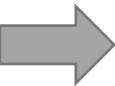


# DEEP LEARNING IN FINANCE



# Big picture: Econometrics vs Machine Learning and Deep Learning



What are we trying to do as researchers? Solve real world problems, right?



Is there a theory?

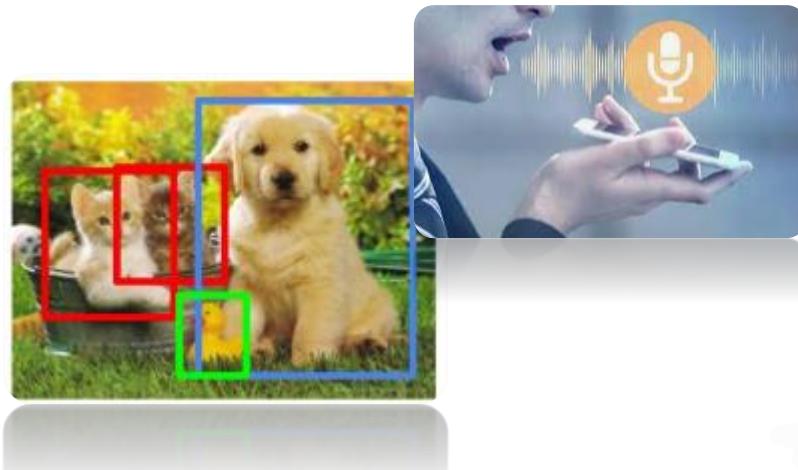
Econ, ML & DL can handle Structured (Tabular) data.

1- What is the **relationship** between

- Quantity demanded and price / income / technology / price of competitors / ... ?
- Wage and education/ age/ gender/ experience/ ...?

2- How about these problems? Object detection, Image Captioning, voice recognition, machine translation, and ...

Only ML & DL can handle Unstructured data, but DL is much better at this.



# → A simple example *(Econometric)*

- Quantifying wage components! (is there a theory?)
- What are the drivers:
  - Demographic variables: Education, age, experience, IQ, ...
  - Social and cultural variables: Ethnicity, race, gender, ...
  - Job characteristic variables: Industry, location, working hours, ...
- Let's build a model (**assuming** a linear functional form!) *(Econometric Setup)*

$$wage = \beta_0 + \beta_1 \text{educ} + \beta_2 \text{age} + \beta_3 \text{exper} + \beta_4 \text{IQ} + \dots + \beta_k \text{hours} + u$$

- Can you **interpret** this model? Do you care about the interpretability? *Yes*
- Can you make **predictions** using your model? *Yes*
- Can you make this functional form more flexible? What are the caveats?  
*Yes, but the trade-off is we lose interpretability.*

*ML & DL is mostly used for predictive power.*

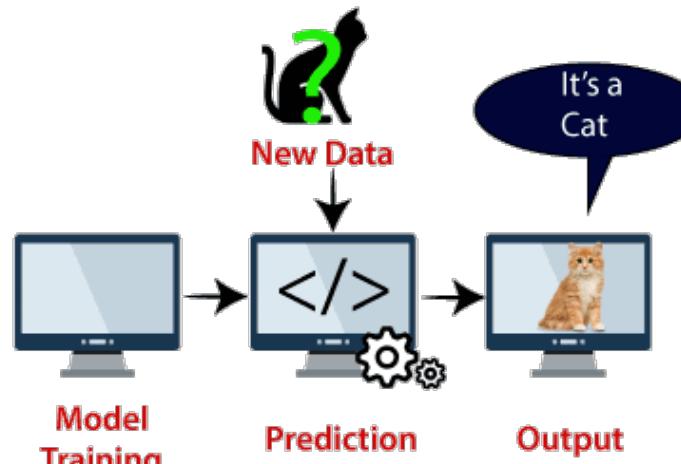
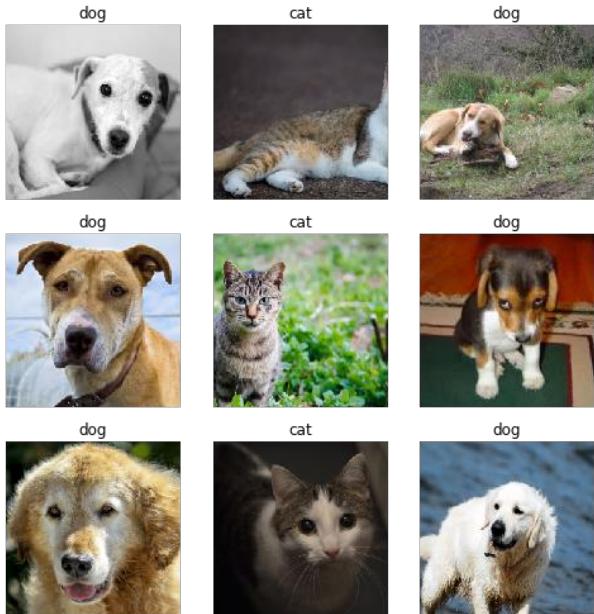


Unstructured data

If feature engineering i.e. (choosing features for model) is done manually then its ML but if the model automatically does feature engineering then its DL.

# A different example (ML/DL)

- Cat vs dog classification problem (image recognition)

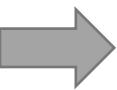


➤ Do you really care about **interpretability** of the model here? *No*

➤ What about accuracy of your predictions? *Yes*

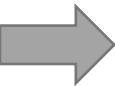
*DL is designed for Unstructured data.*





# Statistical learning vs machine learning

	<i>Econometric / Statistical Learning</i>	<b>Machine Learning / Deep Learning</b>
Focus	Hypothesis testing & interpretability	Predictive accuracy and extracting complex patterns
Driver	Math, theory, hypothesis	Fitting data
<b>Data size</b>	Any reasonable set	Big data
<b>Data type</b>	Structured	Structured, unstructured, semi-structured
Dimensions / scalability	Mostly <b>low</b> dimensional data	<b>High</b> dimensional data
Strength	Understand <b>causal</b> relationship & behavior	Prediction (forecasting and nowcasting)
<b>Interpretability</b>	<b>High</b>	<b>Medium to Low</b>



# Limitations of Econometrics/Structured ML

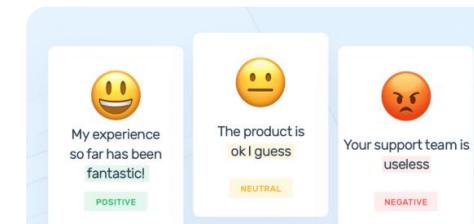
Econometrics/structured ML can **only** handle structured data (tabular data)!

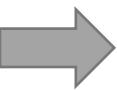
Structured Data

	A	B	C	D
1	Date	Account	Transaction Type	Amount
2	2017-01-12	123	Credit	6089.78
3	2017-01-12	123	Fee	9.99
4	2017-01-12	456	Debit	1997
5	2017-01-12	123	Debit	20996.12
6	2017-01-13	123	Debit	17
7	2017-01-13	123	Debit	914.36
8	2017-01-14	789	Credit	11314
9	2017-01-14	789	Fee	9.99
10	2017-01-14	456	Debit	15247.89
11	2017-01-14	123	Debit	671.28
12	2017-01-15	456	Credit	5072.1
13	2017-01-15	456	Fee	9.99
14	2017-01-16	456	Debit	5109.07
15	2017-01-19	123	Credit	482.01



Unstructured Data  
(everything else!!)





# A more complex example

## Stock price prediction \$\$\$

- What are the classical drivers:

- Company's fundamentals (balance sheet, income statement, cash flow statement)
- Competitors (comparing multiples)
- Technical analysis!
- Seasonality (holidays, months, days, ...)

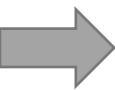
} *Structured data  
ML outperforms  
DL if this is the  
data set*



## What else?

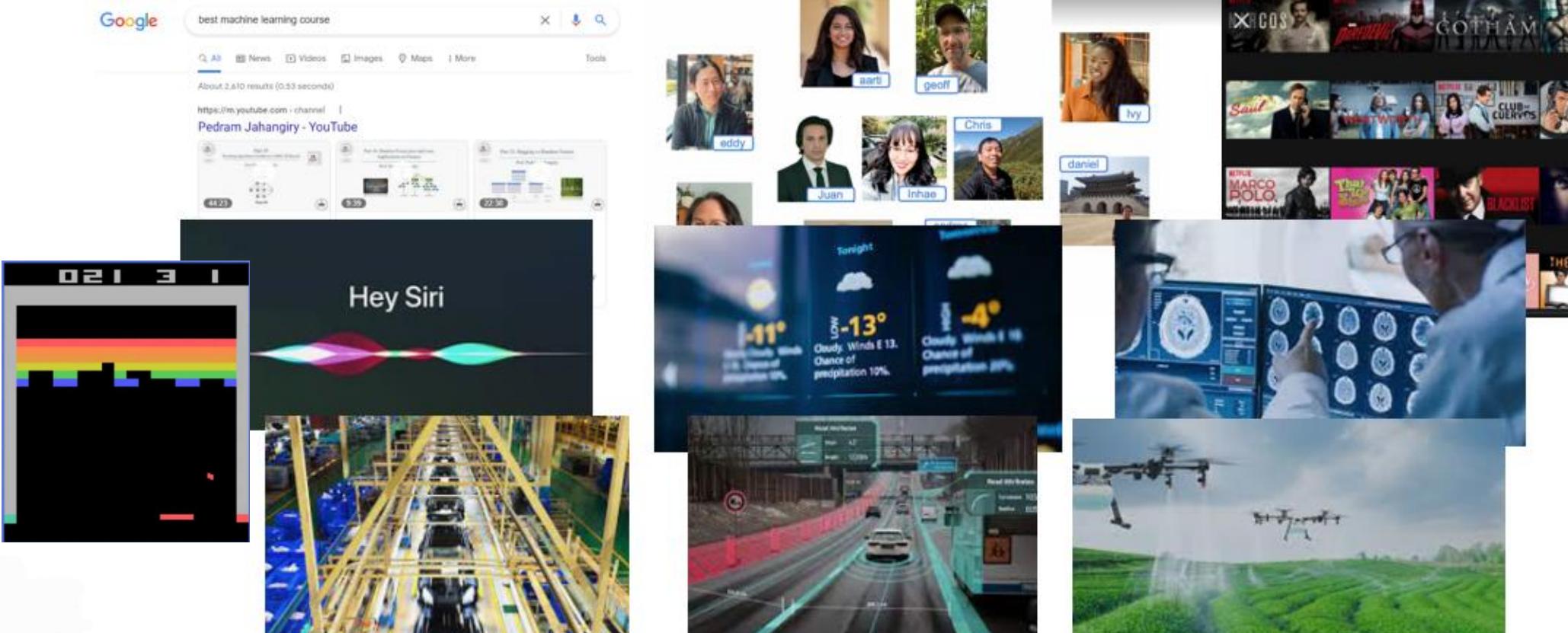
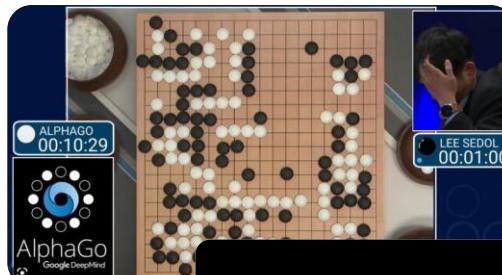
- Market sentiment (news, tweets, blogger opinions, conference calls, ...)
- Satellite images from parking lots!

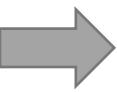
} *Unstructured data  
DL will do better.*



# Why should I learn it?

- It's a bid deal, deep learning is **everywhere!**
- Better career opportunities
- Hedge against next **recession**





# Road map!

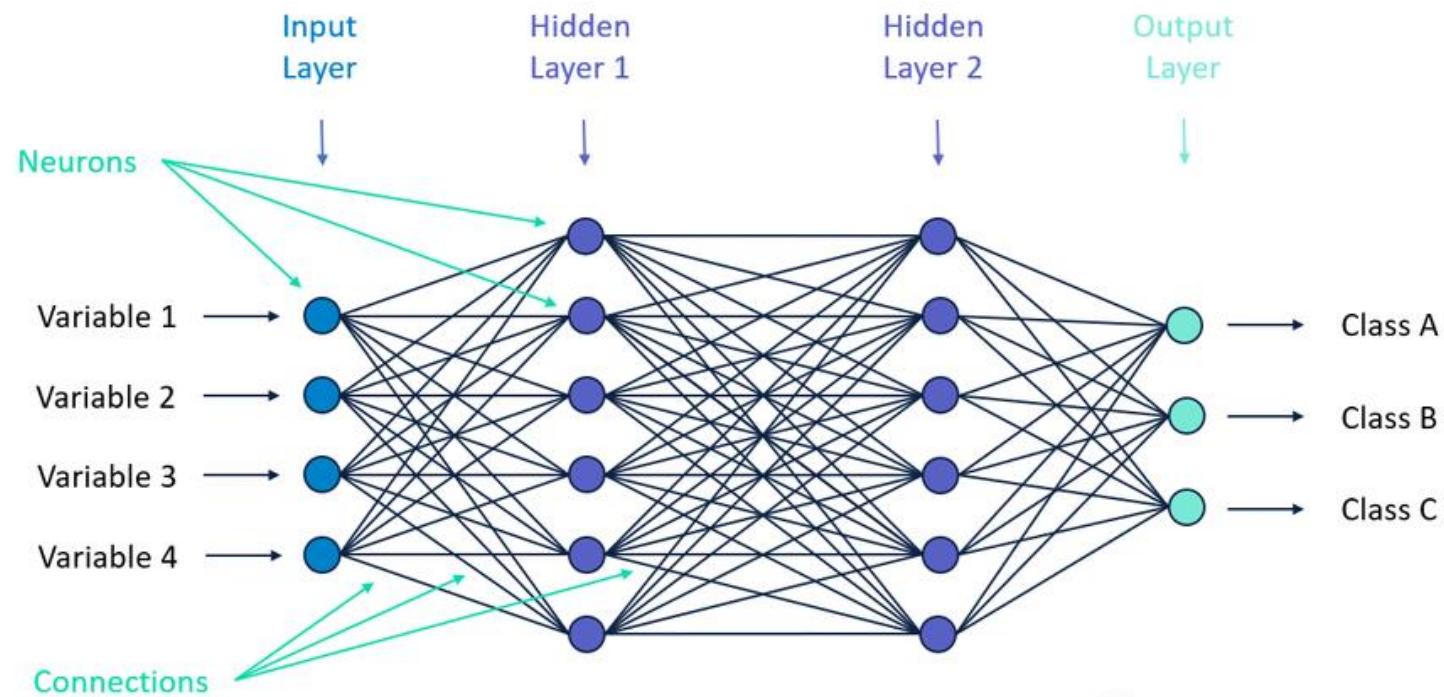
- Module 1- Introduction to Machine Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- Module 6- Deep Sequence Modeling (RNN, LSTM)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

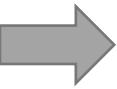


# Module 1 – Part II

## What is Deep Learning?

---





# Artificial intelligence vs Machine learning vs Deep learning

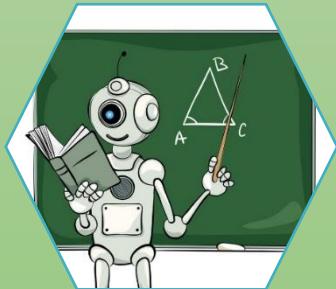
**Artificial intelligence:** Any technique which enables machines to mimic human behavior

**Machine Learning:** Subset of AI that enables computers to learn from data. the model is trained with a set of algorithms

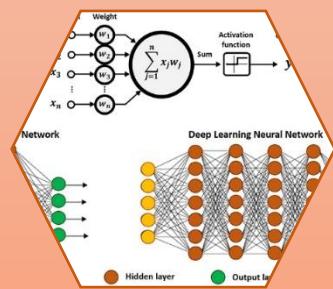
**Deep Learning:** Subset of ML that extract patterns from data using neural networks.



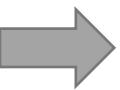
1950's



1980's

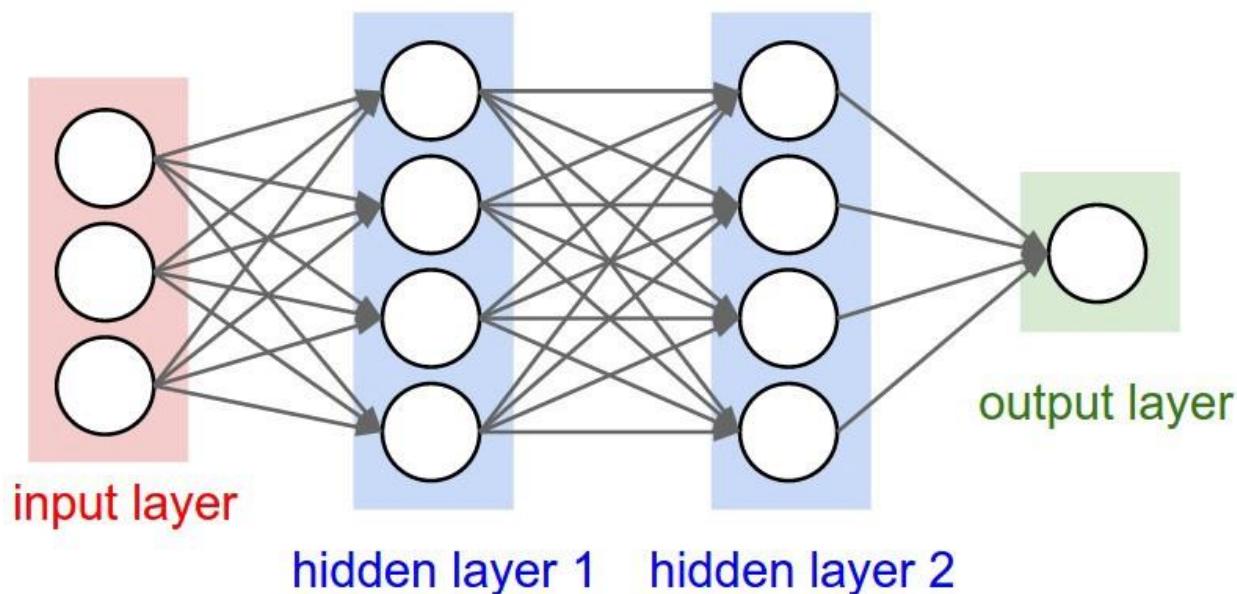


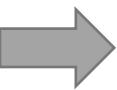
2010's



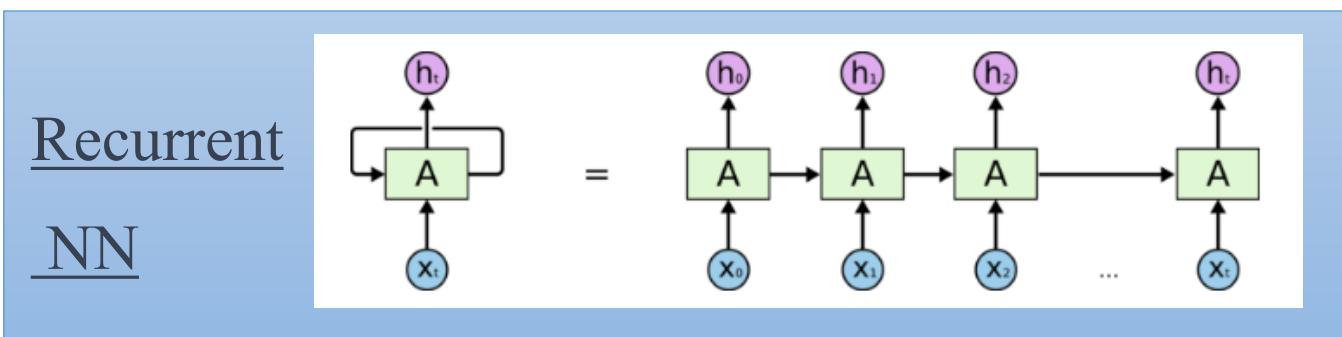
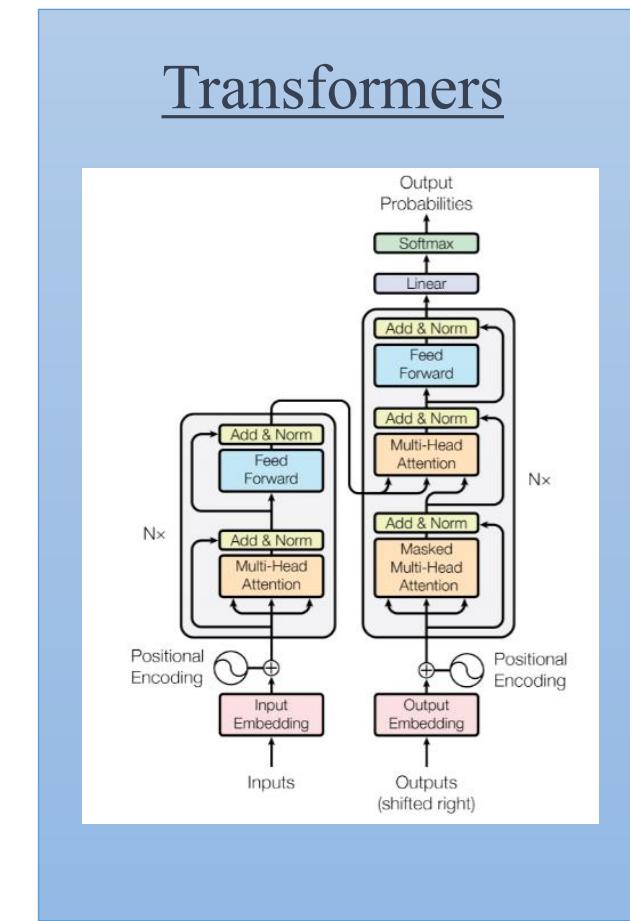
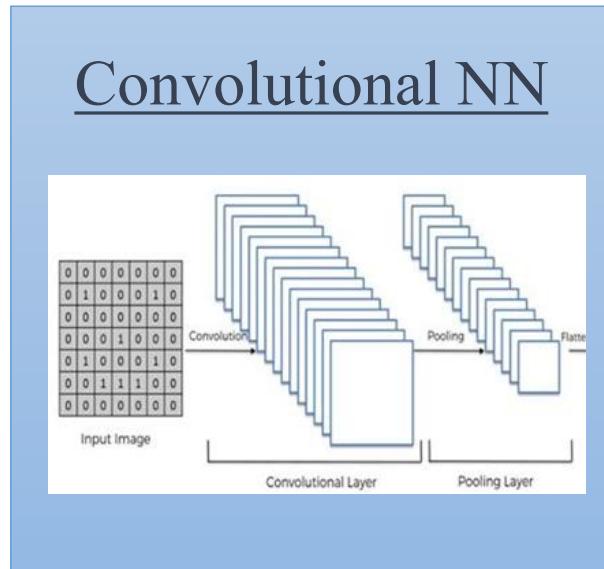
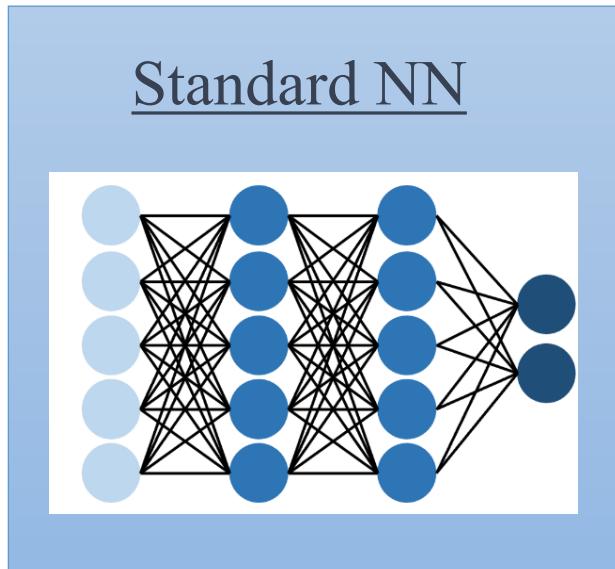
# What is Deep Learning?

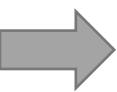
- Deep learning is a type of machine learning that uses **multiple layers** of neurons to process data
- The goal of deep learning is to build a model that can **automatically learn complex patterns** from the data and make **accurate predictions** or decisions





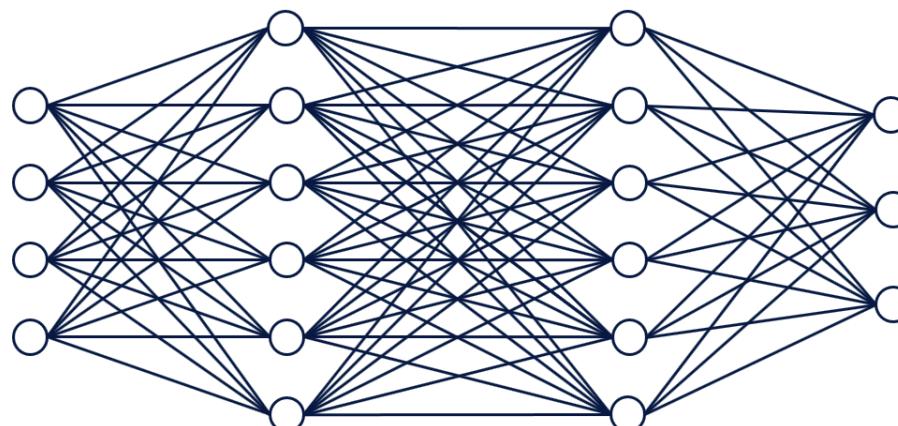
# Network examples

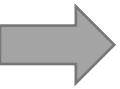




# How Deep Learning Works?

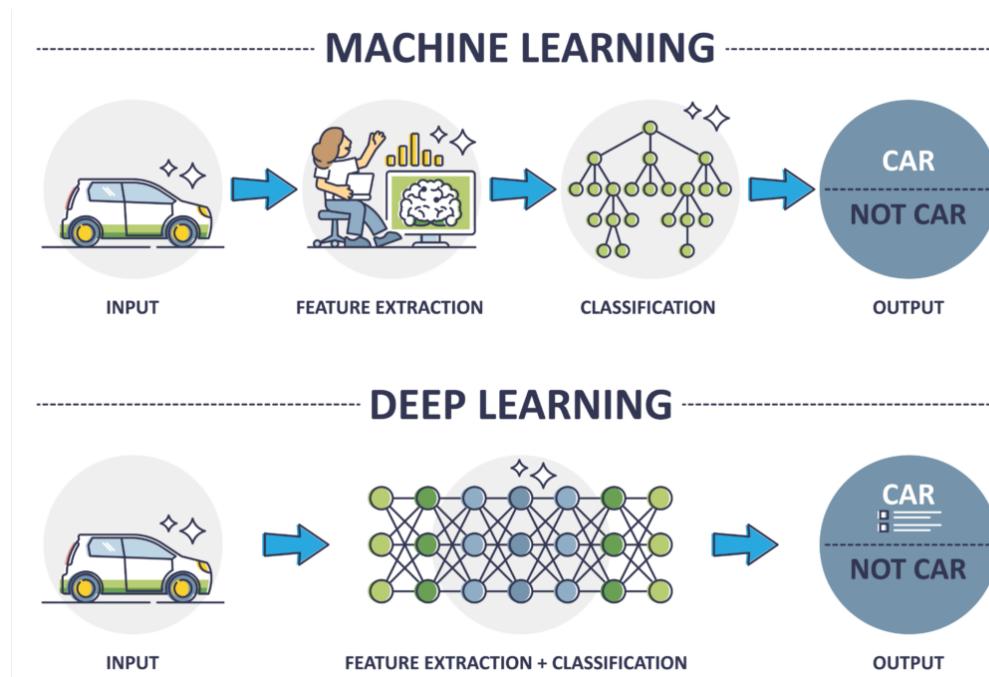
- The network uses a series of interconnected layers, with each layer transforming the input and **passing** it on to the next layer
- The **final layer of the network produces the output**, which is compared to the **true output label** to evaluate the performance of the network
- The network is then **adjusted** to minimize the difference between the predicted and true output labels, and the process is **repeated until the network has learned** to make accurate predictions on new, unseen data

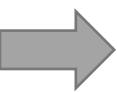




# Why Deep Learning?

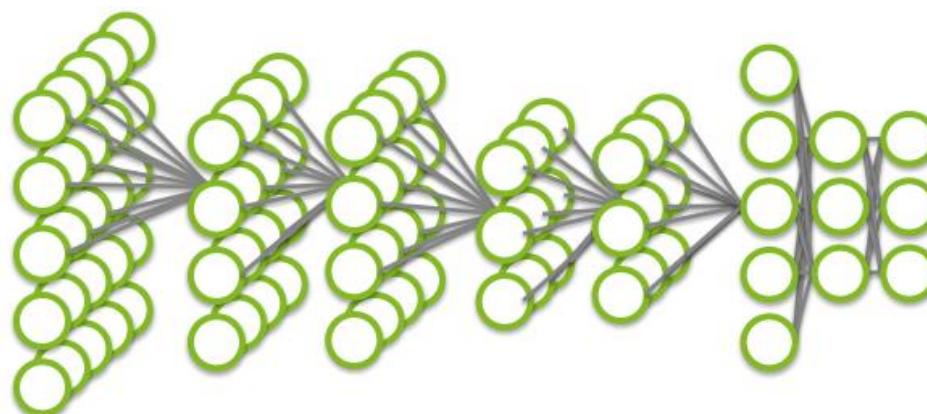
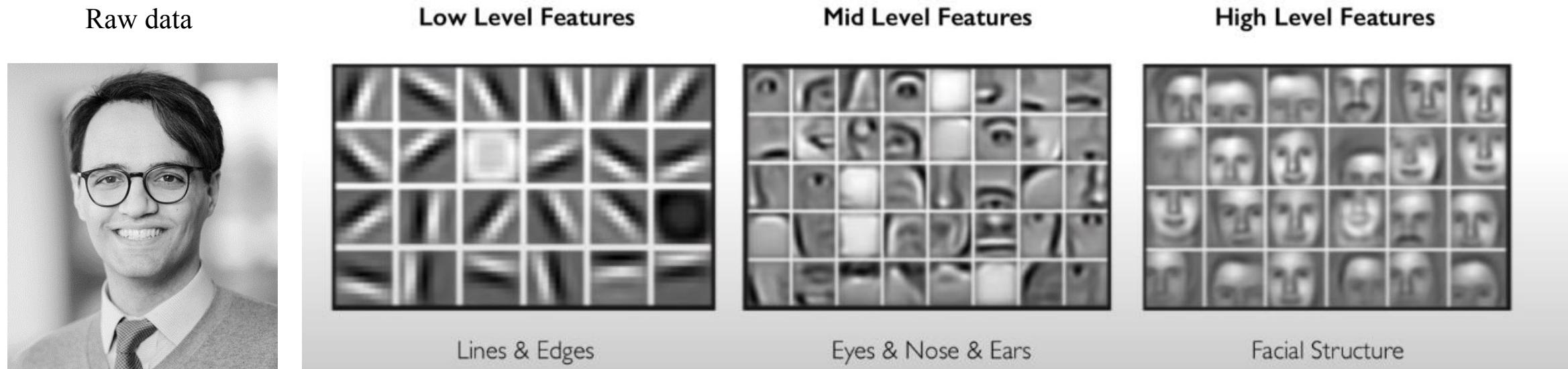
- In deep learning, the model is **not explicitly programmed** with a set of rules or algorithms, but instead learns to recognize patterns and make predictions by adjusting the connections between the neurons in the network!
- Can learn the underlying features **directly from data** and in a hierarchical manner. **No feature extraction/engineering required!**

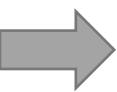




# Why Deep Learning?

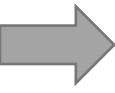
- Hand engineering features for unstructured data is **almost impossible!**





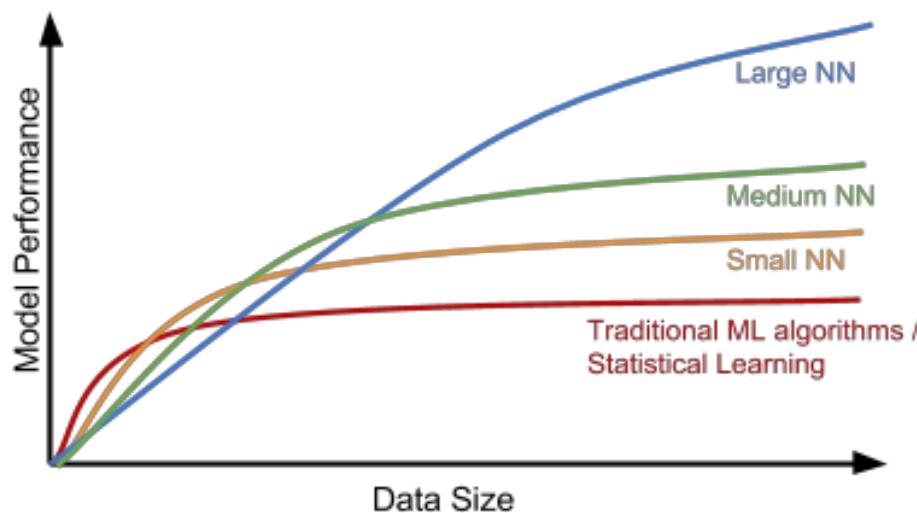
# Summary

	Machine Learning	Deep Learning
Focus	General-purpose predictions	Extracting <b>high-level features</b> from data
Level of Abstraction	Explicitly programmed (rules/algos)	Learn by adjusting neurons
Examples	KNN, SVM, DT, RF, XGBoost, ...	CNN, RNN, LSTM, GAN, Transformers
Requires Feature extraction	Yes	No
Requires high processing power	Not necessarily	Yes
Interpretability	Medium to low	Very low to <b>none</b>
Execution Time on training	Less (compared to DL)	A lot
Execution Time on testing	More (compared to DL)	Little



# Why Now?

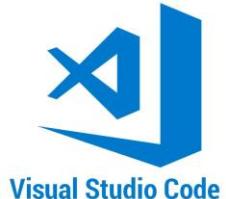
- In small data sets, perhaps traditional machine learning is more effective!
- The performance of deep learning is better when the **data is large**.
- Training large NN is **computationally expensive**.
- **Algorithms** are playing important role in speeding up the training process.

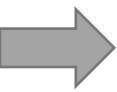


# Module 2

## Setting up Deep Learning Environment

---

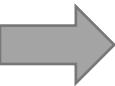




# Road map!

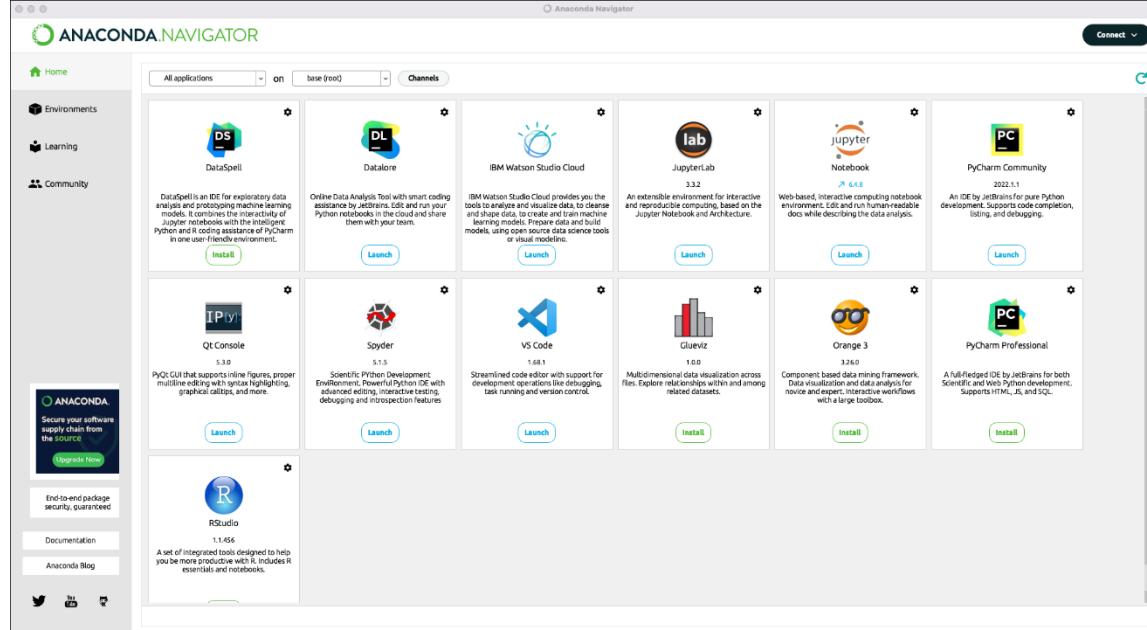
- Module 1- Introduction to Deep Learning
- **Module 2- Setting up Deep Learning Environment**
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, FCN)
- Module 6- Deep Sequence Modeling (RNN, LSTM)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

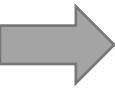




# Install through

- Anaconda is a **distribution** of the Python and R programming languages for scientific computing, that aims to simplify package management with **conda environments**.
- Anaconda offers the easiest way to perform data science and machine learning on a single machine.
- Install Anaconda @ <https://www.anaconda.com/products/distribution>

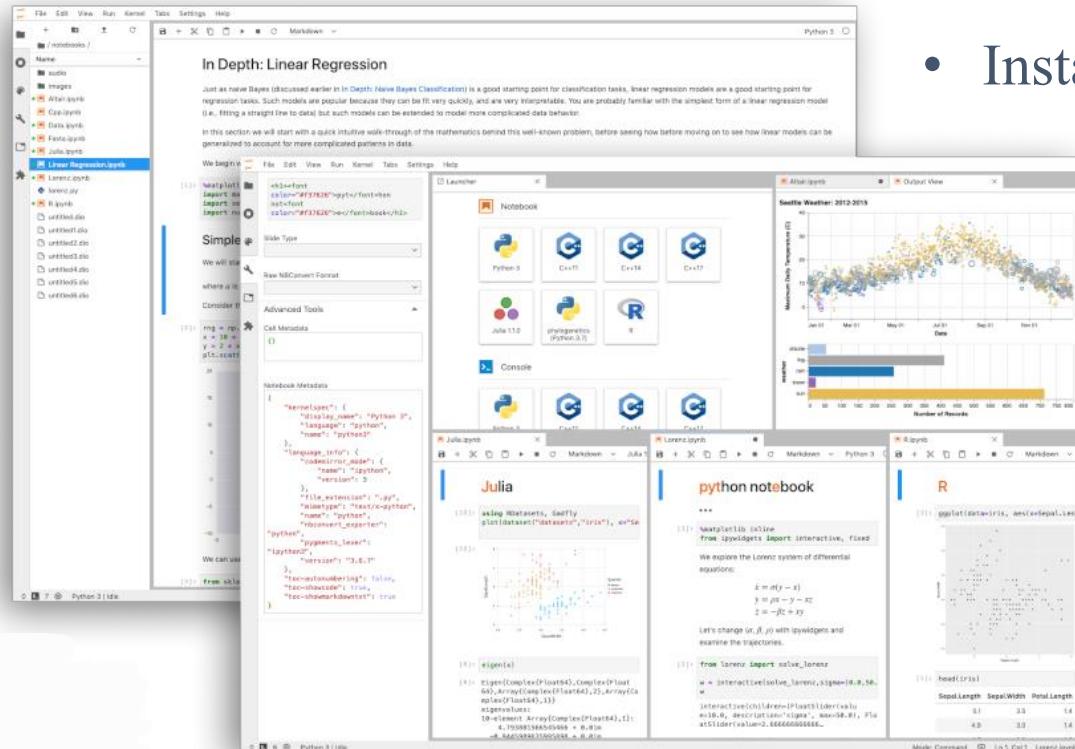




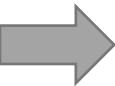
# JupyterLab



- JupyterLab is the latest **web-based interactive development environment** for notebooks, code, and data
- Jupyter's name is a reference to the three core programming languages supported by Jupyter, which are **Julia**, **Python** and **R**



- Install JupyterLab @ <http://jupyter.org/install>

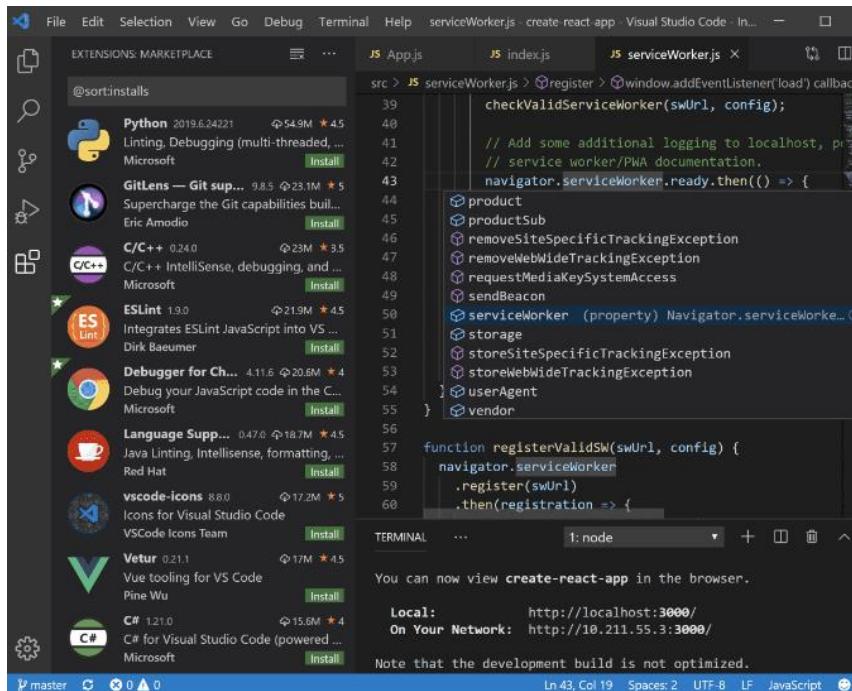


# VS Code

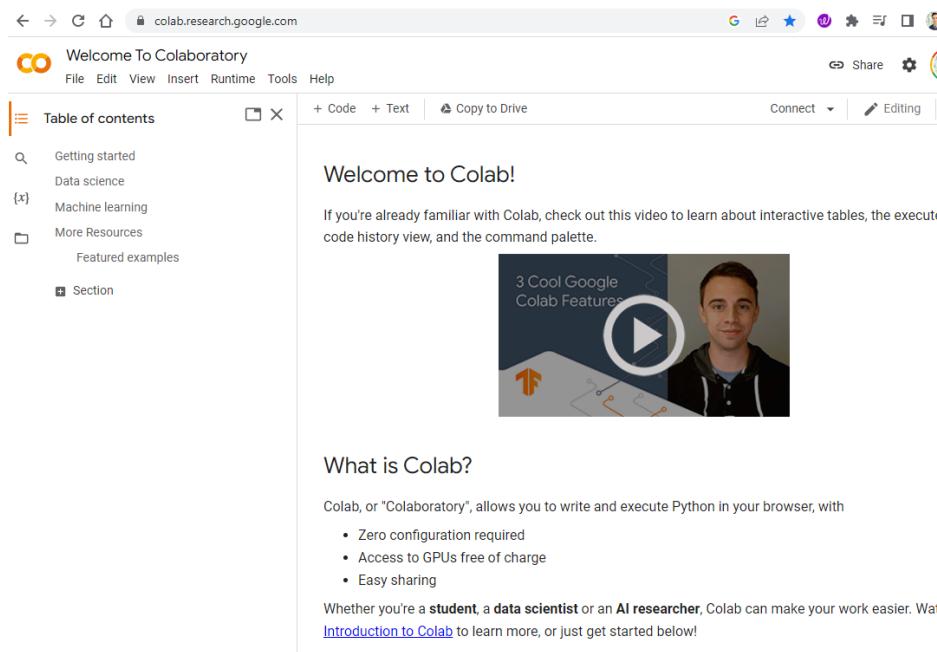


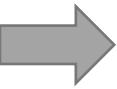
Visual Studio Code

- VS Code is one of the most popular source code editors
- Features include support for **debugging**, syntax highlighting, intelligent **code completion**, code refactoring, and **embedded Git**.
- Install VS code @ <https://code.visualstudio.com/>



- Colab is a free hosted Jupyter notebook-style environment that runs entirely in the **cloud** and requires no setup to use. It also provides access to **machine learning libraries** and computing resources including **GPU**.
- Colab allows anybody to write and execute arbitrary **python code** through the **browser**, and is especially well suited to machine learning, data analysis and education. <https://colab.research.google.com/>

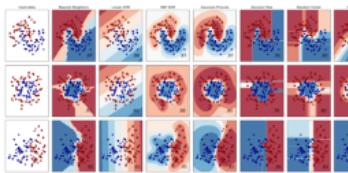




## Classification

Identifying which category an object belongs to.

**Applications:** Spam detection, image recognition.  
**Algorithms:** SVM, nearest neighbors, random forest, and more...

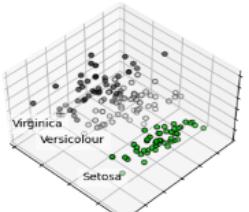


Examples

## Dimensionality reduction

Reducing the number of random variables to consider.

**Applications:** Visualization, Increased efficiency  
**Algorithms:** PCA, feature selection, non-negative matrix factorization, and more...

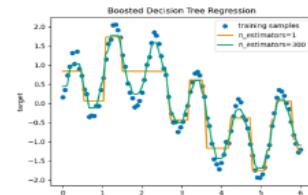


Examples

## Regression

Predicting a continuous-valued attribute associated with an object.

**Applications:** Drug response, Stock prices.  
**Algorithms:** SVR, nearest neighbors, random forest, and more...

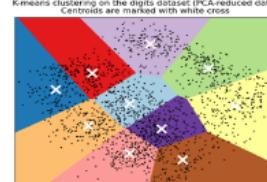


Examples

## Clustering

Automatic grouping of similar objects into sets.

**Applications:** Customer segmentation, Grouping experiment outcomes  
**Algorithms:** k-Means, spectral clustering, mean-shift, and more...

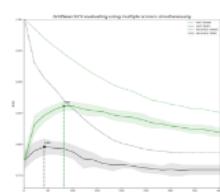


Examples

## Model selection

Comparing, validating and choosing parameters and models.

**Applications:** Improved accuracy via parameter tuning  
**Algorithms:** grid search, cross validation, metrics, and more...

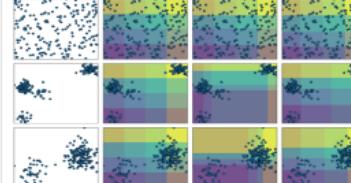


Examples

## Preprocessing

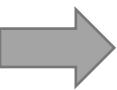
Feature extraction and normalization.

**Applications:** Transforming input data such as text for use with machine learning algorithms.  
**Algorithms:** preprocessing, feature extraction, and more...



Examples

- Scikit-learn is an **open-sourced Python library** and includes a variety of unsupervised and supervised learning techniques.
- It is based on technologies and libraries like Matplotlib, Pandas and NumPy and helps simplify the coding task.
- Install Scikit-learn @ <https://scikit-learn.org/stable/install.html>



- PyCaret is an **open-source**, **low-code** machine learning library in Python that automates machine learning workflows.
- PyCaret is essentially a **Python wrapper** around several machine learning libraries and frameworks
- Install PyCaret @ <https://pycaret.gitbook.io/docs/get-started/installation>

```
# load dataset
import pandas as pd
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

# init setup
from pycaret.classification import *
s = setup(train, target= 'target')

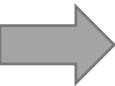
# model training and selection
best = compare_models()

# analyze best model
evaluate_model(best)

# predict on new data
predictions = predict_model(best, data =test )

# save best pipeline
save_model(best, 'my_best_pipeline')
```

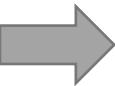
ID	Name
'lr'	Logistic Regression
'knn'	K Nearest Neighbour
'nb'	Naives Bayes
'dt'	Decision Tree Classifier
'svm'	SVM - Linear Kernel
'rbfsvm'	SVM - Radial Kernel
'gpc'	Gaussian Process Classifier
'mlp'	Multi Level Perceptron
'ridge'	Ridge Classifier
'rf'	Random Forest Classifier
'qda'	Quadratic Discriminant Analysis
'ada'	Ada Boost Classifier
'gbc'	Gradient Boosting Classifier
'lda'	Linear Discriminant Analysis
'et'	Extra Trees Classifier
'xgboost'	Extreme Gradient Boosting
'lightgbm'	Light Gradient Boosting
'catboost'	CatBoost Classifier



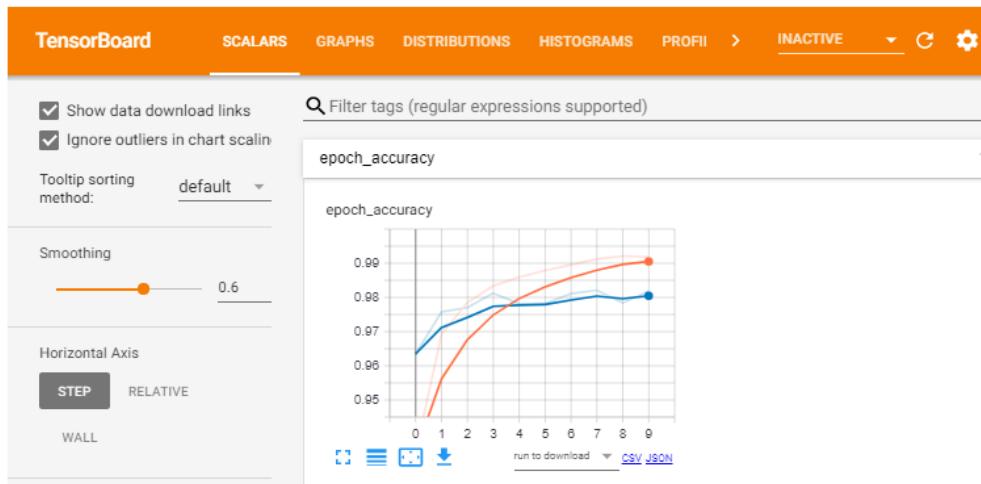
# K Keras

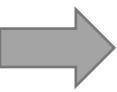
- Keras is a **high-level**, open-source **neural network** library written in Python. It was developed to make it easier for researchers and developers to build and experiment with deep learning models.
- The Keras API became the official high-level API for TensorFlow 2.0 in **2019**.  
<https://keras.io/>

```
model = keras.Sequential(  
    [  
        keras.Input(shape=input_shape),  
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),  
        layers.MaxPooling2D(pool_size=(2, 2)),  
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  
        layers.MaxPooling2D(pool_size=(2, 2)),  
        layers.Flatten(),  
        layers.Dropout(0.5),  
        layers.Dense(num_classes, activation="softmax"),  
    ]  
)  
  
model.summary()
```



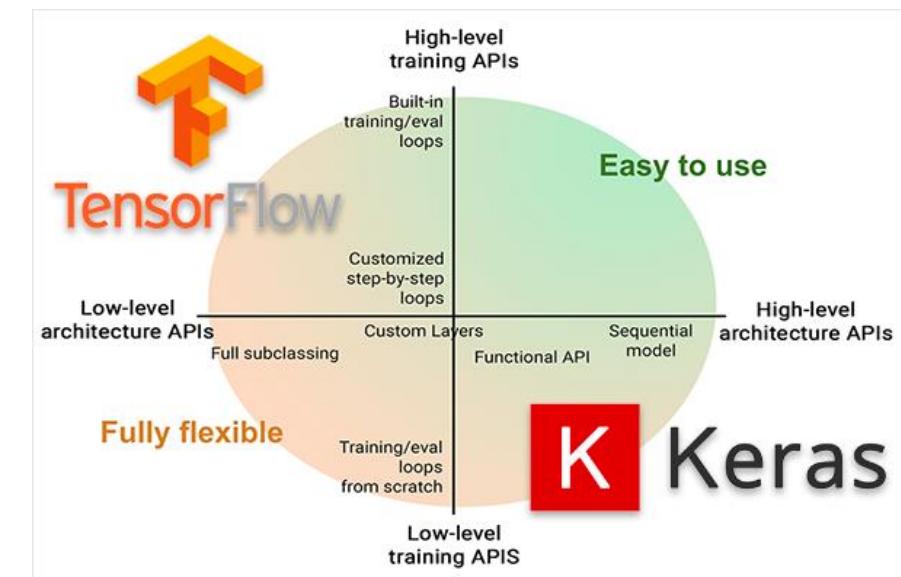
- TensorFlow is a Google-maintained open-source end-to-end platform for prototyping and assessing machine learning models, primarily neural networks.
- TensorFlow also offers **TensorBoard**, a visualization tool for comparing and tracking our learned models.
- It can scale from a single CPU, to a GPU or cluster of GPUs all the way up to a multi-node TPU infrastructure.
- Build-in Google Colab. For local installation visit <https://www.tensorflow.org/install>

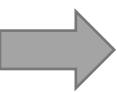




# Keras Vs TensorFlow

- **Level of abstraction:** Keras is a **higher-level** library that provides a more intuitive interface for building and training models, while TensorFlow is a **lower-level** library that provides more flexibility but requires the user to specify more details of the model.
- Keras is a **standalone** library, while TensorFlow includes both a low-level library for numerical computations and a high-level library for building and training machine learning models.
- Keras is a user-friendly interface to TensorFlow





# Available YouTube playlists

Class 3 – Python Crash Course

Pedram Jahangiry



**Python Crash Course**

Pedram Jahangiry

Public

9 videos 1,612 views Updated today

Play all Shuffle

Codes are available on my GitHub account:  
<https://github.com/PJalgotrader/platforms-and-tools>

→ 1.vscode Installation



**Programming tips**

Pedram Jahangiry

Public

13 videos 194 views Updated 3 days ago

Play all Shuffle

Codes are available on my GitHub account:  
<https://github.com/PJalgotrader/platforms-and-tools>

1. Google Colab: Jumpstart!



**Google Colab**

Pedram Jahangiry

Public

3 videos 73 views Updated 6 days ago

Play all Shuffle

Codes are available on my GitHub account:  
<https://github.com/PJalgotrader/platforms-and-tools>

**PYCARET**



Introduction and installation

low-code machine learning

GET STARTED

**PyCaret (Automated machine learning Python package)**

Pedram Jahangiry

Public

3 videos 26 views Updated 3 days ago

Play all Shuffle

All you need from PyCaret Python library to automate your ML workflow.

Codes are available on my GitHub account:  
<https://github.com/PJalgotrader/platforms-and-tools>

# → Platforms and Packages

Listed below are some Python packages and platforms that will be used in the deep learning and deep forecasting courses.



## General Python libraries



## Machine Learning libraries



*dmlc*

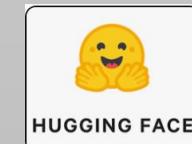
**XGBoost**

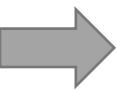
**PYCARET**

**LightGBM**

*(Automated ML)*

## Deep Learning libraries





# Setting up Deep Learning Environment



Personal Workstation



Cloud Platforms



Google Collaboratory

**Pros**

- Full control over hardware and software
- Work offline
- Fixed cost

- Powerful computing resources
- Scalability
- Ease of use
- Cost-effective: Pay-as-you-go
- Collaboration

- Powerful computing resources (GPU, TPU)
- Ease of use
- Collaboration
- No need to set up a local environment

**Cons**

- Scalability
- Maintenance (both hardware and software)

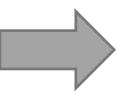
- Expensive for large-scale experiments
- Dependency on the provider
- Limited control
- Internet connection
- Security

- Time limit
- Hardware limitation
- Data storage
- Limited control
- Internet connection
- Security

# → Kaggle Survey 2022

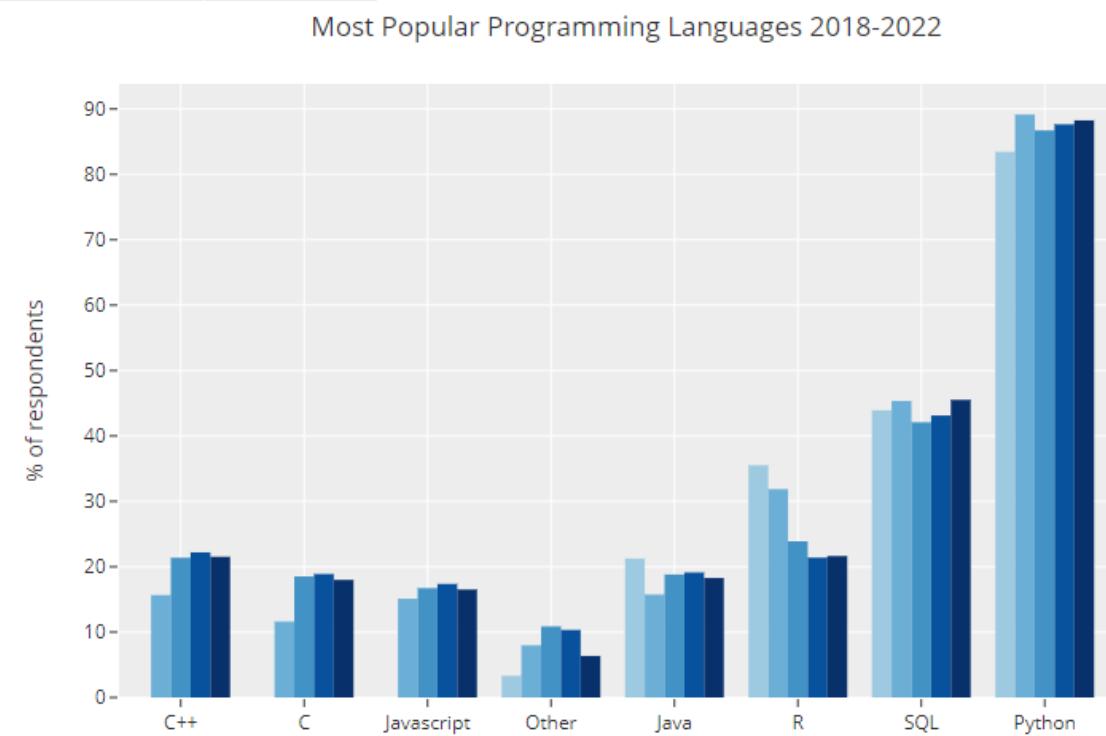
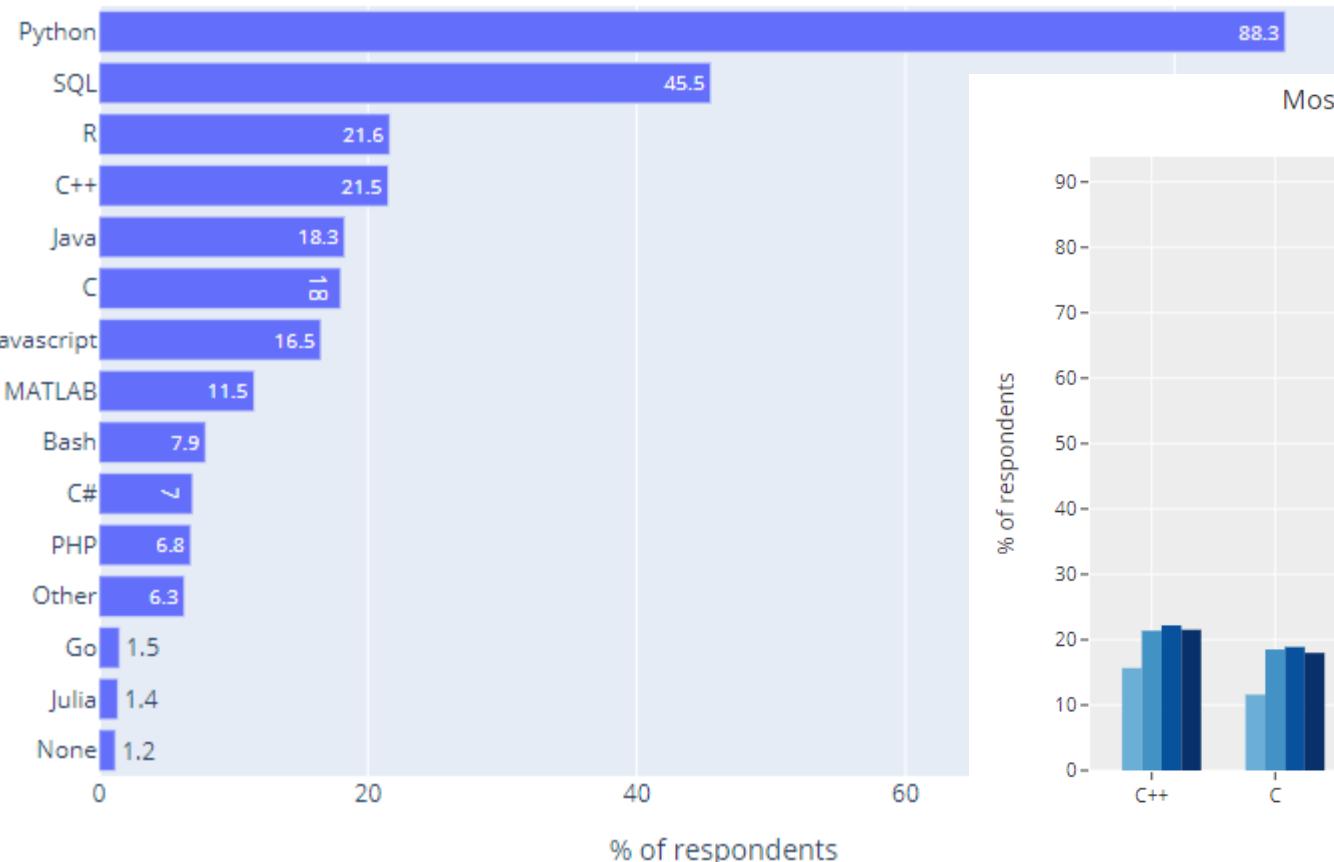


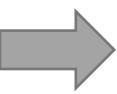
- Kaggle runs a yearly survey among machine learning and data science professionals worldwide.
- This survey is one of our **most reliable** sources about the **state of the industry!!!**



# Programming Languages

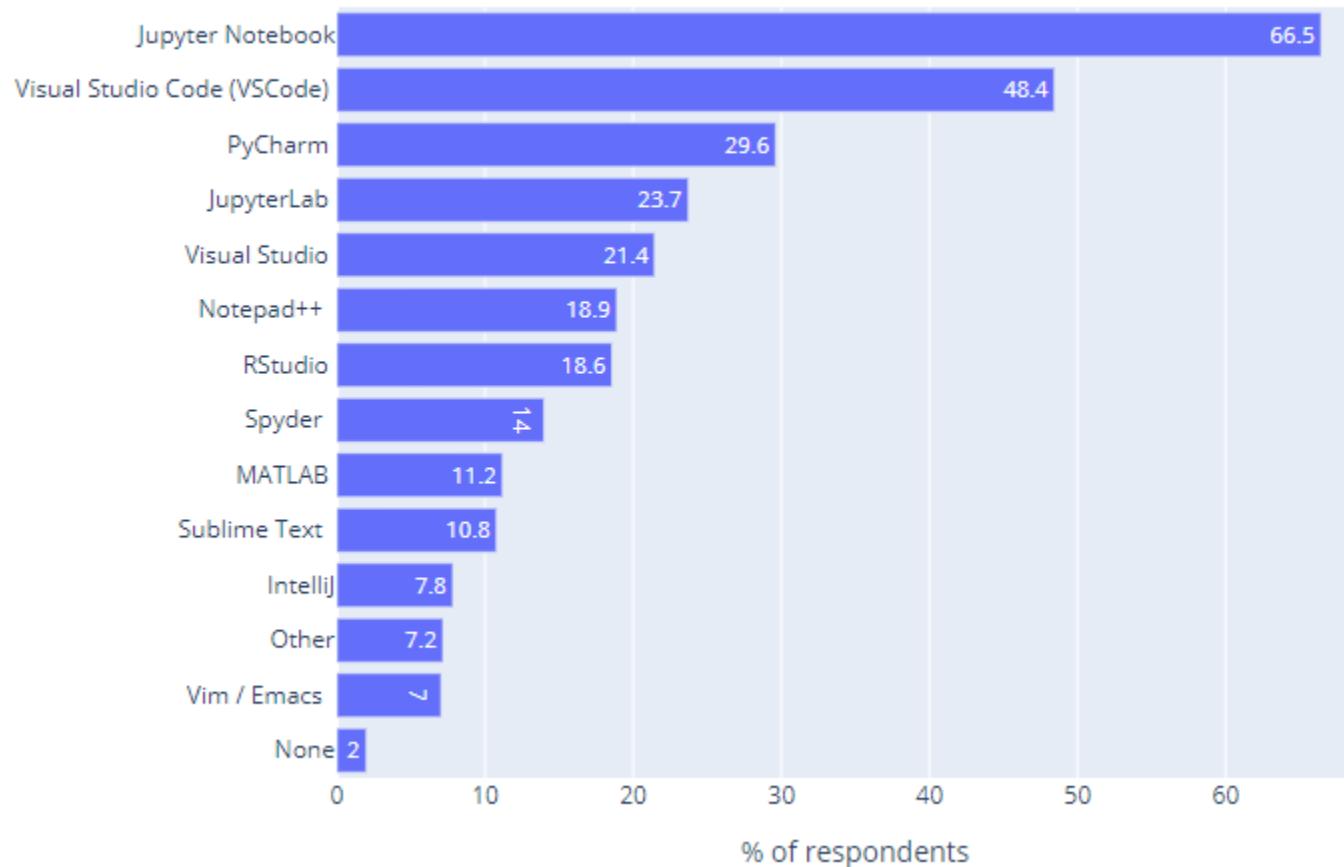
Most Popular Programming Languages in 2022

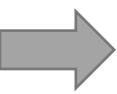




# Popular IDE's

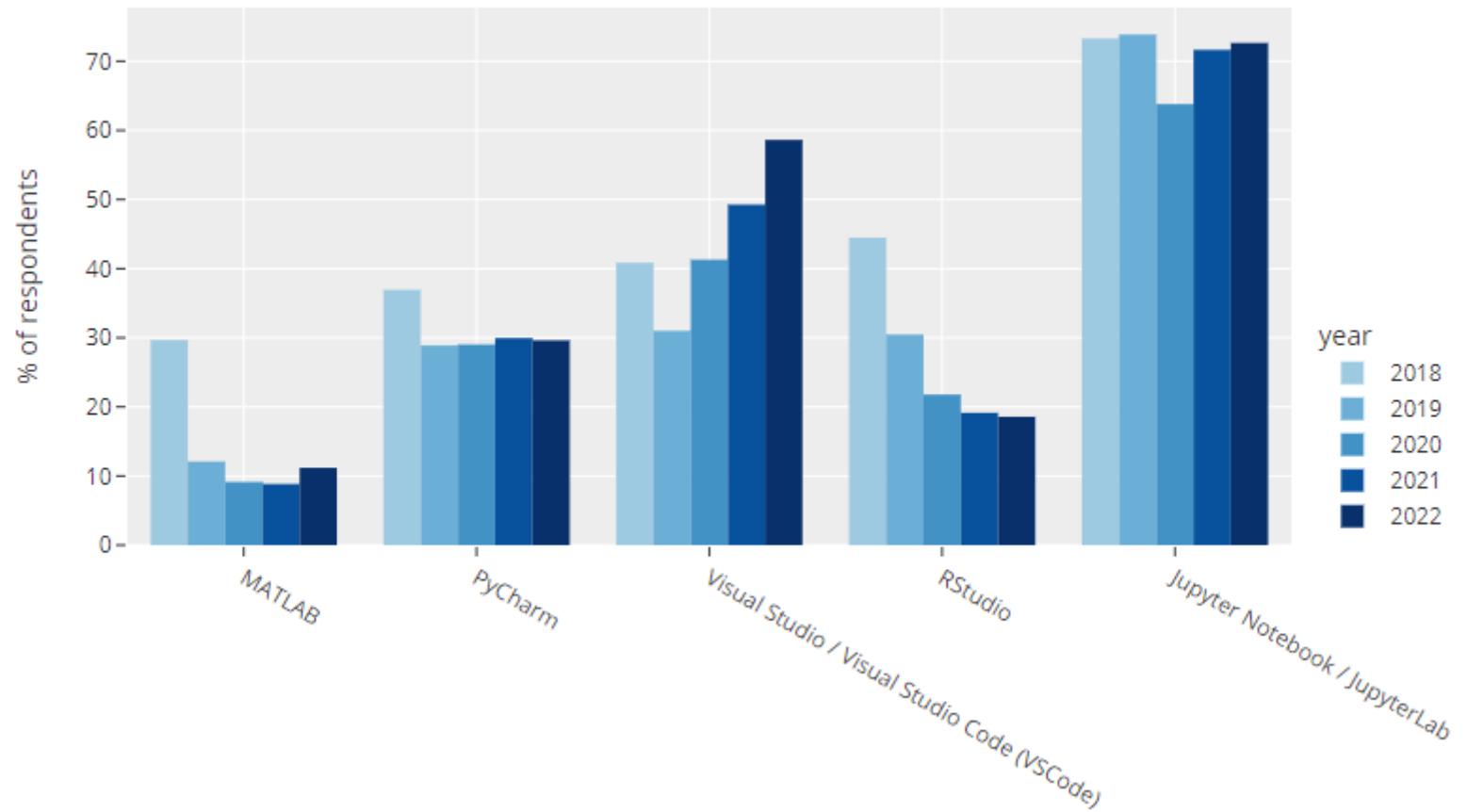
Most Popular IDE's in 2022

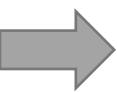




# Popular IDE's

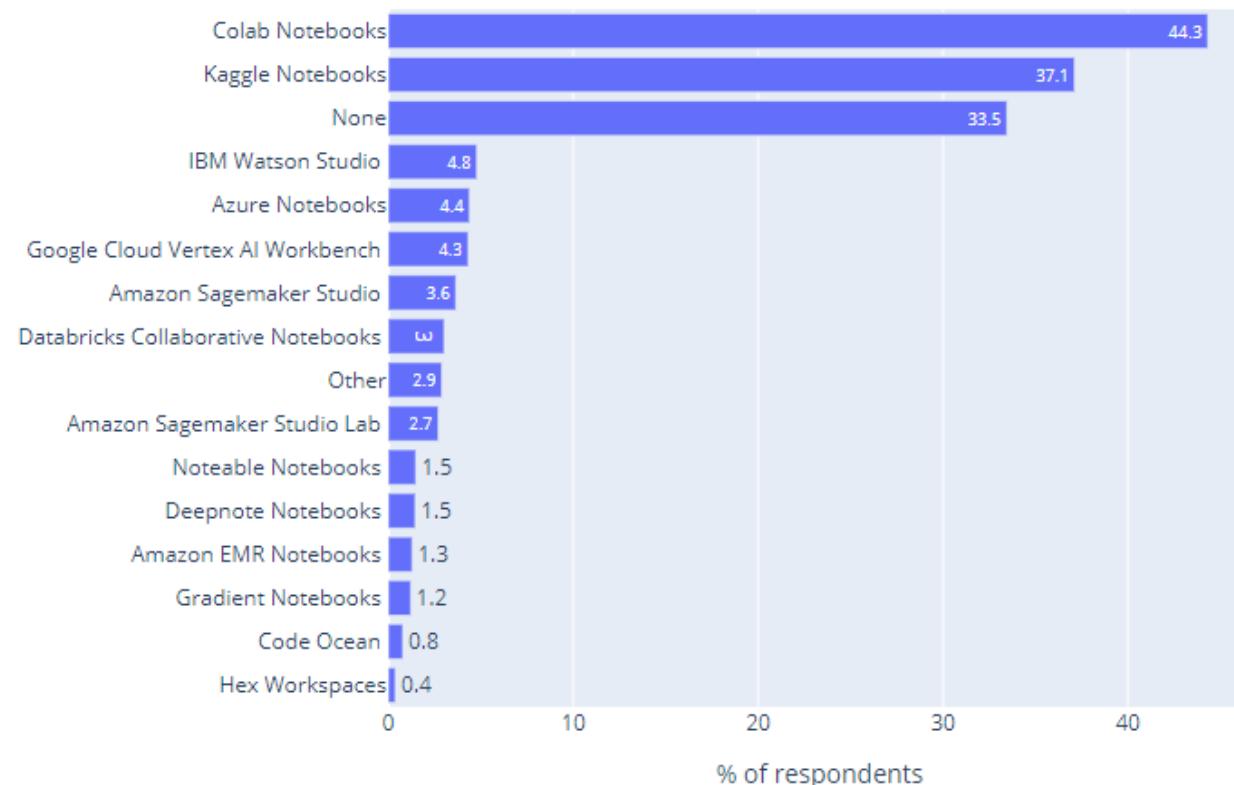
Most Popular IDE's 2018-2022



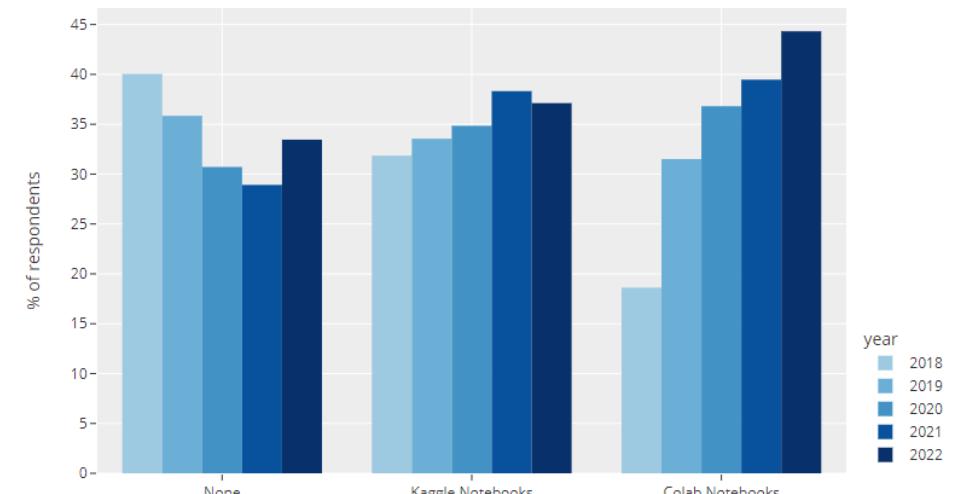


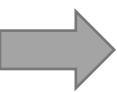
# Popular notebooks

Most popular hosted notebooks in 2022 in 2022



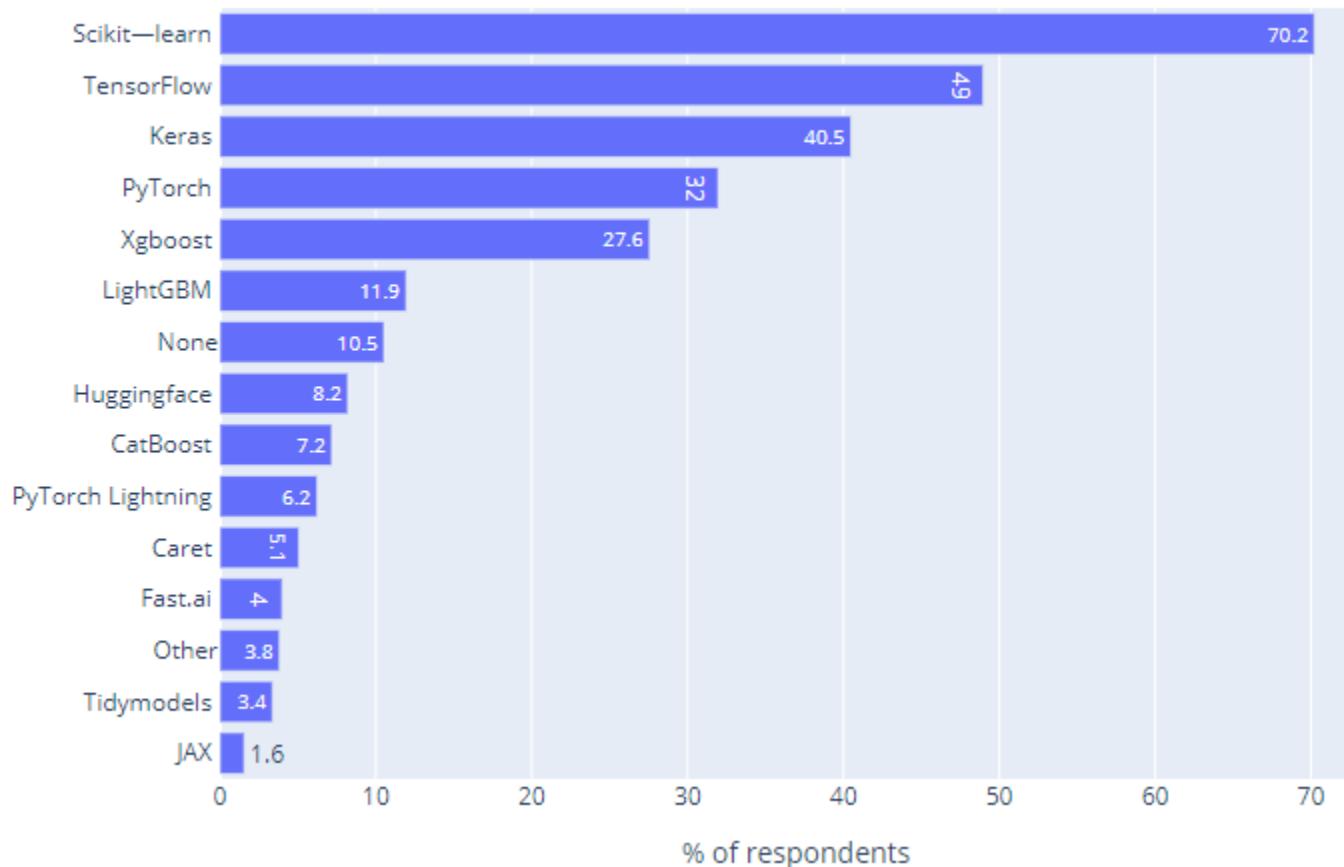
Most popular hosted notebooks products 2018-2022

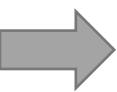




# Machine Learning Frameworks

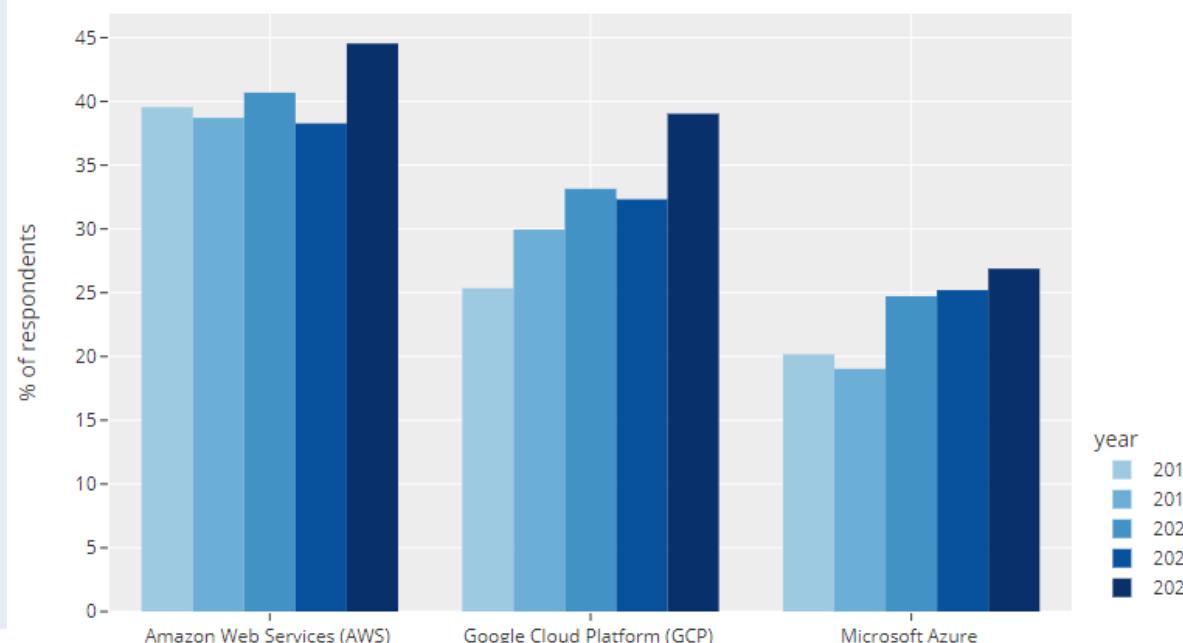
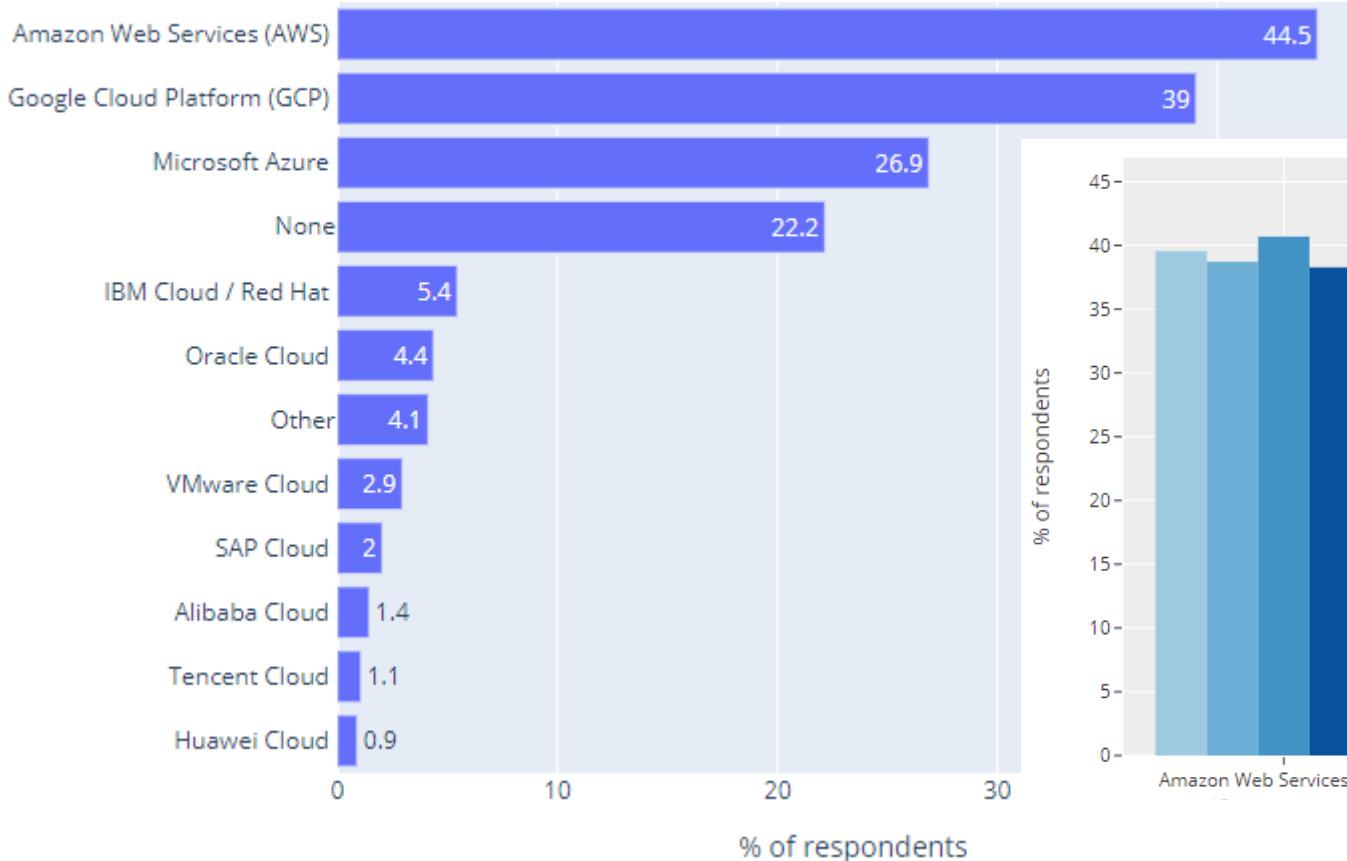
Most popular machine learning frameworks in 2022

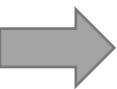




# Cloud computing platforms

Most popular cloud computing platforms in 2022





# The modern machine learning landscape

- From 2016 to 2022, the entire machine learning and data science industry has been dominated by these **two approaches**:
  1. Deep learning
  2. Gradient boosted trees
- Most practitioners of deep learning use **Keras**, often in combination with its parent framework **TensorFlow**.
- This means you'll need to be familiar with **Scikit-learn**, **XGBoost**, and **Keras**

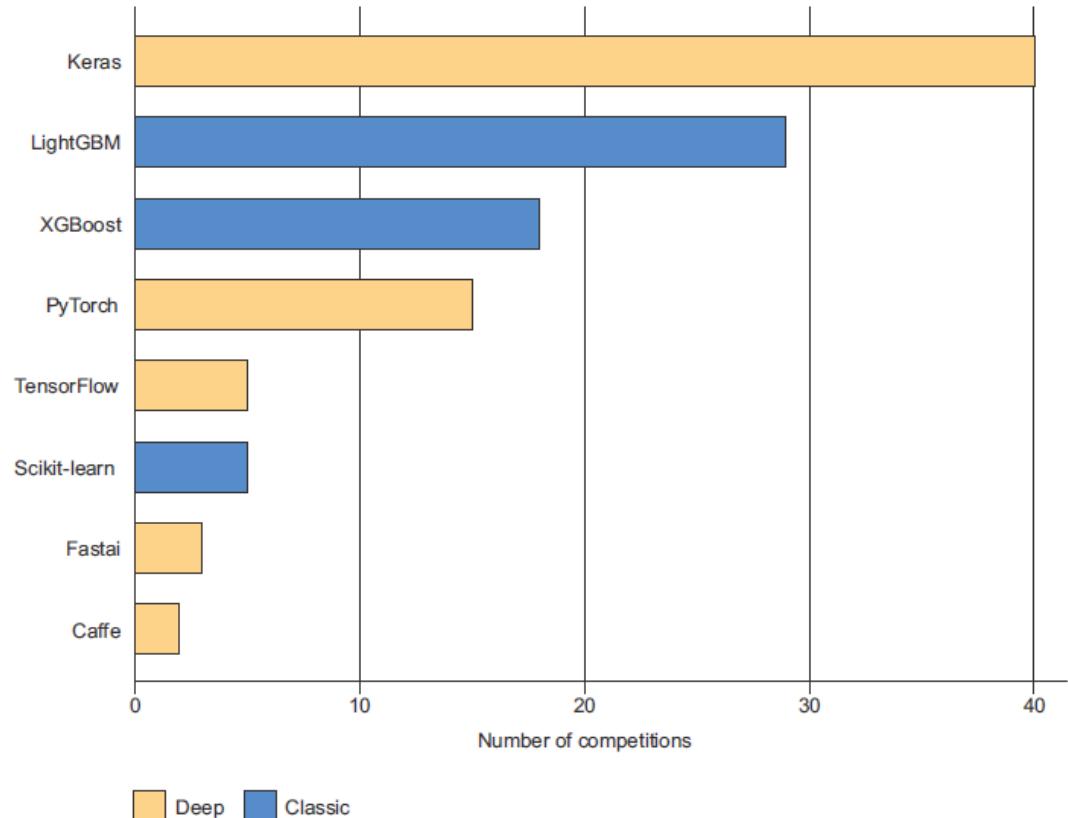


Figure 1.12 Machine learning tools used by top teams on Kaggle

# Module 3 – Part I

## Machine Learning Fundamentals (review)

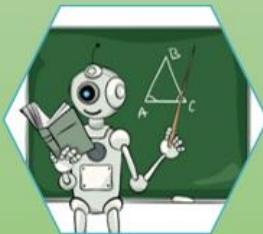
**Artificial intelligence:** Any technique which enables machines to mimic human behavior

**Machine Learning:** Subset of AI that enables computers to learn from data. the model is trained with a set of algorithms

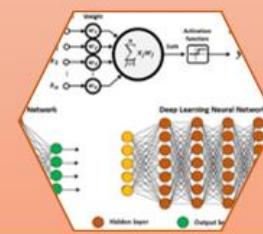
**Deep Learning:** Subset of ML that extract patterns from data using neural networks.



