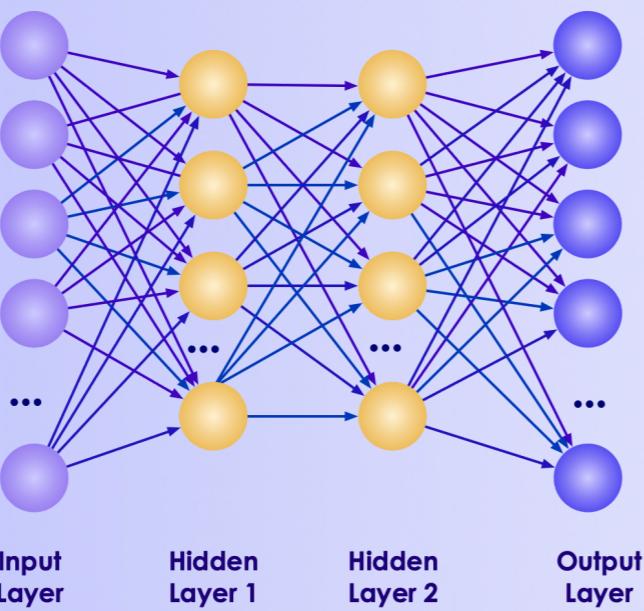


Deep Learning With

TensorFlow & Keras



About This Class

- A practical approach to Deep Learning
- Goals
 - 'Top-Down' learning
 - Learn fundamentals of DL
 - And implement them in an DL environment
- Beyond the scope
 - Deep Math / Stats coverage

Pre-requisites and Expectations

- Basic Python knowledge is assumed
 - if you are new to Python, we will provide some resources
- We don't expect Math / Statistics background
- Curiosity! Ask a lot of questions
- This is a **Intro** Deep Learning class
 - No previous knowledge is assumed
 - Class will be based on the pace of majority of the students



Data Science Totem pole

Data Scientist (PHD)

**Machine Learning
Engineer**

API users



Agenda - 3 Days

- **Day 1 (Introducing TensorFlow and Deep Learning)**

- Deep Learning Intro
- TensorFlow Intro
- TensorFlow Playground

- **Day 2 (TensorFlow / Keras)**

- Keras Intro
- Neural Network intro
- Deep Learning concepts
- Regression
- Classification

- **Day 3 (Deep Learning)**

- CNNs (Images)
- RNNs (Text, Translation)
- Transfer Learning
- Model serving
- Workshop

Our Teaching Philosophy

- Emphasis on concepts & fundamentals
- Highly interactive (questions and discussions are welcome)
- Hands-on (learn by doing)



Introductions

- About Instructor
- About you
 - Your Name
 - Your background (developer, admin, manager, ...)
 - Technologies you are familiar with
 - Familiarity with Python (scale of 1 - 4 ; 1 - new, 4 - expert)
 - Familiarity with TensorFlow / Deep Learning (scale of 1 - 4 ; 1 - new, 4 - expert)
 - Something non-technical about you!(favorite ice cream flavor / hobby...)

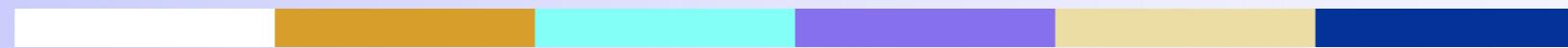


Class Logistics

- Instructor's contact
- Slides
 - For each session, slides will be emailed out or delivered via virtual classroom
- Labs
 - Lab files will be distributed
- Lab environment
 - Provided in the cloud

Let's Get Started!

Machine Learning Resources

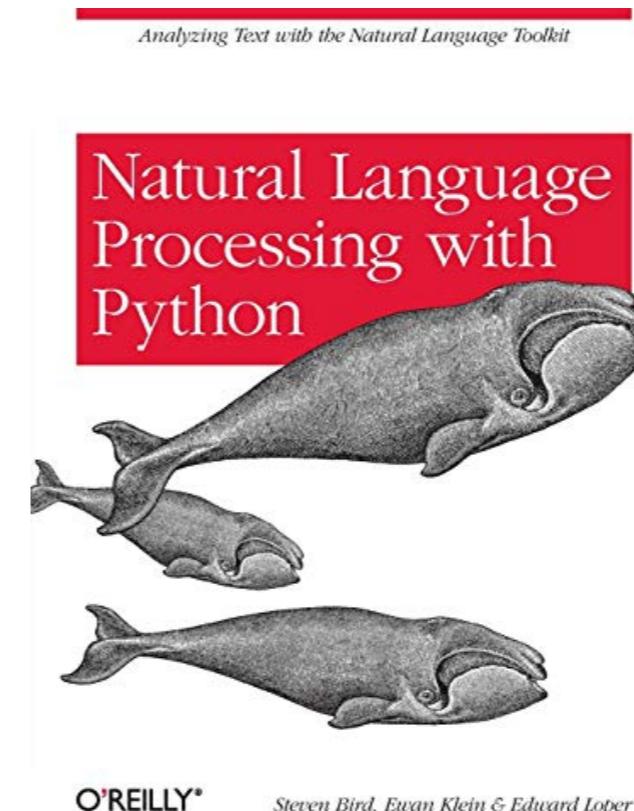
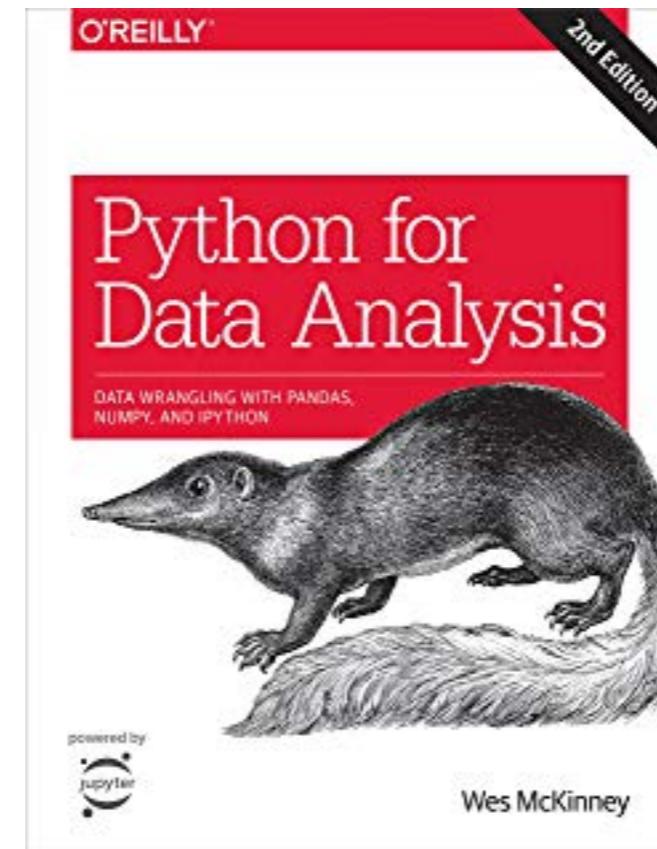


Python Language

- A Byte of Python - (FREE online book) - a very good online book introducing Python
- Dive into Python - (FREE online book)
- Learn Python the Hard Way - (FREE online book)
- Google Python Class - FREE online class
- Python Cookbook - Recipes in Python
- Python tutorials and exercises - For your practice

Python Data Science

- <https://scipy-lectures.org/>
- Best Python data science books
- Book - Python for Data Analysis
- Book - Natural Language Processing with Python



Machine Learning - Meta Links

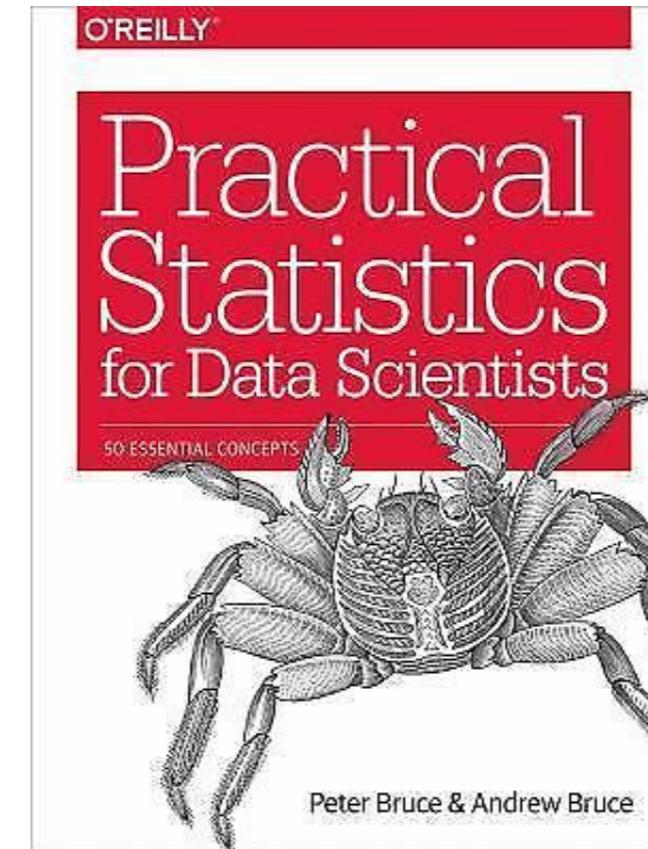
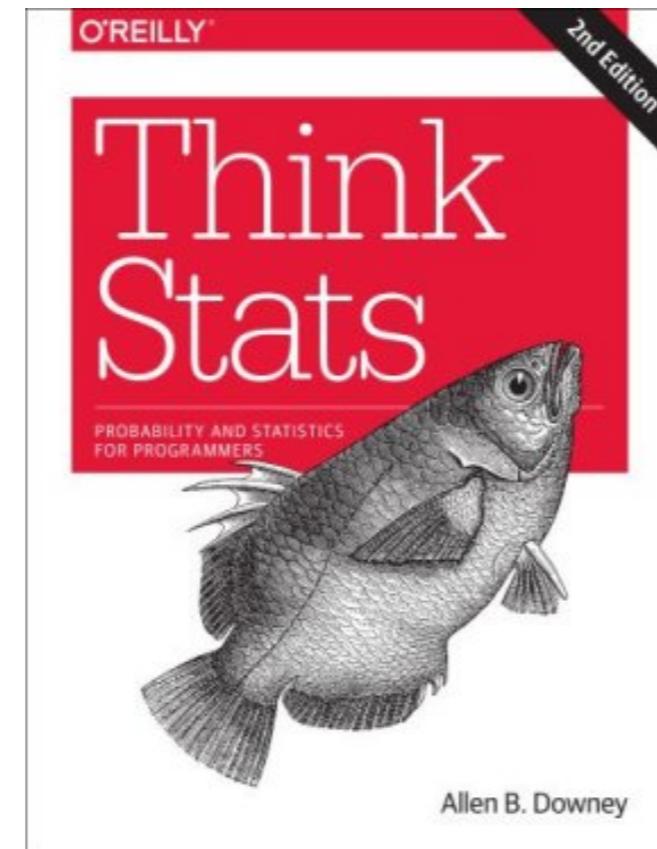
- Awesome Machine Learning
- Machine Learning for Software Engineers

Machine Learning - Gentle Introduction

- Great AI Awakening - New York Times profile of on Google Brain and the people behind it
- Gentle Intro to Machine Learning
- Machine Learning Basics

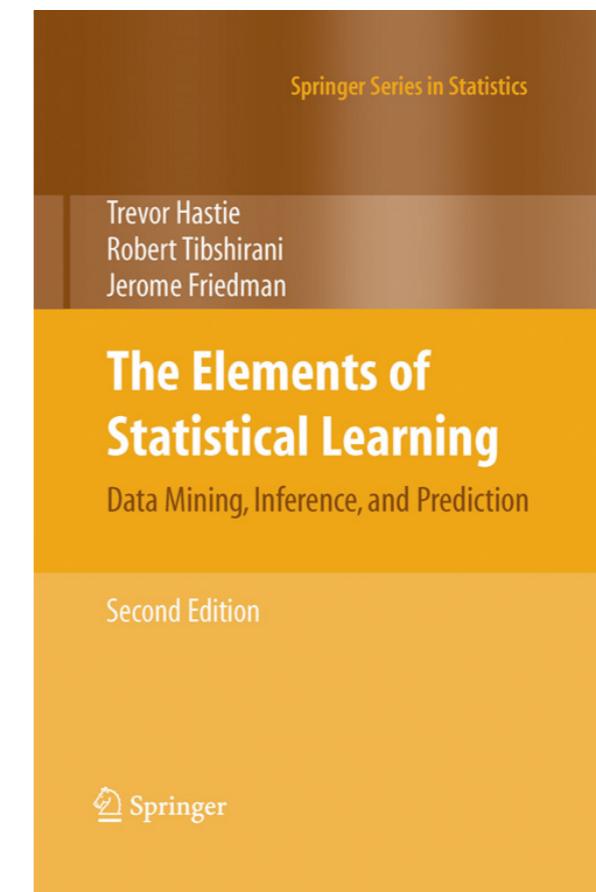
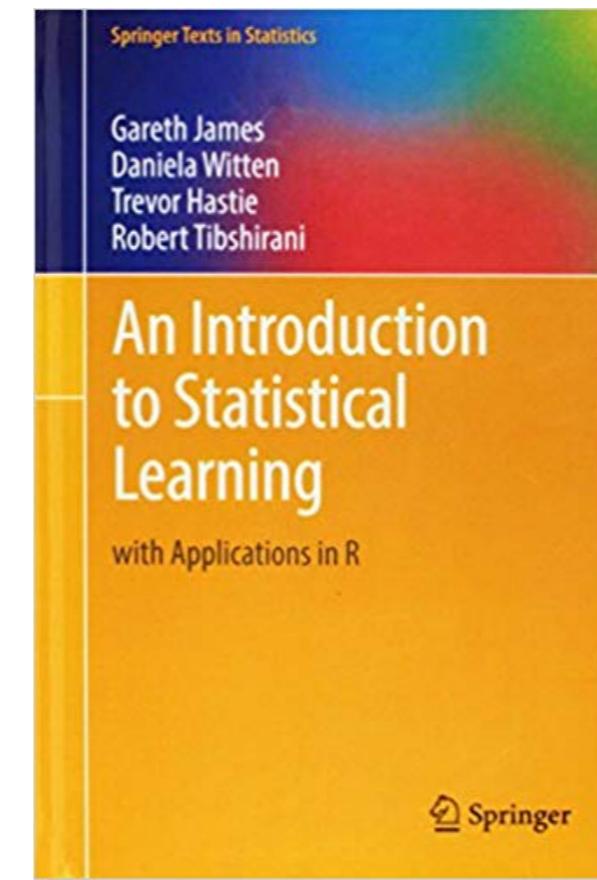
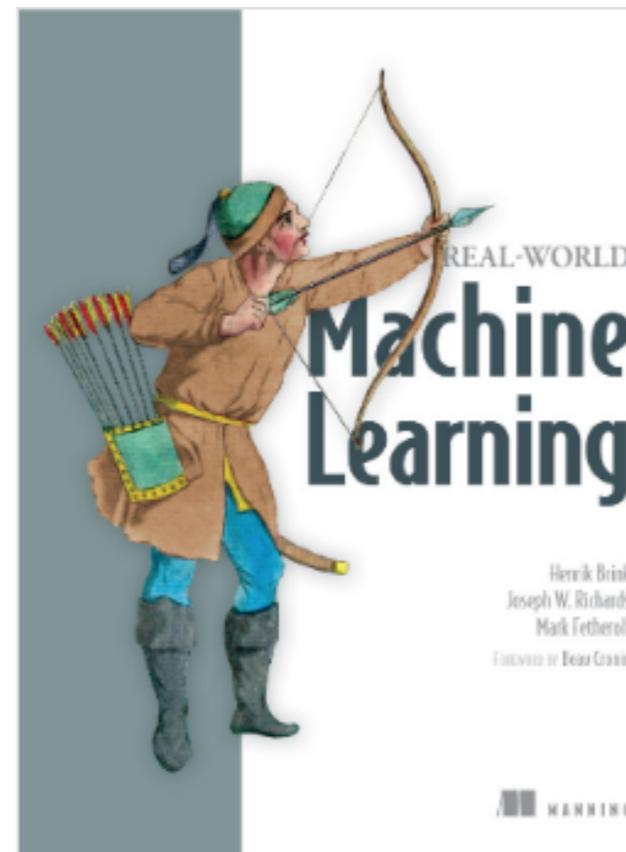
Machine Learning Basics

- Book - Think Stats
- Book - Practical Statistics for Data Scientists
- Basic Stats



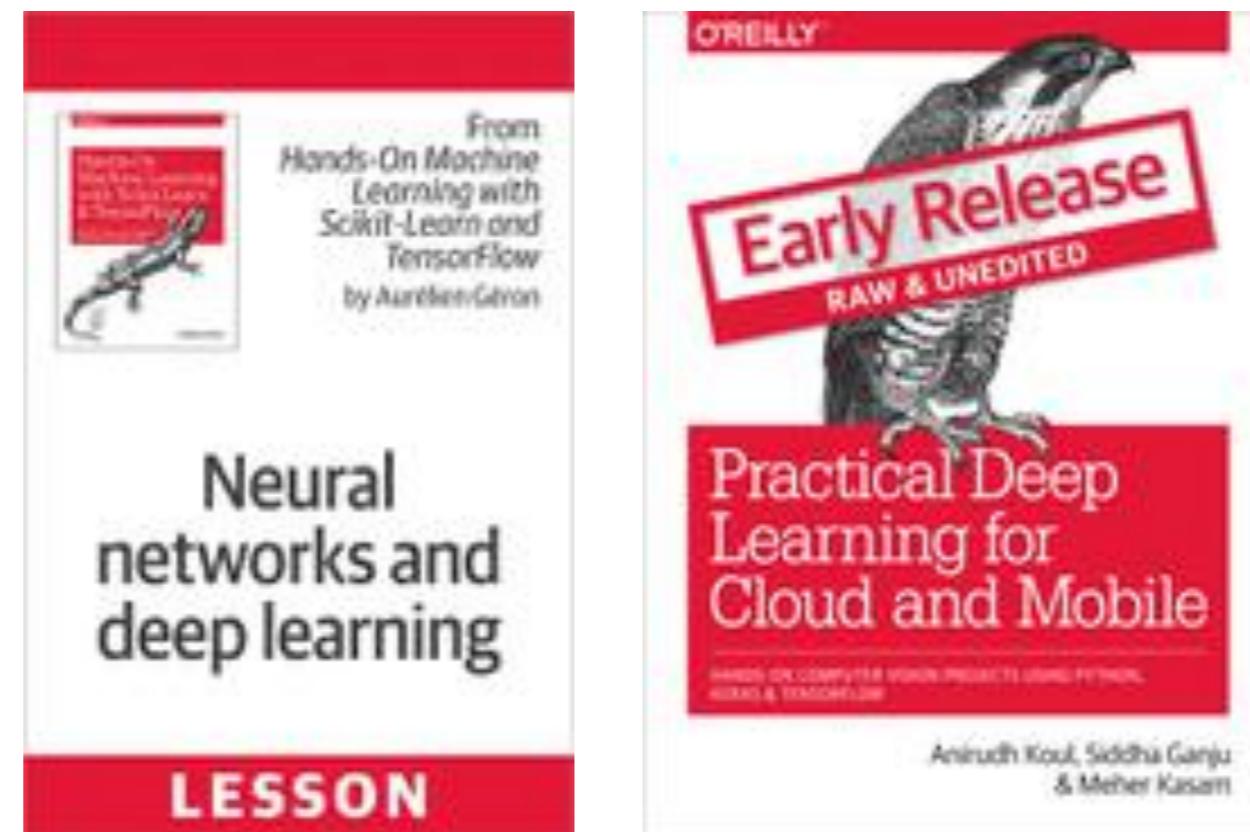
Machine Learning

- Machine Learning Mastery - Good collection of books and blog posts
- Book - Real-World Machine Learning
- Book - An Introduction to Statistical Learning - A gentle introduction to ML theory
- Book - Elements of Statistical Learning - more rigorous math treatment than above



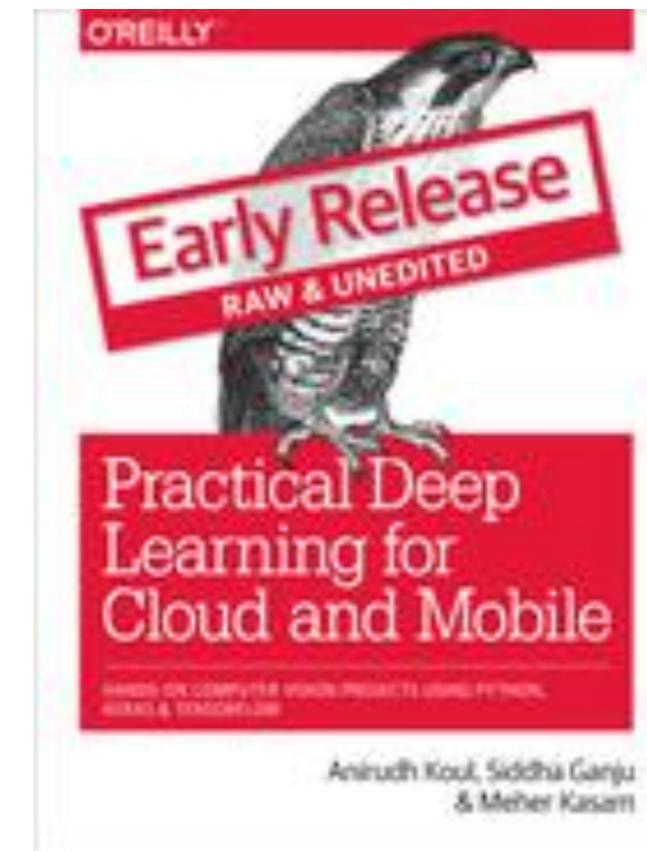
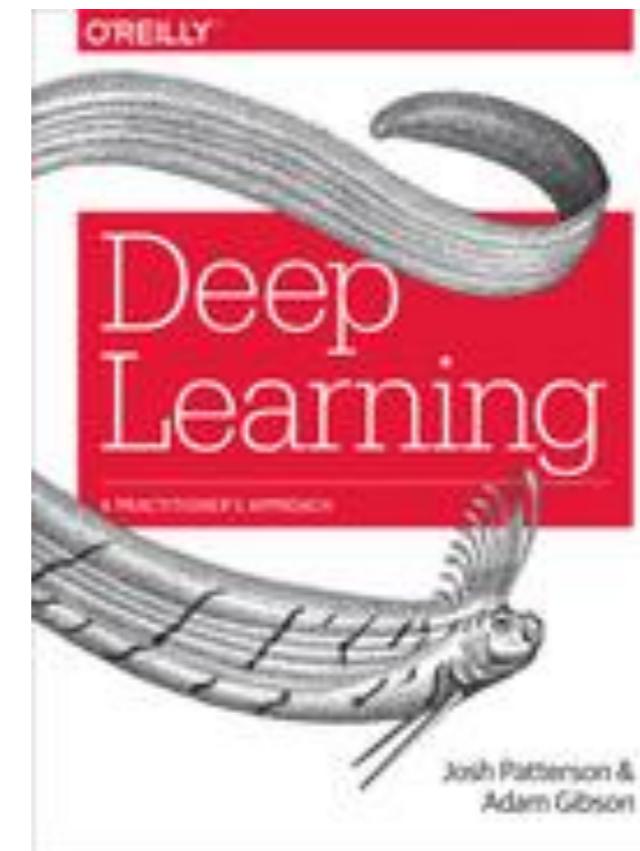
Deep Learning

- Neural networks and deep learning
by Aurélien Géron (ISBN: 9781492037347)
- Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition
by Aurélien Géron (ISBN: 9781492032649)

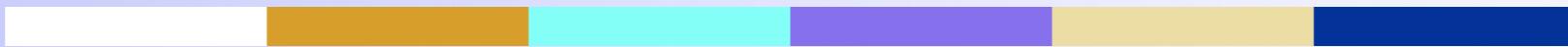


Deep Learning

- Deep Learning
by Adam Gibson, Josh Patterson (ISBN: 9781491914250)
- Practical Deep Learning for Cloud and Mobile
by Meher Kasam, Siddha Ganju, Anirudh Koul (ISBN: 9781492034841)



Machine Learning Primer

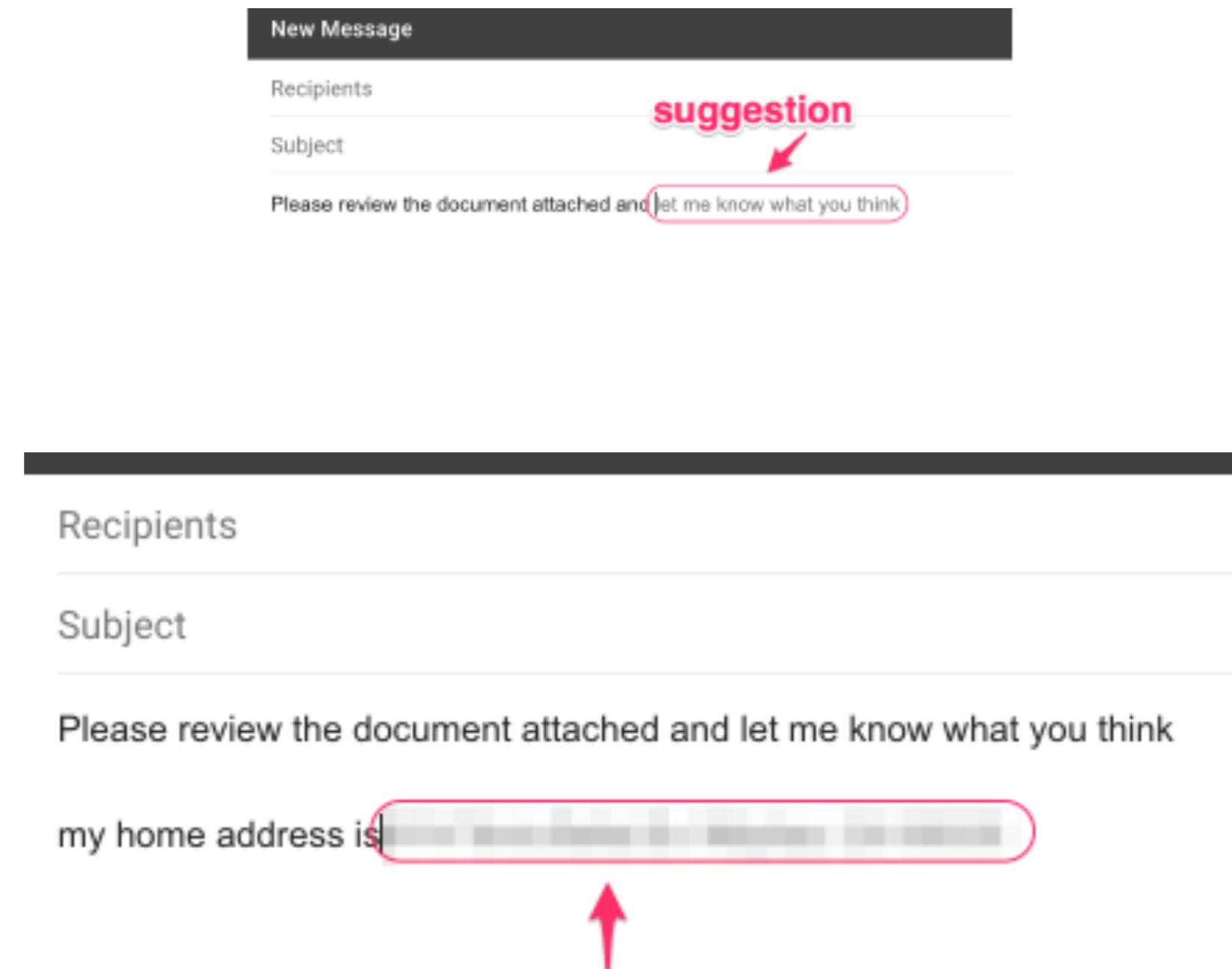


Machine Learning Evolution

Question for Audience

- Think of something you did today / this week that is AI powered?

Demo - Gmail AI Helper

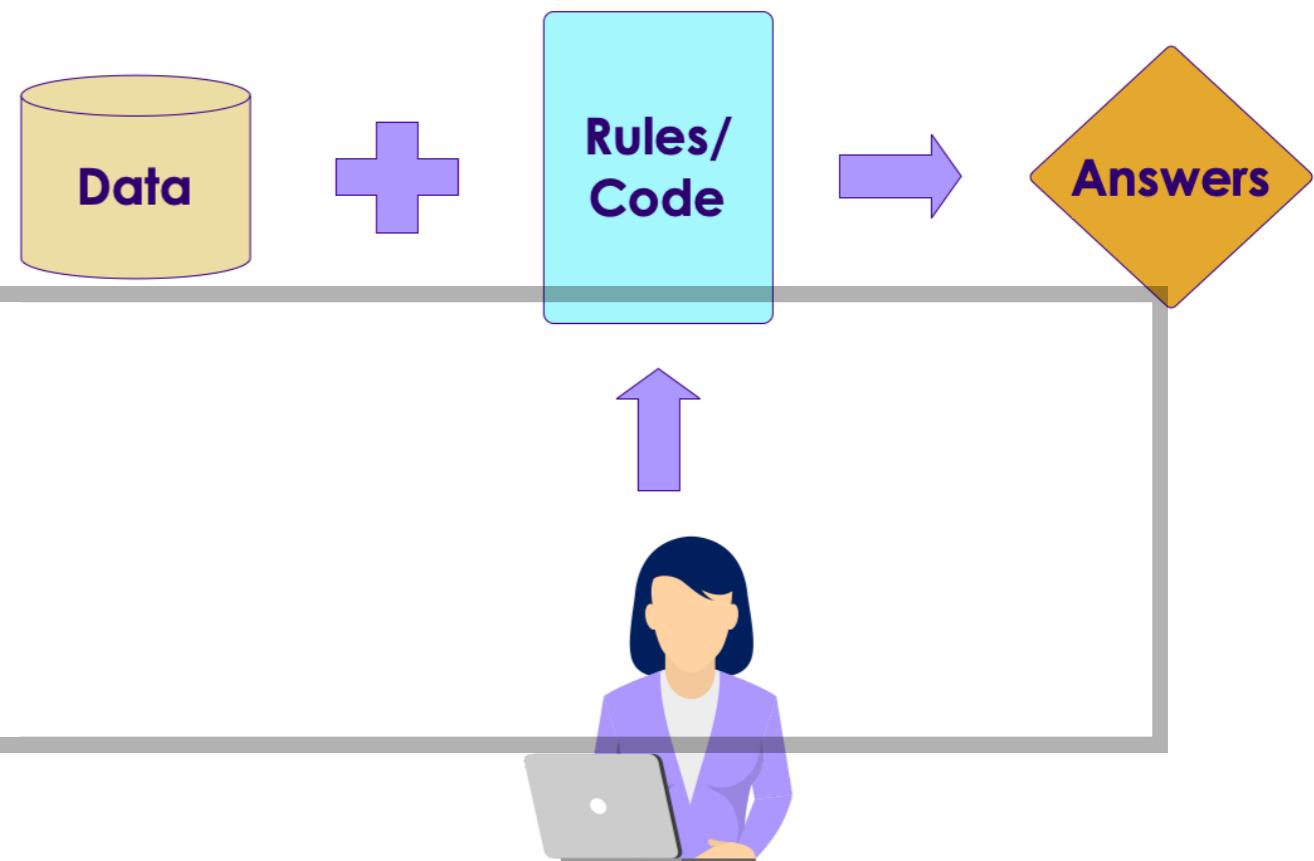


Traditional Programming vs. Machine Learning

- Here is an example of spam detection rule engine
- The rules are coded by developers
- There could be 100s of 1000s of rules!

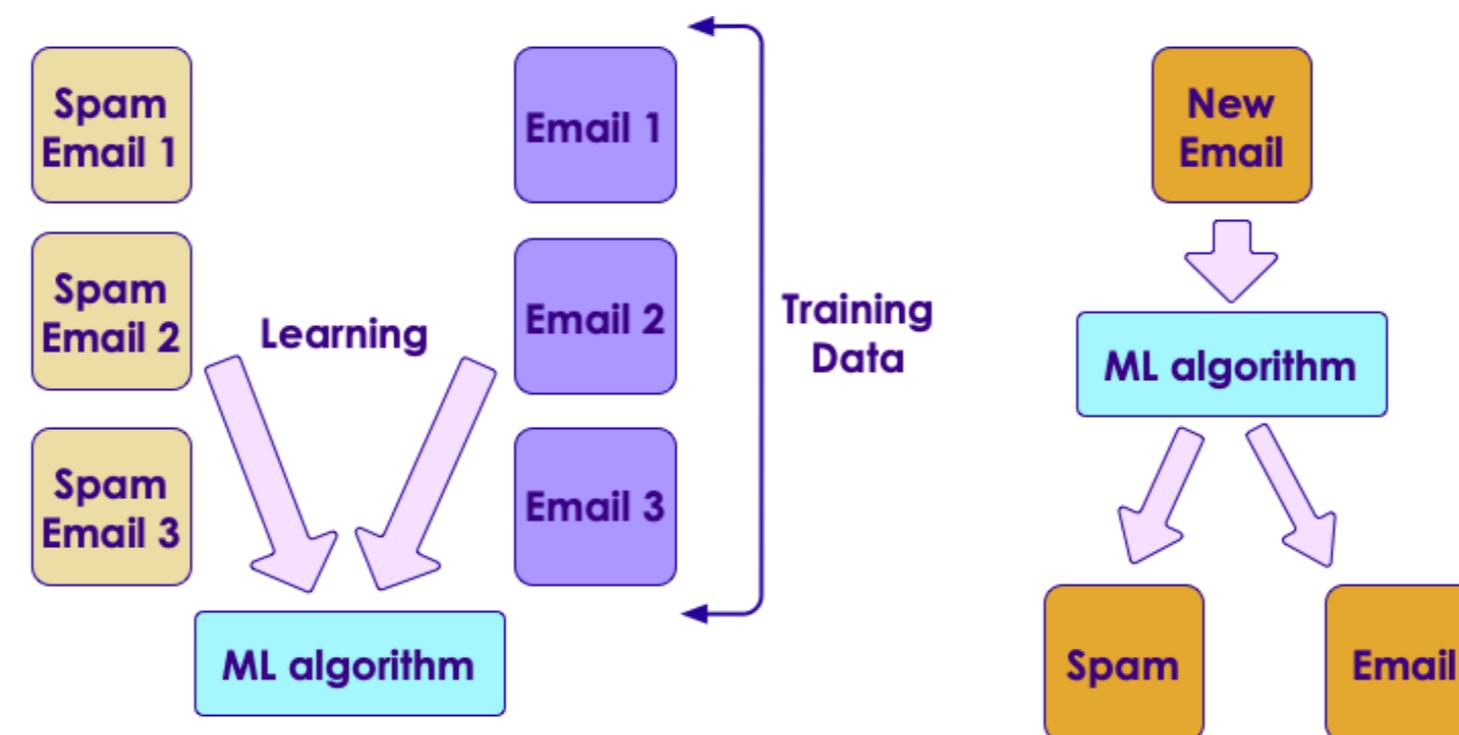
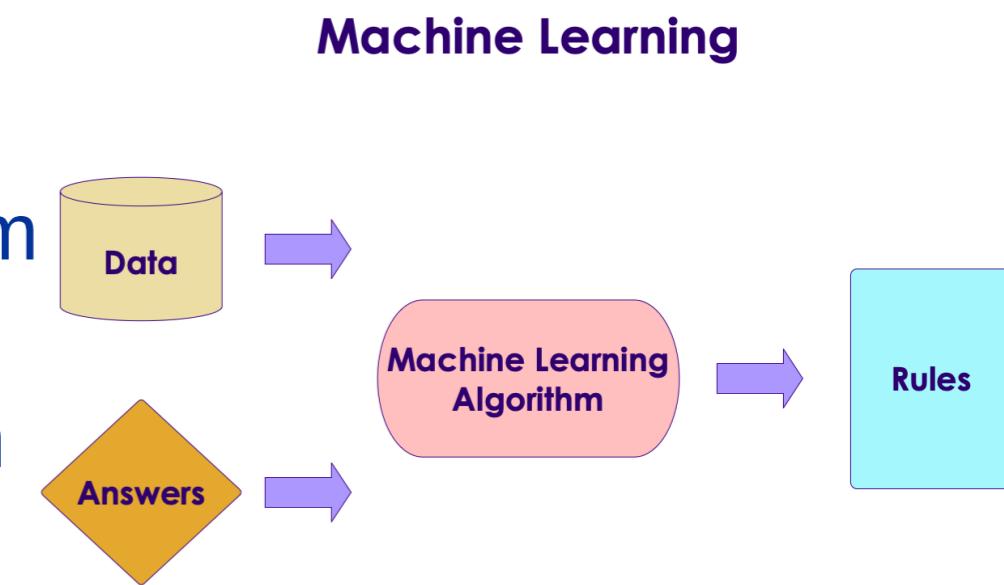
Traditional Programming

```
if (email.from_ip.one_of("ip1", "ip2", "ip3")) {  
    result = "no-spam"  
}  
else if ( email.text.contains ("free loans", "cheap degrees"))  
{  
    result = "spam"  
}
```



Traditional Programming vs. Machine Learning

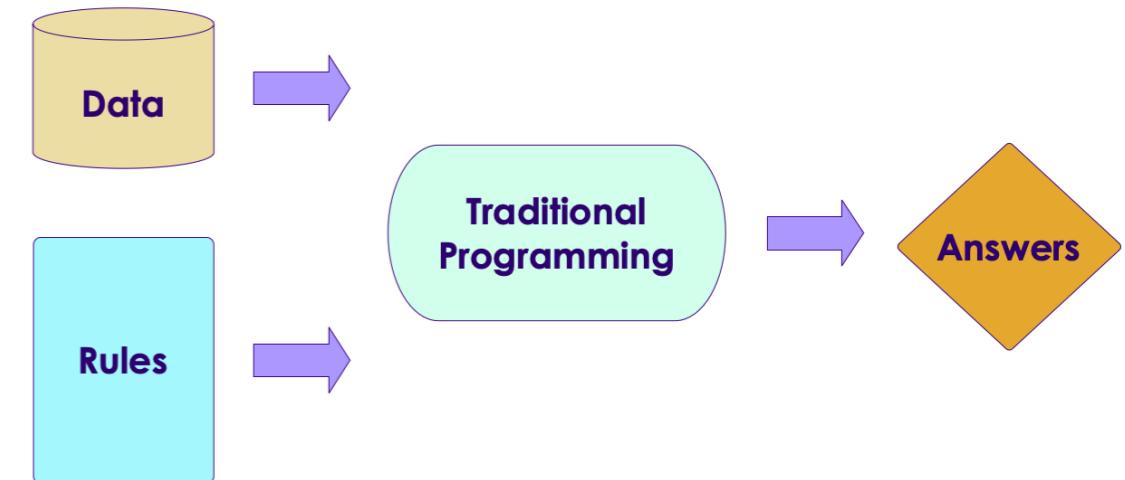
- Here is how we detect spam using ML
- We don't explicitly write rules
- Instead, we show the algorithm with spam and non-spam emails
- Algorithm 'learns' which attributes are indicative of spam
- Then algorithm predicts spam/no-spam on new email



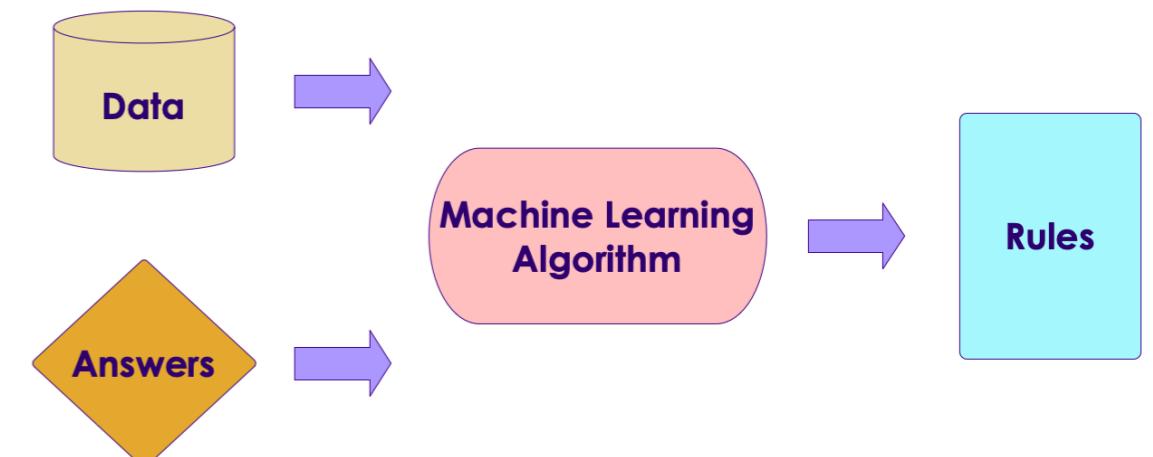
Traditional Programming vs. Machine Learning

- As data size grows so much, ability to humans to write rules to analyze all data can't keep up
- However, we can have machines analyze large amount of data and create comprehensive rules!
- These rules can be applied to provide answers to new questions

Traditional Programming



Machine Learning



Learning From Data

- Let's start with simple housing sales data

Bedrooms (input 1)	Bathrooms (input 2)	Size (input 3)	Sale Price (in thousands) (we are trying to predict)
3	1	1500	230
3	2	1800	320
5	3	2400	600
4	2	2000	500
4	3.5	2200	550

- So our formula for predicting SalePrice is something like this:
- Saleprice = f (Bedrooms, Bathrooms, Size)**
- We need to figure out what f is

Guessing Game

X	Y
1	2
2	5

- I have 2 possible formulas (there may be more)
- $Y = 3X - 1$
- $Y = X^2 + 1$



Guessing Game

X	Y
1	2
2	5
3	10
4	17

- With more data, we can finalize on a formula
- $Y = X^2 + 1$
- Lesson: More (quality) data we have, we can come up with a more precise formula
- **This is the essence of machine learning!**

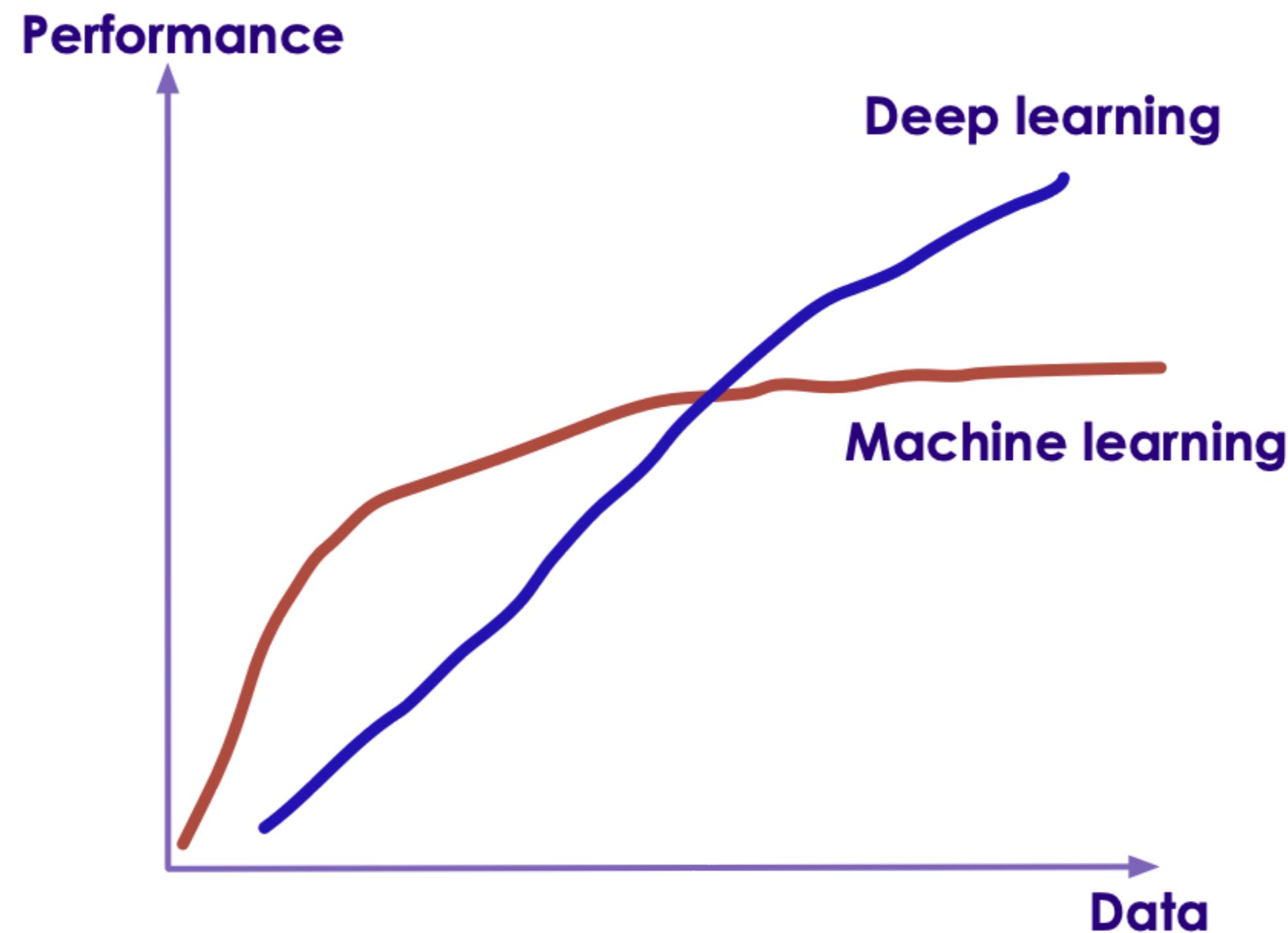


Machine Learning vs. Deep Learning

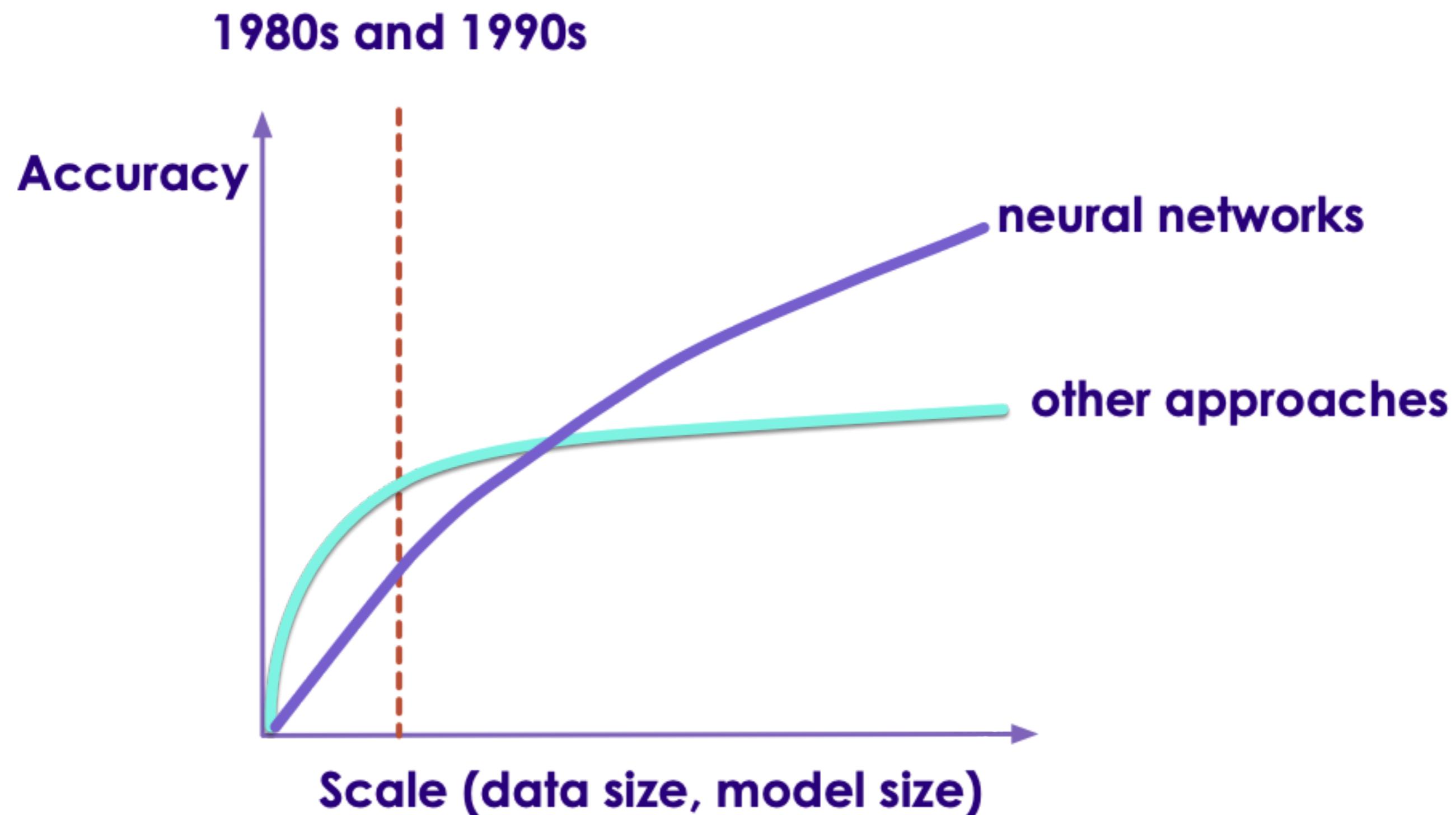
(1/3)

Features	Machine Learning	Deep Learning
==> Data size (see next slide for graph)	Performs reasonably well on small / medium data	Need large amount of data for reasonable performance
Data Type (see next slides)	Works well with structured data	Can handle structured data & unstructured data
Scaling	Doesn't scale with large amount of data	Scales well with large amount of data
Compute power	Doesn't need a lot of compute (works well on single machines)	Needs a lot of compute power (usually runs on clusters)
CPU/GPU	Mostly CPU bound	Can utilize GPU for certain computes (massive matrix operations)
Feature Engineering	Features needs to specified manually (by experts)	DL can learn high level features from data automatically
Execution Time	Training usually takes seconds, minutes, hours	Training takes lot longer (days)
Interpretability	Easy to interpret	Hard to understand the final result

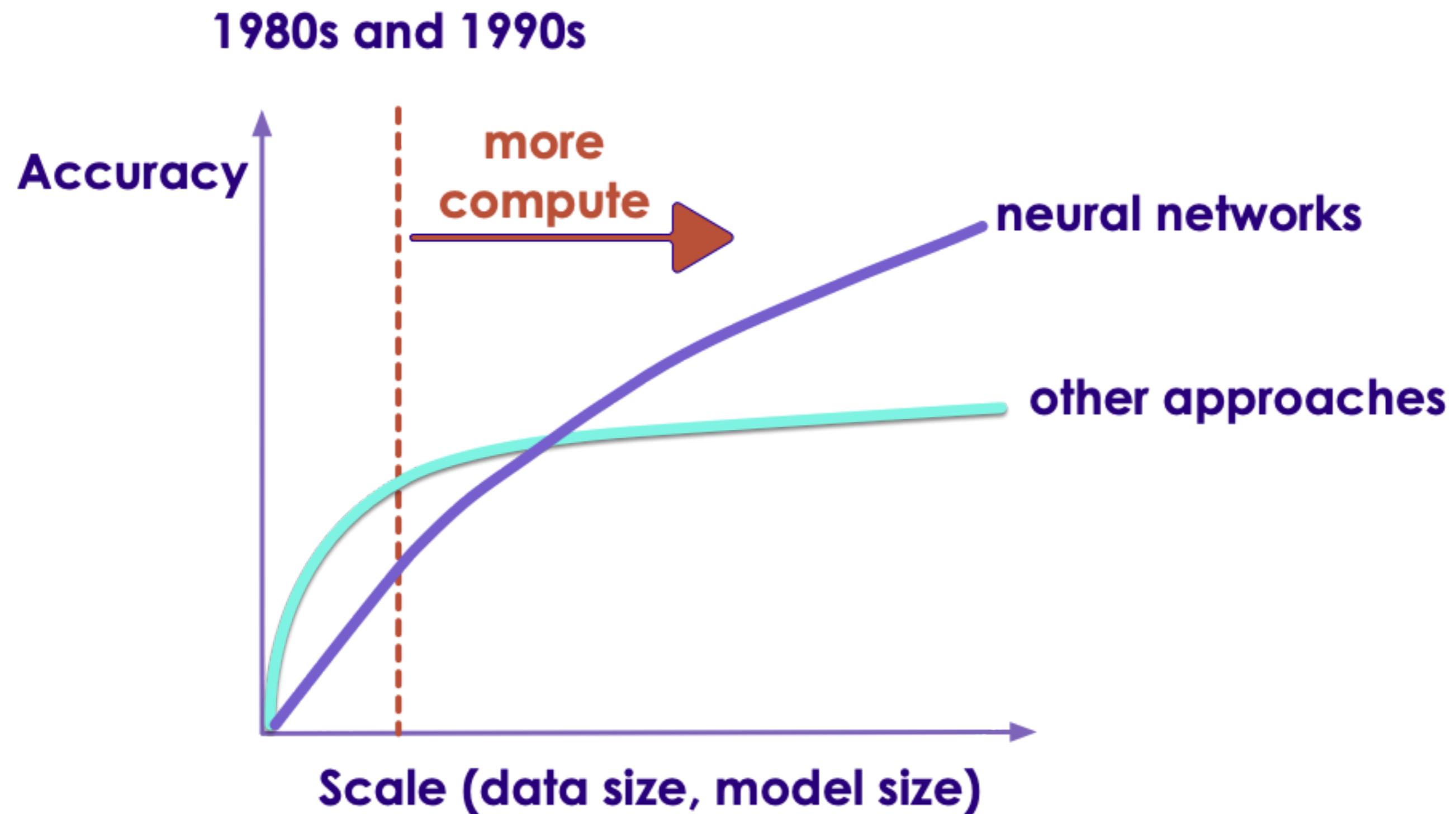
Machine Learning vs. Deep Learning



1980's and 1990's



1990+



Machine Learning vs. Deep Learning

(2/3)

Features	Machine Learning	Deep Learning
Data size (see next slide for graph)	Performs reasonably well on small / medium data	Need large amount of data for reasonable performance
==> Data Type (see next slides)	Works well with structured data	Can handle structured data & unstructured data
Scaling	Doesn't scale with large amount of data	Scales well with large amount of data
Compute power	Doesn't need a lot of compute (works well on single machines)	Needs a lot of compute power (usually runs on clusters)
CPU/GPU	Mostly CPU bound	Can utilize GPU for certain computes (massive matrix operations)
Feature Engineering	Features needs to specified manually (by experts)	DL can learn high level features from data automatically
Execution Time	Training usually takes seconds, minutes, hours	Training takes lot longer (days)
Interpretability	Easy to interpret	Hard to understand the final result

Structured Data Examples

- Pretty much any data stored in a schema database

Bedrooms	Bathrooms	Size	Sale Price (in thousands)
3	1	1500	230
3	2	1800	320
5	3	2400	600
4	2	2000	500
4	3.5	2200	550

- Text data (CSV, JSON) can have structure too

JSON data

```
{ "name" : "Joe",  
  "email" : "joe@gmail.com" }
```

CSV data (Comma Separated Values)

```
joe,joe@gmail.com  
jane,jane@gmail.com
```

Semi-Structured Data

- This is 'between' structured and unstructured
- Data has some structure, but it may not be well defined
- Example, tweet data

```
{ "user_id" : "user123",
  "timestamp" : "2018-09-20 12:00:05 EST",
  "device" : "iPhone X",
  "location" : "34.893, 120.979",
  "tweet" : "Enjoying my first Pumpkin Spice Latte at Starbucks in Seattle downtown #PSL, @Starbucks",
  "image_url" : "https://imgurl.com/1234"
}
```

Question to the class: What data points you can extract from above tweet? Which is structured / unstructured?

Machine Learning vs. Deep Learning

(3/3)

Features	Machine Learning	Deep Learning
Data size (see next slide for graph)	Performs reasonably well on small / medium data	Need large amount of data for reasonable performance
Data Type (see next slides)	Works well with structured data	Can handle structured data & unstructured data
Scaling	Doesn't scale with large amount of data	Scales well with large amount of data
Compute power	Doesn't need a lot of compute (works well on single machines)	Needs a lot of compute power (usually runs on clusters)
CPU/GPU	Mostly CPU bound	Can utilize GPU for certain computes (massive matrix operations)
Feature Engineering	Features needs to specified manually (by experts)	DL can learn high level features from data automatically
Execution Time	Training usually takes seconds, minutes, hours	Training takes lot longer (days)
Interpretability	Easy to interpret	Hard to understand the final result

Deciding Between Machine Learning(ML) and Deep Learning(DL)

- This is not an easy decision, but here are some factors to think about
- Have structured data? Then both ML and DL can be used
- Got unstructured data? Probably deep learning
- Do keep in mind, deep learning usually needs
 - lot of data
 - and lot of compute time to produce good results

Types of Machine Learning

- **Supervised Machine Learning:**

- Algorithm learns from labeled training data
- And predicts on new data

- **Unsupervised Machine Learning**

- Algorithm tries to find natural patterns in the data

- **Semi-Supervised Learning**

- Algorithm is trained with a training set which contains unlabeled (usually lot) and labeled (usually little) data

- **Reinforcement Learning**

- Based on 'game play' (rewards vs penalties)

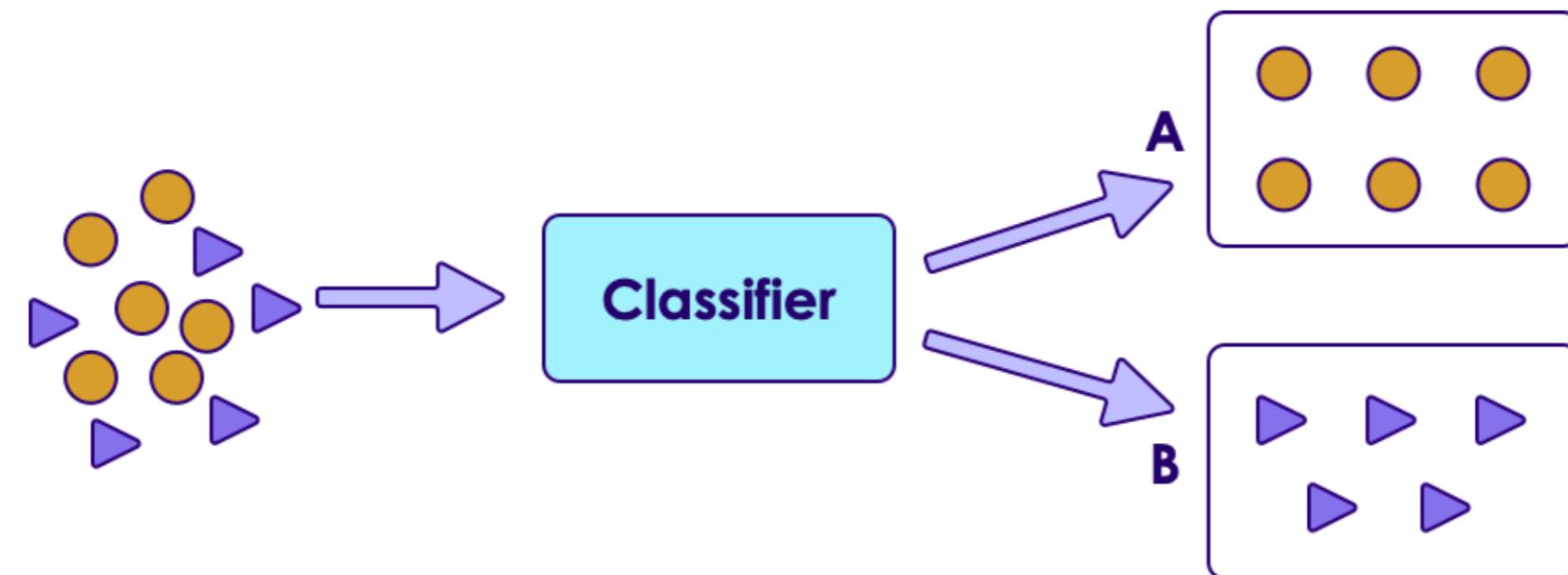
Supervised Learning Example - Regression

- Predicting stock market
- Train the model using training data (past data; already known)
- Test performance using test data (past data; already known)
- If performance is satisfactory, predict on new data (unseen)



Supervised Learning - Classification

- Classification is a model that predicts data into "buckets"
 - Email is **SPAM** or **HAM** (not-SPAM)
 - A cell is **cancerous** or **healthy**
 - Hand-written numbers -> any digits -1, 0, 1,..., 8
- Classification algorithm learns from training data (Supervised learning) and predicts on new data
- In the example below, we input mixed data to the model, and the model classifies them into A or B

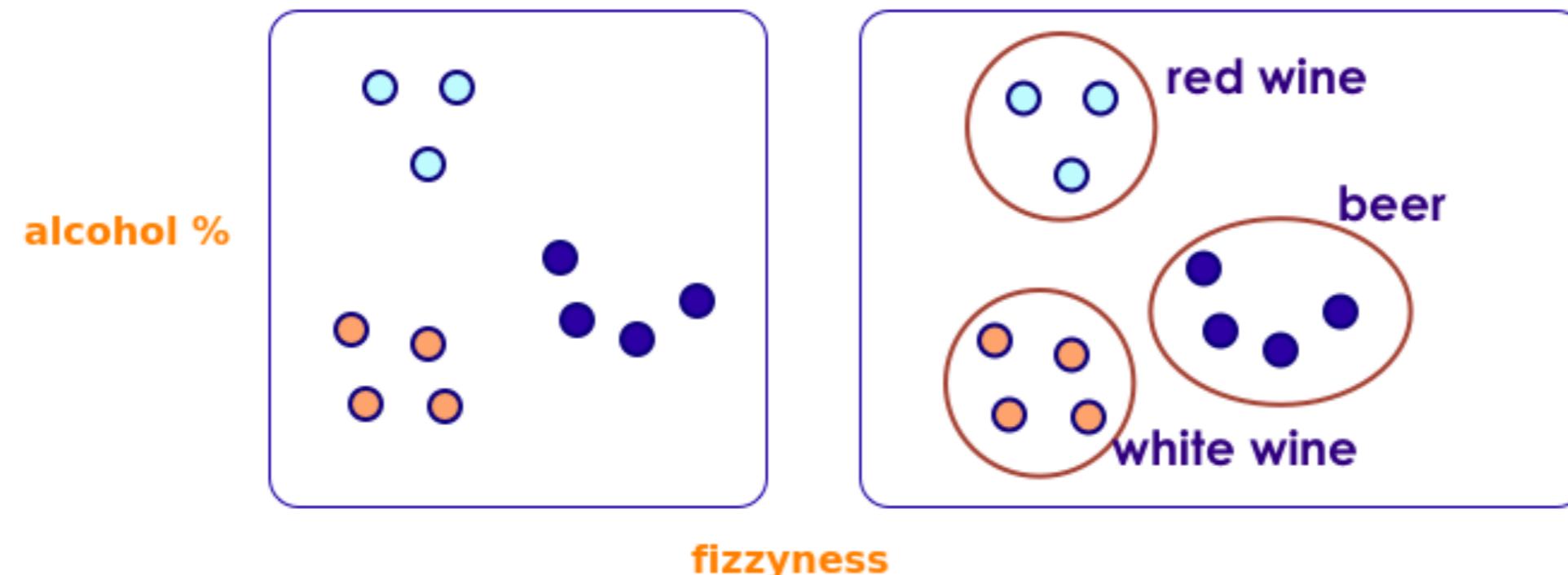


Classification Applications

- Web
 - Email is spam or not
 - Website is authentic or fraudulent
- Medicine
 - Is this cell cancerous or not?
- Finance
 - Credit card transaction fraudulent or not
- OCR
 - Recognizing characters and symbols

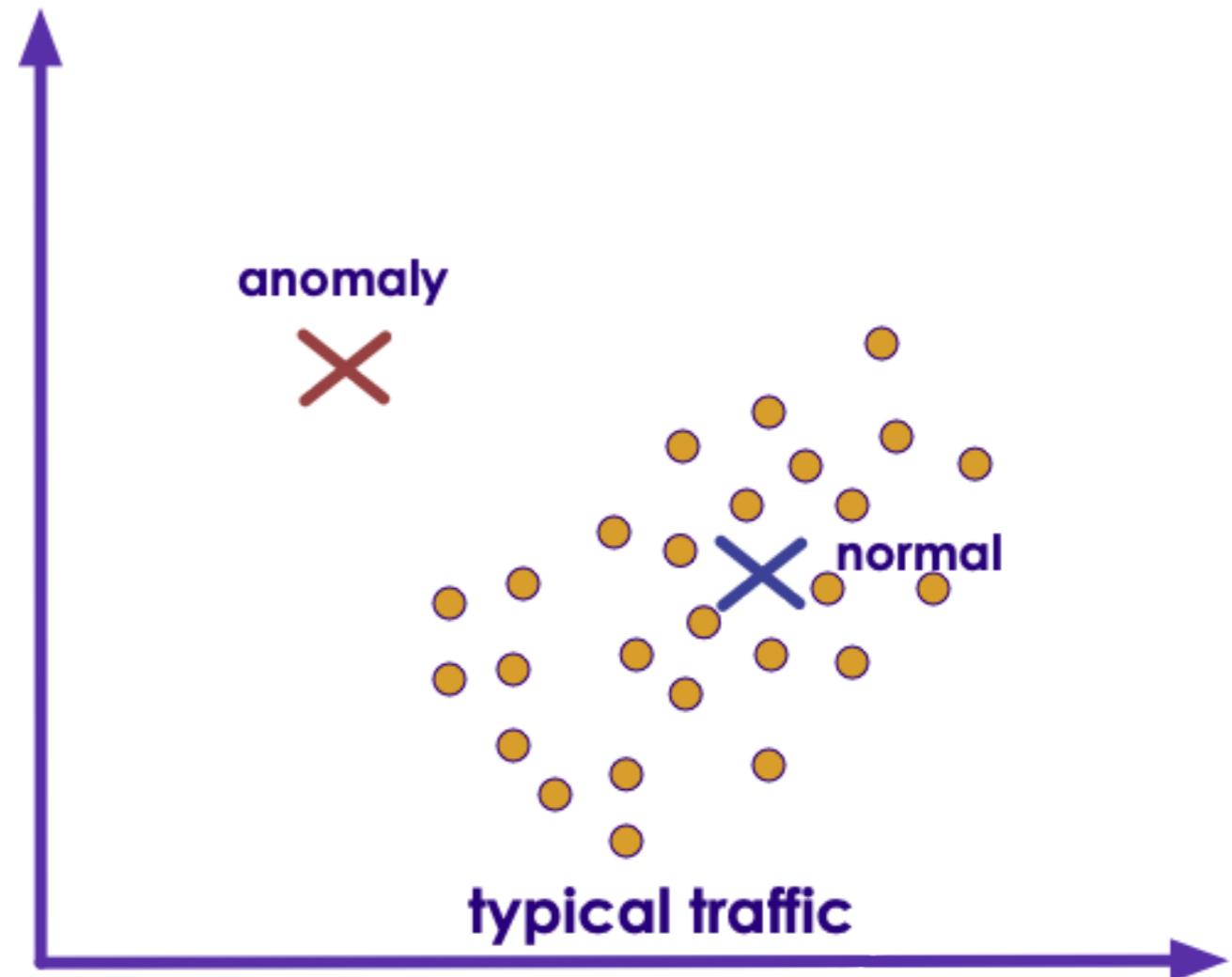
Clustering

- Clustering finds natural groupings in data
- Here we are grouping alcohol beverages according to 2 dimensions (alcohol %, fizziness); And we see similar drinks fall into natural groups
- In real world applications, we could be clustering by many dimensions (10s or 100s)



Clustering Use Cases: Fraud / Anomaly Detection

- Anomaly detection is used to:
 - Find fraud
 - Detect network intrusion attack
 - Discover problems on servers
- Here we see an anomaly (top left) that doesn't fall into the usual pattern (bottom right)



Clustering Applications

- Biology
 - Genomics grouping
- Medicine
 - Xray/CAT image analysis
- Marketing
 - Consumer grouping ("new parents", "gardeners"...etc.) and behavior analysis
- Web
 - Search result grouping
 - News article grouping (Google news)
- Computer Science: Image analysis
- Climatology: Weather pattern analysis (high pressure/warm regions)

Unsupervised Example: Google News

- Google News algorithm automatically groups **related news stories** into sections

Technology

Follow Share

Latest Mobile Gadgets Internet Virtual reality Artificial int >

Samsung Reportedly Delays Galaxy Fold Launch Events in China
Gizmodo • 8 hours ago

- Samsung Galaxy Fold Review: A fragile, uneven step toward the future
Engadget • 14 hours ago

[View full coverage](#)

Fortnite Battle Royale v8.5 update to include huge changes
FortniteINTEL • 11 hours ago

- All Confirmed Bug Fixes and Changes In The Upcoming V8.50 Fortnite Battle Royale Update
Fortnite Insider • 10 hours ago

[View full coverage](#)

New iPhone Exclusive Details Apple's Shocking Design
Forbes • 6 hours ago

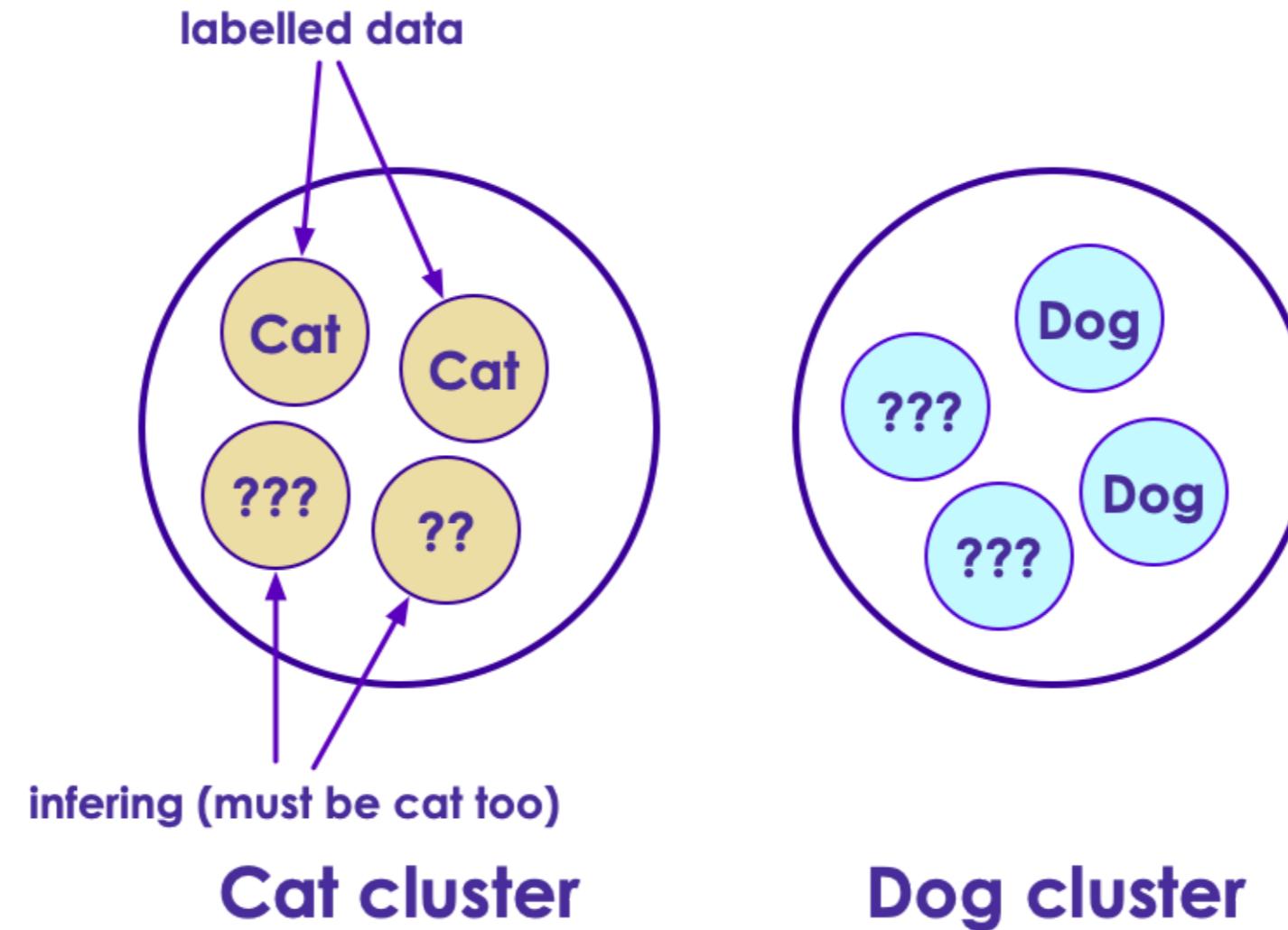
- Alleged case-maker's impressions for iPhone "XI and XI Max" appear to confirm square camera module yet again
Notebookcheck.net • 16 hours ago

[View full coverage](#)

news stories clustered

Semi-Supervised

- We are still learning, but not all data points are 'labelled'
- But by grouping data points together, the algorithm can 'infer' information, even when labels are missing

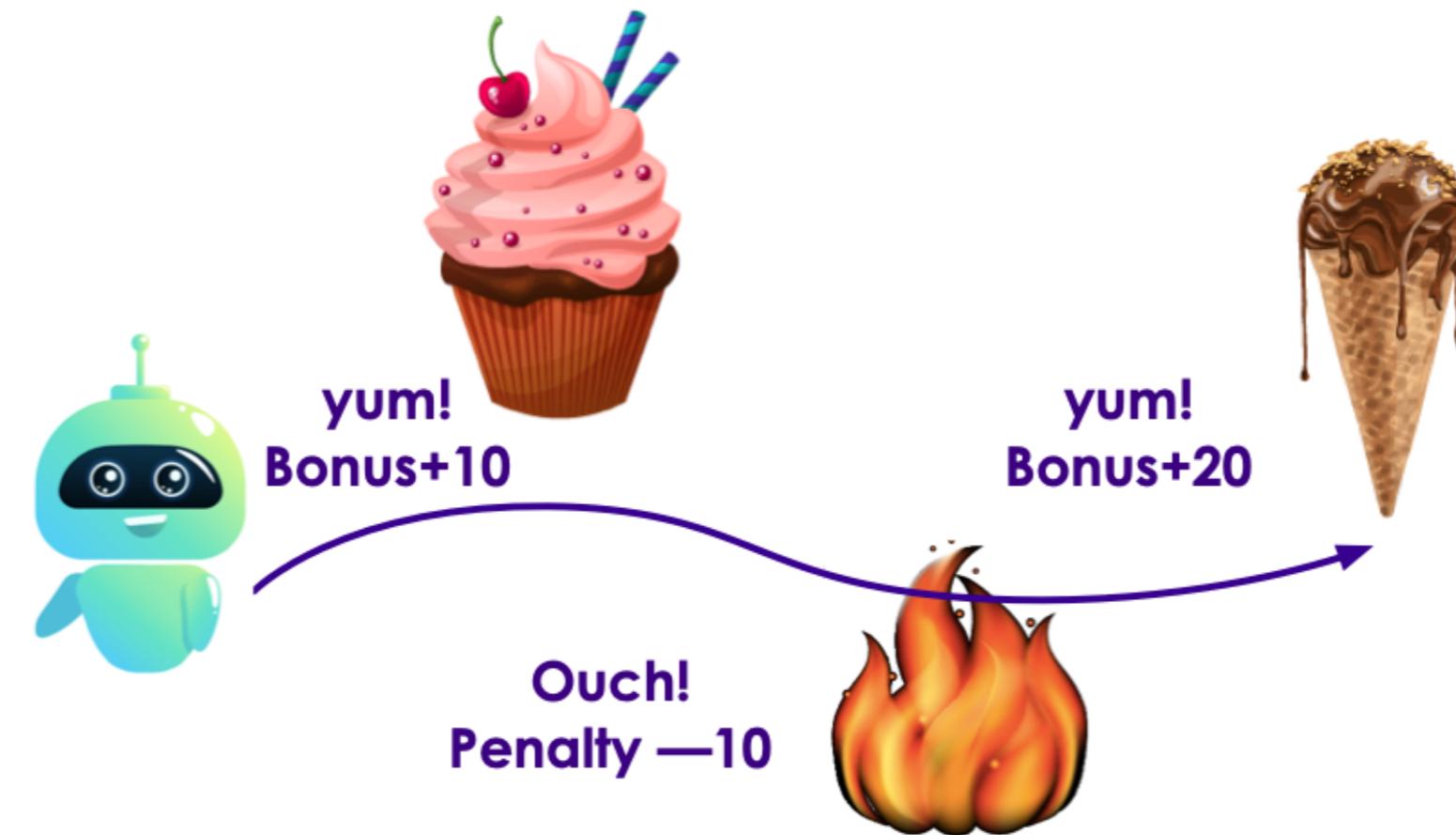


Reinforcement Learning

- Imagine you are playing a new video game. You have no idea how to play it. How will you learn?
- Try a few things:
 - Open a door -> get more money / ammo
 - Jump from a cliff -> got hurt.. Loose health points .. Ouch!
- This is how 'Reinforcement Learning' works.
 - Algorithm tries a few moves.. And learns automatically

Reinforcement Learning

- Here the robot gets rewarded for 'food' and penalized for walking into fire



Reinforcement Learning: Further Reading

- OpenAI trounces Dota-3 players
- Deep Mind's AI beats GO champion

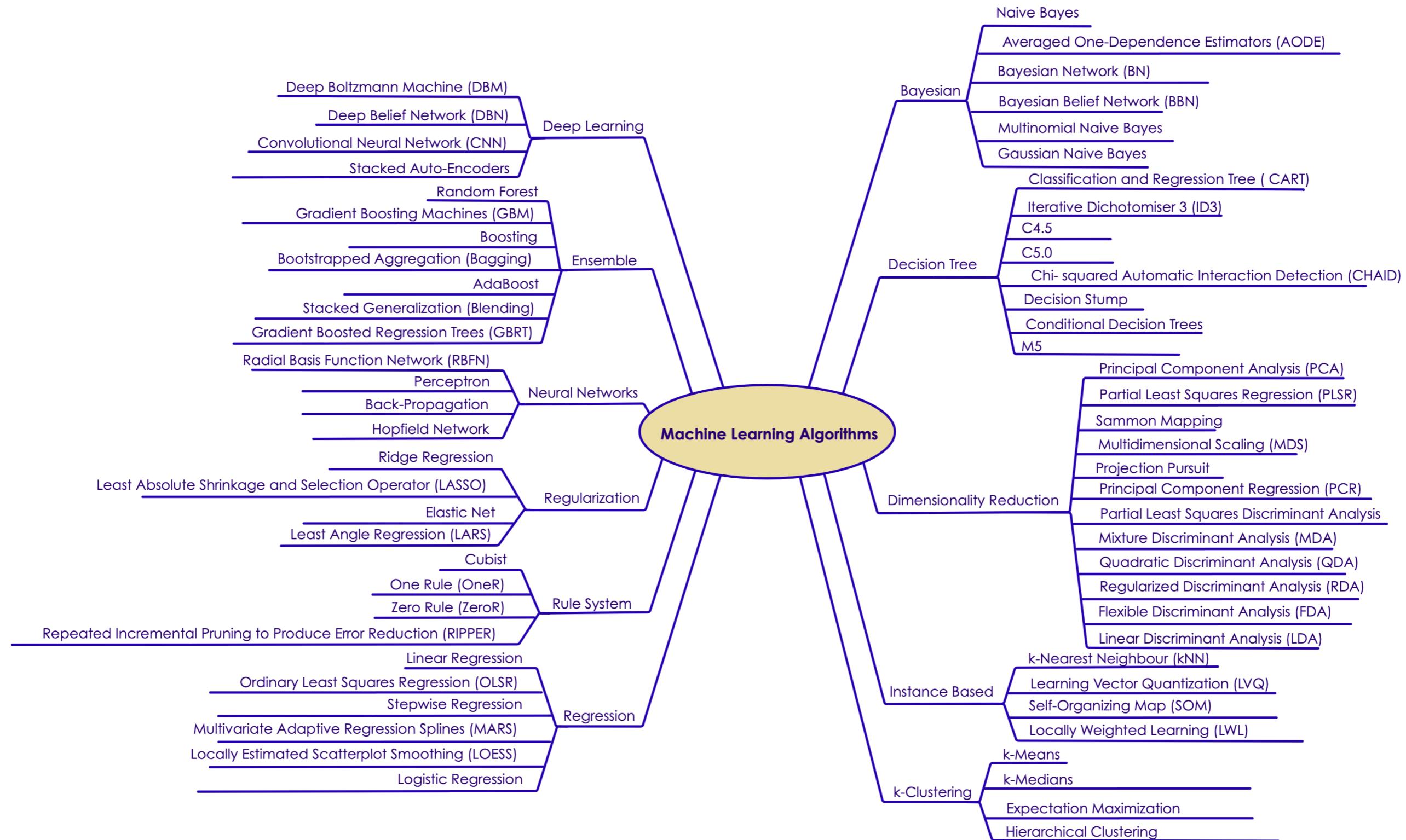
The company's latest AlphaGo AI learned superhuman skills by playing itself over and over

- Google's Alpha-GO defeats GO master
- OpenAI is founded by Elon Musk.
 - To promote AI research for public good

Algorithm Summary

Category	Sub Category	Example	Algorithms
Supervised	Regressions	<ul style="list-style-type: none"> Predict house prices Predict stock price 	<ul style="list-style-type: none"> Linear Regression Polynomial Ridge, Lasso, ElasticNet
	Classifications	<ul style="list-style-type: none"> Cancer or not Spam or not 	<ul style="list-style-type: none"> Logistic Regression SVM Naïve Bayes K Nearest Neighbor(KNN)
	Decision Trees	<ul style="list-style-type: none"> Classification (credit card fraud detection) Regression(predict stock prices) 	<ul style="list-style-type: none"> Decision Trees Random Forests
Unsupervised	Clustering	<ul style="list-style-type: none"> Group Uber trips Cluster DNA data 	<ul style="list-style-type: none"> K-Means Hierarchical clustering
	Dimensionality reduction	<ul style="list-style-type: none"> Reducing number of dimensions in data 	<ul style="list-style-type: none"> PCA
Recommendations		<ul style="list-style-type: none"> Recommend movies 	<ul style="list-style-type: none"> Collaborative Filtering

ML Algorithm Cheat Sheet



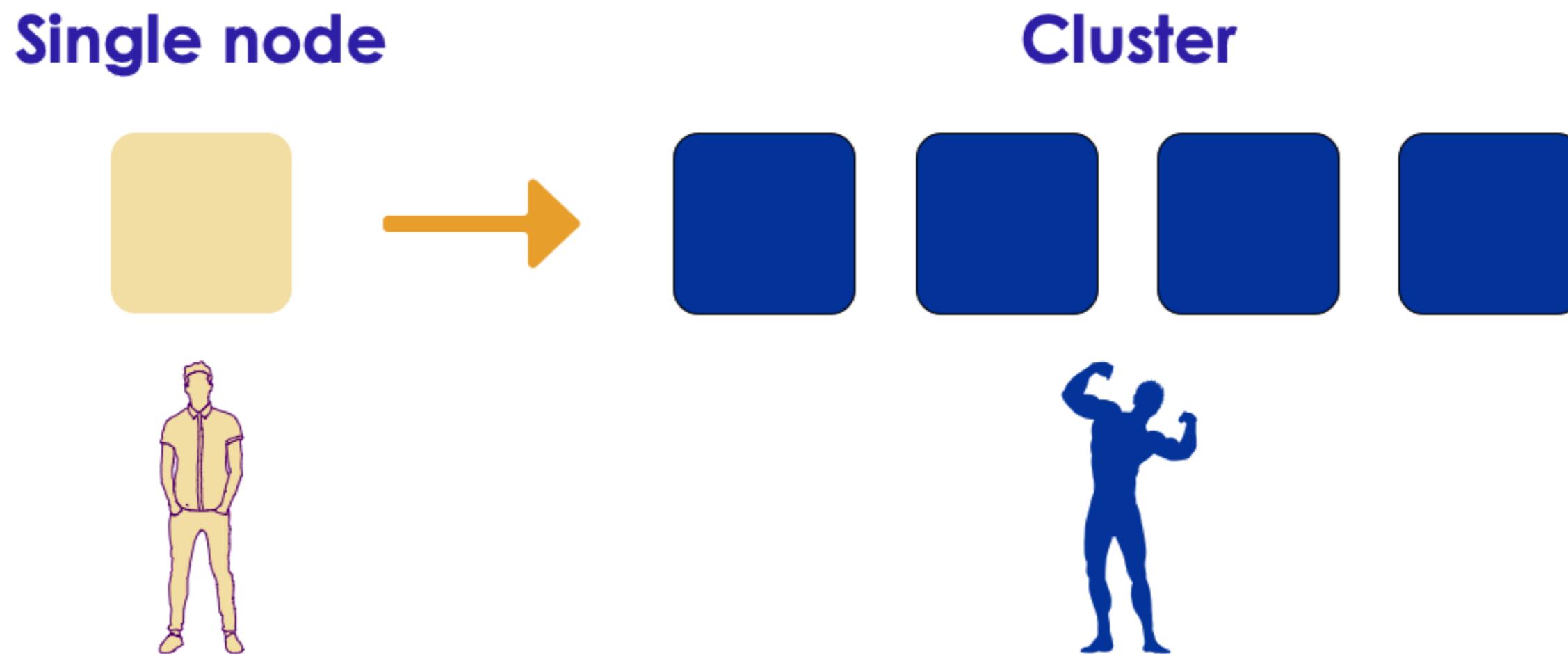
AI Software Eco System

AI Software Eco System

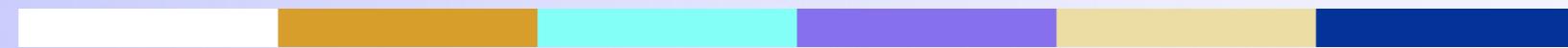
	Machine Learning	Deep Learning
Java	- Weka - Mahout	- DeepLearning4J
Python	- SciKit - (Numpy, Pandas)	- Tensorflow - Theano - Caffe
R	- Many libraries	- Deepnet - Darch
Distributed	- H2O - Spark	- H2O - Spark
Cloud	- AWS - Azure - Google Cloud	- AWS - Azure - Google Cloud

Machine Learning and Big Data

- Until recently most of the machine learning is done on "single computer" (with lots of memory-100s of GBs)
- Most R/Python/Java libraries are "single node based"
- Now Big Data tools make it possible to run machine learning algorithms at massive scale-distributed across a cluster



Introduction to TensorFlow



Lesson Objectives

- Understand the needs that TensorFlow addresses
- Be familiar with TensorFlow's capabilities and advantages
- Gain an understanding of a basic TensorFlow installation

TensorFlow Intro

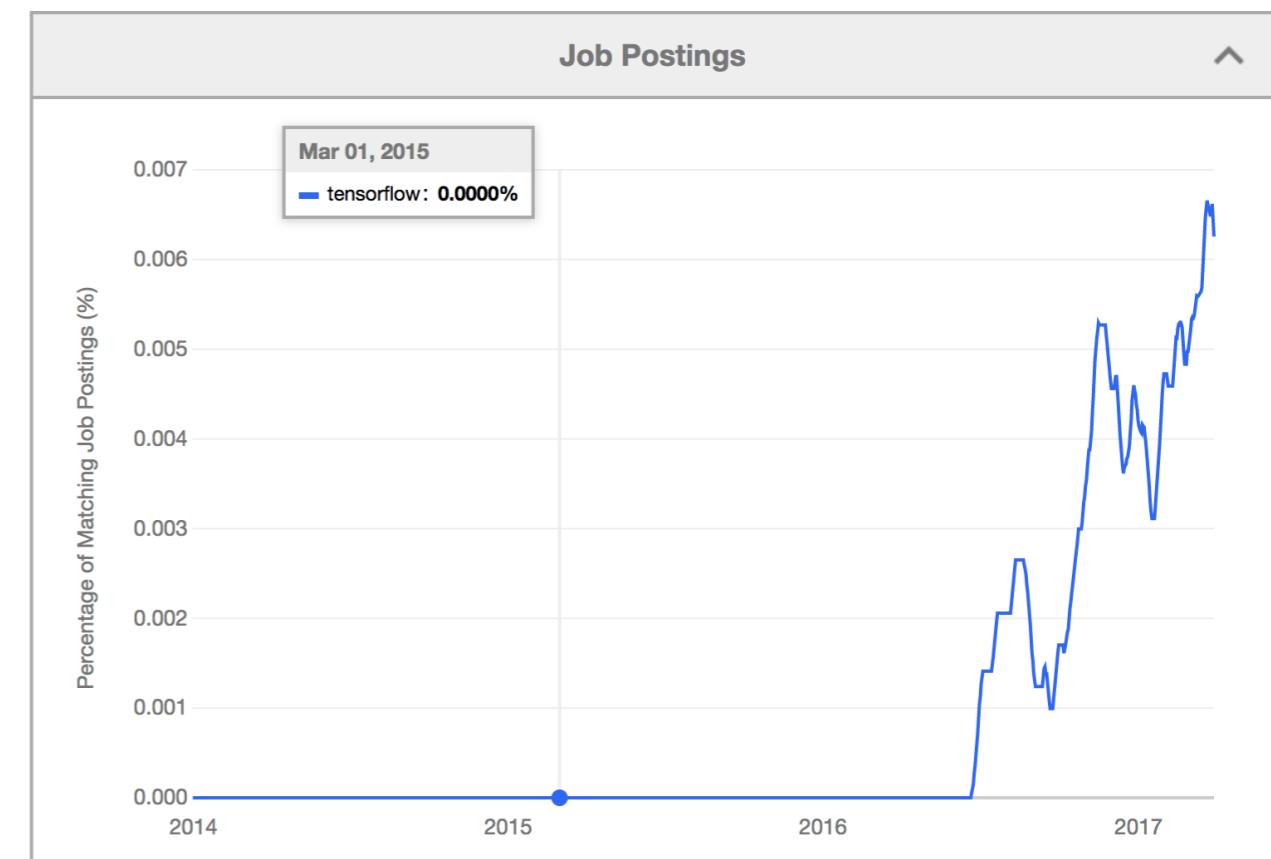
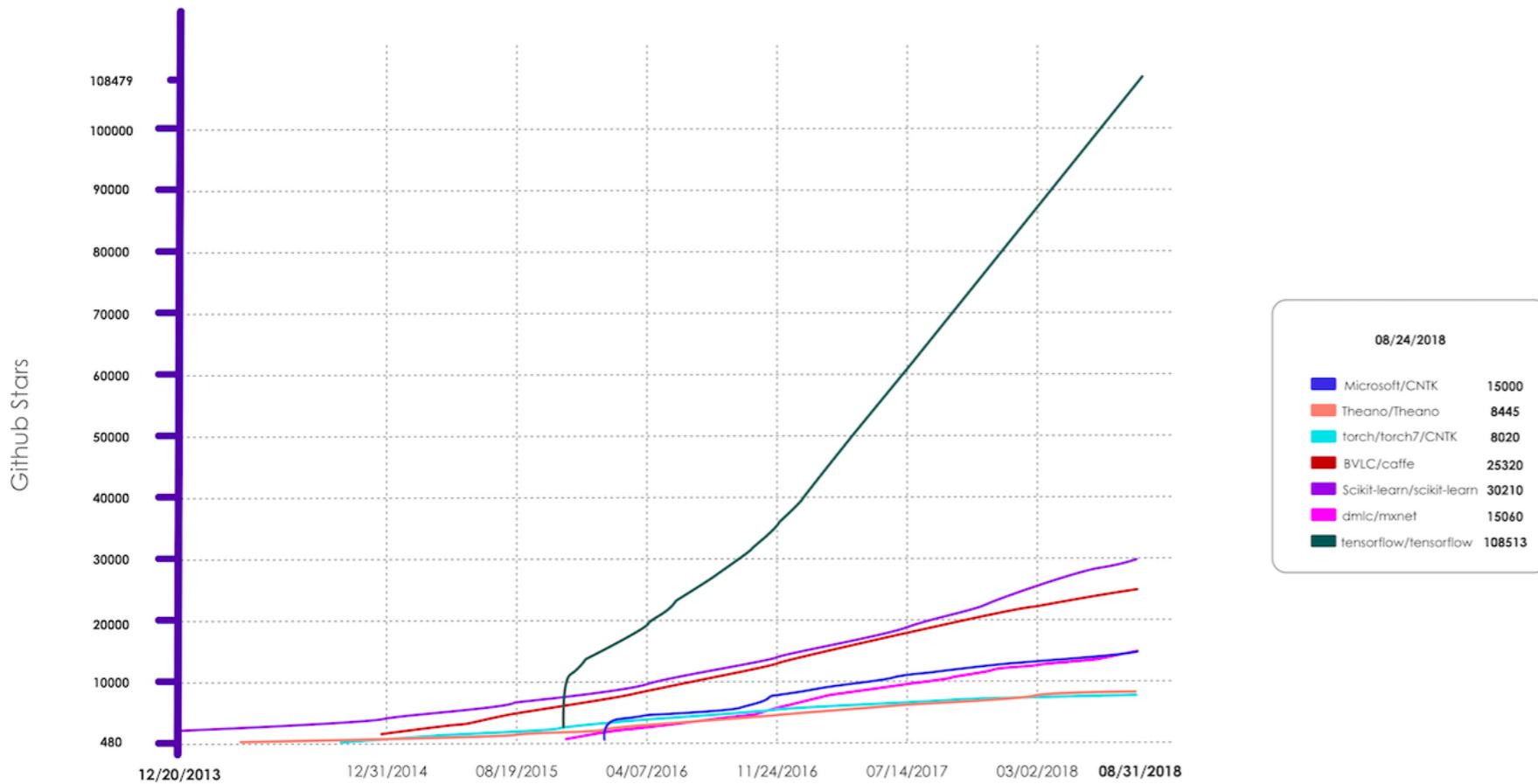
TensorFlow

- **Tensorflow** is a free and open source library for deep learning with neural networks
- Tensorflow was developed by Google Brain team for internal Google use. Google open sourced TensorFlow in Nov 2015
- Gained popularity very quickly because of its
 - clean design
 - flexibility
 - scalability
 - huge community
 - and of course the Google brand :-)
- Google is actively developing and supporting TensorFlow; also offers it in Google Cloud platform
- tensorflow.org



TensorFlow Popularity

- Tensorflow has outpaced its rivals in popularity
- Here is a survey of
 - Github Stars (left) and
 - Indeed job postings (right)



TensorFlow Features

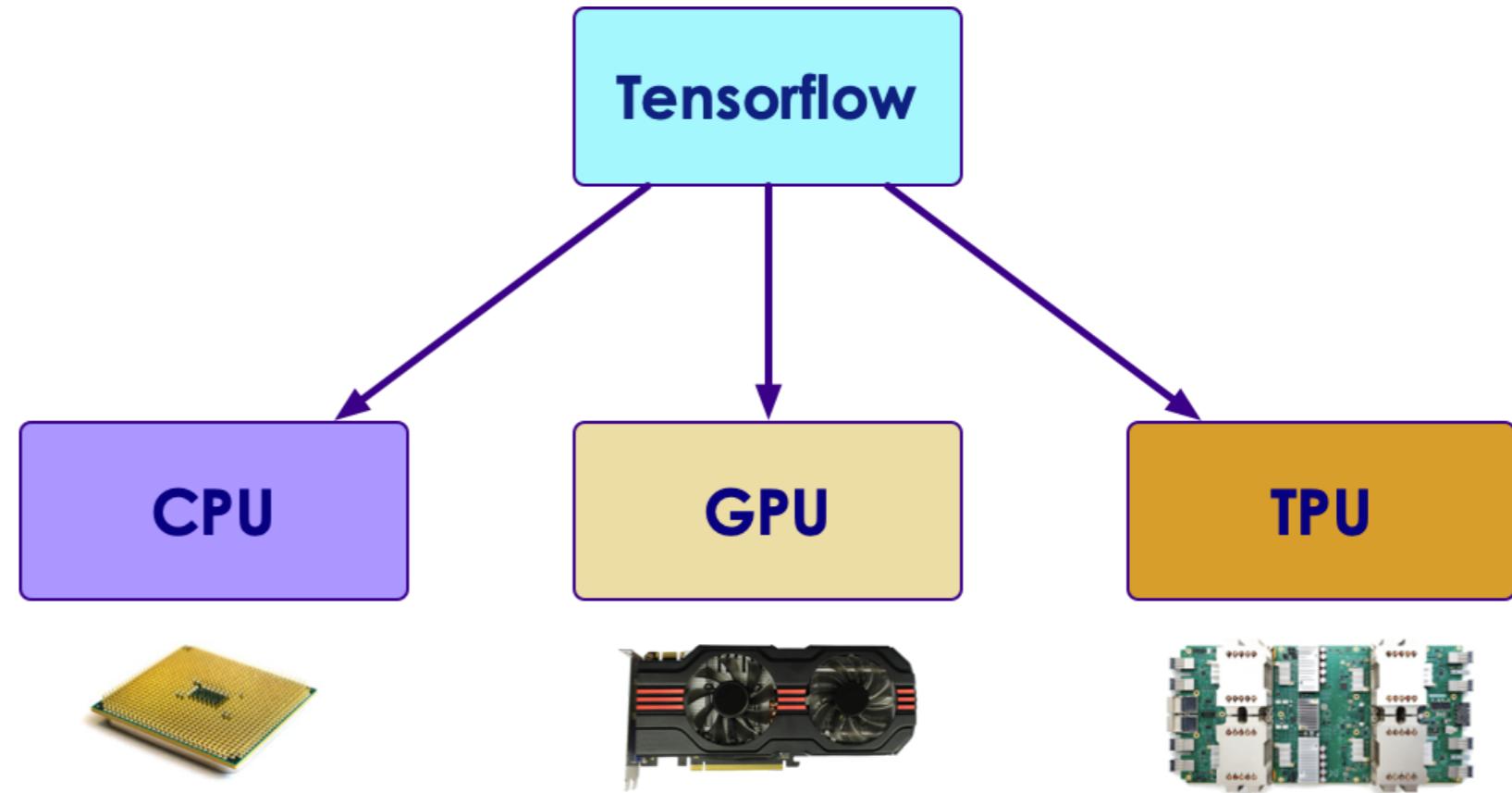
- Runs on all platforms (Windows, Mac, Linux), mobile devices and even in a browser (tensorflow.js)
- Core is written in C++ ; very efficient implementation
 - Wrappers in Python for ease of use
 - Other language support improving : Java, Go, R
- Runs on CPU, GPU and TPU (more on this later)
- Other high level APIs are built on top of TensorFlow
 - Keras and Pretty Tensor
- Has a very nice UI called **Tensorboard** to visualize graphs and learning process
- Great community
 - <https://github.com/jtoy/awesome-tensorflow>

TensorFlow Noteworthy Versions

Version	Release Date	Noteworthy Features
0.01	2015-11	Initial Release from Google
0.20	2016-05	TensorFlow Reaches Maturity
1.0	2017-01	First Stable Release
1.1	2017-07	Installable with Pip/conda
1.14	2019-02	Stable 1.x Release
2.0	2019-10	2.0 Release (big release)

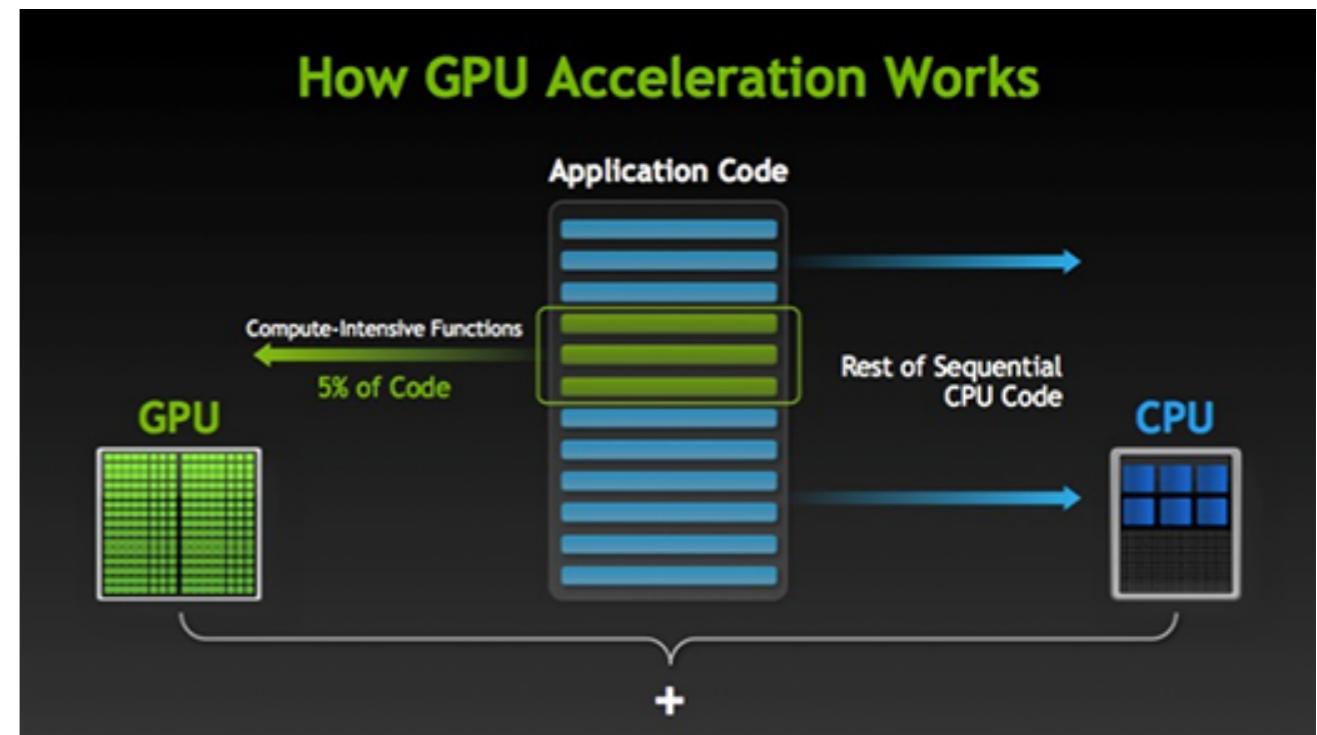
Tensorflow Hardware Support

- TF can run on multiple hardware devices: CPU, GPU, TPU
- CPU: good for inference (prediction)
- GPU: lots of compute power, good for training
- TPU: highly optimized AI-chip



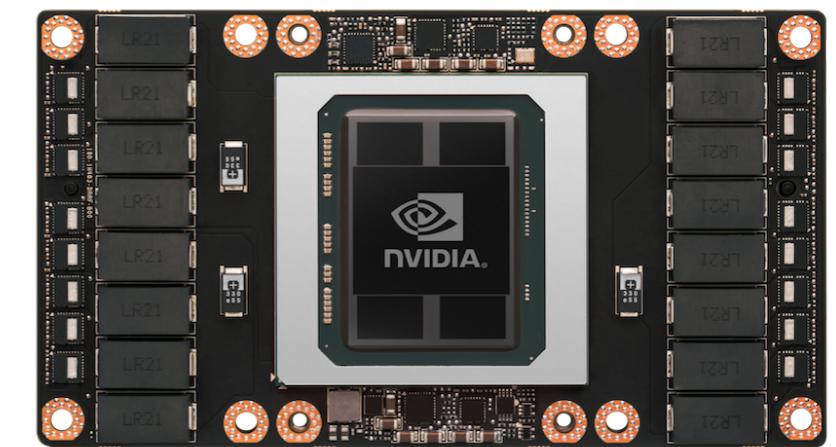
Using GPUs

- One of TensorFlow's most exciting features is using GPUs for compute capacity
 - ML is mainly linear algebra (matrix manipulation)
 - GPUs specialize in fast linear algebra.
 - GPUs + ML = match made in heaven.
- Machines running with GPUs have been shown up to 10x faster.
- TensorFlow will consume GPU + all its memory
 - So, you can't use the GPU for graphics at the same time
 - Servers don't use graphics anyway
 - Workstations should have two NVIDIA cards.



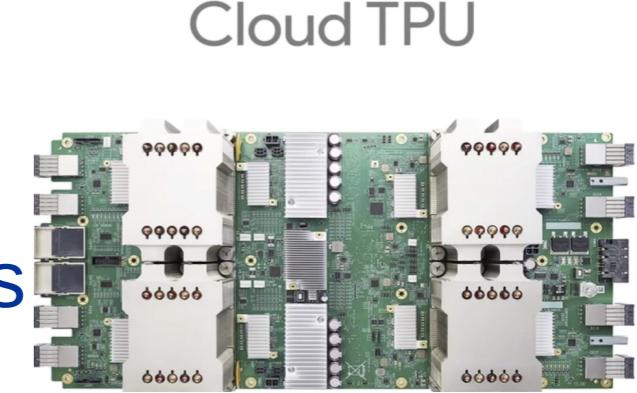
GPU Support

- NVIDIA GPUs are best supported
 - use CUDA library
 - very easy to get set up: install tensorflow-gpu
 - Datacenters: *Must* use **Tesla** GPU line per NVIDIA TOS
 - Workstations: Recommend **Quadro**
 - Home/PC: GeForce (Gaming) GPUs work well
- Tensorflow 2.0 also supports AMD using OpenCL
 - Support a bit new
 - Not as fast as NVidia but hardware is more affordable.
 - Apple/Mac also uses AMD!
- Intel GPUs not currently supported.



TPU Support

- TPU is Google's custom chip built for AI workloads
 - 3rd generation as of March 2018
- Use cases:
 - Processing Google Streetview photos (extract street numbers / text)
 - Image processing Google Photos
 - AlphaGo game
- TPUs are have two very distinct use cases: Training and Inference
- Training TPUs only available in Google Cloud Platform for now
 - High power chip
 - *Free* evaluation with Google Colaboratory
- Edge TPUs
 - Much smaller and consumes far less power compared to 'data center' TPUs'
 - Google Sells physical devices
 - Designed to be used in IOT type devices, robotics, etc



Google Colaboratory

- Google Colaboratory is a **free** hosted environment for AI
- Familiar Jupyter notebook environment
- Provides GPU and TPU platforms!
 - The only **free** GPU and TPU access available!
- Great for light workloads and evaluation
- Serious users will want to upgrade to Google Cloud
 - Security
 - Guaranteed performance access



Cloud Cost

- Cost on Google Cloud Platform: (Hourly)

Device	Type	Gen	Year	Memory	Cost
T4	GPU	Turing	2018	16GB	\$0.95
P4	GPU	Pascal	2016	16GB	\$0.60
K4	GPU	Kepler	2014	16GB	\$0.45
TPU3	TPU	3rd	2019	64GB	\$8.00
TPU2	TPU	2nd	2017	64GB	\$4.50

Tensorflow vs. Scikit-learn

TensorFlow	Scikit-Learn
Focused on Neural Networks and Deep Learning Models (Other Algorithms Available too)	Focused on a wide-variety of ML algorithms
Base Framework low-level (Wrappers available for high-level)	Higher-level API
Distributed Execution Model	Not Distributed
Extensive GPU Optimization	Not GPU Optimized
Support for Distributed File Systems (HDFS)	No Support for Hadoop
Supports Python API (primary), plus Java, C++, Go, R, etc.	Python Support Only

Tensorflow Operations

Simple Helloworld (TF v1)

```
import tensorflow as tf

# Create a Constant op
# The op is added as a node to the default graph.
hello = tf.constant('Hello, TensorFlow!')

# Start tf session
sess = tf.Session()

# Run the op
print(sess.run(hello))
```

Simple Math using Tensorflow (V1)

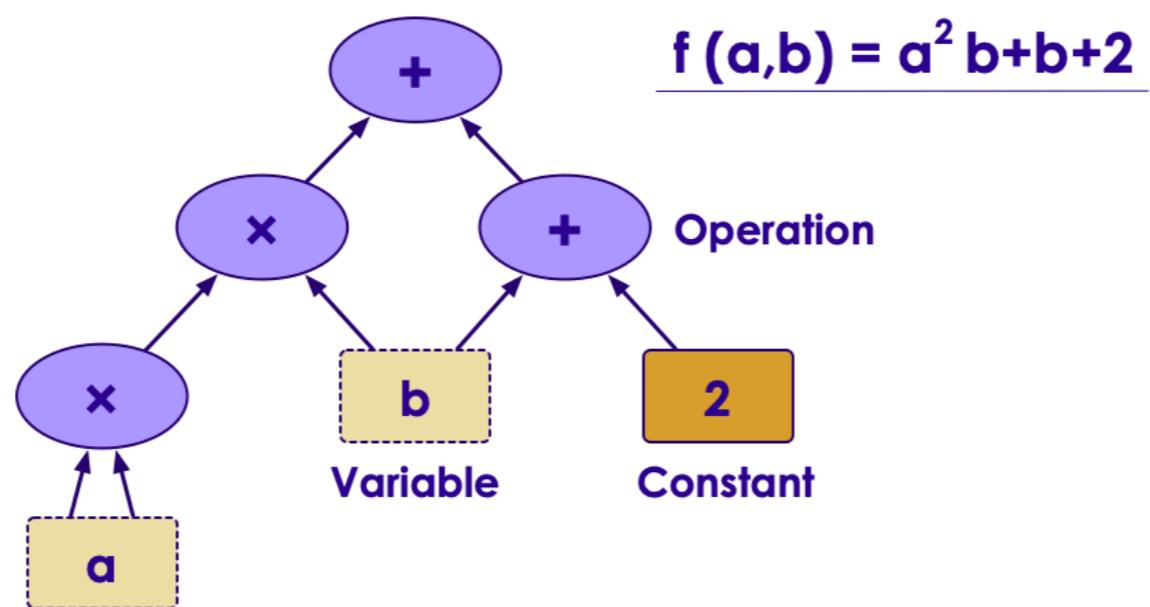
```
import tensorflow as tf

## define placeholders and variables
a = tf.placeholder(tf.int32)
b = tf.placeholder(tf.float32)
c = tf.constant(2)

x = a * a * b + b + c

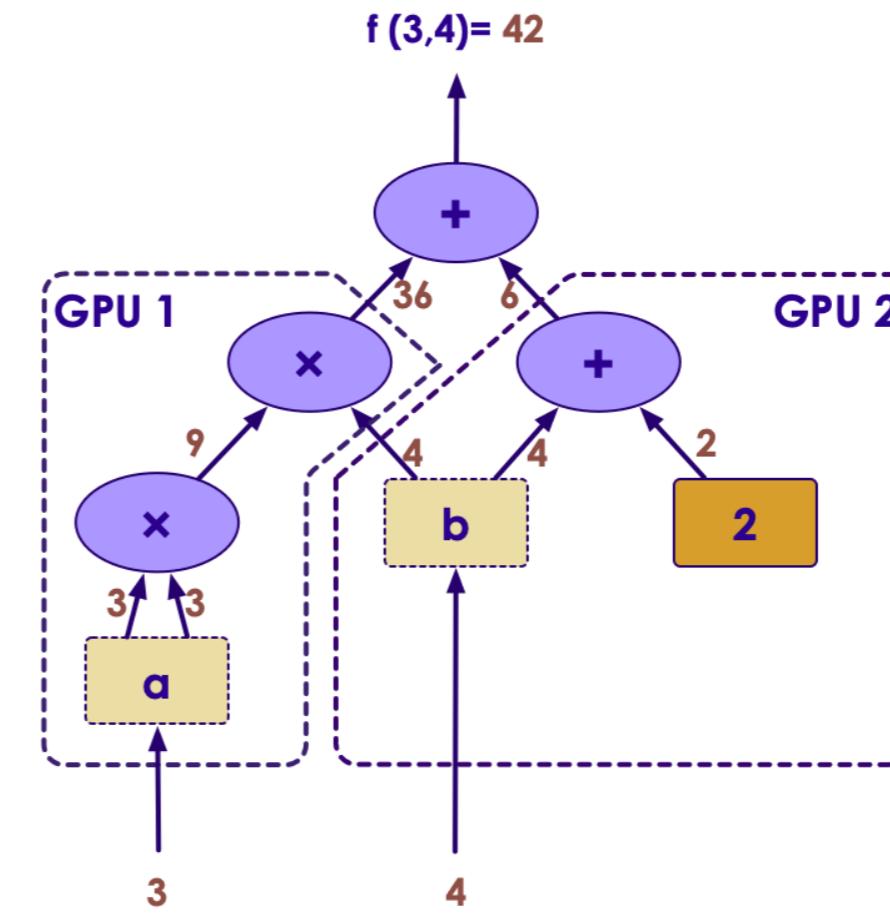
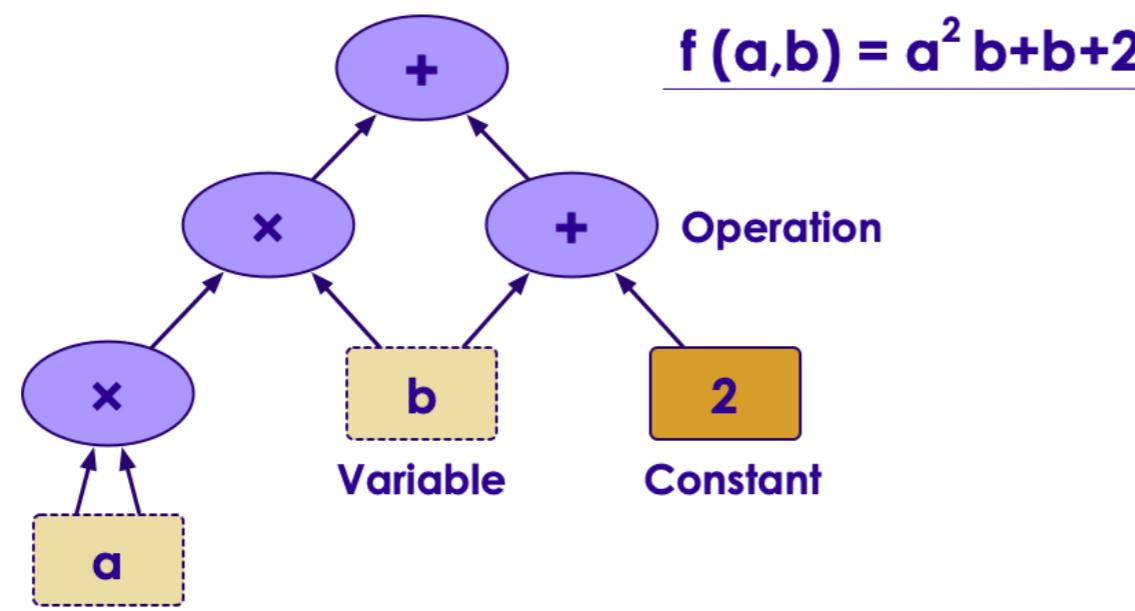
with tf.Session() as sess:
    output = sess.run (x,
                      feed_dict = { a : 10,
                                    b : 5.0})

## output
## x = 10 * 10 * 5.0 + 5.0 + 2
##      = 507.0
```



Tensorflow Operations

- Tensorflow represents operations as mathematical graphs
- Here is an example of
 $x = a^2 * b + b + 2$
- On left we have 'logical graph' and on the right we have 'physical graph' (how the graph is executed on 2 GPUs)



Parallel TensorFlow

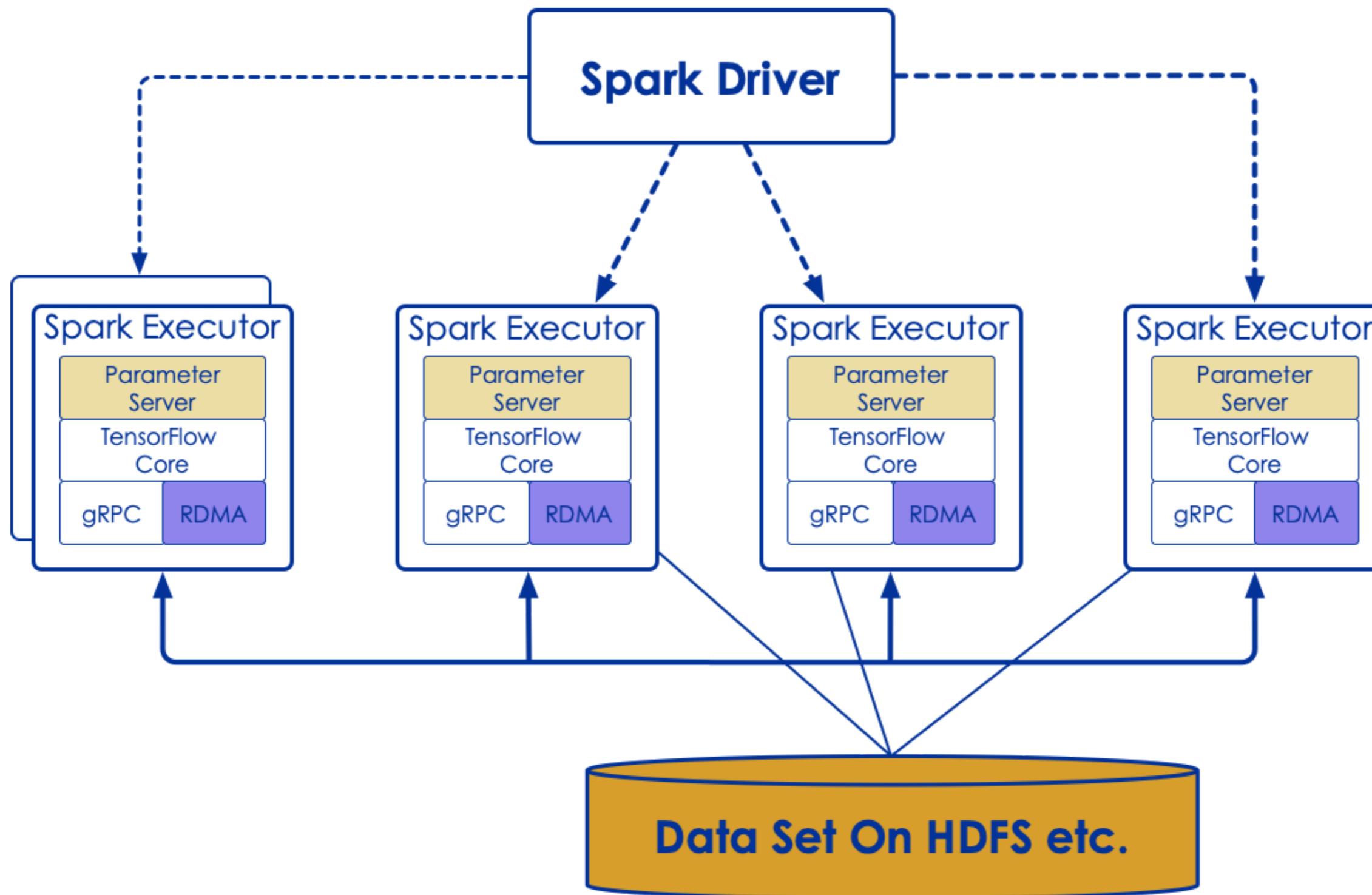
- NN's are known for being difficult to parallelize
- But, TensorFlow can in distributed mode
 - run on multiple CPU/GPU on a single machine
 - run on distributed machines
- Define a distributed master service plus worker services

TensorFlow + Spark = Scale

- Spark is a very popular distributed platform
 - Much faster
 - Better ML support
- Yahoo / Databricks: TensorFlow on Spark (TFoS)
 - Framework for distributing TensorFlow apps on Spark / Hadoop
 - Used by DataBricks in Spark Distribution
- Intel: Analytics Zoo
 - Allows users to do distributed tensorflow/Keras/PyTorch on Spark
 - Does not support GPU acceleration
- Spark v3 will support distributed tensorflow natively!



Tensorflow on Spark



Deep Learning in TensorFlow

- Deep Learning simply means a Neural Network:
 - With more than one hidden layer
- TensorFlow is the world's most popular engine for deep learning
 - Execution Engine is Tuned to Facilitate Deep Learning
 - Runs very fast on GPUs!

Traditional Machine Learning in TensorFlow

- TensorFlow can also be used for traditional Machine Learning
 - `tf.estimator` API often used for this
 - Alternative to scikit-learn
- Traditional Machine Learning Algorithms:
 - Linear Regression
 - Logistic Regression
 - Support Vector Machines
 - Decision Tree Learning
- Other libraries are more extensive in terms of features

Tensorflow Evolution (TF2)

Base TensorFlow is Low Level

- It is more of an execution model
 - Handles the flows of Tensors
 - Does not automatically train models (We can write code to do that)
- Tend to be verbose (lots of code for simple tasks, see below)
- Low Level TensorFlow does **NOT** have built-in training
 - You have to do it yourself with Tensor Transformations

```
import tensorflow as tf

a = tf.placeholder(tf.int32)
b = tf.placeholder(tf.float32)
c = tf.constant(2)

x = a * a * b + b + c

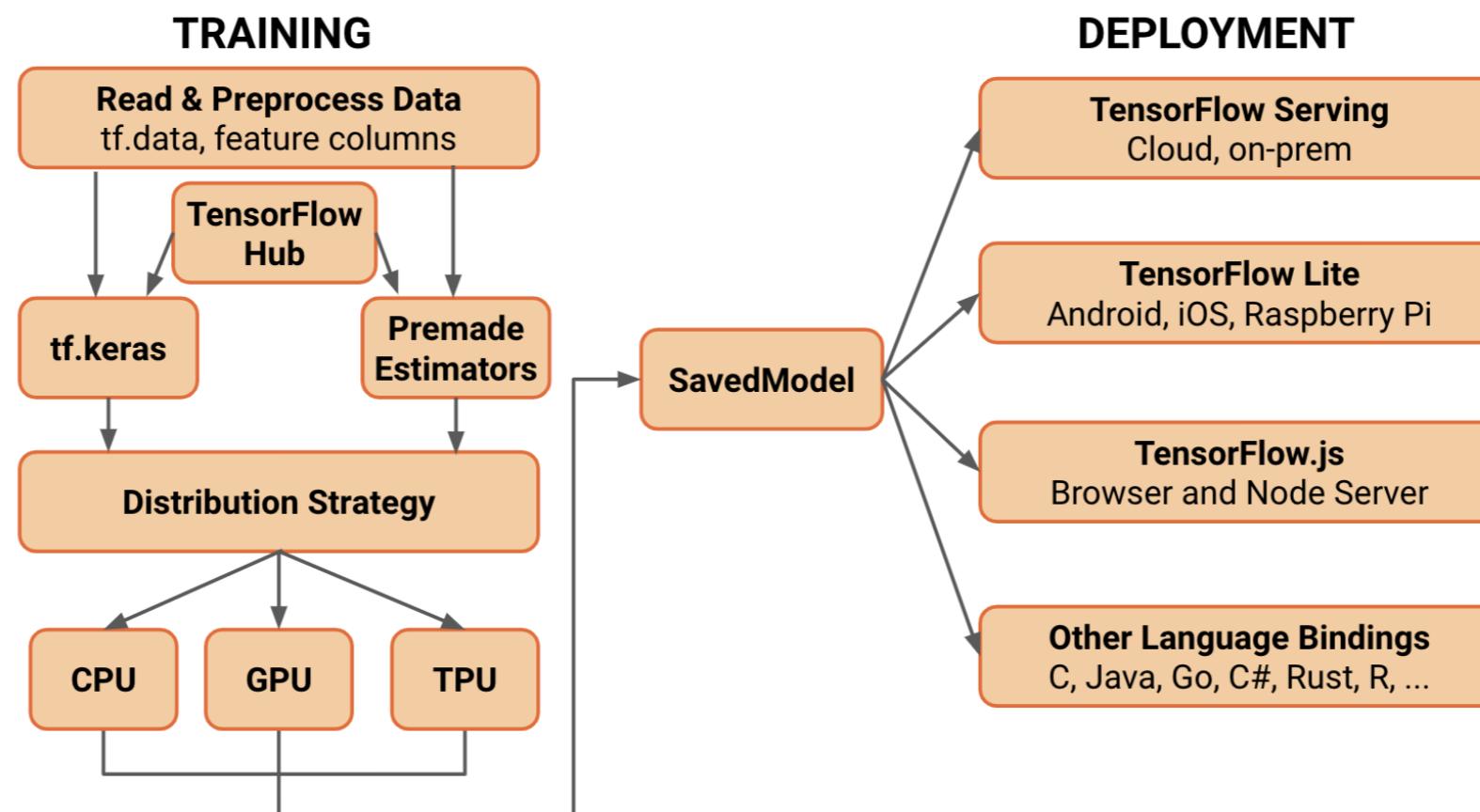
with tf.Session() as sess:
    output = sess.run(x, feed_dict = { a : 10, b : 5.0})
```

Tensorflow 2 (tf2)

- Tensorflow2 (tf2) is in a *big* shift to Tensorflow

- Goals

- Making TF easier to use
- Simplify and cleanup APIs that evolved over time
- Standardize on high level Keras API



TF2 Example

- Here we use high level `tf.keras` API
- Focus on building the network; not low level operations

```
# Simple feed forward network for regression

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense, Sequential

# create a model
model = Sequential()

# And add layers
model.add(Dense(units=10 activation=tf.nn.relu, input_shape=[3]))

model.add(Dense(units=5, activation=tf.nn.relu))

model.add(Dense(units=1))

model.compile(loss='mean_squared_error',
              optimizer= 'adam',
              metrics=[ 'mean_absolute_error', 'mean_squared_error'])

# training will take time
model.fit(training_data)

# prediction is quick
model.predict(test_data)
```

Backward Compatibility

- Honestly, for most of us, there is no real reason to use V1 APIs
- In TF2, there is a backward compatible module `tf.compat.v1` available; This has all the older APIs
- TF2 comes with a `automatic conversion script` to port v1 --> v2
- Read `migration guide` for details

Enhancements in TF2

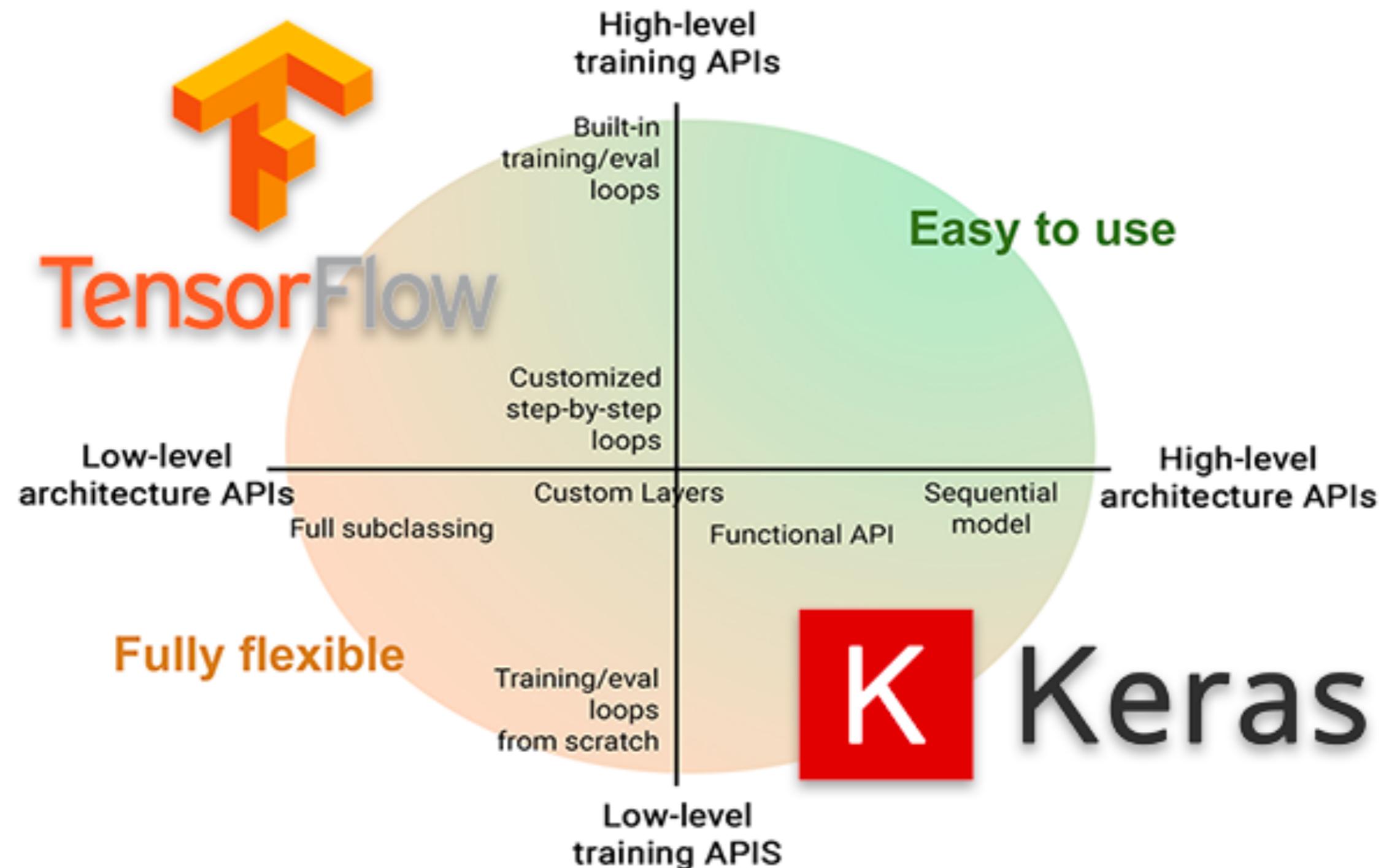
- (These features are explained in detail in the next few slides)
- **Standardizing on Keras style APIs** (big change)
- Eager execution
- Garbage collection of variables
- User defined functions
- Autograph
- Dataset API

tf.keras

- Tensorflow 2.0 **encourages** use of the `tf.keras` API
 - Much easier to use
- Keras (keras.io) is an API that pre-dates tensorflow
 - The `keras.io` project is a API that can use tensorflow, CNTK, or Theano
 - Able to write code that is tensorflow independent.
- Tensorflow has its own re-implementation of the Keras API called **`tf.keras`**
 - Better optimized for Tensorflow than `keras.io`
- Tensorflow has said that the `tf.keras` API is **the** API going forward.



Tensorflow + Keras



Eager Execution

- Eager Execution is now the **default** mode of execution
- This means that code will execute like Python normally does
- Graphs and Sessions are more like implementation details
- We will still use the graph/session execution for production workloads
- Allows us to set breakpoints, step through code, etc
 - `tf.config.experimental_run_functions_eagerly(True)`
 - Not for production! But great for development

Globals and Garbage Collection

- In Tensorflow 1.x, all variables were added to the global graph
- No Garbage Collection!
- Very difficult to keep track of old variables littering the Tensorflow Session Graph
 - much like C/C++ memory leaks!
- Now -- If you lose a reference to a variable, it is garbage collected
 - Just like in regular Python (or Java, C#,)

Functions

- Tensorflow 1.x Didn't really have functions
 - You could use a **Python** function, but it was difficult to optimize
 - It was also difficult to share with other models.
- Tensorflow 2.x allows **functions** at the tensorflow Graph Level
 - use the `@tf.function` annotation
 - Kind of like a SQL Stored Procedure
 - Allows you to embed common functionality at the *tensorflow* level
 - More efficient / better reuse
- Allows Shared Libraries and Code
 - Common Functions
 - Can be serialized and deserialized

AutoGraph

- AutoGraph allows python loops like **for**, **while** to be converted into tensorflow graph code
- Allows us to write loops that will allow dynamic placement of layers / cells
- Example (Dynamic RNN):

```
for i in tf.range(input_data.shape[0]): # Note the For loop
    output, state = self.cell(input_data[i], state)
    outputs = outputs.write(i, output)
```

Dataset API

- The Dataset API is much expanded
 - Handles Data Management side of things!
 - Very important part of Data Science
- Used now with Keras API, not just Estimator API
- Recommended way to handle structured data
 - or when mixing structured data with unstructured data in the same model.
- Dataset API allows us to use very large datasets (too big for Pandas dataframes, etc)

Installing TensorFlow

Installation

- Can install with pip!
 - TensorFlow 2 packages require a pip version >19.0.
- You may need to have python-dev libraries installed.
- Recommended to install in virtualenv (or conda environment).
- This will install the latest TF (v2.x)

```
$ (sudo) pip install tensorflow
$ (sudo) pip install tensorflow-gpu # For GPU
```

- Anaconda: Now officially supported

```
$ conda install tensorflow
$ conda install tensorflow-gpu # For GPU
```

Lab: Hello World in TensorFlow

- **Overview:**

- In this lab, we will do a hello world for TensorFlow and Keras.

- **Approximate time:**

- 15-20 minutes

- **Instructions**

- Follow **tensorflow-1** lab
 - Follow **tensorflow-2** lab



Review and Q&A

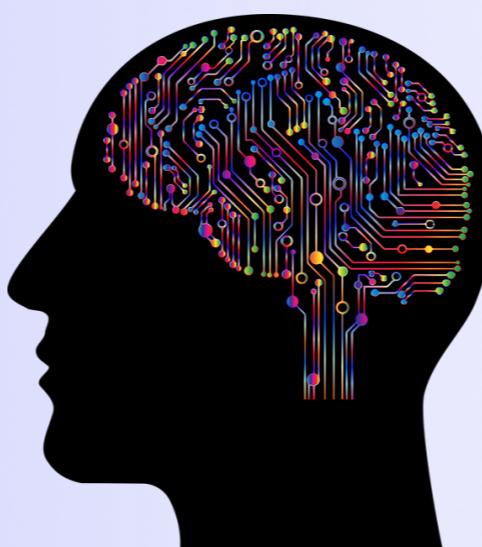
- Let's go over what we have covered so far

- **Questions**

- What hardware platforms does TF support?
- What are some new features in TF2?

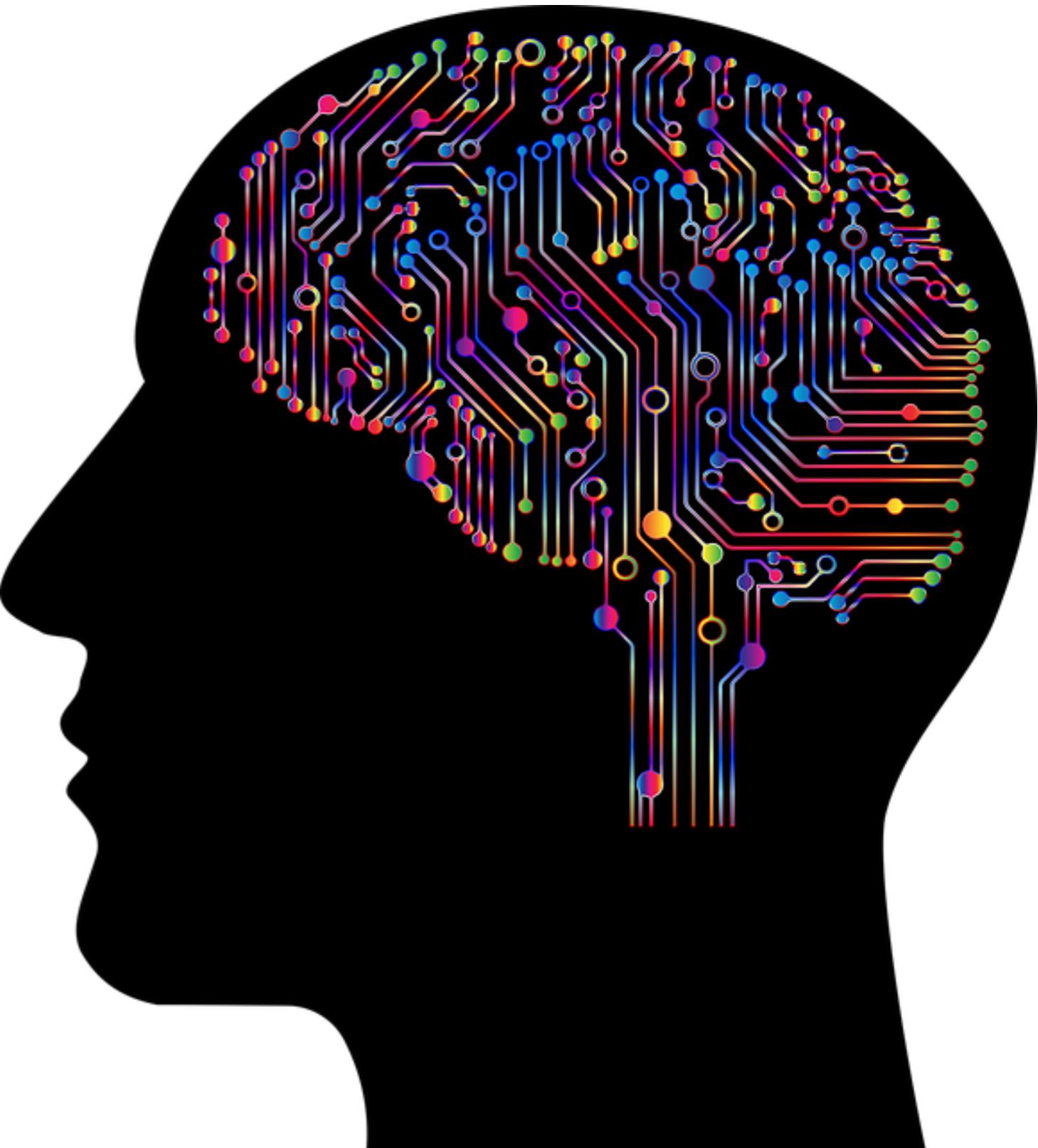


Introduction to Neural Networks



Lesson Objectives

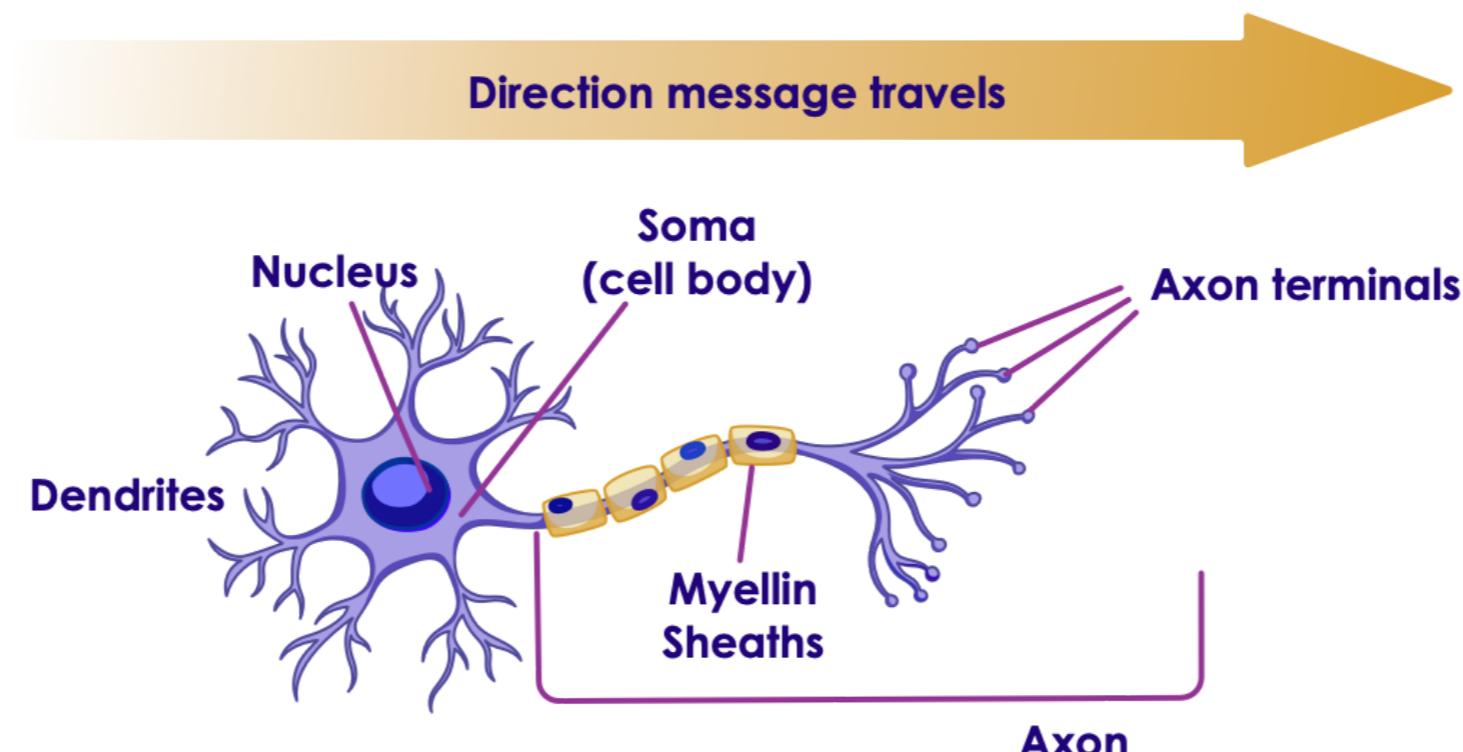
- Understand neural network architectures
- How to size and build neural networks



Artificial Neural Networks (ANN)

Artificial Neural Networks (ANN)

- ANNs are at the core of Deep Learning
 - they are powerful, scalable and can solve complex problems like classifying billions of images (Google Images)
- ANNs were inspired by neurons in human brain
- However ANNs have evolved quite a bit from their original inception. For example planes are inspired by birds first, but now modern planes have evolved a lot from their original designs

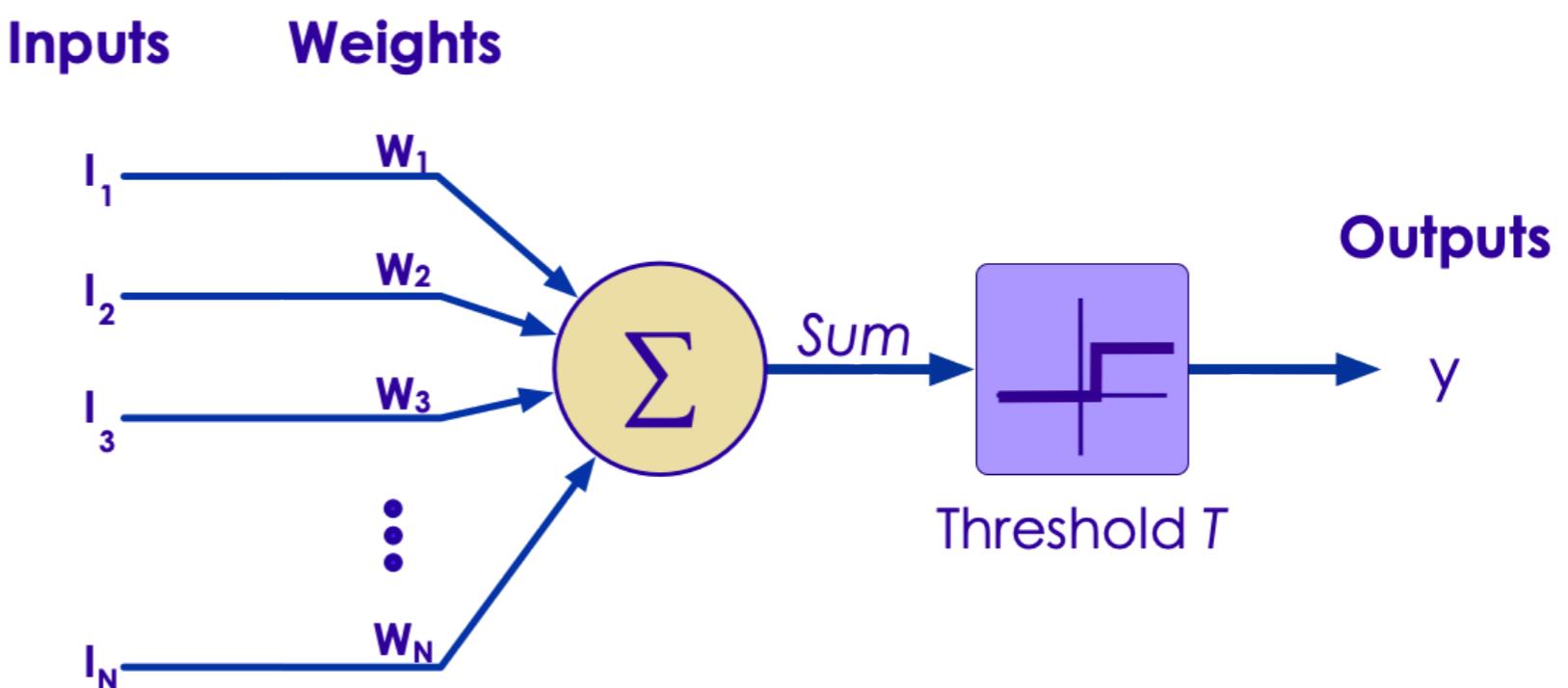


Neural Networks History

- 1943: McCulloch Pitts Neural model
- 1962: Frank Rosenblatt invented the Perceptron:
- 1969: Marvin Minsky's paper threw cold water on ANNs.
He demonstrated the ANNs can't solve a simple XOR problem
- 1970s: First AI Winter
- 1980s: some revival in ANNs (new models + training techniques)
- 1986: D. E. Rumelhart et al. published a groundbreaking paper introducing the backpropagation training algorithm.
- 1990s: Second AI winter (Methods like SVMs were producing better results)
- 2010s: huge revival in AI after some promising results
- Now: The race is on!
- References : [1](#) , [2](#)

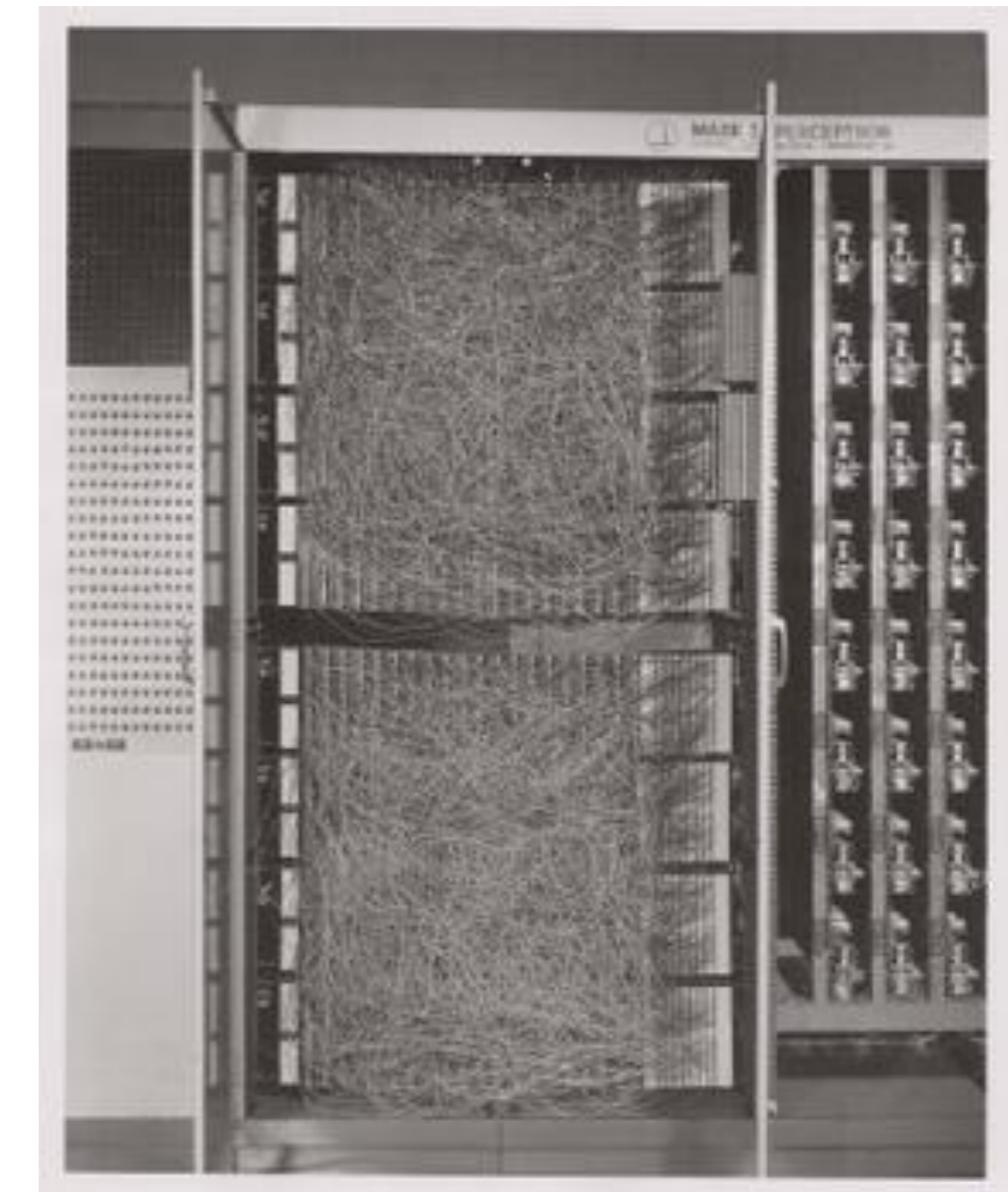
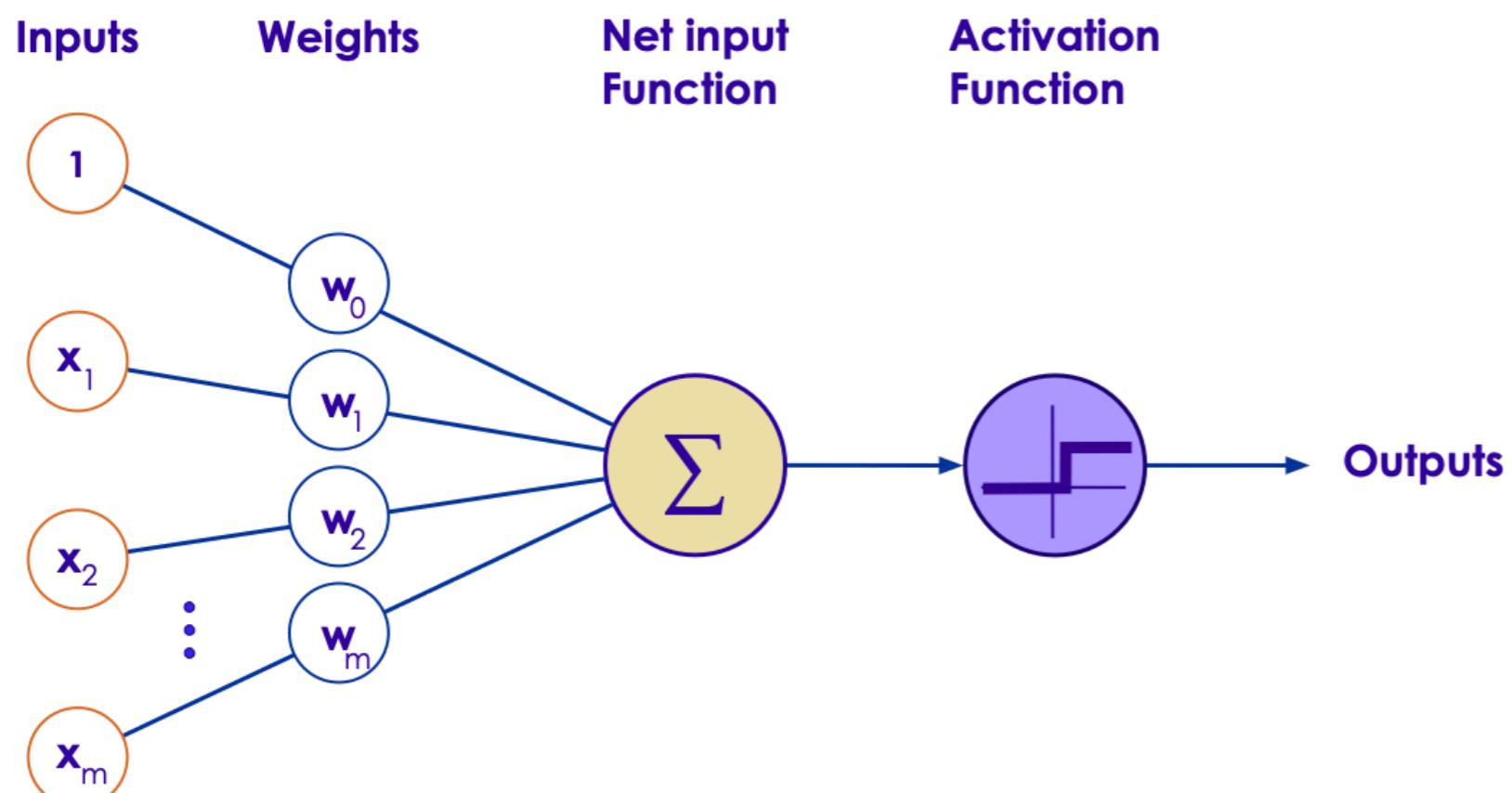
1943: McCulloch Pitts Neural Model

- McCulloch and Pitts defined a simple model of a Neuron (paper)
- It consisted of N inputs I_n and N Weights
- Inputs are **binary (on/off)**
- Inputs and weights are summed up and a threshold function produces output
- Limitations:
 - Binary input / output
 - Weights (W_n) were set manually; No way to automatically train weights



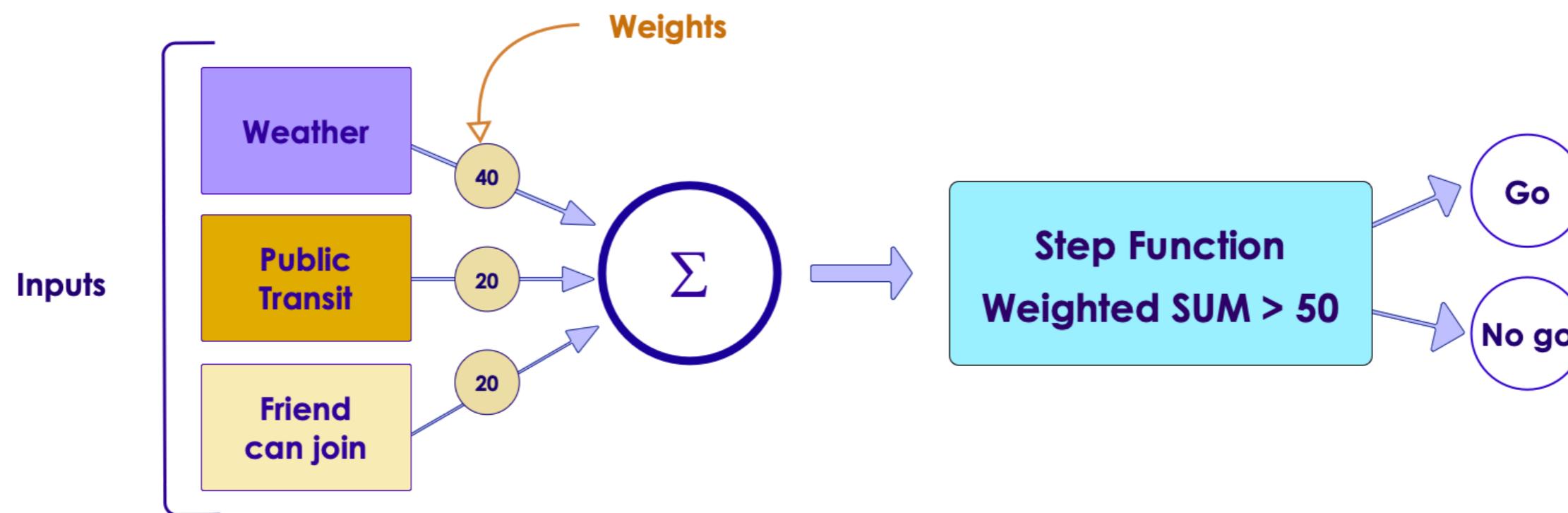
1962: The Perceptron

- Frank Rosenblatt invented the Perceptron
- Inputs are **numbers (not binary as before)**
- Simplest type of Feedforward neural network



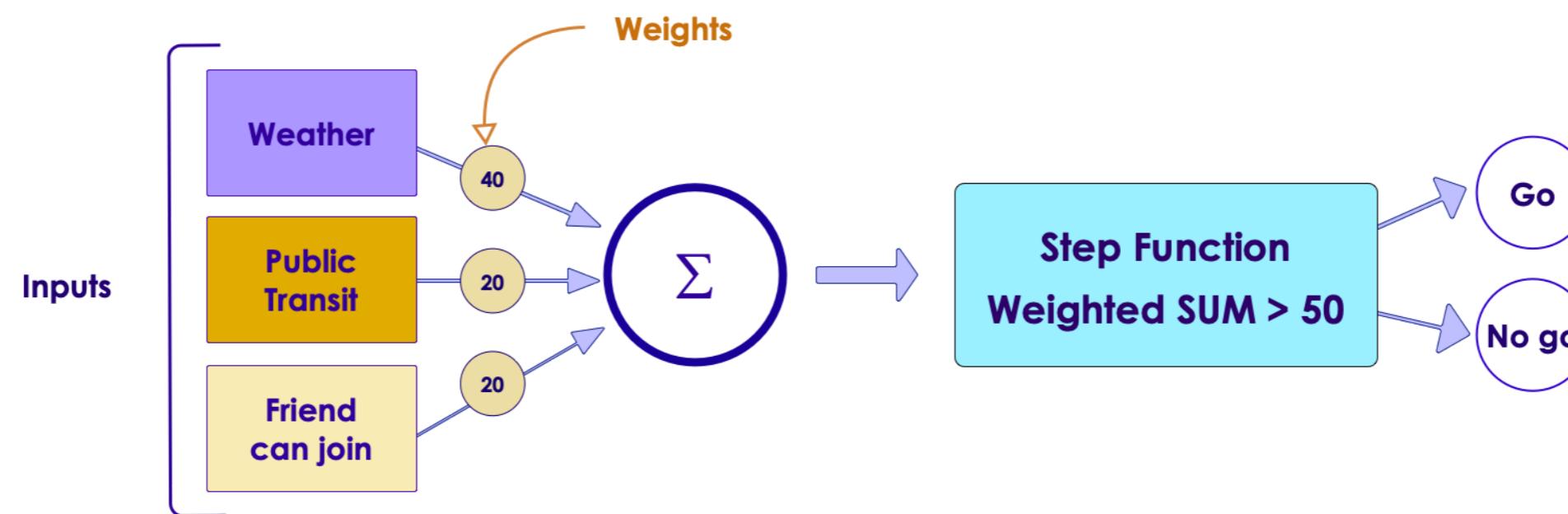
Simple Perceptron Example

- Design a perceptron that will decide if I should go to a concert, based on a few inputs
- Inputs: 1) Weather, 2) Close to public transit, 3) friend can join
- Assign weights to each of the above inputs
- Output: YES / NO
 - If the final score is > 50, then the answer is YES, otherwise NO



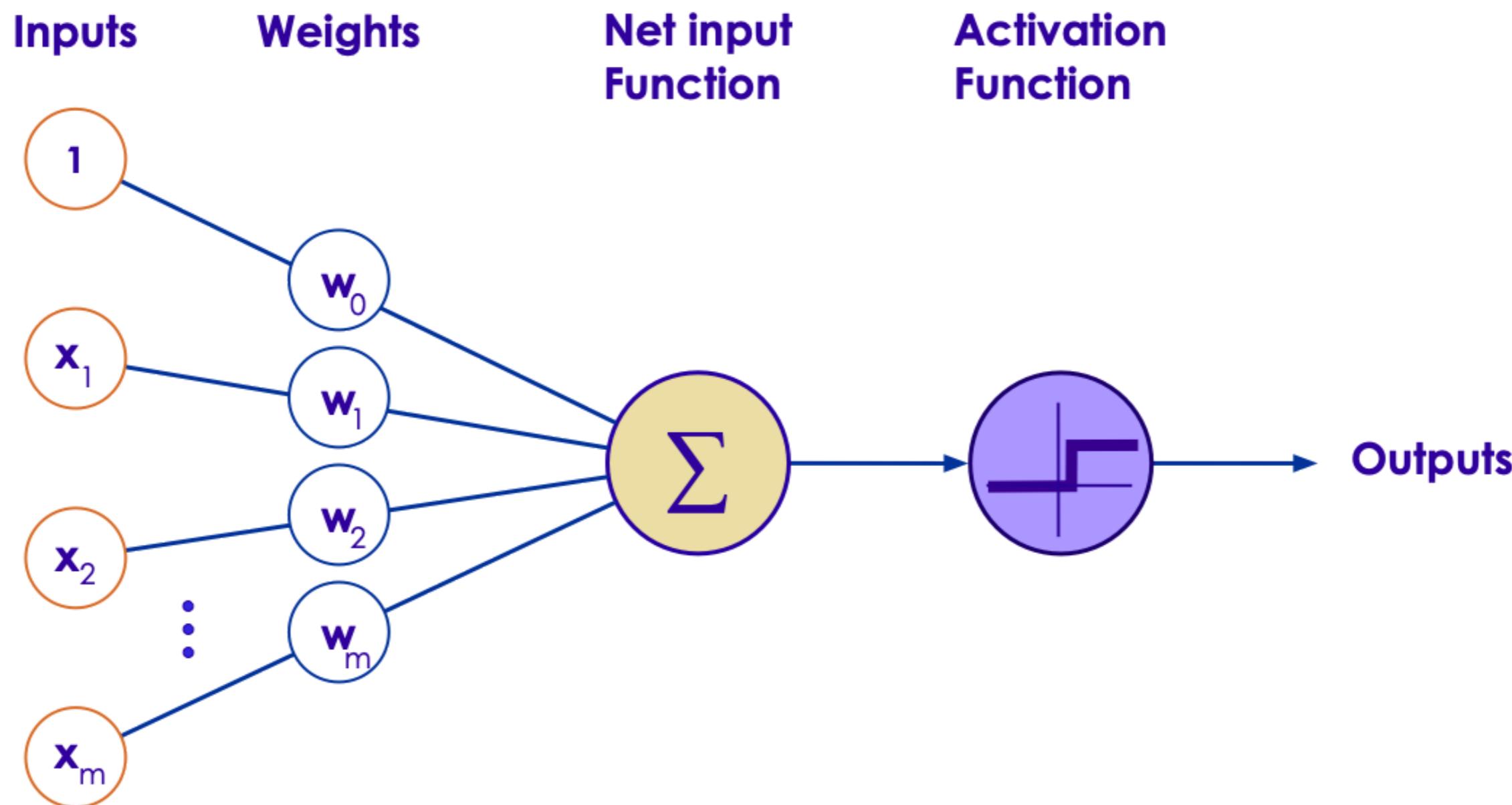
Simple Perceptron Example

- What is the outcome if
 - Weather is good
 - And a friend can join?
- What is the outcome if
 - You can go by public transit
 - And a friend can join
- Please note, here the weights are assigned manually



A Generalized Perceptron

- Here we are adding more inputs (X_1, X_2, \dots, X_m)
- Each input has their weights (W_1, W_2, \dots, W_m)
- The input '1' and weight 'W₀' is bias term



Perceptron Operations

- Step 1: Calculate sum of inputs and weights

$$z = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n$$

- In matrix operations this is:

$$w^T \cdot x$$

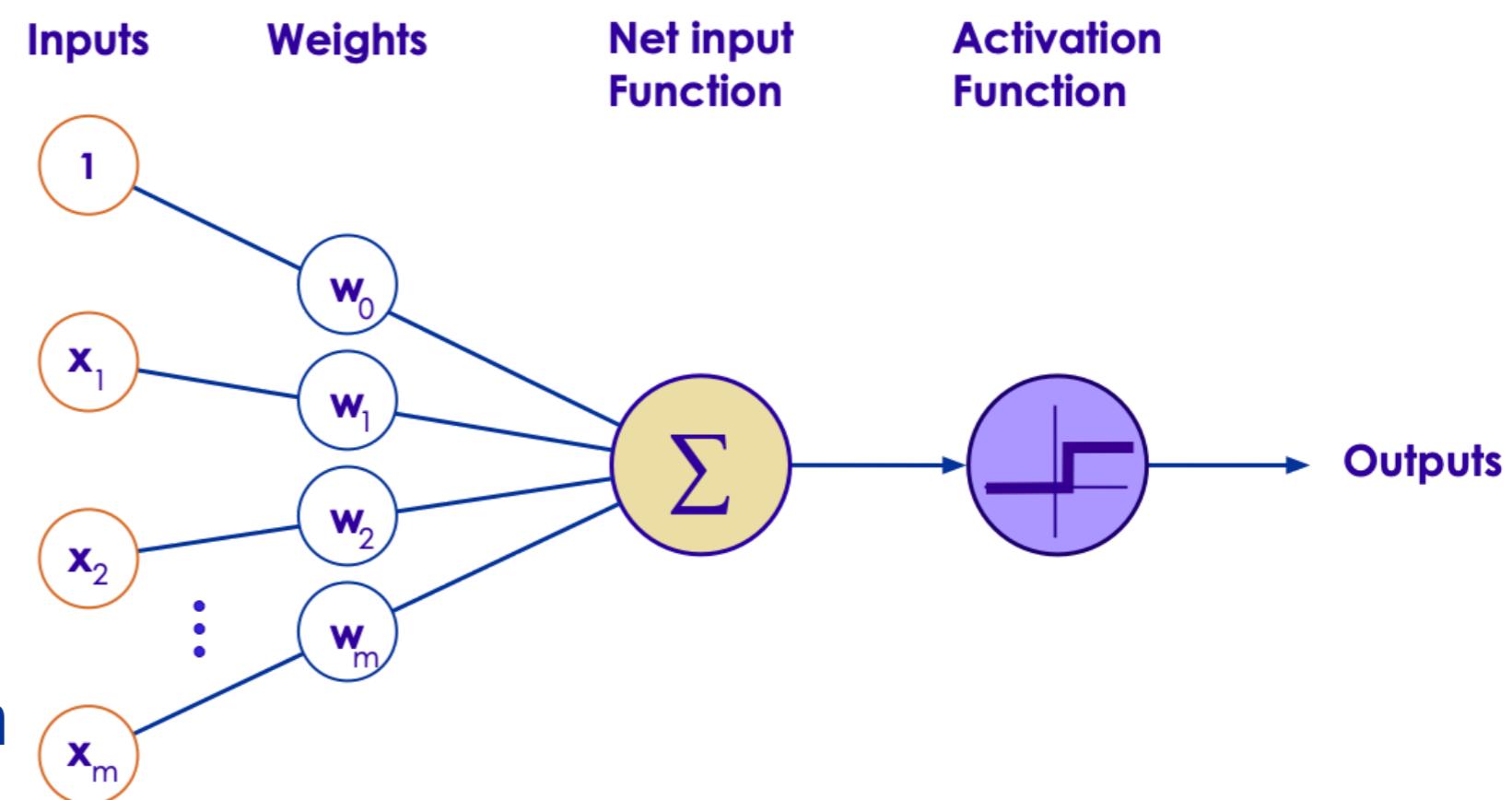
- Step 2: Apply Step function to the sum

$$h_w(x) = \text{step}(z)$$

$$h_w(x) = \text{step}(w^T \cdot x)$$

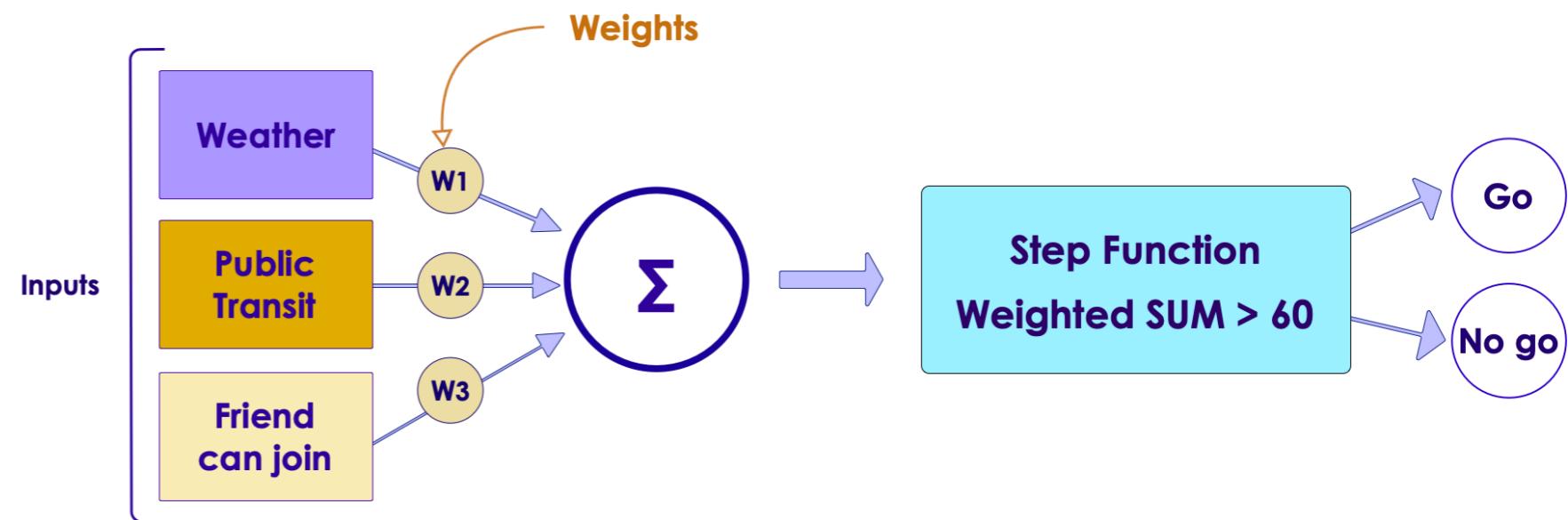
- Simple step function

- if sum is positive ($z \geq 0$) --> output is 1
- otherwise, output is 0



Quiz: Guessing the Weights

- Let's revisit our 'concert going' perceptron.
- Here is some new training data.
- Can you guess the weights w_1 / w_2 / w_3 to match the training data?
- (Threshold is now changed to 60)



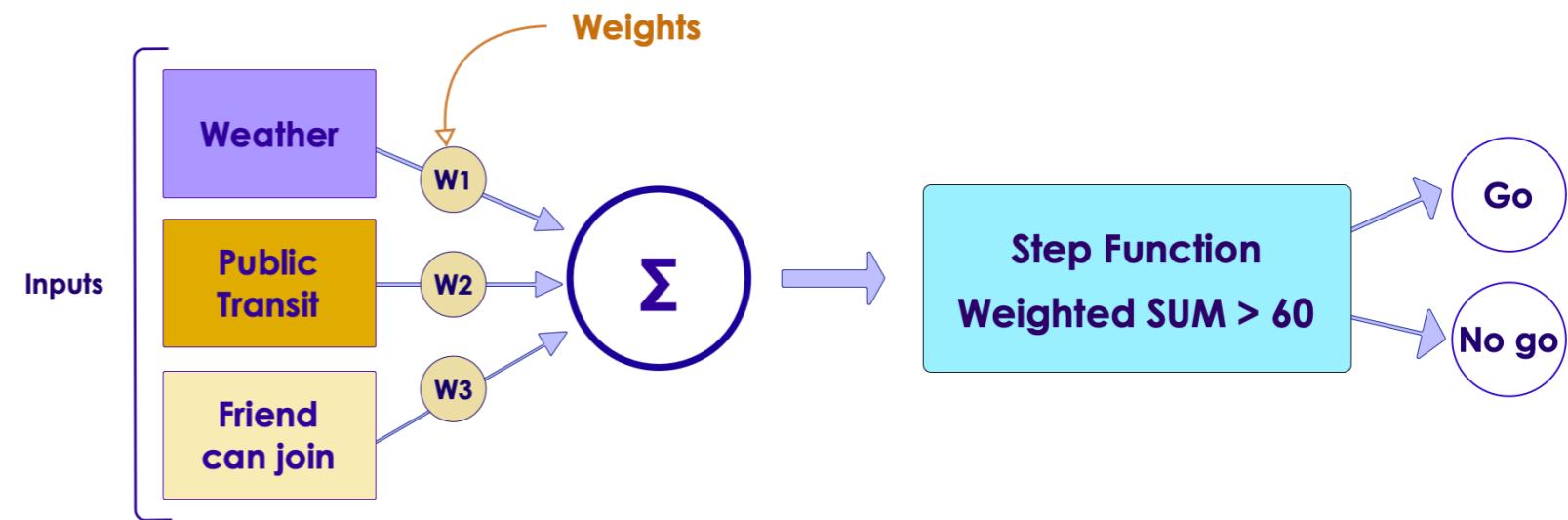
Weather	Public Transit	Friend Can Join	Outcome (0 / 1)
1	1	1	1
1	0	1	1
1	0	0	0
1	1	0	0
0	1	1	0
0	1	0	0
0	0	1	0

Quiz: Guessing the Weights

- Let's start with equal weights for all inputs:

weather = 33, public transit = 33,
friend = 33

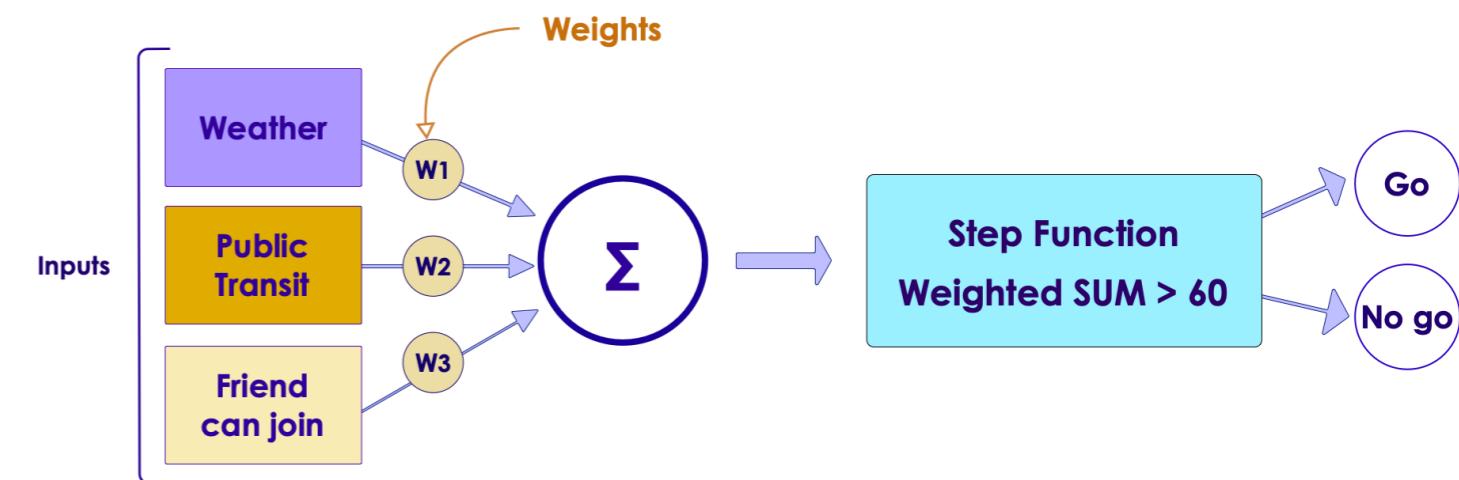
- We got 2 wrong!
- Can we do better?



Weather	Public Transit	Friend Can Join	Total	Predicted (> 60)	Actual
1 * 33	1 * 33	1 * 33	99	1 - ok	1
1 * 33	0 * 33	1 * 33	66	1 - ok	1
1 * 33	0 * 33	0 * 33	33	0 - ok	0
1 * 33	1 * 33	0 * 33	66	1 - wrong	0
0 * 33	1 * 33	1 * 33	66	1 - wrong	0
0 * 33	1 * 33	0 * 33	33	0 - ok	0
0 * 33	0 * 33	1 * 33	33	0 - ok	0

Quiz: Guessing the Weights

- Looks like 'public transit' isn't as important as 'friend'
- And 'weather' seems important
- Let's adjust the weights to reflect this
- weather = 35, public transit = 20, friend = 30
- we got all right !!**



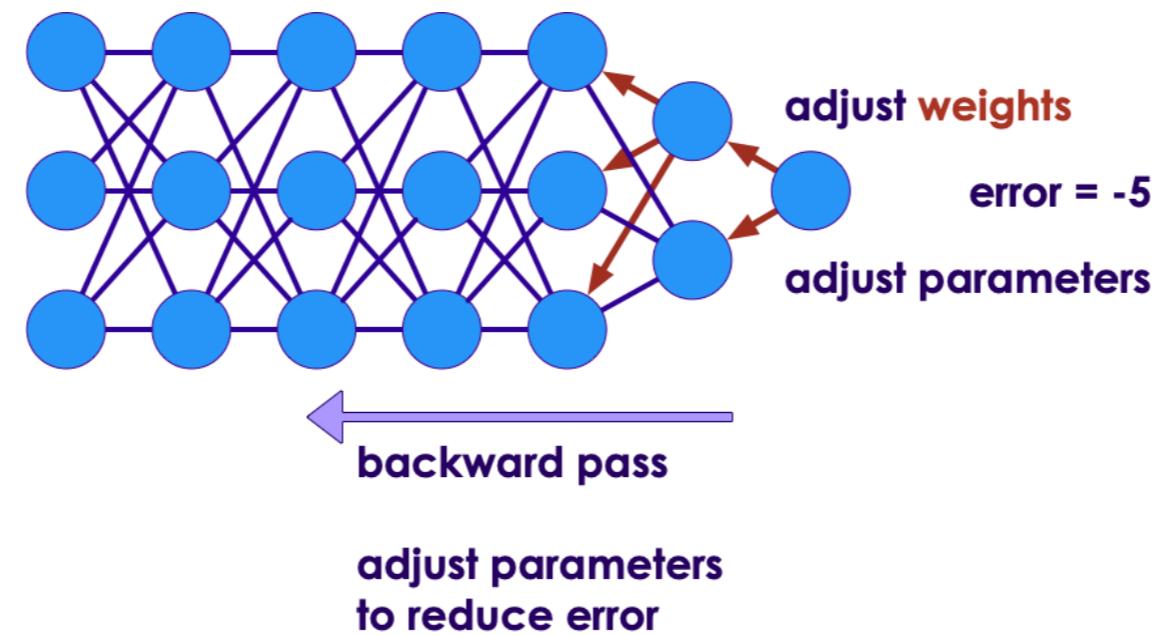
Weather	Public Transit	Friend Can Join	Total	Predicted (> 60)	Actual
1 * 35	1 * 20	1 * 30	85	1 - ok	1
1 * 35	0 * 20	1 * 30	65	1 - ok	1
1 * 35	0 * 20	0 * 30	35	0 - ok	0
1 * 35	1 * 20	0 * 30	55	0 - ok	0
0 * 35	1 * 20	1 * 30	50	0 - ok	0
0 * 35	1 * 20	0 * 30	20	0 - ok	0
0 * 35	0 * 20	1 * 30	30	0 - ok	0

How do We Train?

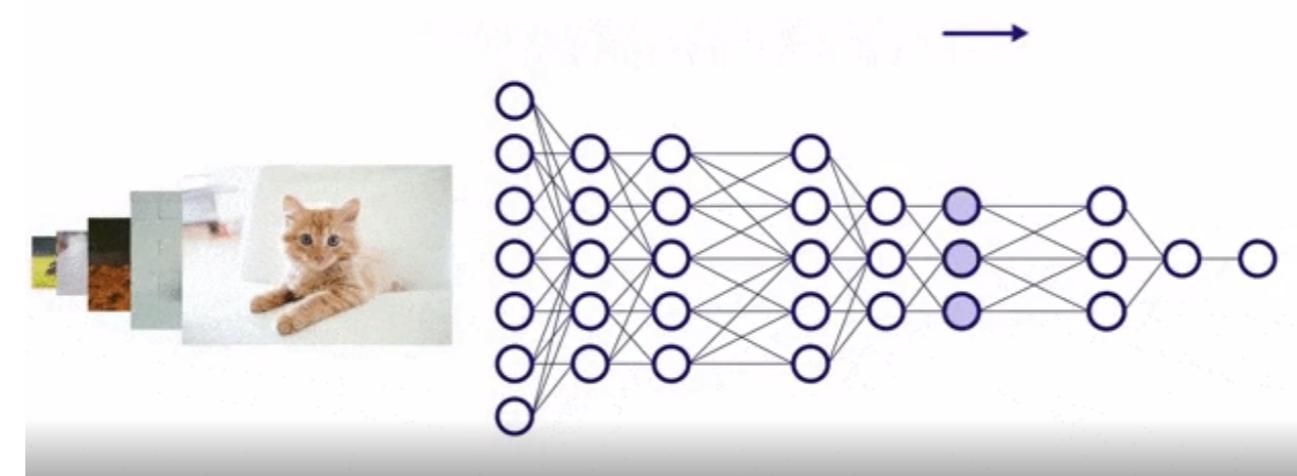
- In the last example, we lucked out by guessing the weights in 2 rounds (2 iterations)
- But for large complex datasets, we need a more systematic way of calculating and adjusting weights
- We use a technique called **back propagation**

Backpropagation Demos

- Animation (Regression) : [link-youtube](#), [link-S3](#)



- Animation (Classification) : [link-youtube](#), [link-S3](#)



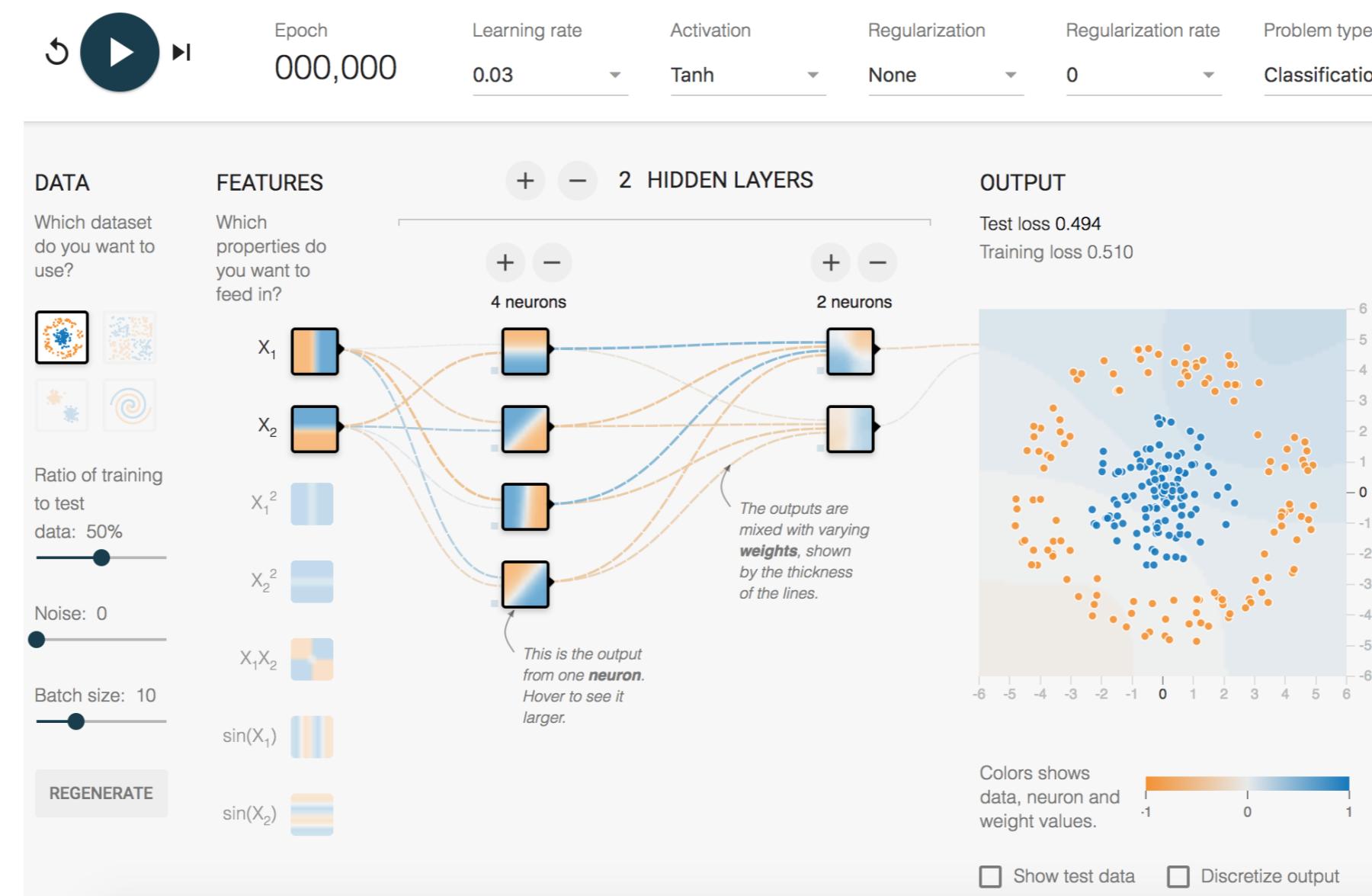
Tensorflow Playground

- For Instructor:
 - Go over the following section as needed and if time permits

Introduction to Tensorflow Playground

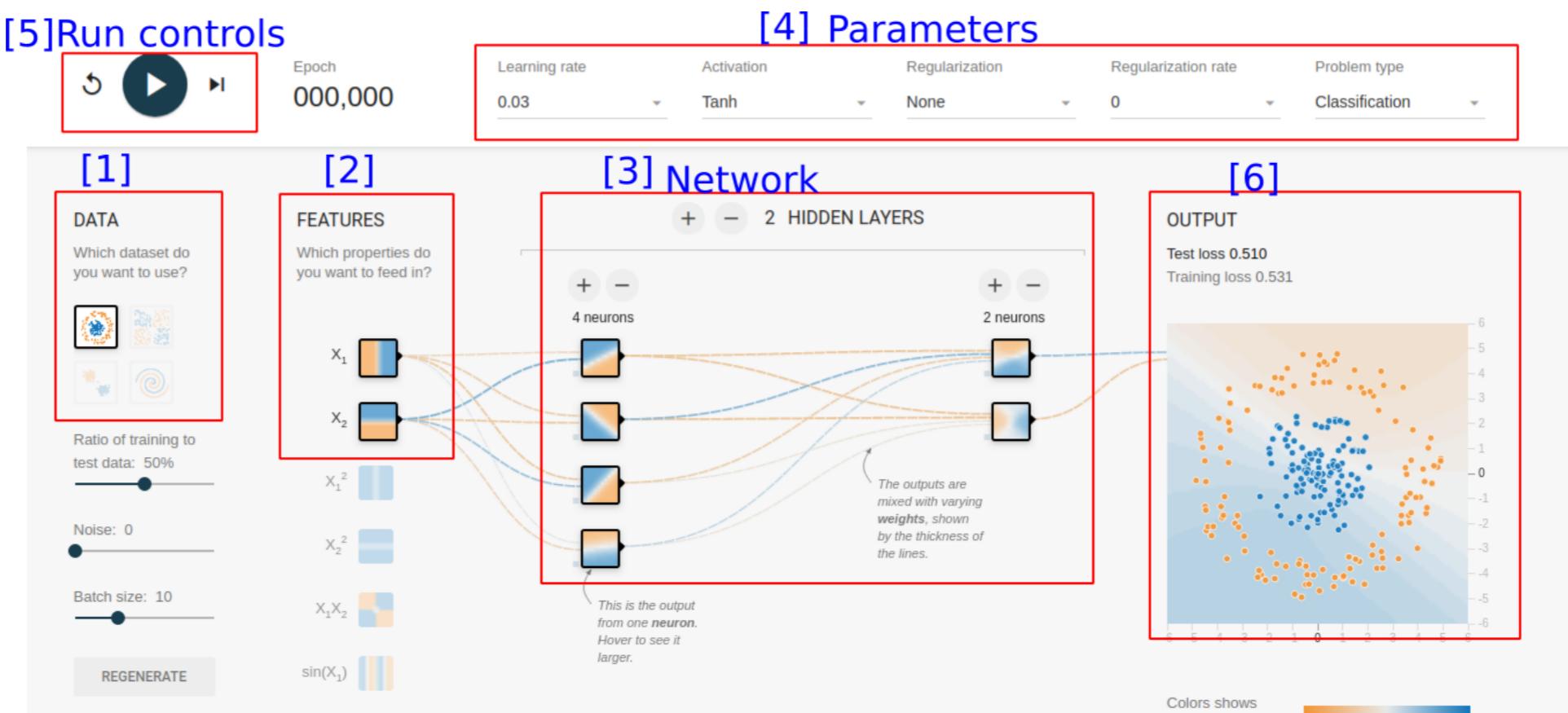
Introducing the Playground

- Navigate in your browser to <http://playground.tensorflow.org>
- This is a playground that we will use to play with some concepts
- It will be fun!
- When you start, you should see this



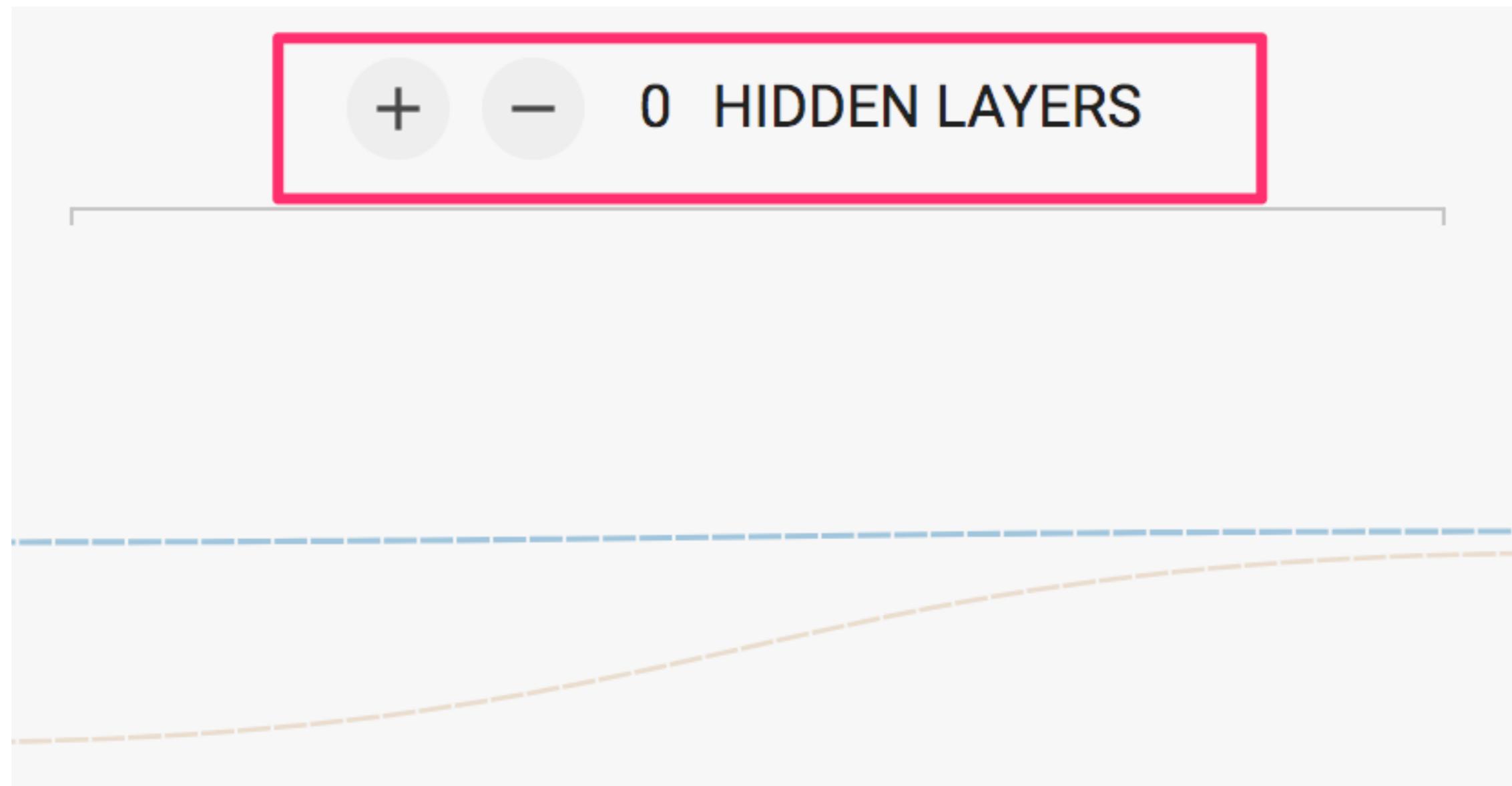
Playground Overview

- Step 1: Select data
- Step 2: Select features
- Step 3: Design neural network
- Step 4: Adjust parameters
- Step 5: Run
- Step 6: Inspect the results



Hidden Layers

- We will start out with **no** hidden layers
- Click the "minus" icon to get to no hidden layers



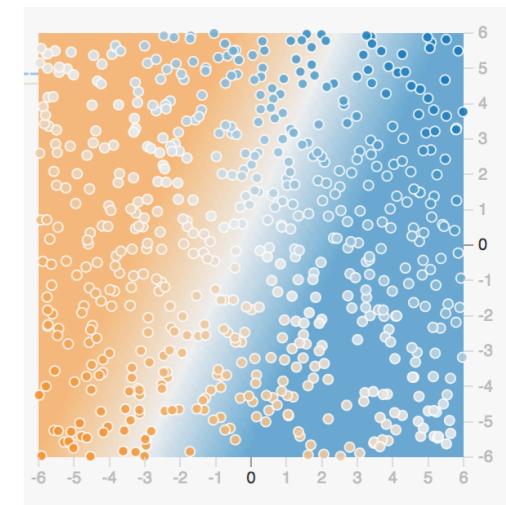
Playground Linear Regression

Linear Regression: Setup

- Click on the dropdown at the upper right, select 'Regression'



- Select the dataset in lower left



- Select the **lowest** setting of Learning Rate



Linear Regression: Parameters

Learning rate	Activation	Regularization	Regularization rate	Problem type
0.00001	Linear	None	0	Regression

- Learning Rate
 - This is the "step size" we use for Gradient Descent
- Activation Function
 - This is what we do to the output of the neuron
 - More on this later.
- Regularization / Regularization Rate
 - L1 / L2 are penalties to help reduce overfitting
 - How much to add

Linear Regression: Run!

- Let's try pressing the PLAY Button
- Look at the "Output" curve:
- **TOO SLOW!!! (Why??)**
 - How long (how many epochs) does it converge?
 - Do you ever get to loss = 0.0?
- What is the meaning of "loss?"
 - It's another way of saying "error"
 - In this case, it's the RMSE (Root Mean Squared Error)
- Is this dataset linearly separable?
 - Is it **possible** to get to zero loss?



Linear Regression: Adjust the Learning Rate

- Hit the reset button to the left of "play"
- Adjust the learning rate dropdown to something higher.
- Try hitting play again.
- What happens if you set a really **big** rate?
 - Note the loss is NaN (Not a Number)
 - The data is only -6.0 to +6.0.
 - A "big" value causes overshoot
- Challenge: What is the "optimal" learning rate?
 - Get to zero loss in the fewest epochs.



OUTPUT

Test loss NaN

Training loss NaN

Lab Review

- What is the impact of 'learning rate'
 - how does it affect convergence



Classification Examples 1

Linear Classification: Setup

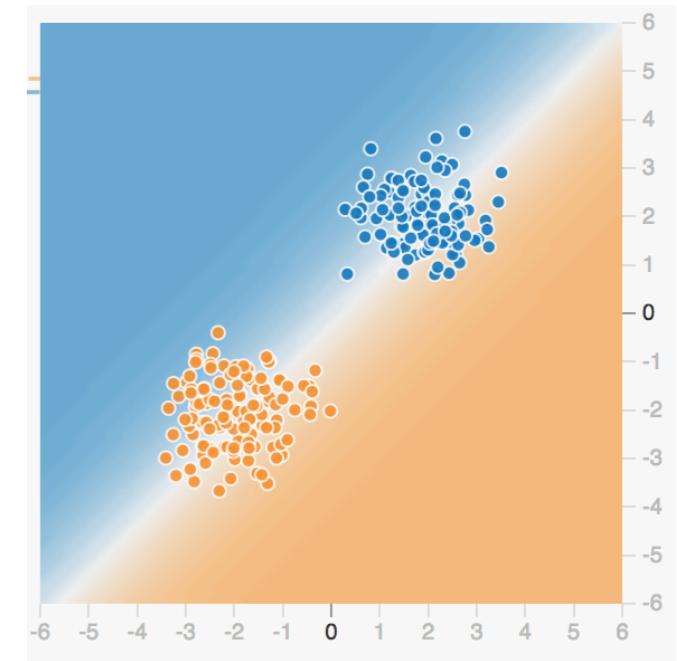
Learning rate	Activation	Regularization	Regularization rate	Problem type
0.03	Tanh	None	0	Classification

- Parameters

- Select 'Classification' on the dropdown at the upper right
- Activation : Tanh
- Learning Rate: 0.01

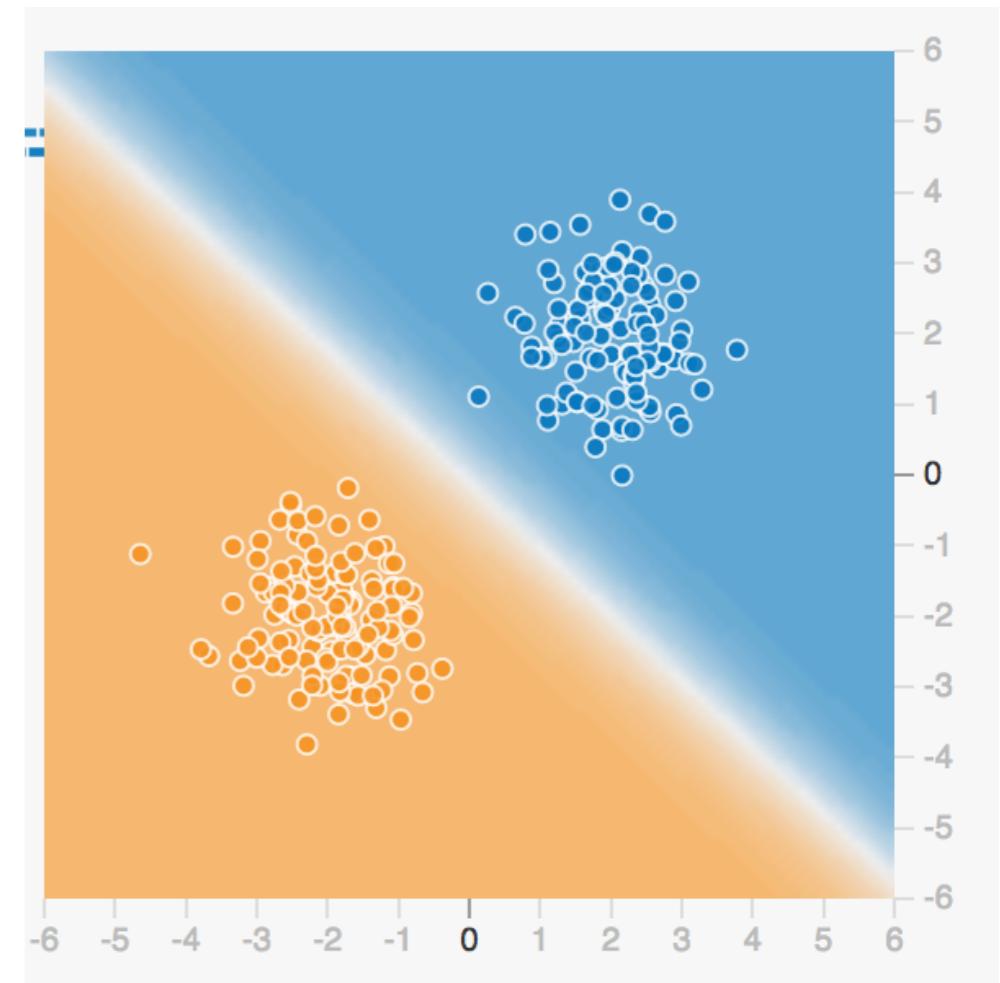
- Select the Two-Blob Datasets

- Is this dataset linearly separable?



Linear Classification: Run

- The separated dataset might look like below
- You may not get zero loss, especially if you introduce noise
- Challenge: Adjust the learning rate to get to minimum loss in as few epochs as possible.



Lab Review

- Why didn't we need hidden layers to converge on a solution?
- What would happen if the dataset wasn't linearly separable?

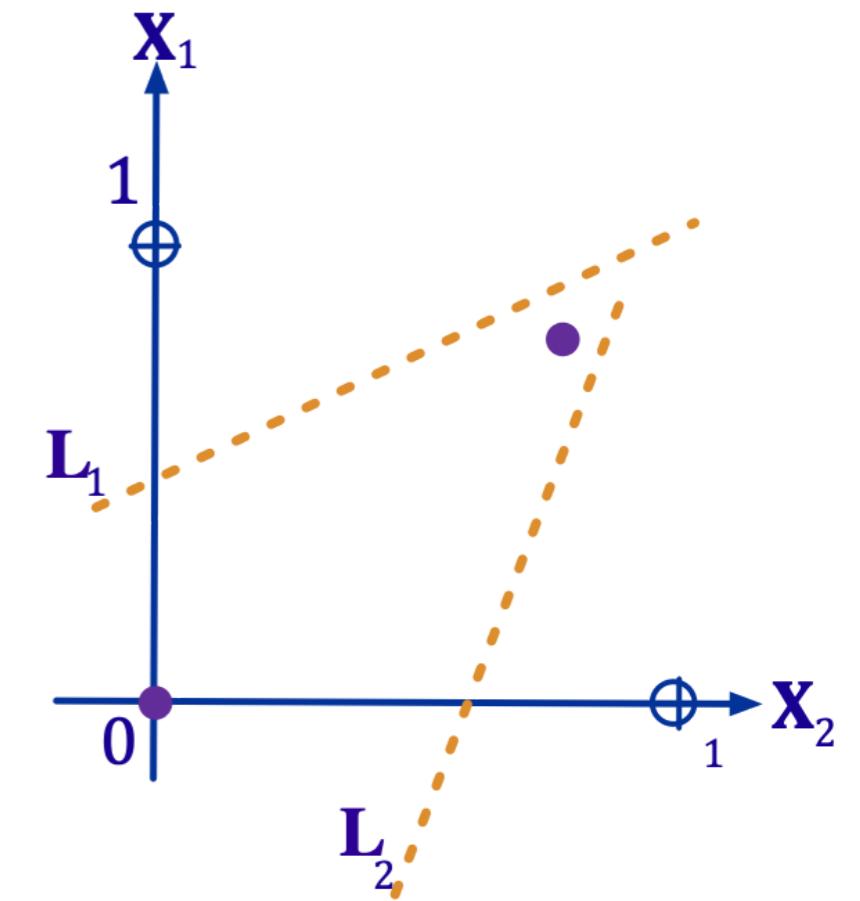


Hidden Layers

The XOR problem

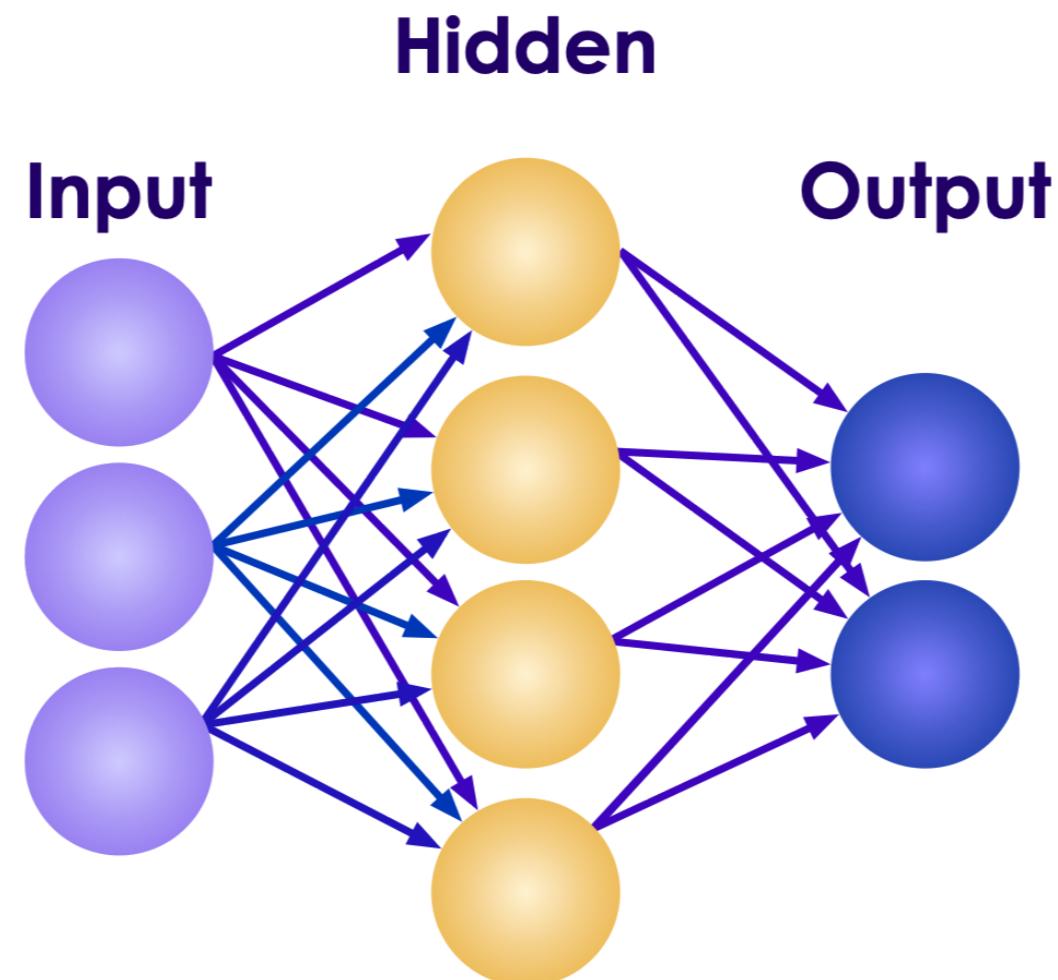
- What happens if we don't have linear separability?
- For example, can we learn a function that does an exclusive or?
- There is no line that can separate these.
 - And so the single-layer perceptron will never converge.
 - This is known as the XOR problem (though many other datasets are not linearly separable).
- Solution : We need **hidden layers**

\mathbf{x}_1	\mathbf{x}_2	y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y = \mathbf{X}_1 \oplus \mathbf{X}_2$$


The Solution: a Hidden Layer

- Our problem is that our solution to a single layer neural network is linear.
 - We call the solution the "decision boundary"
 - What if we could create a nonlinear decision boundary?
 - How would we do that?
- What if we add a new layer to our network?



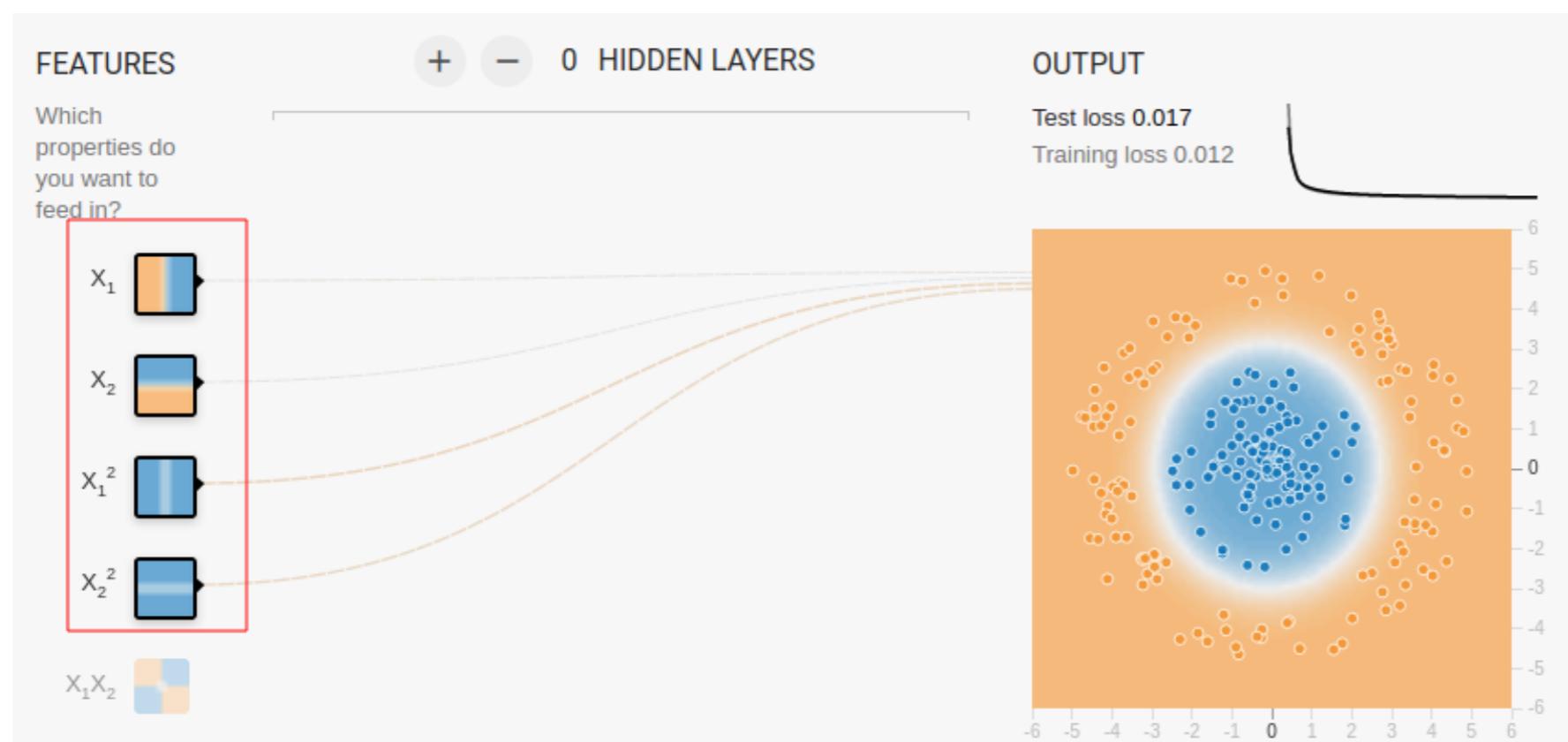
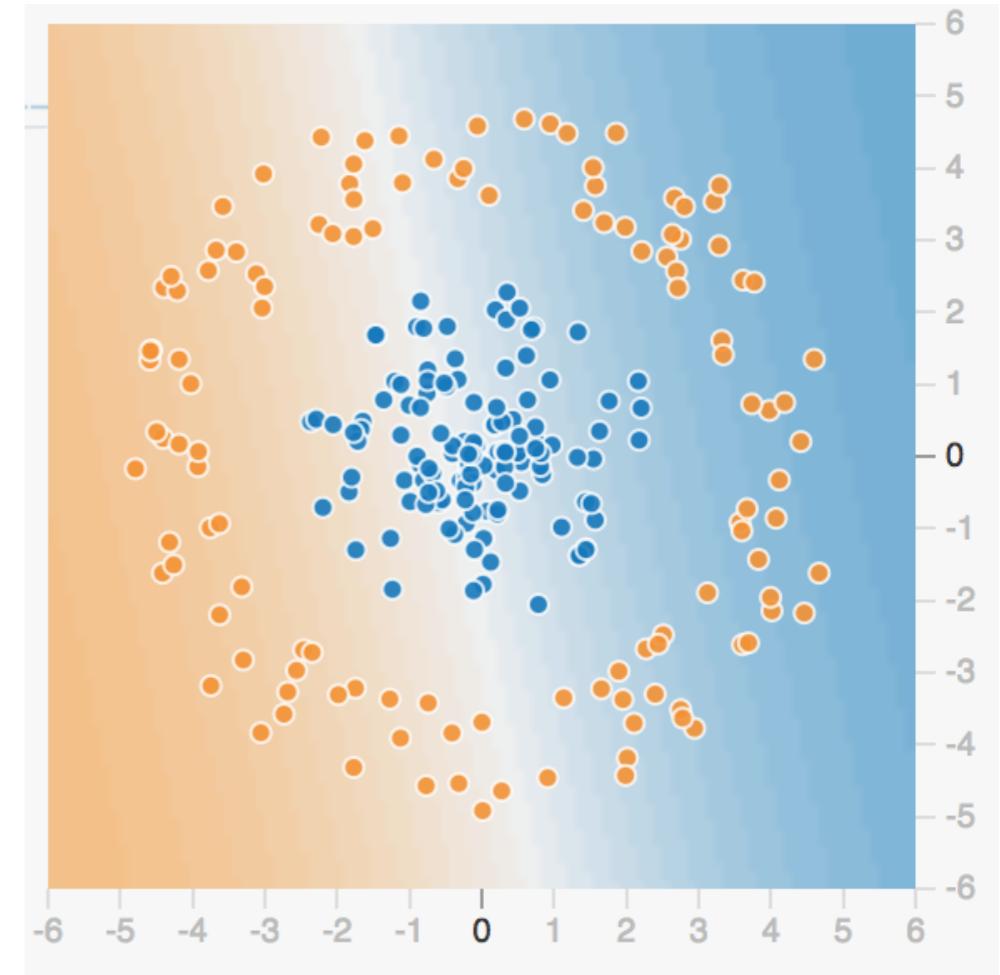
Why Hidden Layers

- Hidden Layers allow us to solve the "XOR" problem
 - Creating a nonlinear decision boundary
- How Many Hidden Layers?
 - Most nonlinear problems solvable with one hidden layer.
 - Multiple Hidden Layers allow for more complex decision boundaries
- One Hidden Layer is Enough
 - It has been proven that any function can be represented by a sufficiently large neural network with one hidden layer
 - Training that network may be difficult, however.
- But it's not enough
 - Current training methods mean that more than one layer is required in many cases.

Classification Examples 2

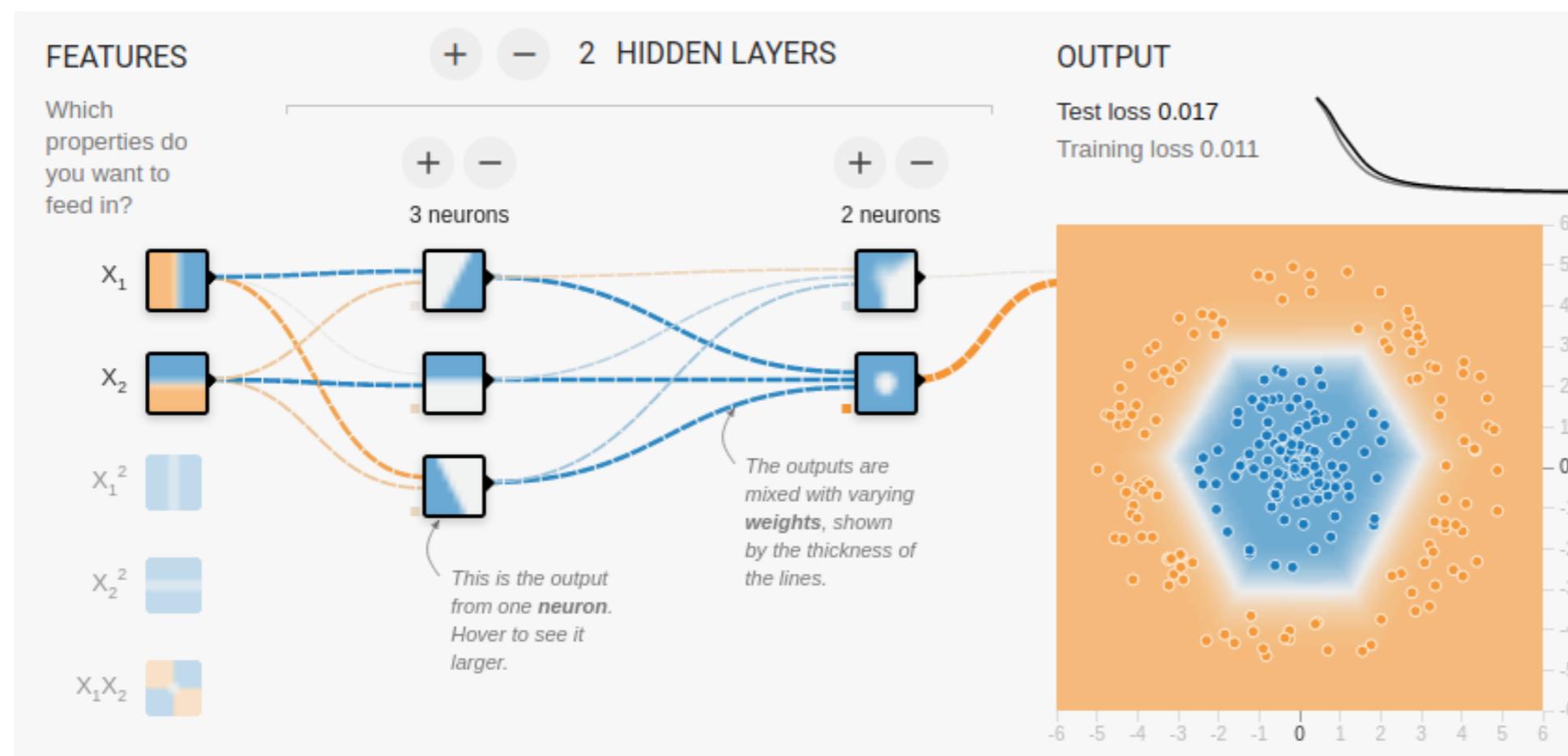
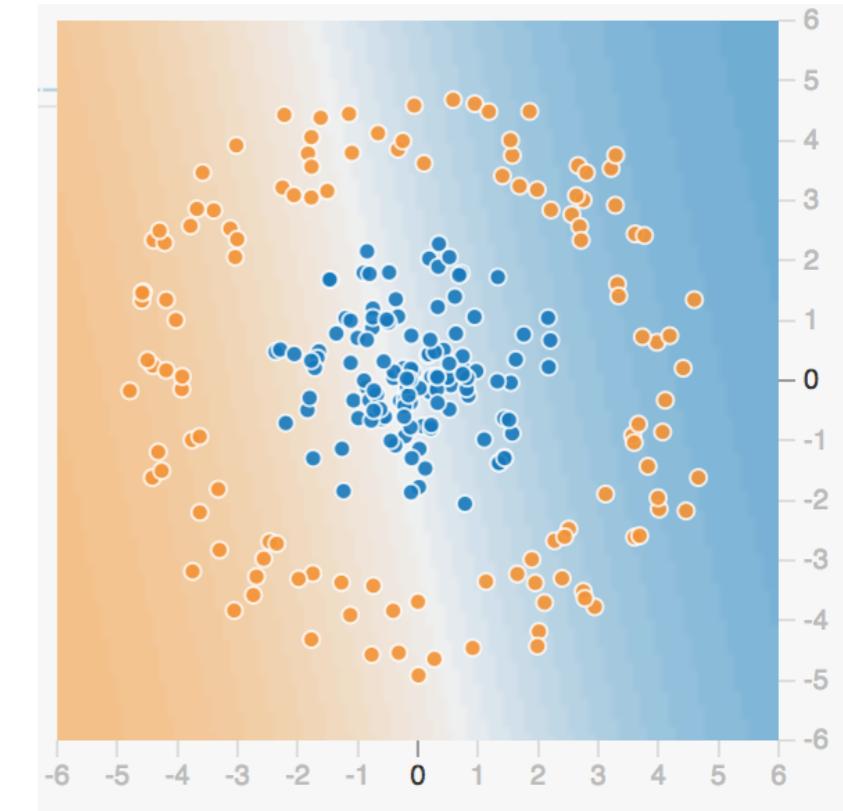
Circle Dataset

- Select the circle dataset
- Can we linearly separate this dataset?
 - No amount of fiddling with learning rate will help!
 - It's not linearly separable.
- Solution-1: Include other features
 - $x_1 + x_2 + x_1^2 + x_2^2$



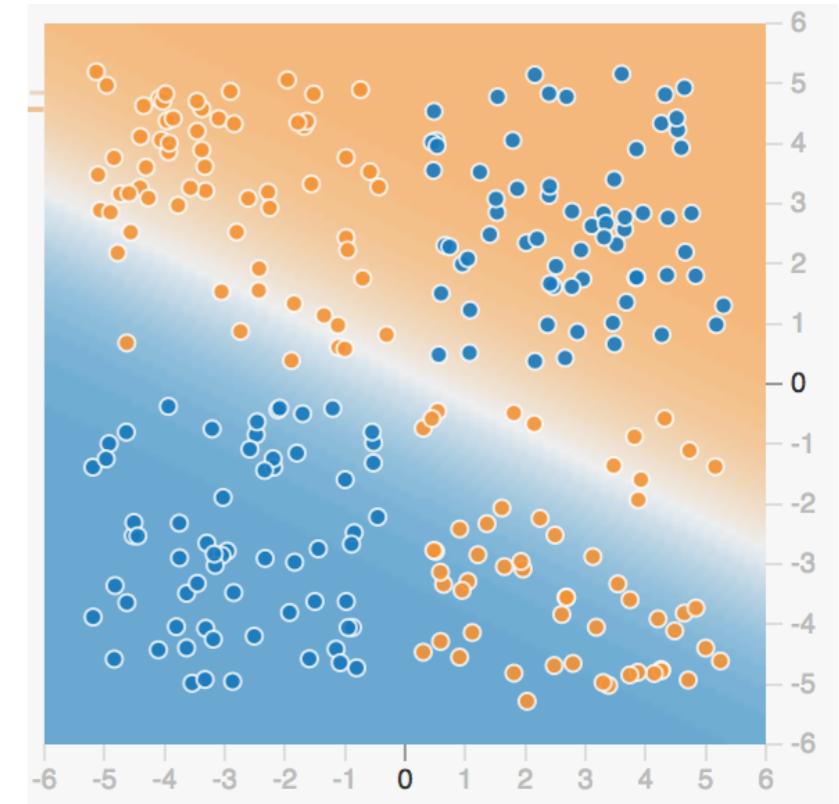
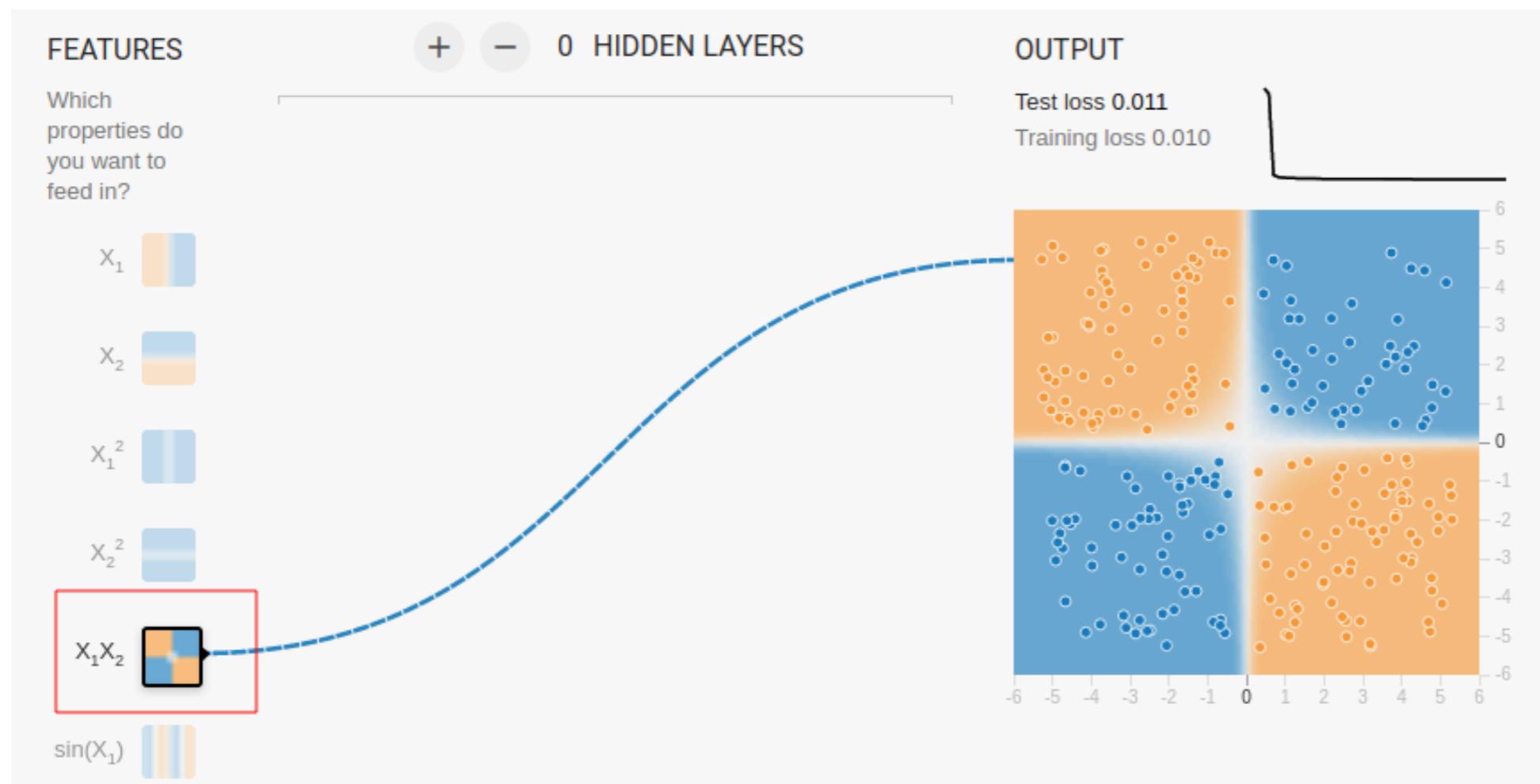
Circle Dataset With Hidden Layers

- Select the circle dataset
- Select only X_1 and X_2 as features
- Add a Hidden Layer
- Can you get a solution with 1 hidden Layer
- You can add more neurons to the hidden layer
- Can you solve it with only one hidden layer?
- If not, add another hidden layer



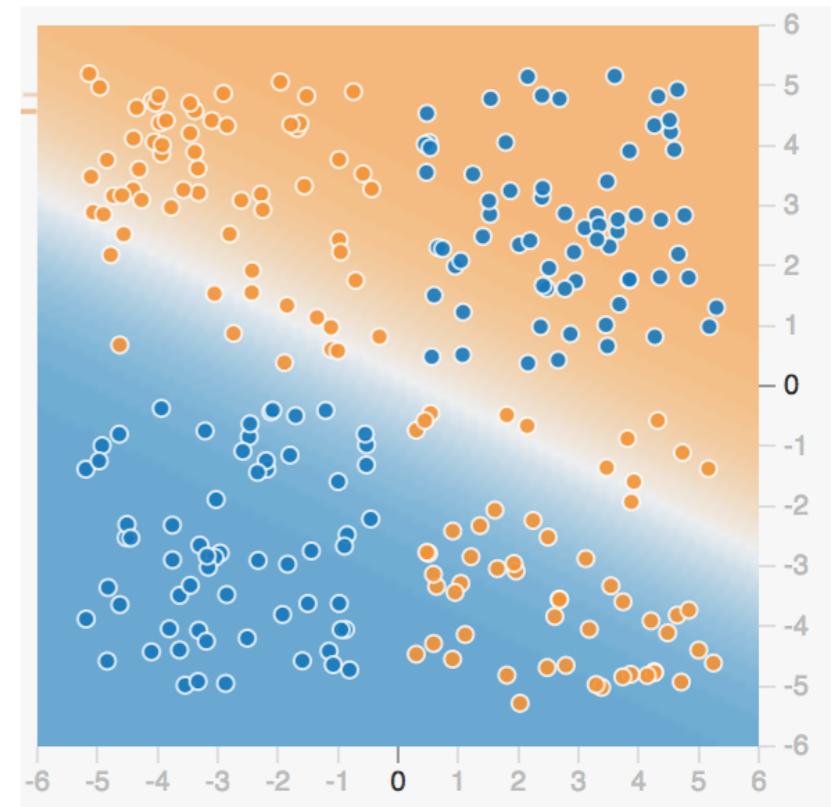
Four Square Dataset

- Set the Four-Square dataset
- Try setting the input to $X_1 . X_2$ with no hidden layers



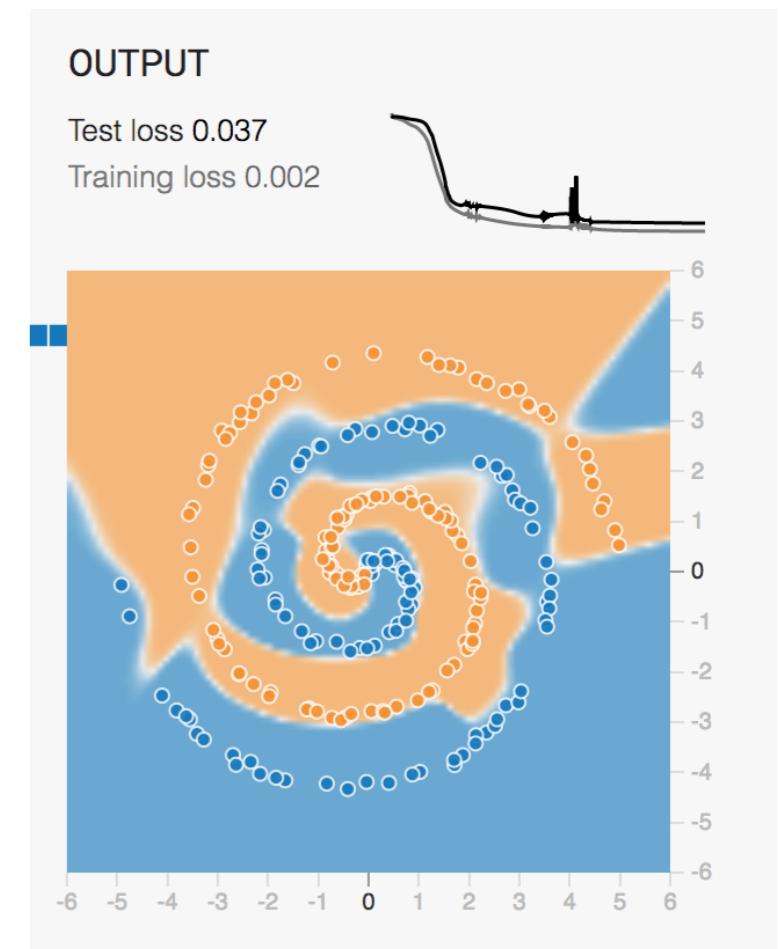
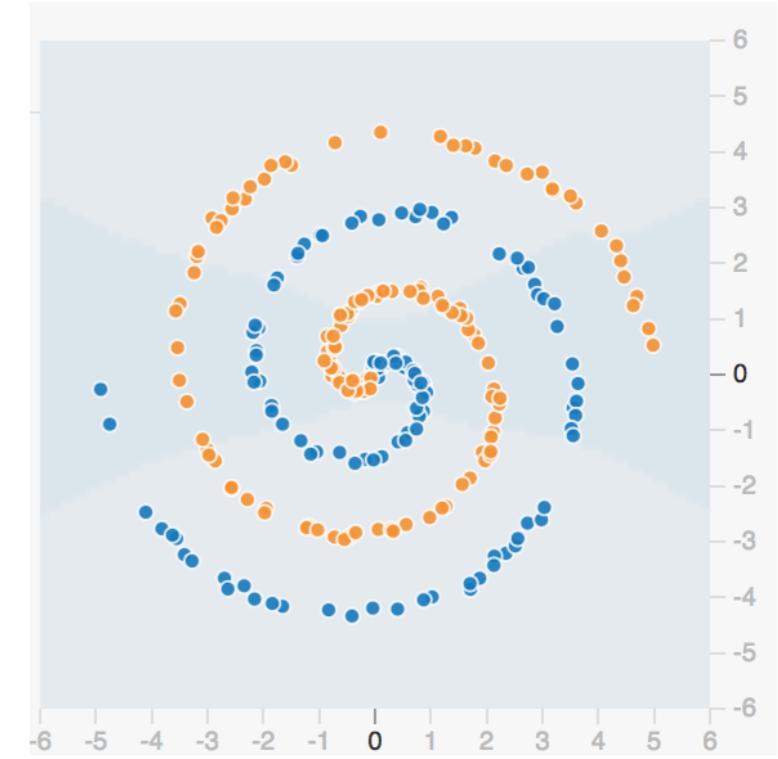
Four Square Dataset With Hidden Layers

- Solve the **Four Square** using hidden layers
- Set the inputs to **X1** and **X2**
- Try adding 1 or 2 hidden layers
- **Instructor** : Offer hints from the notes section



Spiral Dataset

- Set the Spiral dataset
- This is a **challenging** dataset
- Try
 - multiple features
 - multiple hidden layers
- Can you get this result?
- **Instructor** : Offer hints from the notes section



Lab Review

- What's the minimum number of hidden layers required to correctly classify all the test data?
- Does adding any additional features help at all?
- Do we necessarily get better results with more neurons and/or hidden layers?



Neural Network Design

Popular Neural Network Architectures

- Feedforward neural network (FFNN)
- Convolutional neural network (CNN)
- Recurrent neural network (RNN)
- We will cover these in detail in the upcoming sections

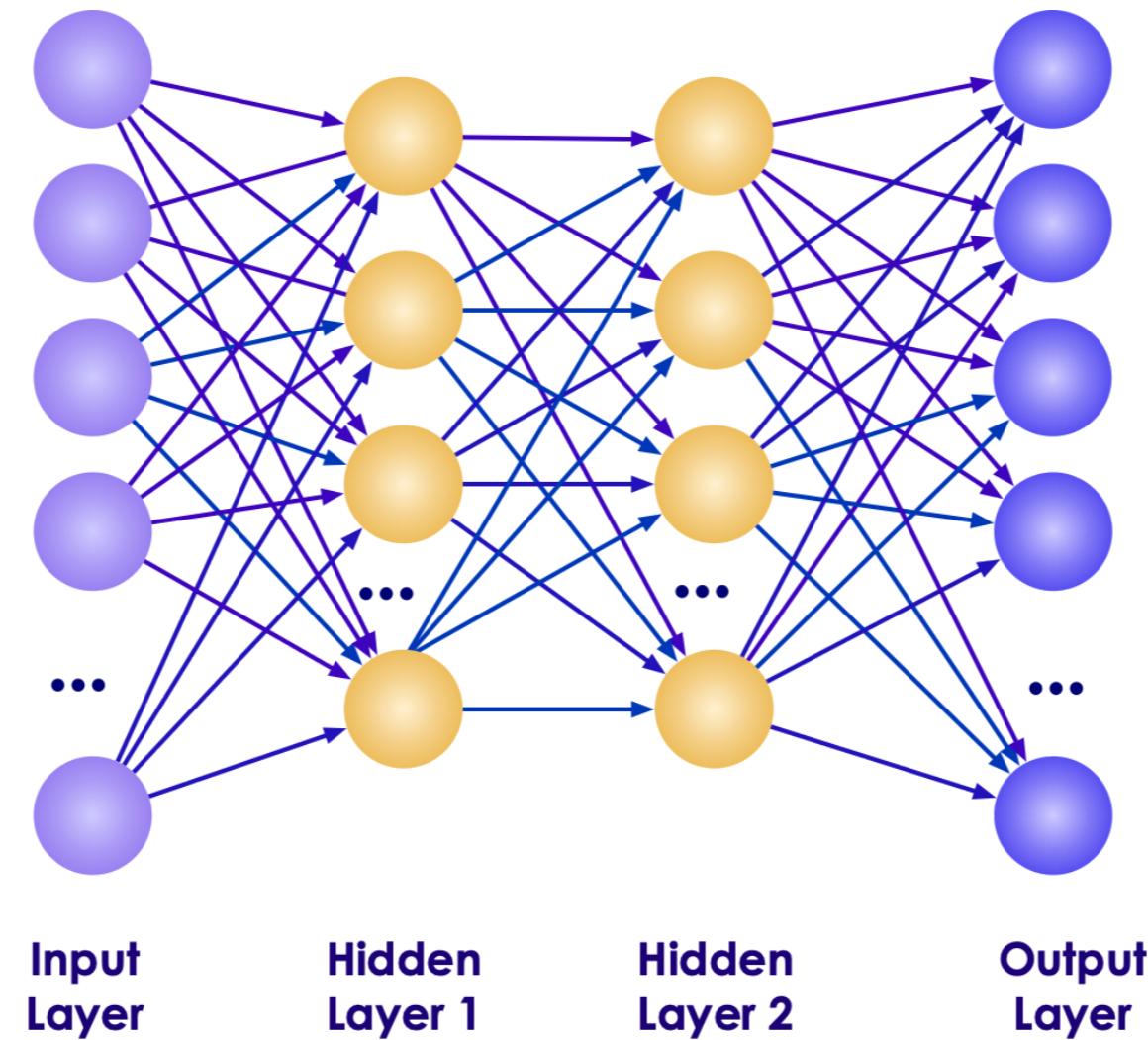
Feed Forward Neural Networks

Feedforward Neural Networks (FFNN)

- As we have seen earlier, single layer NNs can solve simple, linear problems
- To solve more complex problems (with non-linear solutions) we need a more complex setup
- This is where **Feedforward Networks** come in
- Also known as
 - **Multi Layer Perceptrons (MLP)**
 - **Deep Feedforward Neural Networks (DFNN)**

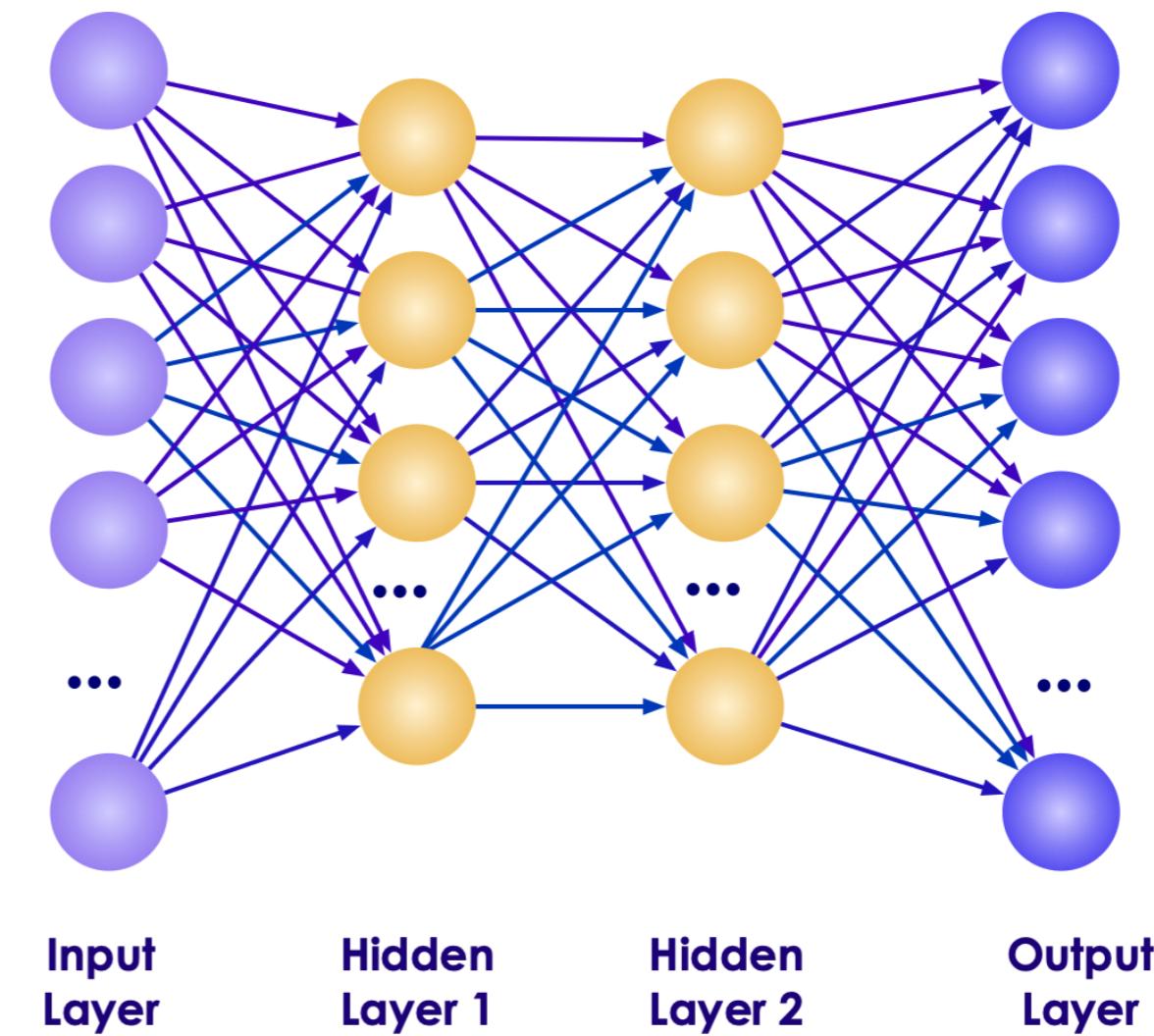
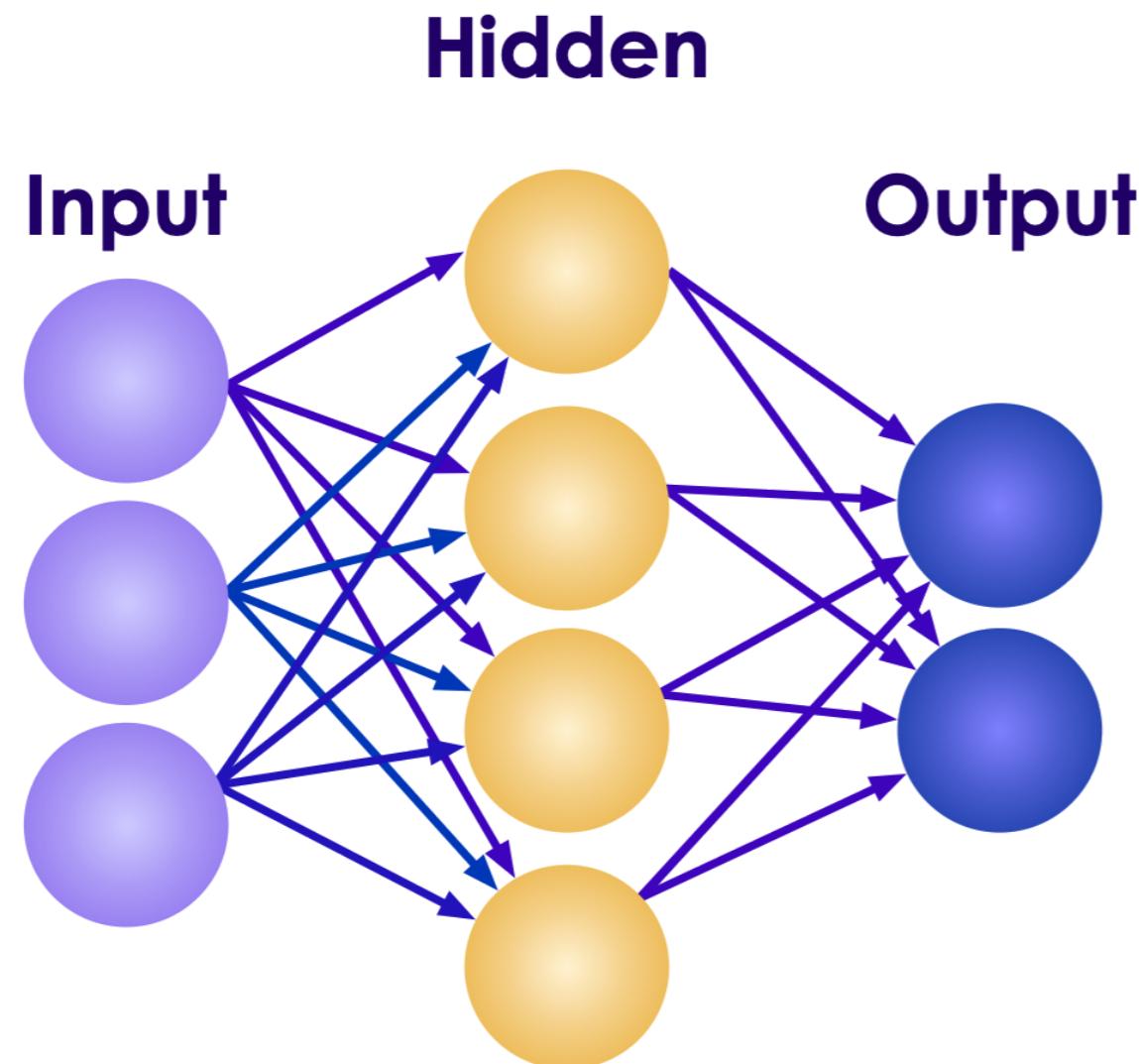
Feedforward Network Design

- There are multiple layers
- Each layer has many neurons
- Each neuron is connected to neurons on previous layer
- Information flows through ONE-WAY (no feedback loop)
- Composed of : Input, Output and Middle (Hidden) layers



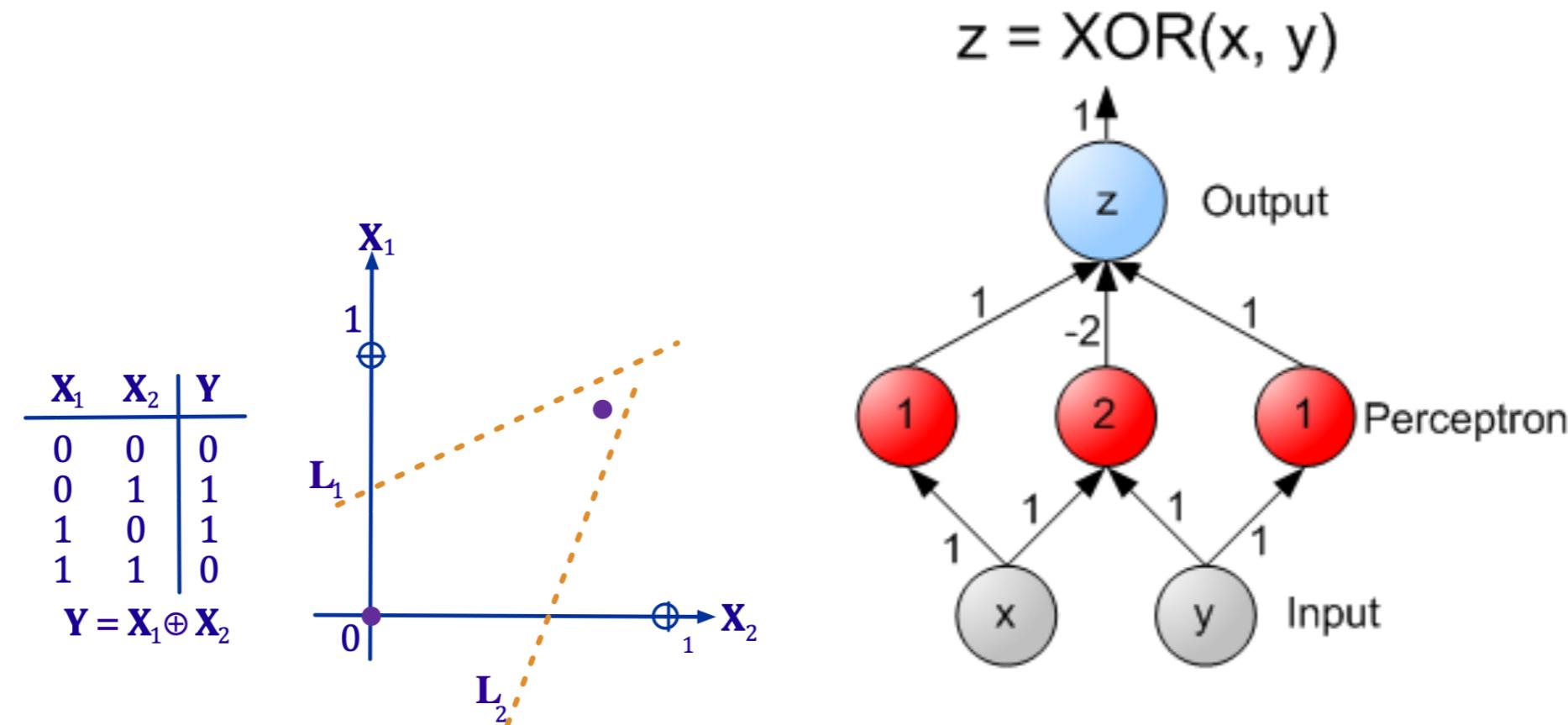
Hidden Layers

- Layers between Input and Output are called **Hidden Layers**
- If there is more than one hidden layers, it is called **Deep Neural Network**



Non-Linearity

- Hidden layers can model 'non-linear' functions
 - one hidden layer can model any continuous functions
 - two or more hidden layers can model discontinuous functions
- Remember the XOR problem? We couldn't solve it using single layer perceptron.
- But using hidden layers, we can solve it very easily

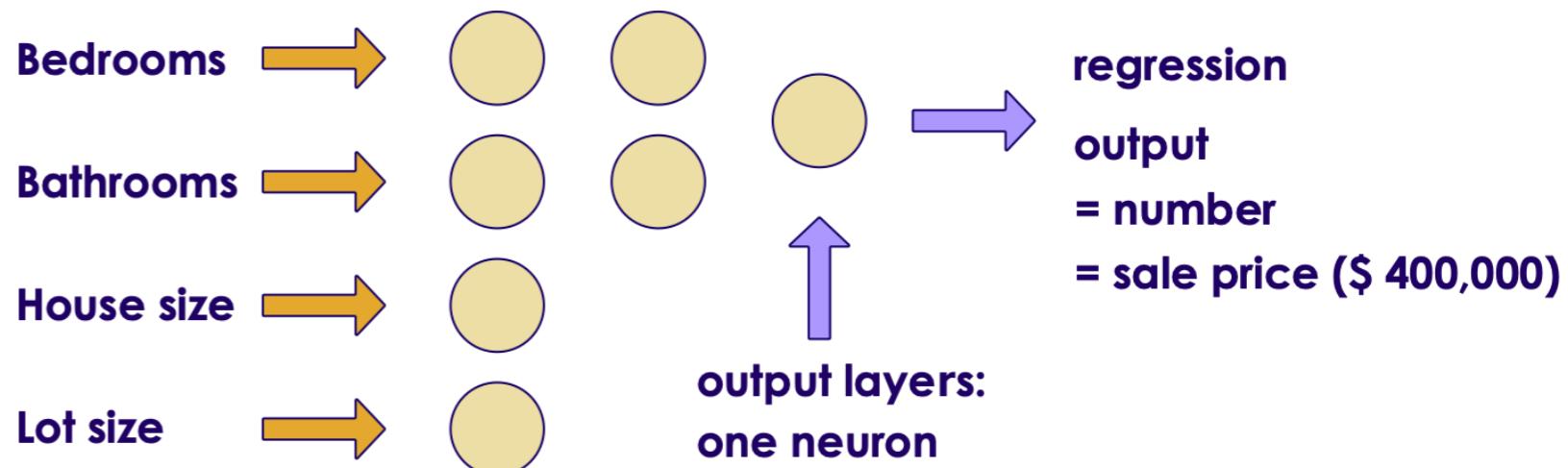


Sizing Neural Networks

Sizing a Regression Network

Bedrooms (input 1)	Bathrooms (input 2)	House Size (input 3)	Lot Size (input 4)	Sale Price (in thousands) (OUTPUT)
2	1	1200	2000	229
3	1	1500	3500	319
4	2	2300	5000	599

- Input layer sizing:
 - Match the number of input dimensions = 4 (bedrooms, bathrooms, house size, lot size)
- Output layer sizing:
 - Only one neuron
 - As we are predicting a number (sale price)
- Hidden layer sizing:
 - flexible



Sizing Binary Classification Network

Income (input 1)	Credit Score (input 2)	Current Debt (input 3)	Loan Approved (output)
40,000	620	0	0
80,000	750	100,000	1
100,000	800	50,000	1

- Input layer sizing

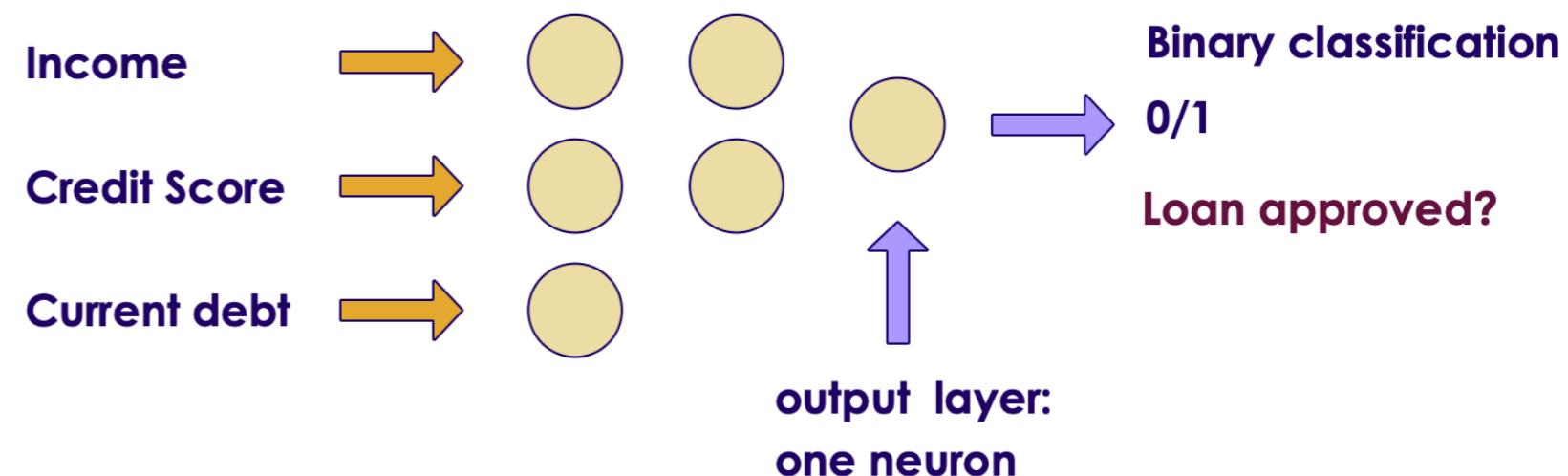
- Same as input dimensions = 3 (income, credit score, debt)

- Output layer sizing

- one neuron for 0/1 output

- Hidden layer sizing

- flexible



Sizing Multi Classification Network

- Input layer sizing

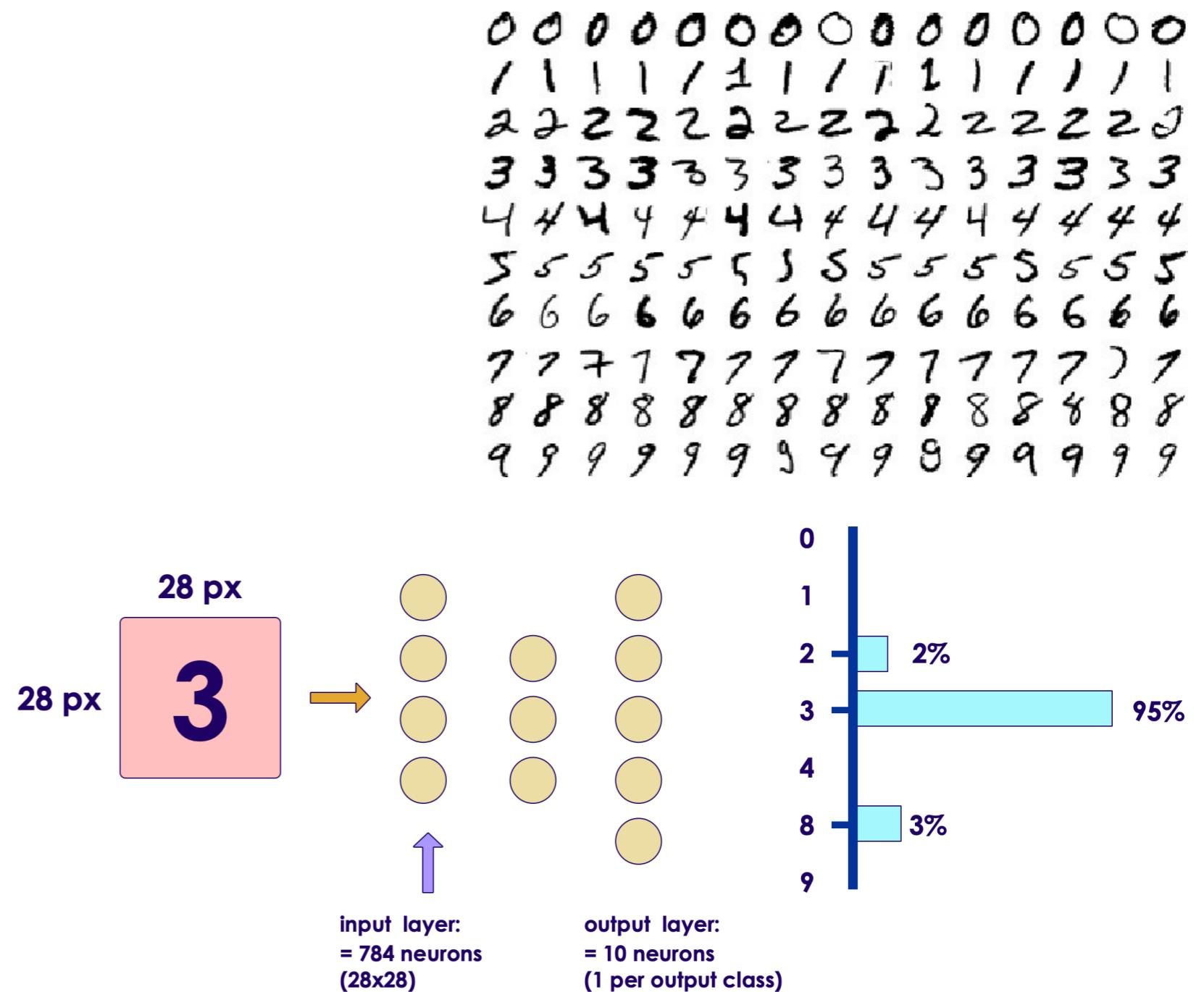
- Match input dimensions
- $784 = 28 \times 28$ pixels

- Output layer sizing

- Softmax layer
- one neuron per output class
- 10 (one for each digit; 0, 1, ..8,9)

- Hidden layer sizing

- flexible



Softmax Output

Output Class	0	1	2	3	4	5	6	7	8	9
Probability (Total 1.0)	0.0	0.0	0.02	0.95	0.0	0.0	0.0	0.0	0.03	0.0

- The Softmax function produces probabilities for output values.
- Here output is predicted as '3' as it has the highest probability (95%)
- The resultant array must add up to 1, because the output enumerates all probabilities

Feedforward Network Sizing

Summary

- **Input Layer**

- Size: Equal to Number of Input Dimensions
- Possibly add one extra neuron for bias term.
- What if we have thousands of sparse dimensions?
- Consider Wide and Deep Neural Network

- **Hidden Layer(s)**

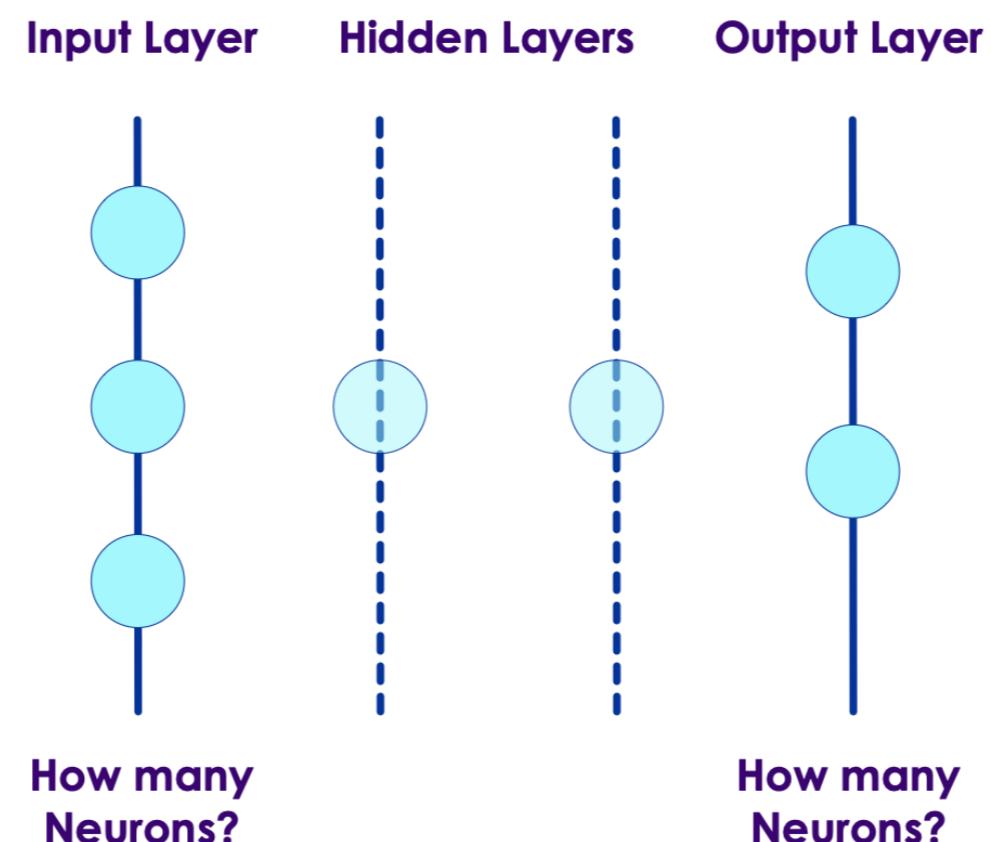
- Deep Learning= Multiple Hidden Layer (more than 2)
- Size depends on training sample, input features, outputs

- **Output Layer**

- Regression: 1 single neuron (continuous output)
- Binary Classification: 1 single neuron (binary output)
- Multi-class Classification: Softmax Layer
 - Size: 1 node per class label

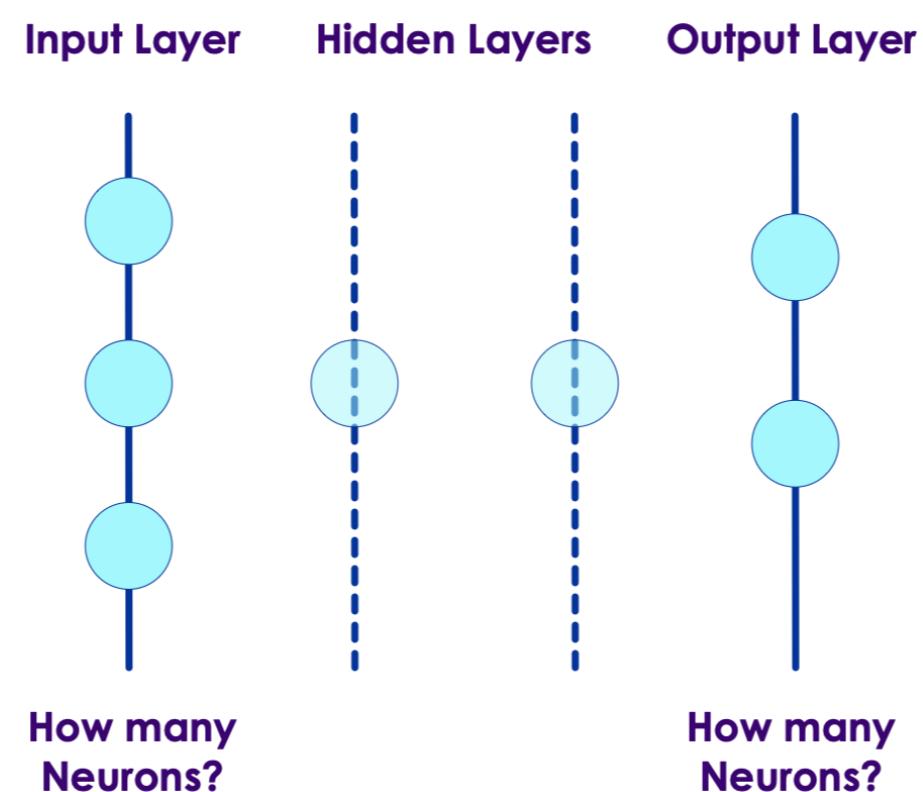
Quiz: Design Neural Networks

- Scenario 1: - We want to predict stock market index (e.g. DOW)
- Inputs: 30 features
- Output: price of index (single value - numeric)
- Is this a classification or regression?
- Design Input/Output layers of NN



Quiz: Design Neural Networks

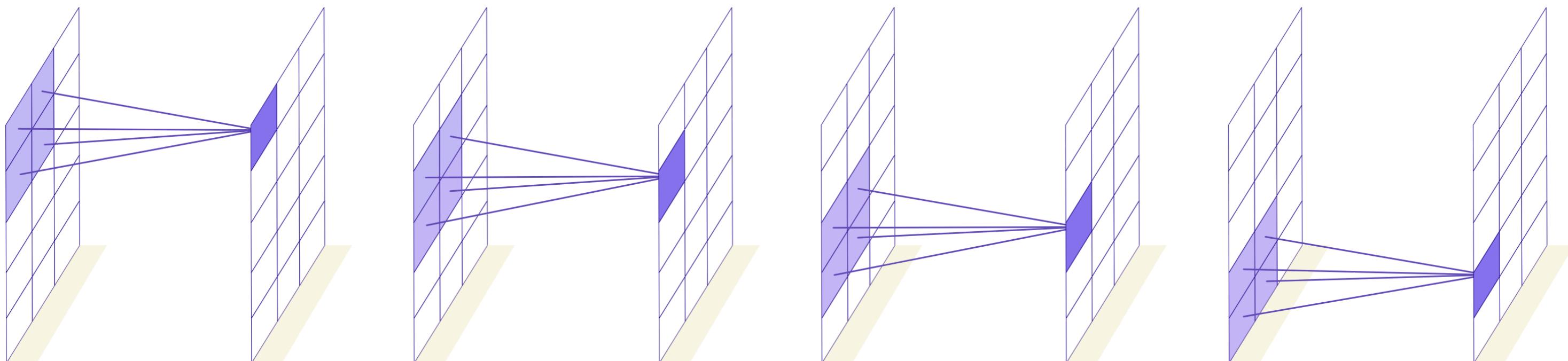
- **Scenario 2:** - We want to analyze customer tickets and assign priority
- Inputs: 10 features selected from 25 total features of a ticket (e.g. ticket topic, keywords ..etc)
- Output: Ticket priority: High / Medium / Low
- Is this a classification or regression?
- Design Input/Output layers of NN



Convolutional Neural Networks (CNN)

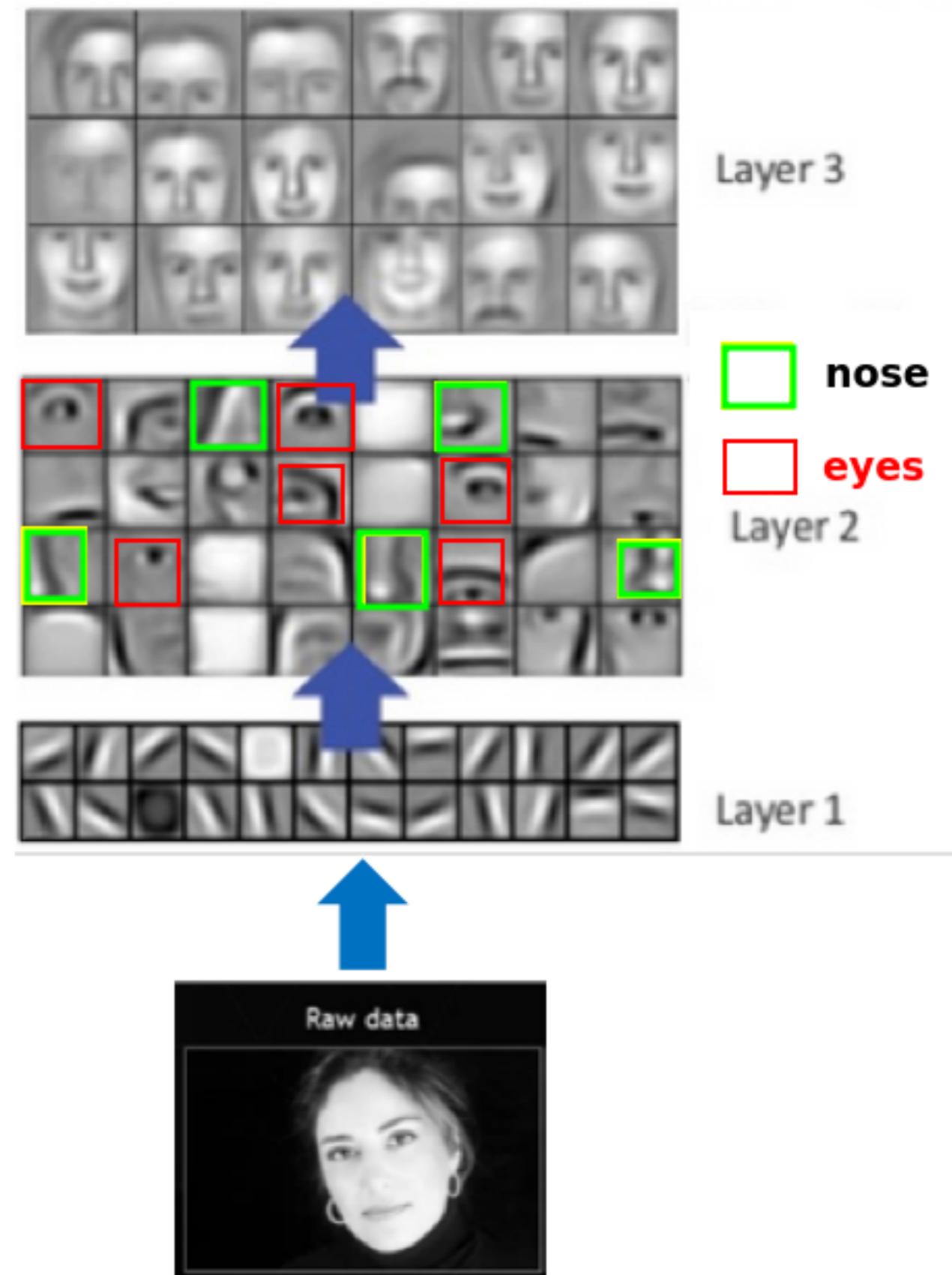
Convolutional Neural Networks (CNN)

- CNNs are designed specially to handle images
- Convolution layer neurons scan a particular area of image (their 'field of vision')
 - And they pick up patterns in images (e.g. eyes, wheels)



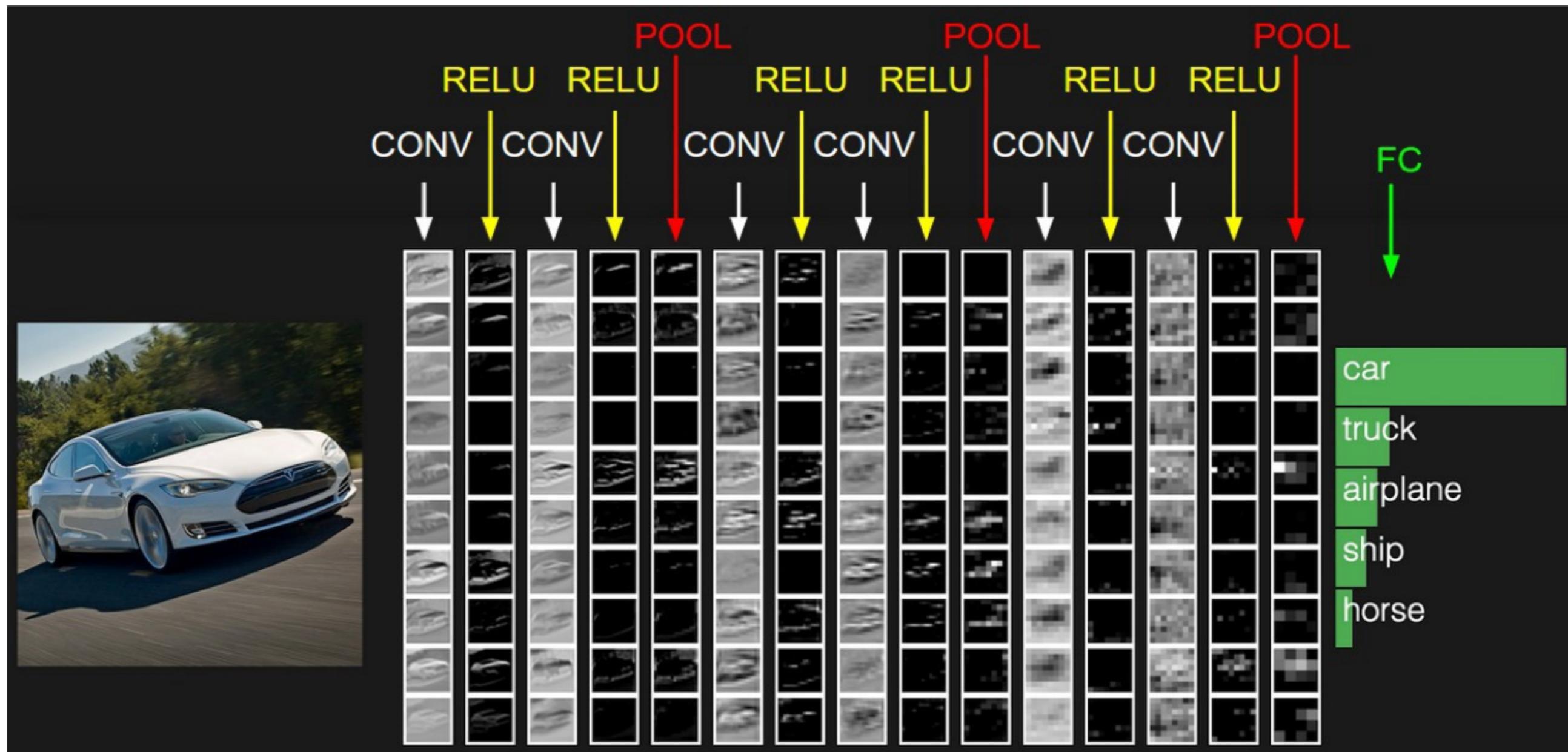
CNNs

- Each layer builds on previous layer's work
- First layer detects simple shapes - horizontal lines, slanted lines ..etc
- Second layer recognizes more complex features: eyes / nose ..etc
- Third layer recognizes faces



CNN Use Cases

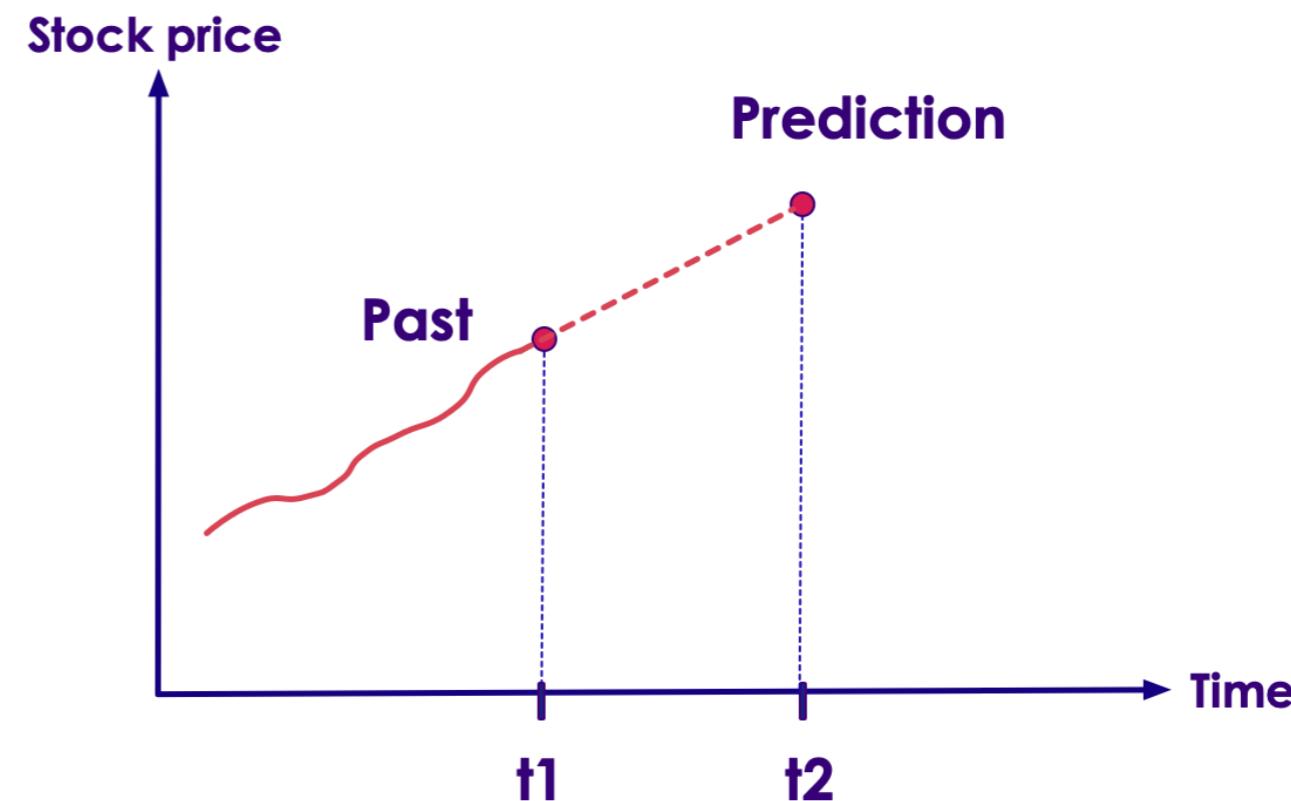
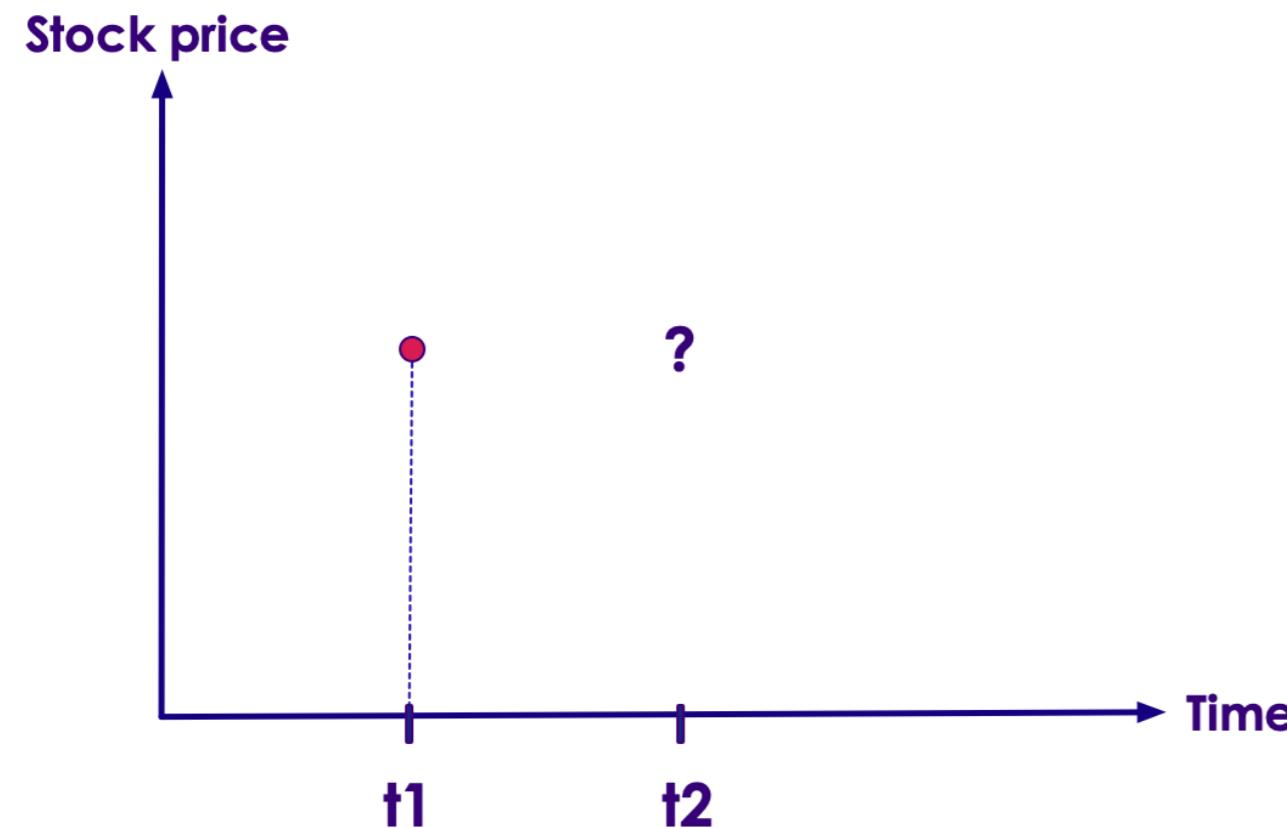
- CNNs are heavily used in **computer vision** applications



Recurrent Neural Networks (RNN)

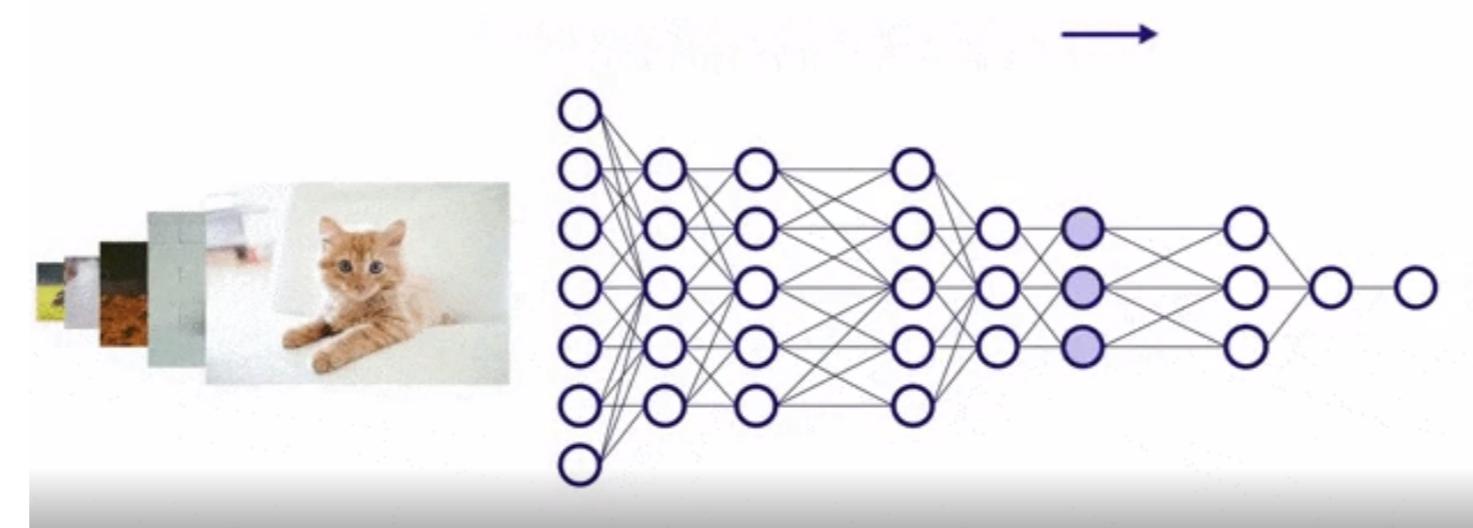
Time Series Data

- To predict time series data, we need to know the past behavior
- For example, what is the stock price in time t_2 ?



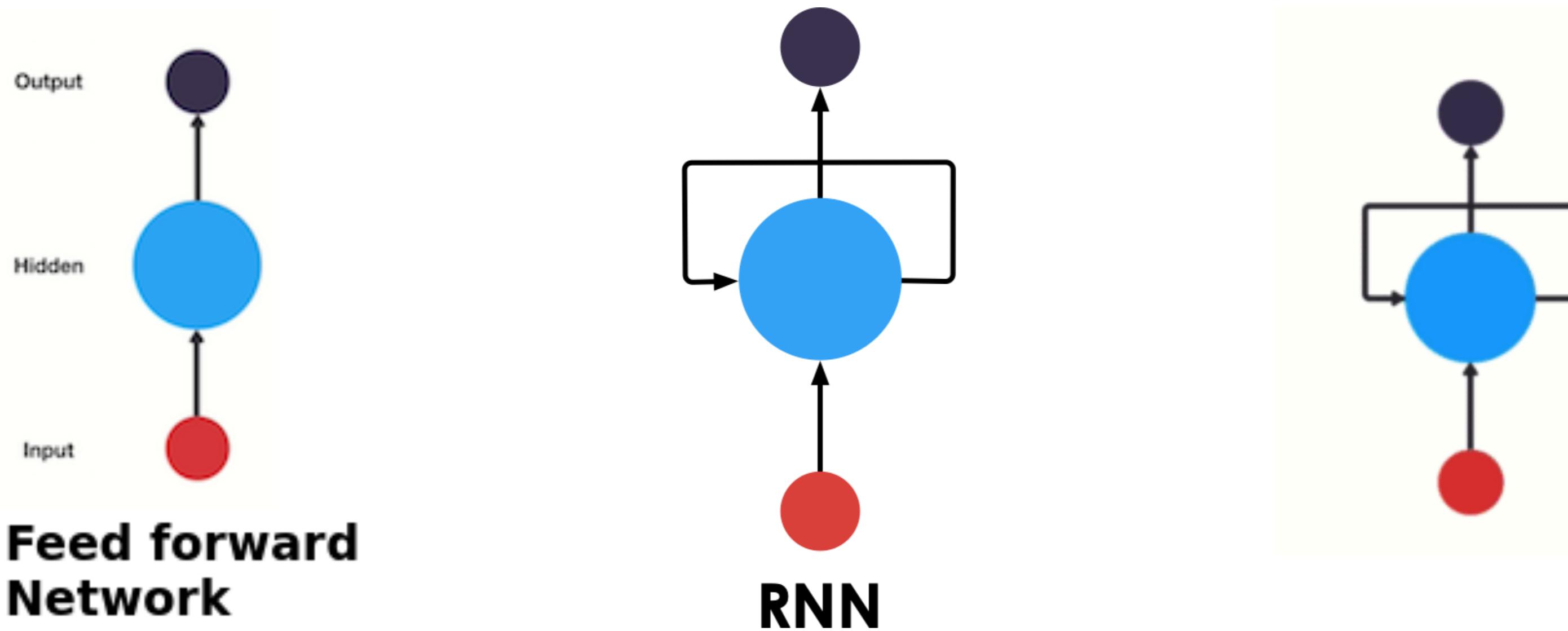
Problems with Feedforward Networks

- Feedforward Neural Networks can model any relationship between input and output.
- However they can't keep/remember **state**
 - The only state retained is weight values from training.
 - They don't remember previous input!
 - For example, in this example, the network doesn't remember the 'previous input' (cat) when predicting the current input
- **Animation** below: [link-youtube](#) | [link-S3](#)



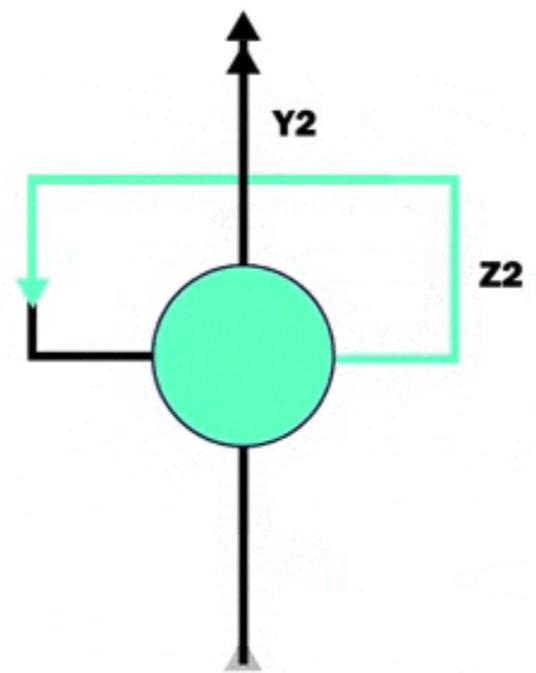
Recurrent Neural Network (RNN)

- In Feedforward Networks, data flows one way, it has **no state or memory**
- RNNs have a 'loop back' mechanism to pass the current state to the next iteration



[Animation link](#)

RNN Animation



- **Animation:** [Link-Youtube](#) | [Link-S3](#)

RNN Implementations

- **RNN (Recurrent Neural Network)** is basic design
 - It implements state
 - But suffers from 'short term memory' problem
- **LSTM (Long Short Term Memory)** improves on this
 - They can remember 'longer memory sequences'
 - But tend to be more complex, and require more time to train
- **GRU (Gated Recurrent Unit)** is an updated implementation
 - It simplifies LSTM, easier to train

RNN Use Cases

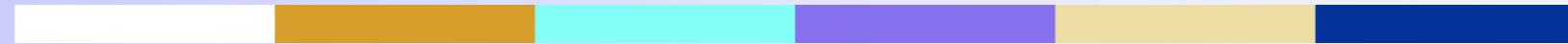
- RNNs are used for any data where sequence matters
- Time series data (like stock quotes)
- NLP (Natural Language Processing)
 - Order of words matter in language
 - Language translation tasks

Review and Q&A

- Let's go over what we have covered so far
- Questions



Neural Network Concepts



Part 1: Essentials

NN Concepts (Overview)

- Training a neural network
- Backpropagation
- Batch size, epoch, iterations
- Activation Functions
- Learning rate, Loss function, Cost
- Vanishing / Exploding gradient problem
- Gradient clipping
- Optimizers

Data Instance / Features

- **Sample** is a single row of data
- Sample = instance / observation / input vector / feature vector
- Each sample has **inputs (vectors) and output**
- Algorithm makes a prediction from inputs, and compares the prediction with actual (expected) output
- In the following example, we have 5 data points / samples (instances 1 - 5)
 - And 3 dimensional input / features: Inputs A,B,C

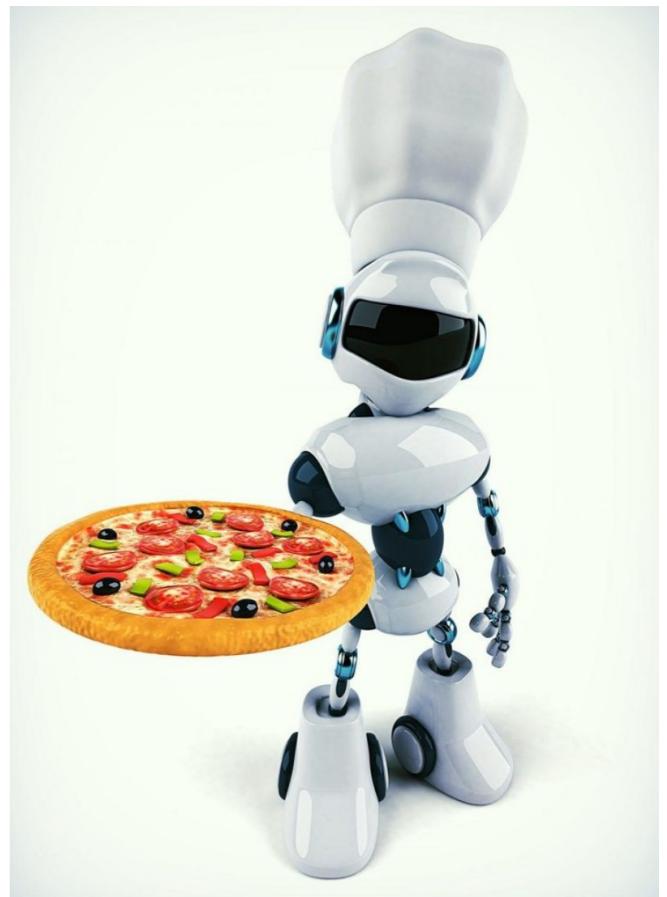
Instance	Input A	Input B	Input C	Output Y
Instance 1	a1	b1	c1	y1
Instance 2	a2	b2	c2	y2
Instance 3	a3	b3	c3	y3
Instance 4	a4	b4	c4	y4
Instance 5	a5	b5	c5	y5

Training the Model

- Rosenblatt's original algorithm for training was simple:
 - Iterate through weights and look at the output error
 - Adjust weights until output error was zero.
 - Will converge only if the outputs are linearly separable.
- The Problem:
 - With hidden layers, we now have a much greater number of weights to try.
 - This kind of "brute force" method will take too long to train.

A Restaurant on an Alien Planet

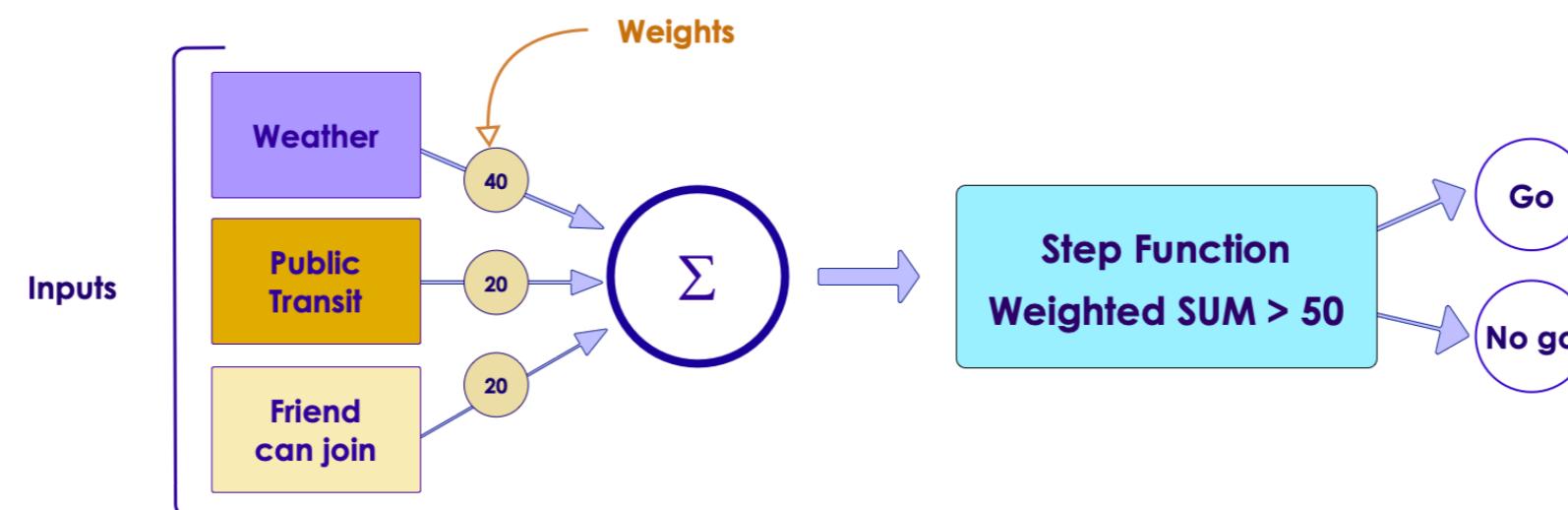
- Imagine you are at restaurant, but you and the waitstaff don't speak the same language (remember they are aliens!)
- You tell them what you want ("*I want a cheese pizza*")
- The kitchen will try to make your pizza and bring it out
- If it is not to your liking, you send it back with feedback ("*it is not cooked well*", "*it needs more sauce*" ..etc)
- You keep sending the pizza back until they get your order right!
- This happens until every one's order is cooked correctly!!