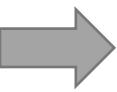
1950's



1980's



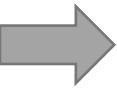
2010's



# Road map!

- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- **Module 3- Machine Learning review (ML fundamentals + models)**
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, FCN)
- Module 6- Deep Sequence Modeling (RNN, LSTM)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

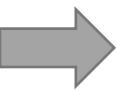




# Machine Learning Fundamentals

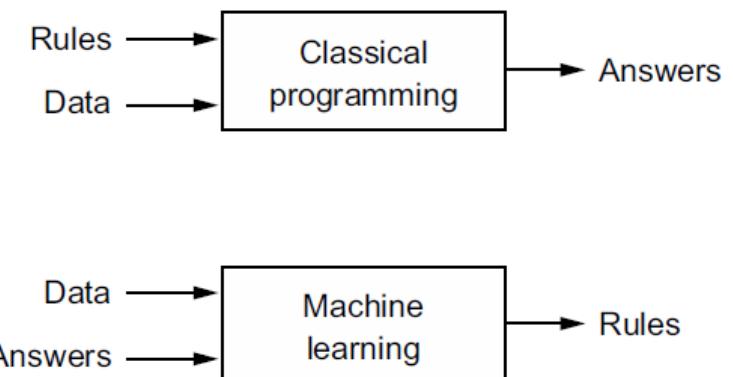
- ML vs traditional programming
- Types of ML
- The Model
- Evaluation metrics
- Bias-Variance tradeoff, overfitting
- Train, Test, Validation
- Resampling methods
- Cost Function
- Solvers/learners (GD, SGD)
- How do machines actually learn?

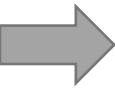




# What is Machine Learning?

- A machine learning system is **trained** (with algorithms) rather than explicitly **programmed**.
- Machine Learning is a subset of AI that enables computers to **learn** from data.
- ML involves automated detection of meaningful **patterns** in data and apply the pattern to make **predictions** on **unseen data!** The purpose is to **generalize**.
- This is done by **minimizing** the loss on the training data.
- The goal is to **maximize** the performance on the unseen data.



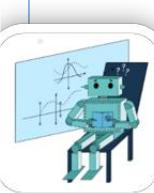


# Types of Machine Learning



Supervised *(labeled data)*

- Regression
- Classification



Unsupervised *(No labels)*

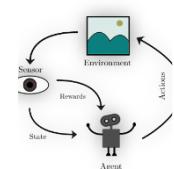
- Clustering
- Anomaly detection
- Dimensionality reduction



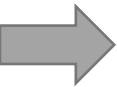
Semi-supervised



Self-supervised



Reinforcement  
Learning



# The Model

---

$$y = f(X, \theta) + \epsilon = f(X_1, X_2, \dots, X_m, \theta_1, \theta_2, \dots, \theta_k) + \epsilon$$

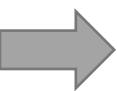
y : response, dependent variables, output, Target

X: predictors, independent variables, input, Features

$\theta$ : estimates, specifications, Parameters

✓ It is all about estimating  $f$  by  $\hat{f}$  for two purposes:

- 1) Inference (interpretable ML)
- 2) Prediction

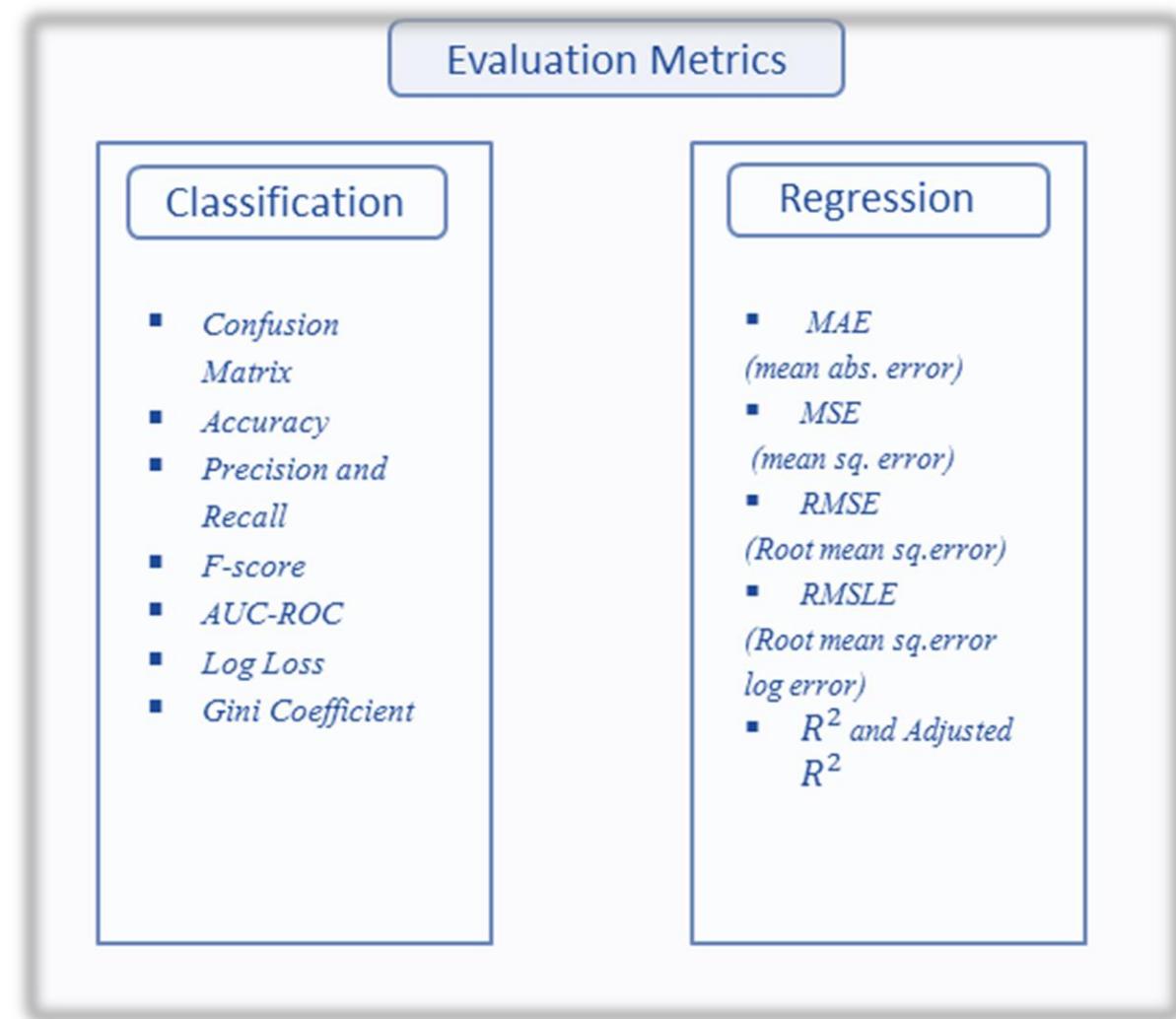


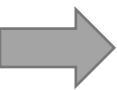
# Evaluation metrics

In general, we want to compare how close are the predictions to the actual numbers in the **test set**.

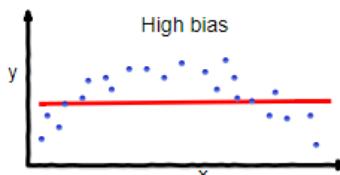
This is typically assessed using

- MSE for **quantitative** response
- Misclassification rate for **qualitative** response

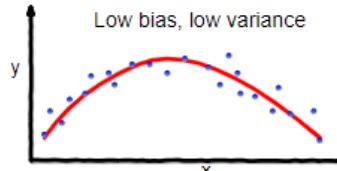




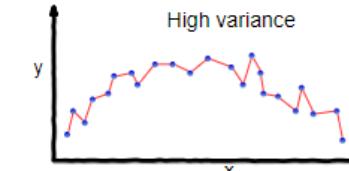
# Representations of the bias-variance tradeoff



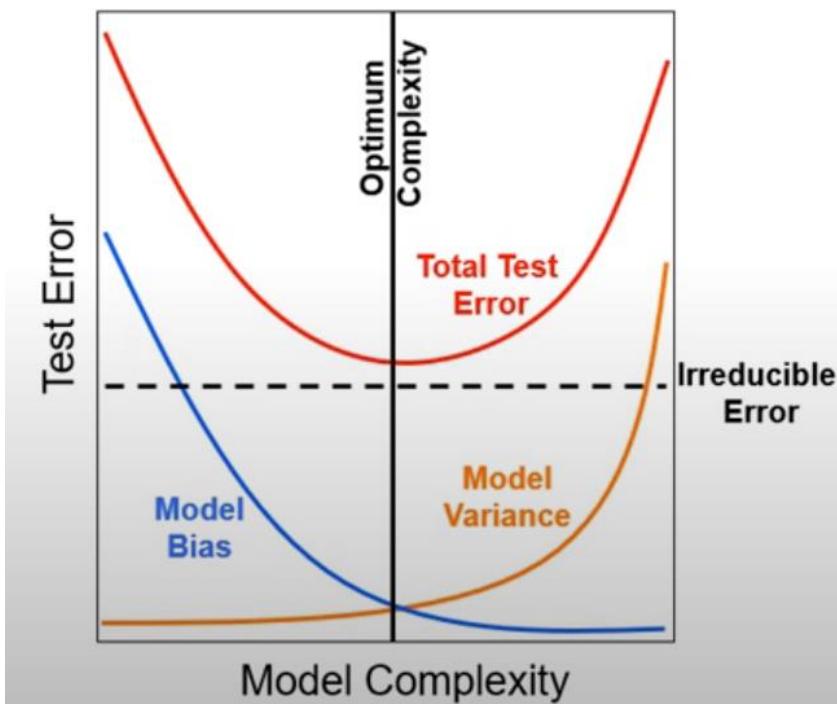
underfitting



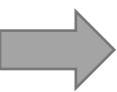
Good balance



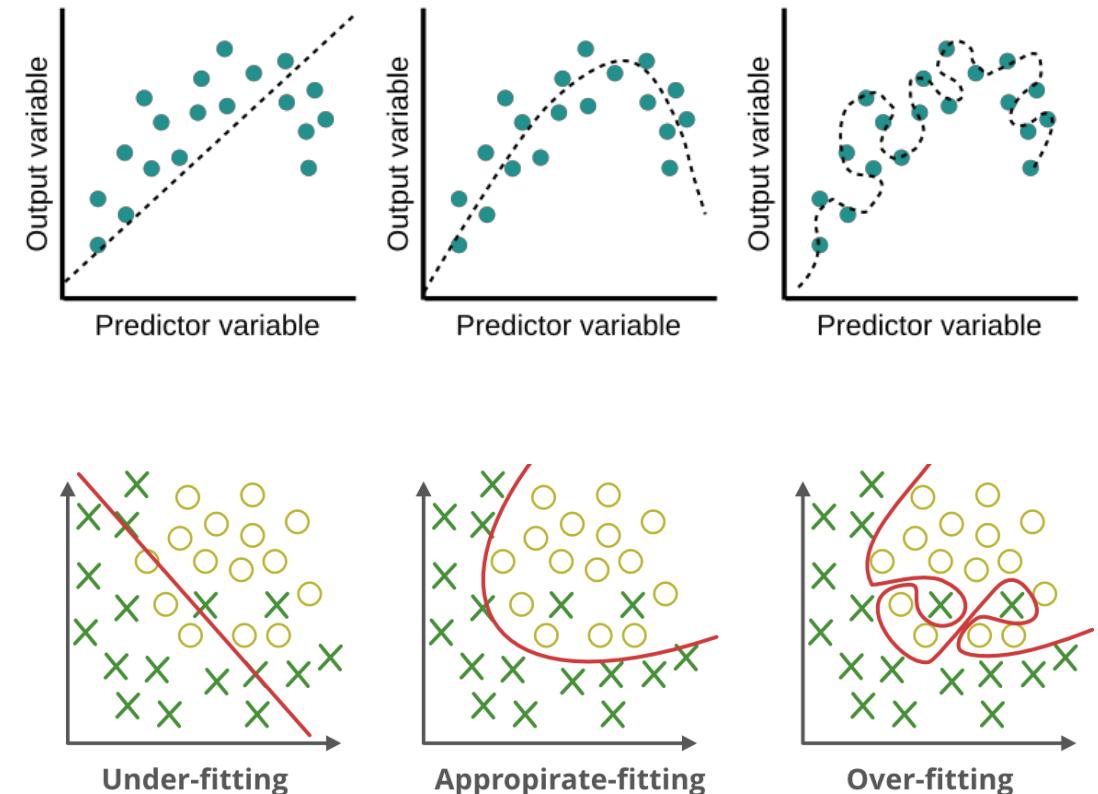
overfitting

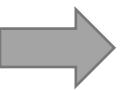


Optimization (optimizes well in Train set)  
Vs  
Generalization (generalizes well in Test set)



# Overfitting

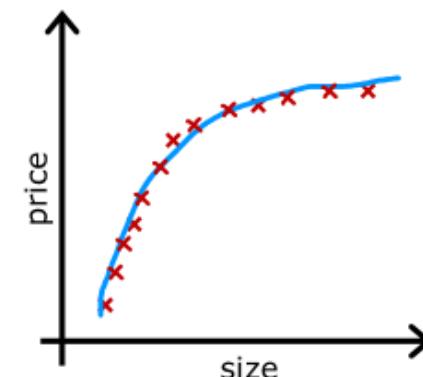
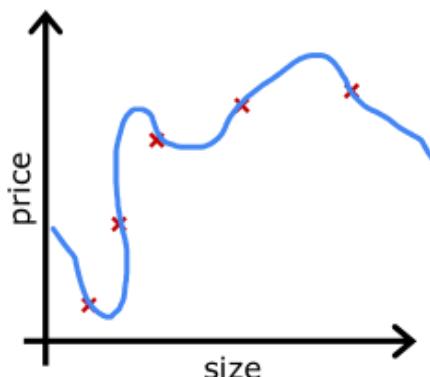




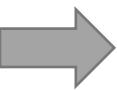
# Mitigate overfitting

The main techniques used to mitigate overfitting risk in a model construction are:

- 1) Collect **more data** (Can reduce bias AND variance)
- 2) **Complexity** reduction (regularization, feature selection)
- 3) Cross validation (estimate the performance in test set)



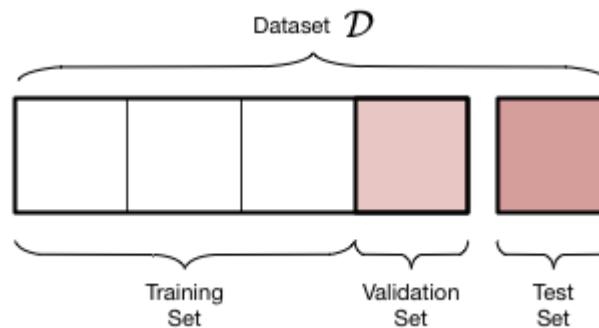
With more training example



# Partitioning of the dataset

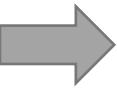
The data set is typically divided into three non-overlapping samples:

- 1) Training set: to train the model
- 2) Validation set: to validate and tune the model (*Validate & tune hyperparameters*)
- 3) Test set: to test the model's ability to predict well on new data (**generalize**)



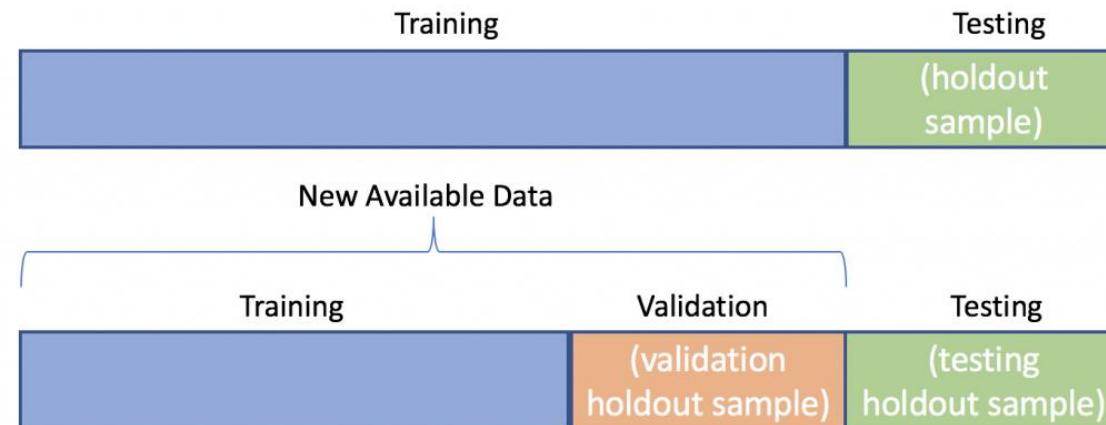
To be valid and useful, any supervised machine learning model **must** generalize well beyond the training data.

Large dataset is needed! But what if we don't have it?



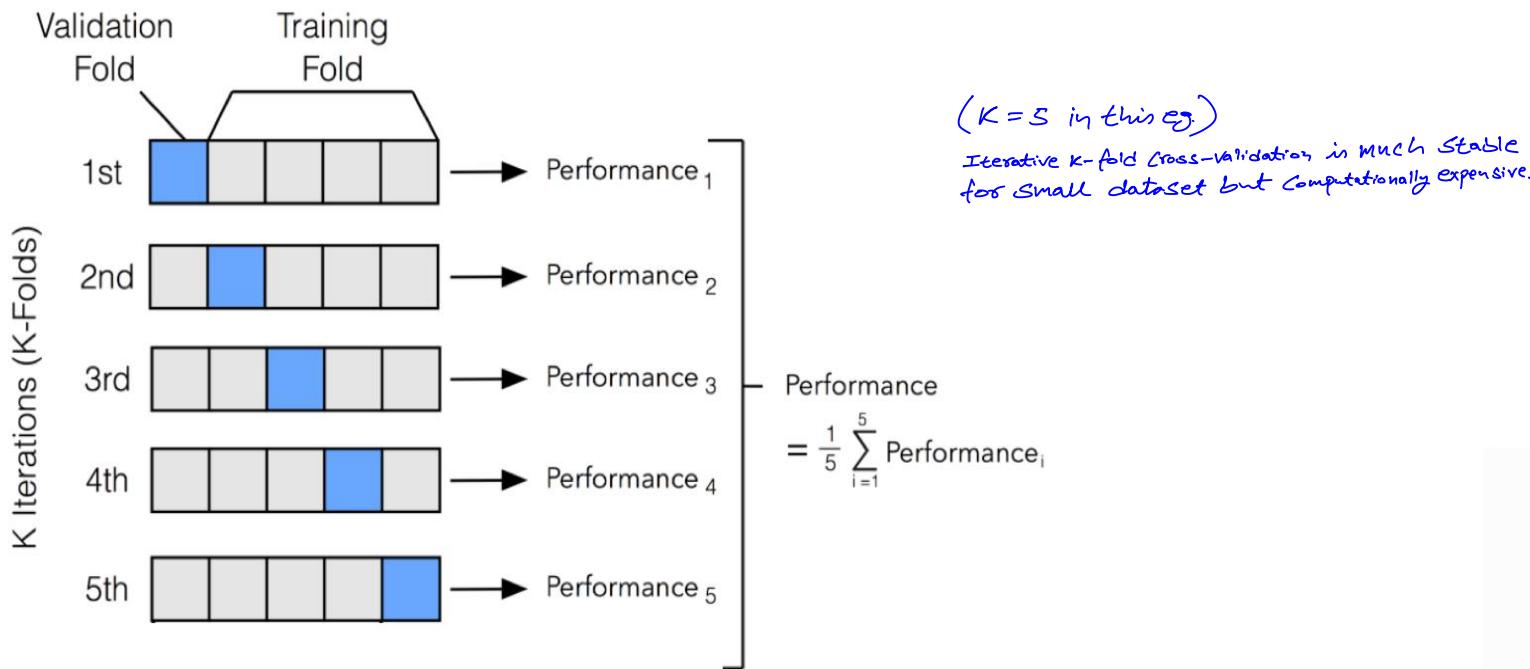
# Resampling methods

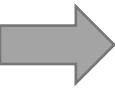
- Sometimes we cannot afford to split the data in three because the algorithm may **not learn** anything from a **small training dataset**!
- **Small validation set** is also problematic because we cannot tune the hyperparameters properly! **Unstable** model performance in validation set!
- **Solution:** combining the training and validation sets and use cross validation!



# → K-fold Cross Validation

- 1) Divide the training data into  $K$  roughly equal-sized non-overlapping groups. Leave out  $k^{th}$  fold and fit the model to the other  $k - 1$  folds. Finally, obtain predictions for the left-out  $k^{th}$  fold.
- 2) This is done in turn for each part  $k$  and then the results are combined.





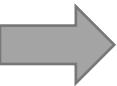
# Why do we use Cross Validation?

Cross validation is mainly used for two purposes:

1. Model **architecture** selection (optimization vs generalization) (*hyperparameters*).
2. Estimation of model performance in the test set

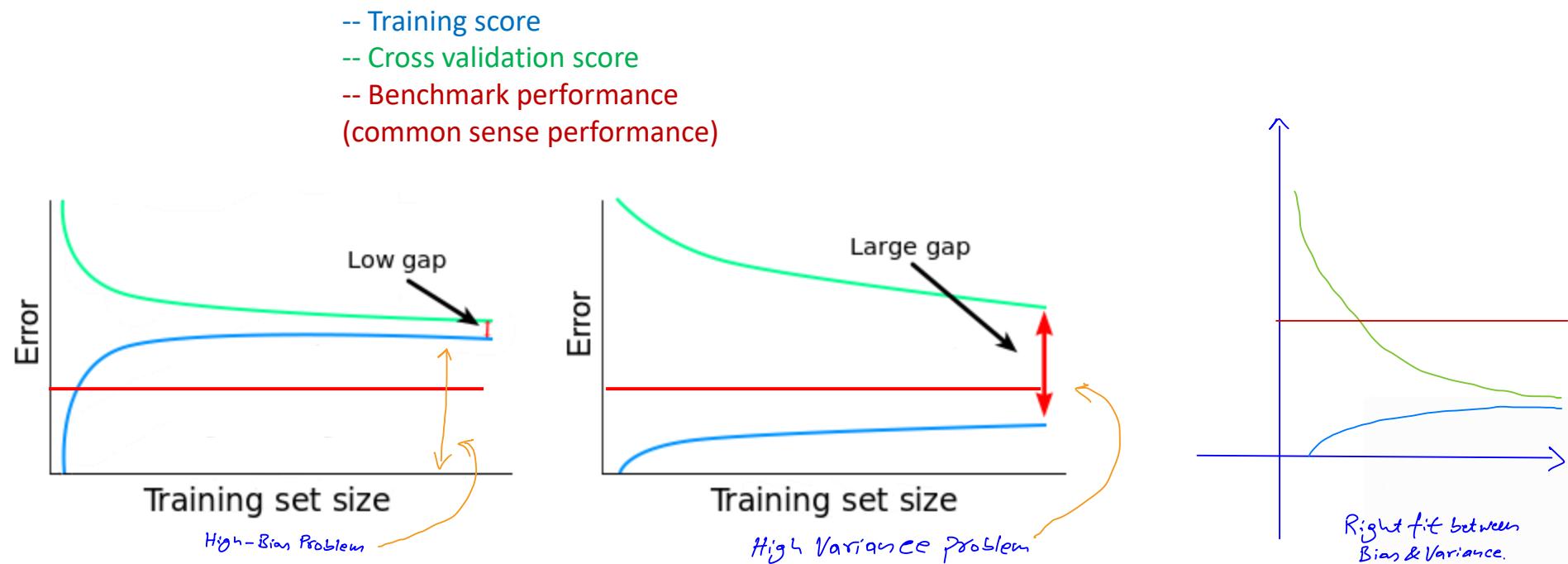


- After selecting the **best model architecture**, we estimate the generalization error using the test set.
- Different model comparison is based on **test set** performance!

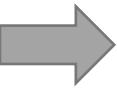


# The Learning Curve: Do we need to collect more data?

- A learning curve is a plot that shows the relationship between the amount of **training data** and the **performance** of a machine learning model.
- It is used to diagnose whether a model has high **bias**, high **variance**, or is **just right**.

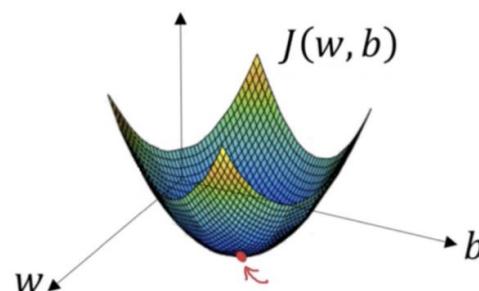


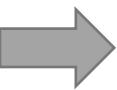
# How do Machines Learn?



# Terminology

- Learning: Finding the model **weights** (parameters' values)
- **Cost Function**: Tells us “**how good**” our model is at making predictions for a given set of parameters.
- The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate.
- The two most frequently used optimization algorithms when the cost function is **continuous** and **differentiable** are Gradient Descent (GD) and Stochastic GD.



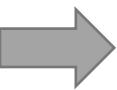


# Solvers (learners)!

- Gradient Descent: is an iterative optimization algorithm for finding the minimum of a function.
- We start at some random point and take steps proportional to the negative of the gradient of the function at the current point.

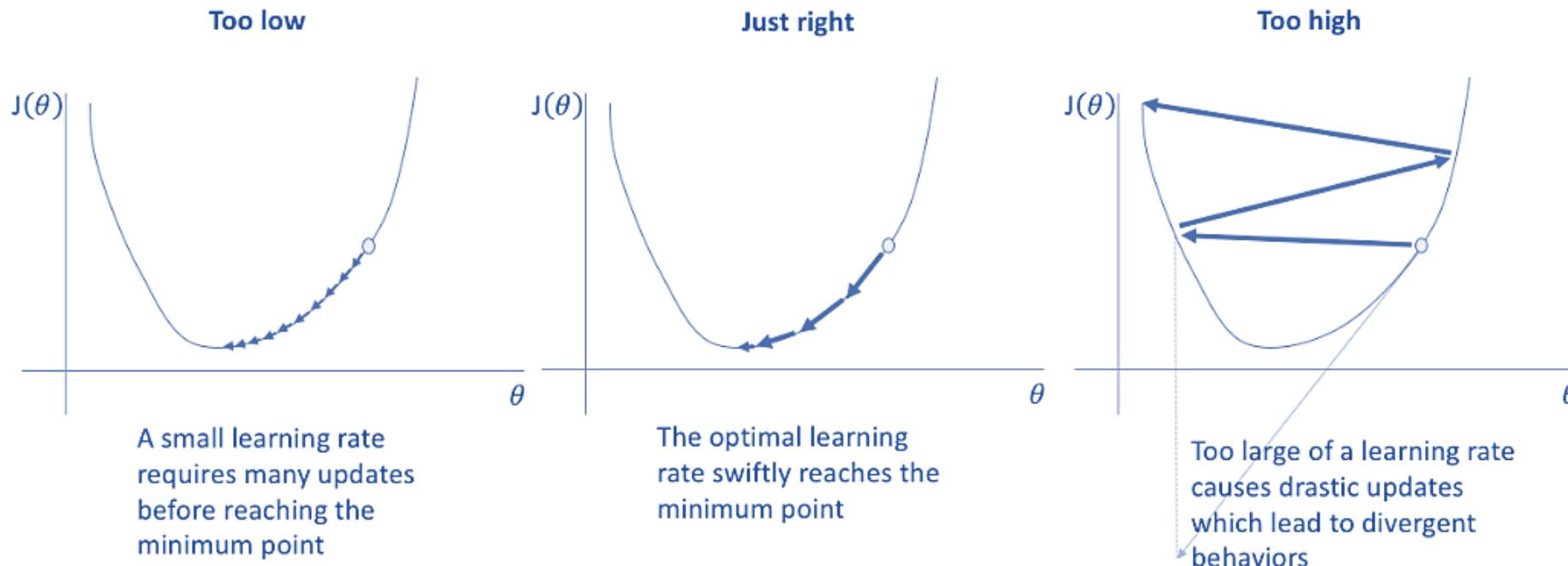
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

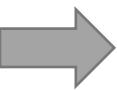
- $\theta_j$  is the model's  $j^{th}$  parameter
- $\alpha$  is the learning rate
- $J(\theta)$  is the cost function (which is differentiable)



# Choice of learning rate

- If  $\alpha$  is **too small**, gradient descent can be **slow**
- If  $\alpha$  is **too large**, the gradient descent can even **diverge**.





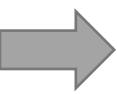
# Beyond Gradient Descent?

**Disadvantages** of gradient descent:

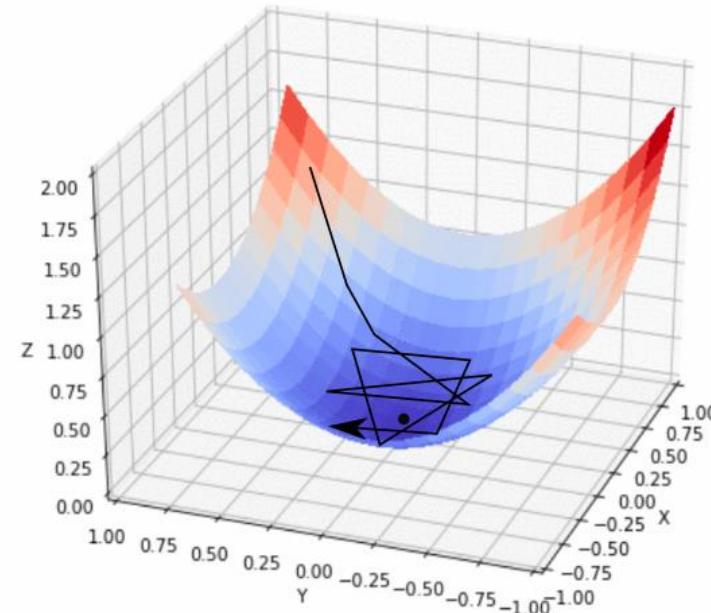
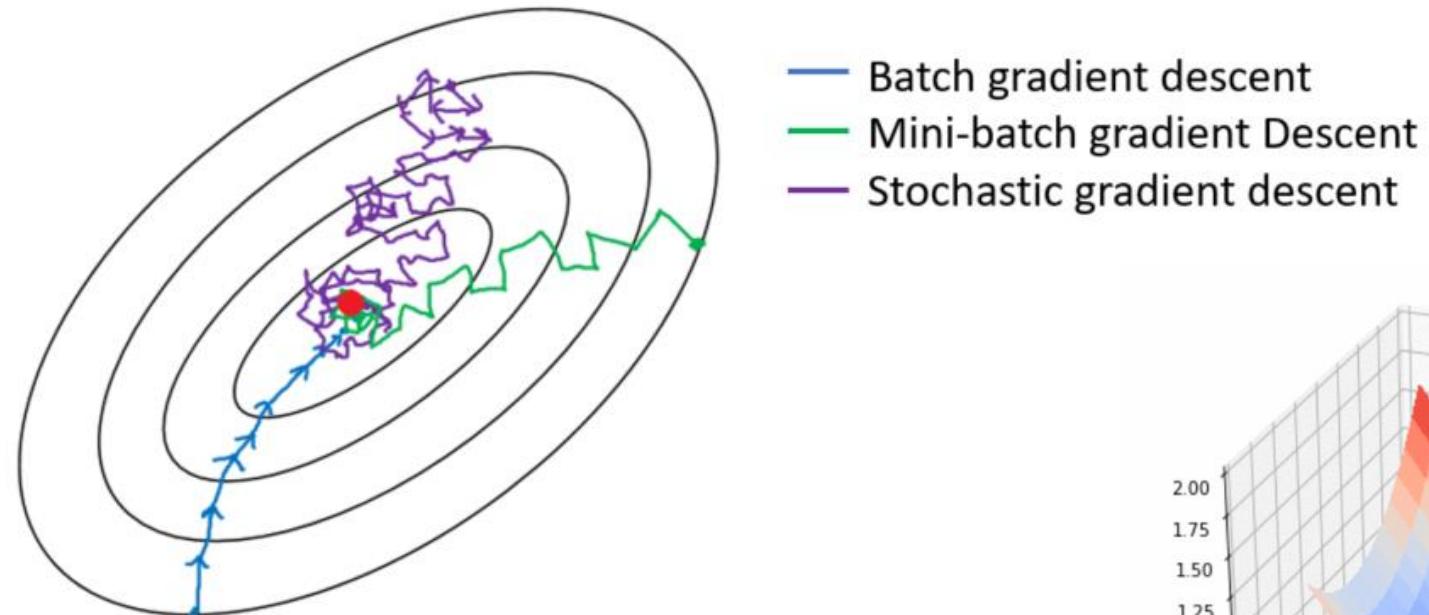
- Single batch: use the entire training set to update parameters!
- Sensitive to the choice of the learning rate
- Slow for large datasets

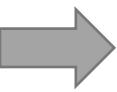
**(Minibatch) Stochastic Gradient Descent**: is a version of the algorithm that speeds up the computation by approximating the gradient using **smaller batches** (subsets) of the training data. SGD itself has various “upgrades”.

↓  
batch size = 1 random observation.



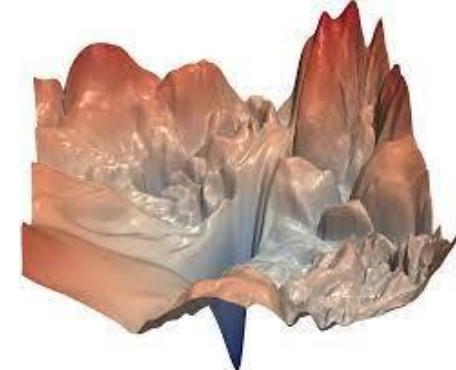
# SGD vs GD





# Beyond SGD?

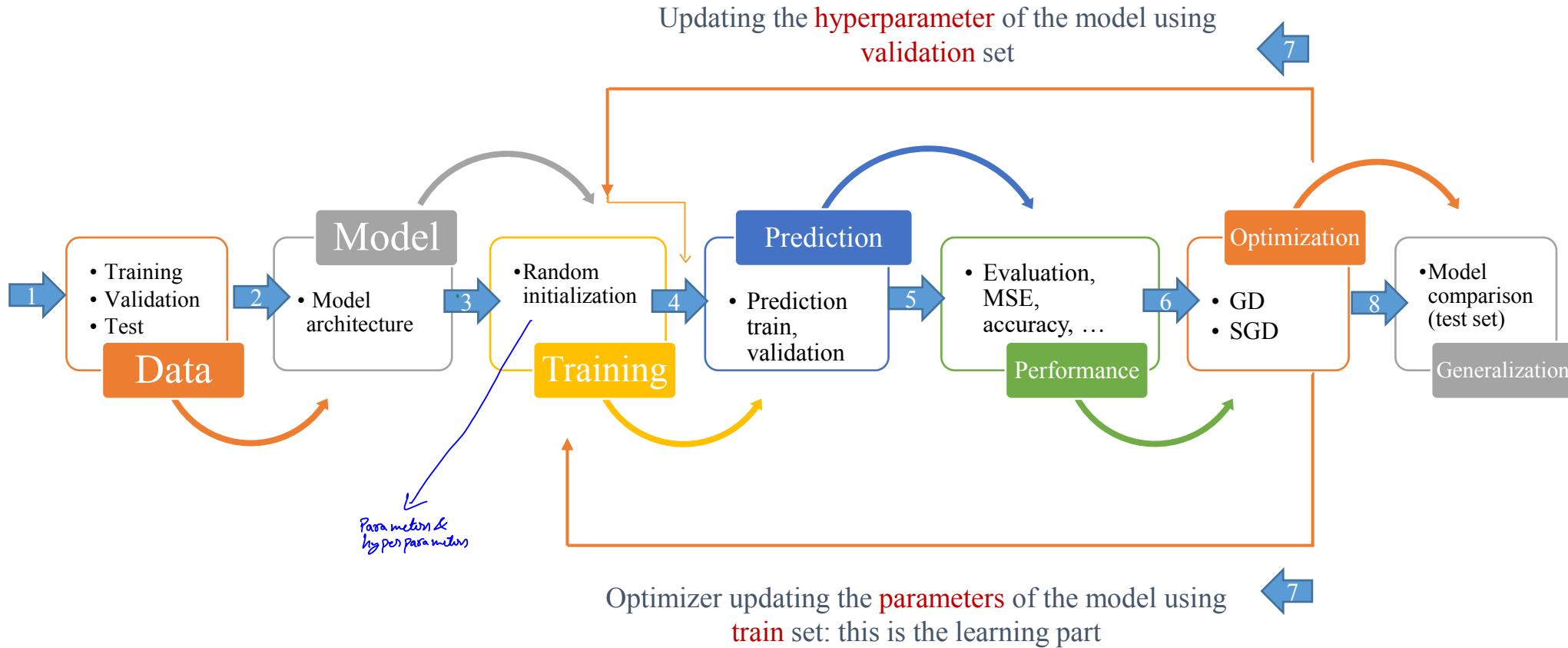
- Loss functions can be difficult to optimize!
- *Visualizing the loss landscape of neural nets*, Li et all, 2018



- **Solution:** Designing an adaptive learning rate that can **adapt** to the loss landscape.
- Rather than just looking at **the current gradient**, consider the **previous weight updates**.
- This is called, **momentum**!
- Examples: **Adam**, Adadelta, Adagrad, **RMSProp**!

Widely used in Neural Networks

# How do machines actually learn?

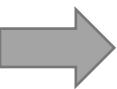


# Module 3 – Part II

## Machine Learning Boosting models

---





# The modern machine learning landscape

- From 2016 to 2020, the entire machine learning and data science industry has been dominated by these **two approaches**:
  1. Deep learning
  2. Gradient boosted trees
- Most practitioners of deep learning use **Keras**, often in combination with its parent framework **TensorFlow**.
- This means you'll need to be familiar with **Scikit-learn**, **XGBoost**, and **Keras**

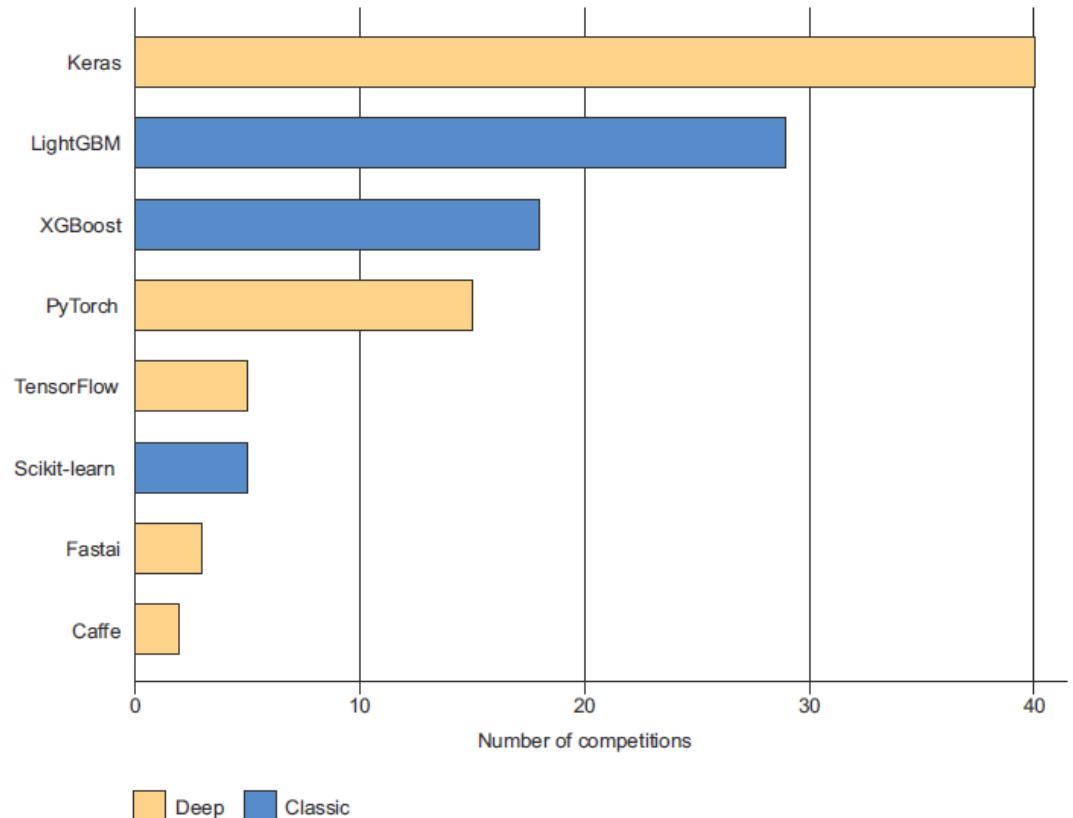
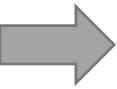
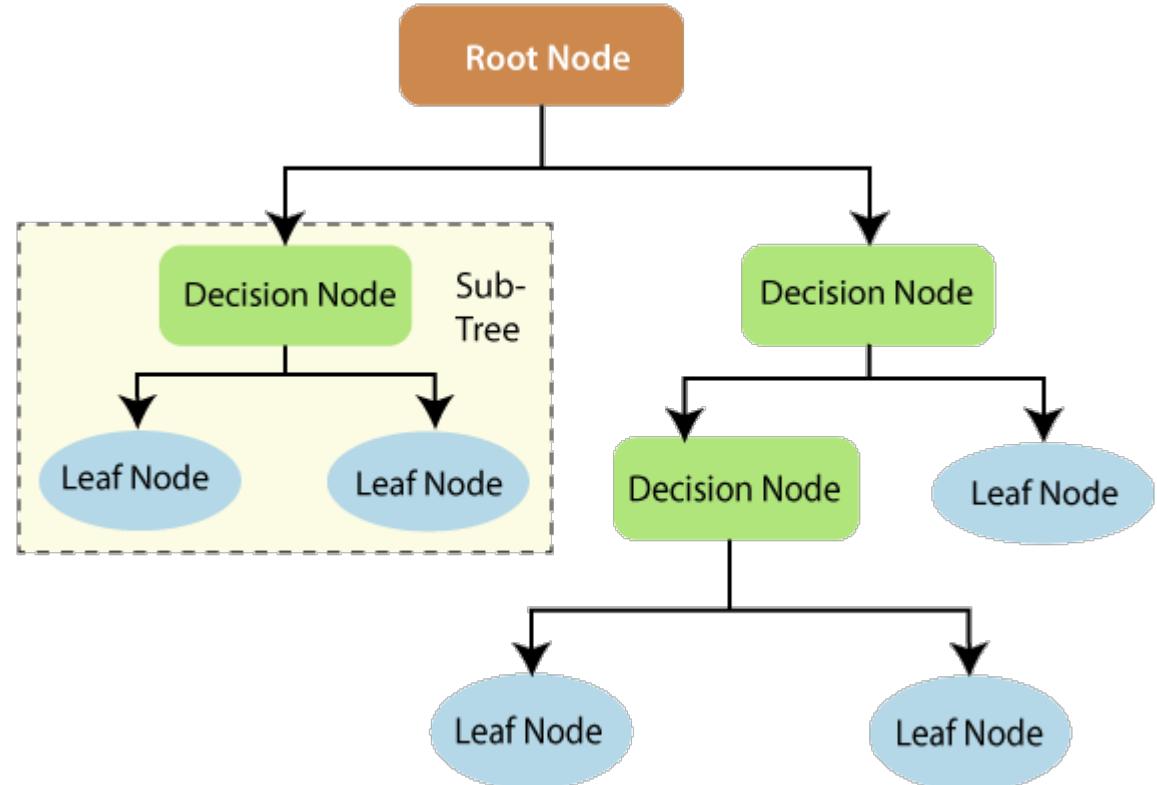


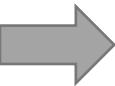
Figure 1.12 Machine learning tools used by top teams on Kaggle



# Decision Trees Fundamental questions

- Four fundamental questions to be answered:
  - 1) What **feature and cut off** to start with?
  - 2) How to **split** the samples?
  - 3) How to **grow** a tree?
  - 4) How to combine trees?

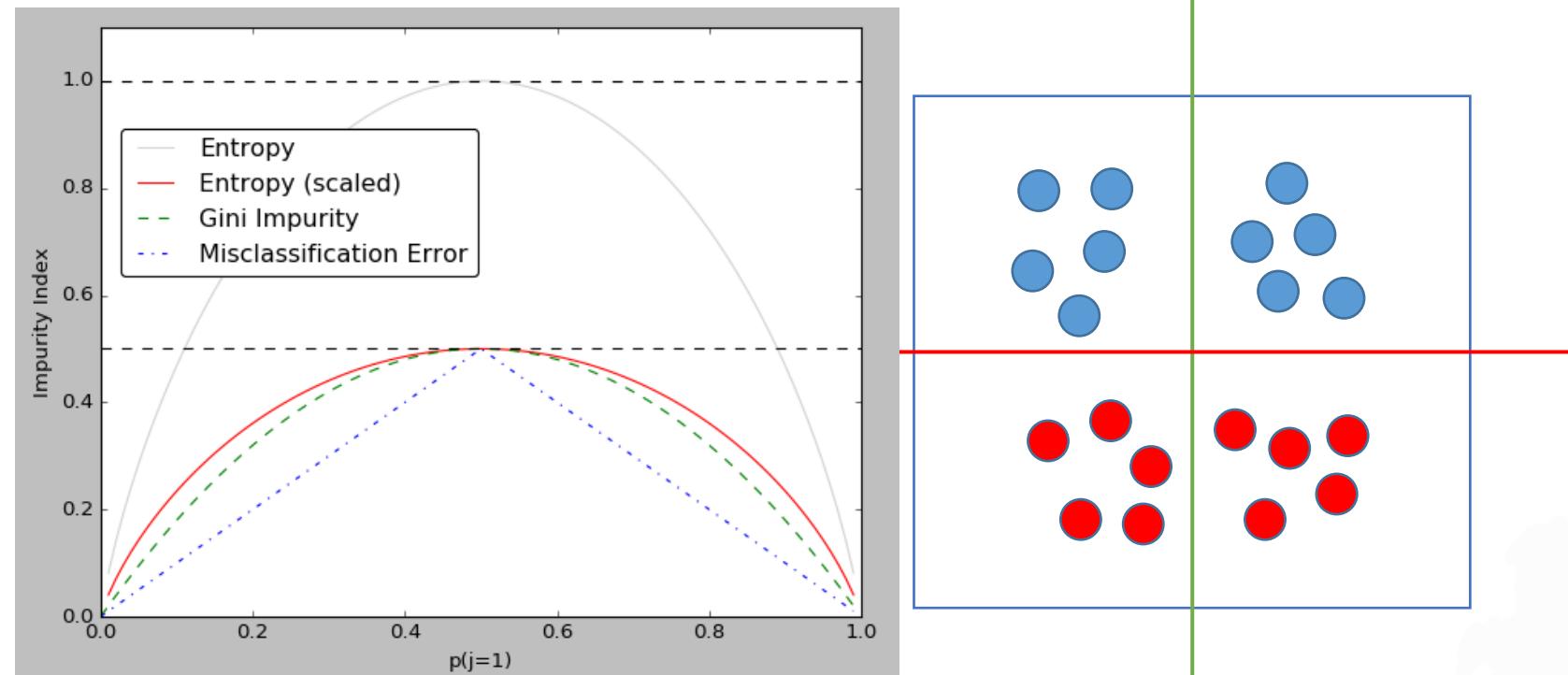


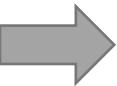


# What feature and cut off to start with?

- Which feature and cut off adds the most information gain (minimum impurity)?
- Regression trees: MSE
- Classification trees:
  1. Error rate
  2. Entropy ✓
  3. Gini Index ✓

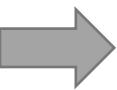
Control how a Decision Tree decides to **split** the data



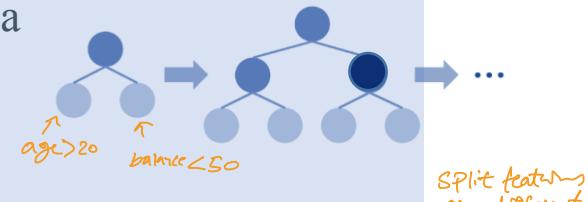
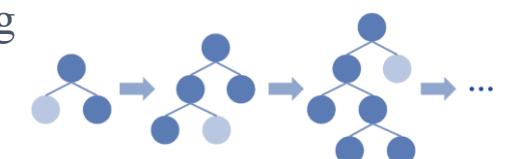
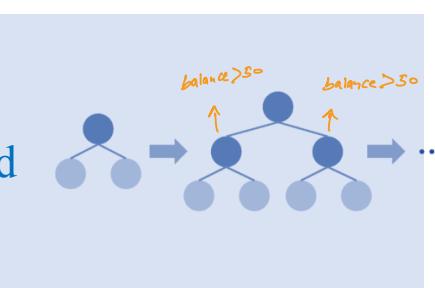


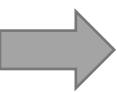
# How to split the samples?

Method	Description
Pre-sorted and histogram based	This method <b>sorts</b> the data and creates <b>histograms</b> of the values before splitting the tree. This allows for faster splits but can result in less accurate trees.
GOSS (Gradient-based One-Side Sampling)	This method uses <b>gradient information</b> as a measure of the weight of a sample for splitting. Keeps instances with <b>large gradients</b> while performing random sampling on instances with <b>small gradients</b> .
Greedy method	This method selects <b>the best split at each step</b> without considering the impact on future splits. This method May result in <b>suboptimal trees</b>



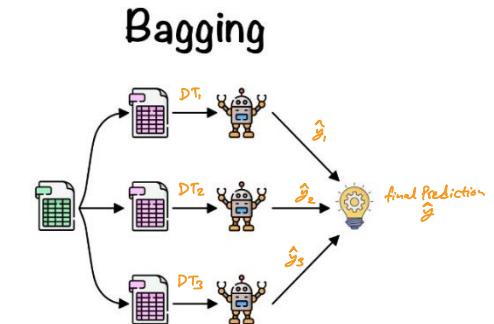
# How to grow a tree?

Algorithm	Description
Depth-Wise Level-Wise	<p>Repeatedly splitting the data along the feature with the highest information gain, until a certain maximum depth is reached. Resulting in a tree with a <b>balanced structure</b>, where all leaf nodes are at the same depth.</p> 
Leaf-wise	<p>Repeatedly splitting the data along the feature with the highest information gain, until all leaf nodes contain only a single class. Resulting in a tree with a <b>highly unbalanced structure</b>, where some branches are much deeper than others.</p> 
Symmetric	<p>Builds the tree by repeatedly splitting the data along the feature with the highest information gain, until a certain stopping criterion is met (e.g. a minimum number of samples per leaf node). Resulting in a more <b>balanced tree structure</b> than leaf-wise growth.</p> 

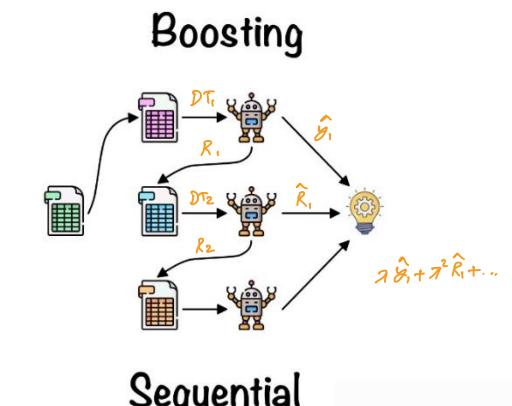


# How to combine trees?

- Bagging consists of creating many “copies” of the training data (each copy is slightly different from another) and then apply the weak learner to each copy to obtain multiple weak models and then combine them.
- In bagging, the bootstrapped trees are **independent** from each other.

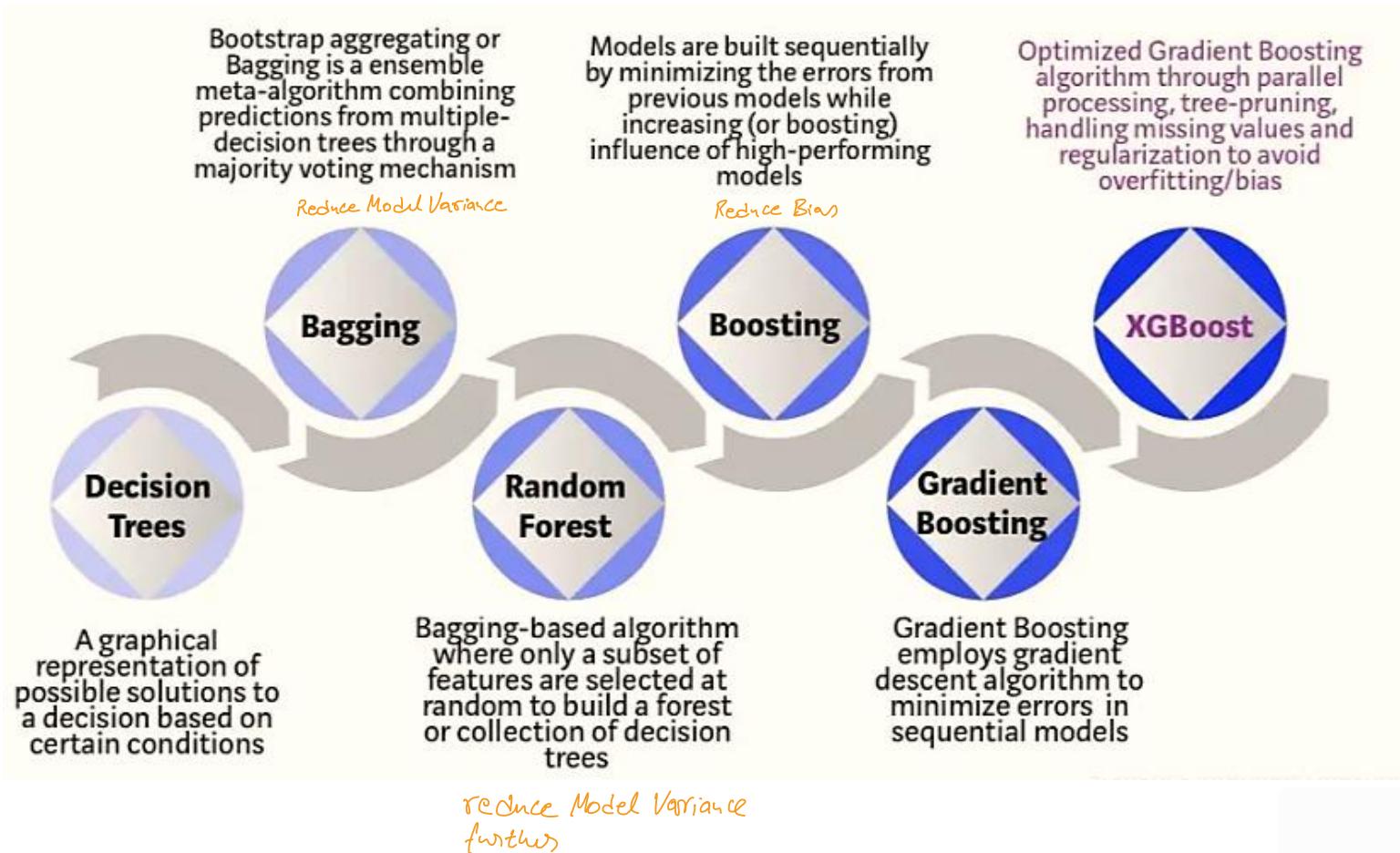
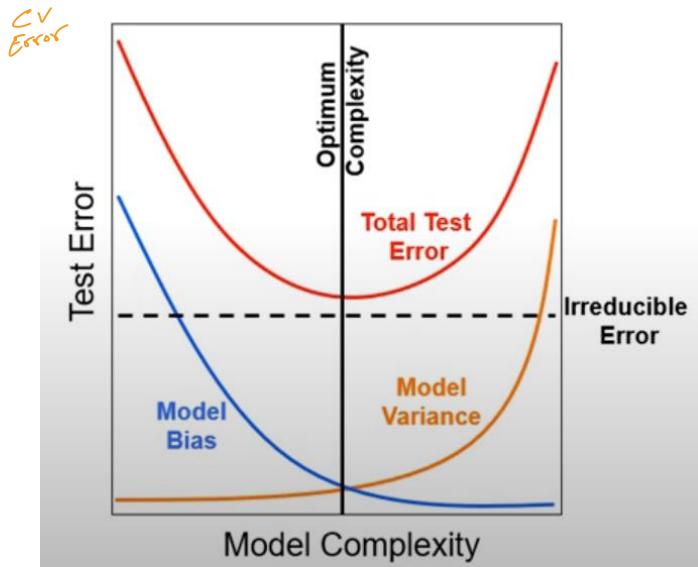


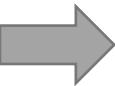
Parallel



Sequential

# → Evolution of XGBoost

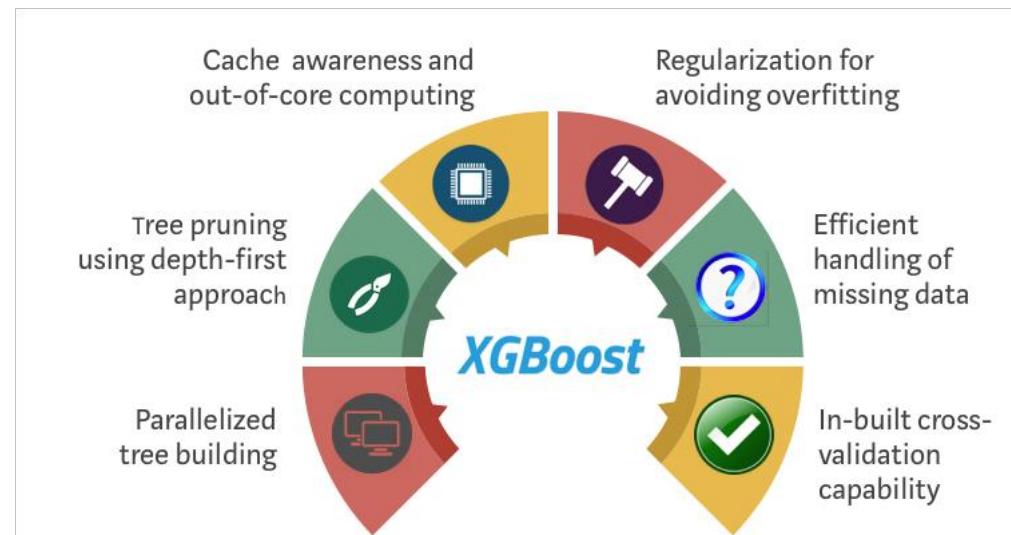


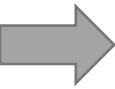


# XGBoost: eXtreme Gradient Boosting

- XGBoost is an open-source gradient boosting library developed by **Tianqi Chen** (2014) focused on developing **efficient** and **scalable** machine learning algorithms.
- **Extreme** refers to the fact that the algorithms and methods have been customized to push the limit of what is possible for gradient boosting algorithms.
- XGBoost includes several other features that can improve **model performance**, such as handling missing values, automatic feature selection, and model ensembling.

*dmlc*  
**XGBoost**

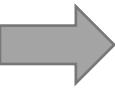




# LightGBM (Light Gradient Boosted Machine)

- LightGBM is an open-source gradient boosting library developed by **Microsoft** (2016) that is fast and efficient, making it suitable for **large-scale learning tasks**.
- LightGBM can handle **categorical features**, but requires one-hot encoding, ordinal encoding or other preprocessing
- LightGBM includes several other features that can improve **model performance**, such as handling missing values, automatic feature selection, and model ensembling.

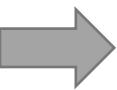




# CatBoost (Category Boosting)

- CatBoost is an open-source gradient boosting library developed by **Yandex** (2017) that is specifically designed **to handle categorical data**.
- CatBoost can handle **categorical features directly**, without the need for one-hot encoding or other preprocessing.
- CatBoost includes several other features that can improve model performance, such as handling missing values, automatic feature selection, and model ensembling.





# XGBoost vs LightGBM vs CatBoost

	XGBoost	LightGBM	CatBoost
Developer	Tianqi Chen (2014)	Microsoft (2016)	Yandex (2017)
Base Model	Decision Trees	Decision Trees	Decision Trees
Tree growing algorithm	Depth-wise tree growth Leaf-wise is also available	Leaf-wise tree growth	Symmetric tree growth
Parallel training	Single GPU	Multiple GPUs	Multiple GPUs
Handling categorical features	Encoding required (one-hot, ordinal, target, label, ...)	Automated encoding using categorical feature binning	No encoding required
Splitting method	Pre-sorted and histogram based	GOSS (Gradient based one-side sampling)	Greedy method

*dmlc*  
**XGBoost**

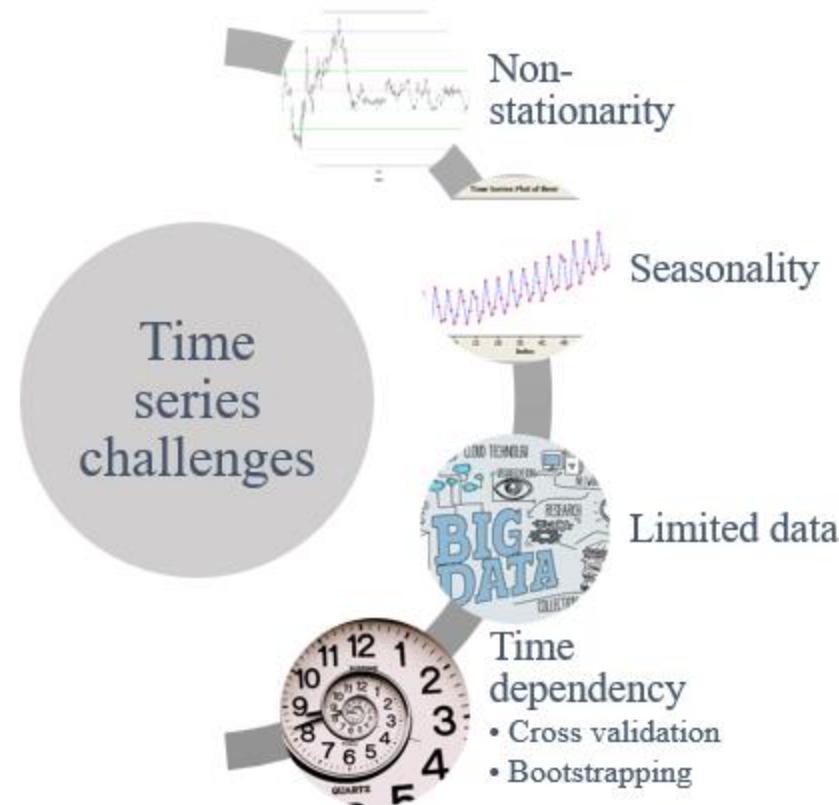
 **LightGBM**

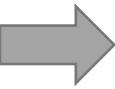
 **CatBoost**

# Module 3 – Part III

## Challenges in Time Series Machine Learning

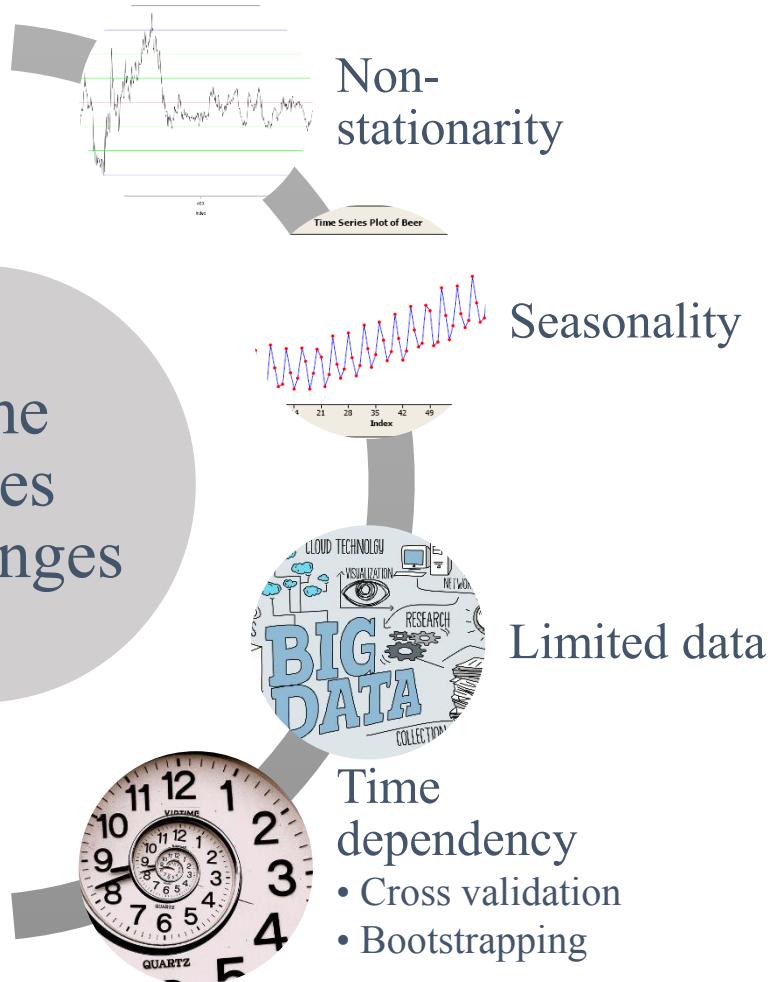
---

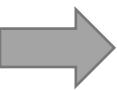




# Challenges in Time Series Machine Learning

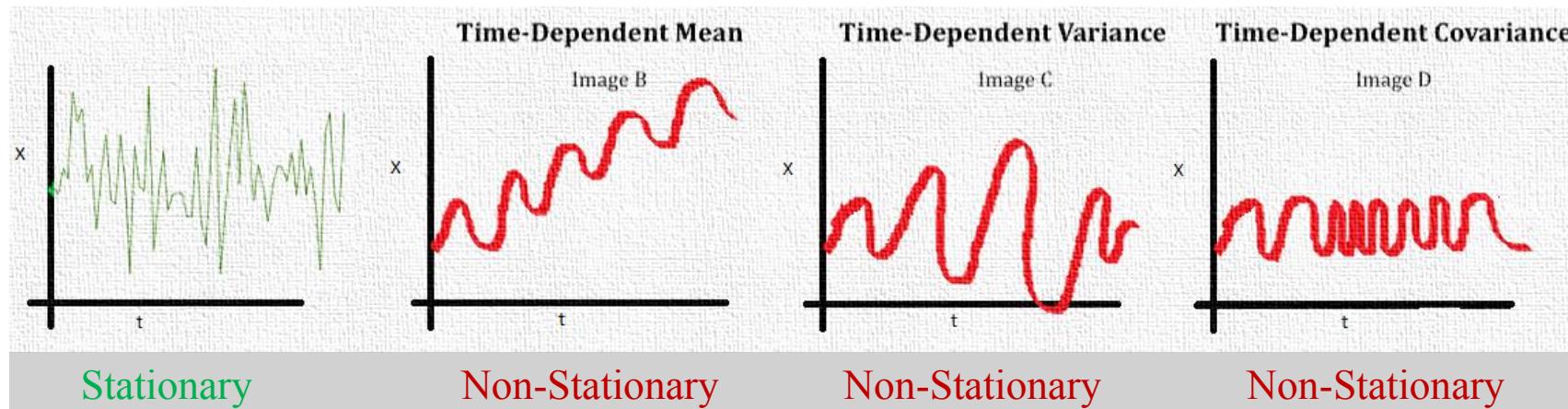
## Time series challenges



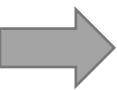


# Stationarity

- Stationary vs Non-Stationary Data. What makes a data set **Stationary**?
- In a stationary timeseries, the statistical properties **do not depend** on the time



- Data with **trend** and **seasonality** are NOT stationary!

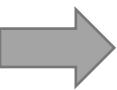


# Time Series Cross Validation

- With time series data, we **cannot shuffle** the data! TS data is not IID.
- We also need to avoid **data leakage**!

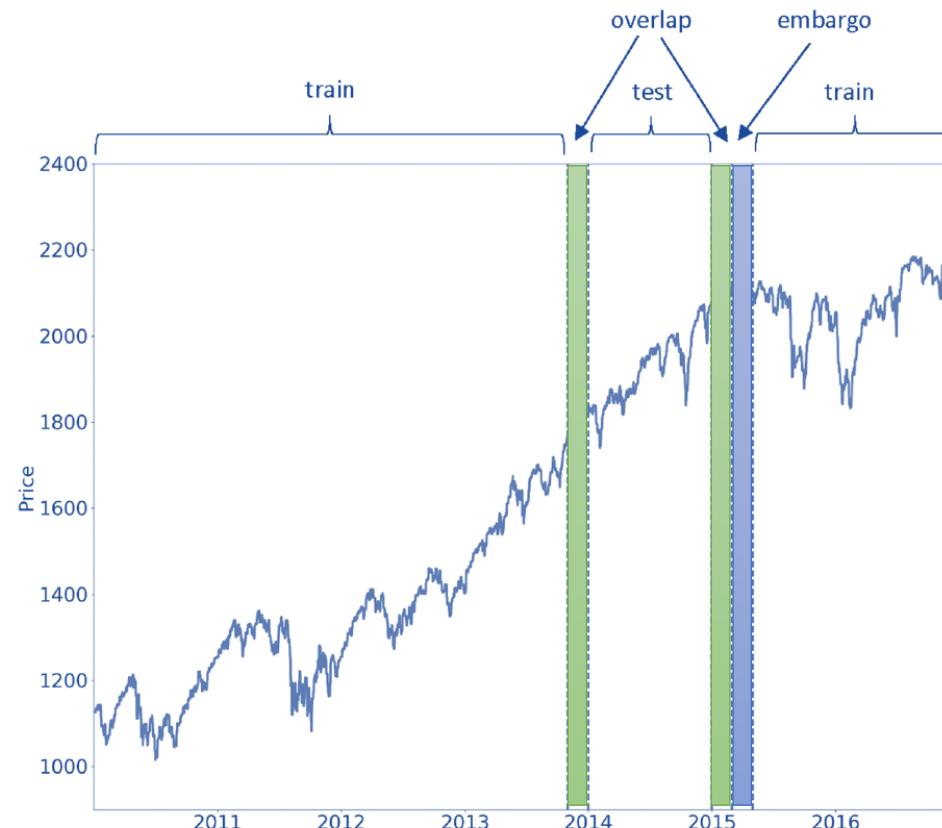
- The main time series CV methods are:
  - Purged K-Fold CV**
  - Walk forward **rolling / expanding window**
  - Combinatorial purged CV**



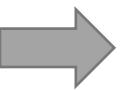


# Purged K-Fold CV

- **Leakage** takes place when the training set contains information that also appears in the testing set.
- Leakage will enhance the model performance
- Solution: **Purging** and **Embargoing**
- Purged K-Fold CV: Adding purging and embargoing whenever we produce a train/test split in K-Fold CV.

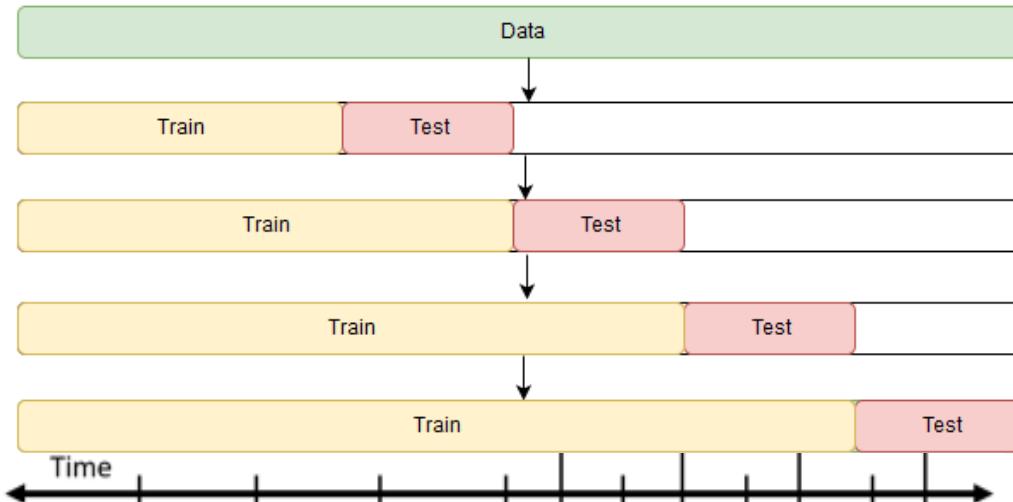


**FIGURE 7.3** Embargo of post-test train observations



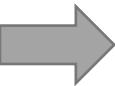
# Walk Forward Cross Validation

Walk forward cross validation  
Expanding windows



Walk forward cross validation  
Rolling windows

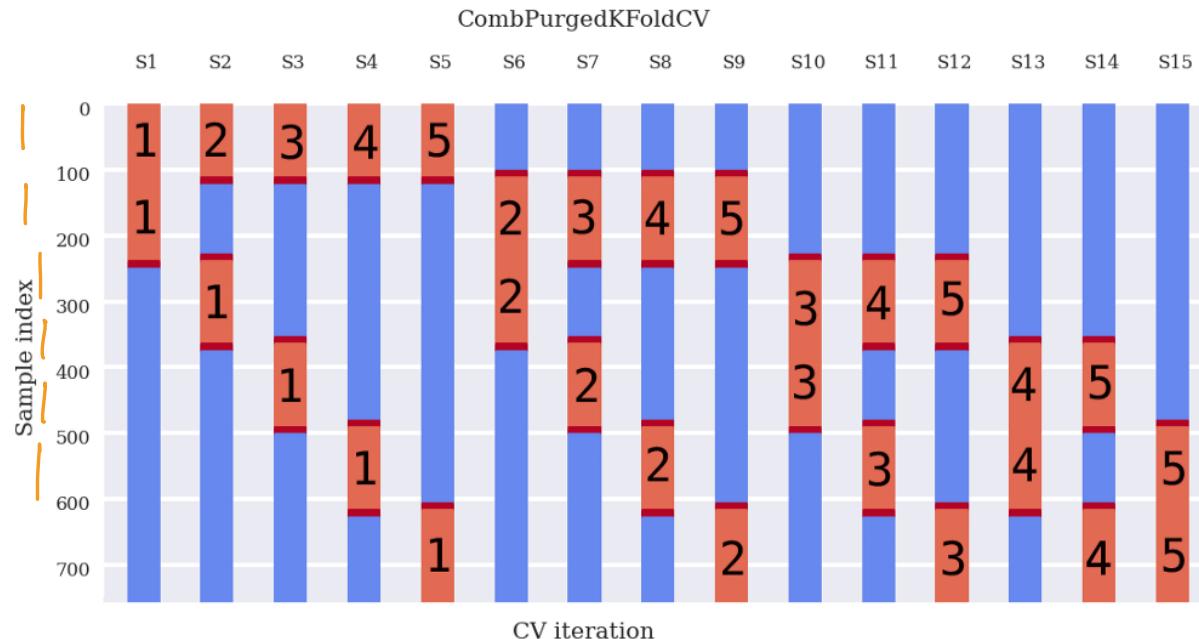


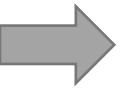


# Combinatorial Purged Cross Validation (CPCV)

- The goal is to generate **multiple unique back-test path** that span the entire data set.
- In each path, we can look at the model's **OOS performance** for the entire time period.

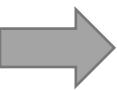
Groups = 6  
Test group = 2  
 $\binom{6}{2} = 15$  splitting ways





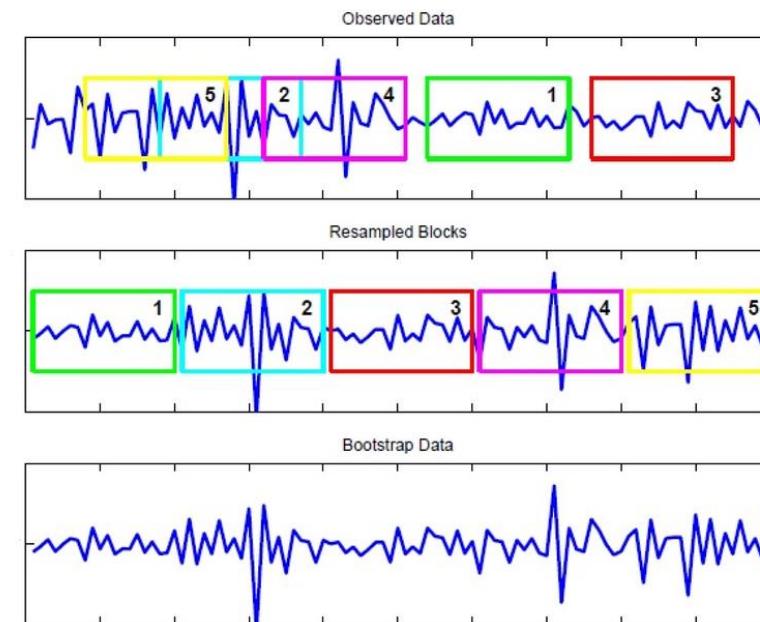
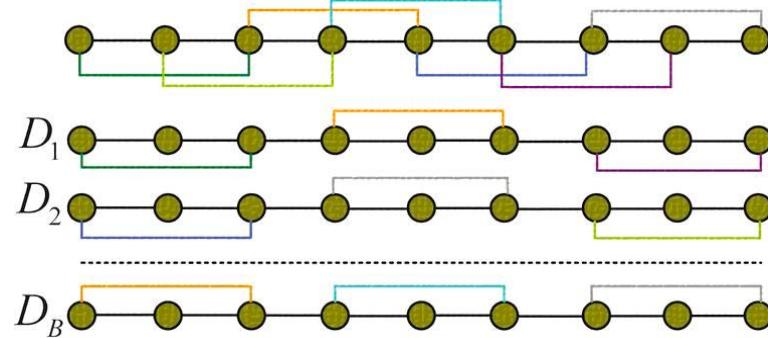
# Time Series Bootstrapping

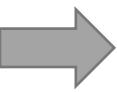
- IID bootstrapping (random sample with replacement) does not work for time series data with temporal dependency.
- Time series Bootstrapping methods:
  - **Parametric** (based on models with **iid residuals** and resampling from residuals.  
Example: ARIMA bootstrap)
  - **Non-parametric block** bootstrap (data is directly resampled. Assumption: blocks can be samples so that they are **approximately iid**)
    - Moving Block Bootstrap (MBB)
    - Circular Block Bootstrap (CBB)
    - Stationary Bootstrap (SB)



# Moving Block Bootstrap (MBB)

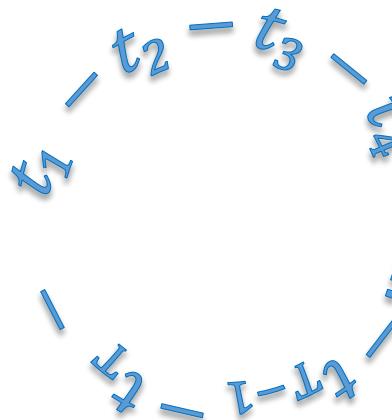
- Moving Block Bootstrap, samples **overlapping fixed size** blocks of  $m$  consecutive observations.
- Blocks starts at indices  $1, \dots, T-m+1$

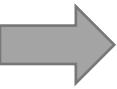




# Circular Block Bootstrap (CBB)

- CBB is a simple extension of MBB which assumes the data live on a circle so that  $y_{T+1} = y_1$ ,  $y_{T+2} = y_2$ , etc.
- CBB has better finite sample properties since all data points get sampled with equal probability.

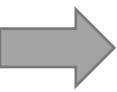




# Stationary Bootstrap (SB)

---

- In SB, the **block size** is no longer fixed.
- Chooses an **average block size of  $m$**  rather than an exact block size.
- Popularity of SB stems from difficulty in determining optimal  $m$
- Once applied to stationary data, the resampled **pseudo time series** by SB are **stationary**.  
This is not the case for MBB and CBB.



# Road map!

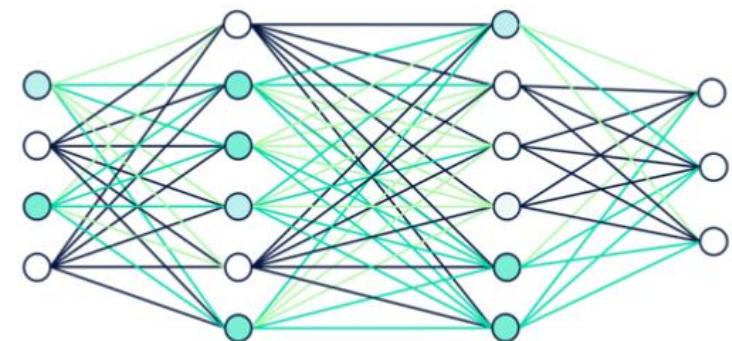
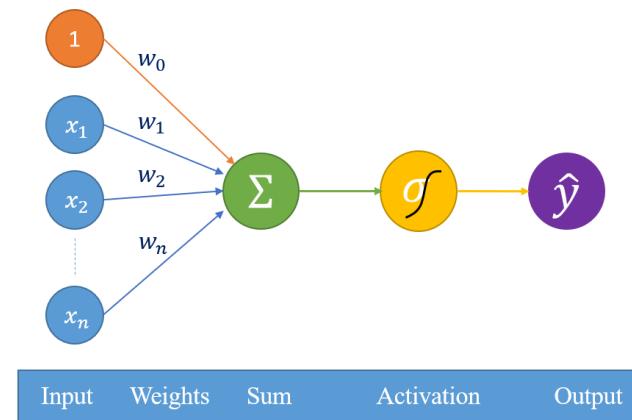
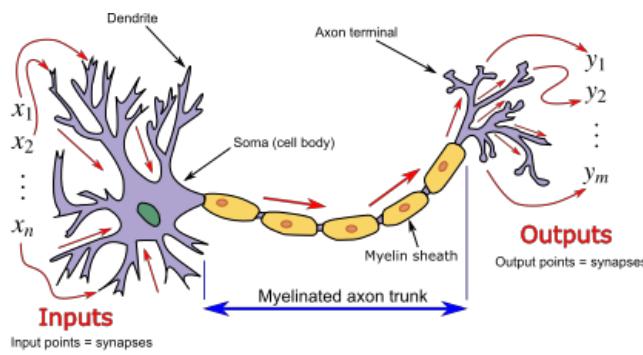
- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- **Module 4- Deep Neural Networks (NN and DNN)**
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- Module 6- Deep Sequence Modeling (RNN, LSTM)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

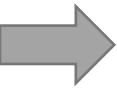


# Module 4 - Part I

## Deep Neural Networks

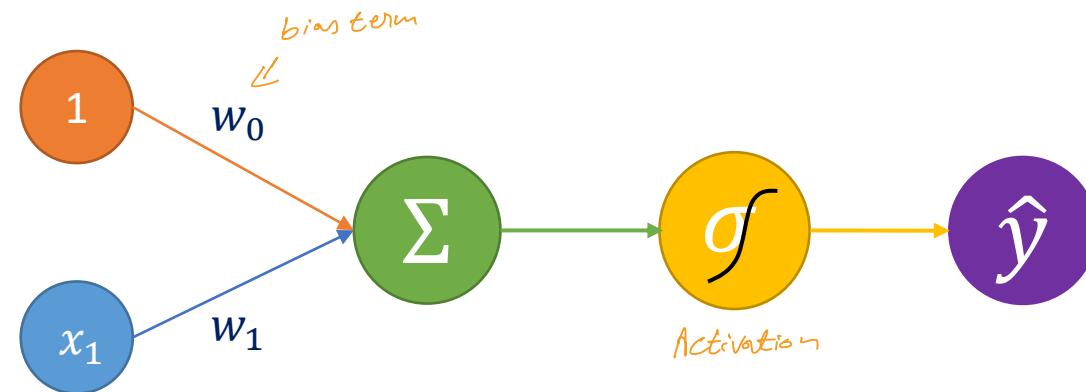
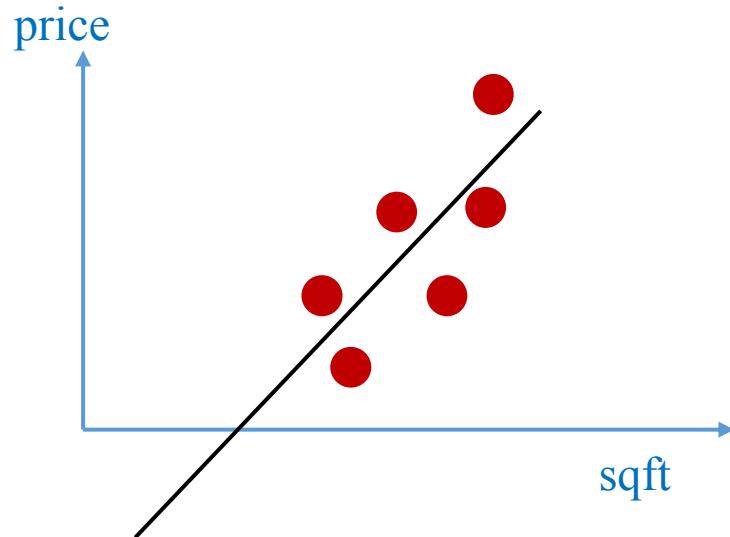
### Basics





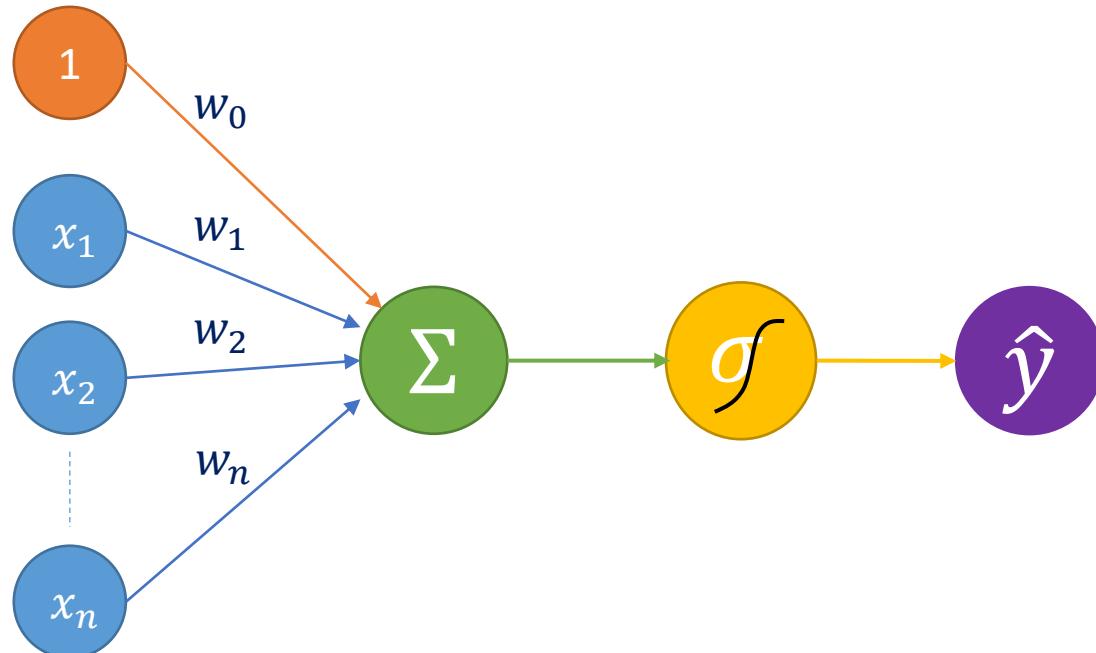
# A simple example!

- Consider the housing price prediction!  $price = f(sqft)$

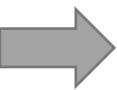


# → Forward Propagation

- Forward propagation is the process of calculating the output of a neural network, given an input.
- $w_0$  is the **bias** term which allows shifting  $\Sigma$  to the left or right.



Input    Weights    Sum    Activation    Output



# What are Neurons?

- Neurons, are the simplest elements or building blocks in a neural network.  
They are inspired by biological neurons that are found in the human brain.
- What happens inside the neurons?

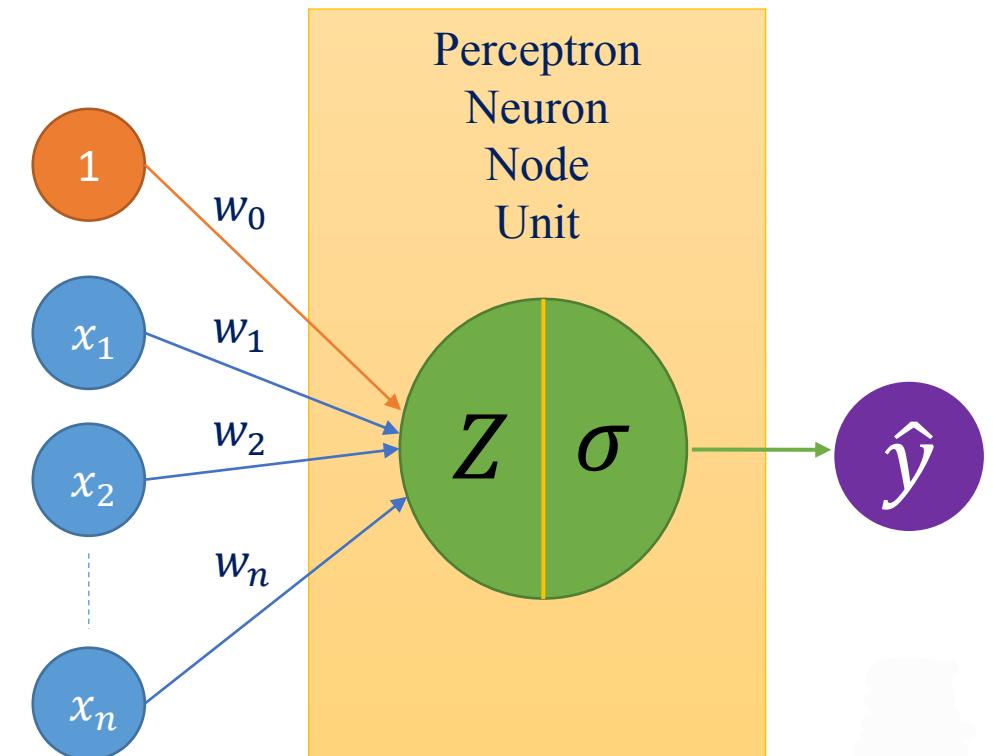
Activation function

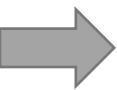
$$\hat{y} = \sigma(w_0 + \sum_{i=1}^n w_i x_i)$$

$$\hat{y} = \sigma(w_0 + W^T X)$$

$$\hat{y} = \sigma(Z)$$

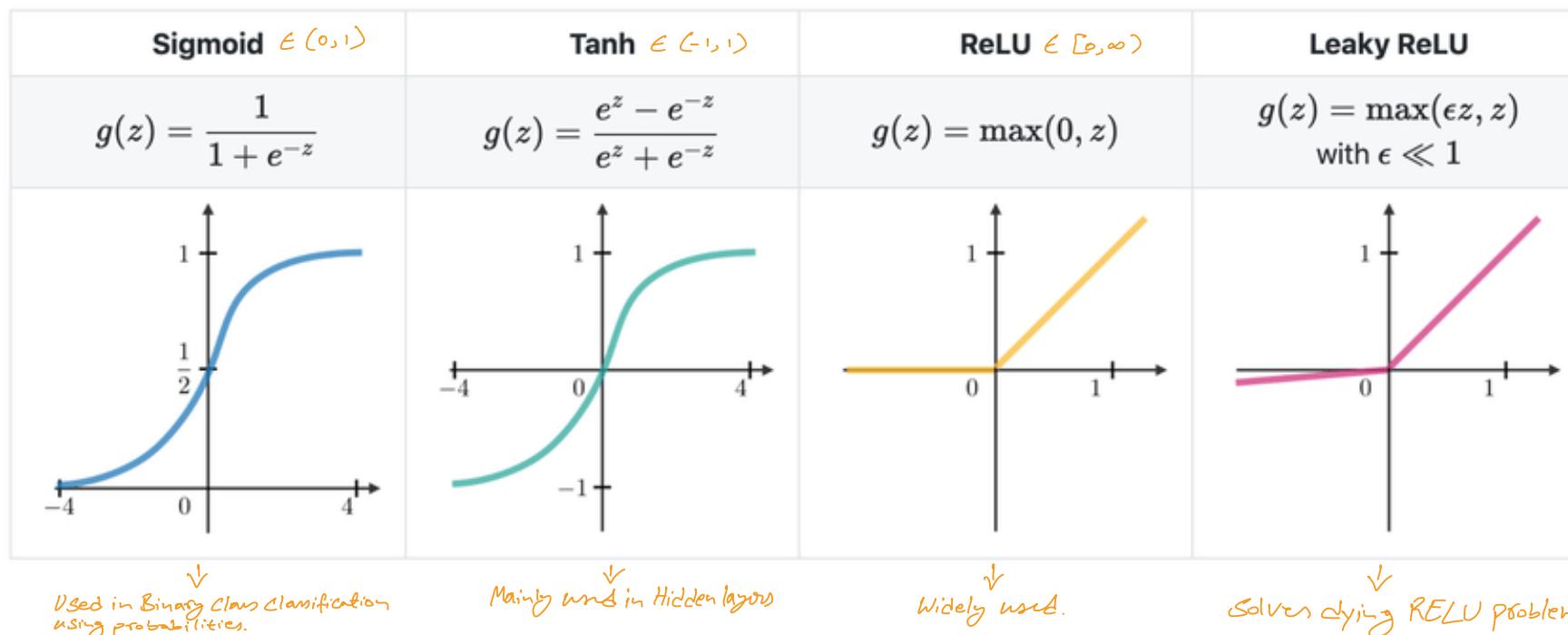
Bias

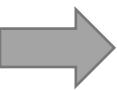




# Activation function

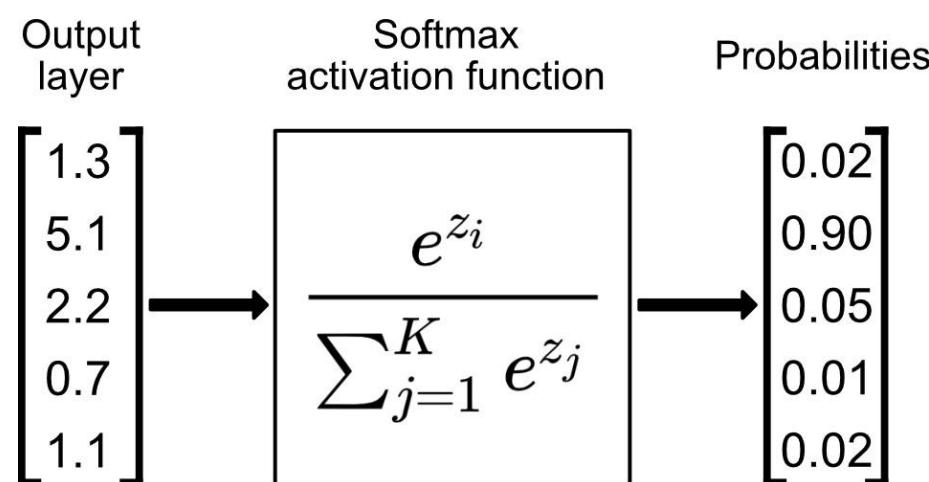
- An activation function is a mathematical function that is applied to the output of each neuron in a neural network
- Activation functions allow the network to learn **non-linearities** and **complex patterns** from the data and make accurate predictions



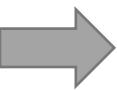


# Softmax Activation function

- Softmax activation function is often used in **classification** tasks, where the goal is to predict which of a fixed set of classes (**more than two classes**) a particular sample belongs to.
- The Softmax function takes in a **vector of real numbers** and converts it into a **probability distribution**, where the sum of all the probabilities is equal to 1.



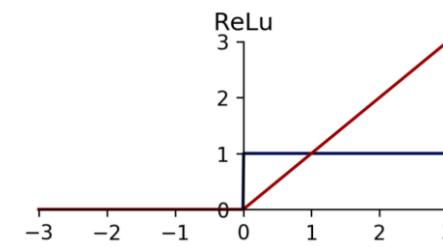
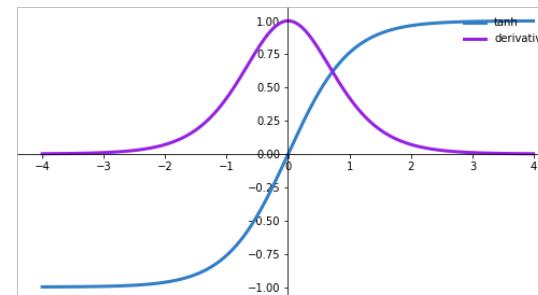
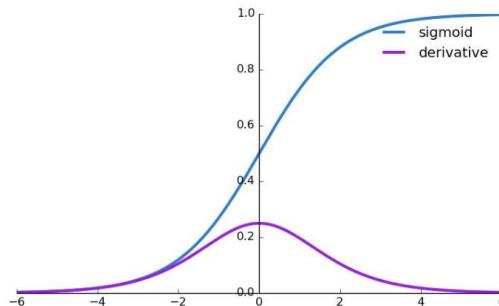
• Widely used in multi-class classifications at the very last layer.  
Softmax is differentiable.



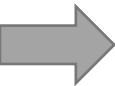
# Which activation function?

Again, there is no “right” answer! However:

- For the hidden layers, almost always **tanh** is better than **sigmoid** because it center the data for the next layer.
- One downside of both sigmoid or tanh is that the gradient is very small for extreme values.

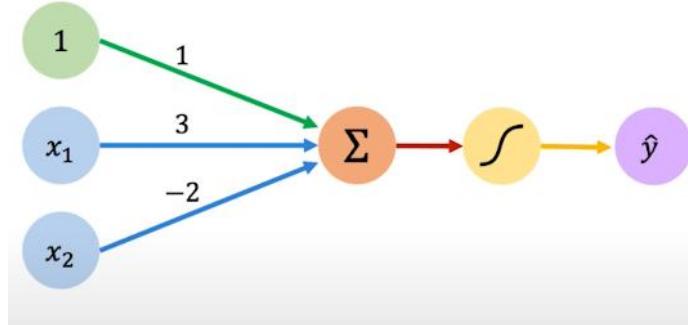


- In practice, RELU works **better/faster** than sigmoid or tanh for hidden layers.
- Leaky RELU might work better than RELU, but if we have enough number of hidden layers, RELU is just fine.
- Sigmoid is mostly used for output layer!



# Sigmoid activation function! example

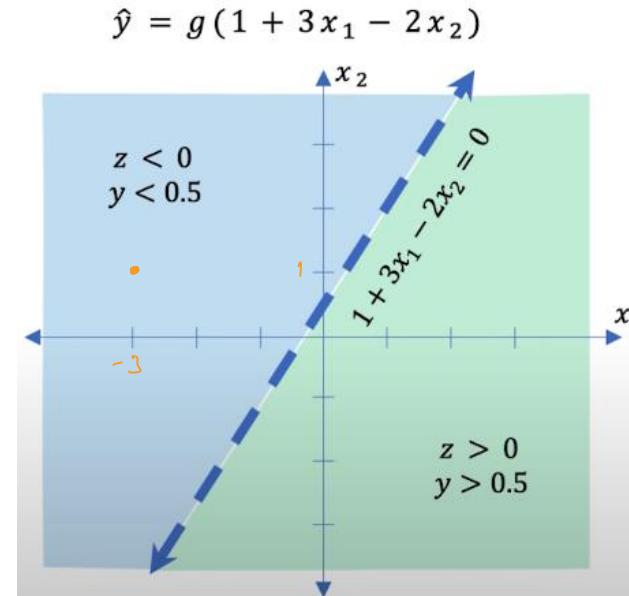
(Binary Class Classification (Blue vs Green))



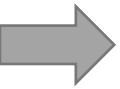
We have:  $w_0 = 1$  and  $\mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{x}^T \mathbf{w}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

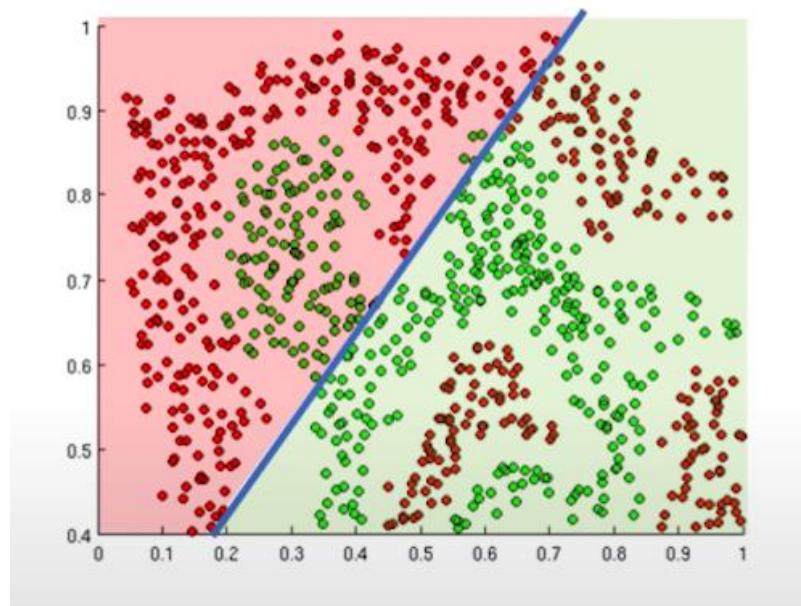


$$g(z) = \frac{1}{1 + e^{-(1+3x_1-2x_2)}} = \frac{1}{1 + e^{-(1+3(-3)+2)}} = \frac{1}{1 + e^{-10}} < 0.5 \Rightarrow \text{Blue}$$

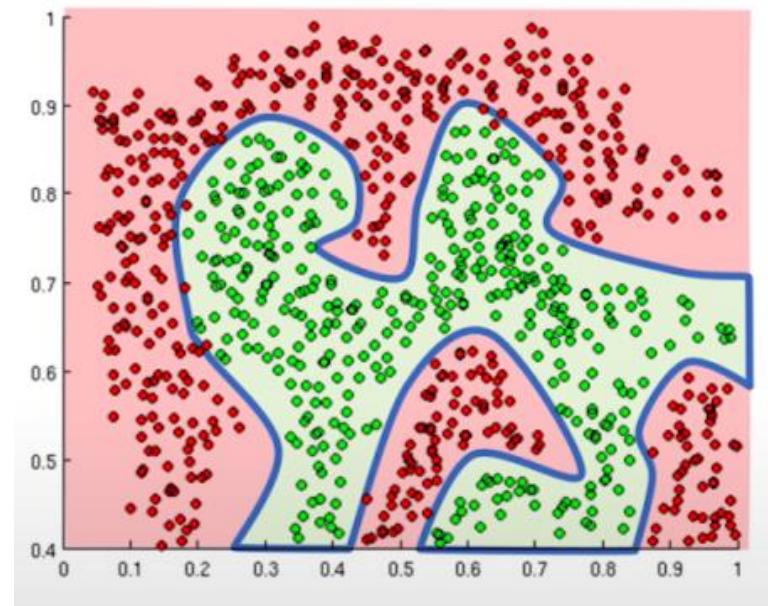


# Why do we need non-linear activation functions?

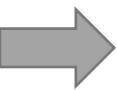
- If we use **linear** activation function  $\sigma(z) = z = w_0 + W^T X$ , then the NN will **always output a linear function of the inputs** regardless of the number of neurons or hidden layers used.
- Linear activation functions produce linear decision boundaries!
- Activation functions allow the network to approximate **complex patterns**!



Linear Decision Boundary

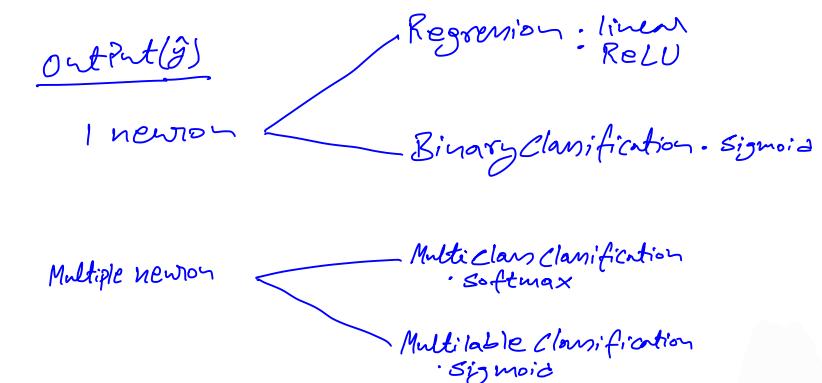
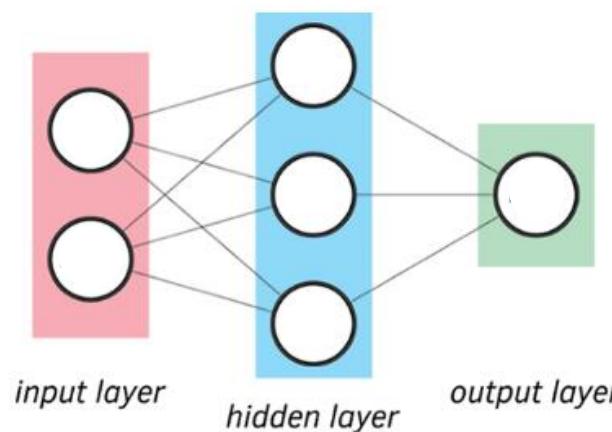
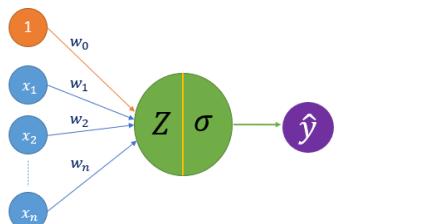


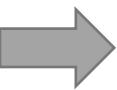
Non-Linear Decision Boundary



# Building a Neural Network

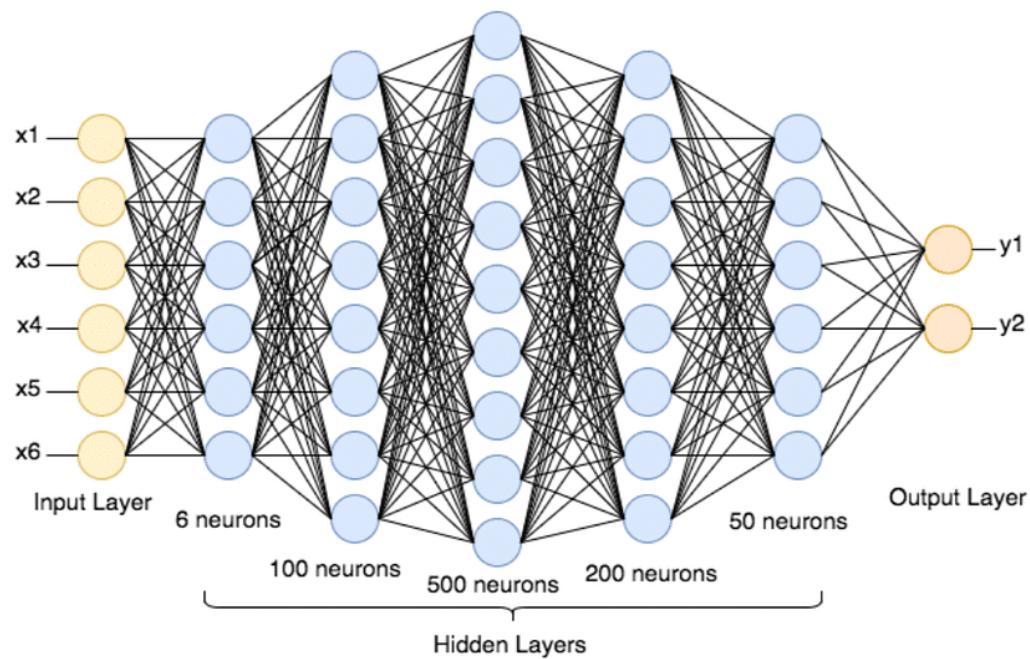
- A neural network is a type of machine learning model that is composed of **layers** of interconnected "neurons" which process and transmit information.
- Each neuron receives input from other neurons, processes it using a **nonlinear activation function**, and then transmits the output to other neurons in the **next layer**. The output of the **final layer** is the **prediction** made by the neural network.
- Because all the inputs are **densely** connected to all outputs, these layers are called **Dense** layers.
- The output layer can be either **single** output or **multiple** output.





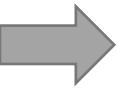
# Building a Deep Neural Network

- Deep neural networks are neural networks **with a large number of layers**, typically consisting of multiple hidden layers.
- Deep neural networks can learn and **model very complex patterns** in data.
- DNNs have been successful in a wide range of applications, including image and speech recognition, natural language processing, and machine translation.



```
import tensorflow as tf

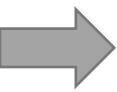
# Define the model architecture
model=tf.keras.Sequential([
    tf.keras.layers.Dense(6 , activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(500, activation='relu'),
    tf.keras.layers.Dense(200, activation='relu'),
    tf.keras.layers.Dense(50 , activation='relu'),
    tf.keras.layers.Dense(2 , activation='sigmoid'),
])
```



# Training a Neural Network, Backpropagation

- To train a neural network, we present it with many examples and **adjust the weights and biases** of the connections between neurons so that the network can accurately predict the output for each example and minimize **the loss function**.
- This process is known as **backpropagation**, and it is done using an optimization algorithm such as stochastic gradient descent.
- The **loss function** quantifies the distance between actuals and predictions. It provides **feedback** to the NN.
- Examples: MSE, Binary Cross Entropy, (Sparse) Categorical Cross Entropy, ...  
*if you use one hot encoding*
- For the full list, visit <https://keras.io/api/losses/#available-losses>





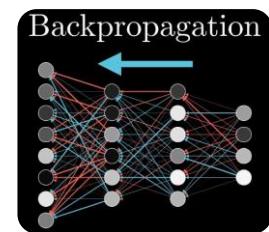
# Backpropagation

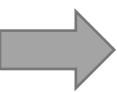
- Backpropagation is a **supervised learning algorithm** that adjusts the weights and biases of the connections between neurons in the network to minimize **the loss function**.
- Steps:  
$$\overset{\text{Optimal Value for weights}}{\longrightarrow} \mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$
- 1) Feed the input data through the neural network to compute the output.
- 2) Calculate the **loss** or **error** between the predicted output and the true output.
- 3) Propagate the error back through the network using the **chain rule** of calculus to calculate the gradient of the loss function with respect to the weights and biases.
- 4) **Update the weights** and biases using the gradient descent algorithm.
- 5) Repeat the process until the loss reaches a satisfactory level or a predetermined iterations.



$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

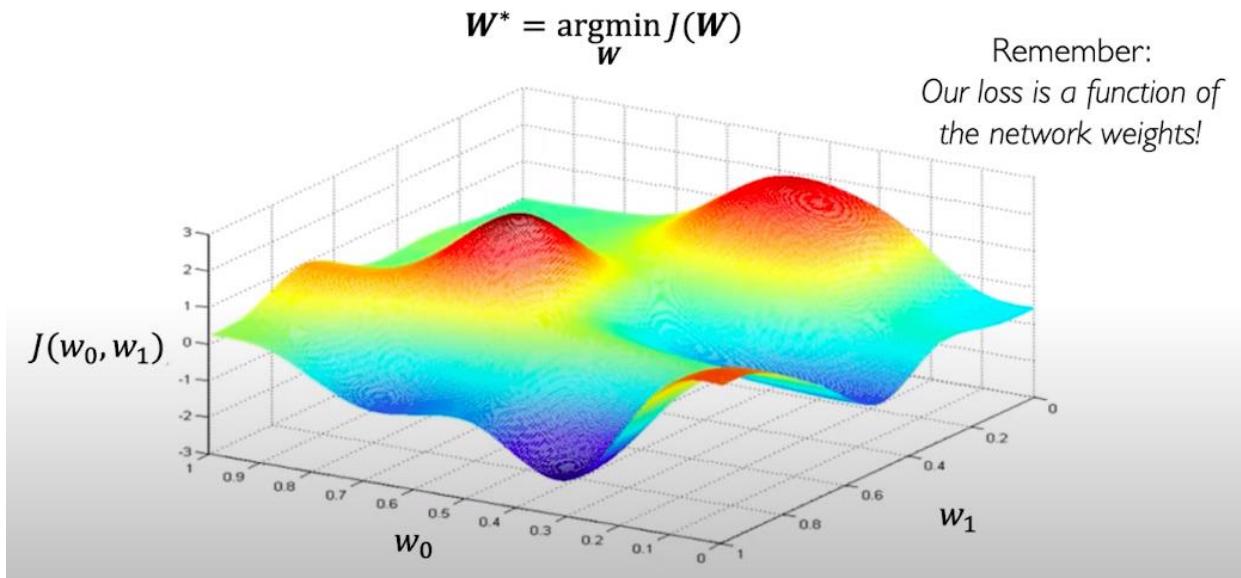
$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$





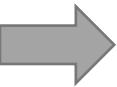
# Loss optimization

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$



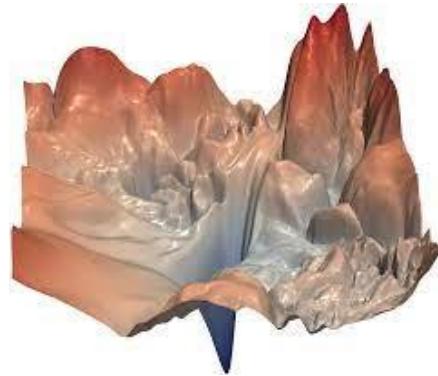
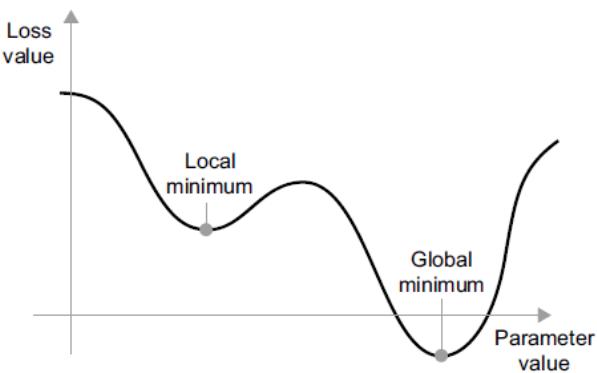
## Mini-batch GD

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

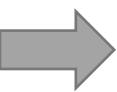


## DNN loss functions can be difficult to optimize!

- *Visualizing the loss landscape of neural nets*, Li et all, 2018



- **Solution:** Designing an adaptive learning rate that can **adapt** to the loss landscape. Rather than just looking at **the current gradient**, take into account the **previous weight updates**. This is called, **momentum**!
- Examples: **Adam**, Adadelta, Adagrad, **RMSProp**!



# Gradient Descent Algorithms

- Available optimizers in Keras:

- SGD
- RMSprop
- Adam
- AdamW
- Adadelta
- Adagrad
- Adamax
- Adafactor
- Nadam
- Ftrl



- ① Sequential (less flexible)
- ② Functional API's (More flexible)
- ③ Subclass

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential()
model.add(layers.Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(layers.Activation('softmax'))
```

```
opt = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

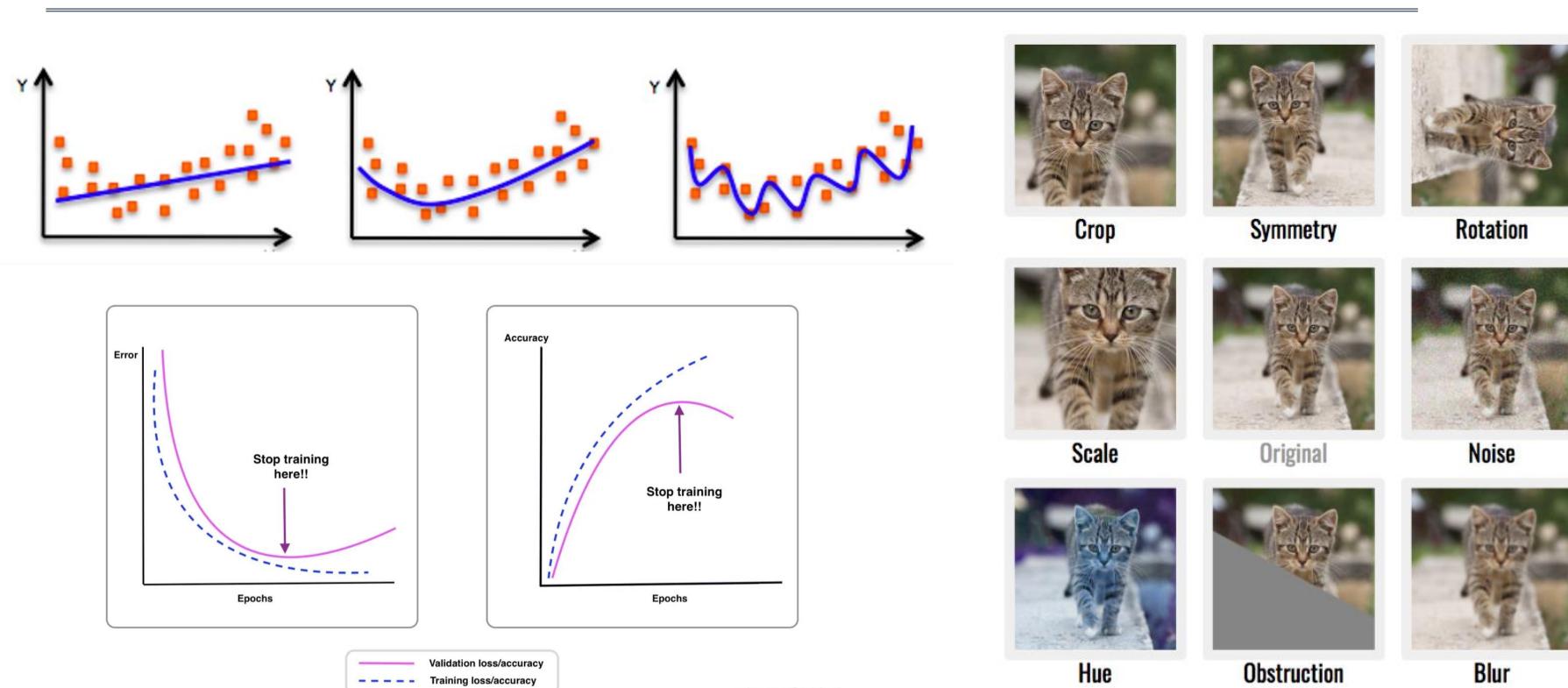
```
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-2,
    decay_steps=10000,
    decay_rate=0.9)
optimizer = keras.optimizers.SGD(learning_rate=lr_schedule)
```

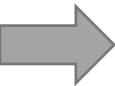
- Which optimizer? There is no “right” answer.

# Module 4 - Part II

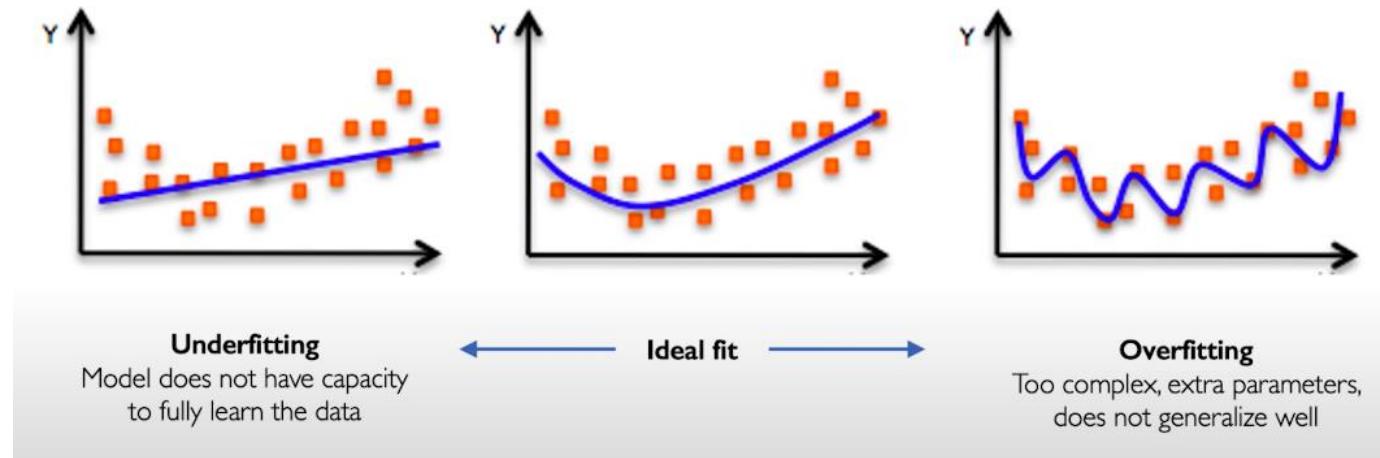
## Deep Neural Networks

### Regularization

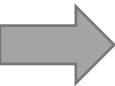




# How to handle overfitting in DNN?



- Regularization refers to a set of techniques that are used to prevent overfitting by **discouraging complex models**.
- Regularization improve **generalization** of our model on **unseen data**.
- Methods include, L1, L2, drop out, Early stopping and Data augmentation.



# Deep Learning Regularization techniques

**L1 regularization:** This technique adds a penalty term to the objective function that is proportional to the absolute value of the model weights. This results in a sparse model, with many weights being set to zero.

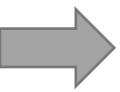
$$J(w) + \sum |w_i|$$

**L2 regularization:** This technique adds a penalty term to the objective function that is proportional to the square of the model weights. This results in a model with small, non-zero weights.

**Dropout:** This technique randomly sets a fraction of the model weights to zero during training, which helps to prevent overfitting

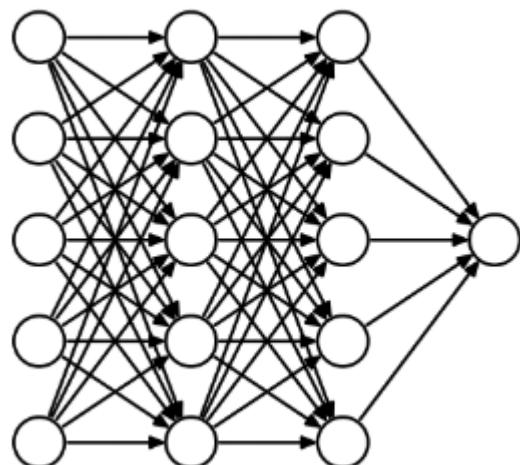
**Early Stopping:** This technique involves training the model until the performance on a validation set begins to degrade, and then stopping the training at that point. This helps to prevent the model from continuing to fit the training data too closely.

**Data Augmentation:** This technique involves generating additional training examples by applying random transformations to the existing training data. This can help to prevent overfitting by providing the model with a more diverse training set.

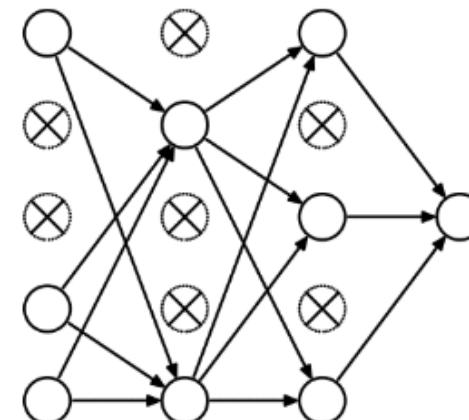


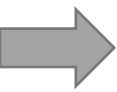
# Dropout Regularization

- Dropout is one of the most popular regularization techniques in deep learning.
- At each training iteration, dropout randomly chooses **different nodes** to ignore
- This prevents the network from **relying on any single neuron**.



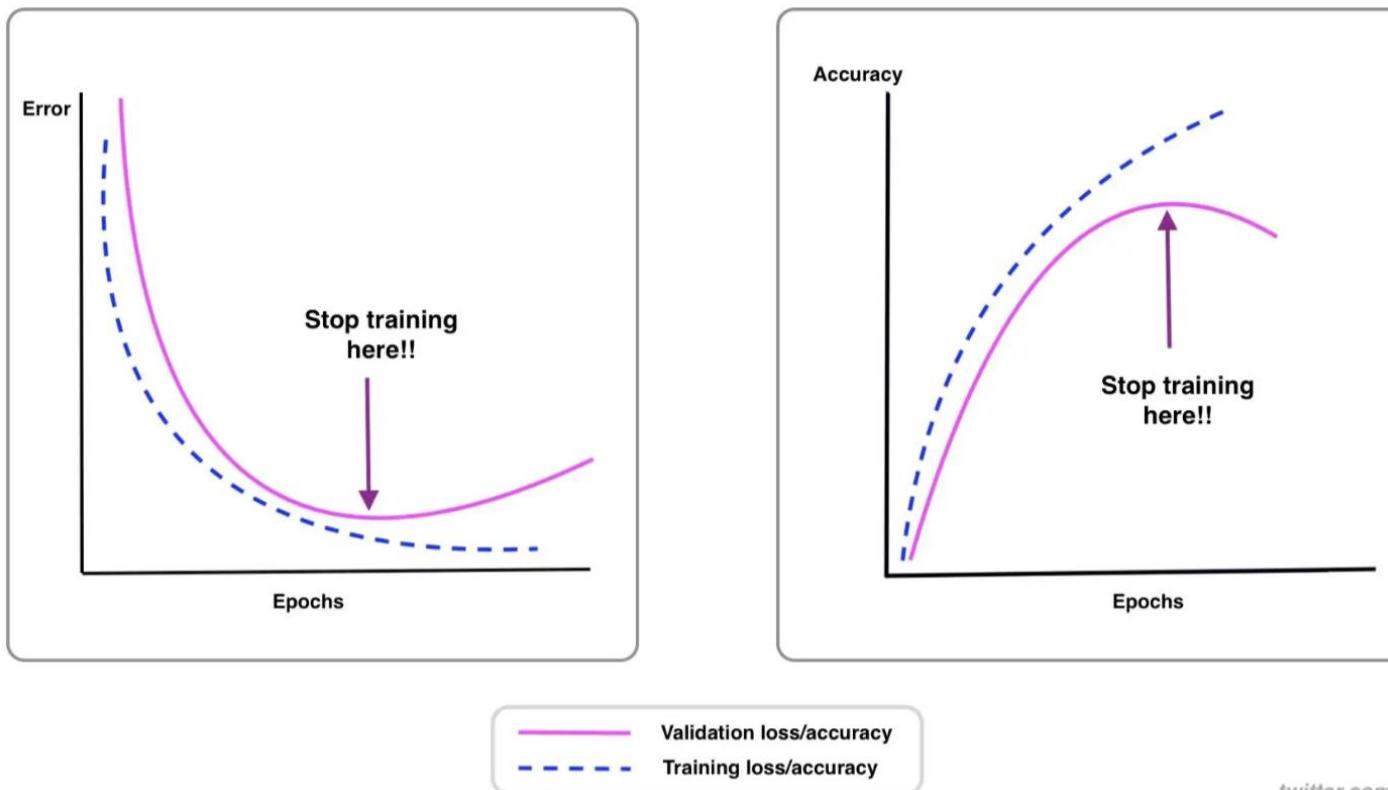
 `tf.keras.layers.Dropout(p=0.5)`



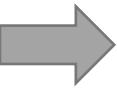


# Early Stopping

- Stop training when the performance on a validation set begins to degrade!

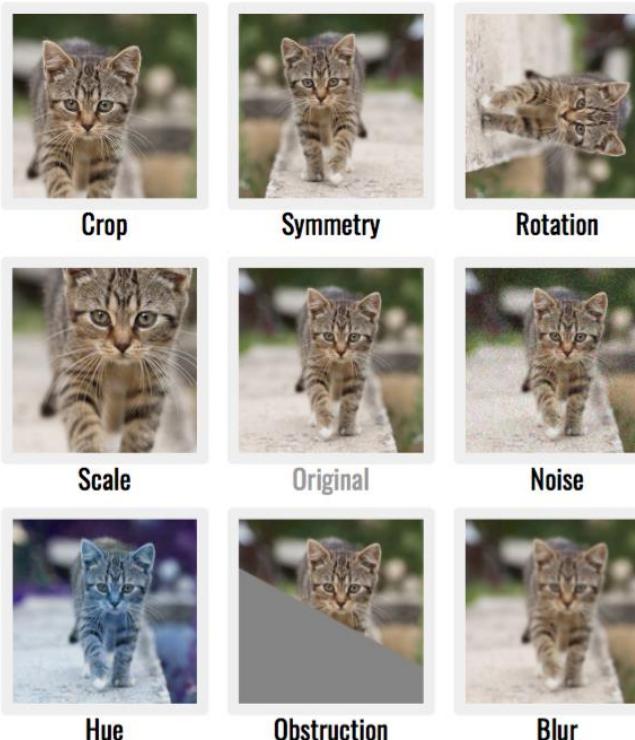


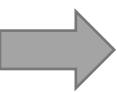
[twitter.com/jeande\\_d](https://twitter.com/jeande_d)



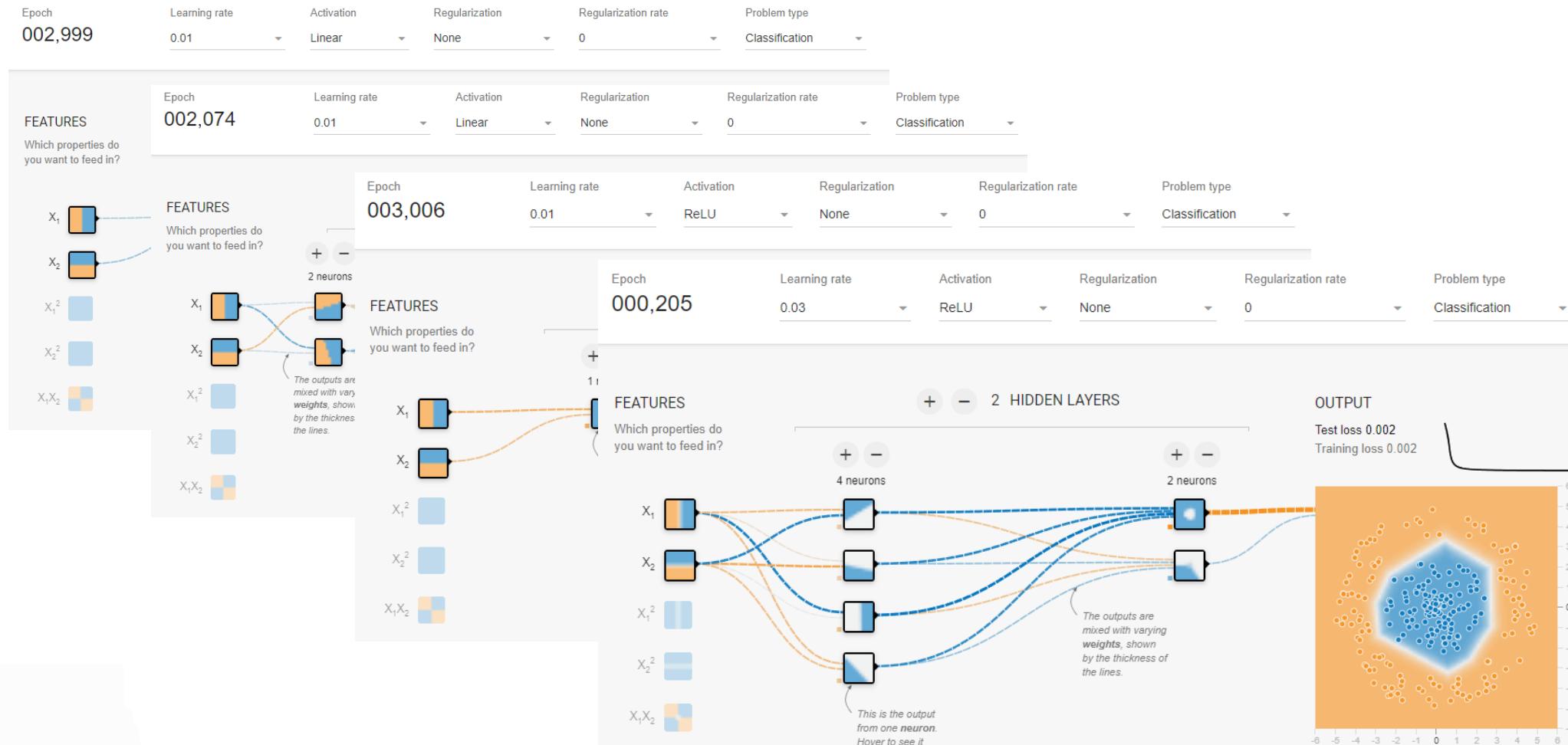
# Data Augmentation

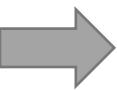
- Generating additional training examples by applying **random transformations** to the existing training data
- The goal of data augmentation is to improve the **generalizability** and robustness of machine learning models by providing them with **more diverse training data**.





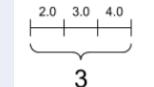
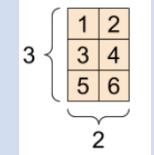
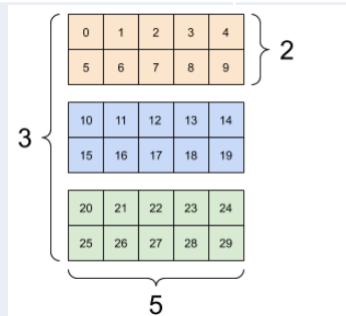
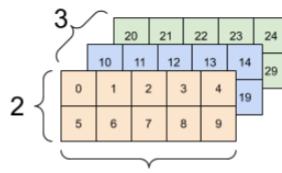
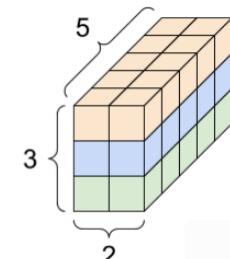
# Neural Network Playground (TensorFlow)

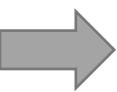




# What is a Tensor?

- Tensor: A **multi-dimensional array** (kind of like np.arrays)
- Rank:** Number of tensor axes , **Size:** The total number of items in the tensor

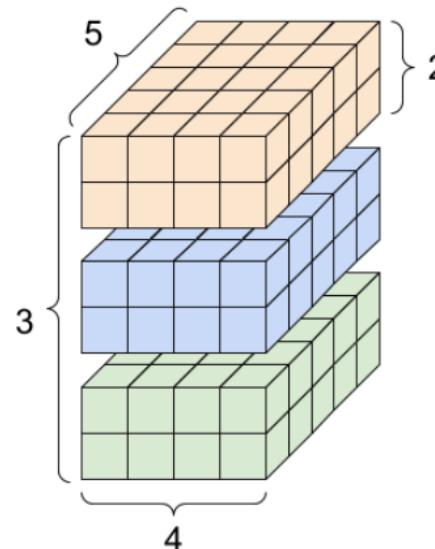
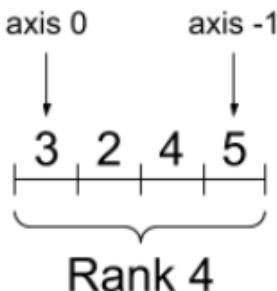
Tensor	tf code	shape	Example
Scalar (Rank-0 tensor)	tf.constant(4)	Shape = ()	4
Vector (Rank-1 tensor)	tf.constant([2.0, 3.0, 4.0])	Shape = (3 , )	
Matrix (Rank-2 tensor)	tf.constant([[1, 2],[3, 4],[5, 6]])	Shape = (3 , 2)	 
Rank-3 tensor	tf.constant([ [0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [[10, 11, 12, 13, 14], [15, 16, 17, 18, 19]], [[20, 21, 22, 23, 24], [25, 26, 27, 28, 29]]])	Shape = (3, 2, 5)	   



# Higher dimension Tensors

```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```

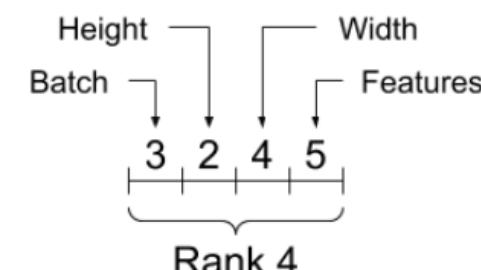
A rank-4 tensor, shape: [3, 2, 4, 5]

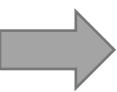