Backpropagation

Difficulty in Training Neural Networks

- In early days (1980s) neural networks' parameters were tuned by hand by experts
- Hand engineering required a lot of knowledge and effort; and usually a very slow process
- Can the networks be trained automatically ?
- Enter **backpropagation**

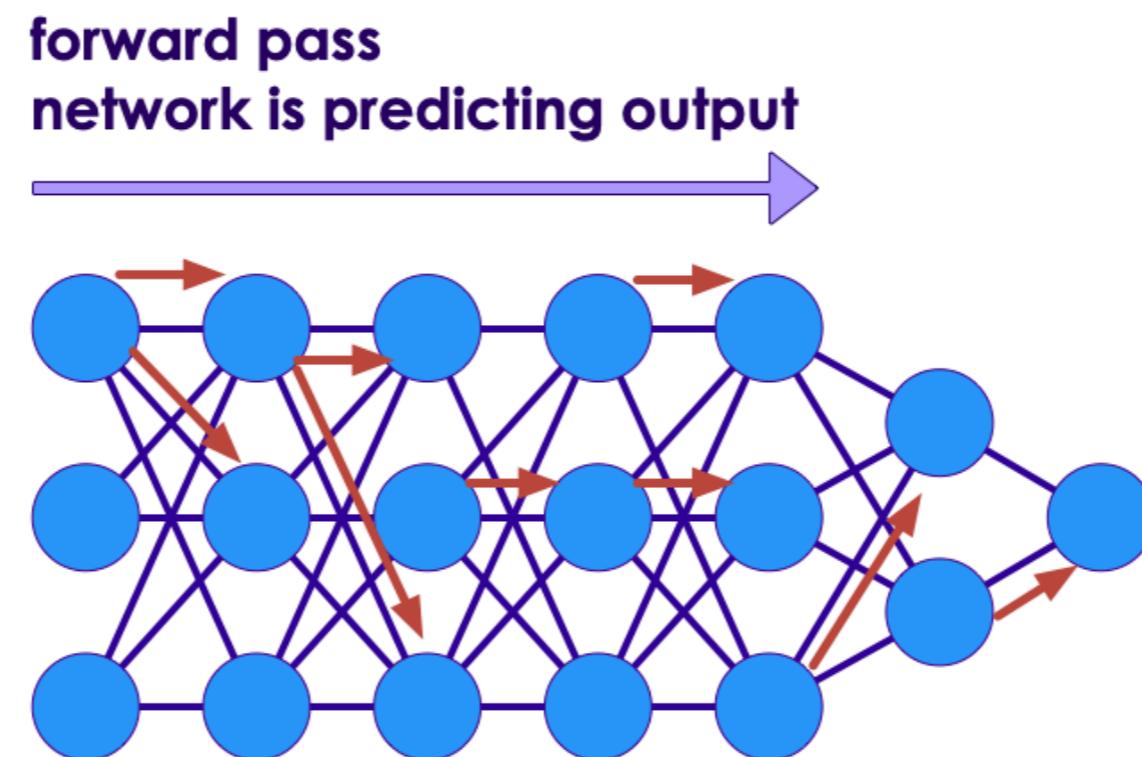


Backpropagation

- Backpropagation algorithm was proposed in 1970s
- But it's usefulness wasn't appreciated until a seminal paper in 1986.
- "**Learning representations by back-propagating errors**"
by David Rumelhart, Geoffrey Hinton, and Ronald Williams
([PDF](#), [Google Scholar](#))
- This paper showed, how backpropagation can be an effective way to train neural networks. And it worked much faster than previous approaches.
- This enabled neural networks to solve difficult problems that were unsolvable before
- This kicked started the current research boom in neural nets

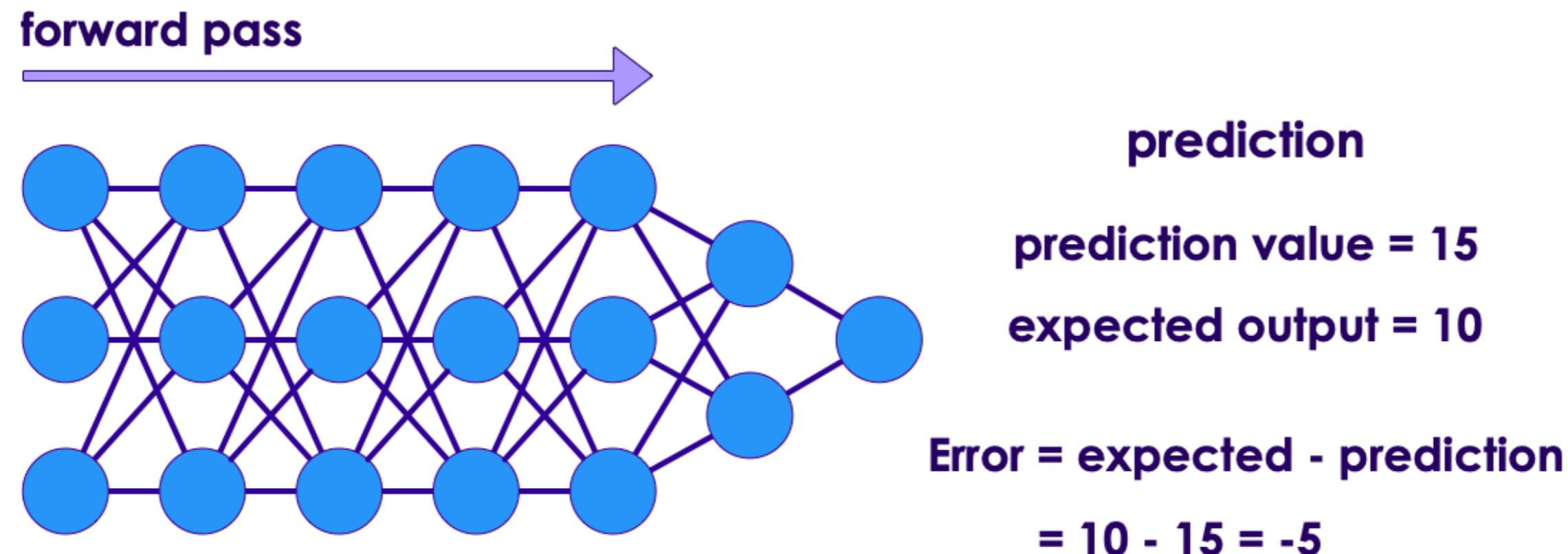
Backpropagation Process: Forward Pass

- During training phase, training data is fed to network
- Neurons in each layer calculate output



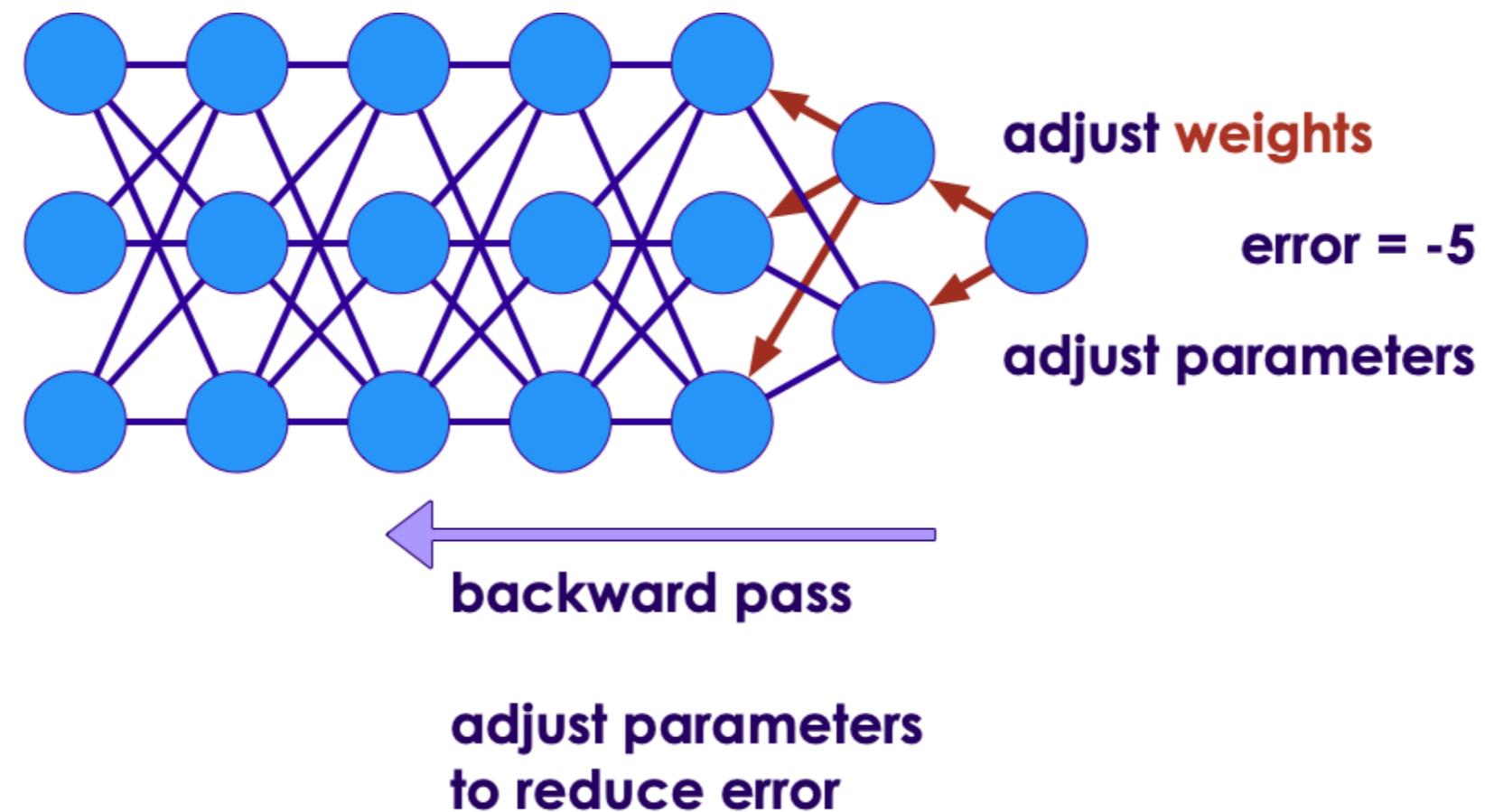
Backpropagation Process: Prediction

- Network predicts an outcome
- This prediction is not usually the same as expected outcome
- Then it measures the error (networks output (prediction) vs. the expected output)



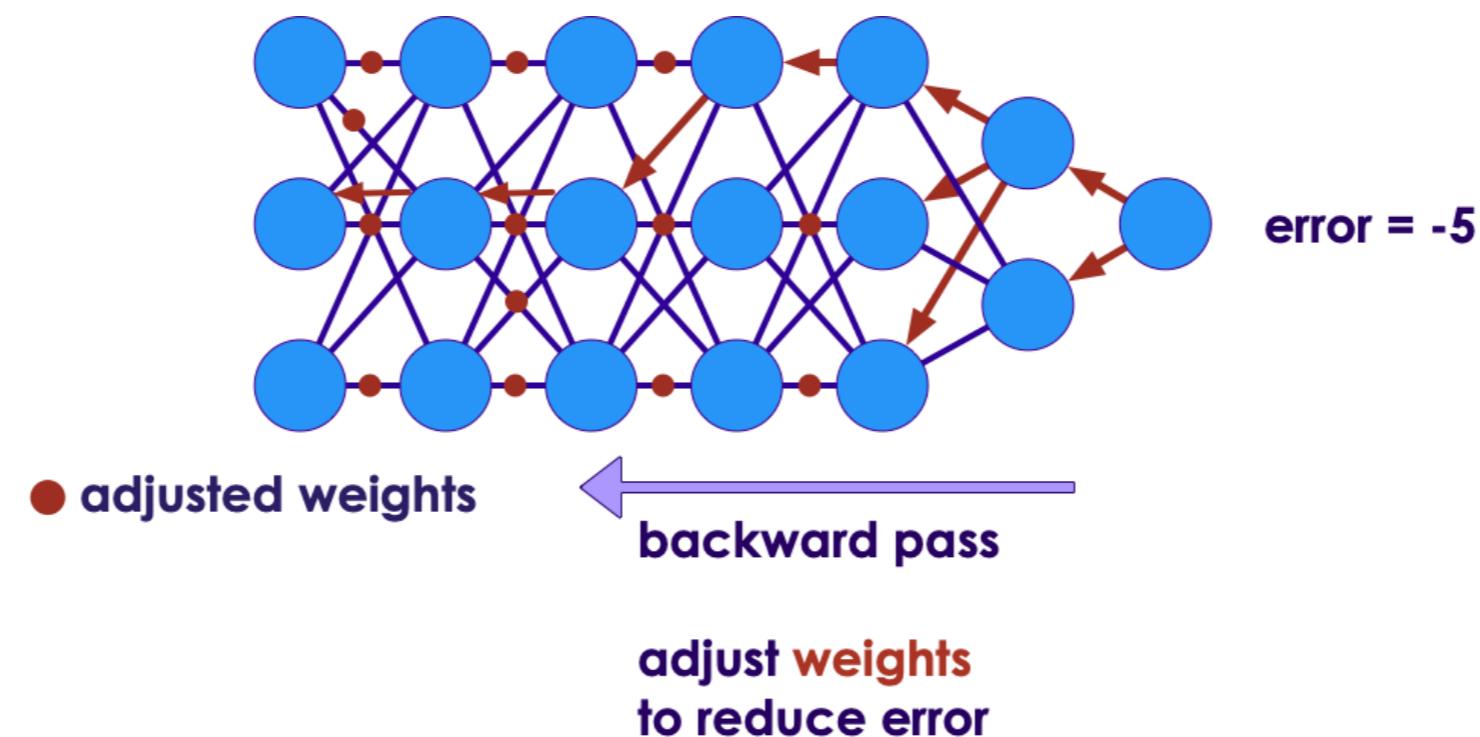
Backpropagation Process: Backward Pass

- It then computes how much each neuron in the last hidden layer contributed to each output neuron's error
- And the network weights are adjusted accordingly to minimize the error



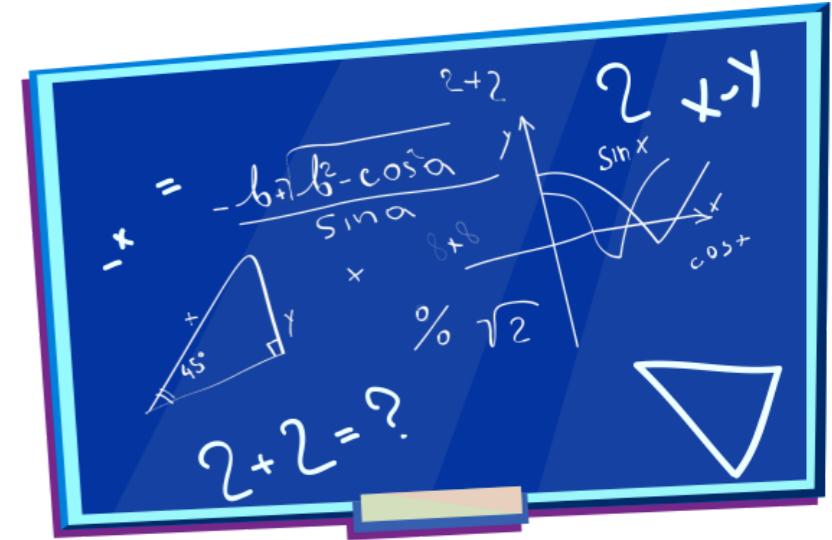
Backpropagation: Backward Pass

- It traverses the network in reverse, computing errors from previous layer
 - until it reaches the input layer
 - this is called 'reverse pass'
 - The reverse pass measures the error gradient across all the connection weights in the network
 - hence called **back propagation**
- During the last step algorithm applies 'Gradient Descent' algorithm on connection weights to tweak them



Backpropagation Math

- Given a cost function C
- weight w in the network
- backpropagation uses partial derivative of $\partial C / \partial w$
- This tells us how quickly cost C changes relative to weight w
- For detailed math please see these links:
 - <http://neuralnetworksanddeeplearning.com/chap2.html>

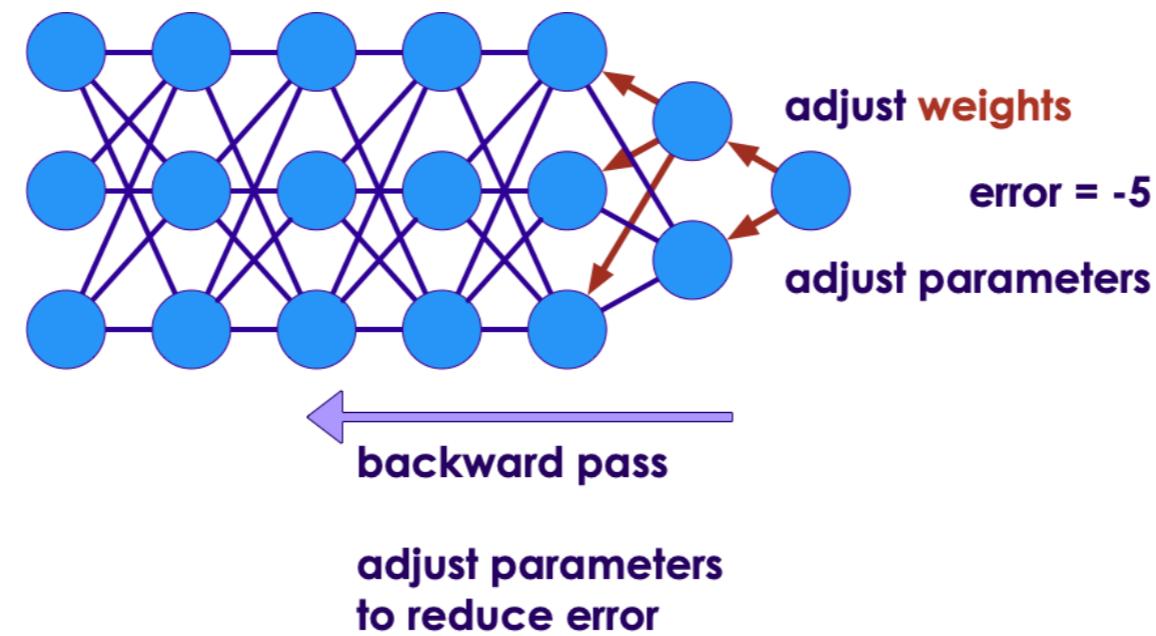


Backpropagation Summary

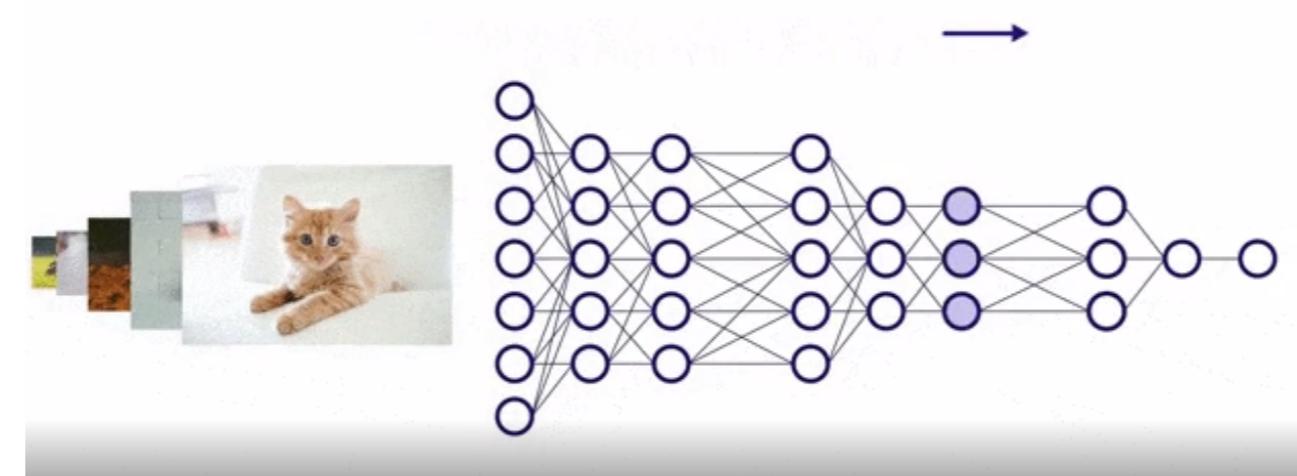
- For each training instance the backpropagation algorithm first makes a prediction (forward pass)
- Measures the error (prediction vs. output)
- Then traverses each layer in reverse to measure the error contribution from each connection (reverse pass)
- And finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).

Backpropagation Demos

- Animation (Regression) : [link-youtube](#), [link-S3](#)

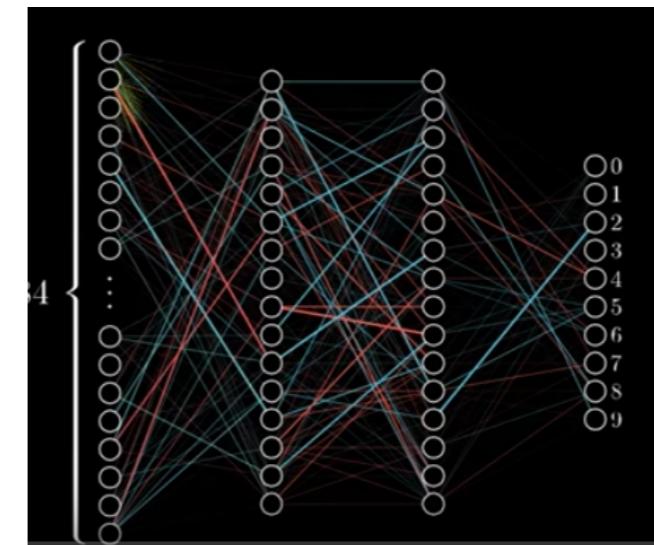
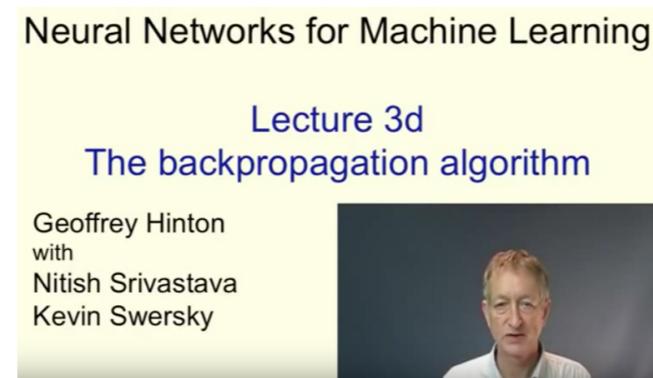
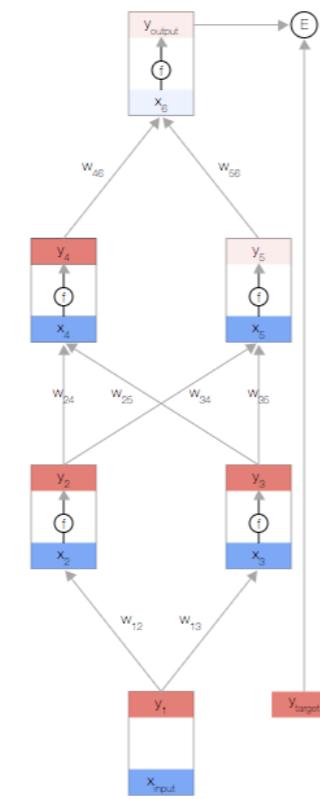


- Animation (Classification) : [link-youtube](#), [link-S3](#)



Backpropagation Demos

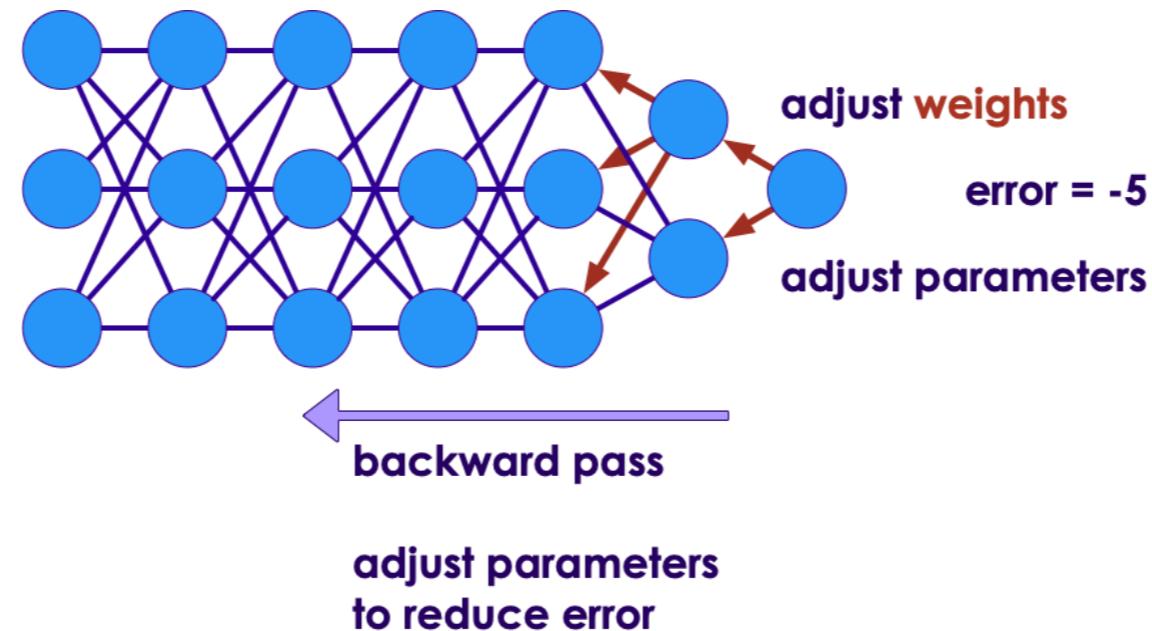
- Demo 1: from Google
- Demo 2 - from Geoffrey Hinton himself ! (~12 mins)
- Demo 3 - Goes through pretty good details (~14 mins)



Controlling Training

Controlling Training

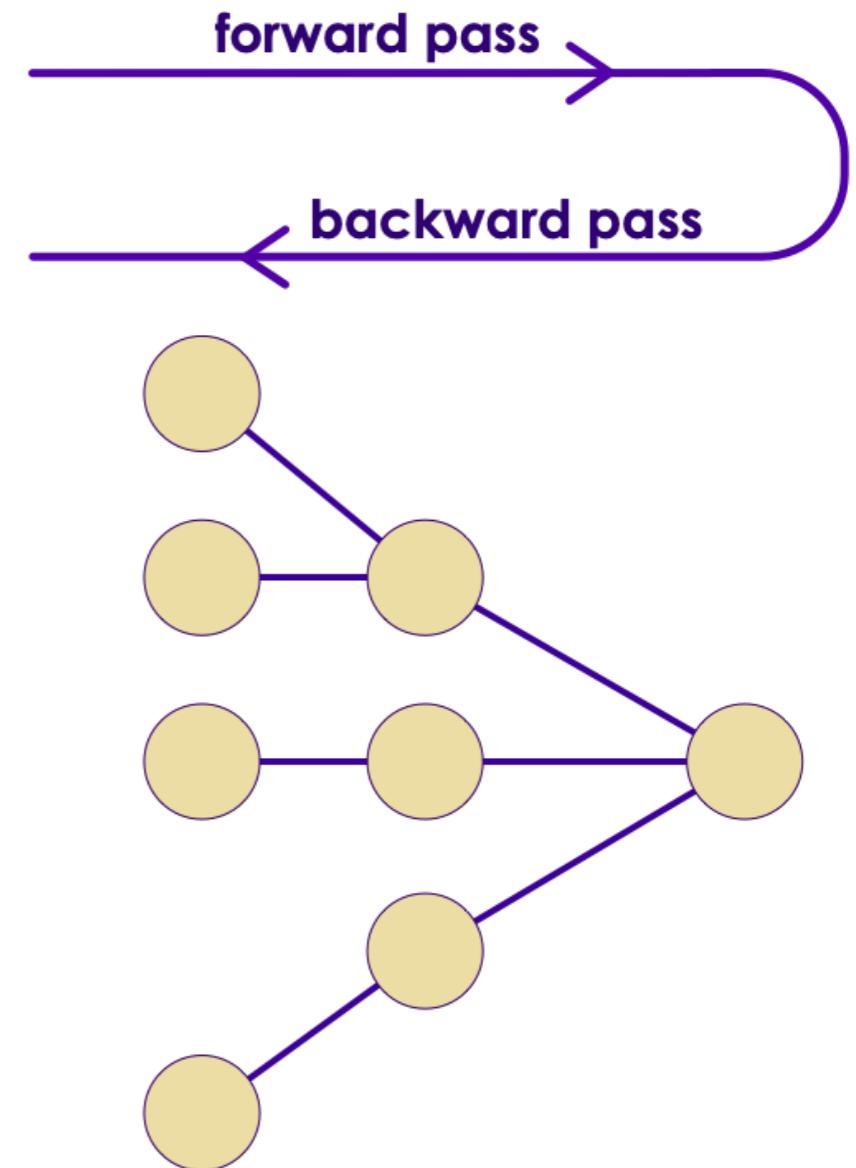
- During training, data goes back and forth through the network
 - Forward passes are for prediction
 - Backward passes are for error correction
- The following parameters control how data flows through the network
 - Epoch
 - Batch size
 - Iteration



Epoch

- One **Epoch** means when an entire dataset passed forward and backward exactly ONCE
 - Restaurant Example: Entire table's meal is sent back once and re-delivered
- Why do we need more than one epoch?
 - Optimizer algorithms try to adjust the weights of neural networks based on training data
 - Just one-pass isn't enough to tweak the weights
 - leads to under-fitting
 - So we need to go back and forth multiple times

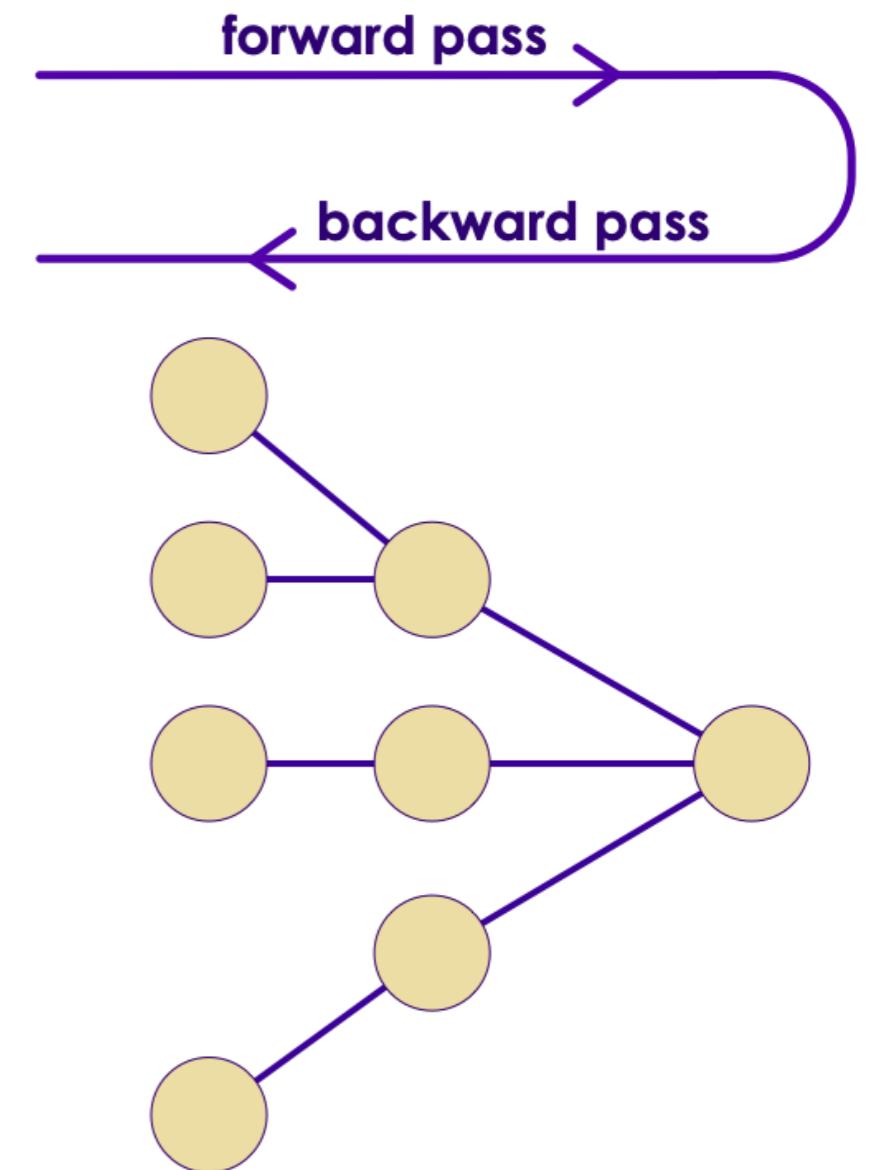
1 epoch = 1 fwd + 1 backward



Epoch

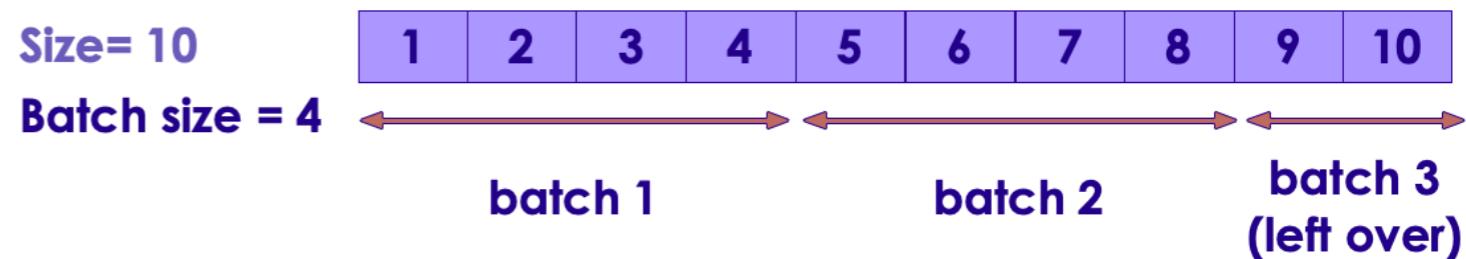
- As we pass the data back and forth multiple times (multiple epochs) the network gets more chance to learn from data and tweak the parameters further
 - model gets more accurate
 - Too many epochs, will lead to overfitting (not good either)
- In practice, the epoch values are typically in hundreds or thousands
- References:
 - Epoch vs. iterations vs. batch

1 epoch= 1 fwd + 1 backward



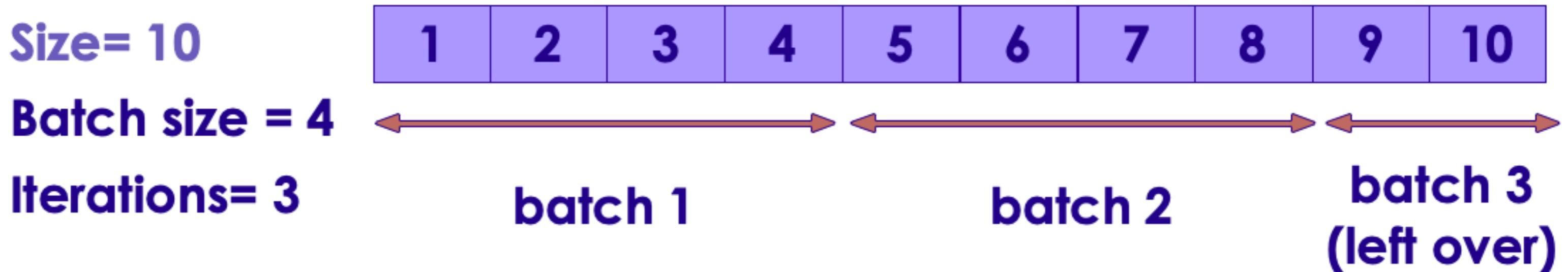
Batch size

- When we are training on large dataset, we can not fit the entire dataset into the network due to memory constraints / processing restraints
- So we send data into batches
- Algorithms (Optimizers) update the weights of neural network after each batch
 - At the end of the batch, predictions are compared with output
 - Error is calculated
 - The algorithm will then calculate error gradient and make the move to minimize the error during the next cycle
- Batch size is usually power of 2 (4, 8, 16, 64 ...)
- Alien Restaurant Example: We send back meals in batches of 3, not the entire table.
- Reference: Batch vs epoch



Batch Size Calculations

- What if data size is not divisible evenly by batch size?
- That is fine, the last batch will have what is left, and will be smaller than previous batches
- For example, if we have 10 data points and batch size is 4
- batch-1 = 4, batch-2 = 4, batch-3 = 2



Iterations

- Iterations is the number of batches needed to complete one epoch.
- Iterations = data size / batch size (round up the result)
- For each epoch, we will need to run iteration amount of times to pass the whole data through the network

```
# think like a nested loop

for e in number_of_epochs {
    iterations = round_up (data_size / batch_size)
    for i in iterations {
        # process batch i
    }
}
```

Epoch / Batch size / Iterations

Batch Size	Algorithm	Description
Size of Training Set	Batch Gradient Descent	All data goes in a single batch
1	Stochastic Gradient Descent	Each batch has one data sample
1 < batch size < size of training set	Mini-Batch Gradient Descent.	Batch size is usually power of 2 (32, 64, 128...)

Size= 10



Batch size = 4



Iterations= 3

batch 1

batch 2

**batch 3
(left over)**

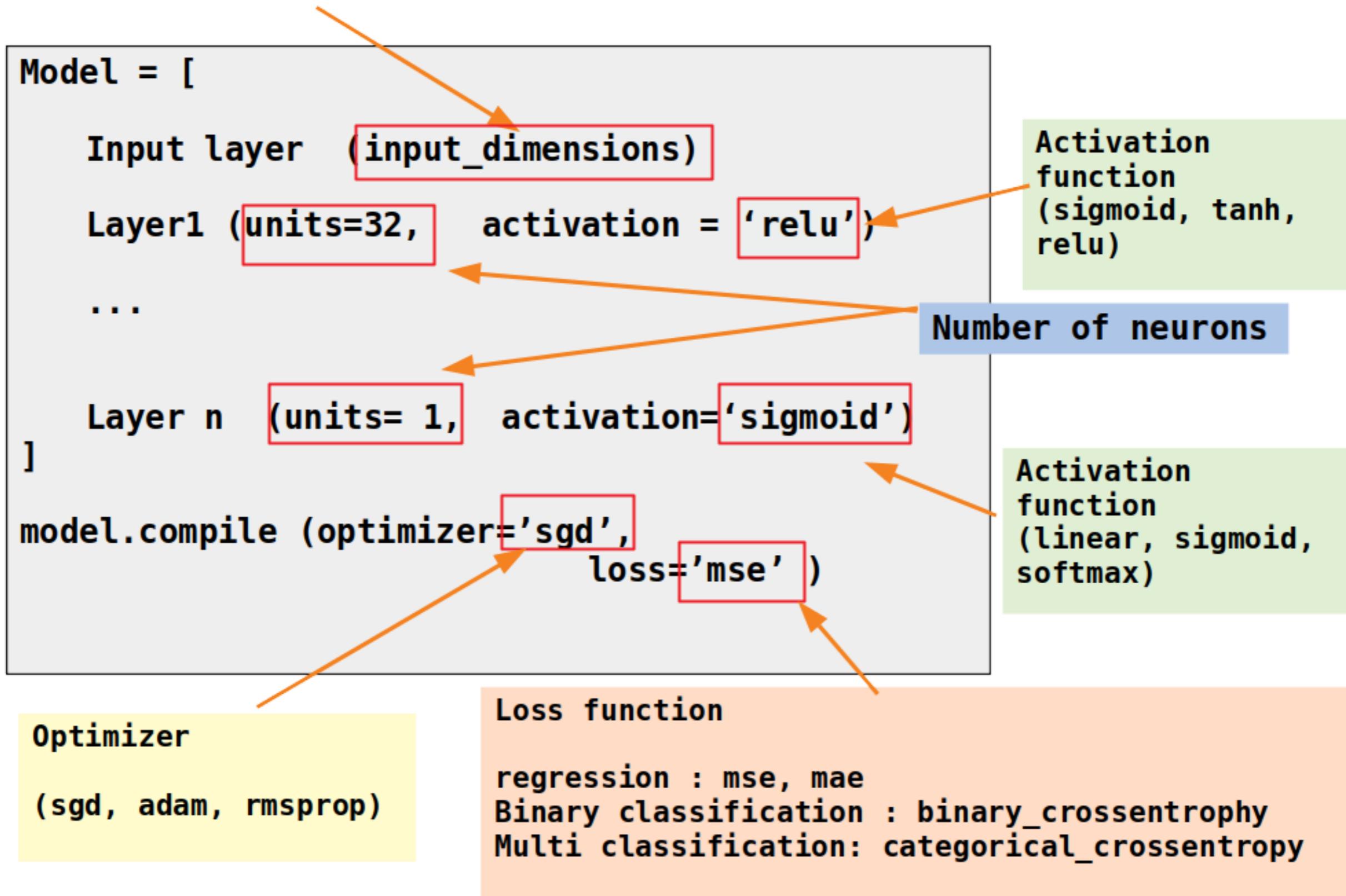
Determining Optimal Values or Batch Size / Epochs

- Typical epochs values are in 100s to thousands
- Batch sizes are powers of 2 (32, 64, 128 ...).
32 is a good value to start with
- One epoch will typically have many iterations
 - Each iteration processing a single batch
- There is no magic formula to calculate the optimal values of batch size and epoch
 - In practice, we try a few runs to figure out optimal values

Training Neural Networks

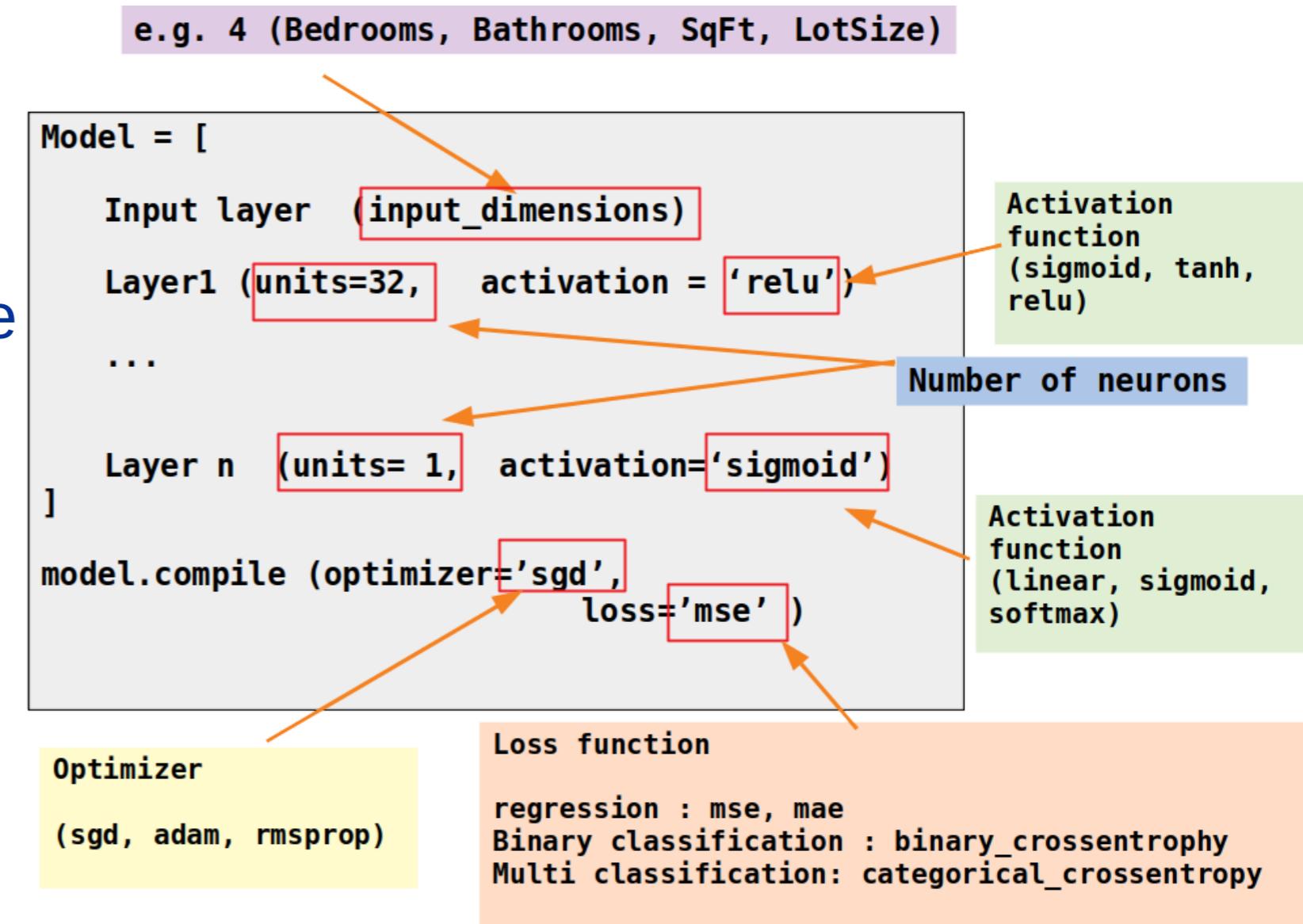
Training Parameters

e.g. 4 (Bedrooms, Bathrooms, SqFt, LotSize)



Training Parameters

- For each layer, we define **number of neurons** and **activation function**
 - more neurons help the network to adopt to complex shape; but more also means increased training time
- Activation functions:** Determine the output each neuron
- Loss functions:** We try to minimize the error/loss
 - For example, when driving, we optimize to minimize time spent



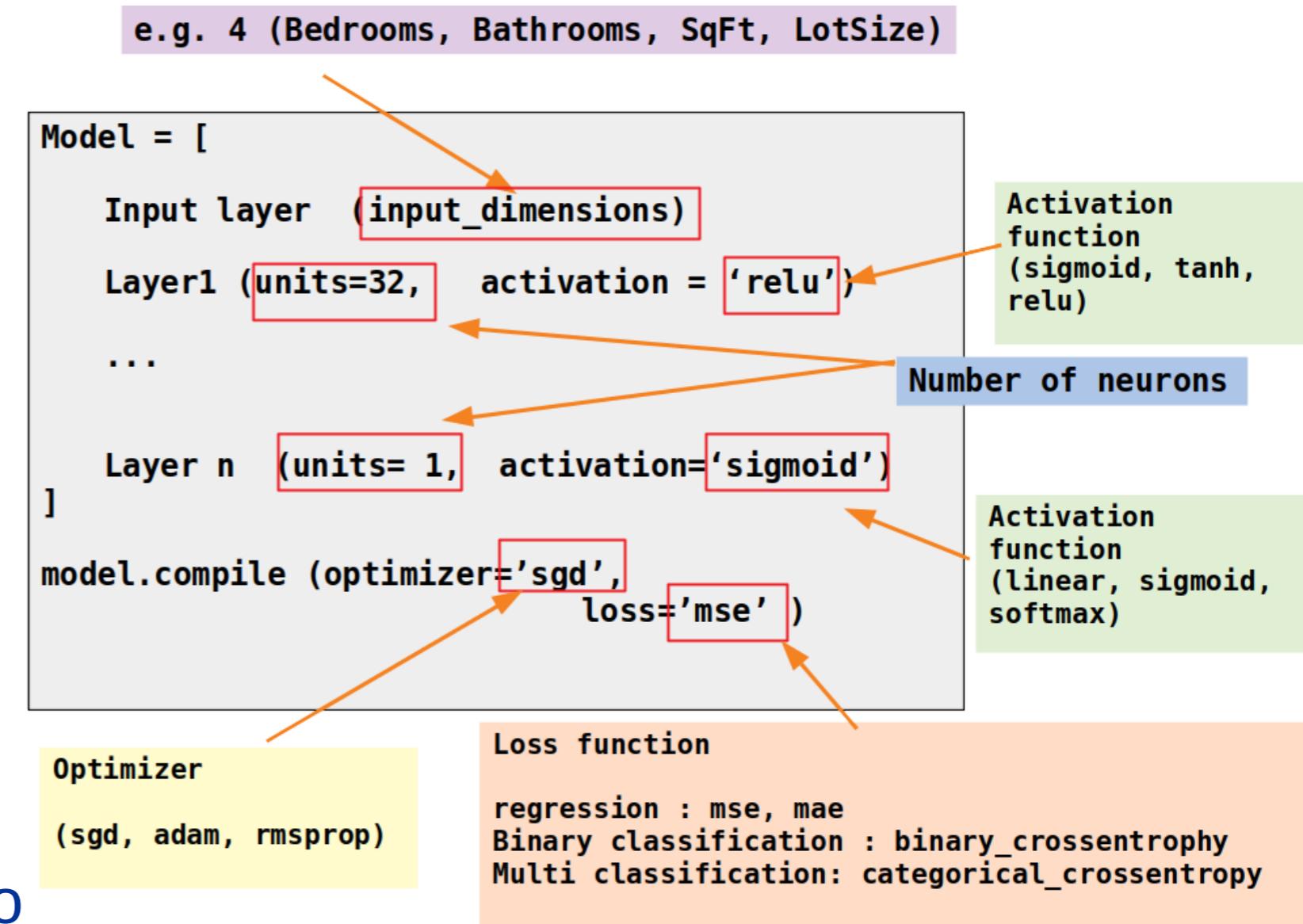
Training Parameters

- **Optimizers:** Helps with adjusting the weights of the network, so our loss can be minimized

- We want to tweak weights efficiently not take random guesses

- **Learning Rate:** This determines how much we update the weights. It is a critical parameter.

- Too small, the training will take too long
 - Too large, the training will bounce around and not converge



Error/Loss Functions for Regressions

Error / Loss Function

- The function that is used to compute the error is known as **Loss Function $J()$**
- Different loss functions will calculate different values for the same prediction errors
- In next few slides, we are going to examine some of the loss functions

Estimating Tips for Meals

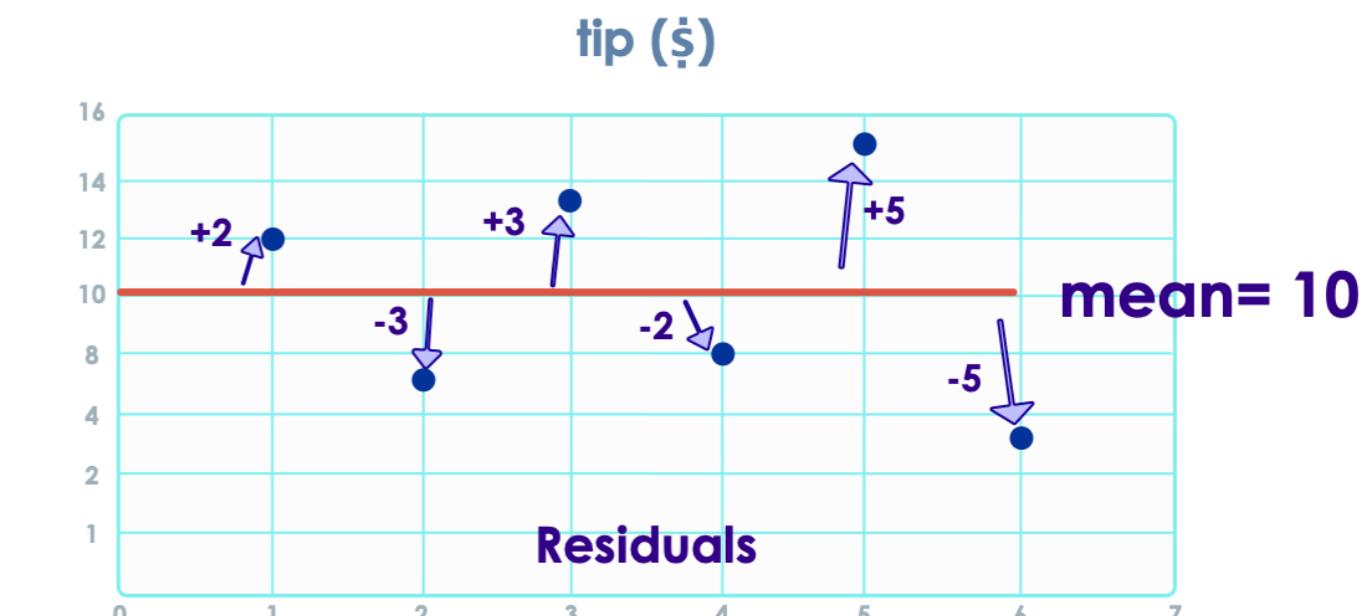
- Let's consider tips at a restaurant

Meal #	Tip (\$)
1	12
2	7
3	13
4	8
5	15
7	5

Understanding Residuals / Errors

- Let's say, our *very naive* model **always predicts tip as \$10 :-)**
- From the table, we can see none of the tip amounts are exactly \$10
 - This difference (delta) is called **Error or Residual**
 - Residual = actual tip - predicted tip**
- Sum of all residuals = **ZERO** (positive and negative errors are canceling each other)

Actual Tip	Predicted Tip	Error or Residual = (Actual - Predicted)
12	10	+2 = (12 - 10)
7	10	-3 = (7 - 10)
13	10	+3 = (13 - 10)
8	10	-2 = (8 - 10)
15	10	+5 = (15 - 10)
5	10	-5 = (5 - 10)
		SUM = 0 (+2 -3 +3 -2 +5 -5)



Sum of Squared Errors (SSE)

- From the previous table, errors can cancel each other out
- Let's square the error:
 - To make them all positive (so negative and positive don't cancel each other out)
 - To amplify 'outliers' (large deviations)
- **Question for the class: Can SSE be zero? :-)**

Actual Tip	Predicted Tip	Error = (Actual - Predicted)	Error Squared
12	10	+2 = (12 - 10)	4
7	10	-3 = (7 - 10)	9
13	10	+3 = (13 - 10)	9
8	10	-2 = (8 - 10)	4
15	10	+5 = (15 - 10)	25
5	10	-5 = (5 - 10)	25
	Total ==>	0	76

Sum of Squared Errors (SSE)

- Also known as
 - **Residual Sum of Squares (RSS)**
 - **Sum of Squared Residuals (SSR)**
- In this formula
 - y_i : actual value
 - \hat{y}_i : predicted value
- Properties
 - A good all purpose error metric that is widely used
 - SSE also 'amplifies' the outliers (because of squaring)
- For example, if SSE for model-A = 75 and SSE for model-B = 50
 - Model-B might be better fit

$$SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Mean Squared Error (MSE) (L2)

Actual Tip	Predicted Tip	Error = (Actual - Predicted)	Error Squared
12	10	+2 = (12 - 10)	4
7	10	-3 = (7 - 10)	9
13	10	+3 = (13 - 10)	9
8	10	-2 = (8 - 10)	4
15	10	+5 = (15 - 10)	25
5	10	-5 = (5 - 10)	25
Total ==>		0	76

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

■ $MSE = (4 + 9 + 9 + 4 + 25 + 25)/6 = 76 / 6 = 12.6$

■ Properties

- Can be sensitive to outliers; predictions that deviate a lot from actual values are penalized heavily
- Easy to calculate gradients (fast)

Mean Absolute Error (MAE)

Actual Tip	Predicted Tip	Error = (Actual - Predicted)	Absolute Error	Error Squared
12	10	+2 = (12 - 10)	2	4
7	10	-3 = (7 - 10)	3	9
13	10	+3 = (13 - 10)	3	9
8	10	-2 = (8 - 10)	2	4
15	10	+5 = (15 - 10)	5	25
5	10	-5 = (5 - 10)	5	25
Total ==>		0	20	76

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

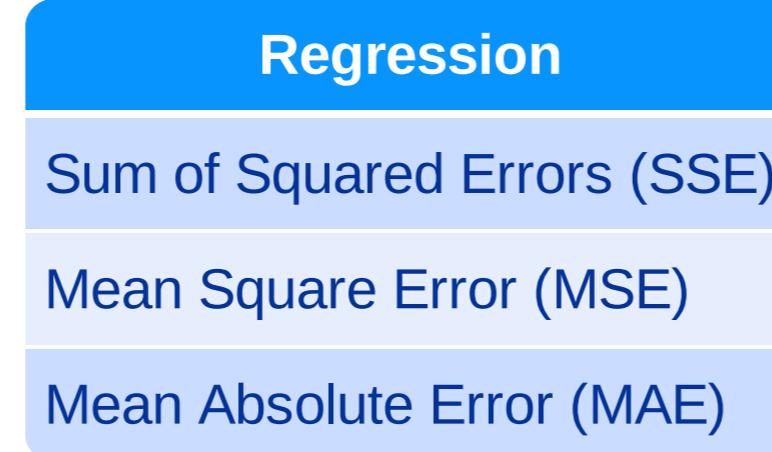
- MAE = $(2 + 3 + 3 + 2 + 5 + 5) = 20 / 6 = 3.33$

- Properties:

- More robust and is generally not affected by outliers
- Use if 'outliers' are considered 'corrupt data' (or not critical part of data)

Regression Error Functions - Summary

- Error functions tell us 'how far off' our prediction from actual value is.
- We have seen 3 popular error functions for regression
- Which one to use?
 - No 'hard' rules!, follow some practical guide lines
 - Try them all and see which one gives better results! :-)
(most ML libraries allow us to configure the error function very easily)



Error/Loss Functions for Classifications

Loss Functions for Classification

- Binary Class Entropy
- Categorical Crossentropy / Sparse Categorical Crossentropy
- Negative Log Likelihood
- Margin Classifier
- Soft Margin Classifier

Binary Classifications: Binary Class Entropy

- Cross Entropy is used in binary classification scenarios (0 / 1)
- Measures the divergence of probability distributions between actual and predicted values

Income (input 1)	Credit Score (input 2)	Current Debt (input 3)	Loan Approved (output)
40,000	620	0	0
80,000	750	100,000	1
100,000	800	50,000	1

$$E = -\frac{1}{n} \sum_{i=1}^n [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Multi Class Classifications: Sparse Categorical Crossentropy

- Here we predict one of many labels (1,2,3)
- Our labels are integers
- Choice: **sparse_categorical_crossentropy**

a	b	c	d	label
6.4	2.8	5.6	2.2	1
5.0	2.3	3.3	1.0	2
4.9	3.1	1.5	0.1	3

Multi Class Classifications: Categorical Crossentropy

- Here we predict one of many labels (1,2,3)
- Our labels are **one-hot-encoded**
- Choice: **categorical_crossentropy**

a	b	c	d	label
6.4	2.8	5.6	2.2	[1,0,0]
5.0	2.3	3.3	1.0	[0,1,0]
4.9	3.1	1.5	0.1	[0,0,1]

Summary of Errors / Loss Functions

Regression	Classification	Embedding
Sum of Squared Errors (SSE)	Binary Class Entropy	Cosine Error
Mean Square Error (MSE)	Categorical Crossentropy	L1 Hinge Error
Mean Absolute Error (MAE)	Margin Classifier	
	Soft Margin Classifier	
	Negative Log Likelihood	

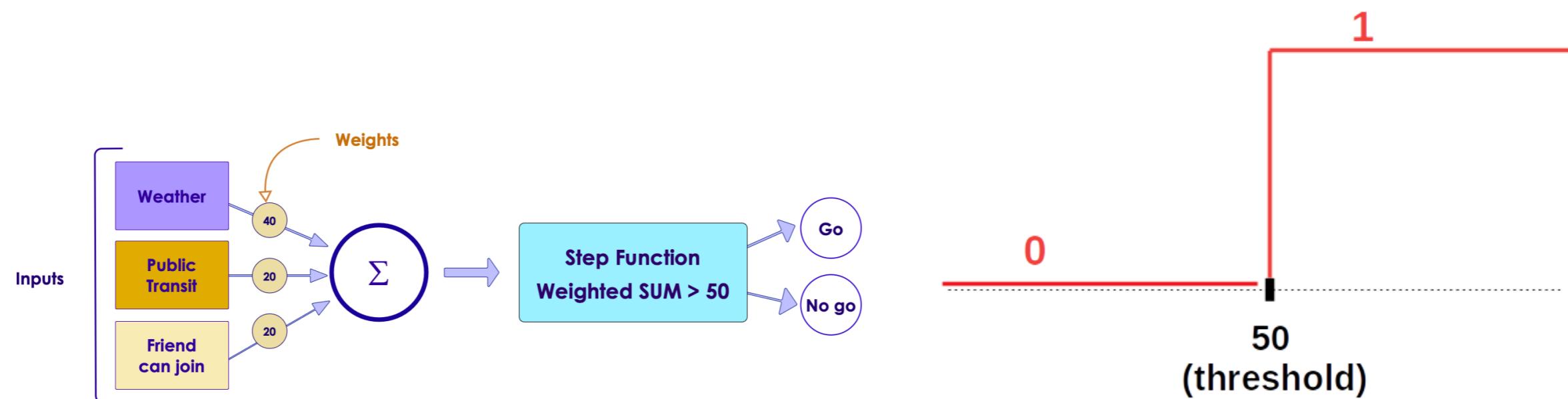
Loss Functions Based on Task

Problem Type	Prediction	Loss Function
Regression	a number	mse, sse, mae
Classification	binary (0/1)	binary_crossentropy

Activation Functions

Activation Functions

- Let's consider our 'simple perceptron' example
- The perceptron sums up all inputs and weights
- The output of the perceptron would be
output = weather * 40 + public * 20 + friend * 20
- Then we defined an **activation function**
 - If the output was greater than threshold (50) then **answer = 1**
 - Else, **answer = 0**
- This is a simple **step function** activation
 - It maps the number to a boolean function (0/1)



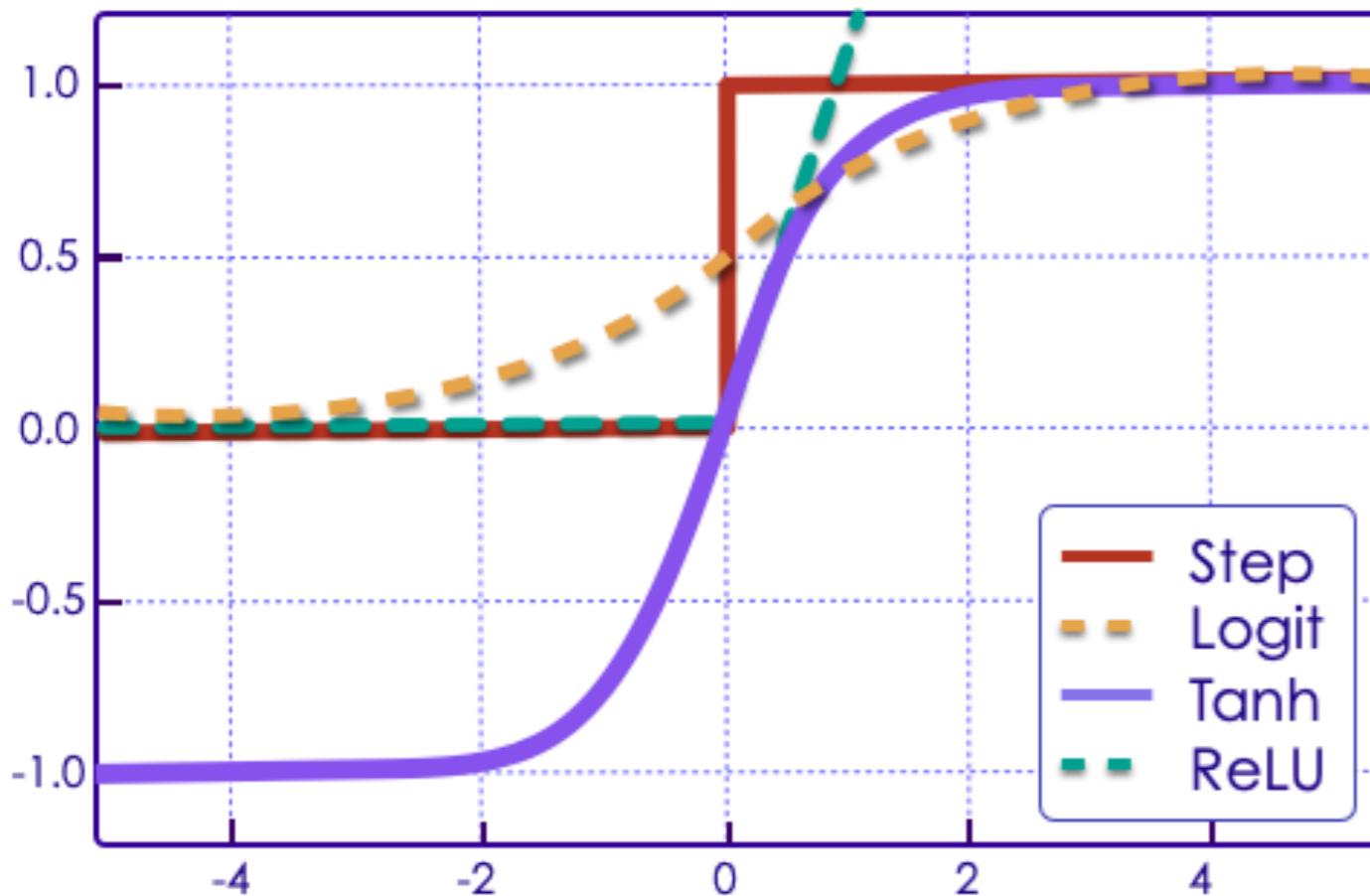
Why Activation Functions?

- Without the activation function, the output of the perceptron would be **linear**
- Linear functions, tend to be simple; they can not solve complex problems
- Neural Nets are seen as '**universal function approximators**'
 - they can compute and learn any function
- So we need '**non linearity**' so our NNs can compute complex functions
 - Hence we need '**activation functions**'

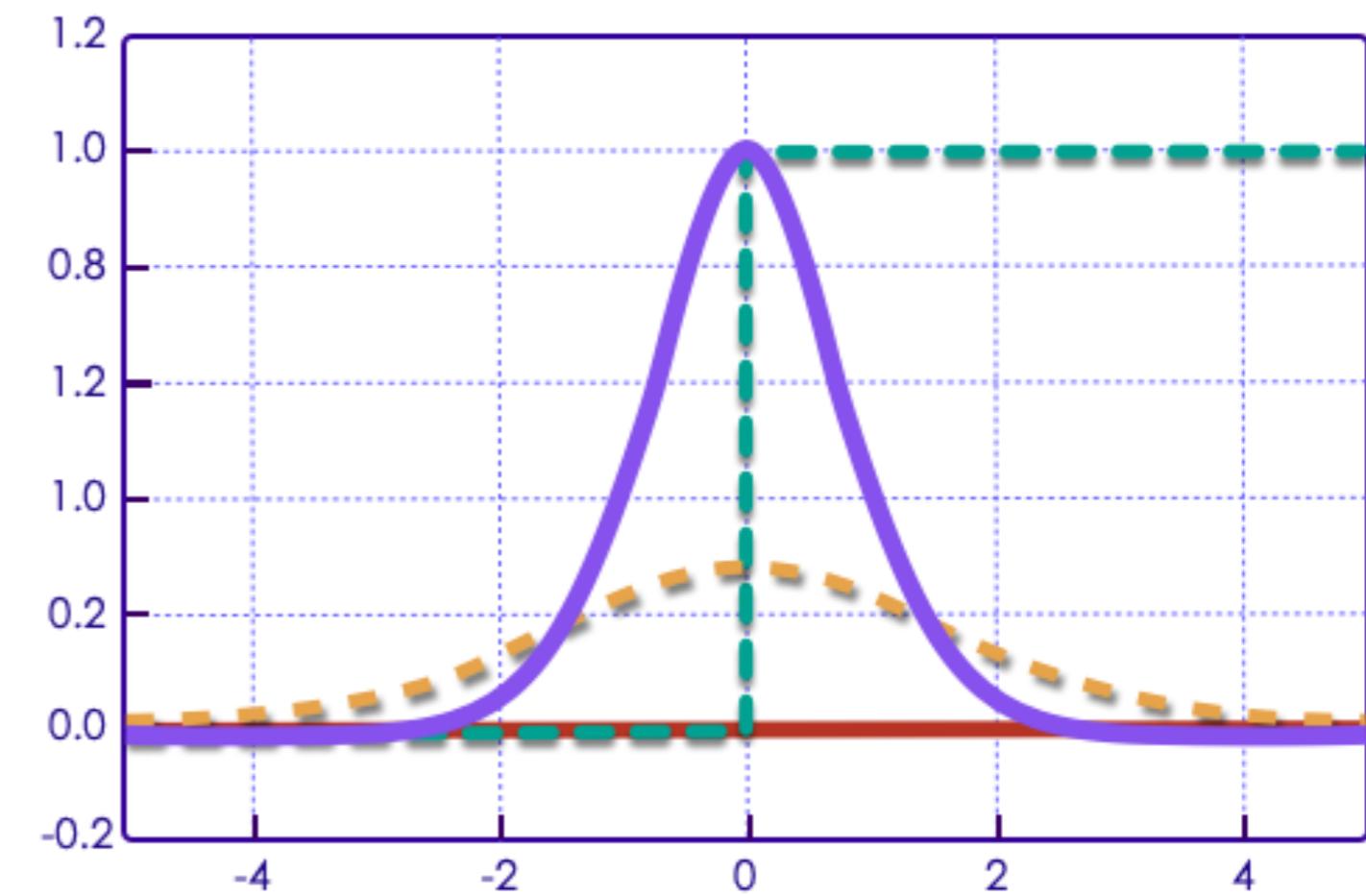
Activation Functions

- None (just use raw output of neuron)
- Linear
- Sigmoid
- Tanh
- ReLU (Leaky ReLU ... etc)

Activation functions

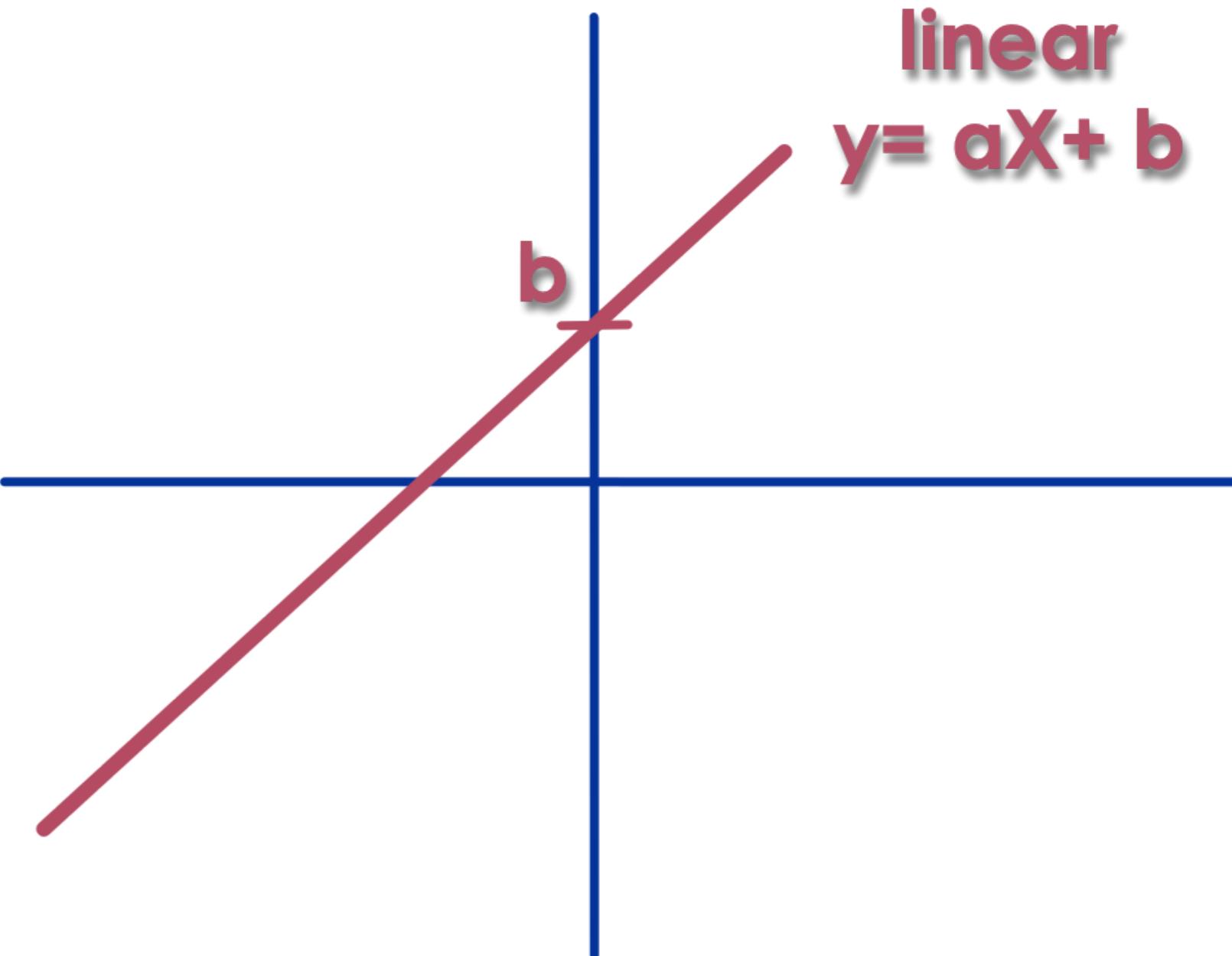


Derivatives



Activation Function - Linear

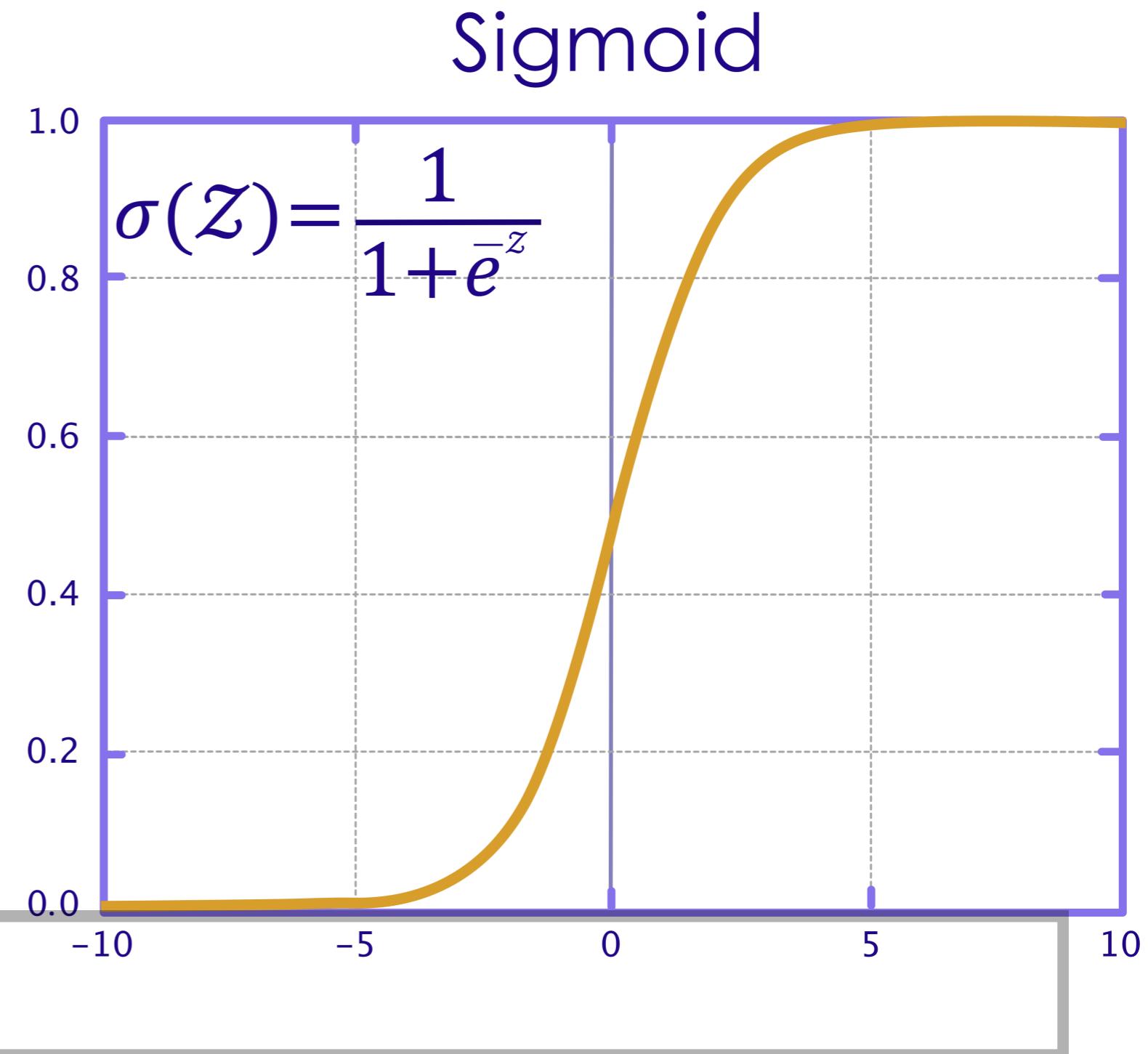
- $y = a \cdot x + b$
- Differentiable
 - So we *can* use gradient descent
- Commonly used for **Regression**
- E.g. Linear Regression:
 - Single Layer (Linear)
 - Linear Activation Function



Activation Function - Sigmoid

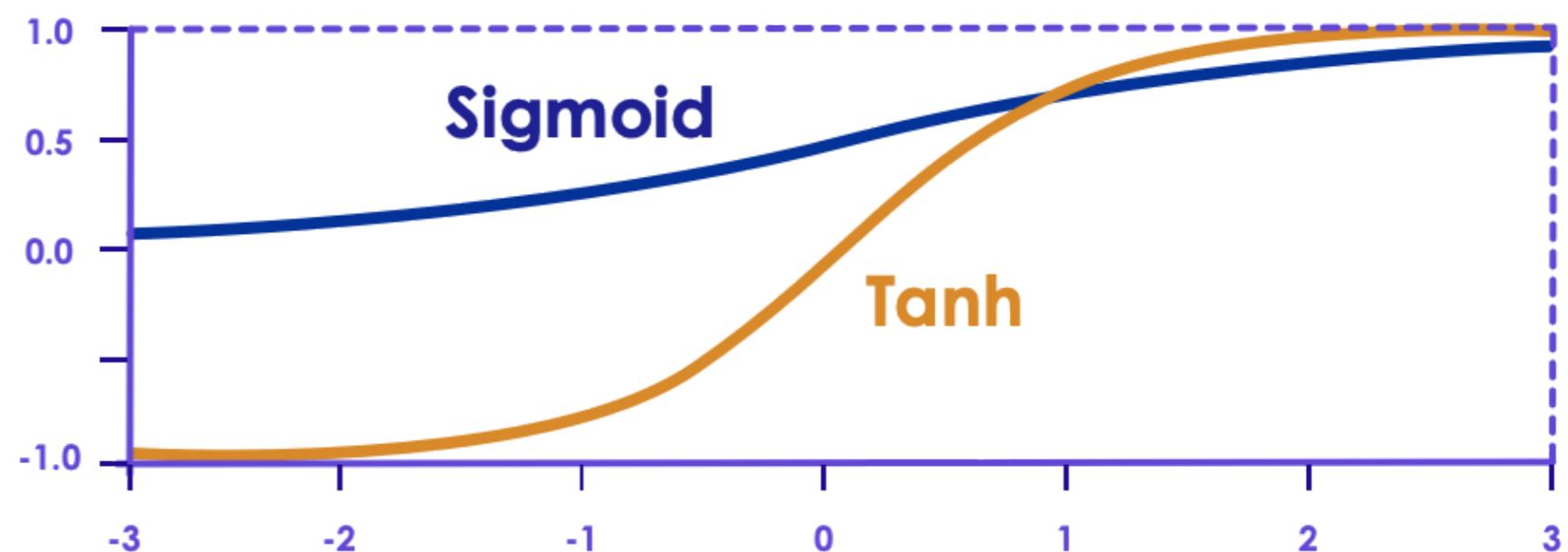
- $\sigma(z) = 1 / (1 + \exp(-z))$
- Sigmoid(Logistic) function has well defined non-zero derivative throughout
 - so we can use Gradient Descent
- Sigmoid function ranges from **0** to **+1**
- Note : Historically Sigmoid function has been very popular.
Recently, ReLU functions work better and are more popular now.

```
sigmoid (0) = 0.5  
sigmoid (10) = 0.99995  
sigmoid (-10) = 0.00005
```



Tanh Activation

- $\tanh(z) = 2\sigma(2z) - 1$
- Just like Sigmoid, Tanh is S-shaped, continuous, and differentiable
- Tanh is symmetric around zero and ranges from **-1 to +1** (sigmoid ranges from **0 to +1**)

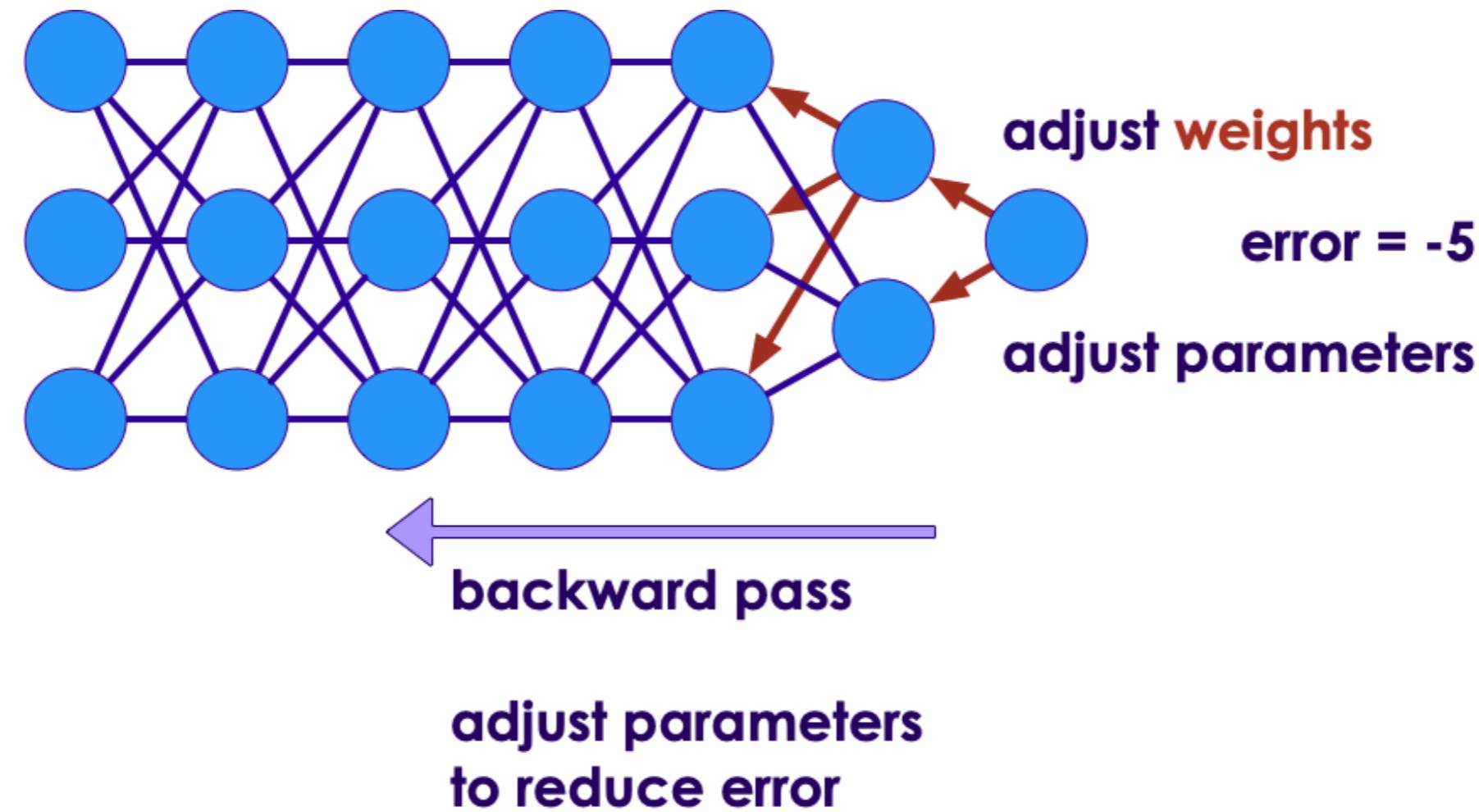


```
Tanh(0) = 0
Tanh(+1) = 0.761594155956
Tanh(+10) = 0.999999995878
Tanh(-10) = -0.999999995878
```

Vanishing/Exploding Gradients Problems

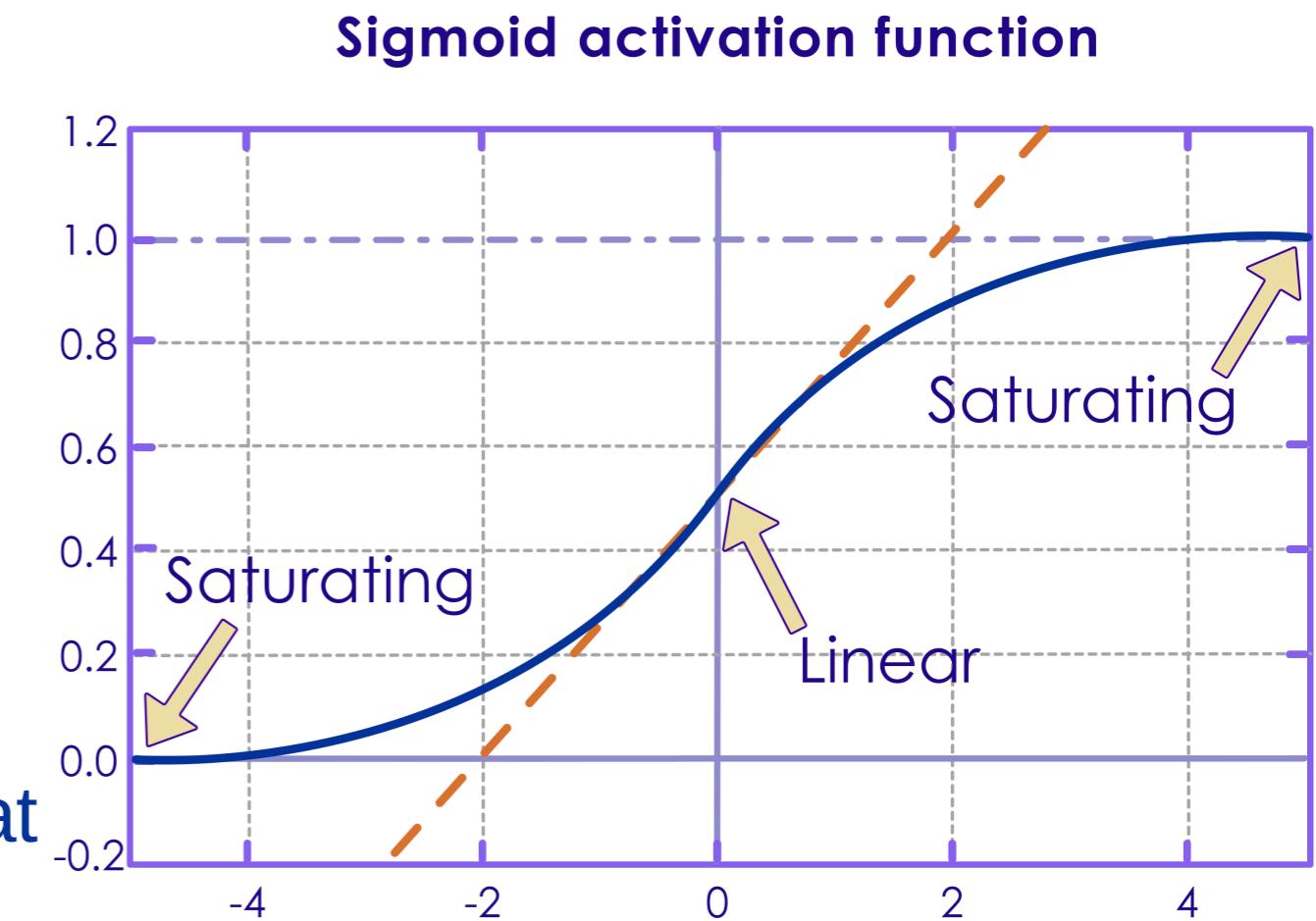
Vanishing Gradients Problem

- As we know backpropagation works by going from output layer to input layer (in reverse)
- It propagates the error gradient backwards
- And the weights are adjusted by multiplying by derivatives
- **Animation** : [link-youtube](#), [link-S3](#)



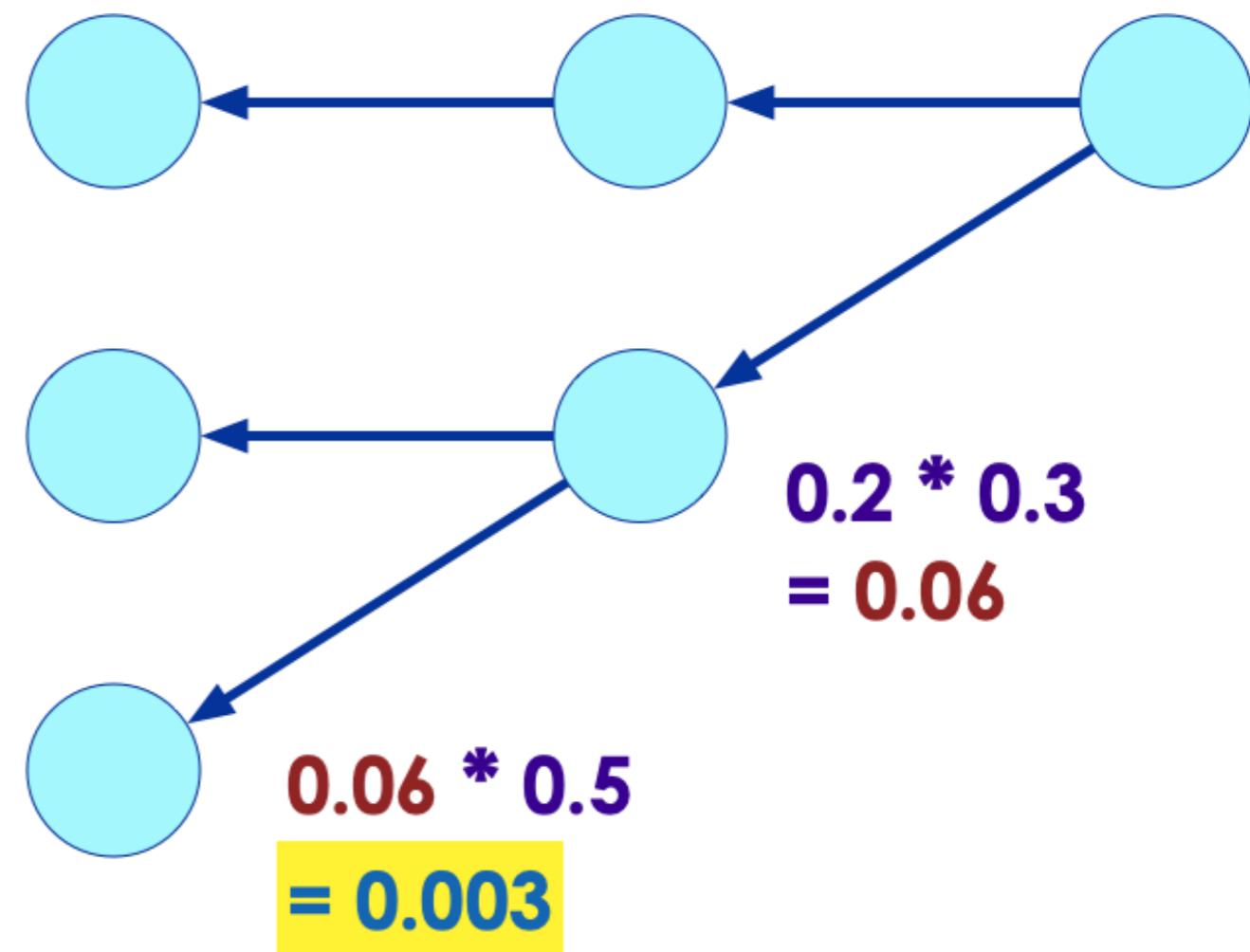
Vanishing Gradient Problem

- Sigmoid and Tanh both suffer from the **Vanishing Gradient** problem.
- The derivative of a Sigmoid is **less than 0.25**
- As the gradients gets multiplied, they will get smaller and smaller
 - e.g. : $0.02 * 0.01 = 0.0002$
- As we propagate that through many layers, that gradient becomes much less.
- And their slopes (derivatives) get closer to zero for large input values
 - this is called **saturating**



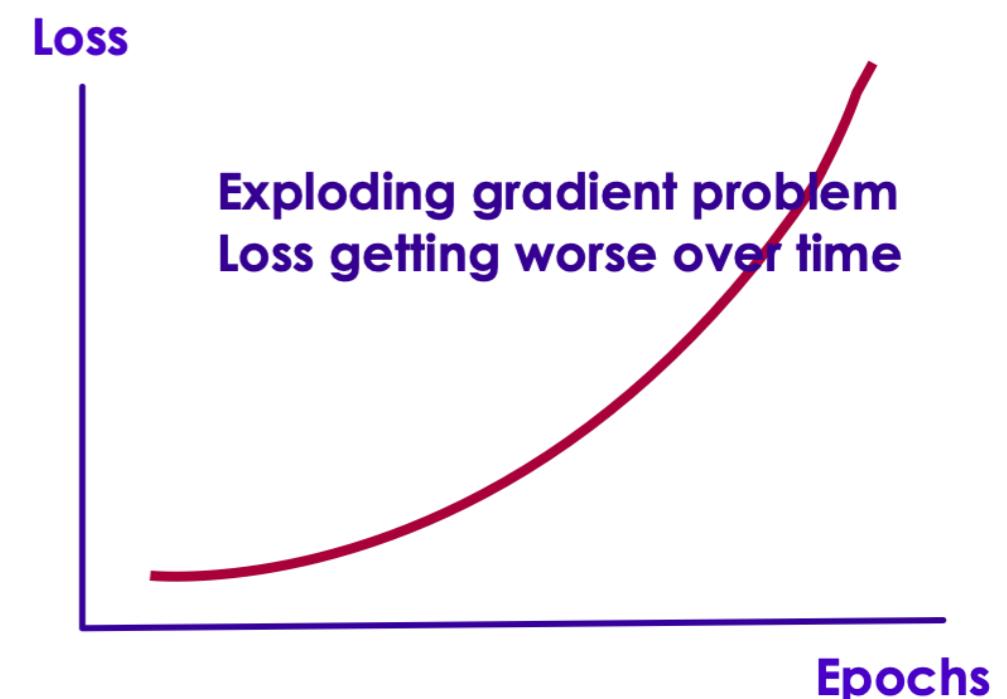
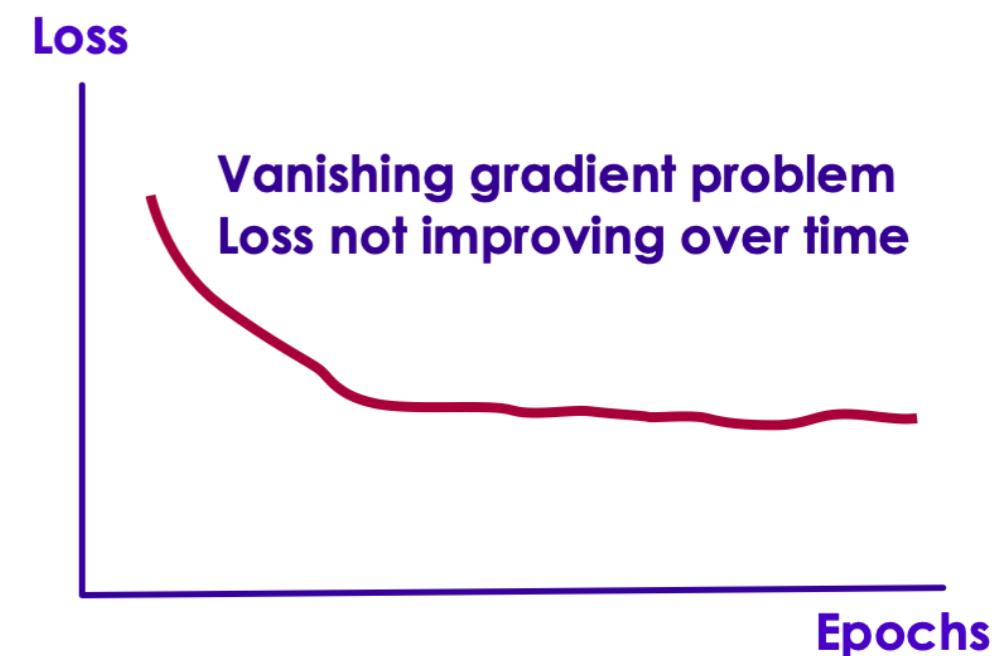
Vanishing Gradients Problem

- Here we are showing how multiplying small numbers yields smaller numbers
 - first step: $0.2 * 0.3 = 0.06$
 - second step: $0.06 * 0.5 = 0.003$
- As we move through the layers, the gradients gets smaller and smaller (approaching zero) as we reach lower layers
- So the Gradient Descent algorithm will leave lower layer connection weights virtually unchanged
 - and training never converges to a solution
 - this is **vanishing gradients problem**



Exploding Gradient Problem

- In some instances, the opposite would happen, the gradients will get larger and larger
- so layers will get huge weight updates
- and the algorithm will be bouncing around, never converging
 - this is **exploding gradients** problem
- Here we see both effects:
 - Vanishing gradient: The loss is not improving
 - Exploding gradient: The loss is actually getting worse



Vanishing/Exploding Gradients Problems

- Vanishing/exploding gradient problem has been observed in deep neural networks in the early days
 - One of the reasons, the progress was stalled
- In 2010 Xavier Glorot and Yoshua Bengio published a game changing paper called 'Understanding the difficulty of training deep feedforward neural networks'
- This paper outlined very good techniques that solved some of the nagging problems of neural nets

Use a Different Activation Function Than Sigmoid

■ Problem:

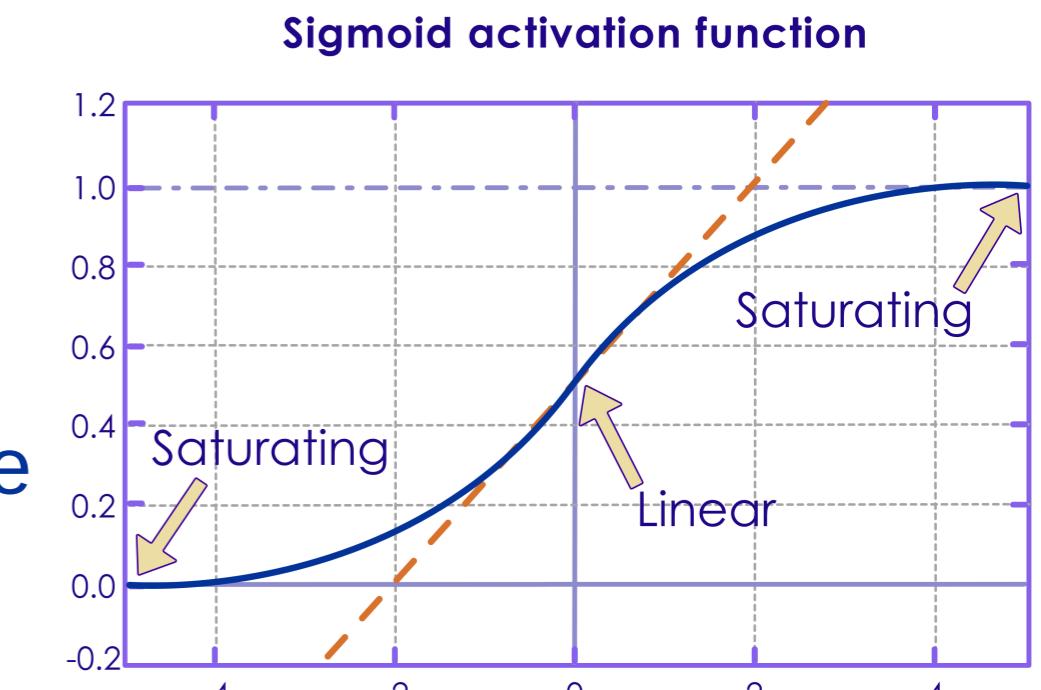
- Sigmoid function was the most popular activation function used at that time
- Because sigmoid like functions are found in biological neurons. (What is good for Mother Nature must be good for us too!)

■ However, Sigmoid functions tend to 'saturate' at high values (towards the edges), that meant derivatives get close to zero.

- Leads to vanishing gradients problem

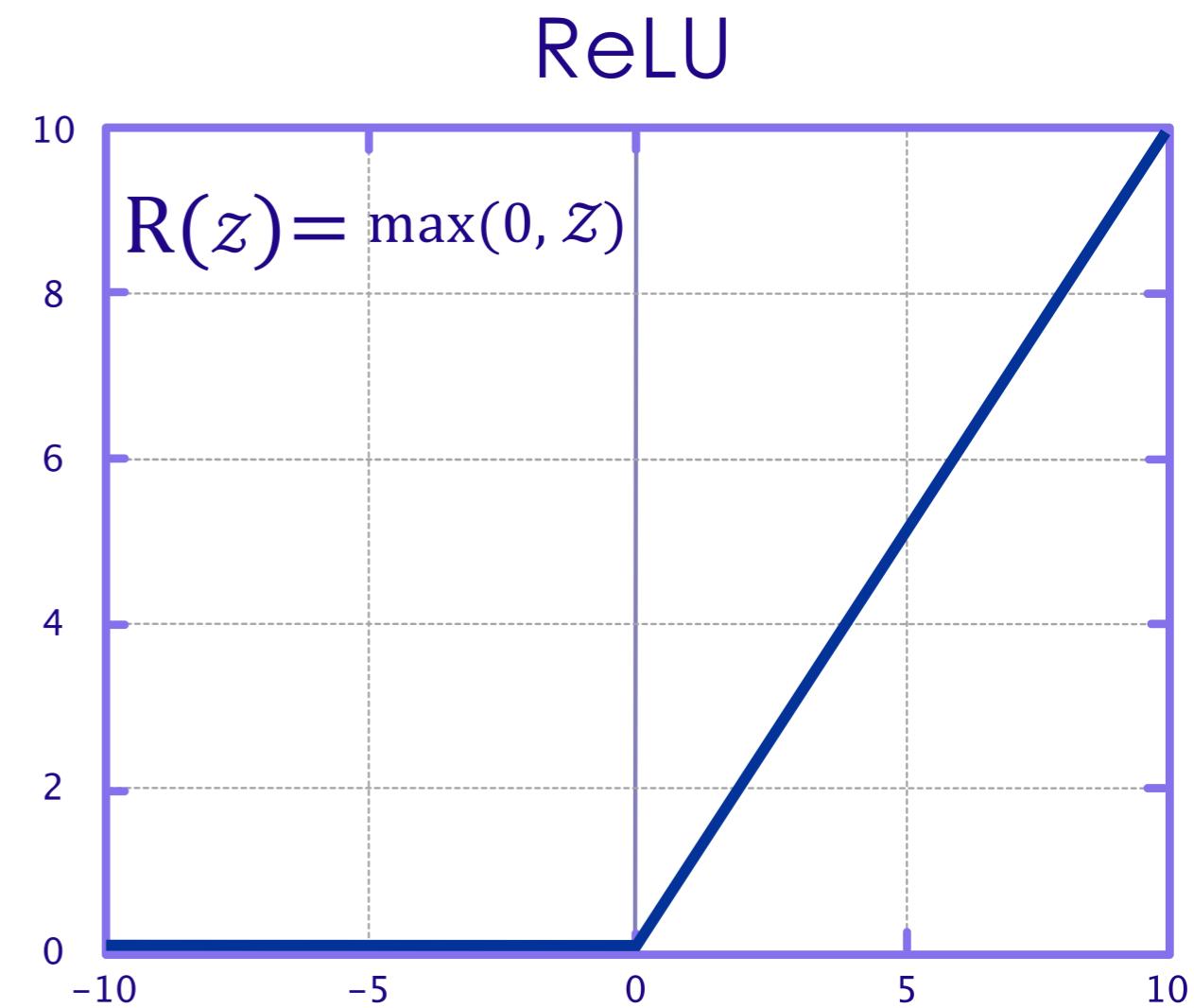
■ Solution:

- Use other activation functions like ReLU or variants (LeakyReLU)



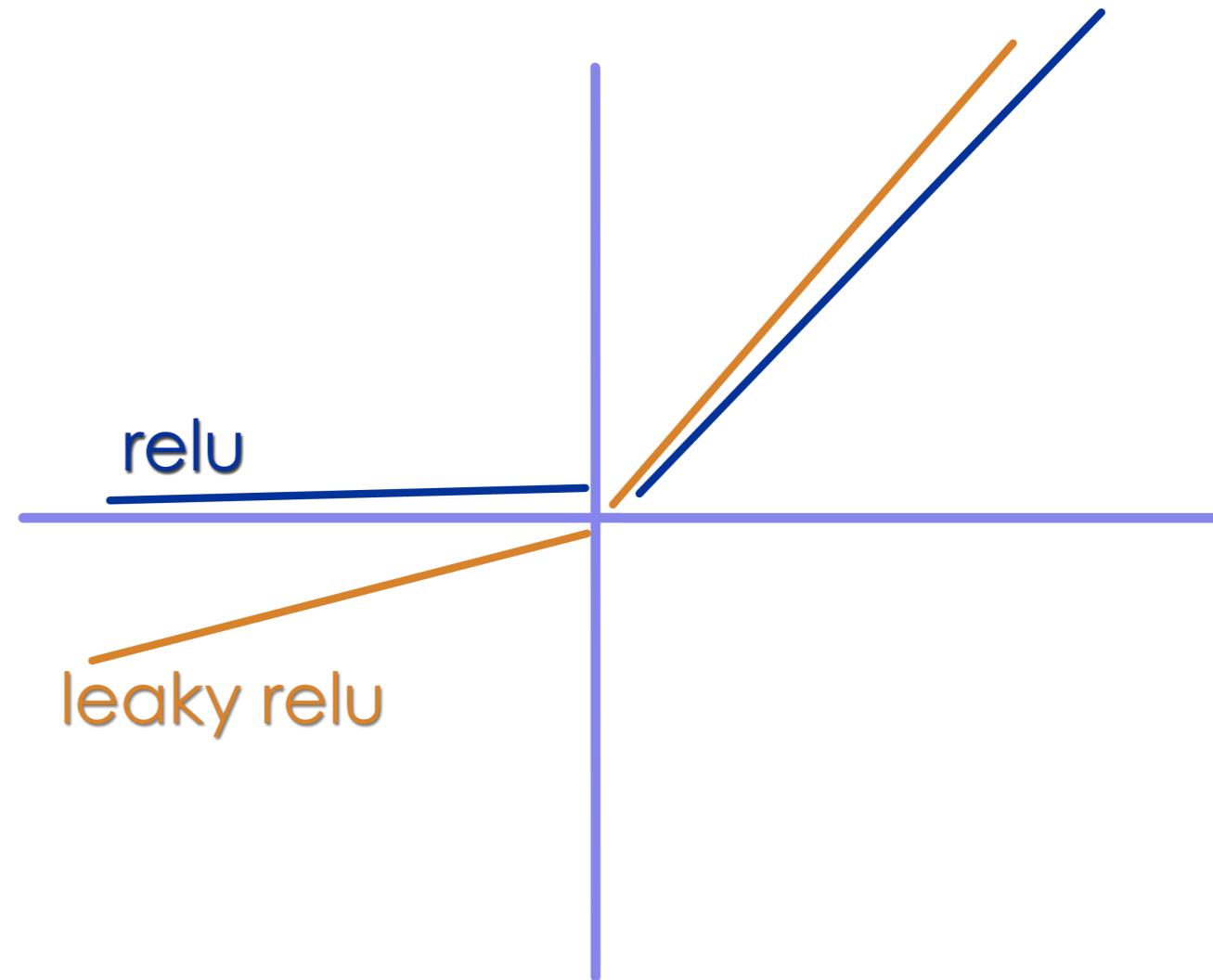
Activation Function - Rectified Linear Unit (ReLU)

- $\text{ReLU } (z) = \max (0, z)$
- ReLU is Linear when greater than zero, and constant (zero) less than zero
- So for positive values, doesn't have a maximum value
- For values less than zero, differential becomes zero
- ReLU is used very heavily
 - Simple: very easy to understand
 - Fast: computationally cheap to compute
 - No Vanishing gradient problem
 - No Exploding Gradient problem
 - and **works well in real life scenarios**



ReLU Variants: Leaky ReLU

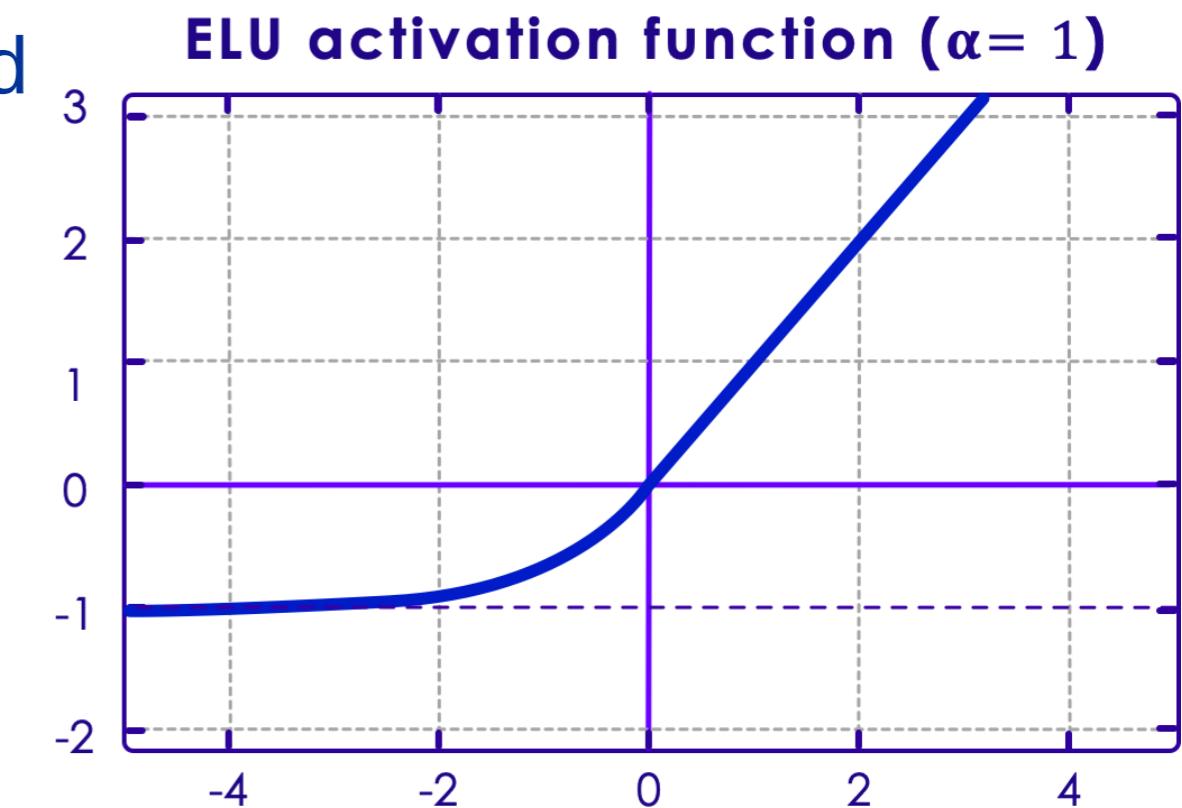
- ReLU isn't perfect; For values at or below zero or values, ReLU derivative is zero
 - Gradient Descent can not be used
 - Called **dying ReLU** problem
- **Leaky ReLU** fixes this by introducing a slope for negative values
- **LeakyReLU $\alpha(z) = \max(\alpha z, z)$**
here $\alpha = 0.001$ (small value, configurable)
- α controls how much the ReLU function 'leaks'
- This leak ensures that the signals never die (zero) and have a chance to wake up during later training phases
(Going into coma vs. death)



ReLU Variants: Exponential ReLU

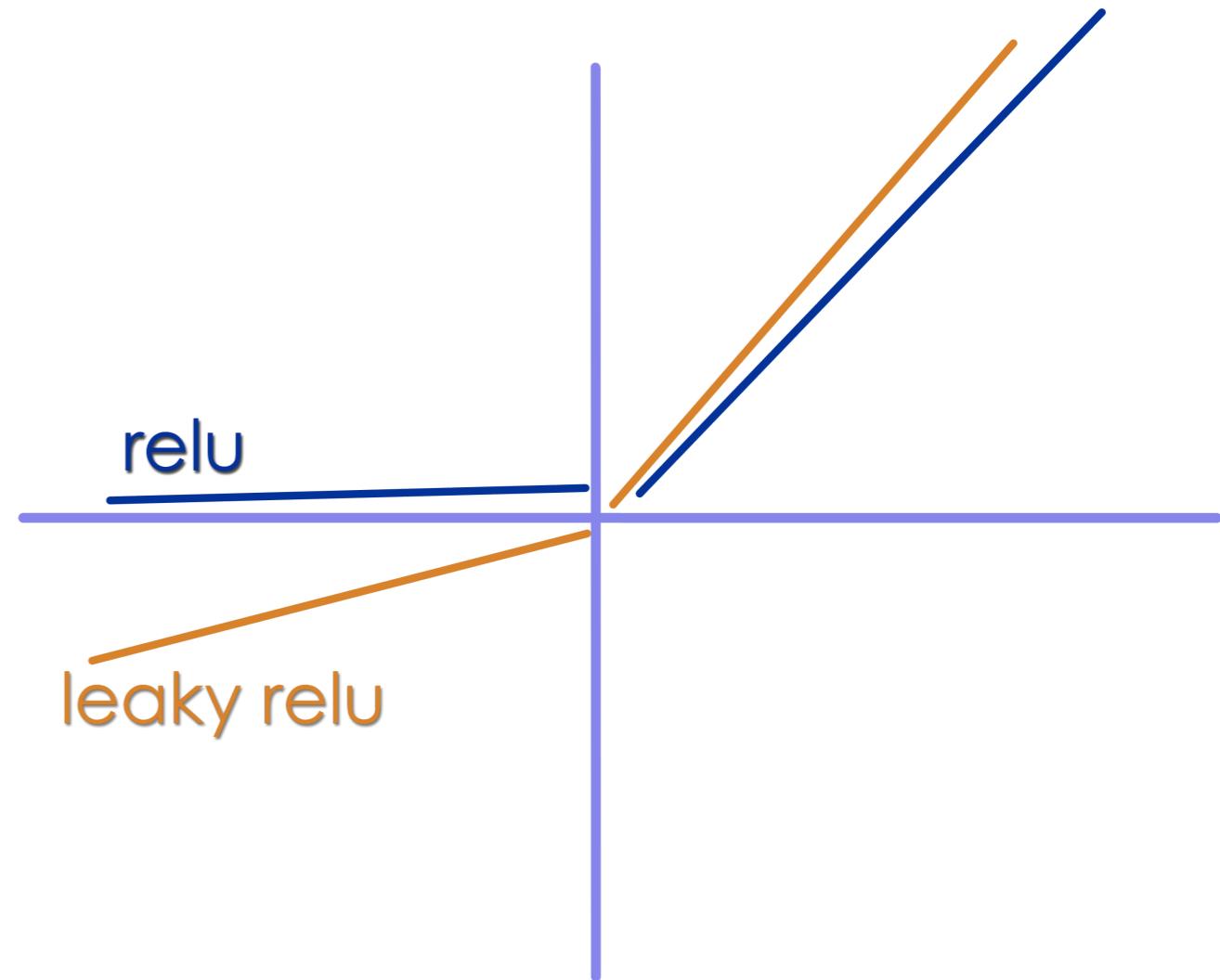
- A 2015 paper by Djork-Arne Clevert, Thomas Unterthiner & Sepp Hochreiter introduced ELUs
- ELU outperformed all other ReLU variants, it trained quicker, and test accuracy was higher too.
- Works for negative values ($z < 0$) (doesn't go to zero).
 - So avoids vanishing gradients problem
- Very smooth function, even at $z = 0$
 - This makes smoother gradient descent convergence; it doesn't bounce around
- Downside: More expensive to compute due to exponential function

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$



Comparing ReLU Variants

- This paper compares various ReLU implementations
- Leaky ReLU seems to perform better in most scenarios than regular ReLU
 - E.g. setting $\alpha=0.2$ (huge leak) seems to outperform $\alpha=0.001$ (small leak)
- Setting α randomly also seems to perform well
 - Got an additional benefit of acting as a regulator, preventing overfitting



Final Word on Activations

- So which activation function to use? :-)

	Advantages	Disadvantages
Sigmoid	- Will not lead to exploding gradients	- will lead to vanishing gradients - computationally expensive to calculate
ReLU	- Not lead to vanishing gradients - Easy to compute than sigmoid	- May lead to exploding gradients

- In practice (real world scenarios), ReLU tend to show better convergence performance than sigmoid
- **ELU > leaky ReLU (and its variants) > ReLU > tanh > sigmoid**
- If enough compute power is available, use **cross validation** to tweak hyper parameters like α
- References:
 - Relu vs. Sigmoid

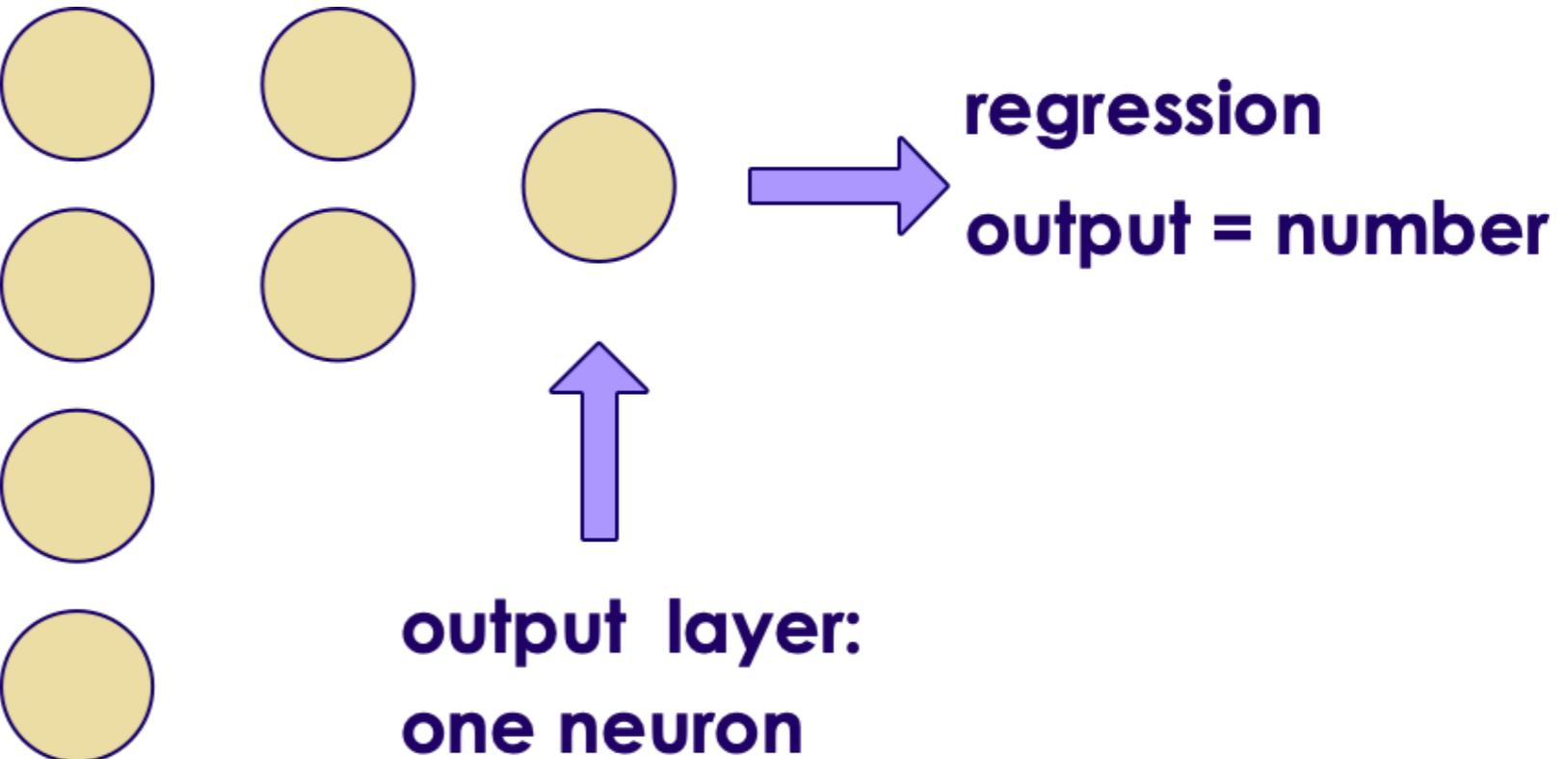
Activation Functions for Regression

- For regression problems, the output neuron generates the response variable (a continuous value)

- e.g. stock price = 60.4

- The following can be used as activation Functions

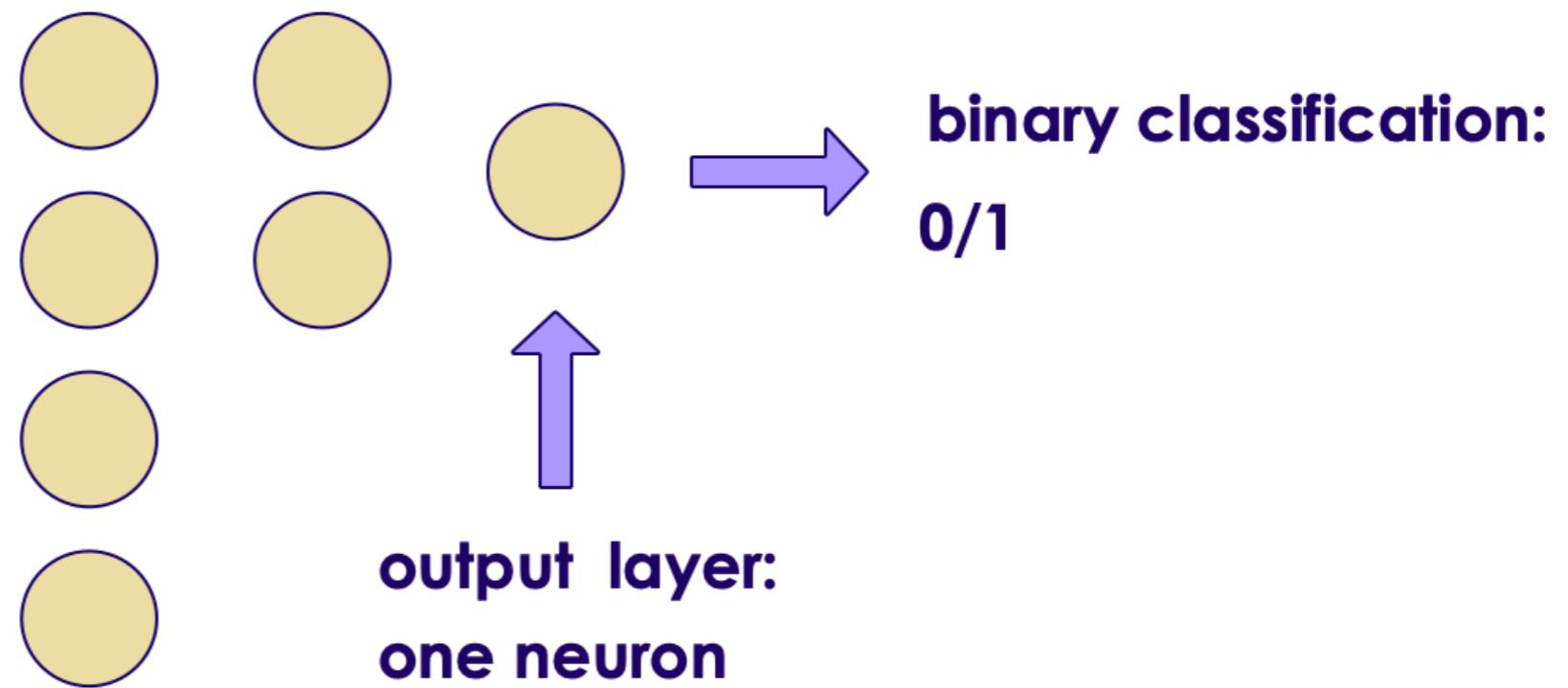
- Linear
- Sigmoid / Logistic
- Tanh
- ReLU variants



Activation Function for Binary Classification

- For classification problems, the output is binary (0/1 or True / False)

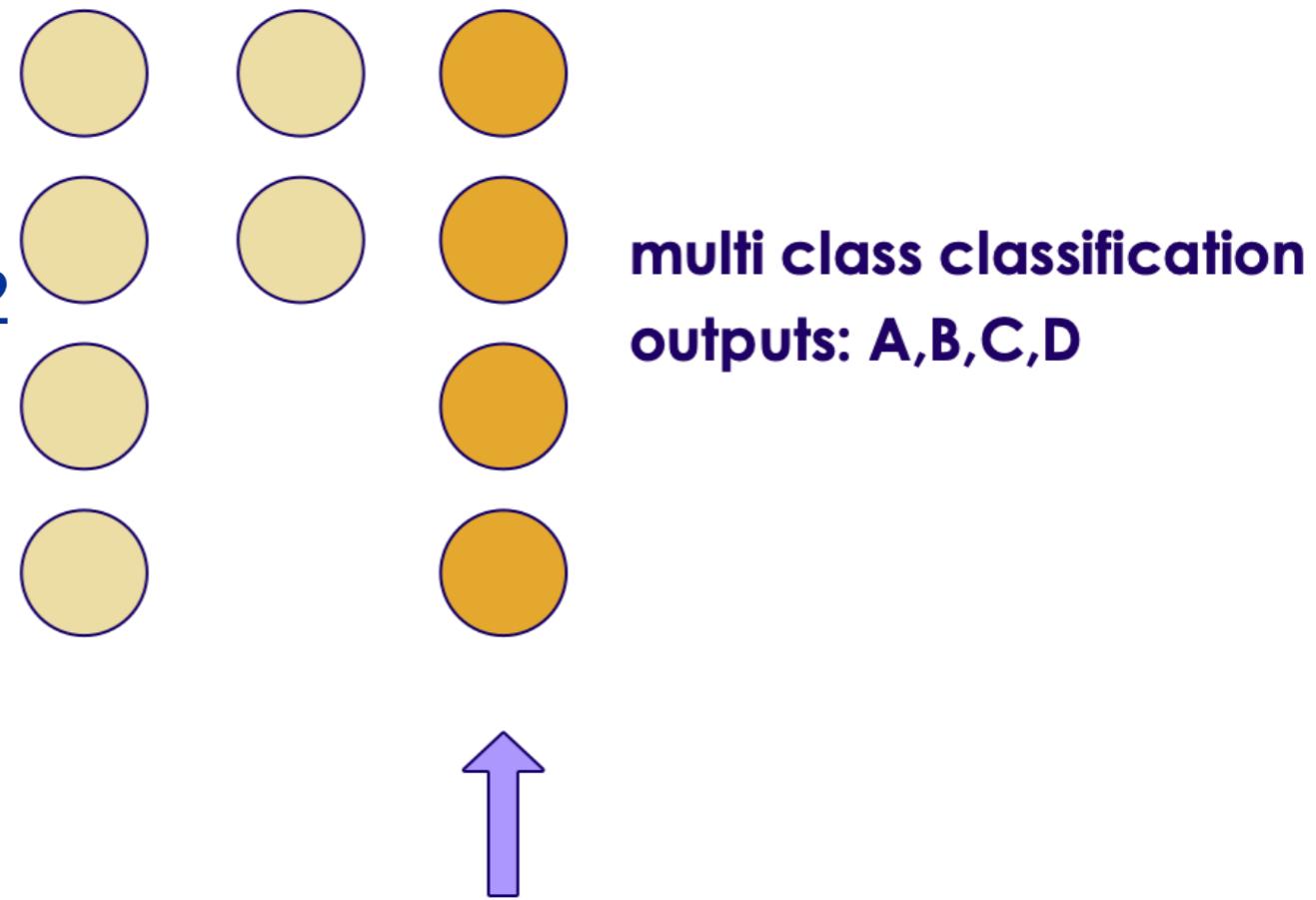
- e.g. is the transaction fraud? True / False
- e.g. will this loan default? True / False



- Again one neuron at the output layer can handle this
- Activation Functions to use:
 - Sigmoid / Logistic

Activation Function for Multi-class (non-binary) Outputs

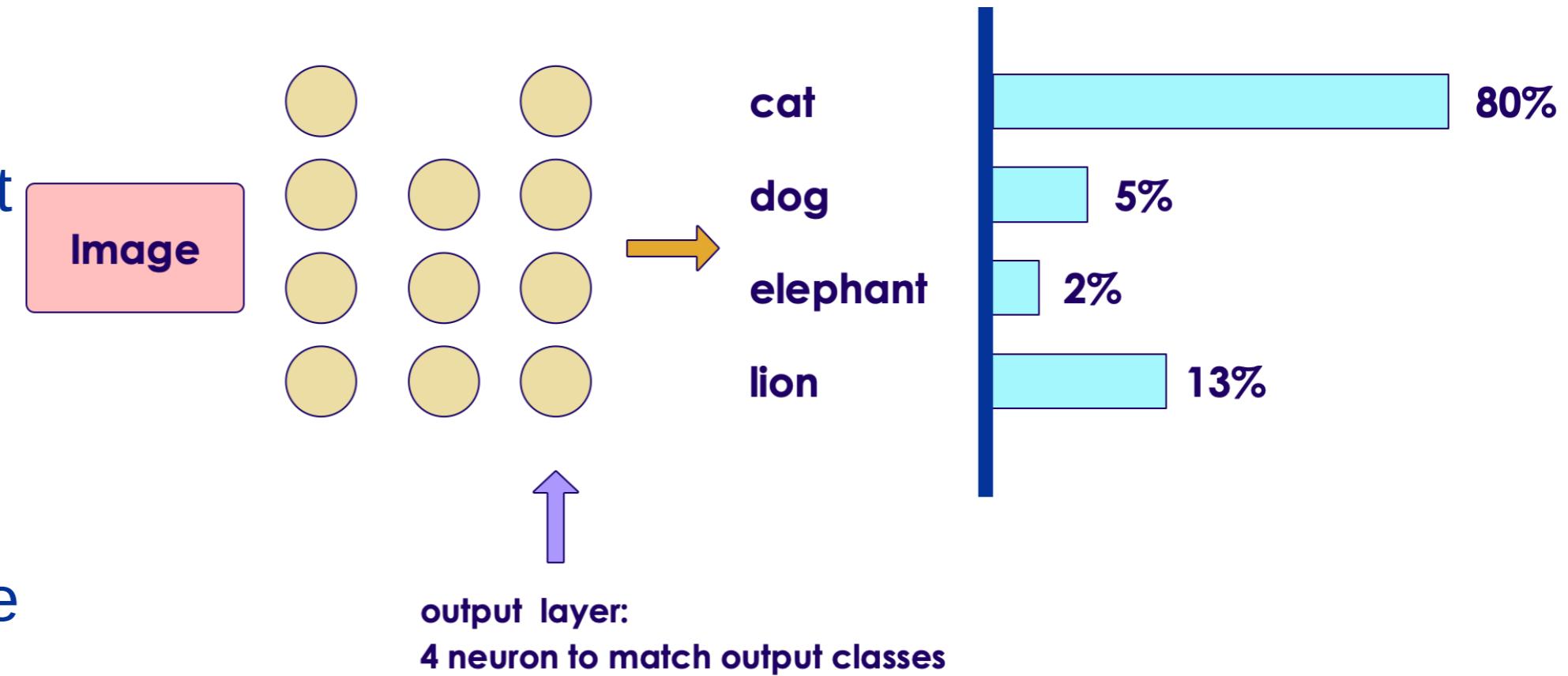
- What about multi-class classification? (non-binary)
 - e.g., classifying a digit into one of $0, 1, 2, \dots, 9$
- We need more than one output neuron.
- Exactly one neuron for each class in classification.
- How do we generate the output classes?
 - We can use a function called Softmax



**output layer:
4 neuron
to match output classes**

Activation Function - Softmax

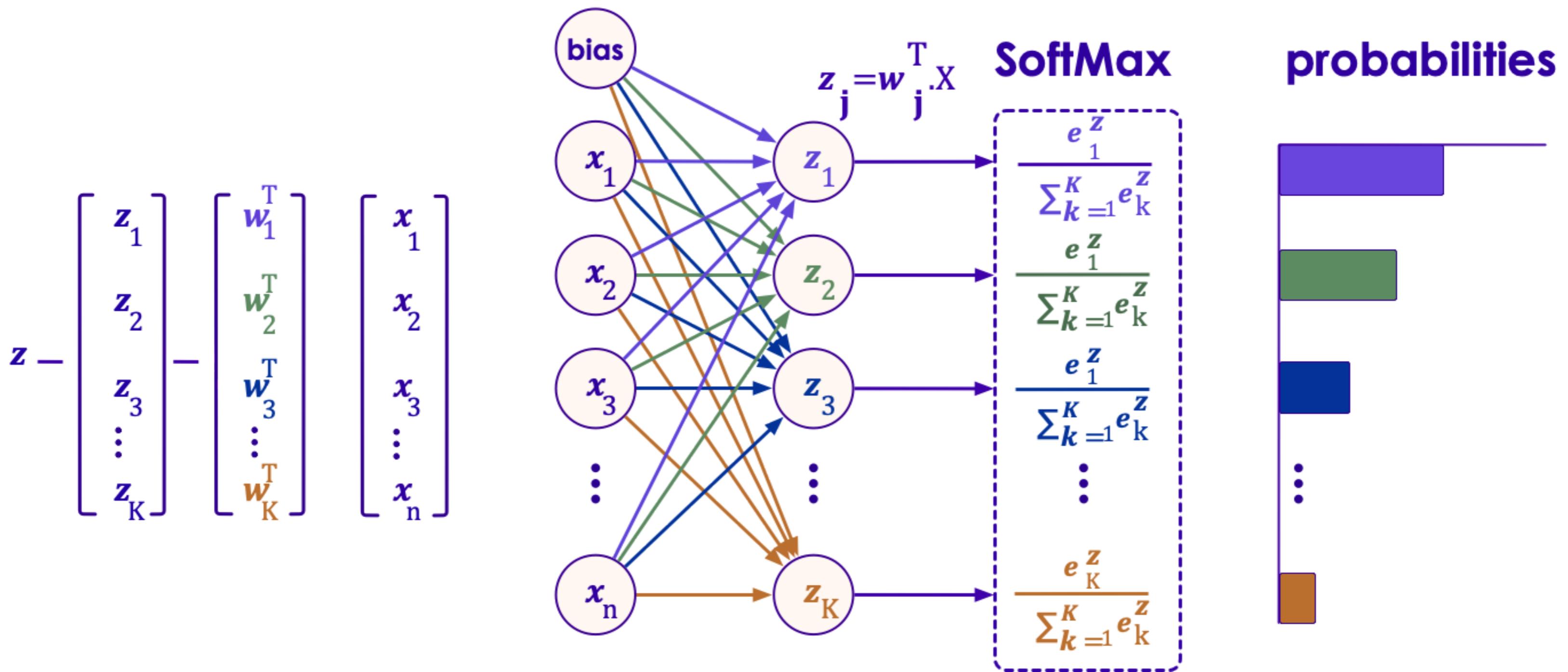
- Assume we have an image classifier that classifies animals into 4 categories: cat / dog / elephant / lion
- So the final layer will have 4 neurons, so they match the output classes (4)
- And the output layer will have **softmax** activation function
- The Softmax function produces probabilities for output values.
- In the above example, prediction is **CAT** as it has the highest probability (80% or 0.8)



Output Class	Cat	Dog	Elephant	Lion
Probability (Total 1.0)	0.80	0.05	0.02	0.13

Softmax Function

Multi-Class Classification with NN and SoftMax Function



Activation Functions - Review

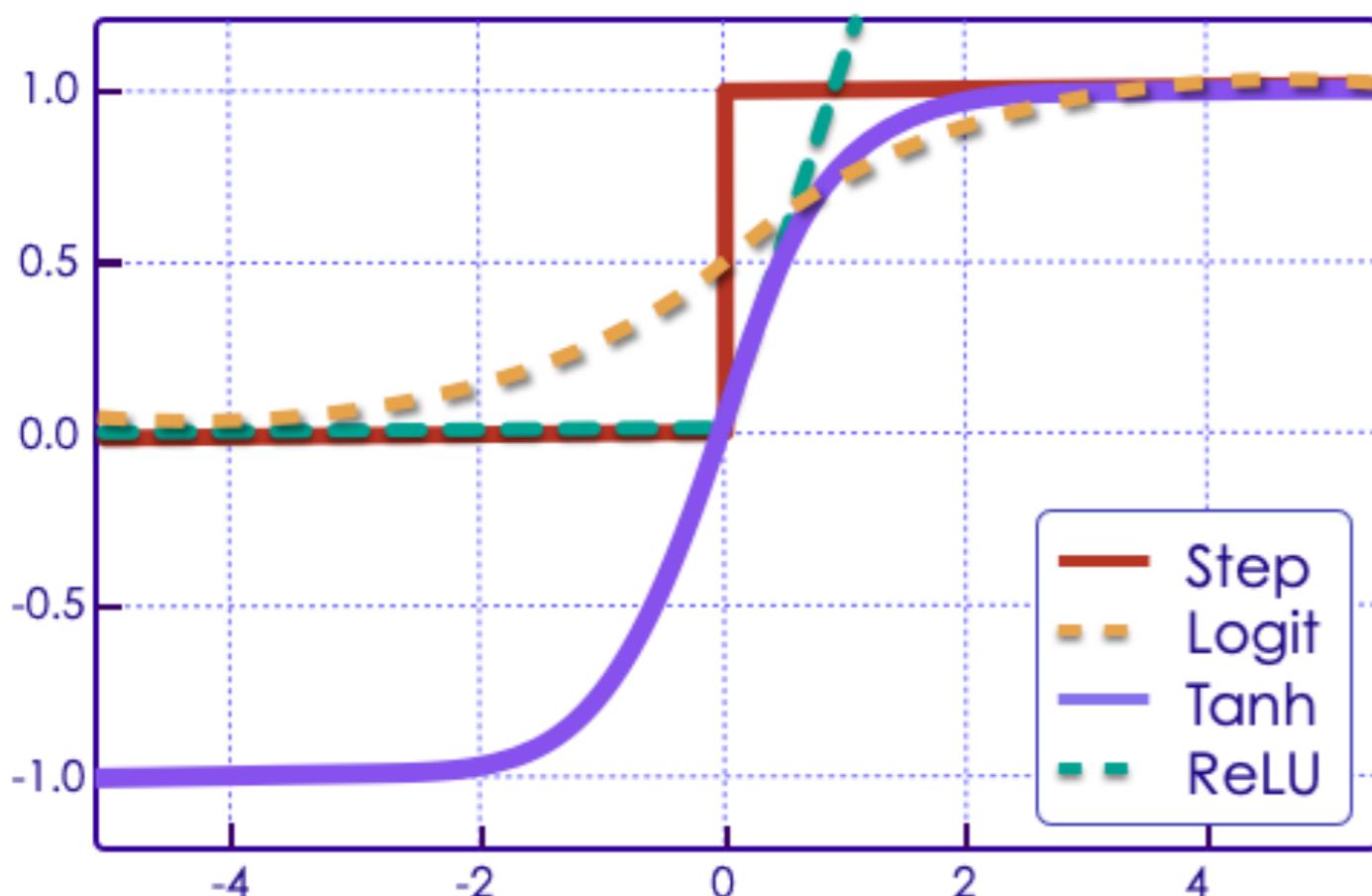
- A **Sigmoid Function** is a mathematical function with a Sigmoid Curve ("S" Curve) and outputs a probability between 0 and 1.
- A rectifier or **ReLU** (Rectified Linear Unit) is a commonly used activation function that allows one to eliminate negative units in an ANN. It helps solve vanishing/exploding gradient problems associated with other activation functions.
- Hyperbolic or **$\tan h$** function is an extension of logistic sigmoid with output stretching between -1 and +1. It enables faster learning compared to a Sigmoid.
- The **Softmax** activation function is used to output the probability of the result belonging to certain classes.

Activation Type Based on the Task

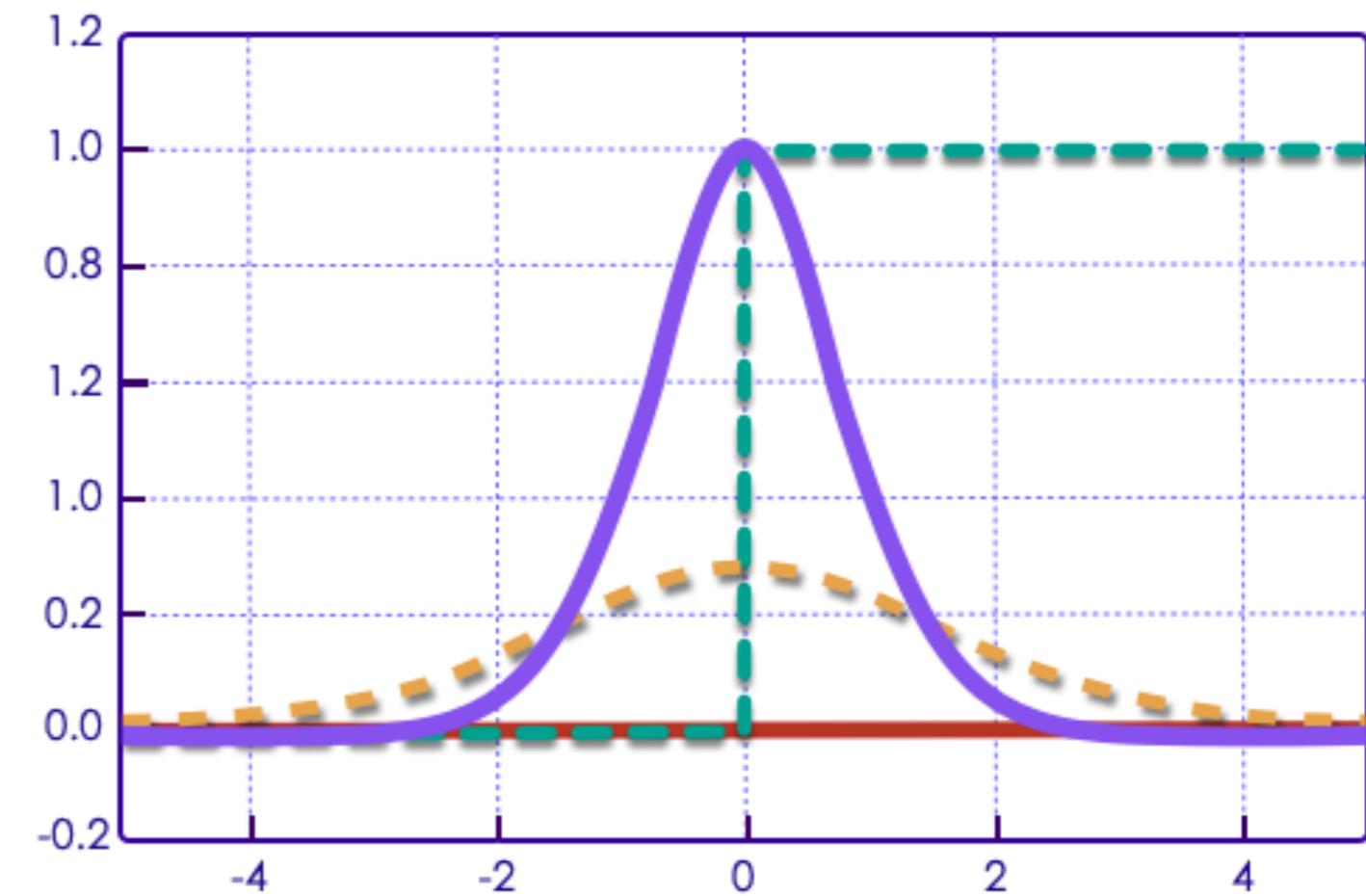
Problem Type	Prediction	Activation on the last layer
Regression	a number	linear, relu
Classification	binary (0/1)	sigmoid

Activation Functions Review

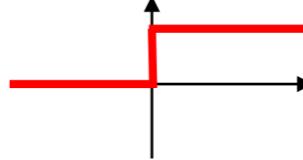
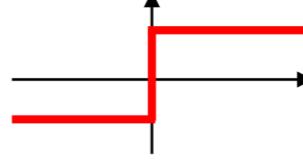
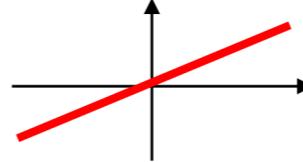
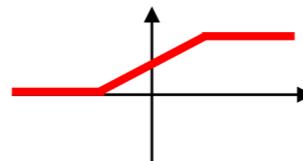
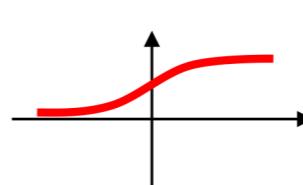
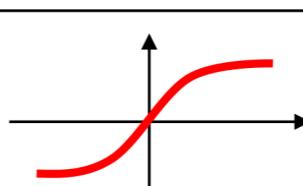
Activation functions



Derivatives



Activation Functions Cheatsheet

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

■ source: [quora](#)

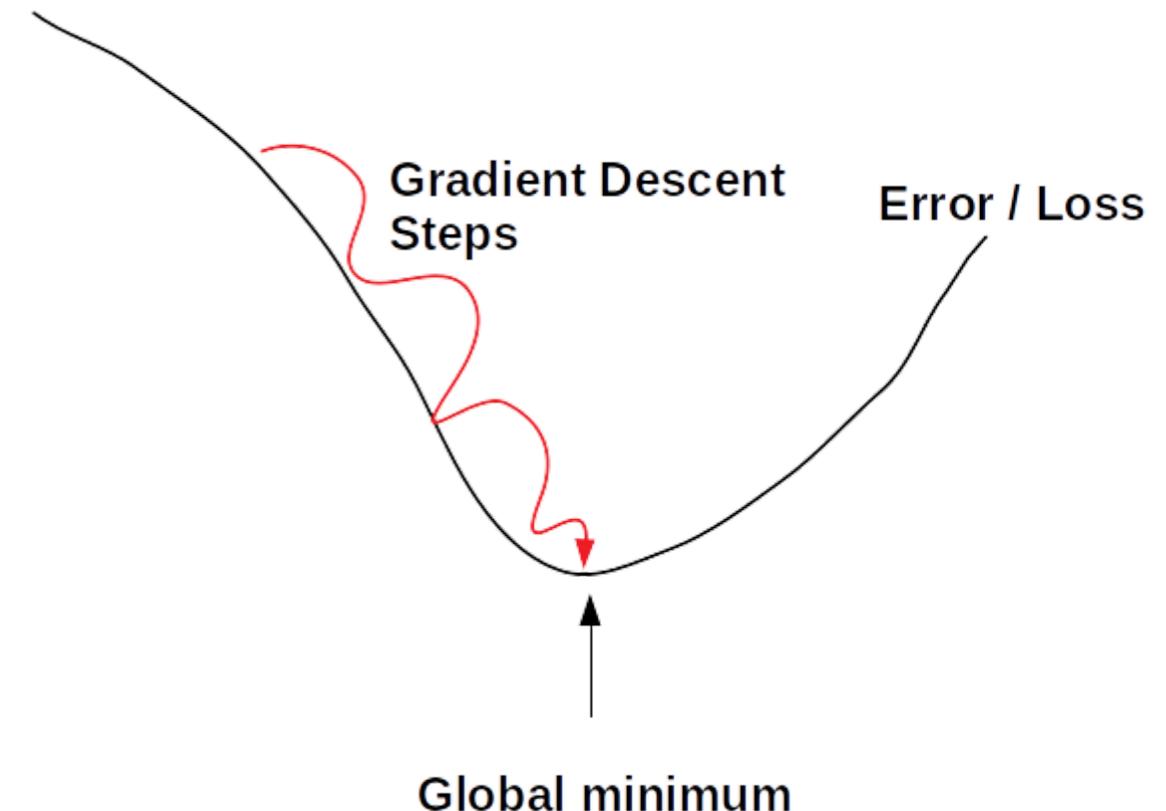
Take-away: Deciding Loss / Activation Functions

Problem Type	Prediction	Loss Function	Activation on the last layer
Regression	a number	mse, sse, mae	linear, relu
Classification	binary (0/1)	binary_crossentropy	sigmoid
	Multi-class (A, B, C , D)	categorical_crossentropy	softmax

Learning Rate

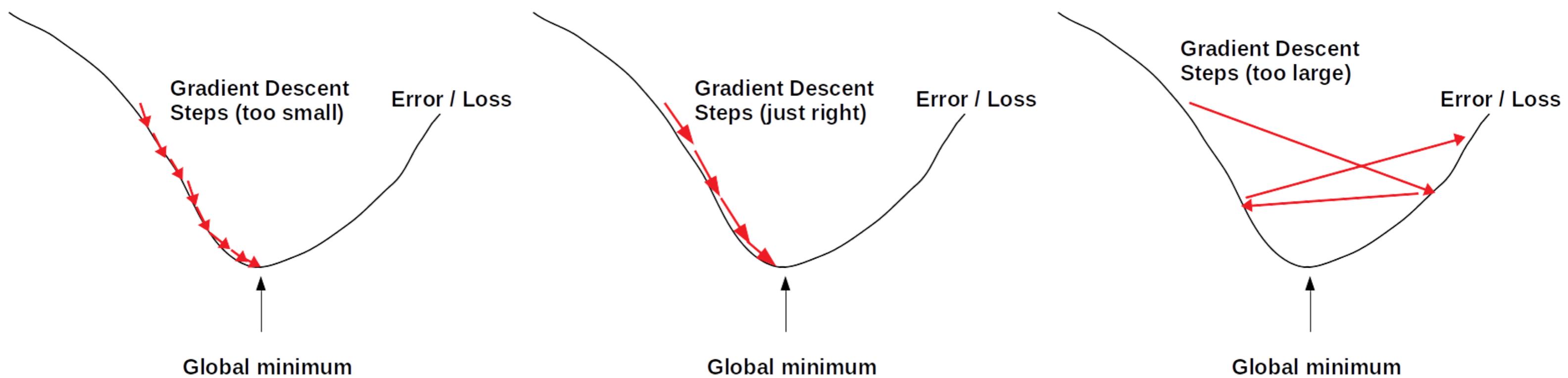
Learning Rate

- Neural Networks update their weights using backpropagation
- The amount the weights are updated is called **step size** or **learning rate**
- Learning rate is a positive number (usually between 0.0 and 1.0 - can be more than 1.0 in some cases)



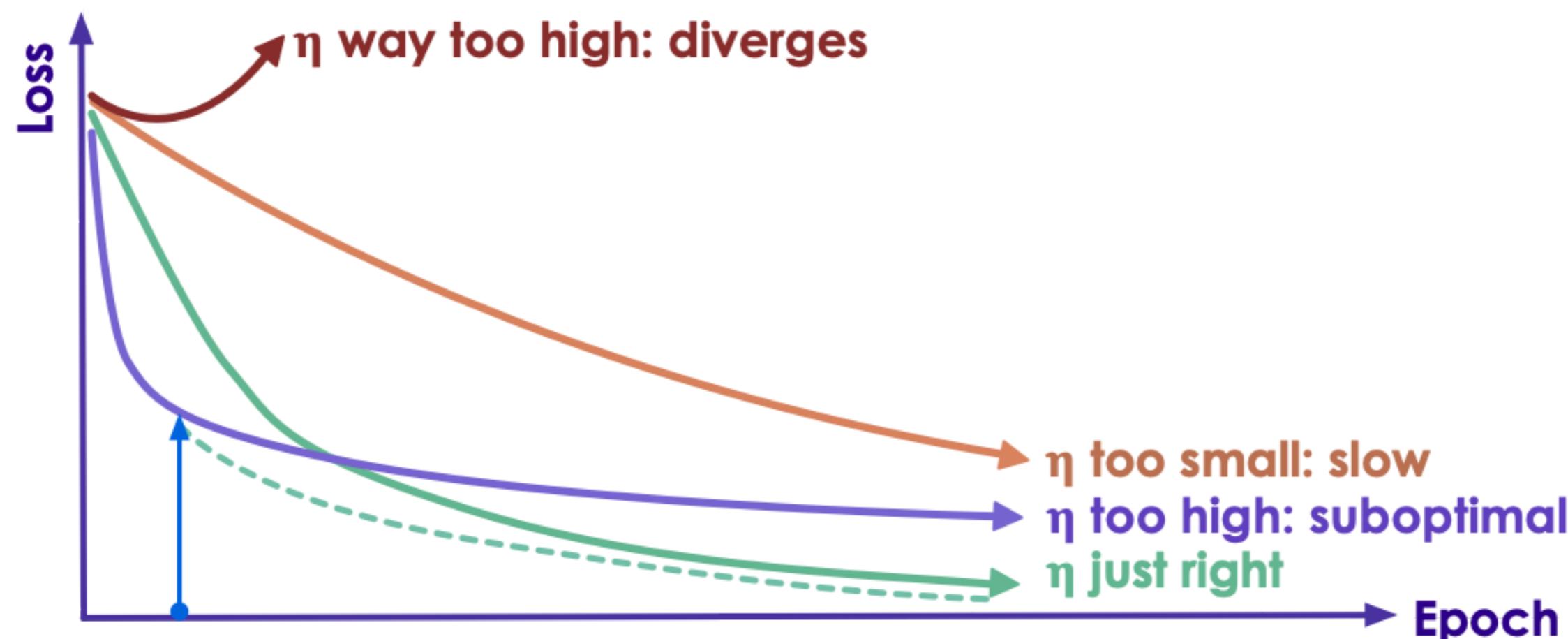
Effect of Learning Rate

- When 'learning rate' is **too small**:
 - The weight updates are small
 - The model may take longer to train (too many steps to find the solution)
- When the 'learning rate' is **too large**:
 - The weight updates are too large
 - It may cause the model to diverge, and bounce around
- Our goal is to find the **right learning rate**



Finding the Optimal Learning Rate

- Learning Rate is a very important factor in the algorithm converging (finding the global minimum)
- We don't want it to be too large or too small
- Set it too high, algorithm may diverge
- Set it too low, algorithm will eventually converge, but will take too many iterations and too long



Finding the Optimal Learning Rate

- Unfortunately, there is **no formula** to calculate the optimal learning rate
- Learning rate is determined by **experimentation** and following **best practices**
- **Learning curve** (from Tensorboard) can give us clues on how effective the learning rate is
- *"The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate."* - Page 429, Deep Learning, 2016.



Andrej Karpathy  @karpathy · Nov 23, 2016
3e-4 is the best learning rate for Adam, hands down.

25

127

466



Determining Learning Rate



Andrej Karpathy ✅ @karpathy · Nov 23, 2016

3e-4 is the best learning rate for Adam, hands down.

25

127

466



Replies



Andrej Karpathy ✅ @karpathy · Nov 23, 2016

Replying to @karpathy

(i just wanted to make sure that people understand that this is a joke...)