```
print("Type of every element:", rank_4_tensor.dtype)
print("Number of axes:", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy())
```

Type of every element: <dtype: 'float32'>  
Number of axes: 4  
Shape of tensor: (3, 2, 4, 5)  
Elements along axis 0 of tensor: 3  
Elements along the last axis of tensor: 5  
Total number of elements (3\*2\*4\*5): 120

- Often axes are ordered from **global to local**.
- The first axis is called the **batch axis** or batch dimension.

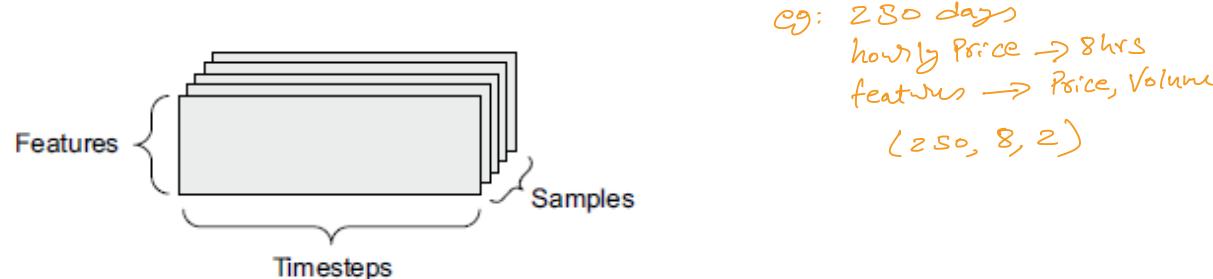


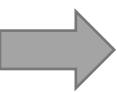


# Real-world examples of data tensors

- The data we'll manipulate almost always fall into one of the following categories:
- ✓ **Vector data:** Rank-2 tensors of shape **(samples, features)**, where each sample is a vector of numerical attributes (“features”)  
$$X = (1000, 50)$$

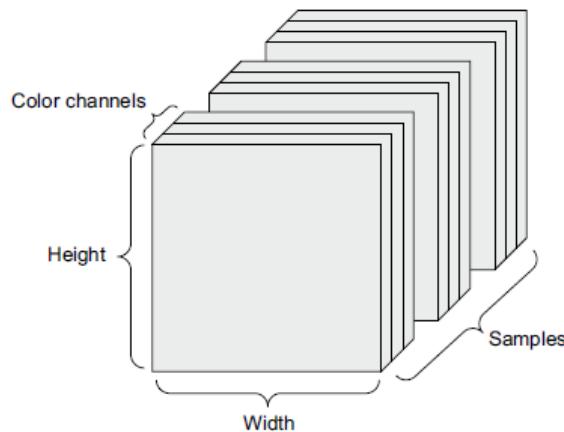

Diagram illustrating a rank-2 tensor  $X$  of shape  $(1000, 50)$ . It shows a vertical stack of 1000 feature vectors  $x_1, x_2, \dots, x_{50}$ . The index  $i$  is shown above the first few vectors, and the total count  $1000$  is indicated at the bottom.
- ✓ **Timeseries data or sequence data:** Rank-3 tensors of shape **(samples, timesteps, features)**, where each sample is a sequence (of length timesteps) of feature vectors





# Real-world examples of data tensors

- ✓ **Images:** Rank-4 tensors of shape **(samples, height, width, channels)**, where each sample is a 2D grid of pixels, and each pixel is represented by a vector of values (“channels”)



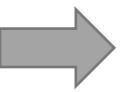
Shape = (2, 1024, 1024, 3)

e.g: 128 grayscale image  $200 \times 200$ .  
⇒ Shape = (128, 200, 200, 1)

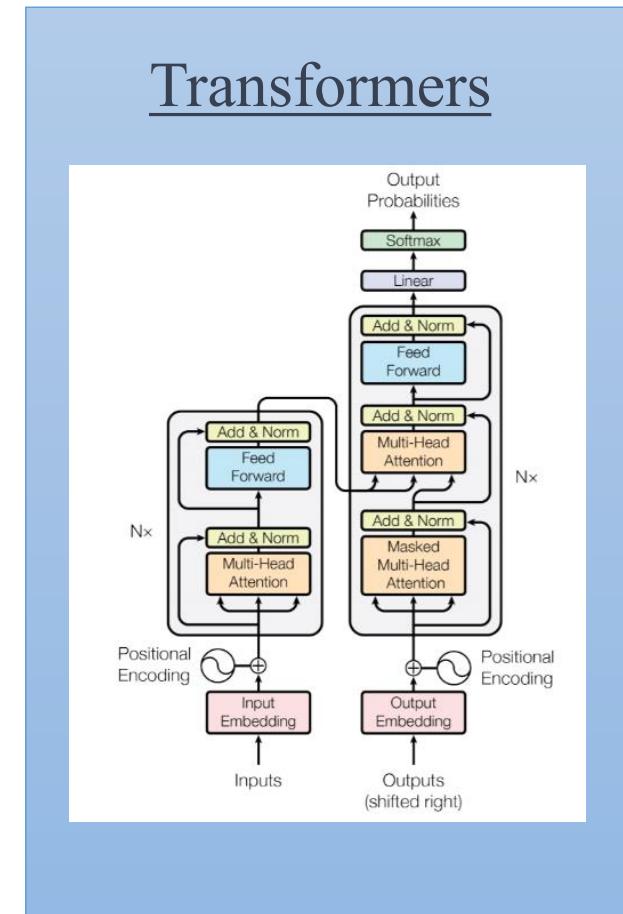
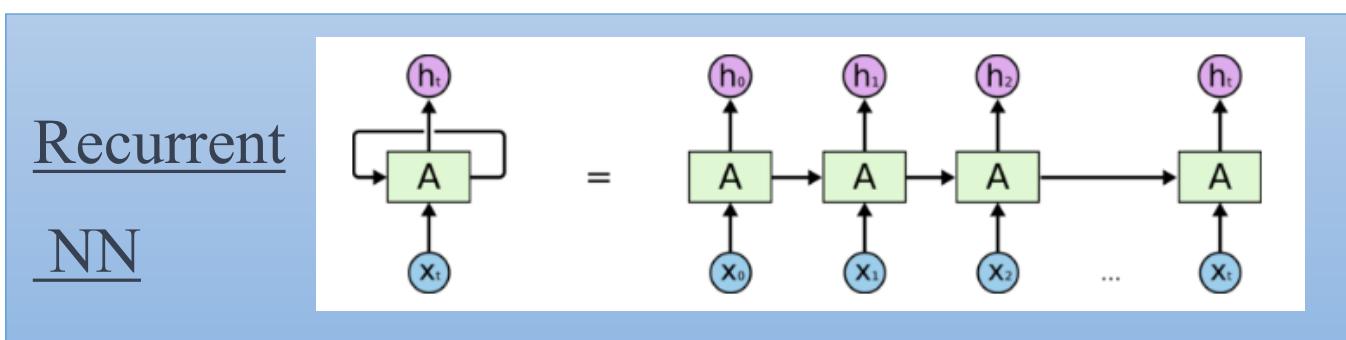
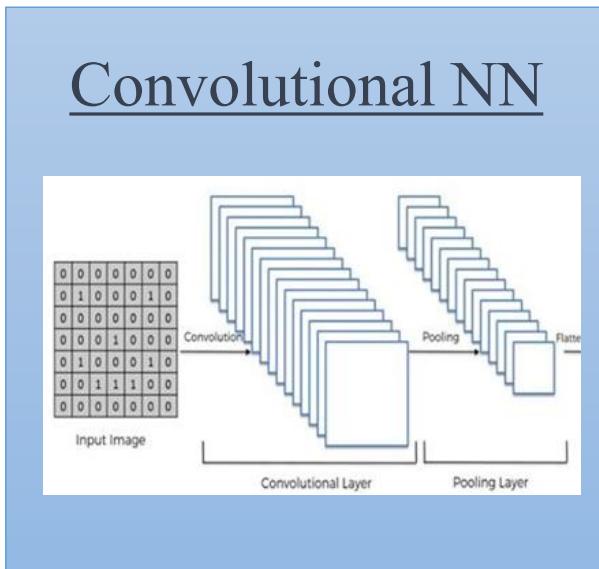
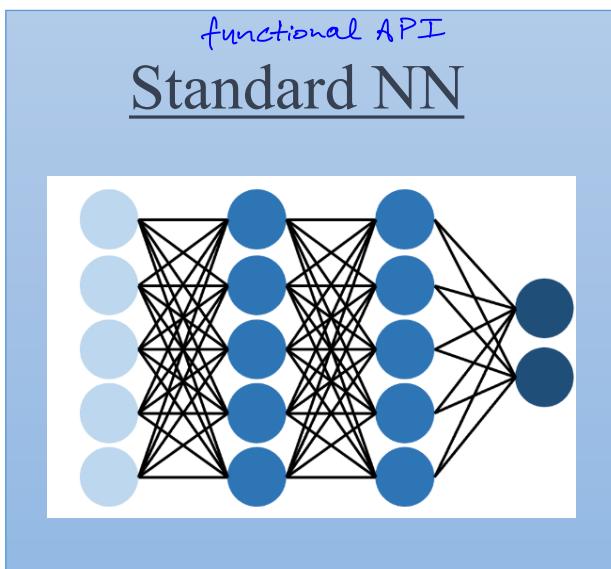
- ✓ **Video:** Rank-5 tensors of shape **(samples, frames, height, width, channels)**, where each sample is a sequence (of length frames) of images.

Example: What is the tensor shape for a 60-second, 144\*256 video clip sampled at 4 frames per second?

Shape = (1, 240, 144, 256, 3)



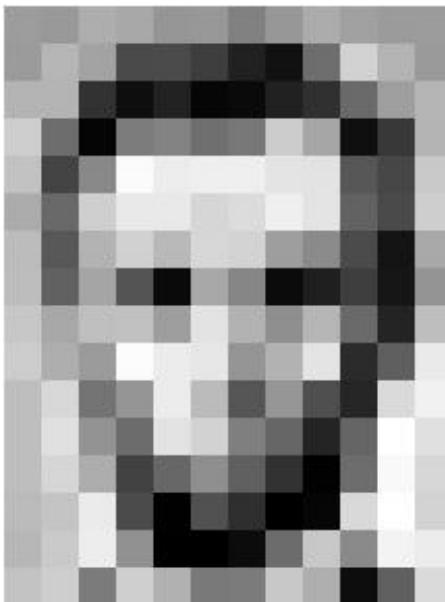
# Types of Neural Networks



# Module 5 – Part I

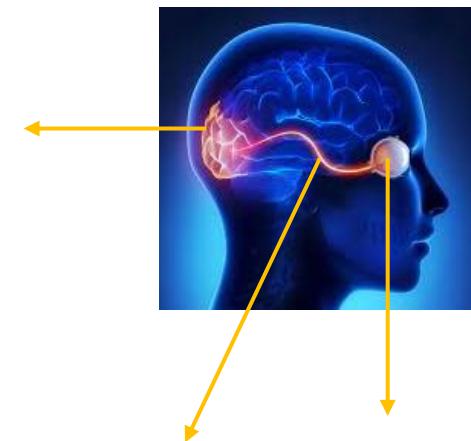
## Deep Computer Vision

### Basics



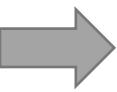
157	153	174	168	150	152	129	151	172	161	155	156
156	182	169	74	75	62	83	17	110	210	180	154
180	180	50	14	84	6	10	83	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	58	137	251	297	299	299	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	36	190
205	174	156	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	106	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	238	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Visual cortex



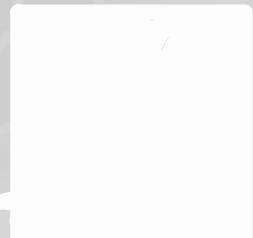
Optic nerve

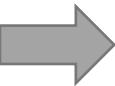
Retina



# Road map!

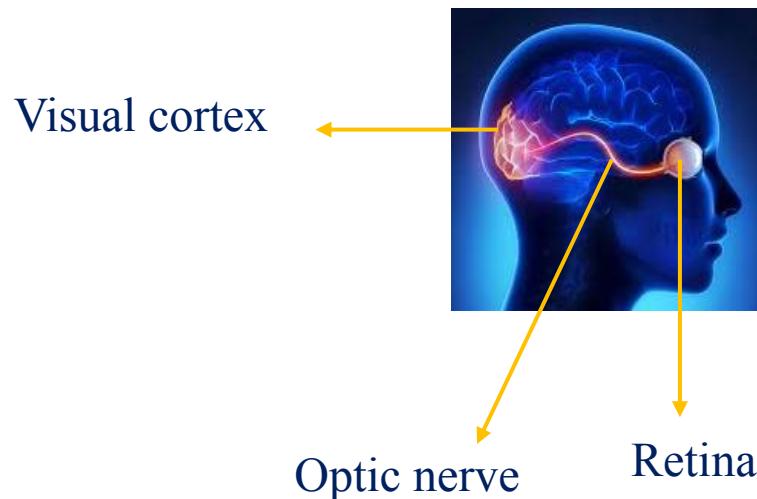
- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- **Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)**
- Module 6- Deep Sequence Modeling (RNN, LSTM)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

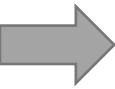




# Human vision: How do we see?

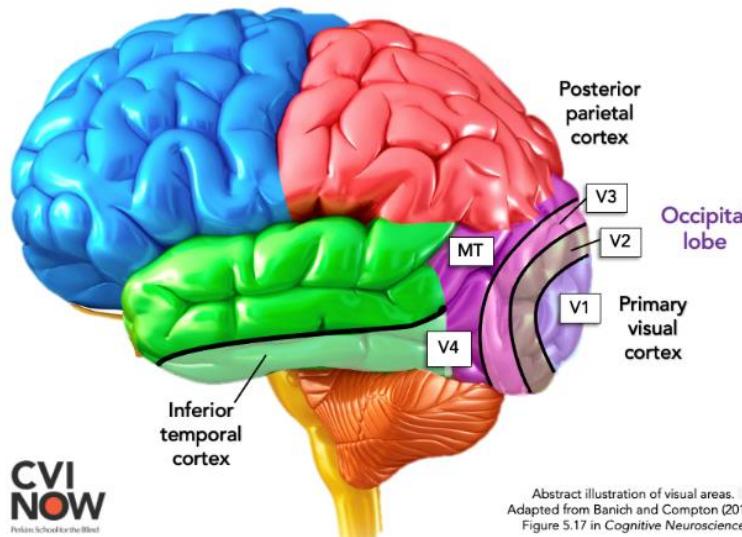
- The visual cortex is the part of the brain that processes visual information from the eyes. When we see an image, light from the image enters the eye and is focused onto the retina.
- The retina converts the light into electrical signals, which are then transmitted through the optic nerve to the brain.
- **Visual Cortex** is made up of **several different areas**, each with a specific role in processing visual information.

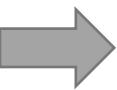




# Human vision: How do we see?

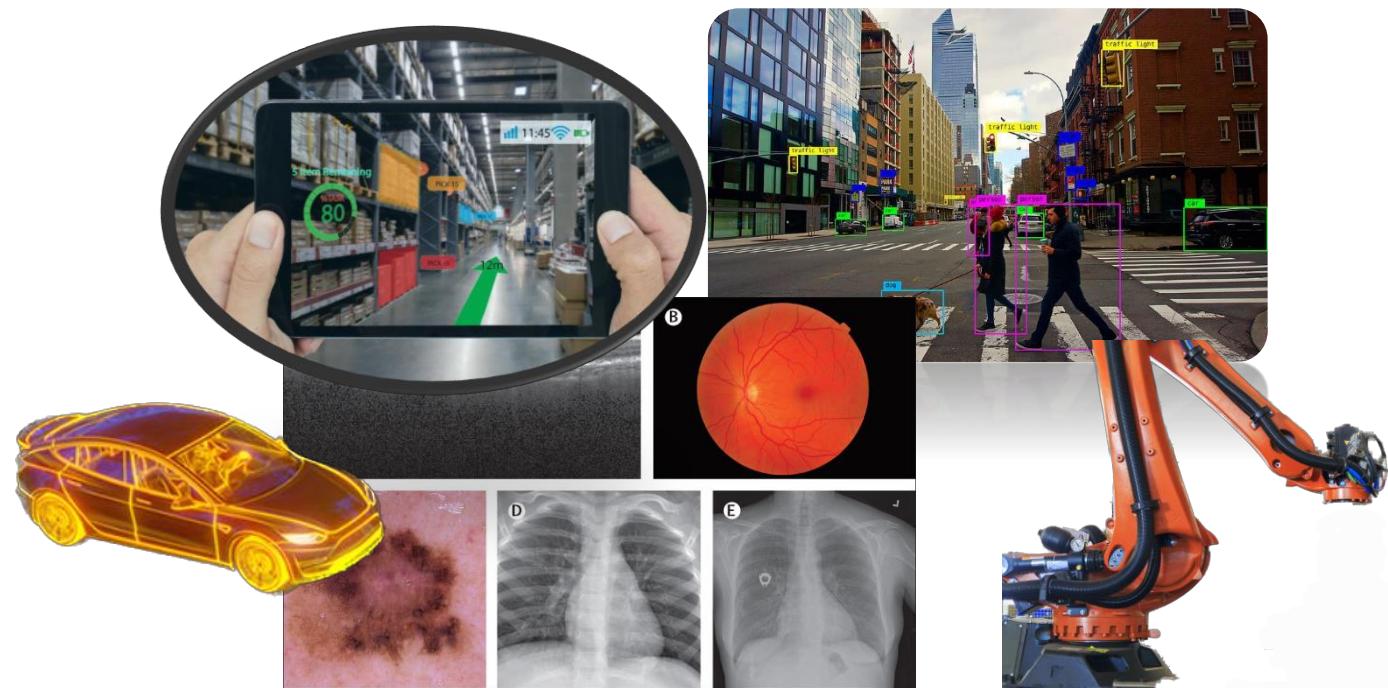
- V1 receives input from the retina and is responsible for early processing of visual information, such as edge detection and basic object recognition
- V2, receives input from V1 and is responsible for more advanced processing, such as color and texture processing.
- V3, receives input from V2 and is responsible for even more advanced processing, such as spatial relationships and depth perception.

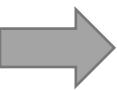




# Computer Vision: What is it?

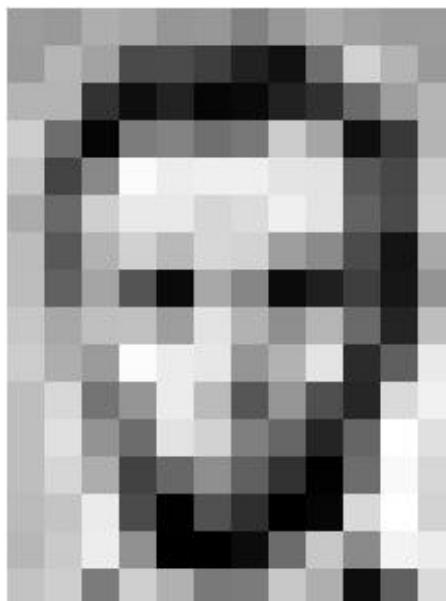
- Computer vision involves the development of **algorithms** and **systems**, to **enable** computers to **see, understand, and interpret** the visual world.
- **History:** The study of computer vision can be traced back to the **1950s**, but it was not until the development of **digital cameras** and the proliferation of cheap computing power in the **1990s** that the field truly took off
- Applications:
  - Robotics
  - Medical imaging
  - Surveillance
  - Augmented reality
  - Image and video analysis





# Computer Vision: What do they see?

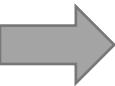
- Computer converts the image into a digital representation!
- This digital representation consists of a grid of pixels, each of which has a numerical value that represents the color and intensity of the light at that point in the image.
- By analyzing these pixels, computers can recognize, detect, and categorize images.



12 × 16

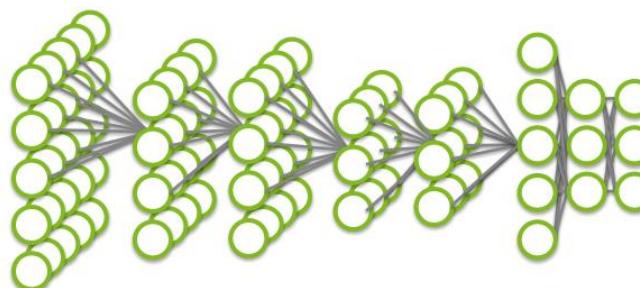
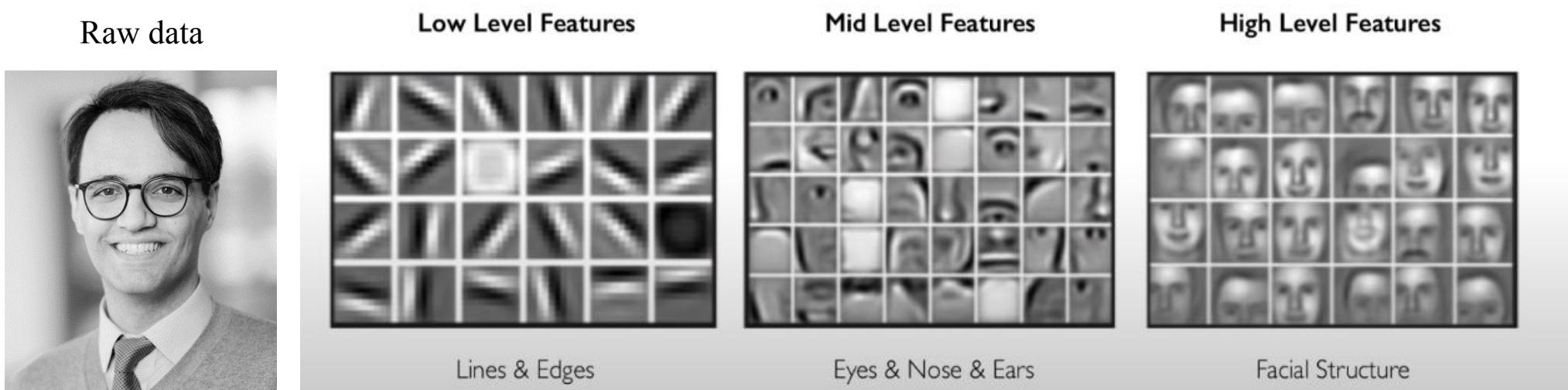
157	153	174	168	150	152	129	151	172	161	155	156
156	182	163	74	75	62	83	17	110	210	180	154
180	180	50	14	84	6	10	83	48	105	159	181
256	109	5	124	131	111	120	204	166	15	56	180
194	58	137	251	257	299	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	158	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

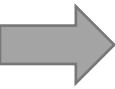
157	153	174	168	150	152	129	151	172	161	155	156
156	182	163	74	75	62	83	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218



# Computer Vision: How do they see?

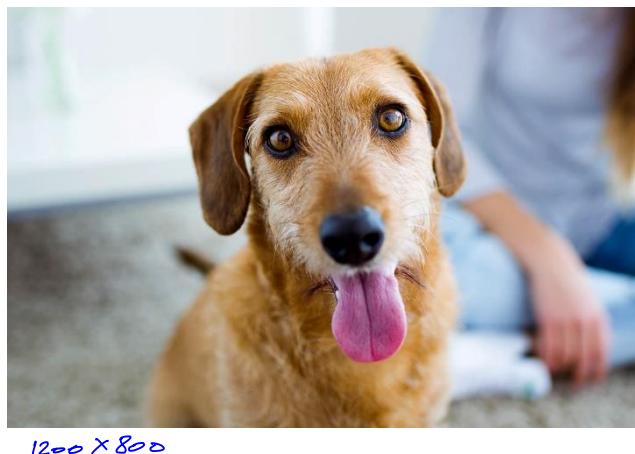
- Computers learn features in a **hierarchical manner**, like what happens in the **visual cortex**.
- **No hand-engineered features!**
- High-level feature detection.





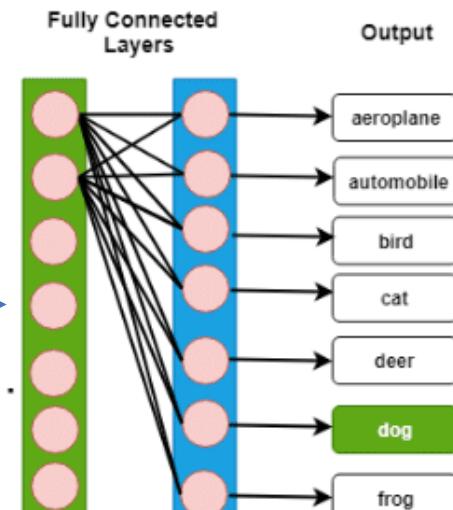
# Fully connected networks?

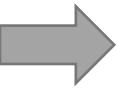
- Could we use fully connected DNN to process image data?
- Theoretically yes, practically no! Why?
  1. Lose **spatial information**
  2. Many **many parameters!!**



Shape = (1, 1200, 800, 3)

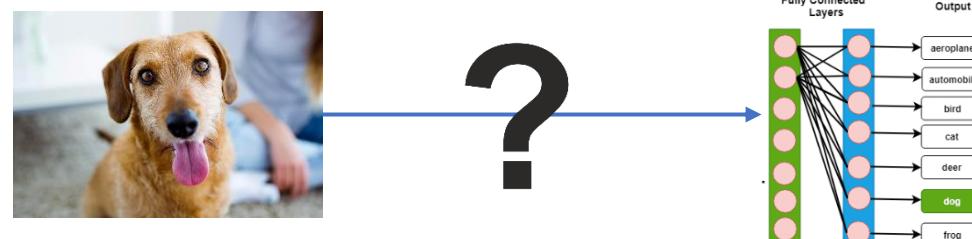
$$= 1200 \times 800 \times 3 = 2.88 \text{ million values}$$

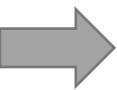




# Introduction to Convolutional Neural Networks (CNNs)

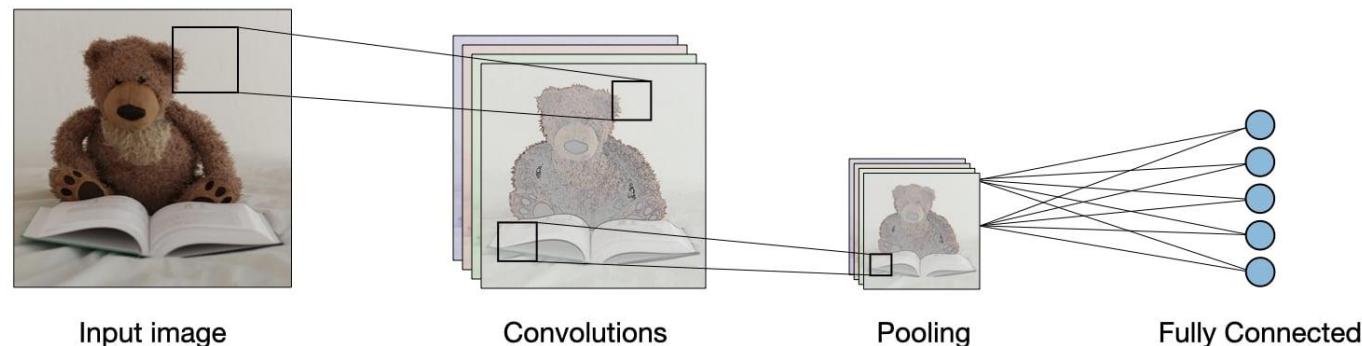
- A new NN **Architecture** is required which can:
  - preserve **spatial structure**
  - requires fewer parameters! Each neuron should see a **patch** of pixels as input instead of the entire vectorized image! **local feature extraction**.
- Convolutional neural networks (CNNs) are artificial neural networks designed specifically for image recognition and processing. They are composed of **multiple layers of nodes** that **each extract specific features from the input image**.
- CNNs were first introduced in the **1980s**, but only became widely used in the early **2010s**, when large amounts of labeled data and powerful hardware became available.

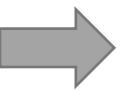




# CNN Architecture

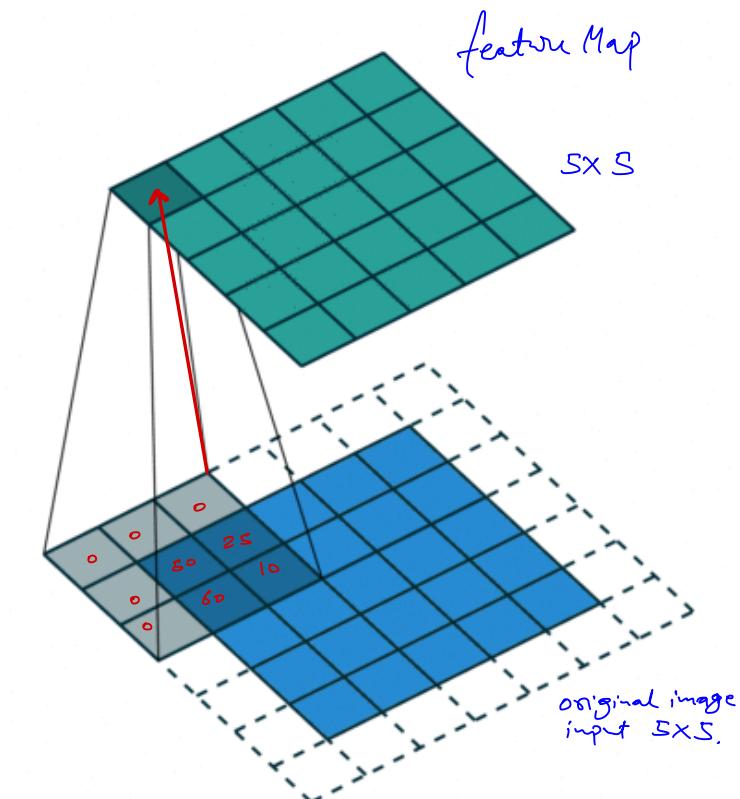
- A CNN typically consists of an **input layer**, one or more **convolutional layers**, one or more **pooling layers**, and one or more **fully connected layers**.
- The **convolutional layers** are responsible for **extracting features** from the input data using a set of learnable filters.
- The **pooling layers** are responsible for **down-sampling** the feature maps produced by the convolutional layers.
- The **fully connected** layers are responsible for **classifying** the features extracted by the CNN

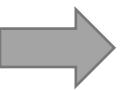




# What is a Convolution?

- In the context of image processing and computer vision, convolution is often used to **extract features from images**.  
*3x3 e.g:  $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$*
- Convolution involves taking a **small matrix of numbers**, called a **kernel** or **filter**, and sliding it over the input image, performing an element-wise multiplication.
- The result of this multiplication is then summed up and forms the output of the convolution at that position. This process is **repeated** for every position of the kernel over the input image.
- The output of this process is a **feature map**, which is a modified version of the input image that has been transformed by the kernel.

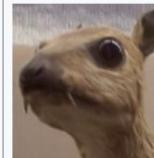


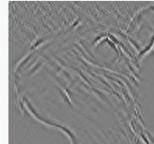
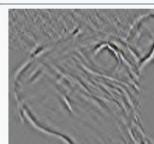
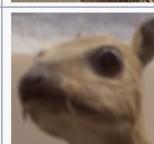


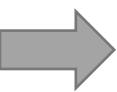
# Filter/Kernel details

- A filter or kernel is a small matrix of weights that is used in CNNs to extract features from images, such as **edges**, **corners**, or **patterns**.
- Different filters can be used to detect different types of features, and **multiple filters** can be applied **to the same input image** to extract a variety of features.

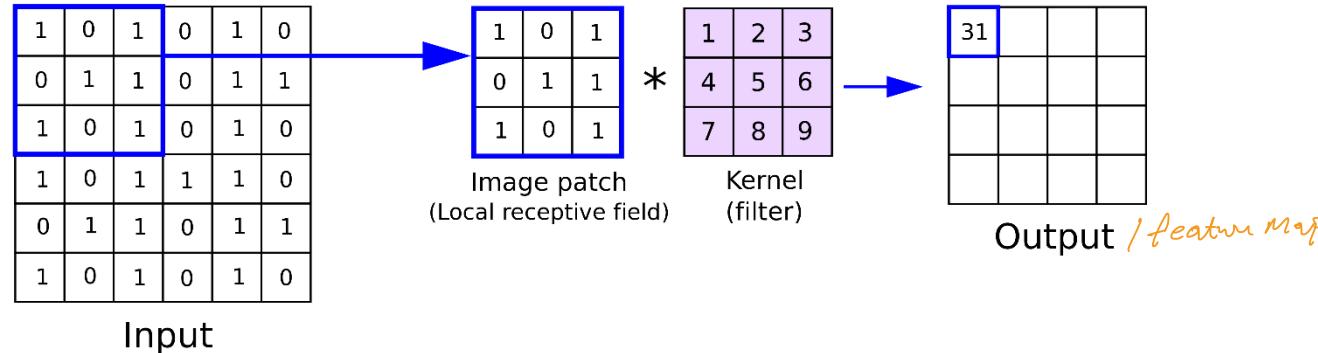
The numbers inside Kernel/Filters matrix are the parameters of model & the model will learn the weights of these parameters.

Operation	Kernel $\omega$	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	

Ridge or edge detection	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & -1 \end{bmatrix}$		<b>Sharpen</b>	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$			<b>Box blur</b> (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ 



# Convolution numerical example



A detailed numerical example of a convolution step:

**Input:**

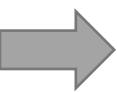
7	2	3	3	8
4	5	3	8	4
3	3	2	8	4
2	8	7	2	7
5	4	4	5	4

**Kernel:**

1	0	-1
1	0	-1
1	0	-1

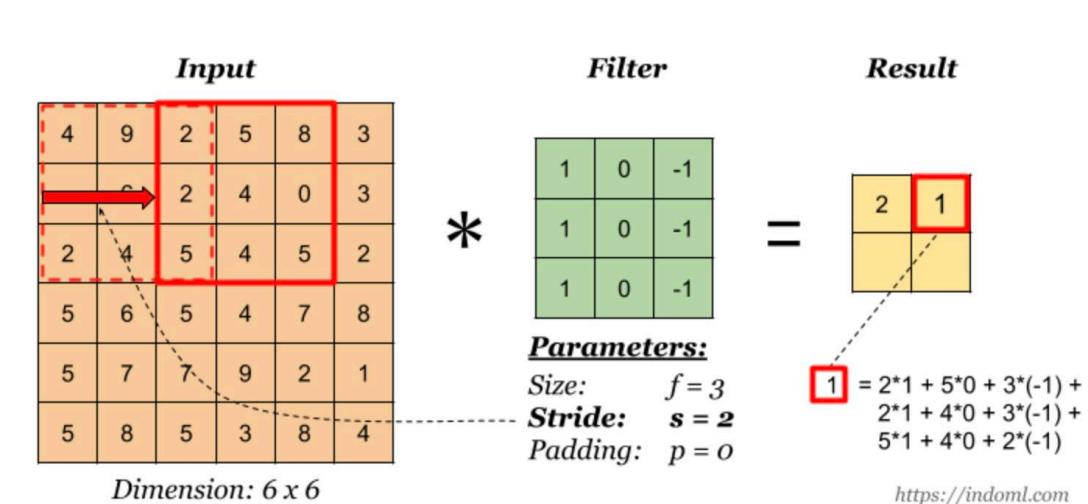
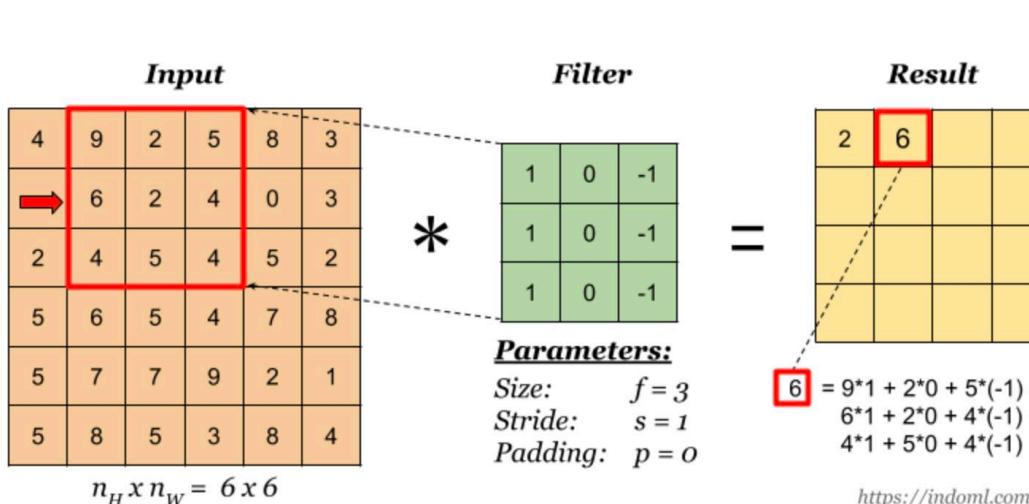
**Calculation:**

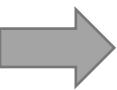
$$7 \times 1 + 4 \times 1 + 3 \times 1 + 2 \times 0 + 5 \times 0 + 3 \times 0 + 3 \times -1 + 3 \times -1 + 2 \times -1 = 6$$



# Filter Size, Stride, Padding

- The **size**, **stride**, and **padding** of a filter are three parameters that control the way in which the filter is applied to the input image during the convolution operation.
- Size:** Refers to the dimensions of the kernel matrix. ex,  $3 \times 3$
- Stride:** Refers to the number of pixels that the kernel is moved at each step as it is slid over the input image. A larger stride results in a smaller output image.



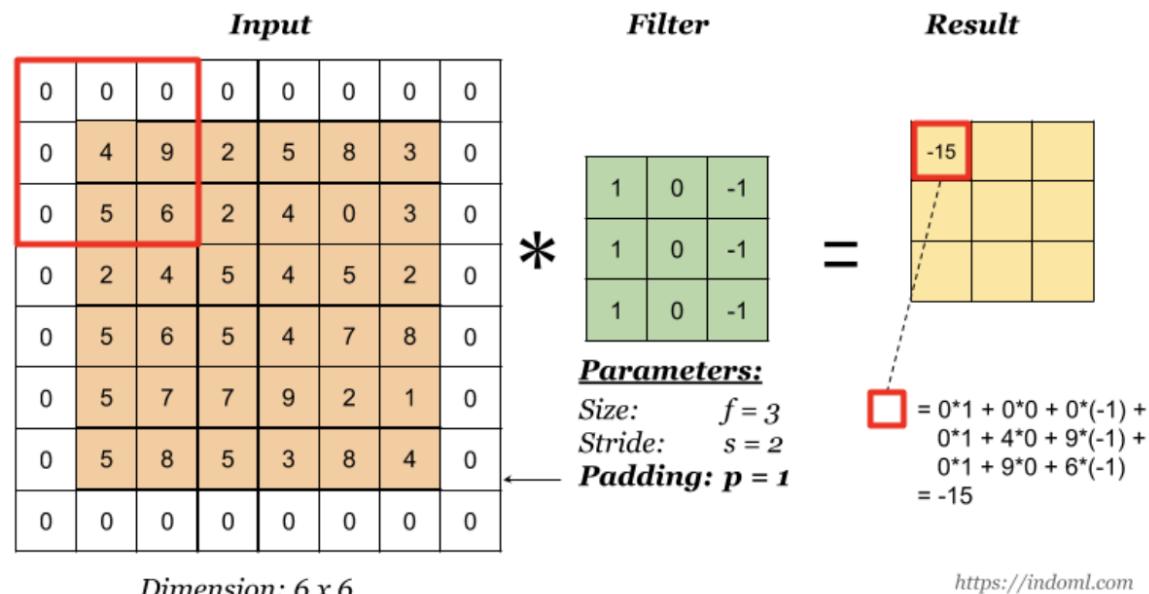


# Filter Size, Stride, Padding

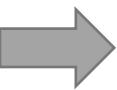
- **Padding:** refers to the addition of extra pixels around the border of the input image.
- Padding is often used in CNNs to **preserve the spatial dimensions** of the input image and to control the size of the output image.
- This is important for building deeper networks, since otherwise the height/width would **shrink** as we go to deeper layers.
- Padding helps us **keep more of the information at the border** of an image.

Padding terminology:

- **Valid** padding: no padding
- **Same** padding: Preserving the input dimension



<https://indoml.com>



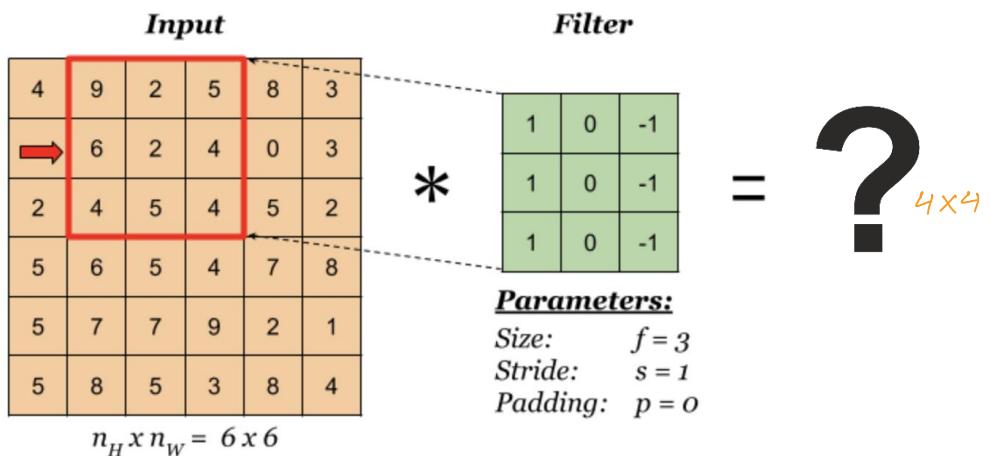
# Calculating the output dimension

$$n^{[l]} = \text{floor} \left( \frac{n^{[l-1]} + 2p^{[l-1]} - f^{[l]}}{s^{[l]}} + 1 \right)$$

Annotations for the formula:

- $n^{[l]}$ : layer dimension
- $n^{[l-1]}$ : dim. of previous layer
- $p^{[l-1]}$ : padding in previous layer
- $f^{[l]}$ : filter size
- $s^{[l]}$ : stride

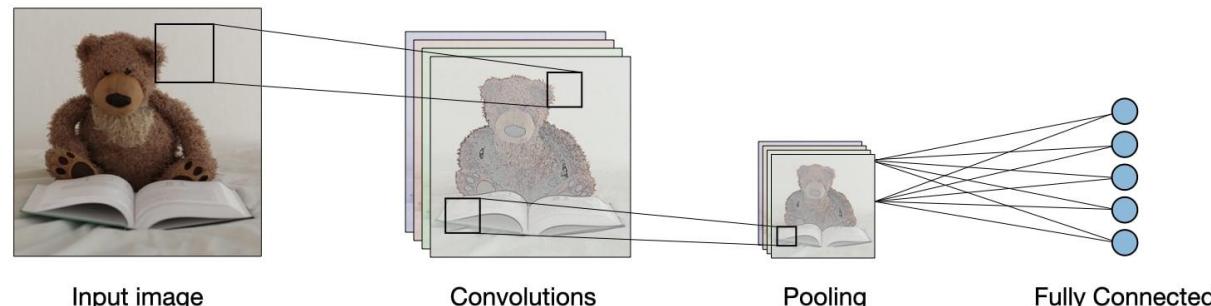
The output dimension, is the floor of the above formula!



$$\begin{aligned} h &= \left\lfloor \frac{6 + 2(0) - 3}{1} + 1 \right\rfloor \\ &= 4 \\ \text{Hence, } &4 \times 4. \end{aligned}$$

# → How CNN operates?

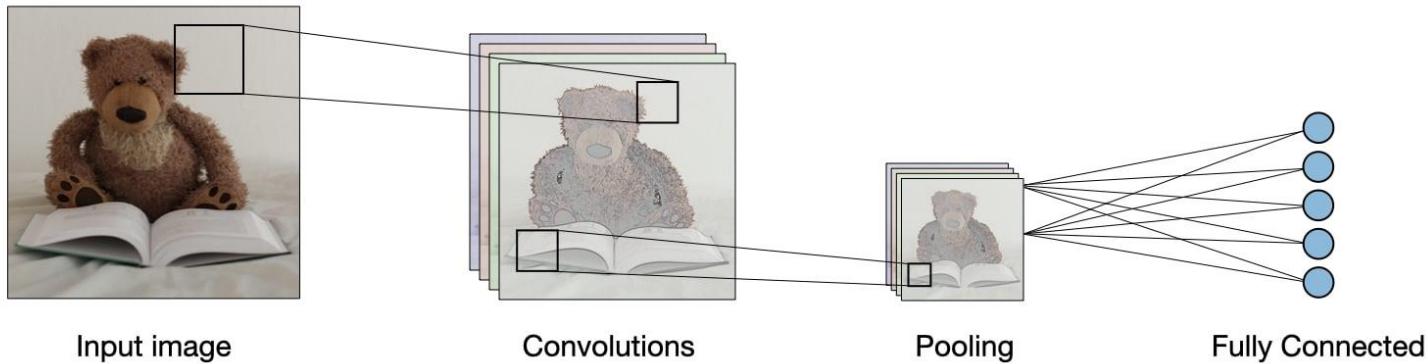
- During training, the CNN is presented with a set of **labeled training examples**. The CNN processes the input image through a **series of layers**, in which the filters are used to extract features from the input image.
- The filters are **updated during the training process** in order to learn features that are relevant for a particular task.
- The output of the CNN is then compared to the true label for the input image, and the **error is calculated**.
- This error is then **backpropagated** through the network, and the **weights of the filters are adjusted** in order to reduce the error. This process is repeated for each training example, and the filters are updated accordingly.

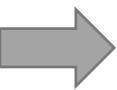


# Module 5 – Part II

## Deep Computer Vision

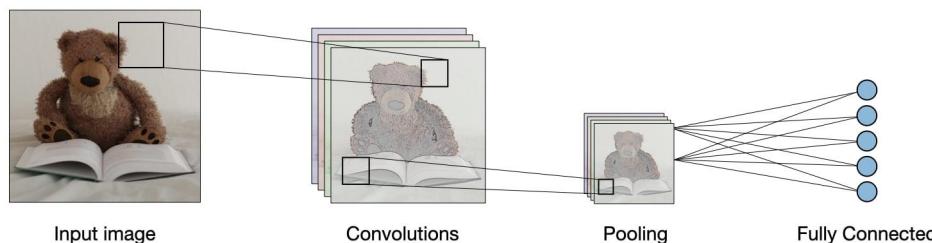
### Convolutional Neural Network (CNN)

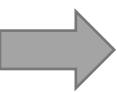




# What is a Convolutional layer?

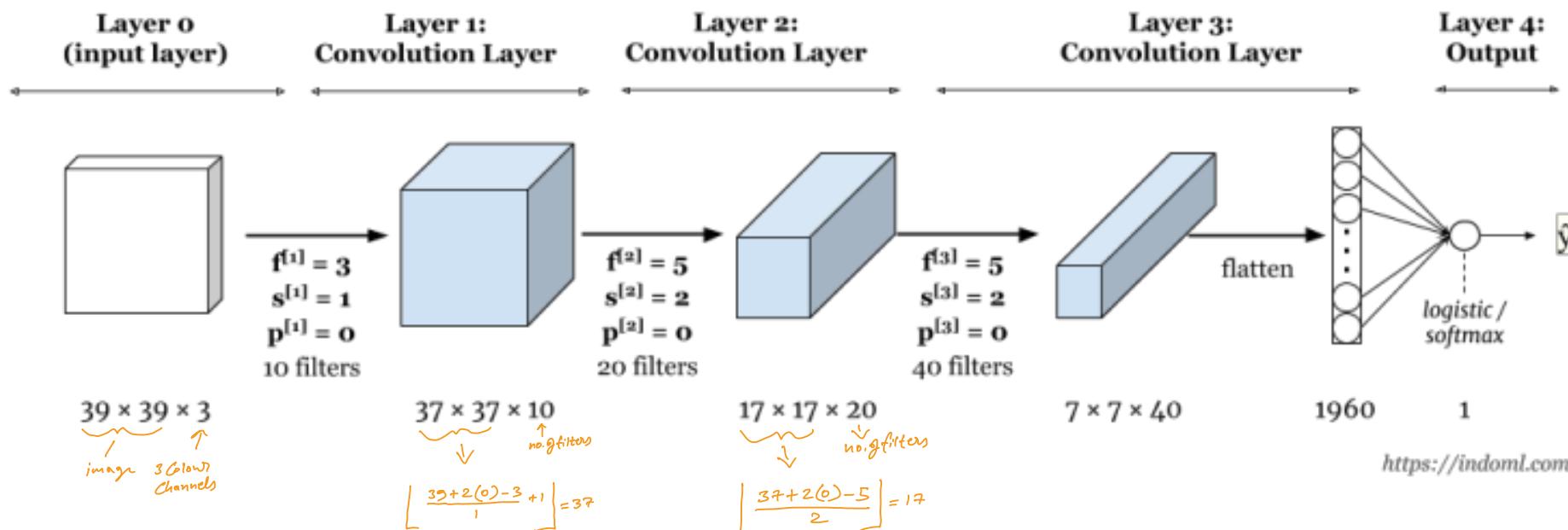
- A convolutional layer consists of a **set of filters** that are convolved with the **input** data to produce a set of **feature maps**.
- Each filter is responsible for extracting a specific feature from the input data, such as **edges, corners, or textures**.
- The **filters are learned during the training process**, allowing the CNN to automatically learn relevant features for a given task
- Why Convolutions?
  1. **Parameter sharing**: a filter (such as edge detector) that's useful in one part of the image is probably useful in another part of the image.
  2. **Sparsity of connections**: in each layer, each output value depends only on small number of inputs.

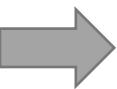




# A more complete example!

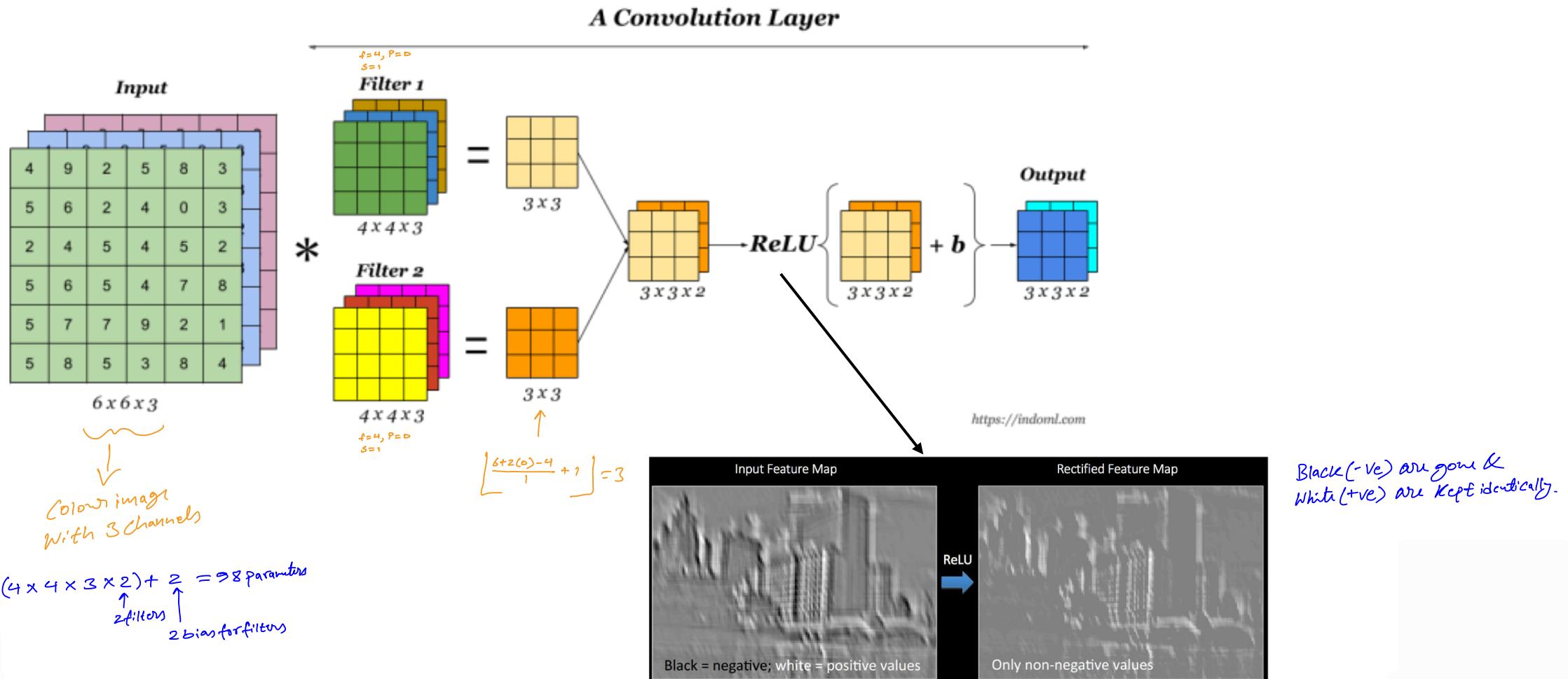
- Stacking multiple convolution layers!

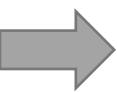




# Parameters in a convolution layer!

- At the last step, a bias is added, and an activation function is applied.





# Pooling Layer

- Pooling layer is used to **reduce the size** of the feature map (generated by the convolutional layer) and to **speed up calculations**, as well as to make some of the features it detects a bit more **robust**. (*i.e. get features that are less sensitive to small changes in input image.*)
- Pooling reduce the size and makes the model scalable while still **preserving spatial structure**. (*keeping it as matrix & not flattening things vertically*)

**Max Pooling**

4	9	2	5
5	6	2	4
2	4	5	4
5	6	8	4

→

9	5
6	8

*Take max. value of each 2x2 Matrix.*

**Avg Pooling**

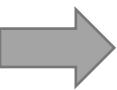
4	9	2	5
5	6	2	4
2	4	5	4
5	6	8	4

→

6.0	3.3
4.3	5.3

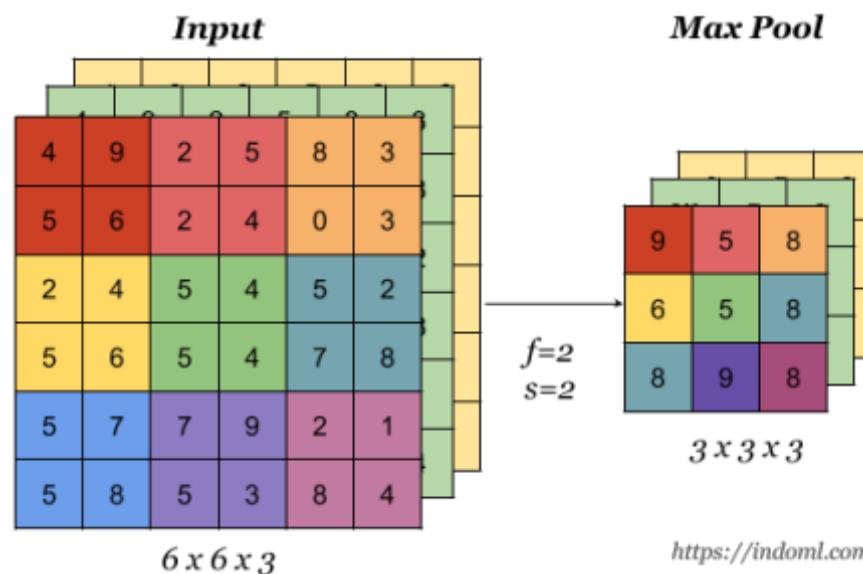
*Take Avg. of each 2x2.*

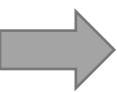
<https://indoml.com>



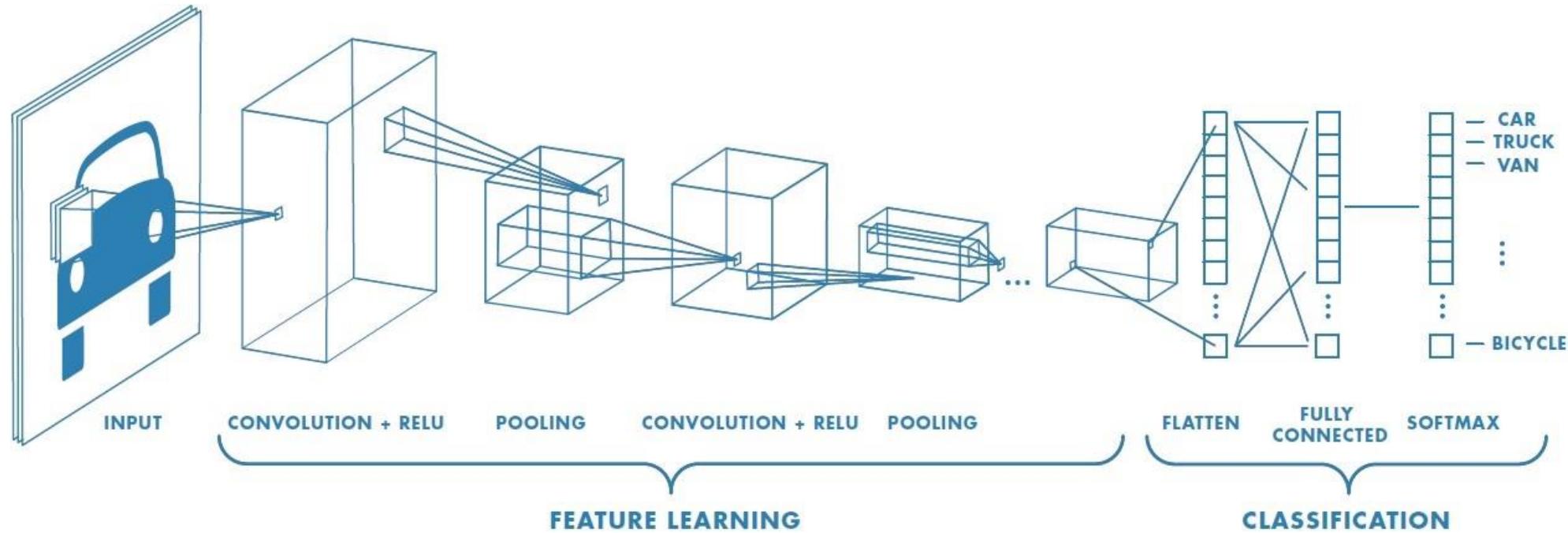
# Pooling Layer Properties

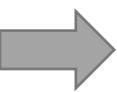
- The hyper parameters of pooling: **size**, **stride**, **type**
- There is **no parameters to be learned**.
- Unlike Convolutional operation, **pooling does not change the number of channels**.



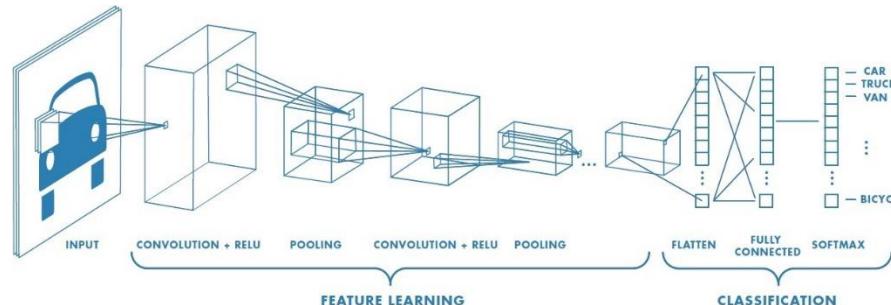


# Putting it together!





# Putting it together!

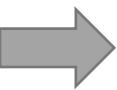


```
def CNN_builder():
    inputs= keras.Input(shape=(28,28,1), name='Input layer')
    x = layers.Conv2D(filters= 32, kernel_size = 3, strides = (1,1) , padding='valid', activation='relu' ,name="conv_layer_1")(inputs)
    x = layers.MaxPool2D(pool_size=2, name="pooling_1")(x)
    x = layers.Conv2D(filters= 64, kernel_size = 3, activation='relu', name="conv_layer_2")(x)
    x = layers.MaxPool2D(pool_size=2, name="pooling_2")(x)
    x = layers.Conv2D(filters= 128, kernel_size = 3, activation='relu', name="conv_layer_3")(x)
    x = layers.Flatten(name="flattening_layer")(x)
    x = layers.Dense(units= 64, activation='relu')(x)
    outputs = layers.Dense(units= 10, activation='softmax', name='output_layer')(x)

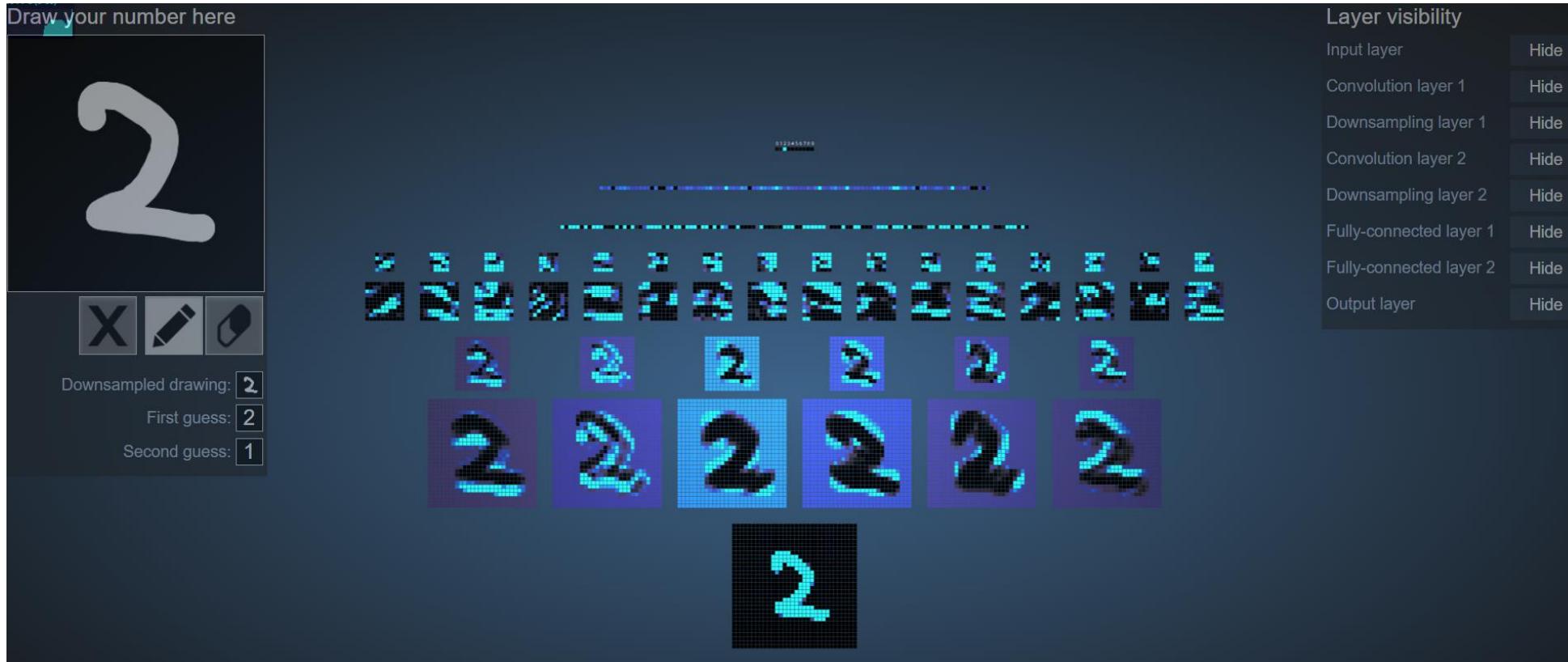
    model = keras.Model(inputs= inputs , outputs=outputs, name='my_first_CNN_model')

    model.compile(optimizer='rmsprop',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

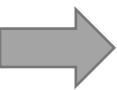
    return model
```



# Interactive Node-Link Visualization of Convolutional Neural Networks

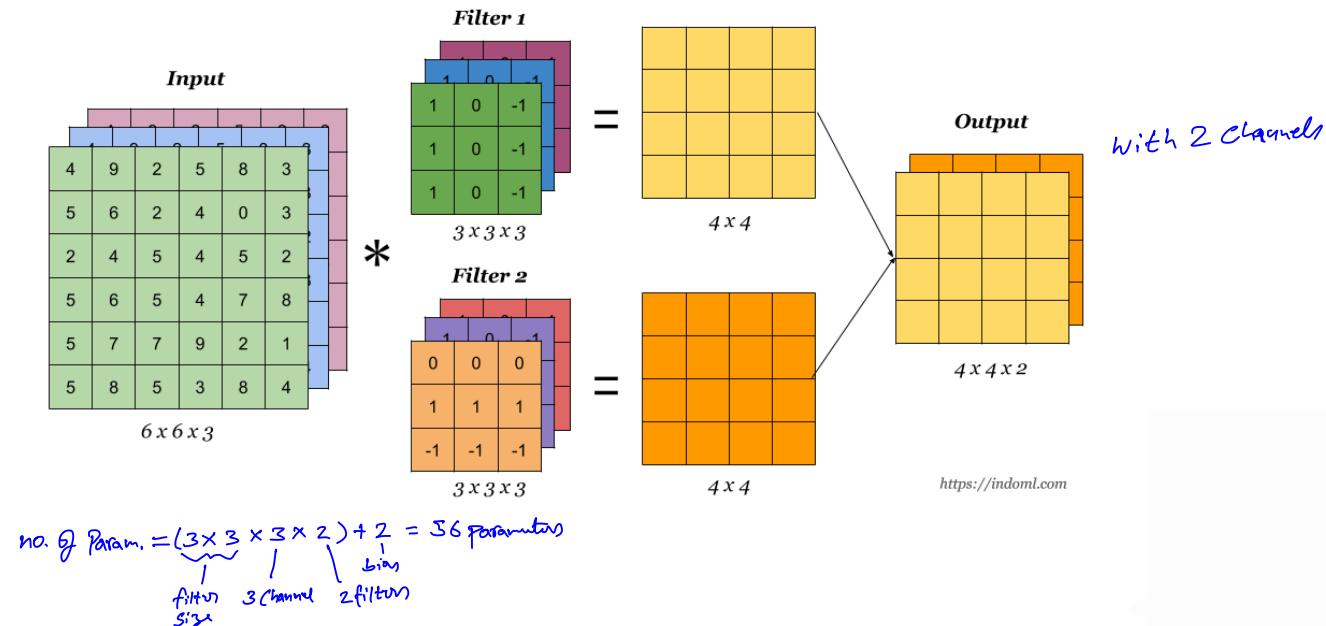
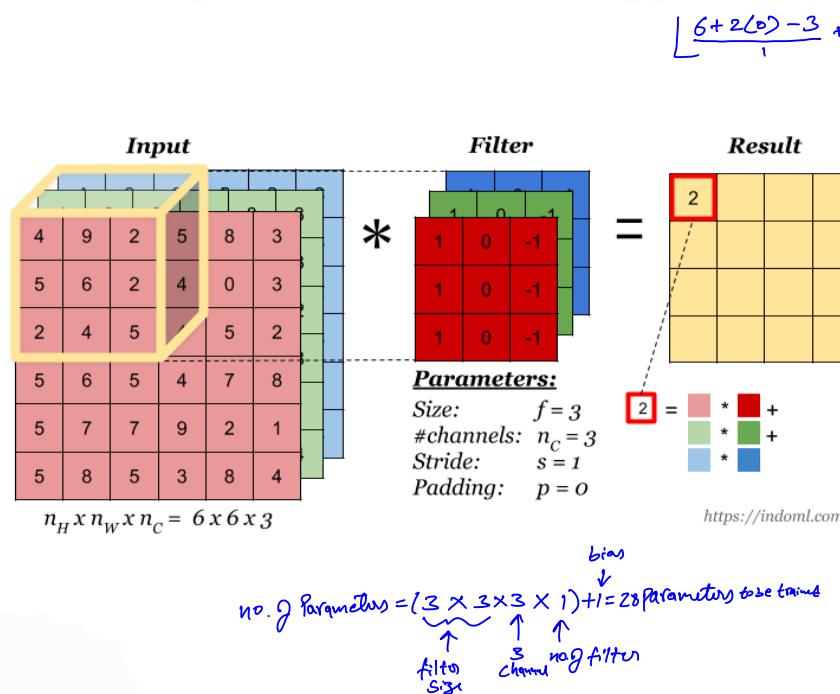


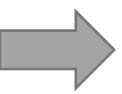
# Other Convolution Operations



# Convolution operation on Volume

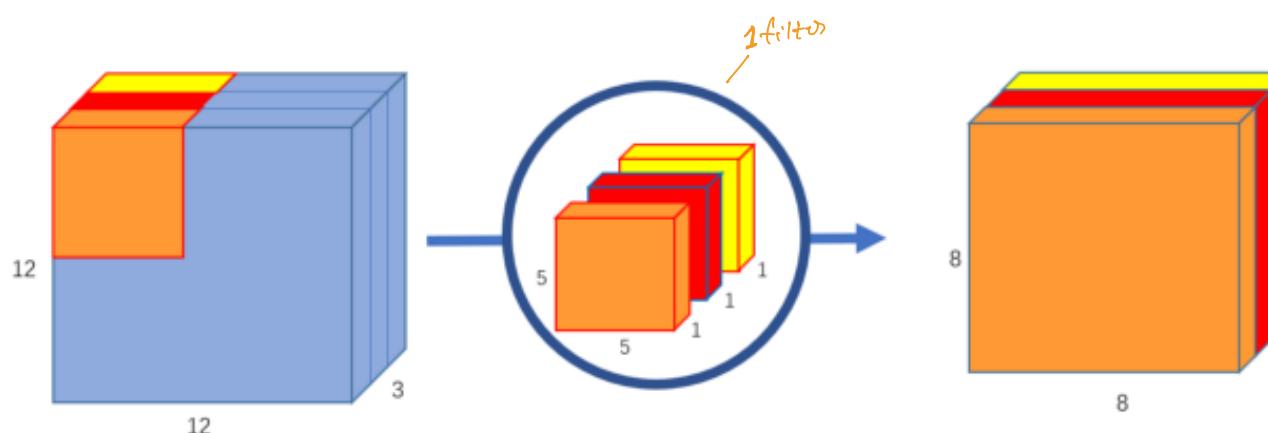
- A filter should have a **matching number of channels** if the input has more than one channel (e.g., an RGB image).
- In order to calculate one output cell, **apply convolution to each matching channel and then add the results together**.
- Multiple filters could be applied to the input image!





# Depthwise Convolution

- Depthwise convolution is a type of convolution operation that operates on each channel of an input volume independently. In other words, the filter has the same depth as the input volume, and the convolution is applied separately to each channel.
- The output of a Depthwise convolution is a **volume with the same number of channels** as the input volume
- The Depthwise separable convolution is so named because it deals not just with **the spatial dimensions**, but with the **depth dimension**

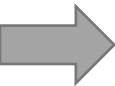


**Parameters:**

Size:  $f = 5$

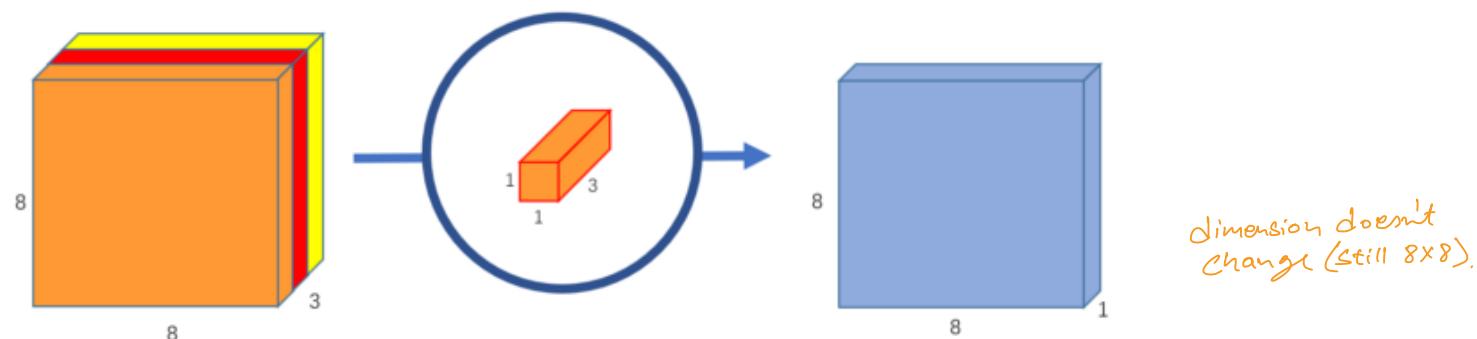
Stride  $s = 1$

Padding  $p = 0$



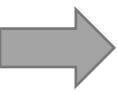
# Pointwise Convolution: 1 x 1 Convolution

- A **pointwise** convolution, is a convolution operation in which the filter has a size of **1x1xC**, where C is the number of **channels** in the input volume
- A **1x1** convolution is used to **reduce the number of channels** in the input volume, or to **combine** multiple channels in the input volume into a single channel in the output volume.
- This convolution can be thought of as a **linear transformation** of the input channels. This makes **1x1** convolutions useful for learning **channel-wise relationships** in the input data.



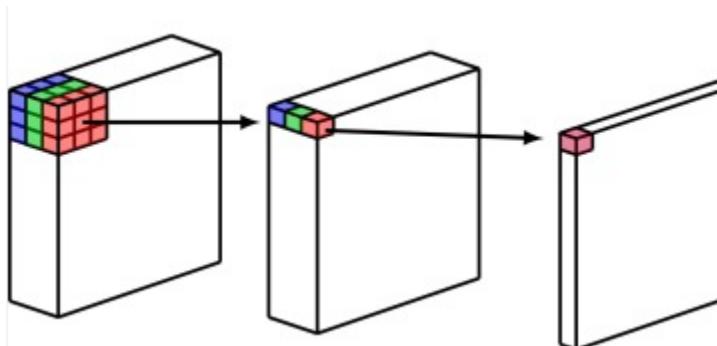
Parameters:

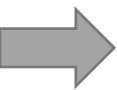
Size:  $f = 1$   
Stride  $s = 1$   
Padding  $p = 0$



# Depthwise Separable Convolution

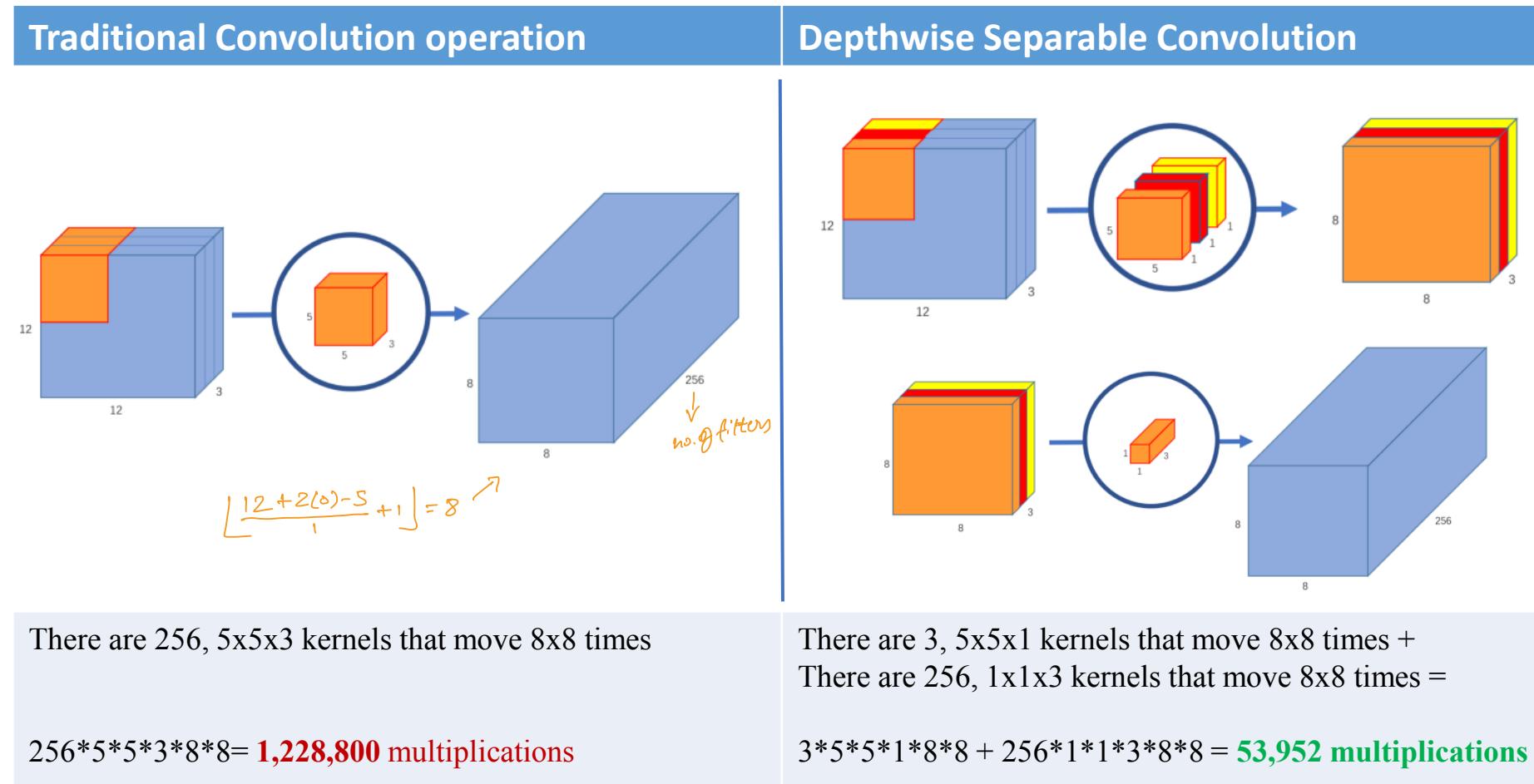
- Depthwise convolution is often used in conjunction with pointwise convolution (a 1x1 convolution) to build a **Depthwise separable** convolutional neural network.
- This type of network architecture can be **more efficient and faster** to train than a traditional convolutional neural network, because it has fewer parameters and requires fewer computations.
- It is often used in **mobile-based applications**, where computational resources are **limited**, to build lightweight and efficient models.
- It is also used in models that need to learn **spatial features** from different channels separately, such as models for image segmentation or object detection.





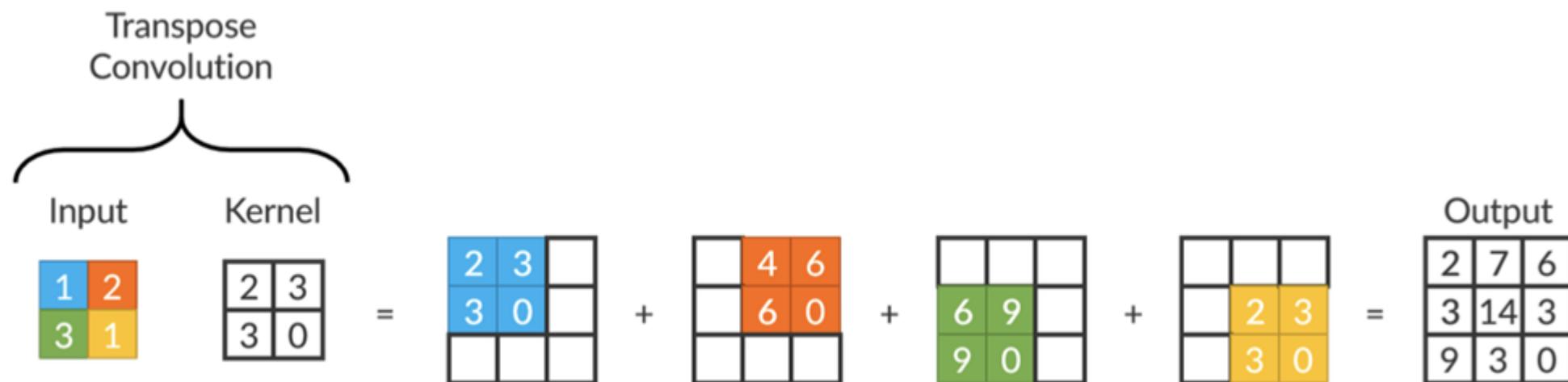
# Why Depthwise Separable Convolution? A numerical example

- Imagine we want to use 256 kernels to create 256 of 8x8x1 output images from a 12x12x3 input.



# Transposed Convolution

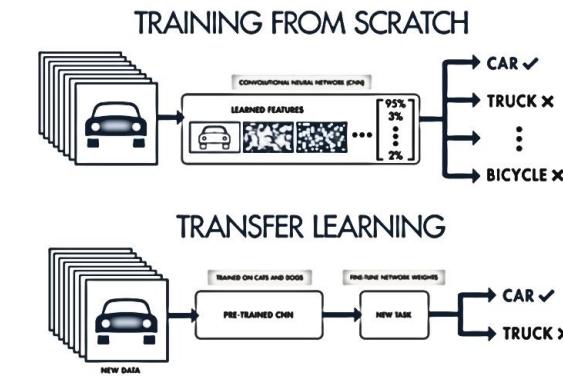
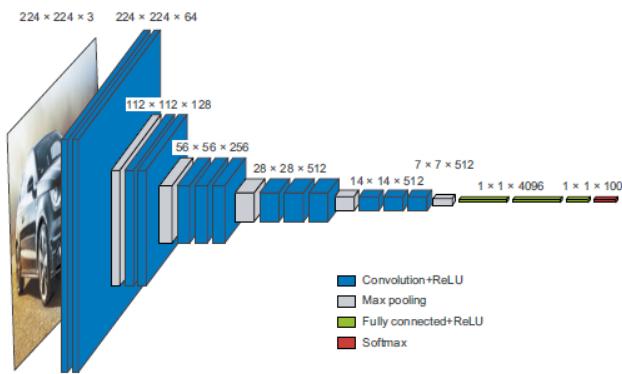
- A transpose convolution, also known as a deconvolution, is a type of convolutional layer that is used to **upsample** the feature map produced by a convolutional neural network.
- It takes an input feature map and produces an output feature map **that is larger in size**, by **inserting zeros** between the elements of the input feature map and convolving with a set of filters



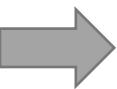
# Module 5 – Part III

## Deep Computer Vision

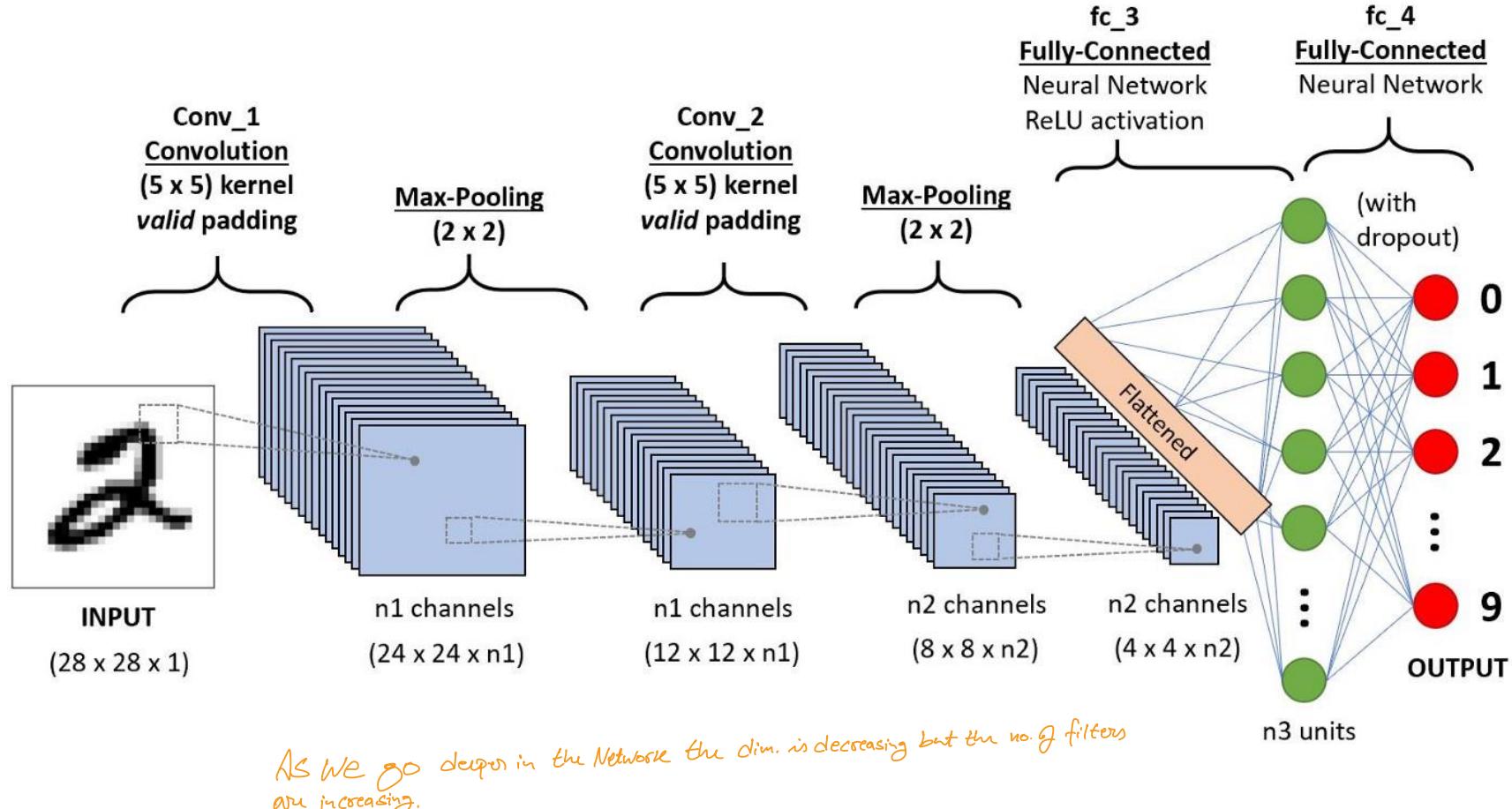
### Architecture, Interpretable CNN and Transfer Learning

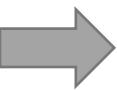


# Convnet Architecture Best Practices



# CNN Architecture

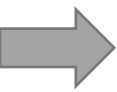




# The MHR formula

- Universal recipe to make a complex system simpler:
  - Structure your complex model into modules (**Modularity**)
  - Organize modules into a **Hierarchy**
  - Start reusing the same modules in multiple places as appropriate (**Reuse**)
- This is the pattern we observe in the **most successful** Deep Learning architectures.

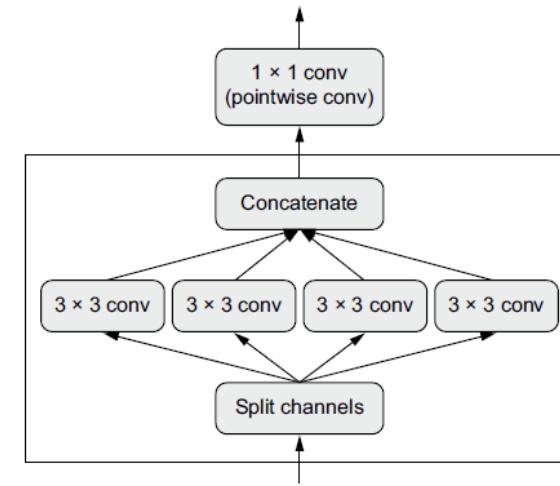
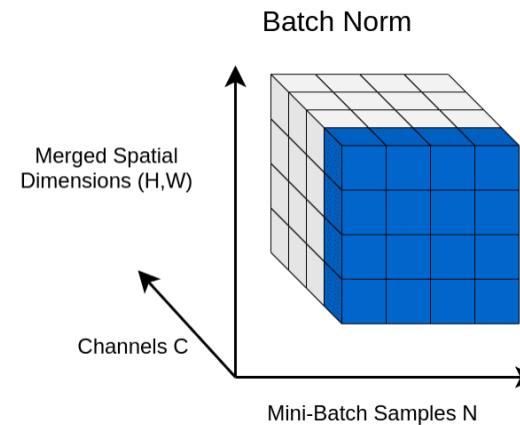
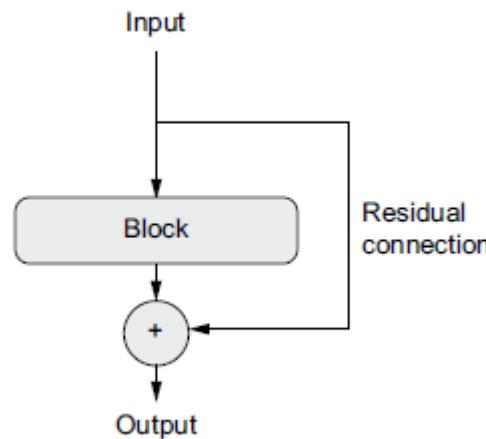


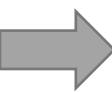


# Convnet architecture best practices

- Model architecture is more an art than a science, however, there are some **best practices!**

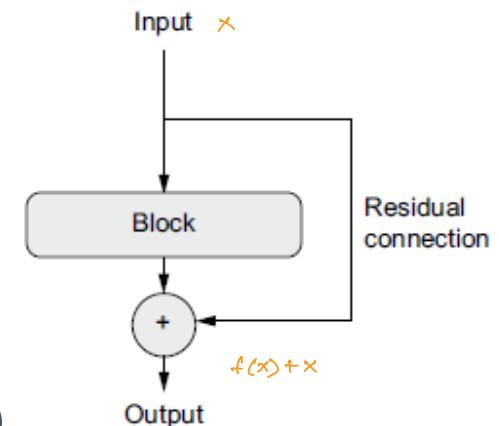
1. Residual Connections
2. Batch Normalization
3. Separable Convolutions

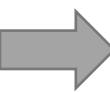




# Residual Connections

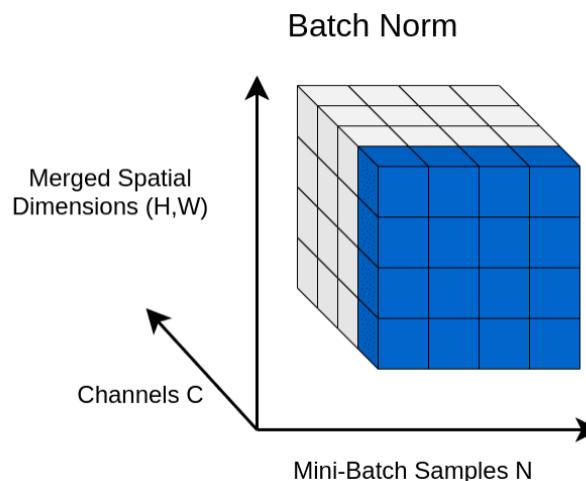
- The game of Telephone (Chinese Whispers)
- If the network is too deep, then successive functions in the chain introduce some **noise!**
- During training (backpropagation), this noise starts **overwhelming** the gradient information.
- **Solution:** add the input of a layer or block of layers back to its output
- Res connections act as *information shortcut* around the noise blocks.
- ResNet family of models was introduced in 2015 (He et al at Microsoft)

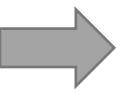




# Batch Normalization

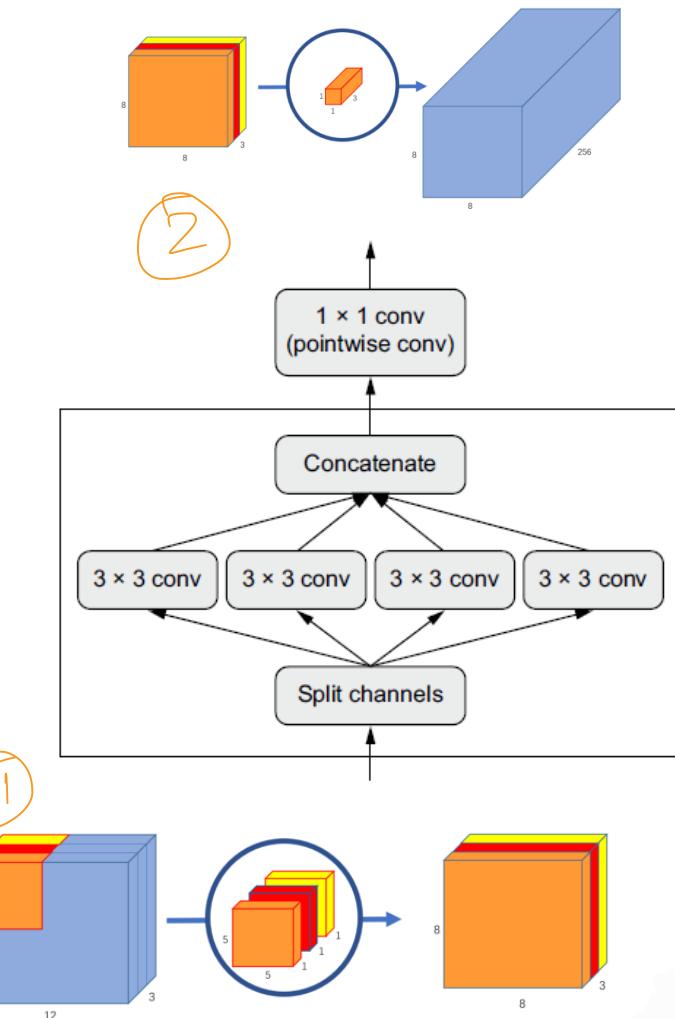
- **Normalization:** Statistical method to make different samples seem more similar to each other.
- Normalization helps the model **learn** and **generalize** better to new data.
- **Batch normalization:** Normalizes the mean and standard deviation for each individual channel /feature map (intermediate activations) **using the current batch of data**.
- Batch normalization helps with gradient propagation, allowing for deeper networks.

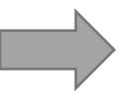




# Separable Convolutions

- Depthwise separable convolution layers, perform spatial convolution on each channel of the input, independently and then mixing output channels using a pointwise convolution.
- Separable convolutions will:
  - Make the model **smaller** (fewer trainable parameters)
  - Require fewer float point operations (seen this before)
  - **Improve** model performance

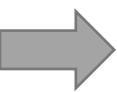




# Best practices summary

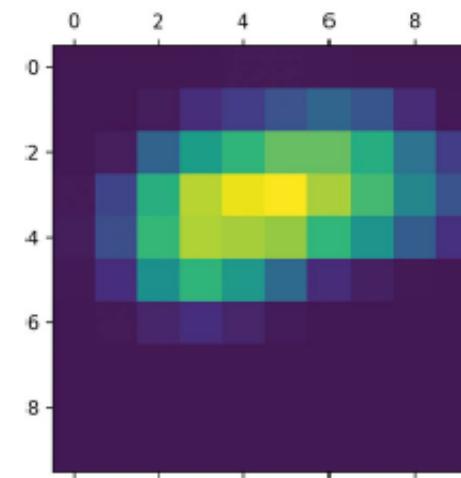
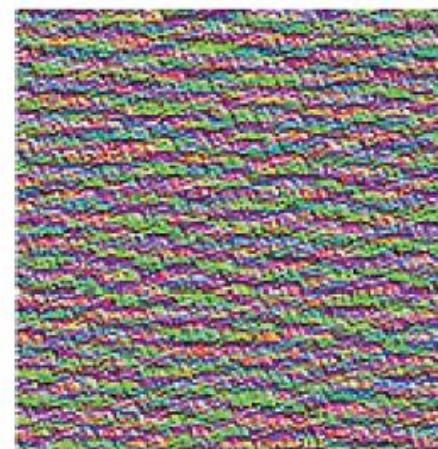
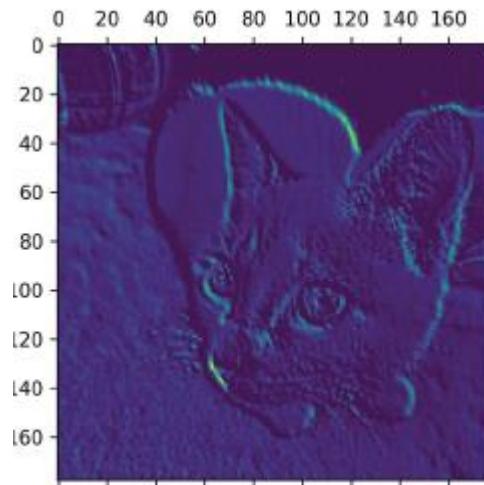
- Your model should be **organized into repeated blocks of layers**, usually made of multiple convolution layers and a max pooling layer.
- The number of **filters** in your layers should increase as the size of the spatial feature maps **decreases**.
- Deep and narrow is better than broad and shallow.
- Introducing **residual connections** around blocks of layers helps you train deeper networks.
- It can be beneficial to introduce **batch normalization** layers after your convolution layers.
- It can be beneficial to replace Conv2D layers with SeparableConv2D layers, which are more parameter-efficient.

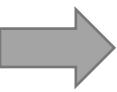
# Interpreting Convnets



# Visualizing what convnets learn

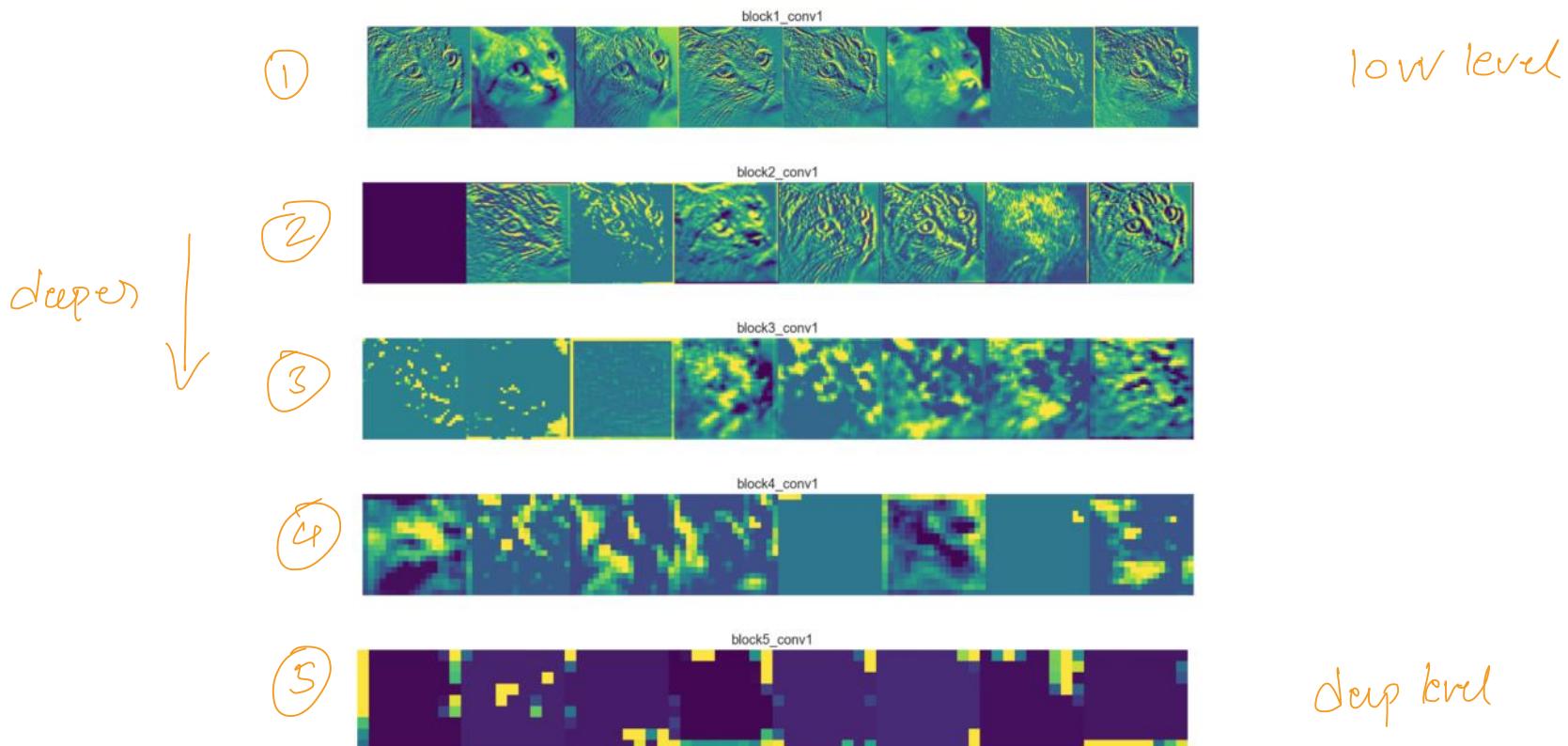
- Visualizing intermediate **convnet output**
- Visualizing **filters**
- Visualizing **heatmaps** (CAM: Class Activation Map)

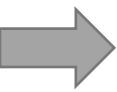




# Visualizing conv outputs

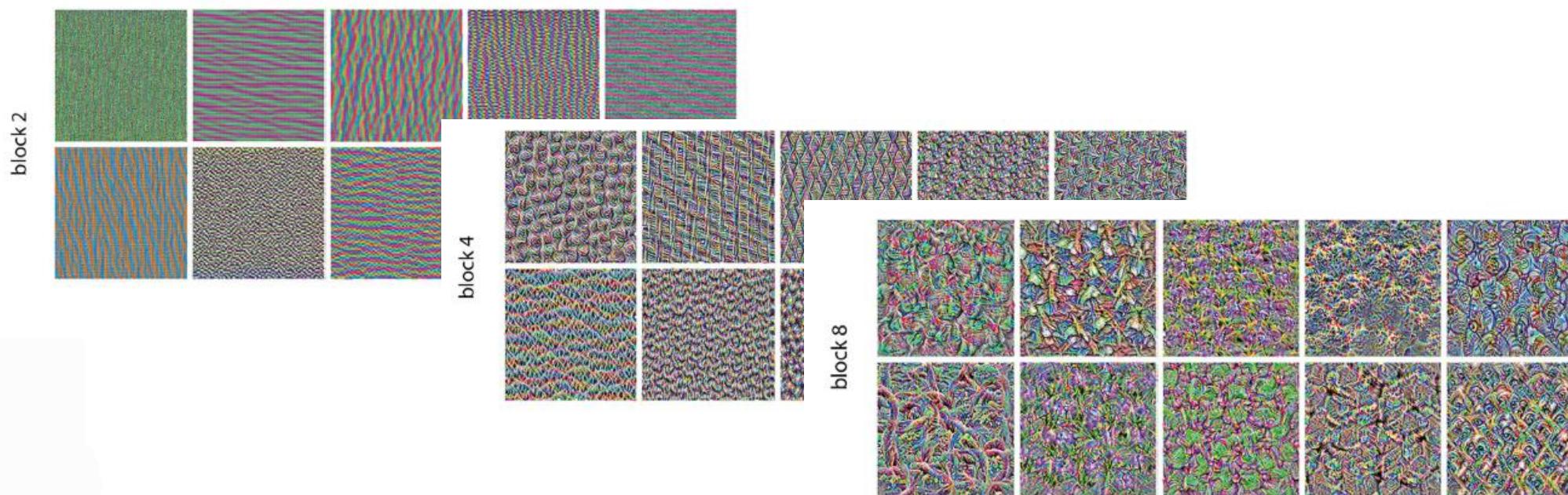
- Low level layers act as a collection of various line and edge detectors
- Deeper presentations carry increasingly less information about the visual contents of the image (more abstract) and more information related to the class of the image.

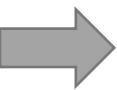




# Visualizing Convnet Filters

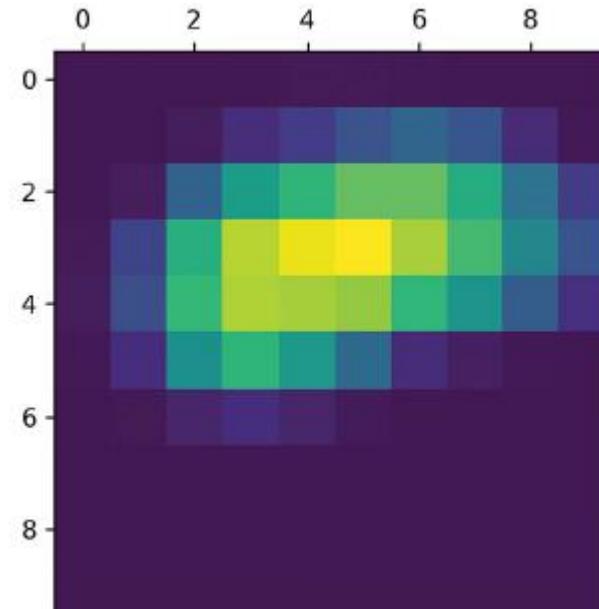
- The filters from the first layers encode simple directional edges and colors
- The filters deeper in the model encode simple textures made from combinations of edges and colors.
- The filters in higher layers begin to resemble textures found in natural images: eyes, ears, and so on.



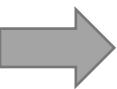


# Visualizing Heatmaps (CAM)

- Class Activation Map (CAM): Producing heatmaps of class activation over input
- CAM indicates how important each location is with respect to the class under consideration.

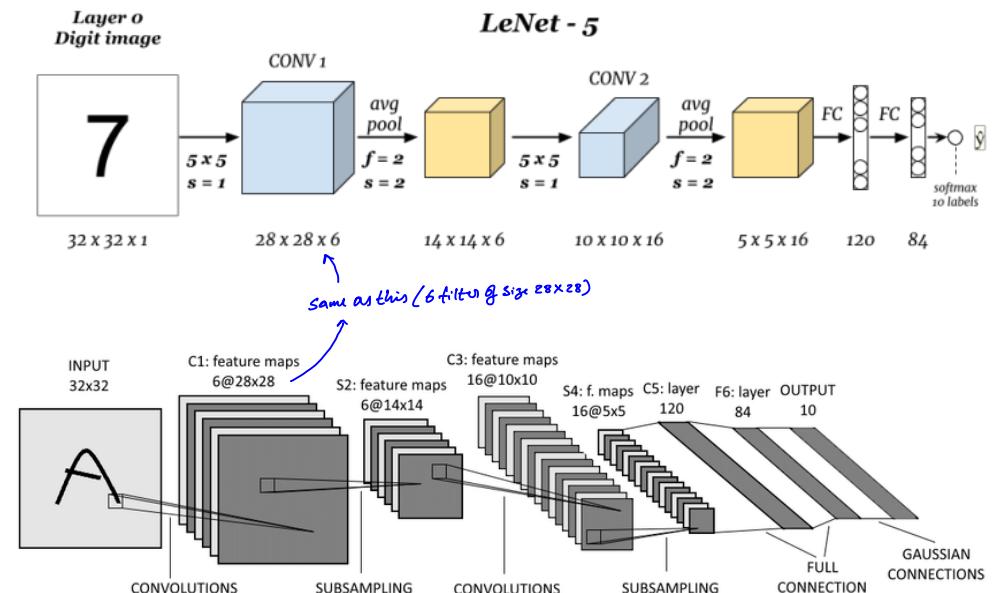


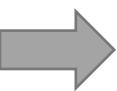
# Transfer Learning



# LeNet – 5

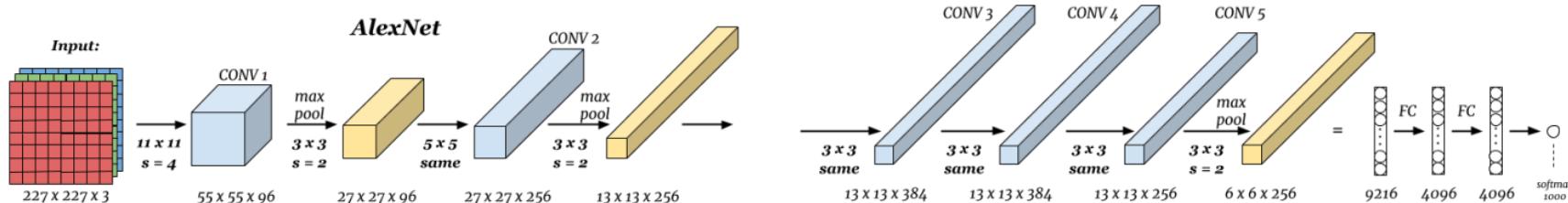
- LeNet is a deep convolutional neural network architecture that was developed by Yann LeCun et al. in **1998**.
- It was one of the first successful applications of deep learning to the task of **handwritten digit recognition** and was a breakthrough in the field of computer vision.
- LeNet architecture consists of a series of **convolutional and pooling layers** (to extract features and reduce the size of feature map) and **few fully connected layers**.
- The LeNet architecture has been influential in the development of subsequent deep learning models and is still widely used as a baseline model for various tasks in computer vision



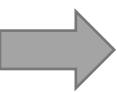


# AlexNet

- AlexNet was developed by **Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton** in **2012**
- AlexNet was one of the **first CNNs to achieve significant success in image classification** and popularized CNN for this task.
- It was the **first CNN to use rectified linear units (ReLUs)** as the activation function, which helped to improve the training speed and accuracy of the model.
- AlexNet used a large number of filters in its convolutional layers, which allowed it to learn **more complex features** from the input data

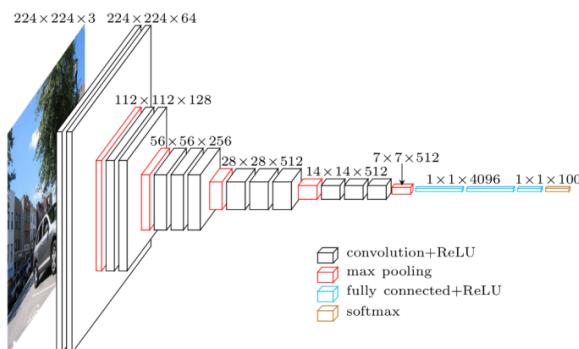


ImageNet Classification with Deep Convolutional Neural Networks paper by Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever (2012).

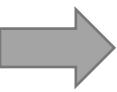


# VGG-16

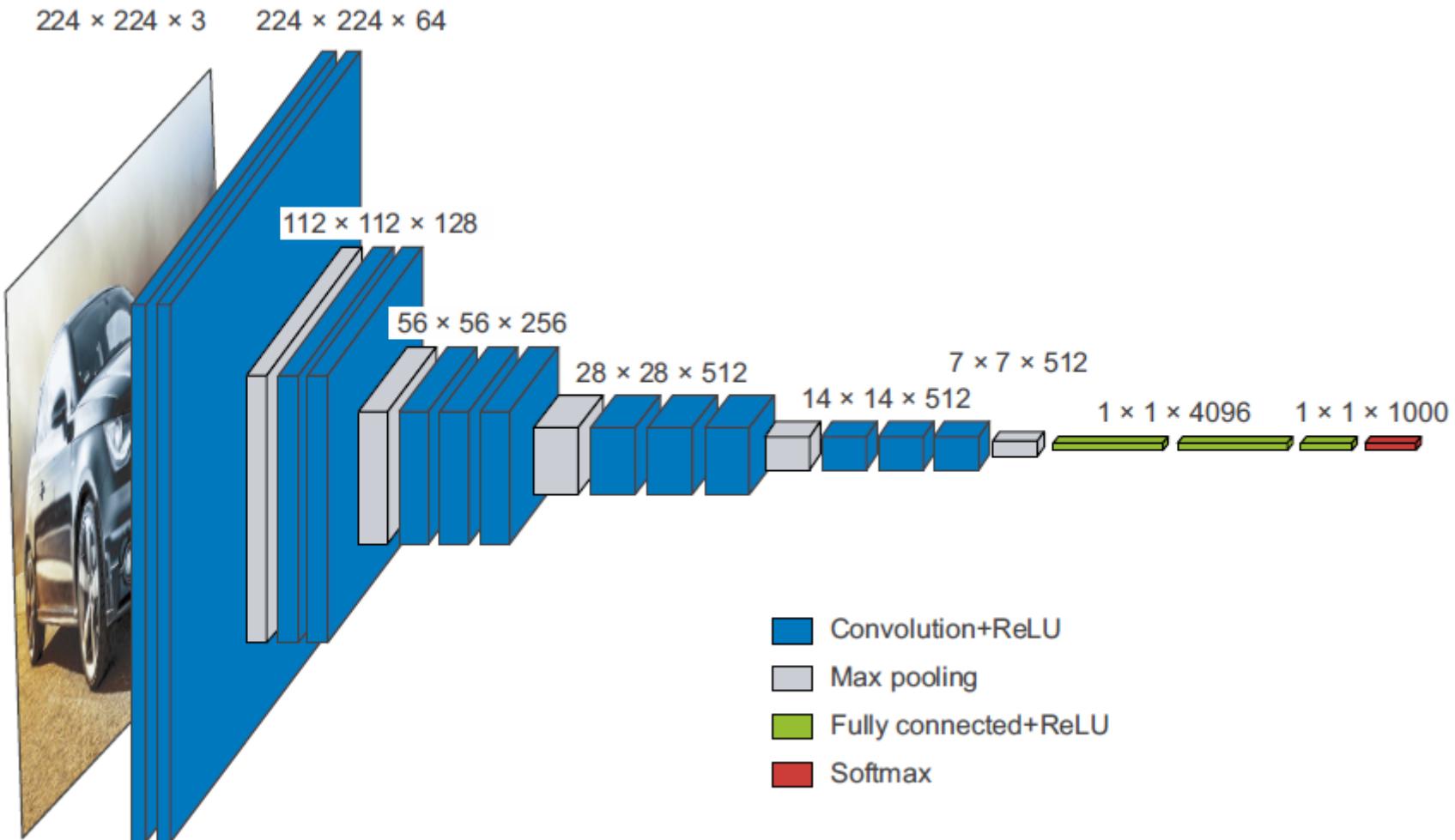
- VGG16 is a CNN developed by [Karen Simonyan](#) and [Andrew Zisserman](#) in [2014](#) at [Visual Geometry Group](#) (VGG) at the University of Oxford.
- The number **16** refers to the fact that the network has **16 trainable layers** (i.e. layers that have weights)
- VGG's strength is in the **simplicity**: the dimension is halved, and the depth is increased on every step (or stack of layers)
- Combination of a **deep architecture**, **simple design**, and **relatively small number of parameters** has contributed to the success of VGG16 for tasks in computer vision/

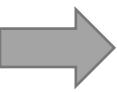


[Very Deep Convolutional Networks for Large-Scale Image Recognition](#) paper by Karen Simonyan and Andrew Zisserman (2014).



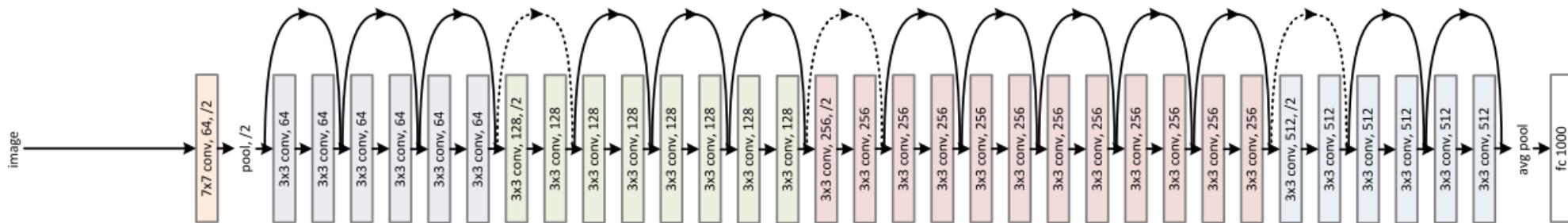
# VGG-16



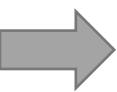


# ResNet

- ResNet (Residual Network) is a CNN that was developed by [Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in 2015](#) at Microsoft.
- The key innovation of the ResNet architecture is the use of [residual connections](#)
- This allows the model to learn [much deeper networks](#) more effectively
- ResNet has been used as a building block for other architectures, such as Mask R-CNN and U-Net.
- There are several variants of the ResNet architecture, including [ResNet-50](#), [ResNet-101](#), and [ResNet-152](#), which differ in the number of layers and the number of parameters.



Deep Residual Learning for Image Recognition. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015)



# Comparing classical models

- The **ImageNet** dataset consists of more than 1 million images and 1000 categories. The categories are organized into a hierarchy, with each category having multiple subcategories.
- ImageNet has played a significant role in the **development** of deep learning and has contributed to the success of many state-of-the-art models for tasks in computer vision.

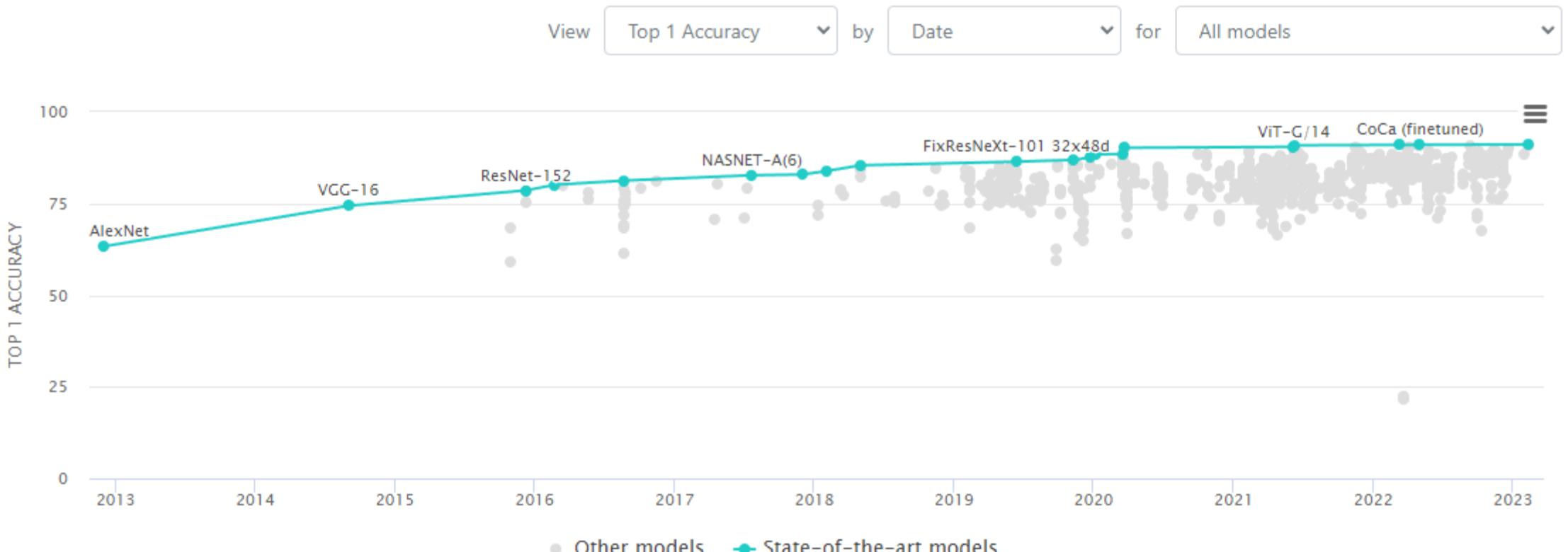
Network	Parameters	Top1 accuracy
LeNet 5	~ 60,000	-
AlexNet	~ 60 million	63.3%
VGG16	~138 million	74.4%
ResNet 152	~ 60 million	78.57%

• Accuracy for Multiclass Classification

Top 1 Accuracy → is my prediction the top prediction

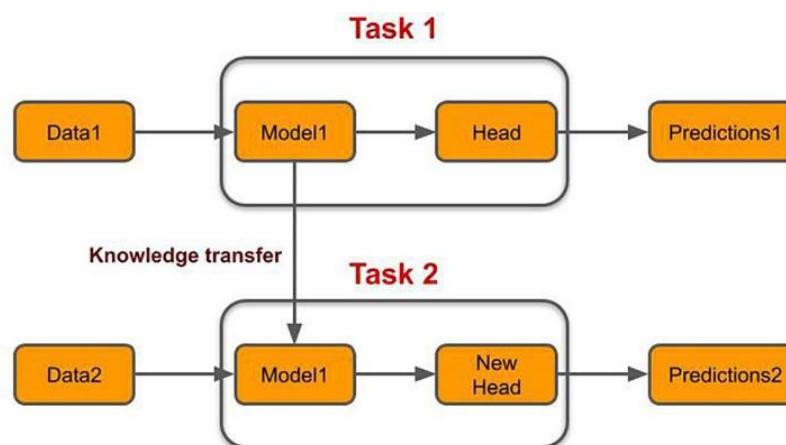
Top 3 Accuracy → is my prediction in the top 3 predictions.

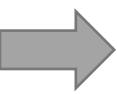
# State-of-the-art models



# Transfer Learning

- Transfer learning is a machine learning technique where a model trained on one task is **re-purposed** on a second related task.
- We use this technique to leverage a model's **backbone** by popping off its **head (the last few layers)** and replacing it with untrained layers that are appropriate for our task.
- Transfer learning is useful because it allows the model to **leverage** the knowledge it has gained from the first task and apply it to the second task, potentially leading to **improved performance** on the second task.

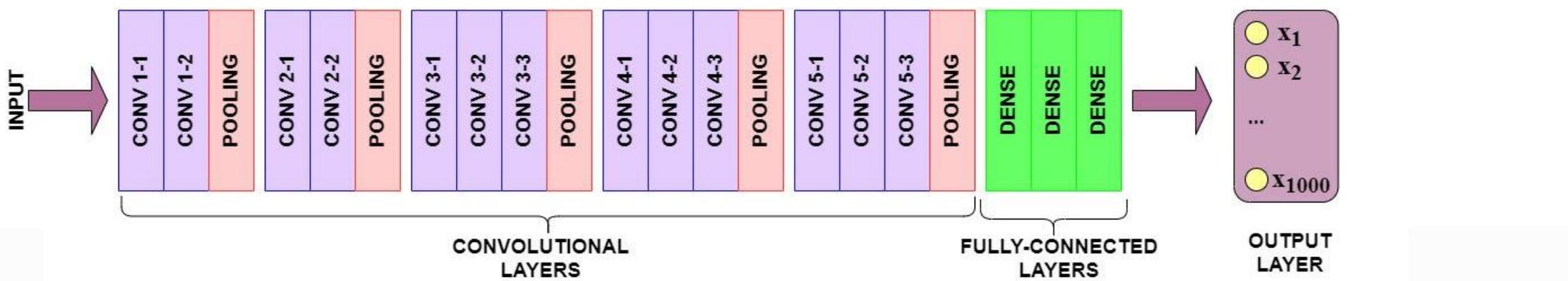




# Transfer Learning

There are two main ways to perform transfer learning:

- **Feature Extraction**: the pre-trained model is used to extract features, and these features are then fed into a new model that is trained to perform the new task.
- **Fine-tuning**: unfreezing some of the layers of the pre-trained model + training and fine tuning these layers on the new task

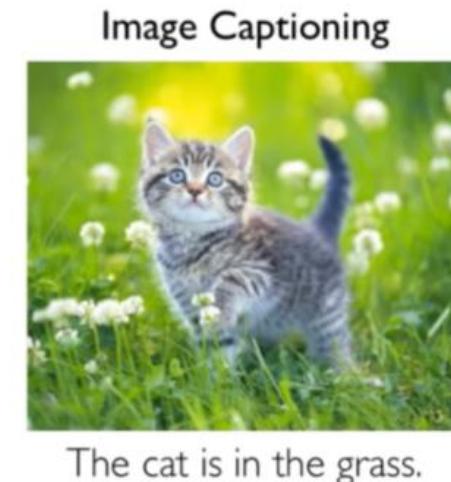
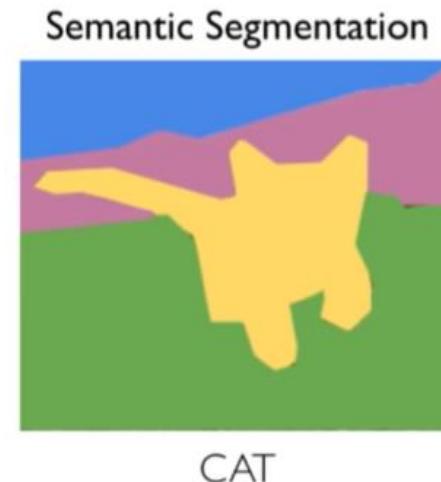
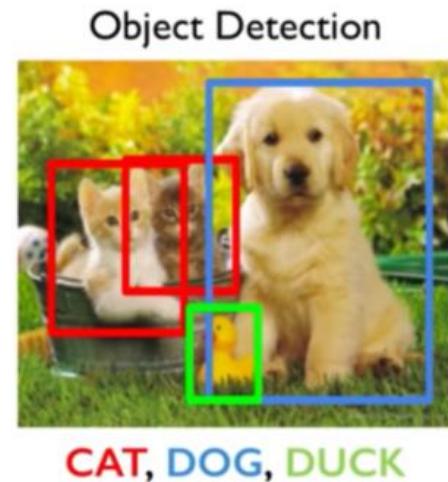


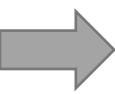
# Module 5 – Part IV

## Deep Computer Vision

### Beyond Classification

---



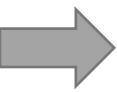


# Beyond Classification

Feature	Object Detection	Image Segmentation	Image Captioning
Task	Identify and locate objects in an image	Classify each pixel in an image into a specific category	Generate a textual description of an image
Output	Bounding boxes and labels for objects	A label for each pixel in the image	A written description of the image
Examples	Identifying and localizing cars, pedestrians, and other objects in an image	Assigning labels to each pixel in an image to create a segmentation map, such as "road", "car", "sky"	"The cat is in the grass"

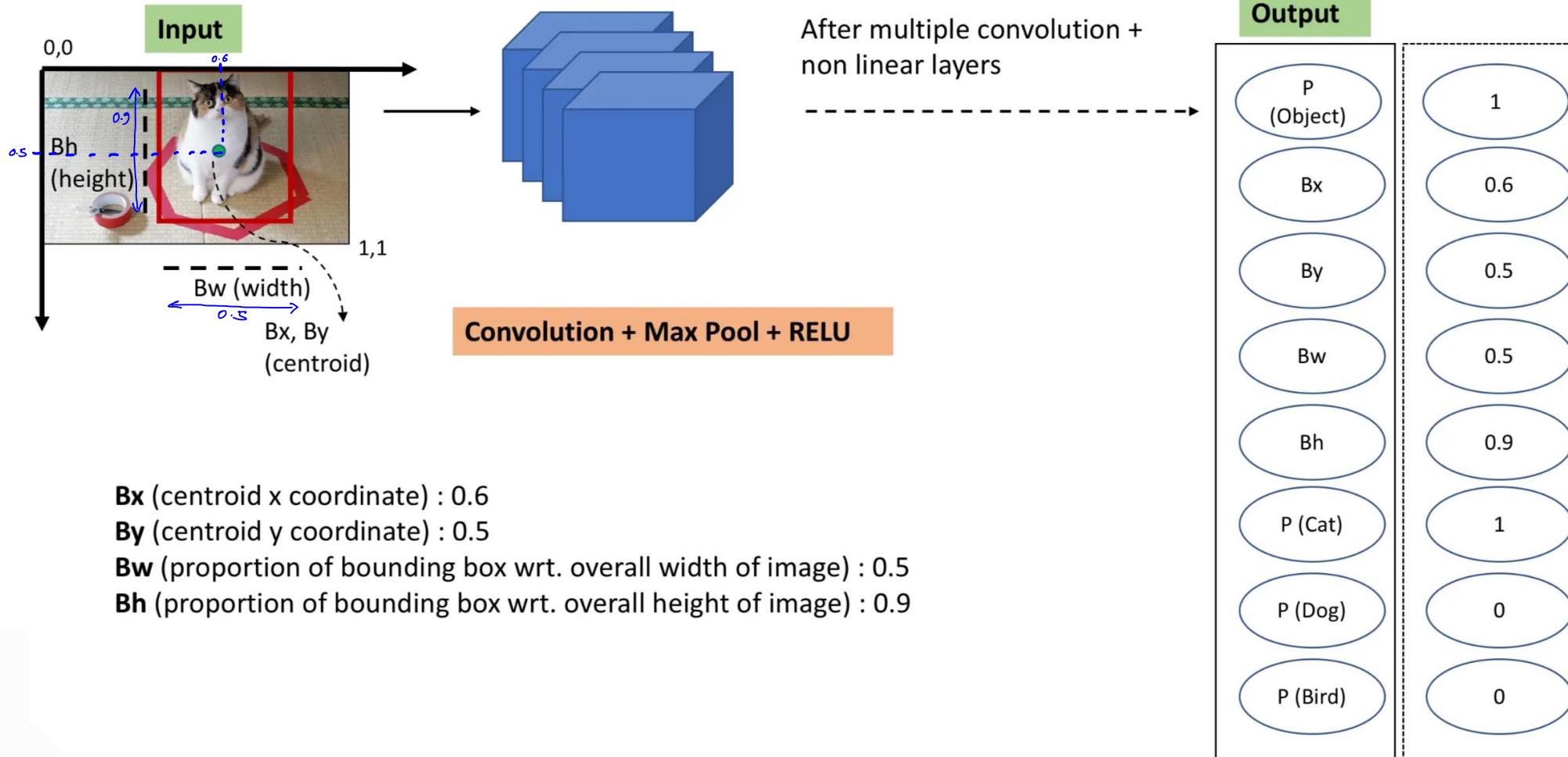


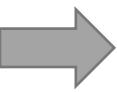
# Object Detection



# Object Classification and Localization

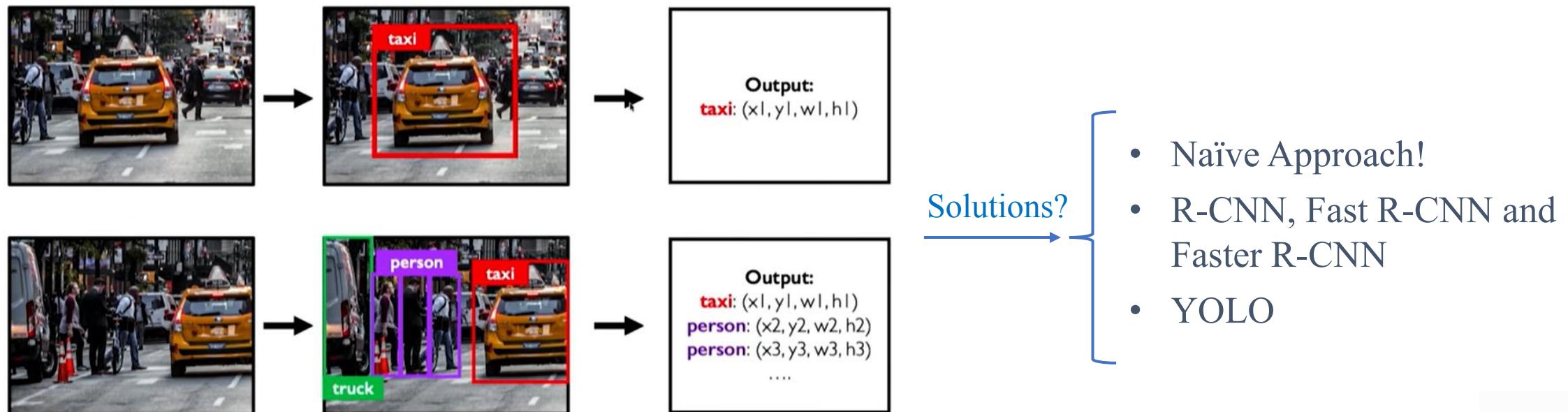
Cat/ Dog/ Bird classifier and localizer

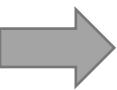




# Object Detection

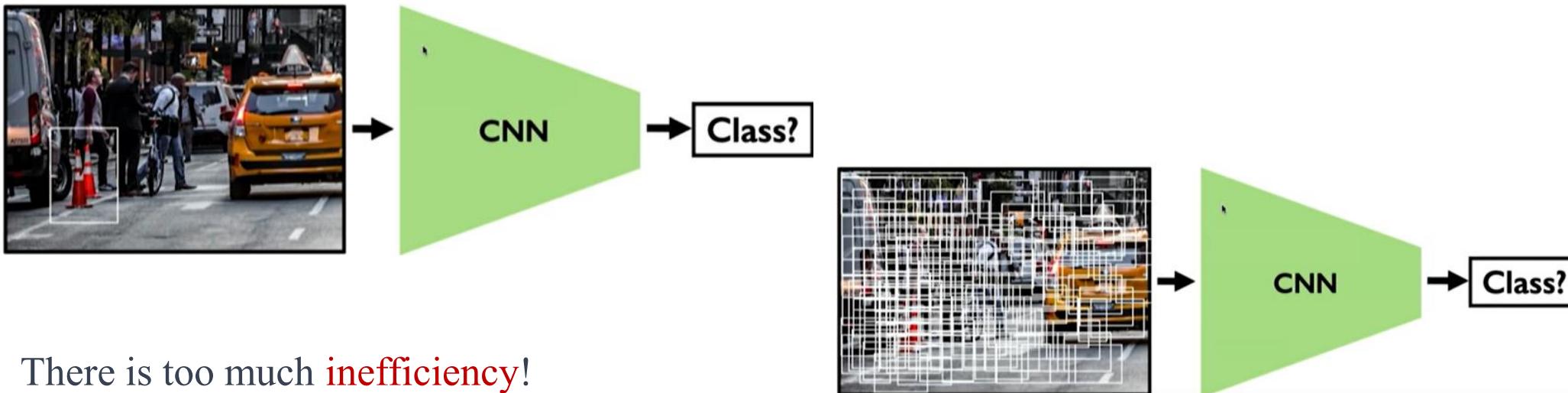
- Object detection is a task that involves **identifying** and **locating** objects in images or video.
- The model learns to recognize the **objects** and predict **bounding boxes** around them in new images.



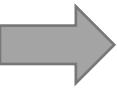


# Object detection: Naïve Approach

- Scan the image with a **sliding window**!
- Place a random **box** (**position** and **size**) somewhere in the image!
- Feed this box into a CNN and predict the class.
- Repeat this with different random boxes!!!

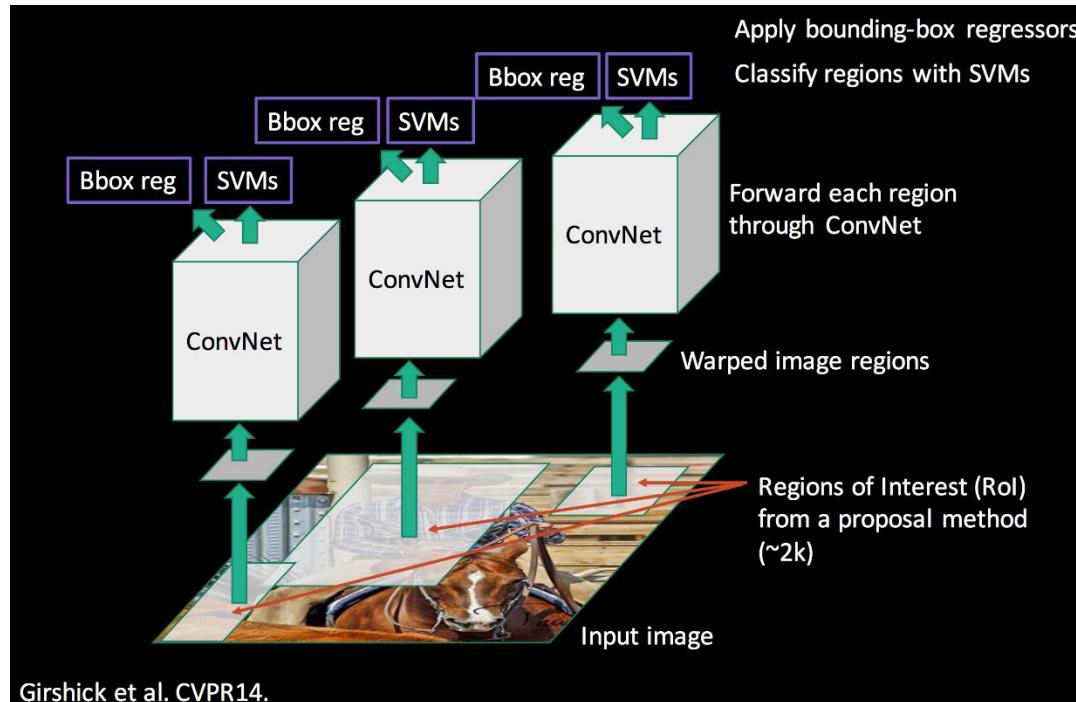


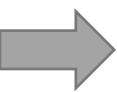
- There is too much **inefficiency**!
- Computationally expensive and **too slow** for real time uses!



# Region-based Convolutional Neural Network (R-CNN)

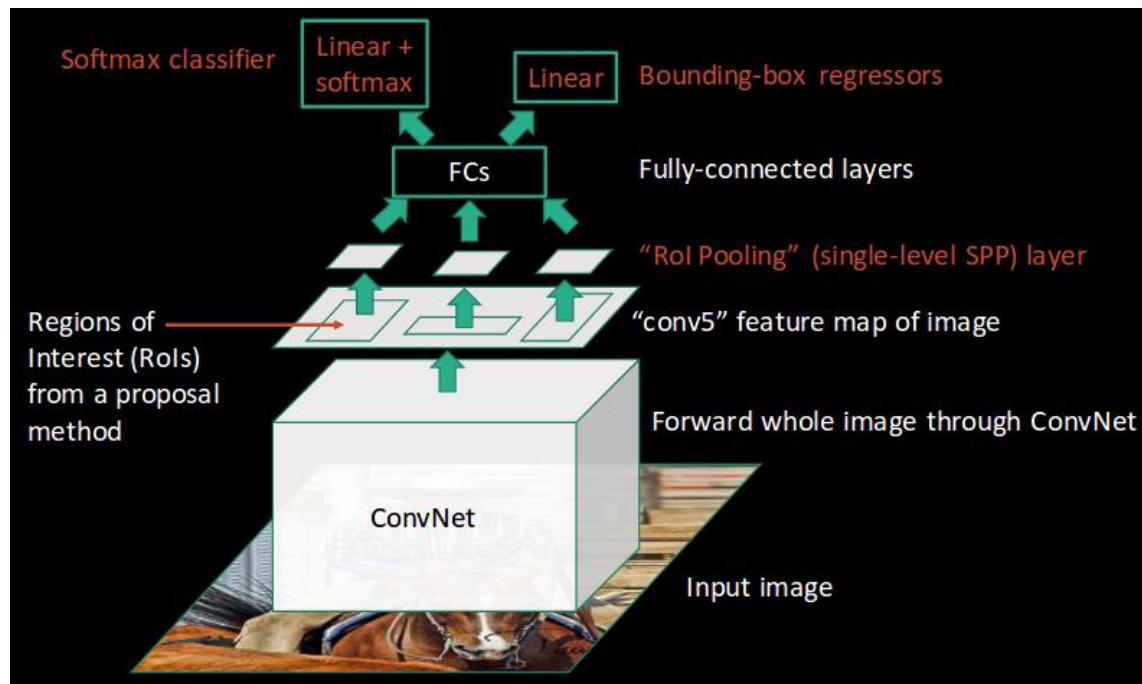
- Strategically selects **interesting regions** to run through the classifier.
- Regions of Interest (**ROI**) are the regions that we **think** have objects with **higher probabilities**.
- Problems:
  - Very **slow** (classify 2k region proposals per image)! No real time use case!
  - How to define **region proposals**? The selective search algorithm is a fixed algorithm detached from the network.

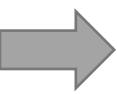




# Fast R-CNN

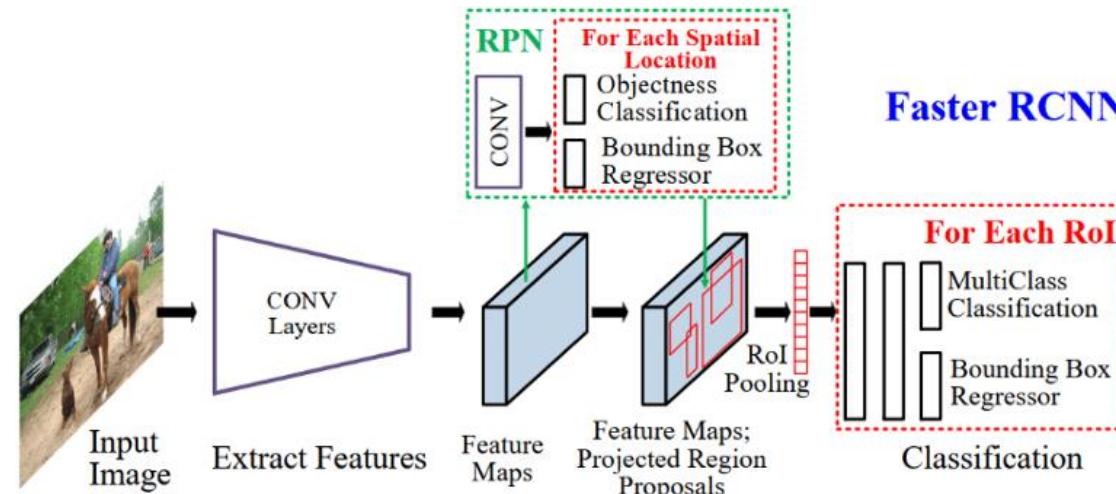
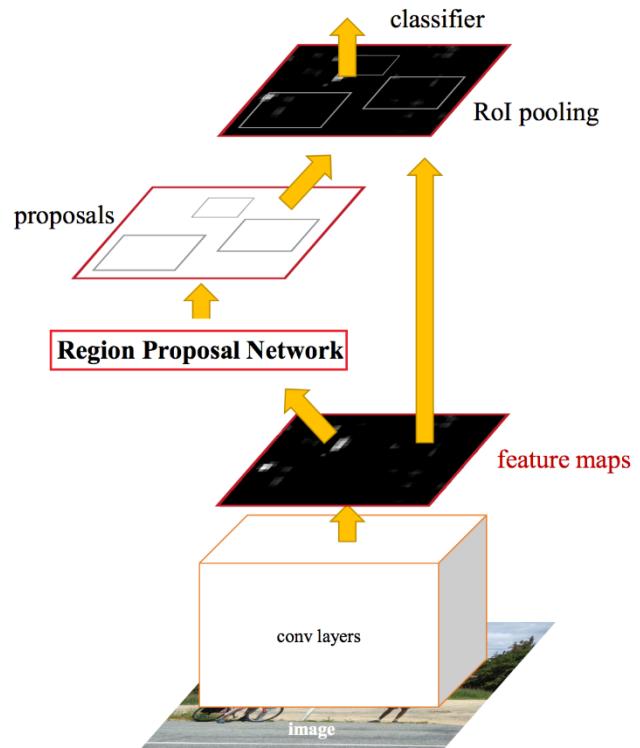
- The approach to Fast R-CNN is similar to the R-CNN algorithm. But, instead of converting 2,000 regions into the corresponding features maps, we convert the whole image once.
- Problems:
  - Still uses selective search to generate the RoIs (detached from the network)

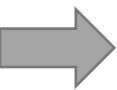




# Faster R-CNN

- In Faster R-CNN, instead of using **selective search algorithm on the feature map** to identify the region proposals, a **separate network** is used to predict the region proposals.
- The predicted region proposals are then **reshaped** using a **RoI pooling layer** which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

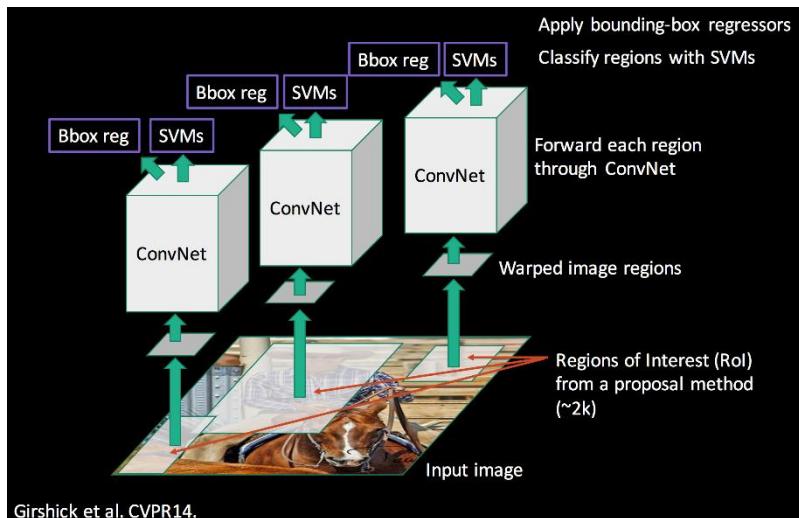




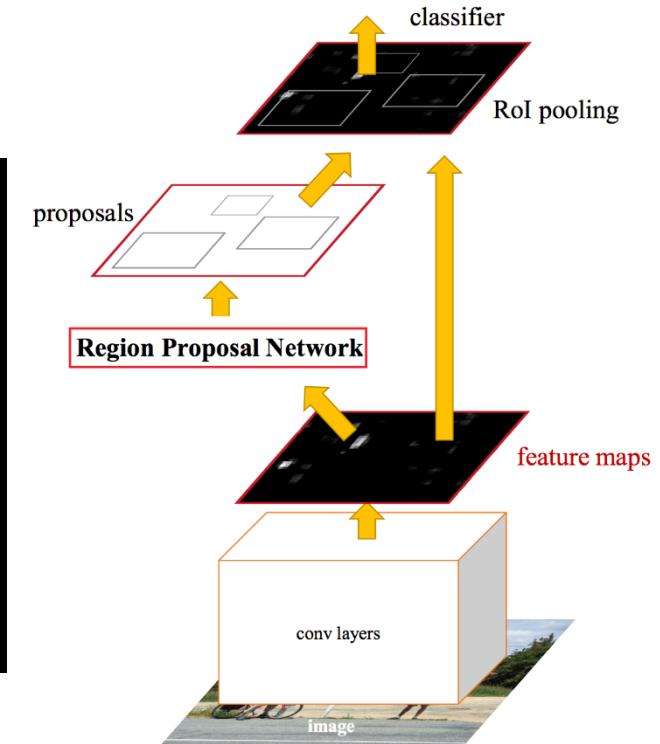
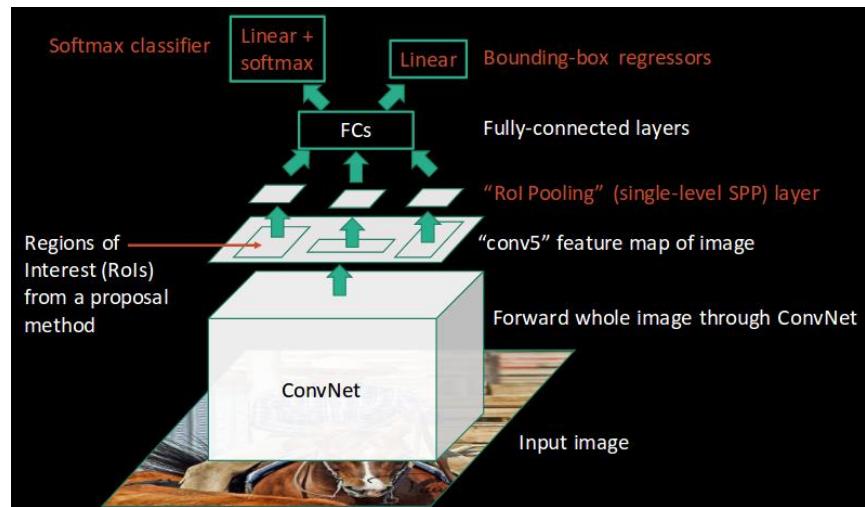
# Comparing R-CNN-based models

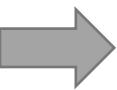
2016 - Faster RCNN

2014 - R-CNN



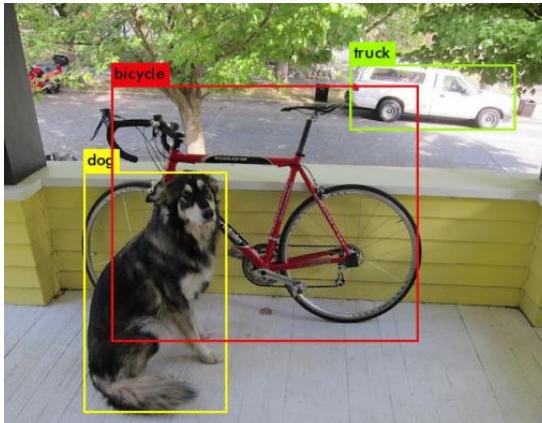
2015 - Fast RCNN

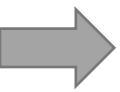




# YOLO (You Only Look Once!)

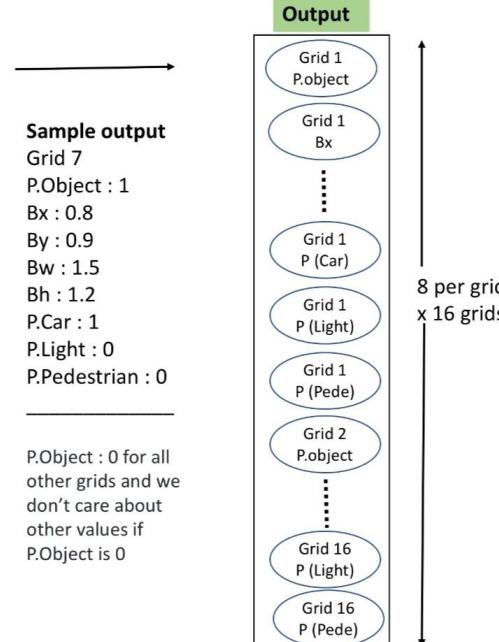
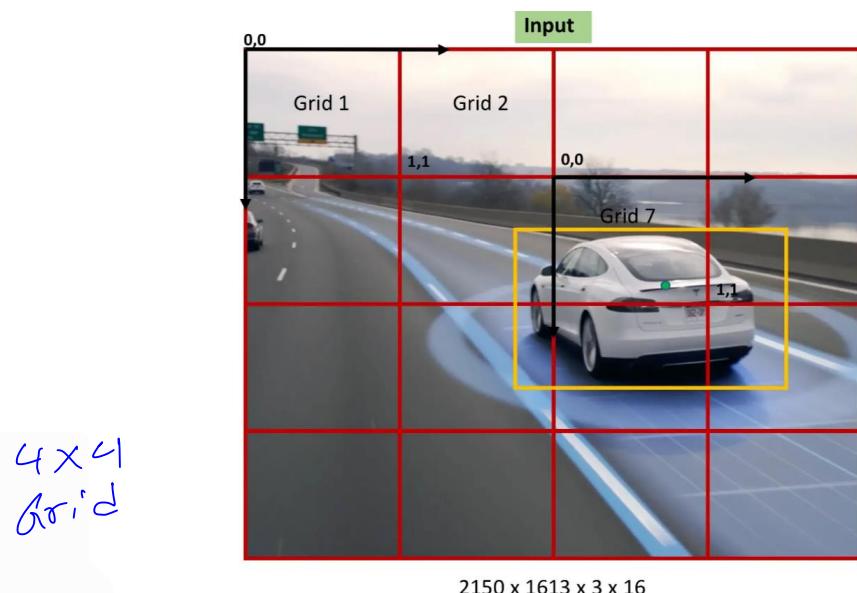
- YOLO is a **real-time** object detection algorithm. It was developed by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi at the University of Washington (2015)
- Yolo is extremely fast because it passes **the entire image at once** into a CNN, rather than making predictions on many individual regions of the image.
- The key idea behind YOLO is to use a single neural network to predict the **bounding boxes** and **class probabilities** for objects in an image





# YOLO (How it works?)

- YOLO divides the input image into a grid of cells and **predicts the presence of objects** in each cell
- If an object is detected in a cell, the algorithm also predicts the **bounding box** and the **class** for the object
- The bounding box coordinates and class probabilities are then used to **localize** and **classify** the objects



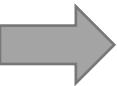
Output Size =  $(S + C) \times (S \times S) = (S+3) \times (4 \times 4) = 8 \times 16 = 128$  outputs to be predicted.

Size of Input image Grid

Prob. of Object  
• By  
• Bx  
• Bw  
• Bh

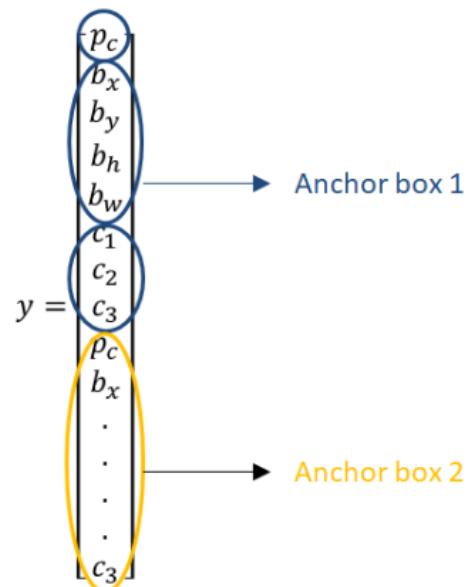
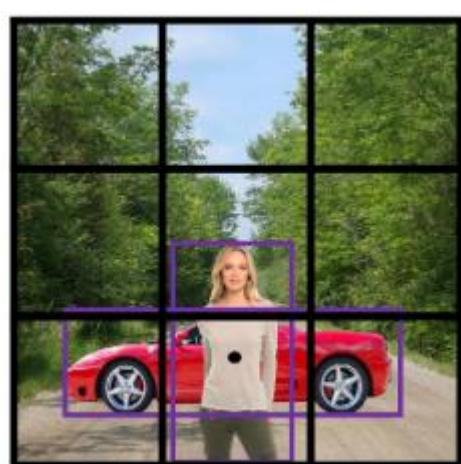
Prob. of classes  
i.e. Car, Light or Pedestrian.  
3 classes.

Output size:  
 $(5+C) \times S \times S$



# YOLO – Anchor boxes

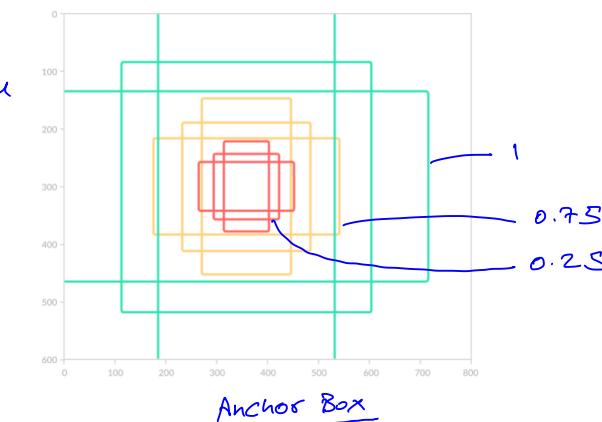
- One of the Caveats of YOLO is that it can't detect multiple objects in same grid.
- Solution: Anchor boxes. It is a **predefined** bounding box used in object detection algorithms.
- The anchor box is used to define the **size** and **aspect ratio** of the window, and it is defined **prior to training** the object detection model. The model is then trained to predict the bounding box coordinates and class probabilities for objects relative to the anchor box.

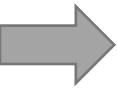


$C \rightarrow 3$  classes, Person, Car, Tree

Output size:  
 $(5+C) \times B \times S \times S$

$B \rightarrow$  Anchor Boxes  
 $(5+3) \times 3 \times 3 \times 3$



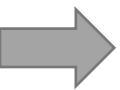


# YOLO – Non-max Suppression (NMS)

- Another caveat of YOLO is the possibility of **detecting one object multiple times!**
- Solution: Non-max Suppression which **removes duplicate or overlapping bounding boxes**.
- In NMS, the algorithm iterates through all the bounding boxes in the image and selects the box with the **highest confidence score**.
- It then removes all other boxes that **overlap** with the selected box by a certain threshold. This process is repeated until all the boxes have been processed.



# Image Segmentation



# Image Segmentation

- Semantic segmentation is the process of assigning a class label **to each pixel** in an image, based on the pixel's visual characteristics and its relationship to the surrounding pixels.
- Instance segmentation involves not only classifying each pixel in the image, but also distinguishing between different instances of the same class.

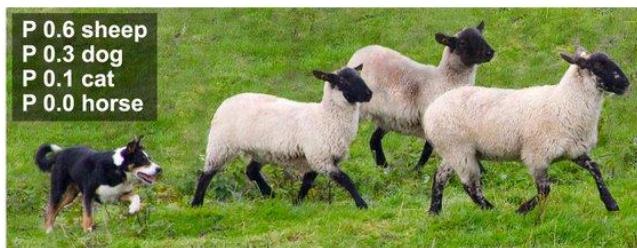
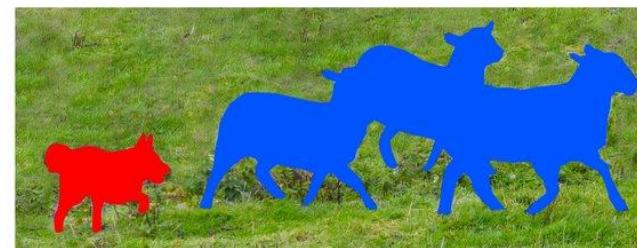
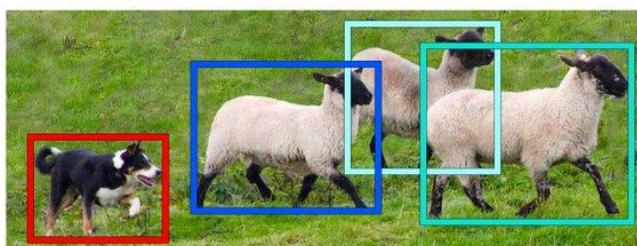


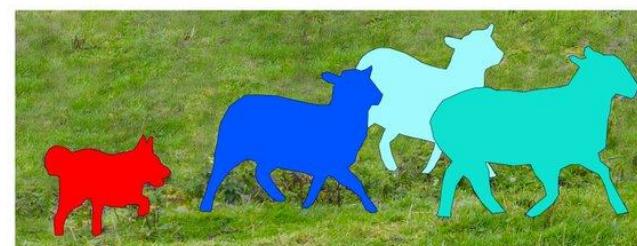
Image Recognition



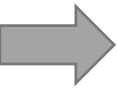
Semantic Segmentation



Object Detection

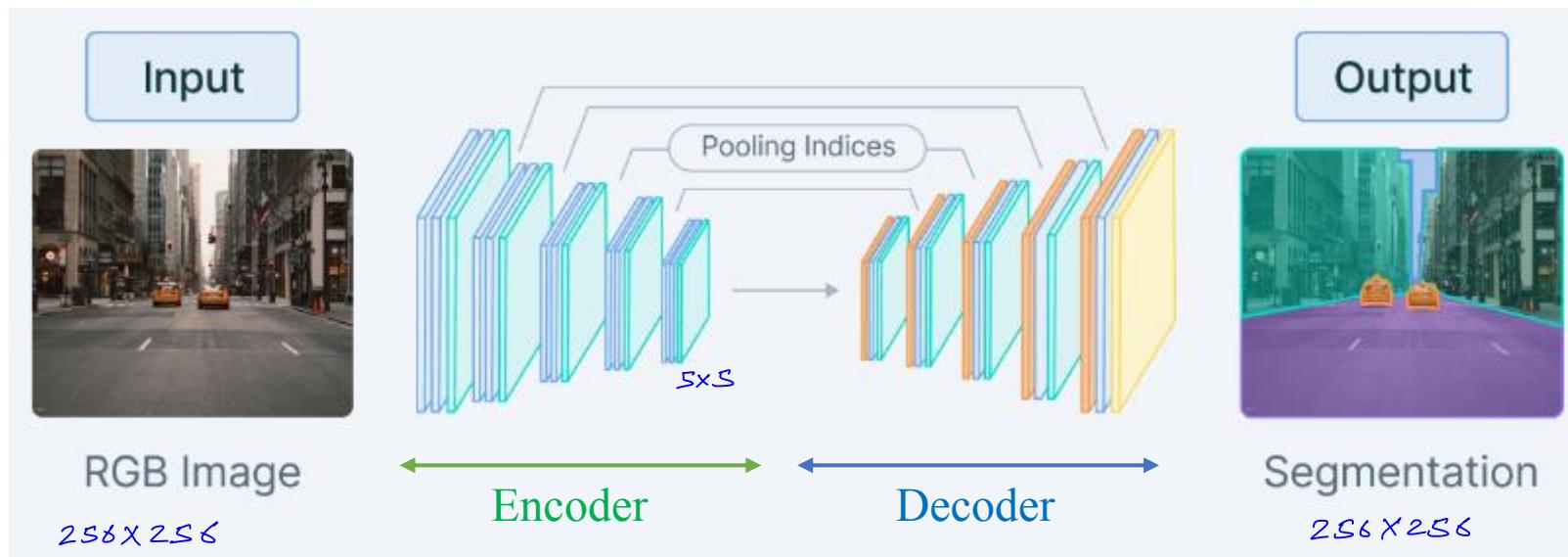


Instance Segmentation

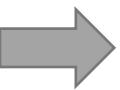


# Fully Convolutional Network (FCN), semantic segmentation

- In an FCN, the input image is passed through a **series of convolutional layers**, which extract features from the image and **reduce its spatial resolution (Encoder)**.
- The final layers of an FCN are typically **transposed convolutional layers**, which **upsample** the feature map to the original resolution of the input image (**Decoder**), and output a dense prediction for each pixel in the image

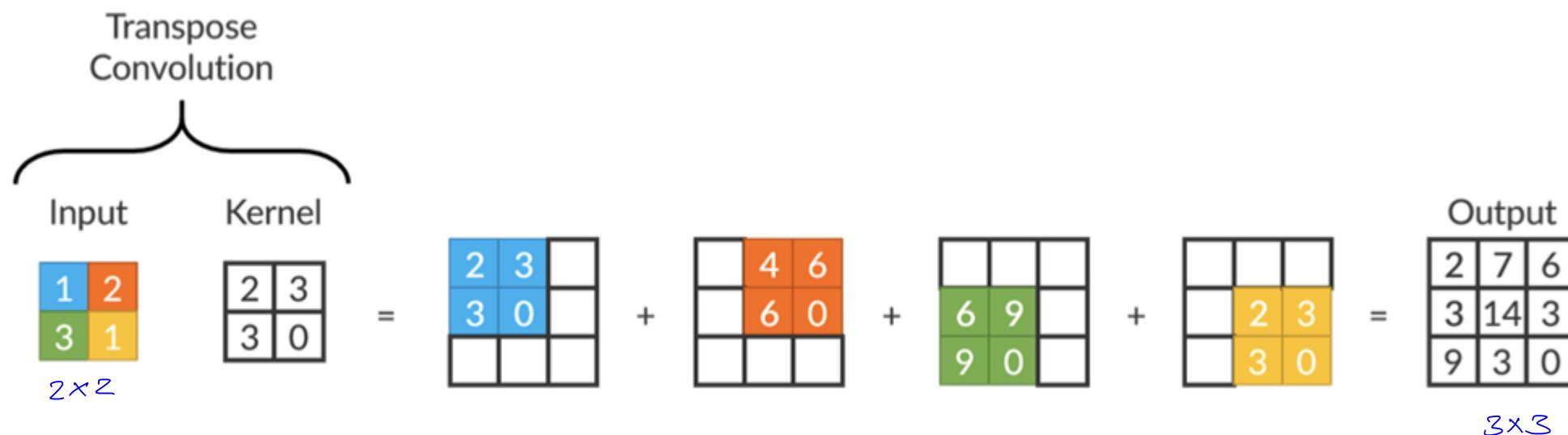


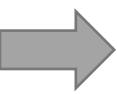
Note: No Fully Connected Layers.



# Transposed Convolution

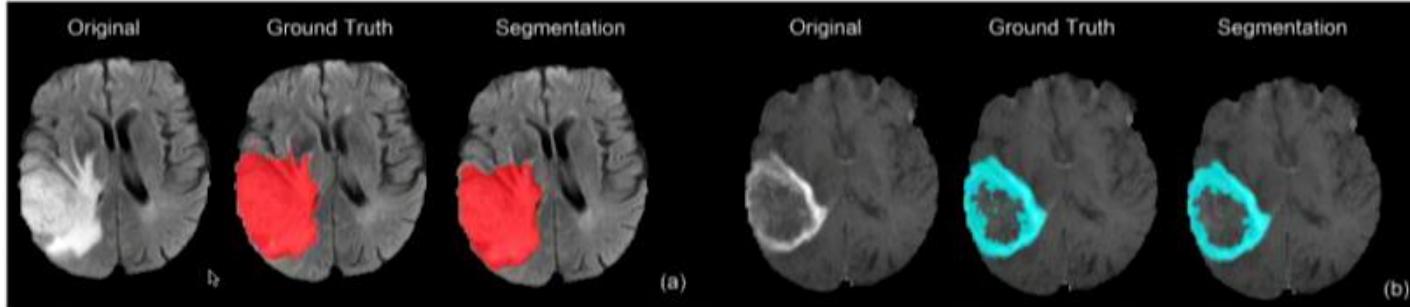
- A transpose convolution, also known as a deconvolution, is a type of convolutional layer that is used to **upsample** the feature map produced by a convolutional neural network.
- It takes an input feature map and produces an output feature map **that is larger in size**, by **inserting zeros** between the elements of the input feature map and convolving with a set of filters



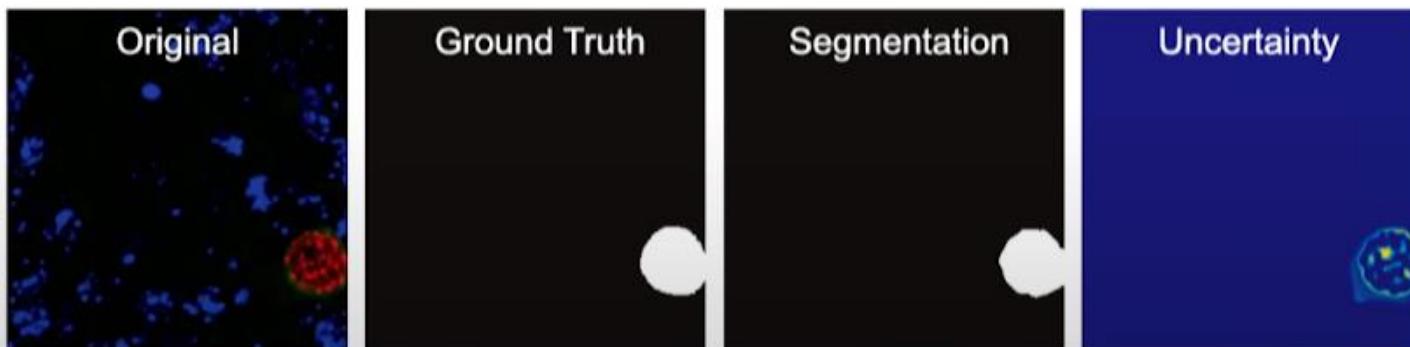


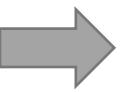
# Semantic Segmentation applications

Brain Tumors  
Dong+ MIUA 2017.



Malaria Infection  
Soleimany+ arXiv 2019.

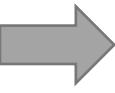




# Semantic Segmentation applications

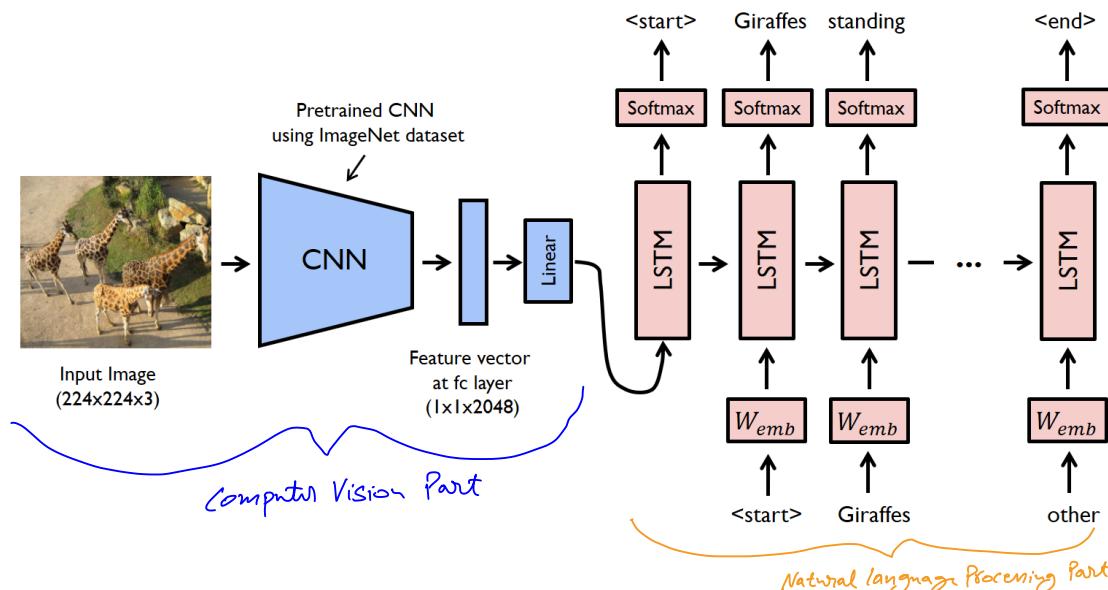


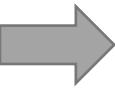
# Image Captioning



# Image Captioning

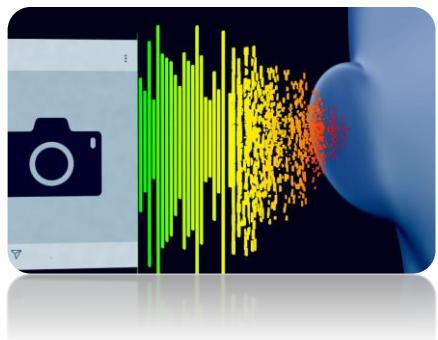
- Image captioning is the task of **generating a textual description of an image**, based on the objects, actions, and scenes depicted in the image.
- Image captioning models typically consist of two parts:
  - A **vision model** that extracts features from the input image (CNN)
  - A **language model** that process the features and generates the caption (RNN, LSTM)





# Image Captioning applications

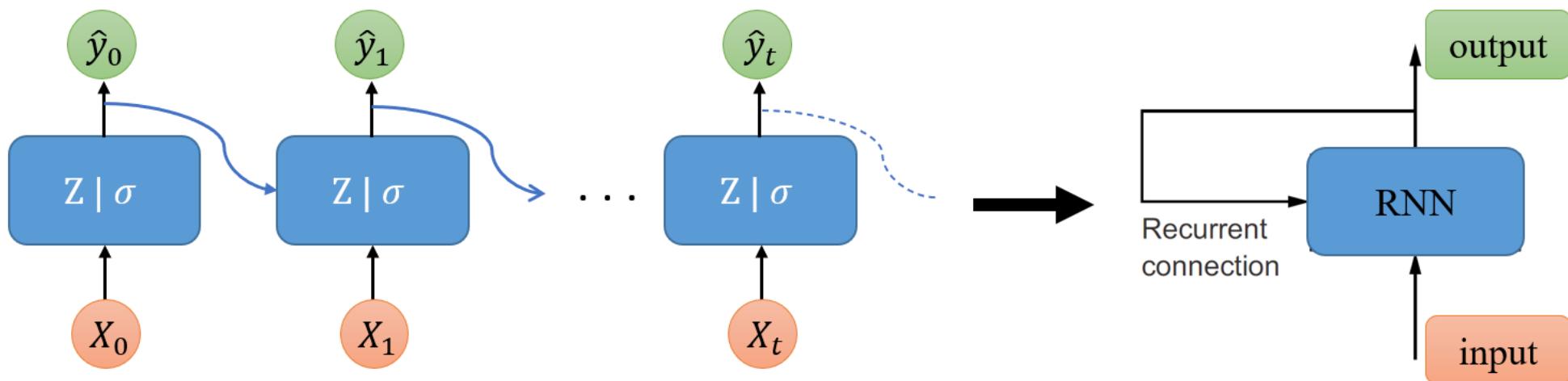
- Generating captions for social media posts and websites
- Creating metadata for image search engines
- Improving the accessibility of images for people with visual impairments

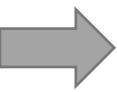


# Module 6 – Part I

## Deep Sequence Modeling

### Recurrent Neural Networks (RNN)

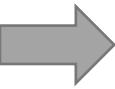




# Road map!

- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- **Module 6- Deep Sequence Modeling (RNN, LSTM, NLP)**
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)



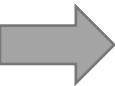


# Different kinds of sequence data

Sequence data refers to any data that has a specific order or sequence to it!

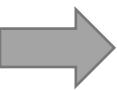
- **Time series data:** sequence of data measured at regular intervals over time. (stock prices, weather patterns, medical records, ...)
  - **Text data:** sequence of data that is composed of words, sentences, or paragraphs. (tweets, news articles, product reviews, ...)
  - **Audio data:** sequence of data that is recorded or generated as sound waves. (speech recordings, music tracks, ...)
  - **Video data:** sequence of data that is represented as a sequence of images or frames. (movie clips, surveillance footage, ...)





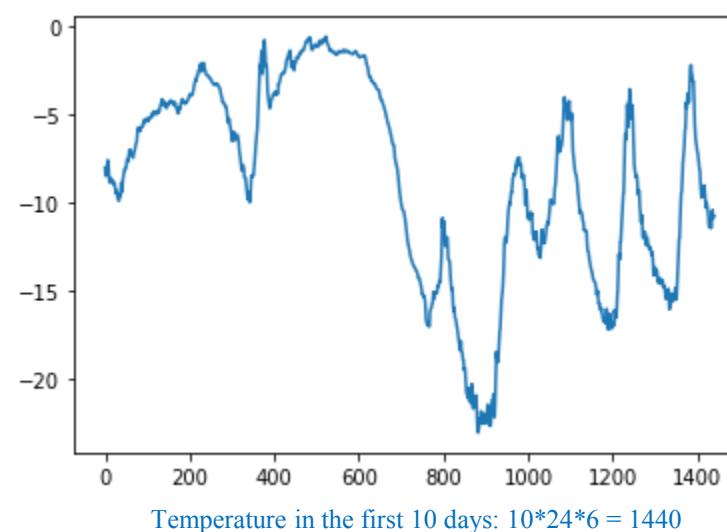
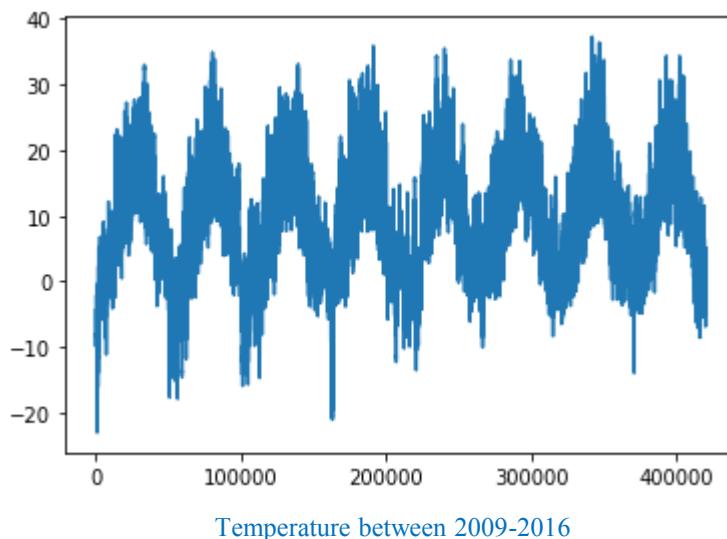
# Different kinds of timeseries tasks

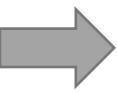
- **Forecasting:** Predicting what happens next  
Electricity consumption, temperature forecast, stock prediction, ...
- **Classification:** Assign one or more labels to a time series  
Classify website visitors as human or bot based on their activity
- **Event detection:** Identify the occurrence of a specific expected event within a continuous data stream  
Hotword detection: “Ok Google” or “Hey Siri”
- **Anomaly Detection:** Detect anything unusual happening within a continuous data stream  
Unusual activity on a network, ...



# A simple timeseries example

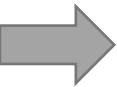
- A temperature forecasting example: [deep-learning-with-python-notebooks](#)
- Predicting the temperature 24 hours in the future
  - Target: temperature
  - Features: 14 different variables including pressure, humidity, wind direction and etc
  - Data recorded every 10 minutes from 2009-2016





# Preparing the data

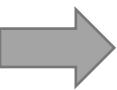
- Given the previous 5 days (120 hours) and samples once per hour, can we predict temperature in 24 hours (after the end of the sequence)?
- Data batches:
  - Sequence length = 120
  - [1,2,3,...,120][144] *Make prediction for this value*  
[2,3,4,...,121][145]  
[3,4,5,...,122][146]
  - Bath size: 256 of these samples are shuffled and batched
  - Sample shape: (256, 120, 14)  $\rightarrow$  Rank 3 Tensor (sample, timesteps, features)
  - Target shape: (256,)  $\rightarrow$  Rank 1 (vector)



# Naïve forecaster: common-sense baseline

- Temperature 24 hours from now = Temperature right now
- This is our random walk with no drift forecaster.
- Performance:
  - Validation MAE = 2.44 degrees Celsius
  - Test MAE = **2.62** degrees Celsius
  - The baseline model is off by about 2.5 degrees on average. Not bad!!

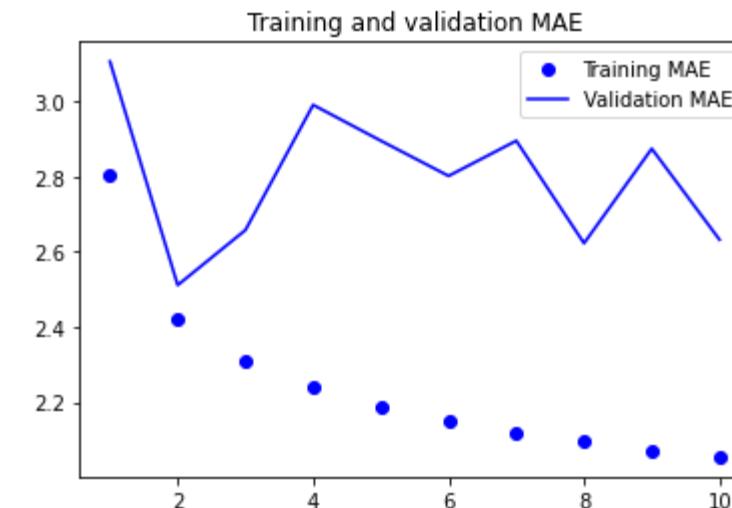




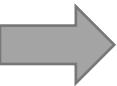
# Let's try DNN (Deep Neural Networks)

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))  
x = layers.Flatten()(inputs)  
x = layers.Dense(16, activation="relu")(x)  
outputs = layers.Dense(1)(x)  
model = keras.Model(inputs, outputs)
```