9

4

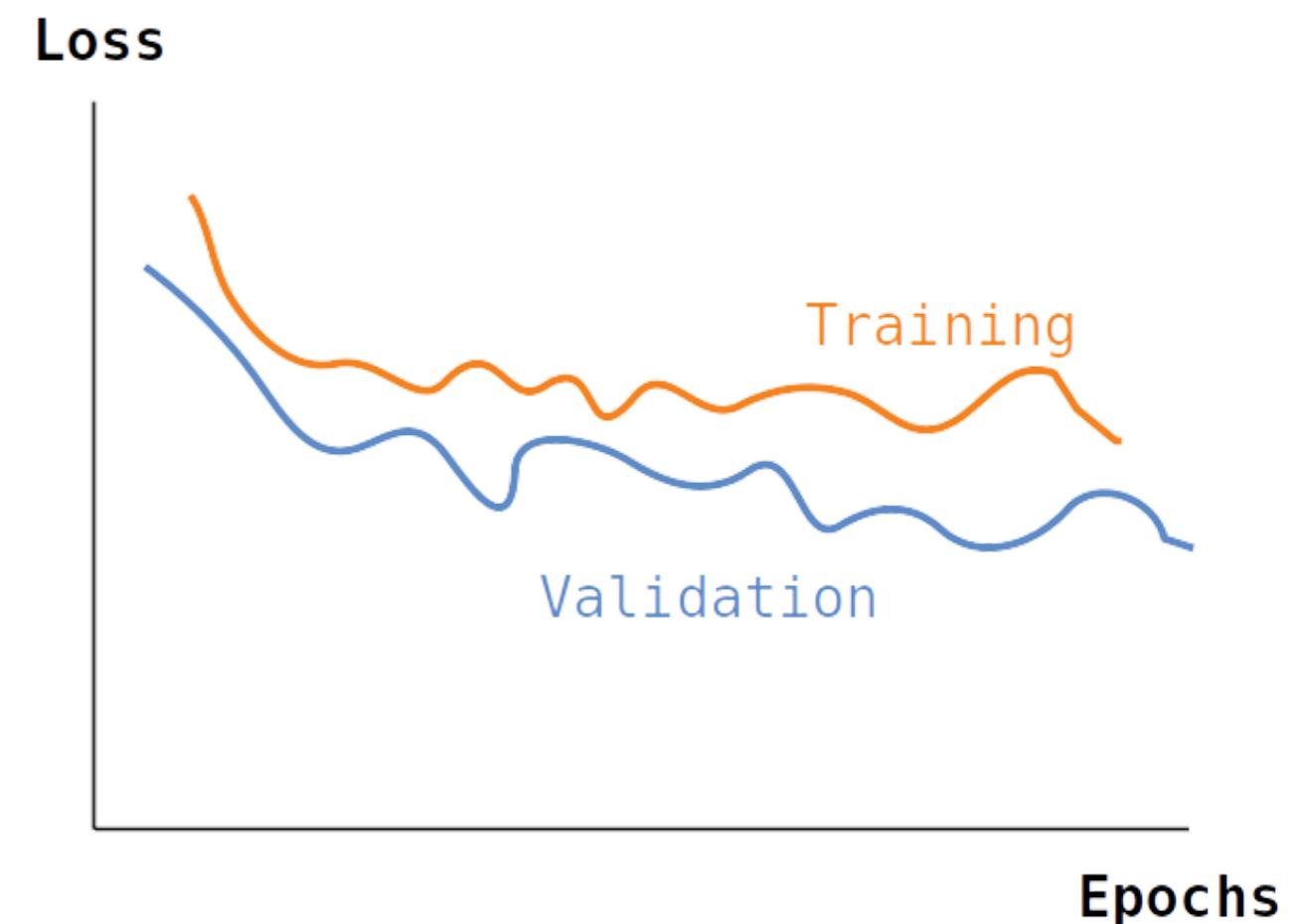
119



- Read the tweet thread for some funny reactions
- Also Andrej's twitter and karpathy.ai

How to Find the Optimal Learning Rate

- Start with learning rate of **0.1 or 0.01**
- Run a few epochs of training
- Watch the convergence using a tool like Tensorboard
- Adjust learning rate, rinse and repeat



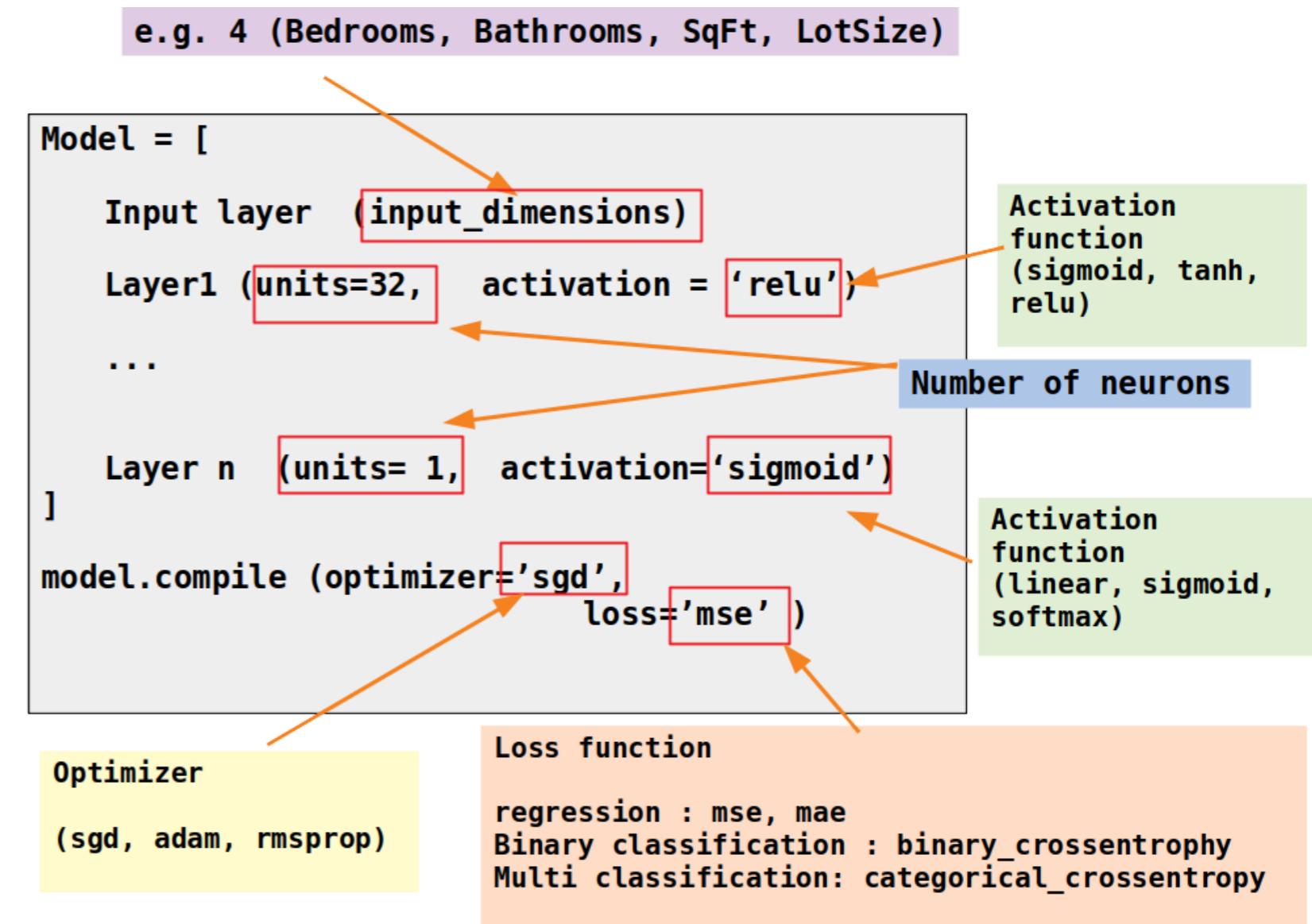
Finding Optimal Learning Rate

- Stochastic Gradient Descent (SGD) algorithm has 'fixed' learning rate
- In practice, **adjusting the learning rate** results in the algorithm converging sooner
- Modern optimizers like Adagrad, RMSProp and Adam have **adaptive learning rate**
 - They can adjust learning rate as training progresses
- We will see more of this in the next section **Optimizers**

Optimizers

Optimizers Overview

- Optimizers help determine weights during training phase
- Various optimizers
 - Gradient Descent
 - Momentum Optimizer
 - Nesterov Accelerated Gradient
 - AdaGrad
 - RMSProp
 - Adam
- References:
 - Various optimizers compared



About This Section

- The first part covers popular optimizers and how to use them
- The math for optimizers can be found in the 'appendix' section (provided as reference; not covered in class)

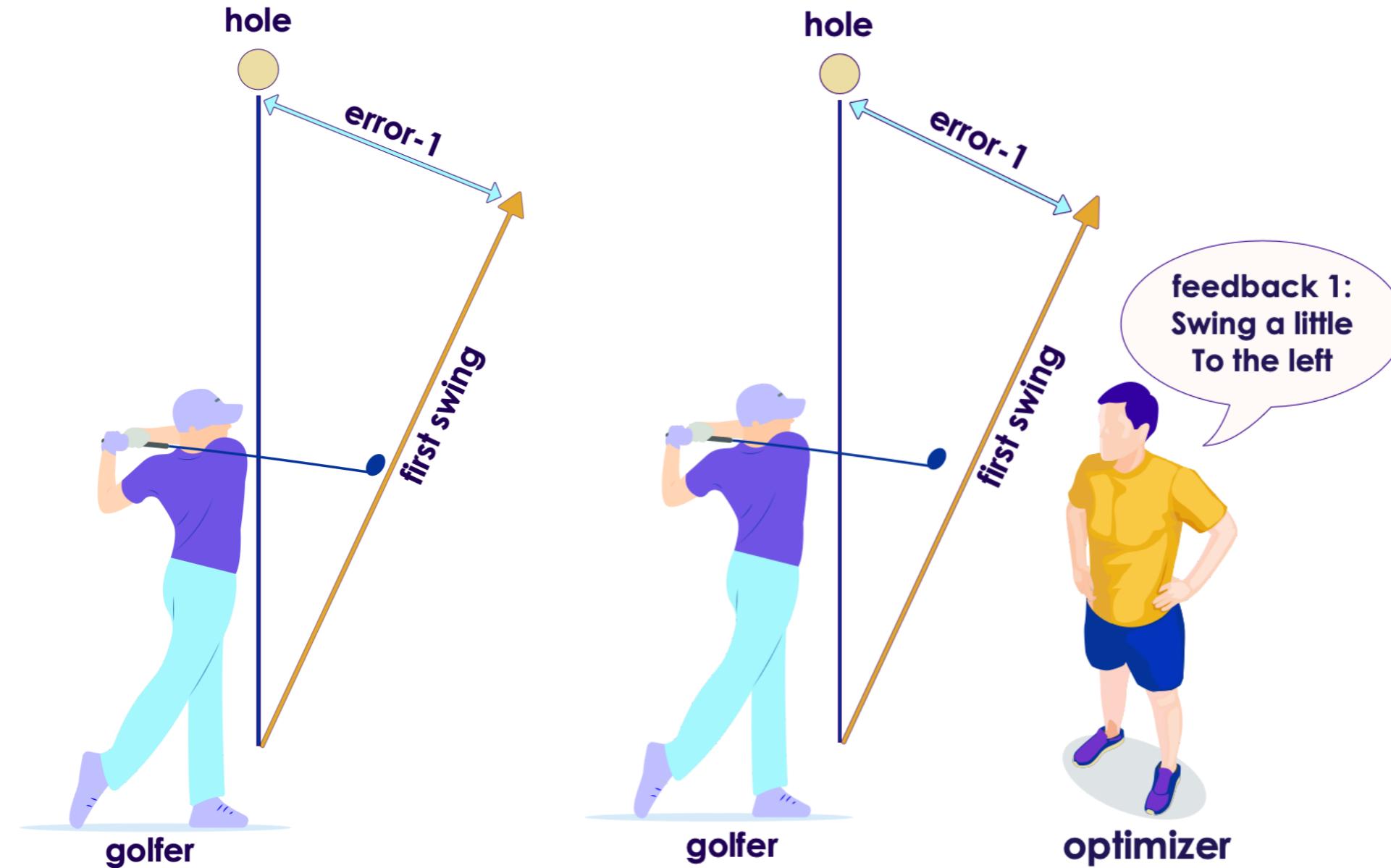
Understanding Optimizers: A Golf Game Analogy :-)

- Imagine a golfer is trying to get the ball in the hole
- He is getting help from a coach
- Coach is giving him feedback after each shot



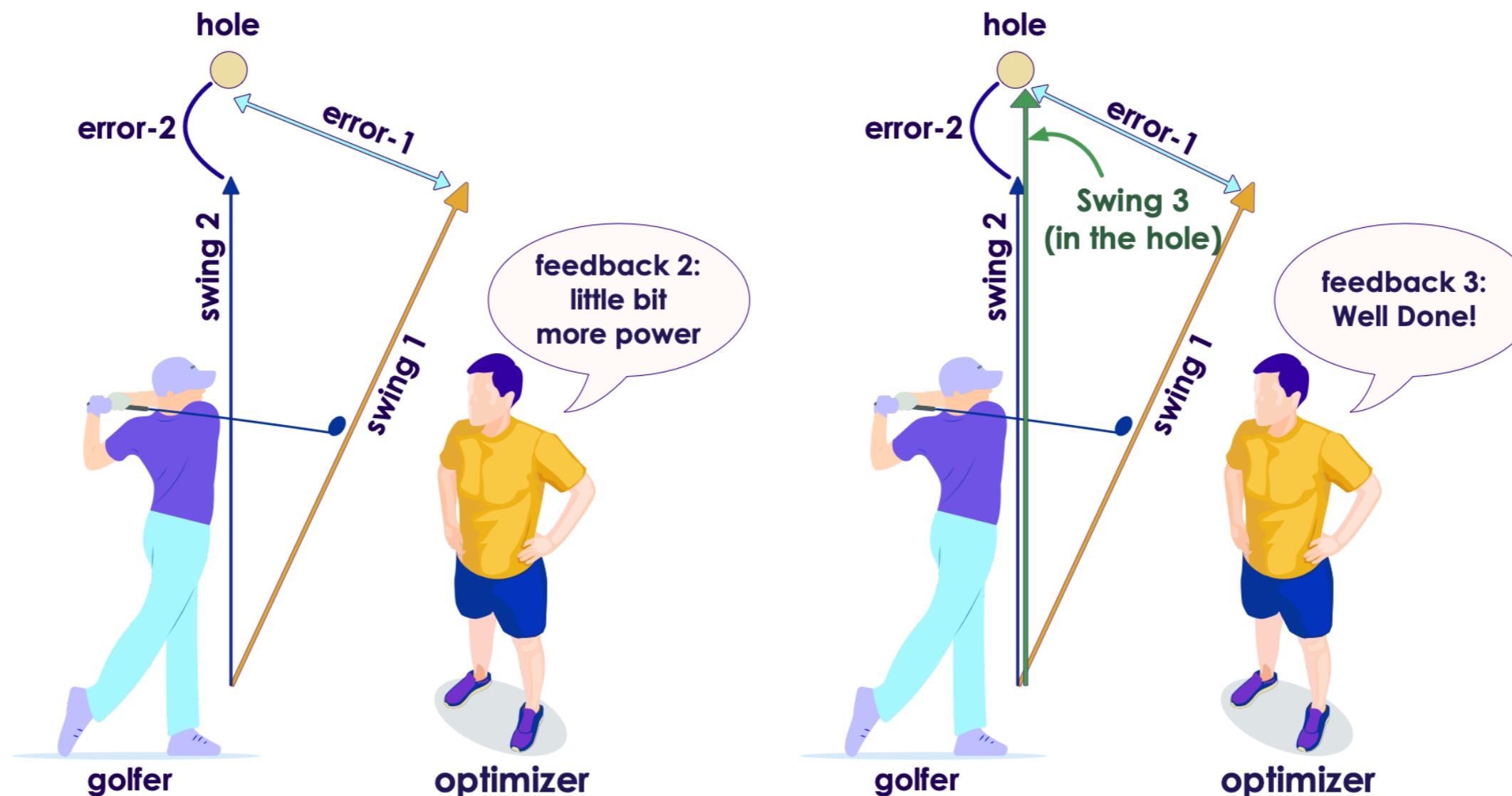
Optimizing a Golf Game

- First swing is off to the right a little
- Error is measured by **loss function** (error-1)
- Optimizer ('golf coach') gives feedback on first shot
 - *"swing a little to left"*

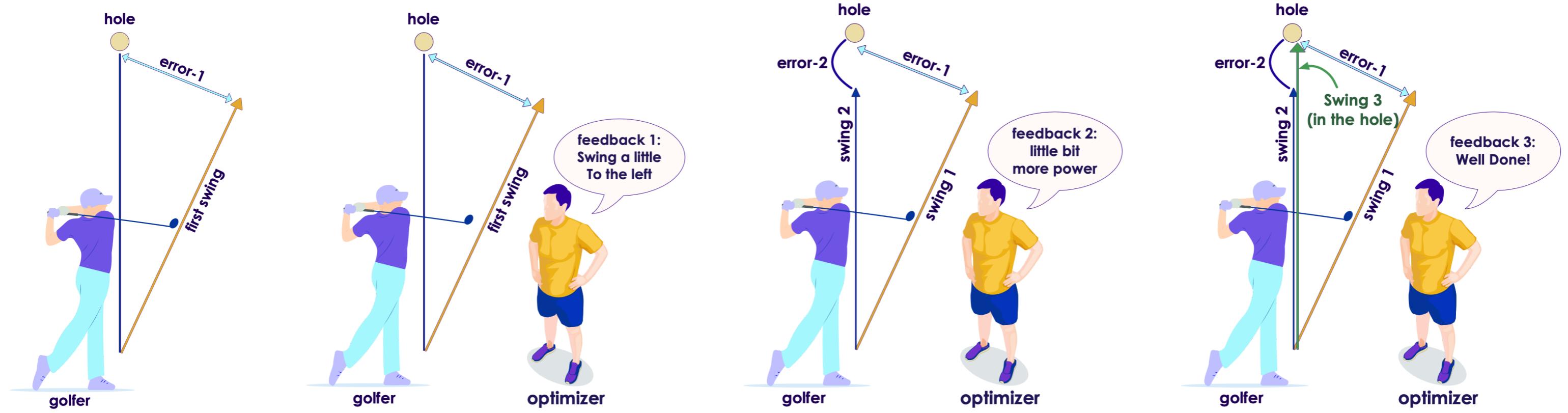


Optimizing a Golf Game

- Second shot is aimed at the hole, but stops a little short (*error-2*)
- Optimizer corrects the swing again
 - *"give it little more power"*
- And the third swing makes the hole!

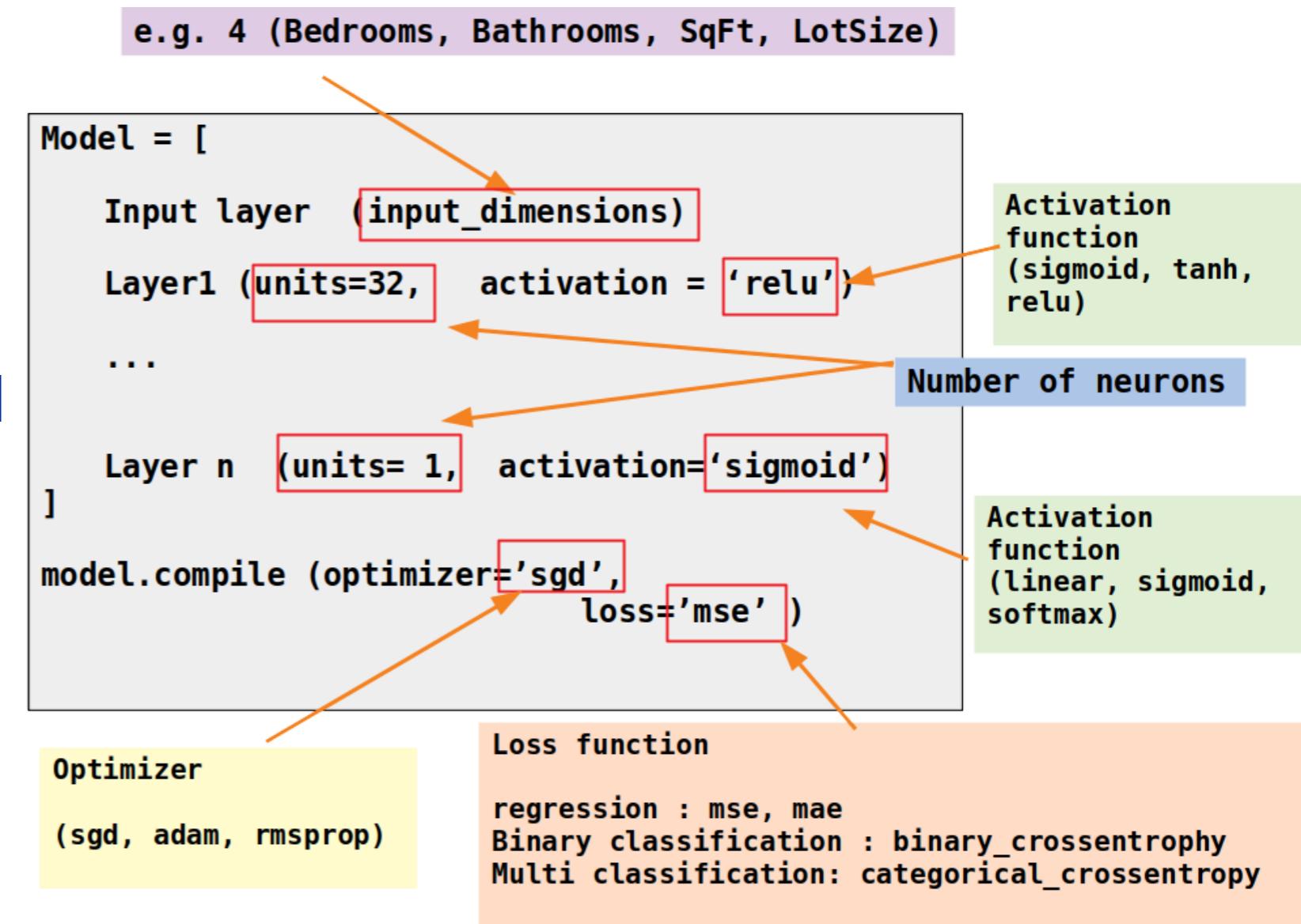


Optimizing a Golf Game - Summary



Popular Optimizers

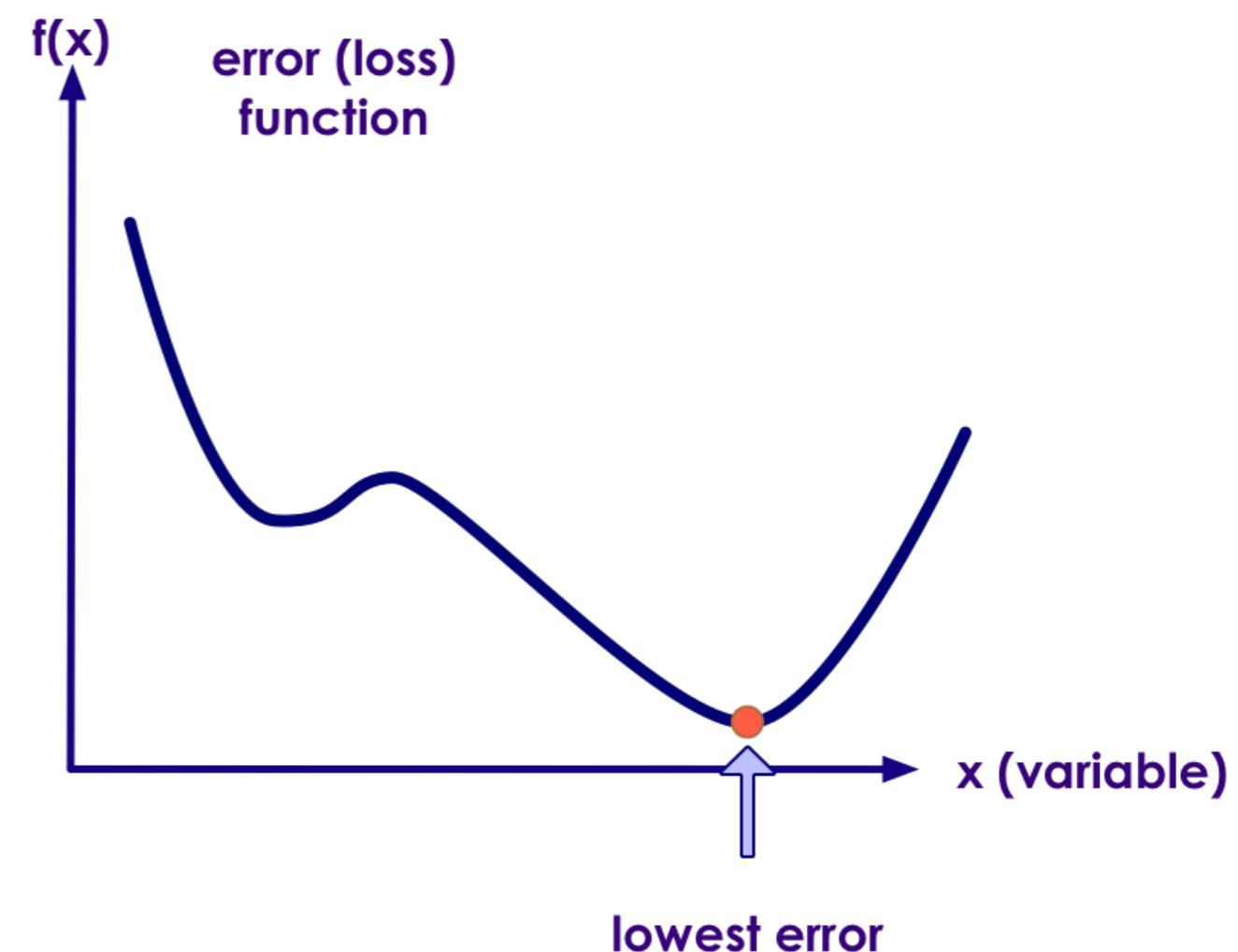
- There are various optimizer implementations; We will focus on 3 most popular ones
- **Stochastic Gradient Descent (SGD)** is the original implementation, and still heavily used
- **RMSProp** is one of the 'go to' optimizers now. It features 'adaptive learning'
- **Adam** is also one of the 'go to' optimizers now. It features 'adaptive learning'



Gradient Descent Algorithm

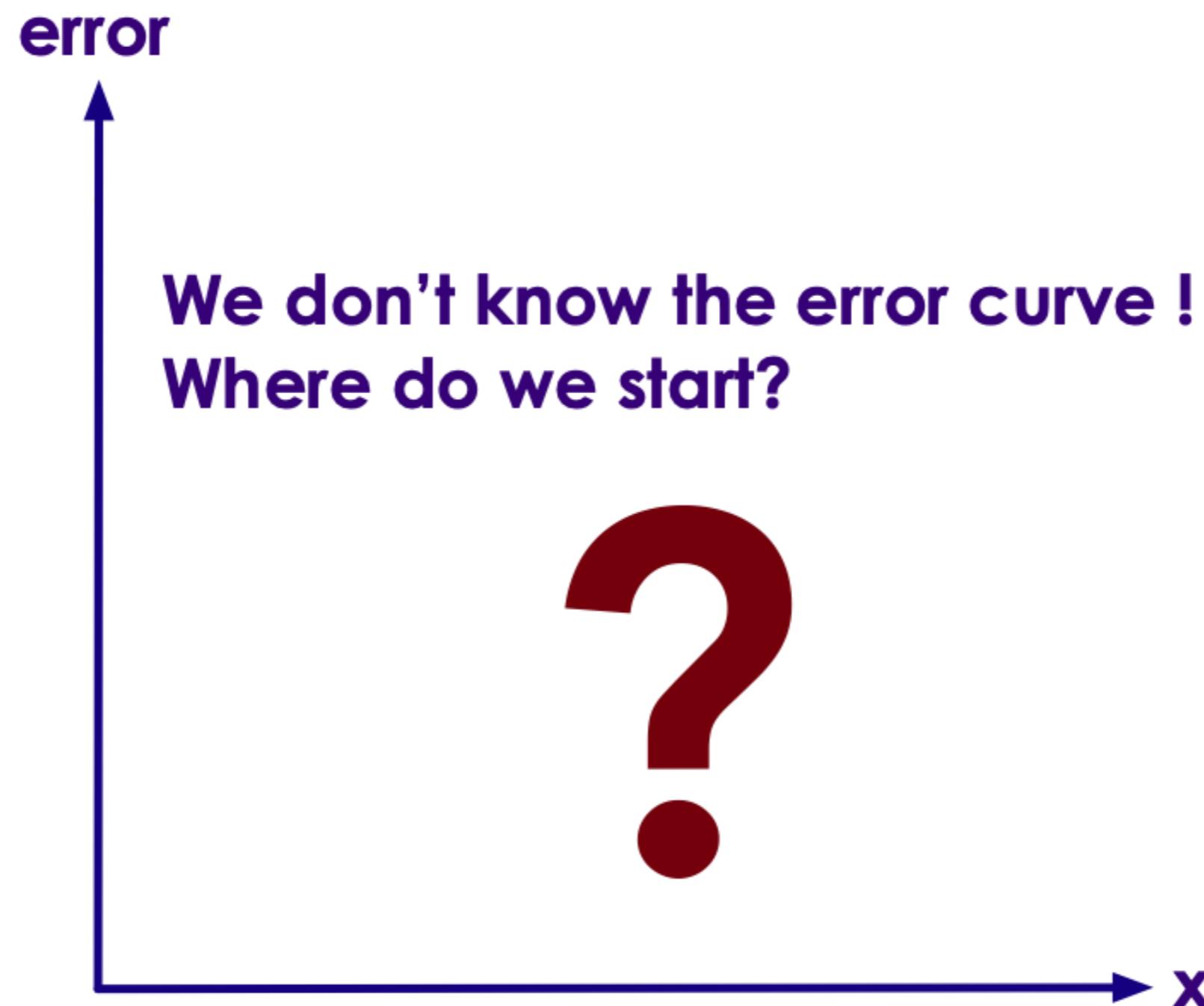
Find the Optimal Value for a Variable

- Now we understand **error/loss functions** let's try this:
- Here when we change variable X, the error function changes
 $\text{Error} = F(x)$
- Goal: find the optimal X that gives me the lowest error
- How ever there is a catch! (See next slide!)



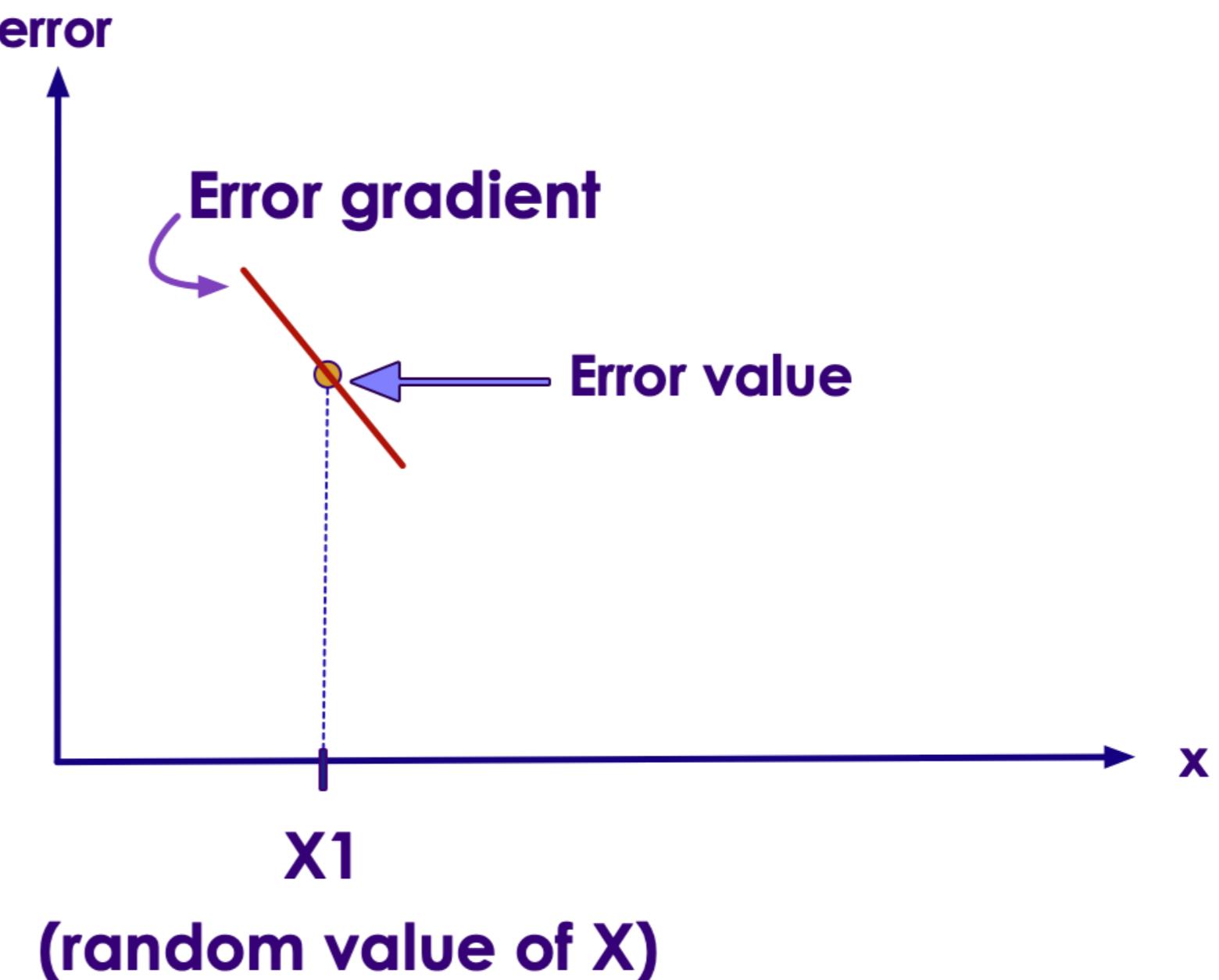
Gradient Descent Process

- Challenge is find the optimal value of X without knowing the graph!



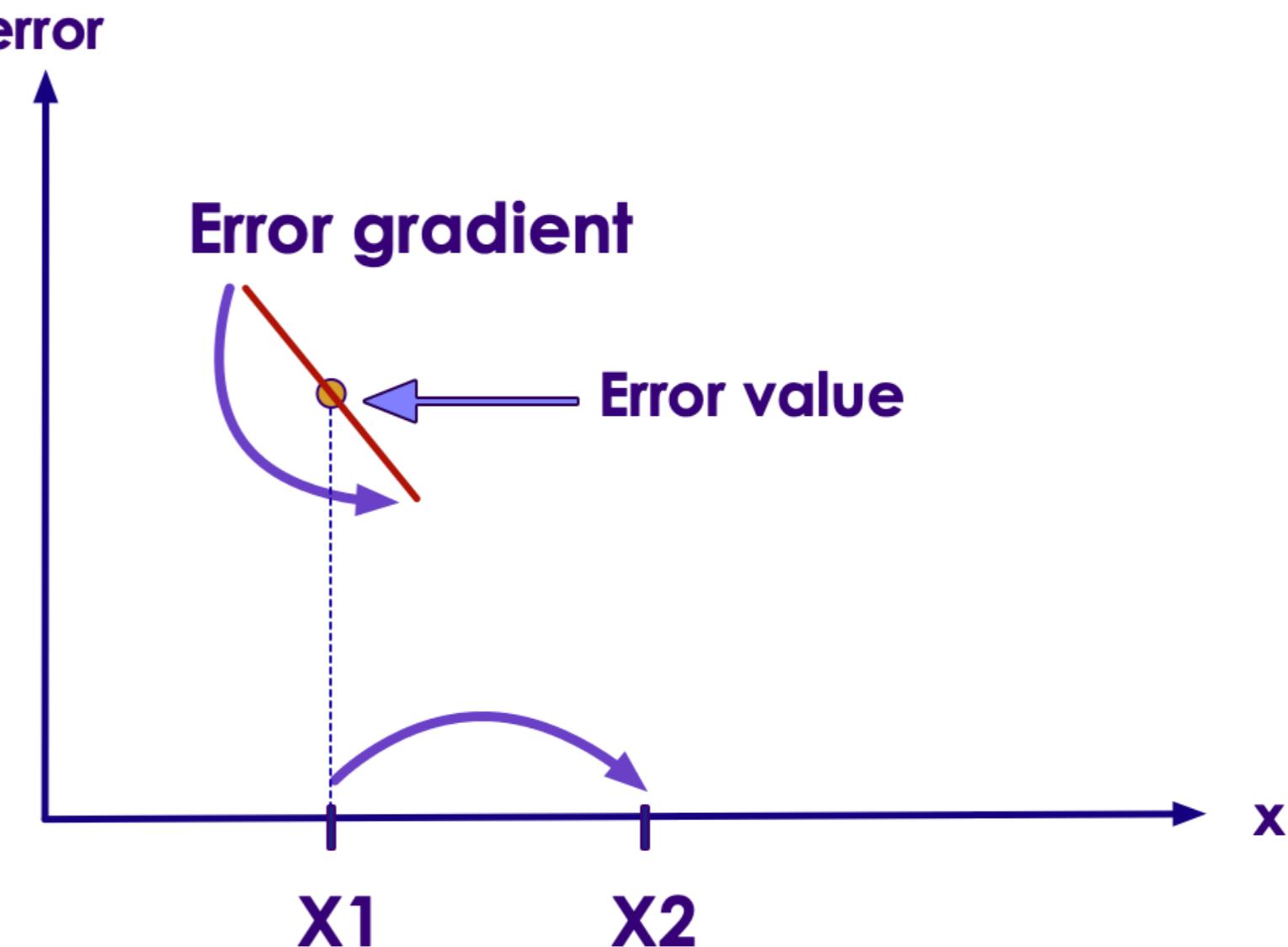
Gradient Descent Process - Step 1

- We are going to start at some random value of X (say x_1)
- We can calculate the error for this x_1
 $\text{error} = f(x_1)$
- We can also measure the error gradient (which way the error function is sloping) for x_1
 - This uses *derivative* functions



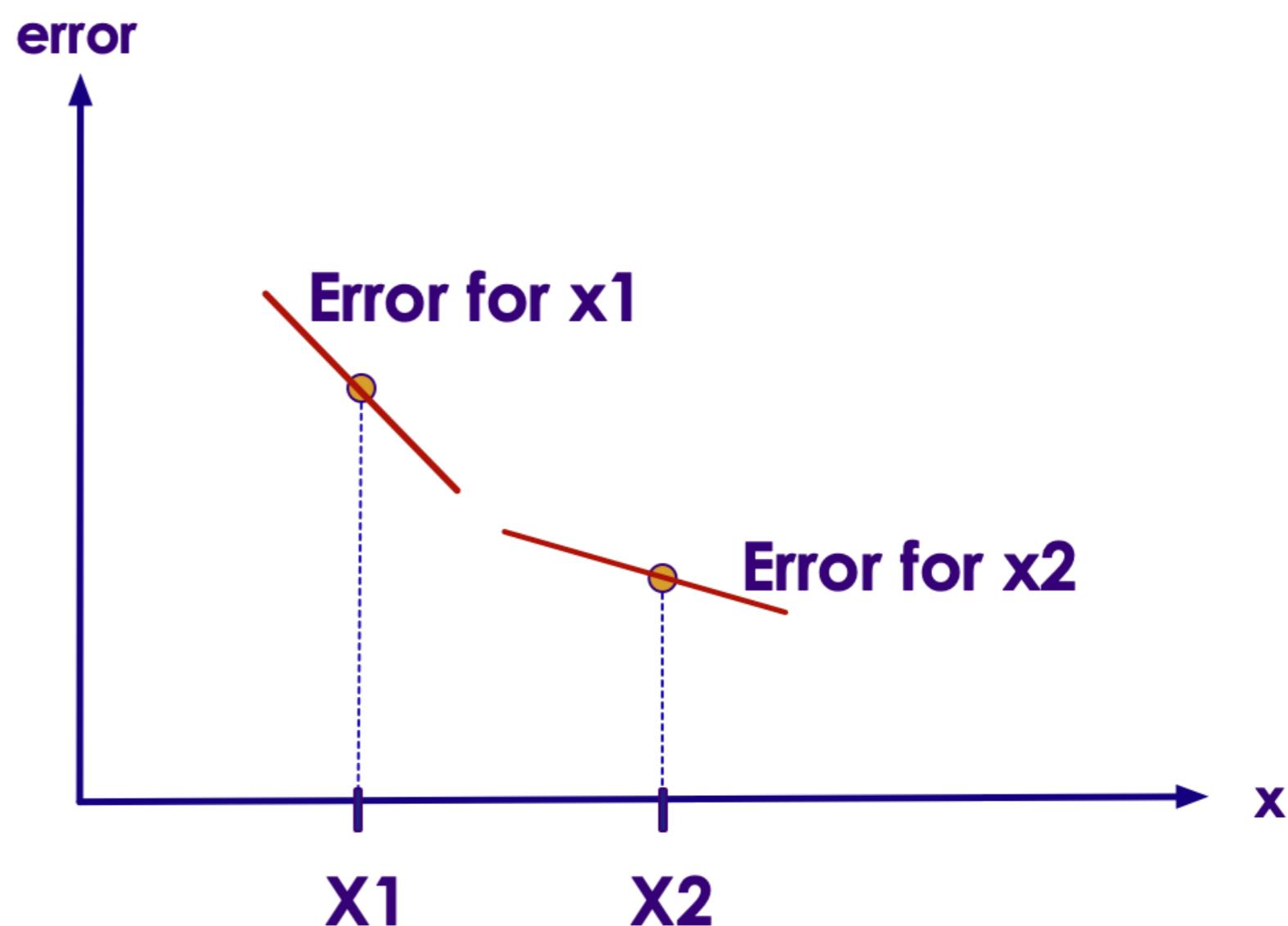
Gradient Descent Process - Step 2

- Using the error derivative, we can see the error function sloping to the right
- That tells us we should be 'moving right' in the graph
- So we calculate x_2 (mostly randomly)
- And repeat the process



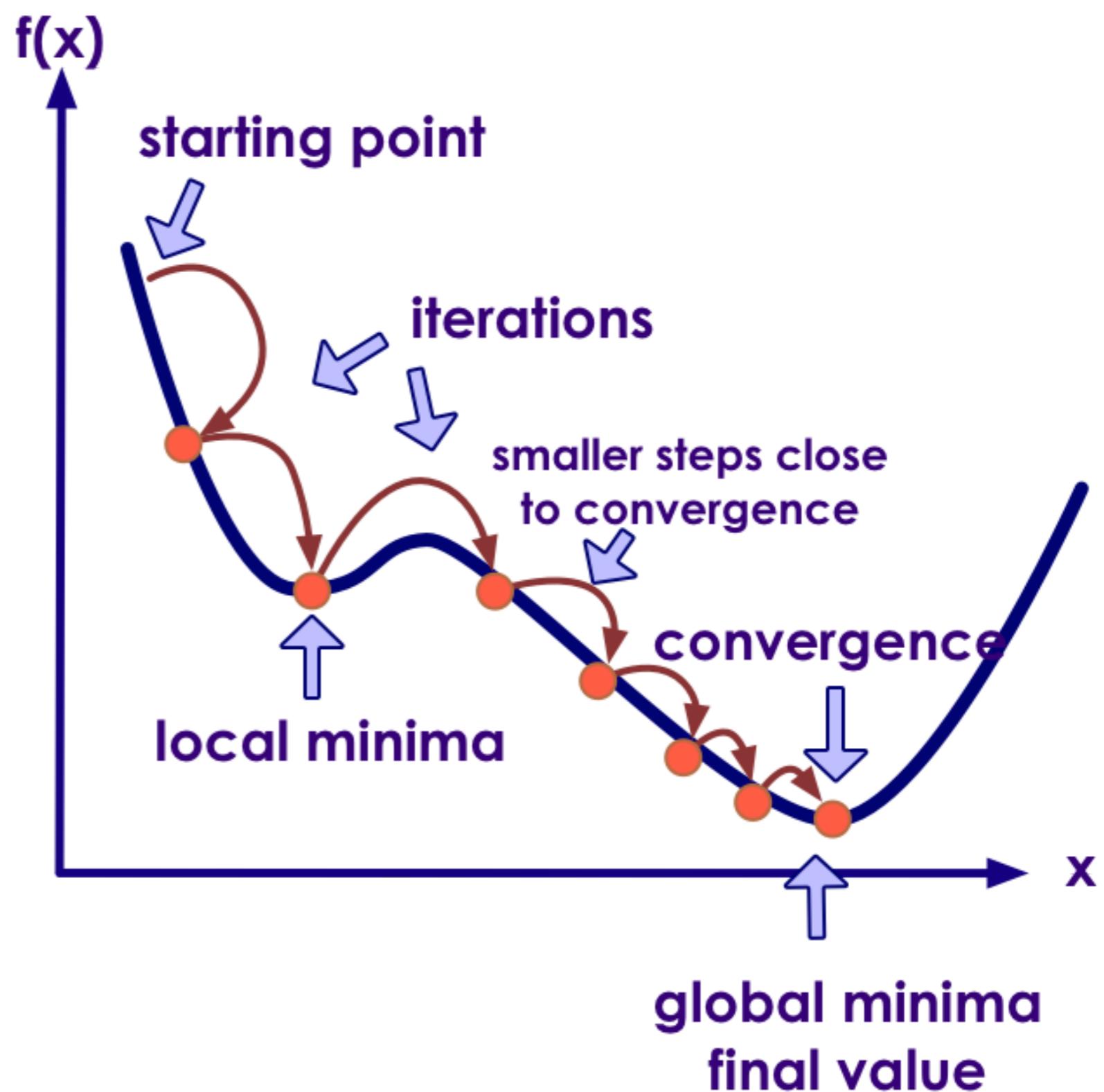
Gradient Descent Process - Step 3

- Now we calculate the error for x_2
 $\text{error} = f(x_2)$
- Measure the error gradient at $f(x_2)$
- And calculate the next X (x_3)
- And repeat the process



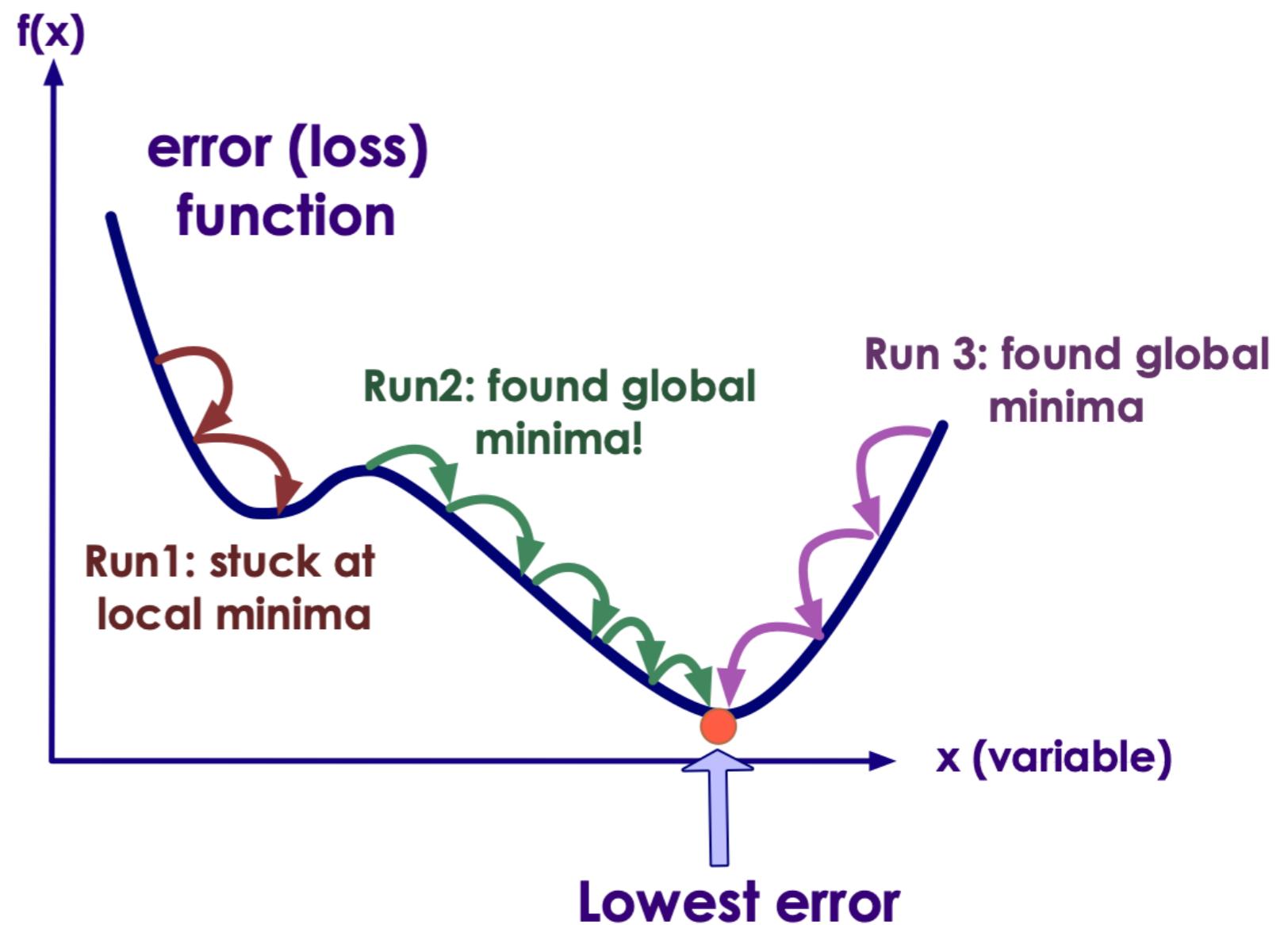
Gradient Descent

- This is basically what a Gradient Descent algorithm does
- Start at a random point, and make 'jumps' towards the minimum
- As it gets closer to convergence the 'steps' gets smaller
 - so we don't overshoot and miss the bottom



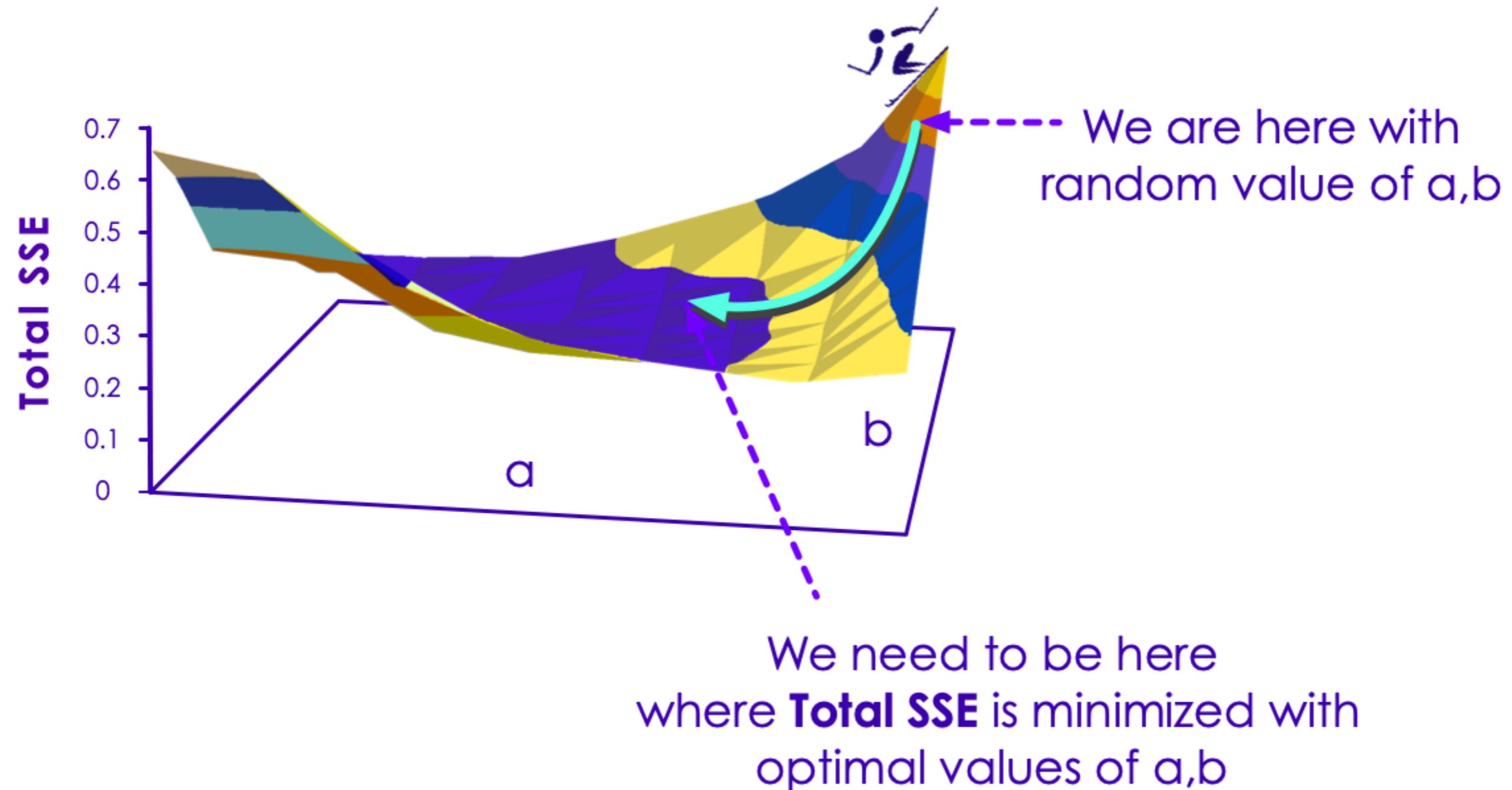
Avoiding Getting Trapped in Local Minima

- Sometimes the descent algorithm will get stuck on local minima
- A practical solution for this problem is to run the descent algorithm multiple times, starting at different random points
- And the algorithm will eventually find the global minimum

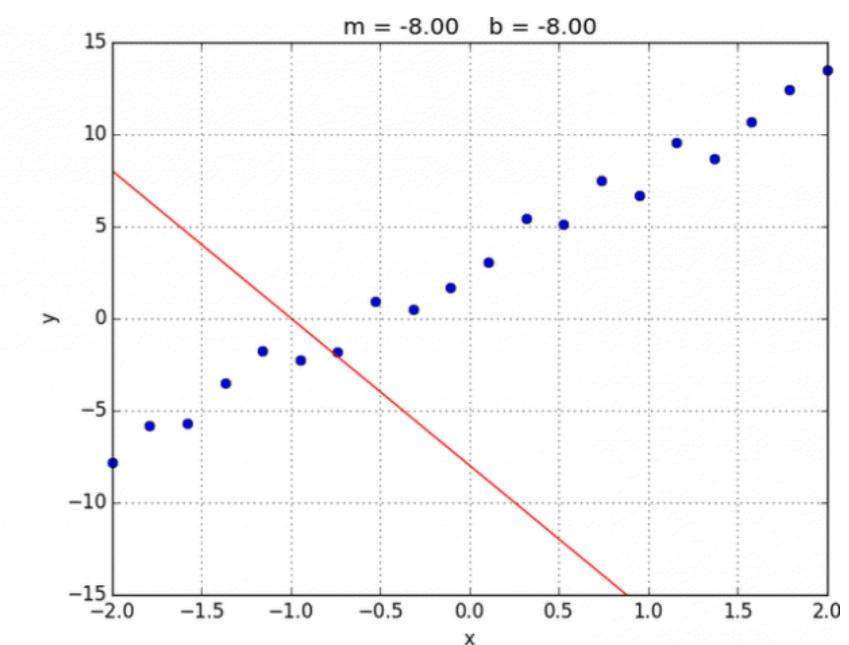
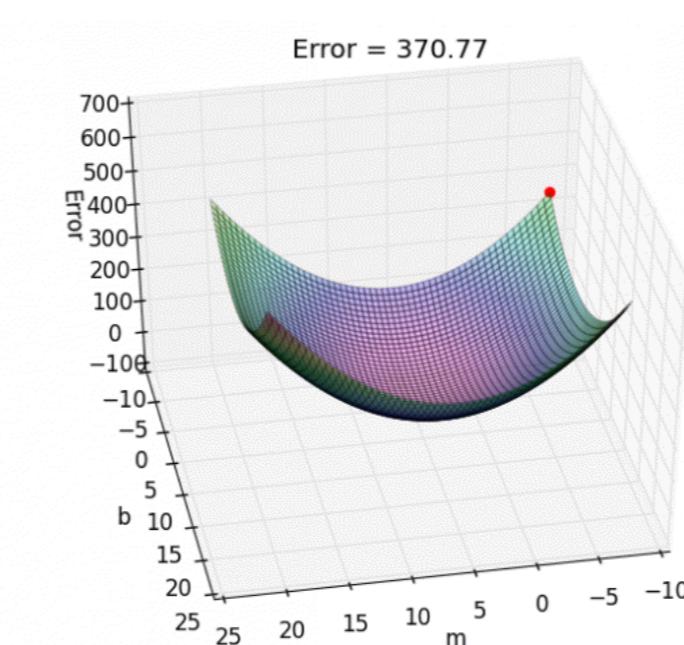
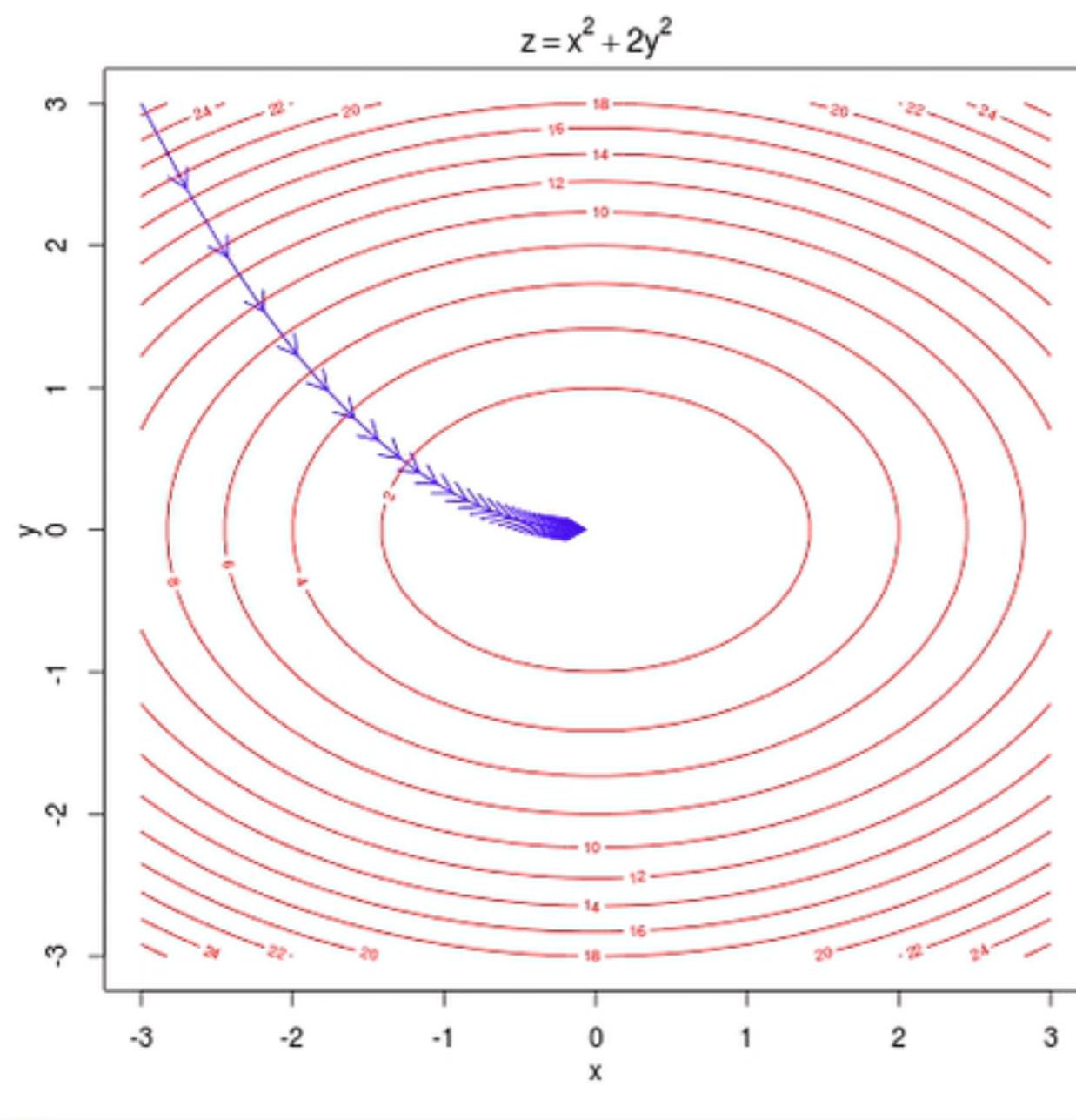


Gradient Descent Algorithm

- Another example in 2D data



Gradient Descent Demo



- Animation 1
- Animation 2

Variations of Gradient Descent Algorithms

▪ Batch Gradient Descent

- The cost is calculated for a machine learning algorithm over the entire training dataset for each iteration of the gradient
- One iteration of the algorithm is called **one batch** and this form of gradient descent is referred to as batch gradient descent

Variations of Gradient Descent Algorithms

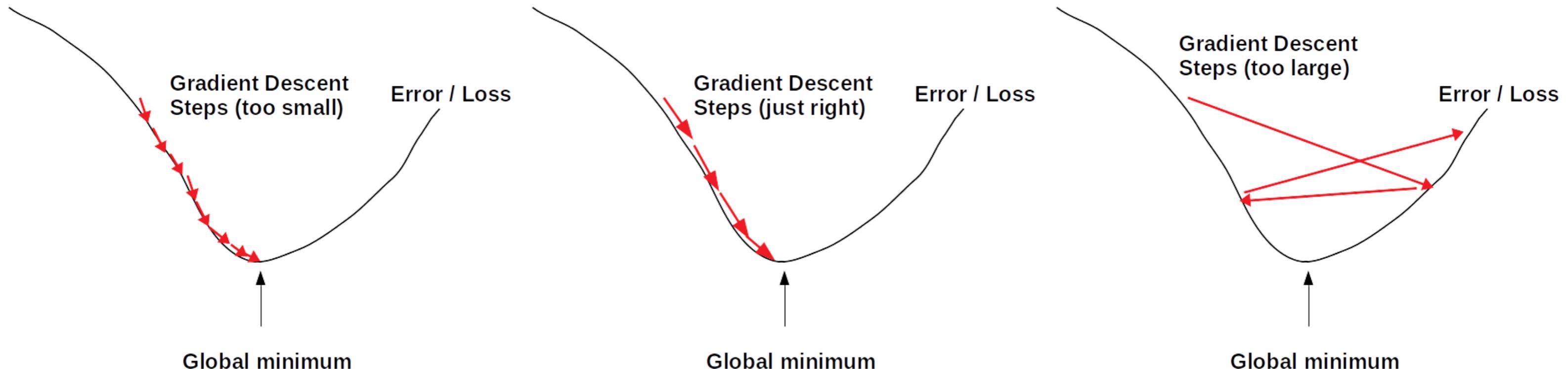
▪ Stochastic Gradient Descent (SGD)

- Classic Gradient Descent can be slow on large datasets (each iteration requires calculation over millions of data points)
- SGD updates coefficients for each training instance, rather than at the end of the batch of instances
- Also randomizes training set to
 - reduce coefficients jumping all over the place
 - And to avoid 'getting stuck' at local minima
- Very effective for large datasets, requires very few passes (usually 10-20) to converge

End: Gradient Descent

Adaptive Learning

- As we saw before, learning rate affects the convergence of SGD
 - Too small, might take too many steps and take long to converge
 - Too large, might not converge at all
- We figure out the optimal learning rate by trial-and-error runs (e.g. hyper parameter tuning)
- The latest optimizers, such as **Adam** and **RMSProp**, can adjust the learning rate automatically; called **adaptive optimizers**

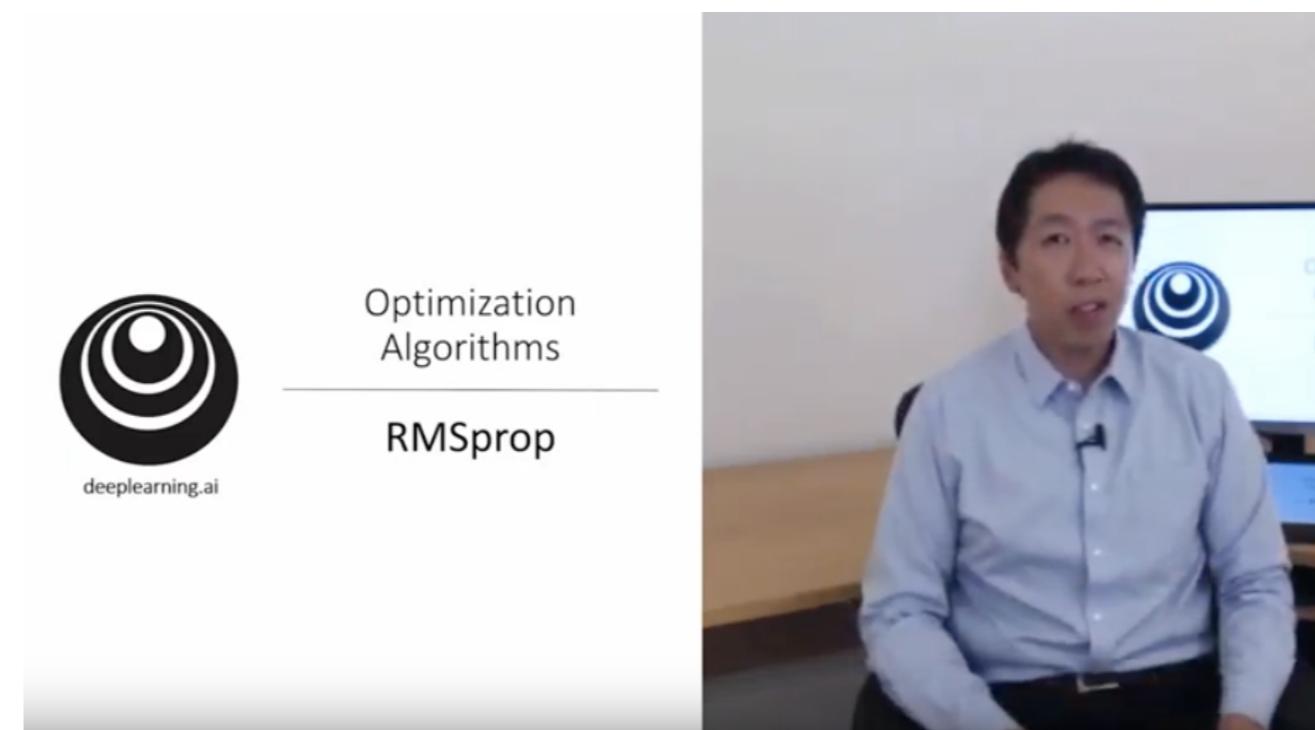
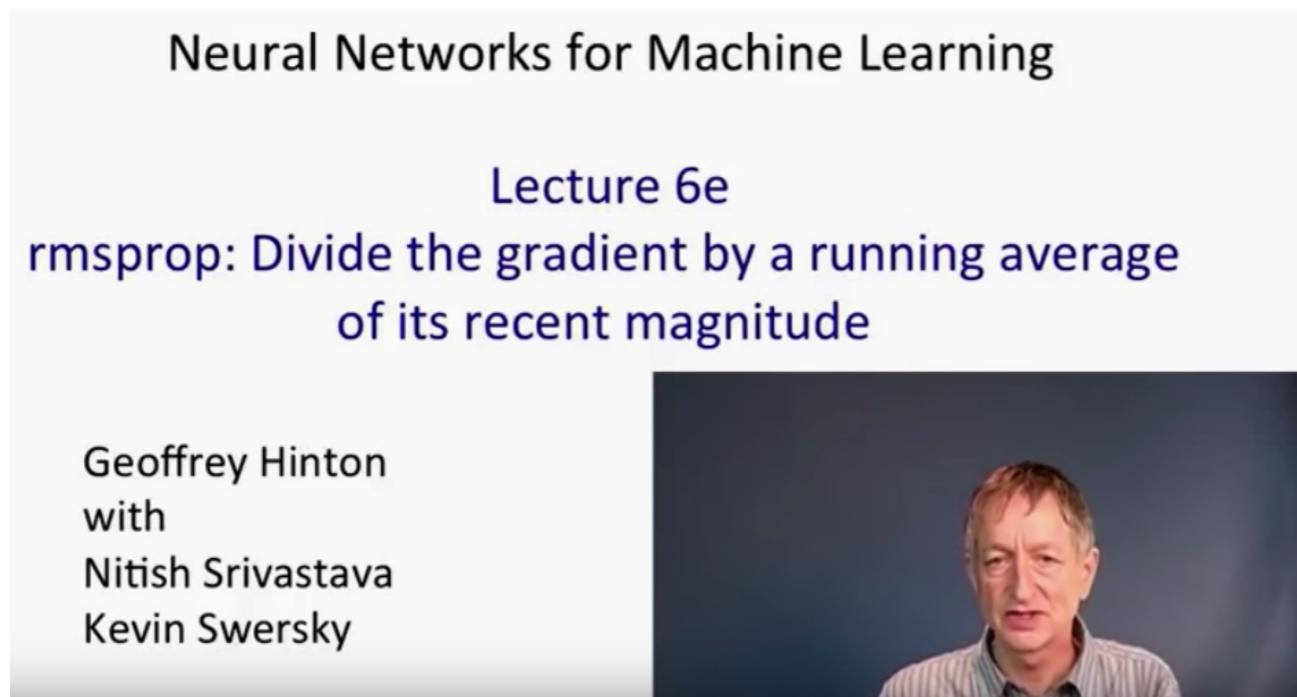


RMS Prop

- Developed by Professor Geoffrey Hinton in his neural nets class
- RMSProp uses **exponential decay** to accumulate only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training)
- Properties
 - Outperforms Adagrad most of the times
 - Was the default choice until 'Adam Optimizer' was devised
- See Appendix for more details and math behind it

RMS Prop Reference

- Lecture by Geoffrey Hinton
- Lecture by Andrew Ng
- References:
 - Class notes for 'lecture 6'
 - A Look at Gradient Descent and RMSprop Optimizers



Using RMSProp

▪ Tensorflow v2

```
from tf.keras.optimizers import RMSprop

# We can use the default values
model.compile(optimizer='rmsprop', loss='...')

# or we can customize
opt = RMSprop(learning_rate=0.1) # <-- initialize the class and provide arguments

# model = ... build model ...

model.compile(optimizer=opt, loss='...')
```

Adam Optimizer

- Adam (Adaptive Moment Estimation) Optimizer ([paper](#)) combines the ideas of Momentum optimization and RMSProp
- Features
 - Currently, the go-to optimizer
 - Since Adam is adaptive, there is very little tuning.
Start with `learning_rate = 0.001`
- References:
 - Paper: '[ADAM: A Method for Stochastic Optimization](#)'

Using Adam Optimizer

▪ Tensorflow v2

```
from tf.keras.optimizers import Adam

# We can use the default values
model.compile(optimizer='adam', loss='...')

# or we can customize
opt = Adam(learning_rate=0.1) # <-- initialize the class and provide arguments

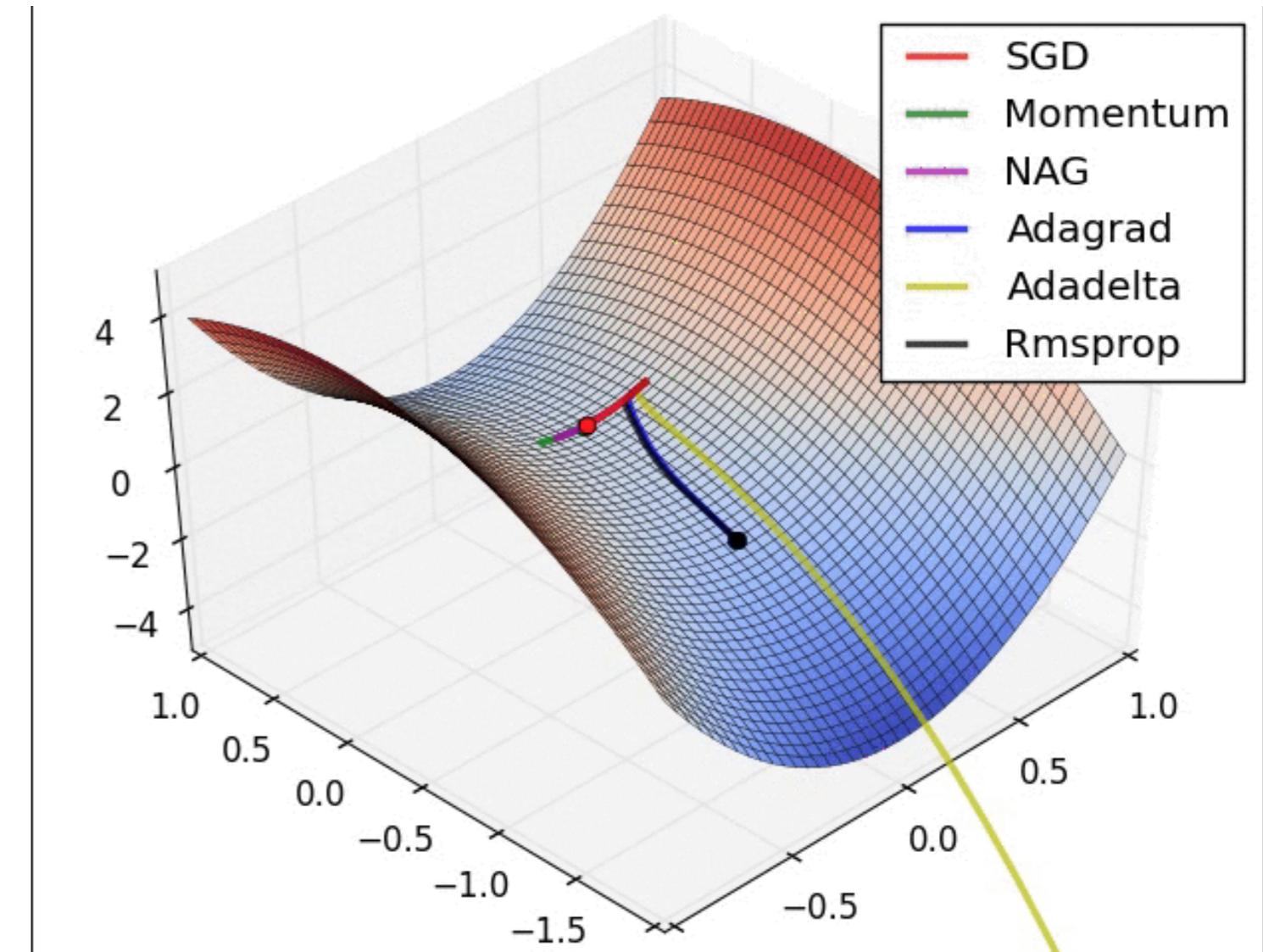
# model = ... build model ...

model.compile(optimizer=opt, loss='...')
```

Comparing Optimizers - Long Valley

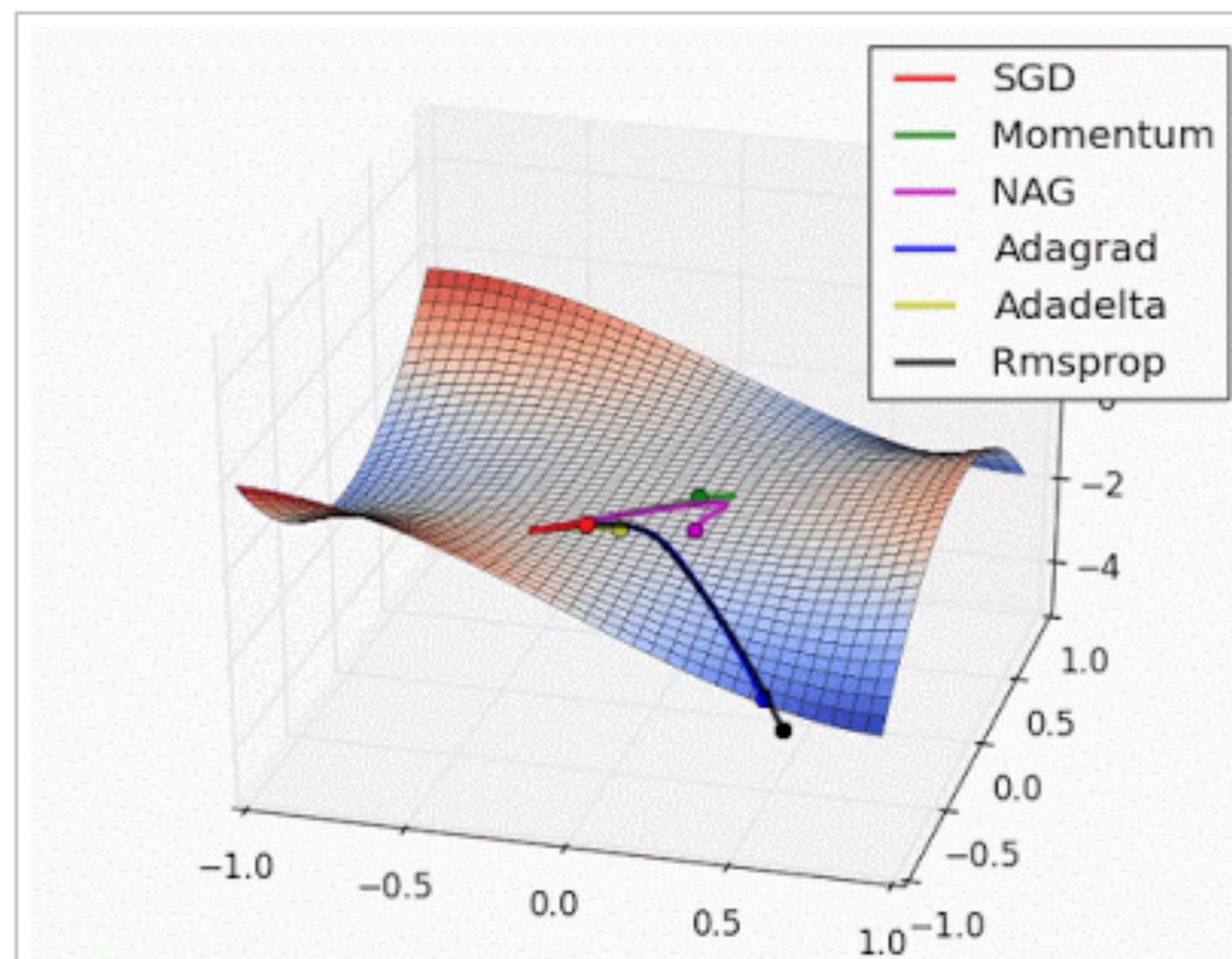
- "Algos without scaling based on gradient information really struggle to break symmetry here - SGD gets no where and Nesterov Accelerated Gradient / Momentum exhibits oscillations until they build up velocity in the optimization direction. Algos that scale step size based on the gradient quickly break symmetry and begin descending quickly"

- Animation
- Source



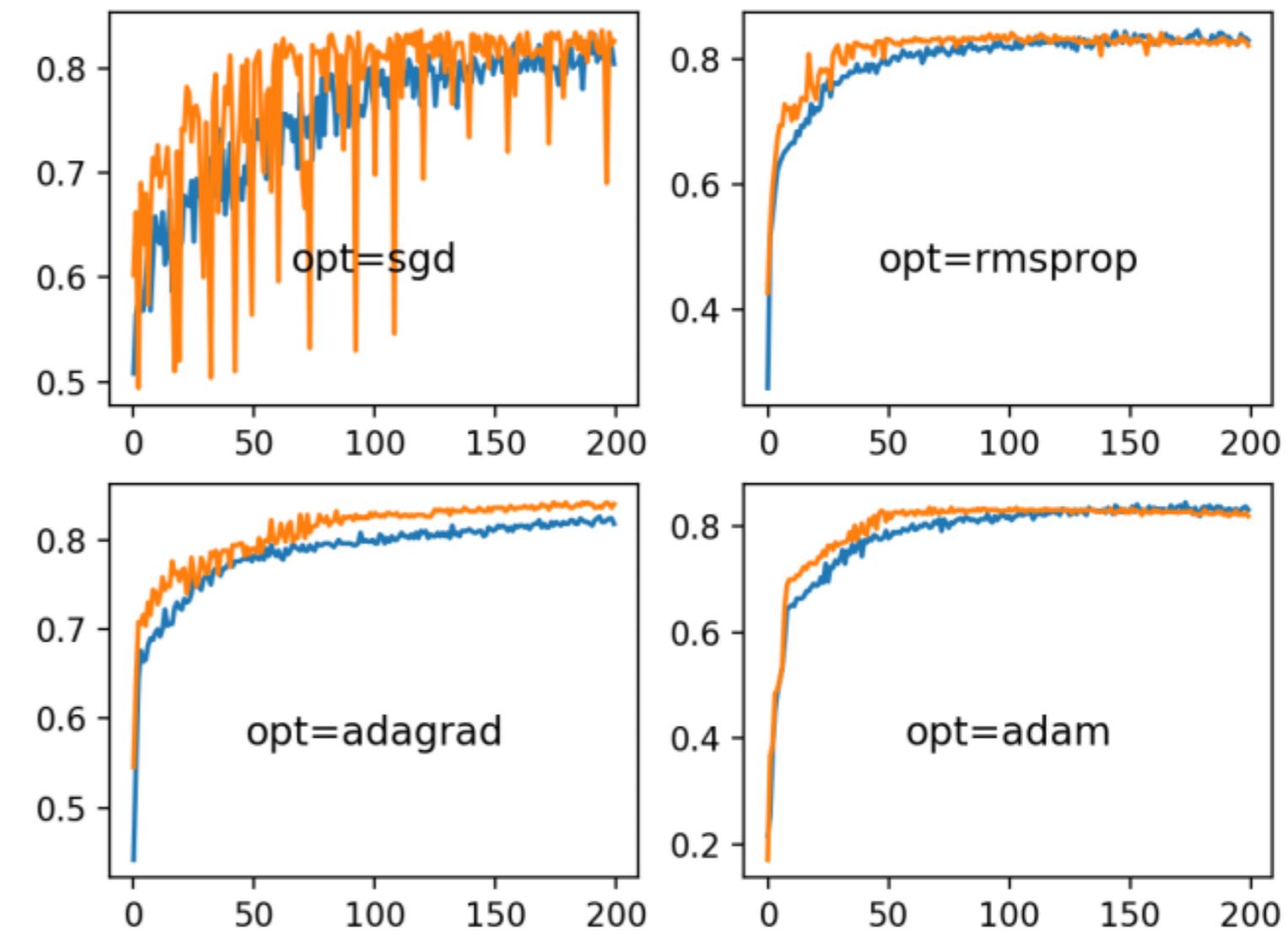
Comparing Optimizers - Saddle Point

- "Behavior around a saddle point. NAG/Momentum again like to explore around, almost taking a different path. Adadelta/Adagrad/RMSProp proceed like accelerated SGD."
- Animation
- Source



Optimizers - Takeaway

- Here we see the progress of our algorithm accuracy (climbing towards 1.0 or 100%)
- SGD's progress is 'bumpy'; While rmsprop and adam are progressing smoothly
- **RMSProp** and **Adam** are the 'go to' optimizers now
- These are **adaptive** algorithms, that adjust learning rate as training progresses.
- No need to fiddle with learning rates!
- Reference: Machine Learning Mastery - Learning rate



Review Questions (Part 1)

- Q: Can you name 3 activation functions and when they are used?
- Q: What are the advantages of the ReLU activation over Sigmoid?
- Q: How many neurons do you need in the output layer to classify emails into spam/ham?
 - how about for classifying digits 0 to 9?
- Q: In which cases you would use the following activation functions: ReLU, tanh, sigmoid, and softmax?



End of Part 1!

- **Instructor:** Pause here. And move onto the next module
- Cover **Part-2** as need basis

Part 2: Advanced Concepts

Cover as necessary

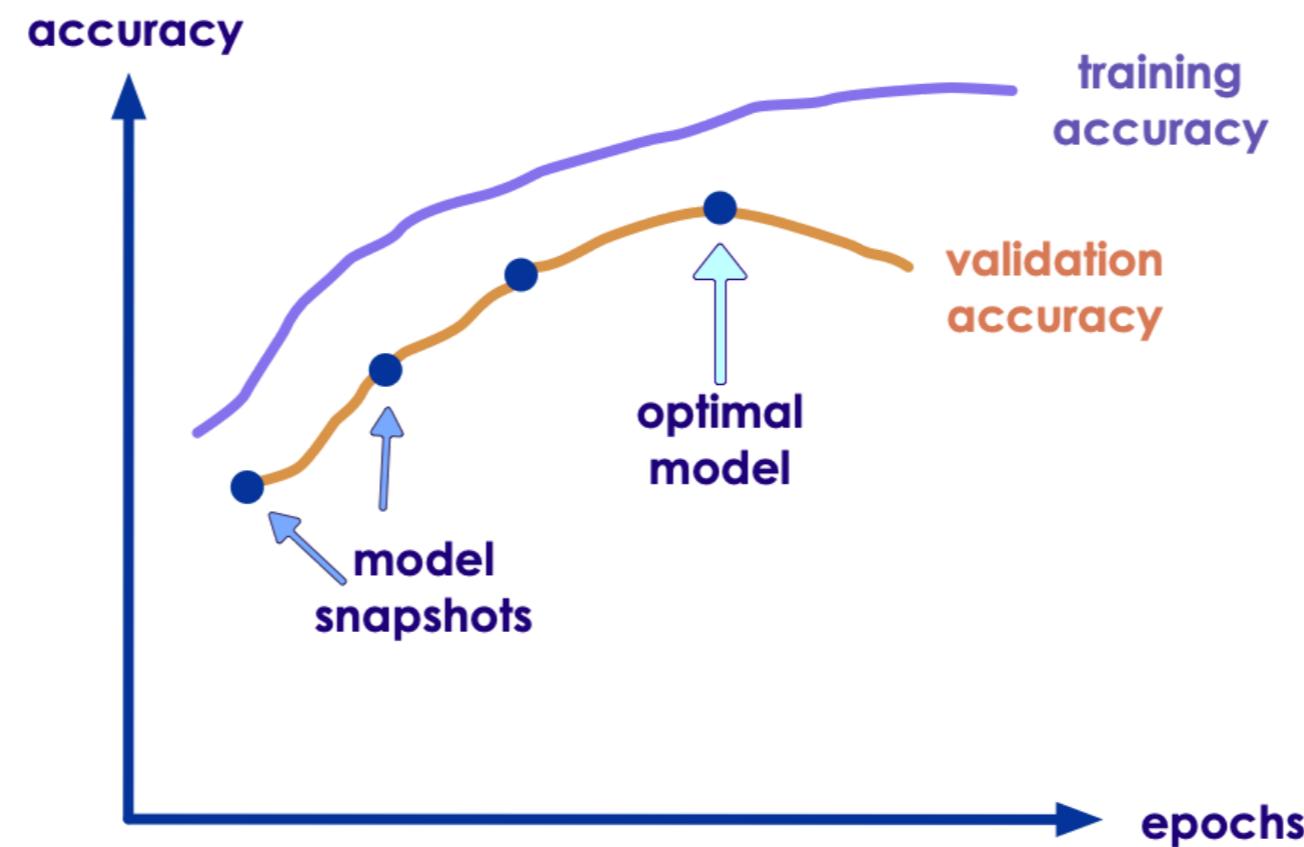
Avoiding Overfitting

Managing Overfitting

- Neural networks have tens of thousands / millions of parameters
- With these many parameters, the networks are very flexible, they can fit very complex data sets
- Also means the network can overfit training data
- How to manage overfitting?
 - Regularization
 - Early stopping
 - Dropout
 - Max-norm regularization
 - Data augmentation

Early Stopping

- Don't train too long
- Interrupt training when its performance on the validation set starts dropping.
- How to do it?
 - Measure validation accuracy every few steps (say 20)
 - If it scores higher than previous snapshot, save the current model snapshot as 'winner'



Regularization

- In conventional ML we often use regularization to control overfitting.
- L1 and L2 are common mechanisms for regularization
- In DL, Regularization is probably not enough
 - Even penalized, certain features will eventually dominate.
 - DL will always overfit, even with L1/L2.
- Is there something else we can do?

Dropout

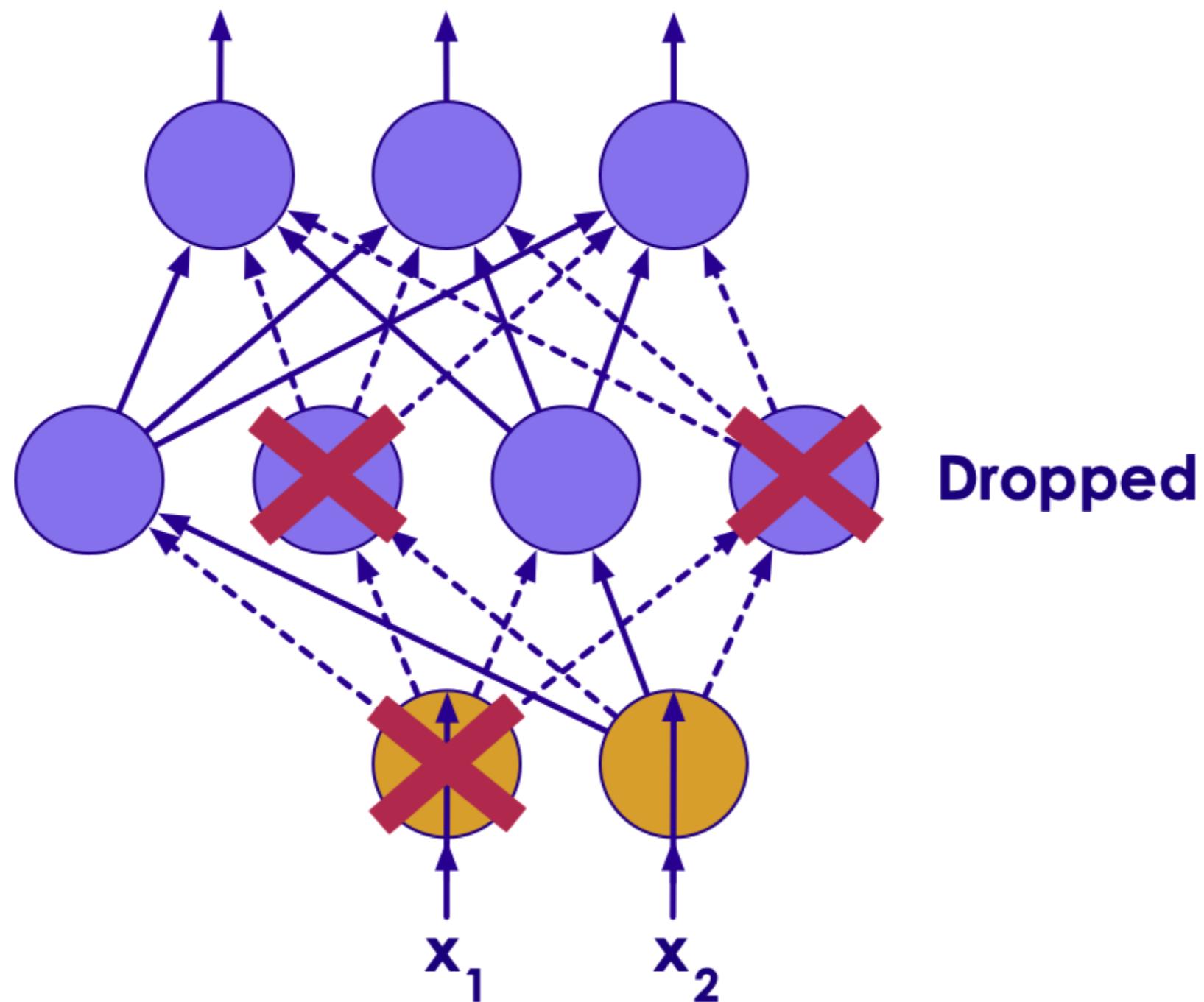
- **Dropout** is the most popular regularization technique for deep neural networks
- It was proposed by Geoffrey Hinton in 2012 ([paper1](#), [paper2](#))
- By omitting half the neurons' signal (50% dropout rate), they were able to increase an accuracy of state of the art model from 95% to 97.5%
 - This may not seem like a lot, but the error rate improved from 5% to 2.5% (that is 50% reduction in error!)

Dropout

- How does it work?

- At every training step, each neuron has a chance (probability) of being 'dropped'.
Meaning, its output ignored during this step
- The neuron can become active during the next step
- Neurons in input layer and hidden layer can be dropped
- Output neurons are not dropped
- The parameter (p) is called 'dropout rate' - varies from 0 to 1.0.
Typically set to 0.5 (50%)

Dropout



Dropout

- It is really surprising, that dropout method works in real life.
Imagine this scenario
- Workers of this 'unicorn' company
 - Every morning they toss a coin
 - 'Heads' they come to work, 'tails' they don't
 - So that means 50% of workers don't show up at any day
 - 'Dropout' method says, this makes the 'company' as a whole, perform better :-)
- Increase dropout rate, if you notice the model is overfitting.
Decrease it if it is underfitting
- Dropout slows down the model convergence, but the model you get is much better at the end

Max-Norm Regularization

- Max-Norm regularization is very popular for neural networks
- for each neuron, it constrains the weights w of the incoming connections such that $\|w\|_2 \leq r$
 - where r is the max-norm hyperparameter and $\|\cdot\|_2$ is the L2 norm

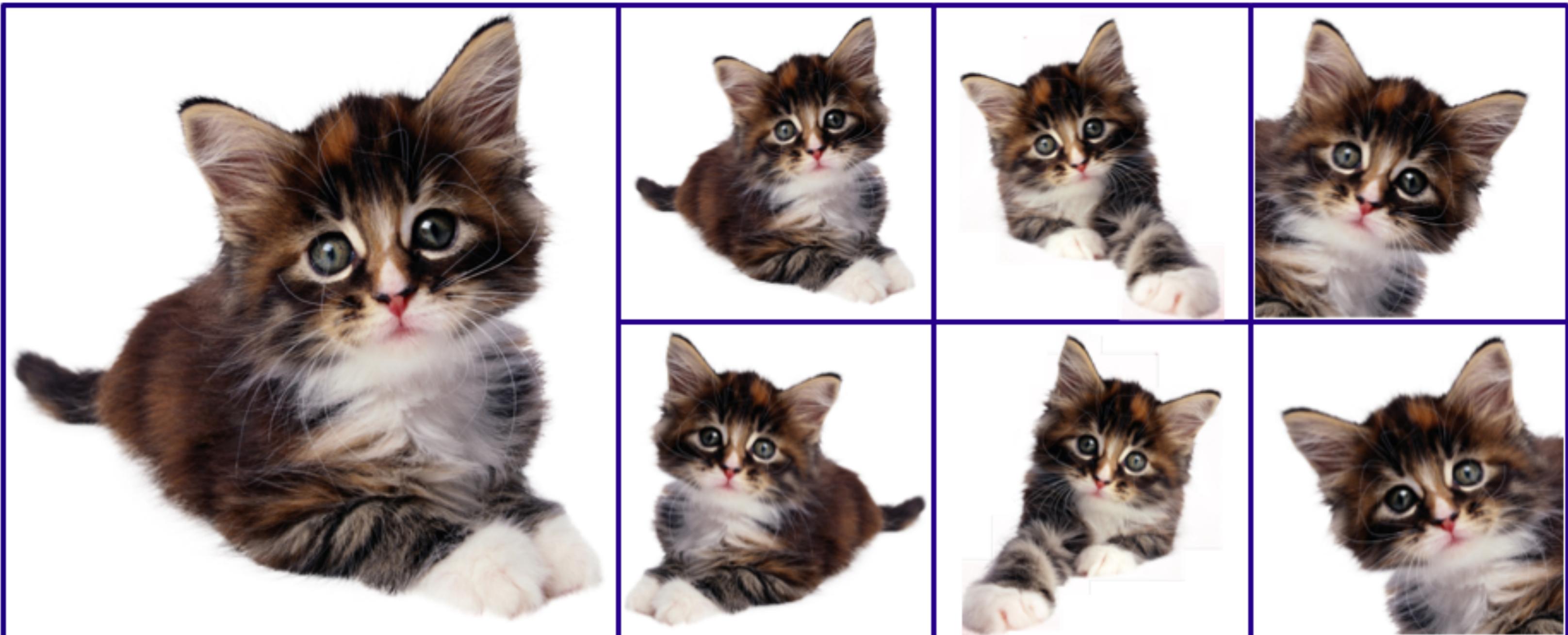
$$\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\|\mathbf{w}\|_2}$$

- Max-norm regularization can also help reduce the vanishing/exploding gradients

Data Augmentation

- **Data augmentation** creates new training instances from existing ones
 - this artificially boosts training set size
- This technique is mostly used in image training
- Common techniques involve:
 - adjusting brightness
 - introducing some noise
 - rotating images slightly clockwise / anti-clockwise (10 to 20 degrees)
 - cropping images / moving centers
- See next slide for an example

Data Augmentation Example



Stochastic Pooling

- Normally, we apply MAX function for pooling
 - sometimes AVG (mean) pooling, but less often these days
- Problem: Selecting MAX tends to overfit!
- What if we do something else?
- "Stochastic" pooling means we randomly choose another one.
- Conform to normal distribution.
- Similar to dropout in that we randomly ignore a preferred weight.

Neural Network Modern Techniques (Advanced / Optional)

Neural Network Modern Techniques

These are discussed in the following sections/slides

- Using ReLU activation functions (we just saw this)
- Xavier and He Initialization
- Batch Normalization
- Gradient Clipping

Xavier and He Initialization

- **Problem**
- We want signals to flow properly in both directions : forward and backwards
 - no dying out or not exploding
- **Solution**
- Make the variance of the outputs of each layer to be equal to the variance of its inputs
(see paper for the math details)
- Connection weights are initialized randomly (see next slide)
- Doing this **Xavier initialization strategy** really sped up learning in neural networks and really kick started the research again

Xavier and He Initialization

- For layer with n-inputs and n-outputs
- Normal distribution with mean 0 and standard deviation σ as follows

$$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

- Or Uniform distribution between $-r$ and r with r

$$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

- When number of inputs == number of outputs, we get a simplified equation

$$r = \sqrt{3} / \sqrt{n_{\text{inputs}}} \quad \sigma = 1 / \sqrt{n_{\text{inputs}}}$$

Xe Initialization Parameters

Activation function	Uniform distribution $[-r, r]$	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Batch Normalization

- So far we have seen **Xe initialization** and **ReLU variants**
- These can help avoid vanishing/exploding gradient problems at the start of training
- however during later phases of training, it may occur
 - Sergey Ioffe and Christian Szegedy proposed a technique called Batch Normalization (BN) in this 2015 paper(<https://arxiv.org/pdf/1502.03167v3.pdf>)
 - This approach adds another operation before the activation function of each layer
- it normalizes input to the layer and zero centers them

Batch Normalization Performance

- Significantly reduced vanishing gradient problems
- They could even try saturating functions like sigmoid and tanh
- Network was less sensitive to initial weight initialization
- Learning time can be reduced by using larger learning rates (converges faster)
- In ImageNet classification it gave 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters
- Also acts as a regularizer reducing overfitting
- Downside:
 - Slower performance during predictions / inferences, because it adds extra compute for each layer
 - Even though the same penalty applies during training phase, it comes out ahead, because training converges quicker (in much fewer steps)

Batch Normalization Implementation

- In Tensorflow

```
tf.layers.batch_normalization
```

- In Keras

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99,  
    epsilon=0.001, center=True, scale=True,  
    beta_initializer='zeros', gamma_initializer='ones',  
    moving_mean_initializer='zeros', moving_variance_initializer='ones',  
    beta_regularizer=None, gamma_regularizer=None,  
    beta_constraint=None, gamma_constraint=None)
```

Batch Normalization Math (Reference Only)

For reference only, please see the paper for underlying math.

$$1. \quad \mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$2. \quad \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

$$3. \quad \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$4. \quad \mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$$

Batch Normalization Math (Reference Only)

For reference only, please see the paper for underlying math.

- μ_B is the empirical mean, evaluated over the whole mini-batch B .
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.
- (i) is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-5}). This is called a smoothing term.
- $z(i)$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

Gradient Clipping

- One way to solve **exploding gradients** during backpropagation is to make sure they don't exceed a certain threshold
 - **gradient clipping**
- See [this paper](#) by Razvan Pascanu, Tomas Mikolov and Yoshua Bengio for details

Final Words

These default values should get you started, and should work well in most scenarios

Parameter	Value
Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam / Nesterov Accelerated Gradient
Learning rate schedule	None

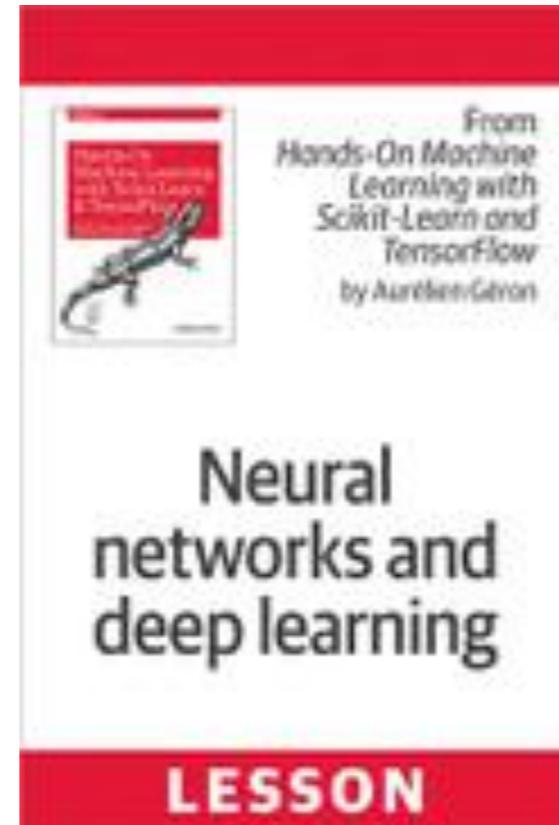
Review Questions - Part 2

- Q: What are the techniques to prevent overfitting?
- Q: Explain how Dropoff works



Resources

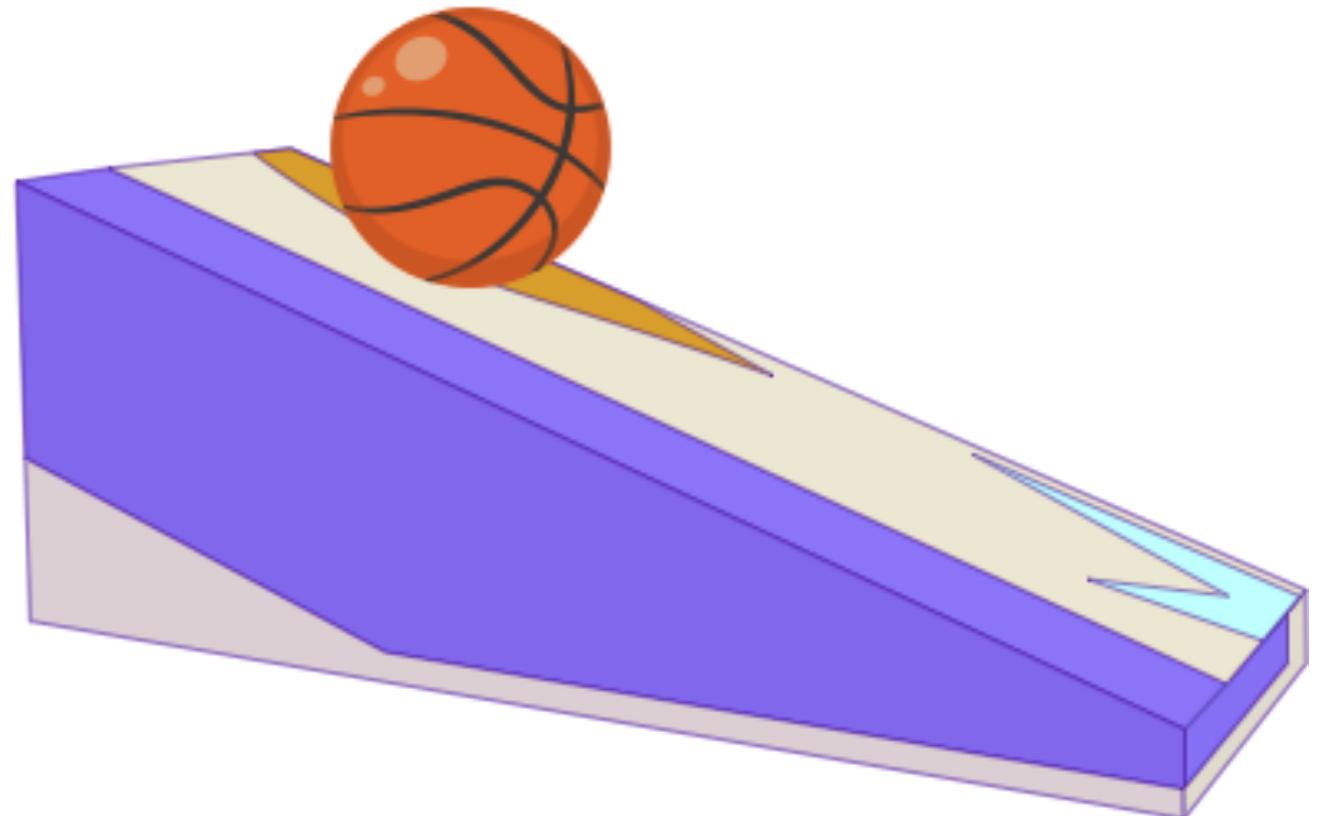
- Neural networks and deep learning
by Aurélien Géron (ISBN: 9781492037347)



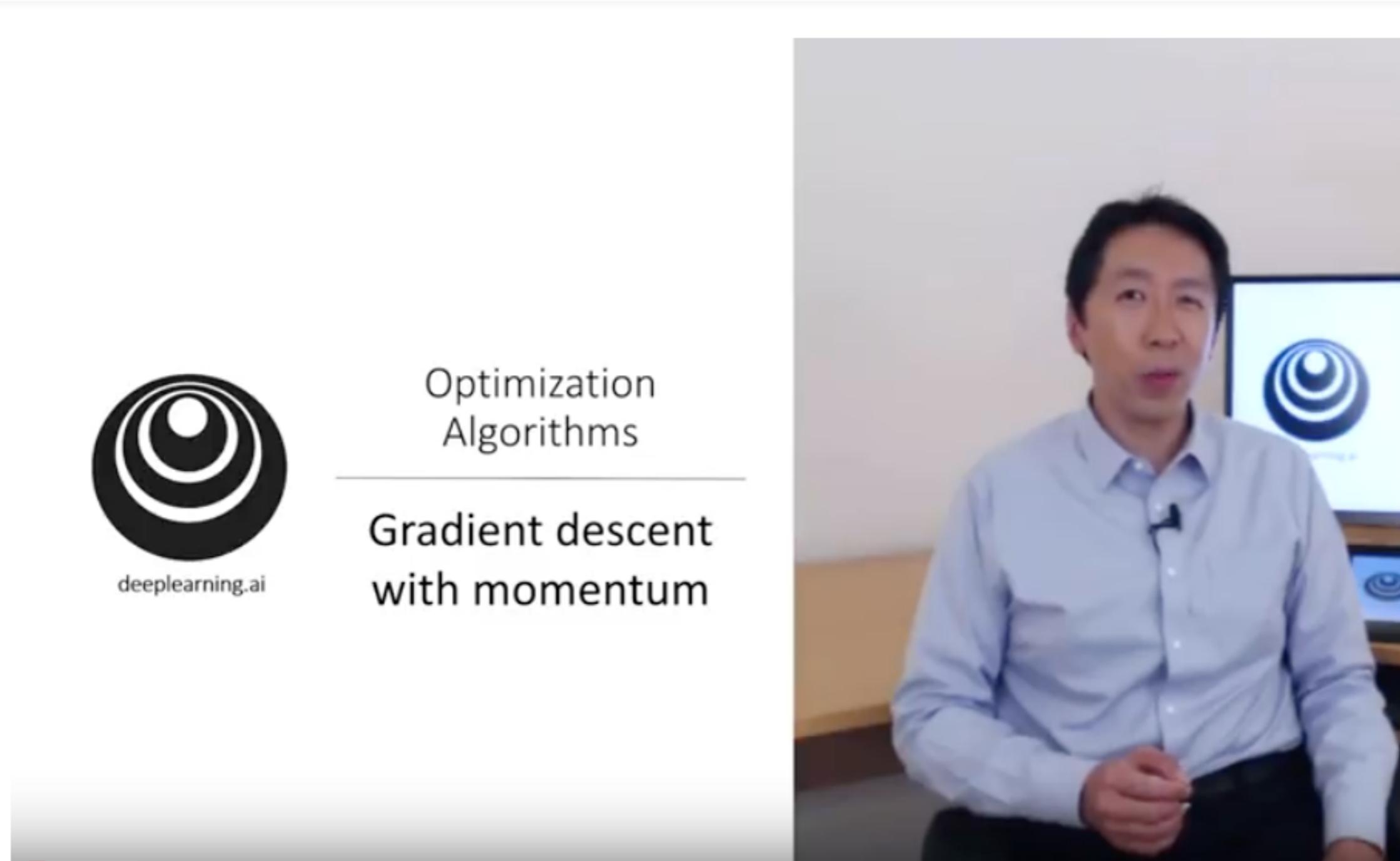
Appendix-Optimizers

Momentum Optimization

- Imagine a ball rolling down a smooth surface; it will start slowly, but keep accelerating and quickly picking up momentum until it reaches terminal velocity
- This is the idea behind **Momentum Optimization** (paper by Boris Polyak, 1964)
- Regular Gradient Descent will get there too, but will take many steps and take longer



Momentum Video Tutorial



- Link

Using Momentum Optimizer

- **Tensorflow v2 (Documentation)**

```
from tf.keras.optimizers import SGD

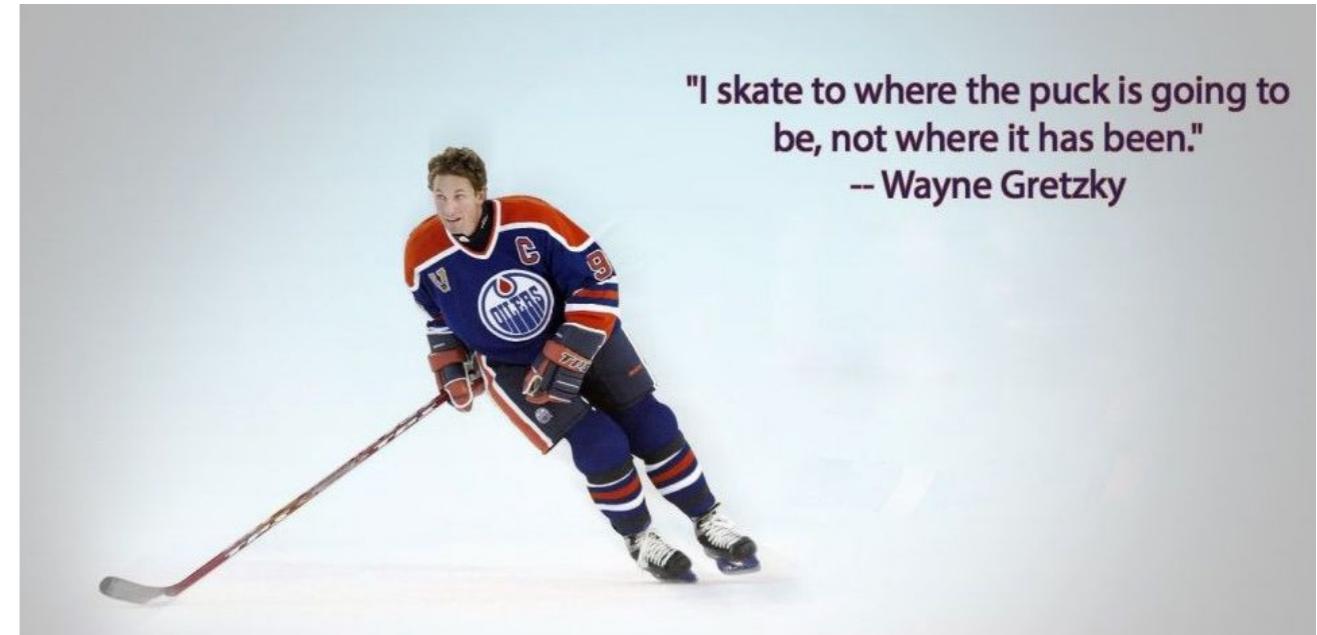
opt = SGD(learning_rate=0.01,
          momentum=0.9) # <-- specify momentum here
# momentum = 0.0 (default value) is plain SGD

# model = ... build model ...

model.compile(optimizer=opt, loss='...')
```

Nesterov Accelerated Gradient

- This is an update to Momentum Descent
- **Nesterov Accelerated Gradient (NAG)** measures the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum
- References:
 - Paper by Yurii Nesterov in 1983
 - Sutskever et al., 2013



Using Nesterov

■ Tensorflow v2

```
from tf.keras.optimizers import SGD

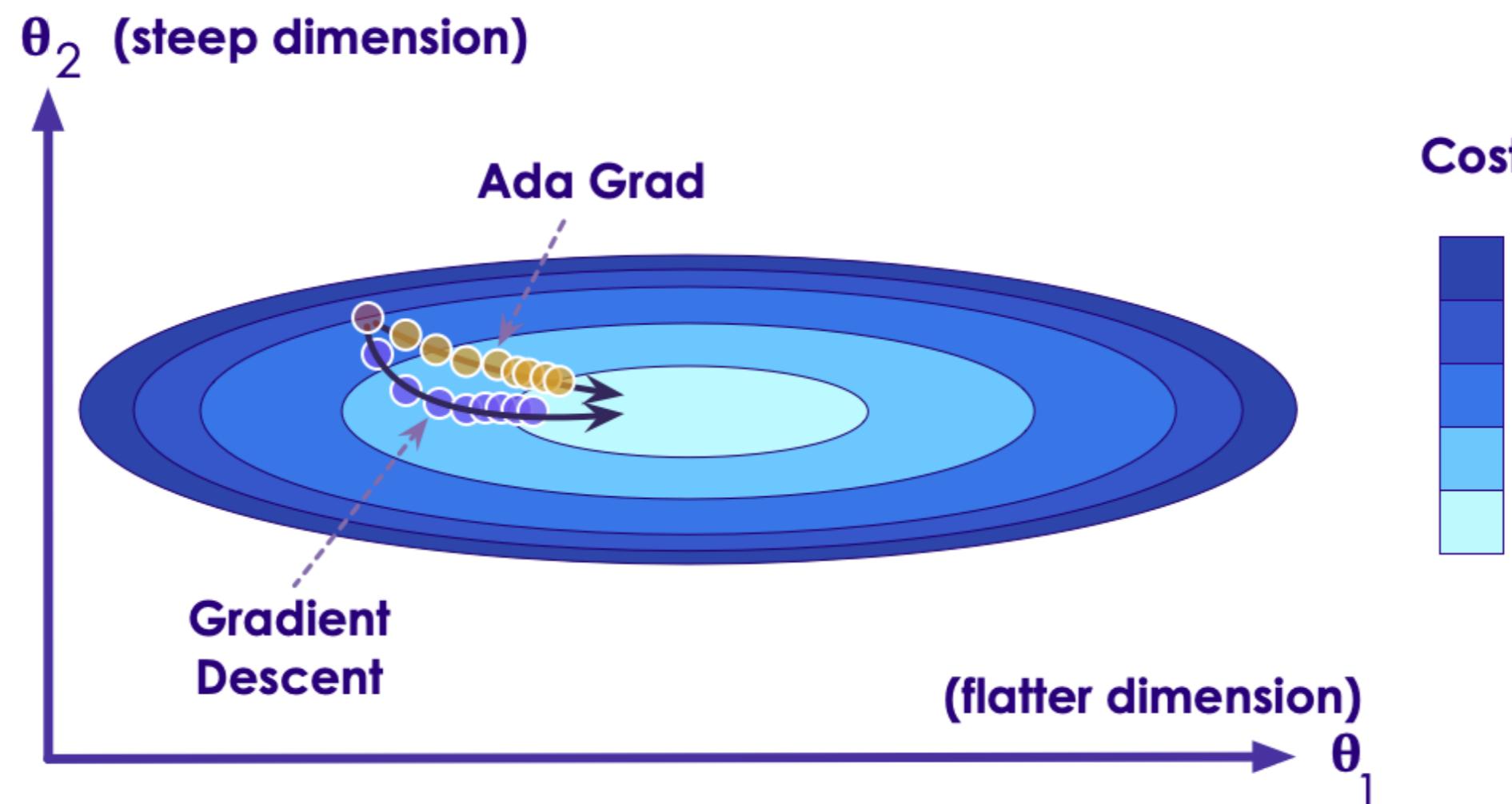
opt = SGD(learning_rate=0.01,
          momentum=0.9,
          nesterov = True) # <-- Apply Nesterov algorithm
# by default nesterov=False

# model = ... build model ...

model.compile (optimizer=opt, loss='...')
```

Adagrad

- In Gradient Descent animation algorithm takes 'smaller steps' when going down 'valleys'
- Adagrad (paper) adjusts the direction and velocity by scaling the direction vector
 - 'points in the right direction (global minimum)' better :-)



Momentum Optimization Theory

- Regular Gradient Descent updates the new weights using learning rate (always constant). if the local gradient is very small, the updates are small too

$$\theta = \theta - \alpha \nabla_{\theta} J(\theta)$$

- Here
 - θ : is current weights
 - α : learning rate
 - $J(\theta)$: cost
 - $\nabla(\theta)$: is derivative

Momentum Optimizer

- Momentum takes into account of what previous gradients were

$$m = \beta m - \alpha \nabla_{\theta} J(\theta)$$

- Calculates the momentum and adds it to the next weight updates

- so it accelerates the updates

- Hyperparameter β , is called the momentum; ranges between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

- Features

- Could be 10x faster than Gradient Descent
 - Also doesn't get trapped in local minima

$$\theta = \theta + m$$

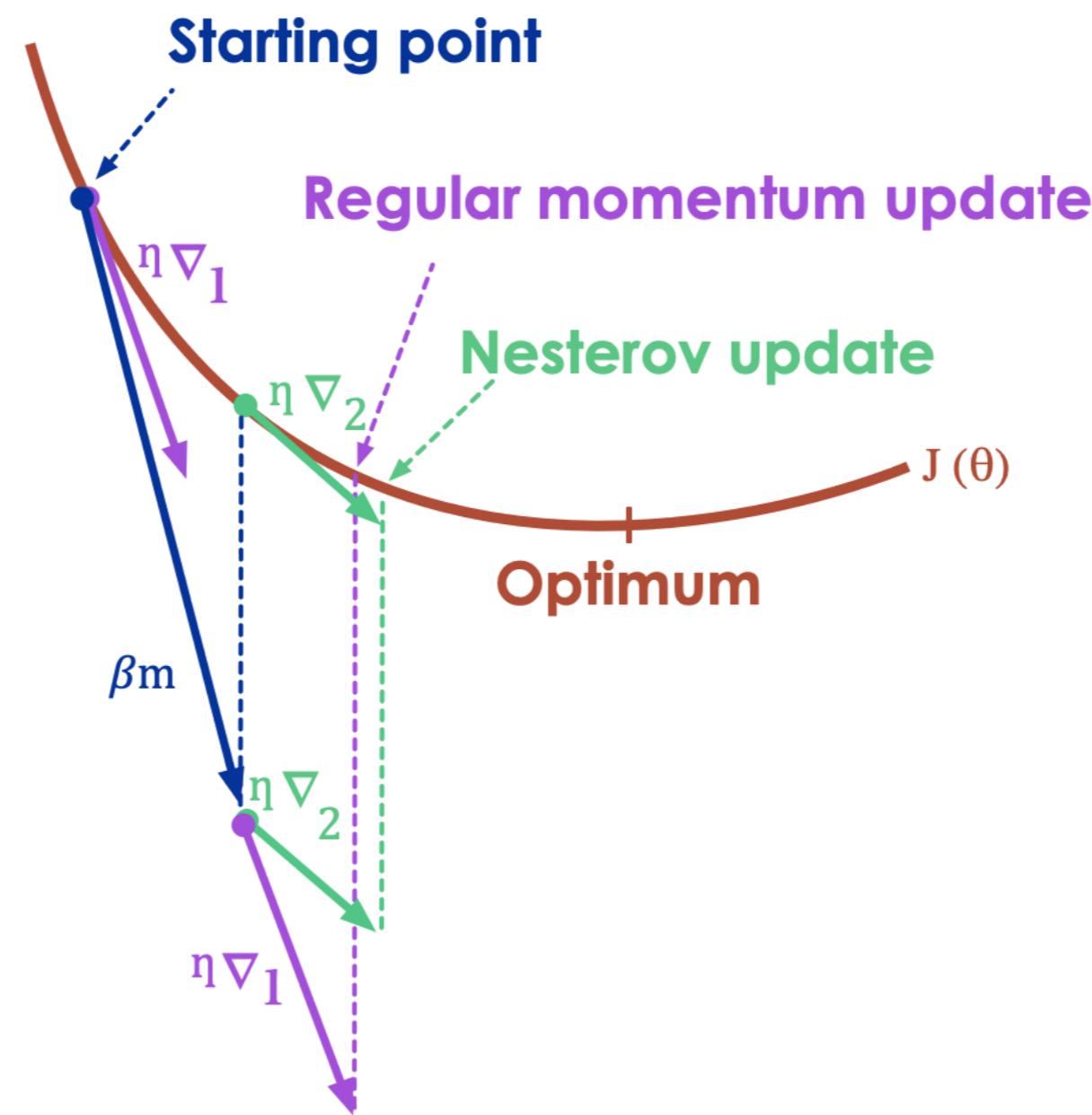
Nesterov Accelerated Gradient

$$m = \beta m - \alpha \nabla_{\theta} J(\theta + \beta m)$$

$$\theta = \theta + m$$

Nesterov Accelerated Momentum

- Here you see Nesterov approach is slightly closer to optimum



RMSProp Math

- Decay rate β is between 0 and 1.0; typically set to 0.9 - that works well in most scenarios

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1 - \beta) \nabla_{\theta}J(\theta) \otimes \nabla_{\theta}J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

Adam Math (Reference Only)

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

Adam Math (Reference Only)

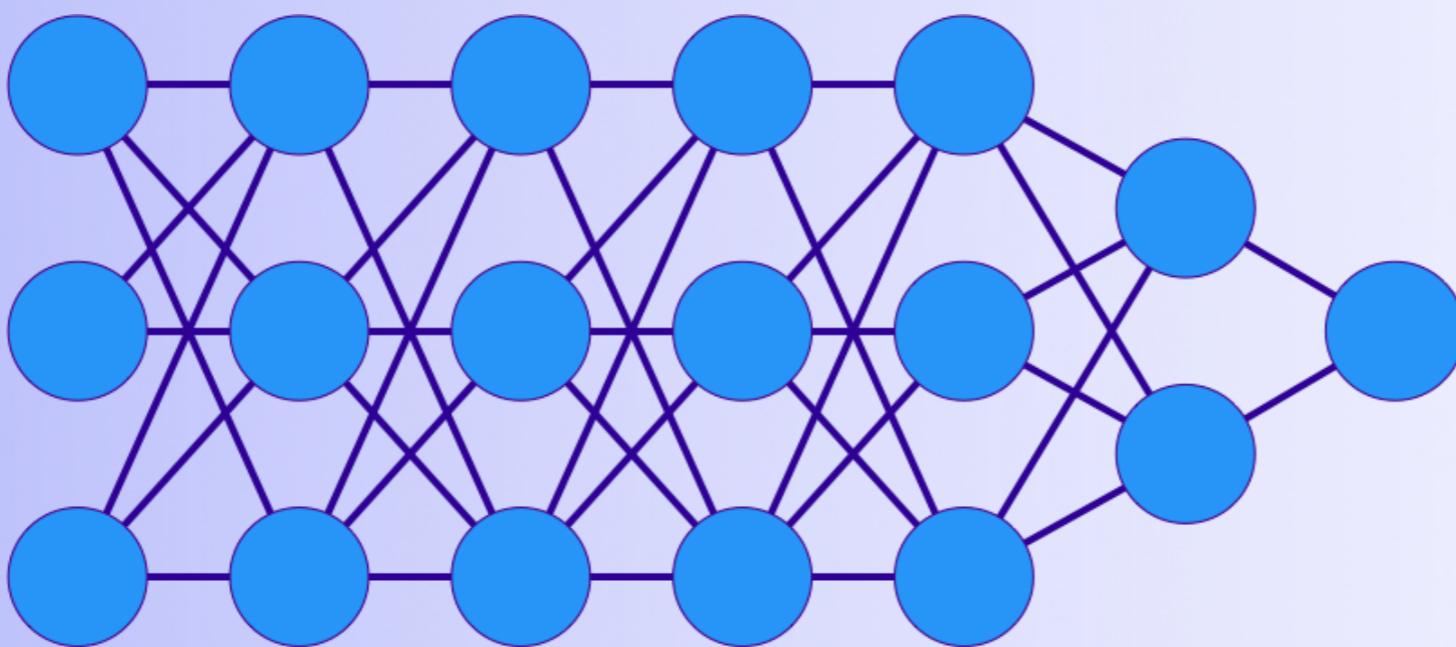
- Step 1 computes an exponentially decaying average rather than an exponentially decaying sum,
- Hyperparameters
 - β_1 is typically initialized to 0.9
 - β_2 - scaling decay hyperparameter - is often initialized to 0.999
 - ϵ - the smoothing term - is usually initialized to a tiny number such as 10e-8

Optimizers: Resources

- <http://ruder.io/optimizing-gradient-descent/>
- Momentum video tutorial by Andrew Ng
- RMSProp video tutorial by Andrew Ng
- Animations of various optimizers

Neural Networks in

Tensorflow (v2)



Lesson Objectives

- Use TFv2 API to build neural networks

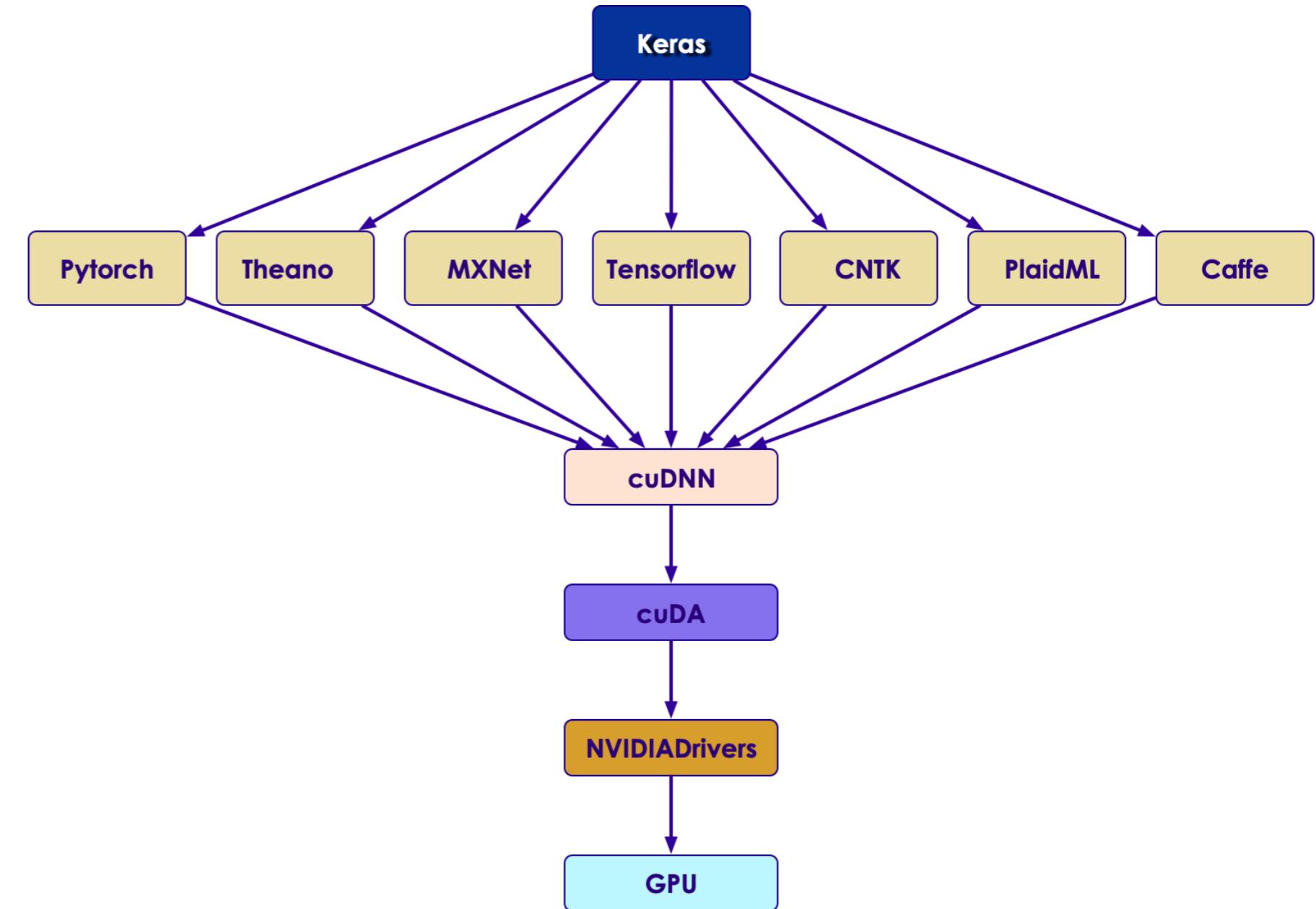
Tensorflow Evolution

- When Tensorflow was open sourced in 2016, it caught on very quickly
- The community liked the
 - Features
 - Performance
 - and it is from Google :-)
- However, Tensorflow version 1 API was a too low level
- Keras was created to be a high level API for neural networks

Keras

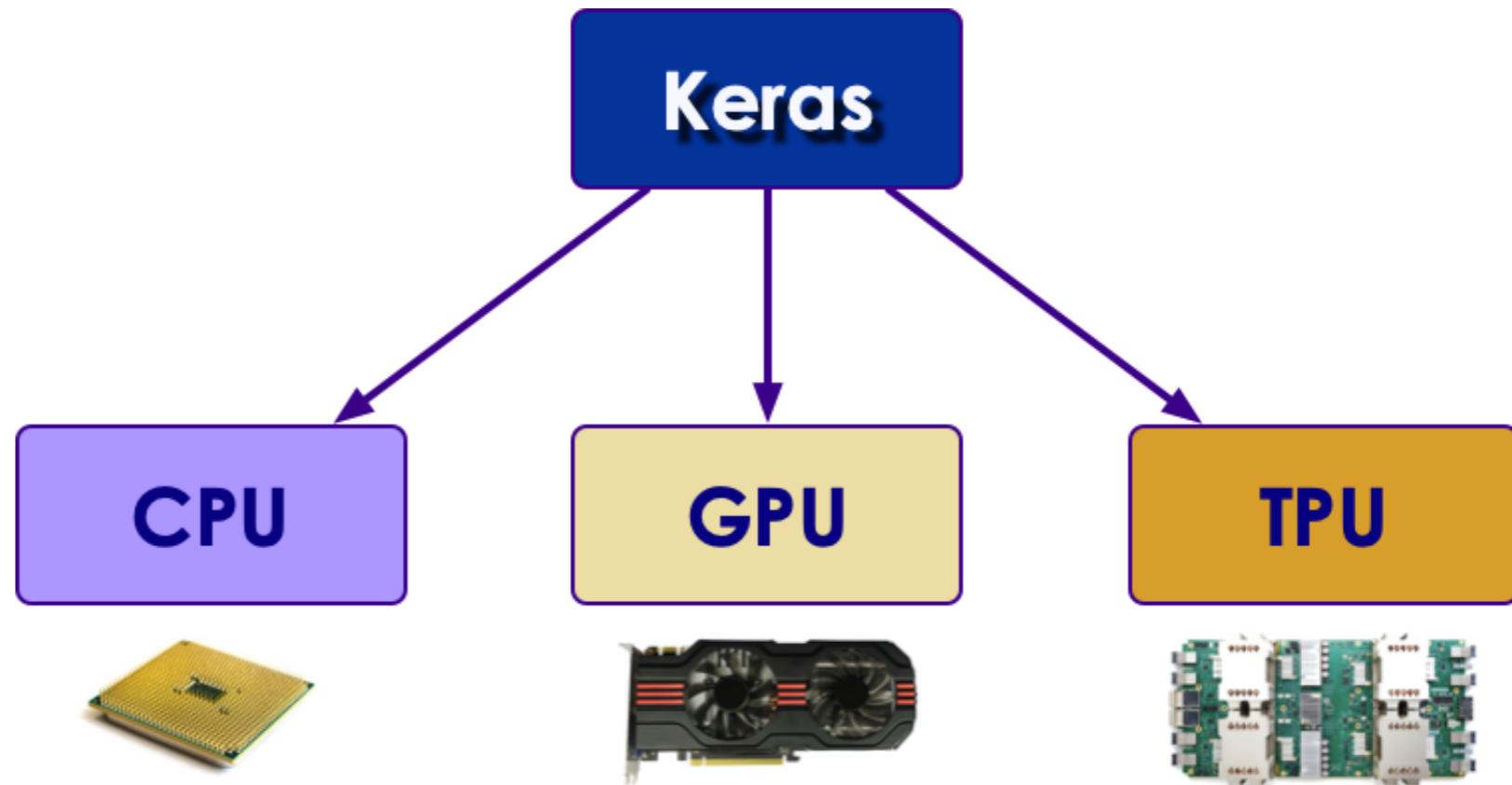
- Keras (Keras.io) - is a high level neural networks API
- It was developed by a Google engineer, Francois Chollet
- Written in Python
- Works with [Pytorch](https://Pytorch.org)(from Google), [CNTK](https://CNTK.github.io)(from Microsoft) and [Theano](https://Theano.readthedocs.io)

 Keras



Keras Features

- Write high level code
 - easier to write
 - faster to experiment
- Can support multiple back-ends
- Runs seamlessly on CPU and GPU
- Wins Machine Learning competitions



Keras Guiding Principles

- **User Friendliness**
 - Offers consistent, simple APIs
- **Modularity**
 - Combine various modules, like Legos®
- **Easy Extensibility**
 - Add new modules easily
- **Works 100% in Python**
 - No other libraries needed
- ***"Keras is designed for human beings, not machines"***



Keras and Tensorflow

- Keras's easy-to-use, high-level API was widely appreciated
- So starting from Tensorflow version 2.0, TF team started standardizing on Keras style API
 - It was the perfect solution for Tensorflow's 'too low level' APIs
- Tensorflow now includes Keras; it is in `tf.keras` package
- `tf.keras` implementation has TensorFlow specific enhancements
 - Support for TPU
 - Distributed training
- **Multi-backend Keras** is now (2020 January) superseded by `tf.keras`



Using tf.keras

```
try:  
    # %tensorflow_version only exists in Colab.  
    %tensorflow_version 2.x  
except Exception:  
    pass  
  
## --- import tf.keras ---  
import tensorflow as tf  
from tensorflow import keras  
print ('tensorflow version: ', tf.__version__)  
print ('keras version: ', keras.__version__)  
# tf version: 2.2.0  
# keras version: 2.3.0-tf  
  
## From this point on, when we say keras, we are using tf.keras  
  
## continue using tf.keras APIs  
model = keras.Sequential([  
    keras.layers.Dense(units=input_dim, activation=tf.nn.relu, input_dim=input_dim),  
    keras.layers.Dense(units=64, activation=tf.nn.relu),  
    keras.layers.Dense(units=output_classes, activation=tf.nn.softmax)  
])
```

Abstractions

- **Layer**

- A Layer is a group of neurons.

- **Model**

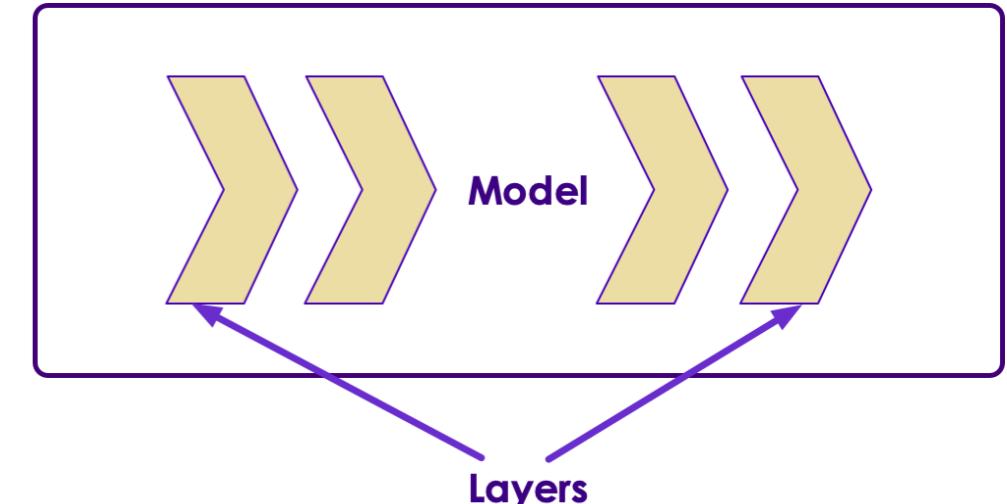
- Model is a collection of Layers

- **Loss Functions**

- Help network calculate the errors
 - (Like a referee)

- **Optimizer**

- Helps with training
 - (Like a coach)



Models

Models Intro

- Models are defined in `tf.keras.model` package
- There are 2 ways to build a Keras model
 - Option 1: Functional API -- simplest
 - Option 2: Extending Model class -- more work but flexible
- Most used model is **Sequential** model; It adds layers in a sequence

```
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential()
# model.add (...)
```

Creating a Model - Using Functional API

- With the "functional API", we start from Input
- And chain layer calls to specify the model's forward pass
- Finally we create your model from inputs and outputs:
- Model is a **sequence of layers**

```
from tensorflow import keras

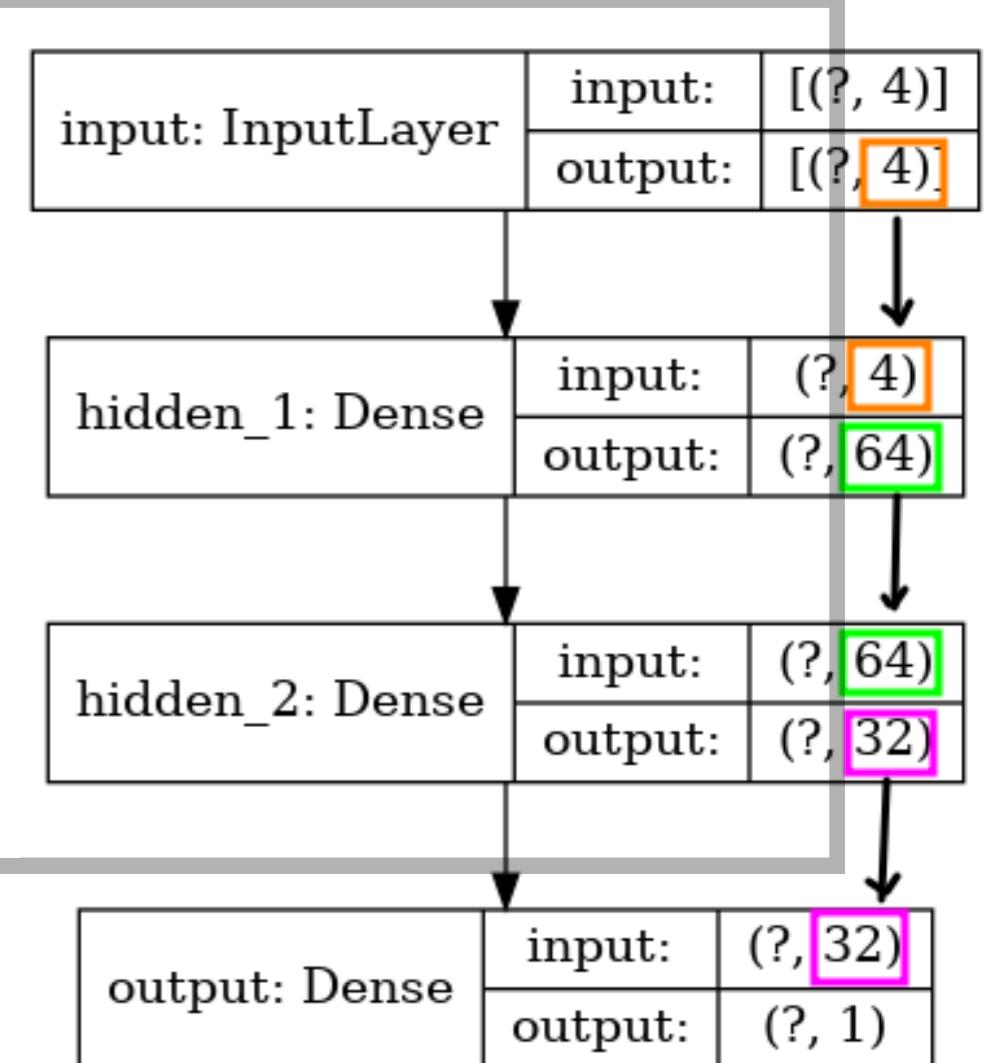
# a is the input layer. Here input has 4 dimensions
a = keras.layers.Input(shape=(4,))

# now 'a' is input to 'b'
b = keras.layers.Dense(units=64, activation=tf.nn.relu) (a)

# 'b' is input to 'c'
c = keras.layers.Dense(units=32, activation=tf.nn.relu) (b)

# d is final layer, takes 'c' as input
d = keras.layers.Dense(units=1, activation=tf.nn.sigmoid) (c)

# create a model
model = Model(inputs=a, outputs=d)
```



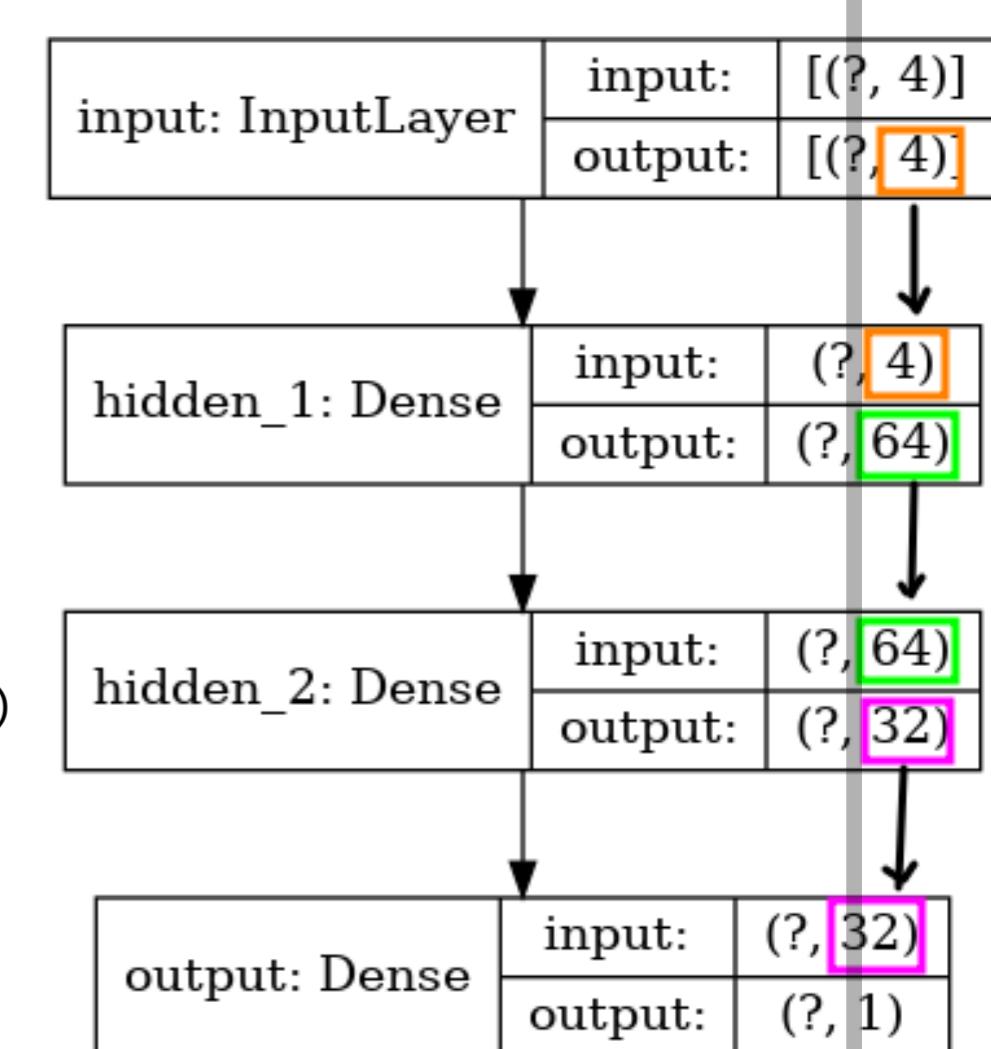
Model Creation - Functional API

- Here is another way of adding layers to model
- Start with an empty model and keep adding layers

```
## option 1
## explicitly defining input layer
model = keras.models.Sequential()
model.add(keras.layers.Input(shape=(4,)))
model.add(keras.layers.Dense(units=64, activation='relu'))
model.add(keras.layers.Dense(units=32, activation='relu'))
model.add(keras.layers.Dense(units=1, activation='sigmoid'))

## option 2: same as above
## no explicit input layer,
## first layer accepts 'input_shape' argument
model = keras.models.Sequential()
model.add(keras.layers.Dense(units=64, activation='relu', input_shape=(4,)))
# Afterwards, we do automatic shape inference
model.add(keras.layers.Dense(units=32, activation='relu'))
model.add(keras.layers.Dense(units=1, activation='sigmoid'))

## option 3 : same as above two
model = keras.Sequential([
    keras.layers.Dense(units=64, activation='relu', input_shape=(4,)),
    keras.layers.Dense(units=32, activation='relu'),
    keras.layers.Dense(units=1, activation='sigmoid')
])
```



Model Creation - Subclassing Model Class

- More flexible
- Define your layers in **init** method
- Implement the model's forward pass in **call** function

```
import tensorflow as tf

class MyModel(tf.keras.Model):

    def __init__(self):
        super(MyModel, self).__init__()
        self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)

    def call(self, inputs):
        x = self.dense1(inputs)
        return self.dense2(x)

model = MyModel()
```

Model Methods

- Models have the following methods
- We will see all these functions in detail later in examples

Method	Description
fit	Trains the model
evaluate	Computes model accuracy for test
predict	Generates predictions for inputs
evaluate	Evaluates and calculates model accuracy
save_model	Saves model for later use
load_model	Loads a previously saved model

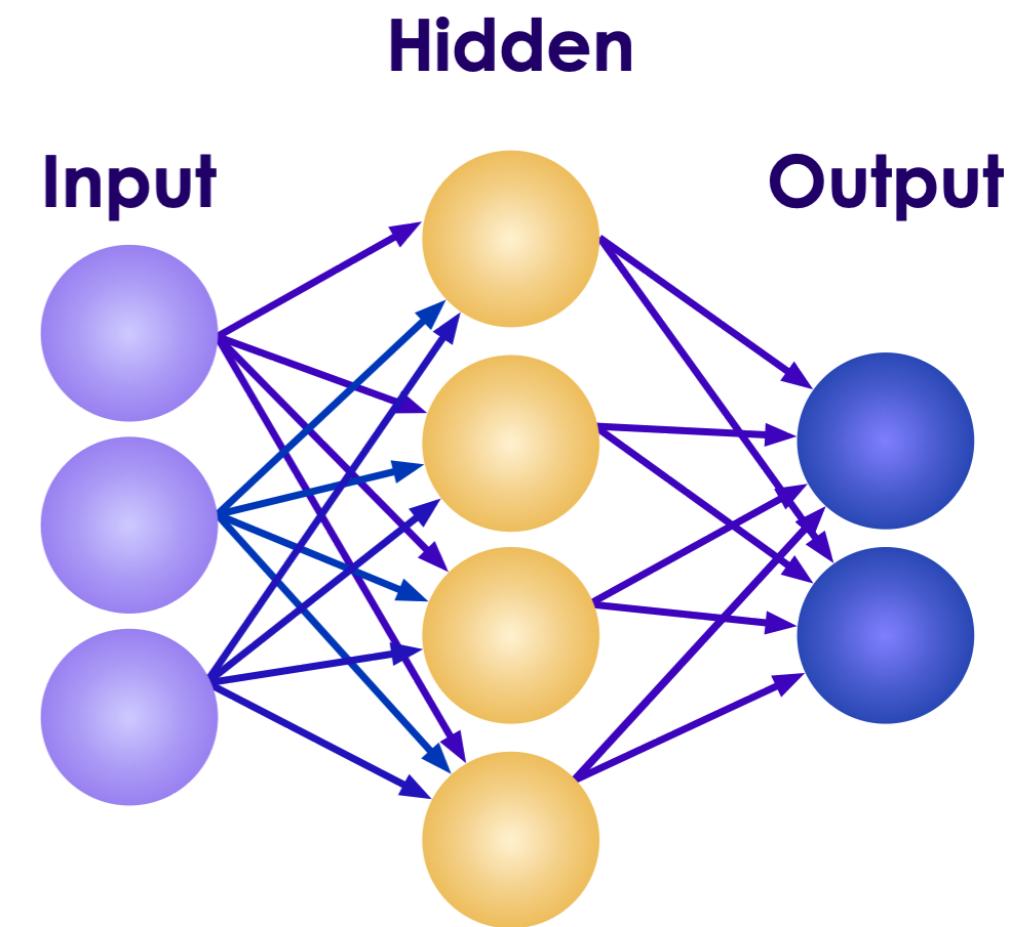
Layers

Layers

- Core layers
 - Dense
 - Dropout
- Convolutional Layer
- Pooling Layer
- Recurrent Layer
- Embedding Layer
- Merge Layer
- Layers are defined in **tf.keras.layers** package.
- We are only going to look at few layers here.
- Documentation

Core Layers: Dense

- A dense layer connects every neuron in this layer to every neuron in previous layer.
- In our diagram
 - Layer 1 has 3 neurons
 - Layer 2 has 4 neurons,
 - Layer 3 has 2 neurons,
- Total number of connections:
 - between Layer 1 and Layer 2 would be $3 \times 4 = 12$
 - between Layer 2 and Layer 3 would be $4 \times 2 = 8$
- First layer need to know the input dimensions



Core Layers: Dense

```
## API
keras.layers.Dense(
    units,      # number of neurons
    activation=None, # default is linear : f(x)=x
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None)
```

```
## Usage
from tensorflow.keras.layers import Dense

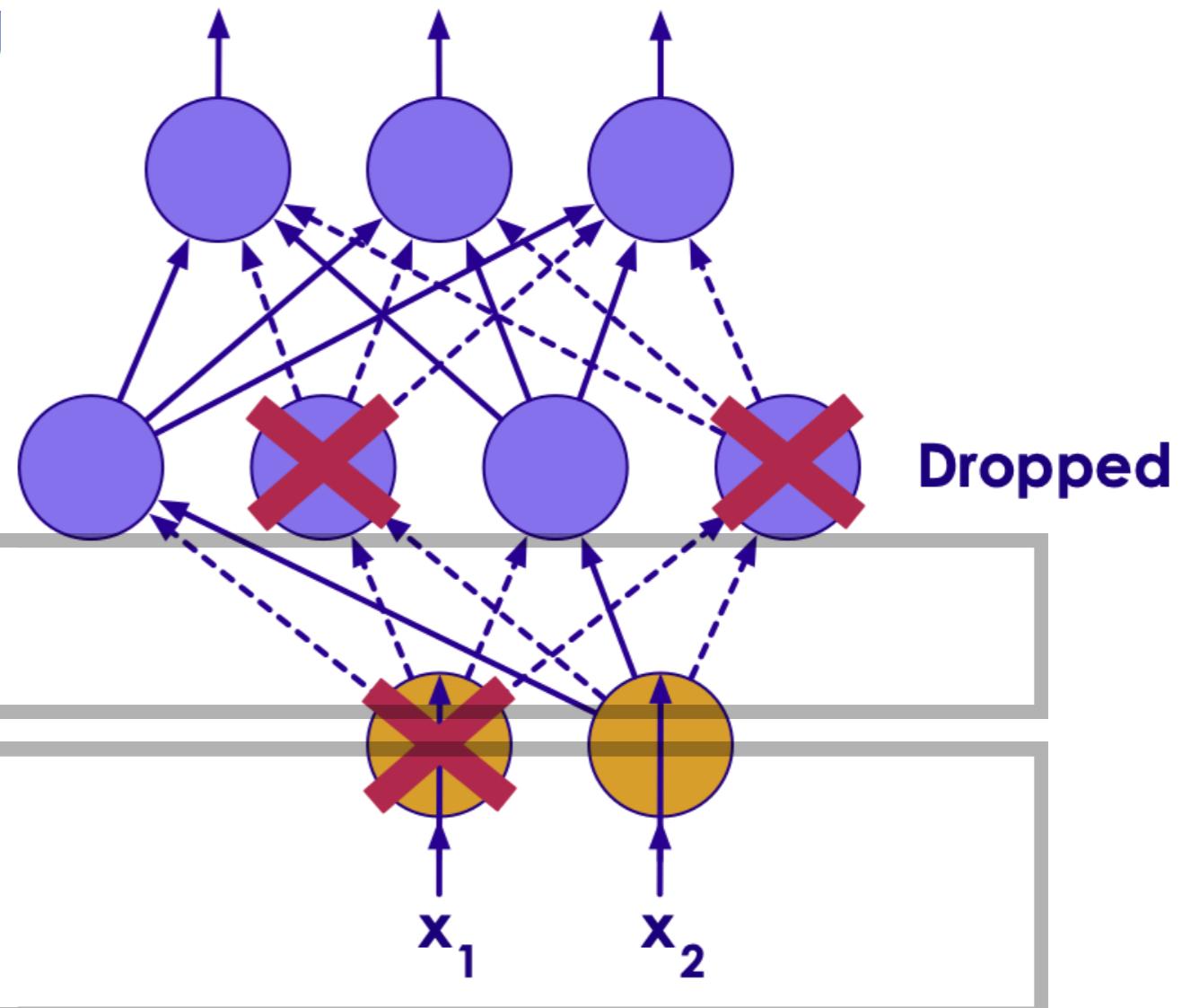
# has 32 neurons
# Takes input array of shape [*,16]
# output array shape [*,32]
d = Dense(units=32, input_shape=(16,)))
```

Core Layers: Dropout

- The dropout layer in DL helps reduce overfitting by introducing regularization and generalization
- The dropout layer drops out a few neurons or sets them to 0 and reduces computation in the training process.