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 120, 14)]	0
flatten (Flatten)	(None, 1680)	0
dense (Dense)	(None, 16)	26896
dense_1 (Dense)	(None, 1)	17
=====		
Total params:	26,913	
Trainable params:	26,913	
Non-trainable params:	0	



- Test MAE = **2.62** degrees Celsius
- No improvement!!
- Flattening a timeseries data is not a good idea!

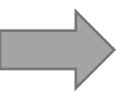


# Let's try CNN (Convolutional Neural Networks)

- Motivation: Maybe a temporal convnet could reuse the same representations across different days, much like a spatial convnet can reuse the same representations across different locations in an image!

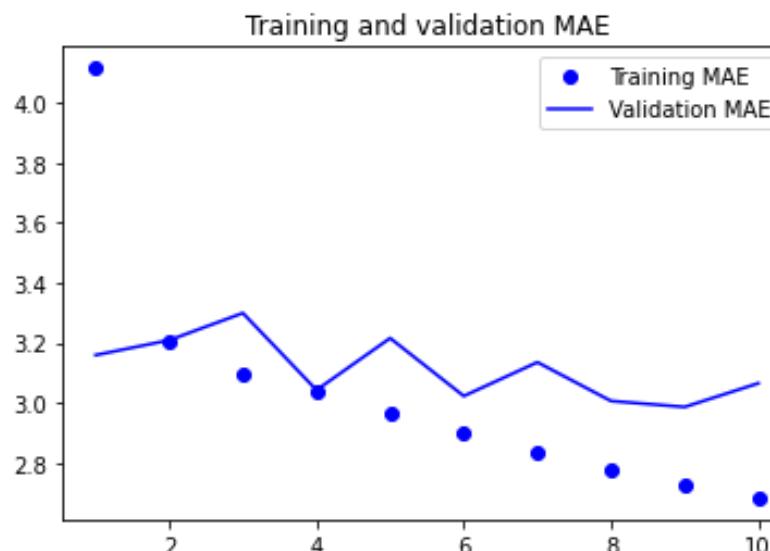
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

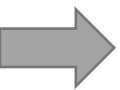
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 120, 14)]	0
conv1d (Conv1D)	(None, 97, 8)	2696 $(4 \times 24 \times 8) + 8 = 2696$
max_pooling1d (MaxPooling1D)	(None, 48, 8)	0
conv1d_1 (Conv1D)	(None, 37, 8)	776
max_pooling1d_1 (MaxPooling1D)	(None, 18, 8)	0
conv1d_2 (Conv1D)	(None, 13, 8)	392
global_average_pooling1d (GlobalAveragePooling1D)	(None, 8)	0
dense_2 (Dense)	(None, 1)	9
<hr/>		
Total params: 3,873		
Trainable params: 3,873		
Non-trainable params: 0		



# CNN performance

- Test MAE = **3.10** degrees Celsius
- Even worse than the densely connected model!! !
  - CNN treats every segment of the data the same way!
  - Pooling layers are destroying order information.
  - *Translation invariance Assumption doesn't Work here. e.g: 5pm - 7pm  $\neq$  7pm - 9pm.*

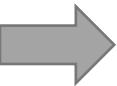




# Sequence Modeling

To model sequence data efficiently, we need a new architecture that:

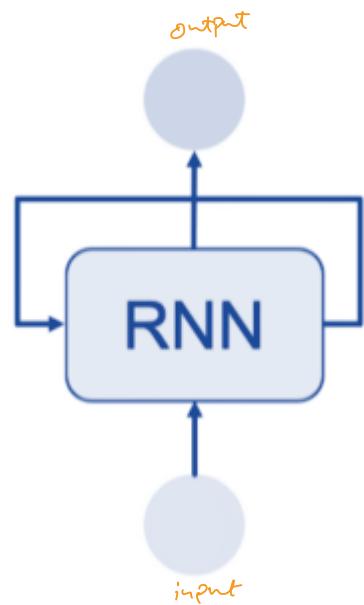
- Preserve the **order** (*For TimeSeries it's a must*)
- Account for **long-term dependencies**
- Handle **input-length** (*Variable input lengths*)
- Share parameters across the sequence

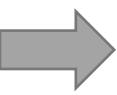


# What is RNN (Recurrent Neural Network)?

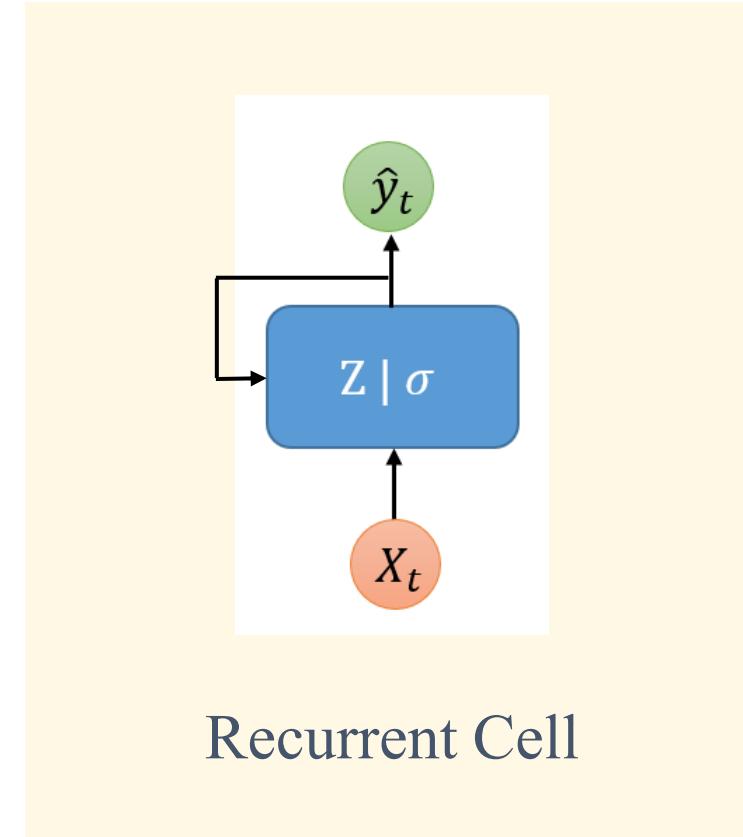
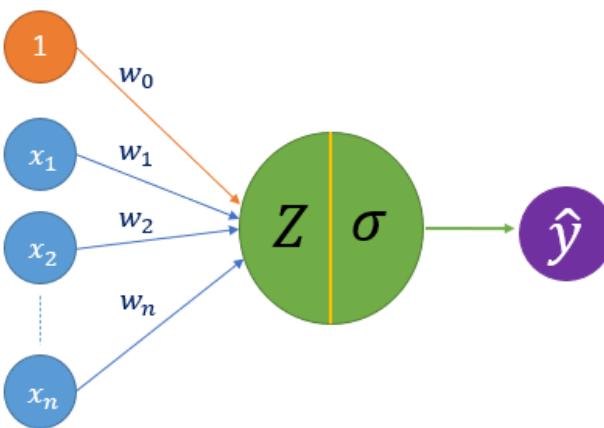
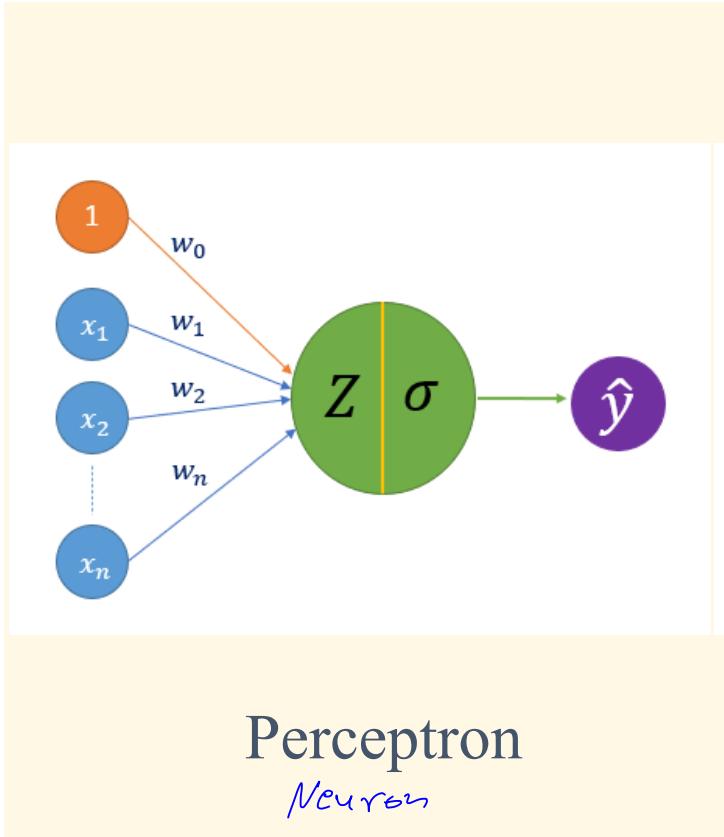
RNN preserves order, handles variable input length & can also share parameters across the sequence but can't do long-term dependencies.

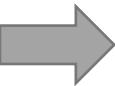
- The architecture of RNNs is inspired by the way **biological intelligence** processes information **incrementally** while maintaining an internal model of what it is processing.
- This ability to **remember previous inputs** and incorporate them into the current output allows RNNs to model sequential data.
- RNN maintains a **state** that contains information relative to what it has seen so far
- RNNs can be thought of as neural networks with an **internal loop**, which allows them to process sequences of varying lengths and learn from temporal dependencies.





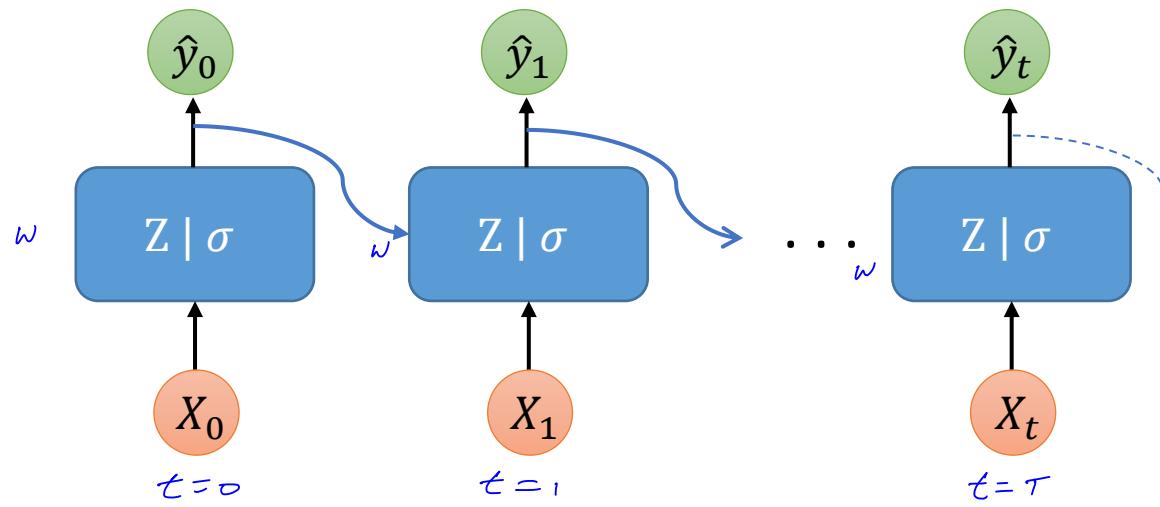
# Perceptron vs Recurrent Cell



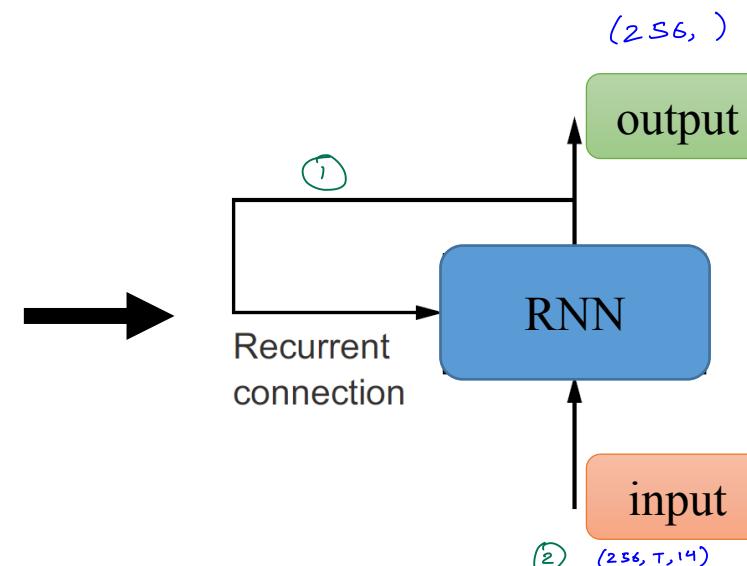


# Unrolling the Recurrent Cell

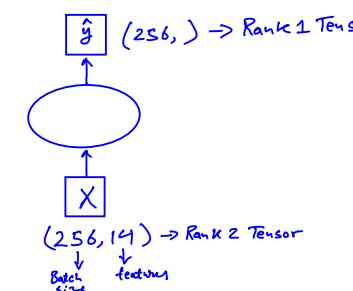
Key Differences between RNN & Neural Network  
is:  
① Loop  
② Input Tensor Rank



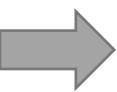
Same weights  $w$  used throughout.



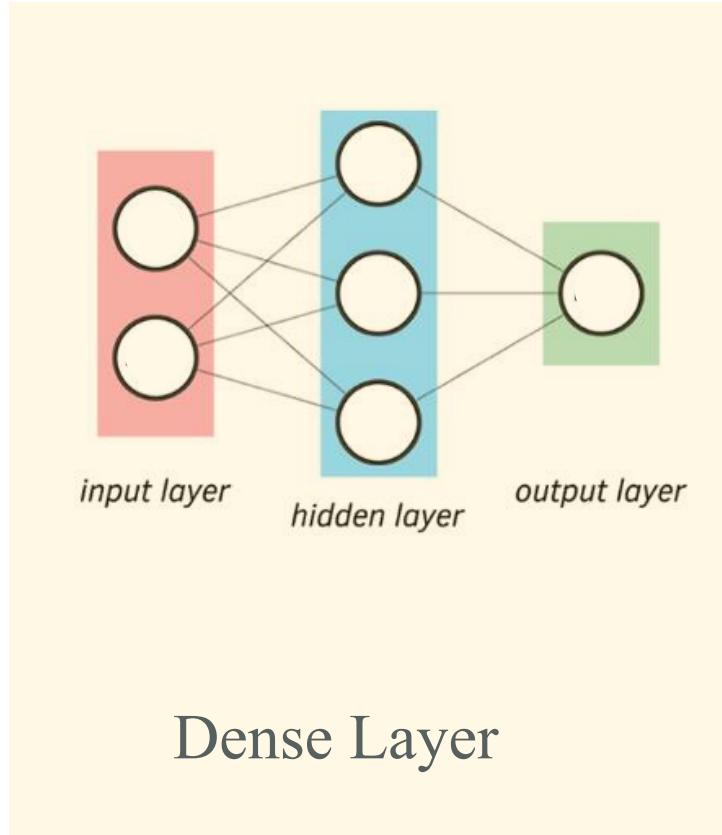
In Simple neuron we had:



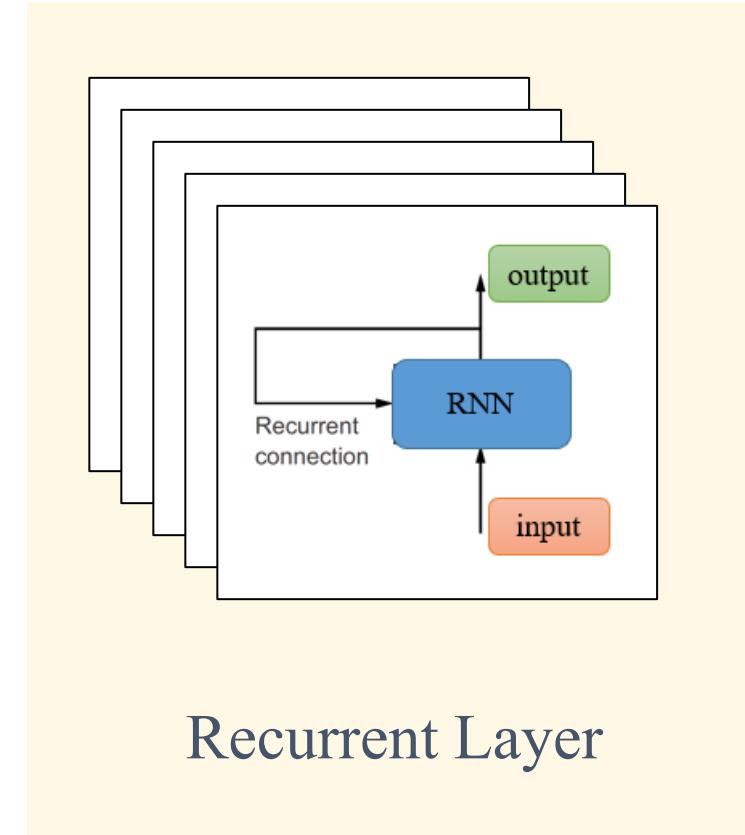
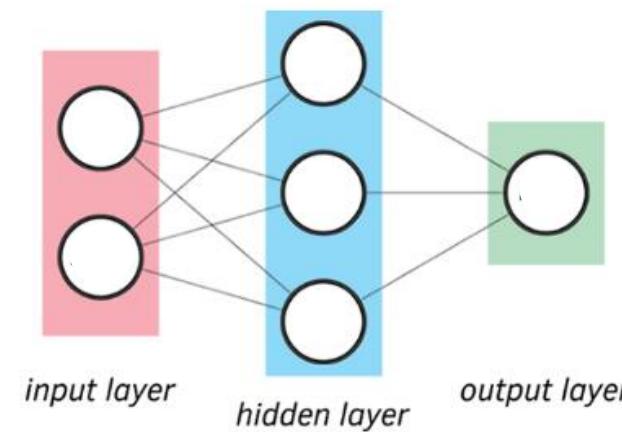
In RNN we have this extra dimension (Time Step)  
Compared to Simple Neural network.



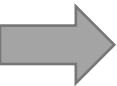
# Dense Layer vs Recurrent Layer



Dense Layer

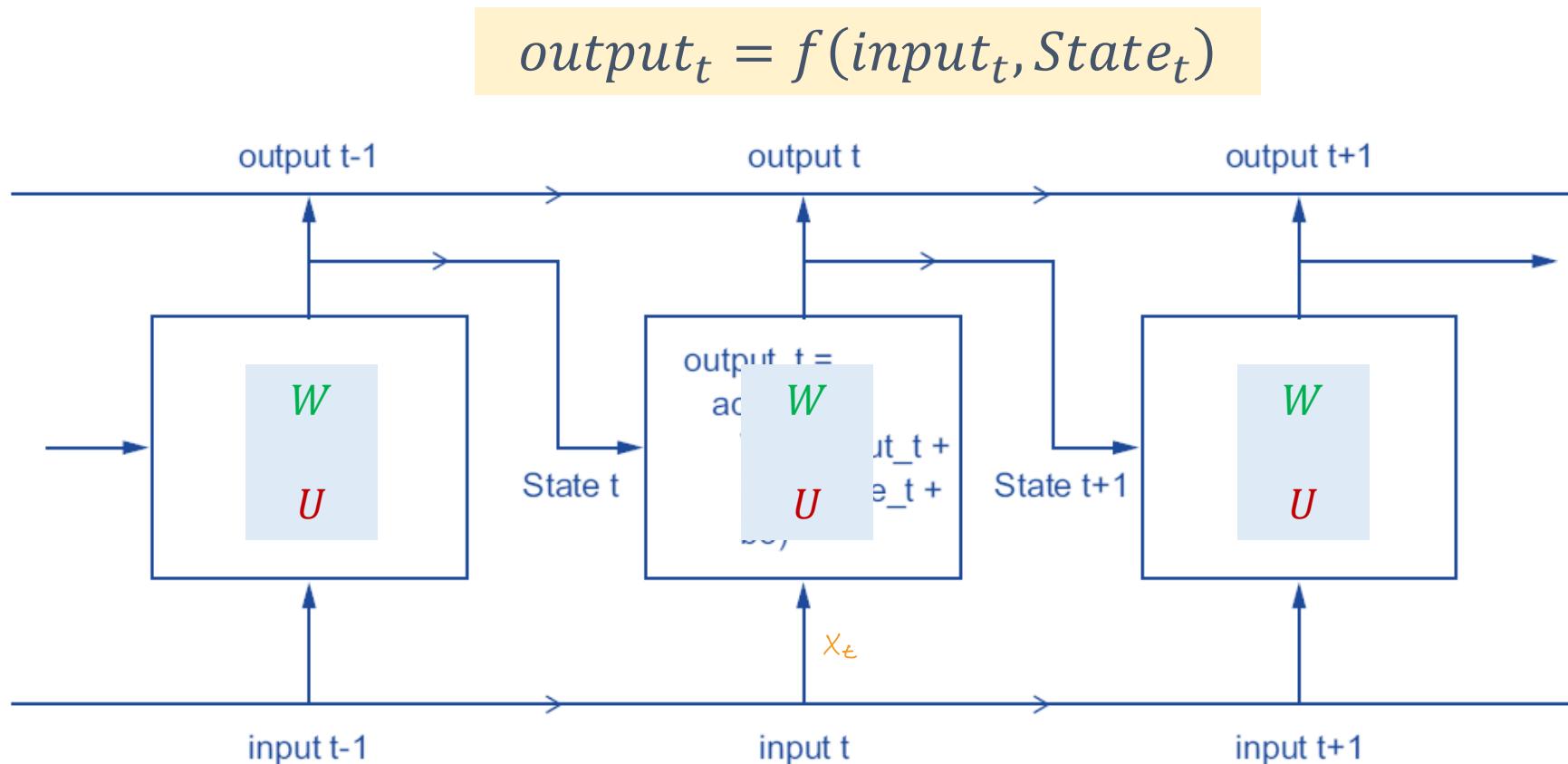


Recurrent Layer

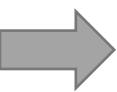


# Inside the Recurrent Cell

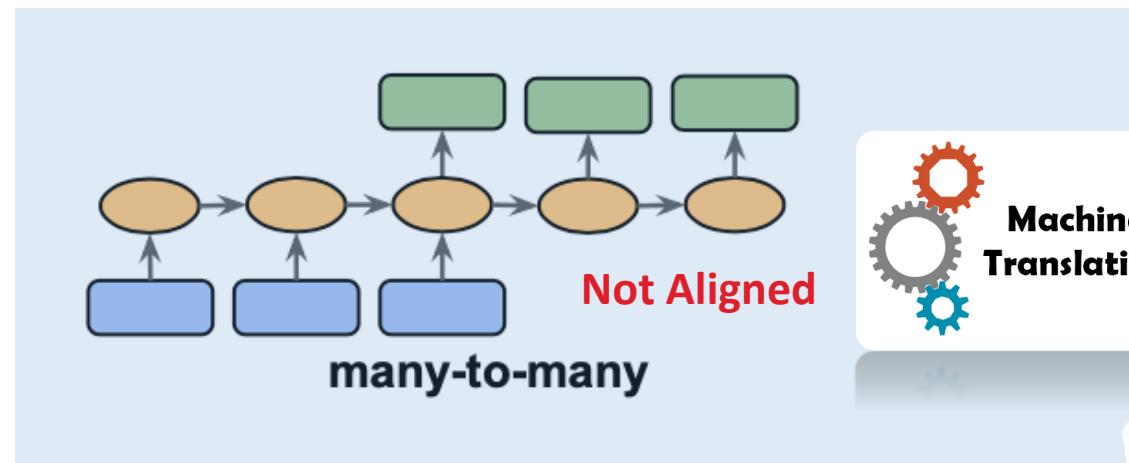
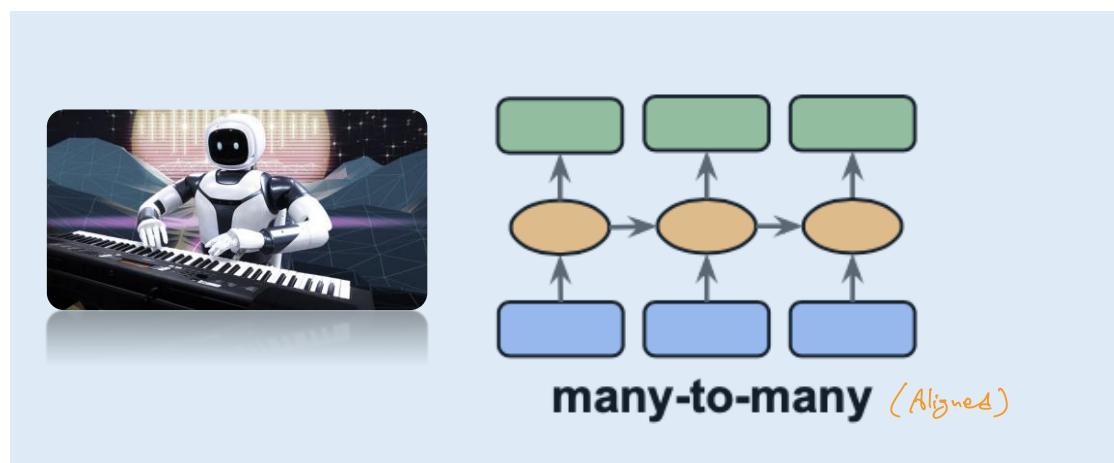
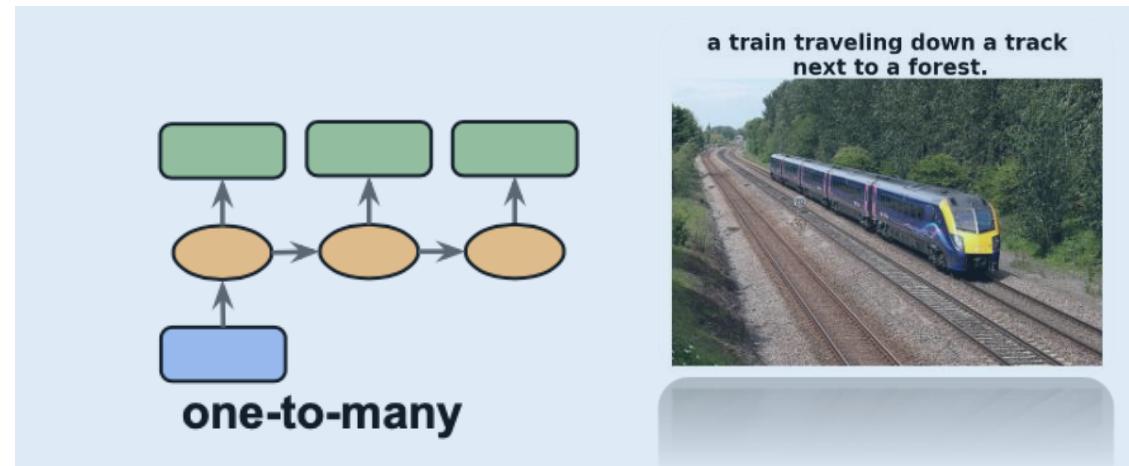
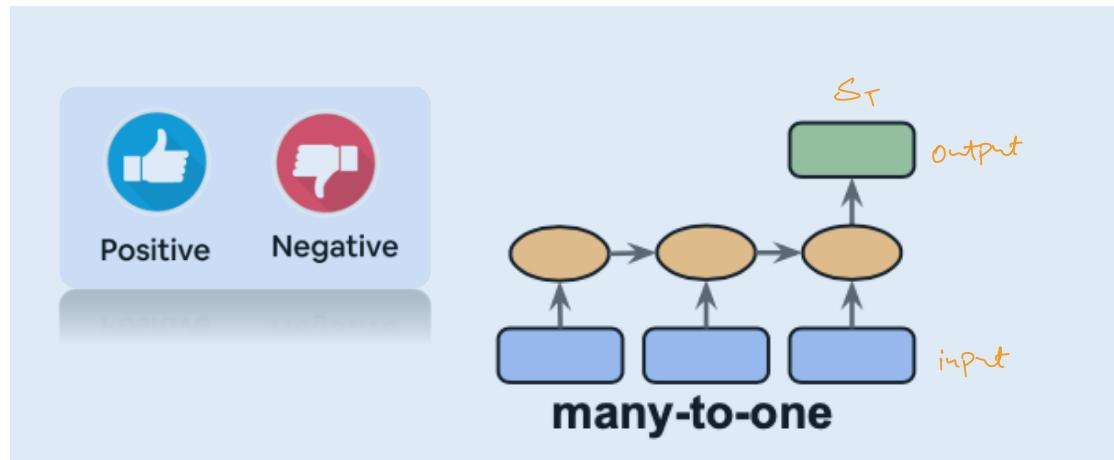
*W = Weight of input  
U = Weight for state variable*

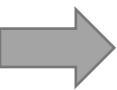


$$s_{t+1} = activation(WX_t + US_t + b)$$



# RNN architectures





# How does RNN learn representations?

- Backpropagation Through Time (BPTT)

- $\frac{\partial J}{\partial P}$  P are the parameters

- $\frac{\partial J}{\partial W} = \frac{\partial J_0}{\partial W} + \frac{\partial J_1}{\partial W} + \dots$

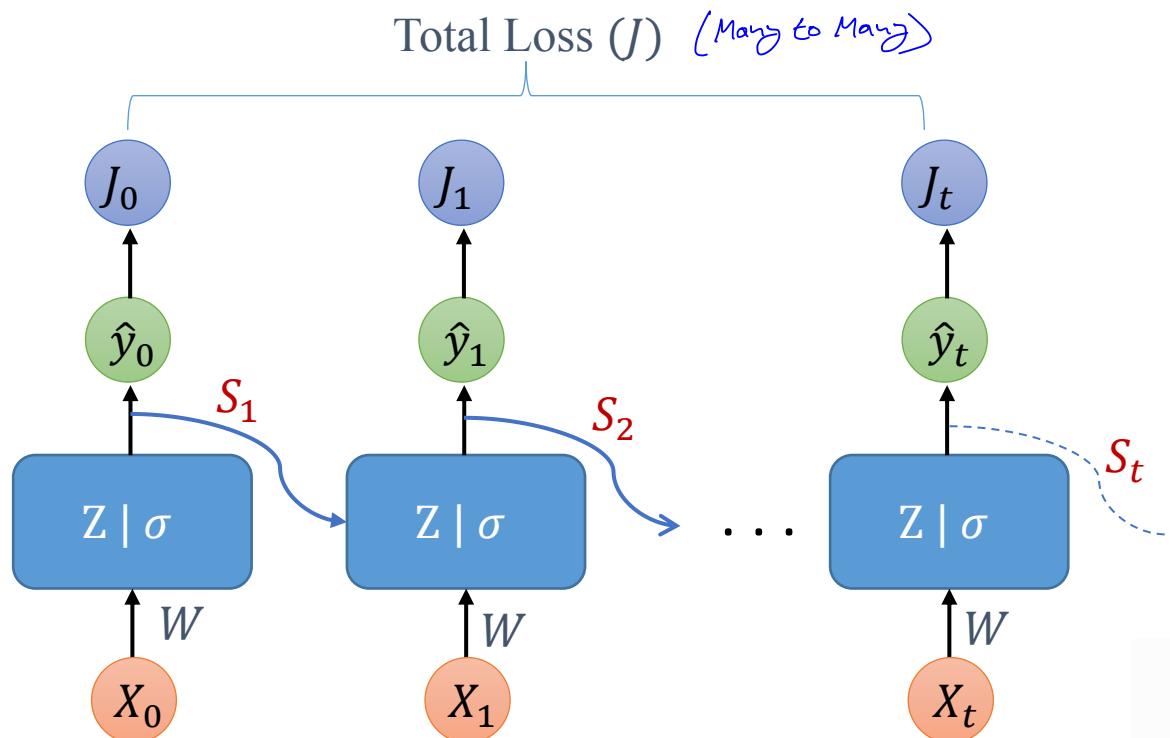
- $\frac{\partial J_0}{\partial W} = \frac{\partial J_0}{\partial y_0} \frac{\partial y_0}{\partial s_0} \frac{\partial s_0}{\partial w}$

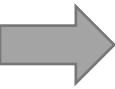
- $\frac{\partial J_1}{\partial W} = \frac{\partial J_1}{\partial y_1} \frac{\partial y_1}{\partial s_1} \frac{\partial s_1}{\partial w}$  ,  $\frac{\partial s_1}{\partial W} = \frac{\partial s_1}{\partial s_0} \frac{\partial s_0}{\partial w}$

- ...

- $\frac{\partial J_t}{\partial W} = \sum_{k=0}^t \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial w}$

$$\frac{\partial J_t}{\partial W} = \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_0} \frac{\partial s_0}{\partial w}$$



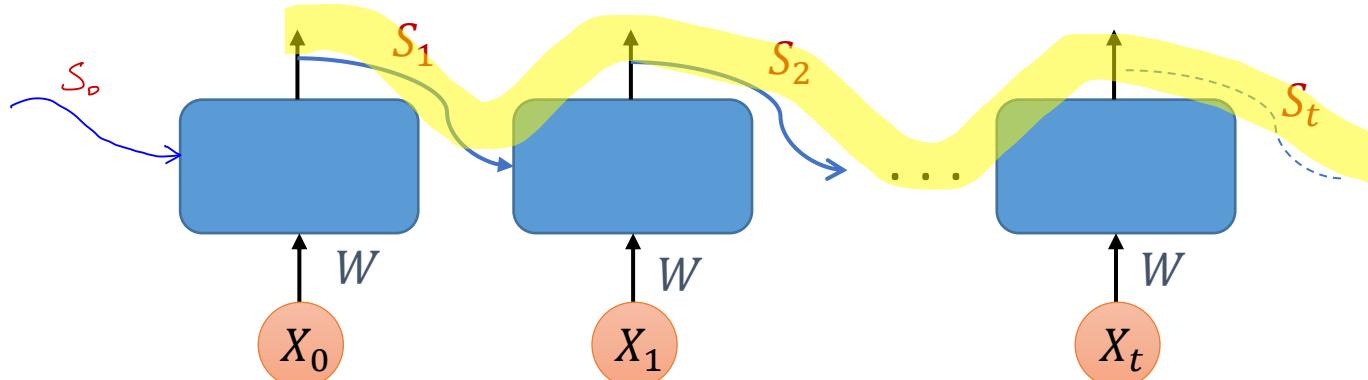


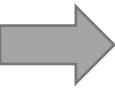
# Vanishing Gradient Problem

- As the time horizon gets bigger, this product gets longer and longer.
- We are multiplying a lot of small numbers → smaller gradients → biased parameters unable to capture long term dependencies.

$$\begin{aligned}\frac{\partial J_t}{\partial W} &= \sum_{k=0}^t \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W} \\ \frac{\partial s_{10}}{\partial s_0} &= \frac{\partial s_{10}}{\partial s_9} \frac{\partial s_9}{\partial s_8} \frac{\partial s_8}{\partial s_7} \frac{\partial s_7}{\partial s_6} \dots \frac{\partial s_1}{\partial s_0}\end{aligned}$$

$$s_t = \text{activation}(Wx_{t-1} + Us_{t-1})$$

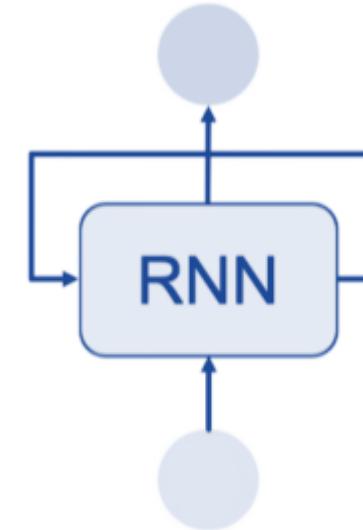




# Beyond RNN

RNN can handle the following sequence modeling criteria:

- Preserve the **order**
- Handle **input-length**
- Share parameters across the sequence



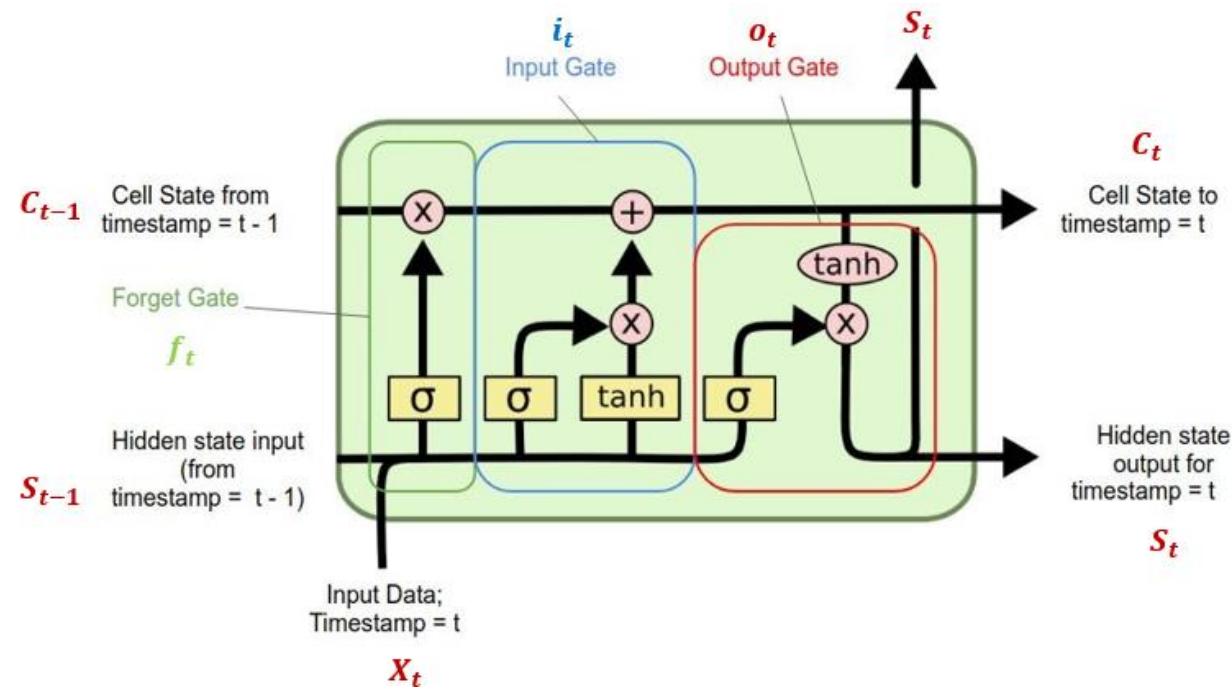
RNN limitations:

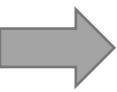
- Does not account for **long-term dependencies** (only remember short term history )
- Vanishing Gradient Problem

# Module 6 – Part II

## Deep Sequence Modeling

### (Gated cells, LSTM)

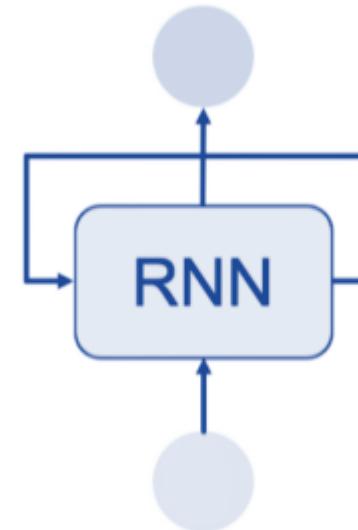




# Beyond RNN

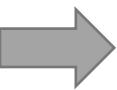
RNN can handle the following sequence modeling criteria:

- Preserve the **order**
- Handle **input-length**
- Share parameters across the sequence



RNN limitations:

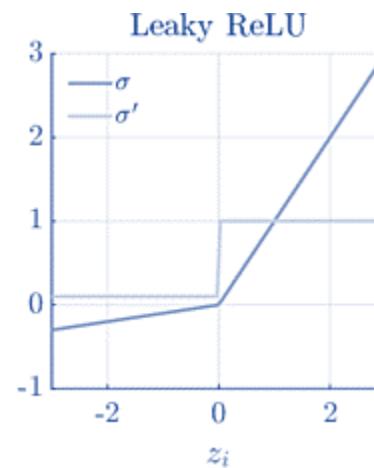
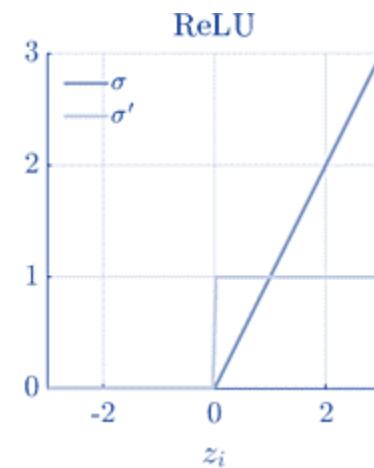
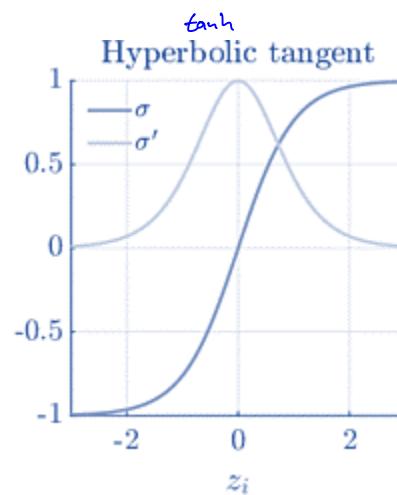
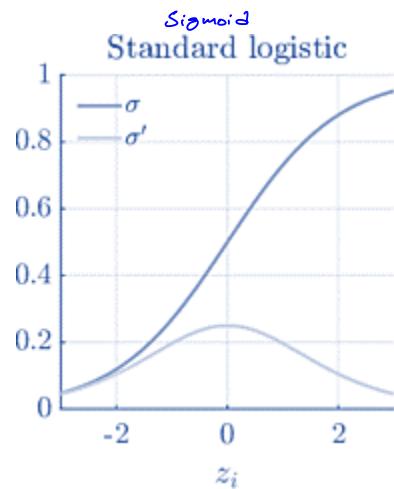
- Does not account for **long-term dependencies** (only remember short term history )
- Vanishing Gradient Problem



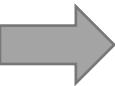
# How to solve vanishing gradient problem

1. Use **Activation Function** that prevents fast shrinkage of gradient *Like ReLU or Leaky ReLU.*

$\sigma'$  = Gradients



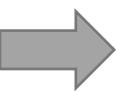
$$S_t = \text{activation}(W X_{t-1} + U s_{t-1})$$



# How to solve vanishing gradient problem

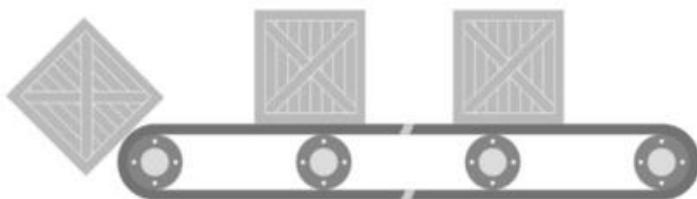
*(Can be applied to any Deep Learning Networks like DNN, CNN, RNN etc.)*

- 1. Use Activation Function that prevents fast shrinkage of gradient
- 2. Use **weight initialization** techniques that ensure that the initial weights are not too small
- 3. Use **gradient clipping** which limits the magnitude of the gradients from becoming too small (vanishing gradient) or too large (exploding gradient)
- 4. Use **batch normalization**, which normalizes the input to each layer and helps to reduce the range of activation values and thus the likelihood of vanishing gradients.
- 5. Use a different **optimization algorithm** that is more resilient to vanishing gradients, such as Adam or RMSprop.
- 6. **Gated cells:** Use some sort of **skip connections**, which allow gradients to bypass some of the layers in the network and thus prevent them from becoming too small.



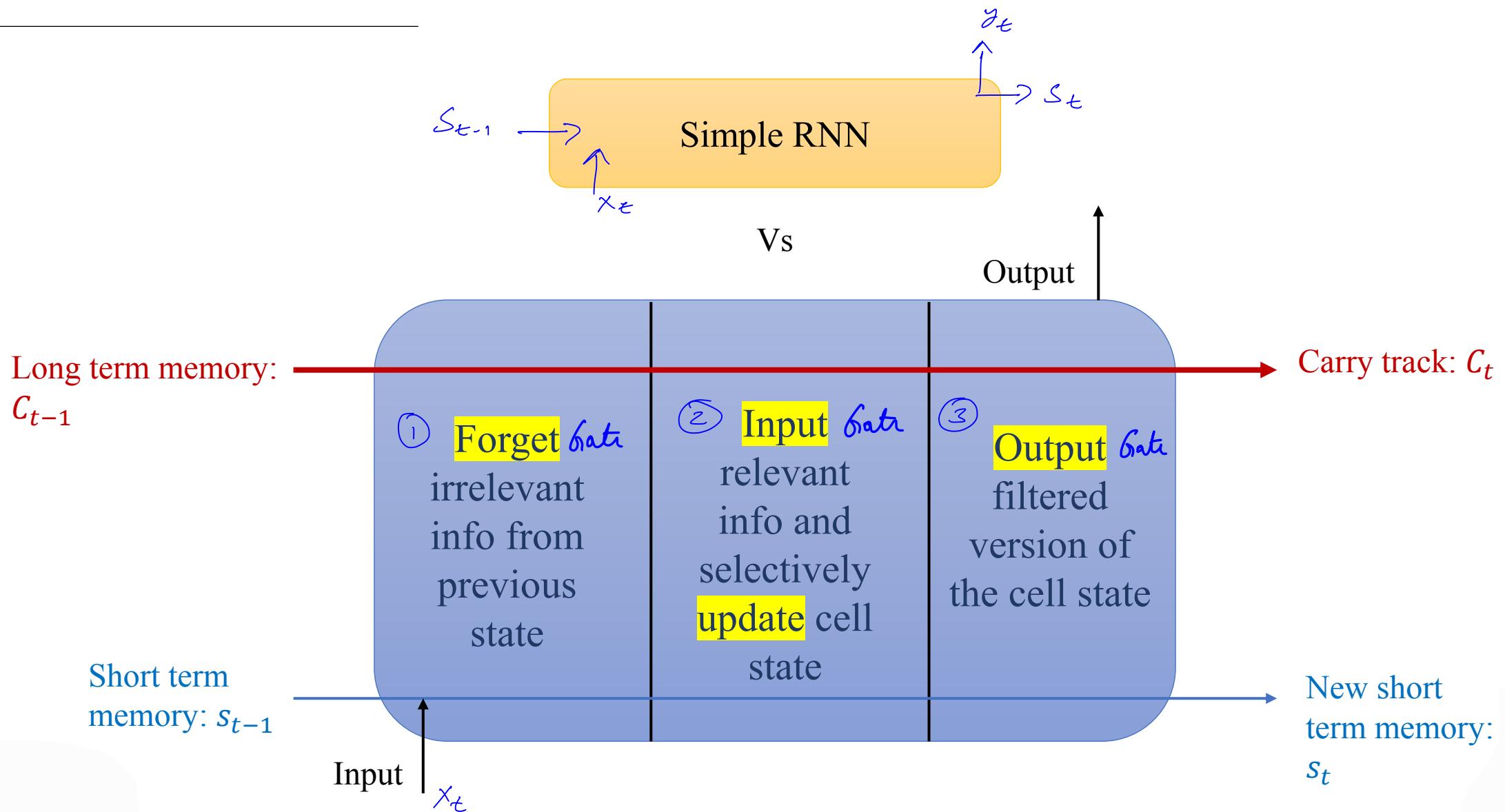
# Gated cells

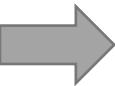
- Instead of using a simple RNN cell, let's use a **more complex cell** with gates which **control the flow of information**.
- Think of a **conveyer belt** running parallel to the sequence being processed:
  - Information can jump on → transported to a later timestep → jump off when needed.
  - This is what a gated cell does! Analogous to **residual connections** we saw before.



- Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two examples of gated cells that can **keep track of information throughout many timesteps**.

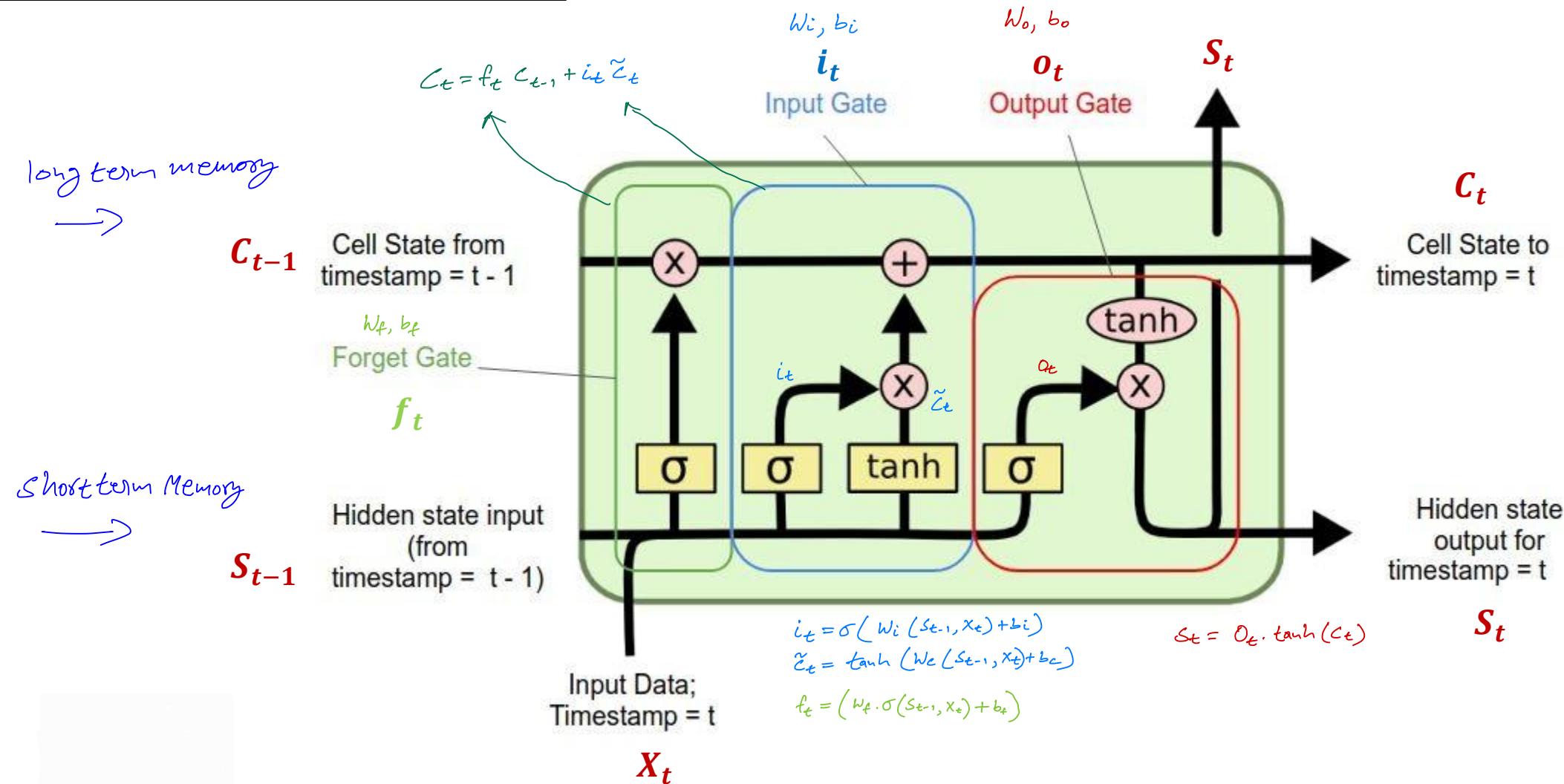
# Inside the LSTM cell

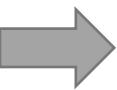




# LSTM details

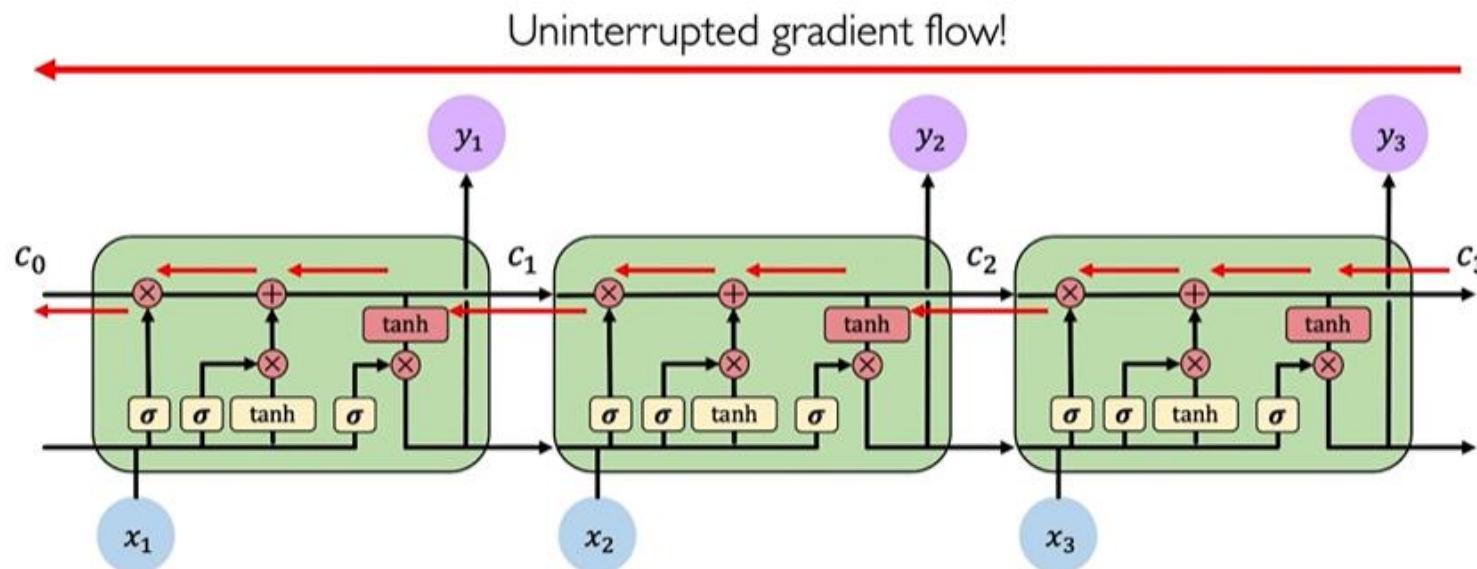
Each Gates have their own weights & bias terms.

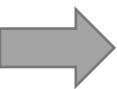




# LSTM takeaway

- LSTM uses gates to regulate the information flow (allows past information to be reinjected later)
- This new cell state (carry) can better capture longer term dependencies
- LSTM fights the vanishing gradient problem



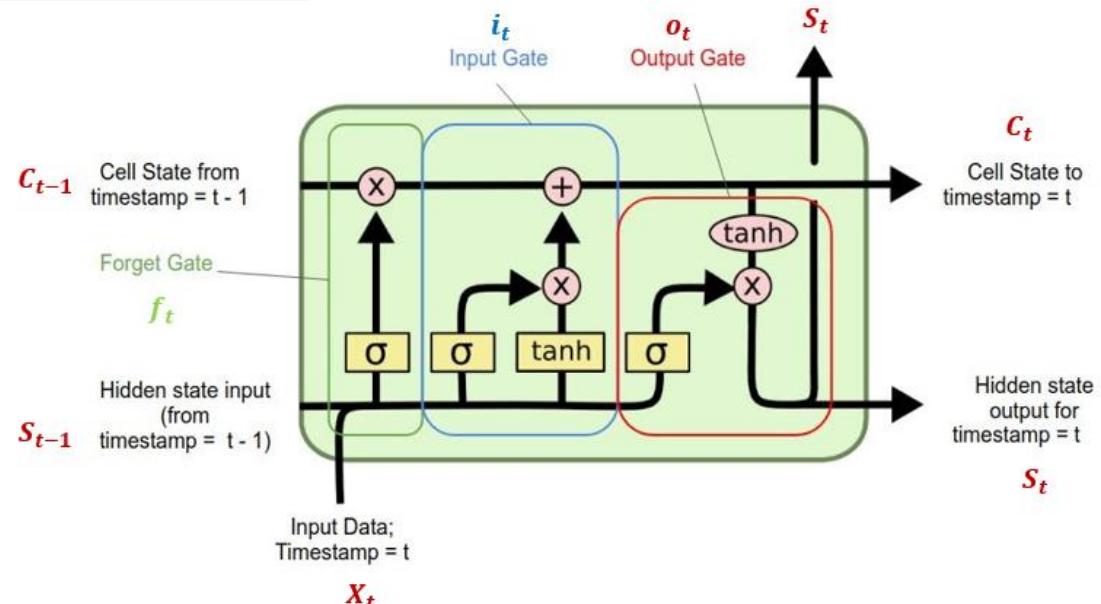


# Let's try LSTM on the temperature example

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))  
x = layers.LSTM(16)(inputs)  
outputs = layers.Dense(1)(x)  
model = keras.Model(inputs, outputs)
```

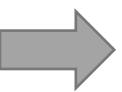
Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[None, 120, 14]	0
lstm (LSTM)	(None, 16)	1984
dense_3 (Dense)	(None, 1)	17

Total params: 2,001  
Trainable params: 2,001  
Non-trainable params: 0



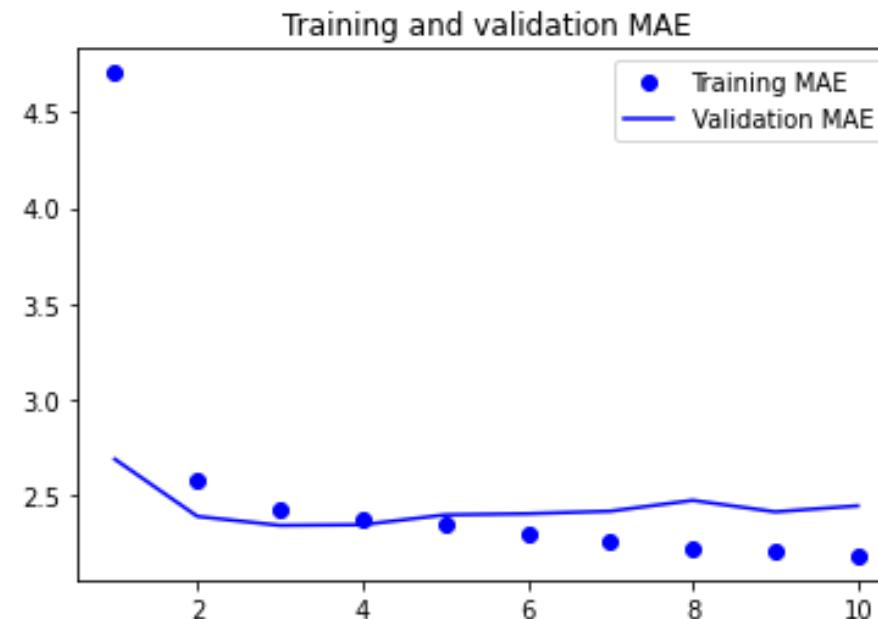
```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)  
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)  
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)  
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

$$(4 \times 16) \times 4 + (16 \times 16) \times 4 + 4 \times 16 = 1984$$

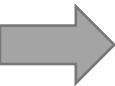


# LSTM performance

- Baseline Test MAE = 2.62
- Simple LSTM Test MAE = **2.53**
- Finally beat the naïve forecaster.
- Overfitting?



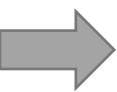
Can we do better?



# Improving the simple LSTM model

- We can improve the performance of the simple LSTM model by:
  1. **Recurrent Dropout** : use drop out to fight overfitting in the recurrent layers (in addition to drop out for the dense layers)
  2. **Stacking recurrent layers**: increase model complexity to boost representation power
  3. **Using bidirectional RNN**: processing the same information differently! Mostly used in NLP.





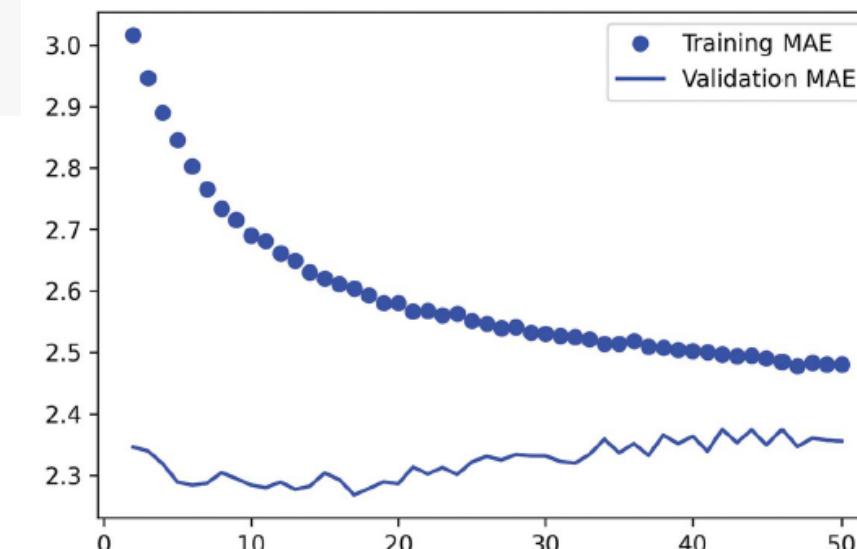
# Recurrent Drop out

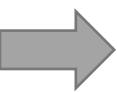
- The same dropout pattern should be applied at every timestep

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

ordinary  
dropout

- Baseline Test MAE = 2.62
- Simple LSTM, Test MAE = 2.53
- LSTM with dropout, Test MAE = 2.45



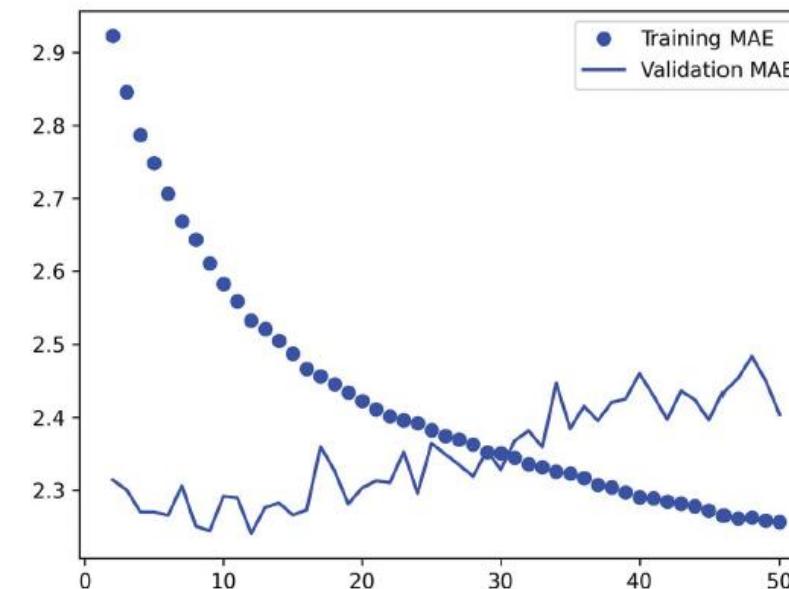


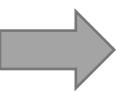
# Stacking Recurrent Layers

- Let's train a dropout-regulated, stacked GRU model.
- GRU is a slightly simpler version (hence, faster) of LSTM architecture

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- Baseline Test MAE = **2.62**
- Simple LSTM, Test MAE = **2.53**
- Stacking GRU, Test MAE = **2.39**

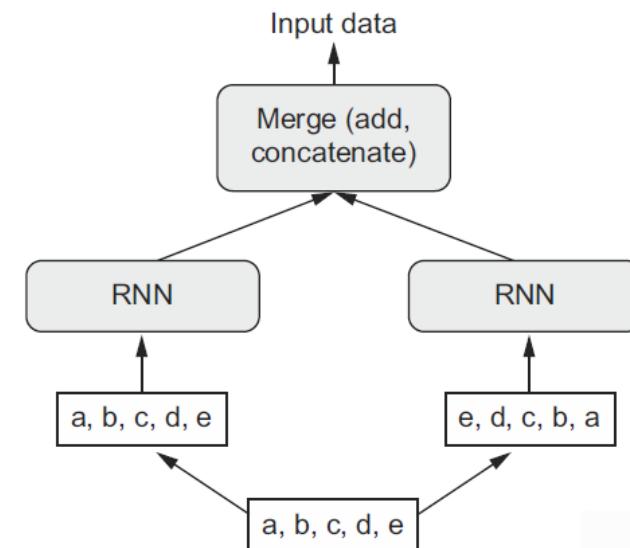


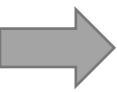


# Bidirectional RNN

- Bidirectional RNN process the input sequence both chronologically and antichronologically.
- Idea: capturing patterns (representations) that might be overlooked by a unidirectional RNN.
- For the temperature example, the bidirectional LSTM strongly underperforms even the common-sense baseline.

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```





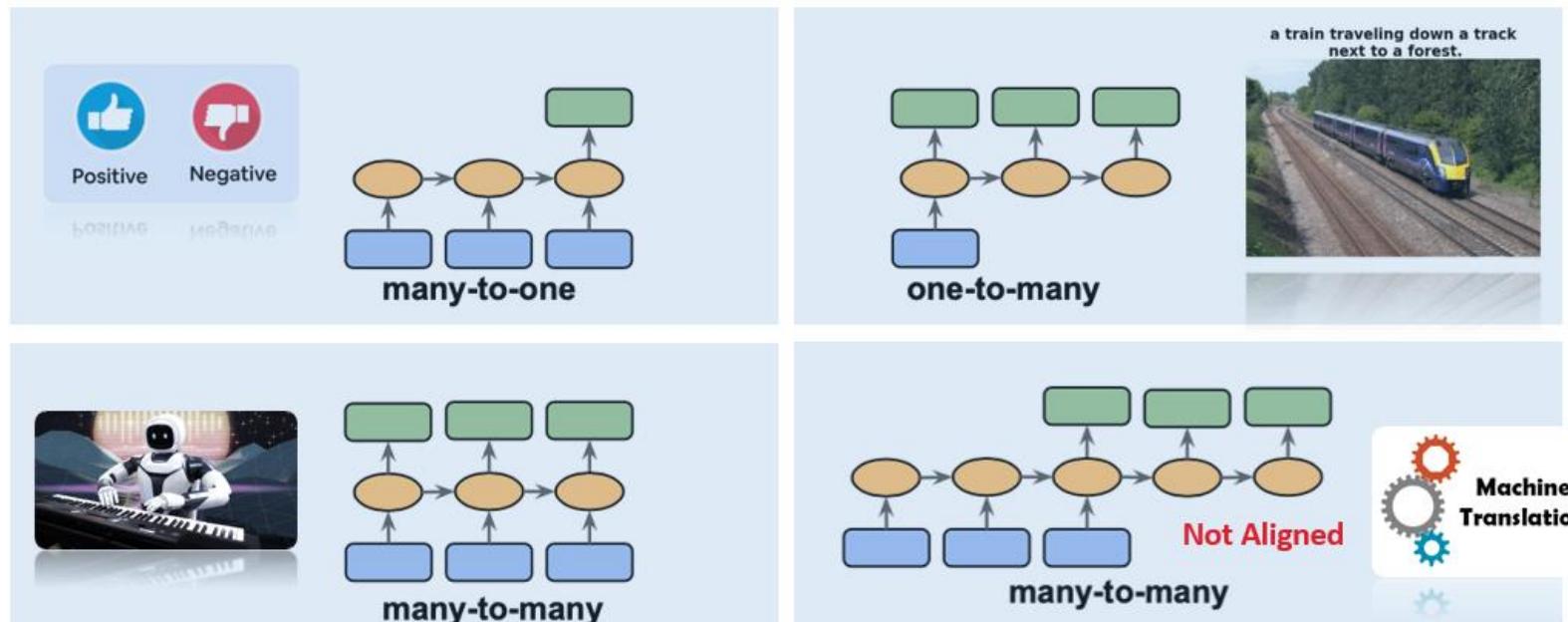
# Final message

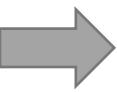
- Deep learning is more an art than science! Too many moving part!
  - Number of units in each recurrent layer
  - Number of stacked layers
  - Amount of dropout and recurrent dropout
  - Number of dense layers
  - Sequence horizon!
  - Optimizers, learning rates and etc
  - ....
- Apply RNN to datasets that past is a good predictor of the future! **Not the stock market!**



# Module 6 – Part III

## Deep Sequence Modeling for Text Natural Language Processing (NLP)

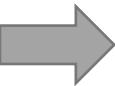




# Road map!

- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- **Module 6- Deep Sequence Modeling (RNN, LSTM, NLP)**
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

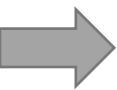




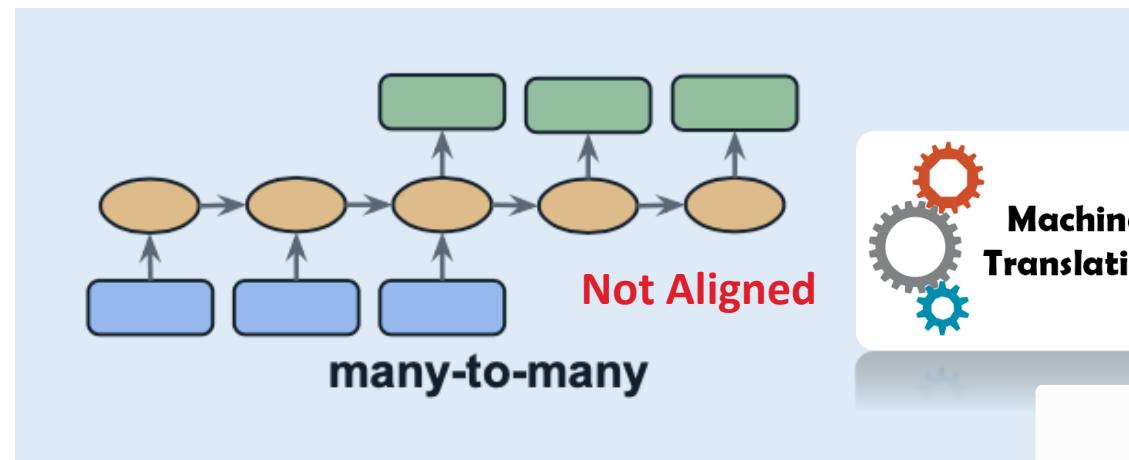
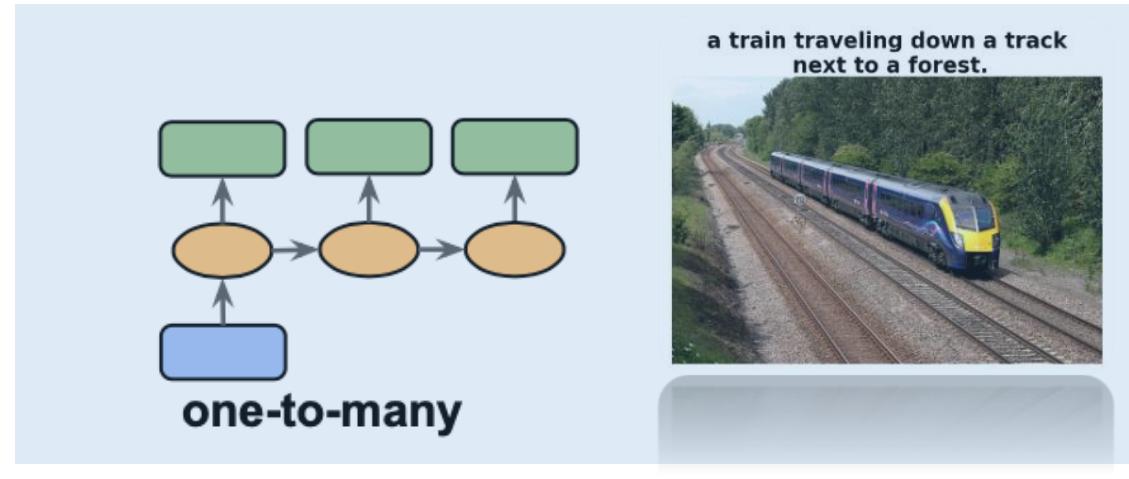
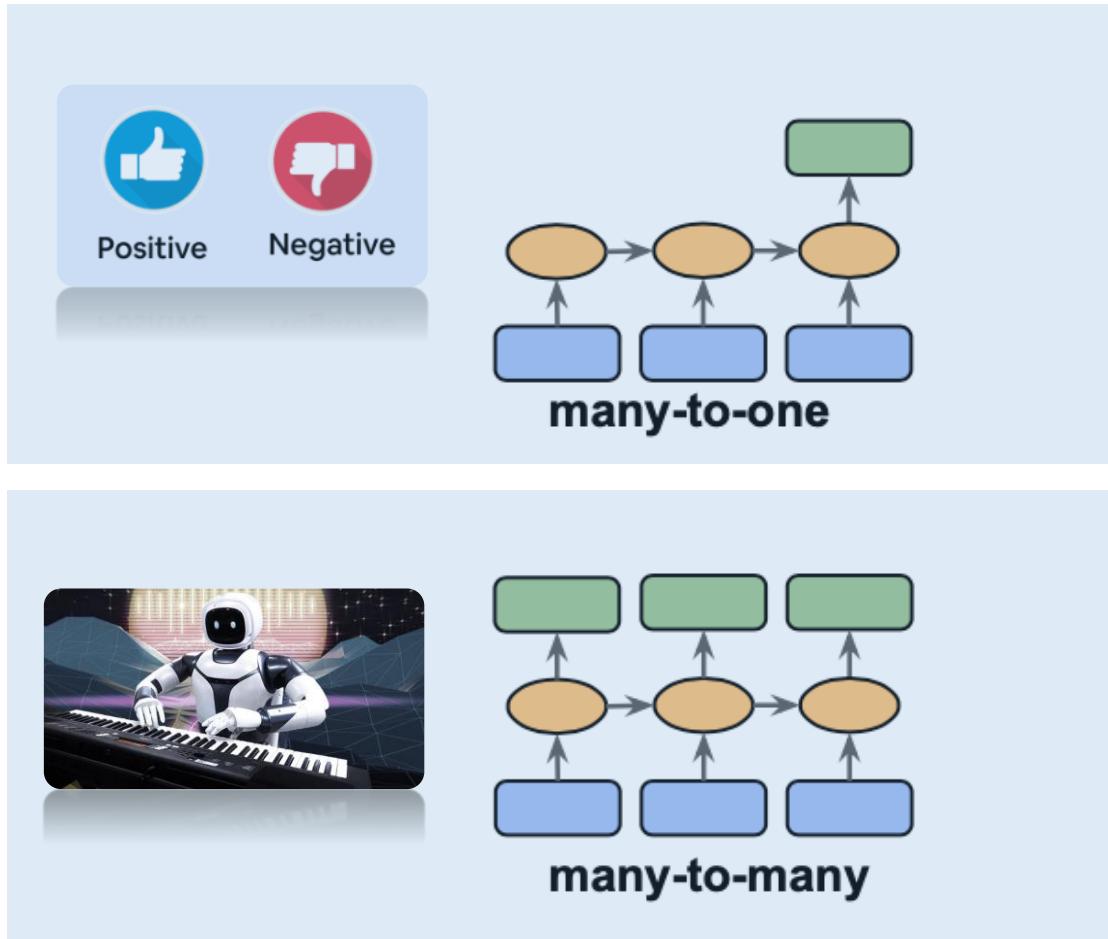
# Different kinds of sequence data

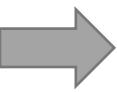
Sequence data refers to any data that has a specific order or sequence to it!

- **Time series data:** Time series data is a sequence of data points that are measured at regular intervals over time. (stock prices, weather patterns, medical records, ...)
- **Text data:** Text data refers to any type of data that is composed of words, sentences, or paragraphs. (tweets, news articles, product reviews, ...)
- **Audio data:** Audio data refers to any type of data that is recorded or generated as sound waves. (speech recordings, music tracks, ...)
- **Video data:** Video data refers to any type of data that is represented as a sequence of images or frames. (movie clips, surveillance footage, ...)



# RNN architectures



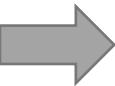


# Human language vs machine language

- In computer science: **Human language** → **Natural language** (shaped by an evolution process)
- **Machine language** designed for machines (XML, Assembly, ...)



Factors	Human Language	Machine Language
<b>Complexity</b>	Highly complex	Structured and logical
<b>Creativity</b>	Can be highly creative	Limited creativity
<b>Ambiguity</b>	High levels of ambiguity	Little to no ambiguity
<b>Learning</b>	Acquired naturally	Taught through programming
<b>Adaptability</b>	Highly adaptable	Fixed and rigid

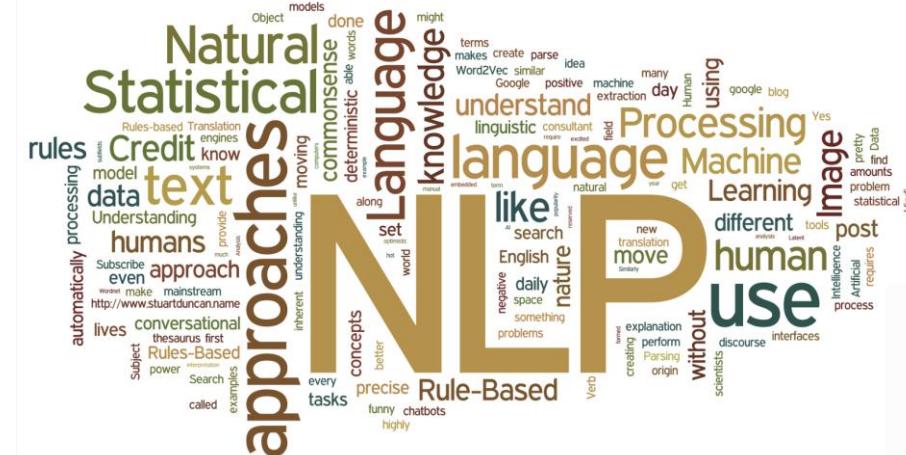


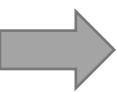
# Natural Language Processing

- Making sense of human language through algorithms is a big deal!
- NLP is a subfield of computer science and AI that deals with the interaction between computers and human language
- Traditional NLP started with finding **handcrafted rules** to **understand** human language (1960-1990)
- Modern NLP is all about using machine learning to automate the search for these rules and enabling computers to do the followings:
  - Sentiment analysis
  - Machine translation
  - Content filtering
  - Text classification and ...

# → NLP, the goal and toolset

- Goal: automated **Pattern recognition** → statistical regularities
  - Computer vision is **pattern recognition** applied to **pixels**,
  - NLP is **pattern recognition** applied to **text** (words, sentences, and paragraphs)
  - NLP toolset:
    - 1990-2010: ML models like **decision trees** and **logistic regression**
    - 2010-2017: DL models like **RNN** and **LSTM**
    - 2018-present: **Transformers**





# Preparing text data

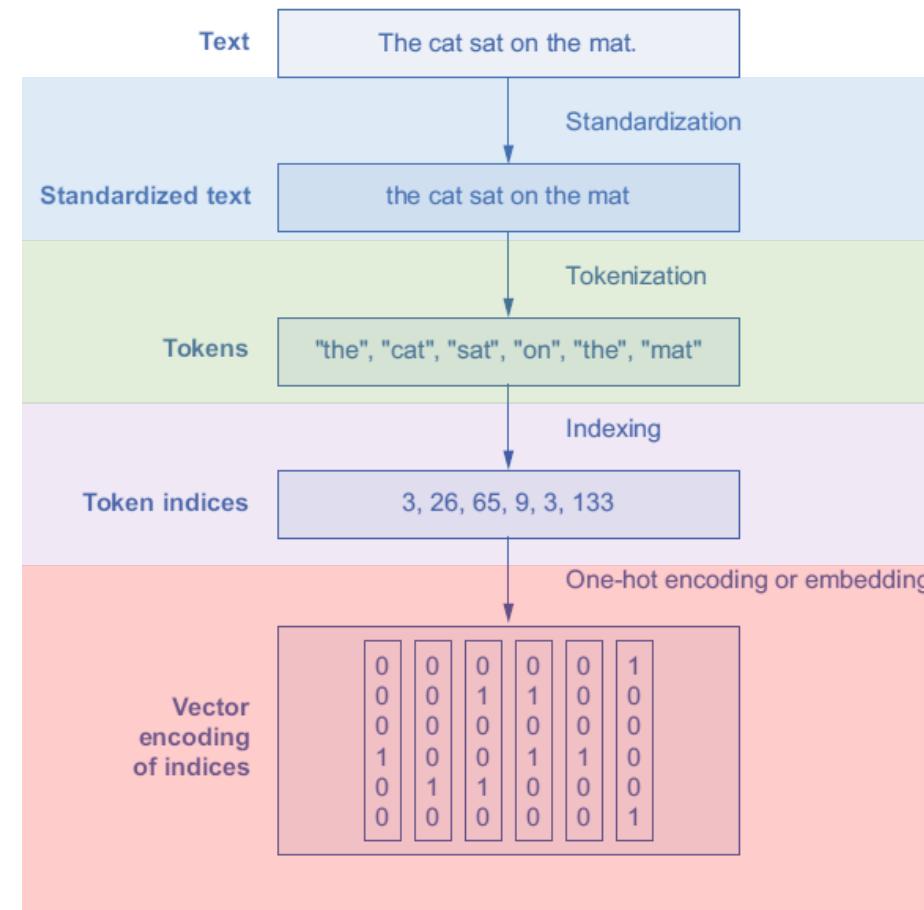
- **TextVectorization:** the process of transforming text into numeric values.

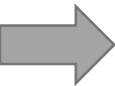
## 1. Standardization

## 2. Tokenization

## 3. Indexing

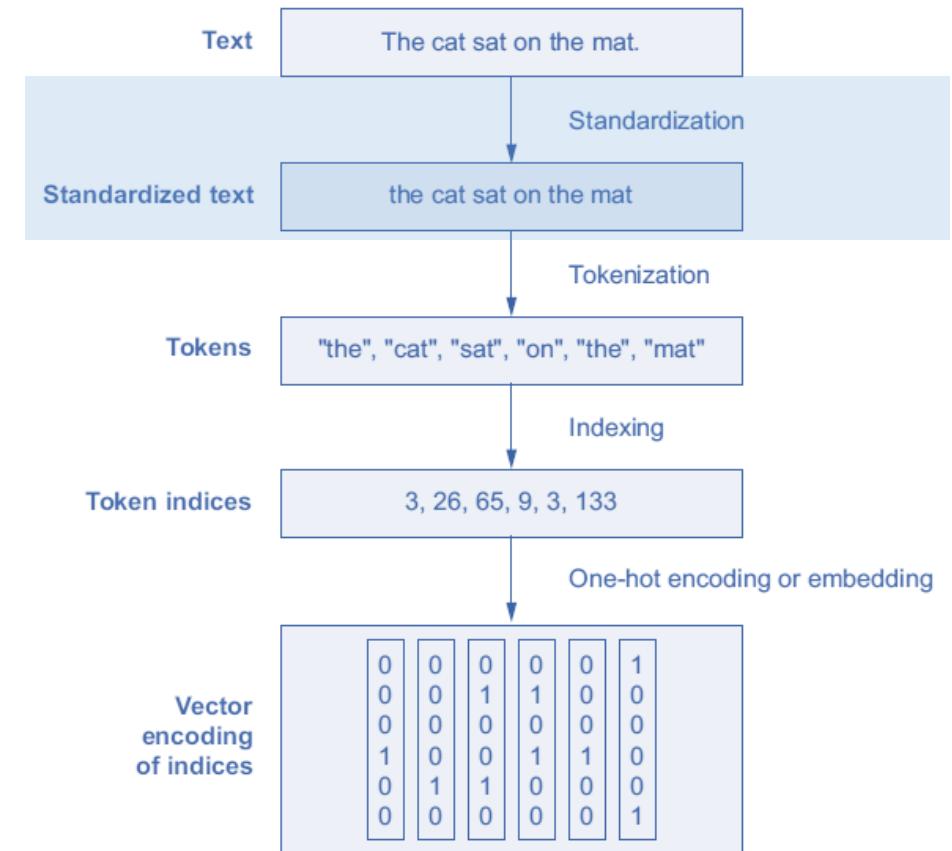
## 4. Encoding

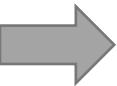




# TextVectorization: standardization, tokenization, indexing, encoding

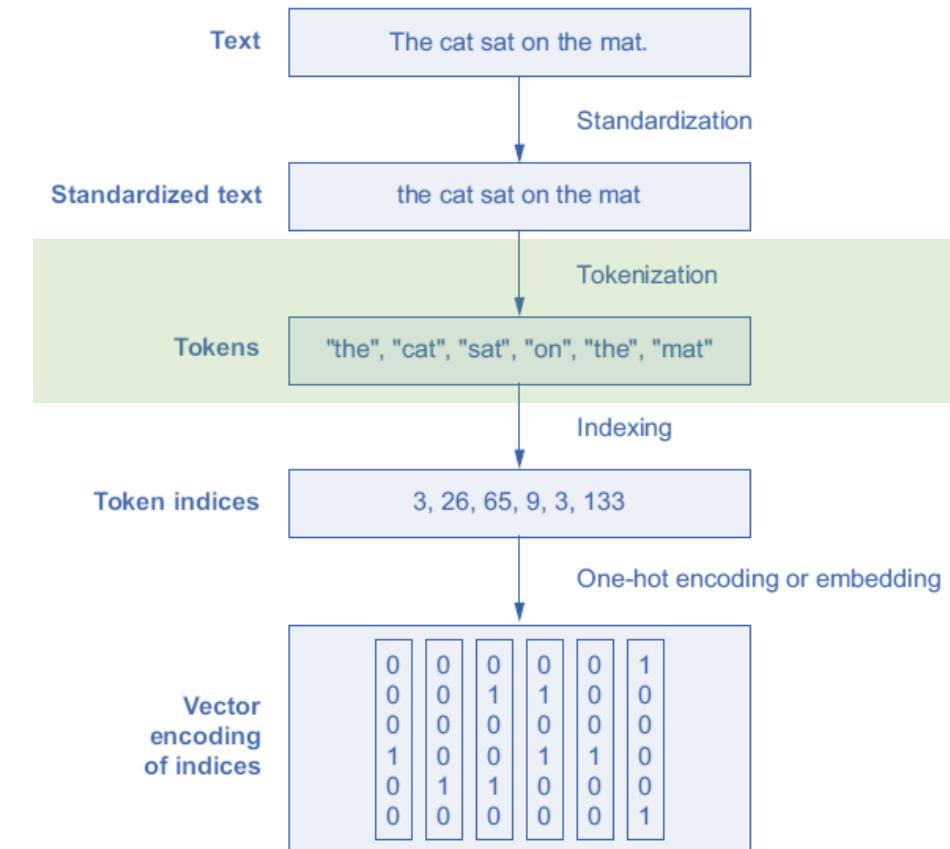
- Standardization:
  - Converting to *lowercase*,
  - Removing *punctuations*,
  - Converting *special characters*,
  - **Stemming** (converting variations of a term into a single shared representation)
- Standardization is a basic form of **feature engineering**
- With standardization, your model requires **less training data** and **generalize better**.

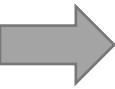




# TextVectorization: standardization, **tokenization**, indexing, encoding

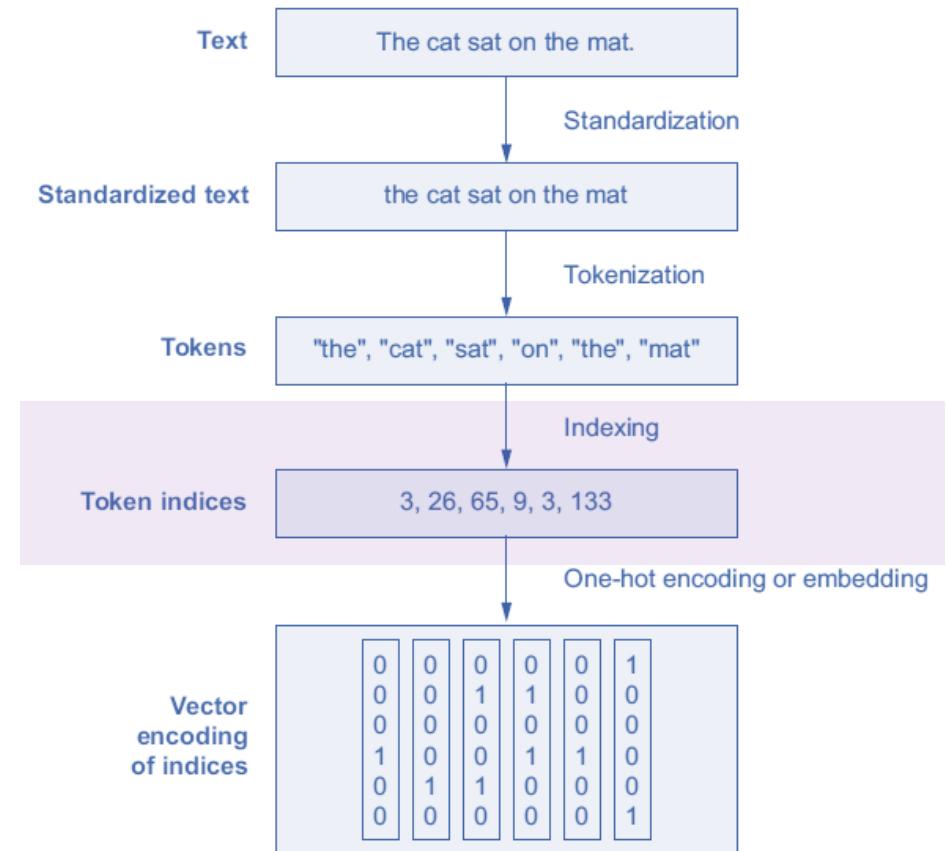
- Tokenization: Splitting text into units/token
  - *Word-level* tokenization: space-separated tokens
  - *N-gram* tokenization: groups of **N (or fewer)** consecutive words
  - *Character-level* tokenization: each character is a token
- Two kinds of text processing models:
  - Bag-of-words models (discarding original order)  
→ N-gram tokenization
  - Sequence models (word order matters)  
→ word-level tokenization

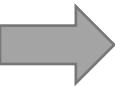




## TextVectorization: standardization, tokenization, **indexing**, encoding

- Indexing:
    - *Integer*: return sequences of words encoded as integer indices
    - Restrict to top 20k or 30k most common words
    - OOV token (out of vocabulary) index 1
    - Mask token index 0 (ignore it, used for padding)

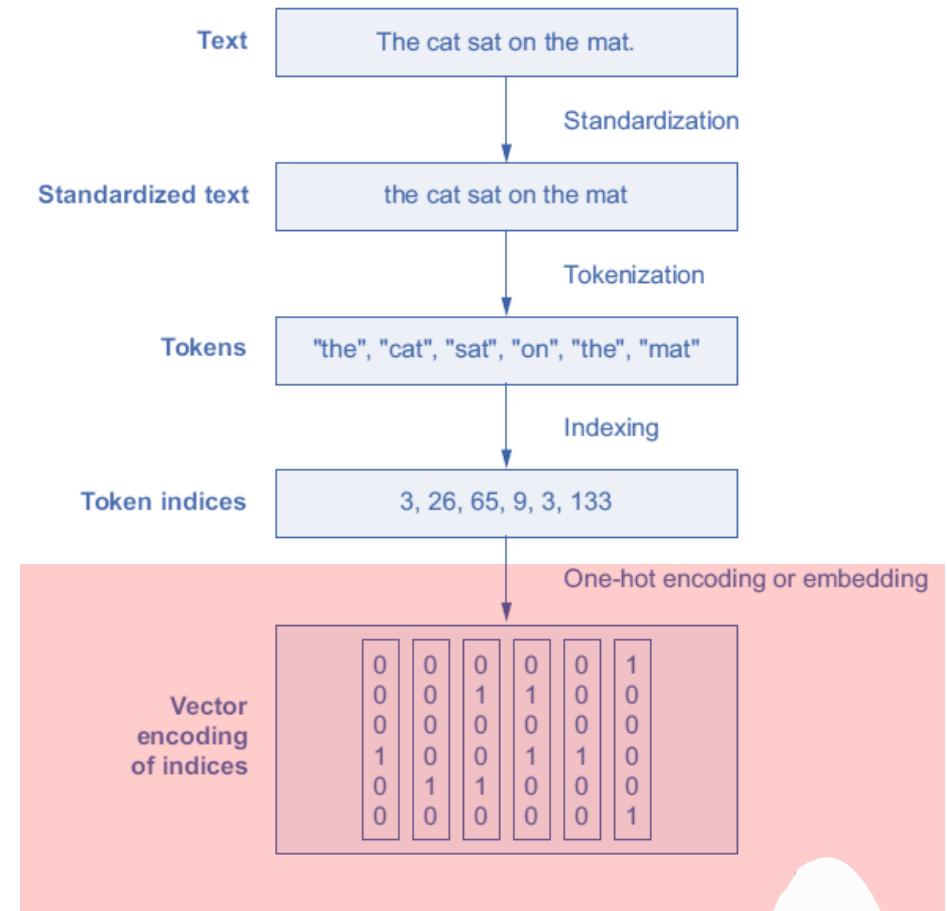


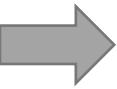


# TextVectorization: standardization, tokenization, indexing, **encoding**

- **Encoding:**

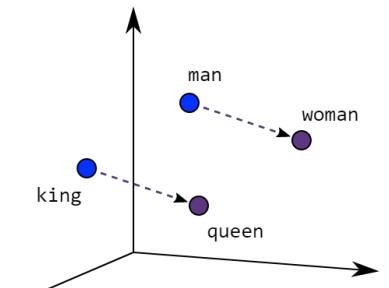
- ***Multi-hot (binary)***: Encode the tokens as a multi-hot binary vector
- ***One-hot***: Encode the tokens as one-hot binary vectors
- ***TF-IDF***: term frequency, inverse document frequency



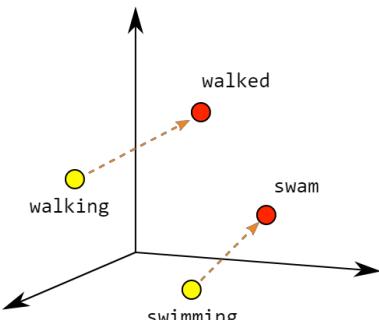


# Word Embedding

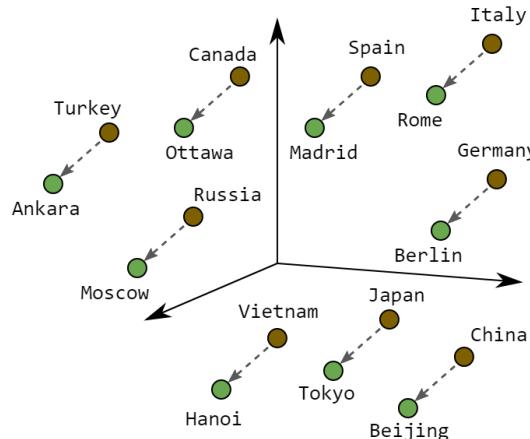
- Word embedding are **dense**, **lower dimensional**, **structured** representations **learned from data**.



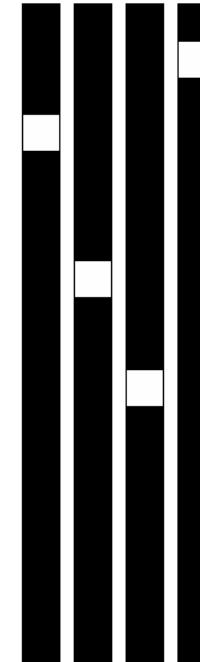
Male-Female



Verb Tense

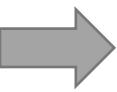


Country-Capital



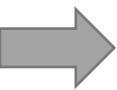
Word embeddings:  
- Dense  
- Lower-dimensional  
- Learned from data

One-hot word vectors:  
- Sparse  
- High-dimensional  
- Hardcoded



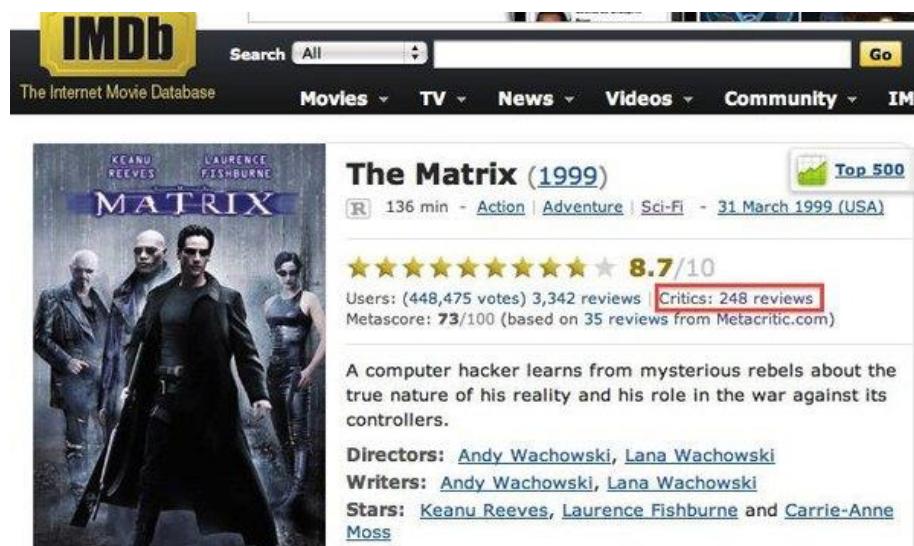
# Representing groups of words

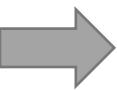
	Bag-of-Words Approach	Sequential Modeling Approach
<b>Input</b>	Collection of text documents	Sequence of words
<b>Order</b>	Ignores the order of words	Captures the order of words
<b>Context</b>	Ignores the context of words	Captures the context of words
<b>Features</b>	Each word is a feature	Features depend on the model architecture
<b>Dimensionality</b>	High dimensionality	Lower dimensionality
<b>Sparsity</b>	High sparsity	Lower sparsity
<b>Performance</b>	Faster training and inference	Slower training and inference
<b>Suitability</b>	Good for simple tasks like document classification	Good for complex tasks like language modeling and machine translation



# A simple example

- Historically, most early applications of machine learning to NLP involved bag-of-words models.
- Interest in sequence models started rising in 2015, with the rebirth of RNN
- Today, both approaches remain relevant. Let's look at a simple example: [IMDB movie review](#)
- Task: sentiment-classification of IMDB movie reviews (positive-negative)





## IMDB review example (bag-of-word approach)

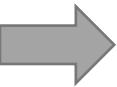
- Single words (**Unigram**) with binary encoding (**multi-hot**)
- You can represent the entire text as a single vector!
- The cat sat on the mat → {"cat", "mat", "sat", "on", "the"} → (0,0,1,..,1,0,...,1,0,..,1,..1)
- Bag-of-words model is mainly made of **dense layers** (No recurrent layers)

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(max_tokens=20000, hidden_dim=16):
    inputs = keras.Input(shape=(max_tokens,))
    x = layers.Dense(hidden_dim, activation="relu")(inputs)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model
```

```
text_vectorization = TextVectorization(
    max_tokens=20000,
    output_mode="multi_hot",
```

- Naïve baseline = **50%** (balance data)
- Test accuracy = **89.2%**



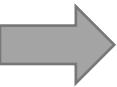
## IMDB review example (improving bag-of-word approach)

- We started with Single words (**Unigram**) with binary encoding (**multi-hot**)
- Two words (**Bigrams**) with binary encoding (**multi-hot**): adding **local order** information
- The cat sat on the mat → {"the", "the cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the mat", "mat"}

### Test accuracy

- Naïve baseline = **50%** (balance data)
- Unigram binary = **89.2%**
- Bigram binary = **90.4%**

```
text_vectorization = TextVectorization(  
    ngrams=2,  
    max_tokens=20000,  
    output_mode="multi_hot",
```



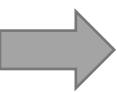
## IMDB review example (improving bag-of-word approach)

- Bigrams with TF-IDF encoding

Test accuracy

- Naïve baseline = 50% (balance data)
- Unigram binary = 89.2%
- Bigram binary = 90.4%
- Bigram TF-IDF = 89.8%
- Typically, TF-IDF increases the model performance by 1%. It wasn't the case for the IMDB dataset.

```
text_vectorization = TextVectorization(  
    ngrams=2,  
    max_tokens=20000,  
    output_mode="tf_idf",
```

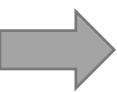


# IMDB review example (sequence modeling approach)

1. Starting by representing the input as sequences of **integer** indices (one integer for one word)
2. Map each integer to a vector to get **sequences of vectors** (one-hot or word embedding)
3. Finally, feed these vectors into a stack of layers such as **1D Convnet, RNN** or a **Transformer**.

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
```