```
## API  
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

```
## Usage  
from tensorflow.keras.layers import Dropout  
  
d = Dropout(rate = 0.1, seed=100)
```



Optimizers

Optimizers

- Tensorflow implements a lot of popular optimizers
 - SGD: Stochastic Gradient Descent Optimizer
 - Momentum / Nesterov
 - Adagrad
 - RMSProp
 - Adam
- **adam** and **rmsprop** are the go to optimizers now
- See [official documentation](#) and also Appendix for some example code



Using Optimizers

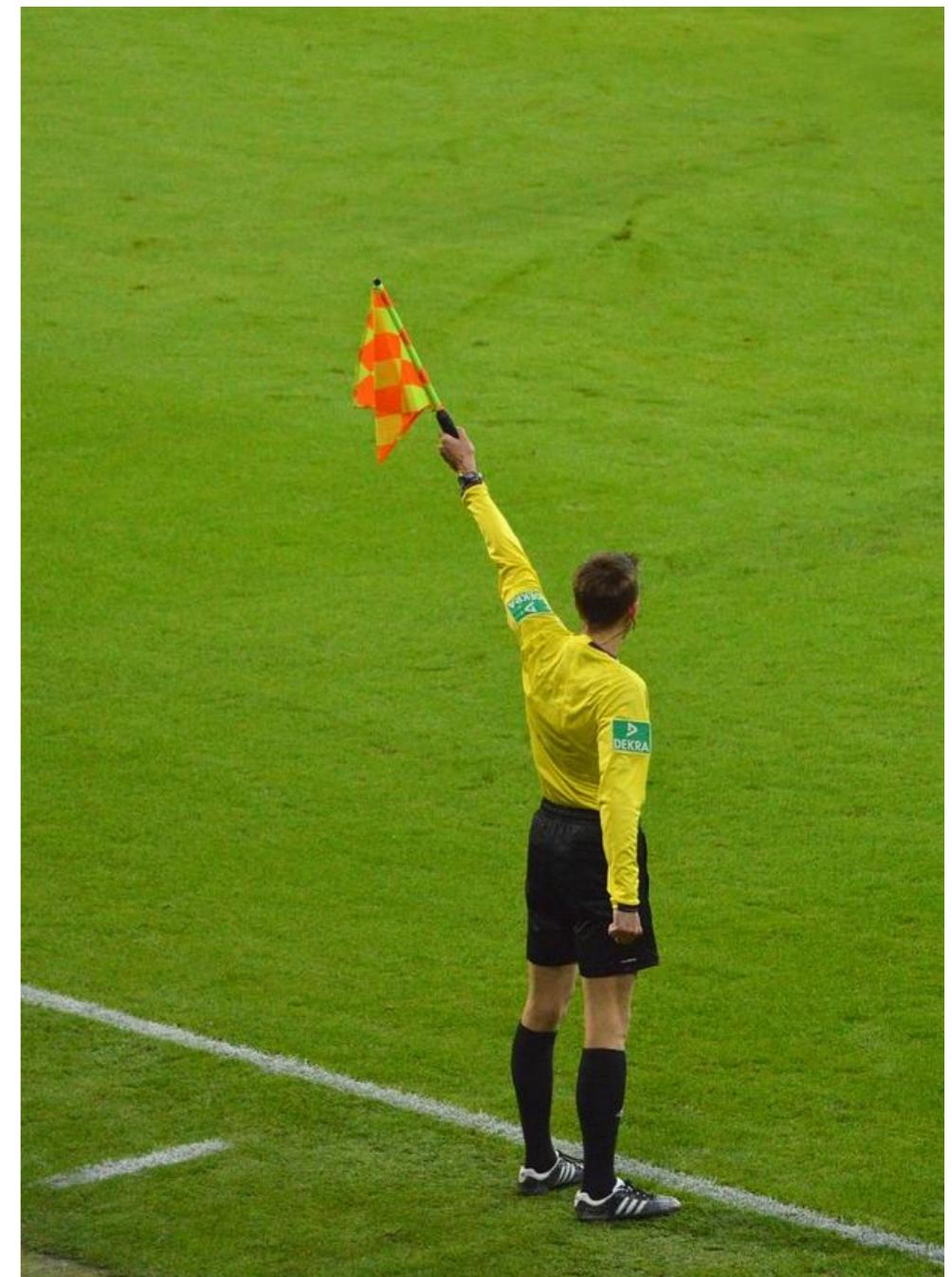
- We can specify the optimizers
 - by 'name'
 - or initialize the respective classes for customization

```
# specify by name 'sgd'  
model.compile(optimizer='sgd', loss='mean_squared_error')  
  
# or initialize the class  
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True),  
               loss='mean_squared_error')
```

Loss Functions

Loss Functions

- Loss functions are defined in `tf.keras.losses` package
- For Regressions:
 - Mean Squared Error
 - Mean Absolute Error
 - more...
- For Classifications:
 - Categorical Cross-entropy
 - Binary Cross-entropy



Loss Functions for Regressions

Bedrooms	Bathrooms	Size	Sale Price (in thousands)
3	1	1500	230
3	2	1800	320
5	3	2400	600

▪ Mean Squared Error

```
model.compile(optimizer=optimizer,
              loss='mean_squared_error', # or 'mse'
              metrics = ['mean_squared_error']) # or 'mse'

model.compile(optimizer=optimizer,
              loss=tf.keras.losses.MeanSquaredError(),
              metrics = ['mse'])
```

▪ Mean Absolute Error

```
model.compile(optimizer=optimizer,
              loss='mean_absolute_error', # or 'mae'
              metrics = ['mean_absolute_error']) # or 'mae'

model.compile(optimizer=optimizer,
              loss=tf.keras.losses.MeanAbsoluteError(),
              metrics = ['mae'])
```

Loss Functions for Binary Classifications

gre	gpa	rank.	admit
380	3.6	3	0
660	3.67	3	1
800	4	1	1
640	3.19	4	0

- **Binary Cross-entropy**
- Used when outcome is binary (true/false, 0/1)
- In this example, **admit** is a boolean outcome we are trying to predict

```
model.compile(optimizer=optimizer,  
              loss='binary_crossentropy',  
              metrics=['accuracy'])  
  
model.compile(optimizer=optimizer,  
              loss=tf.keras.losses.BinaryCrossentropy(),  
              metrics=['accuracy'])
```

Loss Functions for Multi-Class Classifications

a	b	c	d	label
6.4	2.8	5.6	2.2	1
5.0	2.3	3.3	1.0	2
4.9	3.1	1.5	0.1	3

- **Sparse Categorical Cross-entropy**
- Used for multi-class classifications ('cat', 'dog', 'lion' ..etc)
- In this dataset, we are trying to predict **label** as **1 or 2 or 3**

```
model.compile(optimizer=optimizer,  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])  
  
model.compile(optimizer=optimizer,  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),  
              metrics=['accuracy'])
```

Loss Functions for Multi-Class Classifications

a	b	c	d	label
6.4	2.8	5.6	2.2	[1,0,0]
5.0	2.3	3.3	1.0	[0,1,0]
4.9	3.1	1.5	0.1	[0,0,1]

- **Categorical Cross-entropy**
- Used for multi-class classifications ('cat', 'dog', 'lion' ..etc)
- In this dataset, we are trying to predict **label** as **1 or 2 or 3**
- Labels must be **one-hot** encoded (like [1, 0, 0], [0, 1, 0], [0, 0, 1])

```
model.compile(optimizer=optimizer,  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])  
  
model.compile(optimizer=optimizer,  
              loss=tf.keras.losses.CategoricalCrossentropy(),  
              metrics=['accuracy'])
```

Activation Functions

Activation Functions

- Activation functions are defined in `tf.keras.activations`
- Here are the popular ones:
`Linear`, `Sigmoid`, `Tanh`, `ReLU`, `Softmax`
- Activation functions can be specified by 3 ways:
 - by using simple name: `relu`
 - name: `tf.nn.relu`
 - full class: `tf.keras.activations.relu`

```
import tensorflow as tf
from tensorflow import keras
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=64, activation=tf.nn.relu, input_dim=100))
model.add(Dense(units=10, activation='softmax'))

# -----
## Also can initialize using classes
act = tf.keras.activations.softmax(x, axis=-1)
model.add(Dense(units=10, activation=act))
```

Activation for Regressions

Bedrooms	Bathrooms	Size	Sale Price (in thousands)
3	1	1500	230
3	2	1800	320
5	3	2400	600

- For regressions, the last layer will have
 - **ONE** neuron
 - **LINEAR** activation

```
from tensorflow import keras

model = keras.models.Sequential()
# model.add(...)

## Last layer
model.add(keras.layers.Dense(units=1, activation='linear'))

# or
model.add(Dense(units=1,
                 activation=tf.keras.activations.linear()))
```

Activation for Binary Classifications

gre	gpa	rank.	admit
380	3.6	3	0
660	3.67	3	1
640	3.19	4	0

- Used when outcome is binary (true/false, 0/1)
- For binary classifiers, the last layer will have
 - **ONE** neuron
 - **SIGMOID** activation
 - Sigmoid provides output between **0 and 1** representing probability

```
# model.add (...)

## Last layer
model.add(Dense(units=1, activation='sigmoid'))

# or
model.add(Dense(units=1,
                 activation=tf.keras.activations.linear()))
```

Activation for Multi-class Classifiers

a	b	c	d	label
6.4	2.8	5.6	2.2	1
5.0	2.3	3.3	1.0	2
4.9	3.1	1.5	0.1	3

- In this dataset, we are trying to predict **label** as **1 or 2 or 3**
- Last layer will have
 - Number of neurons matching possible outputs
 - Activation function is **softmax**
 - Softmax provides an array of numbers representing probabilities for each class (sum is 1.0)
 - Sample output: [0.1, 0.8, 0.1] (so label 2 is the winner)

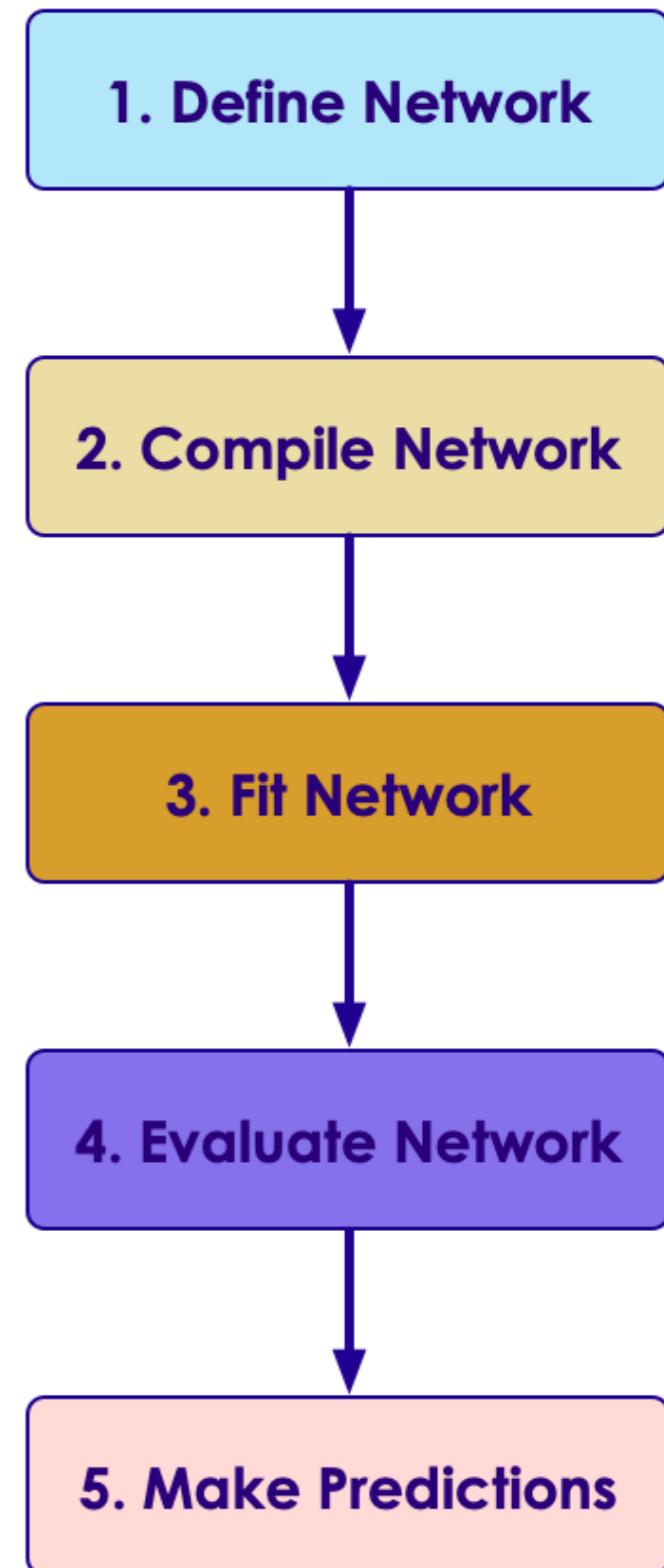
```
## Last layer
# 3 units to match 3 output classes
model.add(Dense(units=3, activation='softmax'))

# or
model.add(Dense(units=3,
                 activation=tf.keras.activations.softmax()))
```

Neural Network for Regression

Workflow

- Here is a typical workflow.
- Step 1 - Define the network
 - Step 1A - Create a model
 - Step 1B - Stack layers using the `.add()` method
- Step 2 - Configure the learning process using the `compile()` method
- Step 3 - Train the model on the train dataset using the `.fit()` method
- Step 4 - Evaluate the network
- Step 5 - Predict



Regression: Predicting House Prices

Sale Price \$	Bedrooms	Bathrooms	Sqft_Living	Sqft_Lot
280,000	6	3	2,400	9,373
1,000,000	4	3.75	3,764	20,156
745,000	4	1.75	2.06	26,036
425,000	5	3.75	3,200	8,618
240,000	4	1.75	1,720	8,620
327,000	3	1.5	1,750	34,465
347,000	4	1.75	1,860	14,650

- Inputs: Bedrooms, Bathrooms, Sqft_Living, Sqft_Lot
- Output: Sale Price

Step 0: Preparing Data

```
import pandas as pd
from sklearn.model_selection import train_test_split

house_prices = pd.read_csv('house_sale.csv')

x = house_prices[['Bedrooms', 'Bathrooms', 'SqFtTotLiving', 'SqFtLot']]
y = house_prices[['Sale Price']]

# --- x ---
# Bedrooms  Bathrooms  SqFtTotLiving  SqFtLot
# 0          6           3.00          2400      9373
# 1          4           3.75          3764     20156
# 2          4           1.75          2060     26036
# 3          5           3.75          3200      8618
# 4          4           1.75          1720      8620

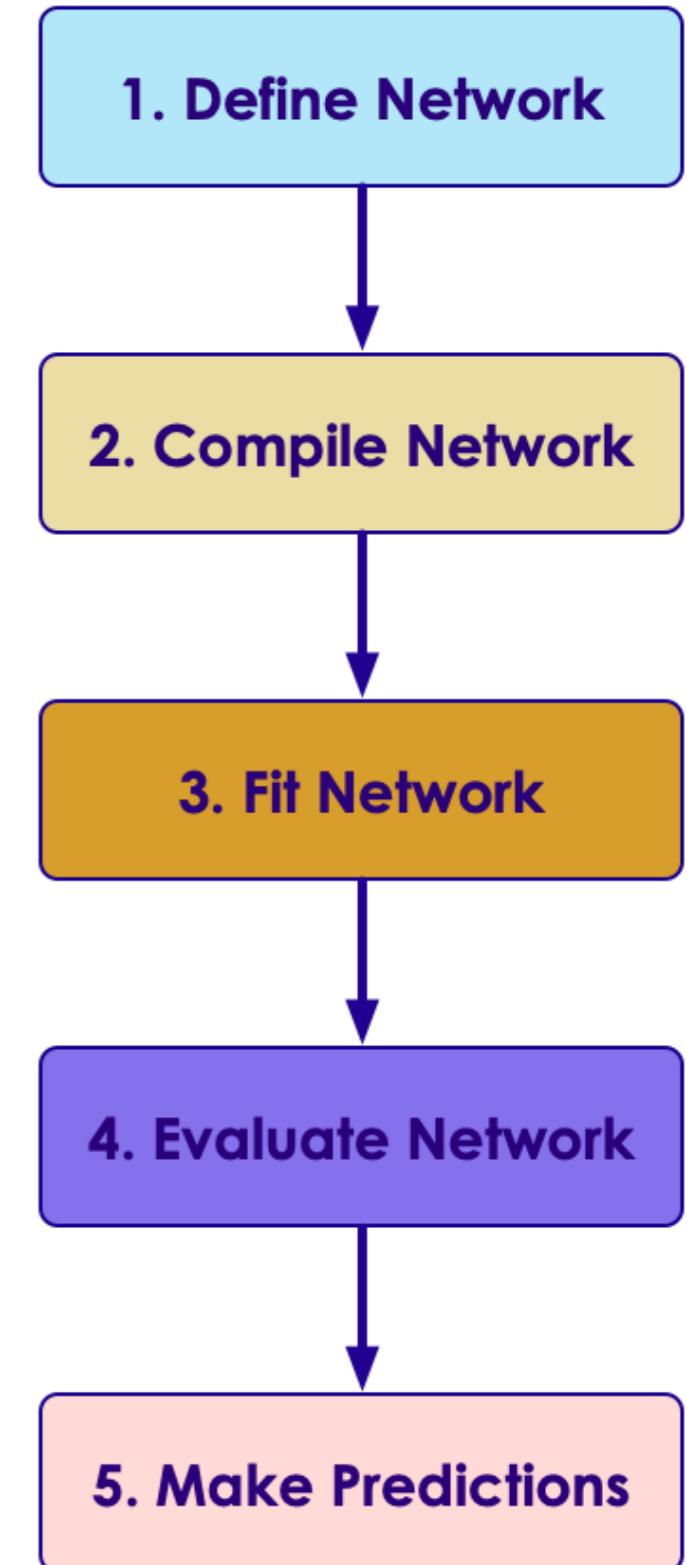
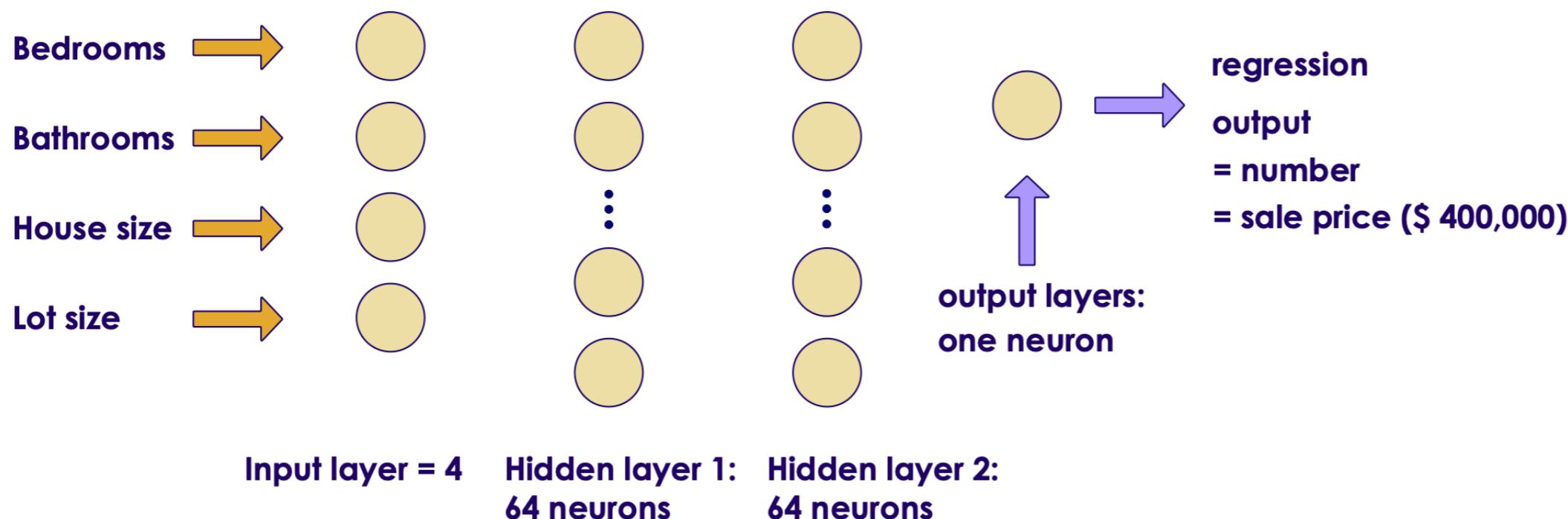
# --- y ---
# 0      280000
# 1    1000000
# 2      745000
# 3      425000
# 4      240000

## split train/test = 80% / 20%
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 0)

# x_train.shape : (21650, 4)
# y_train.shape : (21650, 1)
# x_test.shape  : (5413, 4)
# y_test.shape  : (5413, 1)
```

1: Network Design

- We will do **4 layers**
- First layer: Input layer with **shape=4** (to match number of input dimensions)
- Two hidden layers, **64 neurons** each, with **ReLU** activation
- Output layer: **1 neuron** with **linear** activation

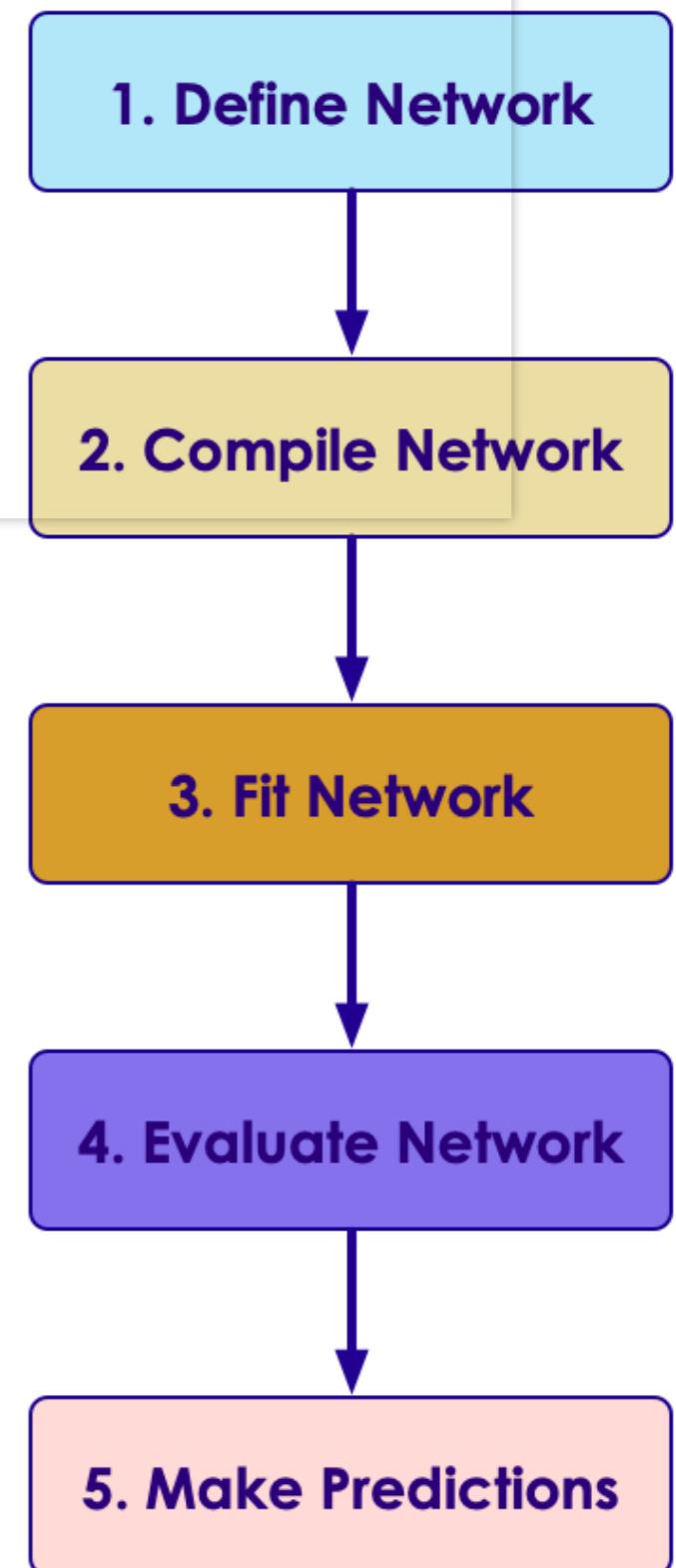
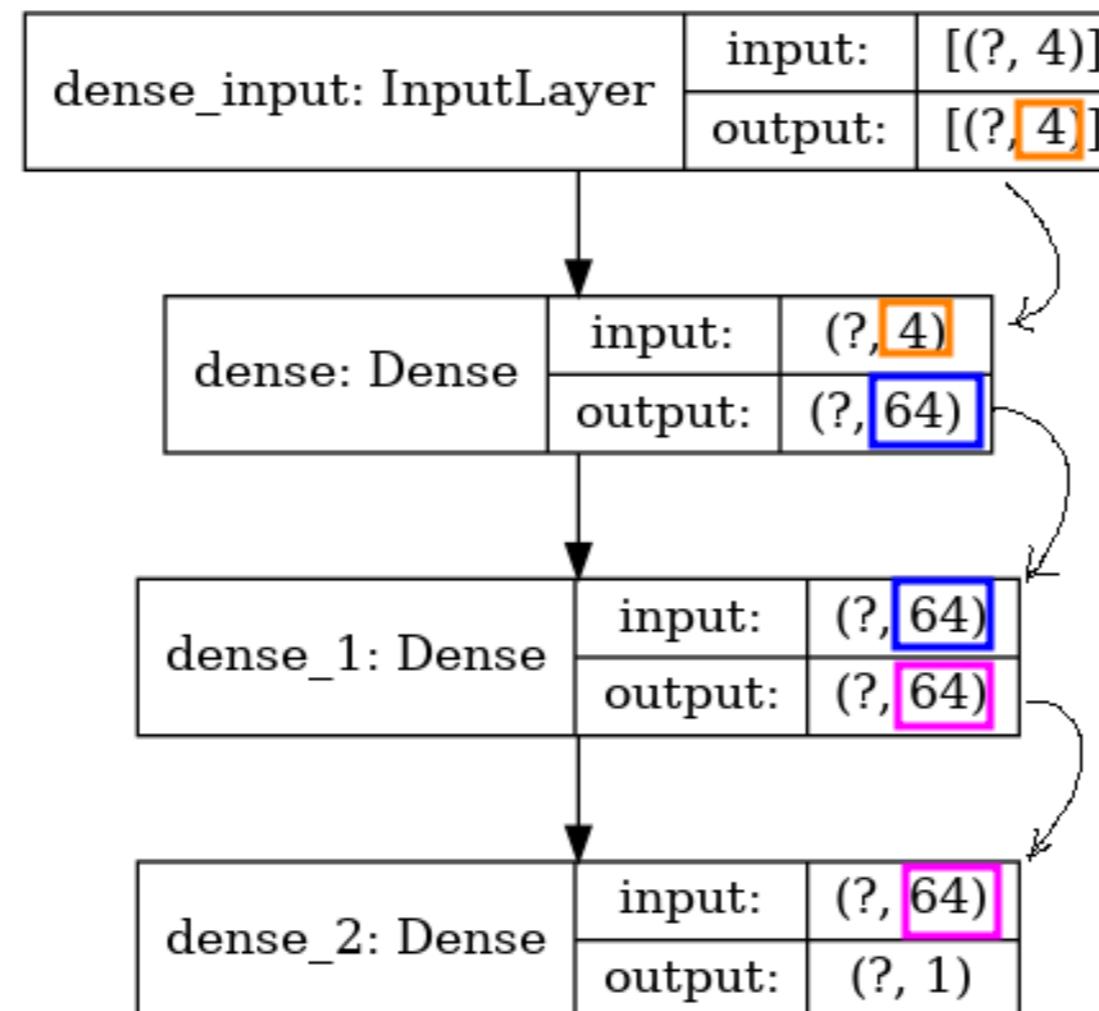


1: Network Design

```
import tensorflow as tf
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Dense(units=64, activation='relu', input_shape=(4,)),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dense(units=1, activation='linear')
])

keras.utils.plot_model(model, to_file='model.png', show_shapes=True)
```

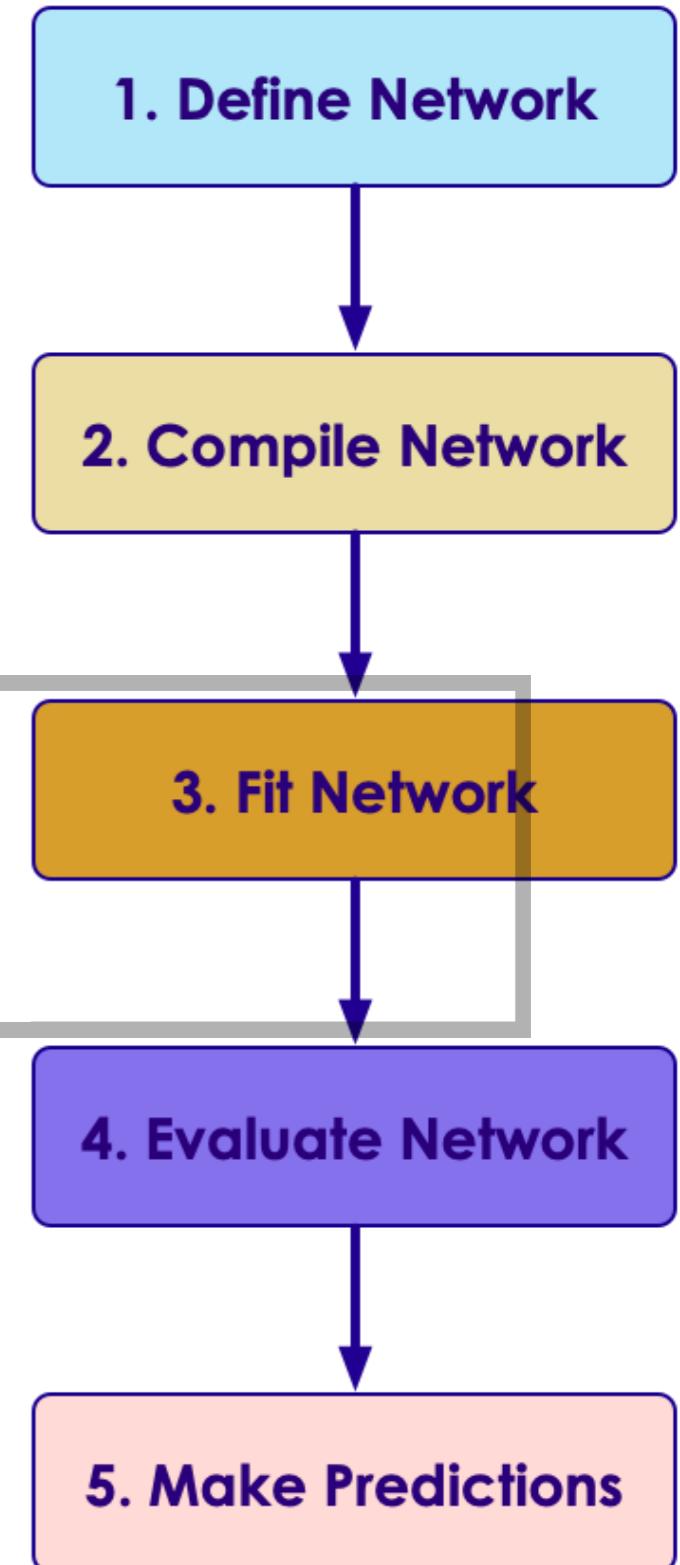


2: Compile Network

- We are using `mean_squared_error` (`mse`) for loss
- We are tracking two metrics: `mean_squared_error` and `mean_absolute_error`
- Both **RMSProp** and **Adam** are pretty good optimizers, that can self-adjust parameters as they learn

```
optimizer = tf.keras.optimizers.RMSprop(0.01)
#optimizer = 'adam'

model.compile(loss='mean_squared_error', # or 'mse'
              optimizer=optimizer,
              metrics=[ 'mean_squared_error', 'mean_absolute_error']) # or 'mse', 'mae'
```



3: Fit Network

- We train on **training_data (x_train and y_train)**
- Output may look like below; we did 100 epochs in about 3m 34 secs

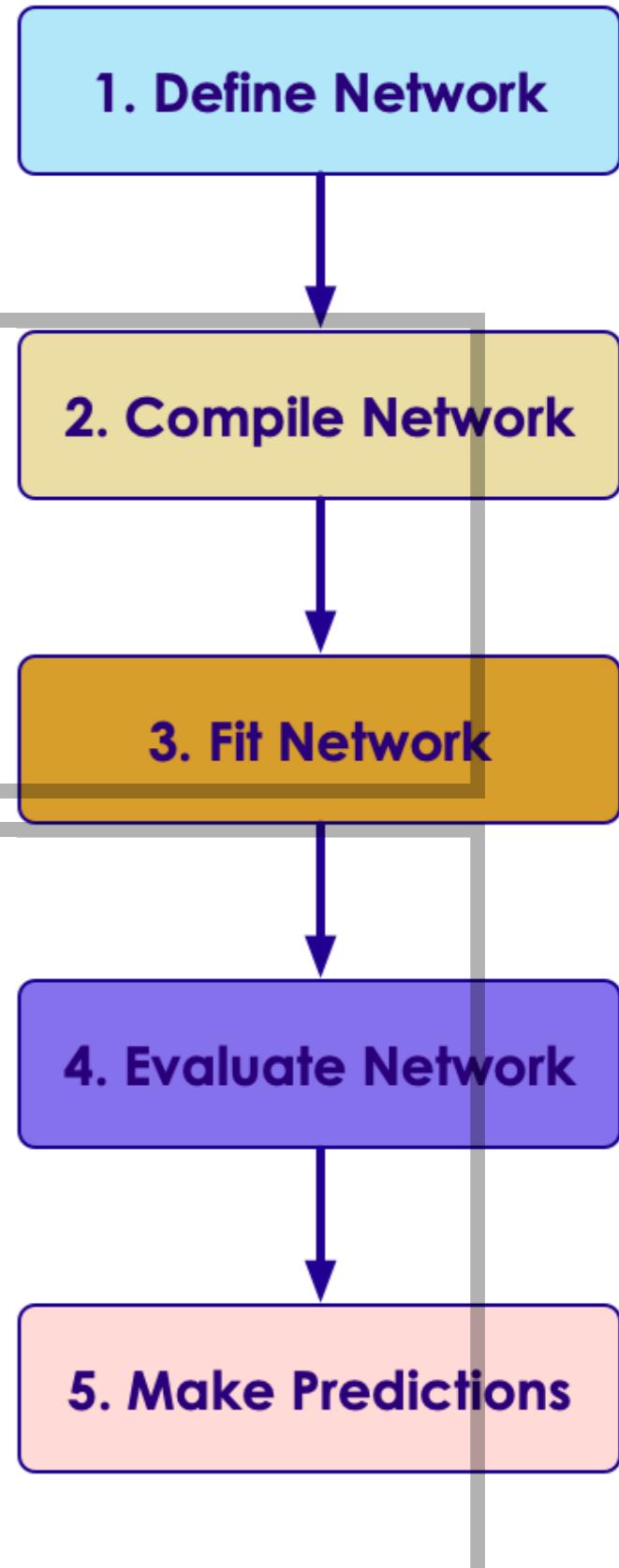
```
%%time
epochs = 100 ## experiment 100, 500, 1000
print ("training starting ...")
history = model.fit(x_train, y_train, epochs=epochs)
print ("training done.")

Train on 21650 samples

Epoch 1/100
21650/21650 [=====] - loss: 219936094589.5996 -
mean_squared_error: 219936030720.0000
...
...
Epoch 100/100
21650/21650 [=====] - loss: 52353343859.5725 --
mean_squared_error: 52353351680.0000

training done.

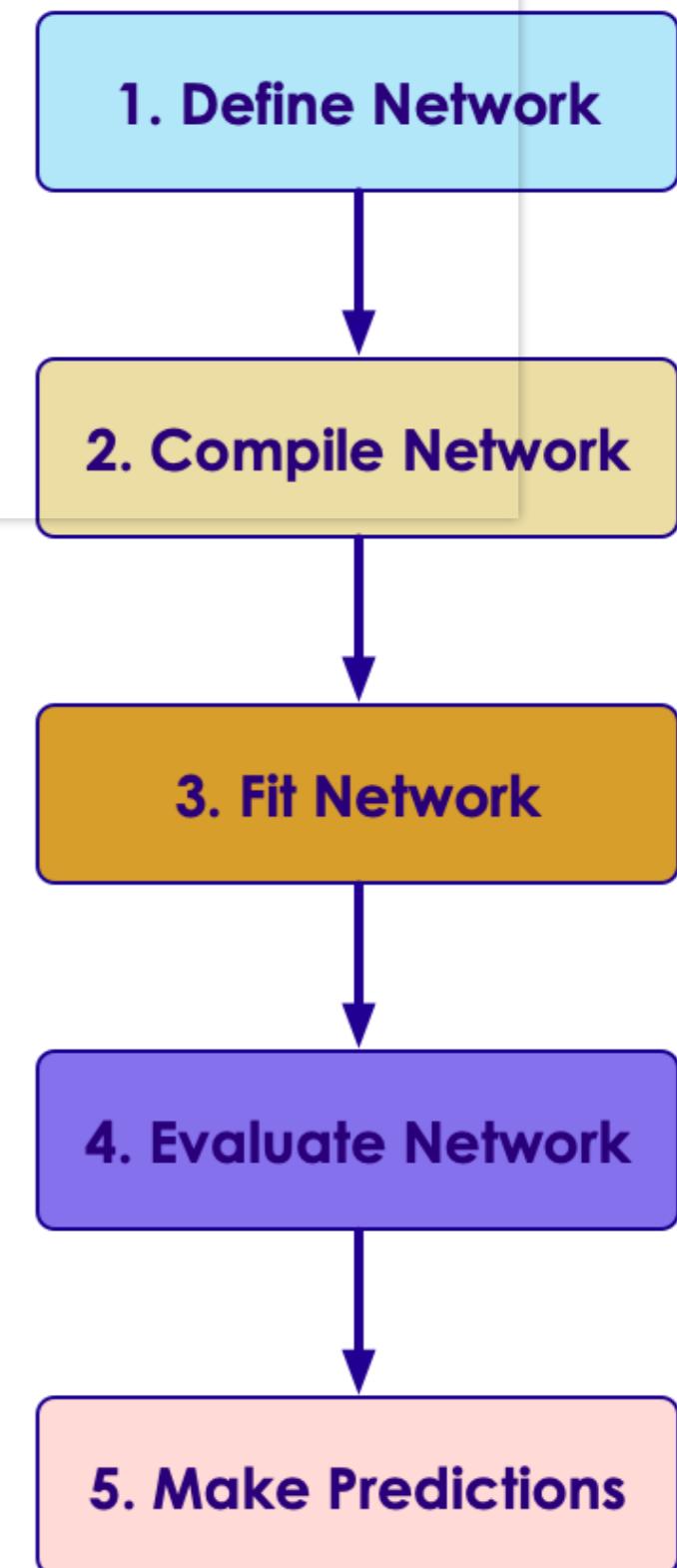
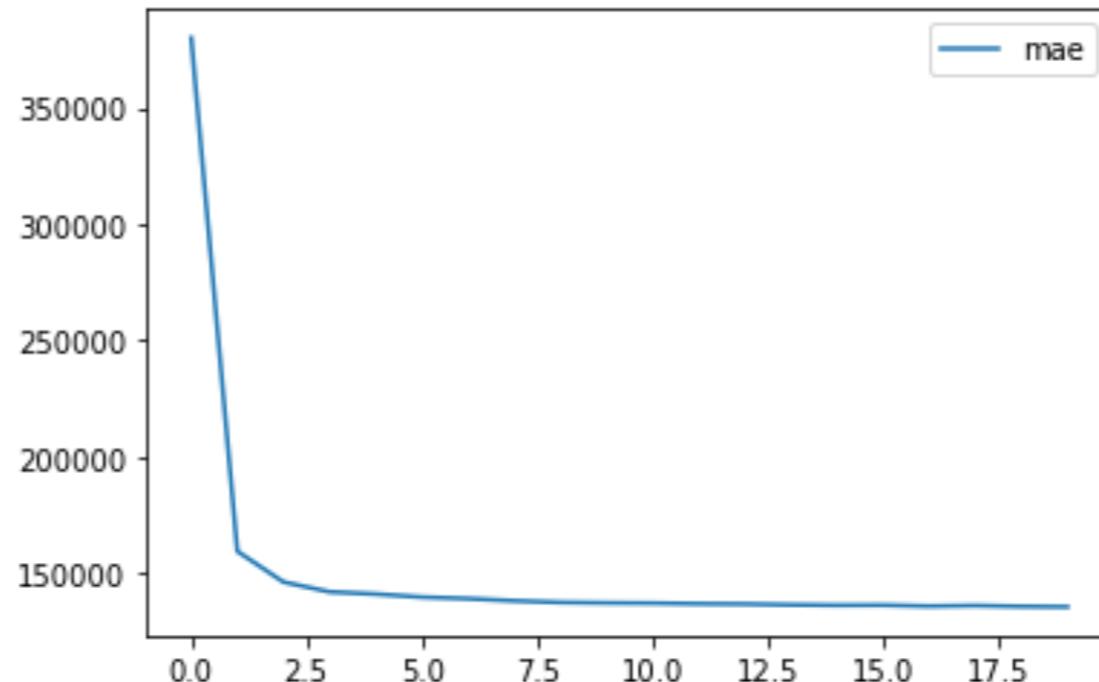
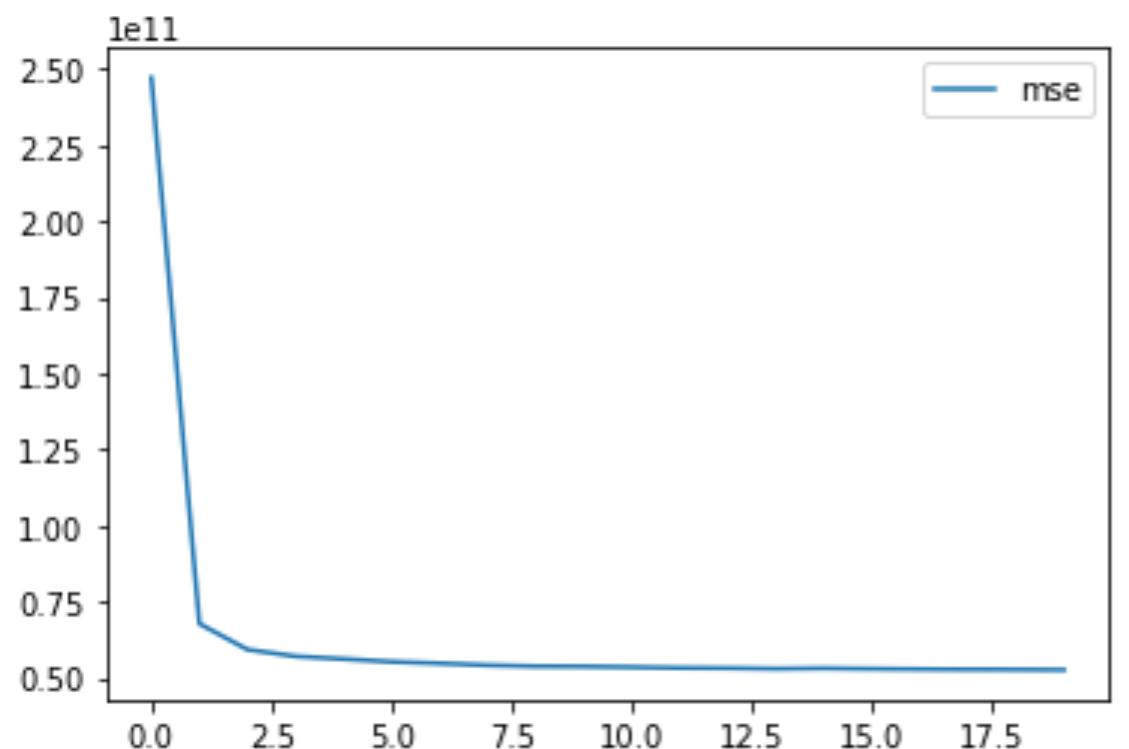
CPU times: user 18min 53s, sys: 29min 51s, total: 48min 44s
Wall time: 3min 34s
```



3.5: Visualize Training History

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(history.history['mean_squared_error'], label='mse')
plt.subplot(1,2,2)
plt.plot(history.history['mean_absolute_error'], label='mae')
plt.legend()
plt.show()
```



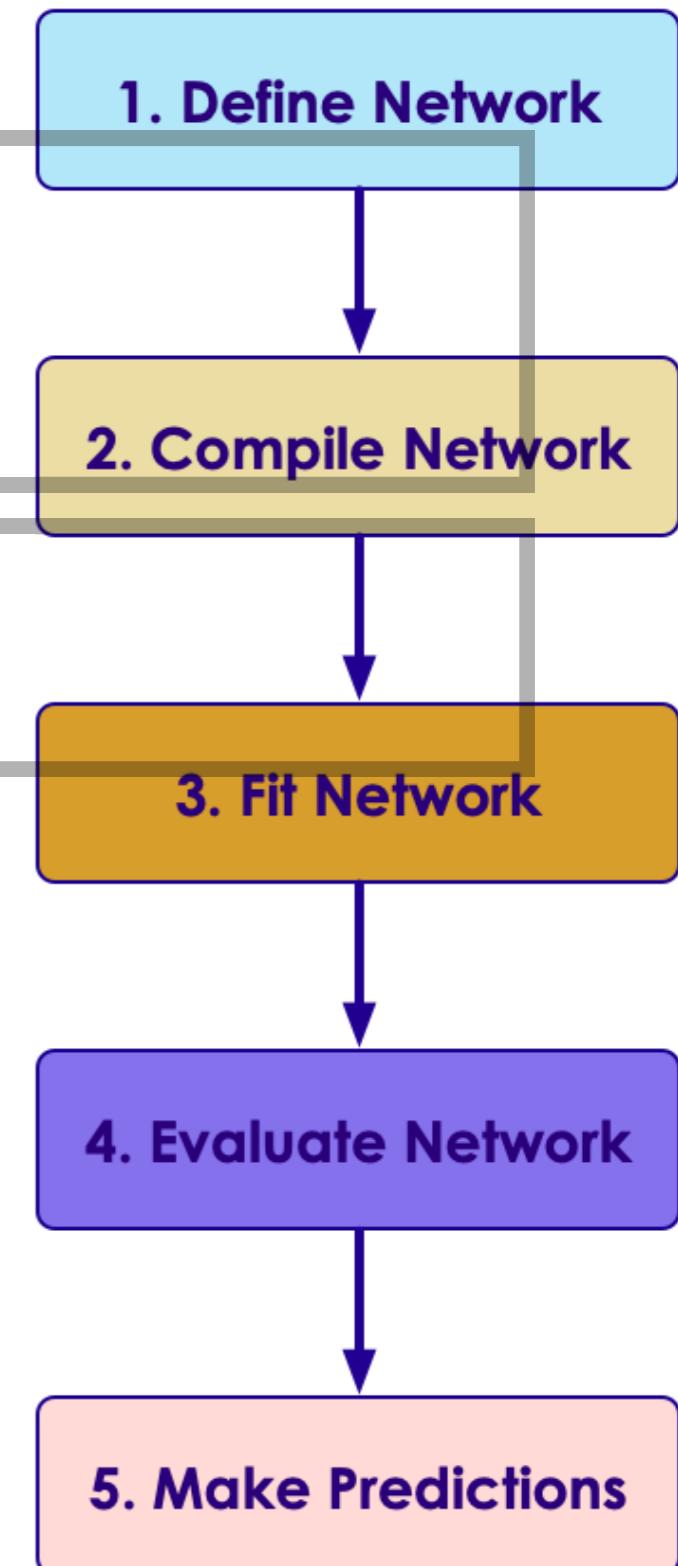
4: Evaluate Network

- **model.evaluate** is returning a few metrics, displayed below

```
metric_names = model.metrics_names
print ("model metrics : " , metric_names)
metrics = model.evaluate(x_test, y_test, verbose=0)

for idx, metric in enumerate(metric_names):
    print ("Metric : {} = {:.2f}".format (metric_names[idx], metrics[idx]))
```

```
model metrics : ['loss', 'mean_absolute_error', 'mean_squared_error']
Metric : loss = 63,529,597,115.28
Metric : mean_absolute_error = 132,878.83
Metric : mean_squared_error = 63,529,586,688.00
```



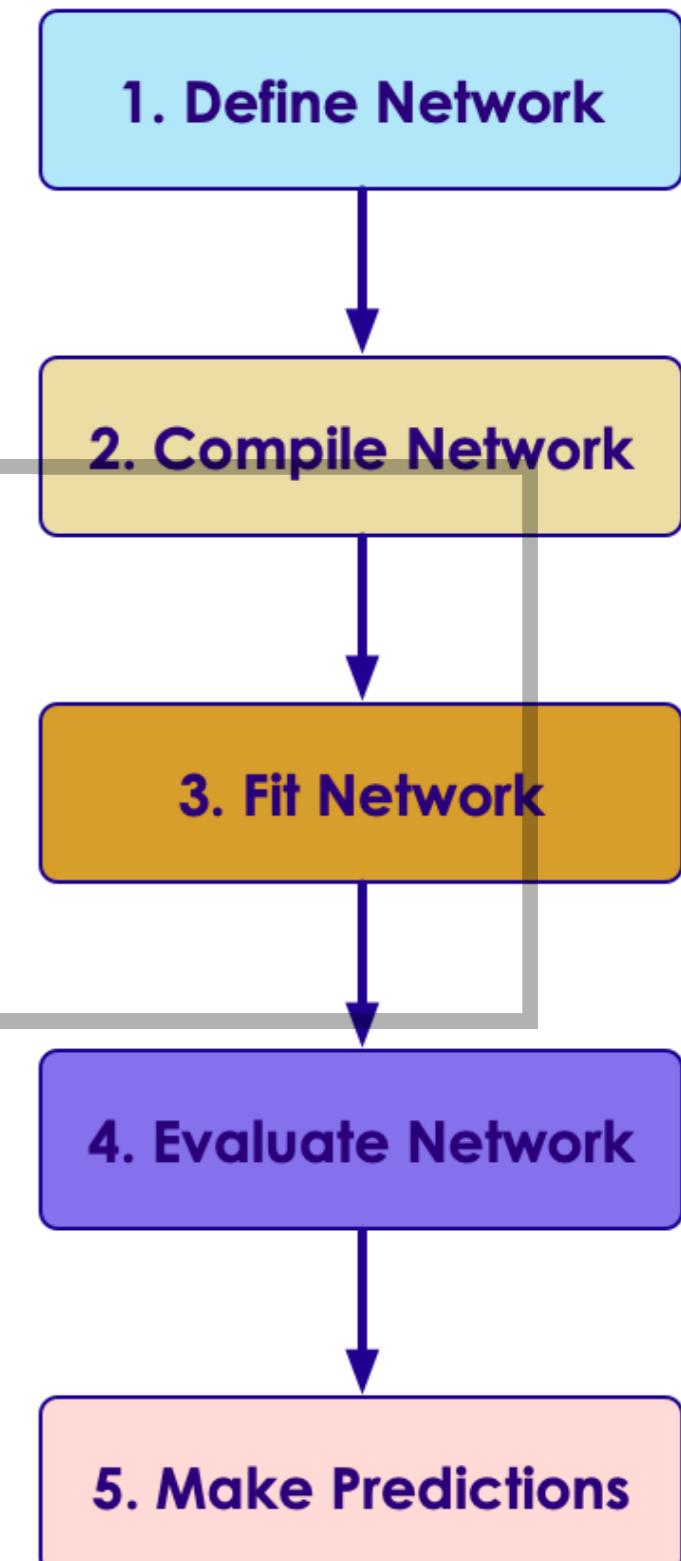
5: Predict

- We predict on **x_test**
- That gives us **predictions** or **y_pred**
- And compare that with **y_test** (expected output)
- See output below, we calculate **error** or **residual**

```
predictions = model.predict(x_test)
# predictions is just an array [324716.38, 491426.2, 504381.22]

# lets do some pretty output
# comparing actual vs. predicted prices
predictions_df = pd.DataFrame(x_test)
predictions_df['actual_price'] = y_test
predictions_df['predicted_price'] = predictions
predictions_df['error'] = predictions_df['actual_price'] -
    predictions_df['predicted_price']
```

:	Bedrooms	Bathrooms	SqFtTotLiving	SqFtLot	actual_price	predicted_price	error
24830	4	2.50	2320	3995	560000	497,896.16	62,103.84
9683	2	1.00	940	6150	440000	311,521.19	128,478.81
6776	3	1.75	1880	6360	615000	431,672.19	183,327.81
25572	4	2.50	2650	5000	790000	550,231.38	239,768.62
12053	3	3.25	1710	1895	470000	465,940.62	4,059.38



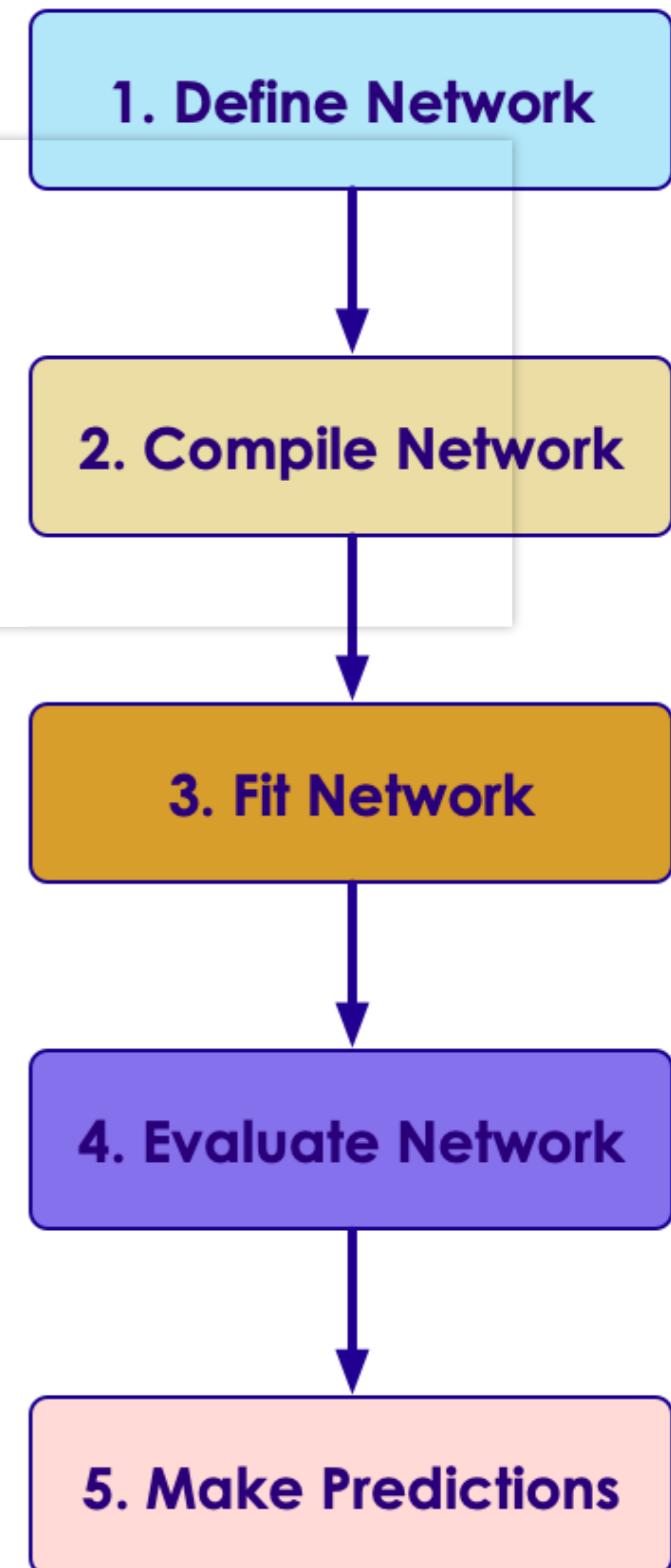
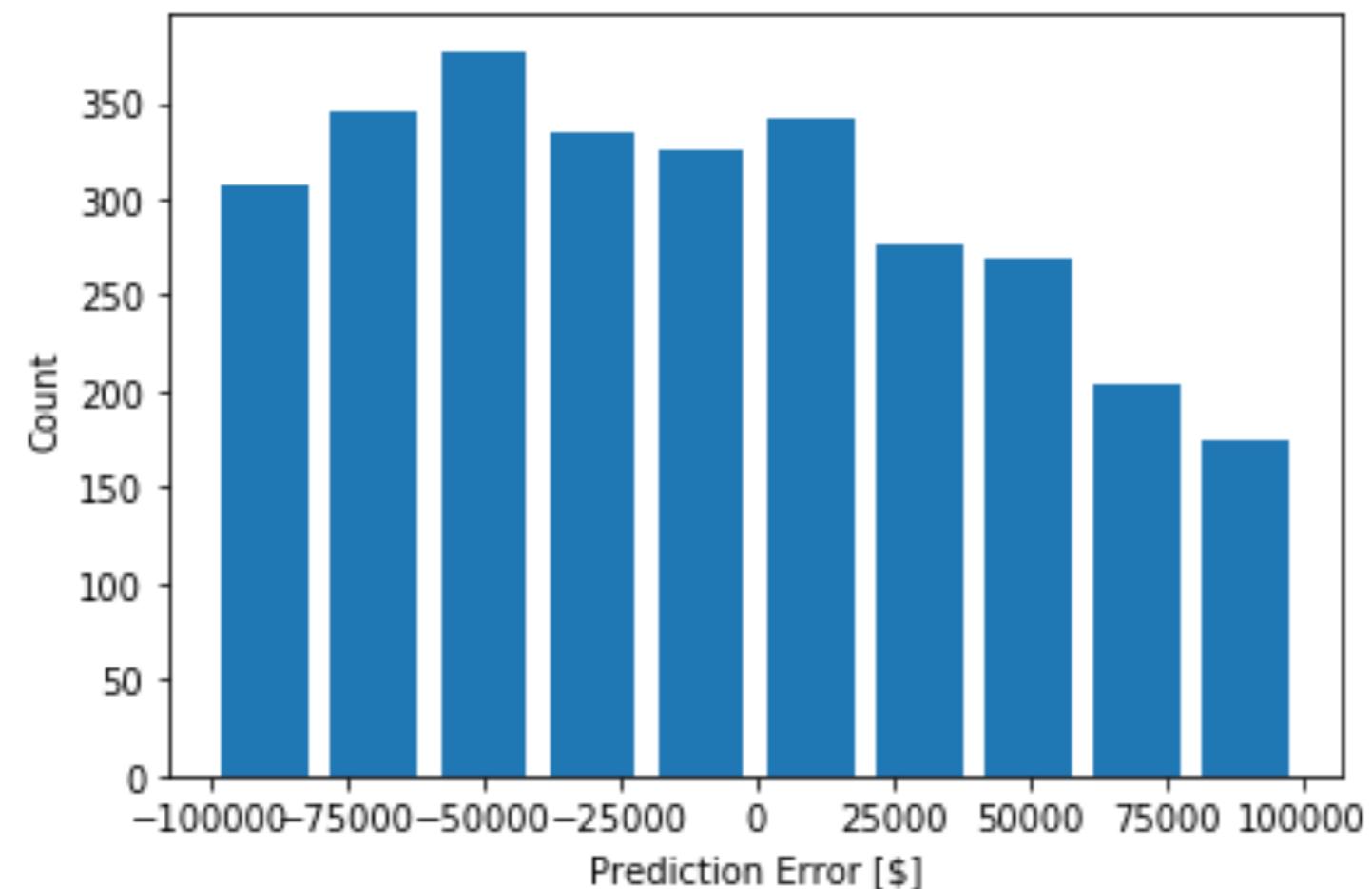
5: Predict - Analyze Error

- We are analyzing error, just displaying a error-distribution

```
%matplotlib inline
import matplotlib.pyplot as plt

predictions_df_filtered =
    predictions_df[predictions_df['error'].abs() < 100000]

plt.hist (predictions_df_filtered['error'], bins=10, rwidth=0.8)
plt.xlabel("Prediction Error [$]")
_ = plt.ylabel("Count")
```



5: Predict - Analyze Error

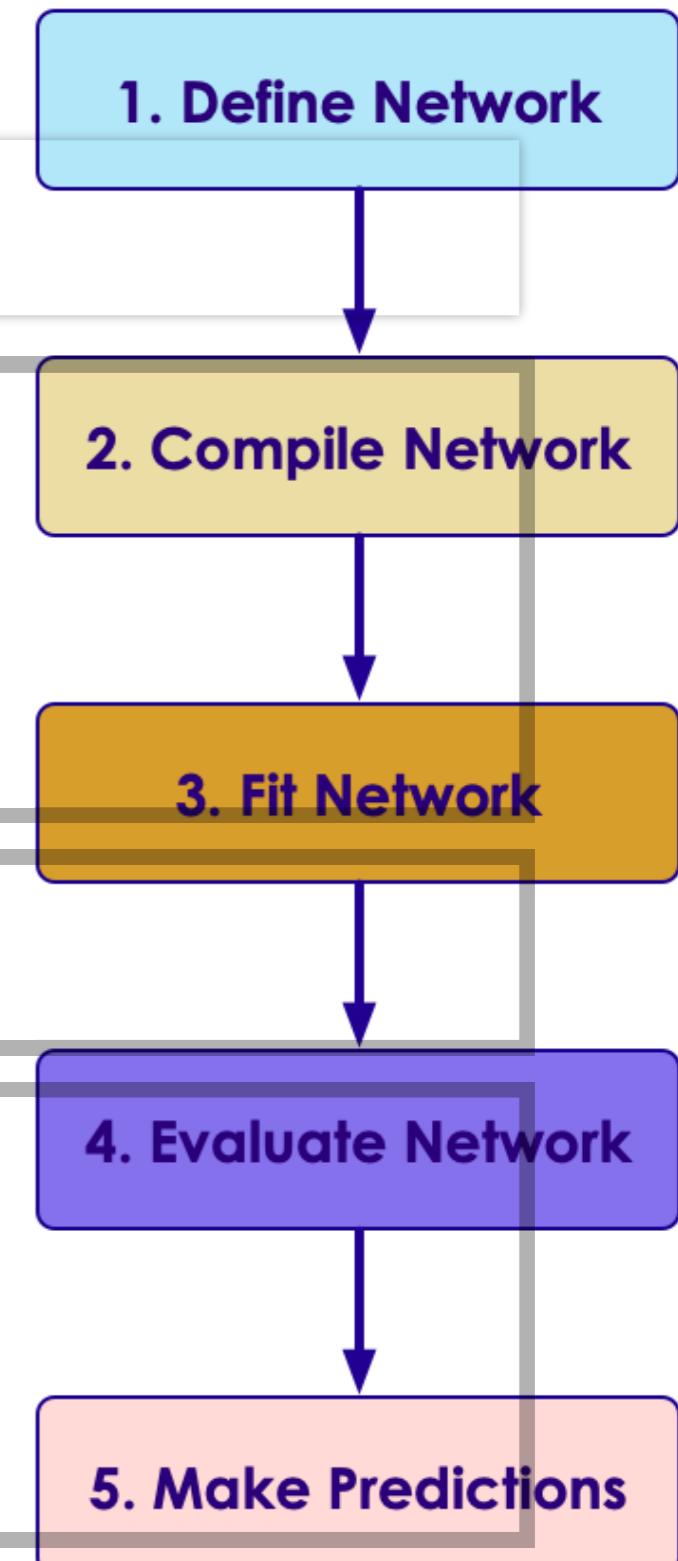
- Inspect the biggest error and smallest error. Can you explain?

```
## which house we got really wrong?  
print ("biggest error: ")  
predictions_df.loc[predictions_df['error'].abs().idxmax()]
```

```
biggest error :  
Bedrooms           6.00  
Bathrooms          6.50  
SqFtTotLiving     7,560.00  
SqFtLot            44,000.00  
actual_price       11,000,000.00  
predicted_price    2,582,545.50  
error              8,417,454.50
```

```
## which house we are spot on?  
print ("lowest error")  
predictions_df.loc[predictions_df['error'].abs().idxmin()]
```

```
lowest error  
Bedrooms           3.00  
Bathrooms          2.00  
SqFtTotLiving     1,310.00  
SqFtLot            5,040.00  
actual_price       385,000.00  
predicted_price    385,004.03  
error              -4.03
```



Lab: Regressions Using NN

- **Overview:**

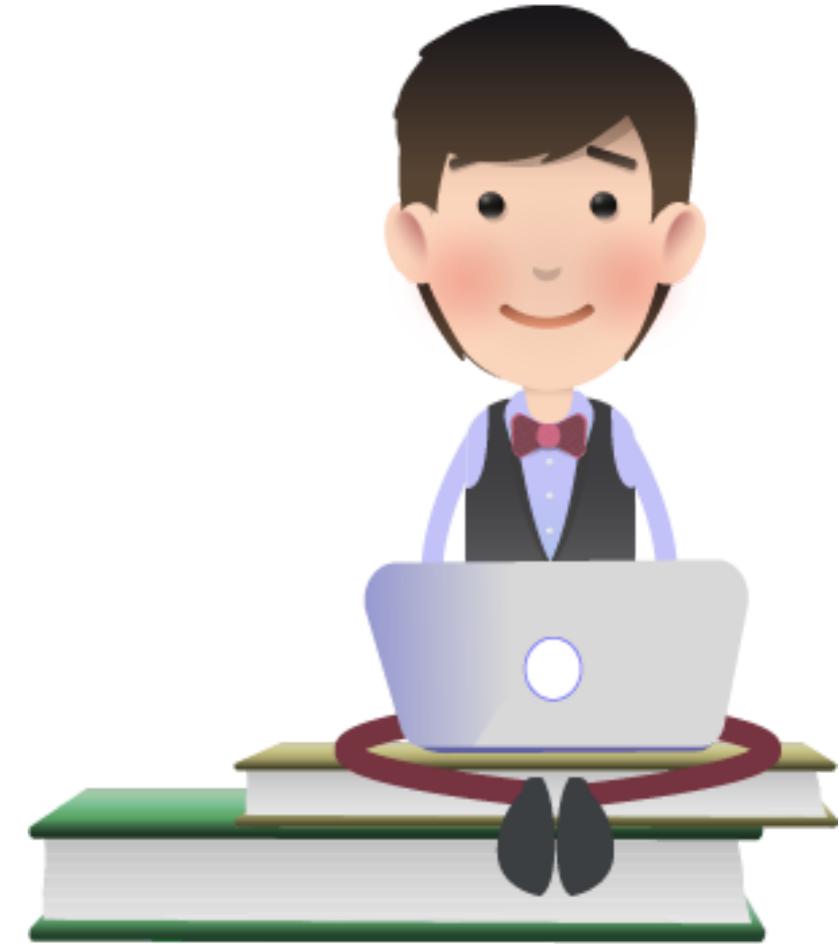
- Solve regression problems with Tensorflow

- **Approximate run time:**

- 40 - 60 mins

- **Instructions:**

- Please follow instructions for
 - **Regression-1:** Bill and Tips
 - **Regression-2:** House Prices



Metrics

Metrics - Console

- During training metrics are printed out on console
- Controlled by **verbose** flag
 - verbose=0: silent
 - verbose=1: progress bar
 - verbose=2: one line per epoch
- Pros:
 - Easy to understand
 - No extra setup required
- Cons:
 - Can get verbose
 - If we are doing 100s of epochs, it can clutter the program output

```
model.fit (x_test, y_test, verbose=1)
```

```
Train on 96 samples, validate on 24 samples
```

```
Epoch 1/100 [=====] - loss: 2.1204 - accuracy: 0.9023
```

```
...
```

```
...
```

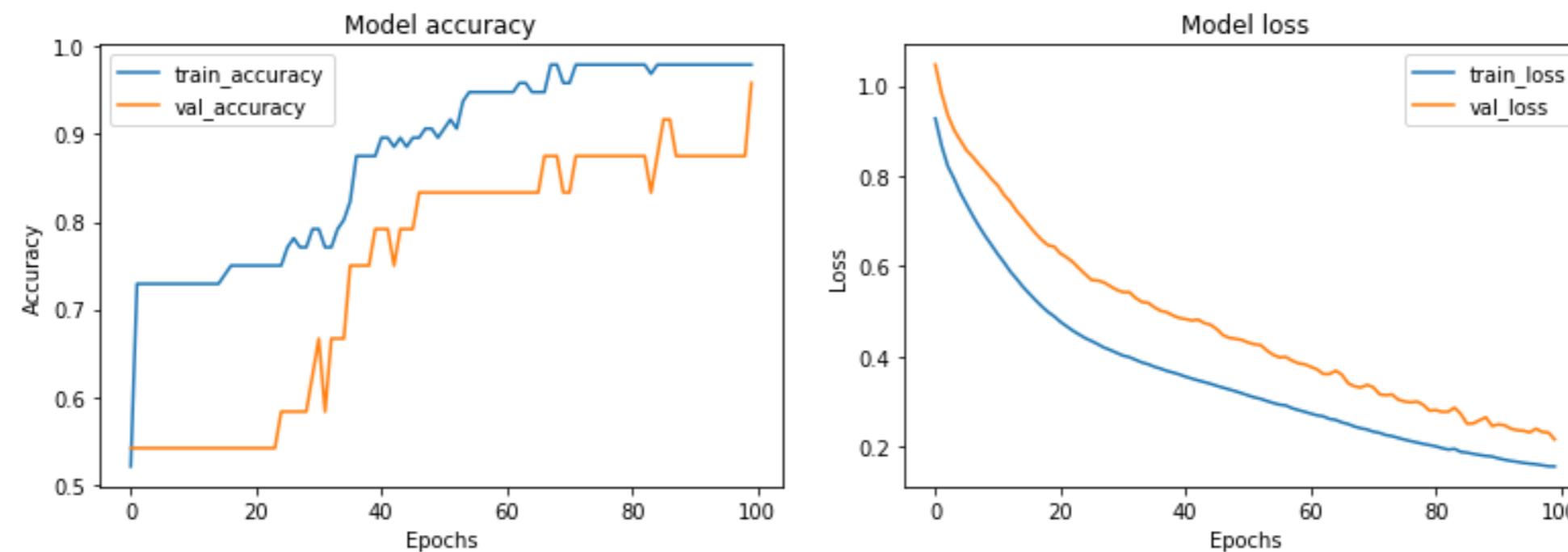
```
Epoch 100/100 [=====] - loss: 0.2375 - accuracy: 0.9583
```

Metrics - History

- The **fit()** method on a Keras Model returns a **History** object.
- The **History.history** attribute is a dictionary recording training loss values and metrics values at successive epochs

```
# training
history = model.fit (x_test, y_test, verbose=1)

# plot
plt.plot(history.history['acc'])
if 'val_acc' in history.history:
    plt.plot(history.history['val_acc'])
# ...
plt.show()
```



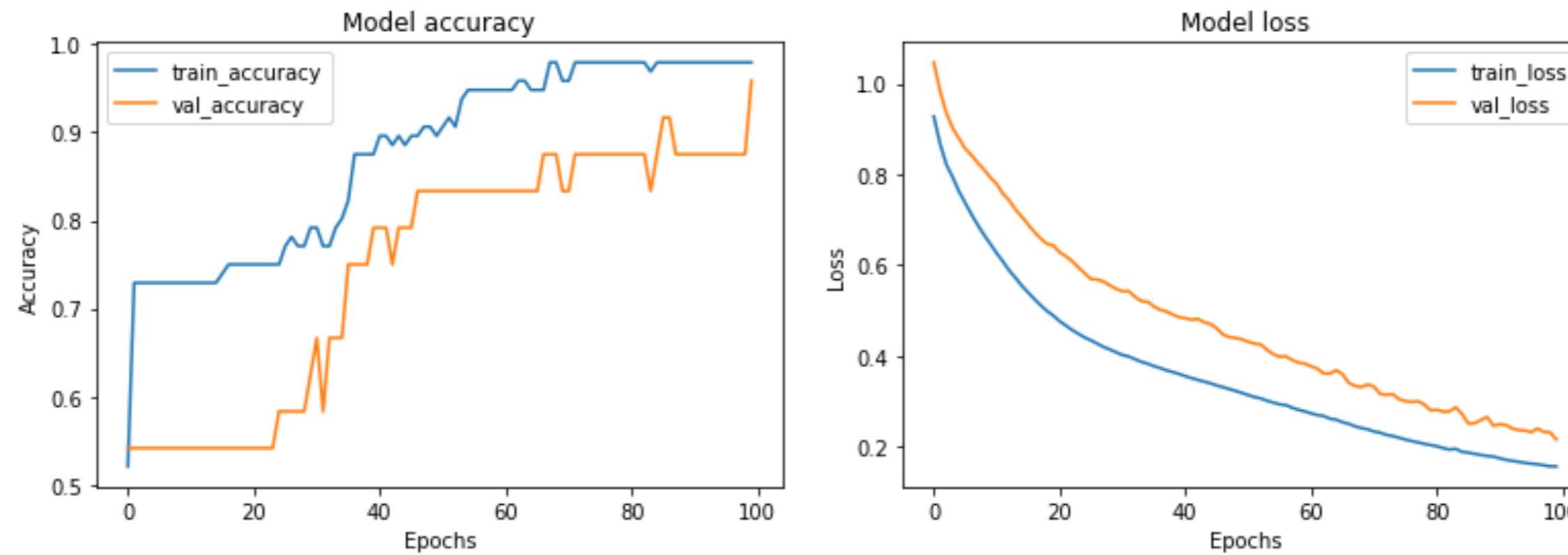
Metrics - History

- Pros:

- Better visualization than plain text
- Just takes a few extra lines of code

- Cons:

- History object available **only after the training is complete**
- So we won't



TensorBoard

TensorBoard

- TensorBoard is a tool for visualizing machine learning
- It is part of TensorFlow library
- TB features:
 - Tracking and visualizing metrics such as loss and accuracy
 - Visualizing the model graph (ops and layers)
 - Viewing histograms of weights, biases, or other tensors as they change over time
 - Projecting embeddings to a lower dimensional space
 - Displaying images, text, and audio data
 - Profiling TensorFlow programs
- See next slide for animation

TensorBoard

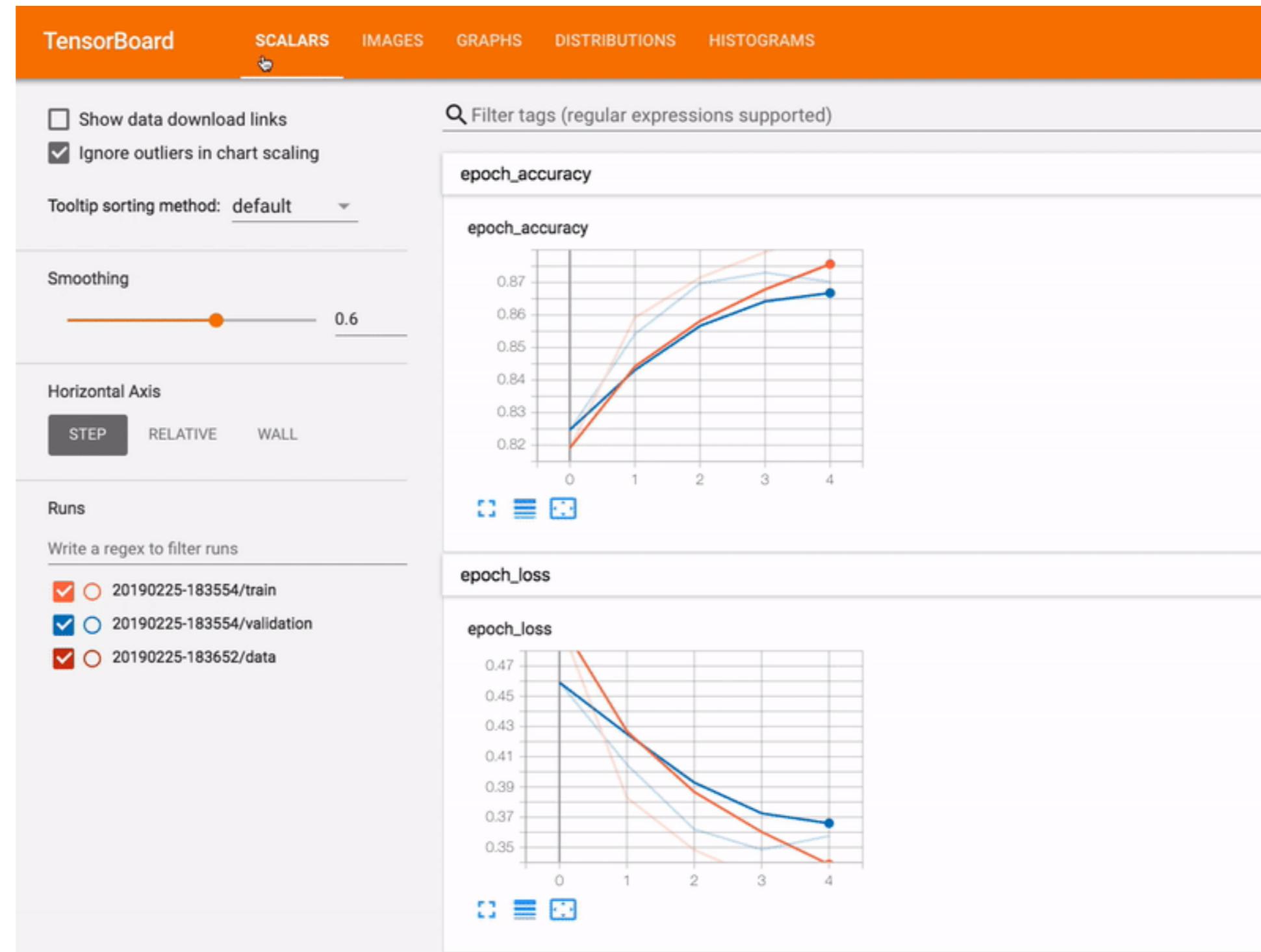
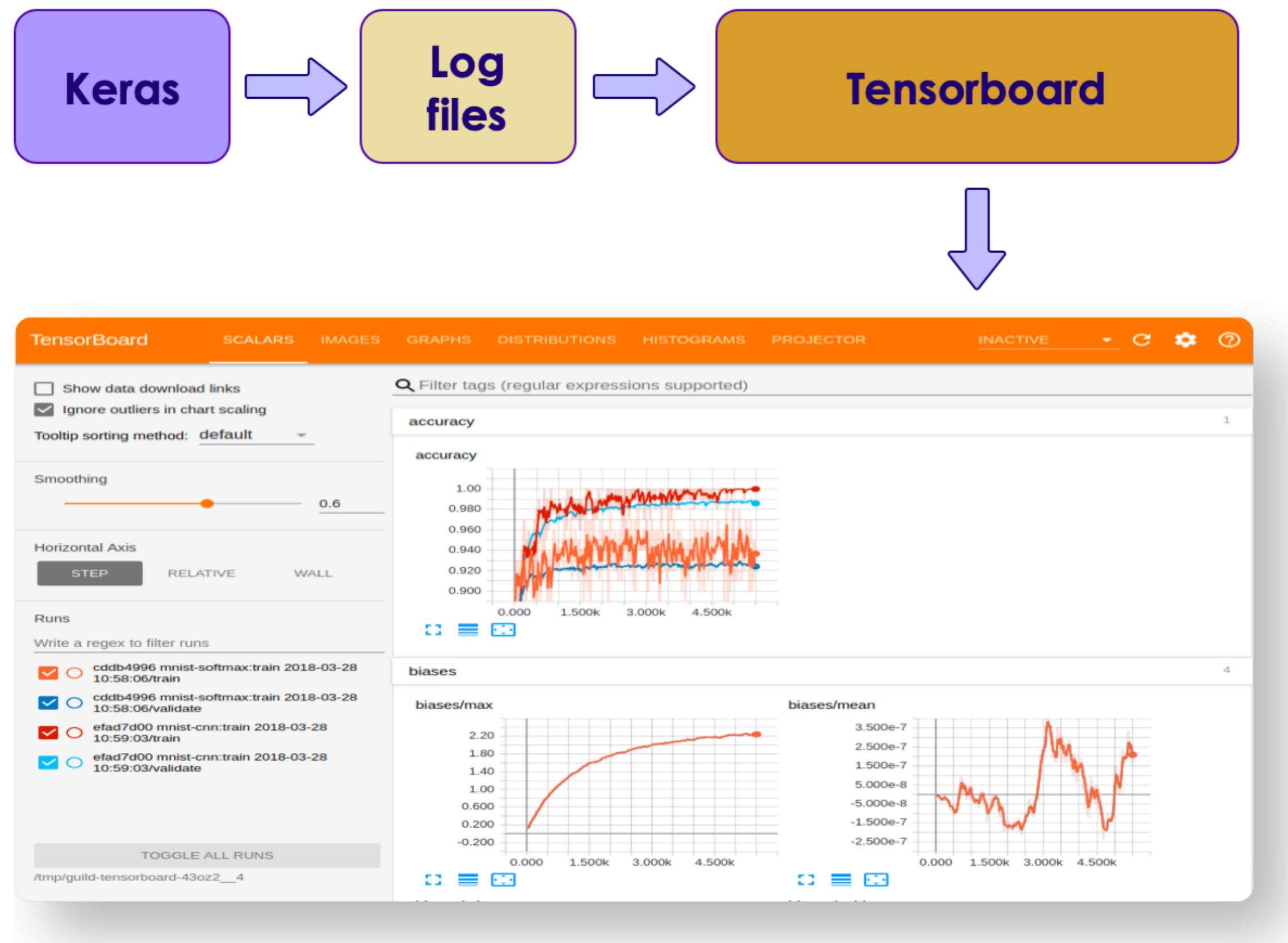


Image source

Keras and TensorBoard



Setting up TensorBoard

```
## Step 1: Run Tensorboard app - it will be monitoring a logs directory
$ tensorboard --logdir=/tmp/tensorboard-logs
```

```
## Step 2: Setup Tensorboard in our application
import datetime
import os

app_name = 'classification-iris-1' # you can change this, if you like

tb_top_level_dir= '/tmp/tensorboard-logs' # this is the top level log dir

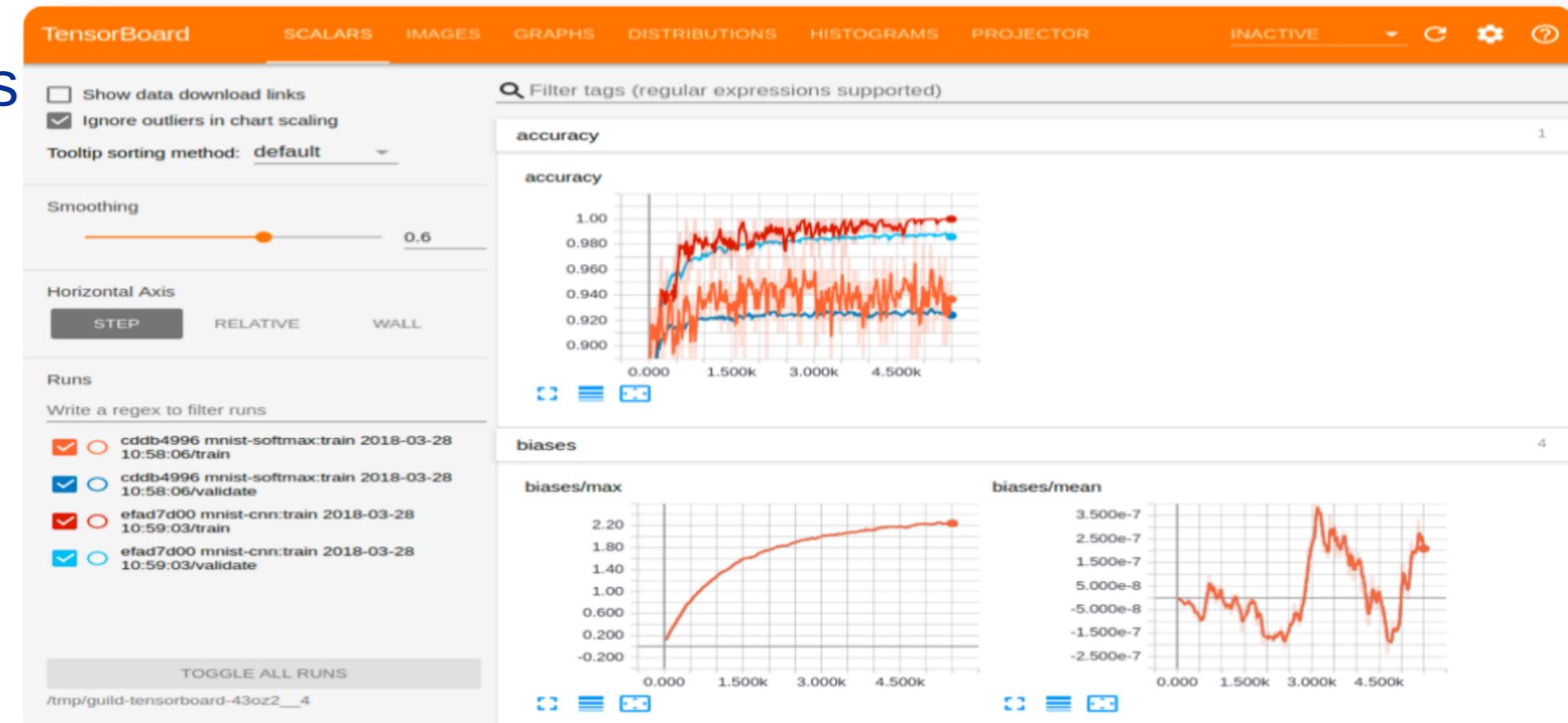
# Create an 'app dir' within logdir
tensorboard_logs_dir= os.path.join (tb_top_level_dir, app_name,
                                    datetime.datetime.now().strftime("%Y-%m-%d--%H-%M-%S"))
print ("Saving TB logs to : " , tensorboard_logs_dir)
# Saving TB logs to : /tmp/tensorboard-logs/classification-iris-1/2020-02-05--18-47-10
```

```
## Step 3: provide tb_callback function during training
tb_callback = tf.keras.callbacks.TensorBoard(log_dir=tensorboard_logs_dir, histogram_freq=1)

model.fit(x, y, epochs=5,
          callbacks=[tb_callback]) ## <-- here is the linkup
```

TensorBoard Log Directory

- Each application will write their own directory within logs directory
- It is also recommended, that each run is its own directory; so we can compare runs
- Each run is timestamped
- Here we see two runs of '**app1**'; check timestamps



```
tensorboard-logs/
├── app1__2020-01-01--12-30-10
├── app1__2020-01-01--12-32-30
└── app2__2020-01-01--12-33-00
```

Lab: Using TensorBoard

- **Overview:**

- Incorporate TensorBoard into NN

- **Approximate run time:**

- 20 mins

- **Instructions:**

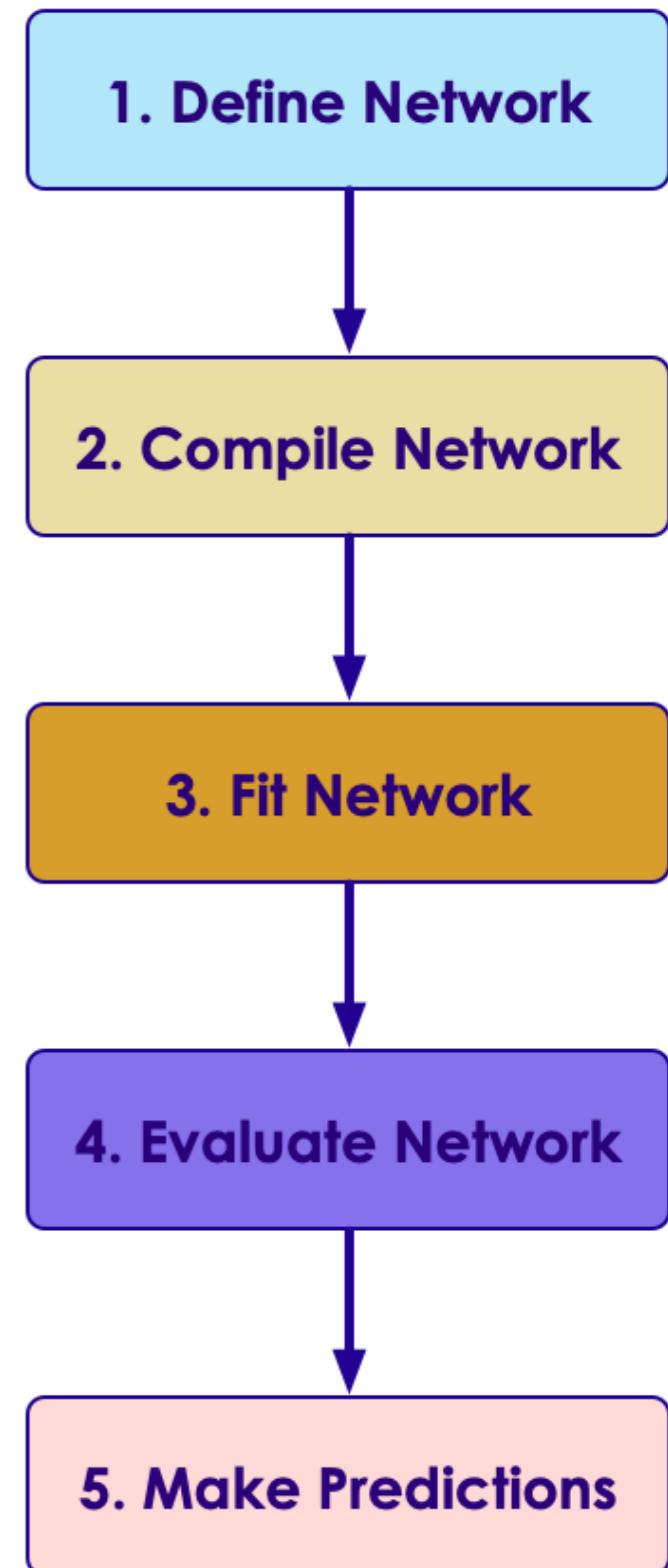
- **Metrics-2:** TensorBoard



Classifications with Neural Networks

Workflow

- Here is a typical workflow
- Step 1 - Define the network
 - Step 1A - Use a model class from **keras.models**
 - Step 1B - Stack layers using the **.add()** method
- Step 2 - Configure the learning process using the **compile()** method
- Step 3 - Train the model on the train dataset using the **.fit()** method
- Step 4 - Evaluate the network
- Step 5 - Predict



Let's use IRIS Dataset

- IRIS is a very simple dataset (a ML classic)
- 4 inputs (a,b,c,d) - representing dimensions of the flower, like 'petal width'
- and 3 output classes label (1,2,3)
- Total samples: 150
- Well balanced, each label (1,2,3) has 50 samples each



a	b	c	d	label
6.4	2.8	5.6	2.2	3
5.0	2.3	3.3	1.0	2
4.9	3.1	1.5	0.1	1

Step 0 - Data Prep

```
### --- read input ---
iris = pd.read_csv('iris.csv')
x = iris[['SepalLengthCm', 'SepalwidthCm', 'PetalLengthCm', 'PetalwidthCm']]
y = iris[['Species']]
# SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
# 0            5.1           3.5           1.4           0.2
# 4            5.0           3.6           1.4           0.2
# -----
#      Species
# 0  Iris-setosa
# 1  Iris-virginica

# ---- pre processing ----
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
y1 = encoder.fit_transform(y.values) ## need y.values which is an array
# [0 0 0 ... 1 1 1 ... 2 2 2]

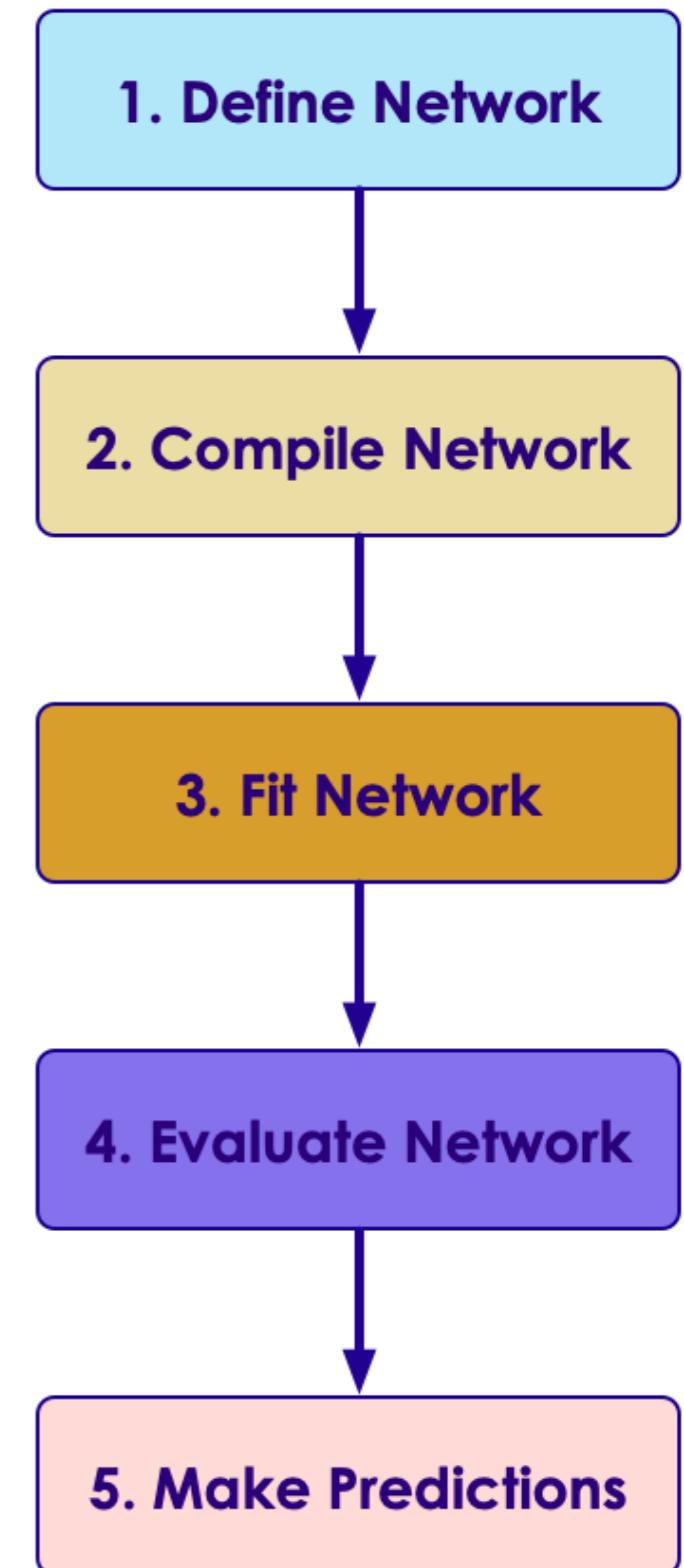
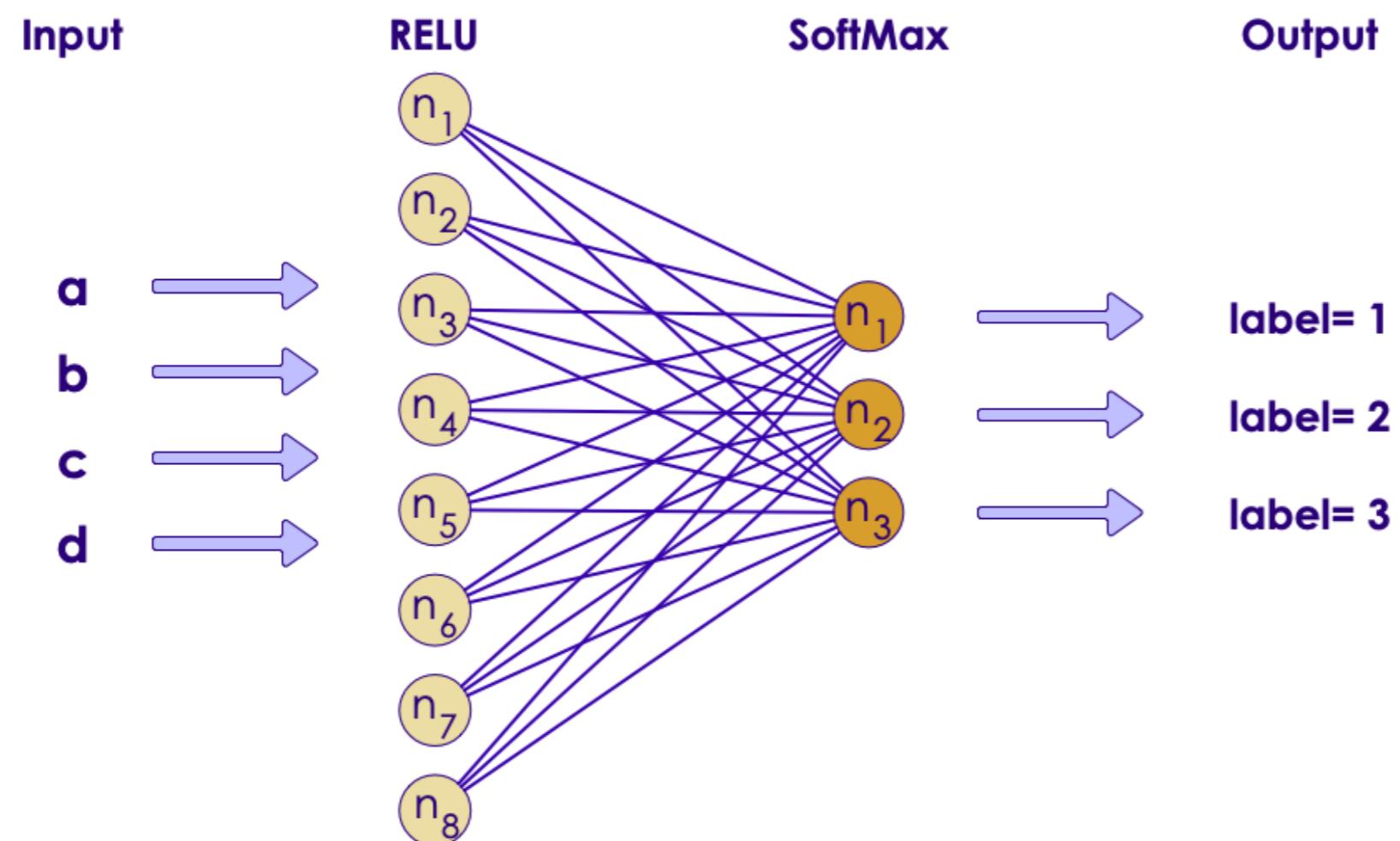
# --- train test split ---
from sklearn.model_selection import train_test_split
x_train,x_test, y_train,y_test = train_test_split(x,y1,test_size=0.2,random_state=0)

print ("x_train.shape : ", x_train.shape)
print ("y_train.shape : ", y_train.shape)
print ("x_test.shape : ", x_test.shape)
print ("y_test.shape : ", y_test.shape)

# x_train.shape : (120, 4)
# y_train.shape : (120,)
# x_test.shape : (30, 4)
# y_test.shape : (30,)
```

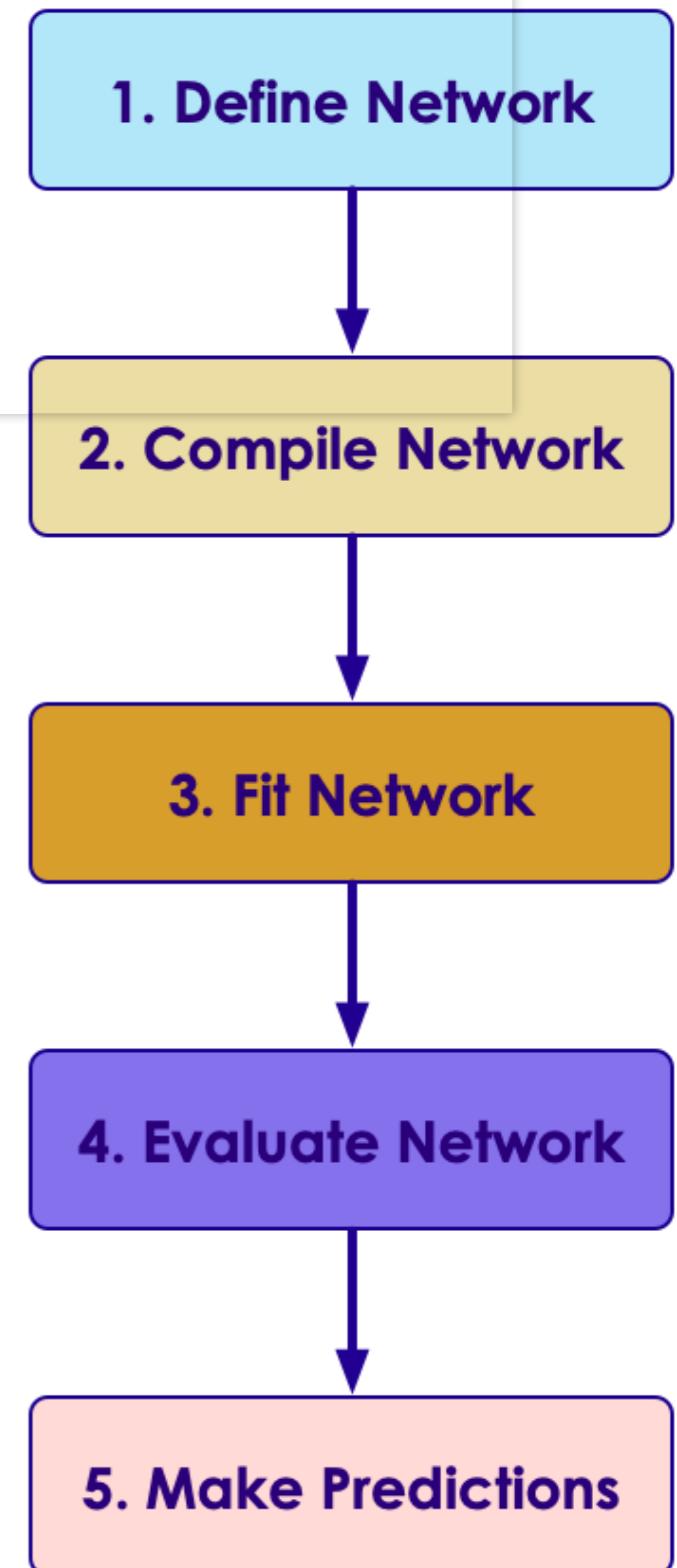
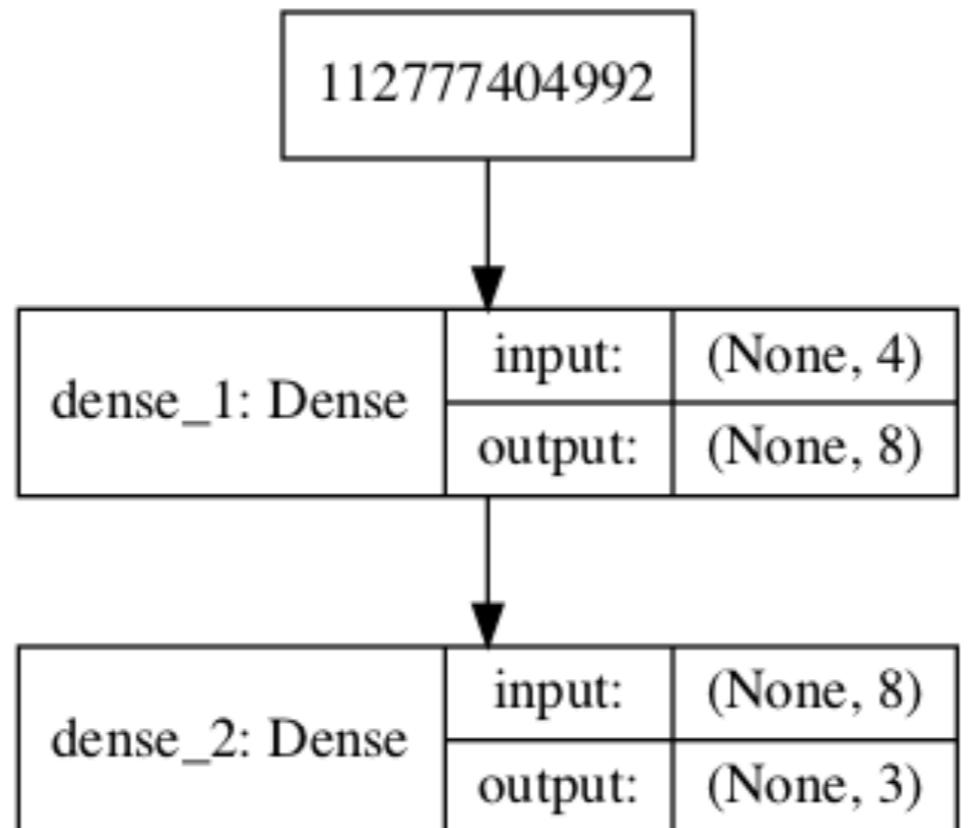
1: Network Design

- Input layer will have **4 neurons** to match input dimensions (a,b,c,d)
- Hidden layer will have **8 neurons**, with **ReLU** activation
- Output layer will have **3 neurons** with **SoftMax** activation



Step 1: Define Model

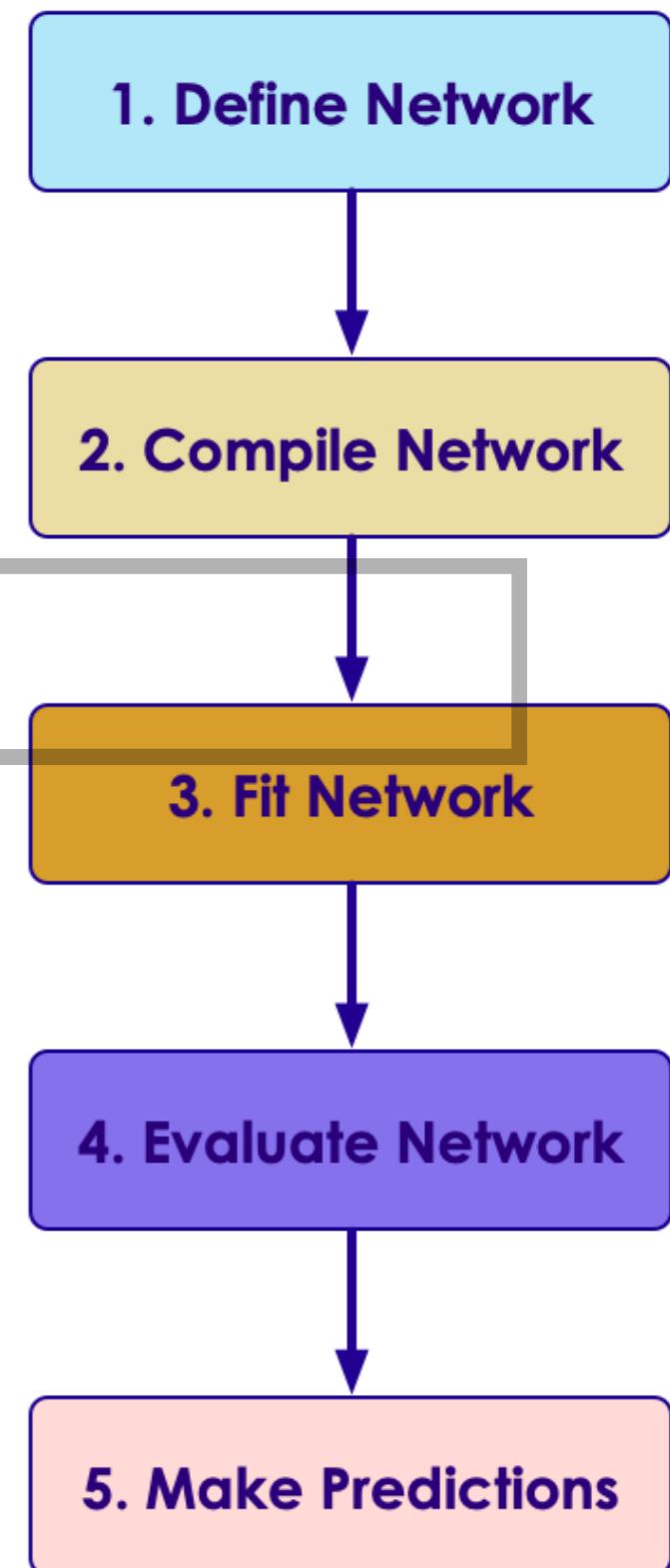
```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense  
  
model = Sequential()  
model.add(Dense(8, input_dim=4, activation='relu'))  
model.add(Dense(3, activation='softmax'))  
  
keras.utils.plot_model(model, to_file='model.png', show_shapes=True)
```



Step 2: Compile the Model

- We are using **sparse_categorical_crossentropy** for loss, because this is a **multi-class classifier**
- We are tracking metric **accuracy**
- **Adam** is a pretty good optimizer, that can self-adjust parameters as they learn

```
model.compile( optimizer = 'adam',
               loss = 'categorical_crossentropy',
               metrics = ['accuracy'] )
```



Step 3: Train the Network

- We train on **training_data (x_train and y_train)**
- Output may look like below; we did 100 epochs in about 10 seconds

```
## without validation
history = model.fit(x_train, y_train, epochs = 100, batch_size = 2**4)

## with validation
history = model.fit(x_train, y_train, epochs = 100, batch_size = 2**4,
validation_split=0.25)

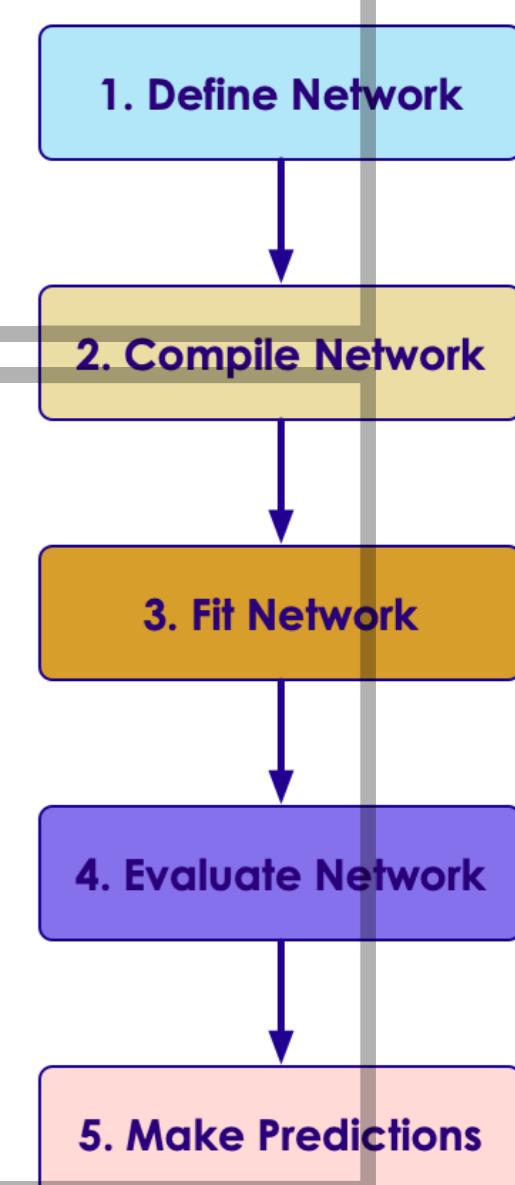
training starting ...

Train on 96 samples, validate on 24 samples

Epoch 1/100 [=====] - loss: 2.1204 - accuracy: 0.2708 -
val_loss: 1.5499 - val_accuracy: 0.4583
...
...
Epoch 100/100 [=====] - loss: 0.2375 - accuracy: 0.9583 -
val_loss: 0.2986 - val_accuracy: 0.9167

training done.

CPU times: user 14.3 s, sys: 18.9 s, total: 33.3 s
Wall time: 10.1 s
```

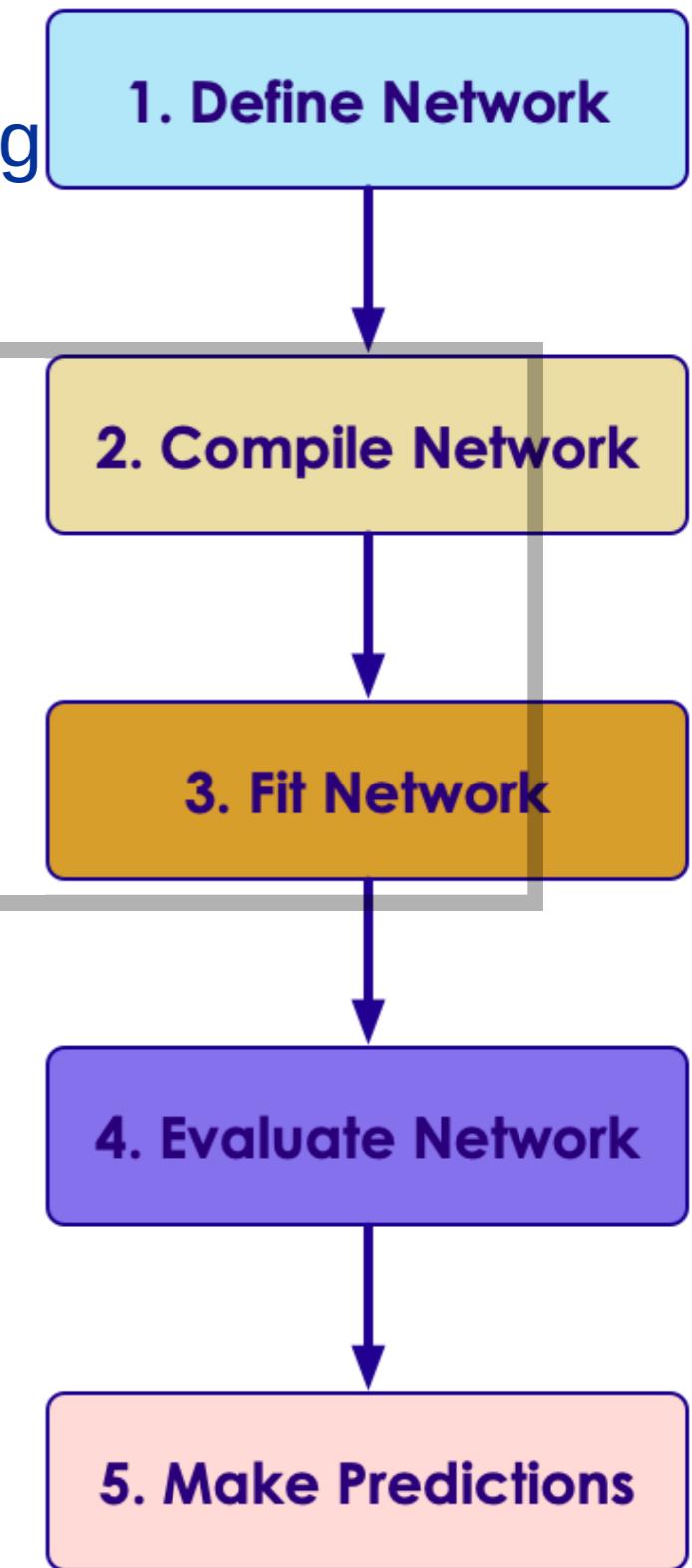
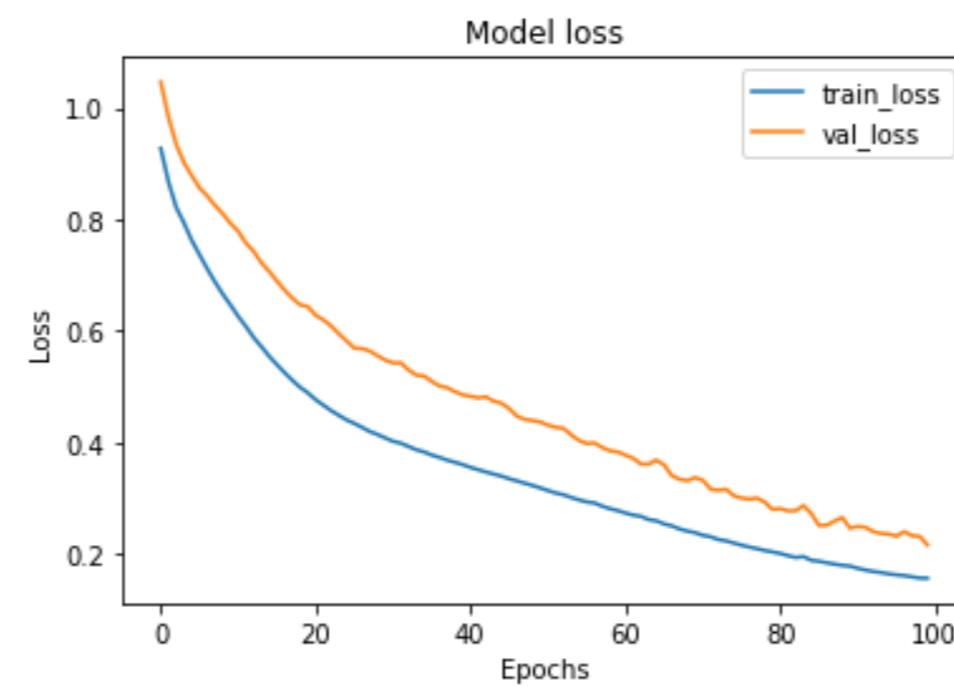
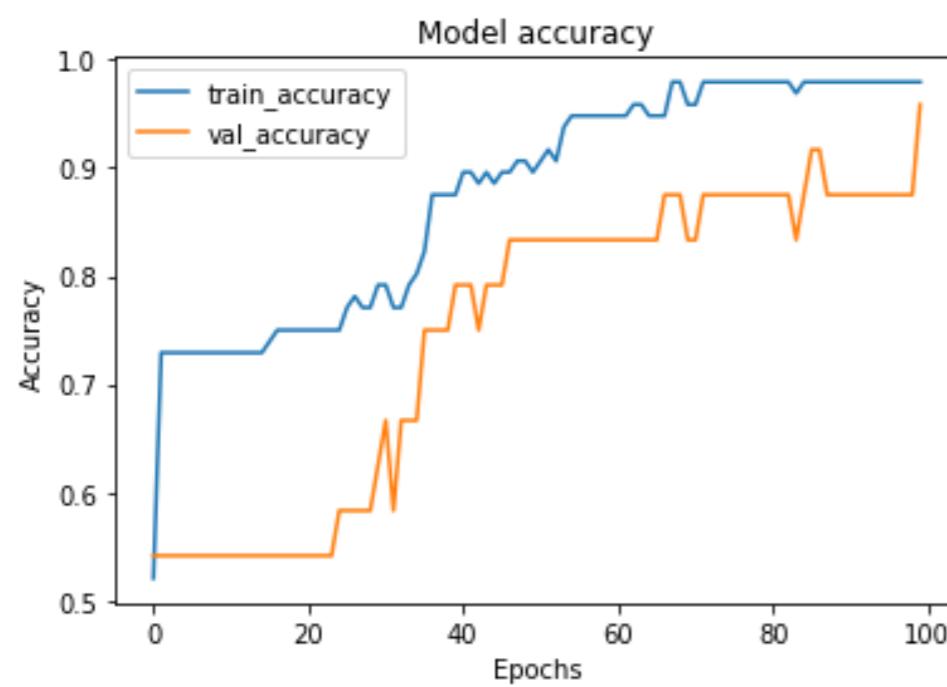


Step 3.5: Visualize Training History

- The `fit()` method on a Keras Model returns a `History` object.
- The `History.history` attribute is a dictionary recording training loss values and metrics values at successive epochs

```
import matplotlib.pyplot as plt

plt.plot(history.history['acc'])
if 'val_acc' in history.history:
    plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



Step 4: Evaluate network

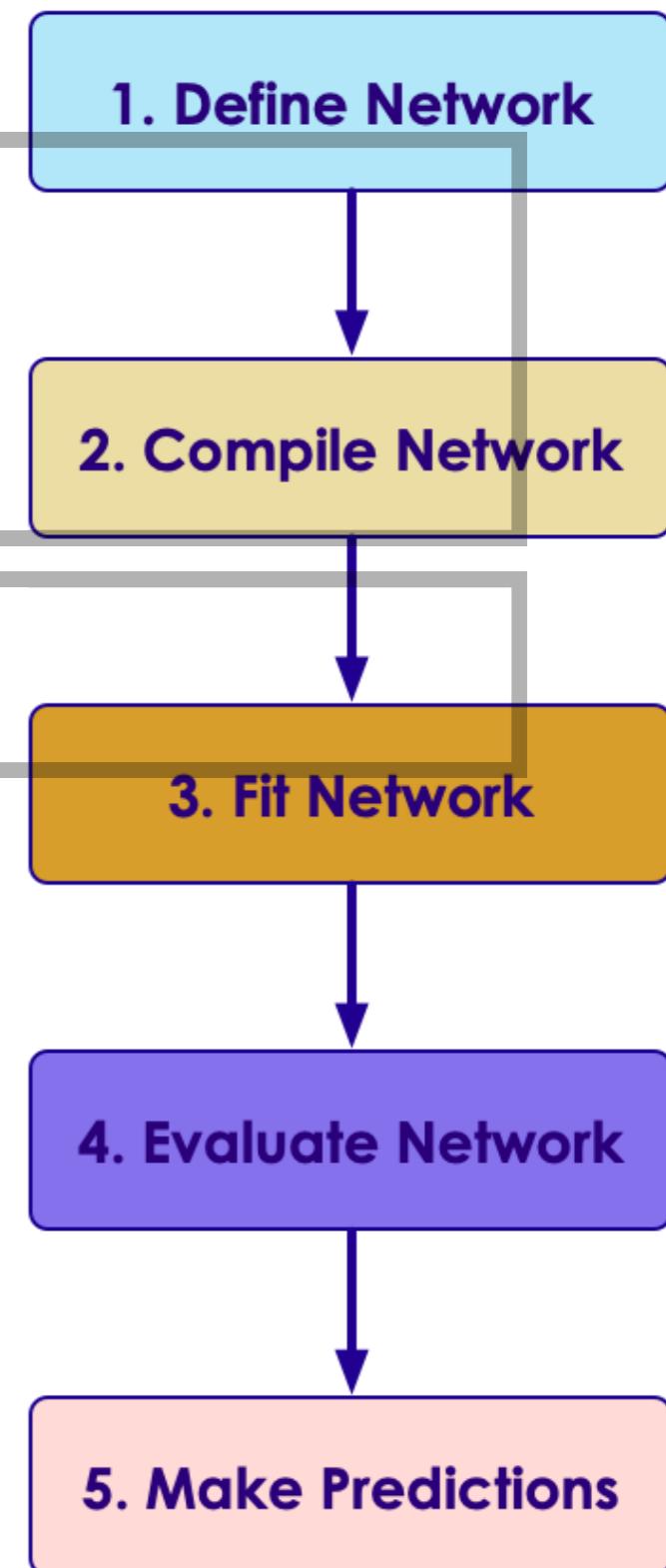
- **model.evaluate** is returning a few metrics, displayed below

```
metric_names = model.metrics_names
print ("model metrics : " , metric_names)

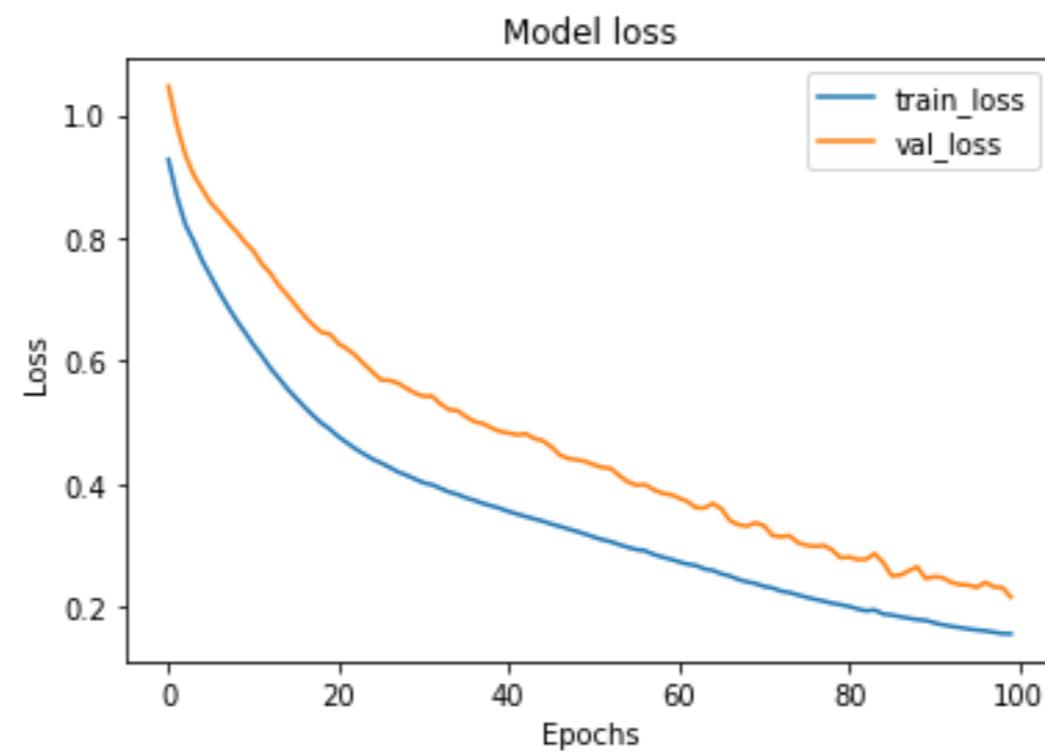
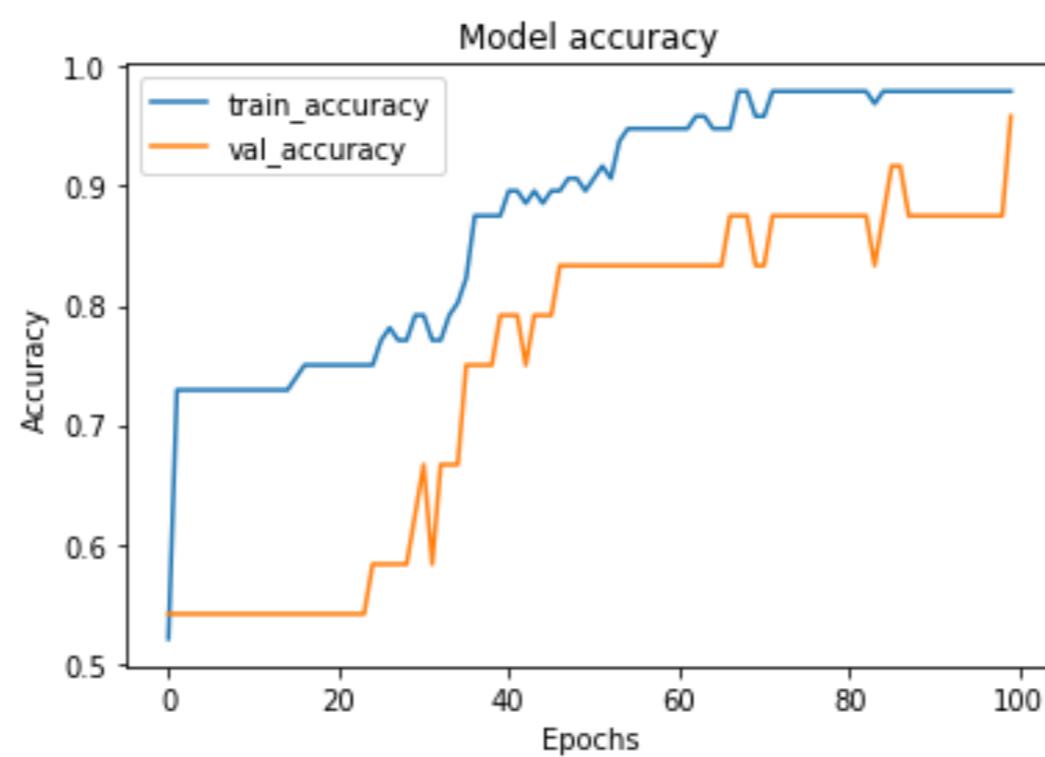
metrics = model.evaluate(x_test, y_test, verbose=0)

for idx, metric in enumerate(metric_names):
    print ("Metric : {} = {:.2f}".format (metric_names[idx], metrics[idx]))
```

```
model metrics : ['loss', 'accuracy']
Metric : loss = 0.27
Metric : accuracy = 0.93
```



Step 4.5: Training Accuracy & Loss



Step 4.5: Visualizing Training History With Tensorboard

- Tensorboard allows us to view training metrics live

```
## Step 1: Setup Tensorboard
import datetime
import os

app_name = 'classification-iris-1' # you can change this, if you like

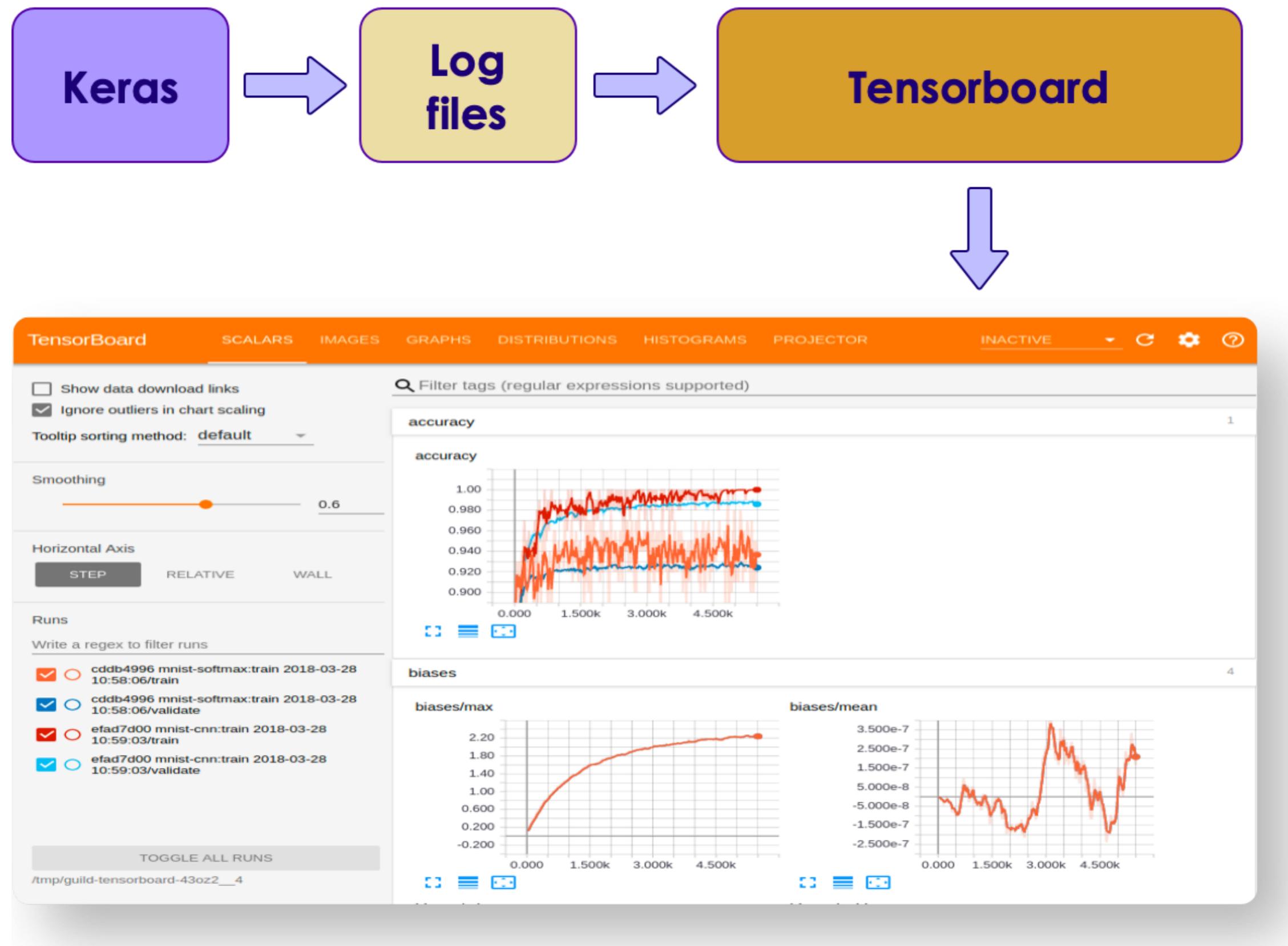
tb_top_level_dir= '/tmp/tensorboard-logs'
tensorboard_logs_dir= os.path.join (tb_top_level_dir, app_name,
                                    datetime.datetime.now().strftime("%Y-%m-%d--%H-%M-%S"))
print ("Saving TB logs to : " , tensorboard_logs_dir)
# Saving TB logs to : /tmp/tensorboard-logs/classification-iris-1/2020-02-05--18-47-10

tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=tensorboard_logs_dir, histogram_freq=1)
```

```
## Step 2: Run Tensorboard
$ tensorboard --logdir=/tmp/tensorboard-logs
```

```
# Step 3: Use tensorboard callback during training
history = model.fit( x_train, y_train,
                      epochs=epochs, validation_split = 0.2, verbose=1,
                      callbacks=[tensorboard_callback]) # <-- here is the TB callback
```

TensorBoard Visualization

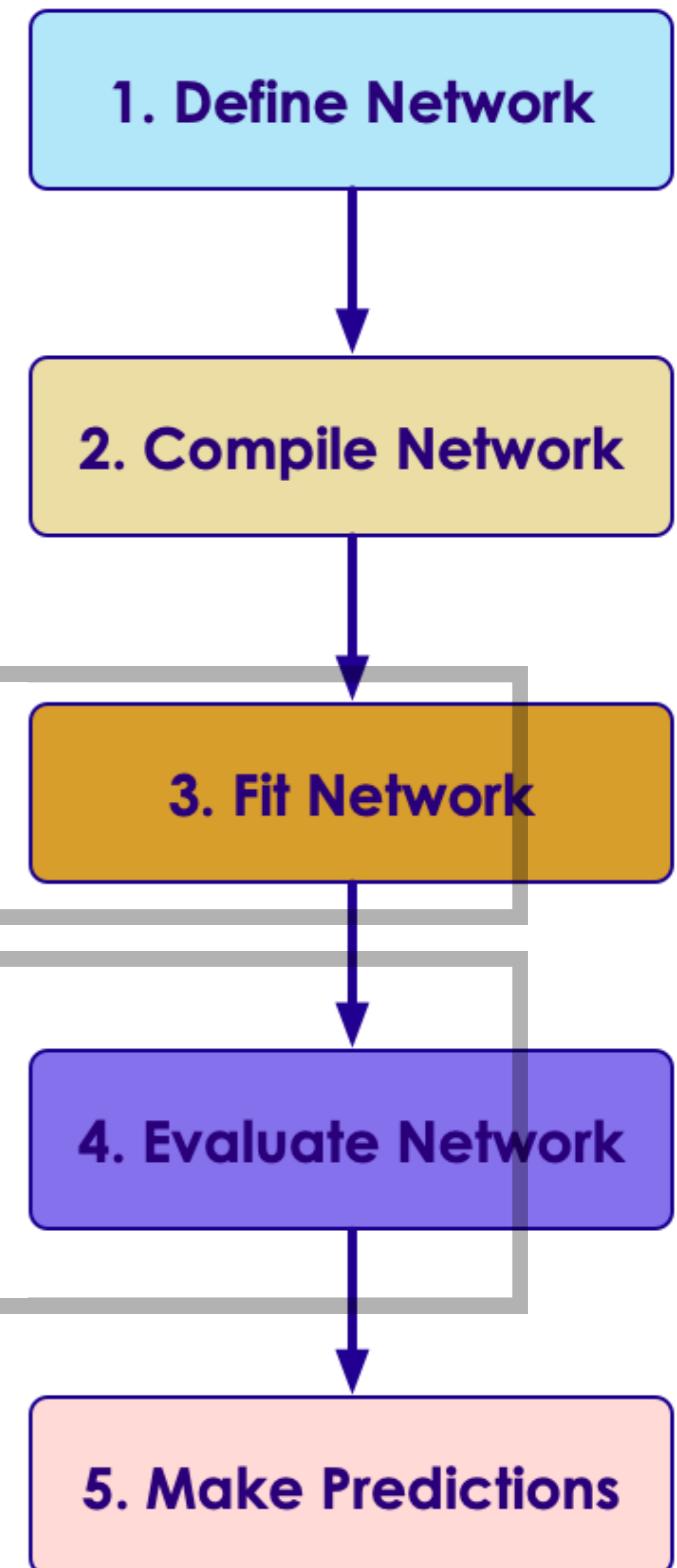


Step 5: Predict

- `model.predict` is run on `x_test`
- It returns `predictions` which is a `softmax` output
- Softmax output is an array of probabilities for each output class (in our case 3);
They should add up to 1.0 (total probability)

```
np.set_printoptions(formatter={'float': '{: 0.3f}'.format})  
  
predictions = model.predict(x_test)  
predictions
```

```
array([[ 0.002,  0.249,  0.749], # winner class-3 (0.749)  
       [ 0.023,  0.602,  0.375], # winner class-2 (0.602)  
       [ 0.987,  0.012,  0.000], # winner class-1 (0.987)  
       [ 0.000,  0.111,  0.889],  
       [ 0.965,  0.034,  0.001],  
       [ 0.000,  0.151,  0.849],
```

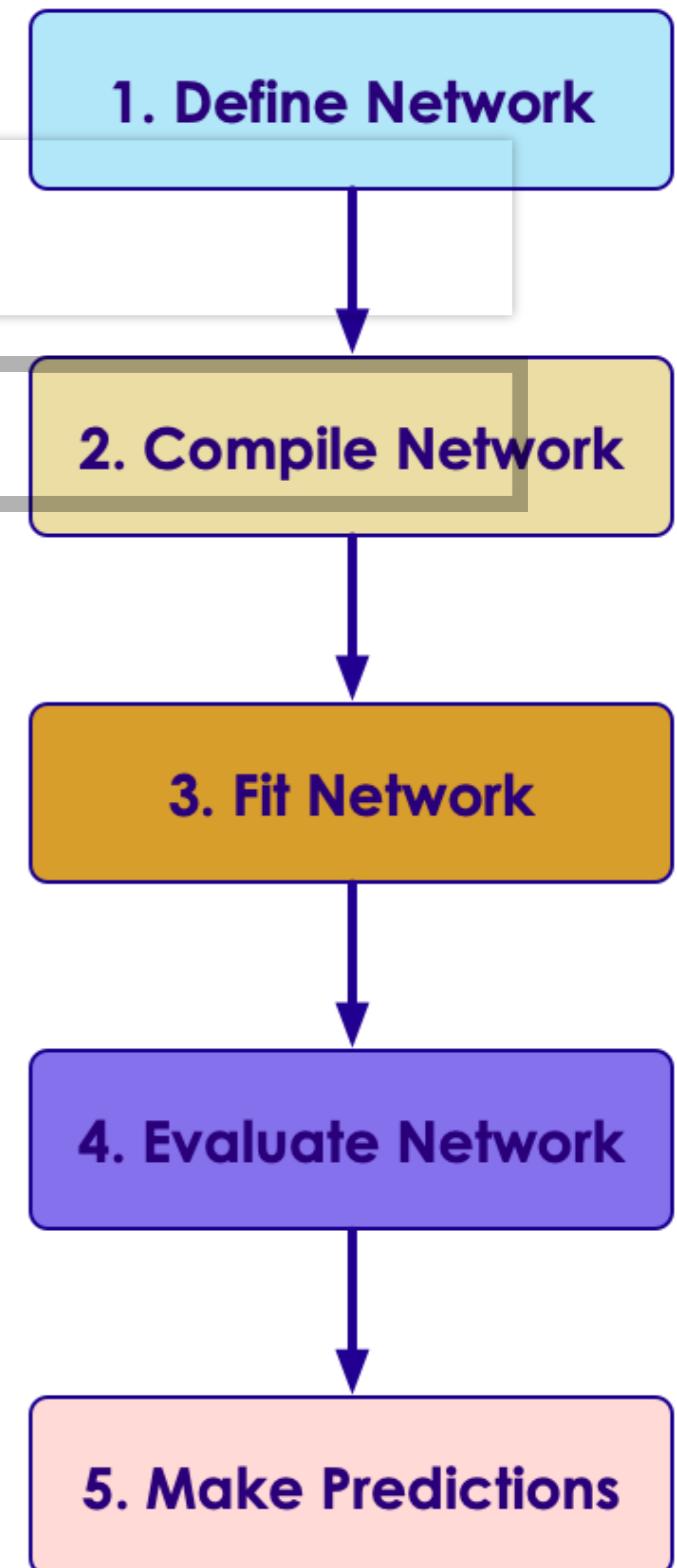


Step 5: Predict

- Here we convert softmax output actual class label

```
predictions = model.predict(x_test)
y_pred = np.argmax(predictions, axis=-1)
print ('prediction classes: ', y_pred)
```

```
array([2, 1, 0, 2, 0, 2, 0, 1, 1, 1, 2, 1, 2, 1, 2, 0, 1, 2, 0, 0, 2, 2,
       0, 0, 2, 0, 0, 1, 1, 0])
```



Evaluation - Confusion Matrix

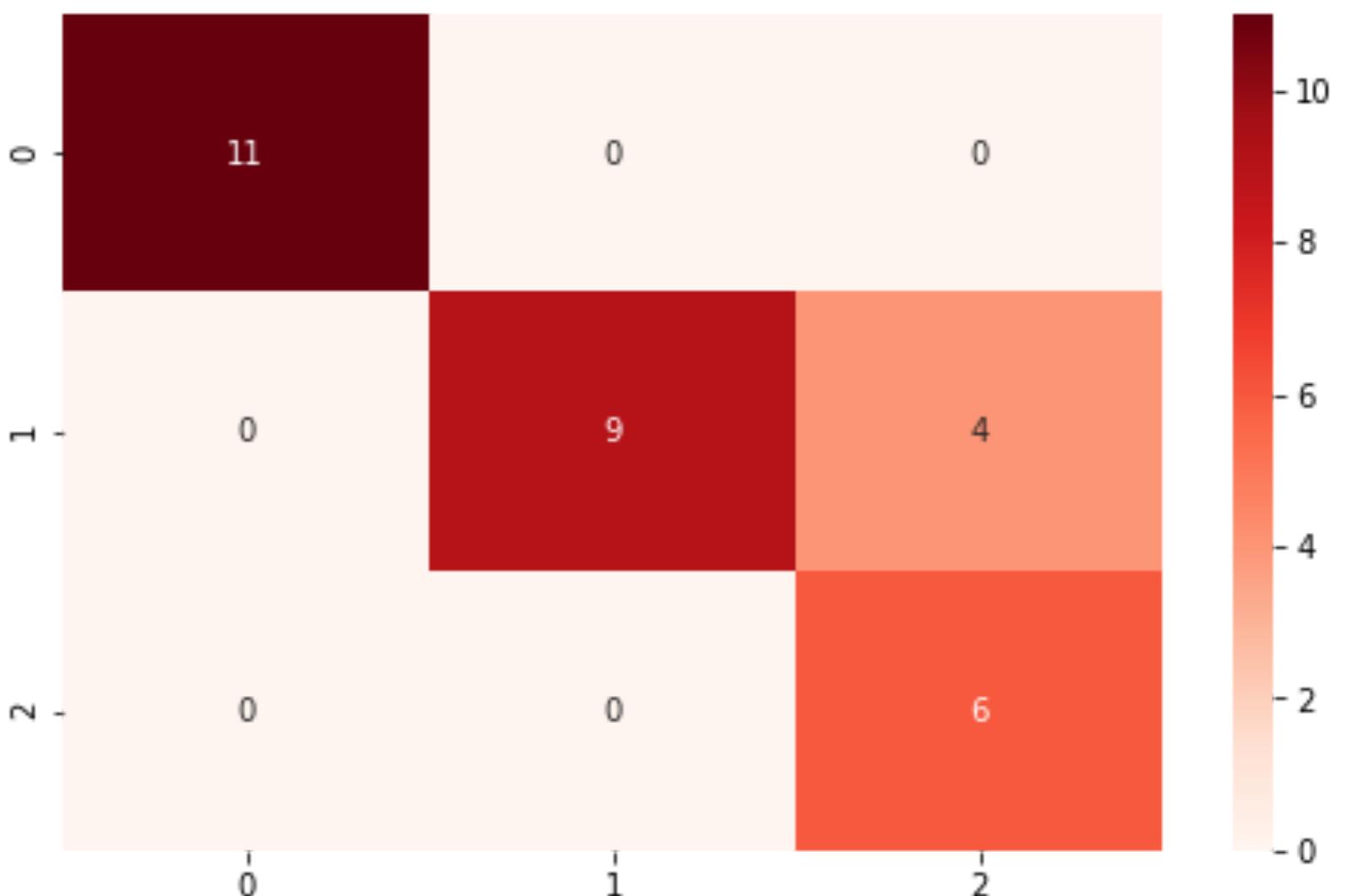
```
## plain confusion matrix

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

cm = confusion_matrix(y_test, y_pred, labels = [0,1,2])

plt.figure(figsize = (8,5))
# colormaps : cmap="YlGnBu" , cmap="Greens", cmap="Blues",   cmap="Reds"
sns.heatmap(cm, annot=True, cmap="Reds", fmt='d').plot()
```

- **To the Instructor:** You may want to cover **Confusion Matrix** from **ML Concepts**
- Can you interpret the confusion matrix?