# IMDB review example (sequence modeling approach)

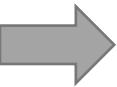
- One-hot encoding approach: Encode the integers into binary 20,000-dimensional vectors.

```
import tensorflow as tf
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

- This model trains very slowly.  $600 * 20,000 = 12$  million floats for a single movie review!
- There must be a better way than one-hot encoding!

## Test accuracy

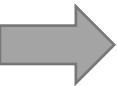
- Naïve baseline = 50% (balance data)
- Unigram binary = 89.2%
- Bigram binary = 90.4%
- Bigram TF-IDF = 89.8%
- Sequence one-hot encode = 87%



# IMDB review example (improving sequence modeling approach)

- Word-embedding approach:
  1. Simultaneously trained word embedding (trained jointly with the weights of a NN)
  2. Pretrained word embedding
- Sometimes it is reasonable to learn a new embedding space for different tasks (for example movie-review is different from legal-document classification)
- Word index → word embedding layer → corresponding word vector

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

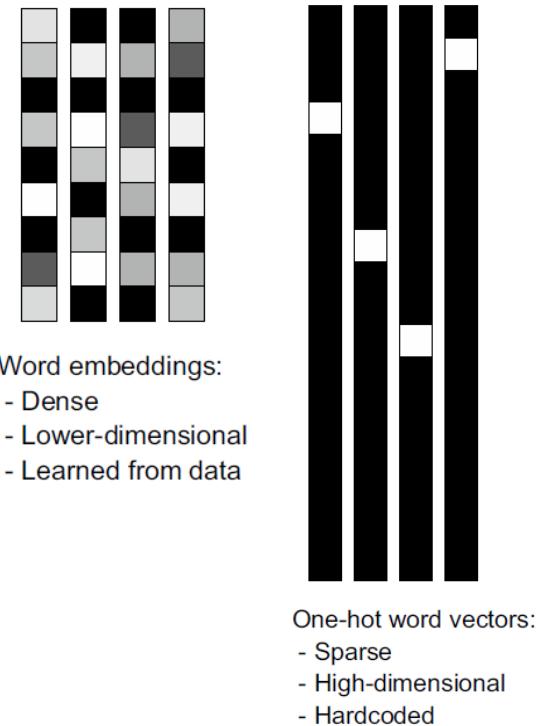


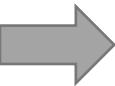
# IMDB review example (improving sequence modeling approach)

- Word-embedding approach performance:
- It trains much faster than the one-hot encoding (256 vs 20k dim)

## Test accuracy

- Naïve baseline = 50% (balance data)
- Unigram binary = 89.2%
- Bigram binary = 90.4%
- Bigram TF-IDF = 89.8%
- Sequence one-hot encode = 87%
- Sequence word embedding simultaneously trained = 87%
- Sequence word embedding simultaneously trained with masks = 88%





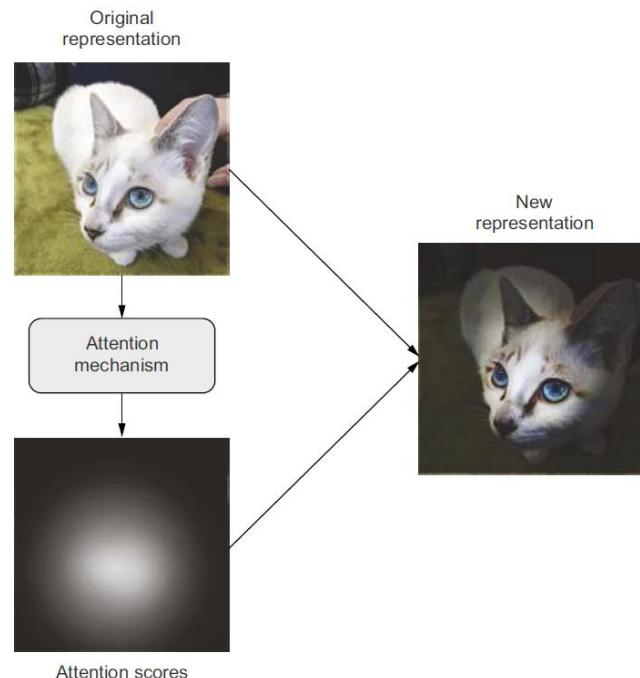
# Pretrained word embedding

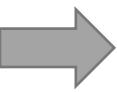
- Motivation: little training data! The network cannot learn an appropriate embedding.
- Analogous to using pretrained convnets in image classification
- Useful when expecting generic features.
- Schemes:
  - The Word2Vec algorithm (Tomas Mikolov at Google 2013): a predictive model that uses a neural network to predict the probability of a word given its context
  - GloVe (Stanford researchers 2014): a count-based model that computes the co-occurrence probabilities between words in a corpus
- Word2Vec is faster to train than GloVe, especially for large corpora

# Module 7 – Part 1

## Transformers prerequisites

### Why Attention is **ALL** you need?

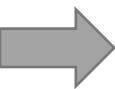




# Road map!

- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- Module 6- Deep Sequence Modeling (RNN, LSTM)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

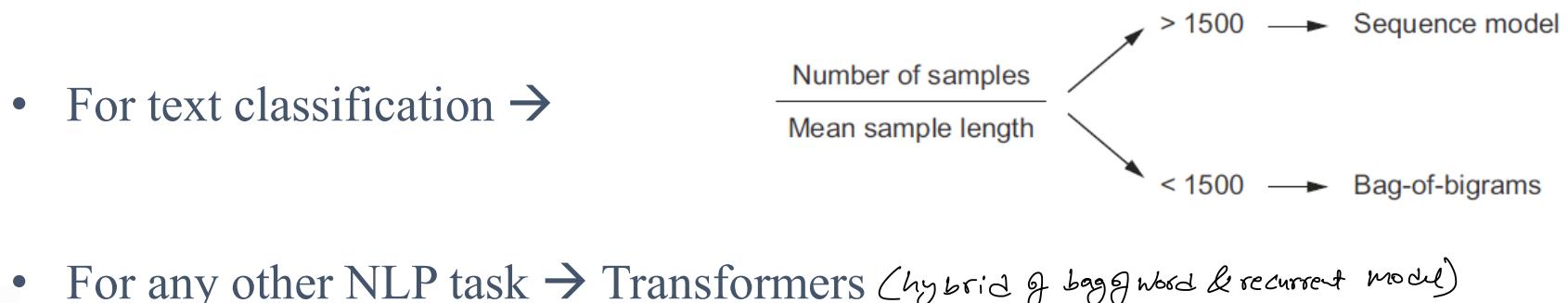




# Transformers for NLP

Research paper  
↑

- Starting in 2017 (**Attention is all you need!**), **transformers** started overtaking RNN across most NLP tasks.
- NLP architecture depends on word representation method
  - **Discard order** and treat text as an unordered set of words → bag-of-words models
  - **Respect order** and treat words one at time (steps in timeseries) → recurrent models
- When to use sequence model over bag-of-words?



- For text classification →

- For any other NLP task → Transformers (*hybrid of bag-of-word & recurrent model*)

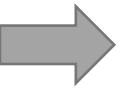
# Transformers vs other sequence models

- Transformer architecture is technically **order-agnostic**, yet it injects word-position information into the representations it processes (**hybrid approach**)
- Transformers simultaneously look at different parts of a sentence (unlike RNNs) while still being order-aware.

The cat, sat on the mat.

NLP Models	Word order awareness	Context awareness (cross-word interactions)
Bag of unigrams	No	No
Bag of Bigrams	Very limited	No
RNN family <small>(i.e. Simple RNN, LSTM, GRU)</small>	Yes	No
Transformer	Yes	Yes



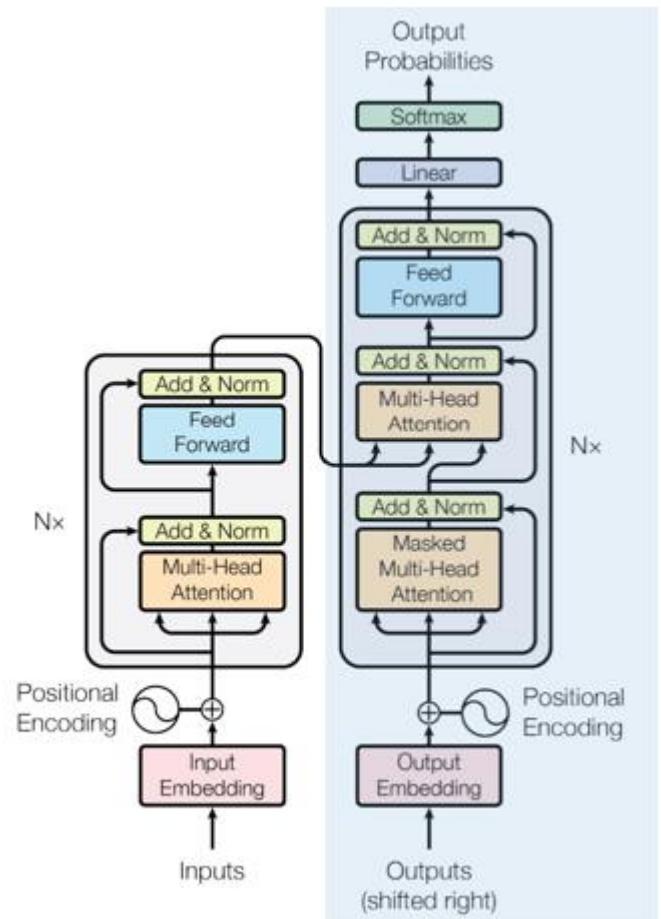
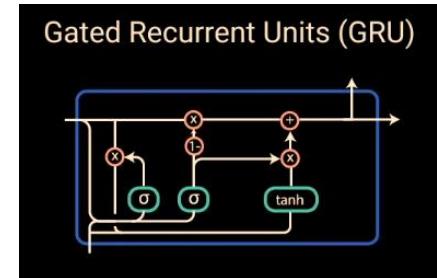
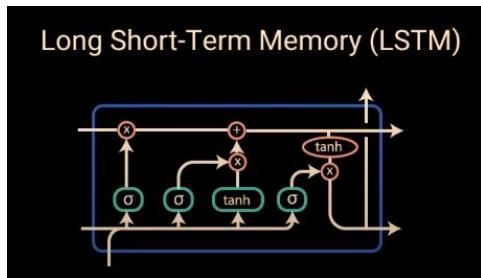
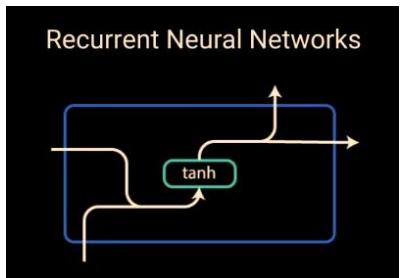


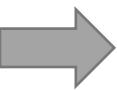
# Sequence Modeling Design Criteria

Transformers

To model sequence data efficiently, we need an architecture that:

- Preserve the **order**
- Account for **long-term dependencies**
- Handle different **input-length**
- Share parameters across the sequence

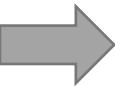




# Applications of Transformers

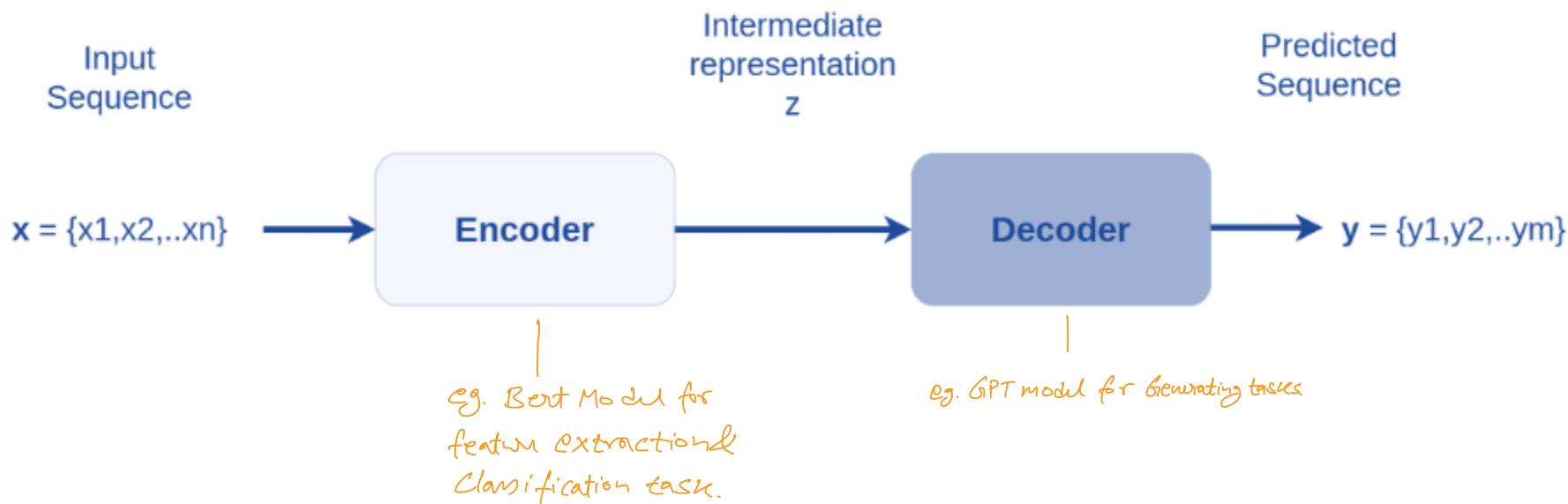
- Transformers are taking NLP, Computer vision and reinforcement learning by storm.
- NLP applications:
  - Machine translation, text generation, text summarization, text classification, chatbots, questions answering etc.
  - BERT, GPT
- Computer vision applications:
  - Image captioning, object detection and segmentation
  - ViT (vision transformers)
- Reinforcement learning applications:
  - Game playing, robotics and autonomous driving

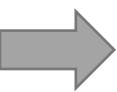




# Sequence-to-sequence modeling

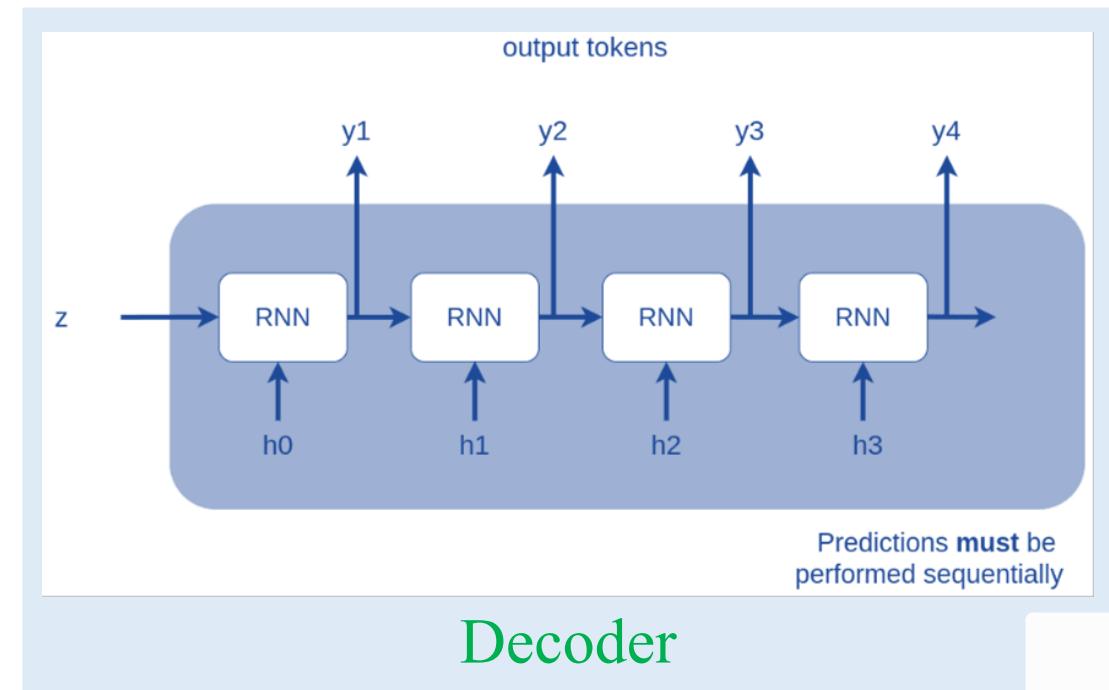
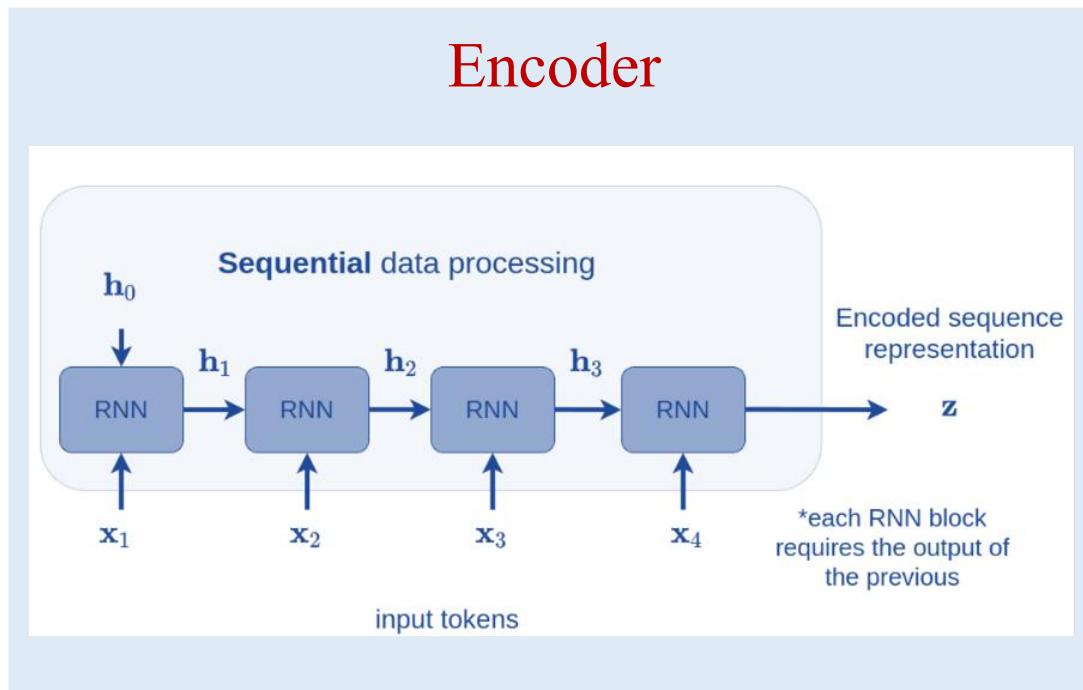
- The goal is to **transform** an input sequence (**source**) to a new one (**target**).

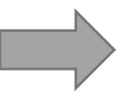




# Encoder – Decoder

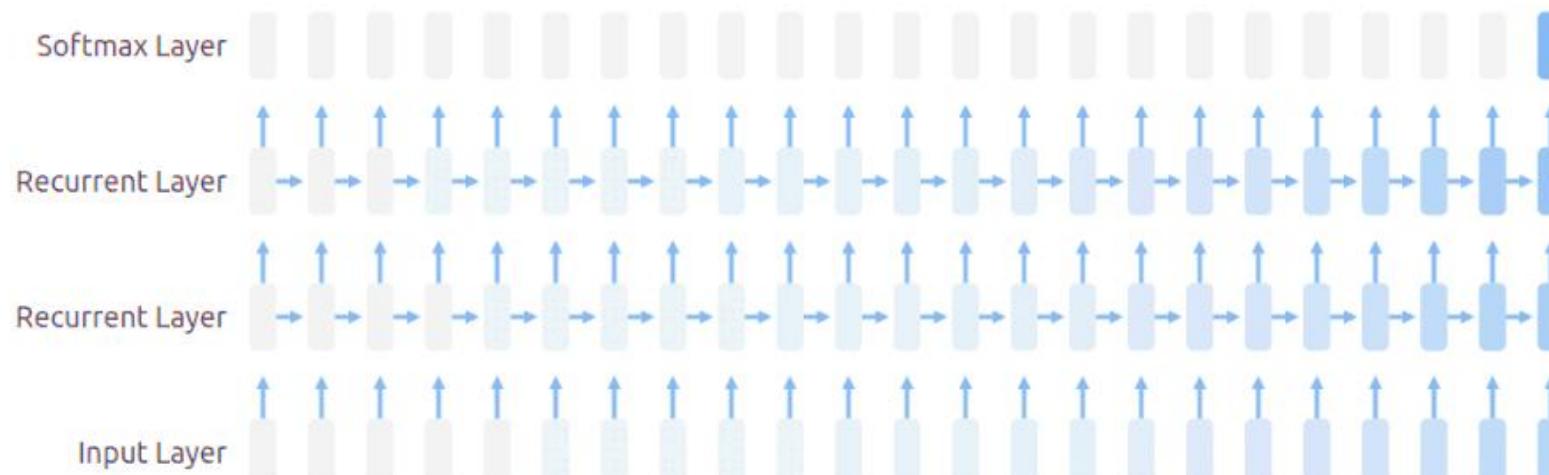
- Recurrent Neural Networks (RNNs) were the prevailing method for sequence-to-sequence learning until Transformers demonstrated superior performance.





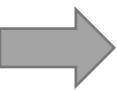
# Limitations of RNN

- **Bottleneck problem:** the Encoder state vector(s) must store the entire input sequence representation → Significant **limitations on** translatable sentence **size** and **complexity**
- RNN tends to progressively **forget about the past** (~100 tokens) and eventually pays more attention to the **last parts** of the sequence.
- **Vanishing gradient** problem:



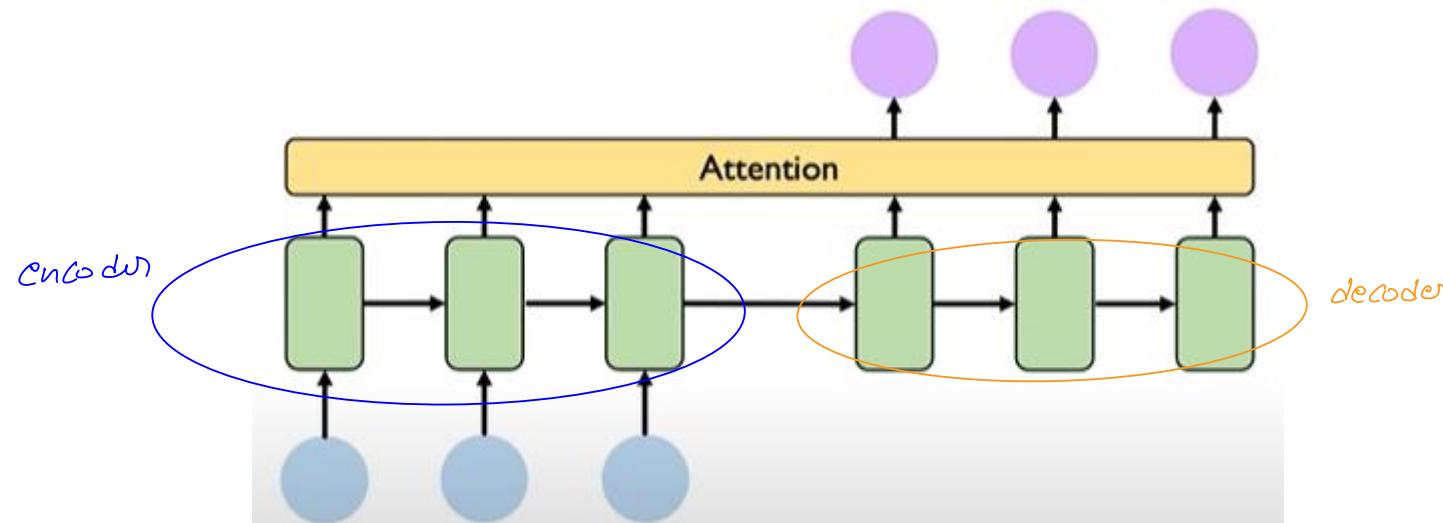
**Vanishing Gradient:** where the contribution from the earlier steps becomes insignificant in the gradient for the vanilla RNN unit. <https://distill.pub/2019/memorization-in-rnns/>

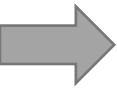
# Attention is ALL you need!



# Attention mechanism

- To avoid **vanishing gradient**, we need to form a **direct connection** with each timestamp.
- By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence (**bottleneck problem**)
- Attention is a mechanism that allows the model to **weigh (score)** and focus on specific parts of the input when generating output.
- Attention mechanism can be applied to any encoder and decoder architecture (RNN, LSTM, GRU, CNN, etc)





# Attention, deep dive

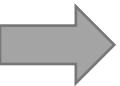
- Introduction to Attention Mechanisms: Inspired by Brandon Rohrer's High-Level Example
- Attention mechanism as a selective second-order model with skips

The screenshot shows the first few paragraphs of a blog post titled "Transformers from Scratch" by Brandon Rohrer. The title and author name are in a dark blue header bar. The main content area has a light orange background.

I procrastinated a deep dive into transformers for a few years. Finally the discomfort of not knowing what makes them tick grew too great for me. Here is that dive.

Transformers were introduced in this 2017 [paper](#) as a tool for sequence transduction—converting one sequence of symbols to another. The most popular examples of this are translation, as in English to German. It has also been modified to perform sequence completion—given a starting prompt, carry on in the same vein and style. They have quickly become an indispensable tool for research and product development in natural language processing.

Before we start, just a heads-up. We're going to be talking a lot about matrix multiplications and touching on backpropagation (the algorithm for training the model), but you don't need to know any of it beforehand. We'll add the concepts we need one at a time, with explanation.



# Attention, deep dive

the train left the station  
① ② ③ ④

- What does it mean **mathematically**?
- **Vocabulary**: collection of symbols in each sequence
- Converting symbols to numbers: **One-hot encoding**
- Dot product can be used to measure **similarity**
- One-hot vectors can pull out a particular row of a matrix!

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}^T = 0$$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}^T = 1$$

Not similar | exactly same.

Transpose version of 3 vectors

① →  $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

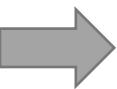
② →  $\begin{bmatrix} .2 & .9 \\ .7 & 0 \\ .8 & .3 \\ .1 & .4 \end{bmatrix}$

③ →  $\begin{bmatrix} .2 & .9 \\ .1 & .4 \\ .8 & .3 \end{bmatrix}$

A                      B

=

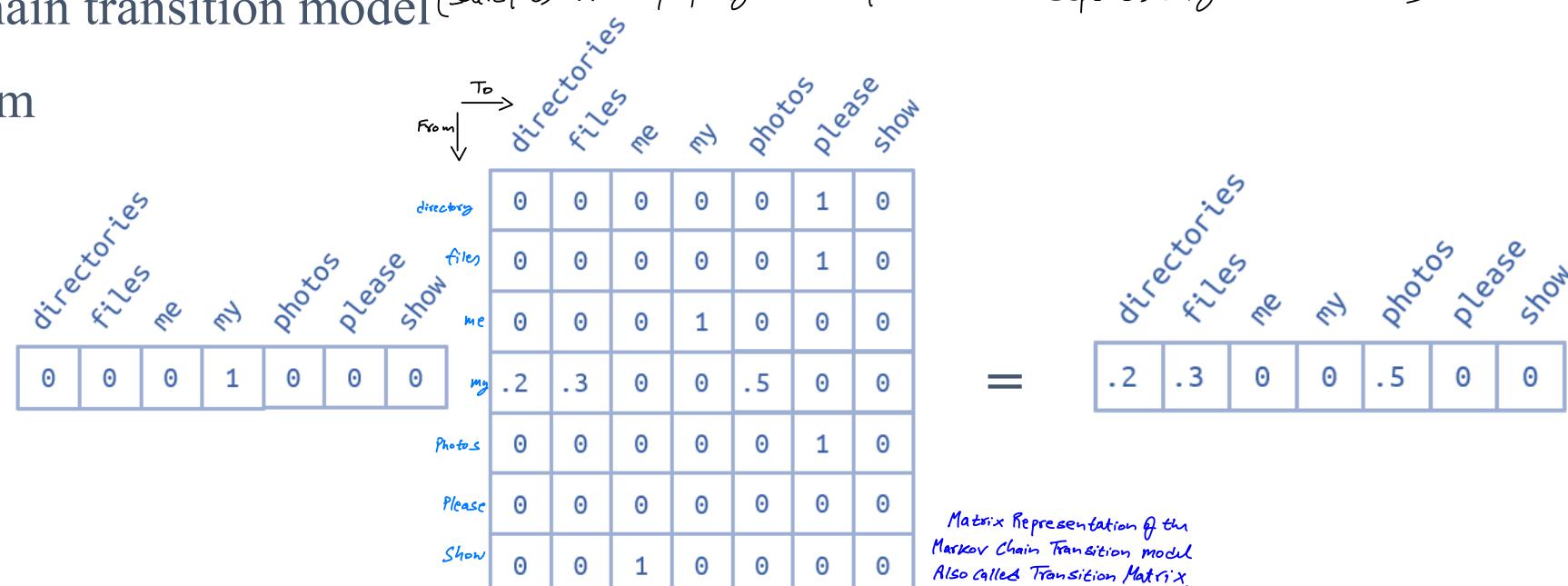
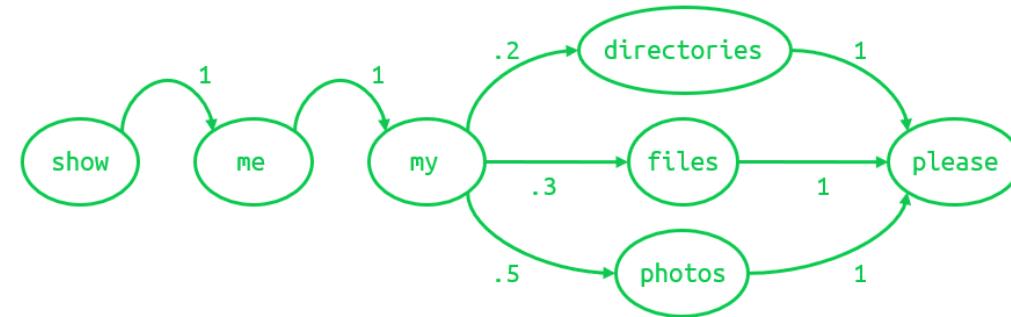
The dot product is extracting rows of matrix B. 1<sup>st</sup> vector extracts row 1, 2<sup>nd</sup> vector extracts row 4 & 3<sup>rd</sup> vector extracts row 3.

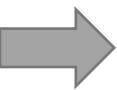


# First order sequence model

- Example:
  - Show me my **directories** please
  - Show me my **files** please
  - Show me my **photos** please
- Vocabulary size = 7 {directories, files, me, my, photos, please, show}.
- Markov chain transition model (satisfies Markov property i.e. Prob. for next word depends only on recent words)
- Matrix form

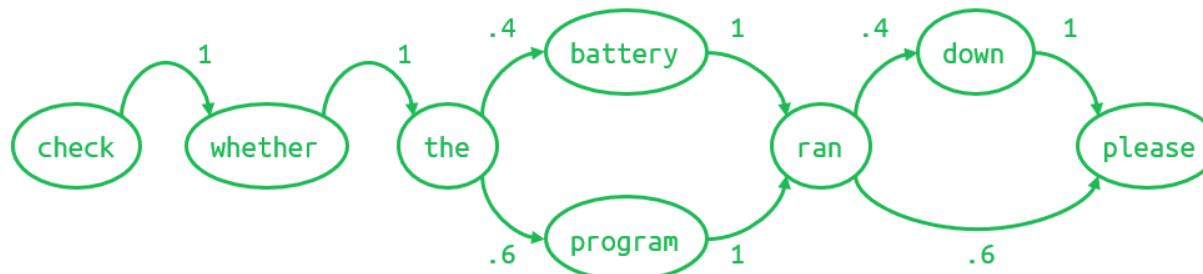
1st order Markov model (only looks at one single most recent word)

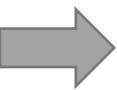




# Second order sequence model

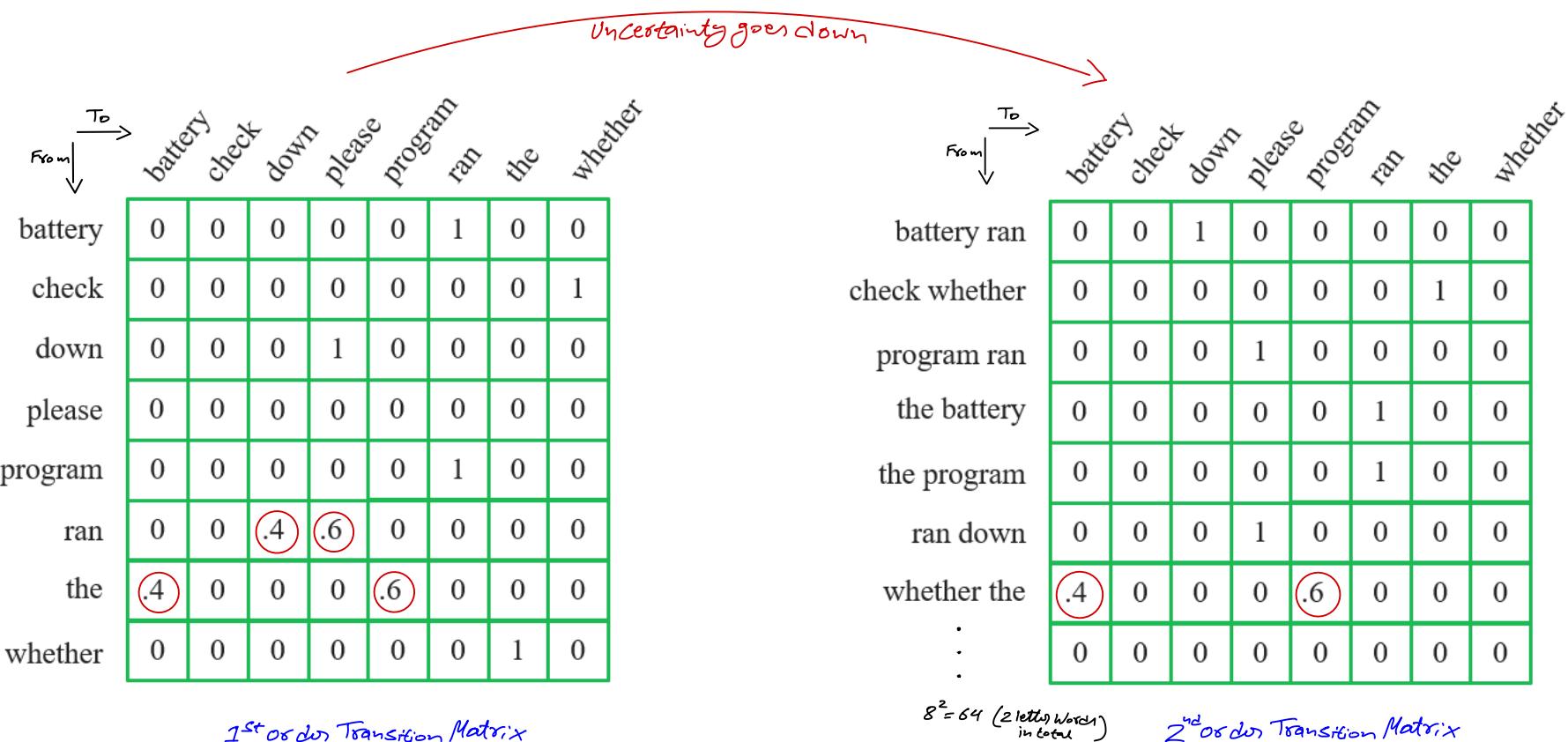
- First order Markov model only looks at the single most recent word.
- Predicting based on only one last word is hard! Let's consider the two most recent words!
- Example: (a 40/60 proportion)
  - Check whether the **battery ran down** please. *(Prob. of 40% this sentence)*
  - Check whether the **program ran** please. *(Prob. of 60% this sentence)*
- First order model:
- How can we remove the uncertainty after the word “ran”? *Look at recent 2 words instead of 1 word.*

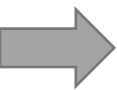




# Second order sequence model

- Check whether the **battery ran down** please.
- Check whether the **program ran** please.
- **Vocabulary:** {battery, check, down, please, program, ran, the, whether} **size = 8**



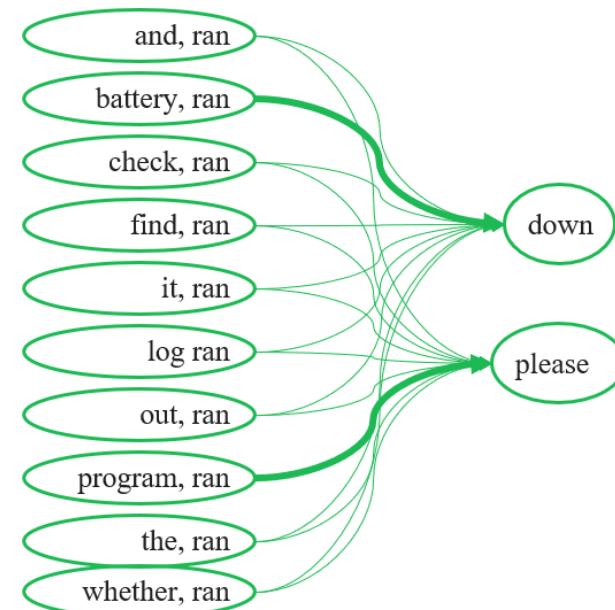
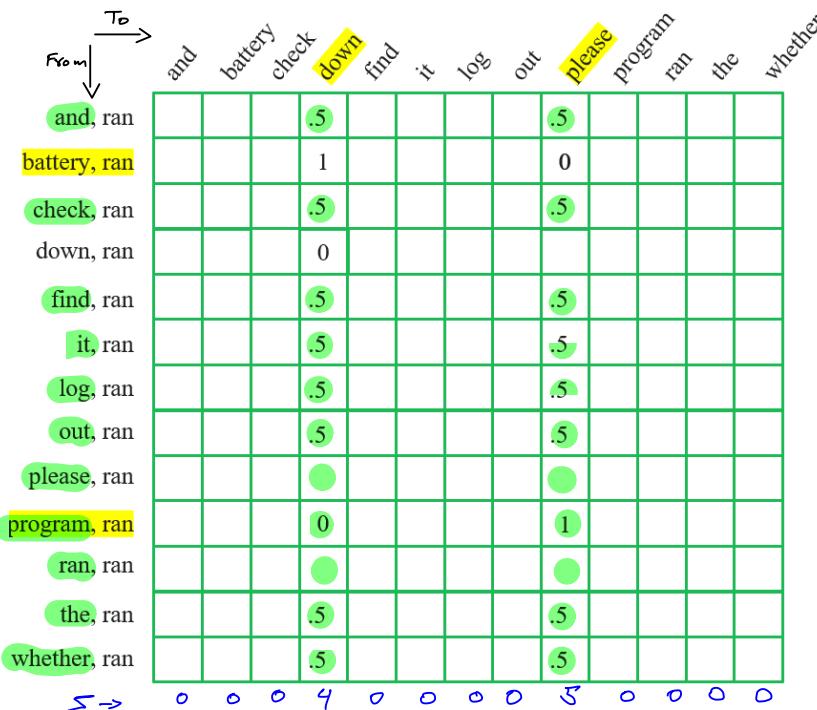


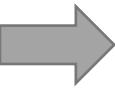
# Higher order sequence models?

- ① • Check the **program** log and find out whether it **ran** please.
- ② • Check the **battery** log and find out whether it **ran** down please.
- What comes after the word “**ran**”? It is unreasonable to investigate 9<sup>th</sup> order sequence model!  
(Vocab Size<sup>9</sup>) combinations!  $13^9$  combinations.
- Solution: Second order sequence model **with skips**

13x13 Transition Matrix

• for the ① sentence since max. is 5, the prediction after word "ran" is "please".



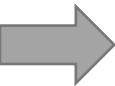


# Masking

- Masking: Crossing out all the uninformative feature votes
- The only important rows are *battery, ran* and *program, ran*. We could mask everything else!

Check the *program* log and find out whether it *ran* please.

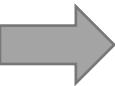




## Masking: selective second order model with skips

- The mask has the effect of **hiding** a lot of the transition matrix
- In this example, it hides the combination of **ran** with everything except **battery** and **program**, leaving **just the features that matter**
- This process of selective masking is the **attention** thing!

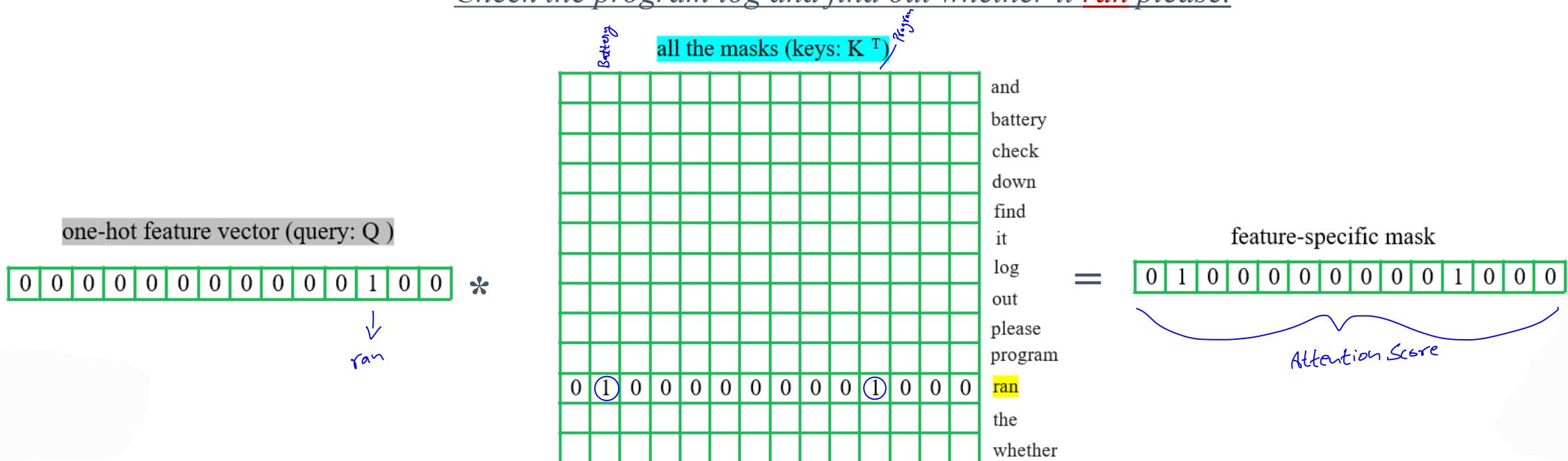
	and	battery	check	down	find	it	log	out	please	program	ran	the	whether
and, ran													
battery, ran				1					0				
check, ran													
down, ran													
find, ran													
it, ran													
log, ran													
out, ran													
please, ran													
program, ran									0		1		
ran, ran													
the, ran													
whether, ran													

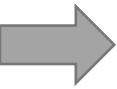


# Attention as Matrix Multiplication

- Stack the mask vectors for every word into a matrix (**Keys**)
- Use one-hot representation of the most recent word (**Query**) to pull out the relevant mask (**attention score**)
- Weight the tokens in the input sequence (**Values**) by the attention scores to create the **context vector**.

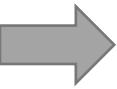
Check the program log and find out whether it *ran* please.





# Connecting the dots

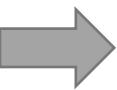
- This high-level description can be connected to the self-attention mechanism used in Transformers as follows: **selective second order model with skips**
- **Selective:** Self-attention mechanisms learn to weigh different parts of the input based on their relevance to the current computation.
  - In the context of Transformers, this is achieved by using the **dot product** between the **query** and **key** vectors to calculate **attention scores**
- **Second order:** After obtaining the attention scores, the self-attention mechanism computes a weighted sum of the value vectors. This is a second order operation (**attention scores \* value**)  $= \text{Context Vectors}$
- **Model with skips:** Self-attention mechanisms can capture long-range dependencies in the input by effectively "skipping" over less relevant parts



# Ways to compute attention

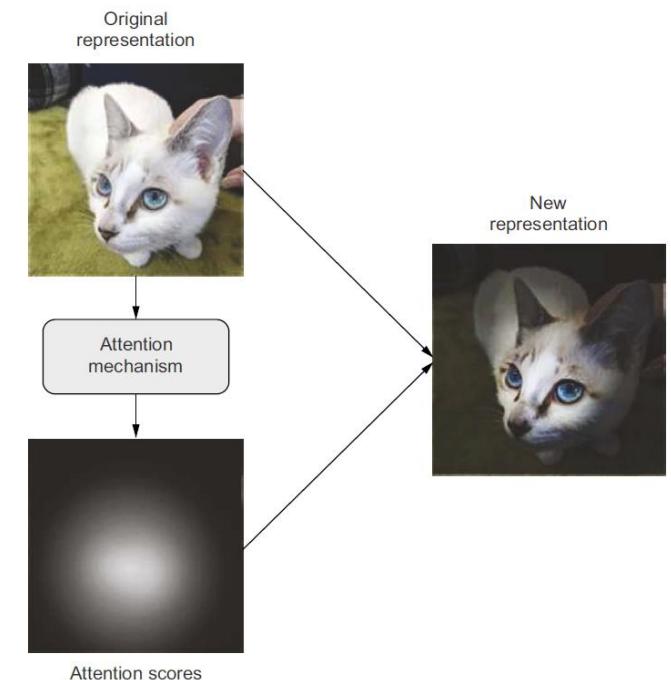
- There are several different ways to compute attention scores, including:
  - ✓ Dot-product Attention
  - ✓ Scaled Dot-Product Attention
  - ✓ Additive Attention
  - ✓ Multiplicative Attention
  - ✓ Location-Based Attention

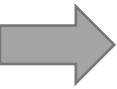
Name	Alignment score function	Citation
Content-base attention	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \text{cosine}[\mathbf{s}_t, \mathbf{h}_i]$	Graves2014
Additive(*)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_t; \mathbf{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017



# Self-Attention

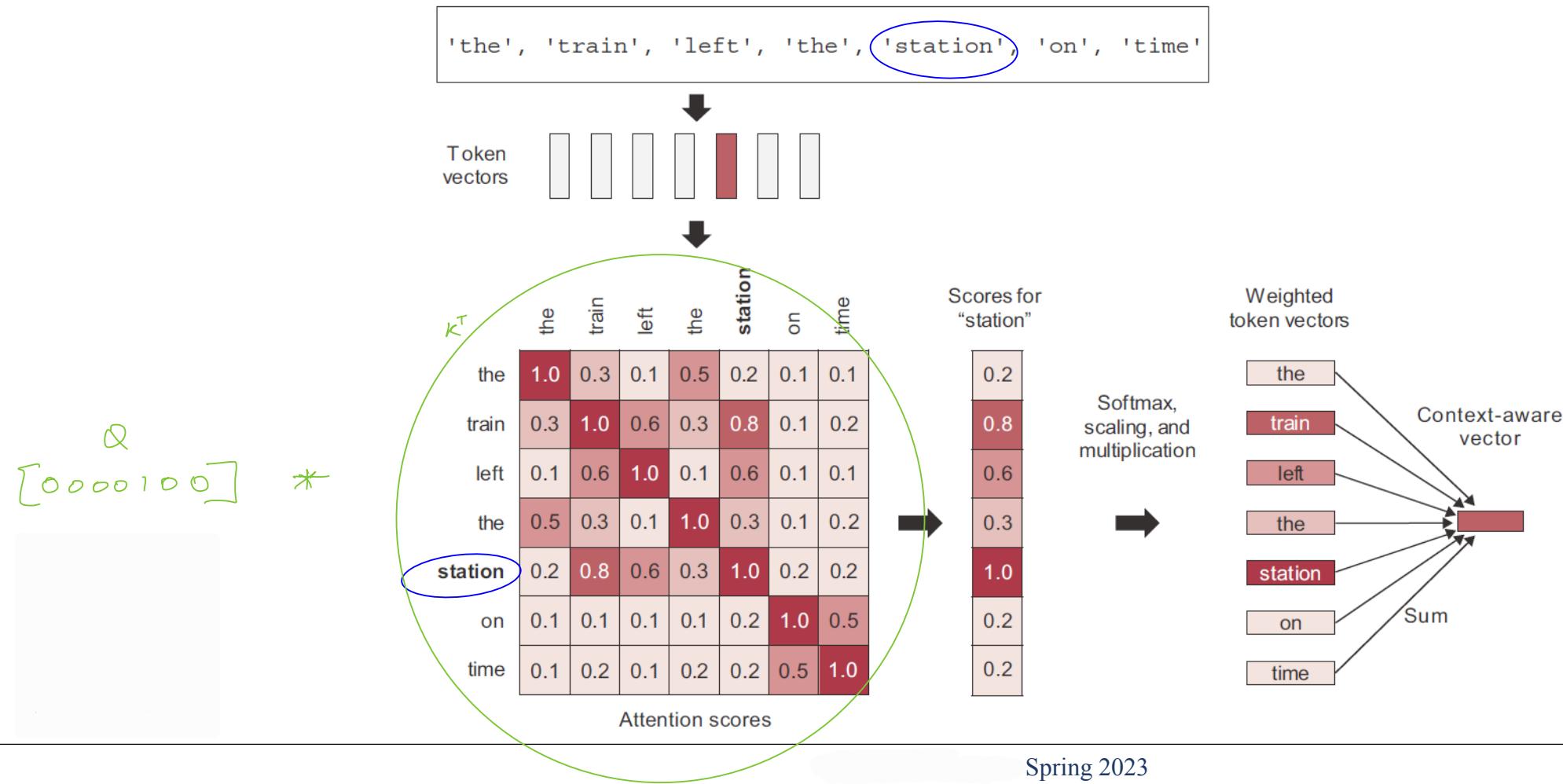
- Self attention is a variation of the attention mechanism where the input and output sequences are the same, meaning the model is attending to its own input.
- Self-Attention starts by computing **scores** for a set of features. High score → more relevant
- Self attention allows the model to **relate** different parts of the input sequence to each other, capturing dependencies and relationships within the sequence itself
- Have we seen this idea before? Max Pooling and TF-IDF

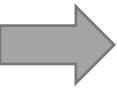




# Self-Attention (context-aware representation)

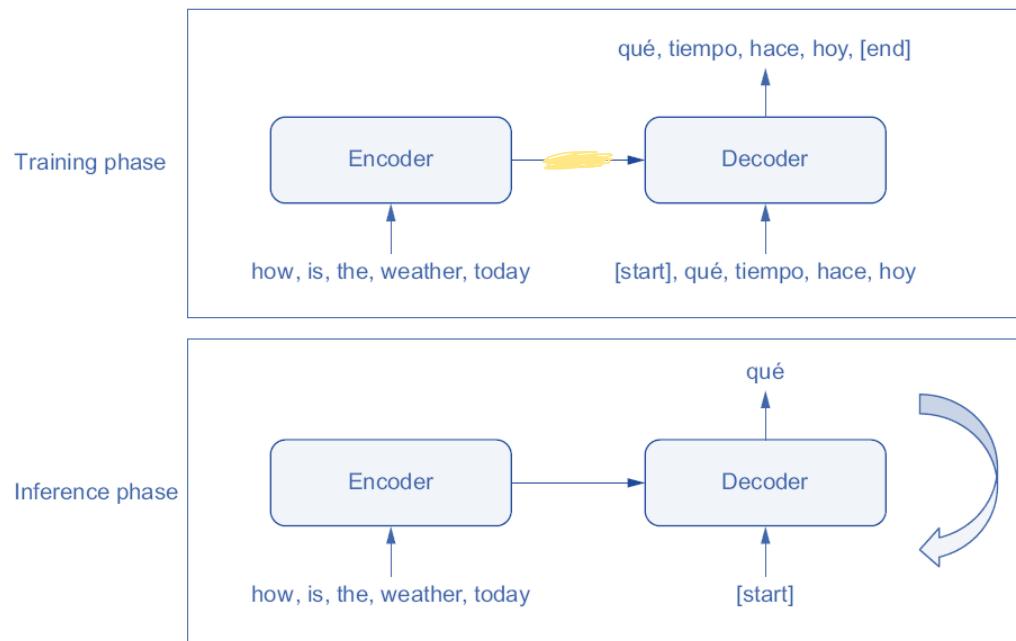
- Self-attention helps to **adjust the representation** of a **token** by considering the information from **related tokens** in the input sequence → **context awareness**

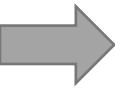




# Encoder–Decoder (Machine Translation)

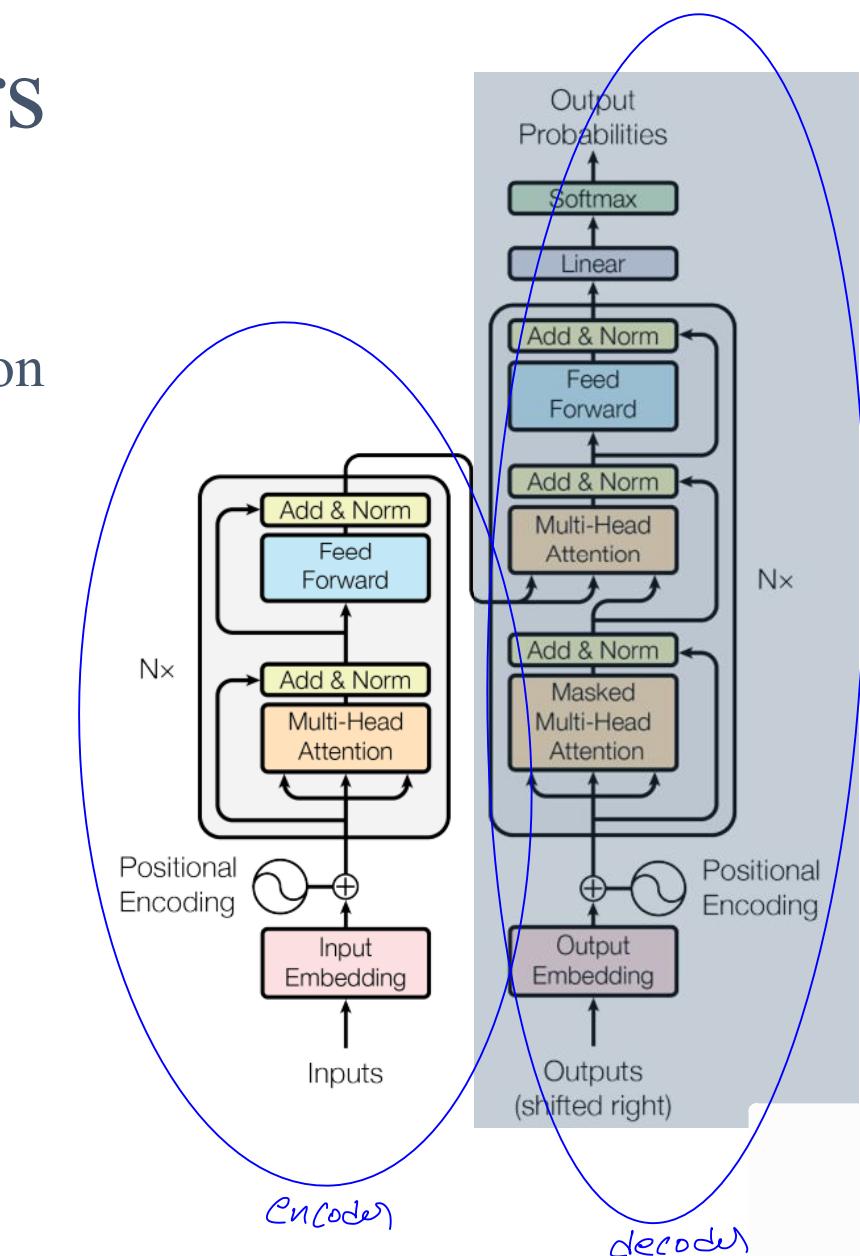
- An **encoder model** turns the source sequence into an **intermediate representation**.
- A **decoder model** is trained to predict the next token in the target sequence by looking at both previous tokens and the encoded source sequence.
- During **inference**, we don't have access to the target sequence (predict it from scratch) → must generate it one token at a time





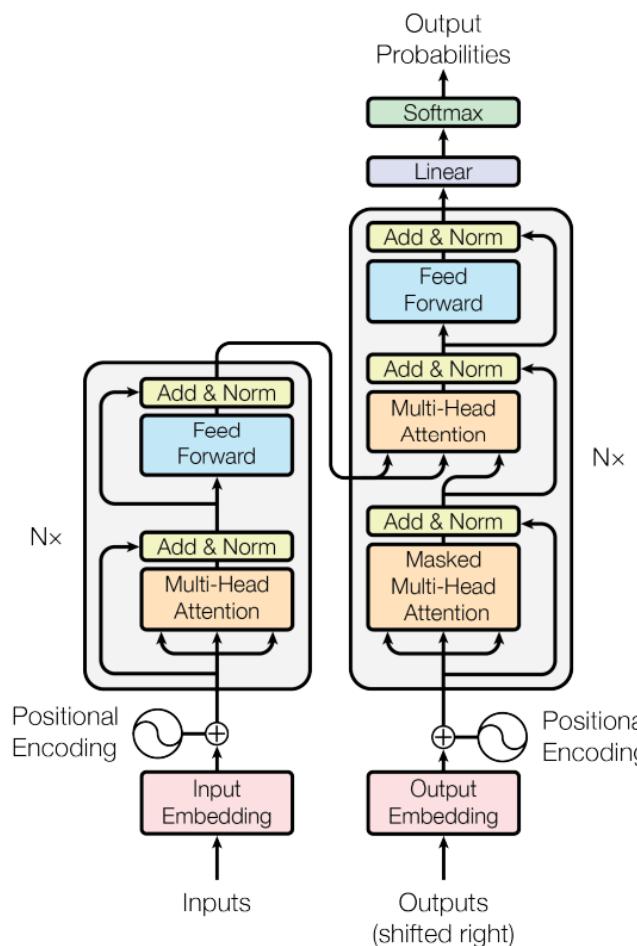
# Self-Attention and Transformers

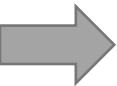
- Transformers **primarily** use **self-attention** mechanisms.
- Both the **encoder** and **decoder** layers apply self-attention to capture relationships and dependencies within the input sequence itself.
- By using self-attention, transformers can:
  - Process inputs in parallel (*Self-attention is Not Sequential*)
  - Identify long-range dependencies
  - Model complex relationships more efficiently compared to RNN and CNN.



# Module 7 – Part 2

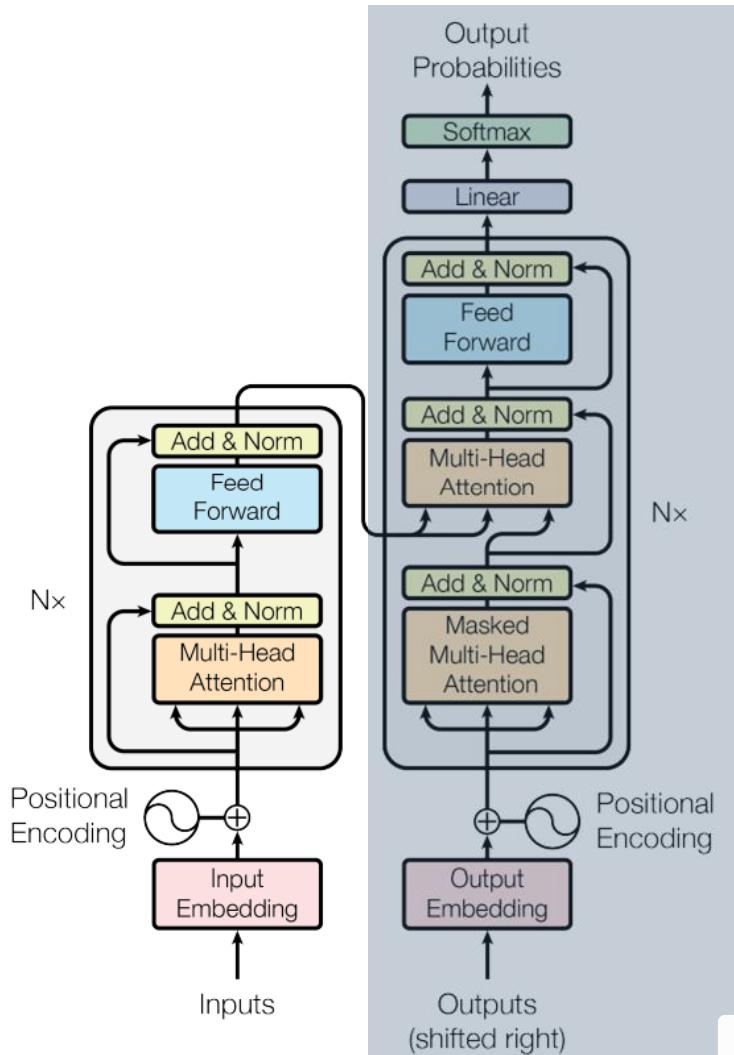
## Transformers Architecture





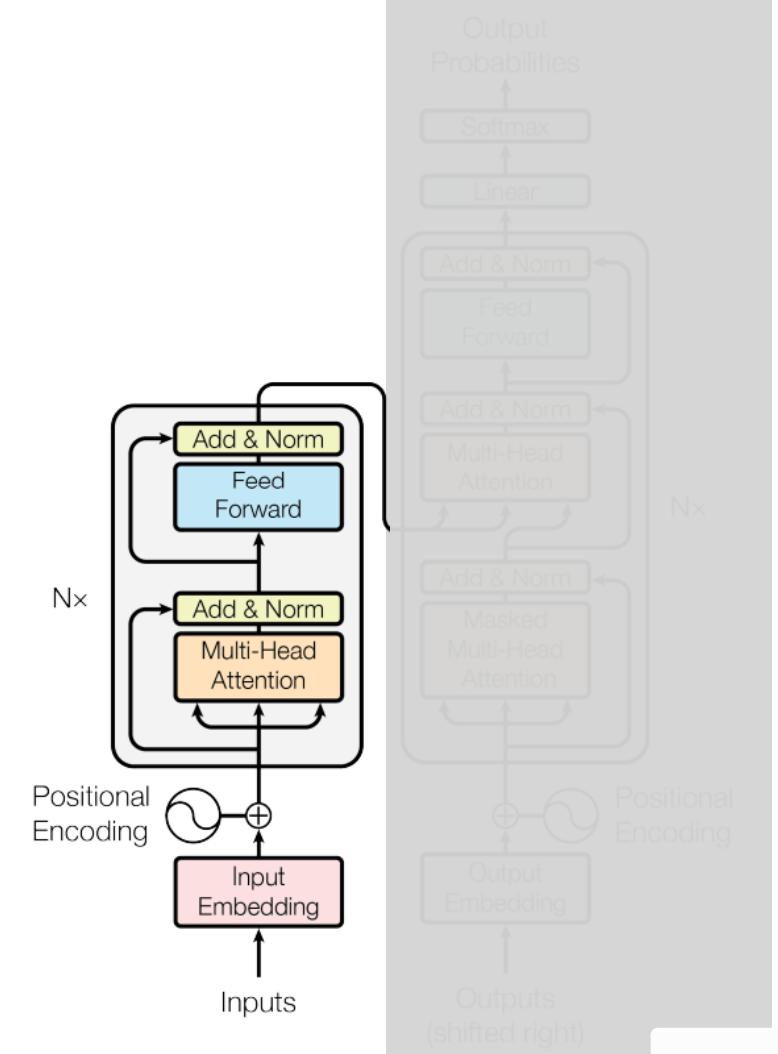
# Transformers outline

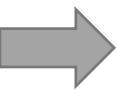
- Encoder – Decoder architecture
  - Embedding and Positional Encoding
  - Self-Attention (scaled dot product attention)
  - Query-Key-Value model
  - (Masked) Multi-head attention
  - Encoder-Decoder attention
  - Residual connections
  - Layer normalization
  - Feed Forward
  - Softmax layer



# Transformer architecture

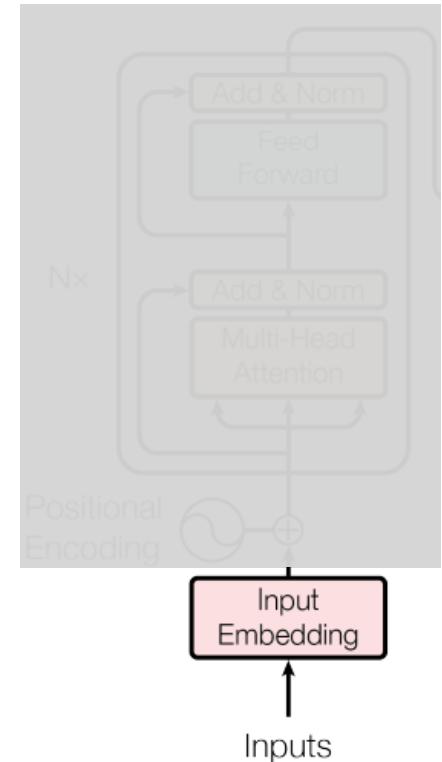
## Encoder

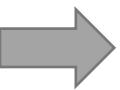




# Input Embedding

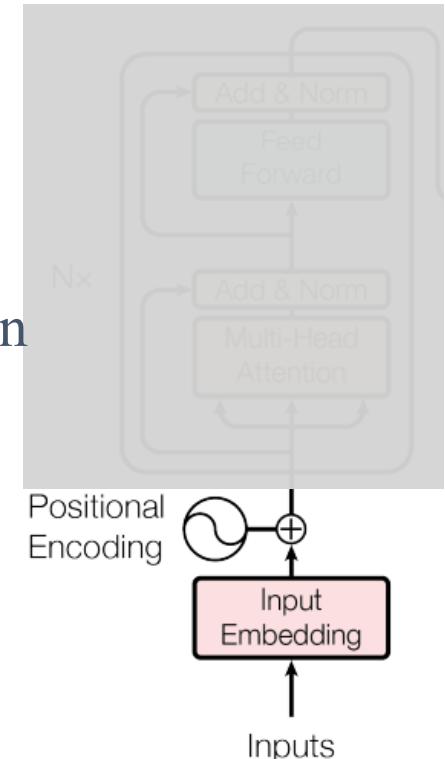
- Word embedding is a numerical representation of words as vectors in a continuous, relatively low-dimensional space compared to the size of the vocabulary, which captures their semantic meaning and relationships with other words.
- Word embedding is learned through a neural network
- Notion of order is lost!
- Example: “I love Transformers!” with embedding dim=10
  - "I" : [-0.2, 0.1, 0.3, -0.4, 0.5, -0.6, -0.1, -0.3, -0.2, 0.4]
  - "love" : [0.3, -0.2, 0.1, 0.5, -0.4, 0.2, -0.6, -0.1, -0.3, 0.2]
  - "transformers" : [-0.4, -0.3, 0.6, -0.1, 0.2, 0.1, 0.4, 0.5, 0.2, 0.3]
  - "!" : [0.1, -0.4, -0.2, 0.3, 0.2, -0.1, 0.5, -0.3, -0.5, -0.4]



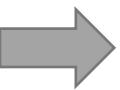


# Positional Encoding

- Positional encoding **reinject** order information
- positional encoding is a set of small constants, which are added to the word embedding vector before the first self-attention layer.
- If the same word appears in a different position, the actual representation will be slightly different, depending on where it appears in the input sentence
- The model **will figure out** how to leverage this additional information.
- Naïve solution: My name is Pedram → {0,1,2,3}
- Better solution: Add a circular wiggle



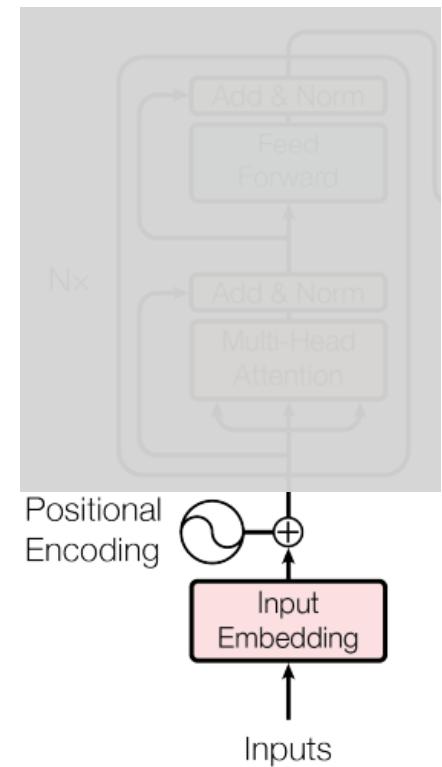
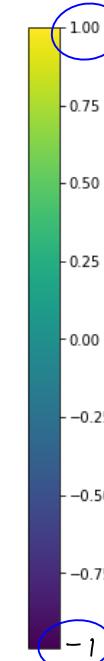
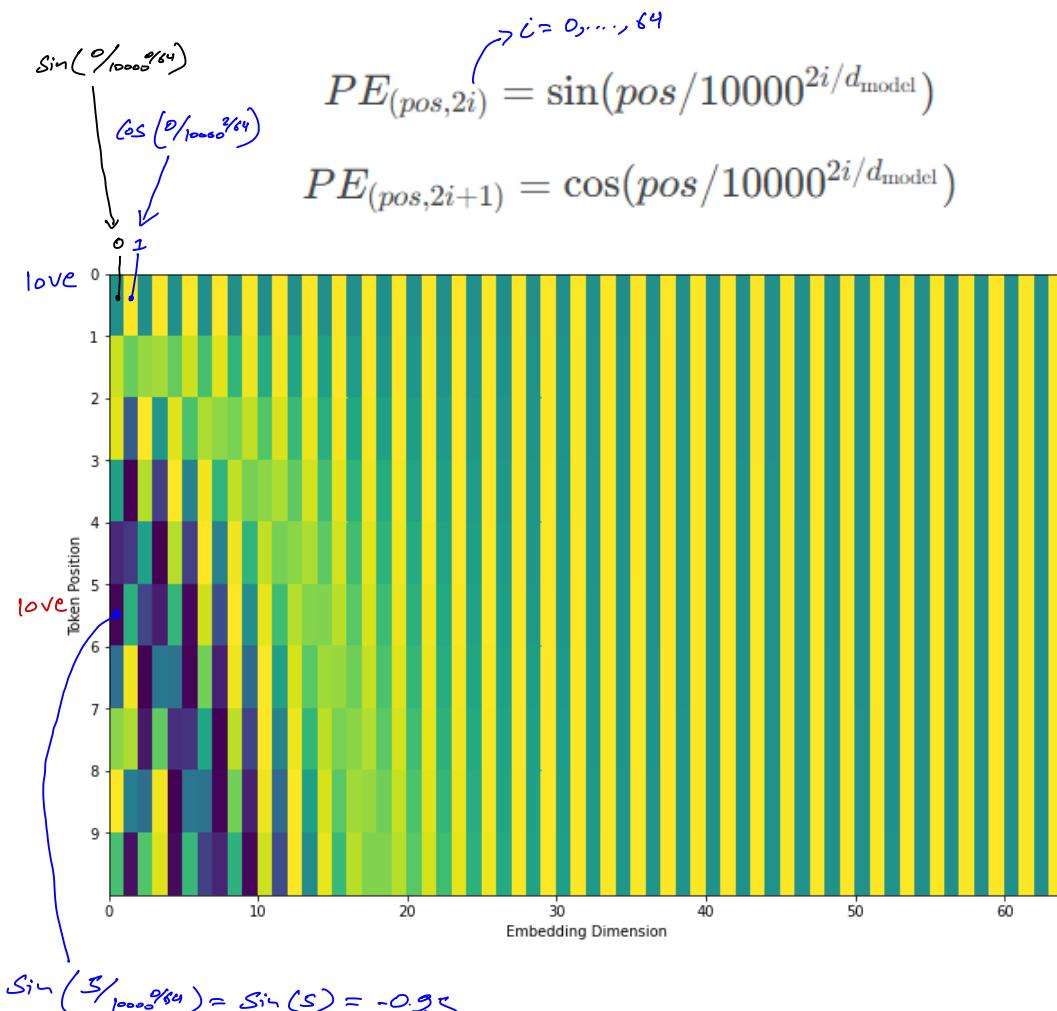
Word embedding + positional encoding =  
positional embedding



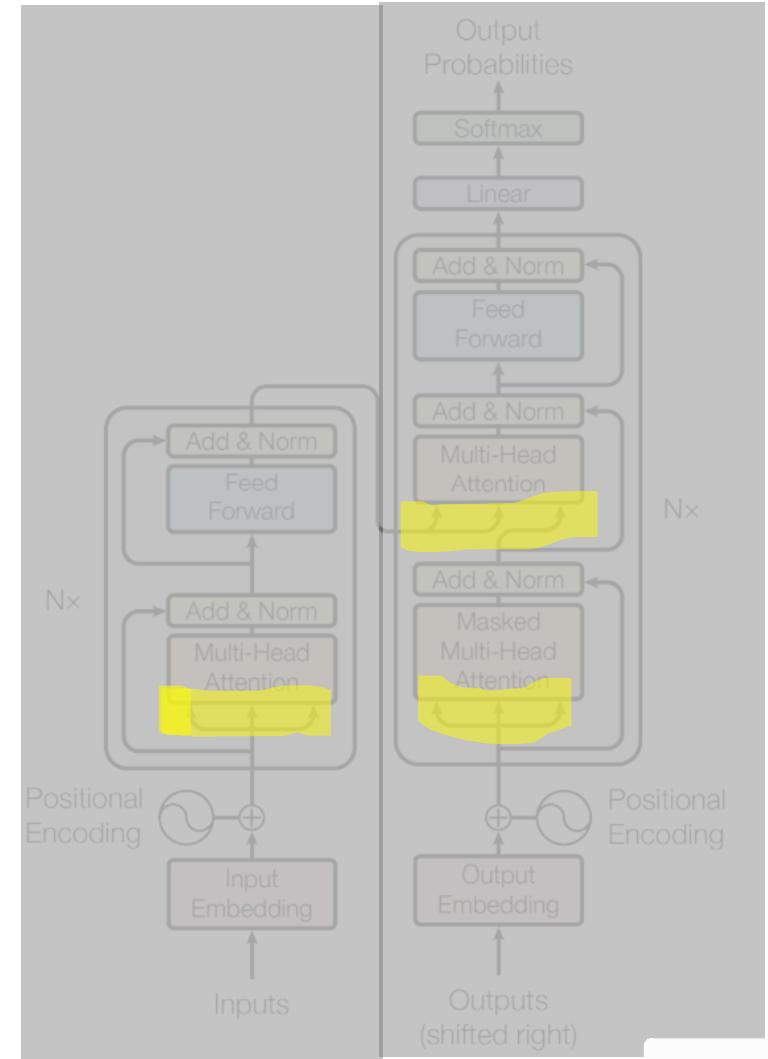
# Positional Encoding

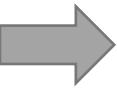
- Better solution: Sine – Cosine function

eg: dim=64.  
love the fact... love...  
love:  $[0.1 \ 0.2 \ 0.3 \dots]^T_{64}$   
 $\begin{array}{r} 0 \\ + \\ 0.1 \ 0 \ 1 \dots \\ \hline 0.1 \ 1.2 \ 0.3 \dots \end{array}^T_{64}$   
love:  $[0.1 \ 0.2 \ 0.3 \dots]^T_{64}$   
 $\begin{array}{r} 0 \\ + \\ 0.85 \ \dots \\ \hline 0.85 \ \dots \end{math>$



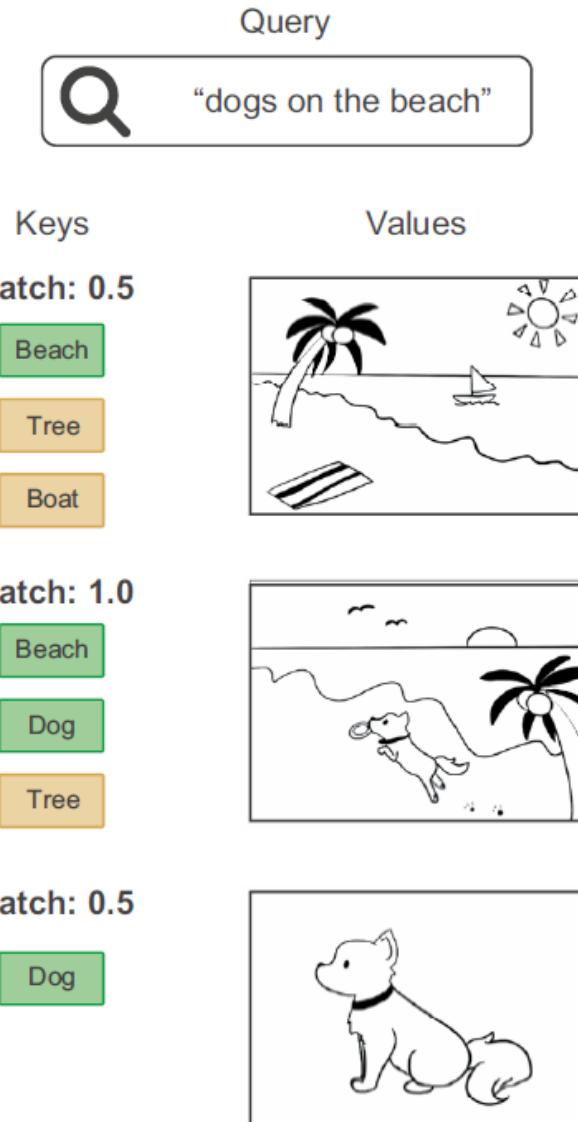
# Self-Attention mechanism Query-Key-Value

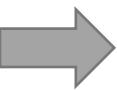




# The Query-Key-Value

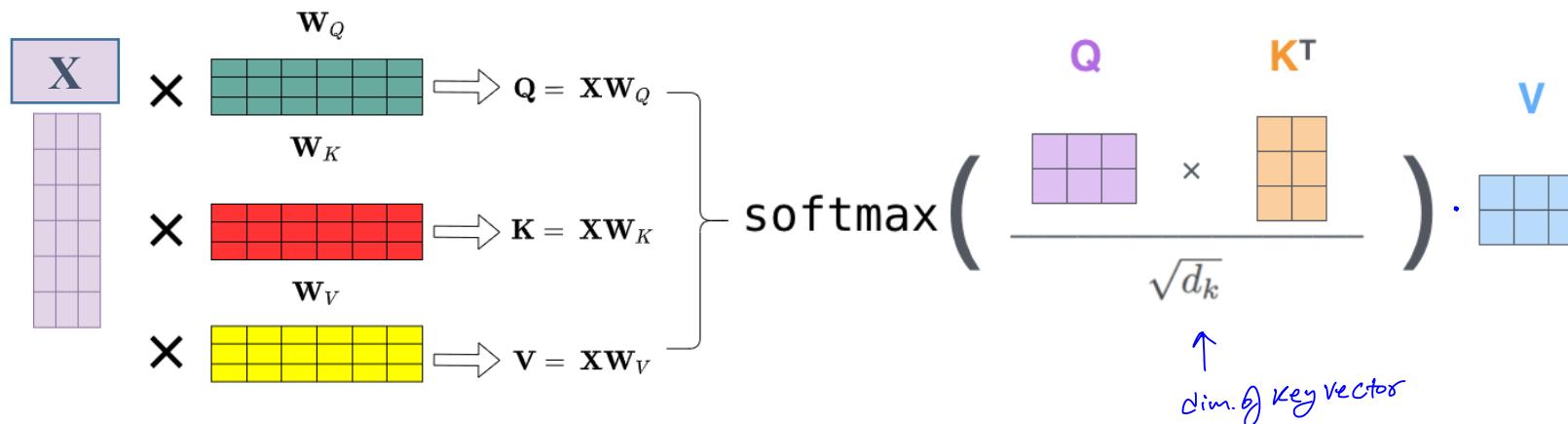
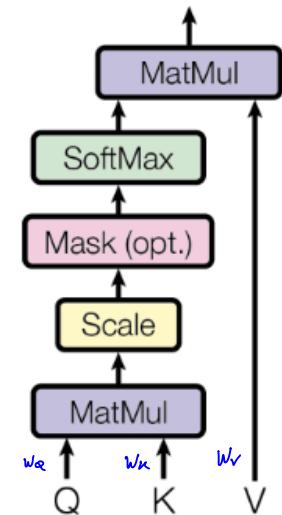
- This terminology comes from **search engines**
- **Query:** What we are looking for?
- **Value:** Body of knowledge that we are trying to extract information from (database)
- **Key:** Set of “keywords” that describes the value in a format that can be readily compared to a query.
- The “query” is **compared** to a set of “keys,” and the match scores are used to **rank** “values” (images in this example).

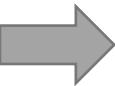




# Scaled dot product attention

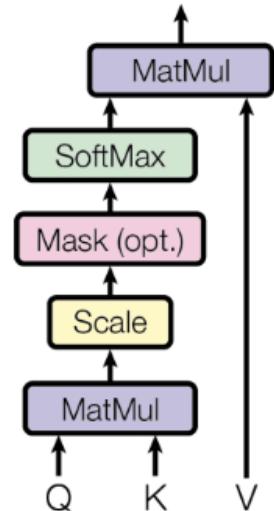
- Goal: Identify which part to attend to and Extracting features with high attention
- Scaled dot product attention utilizes three weight matrices, referred to as  $W_Q$ ,  $W_K$ , and  $W_V$  which are **learned** as model parameters during training.
- These matrices serve to project the inputs into query, key, and value components of the sequence.

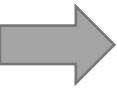




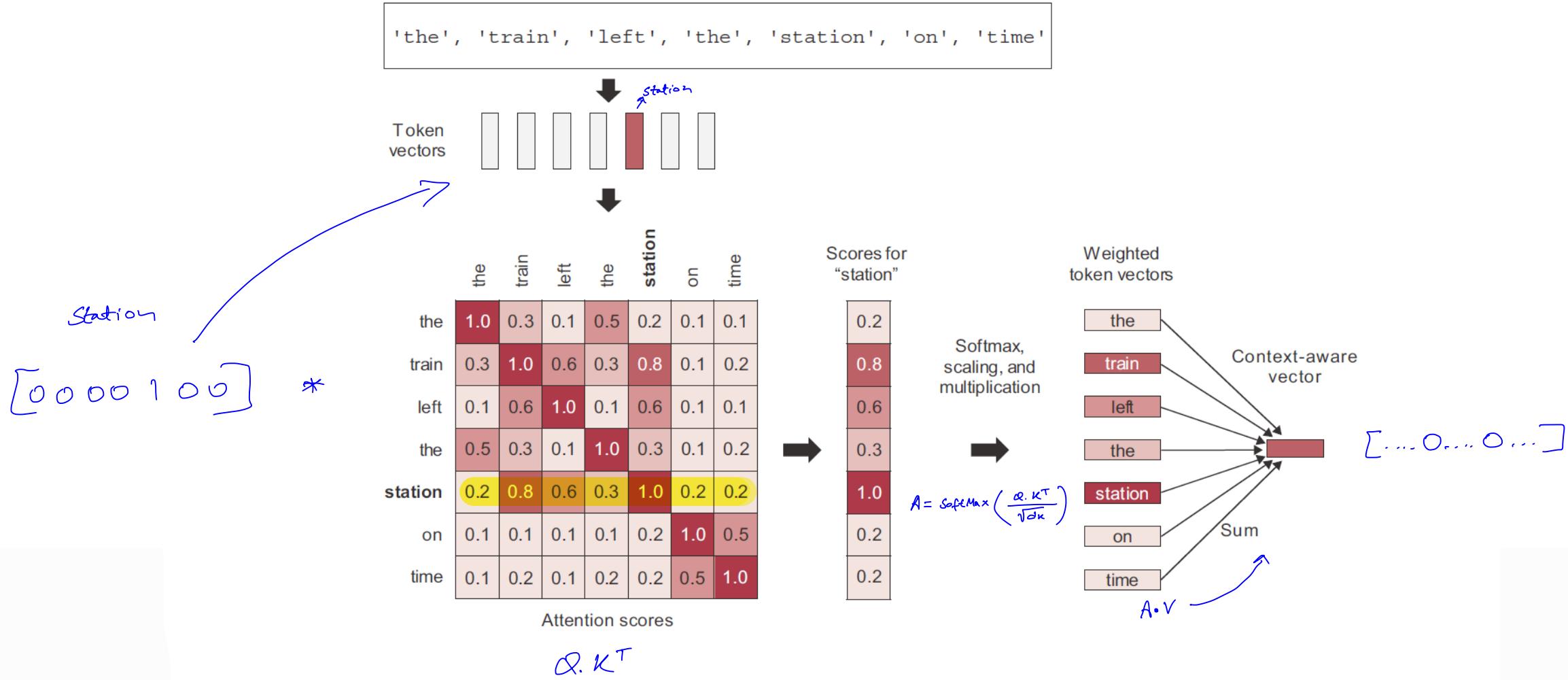
# Self-Attention in Language Models

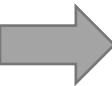
1. The input embeddings are transformed into **query**, **key**, and **value** vectors using different **learned** weight matrices. Each of these vectors is used for different purposes:
  - **Query (Q)**: Represents the current token that the model is focusing on.
  - **Key (K)**: Represents all the tokens in the input sequence, which are compared to the query to compute **attention scores**.
  - **Value (V)**: Represents all the tokens in the input sequence, which are weighted by the attention scores to create the **context vector**.
2. **Attention scores** =  $softmax\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)$
3. The attention scores are then used to weight the value vectors, and the weighted sum of these value vectors forms the **context vector** for the current word.
4. This context vector is then used to generate the output.





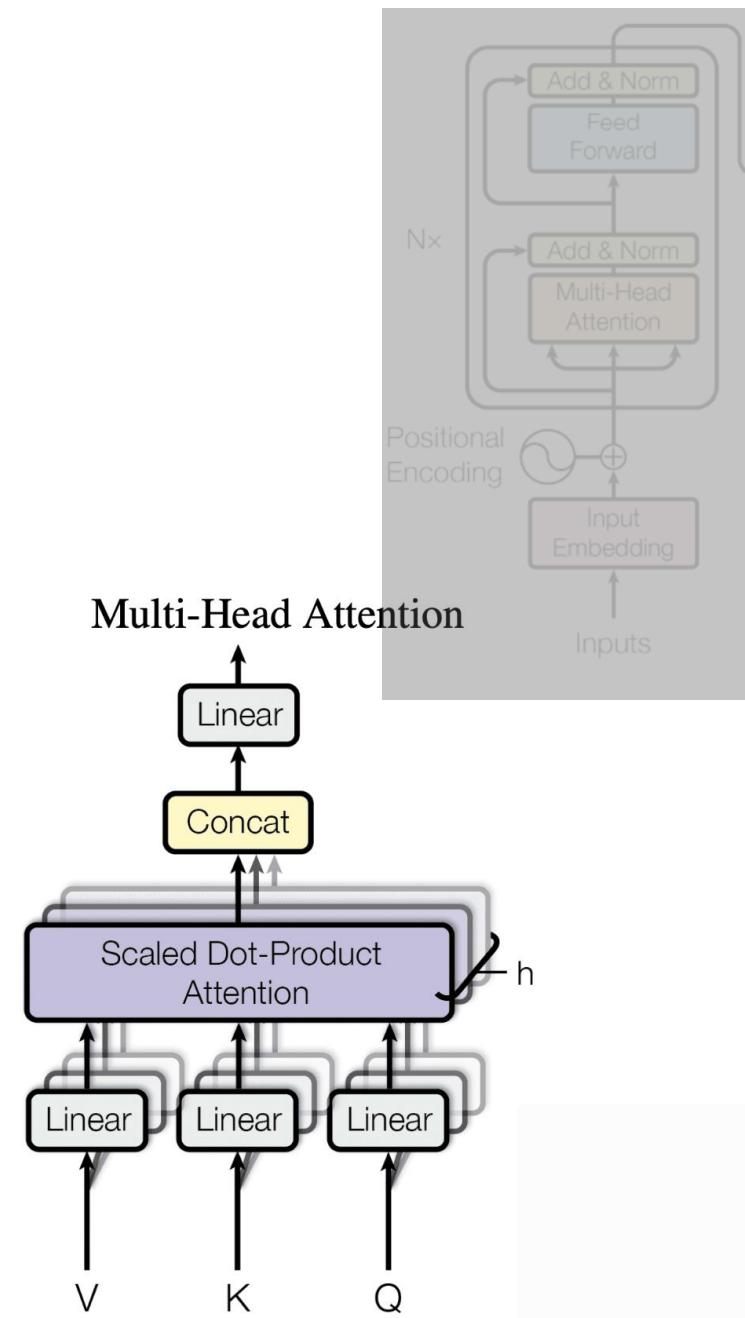
# Self-Attention in Language Models

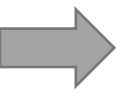




# Multi-Head Attention

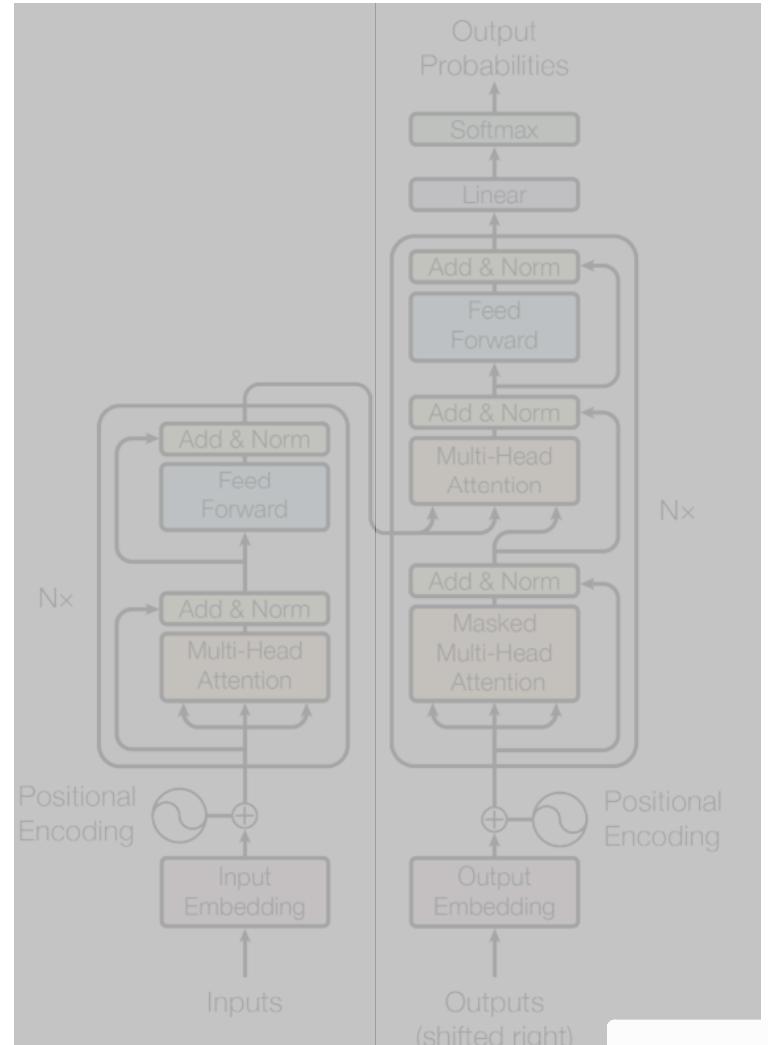
- Intuition: it allows the model to **attend to different parts** of the sequence **differently** each time
- Features **within** a self-attention head are **correlated** but mostly independent from those in other heads, similar to Depthwise separable convolutions treating each channel independently to **obtain more expressive representations**.
- Multi-head attention is simply the application of the same idea to self-attention.
- Independent heads help the layer learn different groups of features for each token.

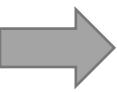




# Add & Norm

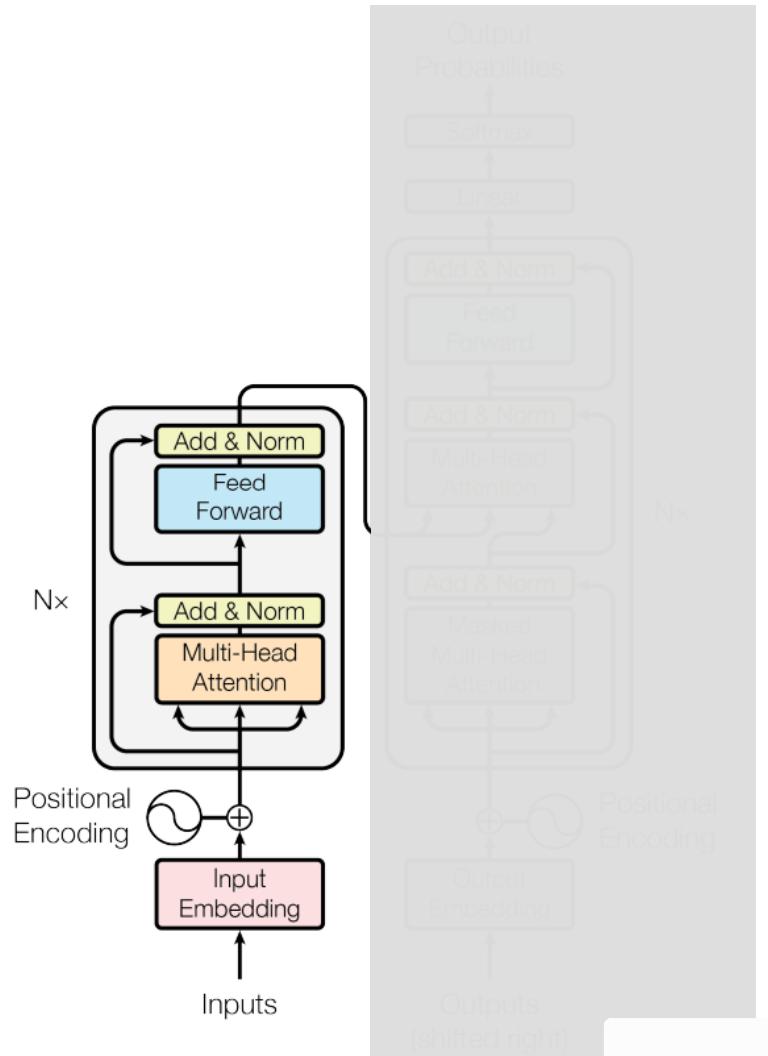
- Residual connections are added to preserve valuable information and prevent information loss during the training process.
- Normalization layers aid gradient flow during backpropagation to improve the training process.
  - Layer Normalization instead of Batch Normalization
  - Normalizing each sequence independently
- Leveraging **standard architectural** patterns such as factoring outputs into multiple independent spaces, adding residual connections, and incorporating normalization layers can enhance the performance of complex models.





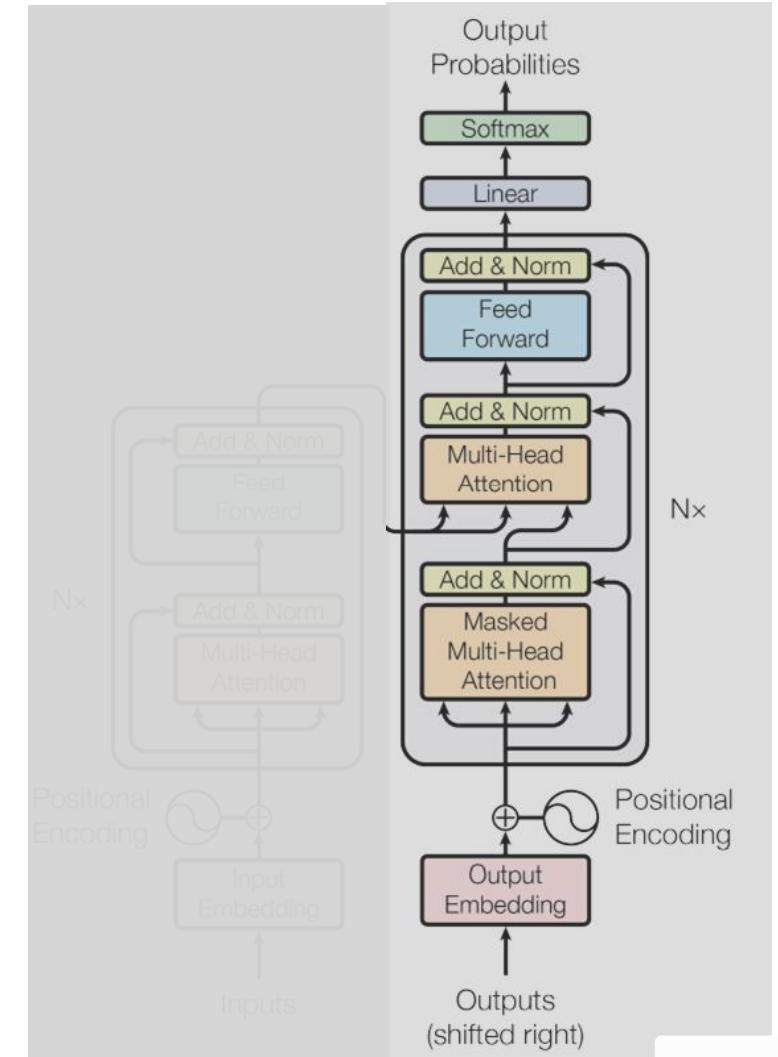
# Encoder summary

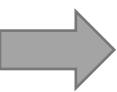
- Input embedding
- Positional encoding
- Encoder block:
  - Multi-head self-attention layer
  - Layer normalization
  - Residual connection
  - MLP (2 linear layers + RELU activation)
  - Second Layer normalization
  - Second residual connection
- Replicating N times (N=6 in the original paper)



# Transformer architecture

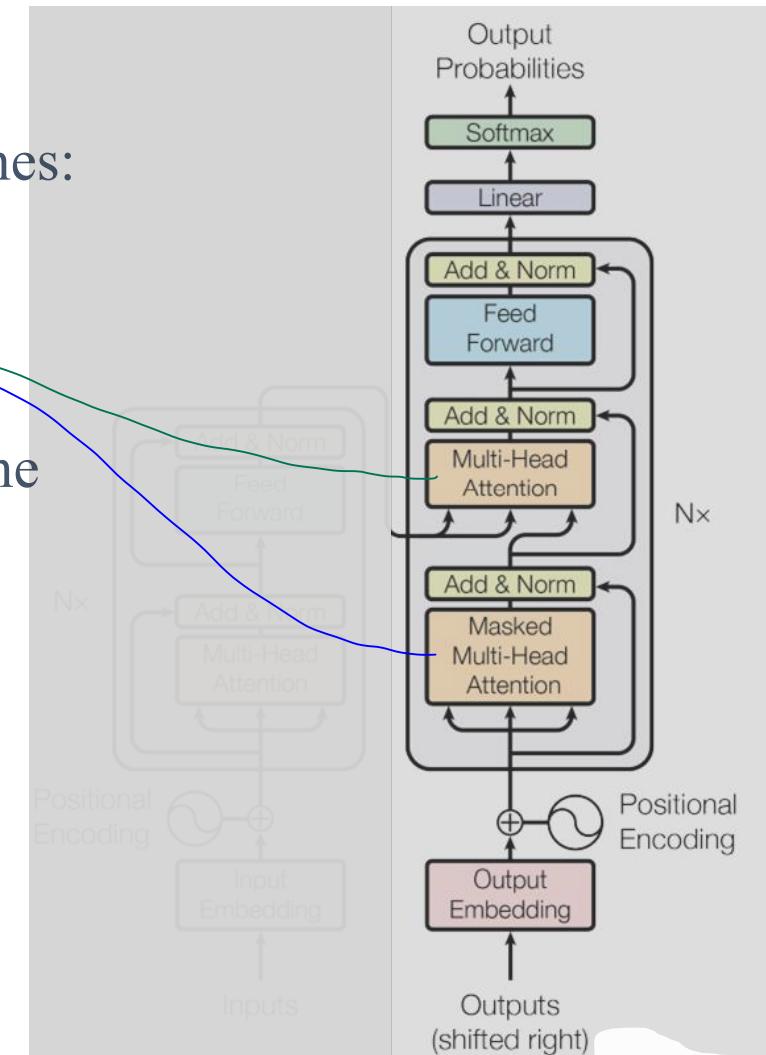
## Decoder

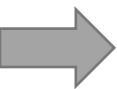




# Decoder

- Decoder consist all the components of encoder plus two novel ones:
  - Masked multi-head self-attention layer
  - Encoder-Decoder Attention layer
- Final decoder block output is passed through a linear layer and the probabilities are calculated with a standard softmax function
- Decoder is autoregressive:
  - Generate output one at a time
  - This is repeated until the <end> token is seen.

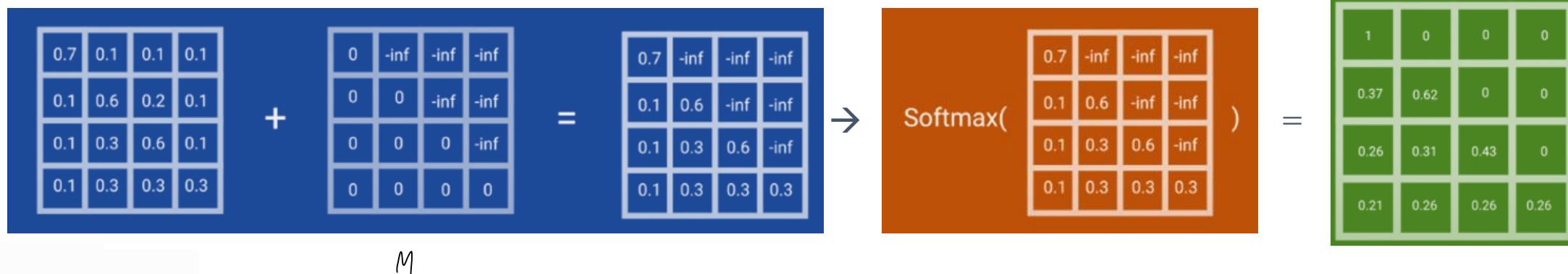
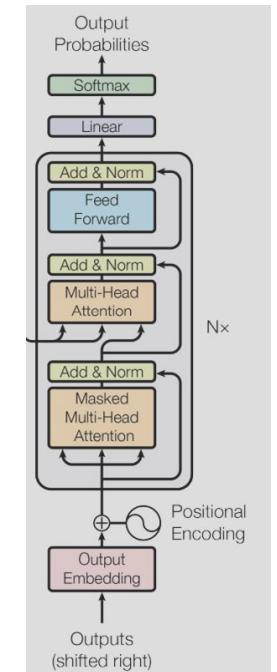


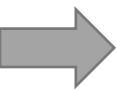


# Masked multi-head self-attention layer

- First Multi-headed attention computes the attention scores for the decoders input.
- For this Multi-headed attention layer, we need to apply **masking** to avoid cheating.

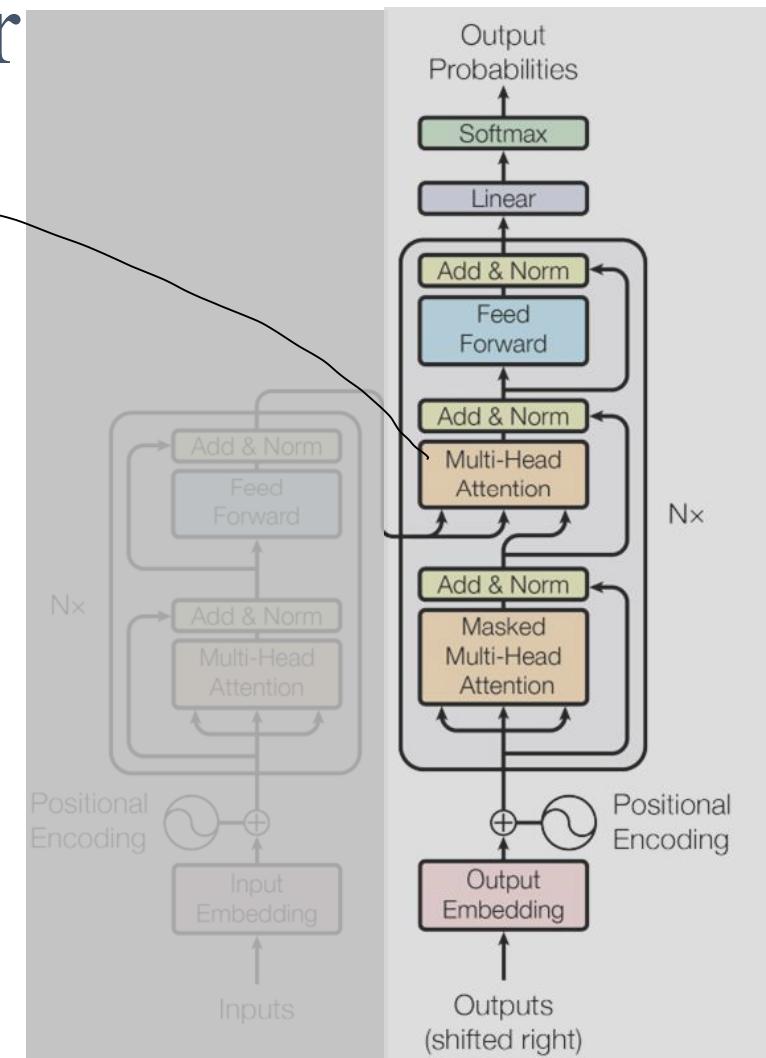
$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left( \frac{\mathbf{Q}\mathbf{K}^T + \mathbf{M}}{\sqrt{d_k}} \right) \mathbf{V}$$

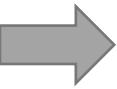




# Encoder-Decoder Attention Layer

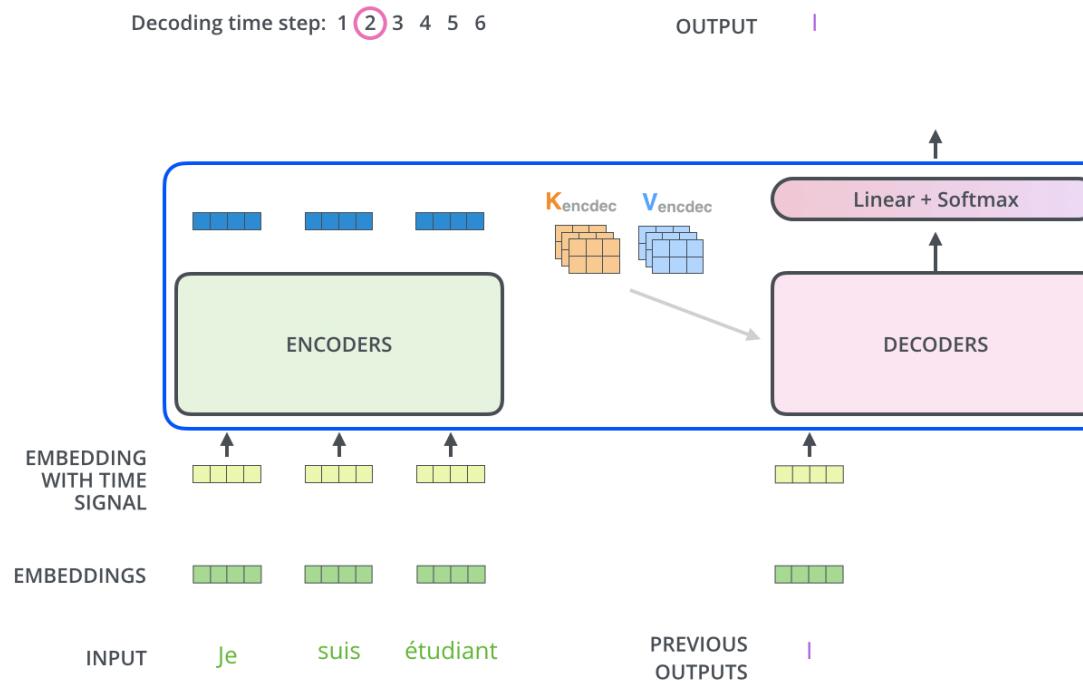
- Transformers also use a form of attention known as "encoder-decoder attention" in the **decoder** layers.
- This is **where the magic happens**, where the decoder processes the encoded representation (K,V)
- This attention mechanism allows the decoder to focus on specific parts of the **input sequence encoded by the encoder** when generating each output element.
- The encoder-decoder attention mechanism helps the model to **align the input and output sequences better**, which is particularly useful in tasks like machine translation.

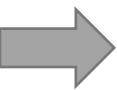




# Encoder-Decoder Attention Layer (cross-attention)

- The encoder output is utilized to generate the **Key** and **Value** matrices.
- On the other hand, the Masked Multi-head attention block's output contains the newly generated sentence, represented as the **Query** matrix in the attention layer.
- This process matched the encoders input to the decoders input allowing the decoder to decide which encoder input is relevant to focus on.





# Training

- Like any other deep learning model, training involves:
  - Feed forward, loss computation, backpropagation, parameter updates
  - Cross-entropy compare two probability distributions.

Target Model Outputs

Output Vocabulary: a am I thanks student <eos>

position #1	0.0	0.0	1.0	0.0	0.0	0.0
position #2	0.0	1.0	0.0	0.0	0.0	0.0
position #3	1.0	0.0	0.0	0.0	0.0	0.0
position #4	0.0	0.0	0.0	0.0	1.0	0.0
position #5	0.0	0.0	0.0	0.0	0.0	1.0

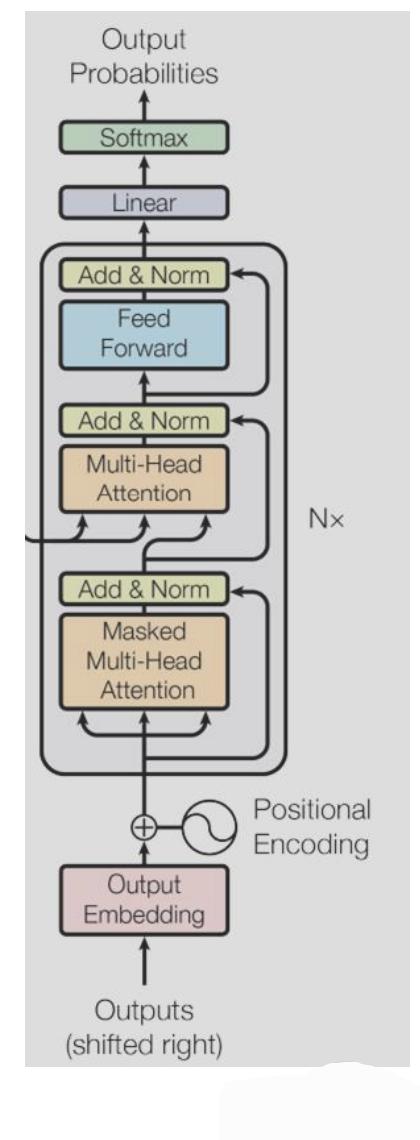
a am I thanks student <eos>

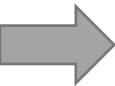
Trained Model Outputs

Output Vocabulary: a am I thanks student <eos>

position #1	0.01	0.02	0.93	0.01	0.03	0.01
position #2	0.01	0.8	0.1	0.05	0.01	0.03
position #3	0.99	0.001	0.001	0.001	0.002	0.001
position #4	0.001	0.002	0.001	0.02	0.94	0.01
position #5	0.01	0.01	0.001	0.001	0.001	0.98

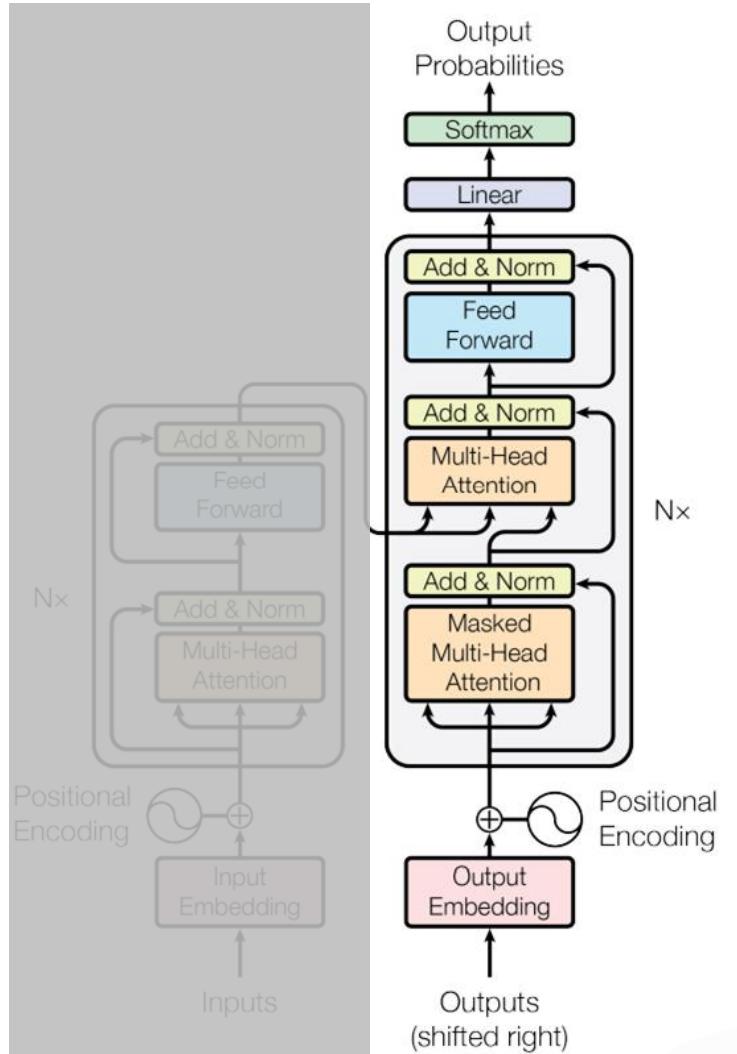
a am I thanks student <eos>

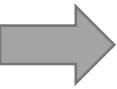




# Decoder summary

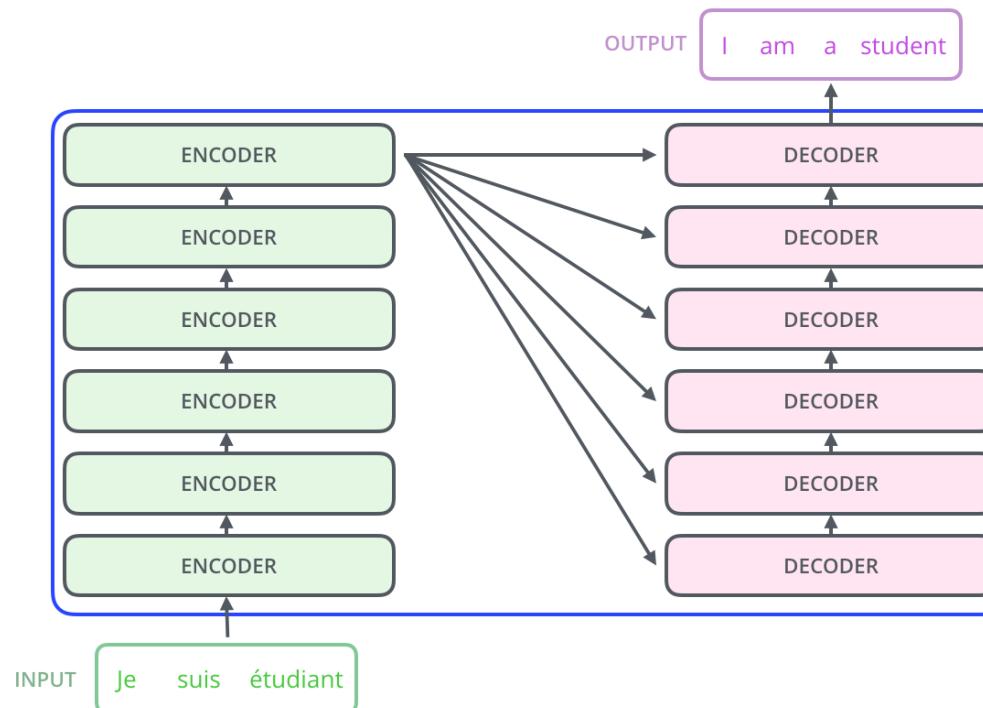
- Input embedding
- Positional encoding
- Decoder block:
  - Masked Multi-head self-attention layer
  - First Normalization and residual connection
  - Encoder-Decoder Multi-head attention
  - Second Normalization and residual connection
  - MLP (2 linear layers + RELU activation)
  - Third Normalization and residual connection
- Linear layer followed by a softmax function
- Replicating N times (N=6 in the original paper)

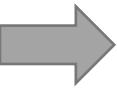




# Why transformers work?

- Multi-head attention (multiple representations of the same input)
- Context awareness through self-attention (data-dependent dynamic weights)
- Stacking multiple encoder and decoder blocks (transformer blocks are **shape-invariant**)

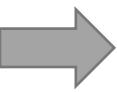




# References

---

- Attention is all you need! *Vaswani et al*
- Deep learning with Python, *Francois Chollet*
- How Transformers work in deep learning and NLP: an intuitive introduction, *Nikolas Adaloglou*
- Transformers from scratch, *Brandon Rohrer*
- The illustrated transformers, *Jay Alammar*



# Road map!

- ✓ Module 1- Introduction to Deep Learning
- ✓ Module 2- Setting up Deep Learning Environment
- ✓ Module 3- Machine Learning review (ML fundamentals + models)
- ✓ Module 4- Deep Neural Networks (NN and DNN)
- ✓ Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- ✓ Module 6- Deep Sequence Modeling (RNN, LSTM)
- ✓ Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)



# Module 7 – Transformers

## Modern Architectures

---