Lab: Classifier with NN

- **Overview:**

- Build a classification network with NN

- **Depends on:**

- None

- **Runtime:**

- 40-60 mins

- **Instructions:**

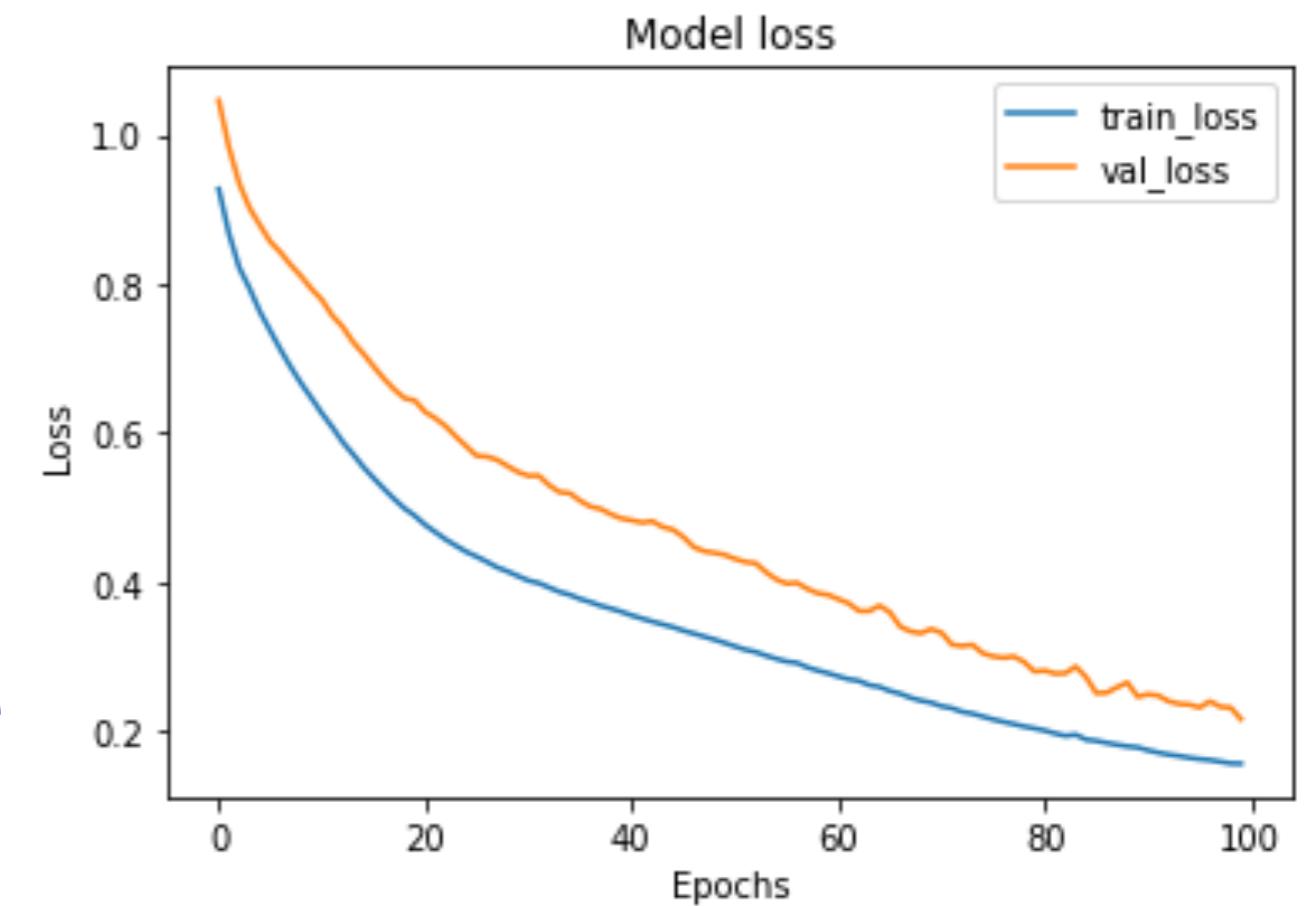
- Follow instructions for
 - **Classification-1** : IRIS
 - **Classification-2** : Propser Loan



Monitoring Training

Monitoring Training

- During training phase, network trains on **training data** and verified on **validation data**
- We can monitor the training progress via Learning Curve
- The **Learning Curve** usually includes 2 plots
 - **Train Learning Curve**: tracks how well the model is learning from training data
 - **Validation Learning Curve**: tracks how well the model is generalizing on validation data

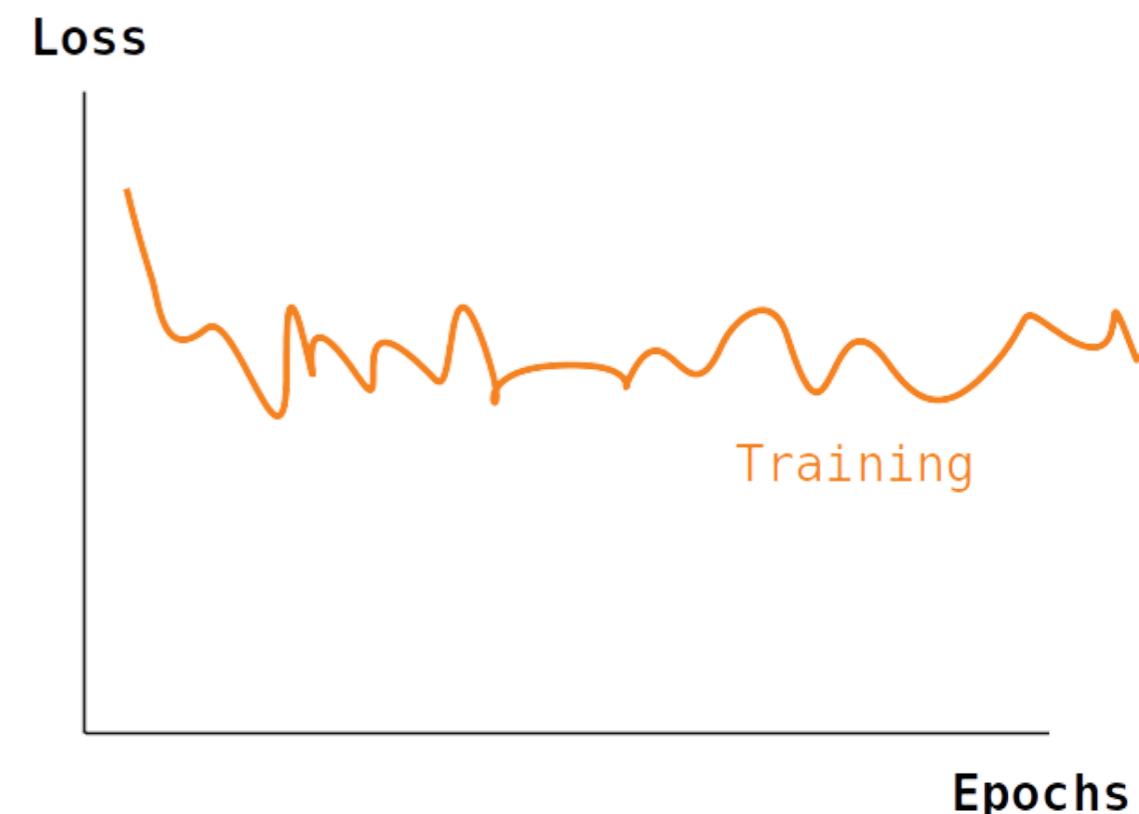
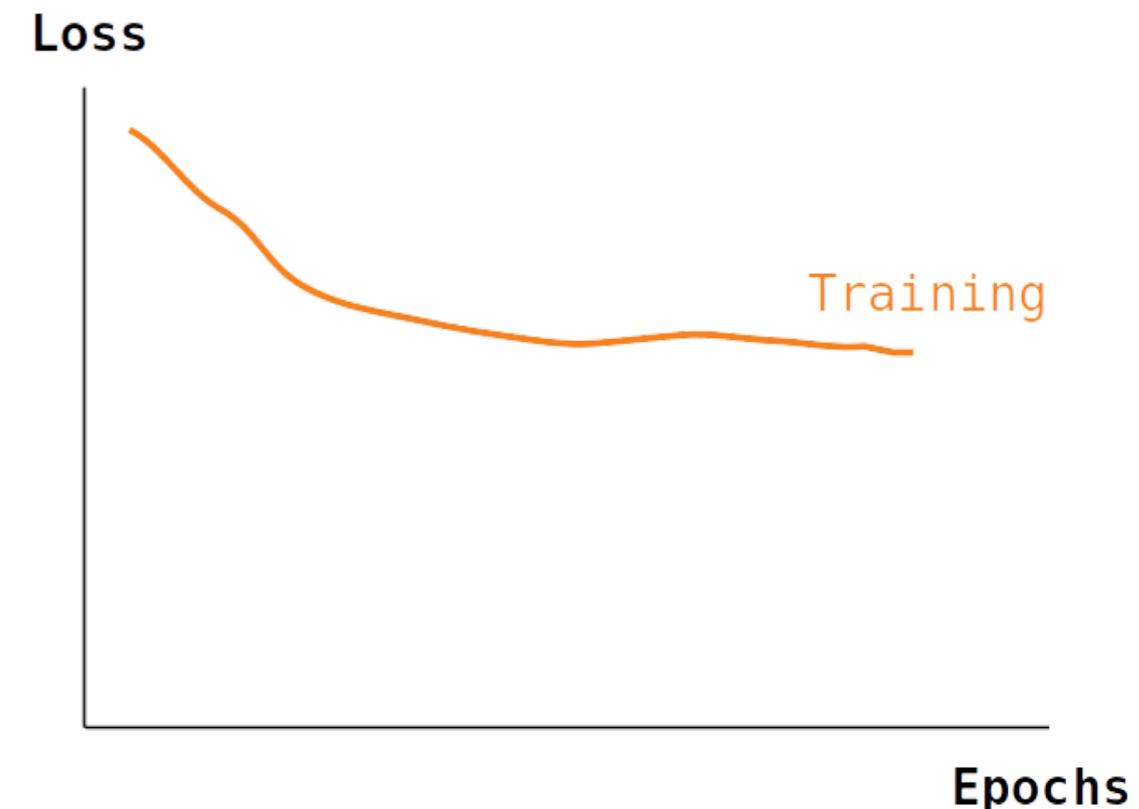


Model Overfit / Underfit / Goodfit

- During training a model **underfit**, **overfit** or **goodfit**
- **Underfit** is the model is not learning enough
- **Overfit** the model is basically memorizing data, instead of learning from it
- **Goodfit** is what we want
- **To instructor:** If appropriate, go through **ML-Concepts # Model Evaluation** section

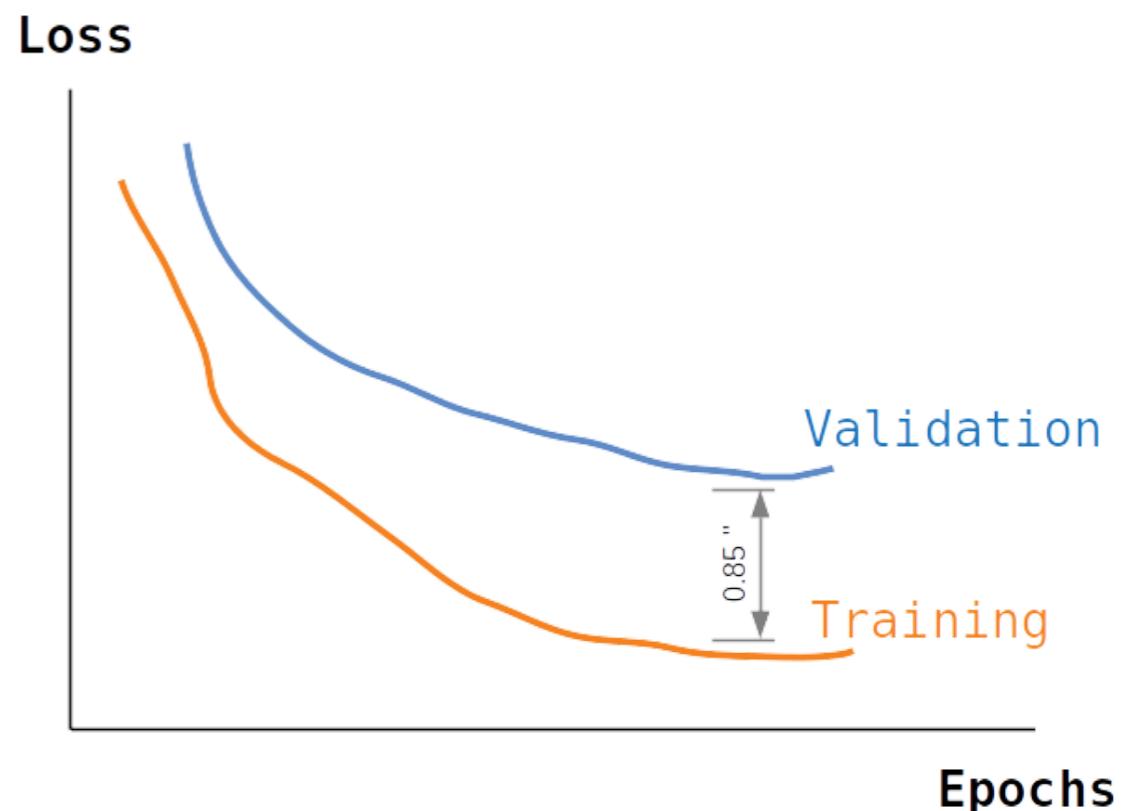
Underfitting

- In this case the network isn't really learning much from training data
- In the top diagram, the loss is not decreasing. The network is no longer learning from training data
- In the bottom diagram, loss function is bouncing around; no improvement
- Causes:
 - The training data is too small; it is not providing enough information for the model to learn
 - The training dataset is not representative
- Possible fixes:
 - Get more training data, if possible
 - Try a more complex model



Overfitting

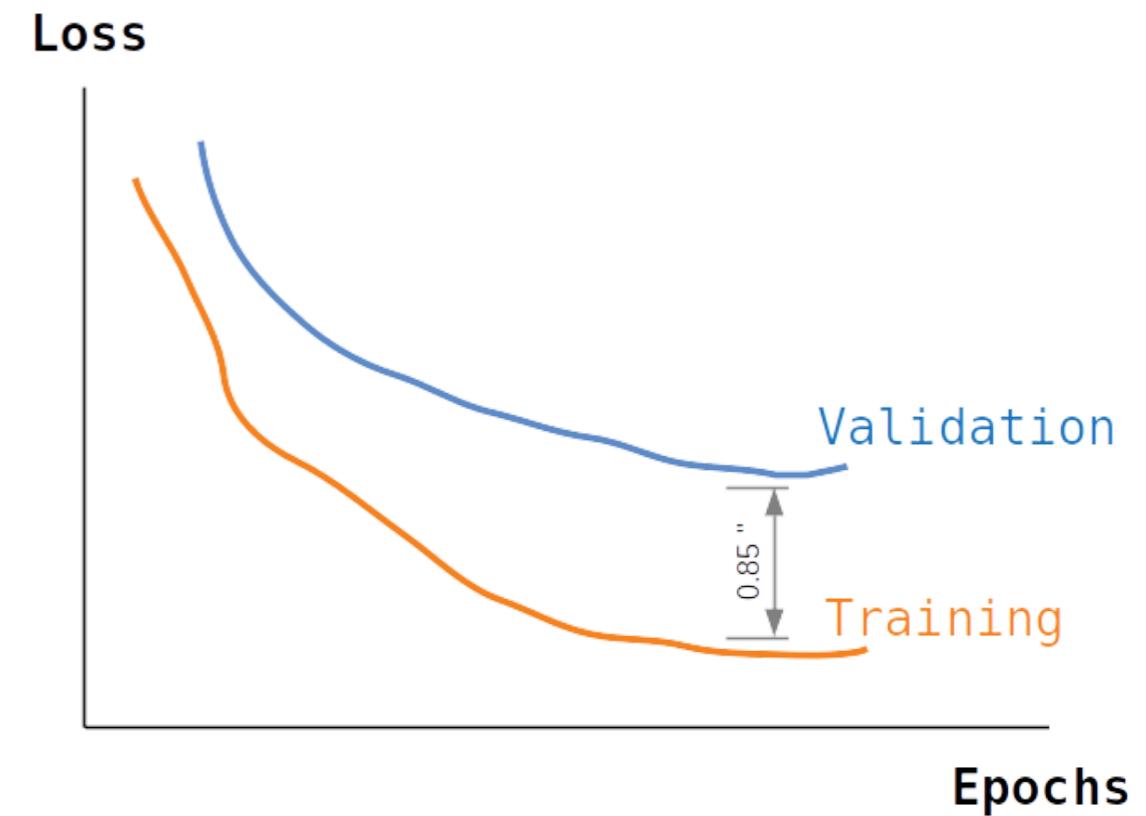
- In this scenario the model is 'memorizing' training data instead of learning from it.
- The model will do well in training data (training loss will be lower (better))
- But the model will not do well with validation data (loss will be higher (worse))
- Here we see training loss is significantly better/lower than validation loss (that is higher)
- This is usually a reliable indication of overfitting



Overfitting

- Causes:

- Not enough training data
- Not enough variety in training data; the model doesn't have enough information to learn from; so it is simply memorizing it
- The model is too complex



- Possible fixes:

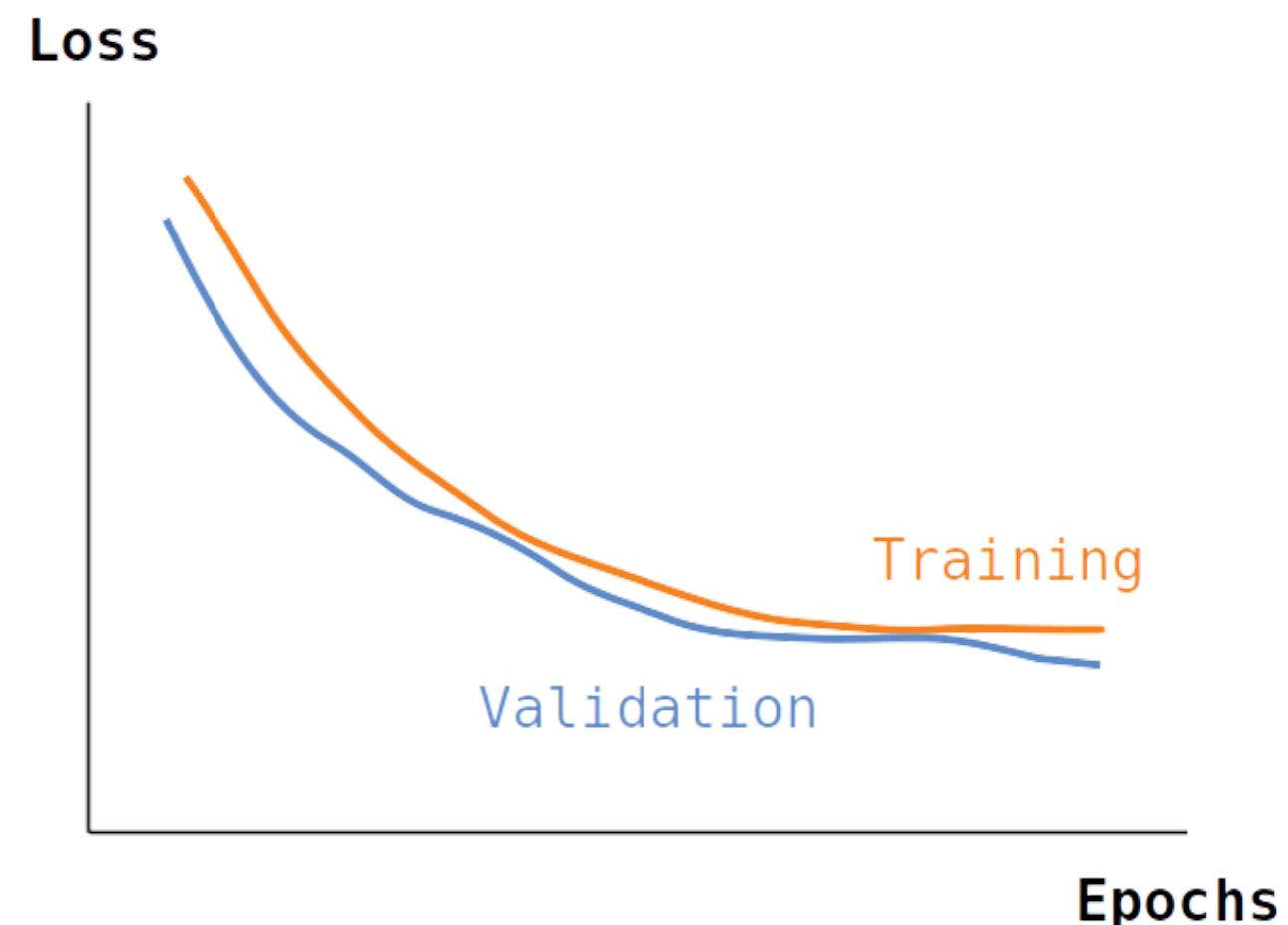
- Provide more training data
- Increase the variety in training data
- Simplify the model
- Add a **dropout layer**
- Add a **batch normalization layer**

Preventing Overfitting

- **Instructor:** If time permits, review **DL-Concepts # Preventing Overfitting** section

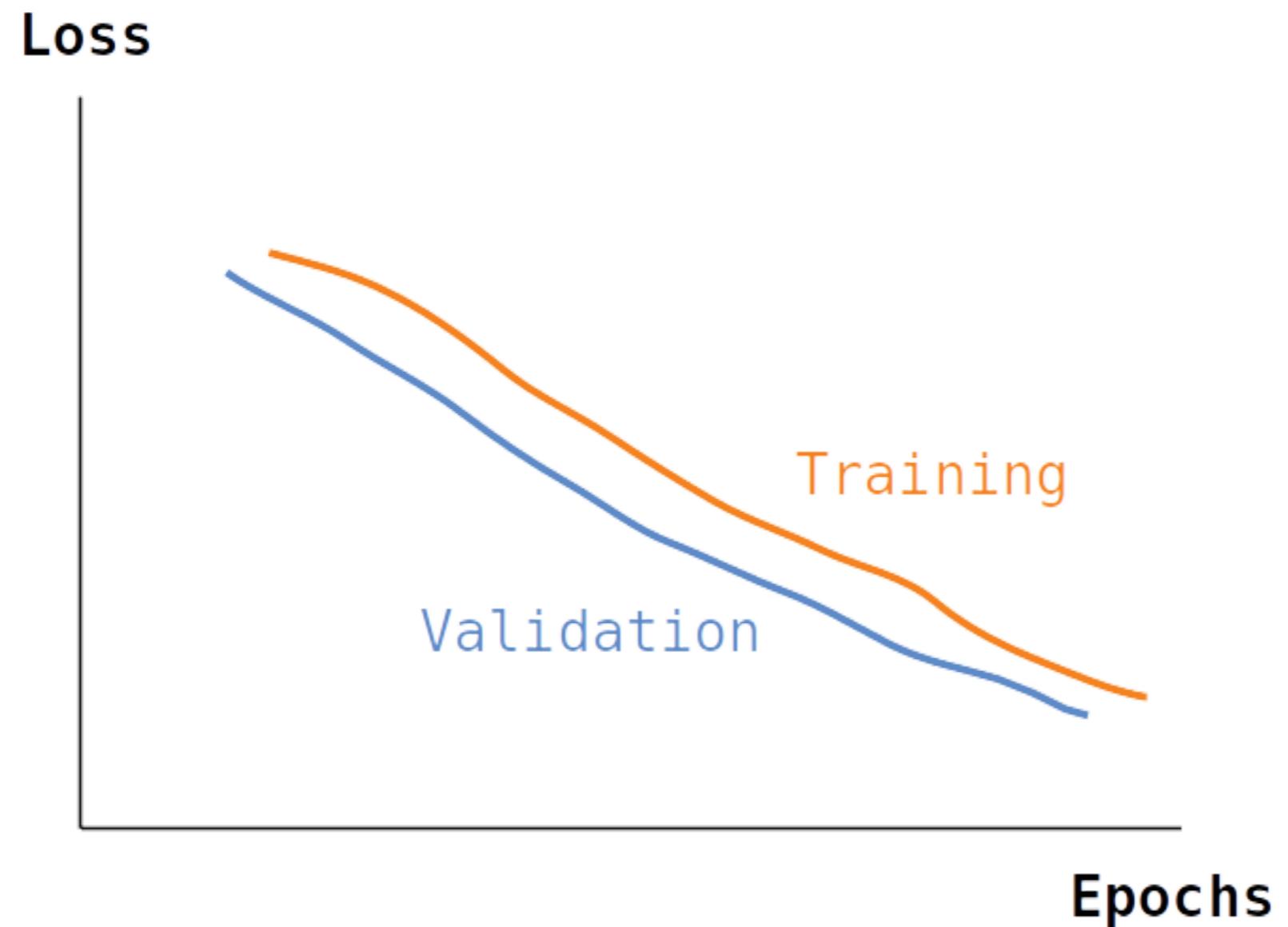
Goodfit

- Here is a good example of **goodfit**
- Both training and validation losses are decreasing in tandem
- They are both smooth (no bouncing around) ; indicating a good convergence
- And there is no large gap between training/validation losses



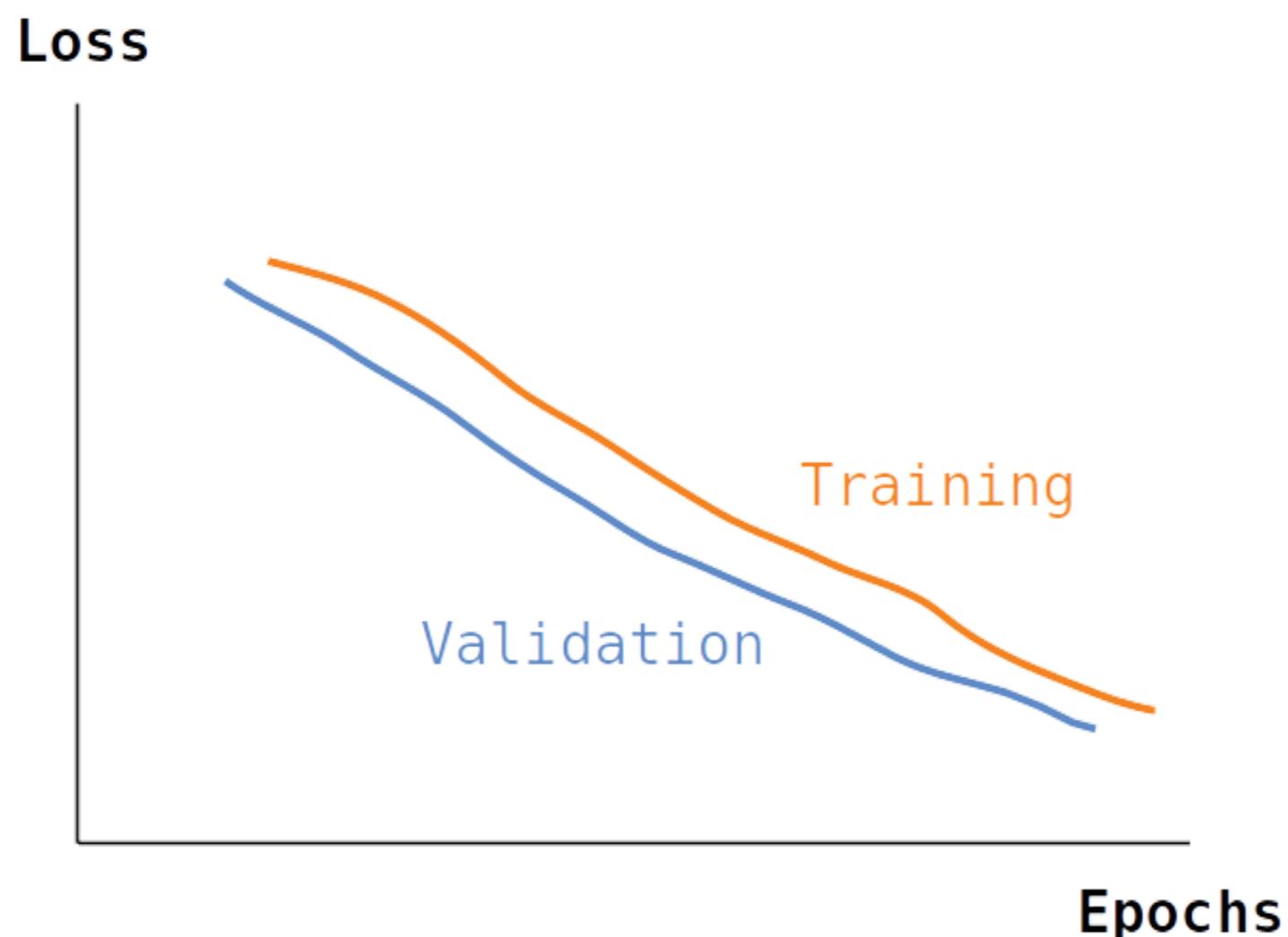
Quiz: Evaluate Training

- What do you think of this training?
- Answer next slide



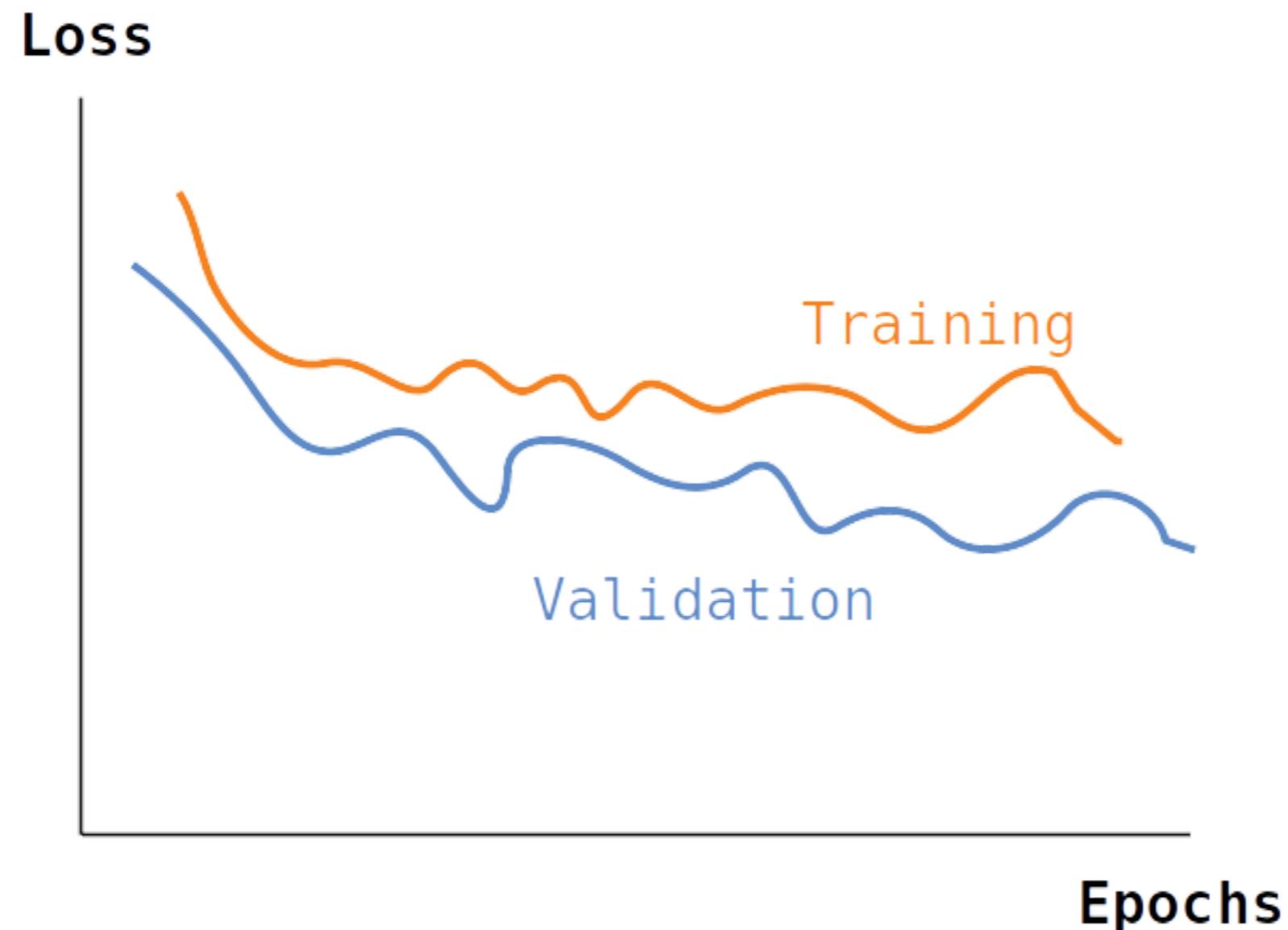
Answer: Evaluate Training

- Here both training and validation losses are improving steadily
- But we stopped the training prematurely
- We should add more epochs the model can keep learning



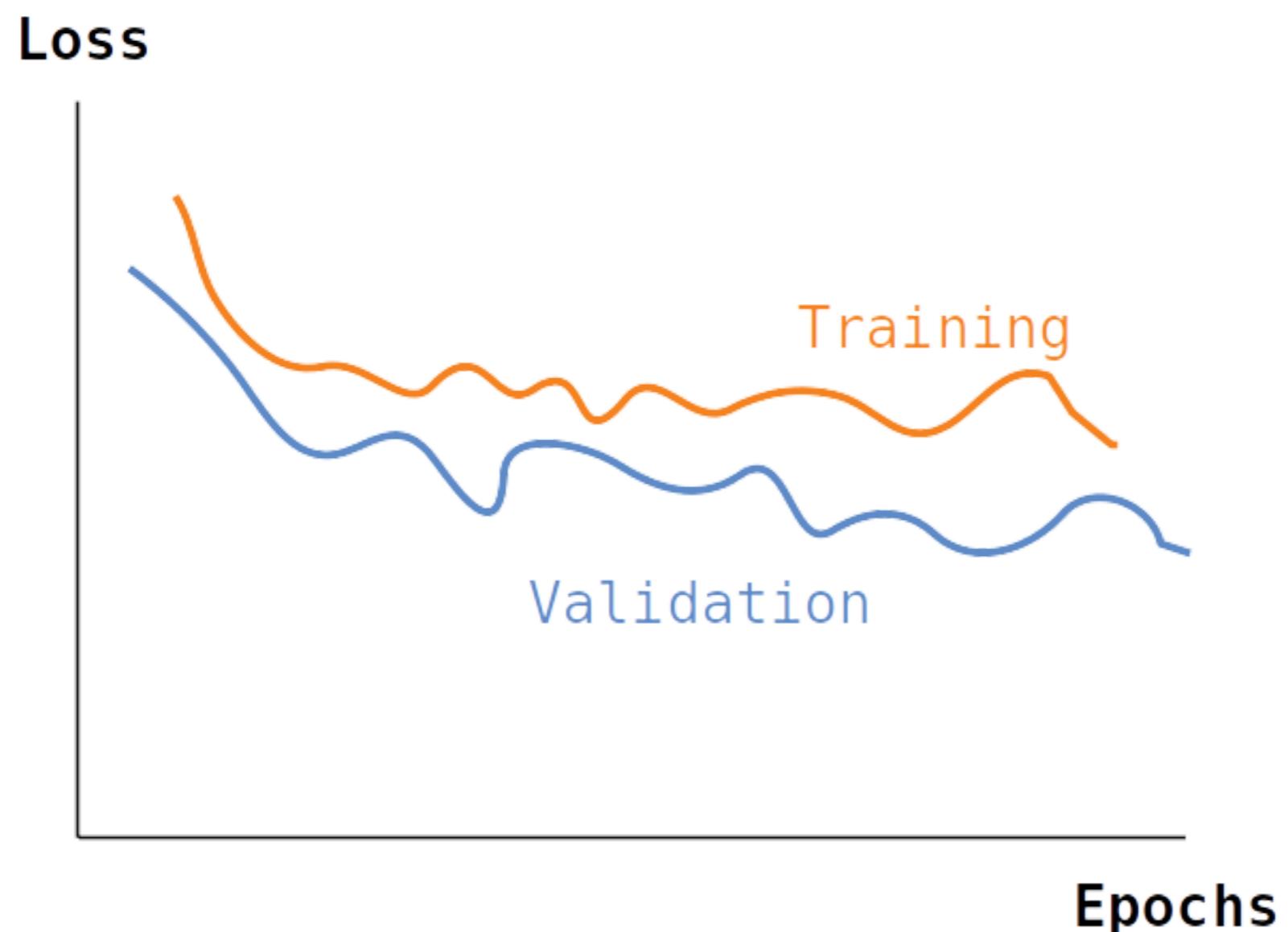
Quiz: Evaluate Training

- What do you think of this training?
- Answer next slide



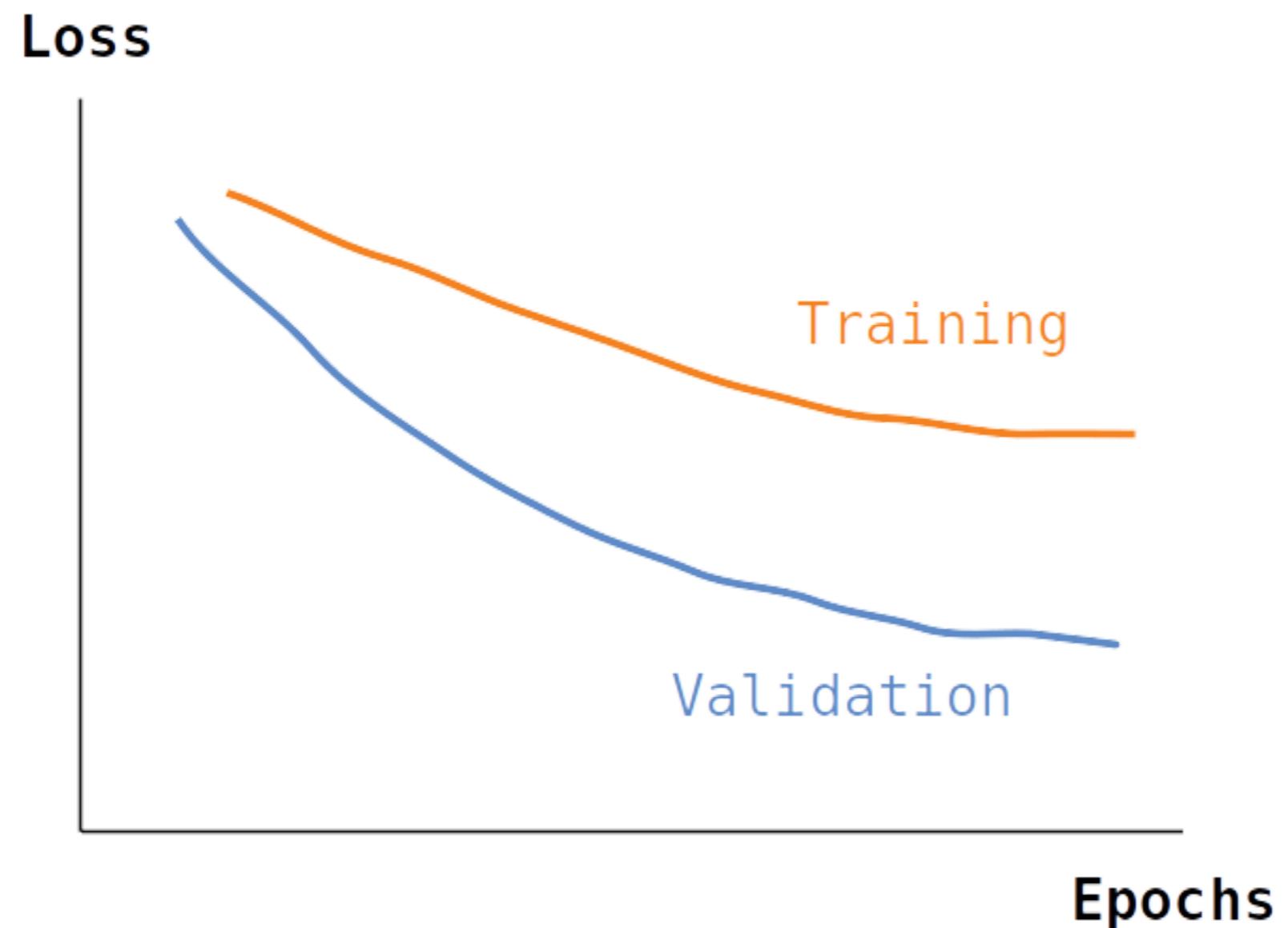
Answer: Evaluate Training

- Here both training and validation losses are not improving steadily
- They are bouncing around a lot
- Model is not stable



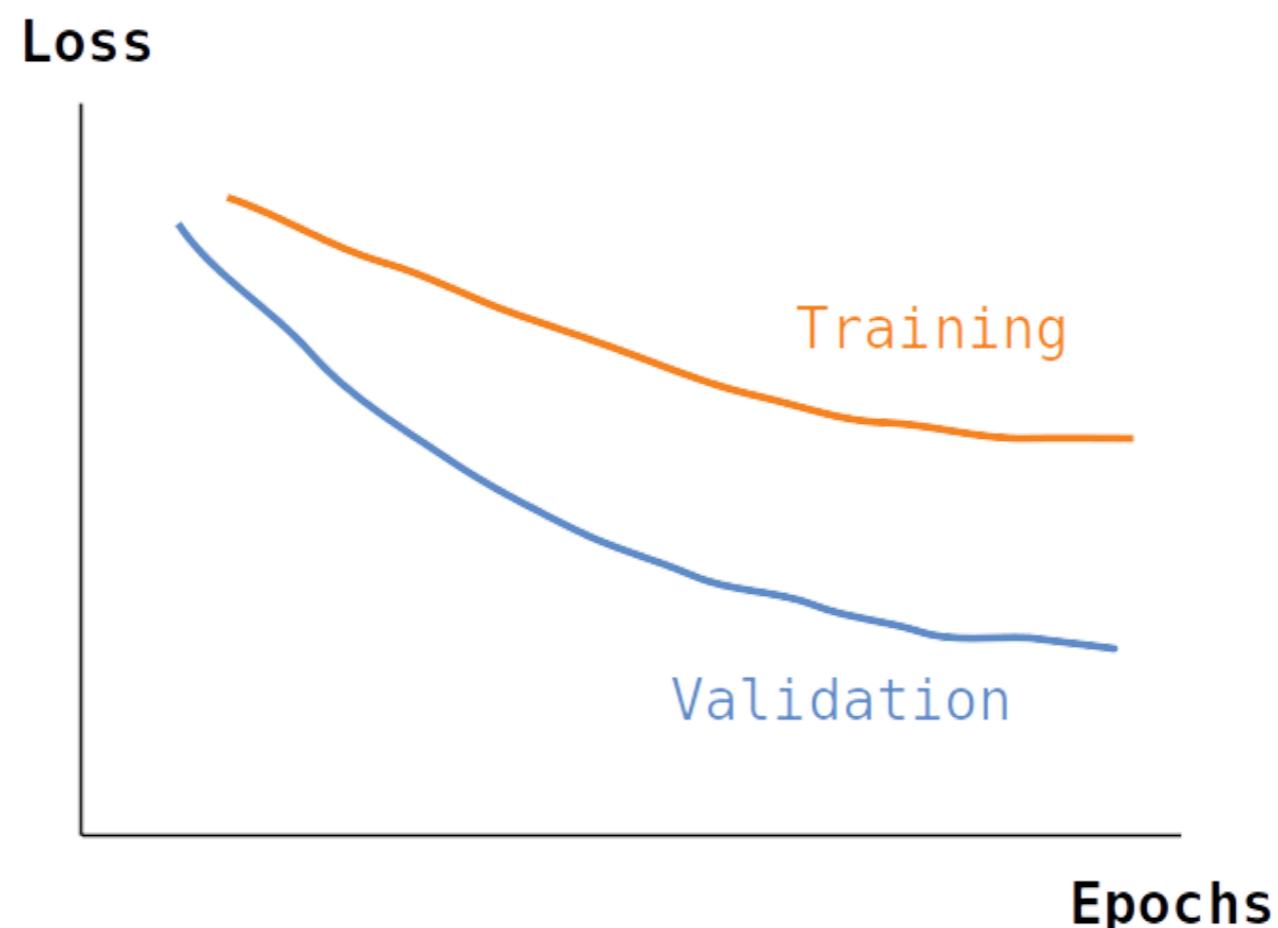
Quiz: Evaluate Training

- What do you think of this training?
- Validation is doing better than training!
- Answer next slide



Answer: Evaluate Training

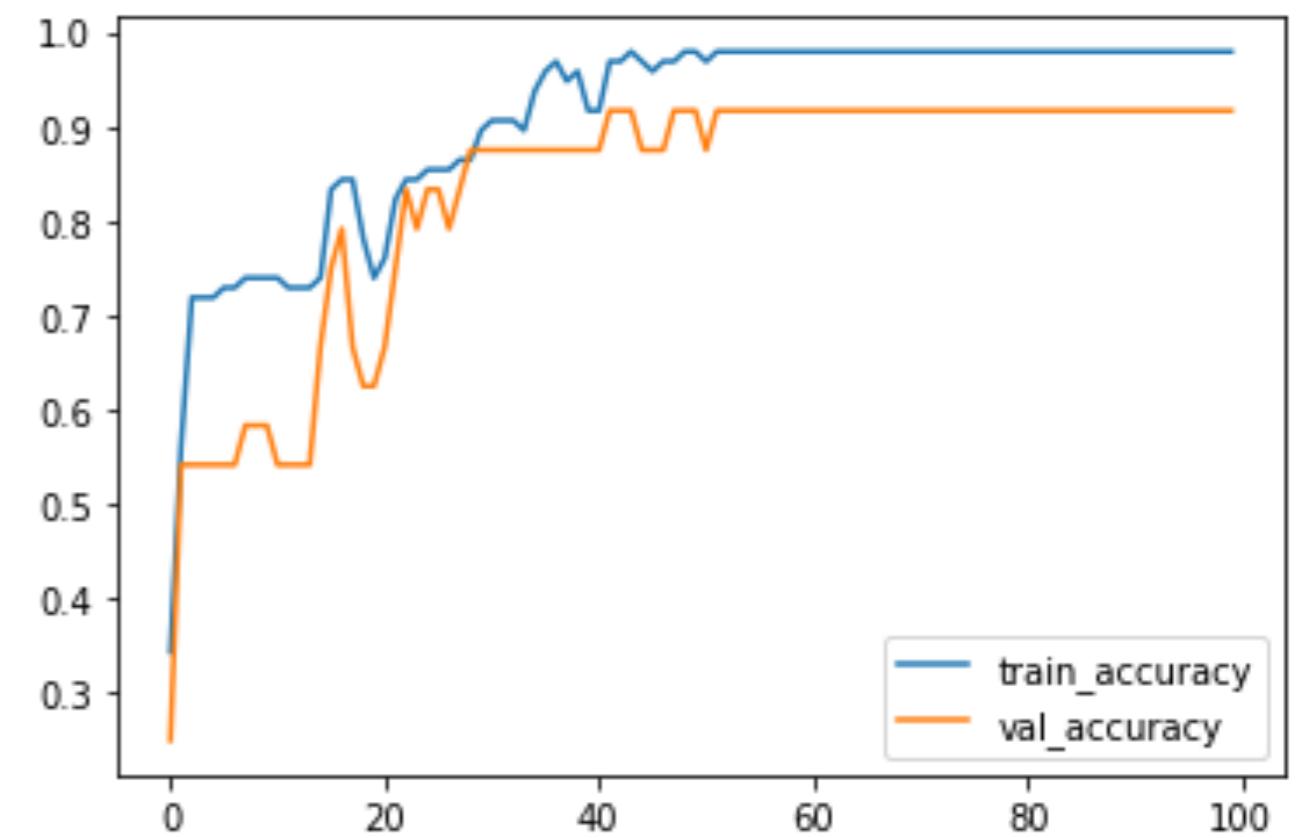
- We are seeing better results for validation than training
- Means, the validation dataset is 'too simple'; so the model is doing better there
- Fixes:
 - Need better validation set



Callbacks

Callbacks

- In our previous lab runs we have noticed with increased epochs the network accuracy increases
- However after certain epochs, the accuracy hardly increases
- Here in IRIS classification, after epoch=60 we don't see much improvement in accuracy
- So can we stop training if accuracy doesn't improve?
- **Callbacks** can help!



Standard Callbacks

- Tensorflow provides following callbacks
 - **EarlyStopping** : Stop training when a monitored metric has stopped improving
 - **ModelCheckpoint** : Callback to save the Keras model or model weights at some frequency
 - **TensorBoard** : Enable visualizations for TensorBoard.
 - and more...
- Callback reference

Early Stopping

- This can stop training when a metric has stopped improving
- Reference

```
cb_early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', min_delta=0.5, patience=3)

## Arguments
##   - monitor: metric to monitor; here it is 'loss'
##   - min_delta: The minimum change in monitored metric that qualifies as improvement
##   - patience: number of consecutive epochs where the metric didn't improve

## This callback will stop the training when there is no improvement in
## the validation accuracy for 3 consecutive epochs.

## Supply the callback during training
history = model.fit (x, y, epochs=100,
                     callbacks = [cb_early_stop]) ## <-- here
```

Using More Than One Callback

- We can provide more than one callbacks

```
## Tensorboard callback
cb_tensorboard = tf.keras.callbacks.TensorBoard(log_dir='tblogs', histogram_freq=1)

## Early stopping callback
cb_early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', min_delta=0.5, patience=3)

## Supply the callbacks during training
history = model.fit (x_train, y_train, epochs=100,
                     callbacks = [cb_tensorboard, cb_early_stop]) ## <-- here
```

Custom Callbacks

- We can also provide our own callbacks!
- Extend `tf.keras.callbacks.Callback`
- Reference

```
on_epoch_end: logs include `acc` and `loss`, and
    optionally include `val_loss`
    (if validation is enabled in `fit`), and `val_acc`
    (if validation and accuracy monitoring are enabled).

on_batch_begin: logs include `size`,
    the number of samples in the current batch.

on_batch_end: logs include `loss`, and optionally `acc`
    (if accuracy monitoring is enabled).
```

Implementing a Custom Callback

```
class CustomCallback(keras.callbacks.Callback):
    def on_train_begin(self, logs=None):
        keys = list(logs.keys())
        print("Starting training; got log keys: {}".format(keys))

    def on_train_end(self, logs=None):
        keys = list(logs.keys())
        print("Stop training; got log keys: {}".format(keys))

    def on_epoch_begin(self, epoch, logs=None):
        keys = list(logs.keys())
        print("Start epoch {} of training; got log keys: {}".format(epoch, keys))

    def on_epoch_end(self, epoch, logs=None):
        keys = list(logs.keys())
        print("End epoch {} of training; got log keys: {}".format(epoch, keys))

    def on_test_begin(self, logs=None):
        keys = list(logs.keys())
        print("Start testing; got log keys: {}".format(keys))

    def on_test_end(self, logs=None):
        keys = list(logs.keys())
        print("Stop testing; got log keys: {}".format(keys))

model.fit(x_train, y_train, epochs=2,
          callbacks=[CustomCallback()])
```

- Output next slide

Implementing a Custom Callback

```
Starting training; got log keys: []
...
Start epoch 0 of training; got log keys: []
...

Start testing; got log keys: []
...
Stop testing; got log keys: []
...

End epoch 0 of training; got log keys:
  ['loss', 'mean_absolute_error', 'val_loss', 'val_mean_absolute_error']
...

Start epoch 1 of training; got log keys: []
...
Start testing; got log keys: []
...
Stop testing; got log keys: []
...

End epoch 0 of training; got log keys:
  ['loss', 'mean_absolute_error', 'val_loss', 'val_mean_absolute_error']
...

Stop training; got log keys: []
```

Implementing a Custom Callback

- We will implement a custom callback that will stop training if a desired accuracy is reached

```
class MyCallback(tf.keras.callbacks.Callback):  
  
    def on_epoch_end(self, epoch, logs={}):  
        if(logs.get('accuracy')>0.6):  
            print("\nReached 60% accuracy so cancelling training!")  
            self.model.stop_training = True  
# end class: MyCallback  
  
## Supply custom callback  
model.fit(x_train, y_train, epochs=2,  
          callbacks=[MyCallback()])
```

Implementing a Custom Callback

```
import tensorflow as tf

class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy')>0.6):
            print("\nReached 60% accuracy so cancelling training!")
            self.model.stop_training = True

mnist = tf.keras.datasets.fashion_mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

callbacks = myCallback()

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer=tf.optimizers.Adam(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10, callbacks=[callbacks])
```

- Output next slide
- Reference

Implementing a Custom Callback

- We ran training with **epochs=10**
- However our callback stops training, as the desired accuracy (60%) is reached

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
...
Epoch 1/10
1872/1875 [=====>.] - ETA: 0s - loss: 0.4773 - accuracy: 0.8299
Reached 60% accuracy so cancelling training!
1875/1875 [=====] - 6s 3ms/step - loss: 0.4771 - accuracy: 0.8299
<tensorflow.python.keras.callbacks.History at 0x7f63ad143c88>
```

Lab: Callbacks

- **Overview:**

- Implement a custom callback

- **Depends on:**

- None

- **Runtime:**

- 20 mins

- **Instructions:**

- **Callback-1**



Review and Q&A

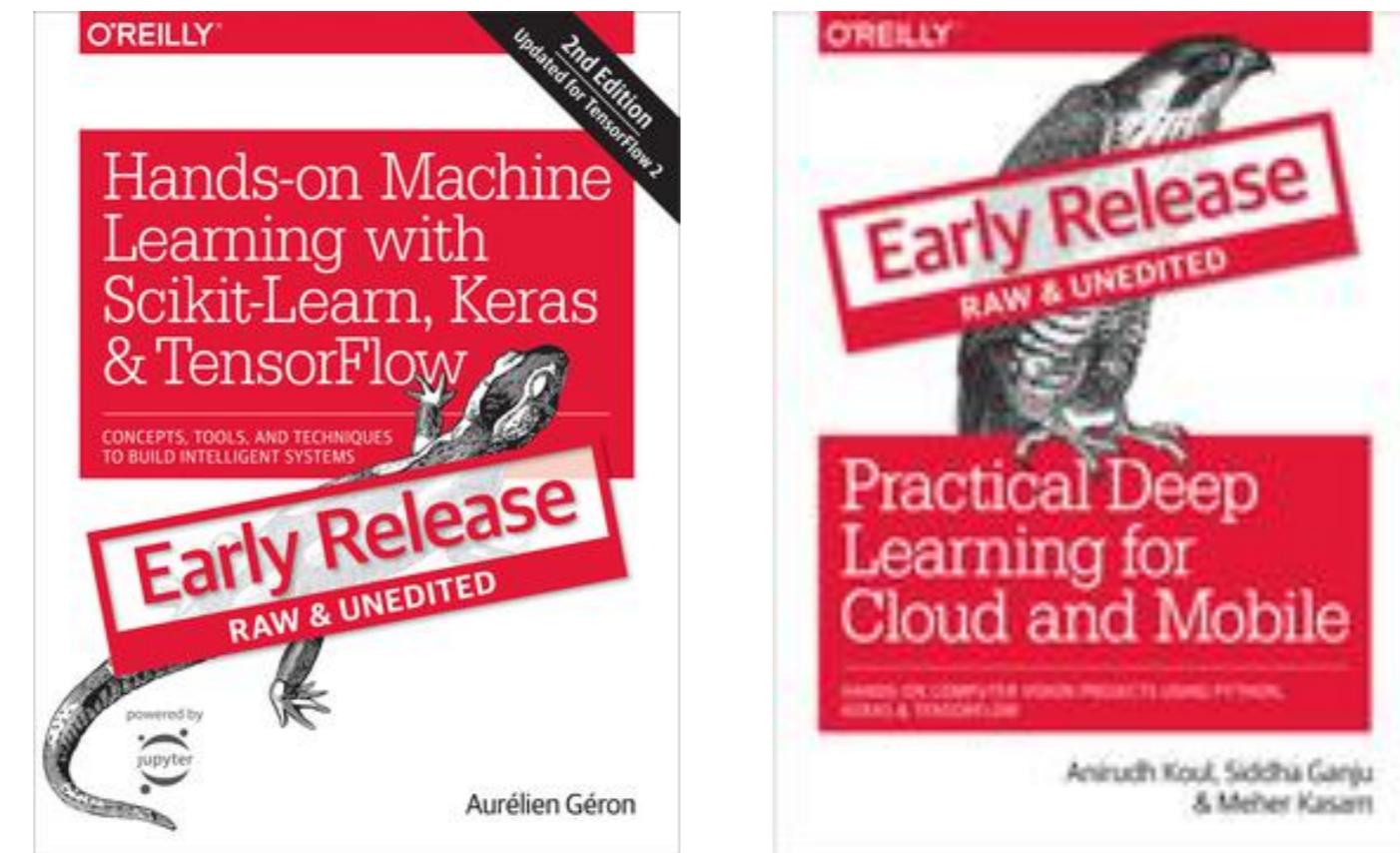
- Let's go over what we have covered so far
- Any questions?
- See following slides for 'resources'



Resources

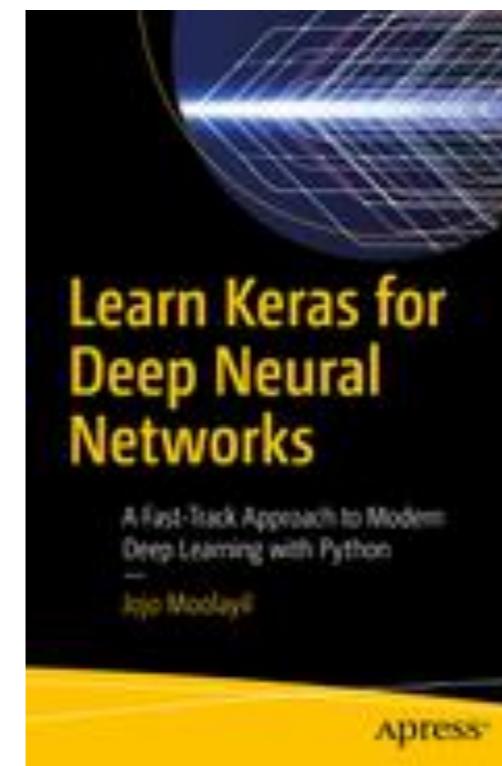
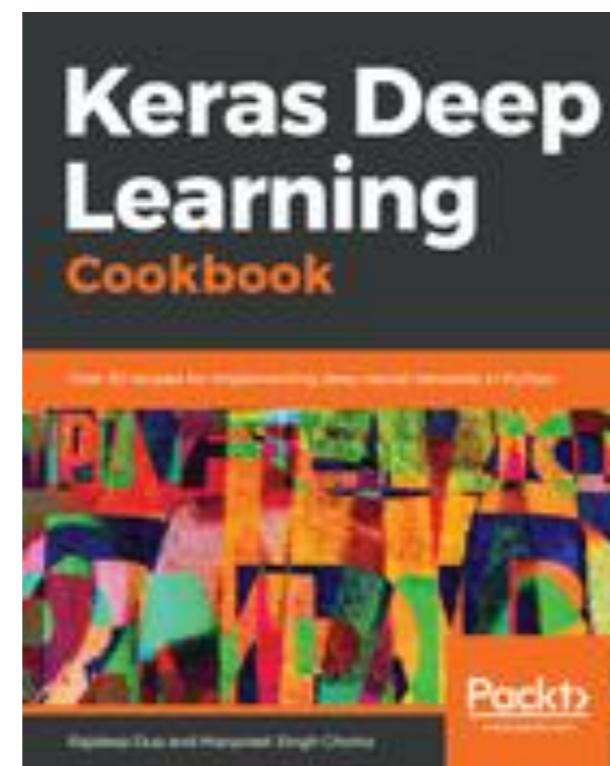
Resources

- [Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition (<https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/>) by Aurélien Géron (ISBN: 9781492032649)]
- Practical Deep Learning for Cloud and Mobile by Meher Kasam, Siddha Ganju, Anirudh Koul (ISBN: 9781492034841)



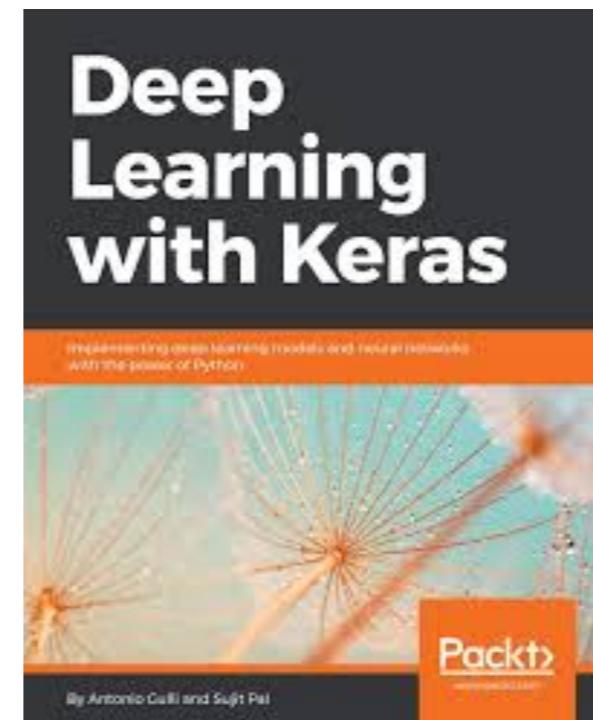
Resources

- Keras Deep Learning Cookbook by Manpreet Singh Ghotra, Rajdeep Dua (ISBN: 9781788621755)
- Learn Keras for Deep Neural Networks: A Fast-Track Approach to Modern Deep Learning with Python by Jojo Moolayil (ISBN : 9781484242407) very good book that explains concepts pretty well



Resources

- Deep Learning with Keras: Implement various deep-learning algorithms in Keras and see how deep-learning can be used in games by Sujit Pal, Antonio Gulli (ISBN: 9781787128422)
- Safari books online, Keras books



Appendix

Loss Function API

Optimizer: SGD

```
# Use default values
model.compile(optimizer='sgd', loss='mean_squared_error')

# ~~~~~

# or Customize
from tensorflow.keras.optimizers import SGD
sgd = SGD(lr=0.01,
           decay=1e-6,
           momentum=0.9,
           nesterov=True) # using Nesterov momentum
model.compile(optimizer=sgd, loss='mean_squared_error')
```

■ Arguments

- lr: float ≥ 0 . Learning rate.
- momentum: float ≥ 0 . Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- decay: float ≥ 0 . Learning rate decay over each update.
- nesterov: boolean. Whether to apply Nesterov momentum.

Optimizer: Adagrad

```
# use default args
model.compile(optimizer='adagrad', ...)

# ~~~~~

# or Customize
from tensorflow.keras.optimizers import Adagrad
adagrad = keras.optimizers.Adagrad(lr=0.01,
                                    epsilon=None,
                                    decay=0.0)
model.compile(optimizer=adagrad, ...)
```

■ Arguments

- lr: float ≥ 0 . Initial learning rate.
- epsilon: float ≥ 0 . If None, defaults to K.epsilon().
- decay: float ≥ 0 . Learning rate decay over each update.

Optimizer: RMSProp

```
# use default values
model.compile(optimizer='rmsprop', ...)

# ~~~~~

# or customize
from tensorflow.keras.optimizers import RMSprop
rmsprop = keras.optimizers.RMSprop(lr=0.001,
                                    rho=0.9,
                                    epsilon=None,
                                    decay=0.0)
model.compile(optimizer=rmsprop, ...)
```

■ Arguments

- lr: float ≥ 0 . Learning rate.
- rho: float ≥ 0 .
- epsilon: float ≥ 0 . Fuzz factor. If None, defaults to K.epsilon().
- decay: float ≥ 0 . Learning rate decay over each update.

Optimizer: Adam

```
# use default values
model.compile(optimizer='adam', ...)

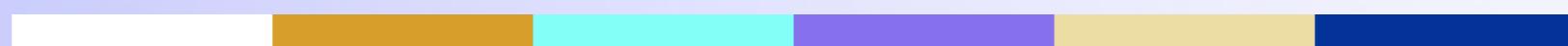
# or customize
from tensorflow.keras.optimizers import Adam
adam = keras.optimizers.Adam(lr=0.001,
                             beta_1=0.9,
                             beta_2=0.999,
                             epsilon=None,
                             decay=0.0,
                             amsgrad=False)
model.compile(optimizer=adam, ...)
```

Optimizer: Adam

▪ Arguments

- lr: float ≥ 0 . Learning rate.
- beta_1: float, $0 < \beta < 1$. Generally close to 1.
- beta_2: float, $0 < \beta < 1$. Generally close to 1.
- epsilon: float ≥ 0 . Fuzz factor. If None, defaults to K.epsilon().
- decay: float ≥ 0 . Learning rate decay over each update.
- amsgrad: boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

Convolutional Neural Networks (CNNs) in TensorFlow



Objectives

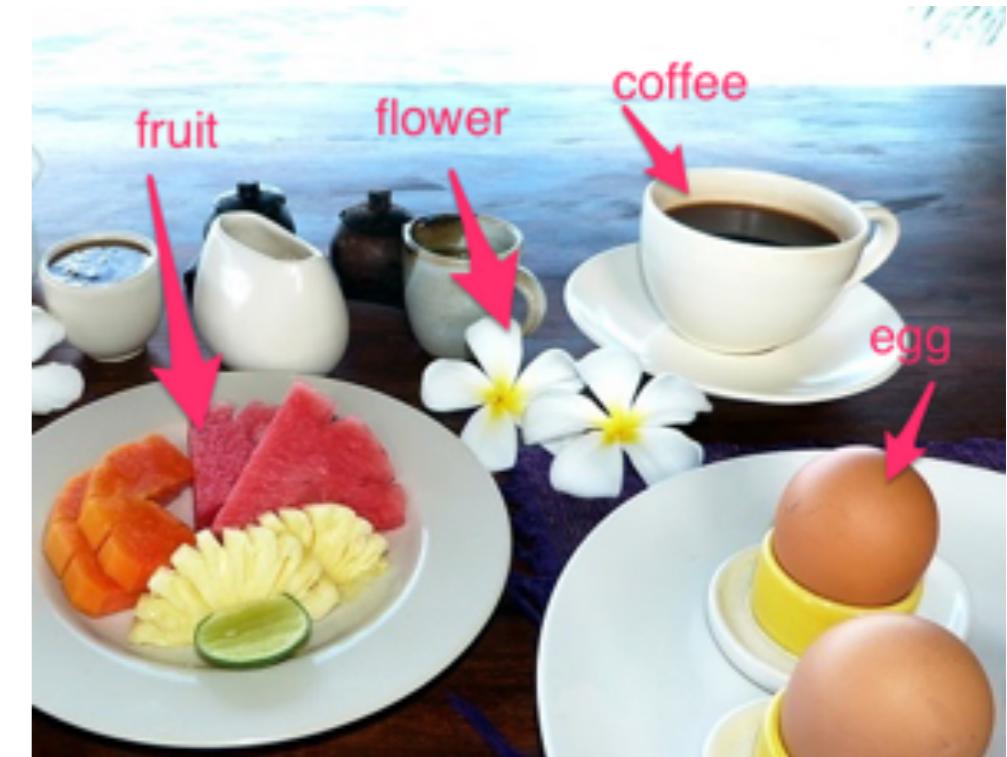
- Learn about CNNs
- Implement CNNs in TensorFlow and Keras

Convolutional Neural Networks (CNN)

Image Recognition

Image Recognition is a Challenging Problem

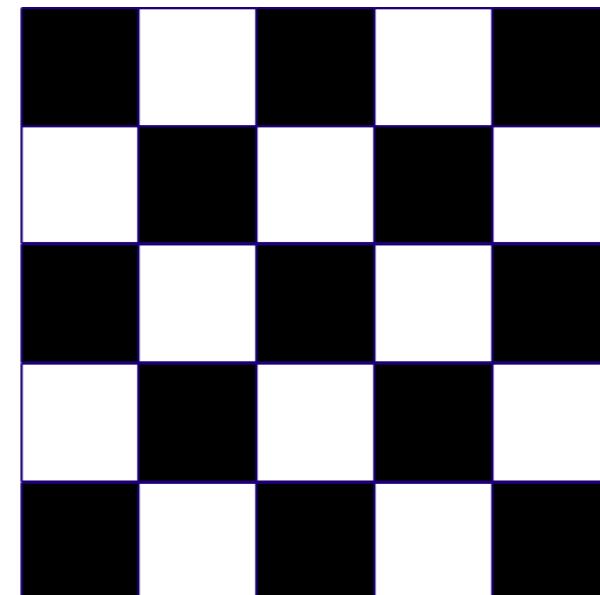
- IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996
- But not until recently (2010) or so, computers were unable to recognize a cat or a dog from an image
- Human brains can do image recognition quite 'effortlessly'
- How ever for computers this is hard
 - Large number of features
 - Features can vary considerably
 - Large Images especially difficult



Dealing With Images

Representing Images

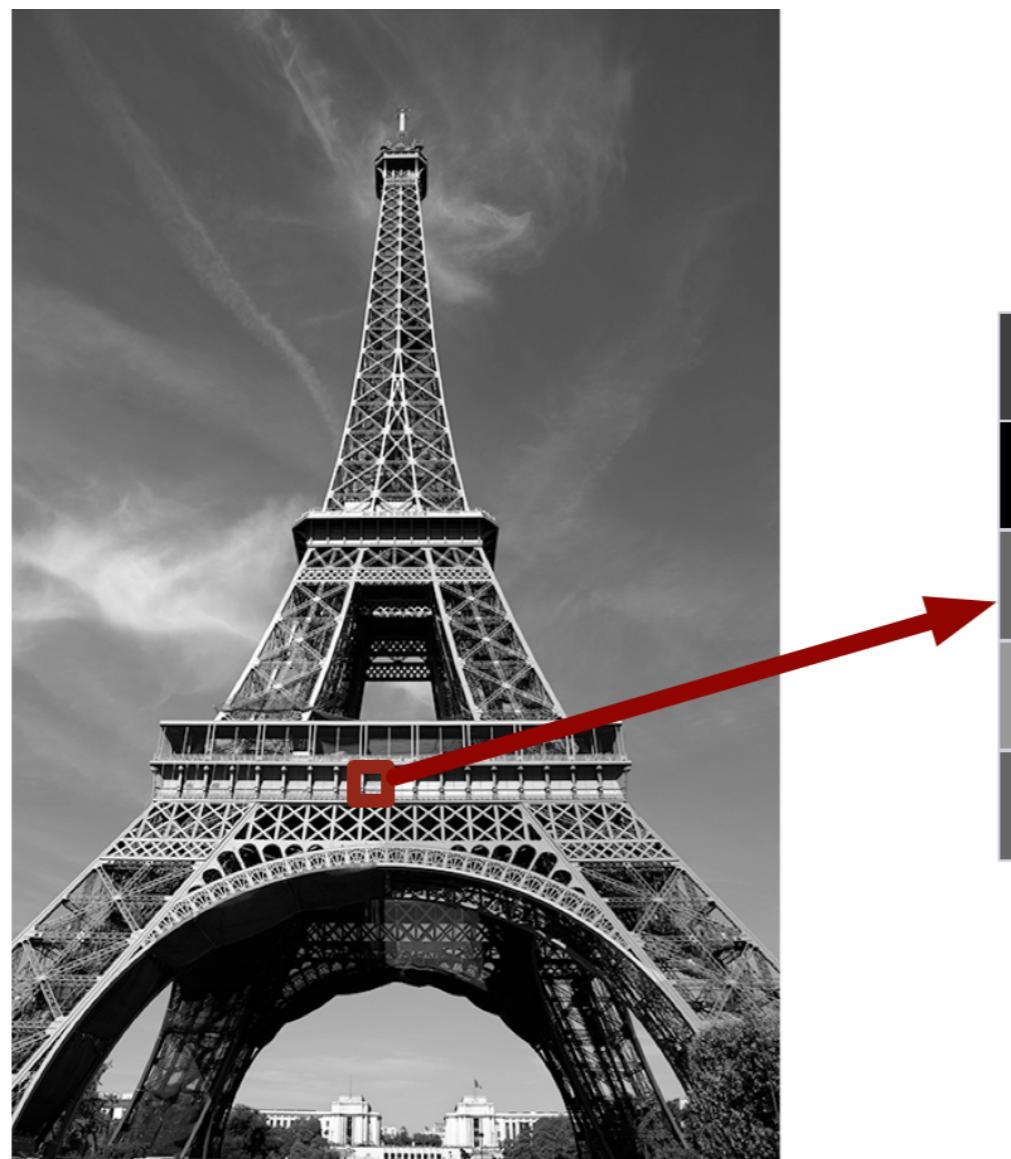
- Images can be represented as a 2D array
- Here is a 5x5 (5 pixels wide, 5 pixels high) image
- for black and white images:
 - black = 0
 - white = 1



0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0

Greyscale Images

- Pixels are represented as numbers ranging from 0 - 255
 - 0: black
 - 255: white
 - in between: grey

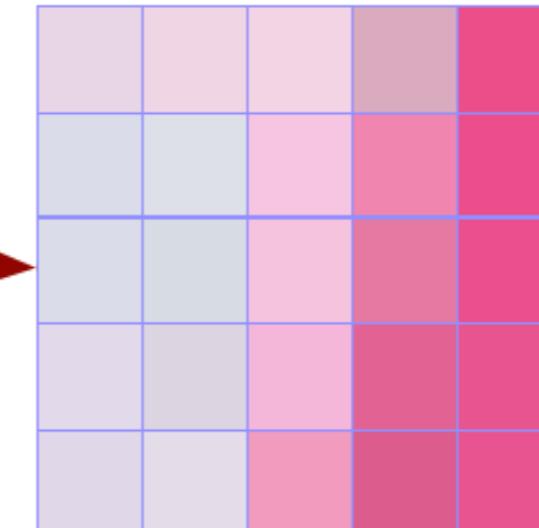


63	106	180	160	159
0	167	255	255	255
105	120	252	255	253
153	175	222	222	219
102	108	200	184	191

63	106	180	160	159
0	167	255	255	255
105	120	252	255	253
153	175	222	222	219
102	108	200	184	191

Color Images

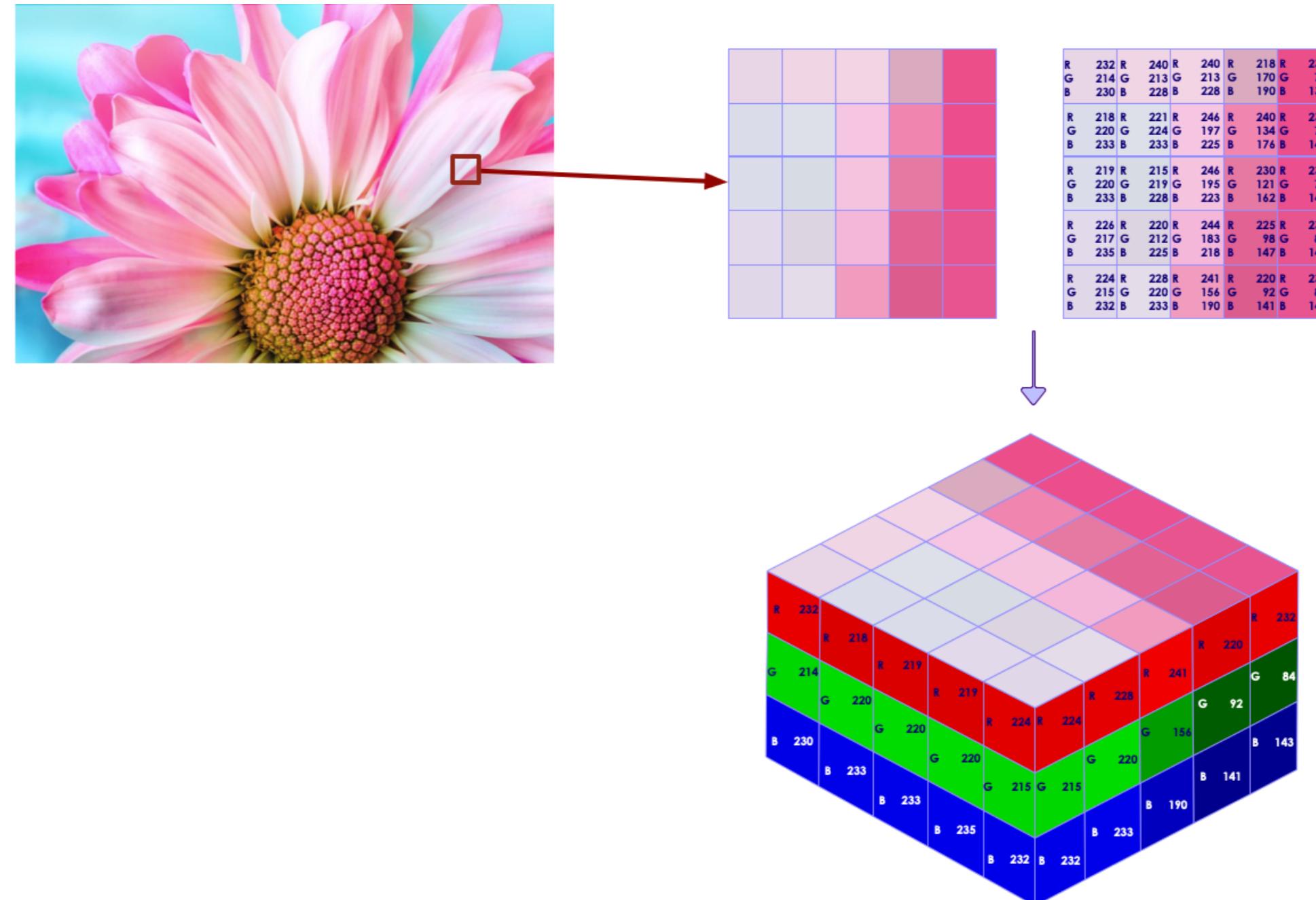
- In color images, each pixel has colors
 - These are represented as RGB (Red, Green, Blue) colors
- Each RGB values are represented as numbers ranging 0-255



R	78	R	240	R	240	R	218	R	236
G	28	G	213	G	213	G	170	G	78
B	195	B	228	B	228	B	190	B	138
R	218	R	221	R	246	R	240	R	236
G	220	G	224	G	197	G	134	G	78
B	233	B	233	B	225	B	176	B	140
R	219	R	215	R	246	R	230	R	236
G	220	G	219	G	195	G	121	G	79
B	233	B	228	B	223	B	162	B	141
R	219	R	220	R	244	R	225	R	232
G	220	G	212	G	183	G	98	G	84
B	233	B	225	B	218	B	147	B	142
R	224	R	228	R	241	R	220	R	232
G	215	G	220	G	156	G	92	G	84
B	232	B	233	B	190	B	141	B	143

Color Images

- Each pixel has 3 'channels' (RGB)
- Color images can be represented as 3D array



Channels in Images

- Images could contain more channels than usual RGB
 - For example, satellite images might have infrared spectrum
- Here is an example of Hubble image taken using multiple cameras in multiple wave lengths, combined together

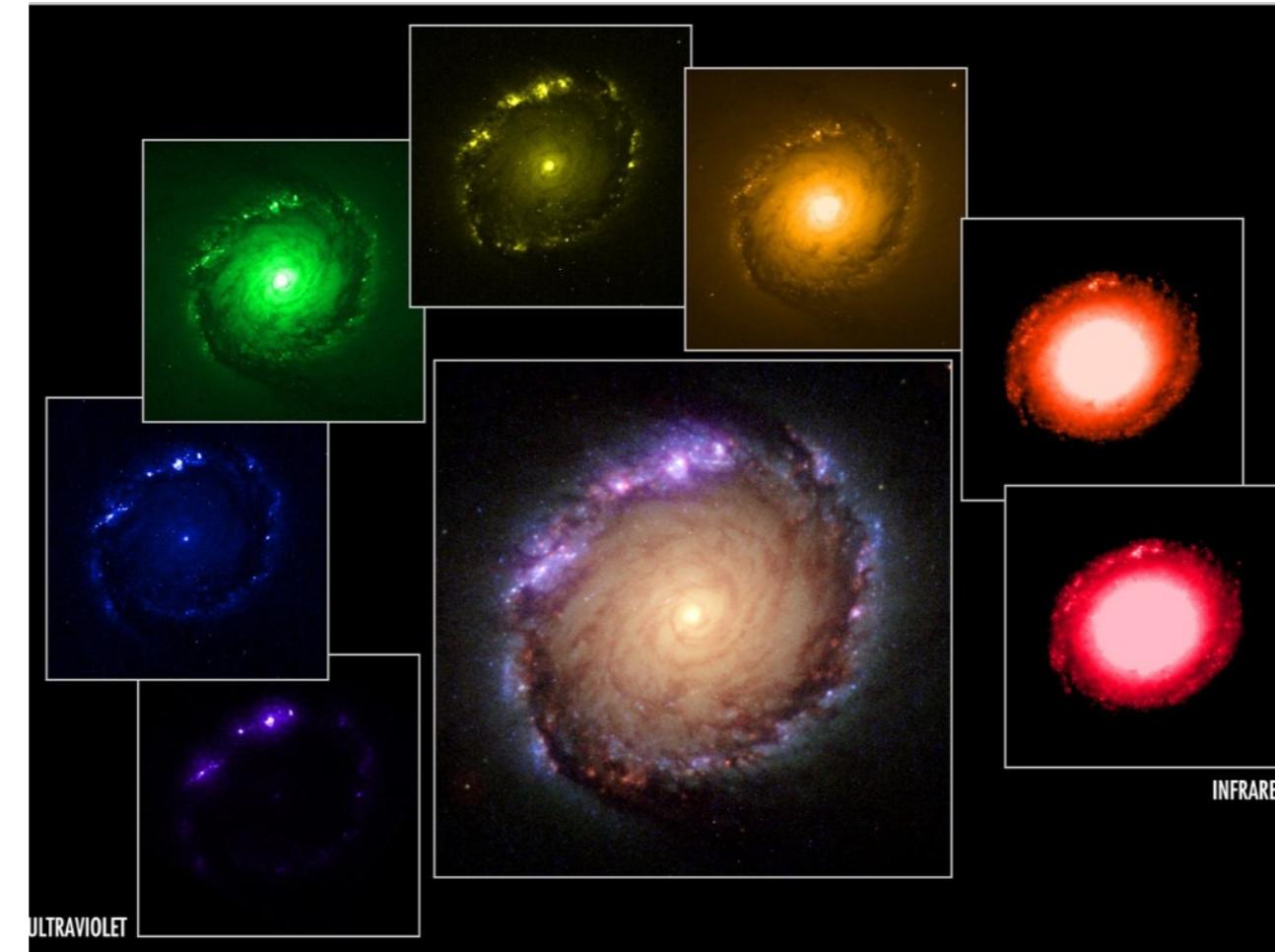
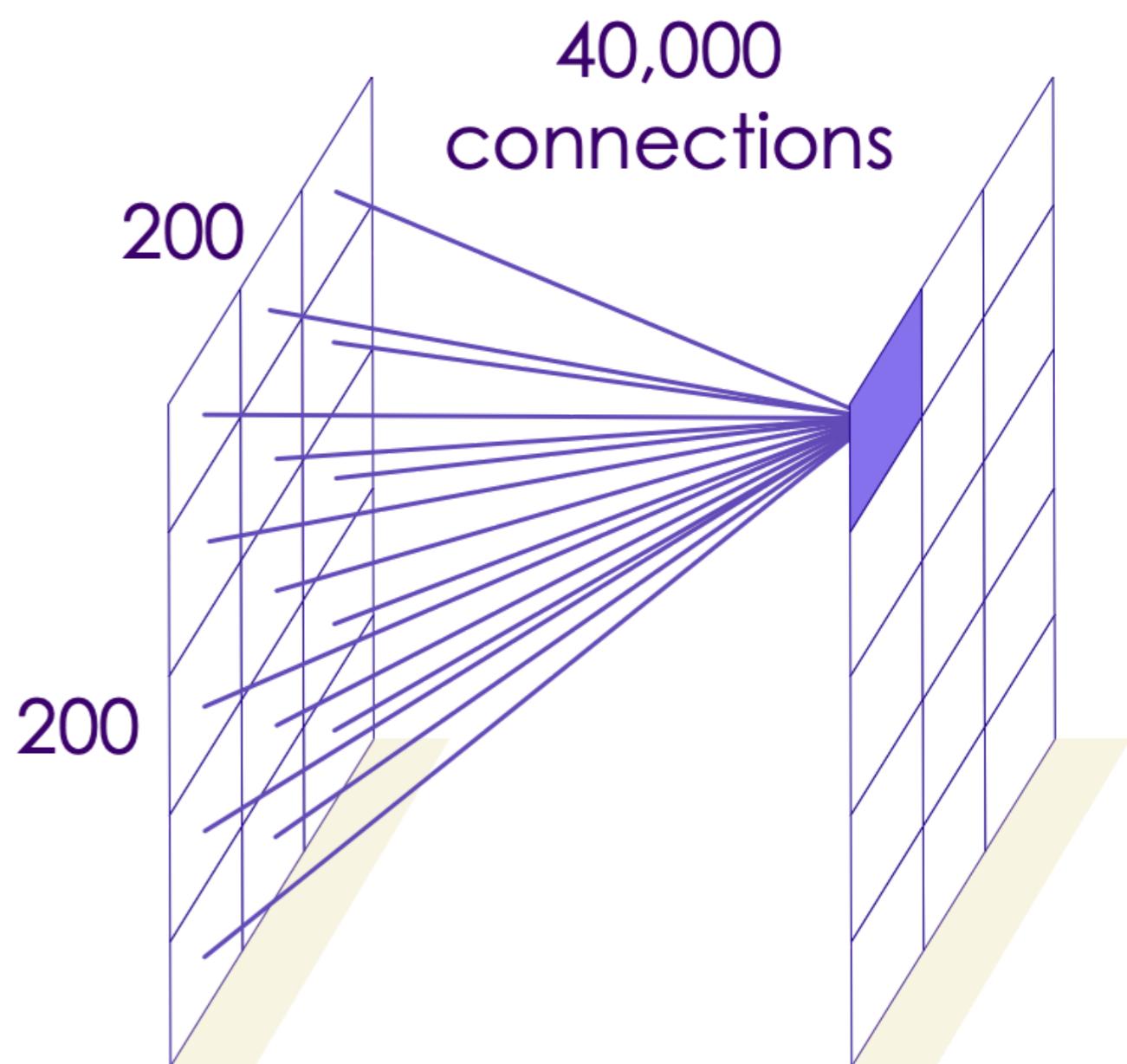


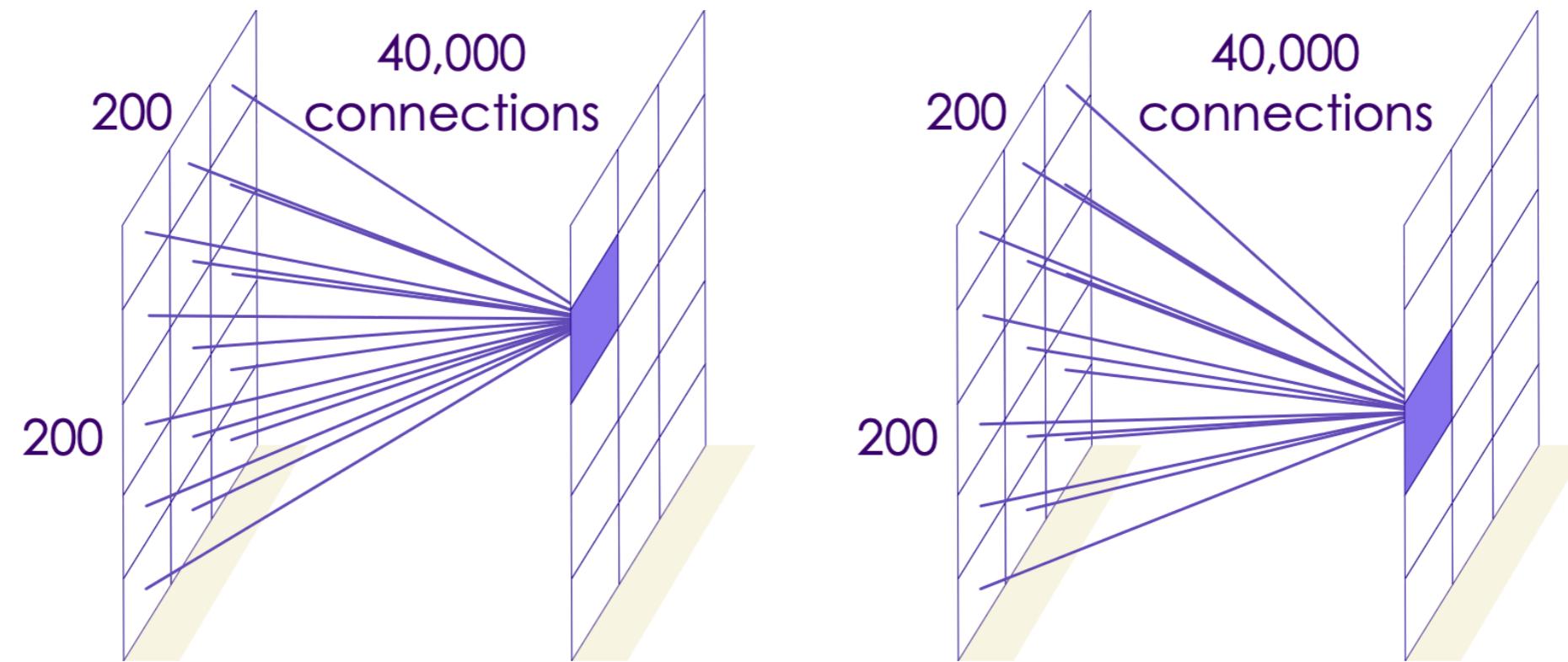
Image Analysis Using Neural Networks

Fully Connected Network's Limitations

- Assume we are analyzing a 200px by 200px image
 - Image has 40,000 (200x200) pixels
- If we connect each neuron on second layer to a neuron in the first layer, each neuron will have 40,000 connections



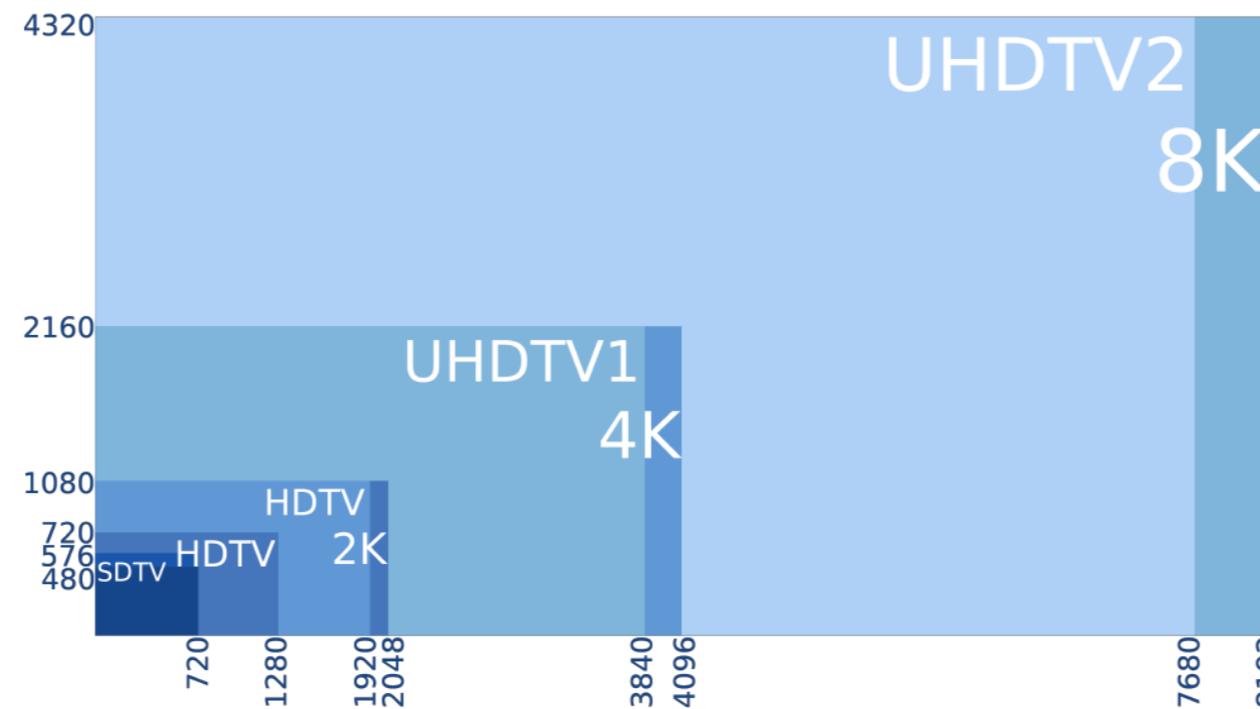
Fully Connected Network's Limitations



- The second layer will have
 - 40,000 connections per pixels \times (200 px \times 200 px) = 1.6 billion connections
 - 10 layers ==> 16 billion connections
- That is way too many connections

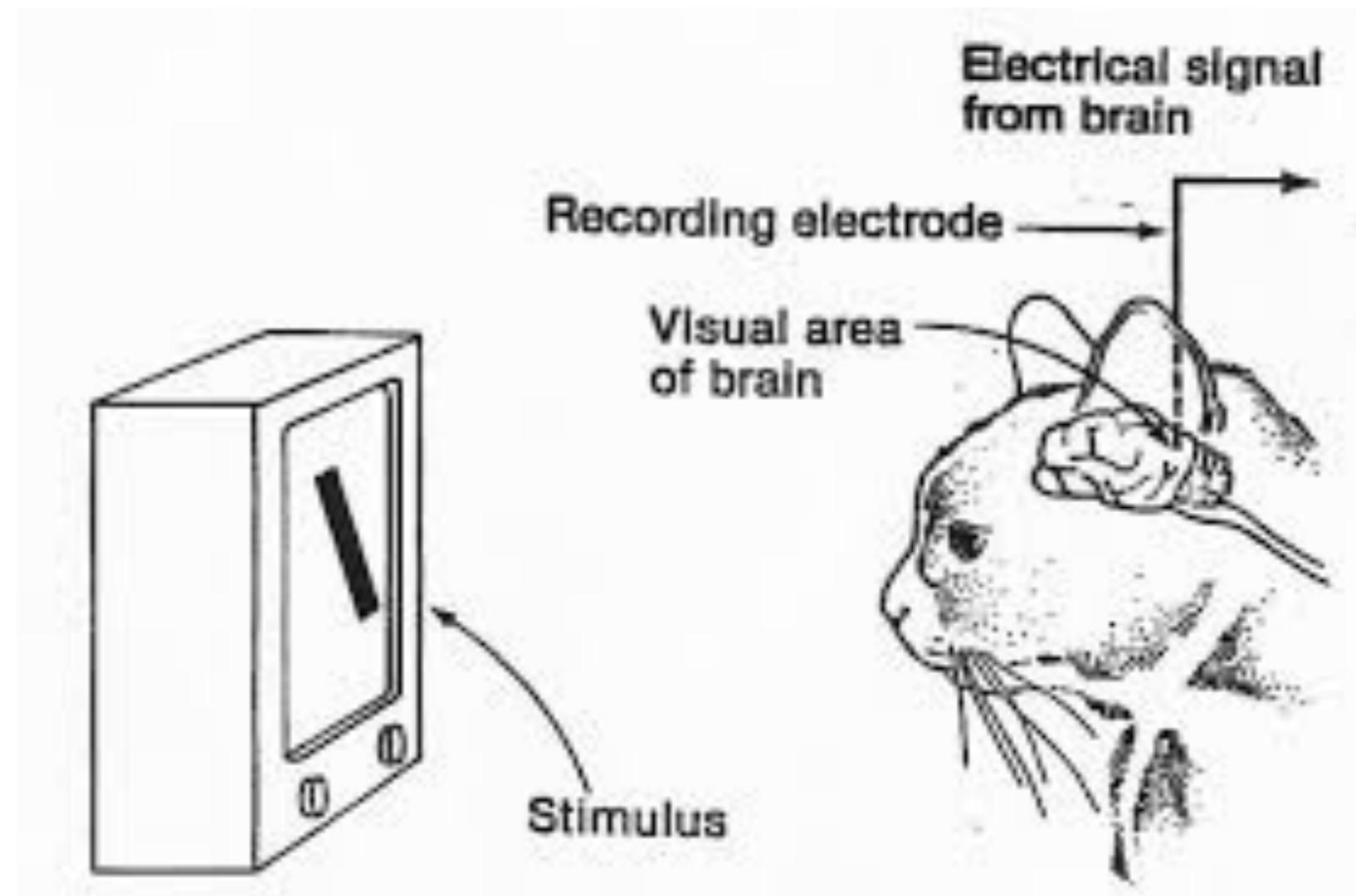
Fully Connected Network's Limitations

- Each layer has 1.6 billion connections; and that many weights to learn
- However, a 200px x 200px image is very small!
- 4K picture is ~ 4000 x 2000 pixels (see picture below, credit [wikimedia](#))
- Can you compute how many weights we will need to compute for a 4K image?
- **Quiz:** Take a picture with your phone. What is the size? (probably 3000 x 2000 pixels)
- **So we need a another approach**



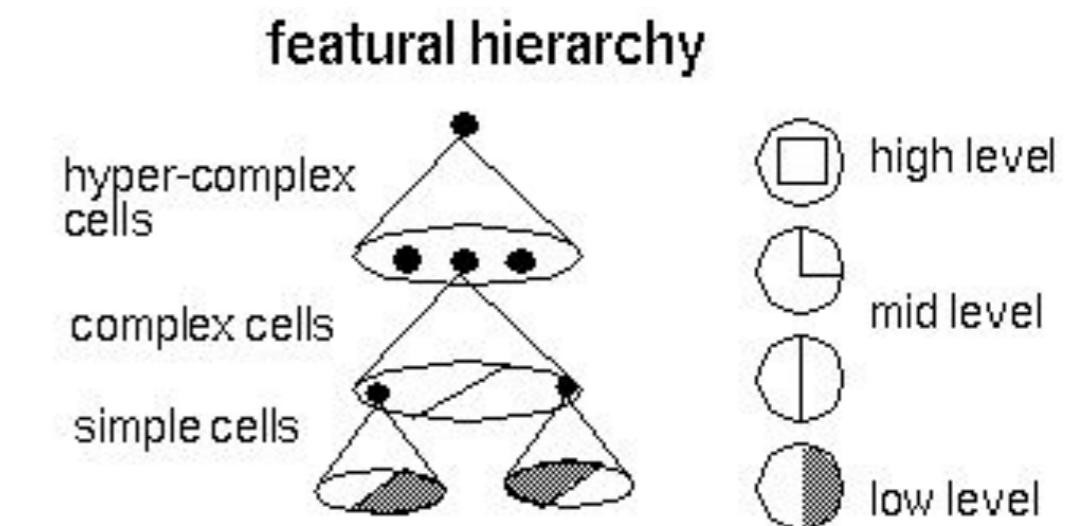
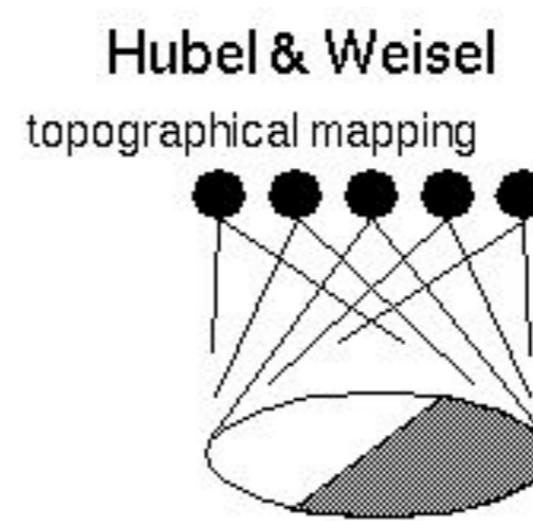
Famous 'Cat Experiment'

- David H. Hubel and Torsten Wiesel performed experiments on cats (1958/1959) that gave us crucial understanding of brain's visual cortex. (paper)
- They hooked up electrodes to a cat's brain (cat is sedated of course!)
- Showed different shapes (dots / lines ..etc) and looked for 'neurons firing'
- But they couldn't get neurons to fire!
- Until they accidentally dragged the shape across the screen, then neurons fired!
- Authors won the Nobel Prize in Physiology or Medicine in 1981 for their work!



Our Visual Cortex is Hierarchical

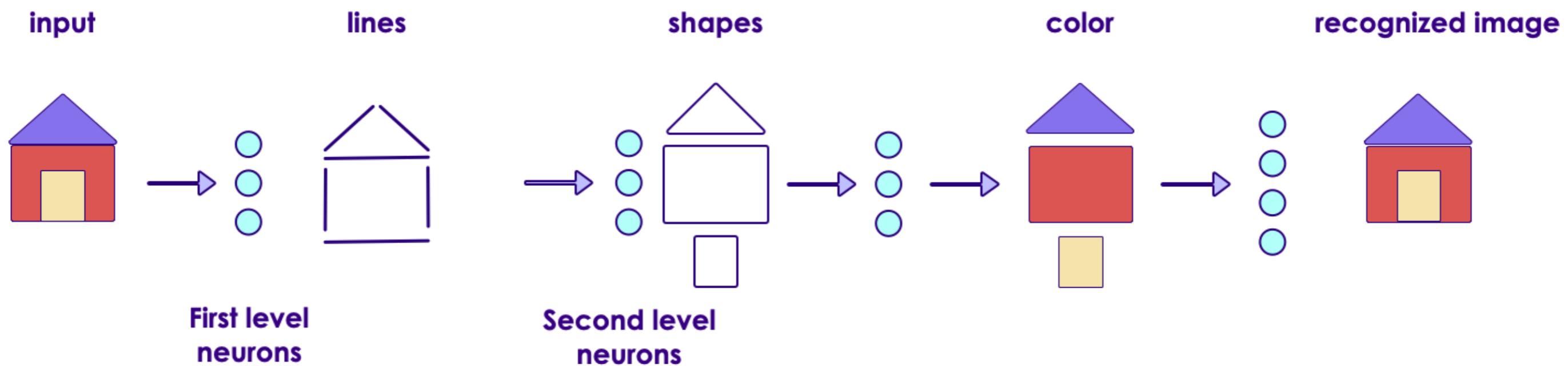
- First level neurons detect simple shapes
- Next level neurons detect more complex shapes, and so on



[Link to video](#)

Hierarchical Visual Cortex

- Our brain's visual system works hierarchically to perceive images
- Some neurons only recognize horizontal lines, some only slanted lines
- Higher level neurons can 'build on' the work done by other neurons
- In the image below, left to right, visual cortex is perceiving 'higher features'
 - Lower level neurons, visualize simple features like lines and shapes
 - Higher level neurons recognize complex shapes



Filters

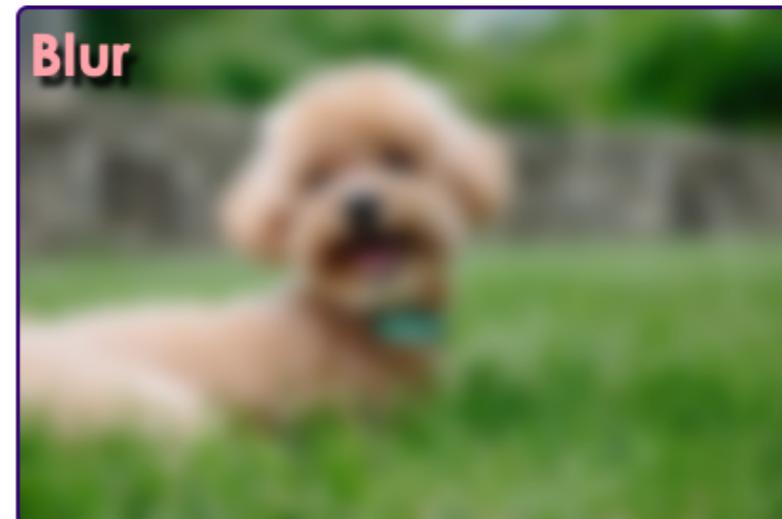
Pre-Processing Images

- Image processing is a standard task for machine learning
- Before we load the images, we can use Photoshop, OpenCV to "clean up" the image
 - remove noise
 - find edges
 - etc
- Even now, this is a common task to help get better results.
- This is *feature engineering*
(Example image, with background blurred)



Image Filter Examples

- Some sample filters
 - Blur filter
 - Sharpen filter
 - Edge detection filter



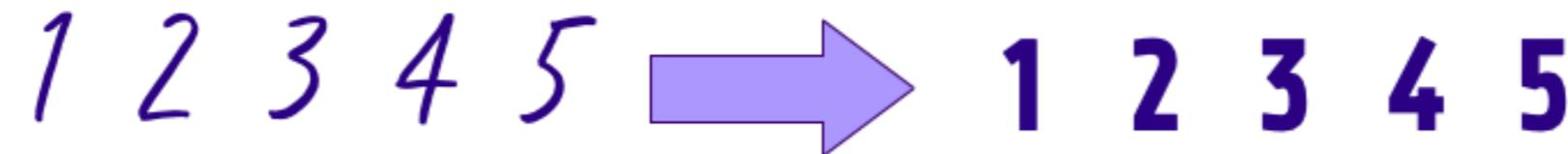
Problems with Image Pre-Processing

- How do we know that one filter will help us
 - Takes a lot of experience!
 - In some cases it might **hurt** rather than help.
- Lots of trial and error!
- What if...
 - Maybe there was a way we could find a filter that gives us better results for sure.
 - Could we automate finding the perfect filter?
 - Maybe if we used more than one filter?

Convolutions

A Little History: Yann LeCun

- Yann LeCun was working on the problem of recognizing digits: MNIST
 - Recognize zip codes in letters for US Postal Service
 - Recognize digits in bank checks
- Classical MLP networks were unable to get very high accuracy on the problem.
 - 96,97% was the best such networks could do.
 - LeCun proposed a new architecture that could be over 99% better.
 - The difference between 96% and 99+% is a big deal!

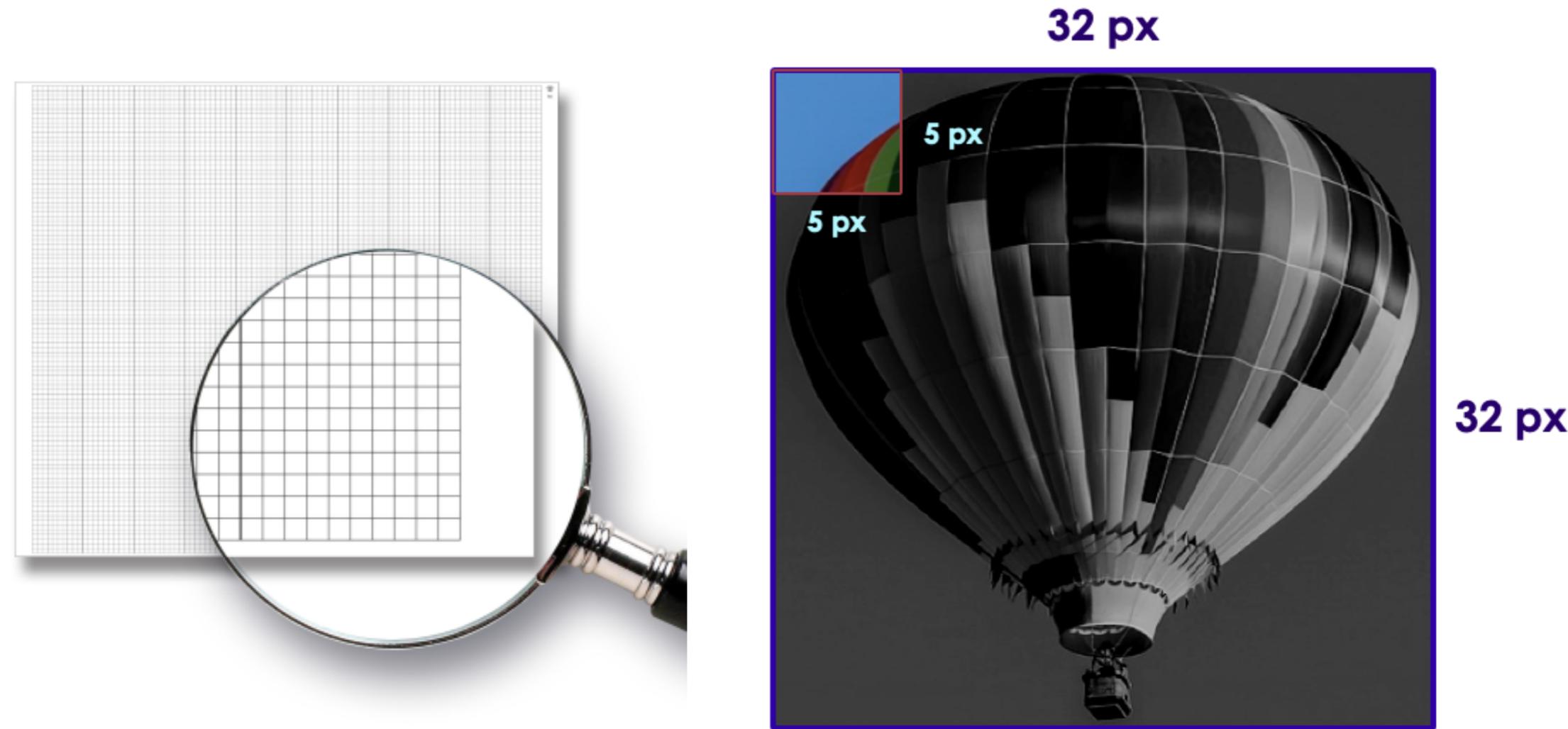


A Little History: Yann LeCun

- LeCun's new architecture is called LeNet (1998 paper)
 - Named after himself!
 - LeNet became the basis of a transfer learning architecture (we will discuss later)
- Lecun's architecture had 2 new elements
 - Convolutional layer
 - Pooling layer

Convolution

- Imagine a small patch being slid across the input image. This sliding is called **convolving** .
- It is similar to a flashlight moving from the top left end progressively scanning the entire image. This patch is called the **filter/kernel**. The area under the filter is the receptive field.



Convolution Example: Edge Detection

- Here we are applying an edge filter to an image
- The resulting image has edges 'highlighted'

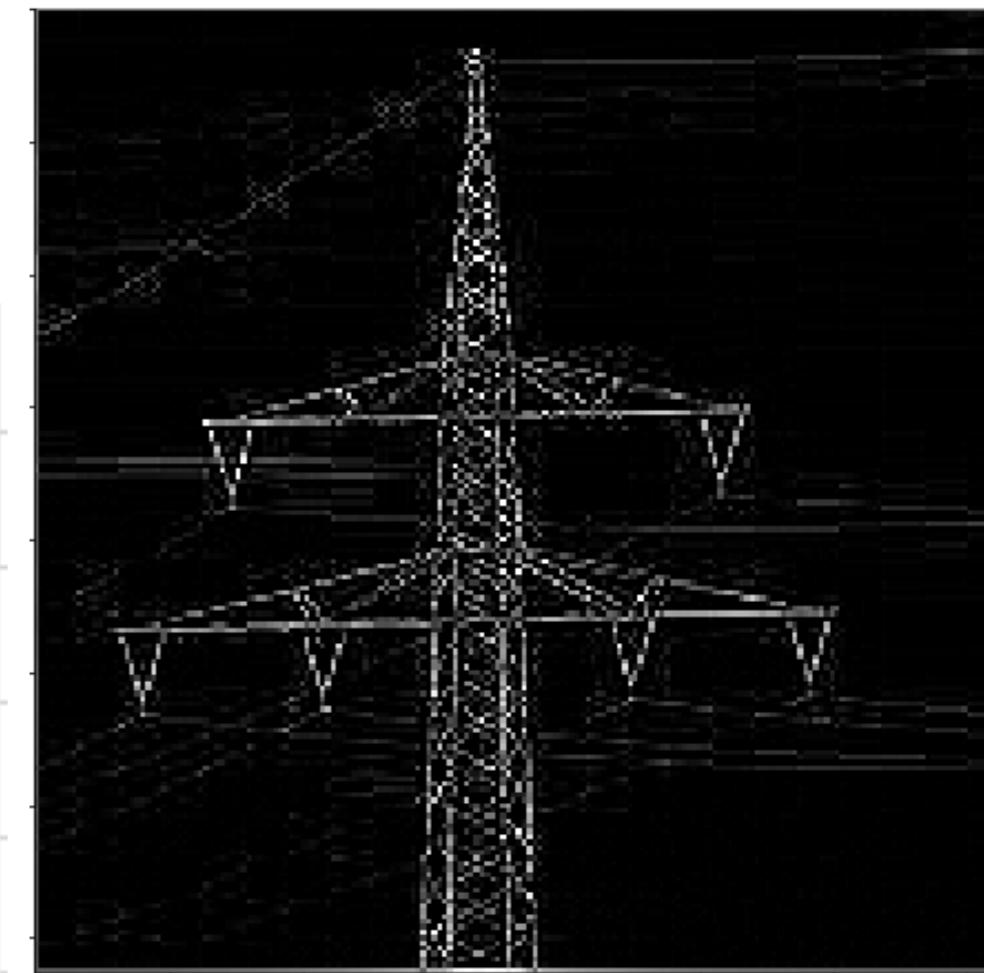
Input Image



Feature Map : Edges

Edge Filter

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$



Convolution Example: Vertical Edges Detection

- The resulting image has vertical edges highlighted

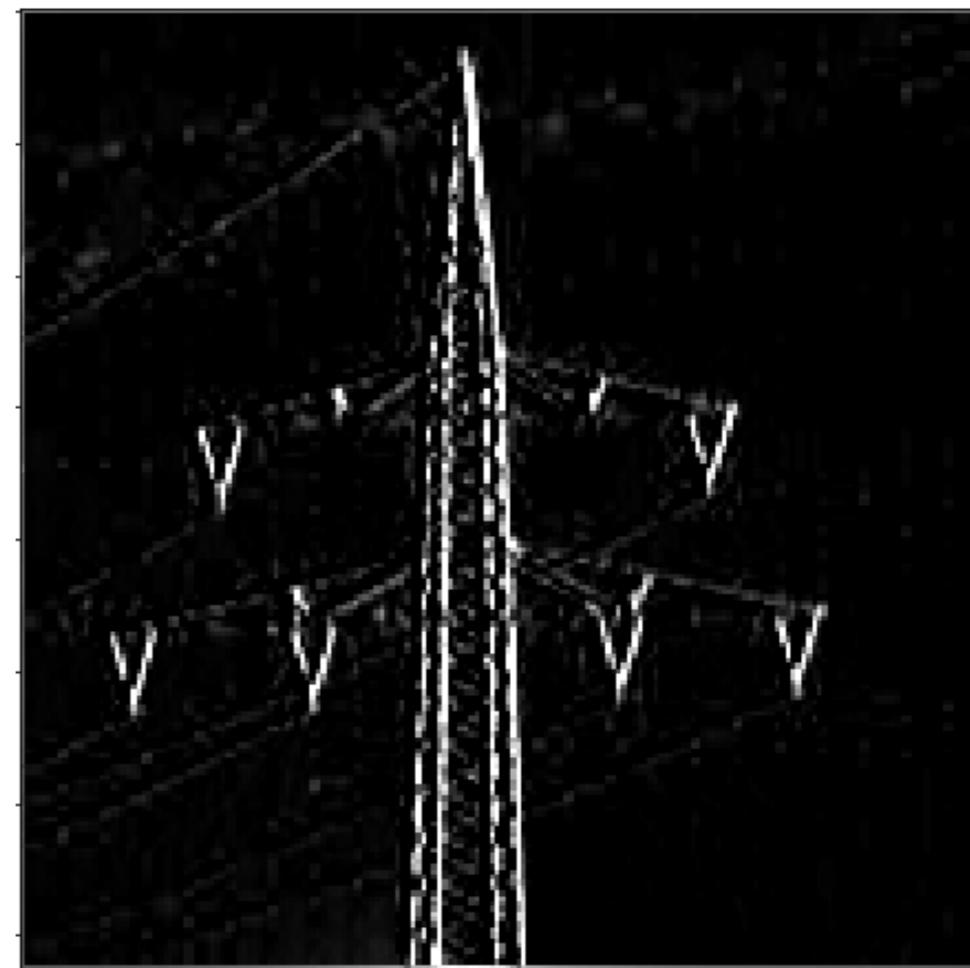
Input Image



Feature Map : Vertical Edges

Vertical Filter

$$\begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$$



Convolution Example: Horizontal Edges Detection

- The resulting image has horizontal edges highlighted

Input Image



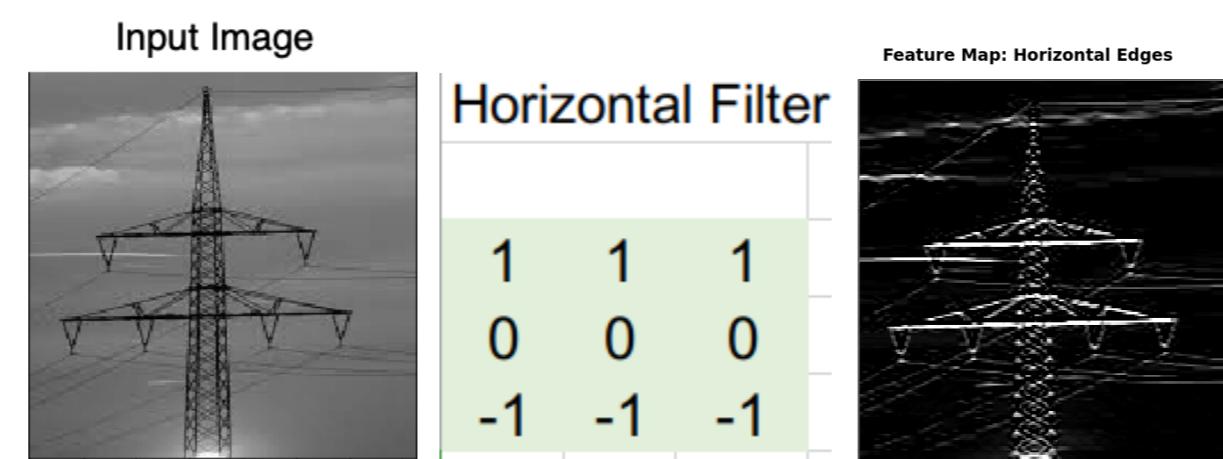
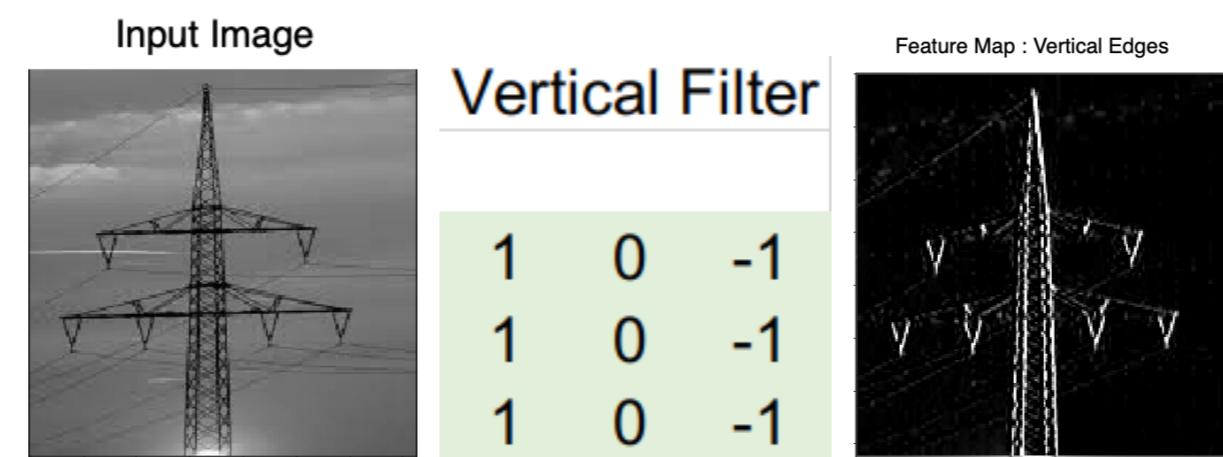
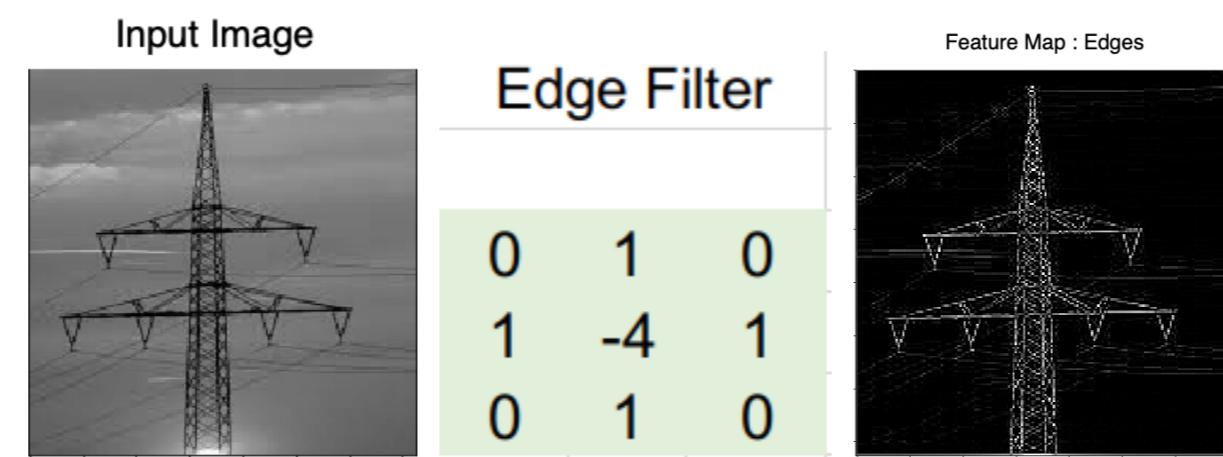
Feature Map: Horizontal Edges



Horizontal Filter

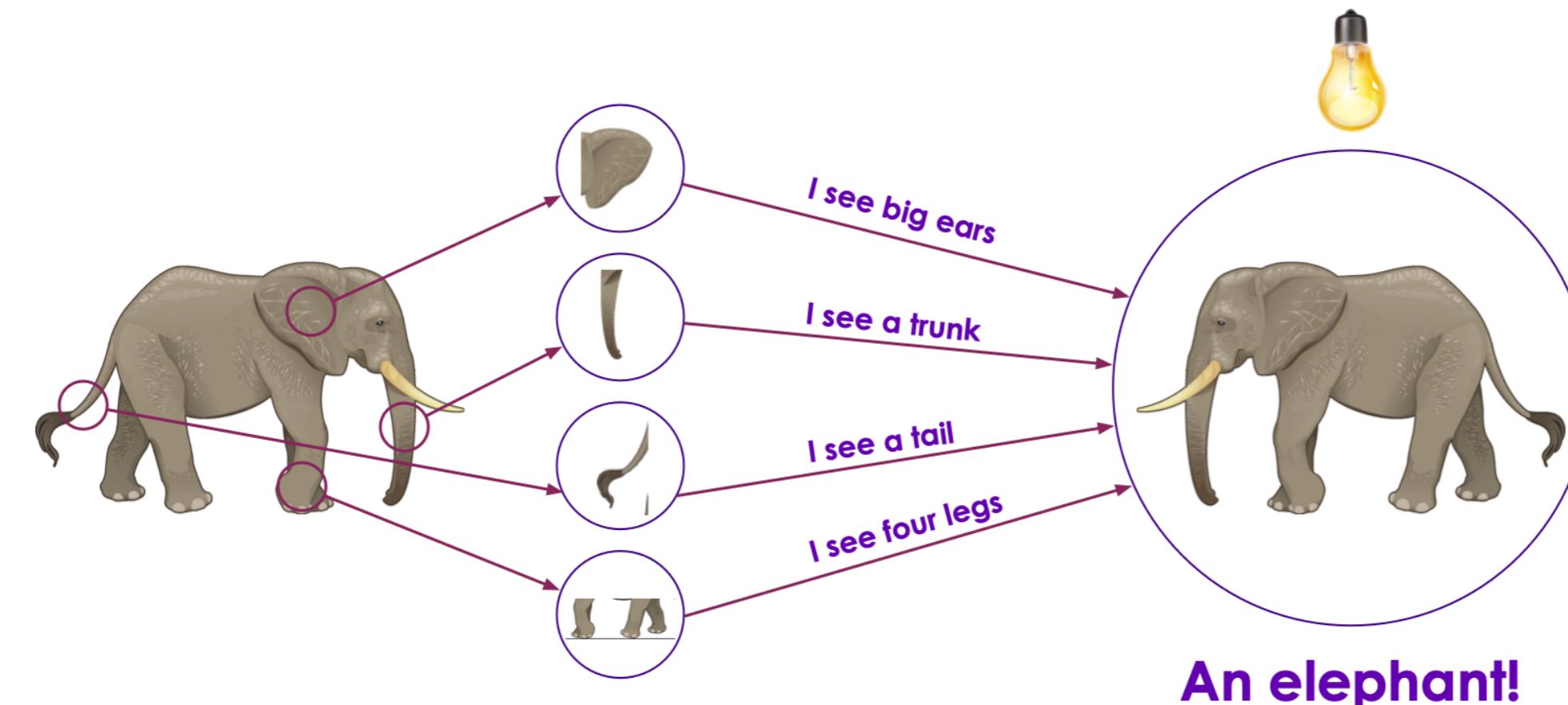
$$\begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{matrix}$$

Convolution Example: Edges



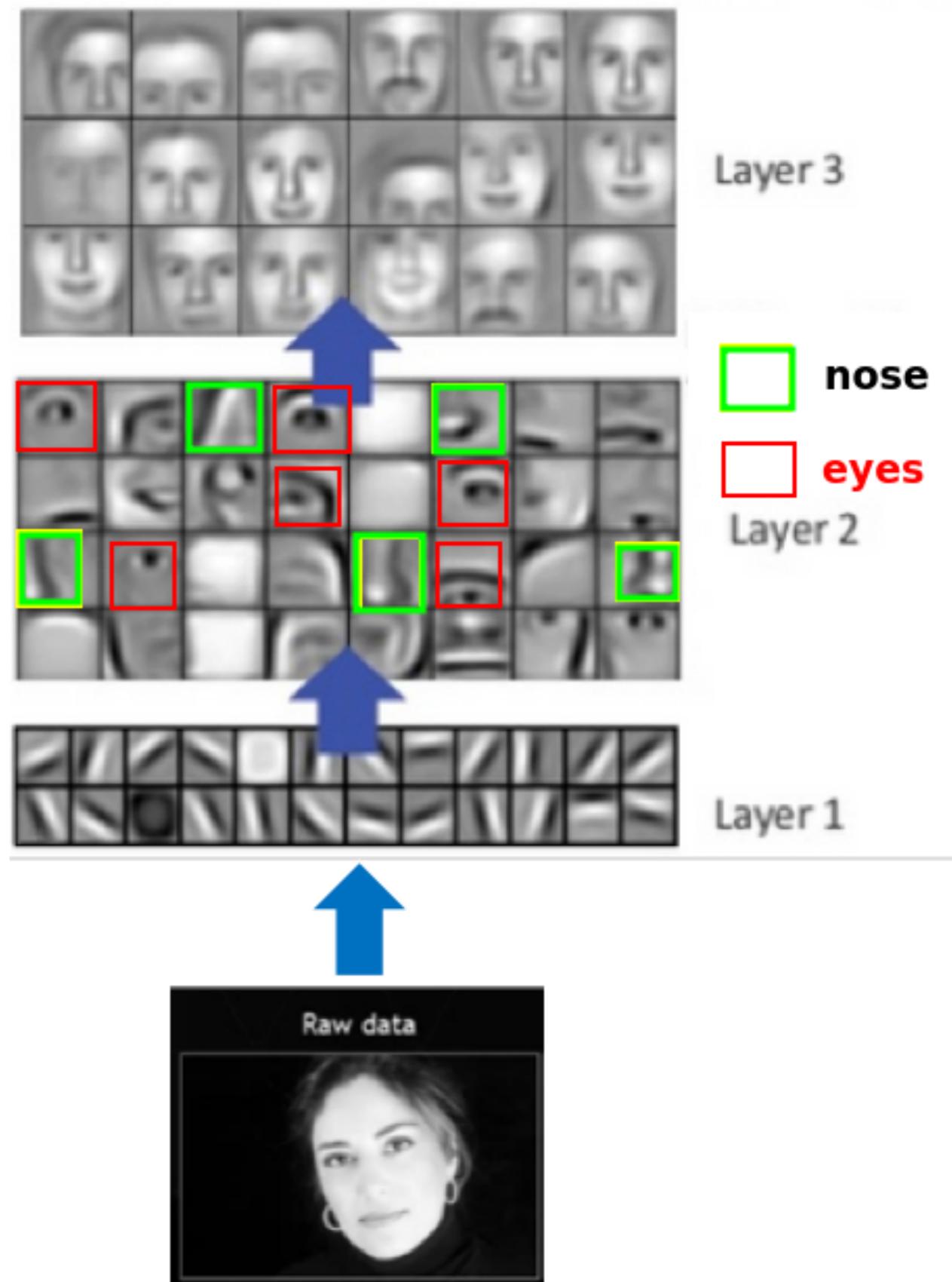
Learning Features

- Here we are analyzing a picture of elephant
- Assume each neuron is 'trained' recognize a certain feature (like ears / trunks / tail ..etc)
- As we scan through the image, the neurons will **'recognize the features'**
- And putting their findings together, we can conclude that we are seeing an elephant



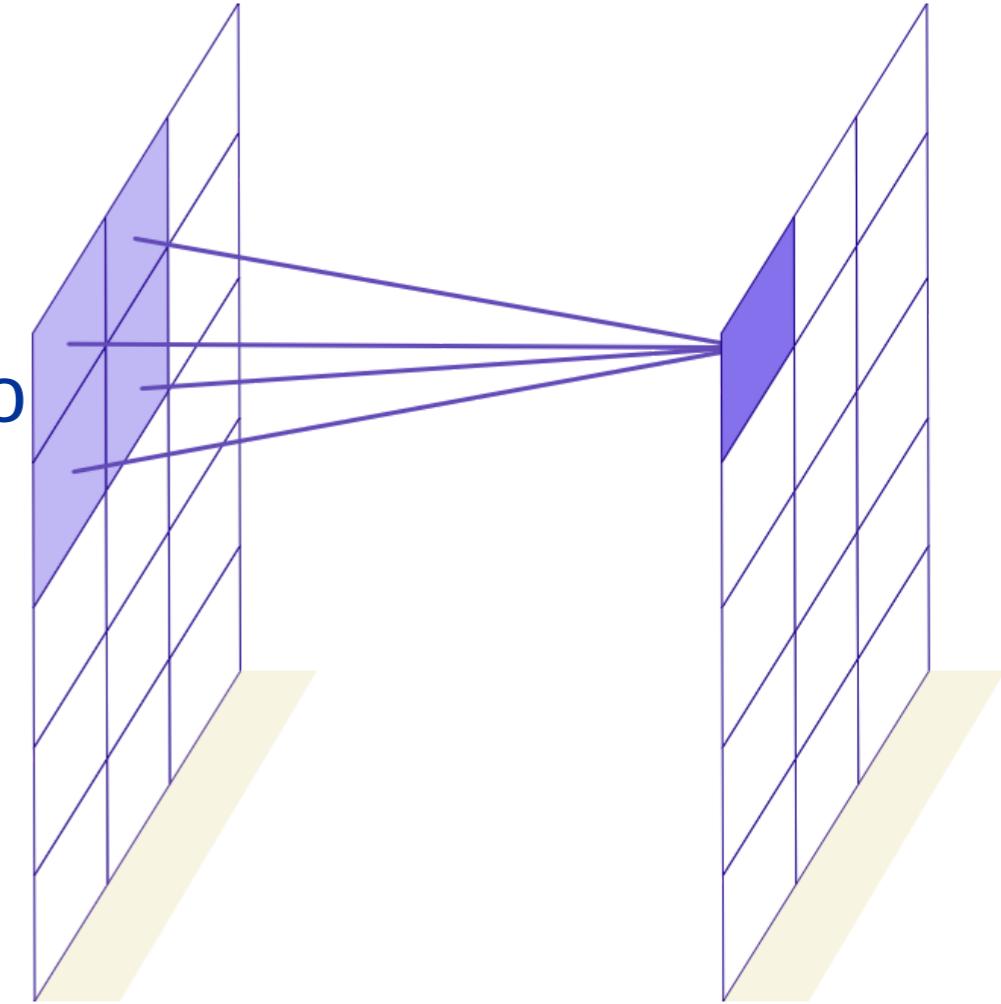
Learning Features

- Each layer builds on previous layer's work
- First layer detects simple shapes - horizontal lines, slanted lines ..etc
- Second layer recognizes more complex features: eyes / nose ..etc
- Third layer recognizes faces



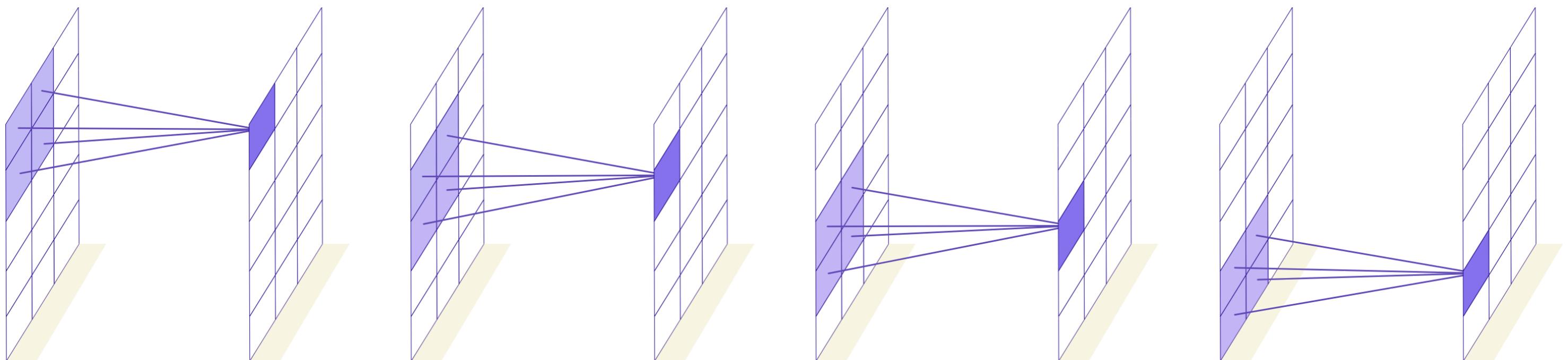
Convolutional Layer

- Here we represent our neurons in a 2D grid format (instead of linear before), this makes visualizing connections easier
- Neurons in the convolutional layer are NOT connected to every single neuron in the layer before
- Instead each neuron is connected to a few pixels/neurons in their **receptive field**
 - The idea is to detect local features in a smaller section of the input space, section by section to eventually cover the entire image.
- This allows the first convolutional layer to concentrate on low level features
- Next layers assemble inputs from previous layers into higher level features

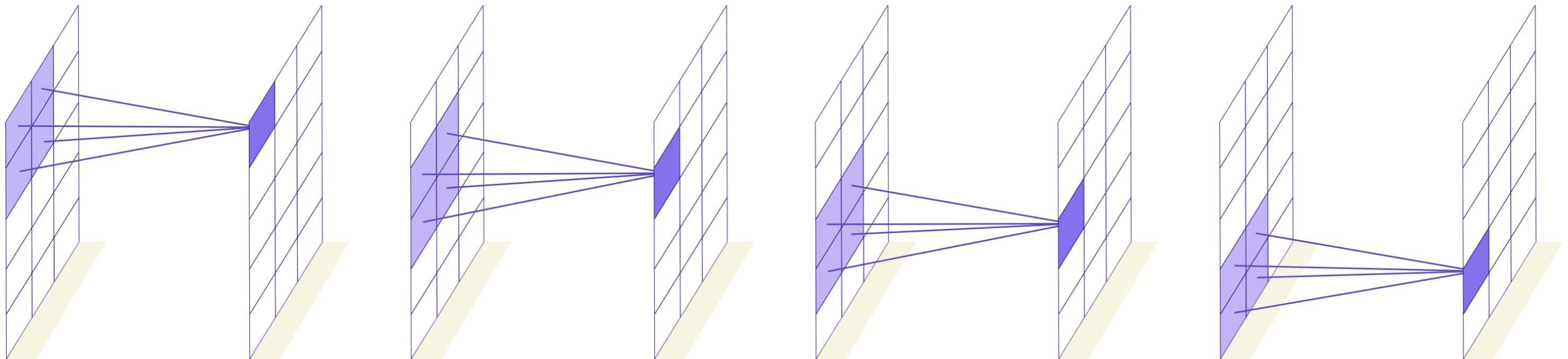
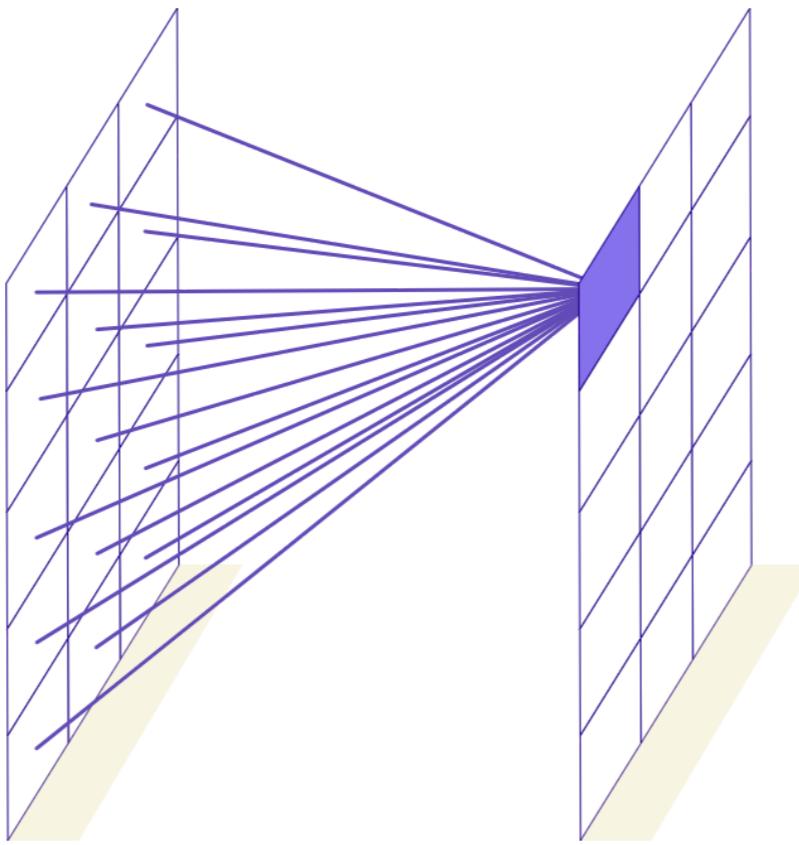


Convolutional Process

- Here we see that in second layer, the neurons only read from a few neurons from previous layer
- They read from their 'field of view'

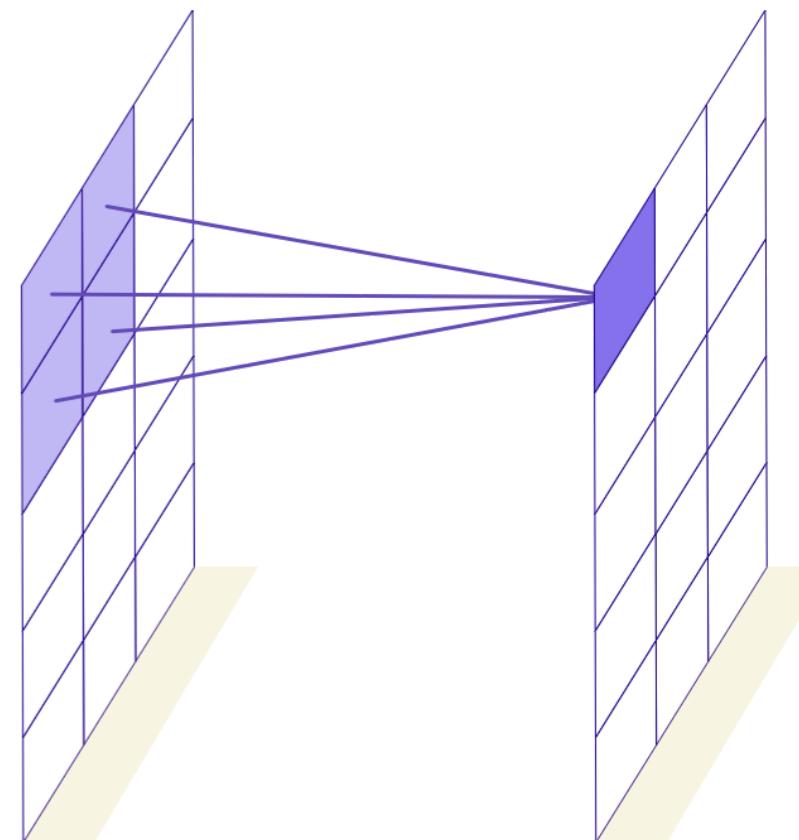
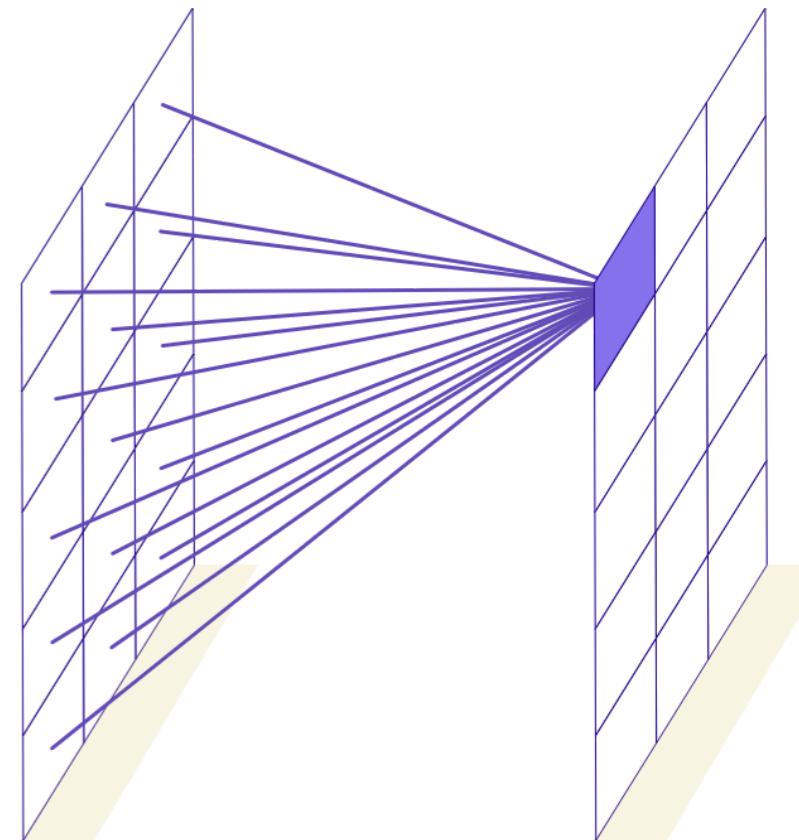


Fully Connected vs. Convolutional



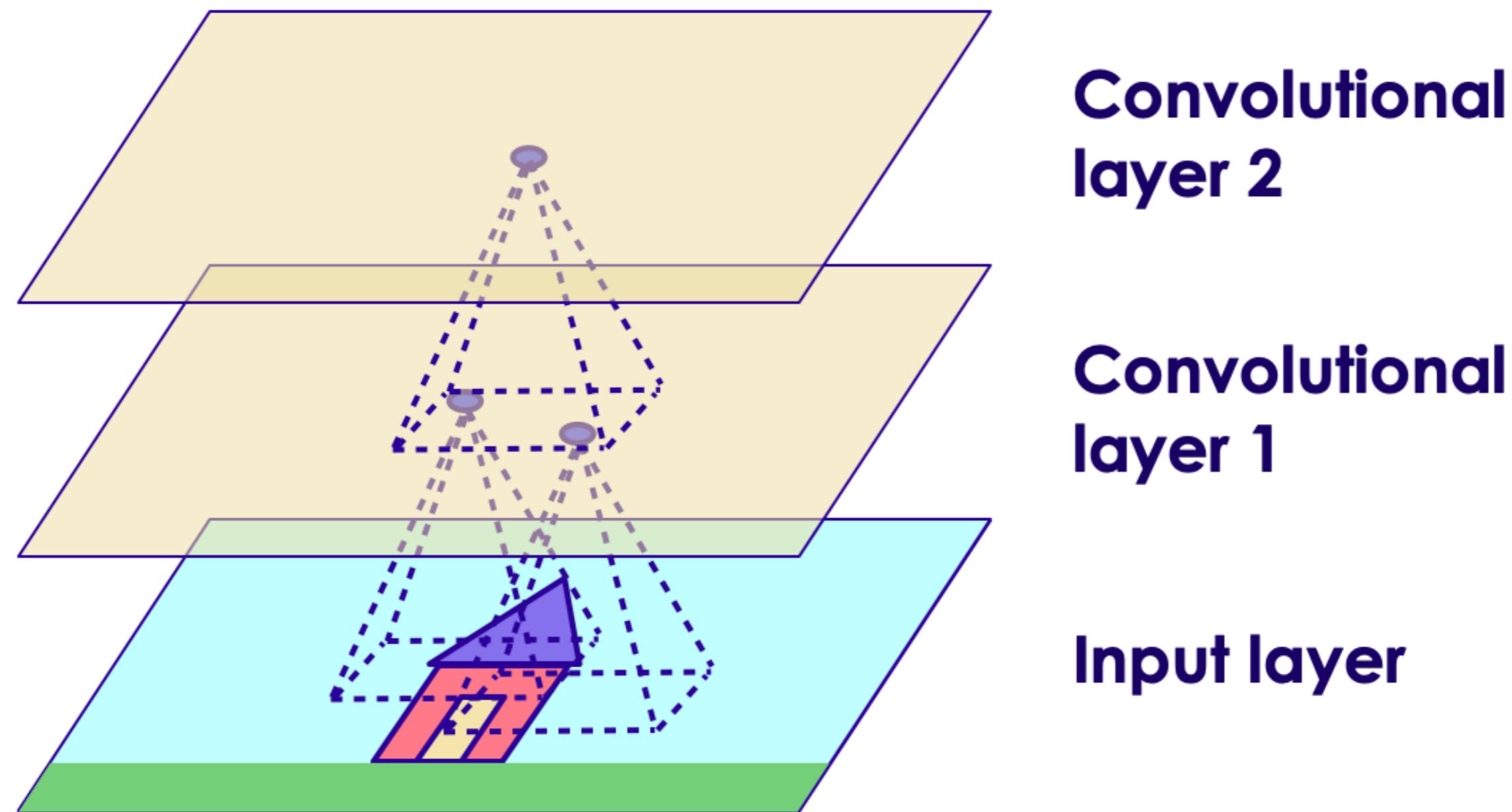
Fully Connected vs. Convolutional

- Fully connected layers connect to every neuron from previous layer
 - This results in too many connections
- But in convolutional layers, the number of connections is greatly reduced
 - Makes computationally feasible



Convolutional Layer: Local Receptive Field

- Here each neuron connects to neurons in its input / perceptive field



How is Convolution Performed?

- The following slides illustrate the math behind convolutions
- Provided as a reference
- Feel free to skip / go-over depending on the time constraint

Process of Convolution

- On the left we have the image matrix (6x6)
- On the right we have filter/kernel matrix (3x3). Also known as weight matrix (W_k)
- The weight matrix is a filter to extract some particular features from the original image. It could be for extracting curves, identifying a specific color, or recognizing a particular voice.

81	2	209	44	71	58			
24	56	108	98	12	112			
91	0	189	65	79	232			
12	0	0	5	1	71			
2	32	23	58	8	208			
4	23	2	1	3	9			

0								
1								
1								

Process of Convolution (Contd.)

- We use matrix multiplication to multiply input matrix vs. weight matrix
 $I \cdot W$
- When the weighted matrix starts from the top left corner of the input layer, the output value is calculated as:
 $(81 \times 0 + 2 \times 1 + 209 \times 1) + (24 \times 1 + 56 \times 0 + 108 \times 0) + (91 \times 1 + 0 \times 0 + 189 \times 1) = 515$

		Input	Layer		
81	2	209	44	71	58
24	56	108	98	12	112
91	0	189	65	79	232
12	0	0	5	1	71
2	32	23	58	8	208
4	23	2	1	3	9

0	1	1	
1	0	0	
1	0	1	

81	0	2	1	209	1
24	1	56	0	108	0
91	1	0	0	189	1

515	0	0	0
0	0	0	0
0	0	0	0

Process of Convolution (Contd.)

- The filter then moves by 1 pixel to the next receptive field and the process is repeated. The output layer obtained after the filter slides over the entire image would be a 4X4 matrix.
- This output is called an **activation map/ feature map**

		Input	Layer		
81	2	209	44	71	58
24	56	108	98	12	112
91	0	189	65	79	232
12	0	0	5	1	71
2	32	23	58	8	208
4	23	2	1	3	9

0		1	1
1		0	0
1		0	1

515	374	0	0
0	0	0	0
0	0	0	0

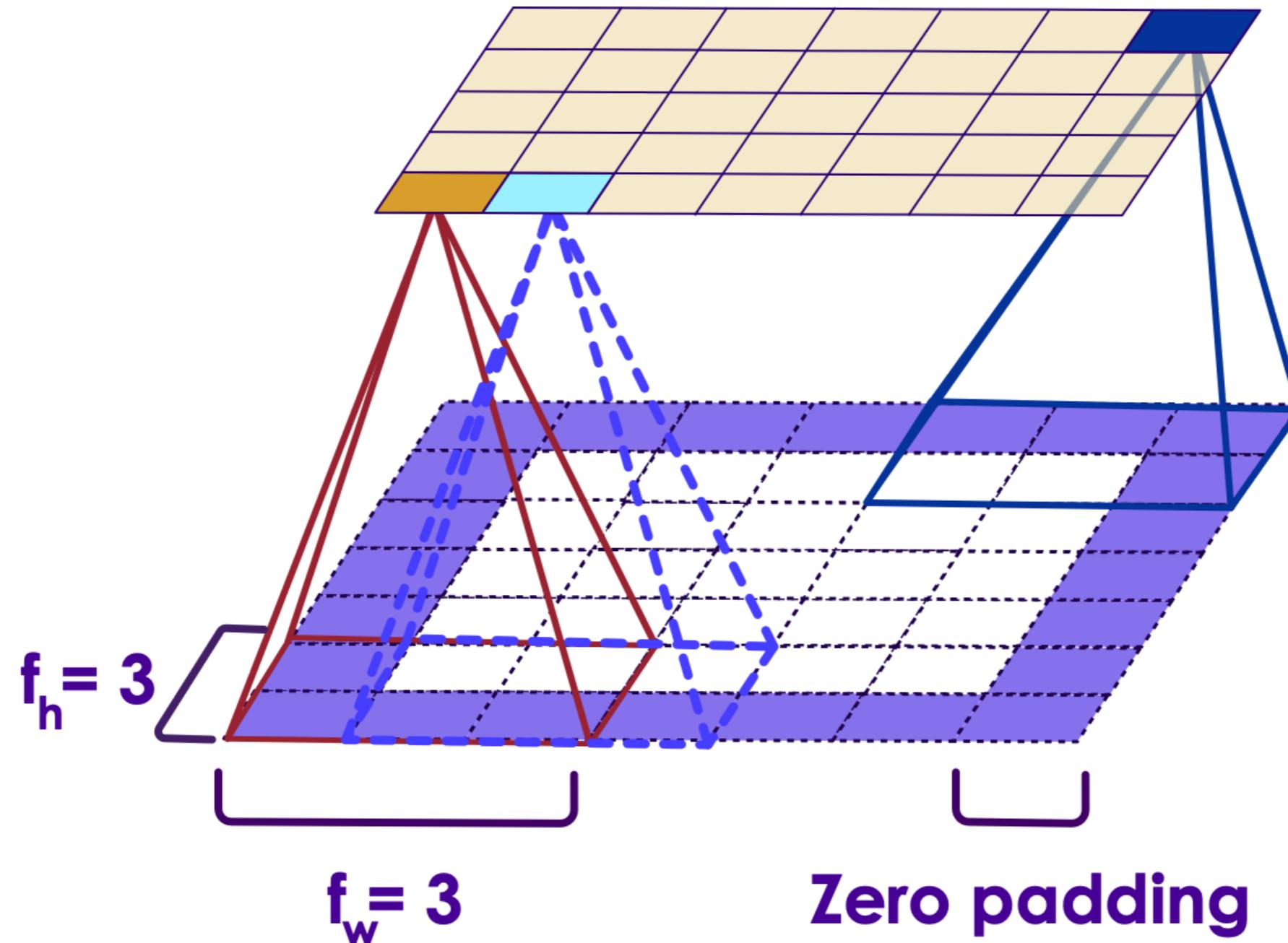
Stride

- The distance between two consecutive receptive fields is called the **stride** .
- In this example stride is 1 since the receptive field was moved by 1 pixel at a time.

		Input	Layer		
81	2	209	44	71	58
24	56	108	98	12	112
91	0	189	65	79	232
	← 12 Stride	0	0	5	1
2	32	23	58	8	208
4	23	2	1	3	9

Zero Padding

- It is common to add zeros around the image (black pixels)
- Called 'zero padding'
- This ensures 'corner pixels' are processed properly



Zero Padding

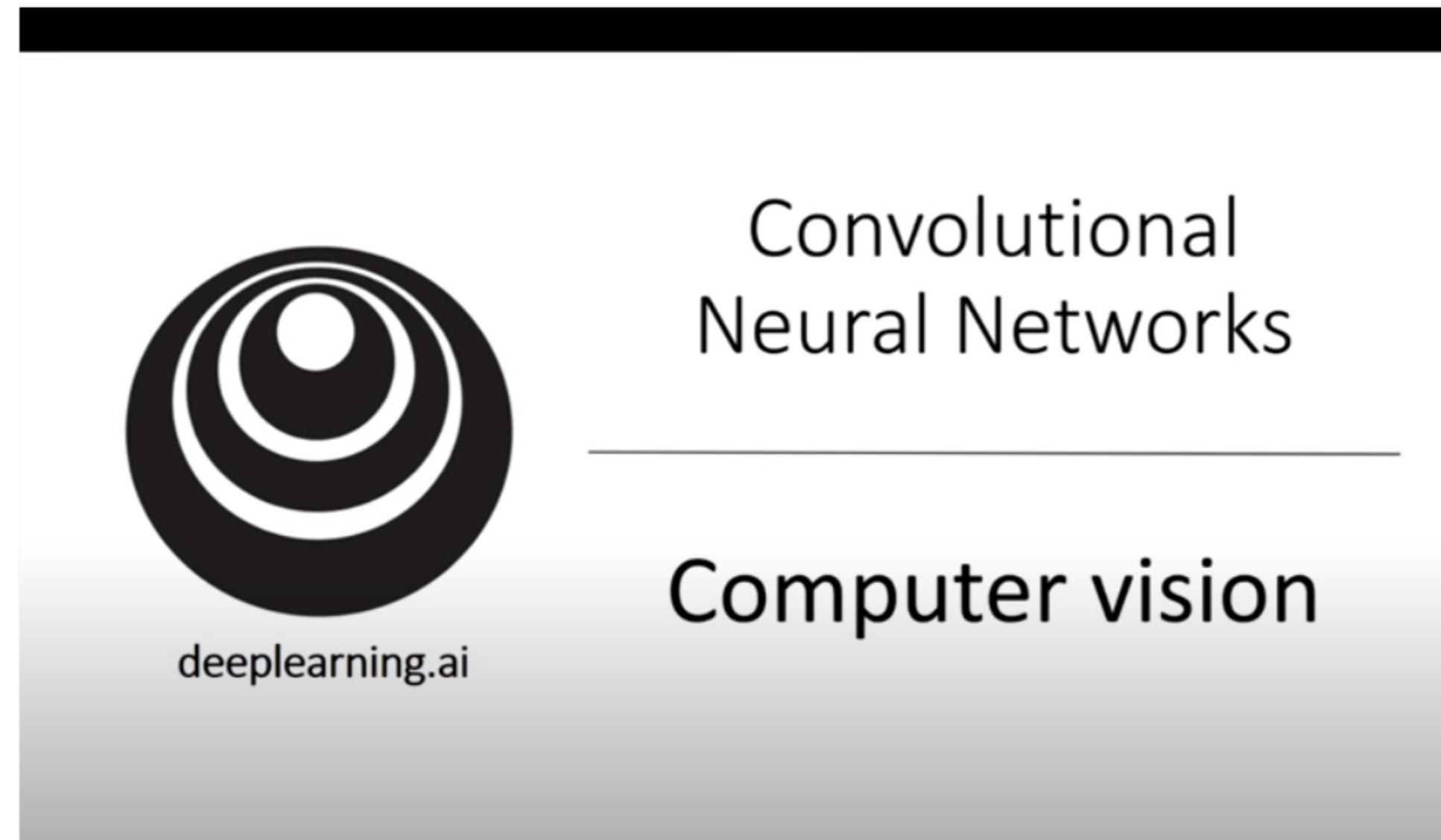
- Assumptions
 - Output image is $32 \times 32 \times 3$
 - Filter size is $5 \times 5 \times 3$
- To achieve the **same** size:
 - You need padding of $(K - 1) / 2$
 - In this case $(5 - 1) / 2 = 2$
- We pad a "frame" around the image
 - black pixels
 - size 2
 - Image is then $36 \times 36 \times 3$
- Output Size:
 - $O = ((W - K - 2P) / S) + 1$

Convolution Parameters

- Three Hyperparameters control the convolution:
 - **Depth:** The number of filters, and the number of neurons per convolution
 - **Stride:** Usually 1 or 2: the number of pixels we "jump" when applying the filter.
 - **Zero Padding:** Creates a "frame" of zero (black) pixels around the border of image.
- Stride > 1 will reduce dimensions
 - Allows us to both do convolution and reduce dimensions in one step
 - Usually, we use pooling layers for reducing dimensions.

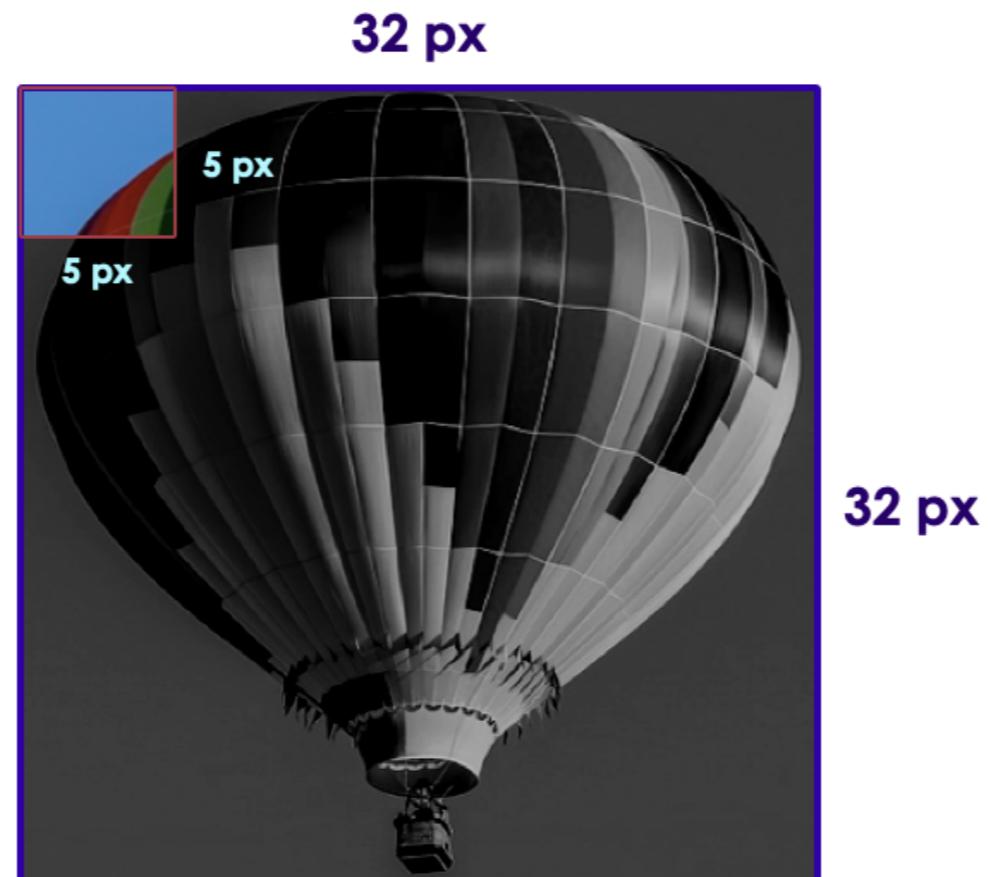
Convolutions: Further Reading

- Tutorial on convolutions by Andrew Ng
- Tutorial on padding by Andrew Ng
- Tutorial on strides by Andrew Ng



Filters / Kernels

- CNNs use filters to detect patterns in images
- Imagine the filter like a flashlight shining on the image
- As 'the flashlight' 'moves' along the image, it 'convolves' the image
 - Output from a filter is called **feature map**
- CNN learns filters during training phase

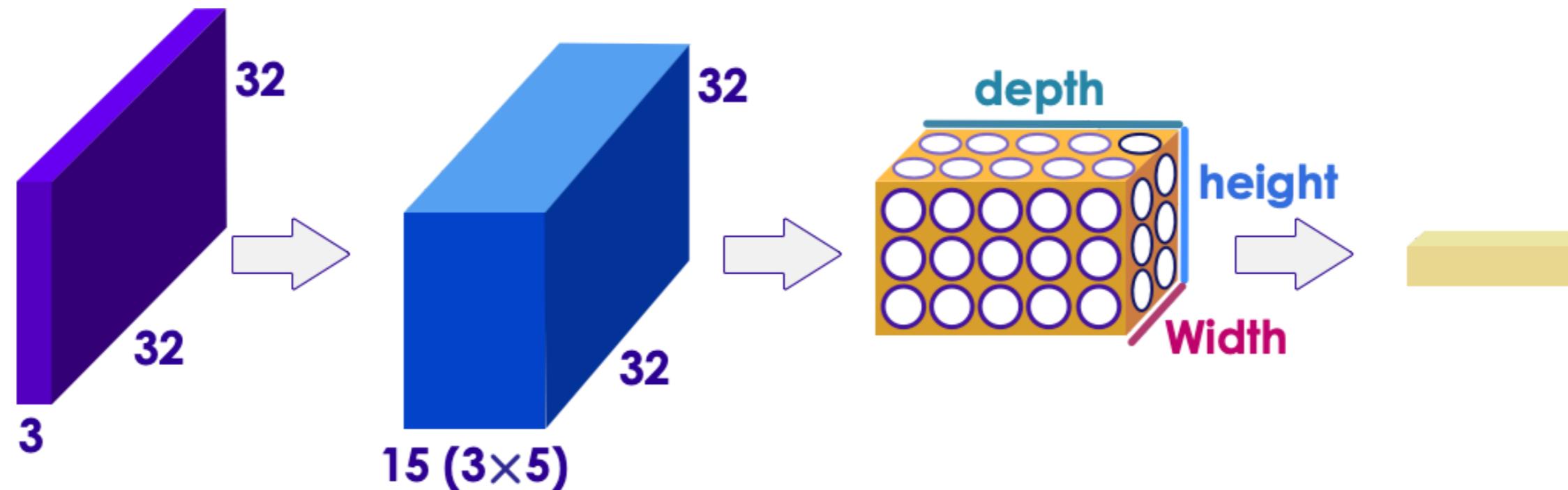


Filters in Computer Vision

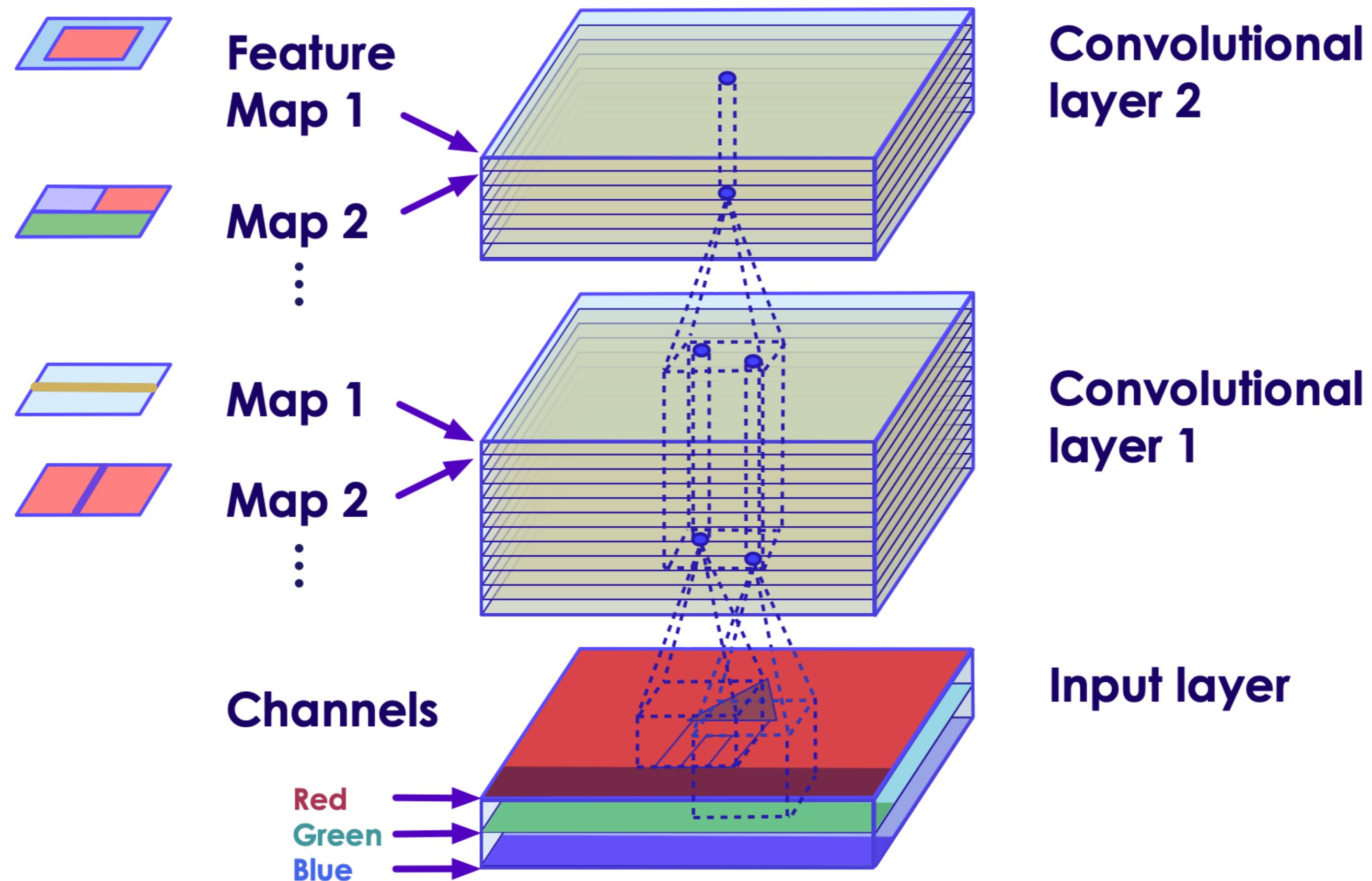
- We just saw a single filter can extract a certain **pattern or feature**
- Convolutional neural networks (CNN) do not learn just a single filter. In fact, they learn multiple features in parallel for a given input
- Typically in practice CNNs learn from 32 to 512 filters
- This gives the model 32, or even 512, different ways of extracting features from an input
- In other words CNNs "learn to see" features from given input

Stacking Feature Maps

- So far, we have seen Convolutional layer as a thin 2D layer
- In fact, **Convolutional layer is composed of series of feature maps**
- These feature maps are 'stacked' forming a 3D structure
- Each filter creates a new volume slice
- Typically have more than one slice (as images tends to have multiple channels like RGB)
- Here we see
 - 3 channels \times 5 filters = 15 feature maps

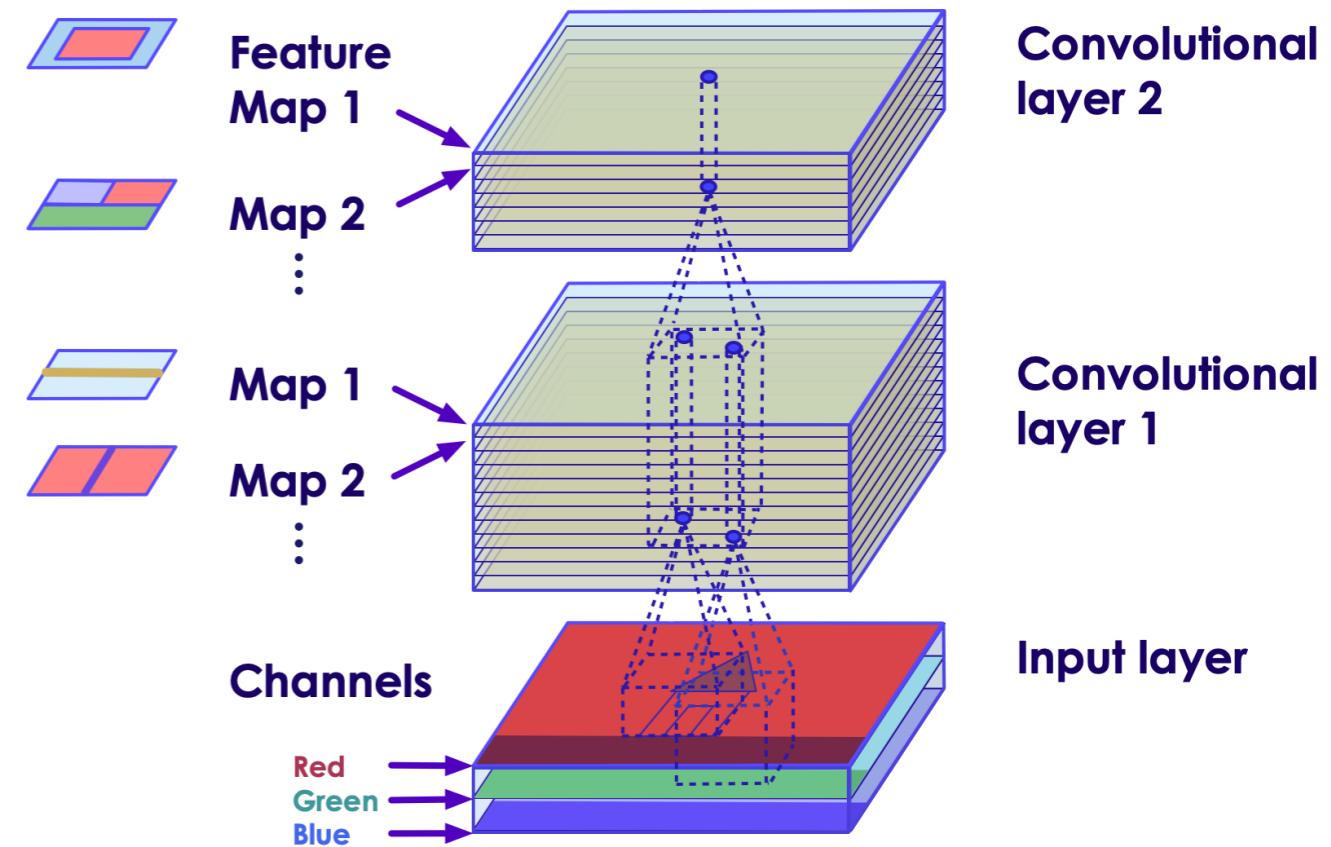


Stacking Feature Maps



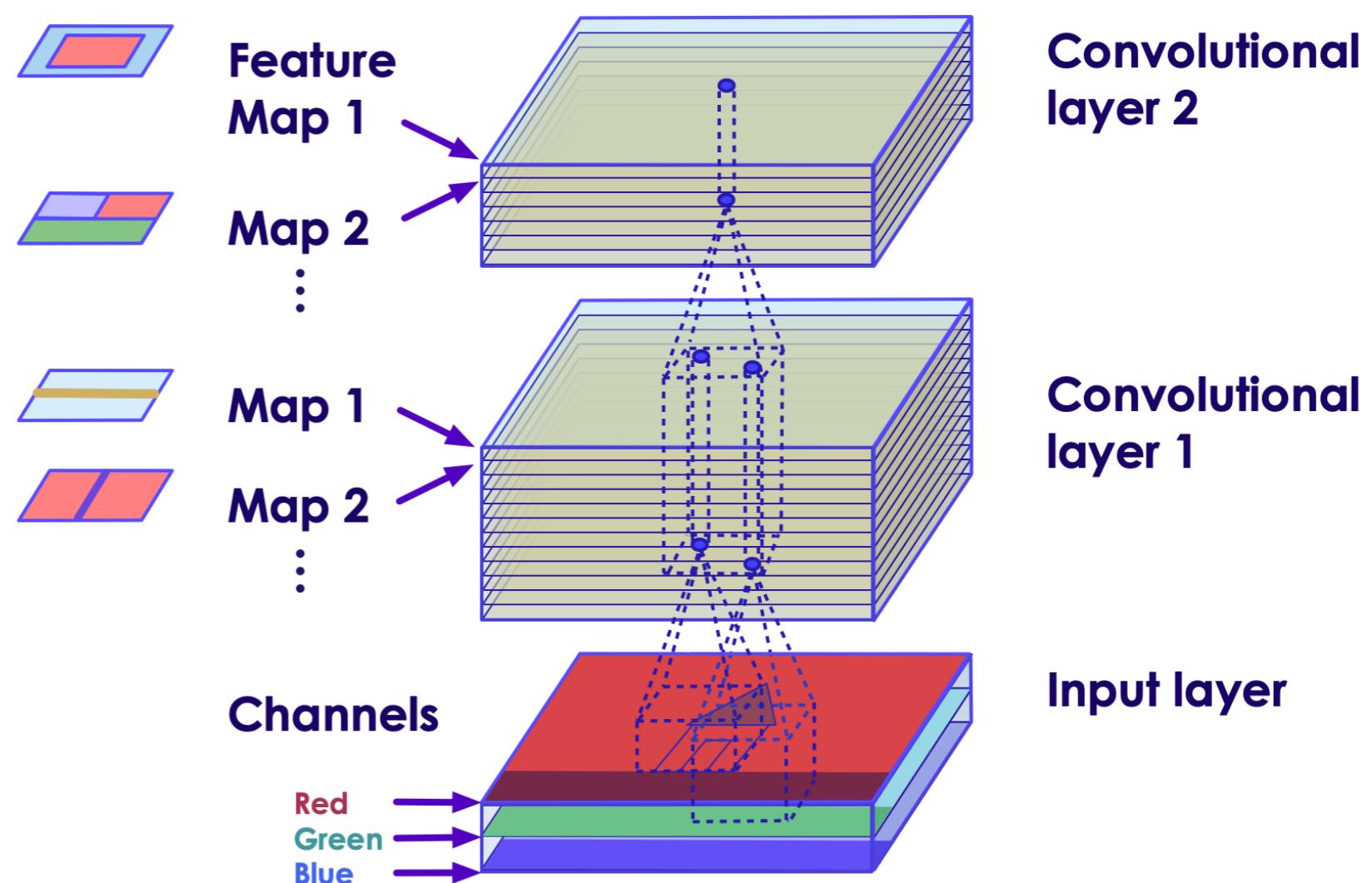
Stacking Multiple Feature Maps

- Within a single feature map all neurons share the same parameters (weights and bias term)
 - This simplifies training
- Different feature maps can have different parameters (weights + biases).
- The receptive field of previous layer extends across all of its feature maps.
- In summary, a **convolutional layer applies multiple filters to the input image, making it capable of detecting multiple features in the input**



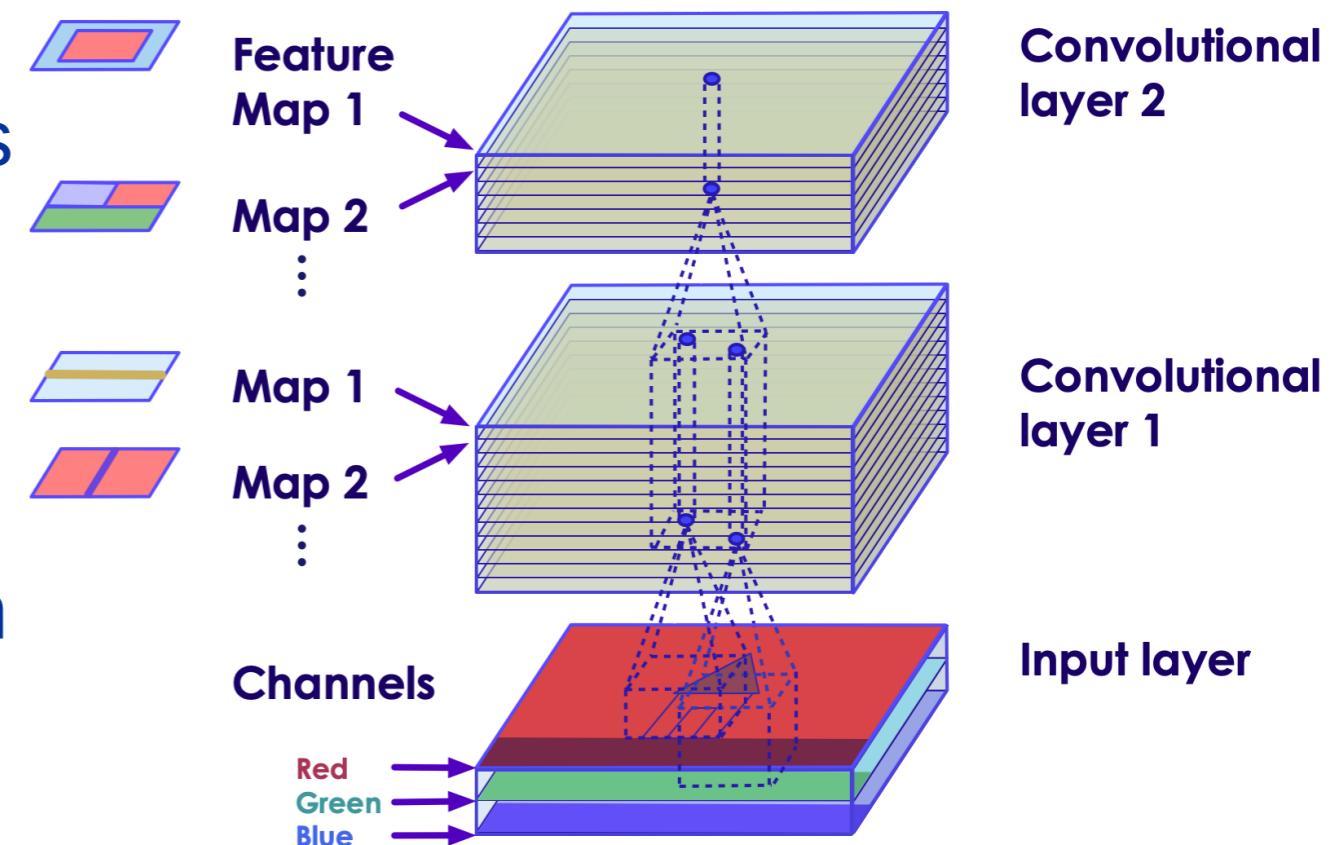
Stacking Multiple Feature Maps (Contd.)

- Images that are greyscale have just one channel. So it needs just 1 sublayer. Colored images have three channels - Red, Green and Blue. So it needs 3 sublayers.
- Satellite imagery that capture extra light frequencies (e.g. infrared) can have more channels.



Stacking Multiple Feature Maps (Contd.)

- The fact that all neurons in a feature map has just one set of parameters dramatically reduces the number of parameters needed.
- This also means that once a CNN has learned to recognize a pattern in one location, it can recognize it in any other location. This is known as location invariance.
 - For example it can detect an 'eye' anywhere in the picture
- In contrast, if a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that location.



Convolutional Layer Hyperparameters

- The hyperparameters of CNN are:
- **Padding type**
- **Filter height and width**
- **Strides**
- **Number of filters**

Convolutions: Further Reading

- Tutorial on convolutions on volumes by Andrew Ng



deeplearning.ai

Convolutional
Neural Networks

Computer vision

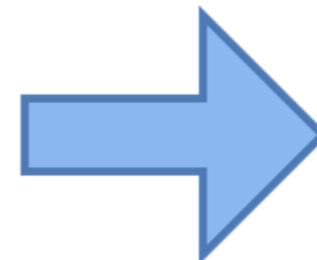
Convolutions Summary

- Images have too many features for a network to learn.
- Convolutions are a standard way that we do image processing
- Convolutional Layers help to extract higher-level features
- Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel
 - It means we take a $n \times n$ *filter* and apply that to the image
- Different filters do different things:
 - Edge Detection filters
 - Sharpening Filters
 - Gaussian blur

Pooling

Pooling Layer

- Pooling layer is used to subsample (i.e., shrink) the input image, while **keeping important features intact**
- Reduces the computational load, reduce memory usage, and reduce the number of parameters (limits overfitting)
- Here we are shrinking the picture, while still keeping the prominent feature (lighthouse)

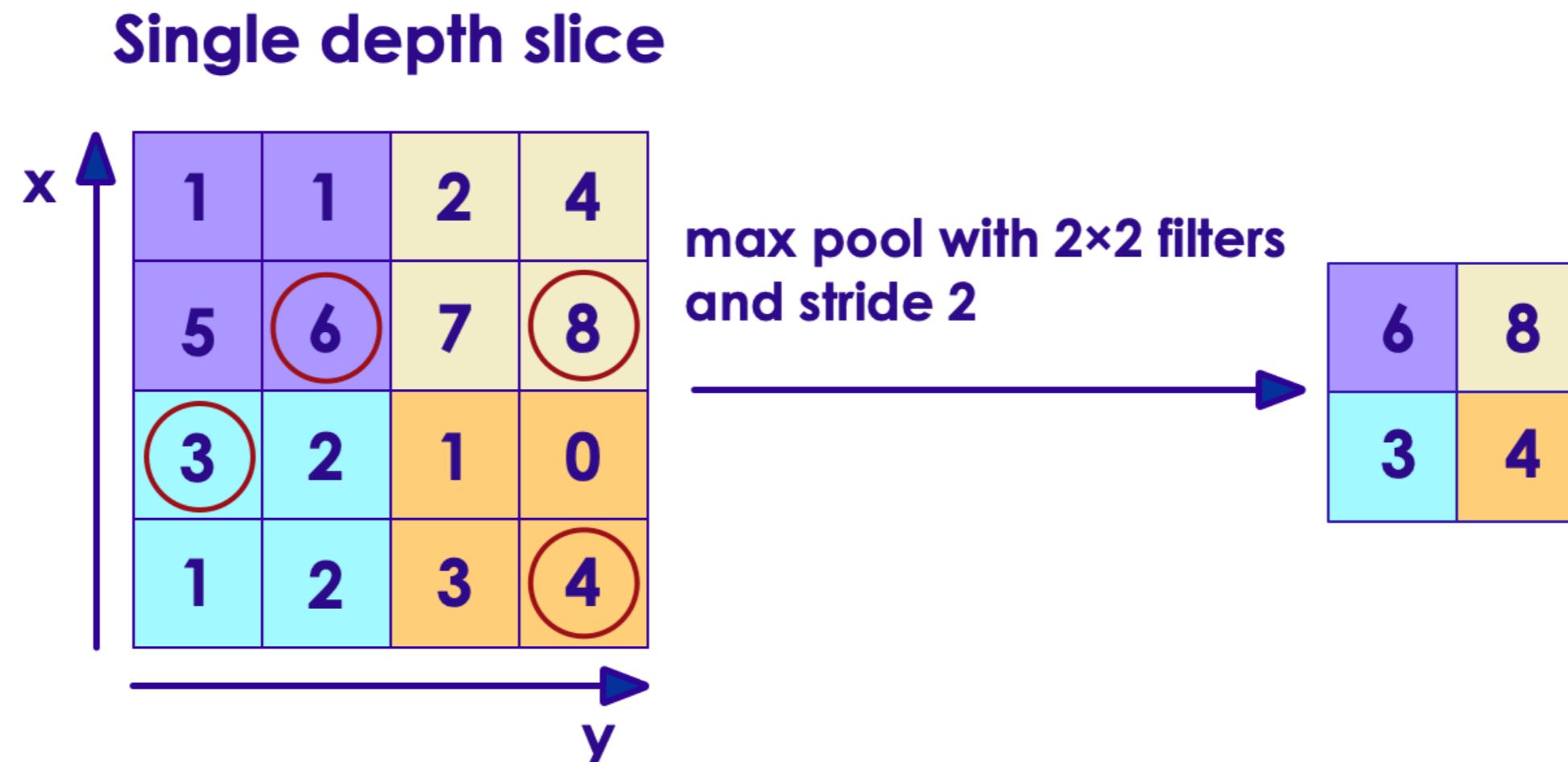


Pooling Types

- Pooling types
 - MAX pooling
 - Average pooling
 - L2-Norm pooling
 - Stochastic pooling

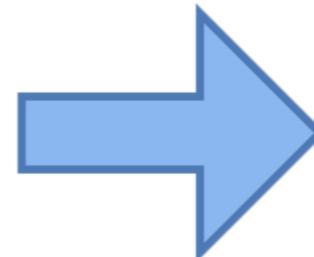
Max Pooling

- MAX pooling is most common.
- Notice in this case we take the max of each receptive field
- 2x2 window with stride 2
- Here input is shrunk by factor of 4 --> 1
so the resulting image is 25% of original image



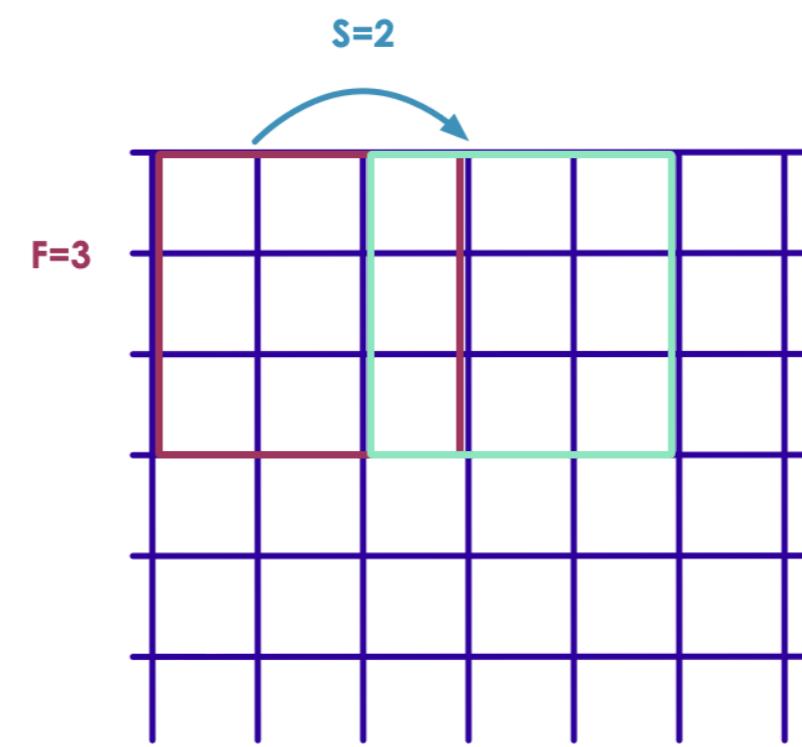
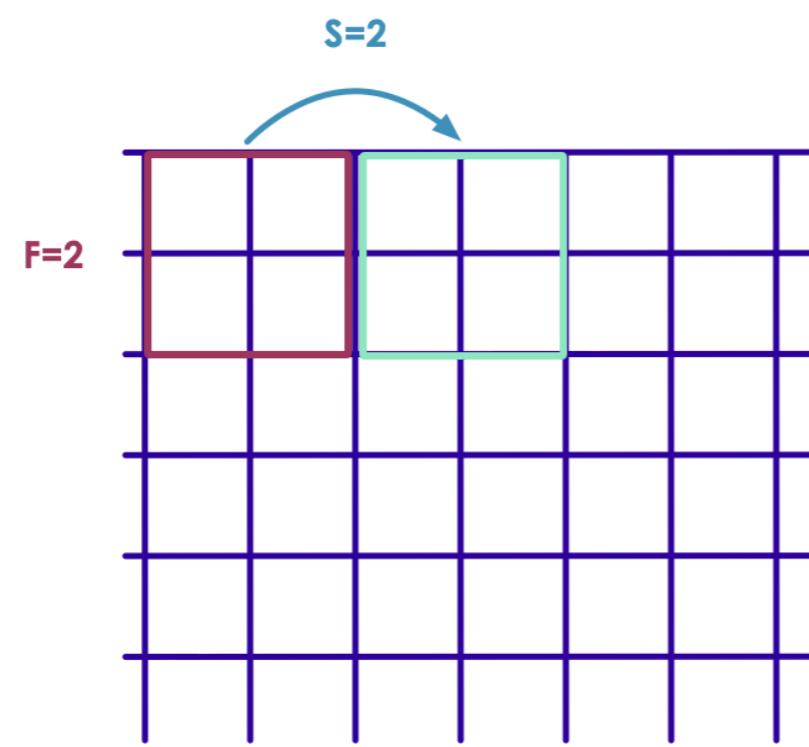
Max Pooling Example

- With 2x2 window, stride=2, and max pooling, the image is reduced to 25% of original size



Pooling Hyperparameters

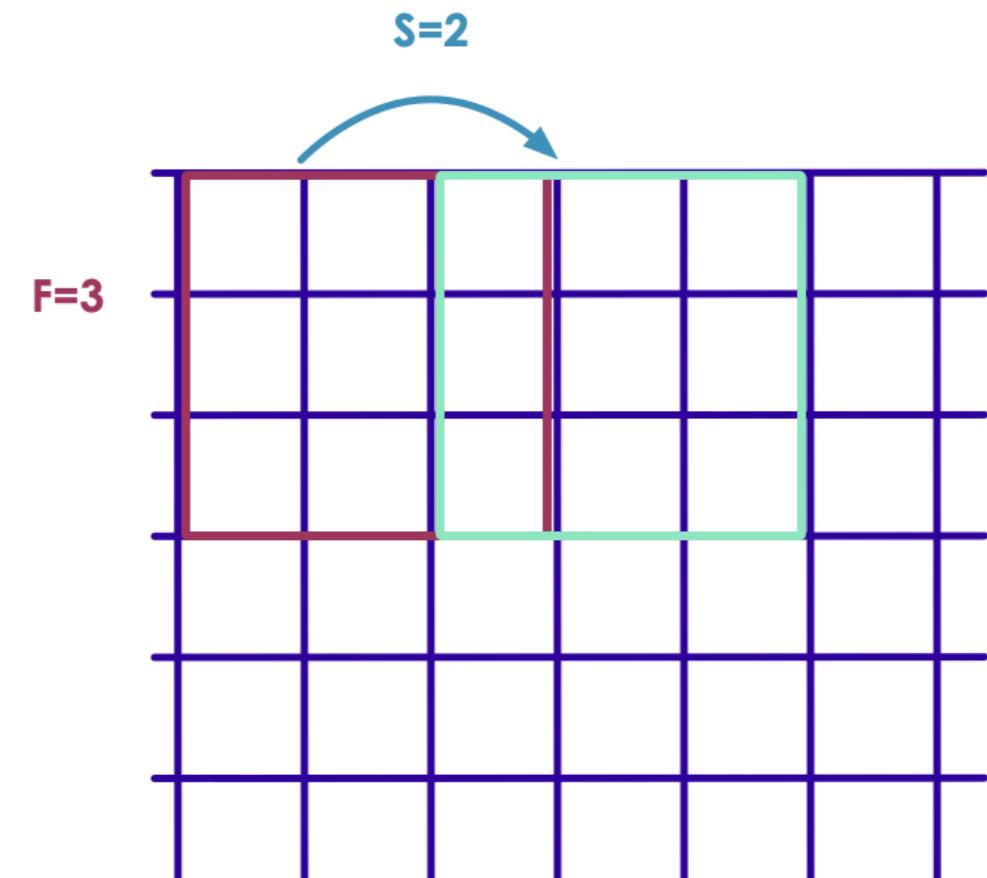
- Filter / Spatial Extent F: (for example, $F = 2$ is 2×2 filter)
- Stride S, how many pixels we "move" (Commonly 2)
- We do not use Zero Padding (black pixel padding) with pooling layers.
- Common Parameters:
 - $F = 3, S = 2$: 3×3 filters, stride 2 : Overlapping pooling
 - $F = 2, S = 2$: 2×2 filters, stride 2: No overlaps



Do We Need Pooling?

- We need to reduce dimensionality somehow!
- Possible to use CONV Layer with Stride=2 instead of a pool.
- We can provide the stride to the convolutional layers to reduce features
- Pooling layers have *traditionally* seemed to help make a network more efficient.
 - particularly with max pooling
- Recent trends have led to all convolutional networks
 - Use a convolution layer with stride greater than 1

		Input	Layer			
81	2	209	44	71	58	
24	56	108	98	12	112	
91	0	189	65	79	232	
	12 Stride 0	0	5	1	71	
2	32	23	58	8	208	
4	23	2	1	3	9	



Pooling: Further Reading

- Tutorial on pooling by Andrew Ng



deeplearning.ai

Convolutional
Neural Networks

Computer vision

Lab: Explore Convolutions and Pooling

- **Overview:**

- Understand convolutions and pooling

- **Approximate run time:**

- 15-20 mins

- **Instructions:**

- Instructor: Please demo **convolutions** lab



Convolutional Neural Networks (CNNs)

Introduction to Convolutional Neural Networks (ConvNets)

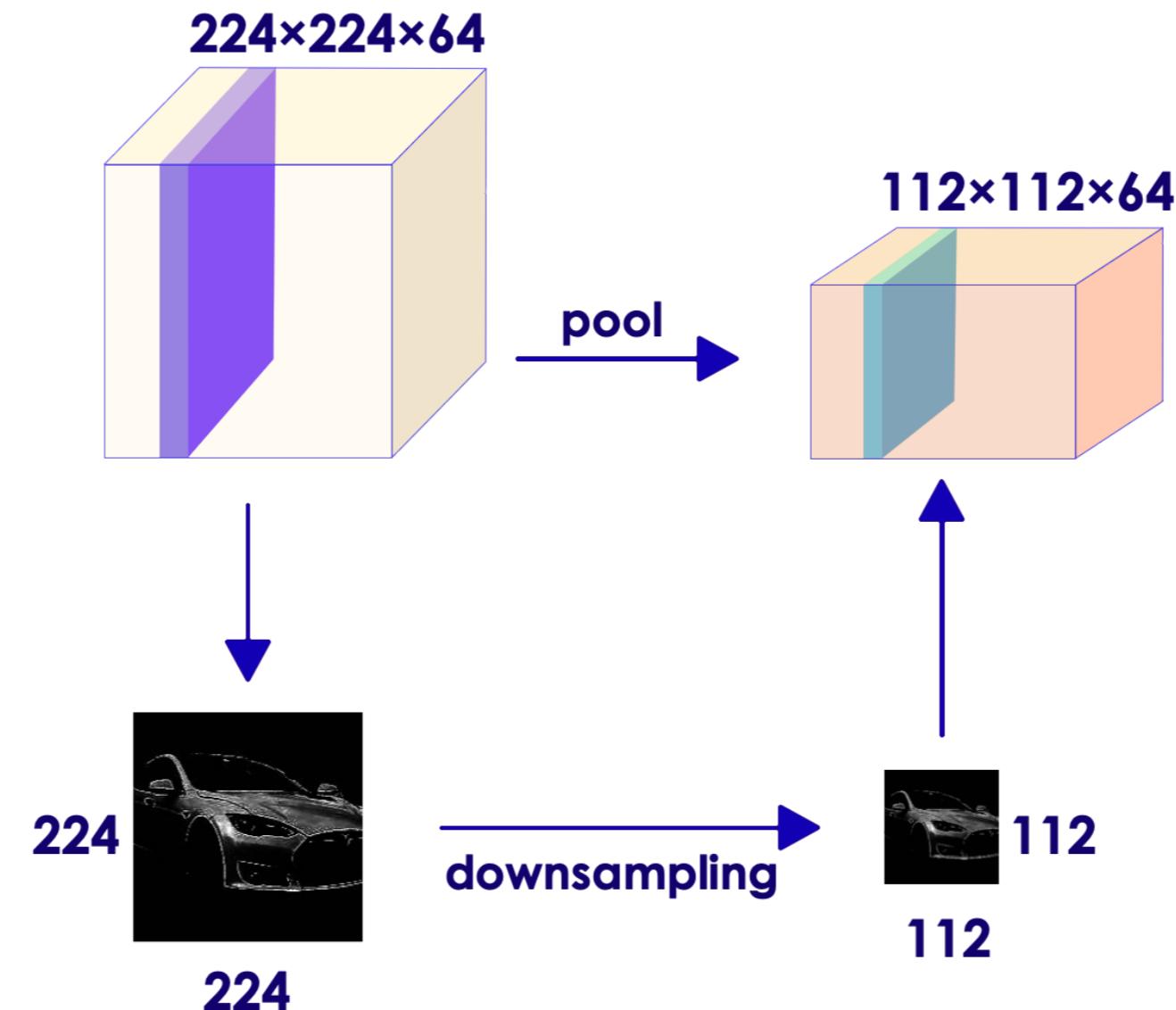
- CNNs are a sequence of layers:

- Input layer
- Convolutional Layer
- ReLU (Rectified Linear Unit) Activation
- Pooling Layer
- Fully Connected Layer(s)
- Many times we have more than one sequence of layers



Pooling Layer

- Apply some function (commonly MAX) to each $n \times n$
- Reduce features!
- Notice the reduced features.
- We can also reduce features simply by resizing the image.



Fully Connected Layers

- Fully Connected Layer
 - Finally, have one or more fully connected layers at the end
- Why Fully Connected?
 - Need to take the convolutional layers and apply to our problem
- Softmax Layer at the end (for classification problems)
 - Classify digits into 0,1,...9
 - Classify images into cats, dogs, elephants



Convolutions Further Reading

- Watch this brilliant [tutorial series](#) on convolutions by Andrew Ng



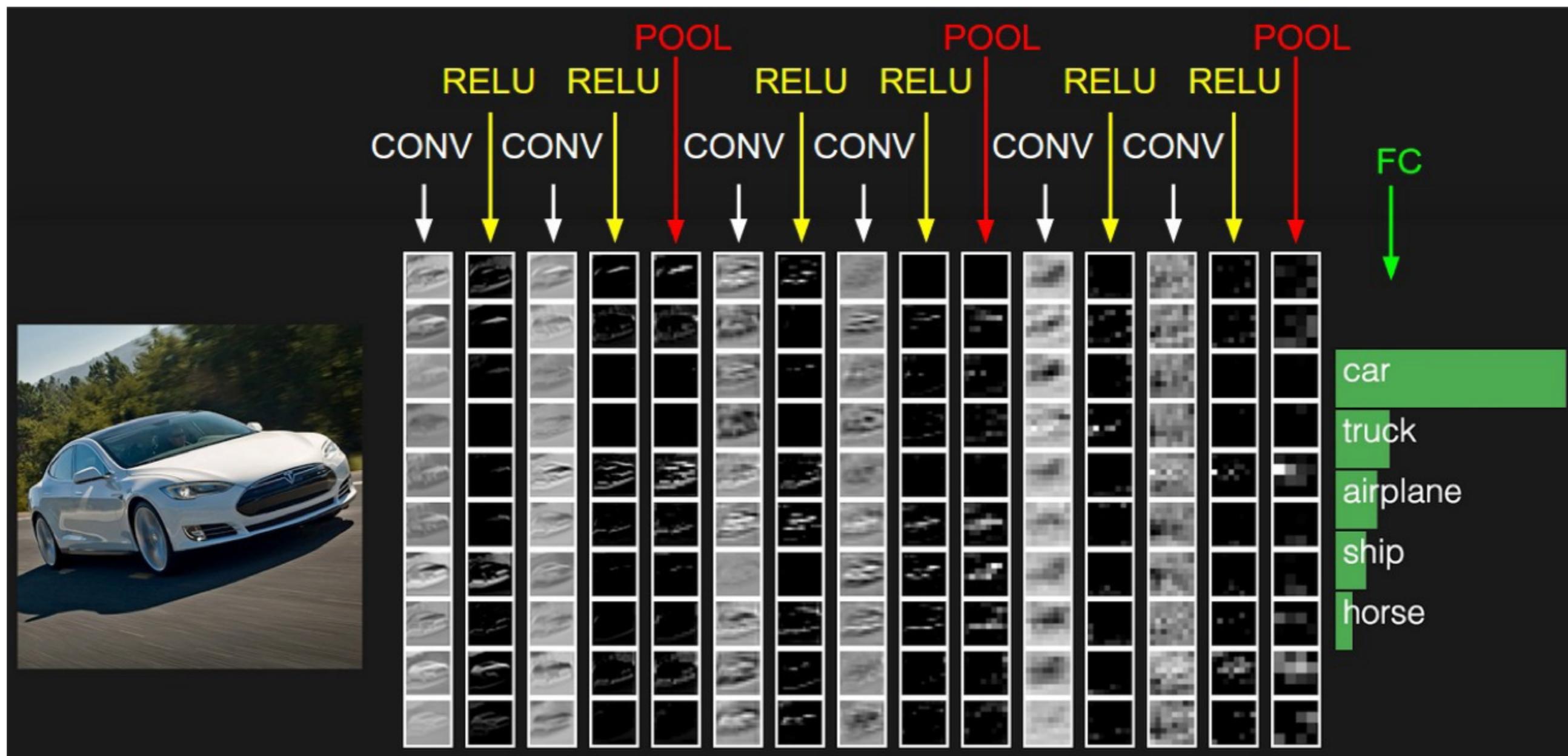
deeplearning.ai

Convolutional
Neural Networks

Computer vision

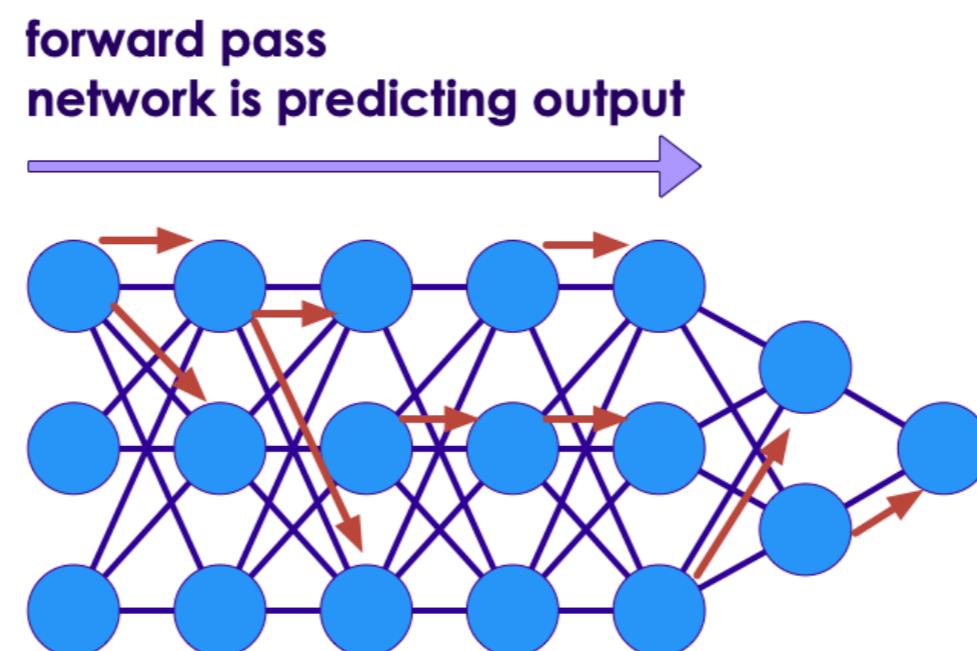
Example Convolutional Network

- Here we have repeating layers of convolutional and pooling layers
- And a final Softmax layer is giving the probabilities of the image class



CNN Training Workflow

- Make predictions on training data images (forward pass).
- Determine which predictions were incorrect and propagate back the difference between the prediction and the true value (backpropagation).
- Rinse and repeat till the predictions become sufficiently accurate.
- It's quite likely that the initial iteration would have close to 0% accuracy. Repeating the process several times, however, can yield a highly accurate model (> 90%).
- **Animation** : [link-youtube](#), [link-S3](#)



CNN Training Batches

- The batch size defines how many images are seen by the CNN at a time
- Each batch should have a good variety of images from different classes in order to prevent large fluctuations in the accuracy metric between iterations.
 - A sufficiently large batch size would be necessary for that.
- However, don't let the batch size too large
 - It could consume too much memory
 - The training process would be slower.
- Usually, batch sizes are set as powers of 2.
64 is a good number to start, then tweak it from there.

CNN Batch Size and Memory

- Image input is: 150×100 RGB image (three channels)
- Convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100
- Then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15,200$ (the +1 corresponds to the bias terms), which is fairly small compared to a fully connected layer
- Each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that's a total of 225 million float computations
- If the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits
- About **11.4 MB of RAM per image**
- If a training batch contains **100 images**, then this layer will use up over **1 GB of RAM!**
- Reference: Neural Networks and Deep Learning, Chapter 3

Data Augmentation

- State of the art image classifiers are trained with millions of images
- What if we only have a small sample (say a few hundred) ?
- NNs could 'memorize' the small dataset.
 - great accuracy on training set
 - but bad accuracy on test set (or new/unseen images)
 - this is classic 'overfitting'
- If available images are limited, then we can use a technique called 'augmentation' to create more data samples
- Only augment training set
 - Do not augment validation/test set because the resulting accuracy metric would be inconsistent and hard to compare across other iterations.

Data Augmentation

- Augmentation techniques

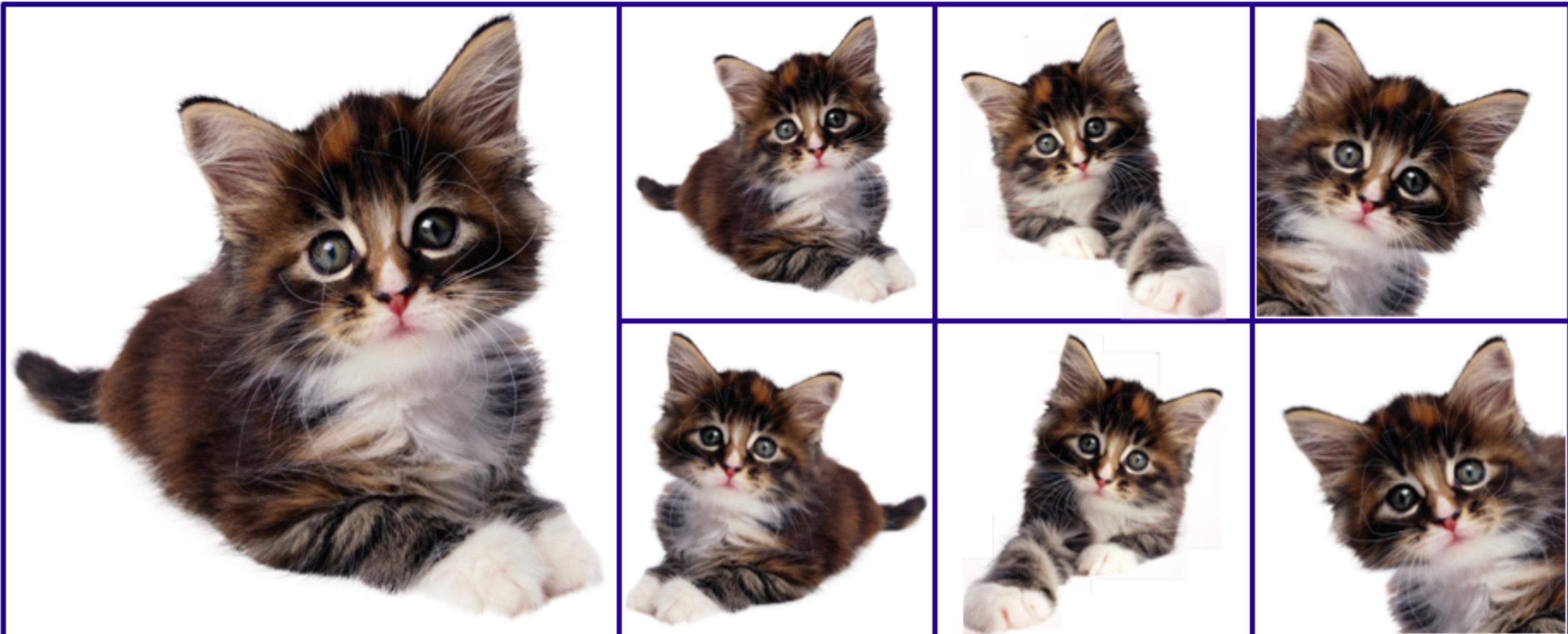
- Rotation (rotate image 10-20' clockwise or counter-clockwise)
- Shift images to left/right of center
- Zoom in/out
- combine the above

- Augmentation tools

- Keras ImageDataGenerator
- imaging library

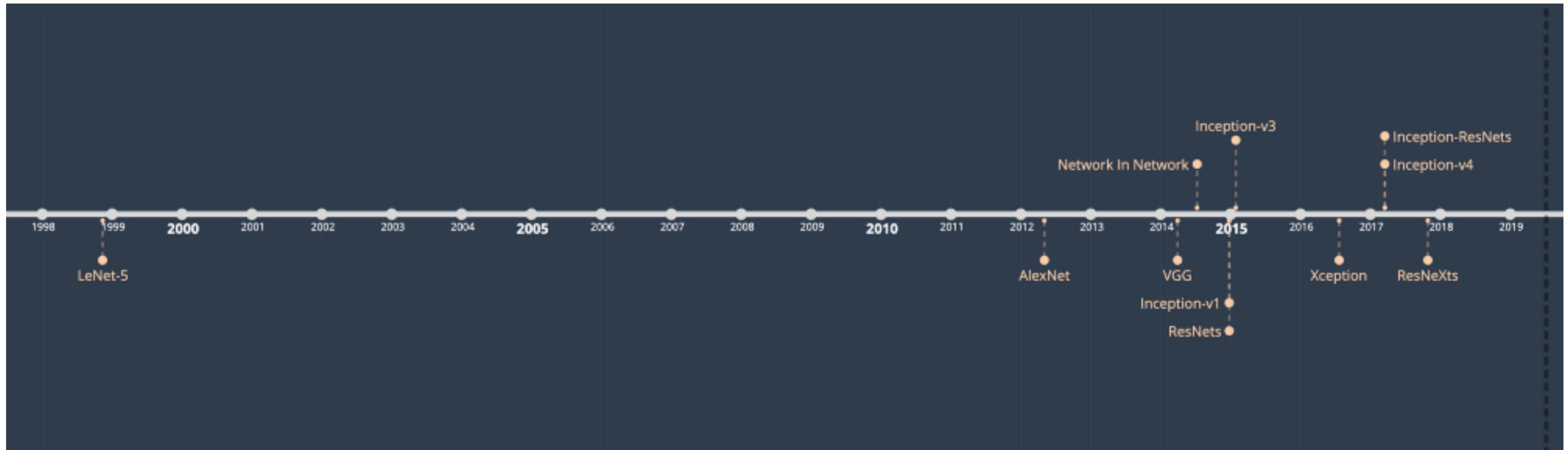
Image Augmentation Example

- Use techniques like: rotate slightly to left and right, flip the image ...etc.



Popular CNN Architectures

Popular CNN Architectures



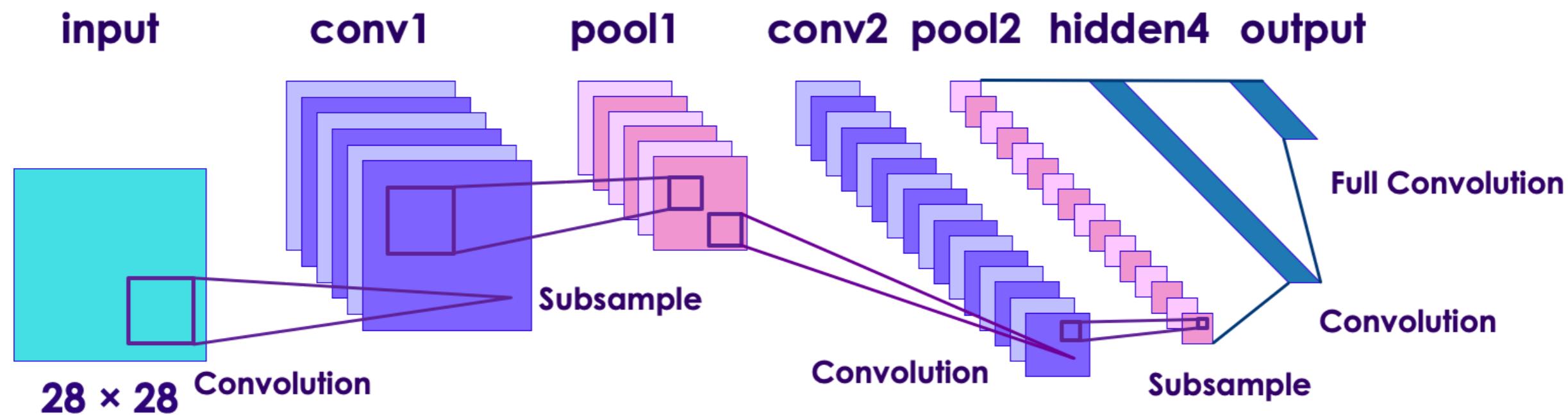
- References:
 - Illustrated: 10 CNN Architectures
 - "Programming PyTorch for Deep Learning" book Chapter 03

Popular CNN Models

Year	Name	Created By	Layers	Parameters	Description
1998	LeNet	Yann Lecun	5	60,000	Simple, but basic template for CNN architecture
2012	AlexNet	Hinton et al	8	60 million	Dominated ImageNet Challenge in 2012. Ignited the NeuralNet boom. Introduced ReLU activation function
2014	VGG-16	Oxford Uni	19	138 million	2014 ImageNet 2nd place. Introduced deeper network (twice as deep as AlexNet)
2014	Inception V1 (GoogLeNet)	Google	22	7 million	Won 2014 ImageNet competition. Introduced Inception model of using dense modules/blocks
2015	Inception V3	Google	42	24 million	2015 ImageNet runner up. Introduced batch normalization technique
2015	ResNet-50	Microsoft	150	26 million	2015 ImageNet winner. Went way deeper than previous models
2016	Xception	Keras Team		23 million	Extension of Inception model
2016	Inception V4	Google		43 million	Updated from previous Inception v3
2016	Inception-ResNet-V2	Google		56 million	A merge of Inception and ResNet

LeNet (1998)

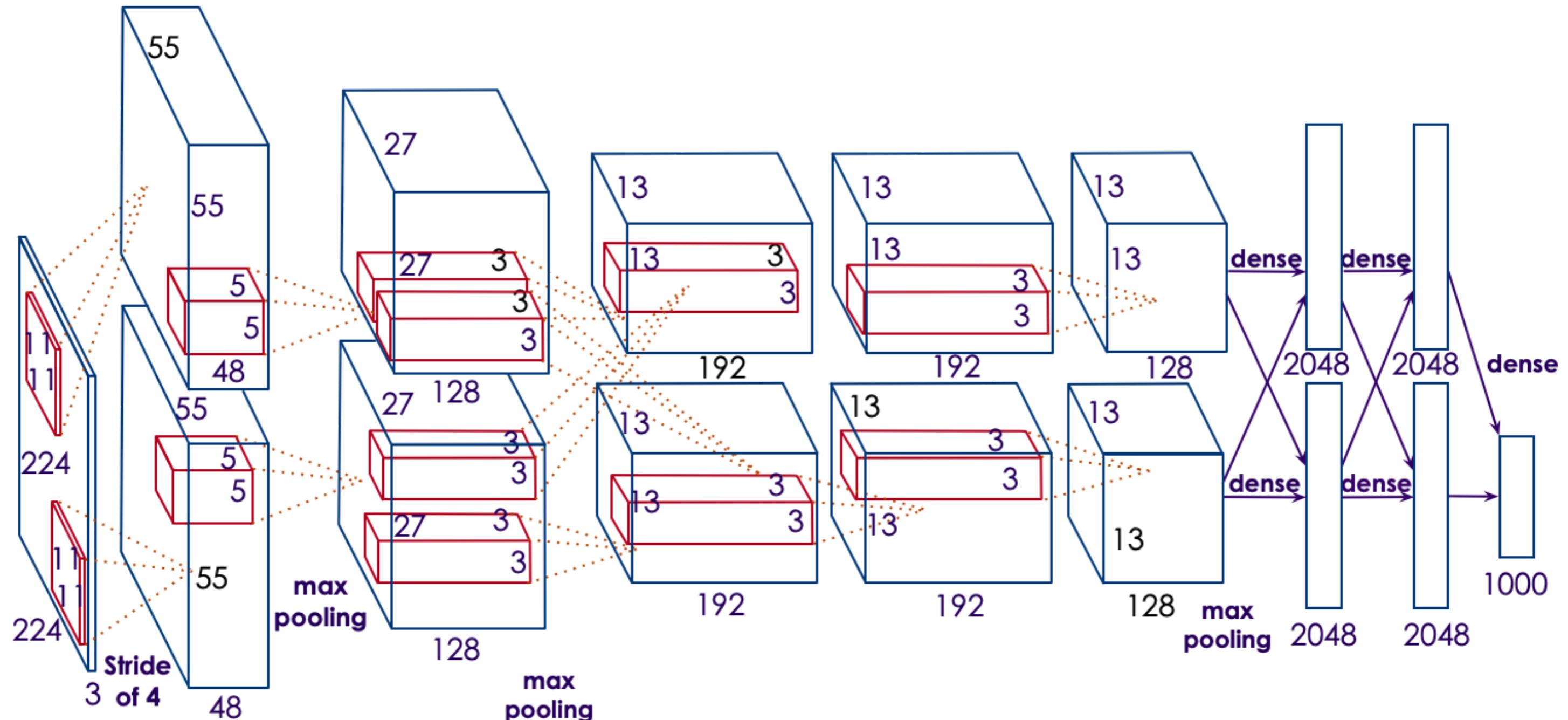
- The 'OG' of CNNs
- Created by Yann LeCun
- Handwritten digits / MNIST type data (10 classes)
 - To identify Zipcodes for USPS
- 28x28x1 monochrome images
- Revolutionary in its time
 - Mostly constrained by resources of the day



AlexNet (2012)

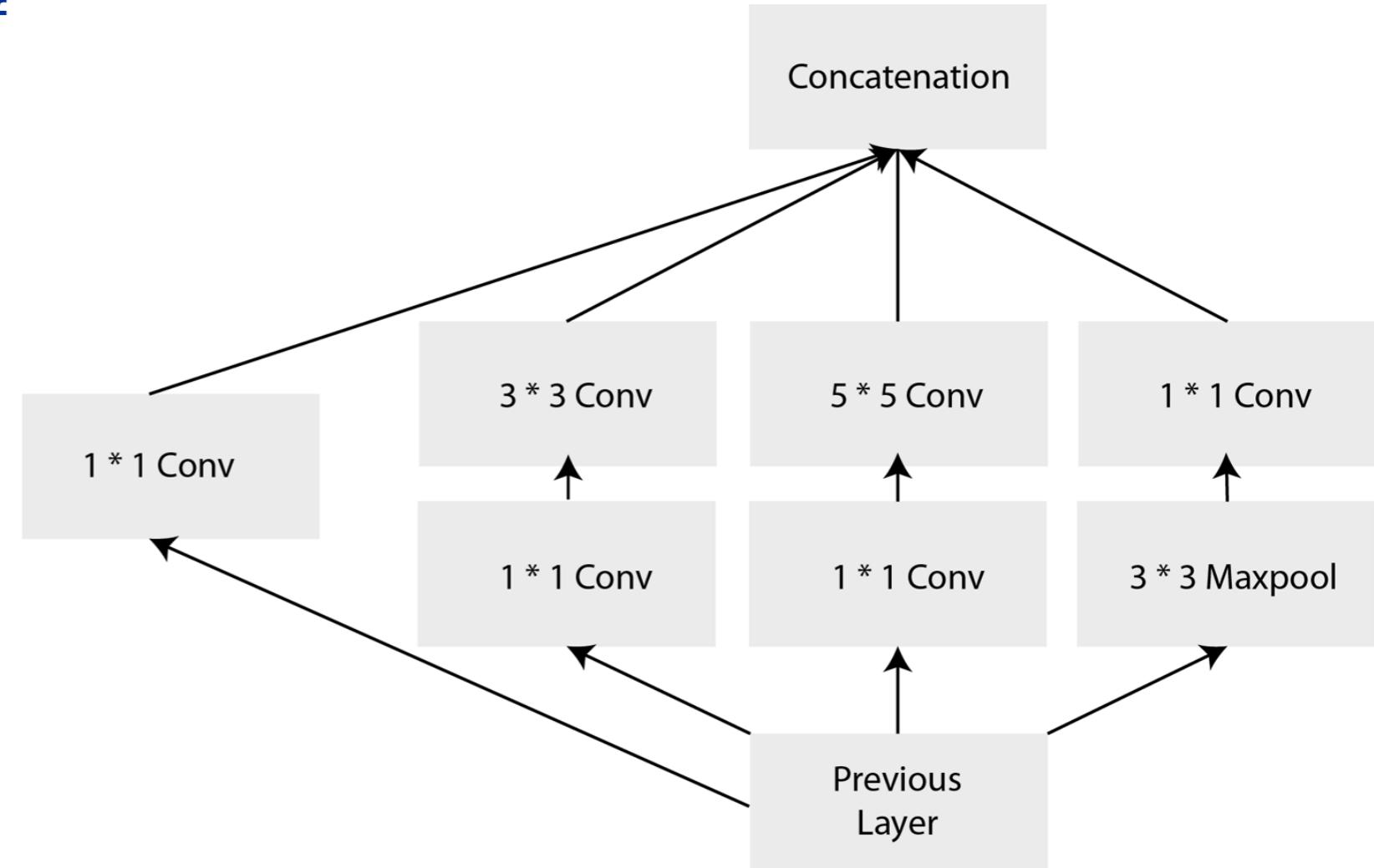
- AlexNet was an important milestone in CNNs. It kick-started the latest boom in CNNs
- In 2012 ImageNet competition, it dominated with top-5 error rate is 15.3%, beating the second place entry with a top-5 error of 26.2%; Almost a 10% gap
- AlexNet was the blueprint for lot of later architectures
- It introduced MaxPool and DropOut concepts, Also introduced ReLU activation function
- Applied LeNet to full size RGB images (224x224x3) with 1000 classes.
- References:
 - ImageNet Classification with Deep Convolutional Neural Networks - original paper
 - Review: AlexNet, CaffeNet — Winner of ILSVRC 2012 (Image Classification)

AlexNet (2012)

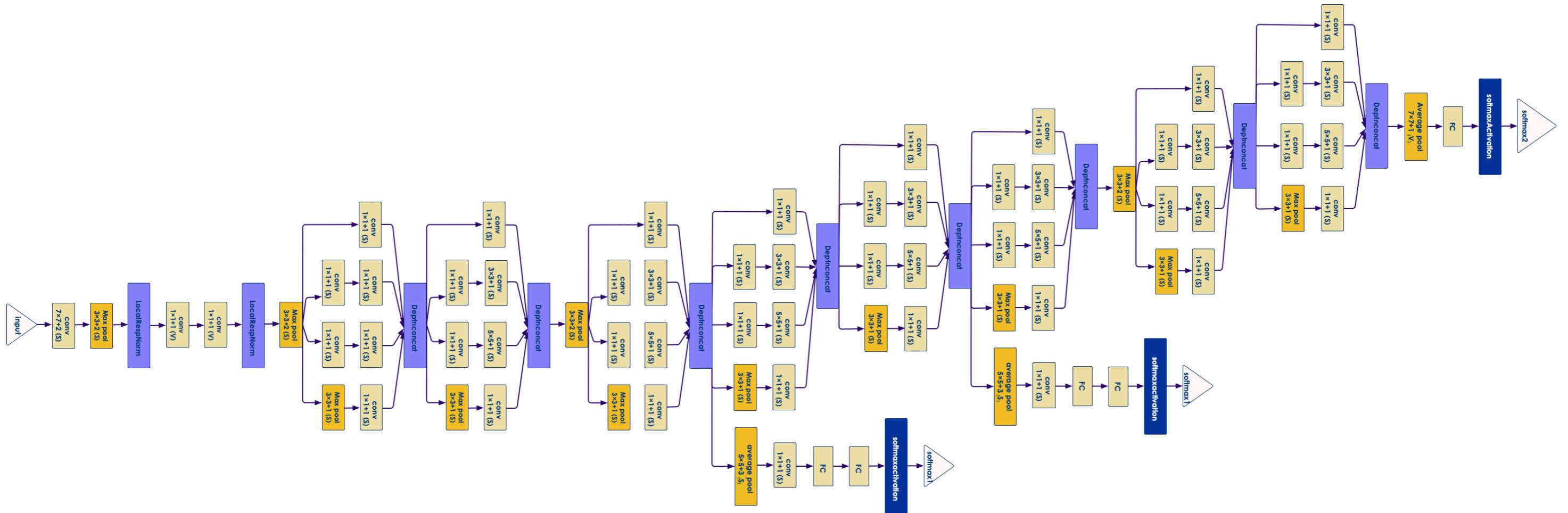


Inception V1 / GoogLeNet (2014)

- 2014 ImageNet winner from Google, introduced a Inception concept
- AlexNet had a fixed size kernel; Inception had various sized kernels
 - So a large kernel can identify a car
 - A smaller kernel can further identify logos ..etc
- The Inception network runs a series of convolutions of different sizes all on the same input, and concatenates all of the filters together to pass on to the next layer
- Number of layers:22,
Number of parameters: 5 million



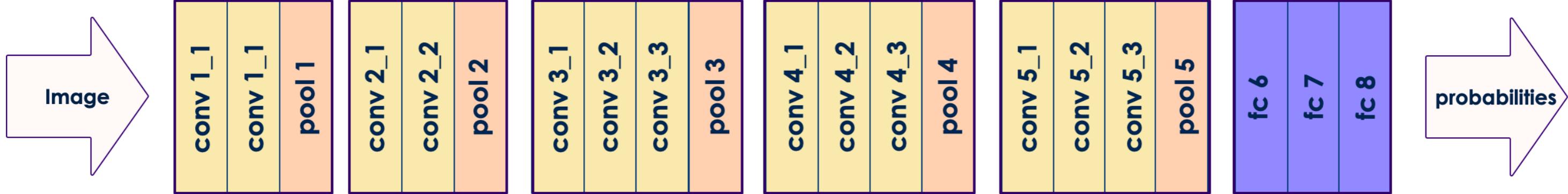
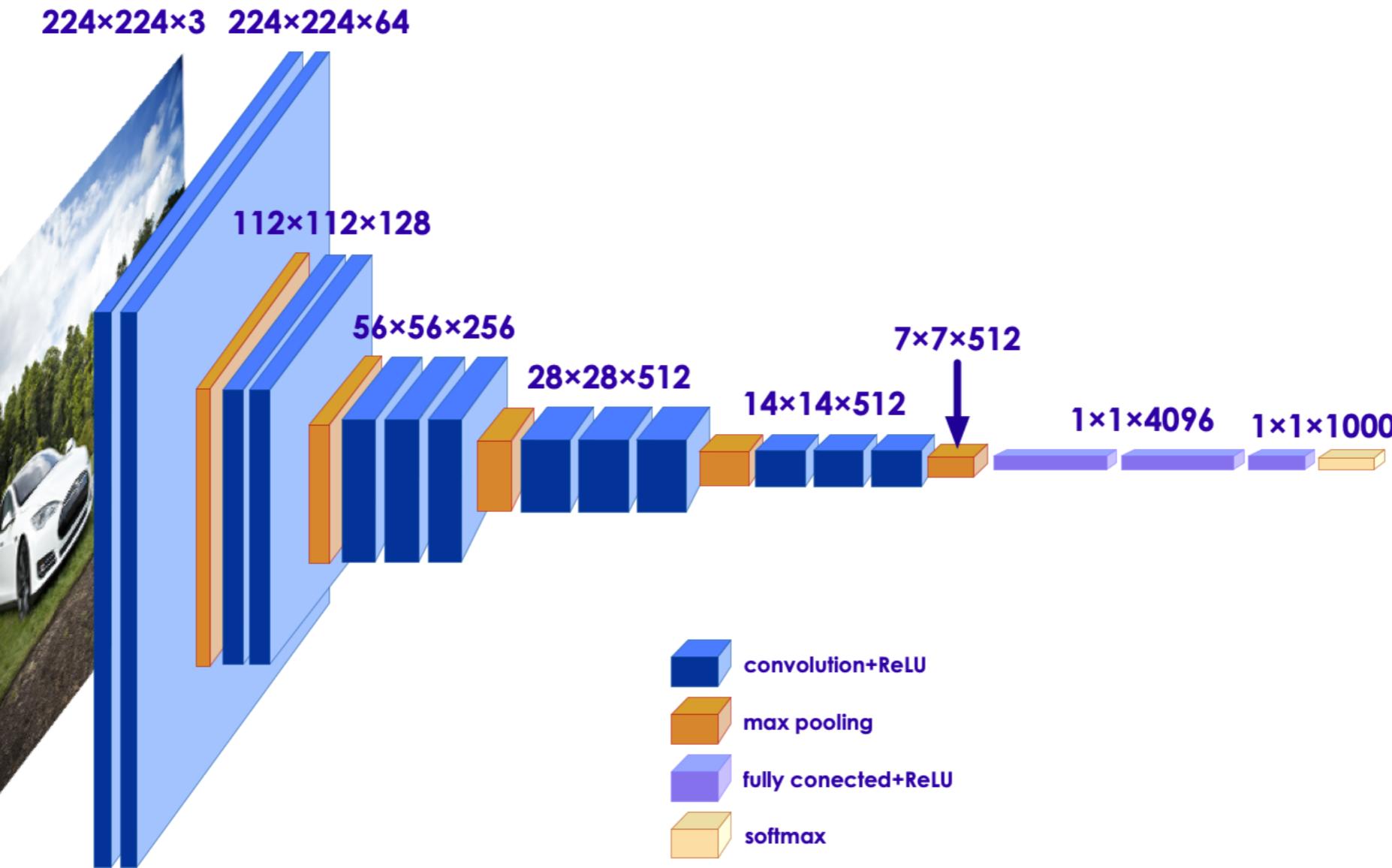
Inception V1



VGG (2014)

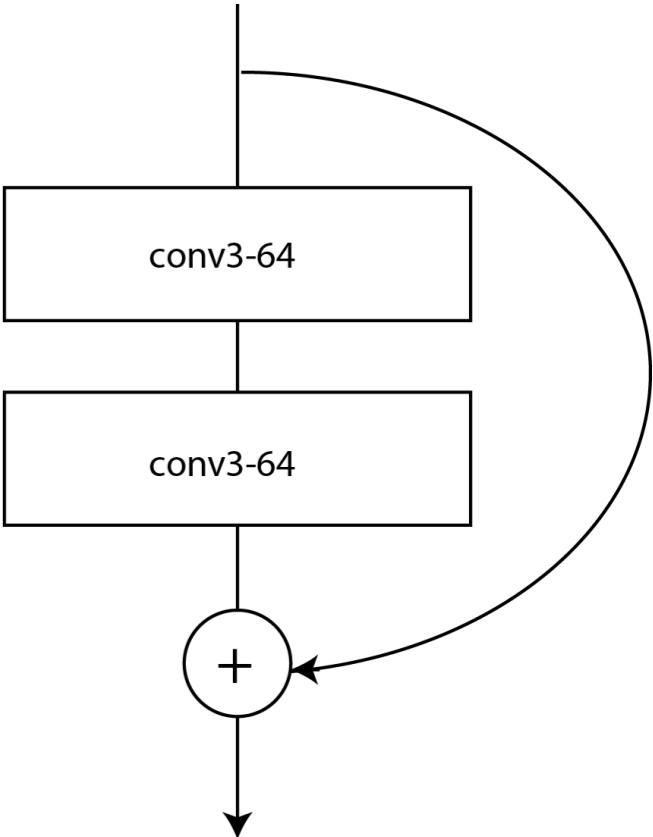
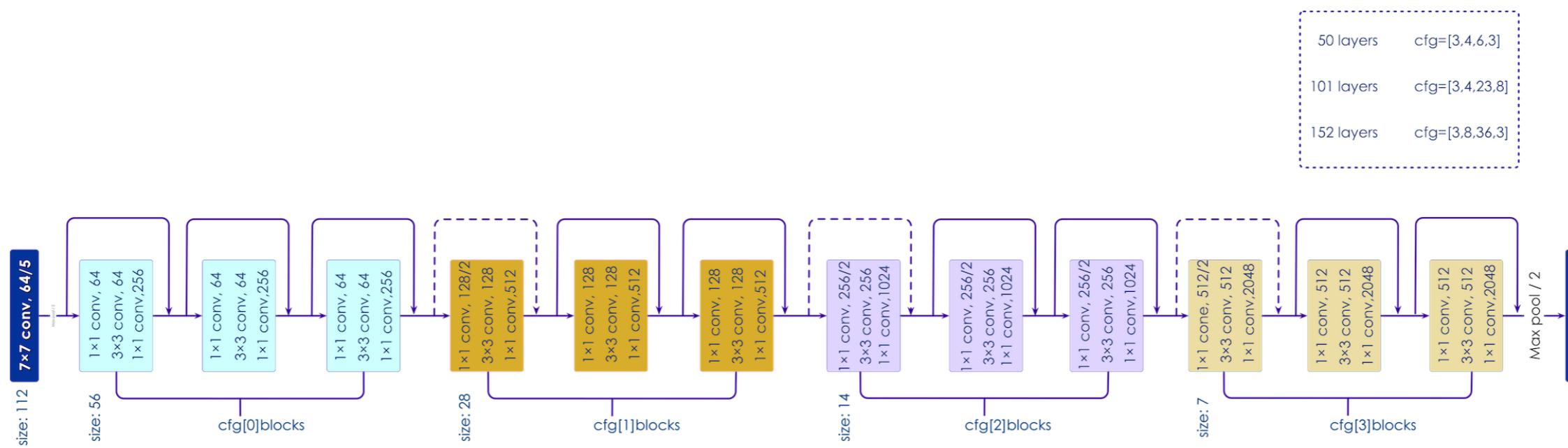
- From University of Oxford—the Visual Geometry Group (VGG)
- Won 2nd place on 2014 ImageNet competition
- It introduced deeper layers (19 layers)
- One downside of VGG is final fully connected layers make the network balloon to a large size, weighing in at 138 million parameters in comparison with GoogLeNet's 7 million
- Despite it's large size, still popular transfer learning architecture
- Number of layers: 19
Number of parameters : 138 million

VGGNet



ResNet (2015)

- ResNet-152 model from Microsoft is the 2015 ImageNet winner with top-5 score of 4.49%
- ResNet uses Residual Neural Networks (RNNs); Not feed forward; Both Residual and Convolutional
- ResNet improved on the Inception-style (stacking bundle of layers approach), wherein each bundle performed the usual CNN operations but also added the incoming input to the output of the block
- Number of layers: 150
Number of parameters: 26 million



CNN Architectures Comparison

- Inception wins on performance, accuracy, and size of network (number of params)
- ResNet is a close second with some valid use cases.
- Inception and ResNet are often ensembled (combined together).

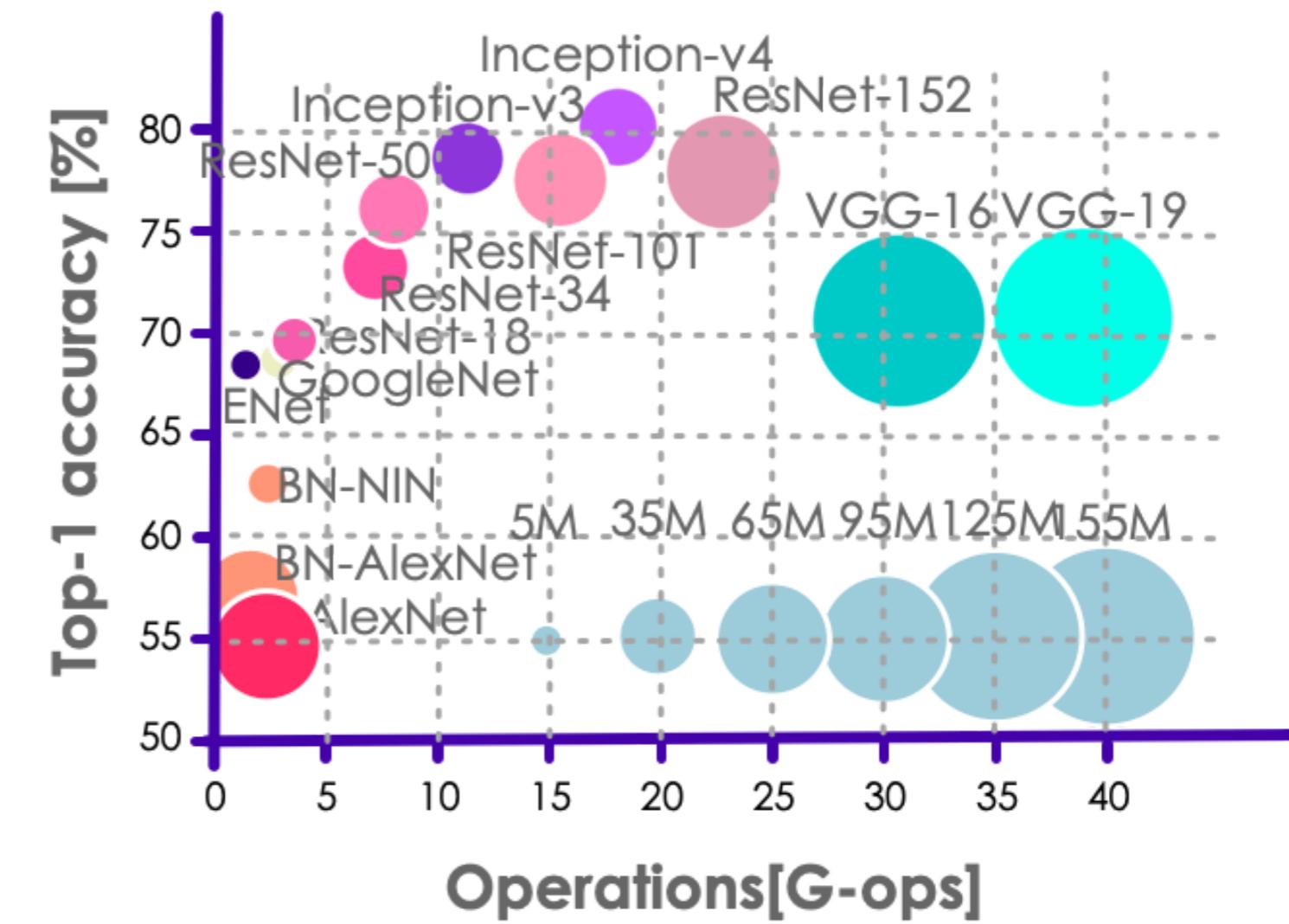
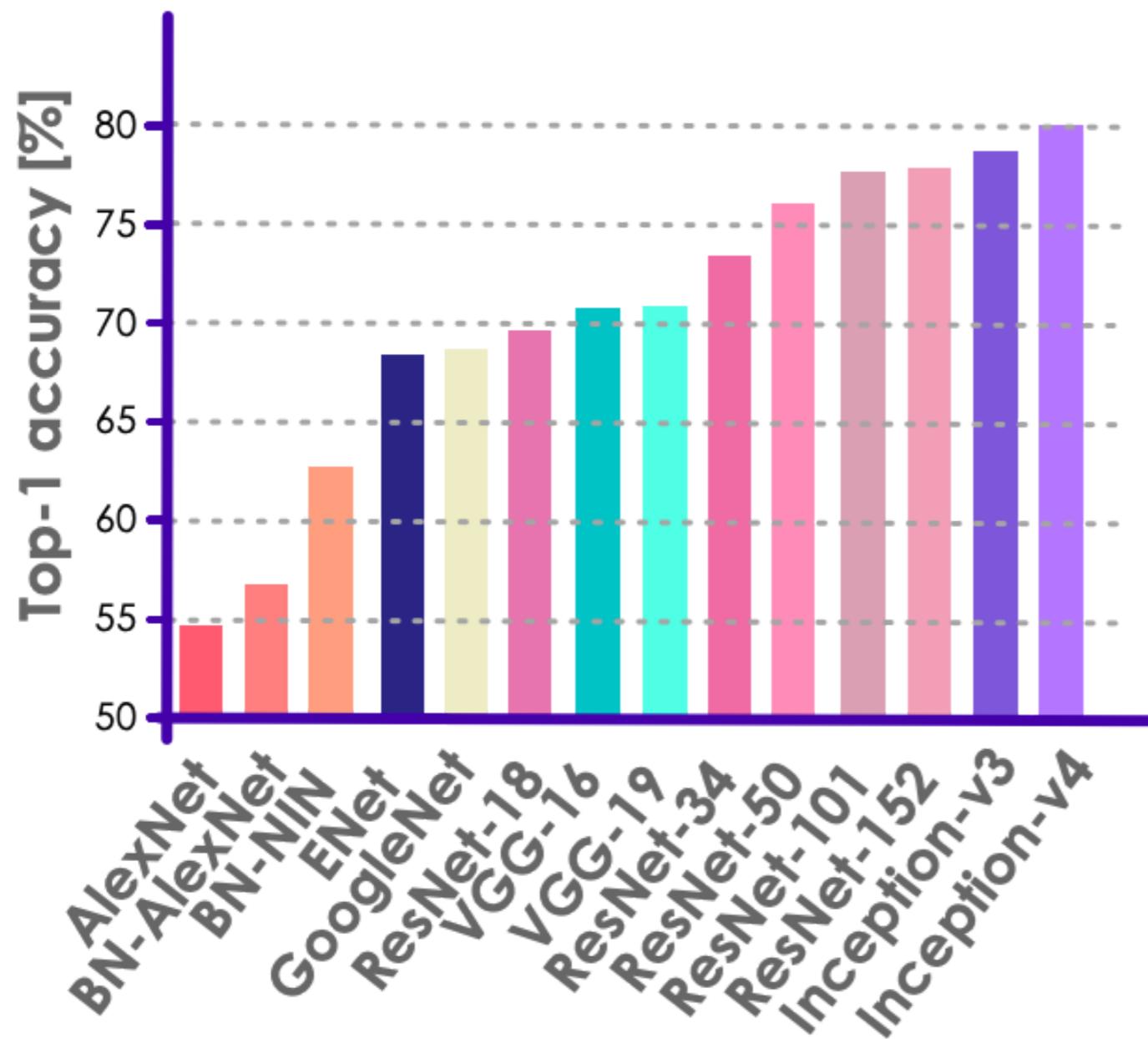


Image Datasets

Popular Image Datasets

- MNIST
- Fashion MNIST
- CIFAR
- ImageNet
- Street View House numbers
- Cats & Dogs

MNIST

- MNIST Dataset is the "hello world" of image recognition
- 28x28 greyscale scanned digits
- Total 70,000 images
 - 60,000 training images (26 MB)
 - 10,000 test images (4.3 MB)
- MNIST is often the first dataset researchers try.

"If it doesn't work on MNIST, it won't work at all",

"Well, if it does work on MNIST, it may still fail on others."

- Reference



Exploring MNIST

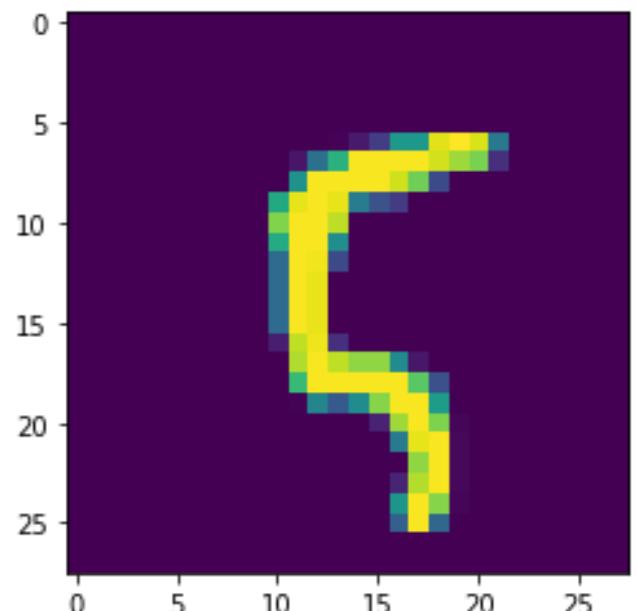
```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()

print("train_images shape : ", train_images.shape)
print("train_labels shape : ", train_labels.shape)
print("test_images shape : ", test_images.shape)
print("test_labels shape : ", test_labels.shape)
# train_images shape : (60000, 28, 28)
# train_labels shape : (60000,)
# test_images shape : (10000, 28, 28)
# test_labels shape : (10000,)

## Display a sample image
index = 100
print("train label [{}] = {}".format(index, train_labels[index]))
# train label [100] = 5
plt.imshow(train_images[index])
```

- Training images are 28x28 pixel images
- Labels are numbers: 0 - 9
- Here we are display 100th image, which is a '5'.
See the image and label



Fashion MNIST

- Fashion MNIST was created as a 'better MNIST'
 - MNIST is too easy; Classic ML algorithms can achieve 97% accuracy, CNNs can achieve 99.7% accuracy!
- Images of 10 fashion items (shirt, sweater, shoe ..etc)
- 28x28 greyscale images
- Total 70,000 images
 - 60,000 training images (26 MB)
 - 10,000 test images (4.3 MB)
- Designed as a 'drop in' replacement for MNIST
- Reference | bigger image



Exploring Fashion-MNIST

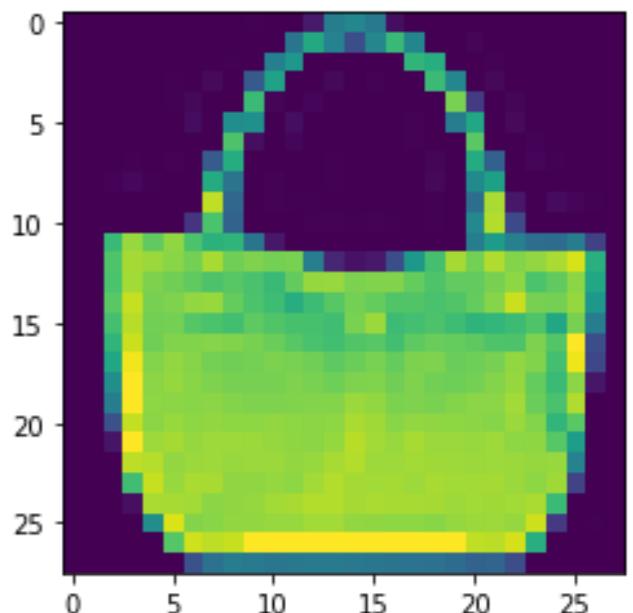
```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = keras.datasets.fashion_mnist.load_data()

print("train_images shape : ", train_images.shape)
print("train_labels shape : ", train_labels.shape)
print("test_images shape : ", test_images.shape)
print("test_labels shape : ", test_labels.shape)
# train_images shape : (60000, 28, 28)
# train_labels shape : (60000,)
# test_images shape : (10000, 28, 28)
# test_labels shape : (10000,)

## Display a sample image
index = 100
print("train label [{}] = {}".format(index, train_labels[index]))
# train label [100] = 8
plt.imshow(train_images[index])
```

- Training images are 28x28 pixel images
- Labels are numbers: 0 - 9
- Here we are display 100th image, which is a '8' (handbag).



Exploring Fashion MNIST

- Why do we use numbers as labels as opposed to 'Dress', 'Coat'
- Numbers are universal; not constrained by languages (English / Japanese ..etc)

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

CIFAR-10

- CIFAR-10 dataset consists of
 - 60,000 color images
 - 32x32 pixels, color
 - 10 classes (6,000 per class)
- 50,000 training images; 10,000 test images
- 5 training batches, 1 test batch
 - 10,000 images per batch
 - Each batch has 1000 random images from each class
- Size ~160 MB



CIFAR-100

- CIFAR-100 is pretty much like CIFAR-10
 - 32x32 pixels (color)
 - 100 classes
 - 600 images per class
- The 100 classes are grouped into 20 superclasses.
- **Size ~160MB**

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers

Using CIFAR data (TensorFlow)

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

ImageNet

- ImageNet is an image database
- 15 millions+ labeled high-resolution images with around 22,000 categories.
- It is organized hierarchically

```
imagenet/
└── animals
    └── domestic
        ├── cat
        │   ├── cat1.jpg
        │   └── cat2.jpg
        └── dog
            ├── dog1.jpg
            └── dog2.jpg
```

Imagenet

IMAGENET

SEARCH

Home About Explore Download

Not logged in. Login | Signup

Dog, domestic dog, *Canis familiaris*

A member of the genus *Canis* (probably descended from the common wolf) that has been domesticated by man since prehistoric times; occurs in many breeds; "the dog barked all night"

1603 pictures 88.15% Popularity Percentile Wordnet IDs

Numbers in brackets: (the number of synsets in the subtree).

ImageNet 2011 Fall Release (3232)

- plant, flora, plant life (4486)
- geological formation, formation (1112)
- natural object (1112)
- sport, athletics (176)
- artifact, artefact (10504)
- fungus (308)
- person, individual, someone, son (10504)
- animal, animate being, beast, brute (766)
- invertebrate (766)
- homeotherm, homiotherm, homoiotherm (4)
- work animal (4)
- darter (0)
- survivor (0)
- range animal (0)
- creepy-crawly (0)
- domestic animal, domesticated animal (4)
- domestic cat, house cat, Feline (4)
- dog, domestic dog, *Canis familiaris* (101)
- pooch, doggie, doggy, hunting dog (101)
- dalmatian, coach dog, carriage dog (2)
- cur, mongrel, mutt (2)
- corgi, Welsh corgi (2)
- Mexican hairless (0)
- lapdog (0)
- Newfoundland, Newfoundland (4)
- poodle, poodle dog (4)
- basenji (0)
- Leonberg (0)

Treemap Visualization Images of the Synset Downloads

Hunting Working

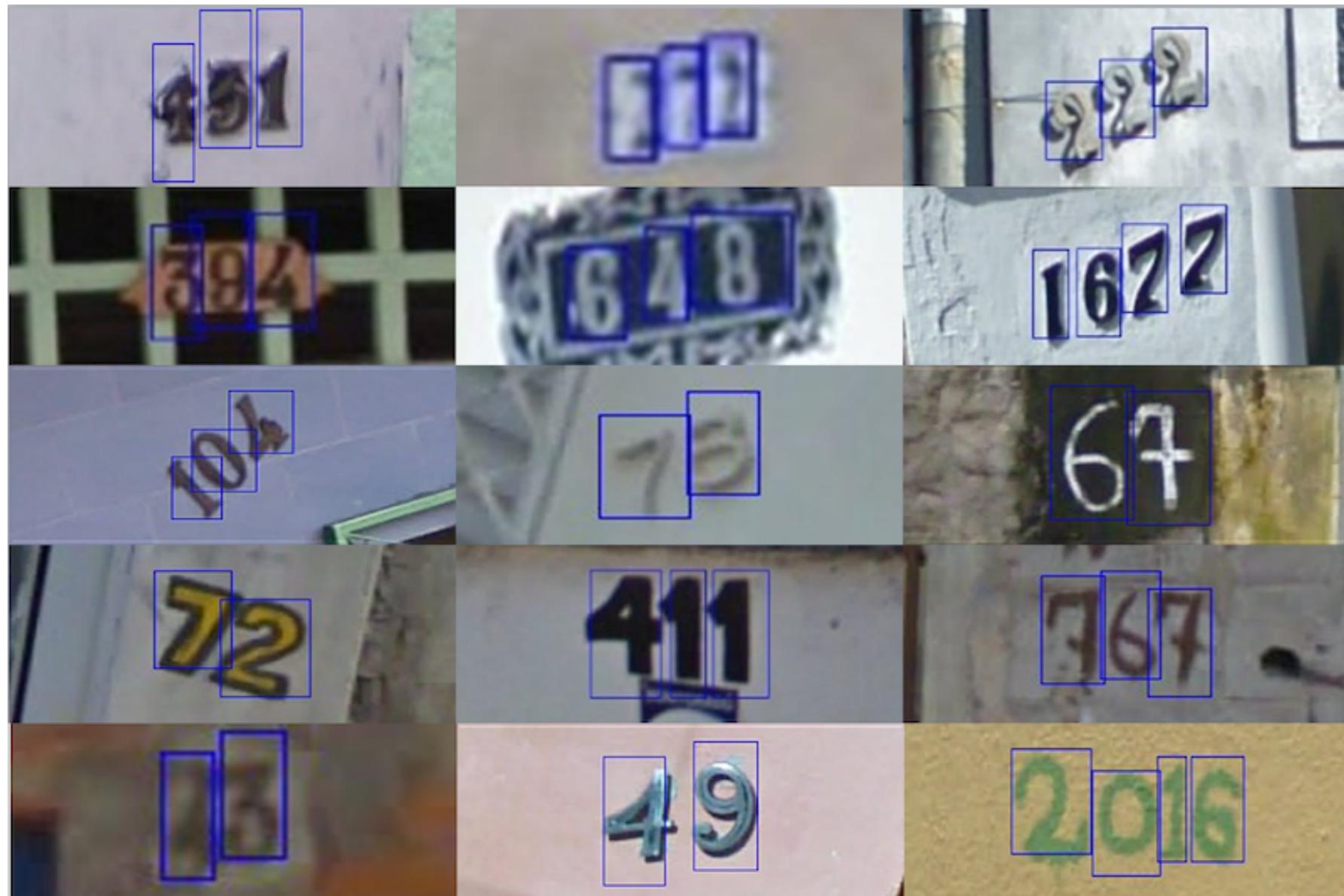
Toy	Leonberg	Pooch	Puppy
Great	Dalmatian	Griffon	
Mexican	Cur	Newfoundland	Poodle
Lapdog	Pug	Basenji	Spitz
			Corgi

Imagenet

- The ImageNet dataset was the basis for the famous ImageNet Large Scale Visual Recognition Challenge (ILSVRC)
- ILSVRC uses a subset of ImageNet of around 1000 images in each of 1000 categories.
~1.2 million training images, 50,000 validation images and 150,000 testing images.
- ILSVRC competition started in 2010
- Since then it is considered the 'Olympics' of image recognition.
Researchers compete to win this prestigious competition
- The accuracy has gone up from 70% to 97%
- Also researchers are sharing models trained with ImageNet, making rapid progress in image recognition

Street View Numbers

- Streetview comes from Google Street View data
- over 600,000 images



Cats & Dogs

- In 2014 Microsoft Research was working on a CAPTCHA system
- For that they were using ASIRRA (Animal Species Image Recognition for Restricting Access)
- 3 million images (800 MB in size)
- Labeled by animal shelters throughout US and also [PetFinder.com](#)
- When the dataset came out the accuracy was around 80%. Within a few weeks the top algorithms were scoring 98% accuracy!
- This image set has become a 'classic' test for image recognition algorithms! (The cuteness doesn't hurt either!)
- [Link to Paper](#) ,
- [Link to download](#)



CNNs in TensorFlow



CNN Example 1 - CIFAR

- CIFAR-10 dataset consists of
 - 60,000 color images
 - 32x32 pixels, color
 - 10 classes (6,000 per class)
- 50,000 training images; 10,000 test images
- 5 training batches, 1 test batch
 - 10,000 images per batch
 - Each batch has 1000 random images from each class
- **Size ~160 MB**



CNN Example 1 - CIFAR

- **tensorflow.datasets** package has convenient way to download CIFAR dataset

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

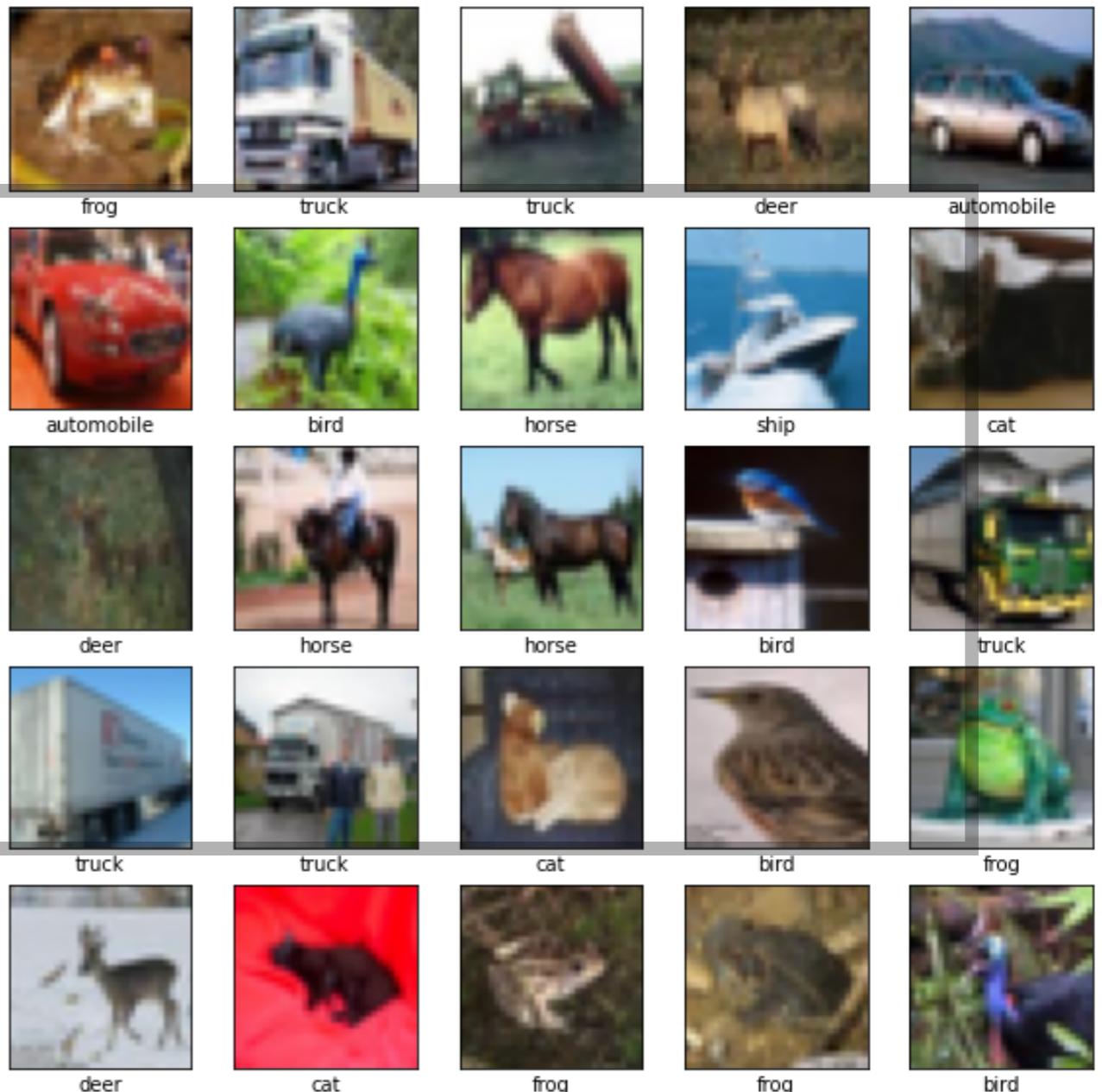
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

CNN Example 1 - CIFAR

- Explore CIFAR dataset
- Reference

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



CNN Example 1 - CIFAR

- Here we are creating a convolutional layer
 - **CONV2D (filters=64, kernel=(3, 3), input_shape=(32, 32, 3))**
 - filters=64 (filters are initialized randomly from a 'uniform distribution' and learned during training - just like any other weights)
 - kernel / convolution size = 3x3
- Input shape is equivalent to image dimensions (32, 32, 3)
 - 32 x 32 pixels
 - 3 channels (RGB) - color images
 - Reference

```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

CNN Example 1 - CIFAR

- So far the model looks like this
- **Can you understand how the input is shaped through the network?**

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36928
<hr/>		

```
Total params: 56,320
```

```
Trainable params: 56,320
```

```
Non-trainable params: 0
```

CNN Example 1 - CIFAR

- We will add **Dense layers** on top
- We need these layers to classify the images
- The final layer will be a **Softmax** layer

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

CNN Example - CIFAR

- Final model shape
- Note **Trainable params: 122,570**

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36928
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 64)	65600
dense_1 (Dense)	(None, 10)	650
<hr/>		
Total params: 122,570		
Trainable params: 122,570		
Non-trainable params: 0		

CNN Example 1 - CIFAR

▪ Compile and run the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                     validation_data=(test_images, test_labels))
```

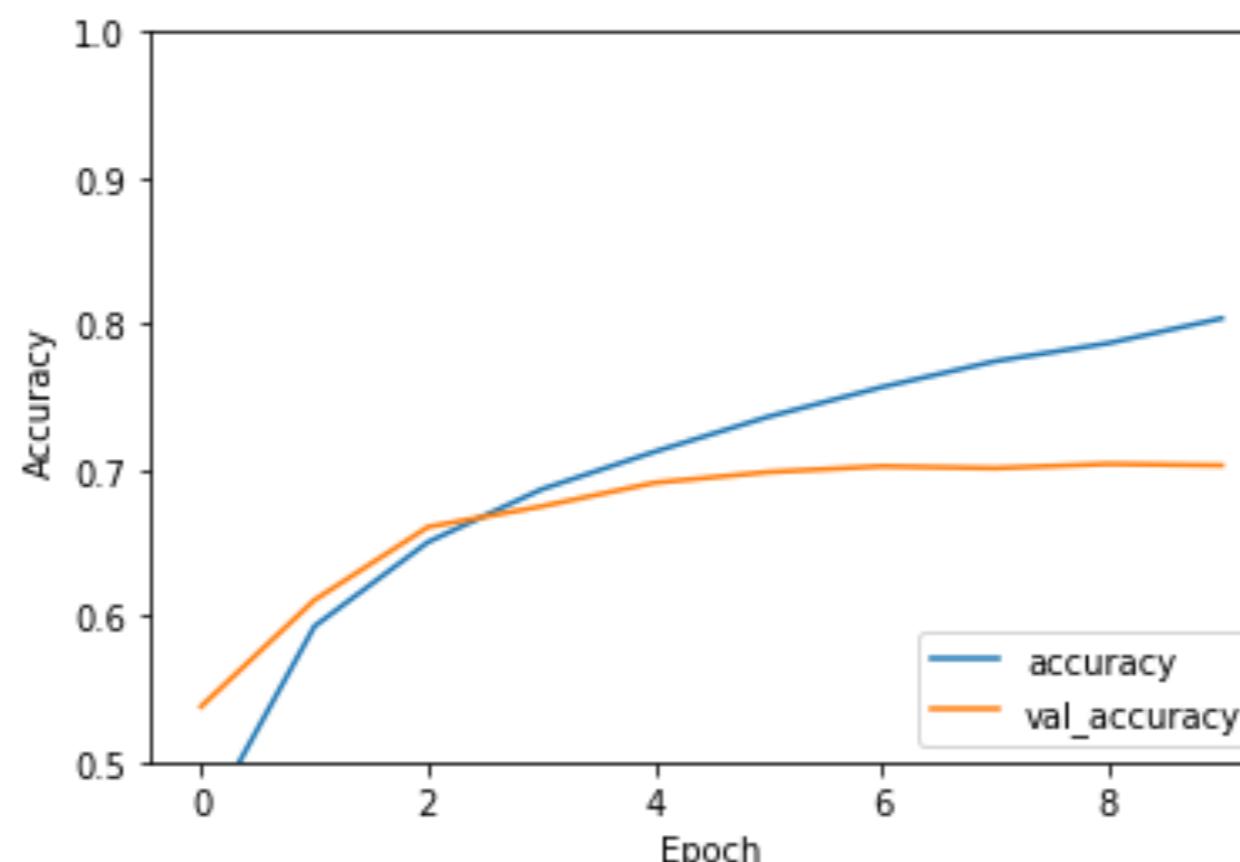
```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [=====] - 7s 141us/sample - loss: 1.5074 - accuracy: 0.4526 -
    val_loss: 1.2683 - val_accuracy: 0.5378
Epoch 2/10
50000/50000 [=====] - 5s 98us/sample - loss: 1.1462 - accuracy: 0.5928 -
    val_loss: 1.0982 - val_accuracy: 0.6108
...
...
Epoch 9/10
50000/50000 [=====] - 5s 99us/sample - loss: 0.6014 - accuracy: 0.7867 -
    val_loss: 0.8803 - val_accuracy: 0.7040
Epoch 10/10
50000/50000 [=====] - 5s 98us/sample - loss: 0.5582 - accuracy: 0.8037 -
    val_loss: 0.9265 - val_accuracy: 0.7030
```

CNN Example 1 - CIFAR

- Evaluate the model
- We can see from the training history plot, the accuracy is steadily increasing during each epoch
- Final accuracy after 10 epochs, is 0.70 (70%)

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
# 10000/10000 - 1s - loss: 0.9265 - accuracy: 0.7030
```



Lab: CNN lab

- **Overview:**

- Implement CNNs in TensorFlow

- **Approximate run time:**

- 30-40 mins

- **Instructions:**

- Try these labs
 - **Computer Vision 1 - MNIST Intro**
 - (optional) **Computer Vision 2 - Fashion MNIST Intro**
 - **Computer Vision 3 - MNIST CNN**
 - **Computer Vision 4 - CIFAR CNN**

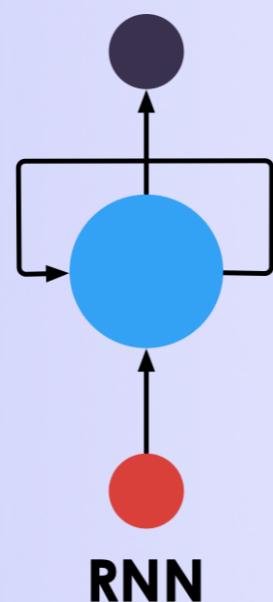


Review and Q&A

- Let's go over what we have covered so far
- Any questions?



Recurrent Neural Networks (RNNs) in TensorFlow



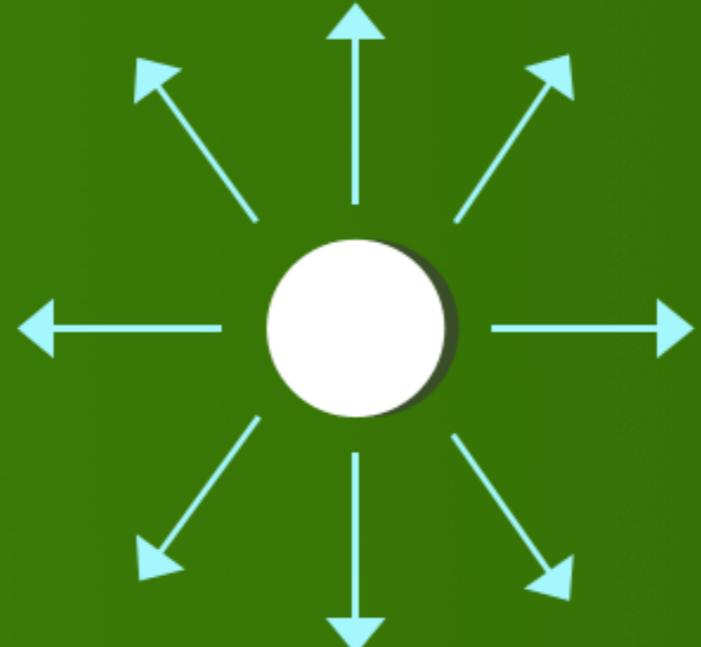
Objectives

- Learn about RNNs
- Implement RNNs in TensorFlow and Keras

RNN Intro

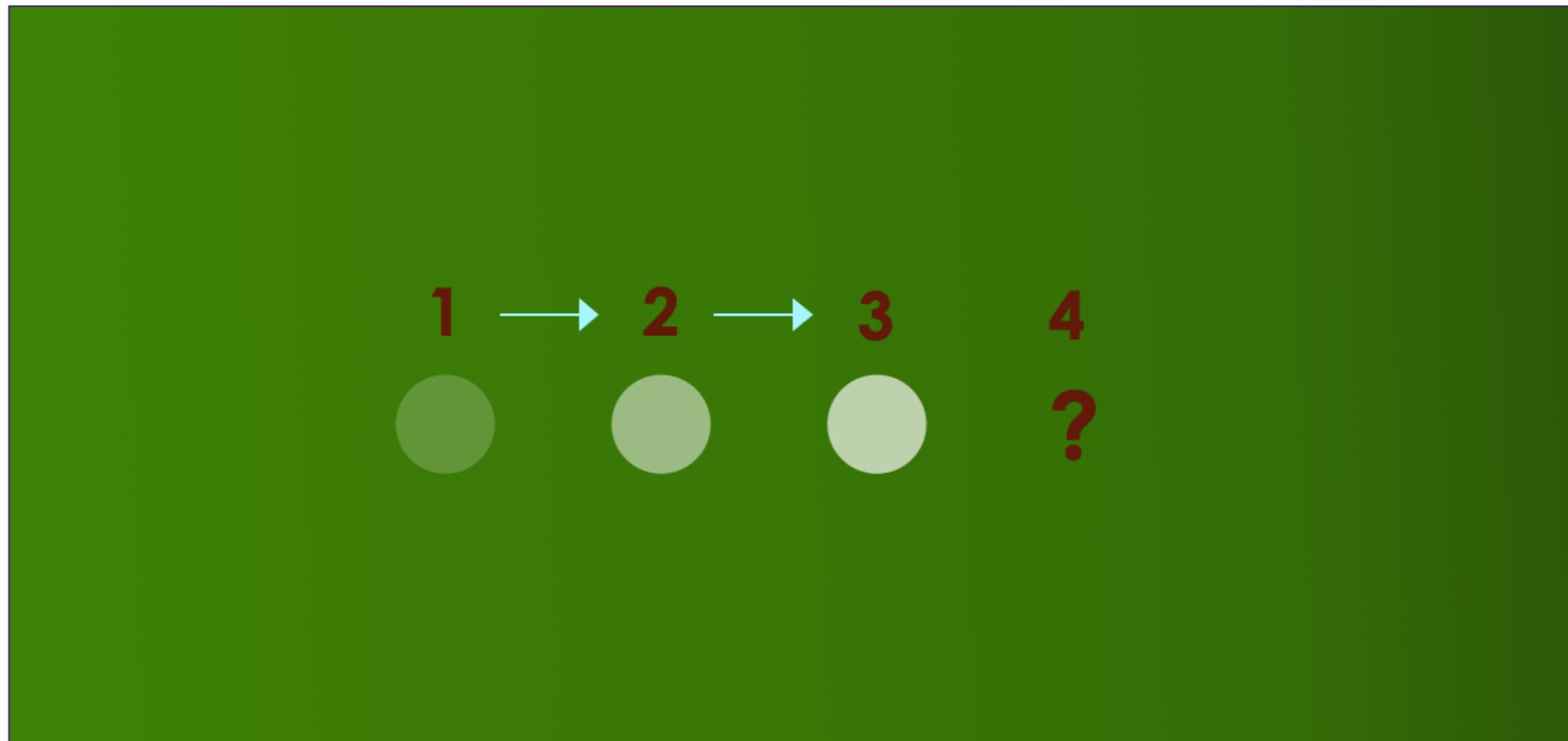
Sometimes to Predict the Future, We Need to Know the Past

- Can you predict the next position of the ball?



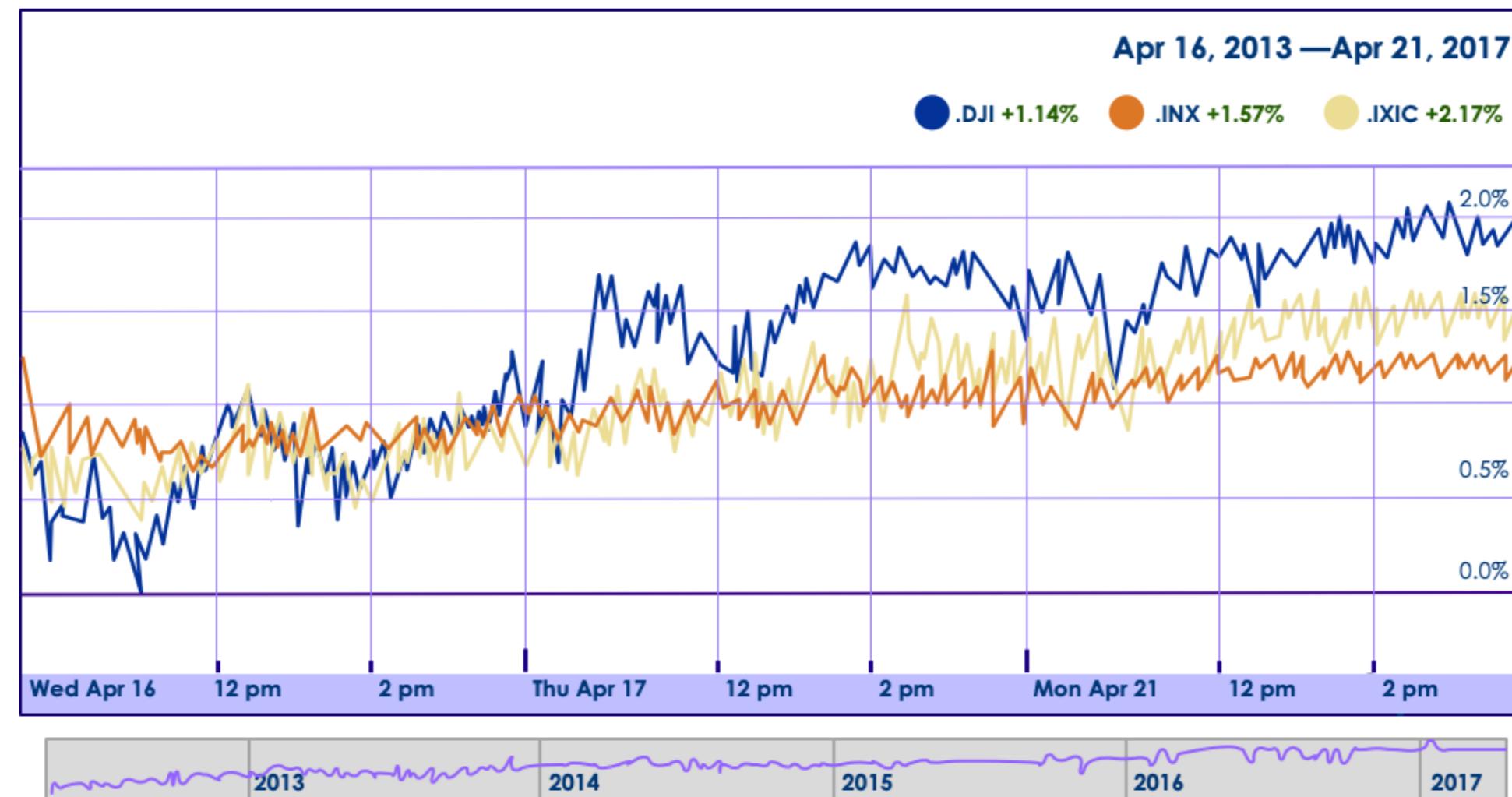
Sometimes to Predict the Future, We Need to Know the Past

- Can you predict the next position of the ball?



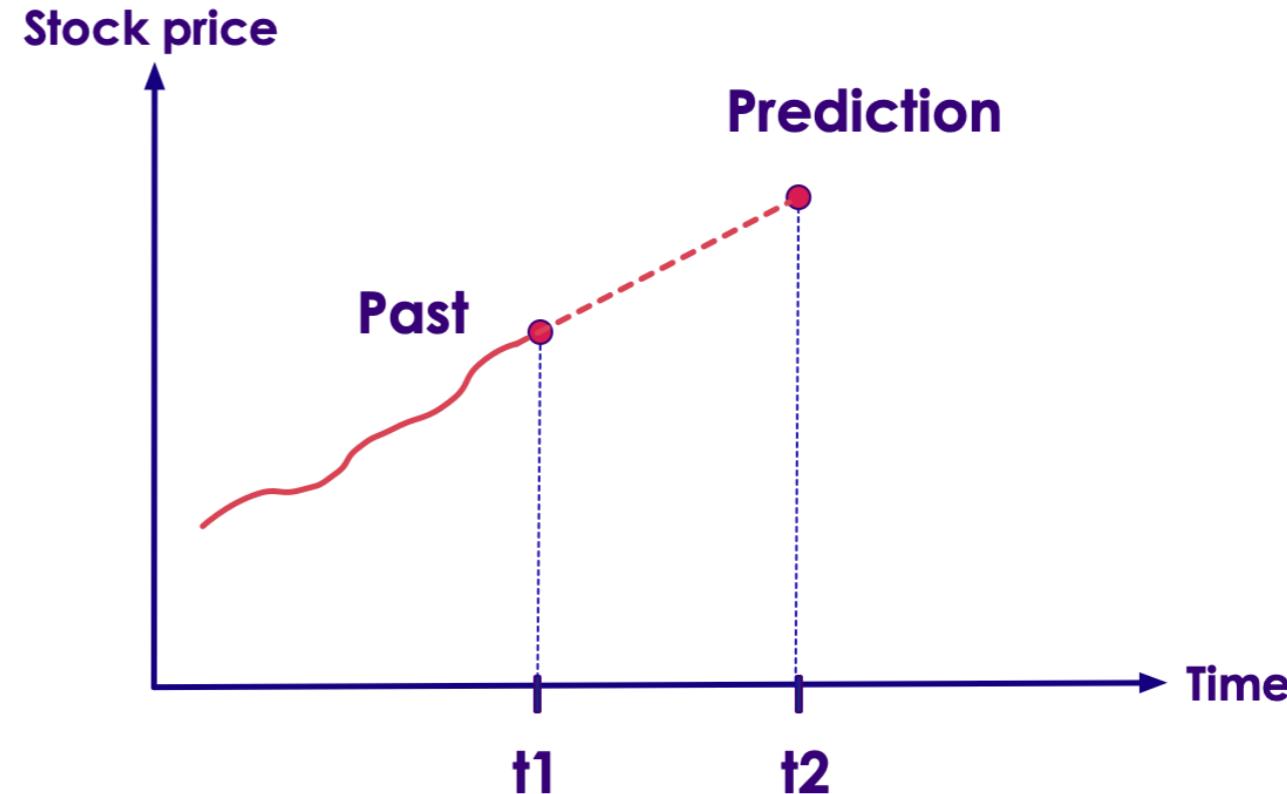
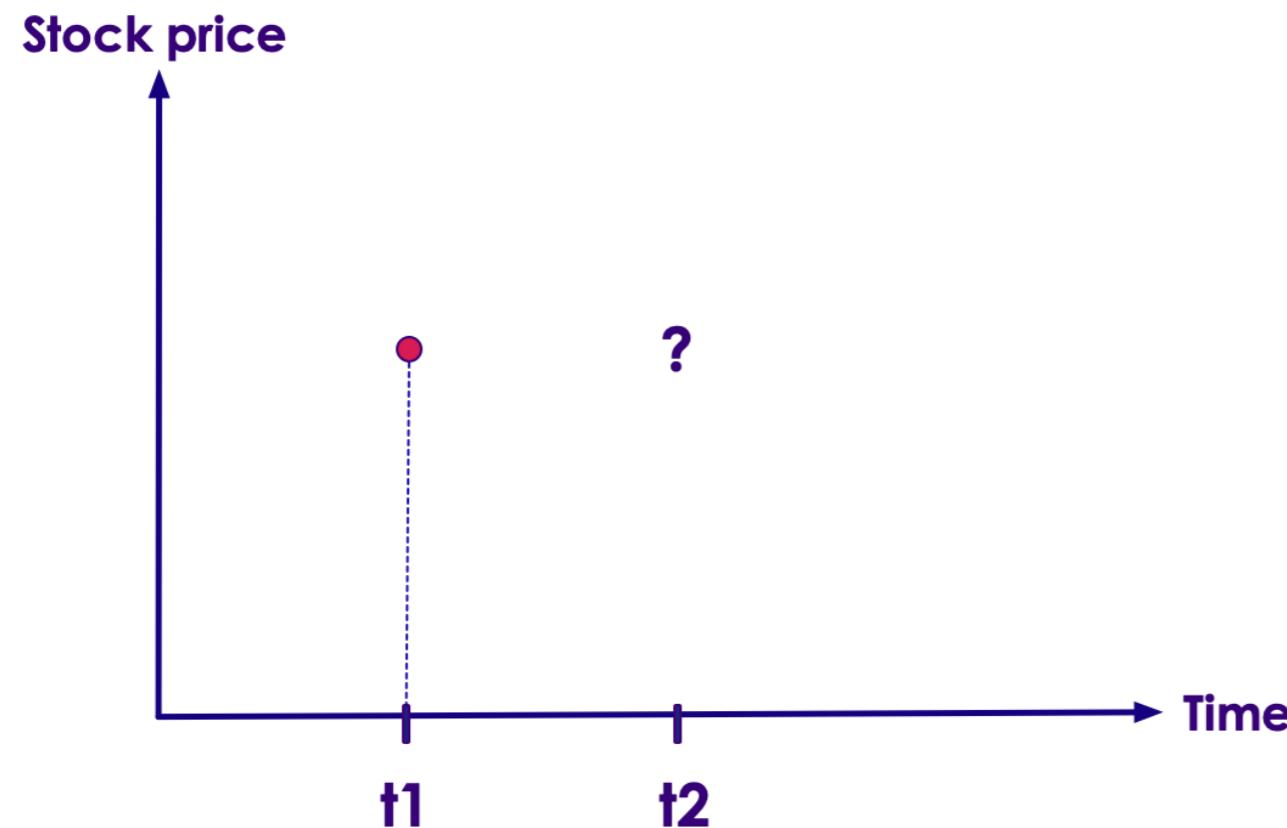
Time Series Data

- Sensor Data (thermostats, weather stations, buoys)
- Stock ticks
- Other cases where the data changes on temporal basis



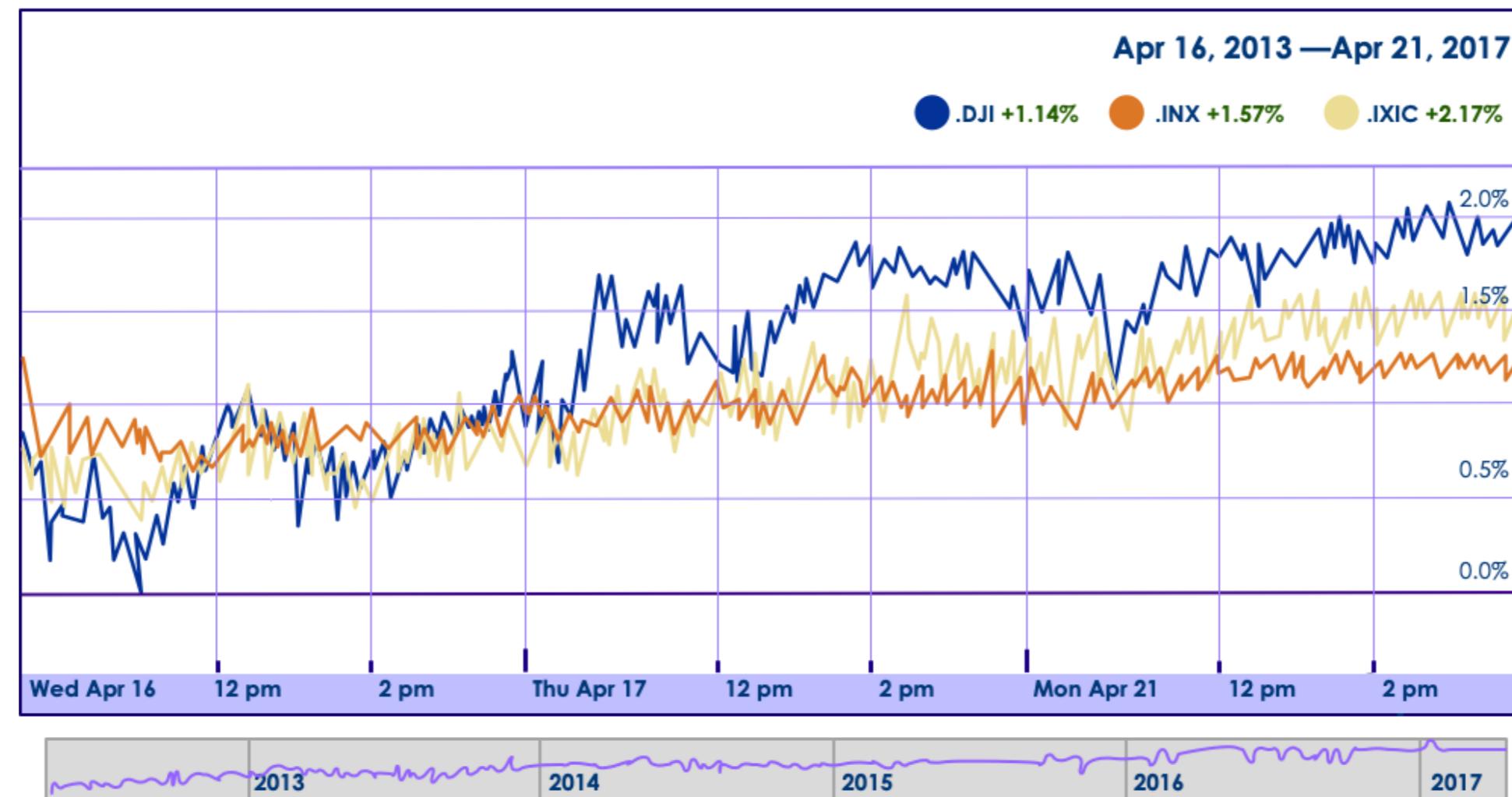
To Predict Time Series Data, We Need to Know the Past Behavior

- For example, what is the stock price in time t_2 ?



Time Series Data

- In Time Series data, the value reflects a change over Time
- So one value isn't so important by itself
- It is the change in the value over time that matters.



Natural Language Processing

Example

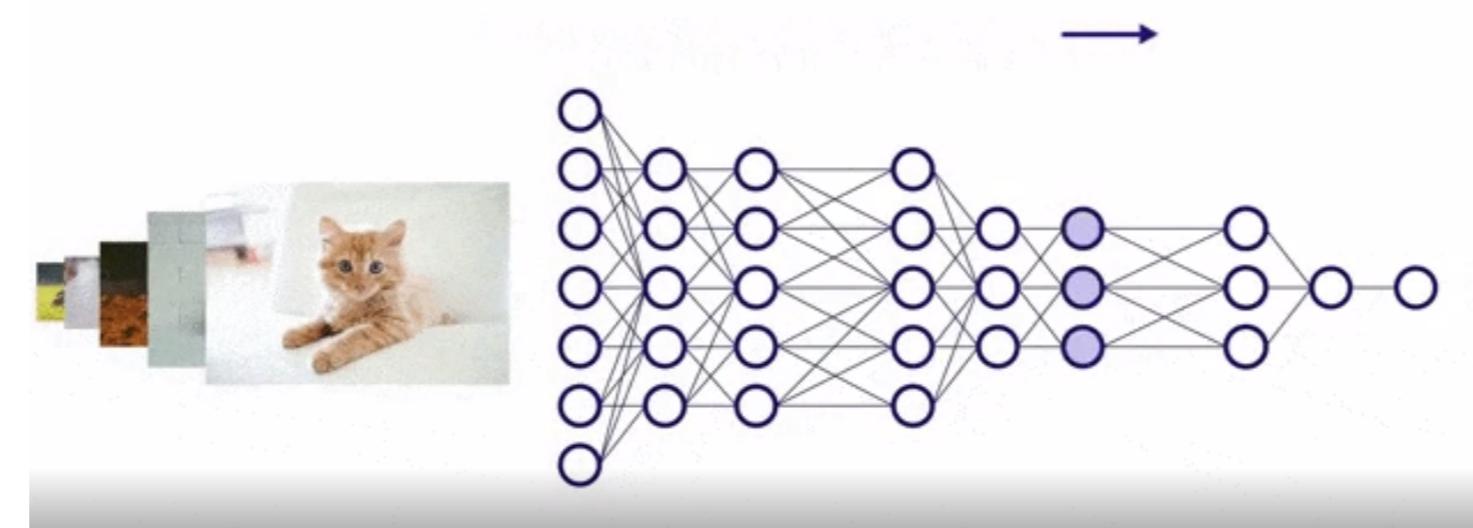
- In NLP, we often need context (history) to determine reference.

```
I was talking with **my** mother about politics.  
"I voted for Hillary Clinton because **she** was most  
aligned with **my** values," **she** said.
```

- Who does the first "she" refer to?
- Who does the "my" refer to?
- We need **state** to be able to figure this out!

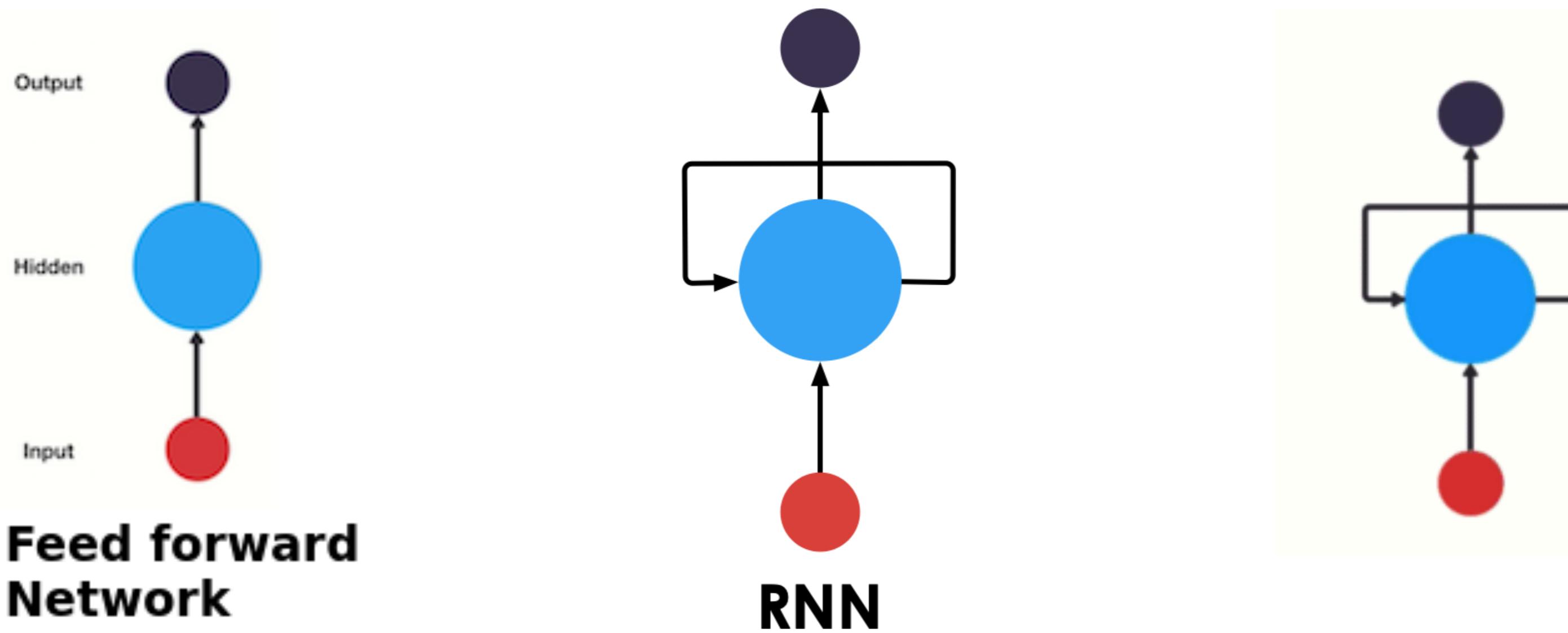
Problems with Feedforward Neural Networks

- Feedforward Neural Networks can model any relationship between input and output.
- However they can't keep/remember **state**
 - The only state retained is weight values from training.
 - They don't remember previous input!
 - For example, in this example, the network doesn't remember the 'previous input' (cat) when predicting the current input
- **Animation** below: [link-youtube](#) | [link-S3](#)



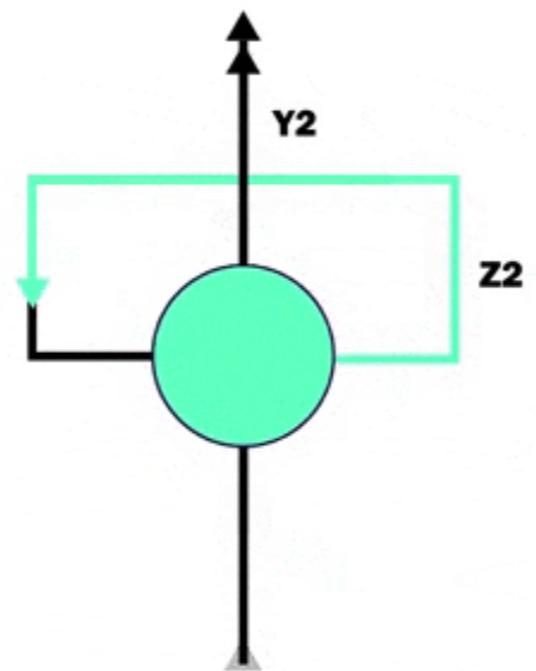
Recurrent Neural Network (RNN)

- In Feedforward Networks, data flows one way, it has **no state or memory**
- RNNs have a 'loop back' mechanism to pass the current state to the next iteration



[Animation link](#)

RNN Animation



- **Animation:** [Link-Youtube](#) | [Link-S3](#)

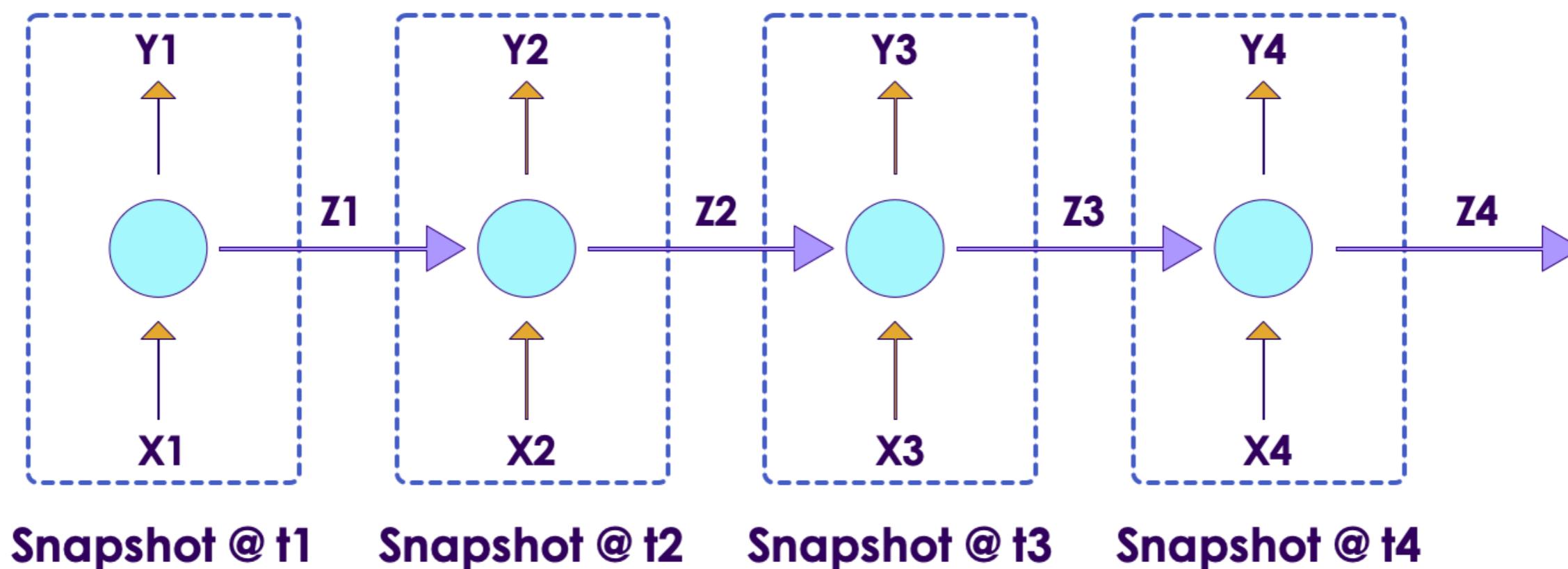
RNN Unrolling Through Time



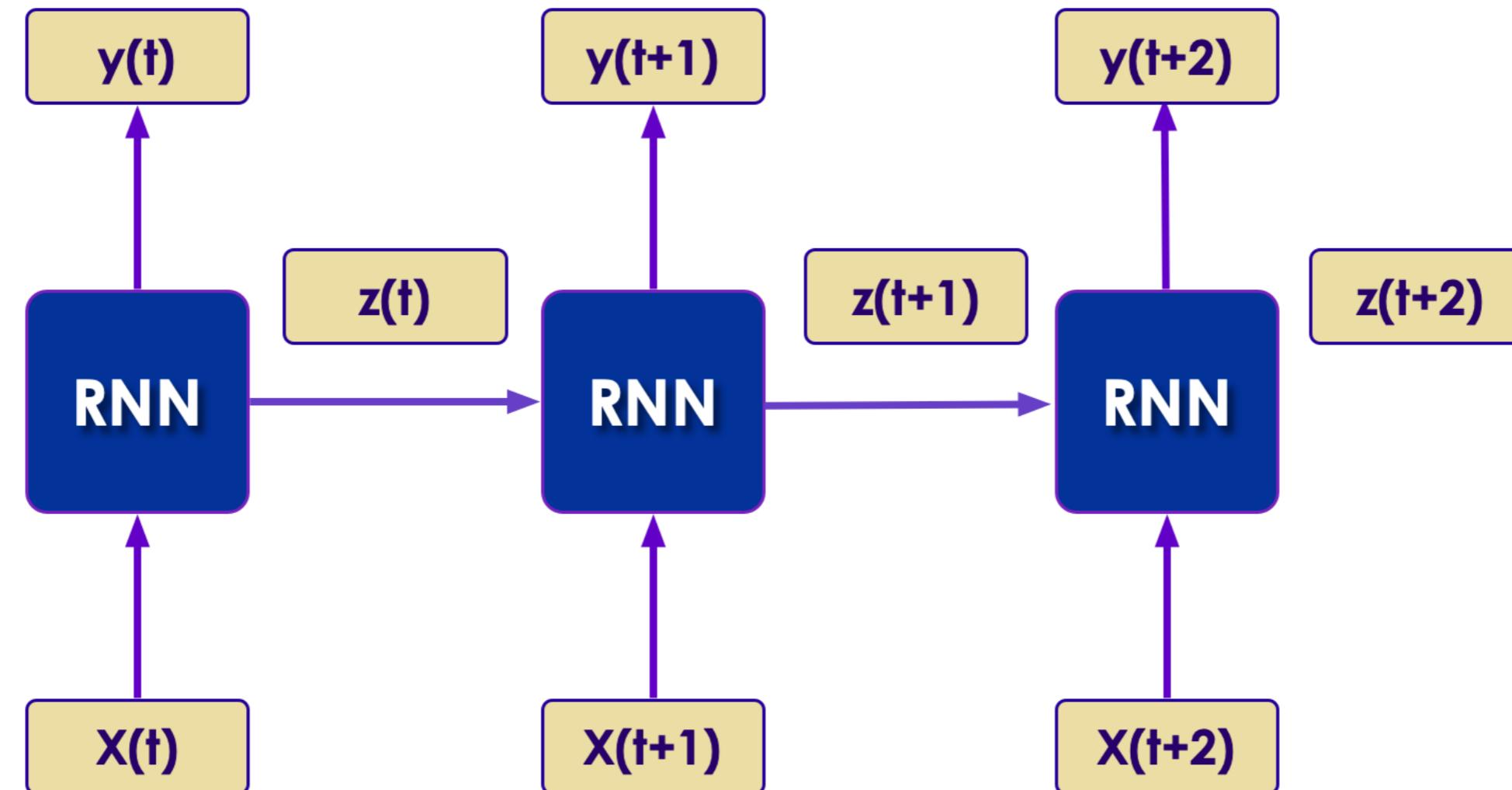
- Animation [link1 \(gif\)](#), [link2 \(mp4\)](#), [\(Source\)](#)

Unrolling Through Time

- A recurrent connection now has a time dimension
- Every output from the neuron goes back to the input at the next time.
- One way to picture this is called unrolling through time
 - Like taking a picture/snapshot of the network for each time frame
- This means that each neuron is like a chain of neurons, one for each time slice.

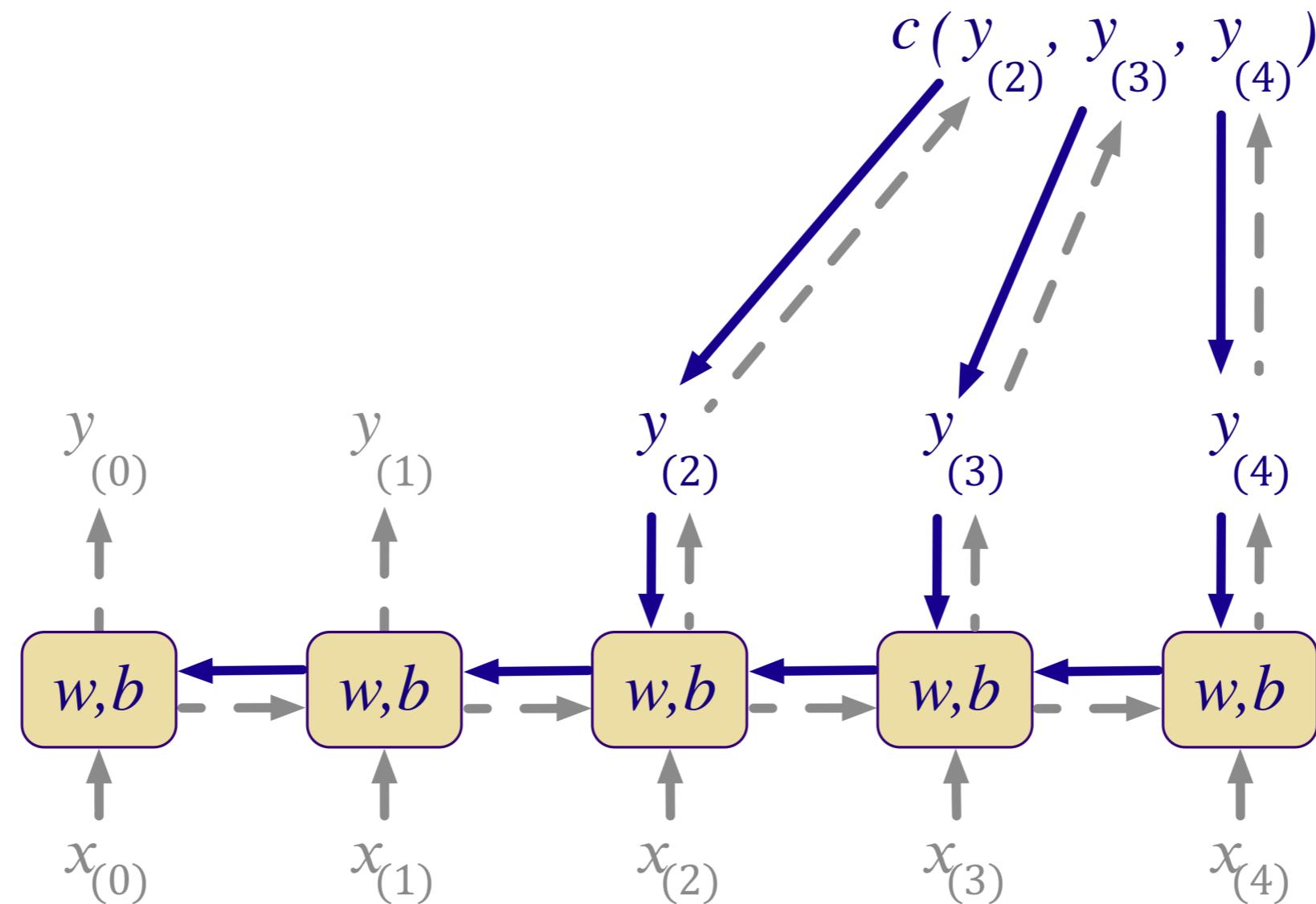


Unrolling Through Time



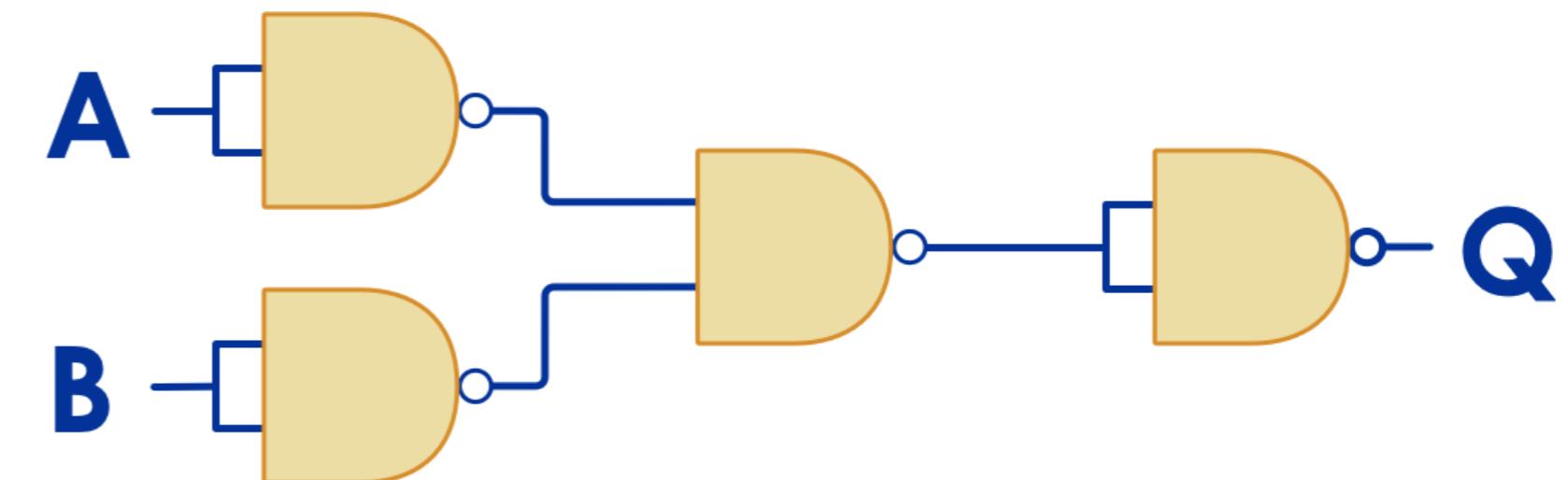
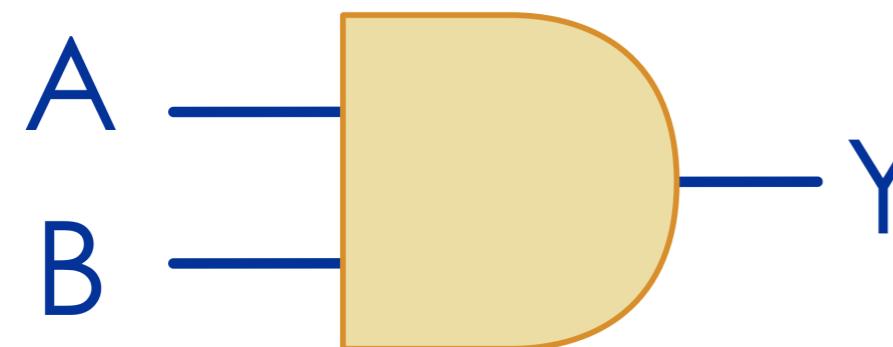
Backpropagation Through Time

- We can use Gradient Descent / Backpropagation to train a recurrent network.
- But now, we have to maintain backpropagation through time.
- For unrolling, we can treat each time step as a new layer.
 - It's really not any different than conventional backpropagation once unrolled.



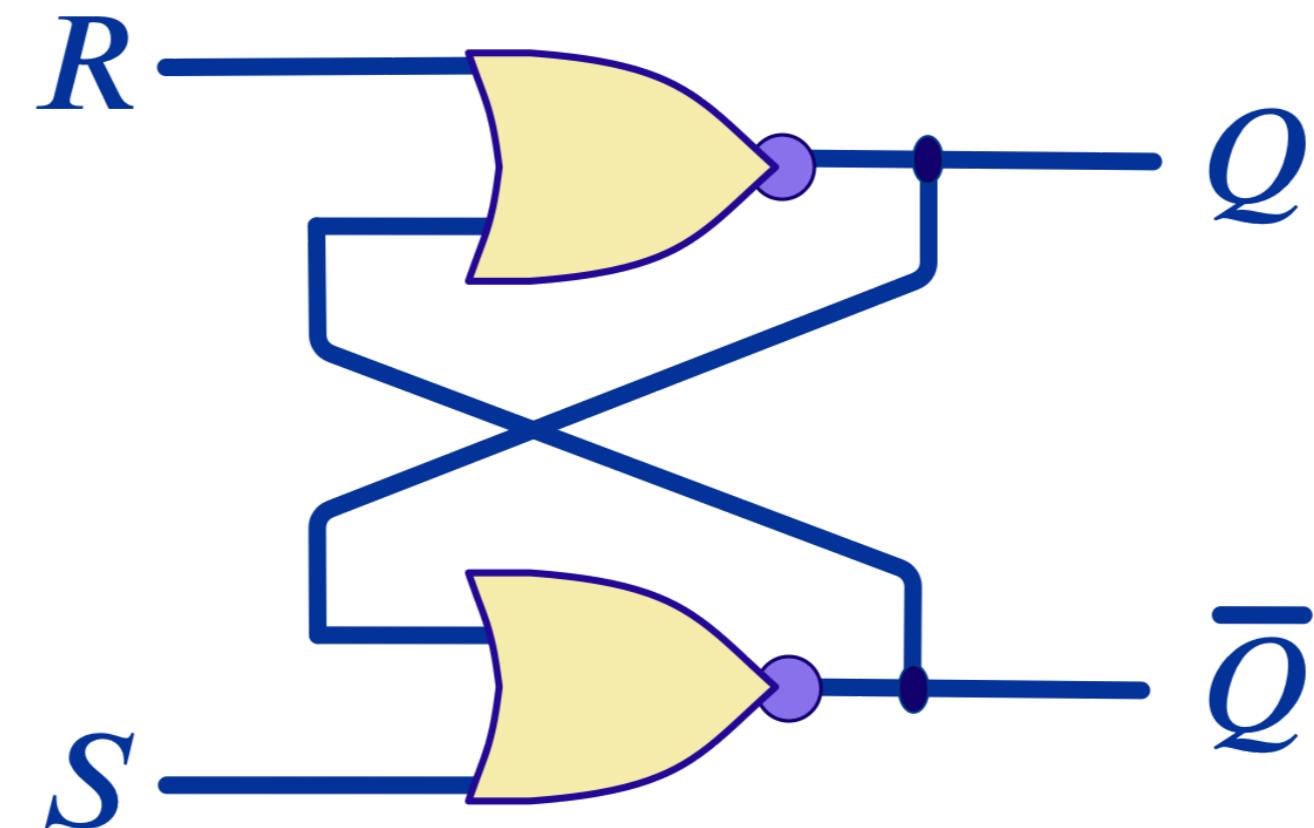
Analogy: Hardware Combinatorial Logic

- In Computer Hardware, we can model logic functions with **gates** :
 - AND, OR, NOR, NAND, etc
 - These can in theory produce any binary output desired from inputs.
 - This is similar to feedforward neural network.
- However, Combinational Logic cannot maintain state.



Feedback Circuits to Maintain State

- To maintain state, we need some kind of memory unit.
- Memory is maintained using feedback, feeding the outputs back into the input.
- By introducing feedback, we can maintain state.
- Note the feedback loops!
- This is a computer hardware memory cell.

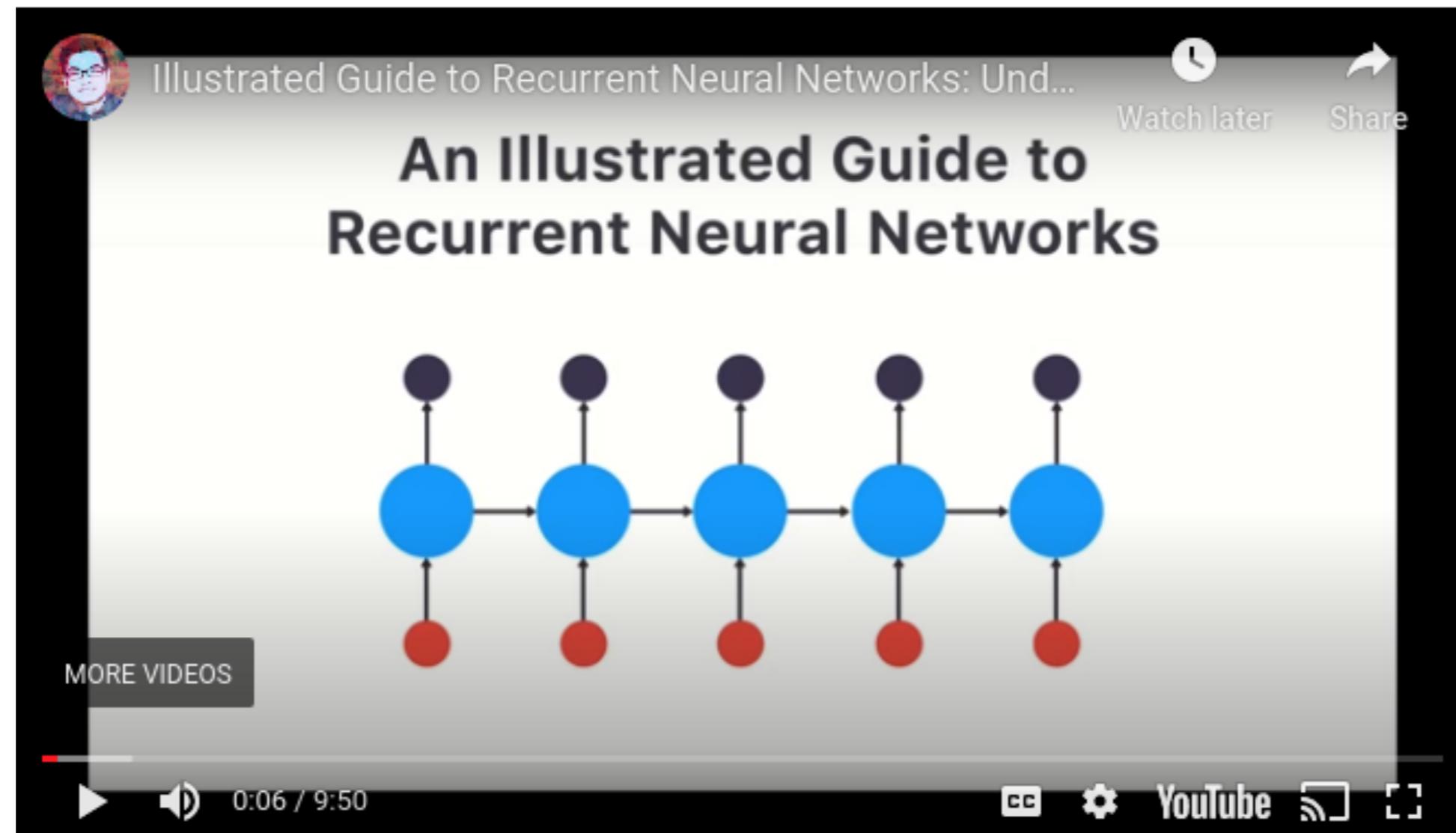


Feedback in RNNs

- RNNs have Feedback
 - The output of some layers feeds back to the input of others
- The Human Brain is a RNN
 - Your brain loops and cycles connections, and allows for state management
- Problem: cycles introduce instability
 - Very hard to train a model with random types of cycles
 - Easily can lead to unpredictable results
 - Positive feedback can lead to instability
 - RNNs are *essentially* positive feedback

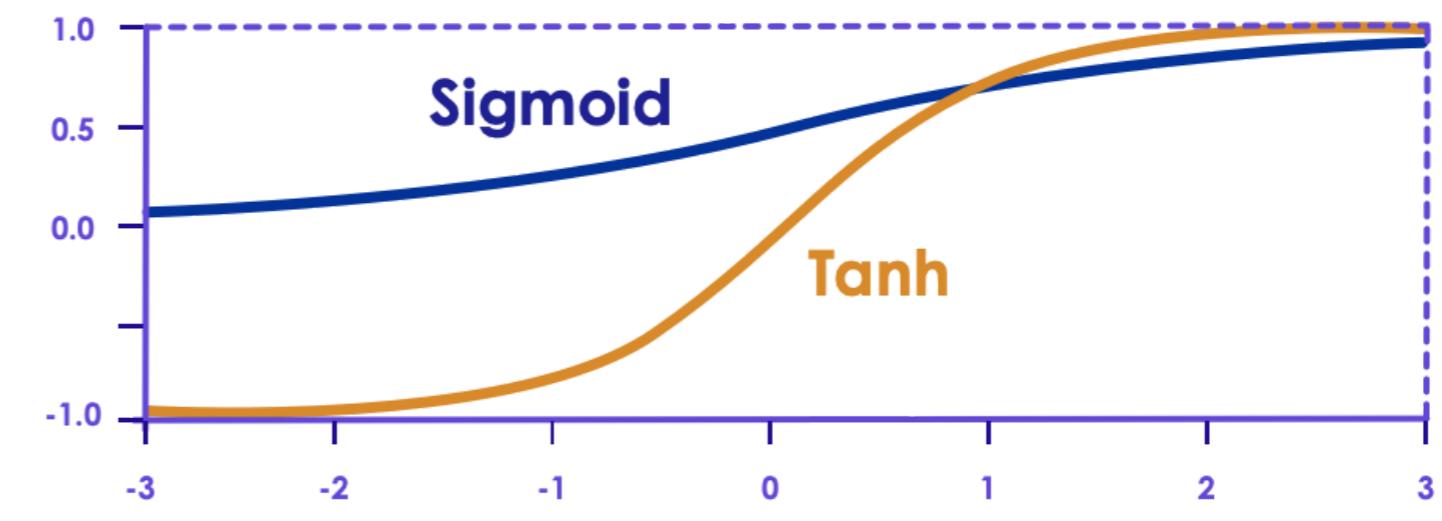
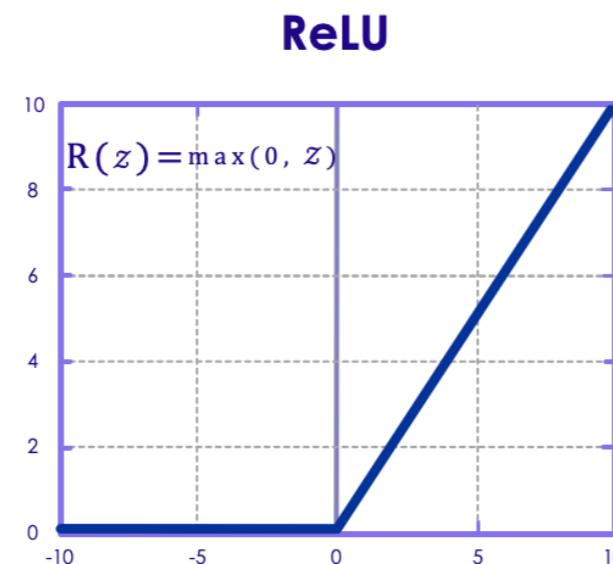
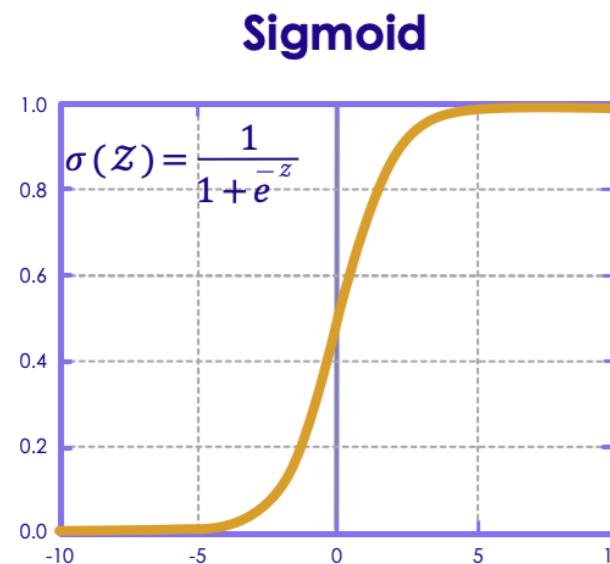
Understanding RNNs

- Great explainer video by Mikael Phi
- And accompanying writeup
- We will discuss a use case starting at 3:50 in the video



Activation Functions for RNNs

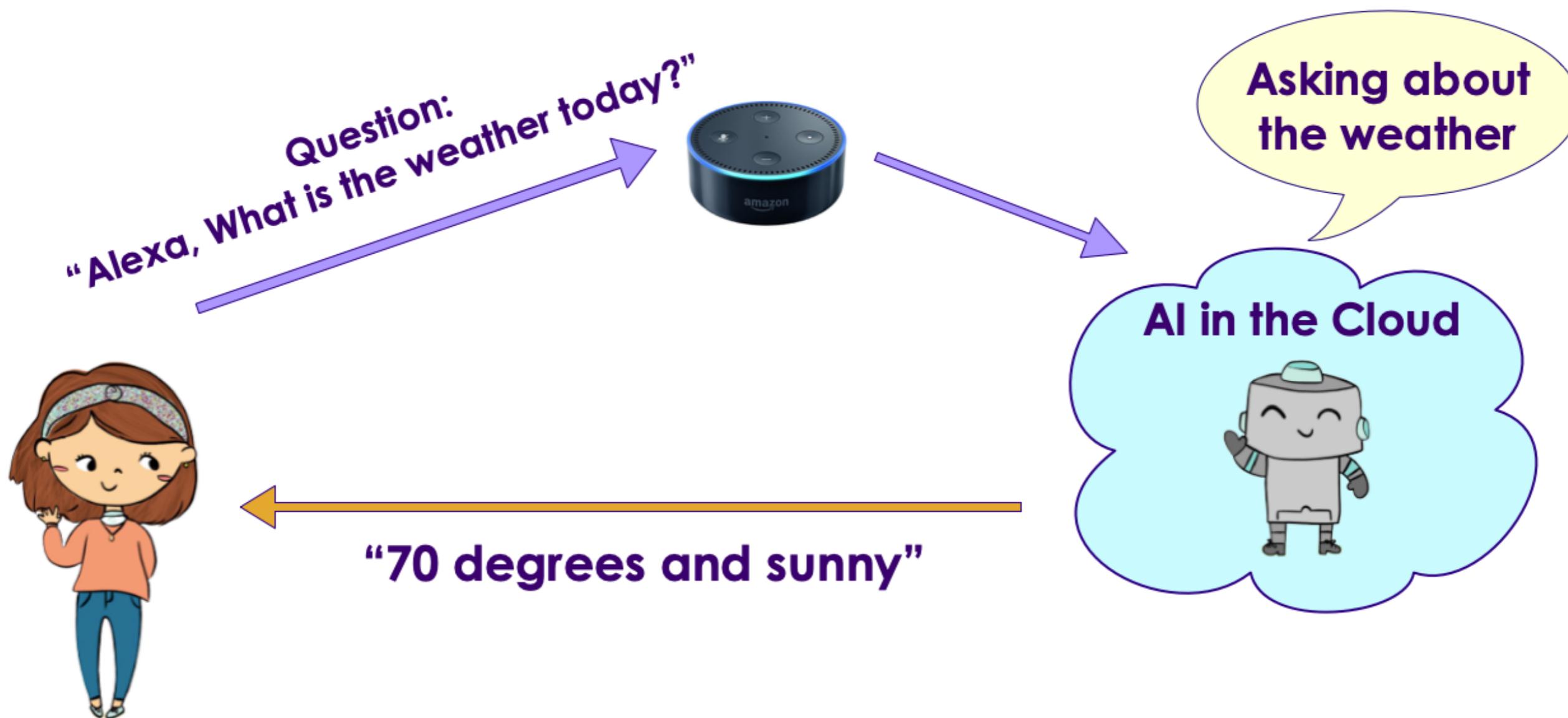
- ReLU and Linear functions have a problem of being unbounded.
 - If we have a self-recurrent loop, they tend to self-reinforce.
 - Think of 'mic amplification feedback'
- Sigmoid is *always* positive (between 0 and 1) which also tends to self-reinforce.
- Tanh is zero centered (between -1 and +1), which is better
- Tanh is the most commonly used in RNNs.



RNN Use Cases

- RNNs are used to analyze sequence data
- RNNs can be used to analyze time series data
 - Stock prices
 - Sensor data
- RNNs can be used for text analysis
 - Language translation
 - Understanding natural text

Text Understanding



Text Processing with RNNs

- This is a usecase illustrated in the explainer video (time 3:50)
 - Adopted with thanks!
- We ask a smart speaker (e.g. Alexa) for time
 - "**What time is it?**"
- First we break the sentence into words
- **Animation** below: Animation link1 (gif), link2 (mp4) | ([Source](#))



Text Processing with RNNs

- Now we feed the sequence of words into RNN
- First word **What** is encoded as a number(vector) **01**
- **Animation** below: Animation link1 (gif), link2 (mp4) | ([Source](#))



Text Processing with RNNs

- Then the next word **time** is fed
- Also the **hidden output** from previous input word **what** is also used
- These two inputs result in the output number **02**
- **Animation** below: [Animation link1 \(gif\)](#), [link2 \(mp4\)](#) | [\(Source\)](#)



Text Processing with RNNs

- The words are fed into sequence
- You can see the color coding of previous outputs influencing the current output
- The final number is **05**
- **Animation** below: [Animation link \(gif\)](#), [link2 \(mp4\)](#) | [\(Source\)](#)



Text Processing with RNNs

- So the text "**What time is it?**" is encoded as **05**
- So if the RNN produces number/vector **05** we know the user asked for time
- If the sequence is changed in anyway (different words and different order) then we wouldn't get **05** as final output
- For example, if the input text is "**What is the time?**" (notice the order is different), the output will definitely NOT be **05**
- **Animation below:** Animation link1 (gif), link2 (mp4) ([Source](#))

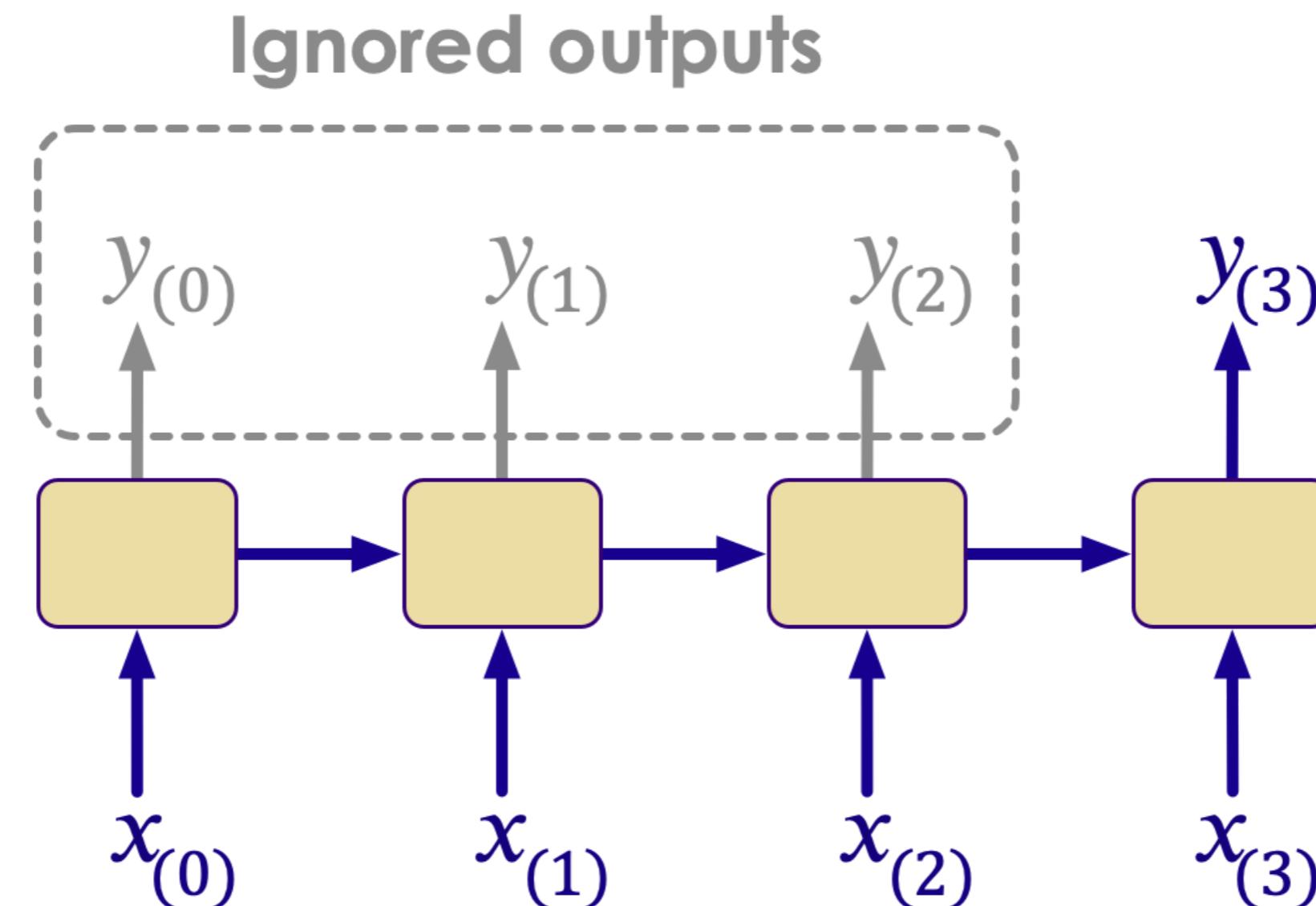


RNN Architectures

- These are the common designs of RNNs
- Sequence to Vector
 - Language processing
- Sequence to Sequence
 - Language translation
- Vector to Sequence

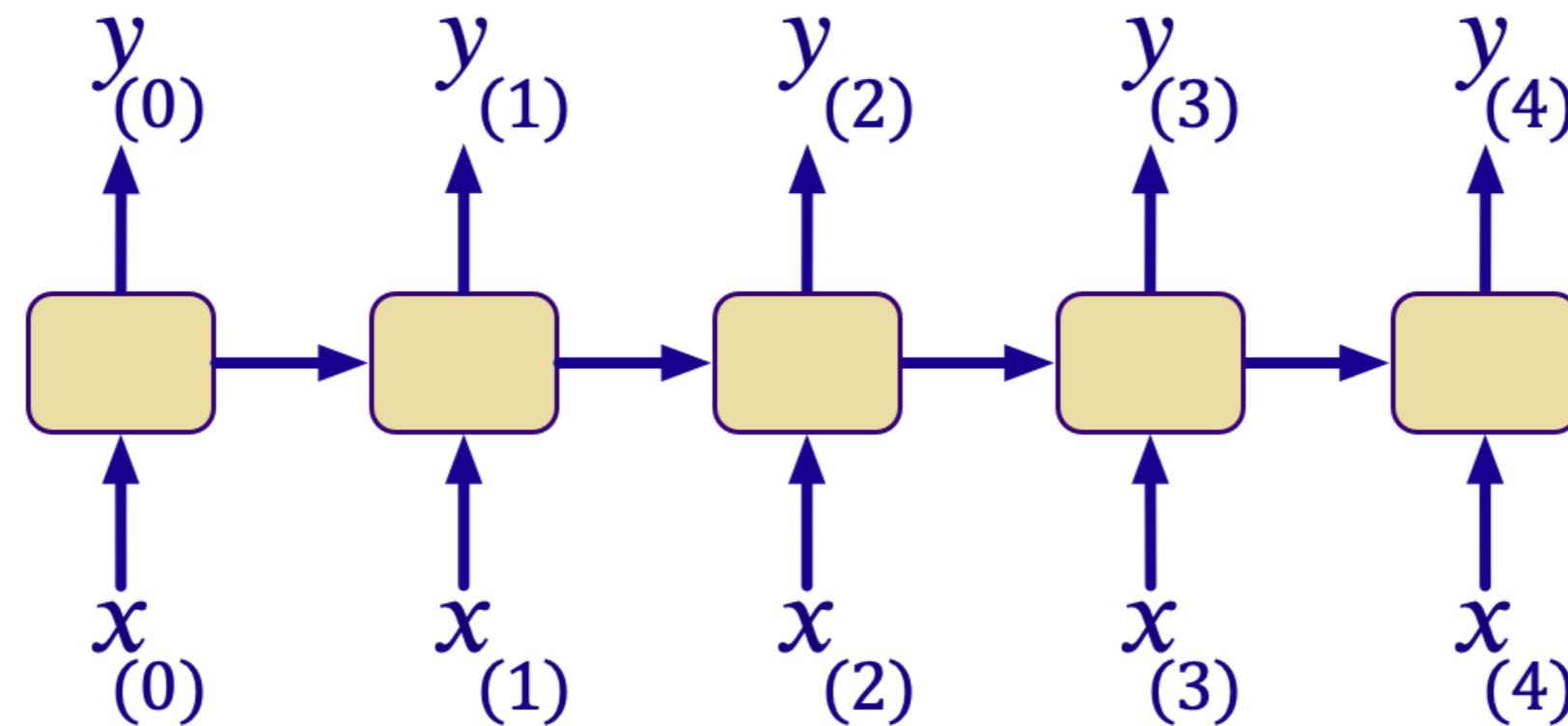
Sequence to Vector

- In the previous example, RNN took sequence as an input (e.g. a sentence) and produced a *vector* as an output
- Natural Language vectorizers can be implemented this way



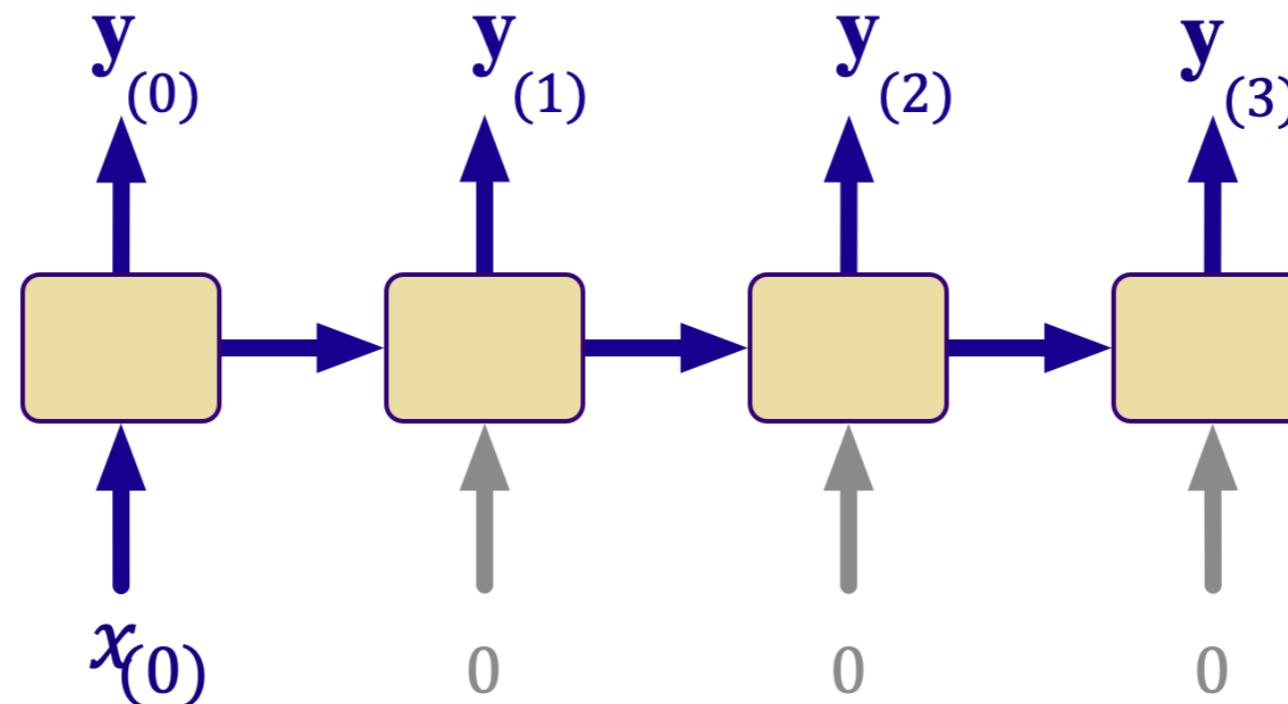
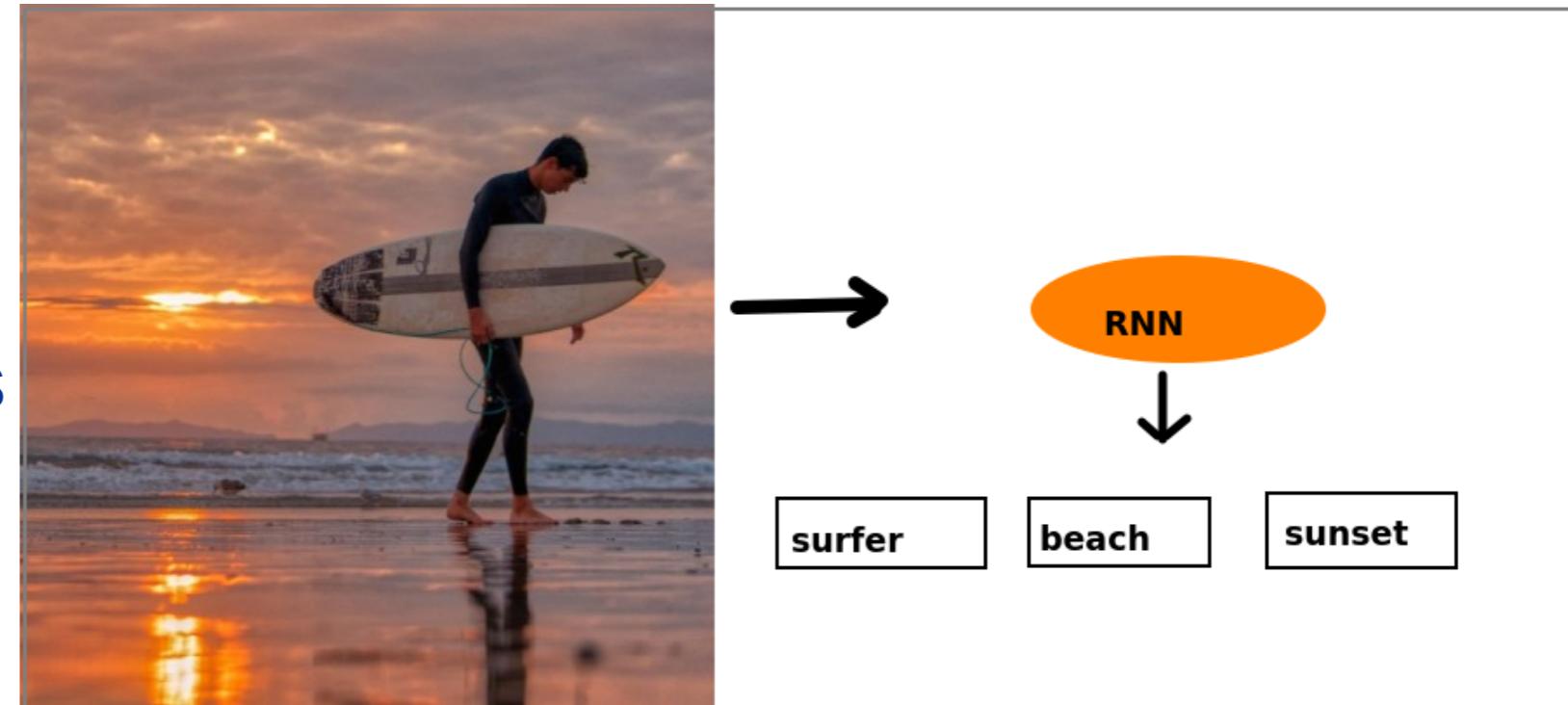
Sequence to Sequence

- An RNN can input a sequence and predict a sequence.
- For example: stock market data
 - The input would be a sequence of stock prices
 - The output would be a prediction of what the next step *would* be
- And language translation (more on this later)



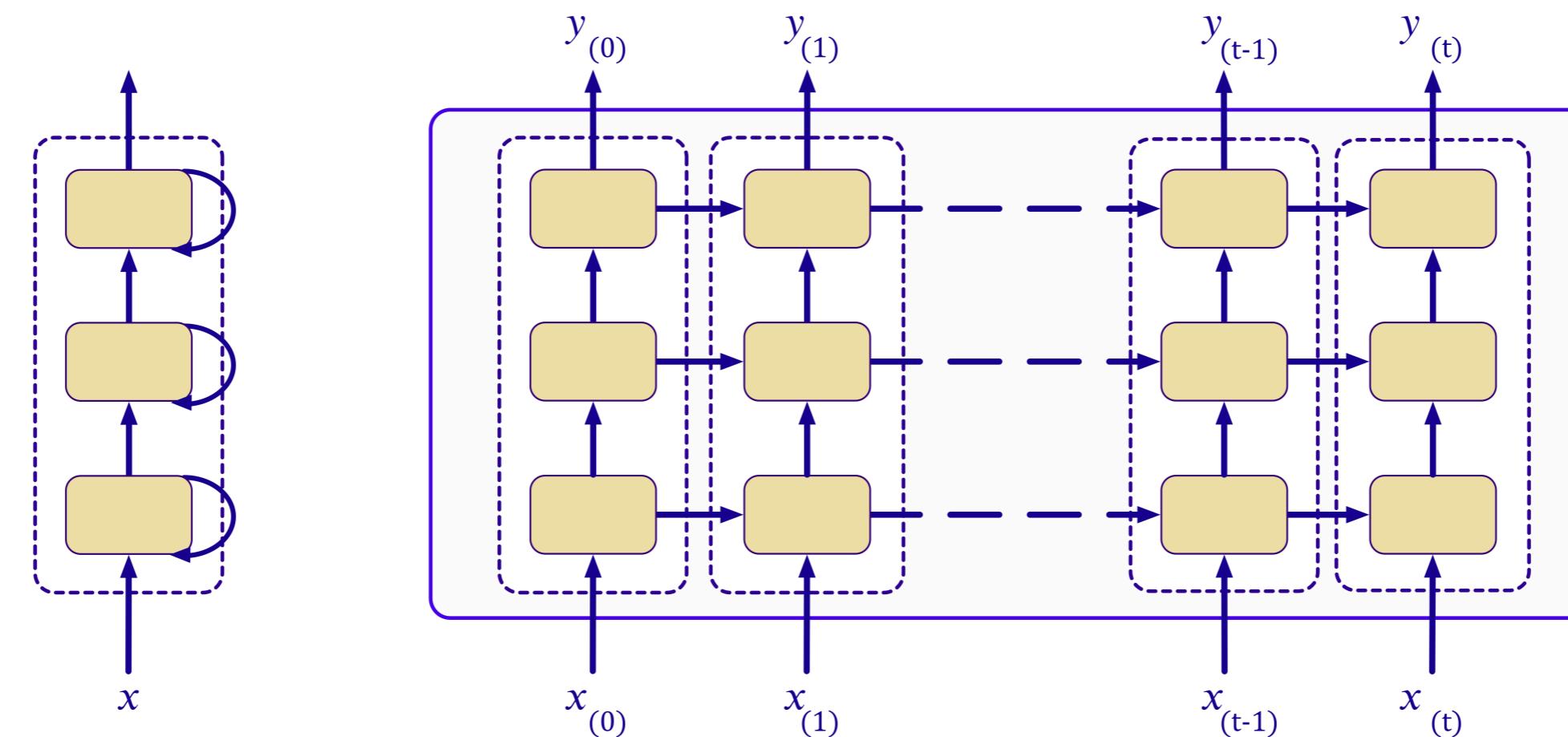
Vector To Sequence

- The Network will take a vector as an input and produce a sequence as an output
- Examples: Image annotation. Image is a vector, annotation is a character sequence.



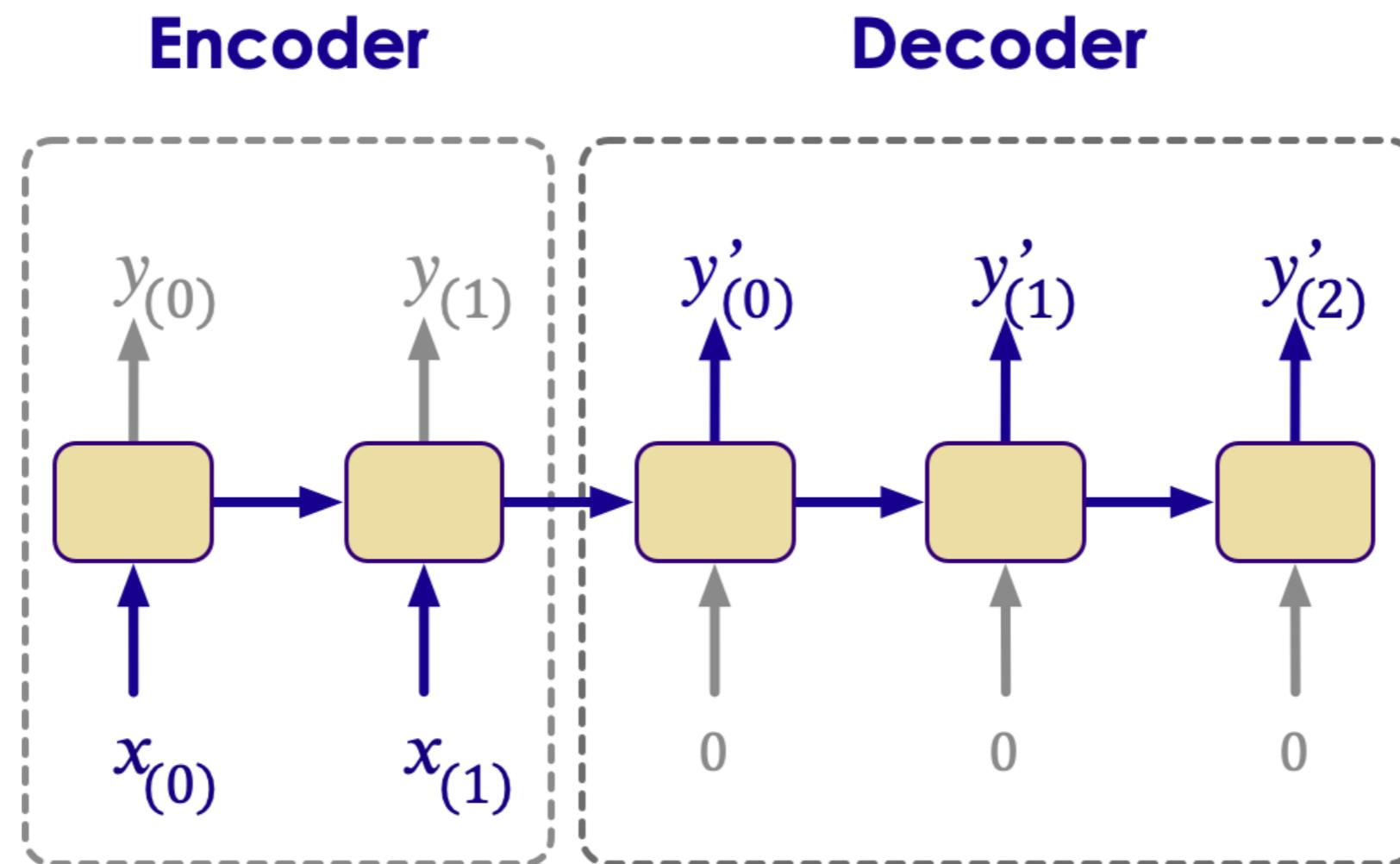
Deep RNNs

- RNNs can also be "deep".
 - Sequentially connected neurons in one layer are not considered "deep".
- So far we have only looked at single layer RNNs.



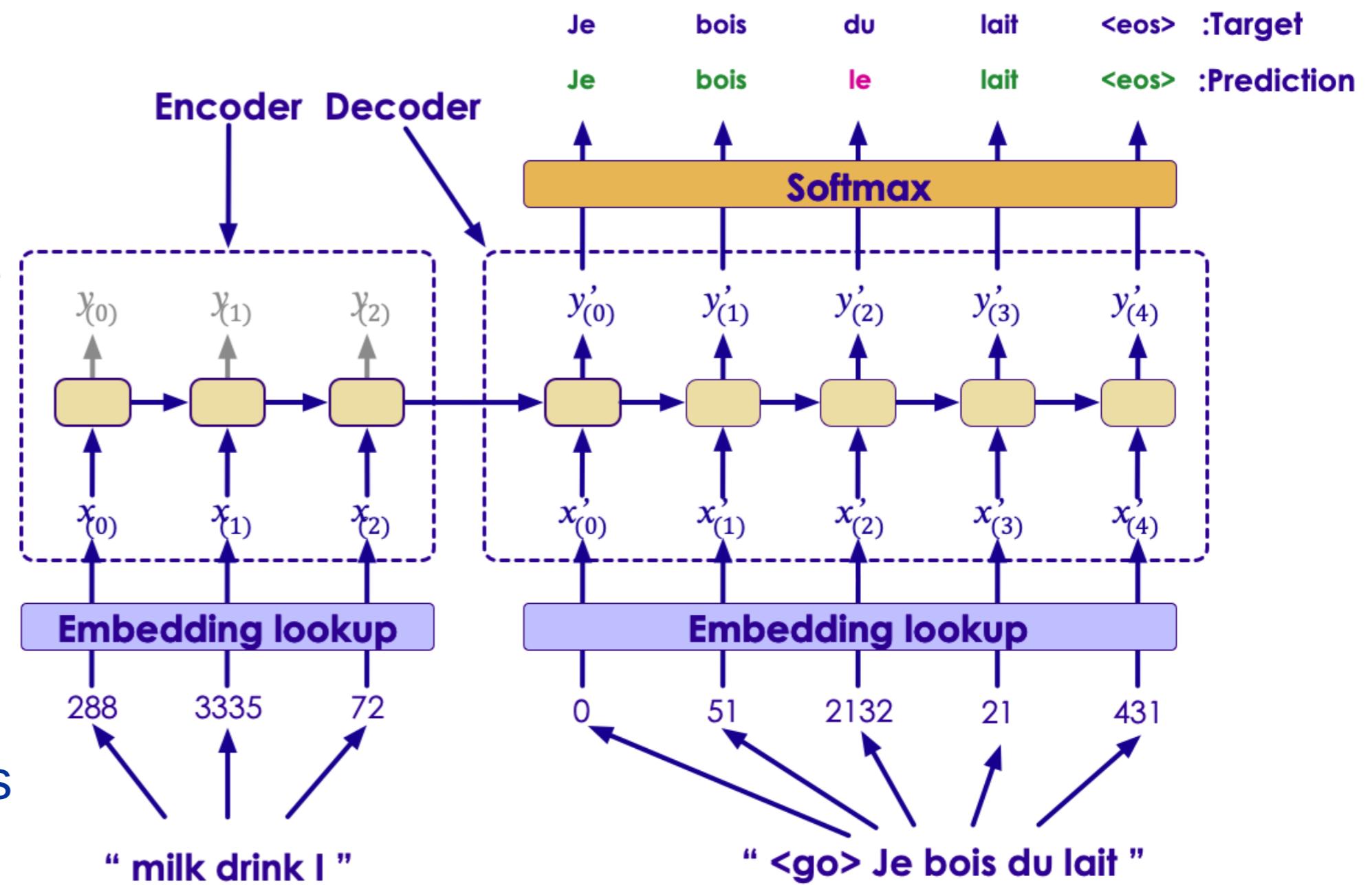
Encoder-Decoder

- An encoder-decoder network takes a sequence as input and produces a sequence as output
- Similar to an autoencoder, but for recurrent neural networks.
- Used in language translation



Machine Translation Model

- Machine translation model is essentially a deep recurrent neural network
- They take an input sequence (English sentence) and produce output sequence (French sentence)
- "I drink milk" \rightarrow "je bois du lait"
- The following example shows how this is done.

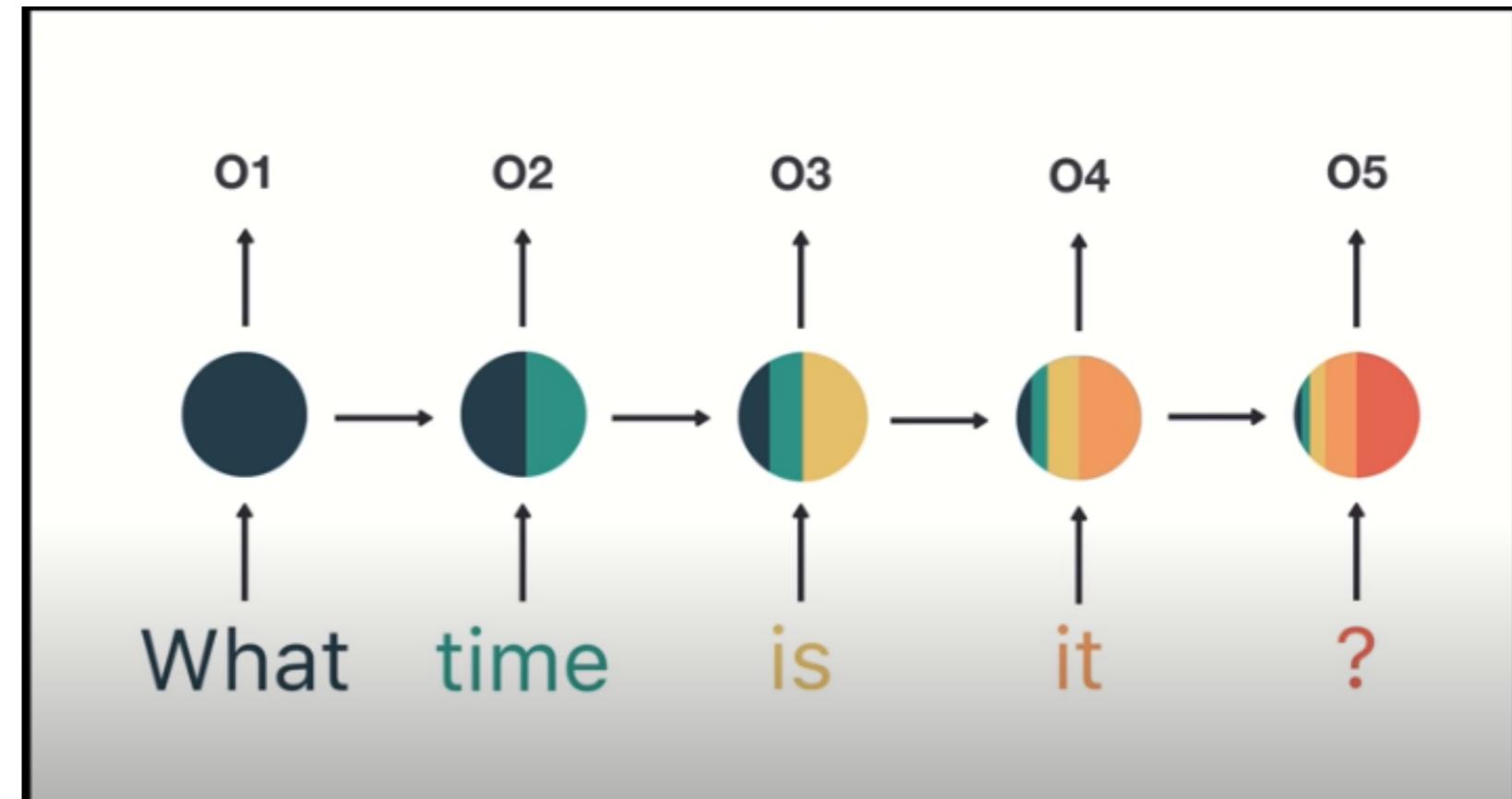


RNN Advantages

- Can process input of any length
 - e.g. input sentences can be arbitrarily long
- Model size not increasing with size of input
 - Same size model can process text of any length (typically)
- Computation takes into account historical information
- Weights are shared across time

RNNs Have Short Term Memory

- The **hidden state fades over multiple steps**
- In this diagram below, we can see the 'influence' of **word 'what'** (**color black**) is diminishing with each step
 - And in the last step **word '?'** the color black is almost non-existing

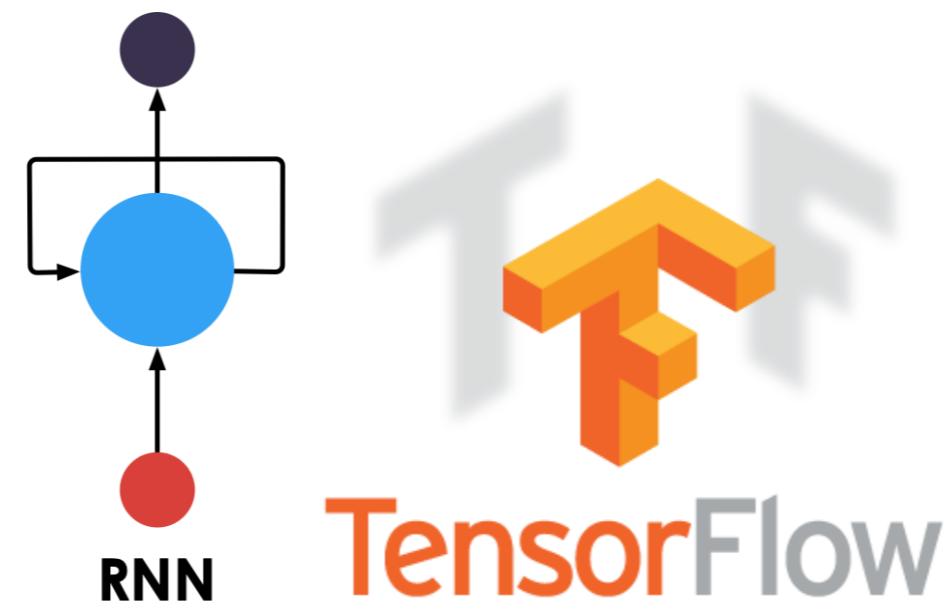


- **RNNs suffer from short term memory**
- What this means is, RNNs can't remember 'long sequences'
 - They can't process long sequences well (e.g. really long sentences)
- Source

RNN Drawbacks

- RNNs can be difficult to train
- Computation being slow
- Stability is a problem
- RNNs suffer from short term memory
 - Difficulty of accessing information from a long time ago (tend to forget earlier information)
- Sequential dependencies limits parallelization opportunities.
- Architectures are complex
- Sometimes CNN can be a better solution.

RNNs in TensorFlow



RNNs in TensorFlow

- RNNs are implemented in `tf.keras.layers.SimpleRNN`

```
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential()

# here is RNN
model.add(keras.layers.SimpleRNN(
    units=32,
    input_shape=(1,step),
    activation="relu"))

## add any other layers
```

A Simple RNN Walkthrough

- Let's walk through a very simple RNN
- We will build a simple RNN to predict a **SINE Wave**

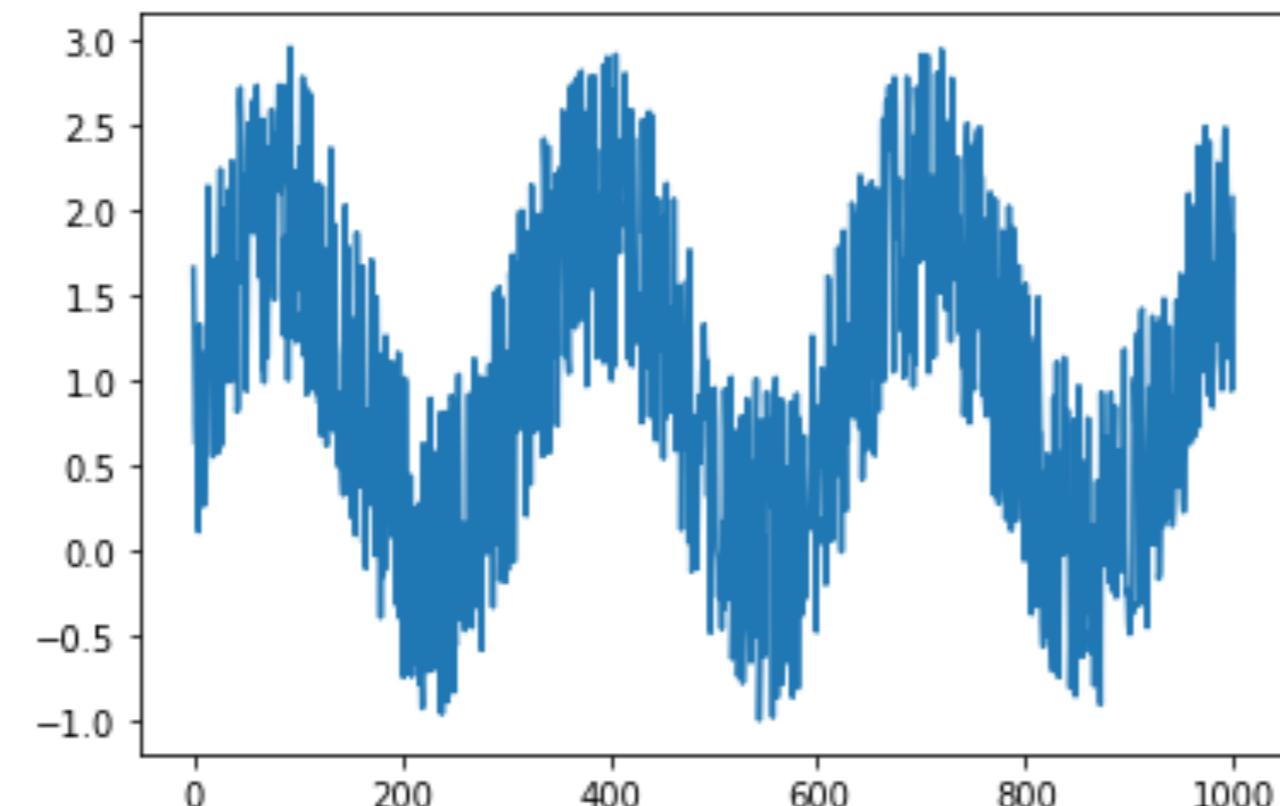
RNN Example: Sine Wave Prediction

```
# Generate 1000 samples
samples = 1000
training_samples = 800

t=np.arange(0,samples)

## Generating a sine wave, with some noise
x=np.sin(0.02*t)+2*np.random.rand(samples)

## print data
df_orig = pd.DataFrame(x)
## plot data
plt.plot(df_orig)
df_orig
```



0	1.6565
1	1.1690
2	0.7687
3	0.6216
4	1.0349
...	...
995	1.5161
996	1.1259
997	1.8623
998	0.9390
999	2.0677
1000 rows × 1 columns	

Shaping Data for RNN

- RNN takes sequence of input and produces output
- This is called **step**
- Consider the following sequence:
 $x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
- For **step=1** the input(x) and prediction(y) looks like this

X	Y
1	2
2	3
3	4
4	5
5	6

Shaping Data for RNN

- Input:

$x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

- Step=2

- Input, Output looks like this:

X	Y
1,2	3
2,3	4
3,4	5
4,5	6
5,6	7

Shaping Data for RNN

- Input:

$x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

- Step=3

- Input, Output looks like this:

X	Y
1,2,3	4
2,3,4	5
3,4,5	6
4,5,6	7
5,6,7	8

Shape Data

```
## vectorize the data

def convertToMatrix(data, step):
    X, Y =[], []
    for i in range(len(data)-step):
        d=i+step
        X.append(data[i:d,])
        Y.append(data[d,])
    return np.array(X), np.array(Y)

x_train,y_train =convertToMatrix(train,step)
x_test,y_test =convertToMatrix(test,step)

print ("x_train.shape", x_train.shape)
print ("y_train.shape", y_train.shape)
print ("x_test.shape", x_test.shape)
print ("y_test.shape", y_test.shape)

## See data
df = pd.DataFrame(x_train, y_train)
df

## Finally, we'll reshape trainX and testX to fit with the Keras model.
## RNN model requires three-dimensional input data.

x_train = np.reshape(x_train, (x_train.shape[0], 1, x_train.shape[1]))
x_test = np.reshape(x_test, (x_test.shape[0], 1, x_test.shape[1]))

print ("x_train.shape", x_train.shape)
print ("y_train.shape", y_train.shape)
print ("x_test.shape", x_test.shape)
print ("y_test.shape", y_test.shape)
```

Shaping Data

```
x_train.shape (800, 4)  
y_train.shape (800, )  
x_test.shape (200, 4)  
y_test.shape (200, )
```

```
x_train.shape (800, 1, 4)  
y_train.shape (800, )  
x_test.shape (200, 1, 4)  
y_test.shape (200, )
```

	0	1	2	3
1.0349	1.6565	1.1690	0.7687	0.6216
0.1124	1.1690	0.7687	0.6216	1.0349
1.3249	0.7687	0.6216	1.0349	0.1124
0.2906	0.6216	1.0349	0.1124	1.3249
1.1567	1.0349	0.1124	1.3249	0.2906
...
-0.0575	0.3849	0.7801	0.6373	0.8977
-0.0575	0.7801	0.6373	0.8977	-0.0575
-0.0575	0.6373	0.8977	-0.0575	-0.0575
-0.0575	0.8977	-0.0575	-0.0575	-0.0575
-0.0575	-0.0575	-0.0575	-0.0575	-0.0575

800 rows × 4 columns

	output	input X			
Y	0	1	2	3	
1.0349	1.6565	1.1690	0.7687	0.6216	
0.1124	1.1690	0.7687	0.6216	1.0349	
1.3249	0.7687	0.6216	1.0349	0.1124	
0.2906	0.6216	1.0349	0.1124	1.3249	
1.1567	1.0349	0.1124	1.3249	0.2906	
...
-0.0575	0.3849	0.7801	0.6373	0.8977	
-0.0575	0.7801	0.6373	0.8977	-0.0575	
-0.0575	0.6373	0.8977	-0.0575	-0.0575	
-0.0575	0.8977	-0.0575	-0.0575	-0.0575	
-0.0575	-0.0575	-0.0575	-0.0575	-0.0575	

800 rows × 4 columns

Creating the Model

- Now the data is ready, we can create the model

```
import tensorflow as tf
from tensorflow import keras

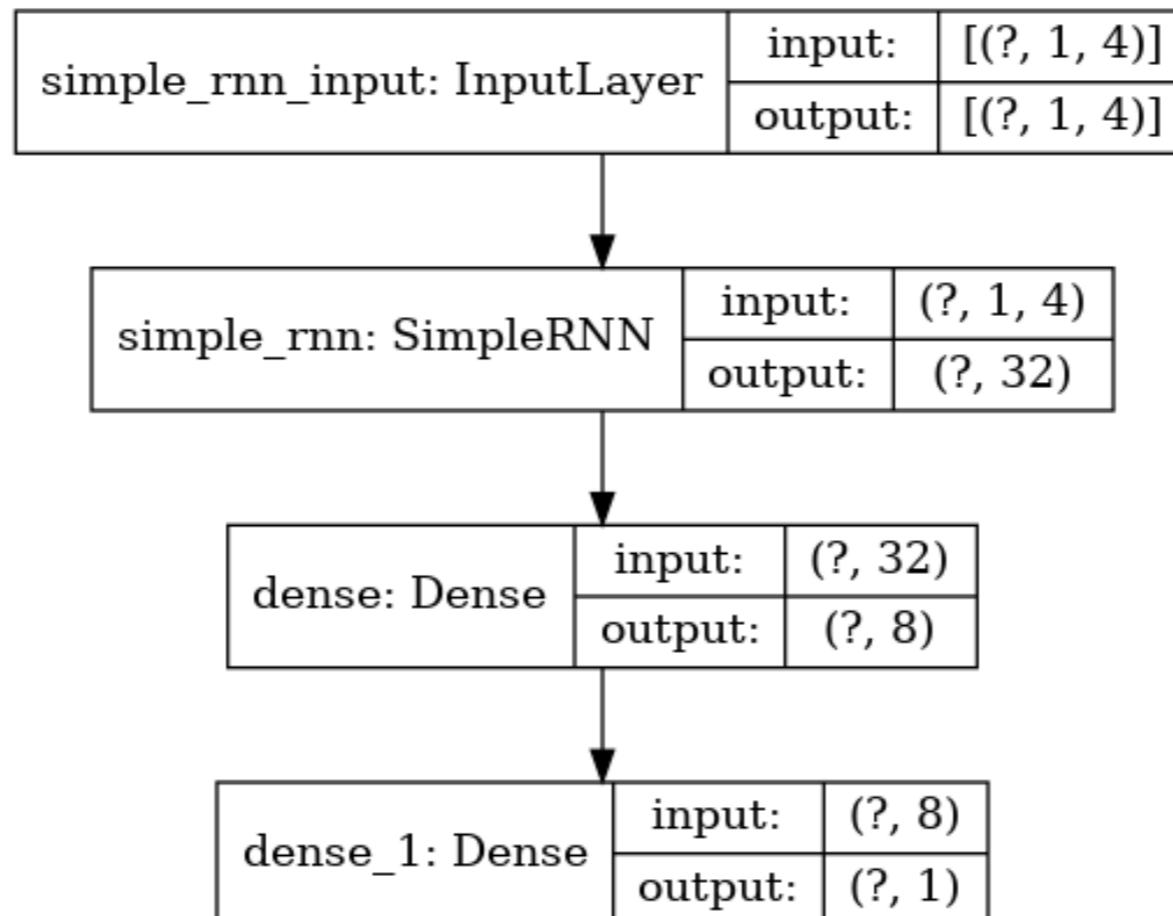
model = keras.models.Sequential()
model.add(keras.layers.SimpleRNN(units=32, input_shape=(1,step), activation="relu"))
model.add(keras.layers.Dense(8, activation="relu"))
model.add(keras.layers.Dense(1))
model.compile(optimizer='rmsprop', loss = 'mse', metrics=['mse'])

model.summary()
tf.keras.utils.plot_model(model, to_file='model.png', show_shapes=True)
```

Model Shape

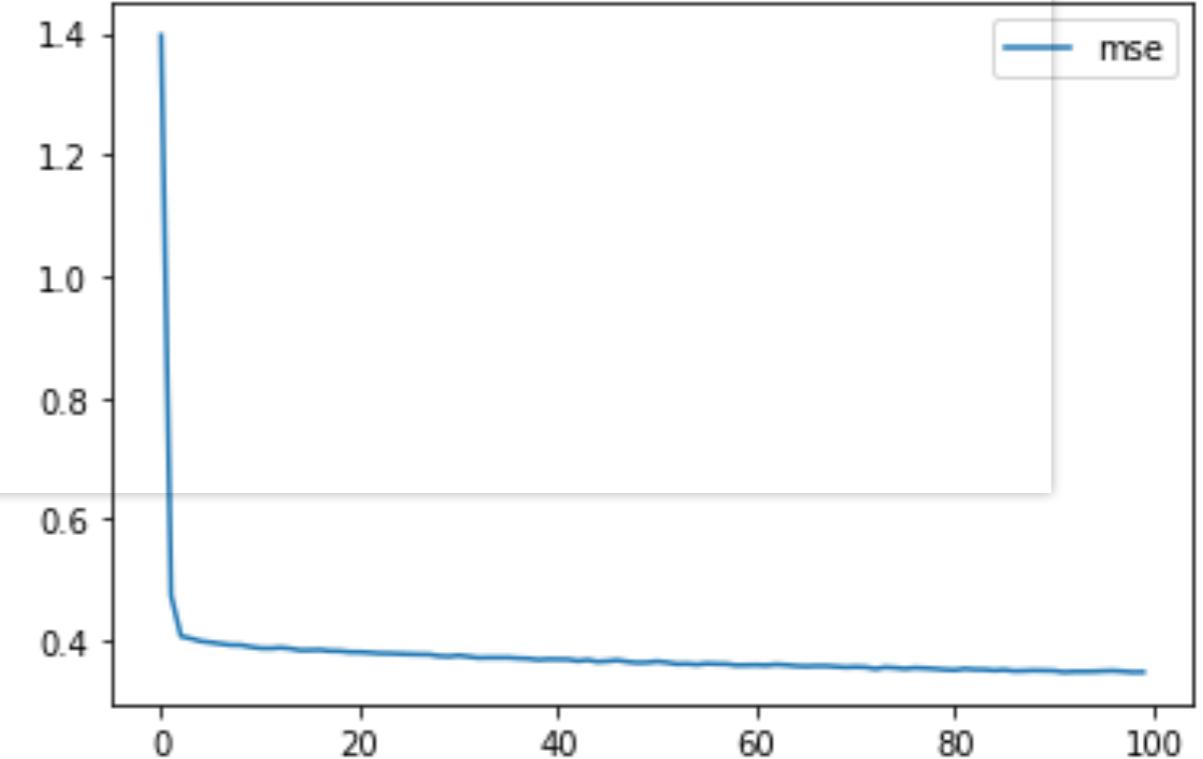
Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
simple_rnn (SimpleRNN)	(None, 32)	1184
dense (Dense)	(None, 8)	264
dense_1 (Dense)	(None, 1)	9
=====		
Total params: 1,457		
Trainable params: 1,457		
Non-trainable params: 0		



Training

```
%time  
  
# Fitting the RNN to the Training set  
history = model.fit(x_train, y_train, epochs=100, batch_size=16)  
  
%matplotlib inline  
import matplotlib.pyplot as plt  
  
plt.plot(history.history['mse'], label='mse')  
plt.legend()
```



```
training starting ...  
Train on 800 samples  
Epoch 1/100  
800/800 [=====] - 1s 1ms/sample - loss: 1.3962 - mse: 1.3962  
Epoch 2/100  
800/800 [=====] - 0s 231us/sample - loss: 0.4745 - mse: 0.4745  
...  
Epoch 99/100  
800/800 [=====] - 0s 194us/sample - loss: 0.3482 - mse: 0.3482  
Epoch 100/100  
800/800 [=====] - 0s 295us/sample - loss: 0.3485 - mse: 0.3485  
training done.  
CPU times: user 1min 56s, sys: 2min 54s, total: 4min 51s  
Wall time: 20.5 s
```

Scoring

```
predict_train = model.predict(x_train)
predict_test= model.predict(x_test)
predicted=np.concatenate((predict_train,predict_test),axis=0)

train_score = model.evaluate(x_train, y_train, verbose=0)
test_score = model.evaluate(x_test, y_test, verbose=0)

metric_names = model.metrics_names
print ("model metrics : " , metric_names)

train_metrics = model.evaluate(x_train, y_train, verbose=0)
for idx, metric in enumerate(metric_names):
    print ("Train Metric : {} = {:.2f}".format (metric_names[idx], train_metrics[idx]))

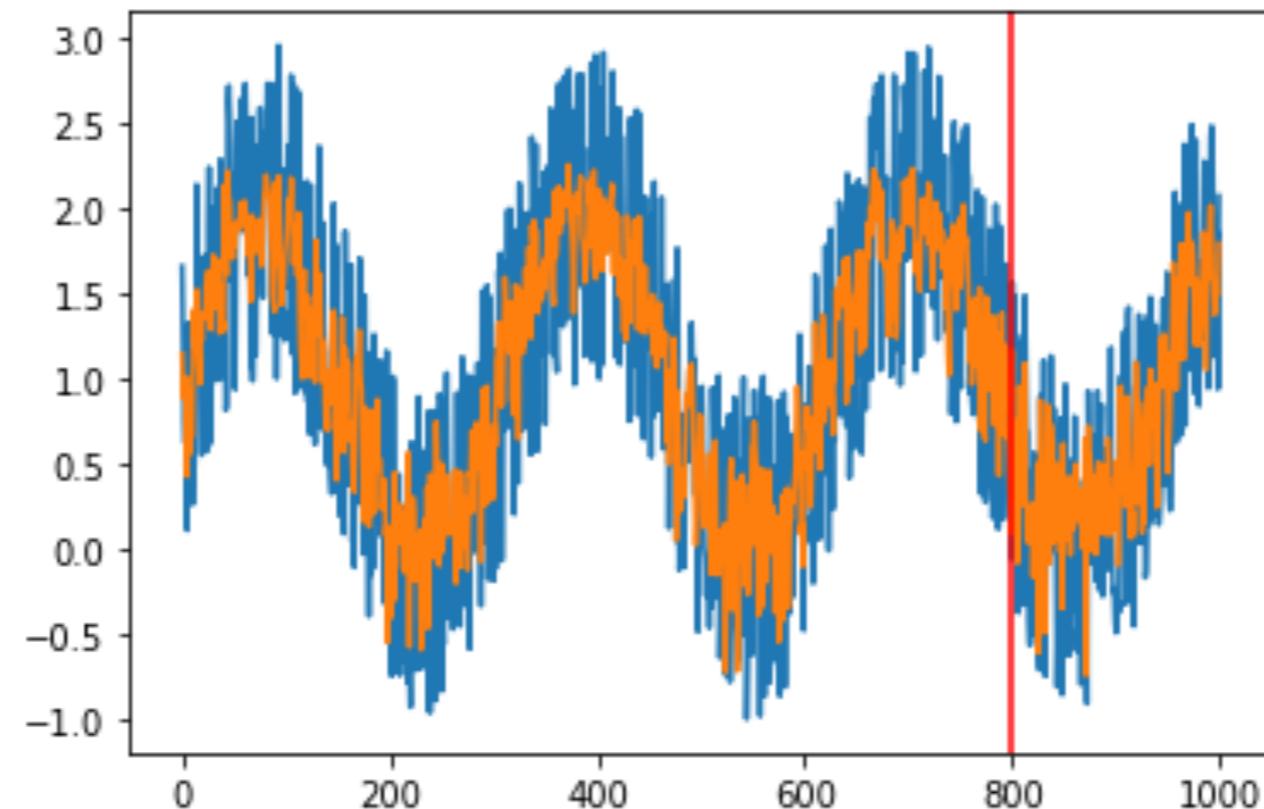
test_metrics = model.evaluate(x_test, y_test, verbose=0)
for idx, metric in enumerate(metric_names):
    print ("Test Metric : {} = {:.2f}".format (metric_names[idx], test_metrics[idx]))
```

```
model metrics : ['loss', 'mse']
Train Metric : loss = 0.34
Train Metric : mse = 0.34
Test Metric : loss = 0.46
Test Metric : mse = 0.46
```

Plot Predictions

- Blue is original data
- Orange is prediction
- It is tracking pretty close!

```
index = df_orig.index.values
plt.plot(index,df_orig)
plt.plot(index,predicted)
plt.axvline(df_orig.index[training_samples], c="r")
plt.show()
```



Lab: RNN lab

- **Overview:**

- Implement RNNs in TensorFlow

- **Approximate run time:**

- 30-40 mins

- **Instructions:**

- Try these labs
 - (Instructor to Demo) **RNN-1** - RNN Intro: Sinewave
 - **RNN-2** - Stock price prediction
 - **RNN-3** - Text generation with RNN

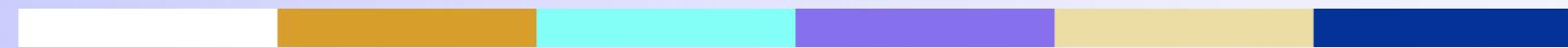


Review and Q&A

- Let's go over what we have covered so far
- Any questions?



LSTMs in TensorFlow



Objectives

- Learn to implement LSTMs in TensorFlow

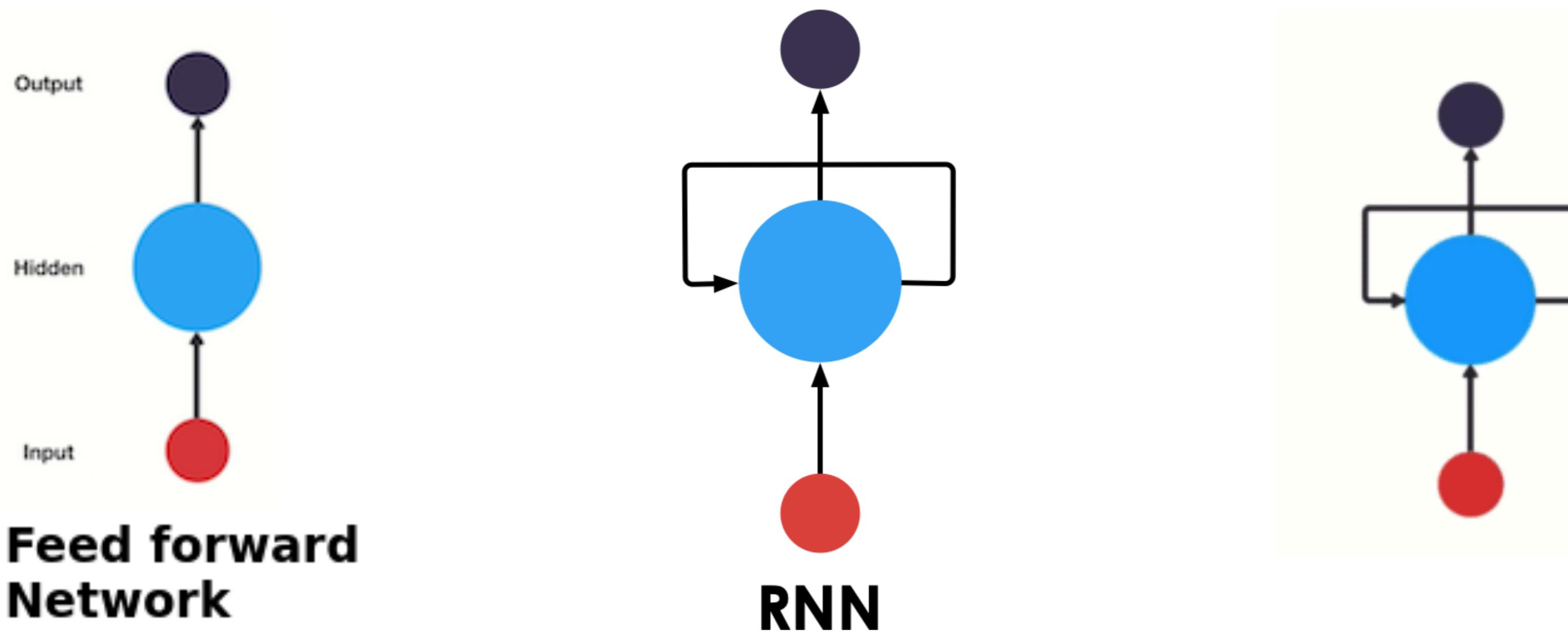
Long Short Term Memory (LSTM) Neural Networks

Lesson Objectives

- Learn about Long Short Term Memory (LSTM) Neural Networks
- Understand how to use LSTM

RNNs - Review

- In Feedforward Networks, data flows one way, it has **no state or memory**
- RNNs have a 'loop back' mechanism to pass the current state to the next iteration



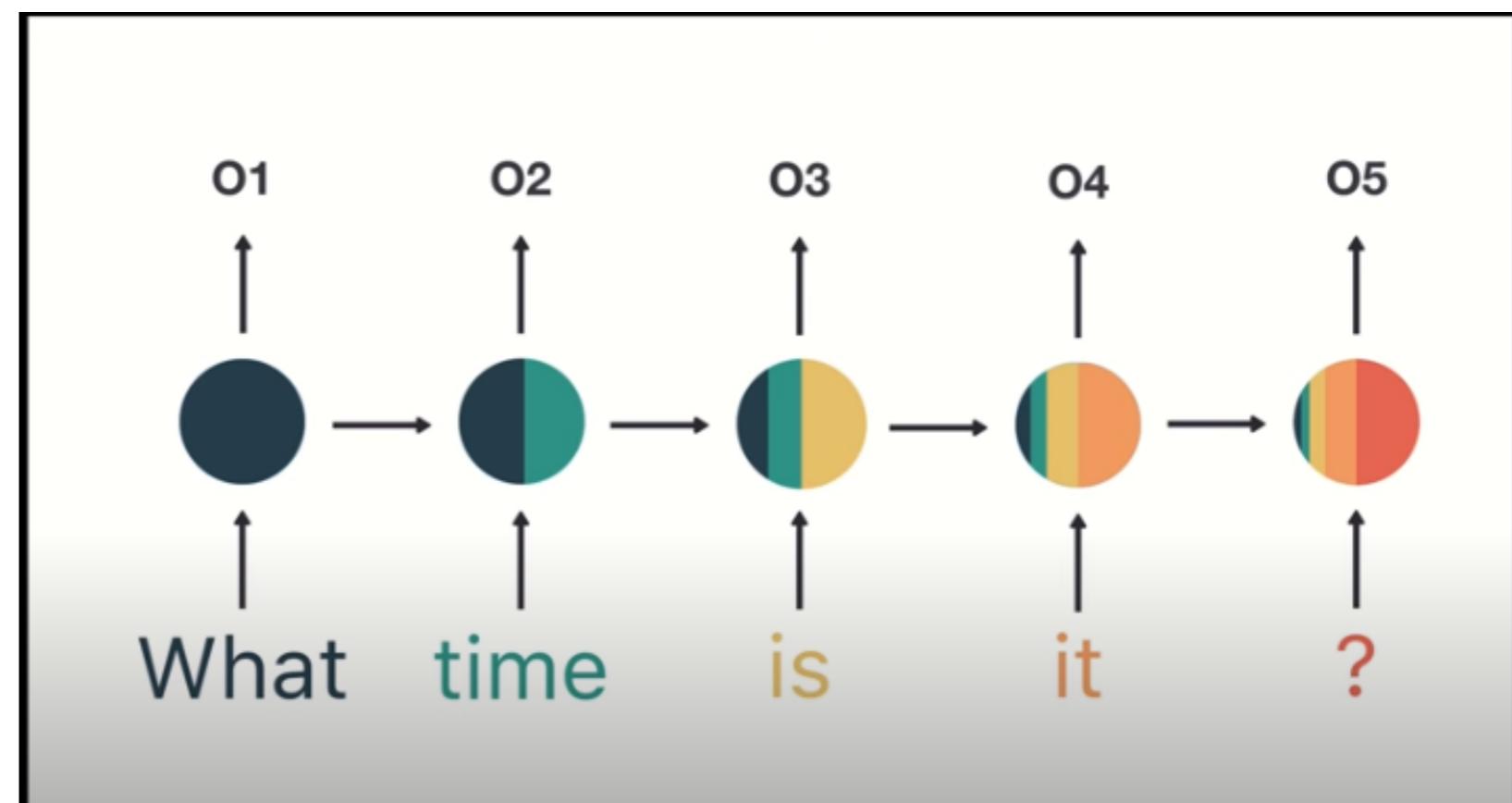
[Animation link](#)

Issues With RNN: Long Training Time

- To train an RNN on long sequences, you will need to run it over many time steps
- making the unrolled RNN a very deep network
- Just like any deep neural network it may suffer from the vanishing/exploding gradients problem (discussed earlier) and take forever to train
- There are solutions (good parameter initialization, non saturating activation functions etc.) for this
- Large amount of inputs (100+) means the training time can still be long

Issues With RNNs: Short Term Memory

- Memory of the first inputs (hidden state) gradually fades away over multiple steps
- In this diagram below, we can see the 'influence' of word **'what'** (**color black**) is diminishing with each step
 - And in the last step word '**?**' the color black is almost non-existing



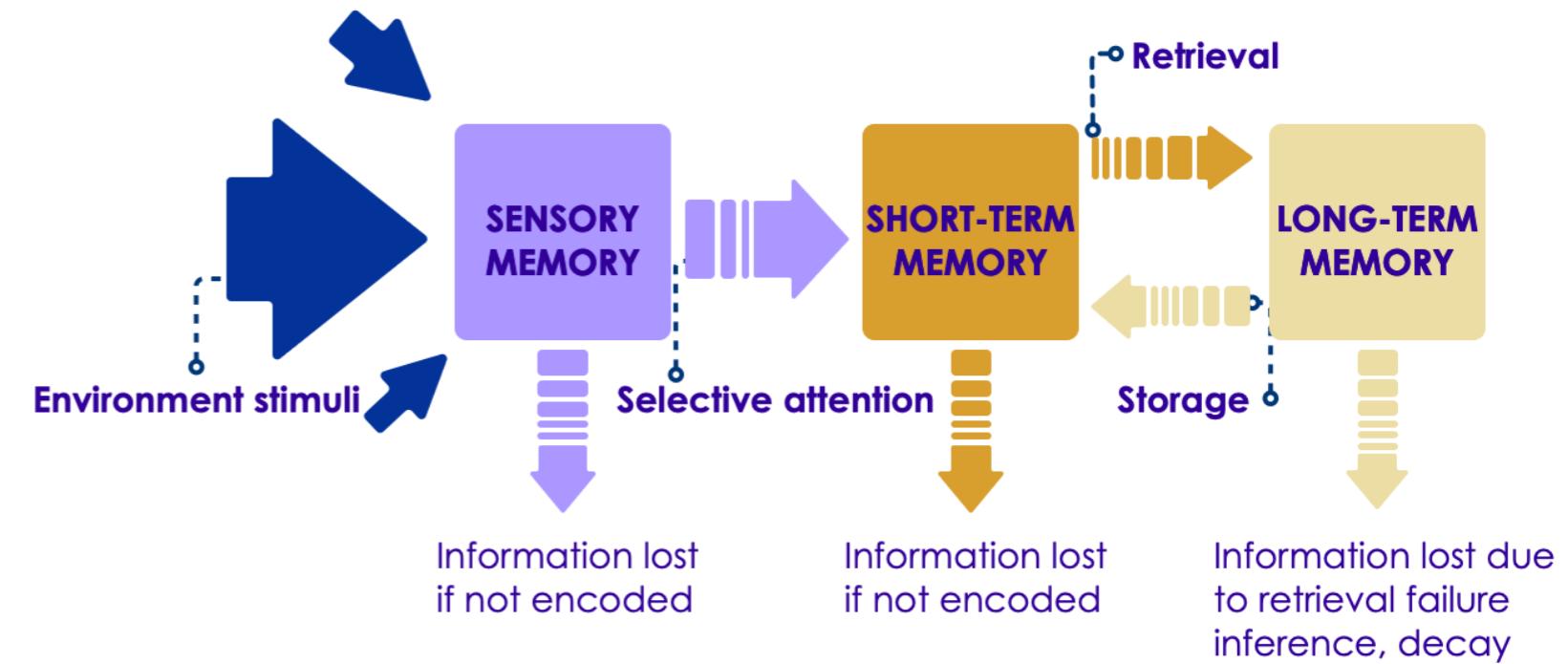
- **RNNs suffer from short term memory**
- What this means is, RNNs can't remember / process 'long sequences' (e.g. long sentences)
- Source

Illustrating Short Term Memory Problem

- Consider the following word completion example
- I lived in China, for most of my teenage years, so I speak fluent _____
- The answer is **Mandarin**
- But which is the key to determining the answer?
 - Not the adjacent words : **my teenage years**
 - But : **lived in China** - from start of the sentence
- So to make the correct prediction, the network has to 'remember' early words (lived in China)
- RNN's short term memory makes it hard to do so

Long-term vs. Short-term Memory

- Human Brains have two types of memory: short and long term
- Short-term memory holds data in the immediate context
 - Around 30 seconds or so
 - After this, data is erased
- Long-term Memory is indefinite
 - Data can be accessed if a path to the memory is found
 - Older data, however, may be "**forgotten**", the path is not found.
- For example, I remember what I had for breakfast this morning; I may not remember what I had for breakfast 2 days ago; But I will certainly remember a nice brunch I had with friends last weekend!

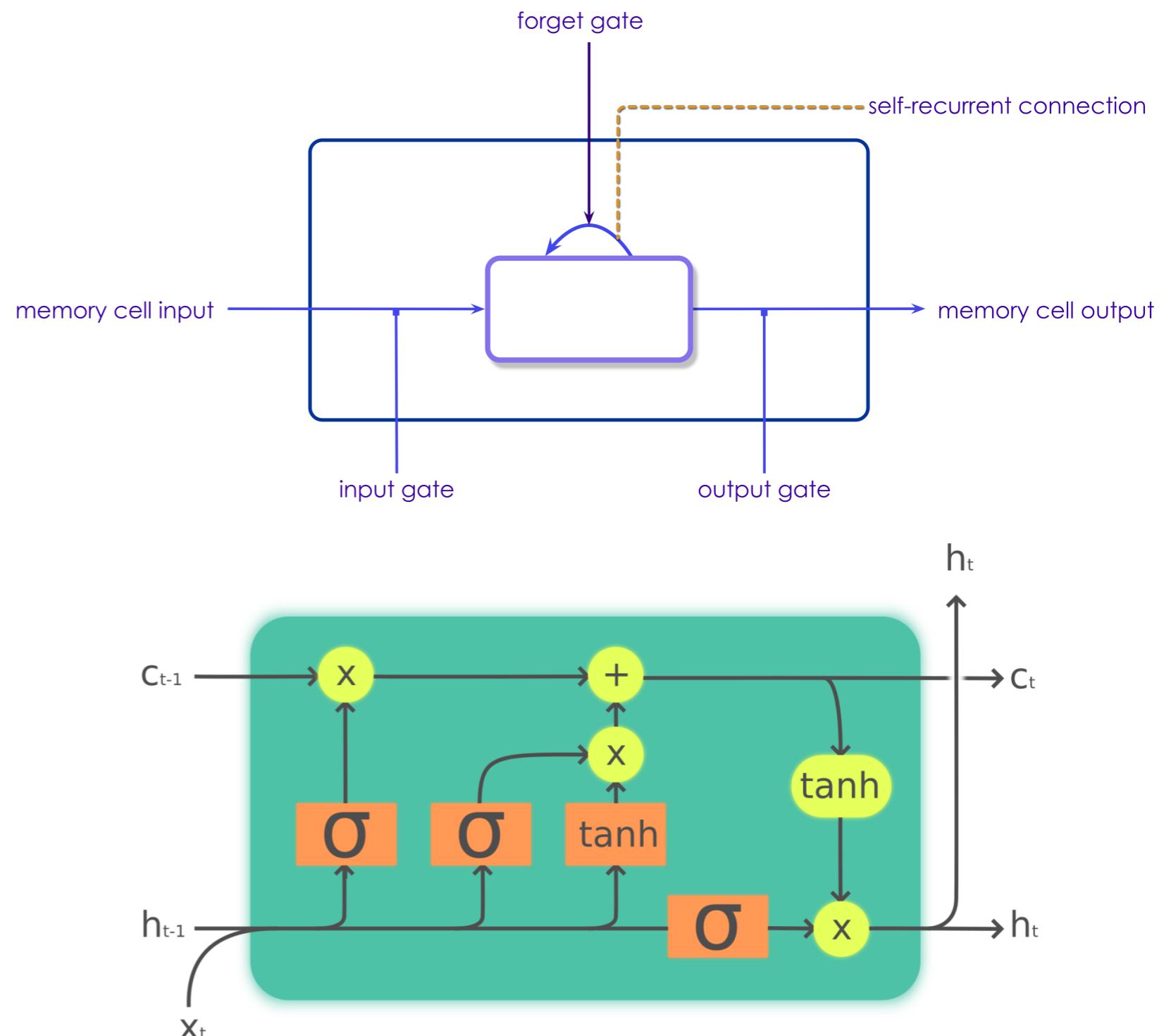


LSTM Networks

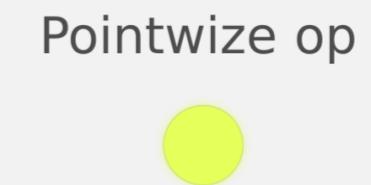
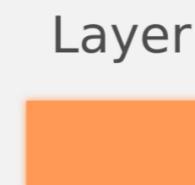
- LSTM networks are the most commonly used variation of Recurrent Neural Networks (RNN)
- LSTM was introduced in 1997 by Sepp Hochreiter and Jürgen Schmidhuber ([paper](#))
- It was gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak,⁴ Wojciech Zaremba
- It is designed to mimic selective memory and forgetfulness

LSTM Design

- Input Gate
- Output Gate
- Forget Gate
- Self-Recurrent Connection (memory)
- (See next slides for details)



Legend:



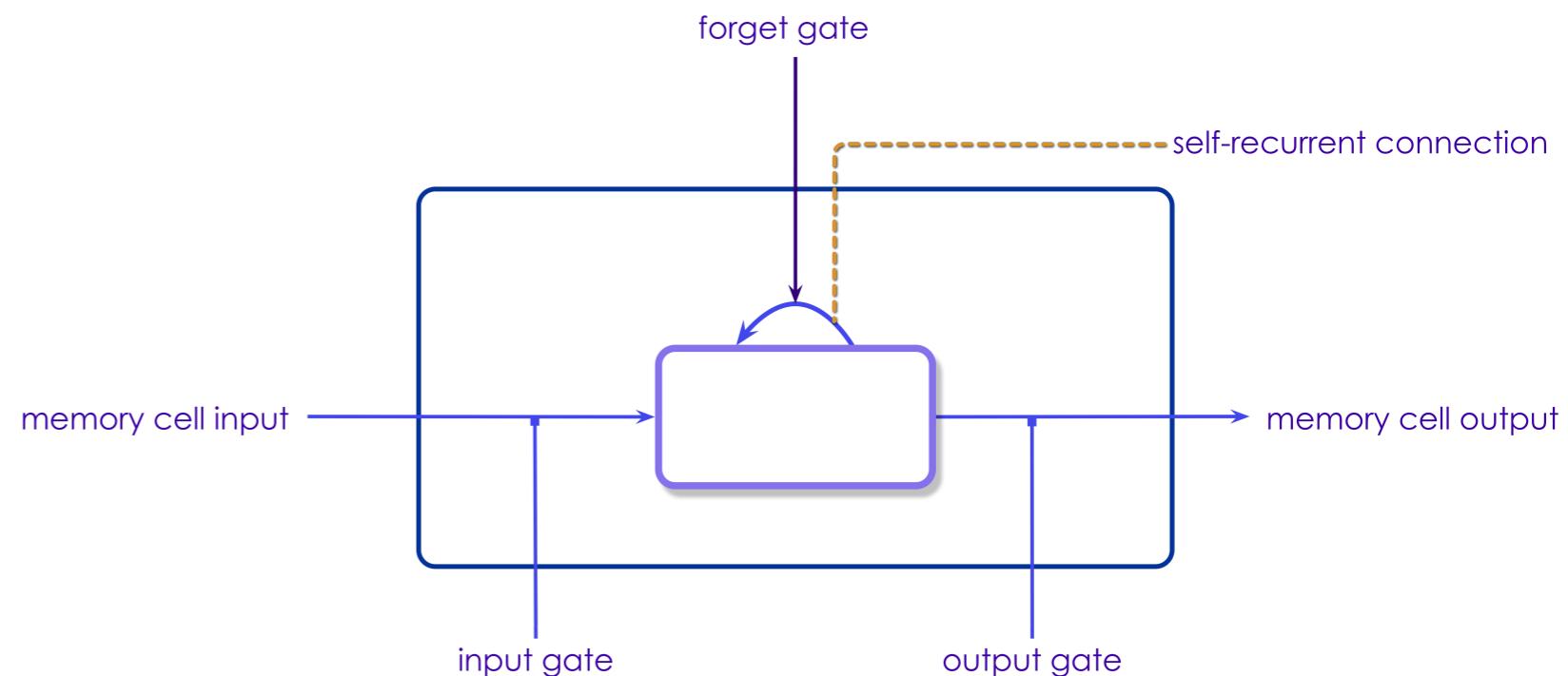
Components of the LSTM

▪ Input Gate

- Input Gate contains new information / input
- Input can be user input or output of last LSTM layer

▪ Output Gate

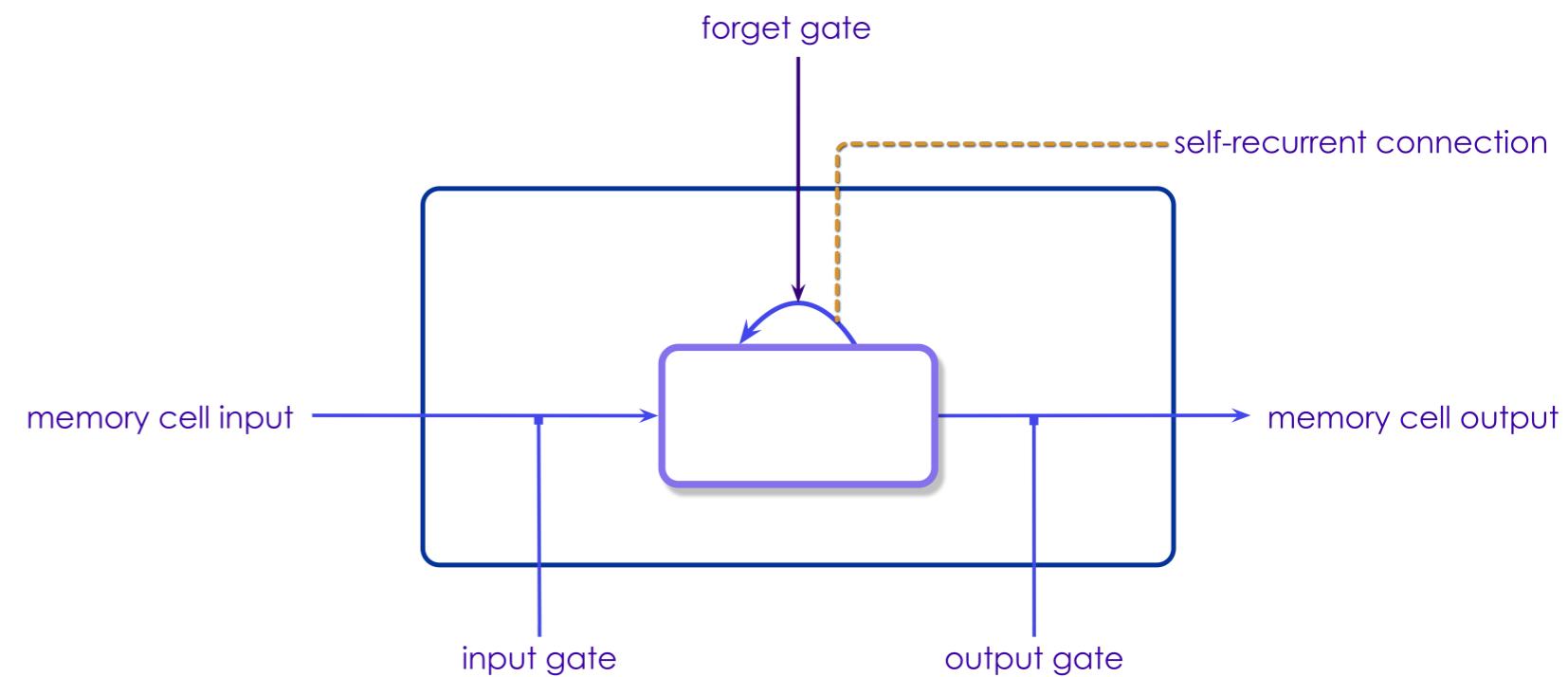
- Sent to next layer (another LSTM or Dense layer)
- The memory cell value can be read as well.



Components of the LSTM

▪ Forget Gate

- The forget gate is an example of *negative feedback*
- It tends to reduce the value of the neuron
- Creates stability, prevents vanishing gradient

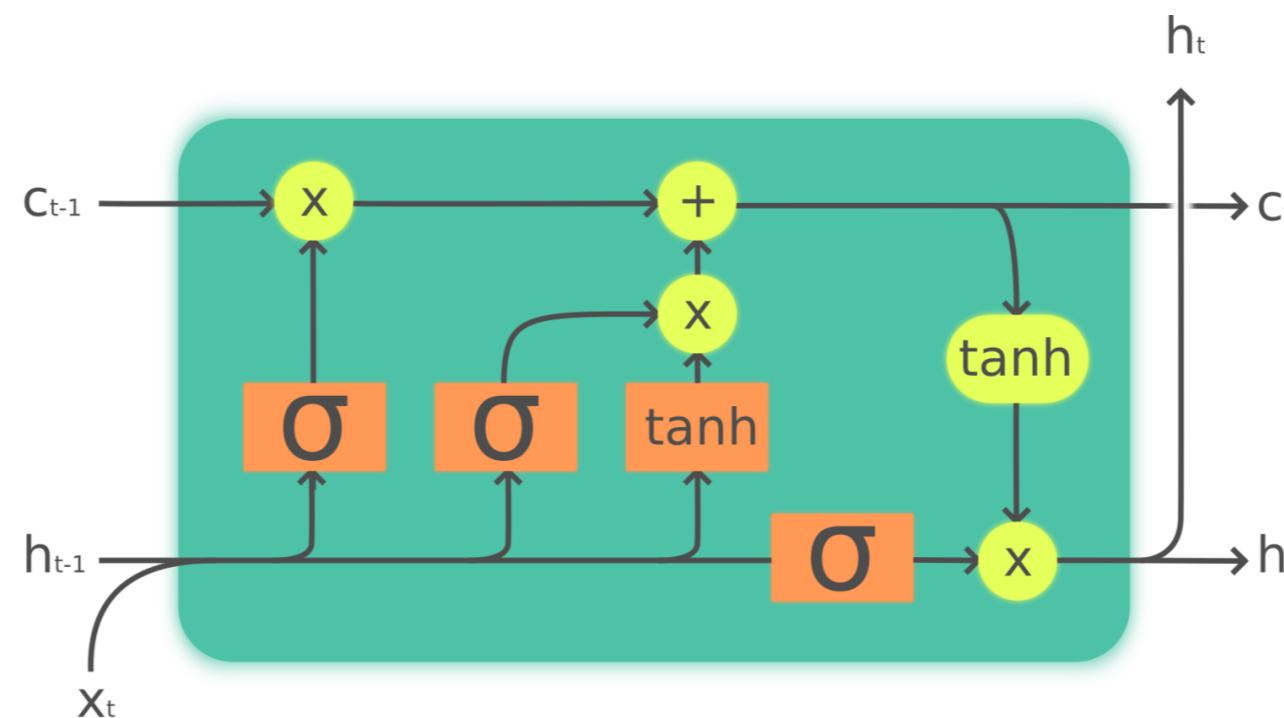
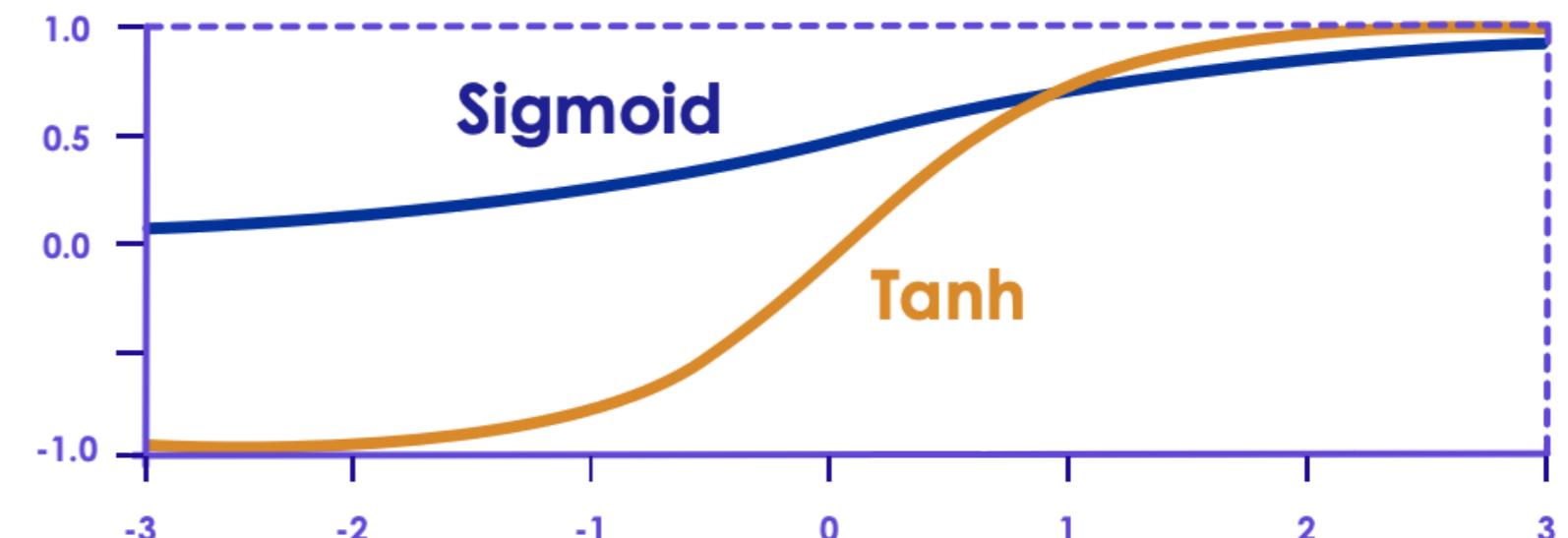


▪ Self-Recurrent Connection (memory)

- Self Recurrence is the *memory* part of LSTM
- It means that the current value will be stored
- The forget gate will cause the current memory to *decay*
- Unless reinforced by the input gate.

Activation Functions in LSTM

- Generally **sigmoid** or **tanh** activations are used (rather than ReLU as in CNN)
- Default activation is tanh.
- Previous State is applied with sigmoid (σ)



Legend:

Layer	Pointwize op	Copy

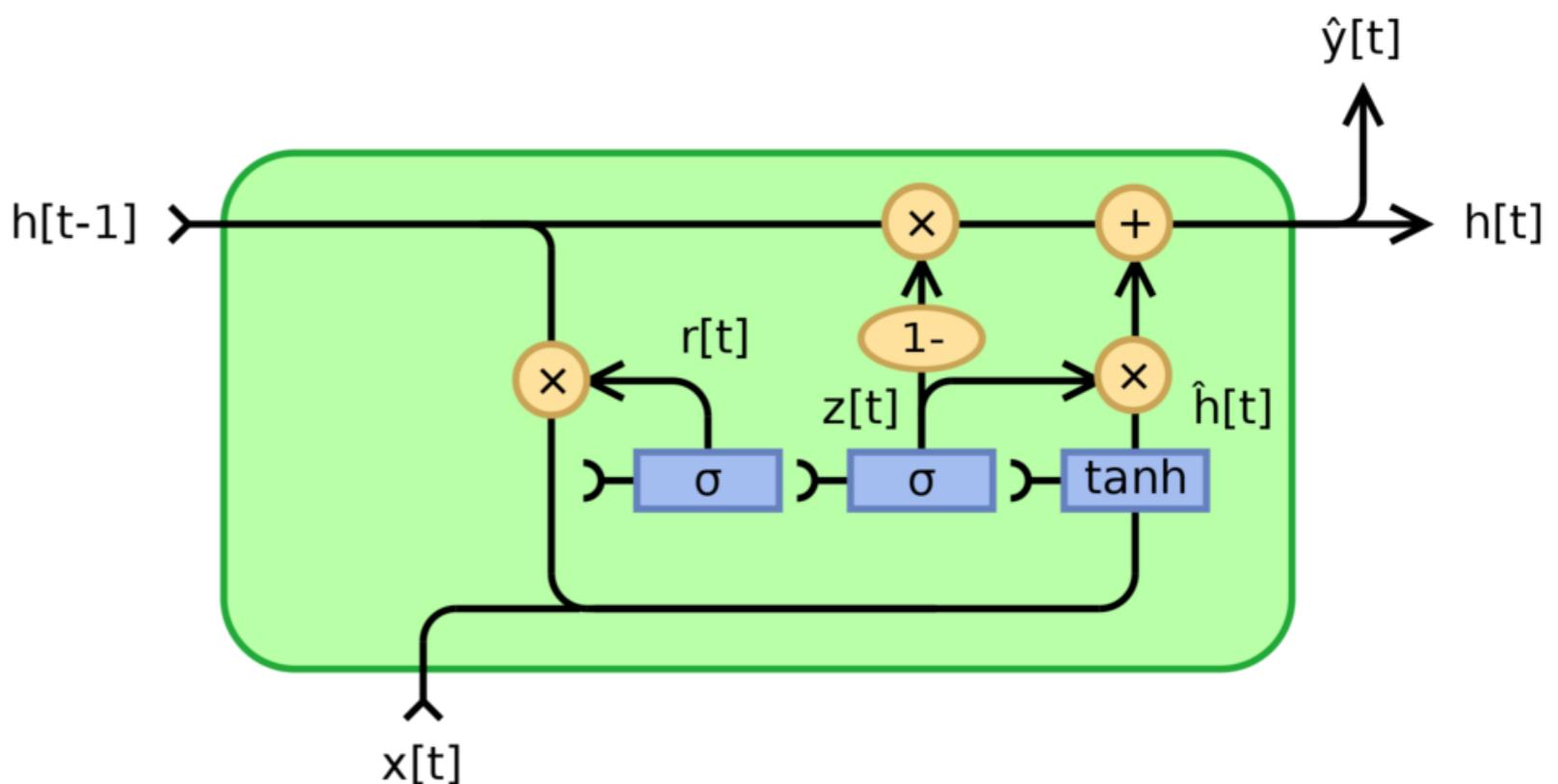
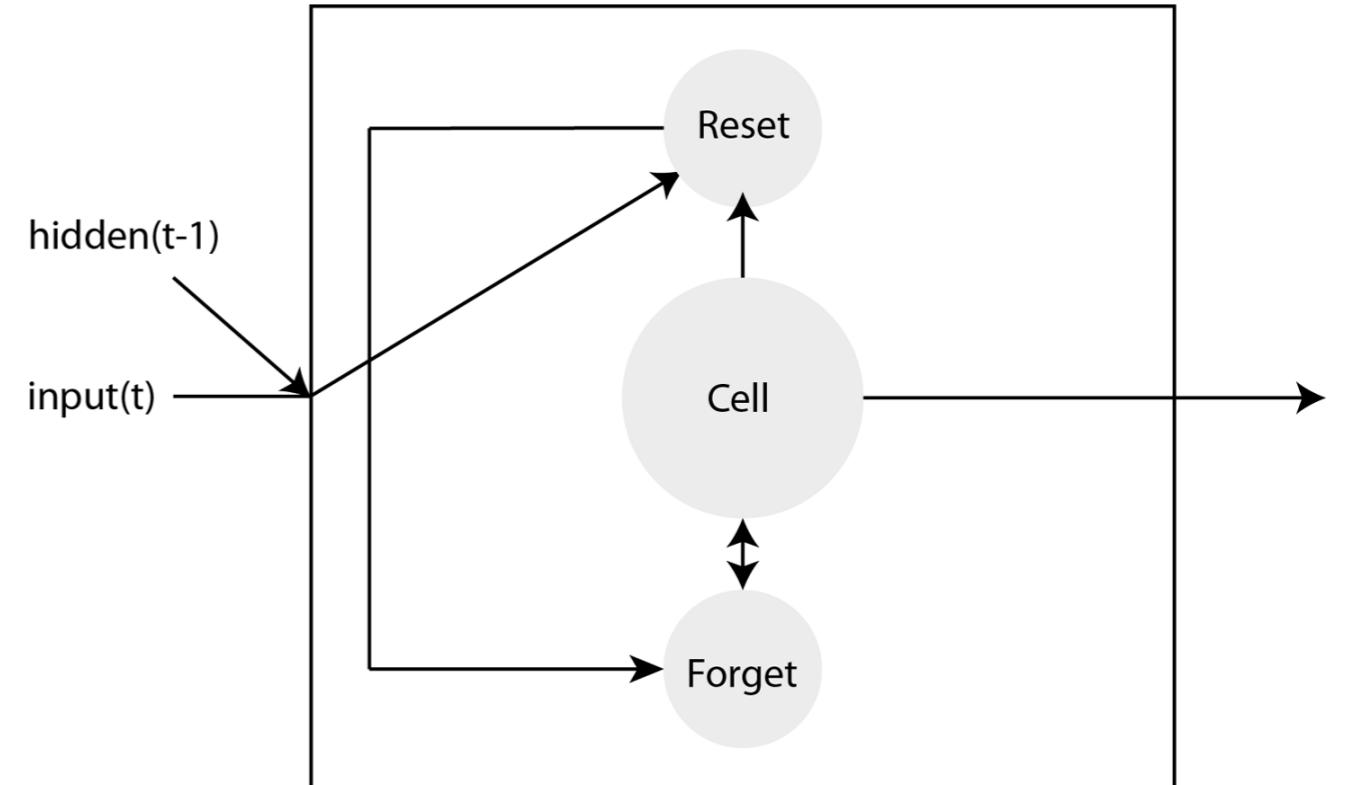
LSTM Variants

Improvements on LSTM

- LSTM proved to be very popular architecture
- It solved few critical issues we had with RNNs
 - Not susceptible to vanishing / exploding gradients
 - Can remember longer sequences
- But LSTMs are also more complex
 - They take longer to train
 - They take up more resources during runtime (prediction)
- Since their introductions in 1997, many variants are created

Gated Recurrent Units (GRU)

- Introduced by Kyungyun Cho (et al) in 2004 (paper)
- GRU has merged the forget gate with the output gate
- This means that GRU has fewer parameters than an LSTM
 - Quicker to train and uses fewer resources at runtime
 - Can be trained with less training data

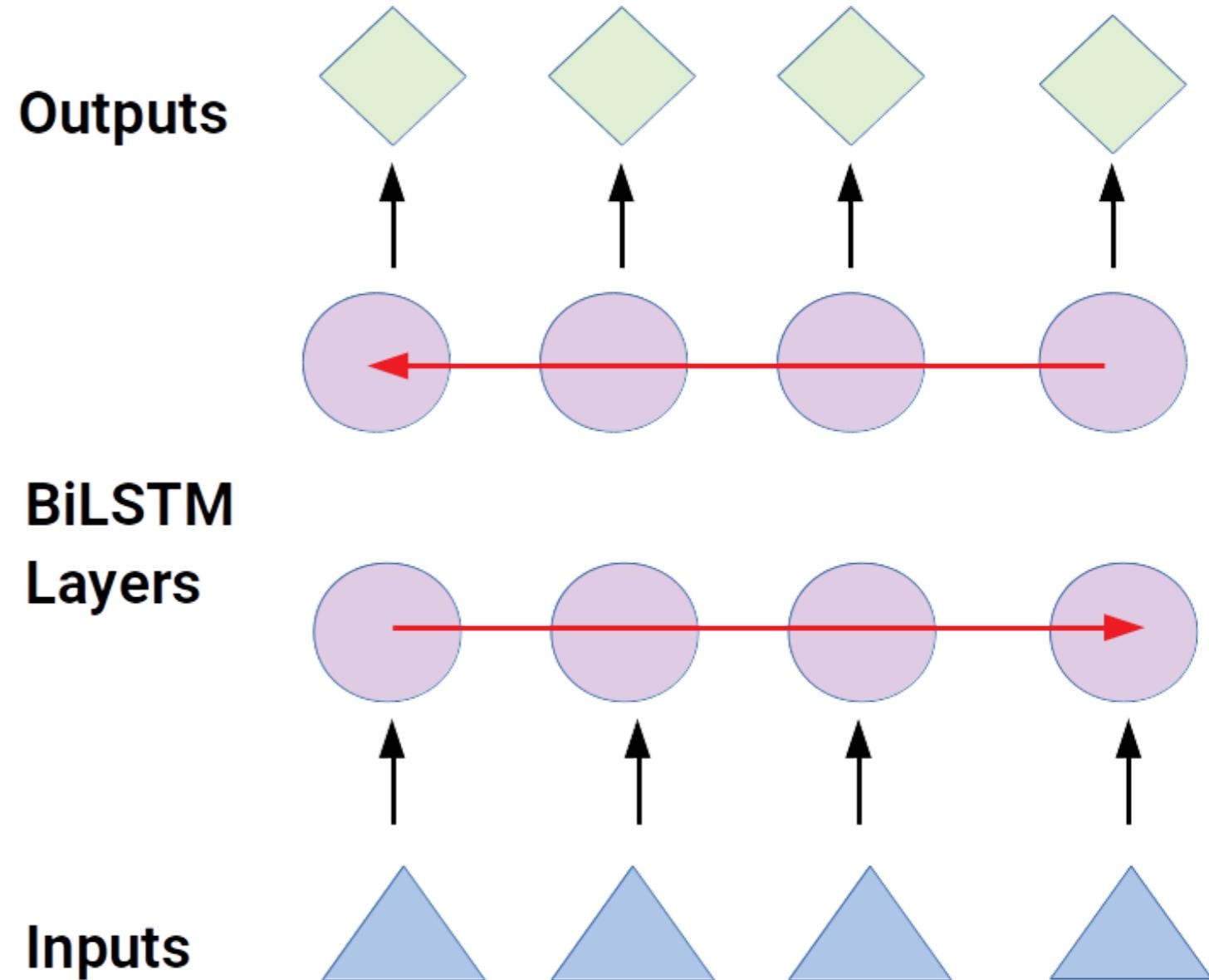


Limitations of GRUs

- GRUs are less powerful than LSTMs because of the merging of the forget and output gates
- GRUs help fix the vanishing gradient problem by "gating" the "hidden state"
- However, GRUs cannot differentiate between short-term and long-term memory.
- They have *one* hidden state.
- This makes them perfect for shorter sequences.
- Performance wise : RNN < GRU < LSTM
- Speed : RNN > GRU > LSTM

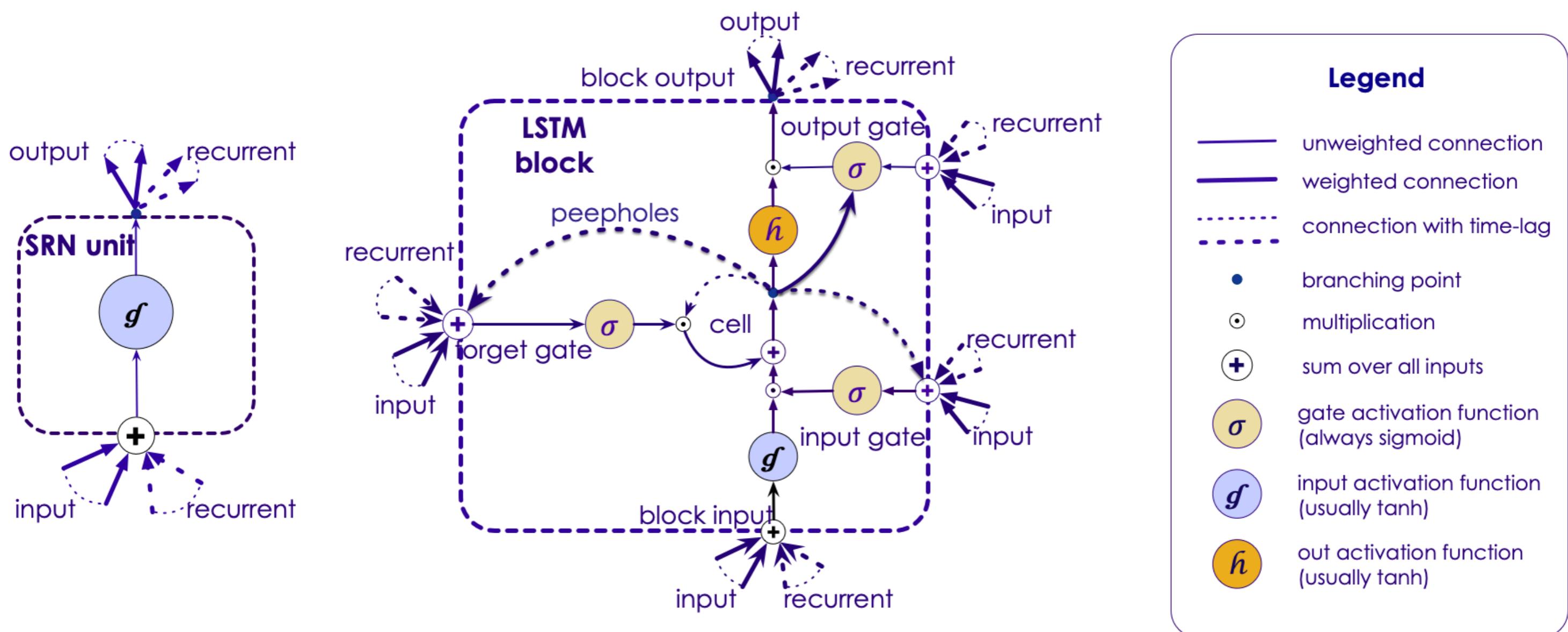
biLSTM (Bidirectional LSTM)

- LSTMs/RNNs look to past data to make decisions as they train
- Sometimes we need to 'look ahead' to make a decision
 - This is specially true for natural language processing (NLP)
- **biLSTM** is essentially two LSTMs stacked
 - Input is sent in one direction in one layer
 - and reversed in other layer
- Example of look ahead
 - "I am learning ____"
 - "I am learning ____ for my trip to Mexico"



Peephole LSTM

- One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding "peephole connections" (Paper)
- Allow the current *state* of the cell to be considered at the input
- Otherwise current state cannot be directly compared with the gated input

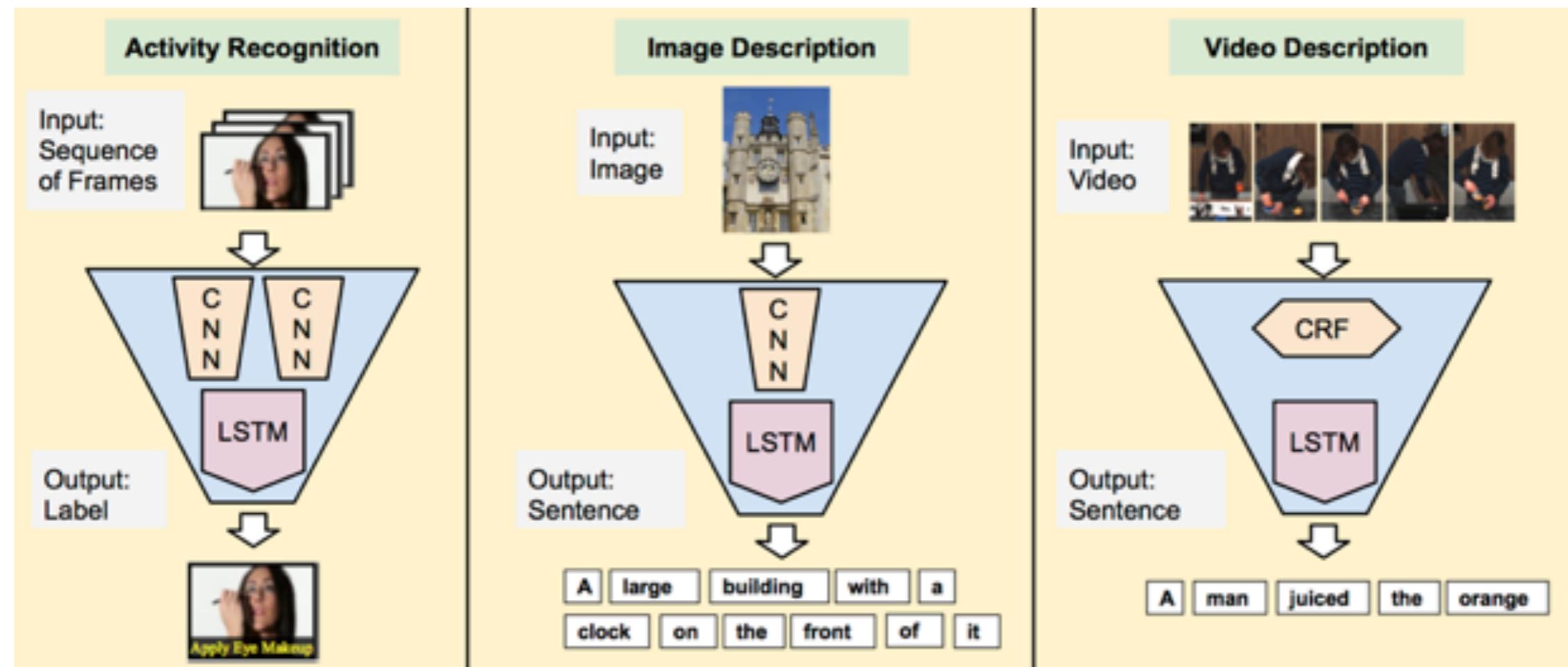


LSTM Applications

LSTM Applications

- LSTMs are good at dealing with **sequence data**
- Natural Language Processing (NLP) tasks
- Generating sentences (e.g., character-level language models)
- Classifying time-series
- Speech recognition
- Handwriting recognition

Image / Video Captioning



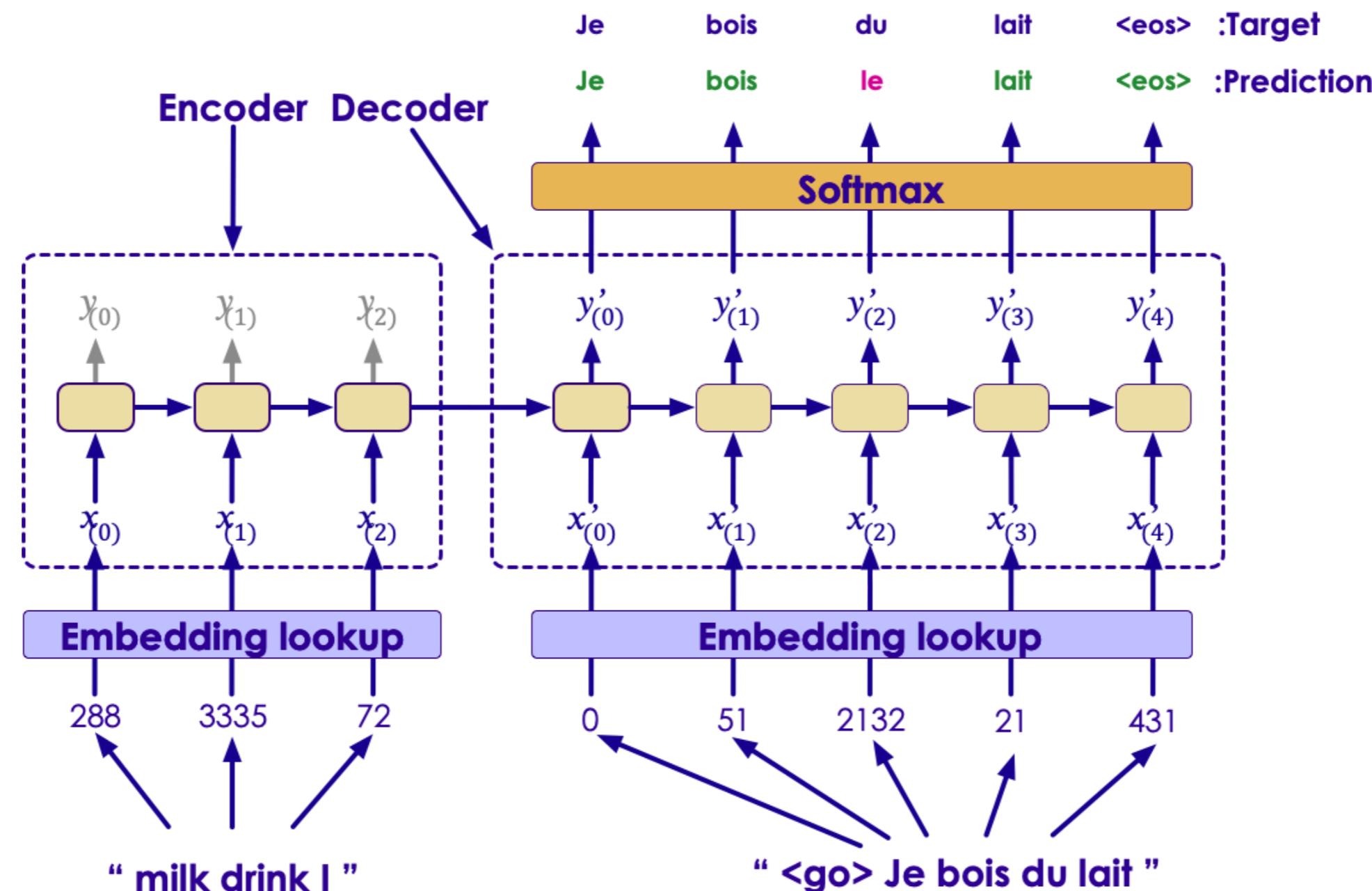
- LM2Text Paper

LSTMs and Natural Language

- NLP is also treated as a sequence.
- Each word is treated as a item in sequence.
- This is far more efficient than the massive sparse vector arrays.
- This allows us to move beyond the "bag of words" approach
- LSTMs are a great way to look at *semantic* models.
- Example:
 - "The cat sat on the mat; then she climbed on the table; then jumped down"
 - where is the cat now?

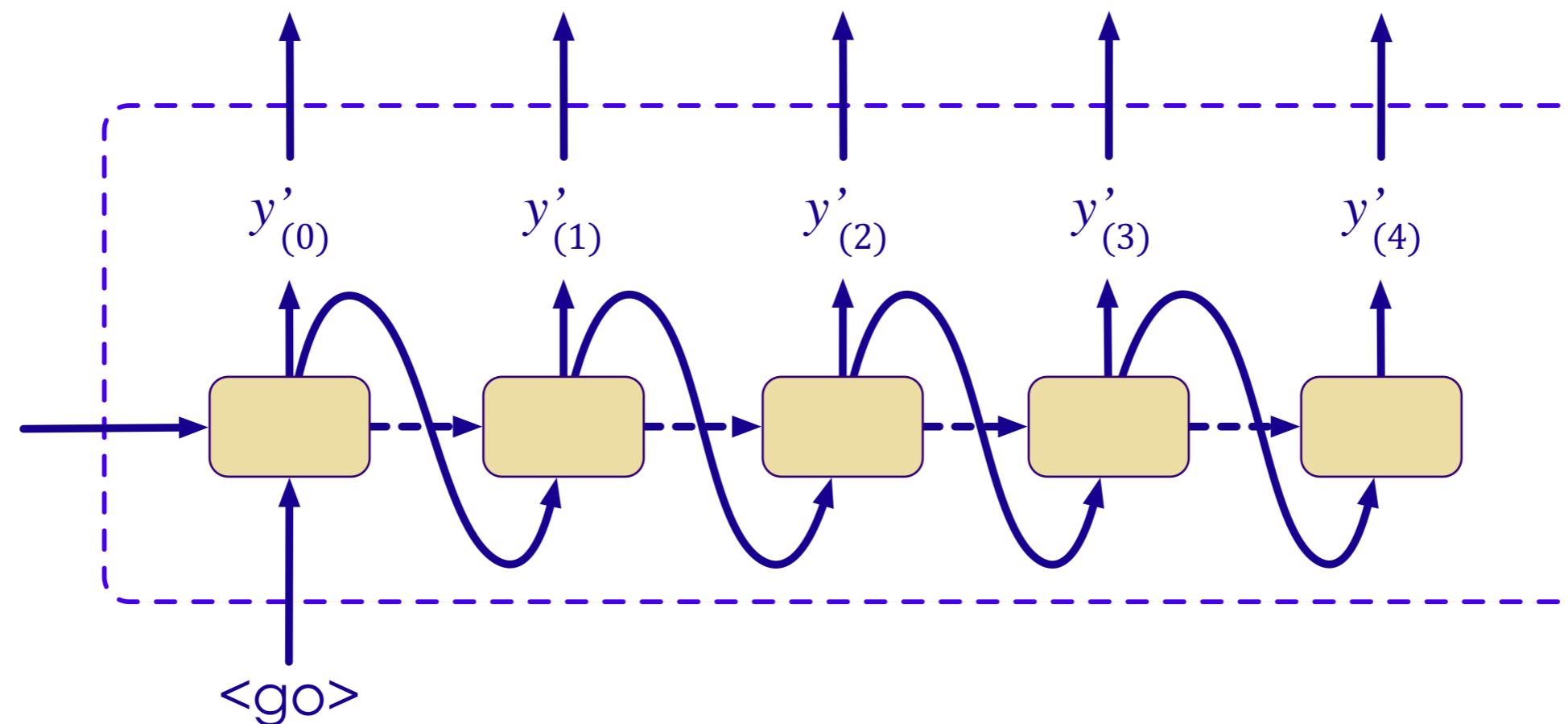
Machine Translation Model

- Machine translation model is essentially a deep recurrent neural network
- The following example shows how this is done.



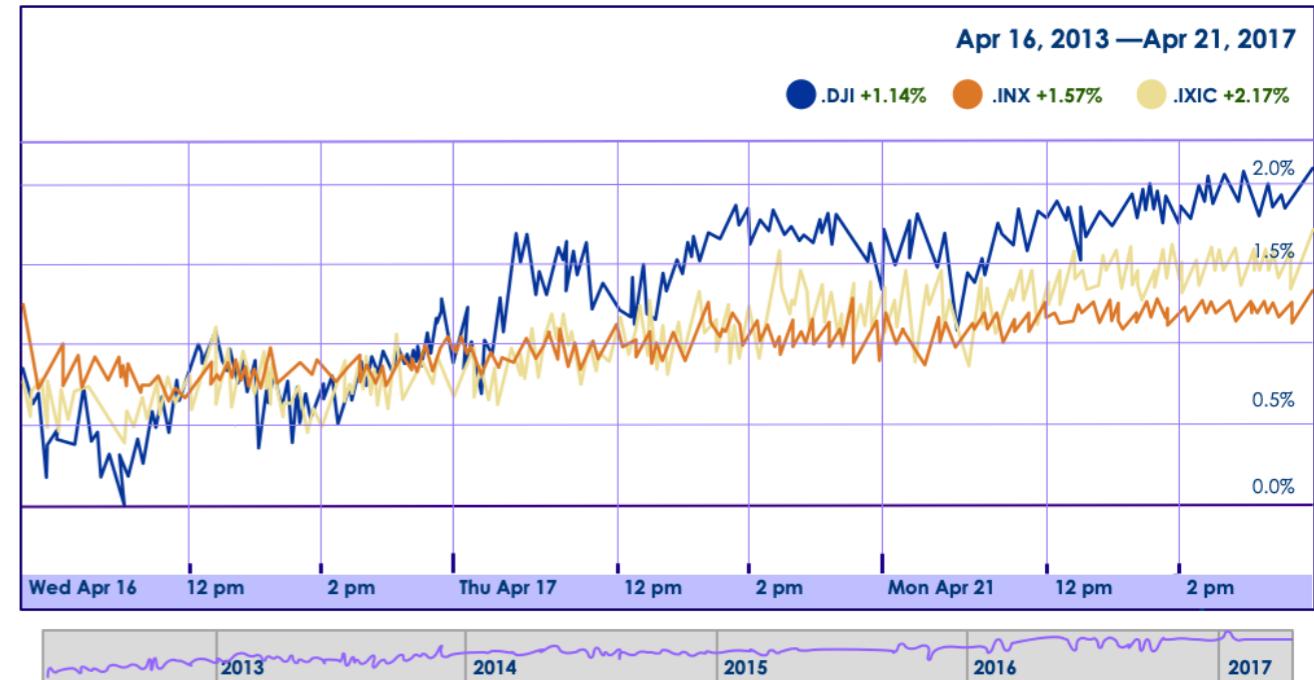
Encoding an Output at Prediction Time

- The previous time step is fed in at the left.
- The next word in sequence is fed from the bottom, for example "go".



Predicting Time Series data

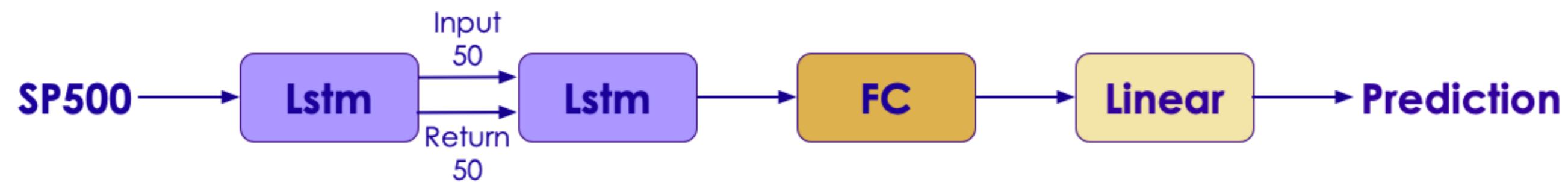
- How do we predict time-series data?
- We can view the SP500 index a series of timestamps and data
 - 2018-01-01 5:00pm 1200
 - 2018-01-02 5:00pm 1210
- So what's the *next* day of S&P500 data?
 - and the next, and so on?
- Is this a Classification problem or Regression?



SP500 Example

- We are going to use a network something like this:

- Input Layer (1 input)
- 2 LSTM Layers
- Fully Connected Layer
- Output Layer with Linear Activation (1 output)

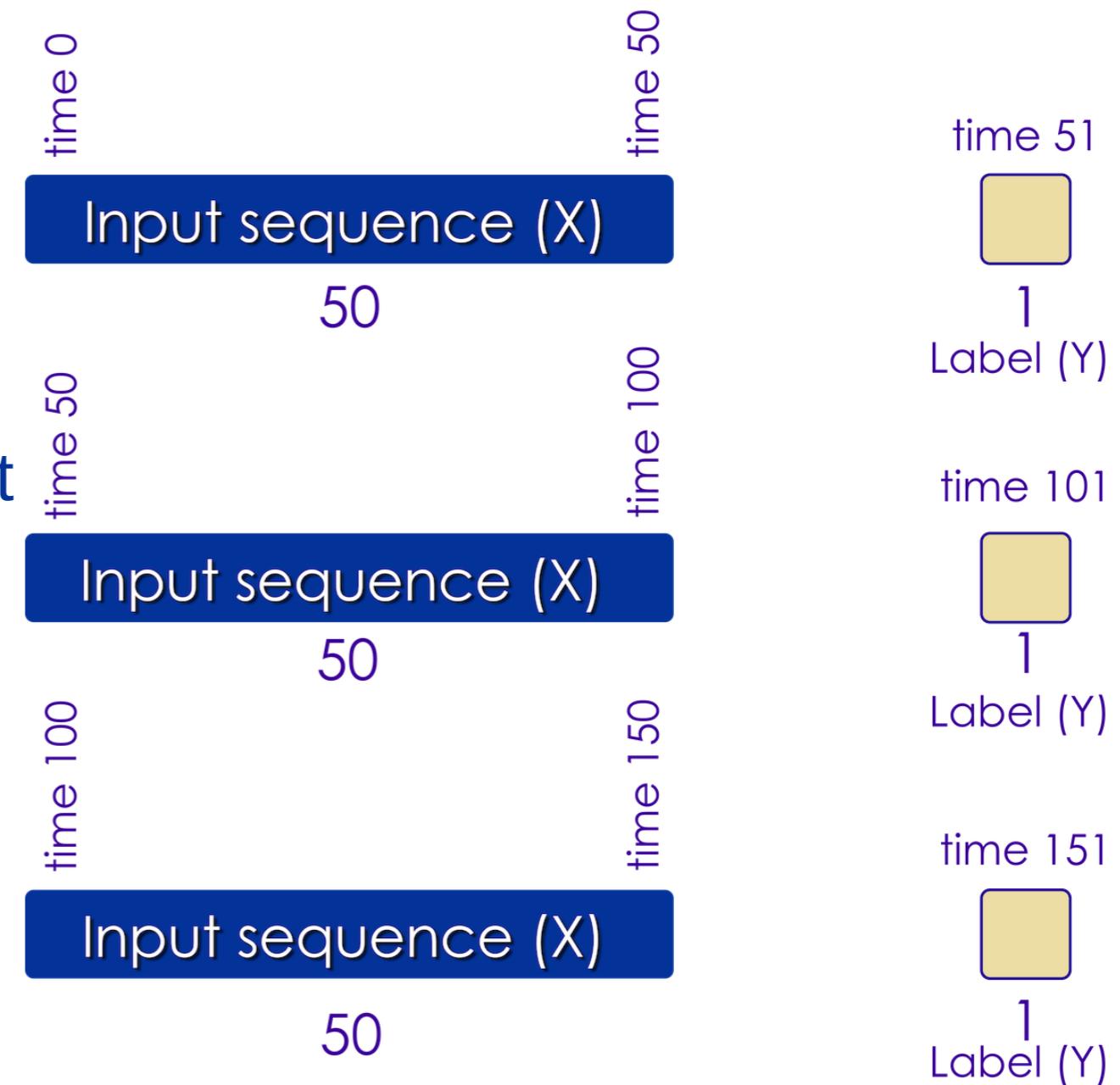


Memory Cells

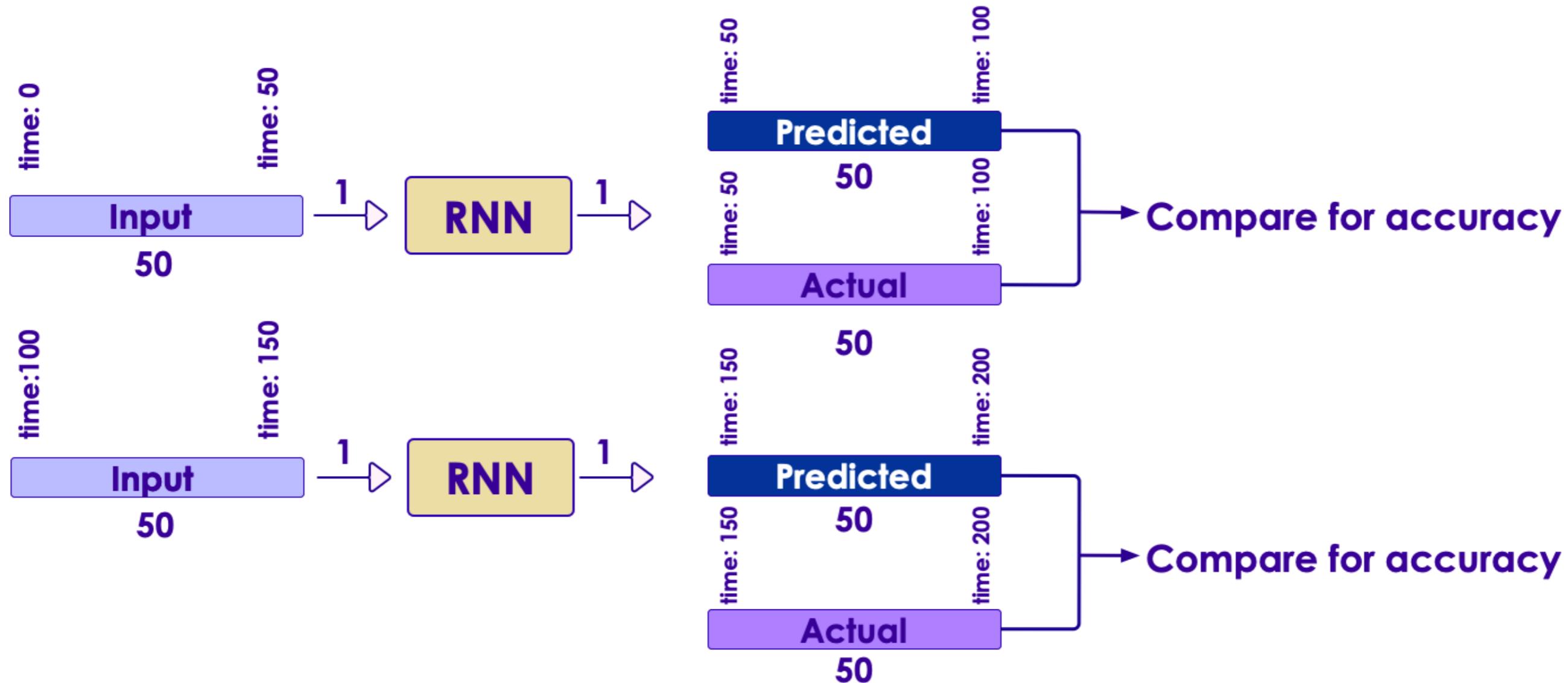
- We have 2 LSTM Layers:
 - One has 50 cells
 - The next has 100 cells
- How long a sequence can we *remember*?
 - The network is limited in how far into the future it can predict.
 - Sequences of 50 are good.
- We will *train* with sequences of 50
- Then we will predict with sequences of 50
 - and predict the next 50 values after.

Training Sequences

- We are training with sequences of 50 prices of SP500.
 - 50 prices
 - No need for timestamps
- We then have a label of size 1 which is the 51st price in the sequence.
- We then make a new training sequence.



Feeding the Sequence

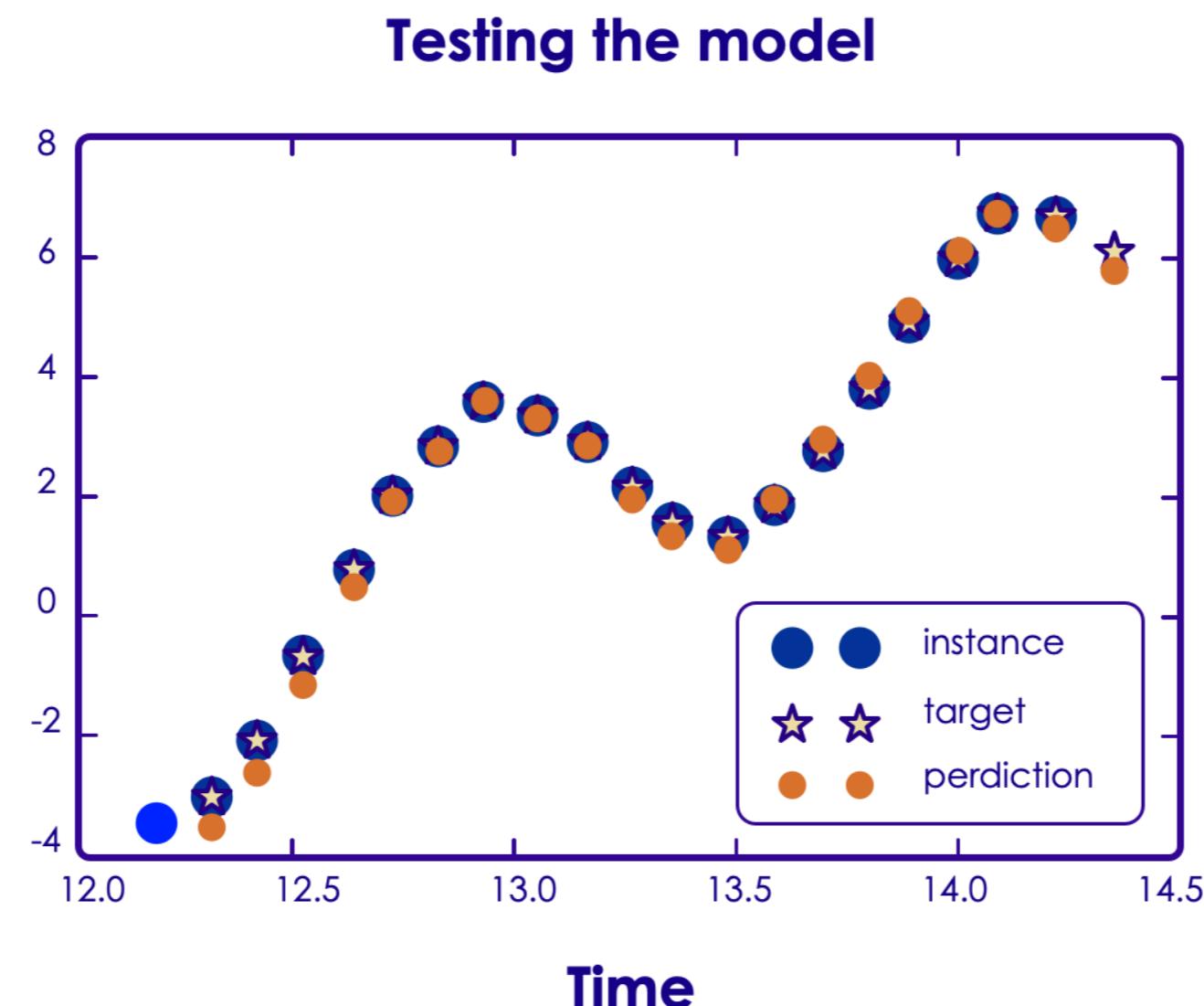


Prediction

- Each Sequence is "fed" sequentially into the neural net (only 1 input)
- Each LSTM cell is connected sequentially with the next in a given layer
 - Layer 1: 50 cells
 - Layer 2: 100 cells
- We also predict with sequences of 50
- We sequentially go through with 50 prices, then predict another 50.
- We then compare the 50 predicted with the actual data.

Testing the Model

- We evaluate the model with the prediction
- Comparing predicted sequences with actual helps us to evaluate the model
- We can't just see if predicted and actual are the same.
- We have to take a window in time.



Our Results

- Does our model predict the future?
 - When the market crashes, does our model predict a downward movement?
 - When the market spikes, does our model predict an upward movement?
- The model is more "general"
 - Is the market tracking upward?
 - Is it tracking downward?
 - Is it flat?
- Warning: Don't try this at home with real money!
 - Real financial models will be far more complex.



LSTM Takeaways

- LSTMs are very sequential
- Limits parallelization opportunities
- Also LSTMs are resource intensive
 - Take longer to train
 - Need more training data
 - And take up more resources at runtime
- CNN variations may be more effective.
- Only use LSTMs when you *must*.

Further Reading

- Good intro to LSTM
- "Neural Networks and Deep Learning"

LSTMs in TensorFlow

LSTMs in TensorFlow

- LSTMs are implemented in `tf.keras.layers.LSTM`

```
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential()

## Add LSTM layer
model.add (tf.keras.layers.LSTM(units=4))
```

- Using **bidirectional LSTM**

```
model.add (tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)) )
```

LSTMs in TensorFlow

- We can use multiple LSTM layers
 - Note LSTM layers except the the last one will have **return_sequences=True**

```
model = tf.keras.Sequential([  
  
    tf.keras.layers.LSTM(64, return_sequences=True),  
    tf.keras.layers.LSTM(64),  
])
```

- Using **GRU** Units

```
model = tf.keras.Sequential([  
  
    tf.keras.layers.GRU(64),  
])
```

LSTM Walkthrough - IMDB Movie Reviews

- We will use a LSTM to predict sentiment of IMDB movie reviews
- **Instructor:** You may walk through the example here or use the lab **LSTM-2-IMDB-sentiment**
- Reference

Step 1 - Grab Data

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

train_data, test_data = imdb['train'], imdb['test']

print ("train_data: ", len(train_data))
# > train_data: 25000

print ("test_data: ", len(test_data))
# > test_data: 25000
```

Step 2 - Shaping Data

```
import numpy as np

training_sentences = []
training_labels = []

testing_sentences = []
testing_labels = []

# str(s.tonumpy()) is needed in Python3 instead of just s.numpy()
for s,l in train_data:
    training_sentences.append(s.numpy().decode('utf8'))
    training_labels.append(l.numpy())

for s,l in test_data:
    testing_sentences.append(s.numpy().decode('utf8'))
    testing_labels.append(l.numpy())

training_labels_final = np.array(training_labels)
testing_labels_final = np.array(testing_labels)
```

Step 3 - Explore Data

```
import random

index = random.randint(0, len(training_sentences)-1)

print ('training_labels_final[{}]\n{}'.format(index, training_labels_final[index]))
print()
print ('training_sentences[{}]\n{}'.format(index, training_sentences[index]))
```

```
training_labels_final[15868]
```

```
0
```

```
training_sentences[15868]
```

(A possible spoiler or two)

 "Soul Survivors" is quite possibly the worst theatrical released movie ever. Nothing makes sense at all, there's some plot about a girl who has strange visions of people who may or may not be dead. The entire movie is just a bunch of random shots of things that don't really tie together, by the end of the film.

 The acting is non-existent, the camera work is jerky and the script is so confusing, it just makes the movie even harder to watch.

 I kept waiting for something to tie the movie together but nothing came. Definitely the worst film of the year. -****1/2 stars.

Step 4 - Vectorize Text

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

VOCAB_SIZE = 10000 # consider top-N words
OOV_TOK = "< OOV >"
EMBEDDING_DIM = 32
MAX_SEQ_LENGTH = 120
TRUNC_TYPE='post'
PADDING_TYPE='post'

tokenizer = Tokenizer (num_words=VOCAB_SIZE, oov_token=OOV_TOK)
tokenizer.fit_on_texts(training_sentences)

training_sequences = tokenizer.texts_to_sequences(training_sentences)
max_seq_length_actual = max([len(x) for x in training_sequences])

training_sequences_padded = pad_sequences(training_sequences, maxlen=MAX_SEQ_LENGTH,
                                         truncating=TRUNC_TYPE)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_sequences_padded = pad_sequences(testing_sequences, maxlen=MAX_SEQ_LENGTH)
```

Step 5 - Text Vectors

```
index = random.randint(0, len(training_sentences)-1)

print ('training sentence [{}]\n{}'.format(index, training_sentences[index]))
print ('training seq [{}]\n{}'.format(index, training_sequences[index]))
print ('training padded [{}]\n{}'.format(index, training_sequences_padded[index]))
```

training sentence [14467]

Interesting mix of comments that it would be hard to add anything constructive to. However, i'll try. This was a very good action film with some great set pieces. You'll note I specified the genre. I didn't snipe about the lack of characterisation, and I didn't berate the acting. Enjoy if for what it is people, a well above average action film. I could go on but I've made my comment.

training seq [14467]

```
[219, 1492, 5, 794, 13, 10, 60, 28, 252, 6, 761, 230, 1, 6, 188, 635, 351, 12, 14, 4, 53, 50, 204,
20, 17, 47, 85, 268, 1326, 488, 852, 11, 1, 2, 510, 11, 159, 1, 42, 2, 581, 5, 7370, 3, 11, 159, 1,
2, 114, 356, 45, 16, 49, 10, 7, 82, 4, 71, 750, 857, 204, 20, 11, 98, 138, 21, 19, 205, 91, 59, 927]
```

training padded [14467]

```
[ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   219 1492   5   794   13   10   60   28   252   6   761   230   1   6   188
635 351   12  14   4   53  50   204   20   17   47   85   268 1326   488   852   11   1   2   510  11
159   1   42   2   581   5 7370   3   11  159   1   2   114  356   45   16   49   10   7   82   4
 71   750  857  204   20   11  98   138   21   19  205   91   59   927]
```

Step 6 - Build a Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(VOCAB_SIZE, EMBEDDING_DIM, input_length=MAX_SEQ_LENGTH),
    tf.keras.layers.LSTM(64),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

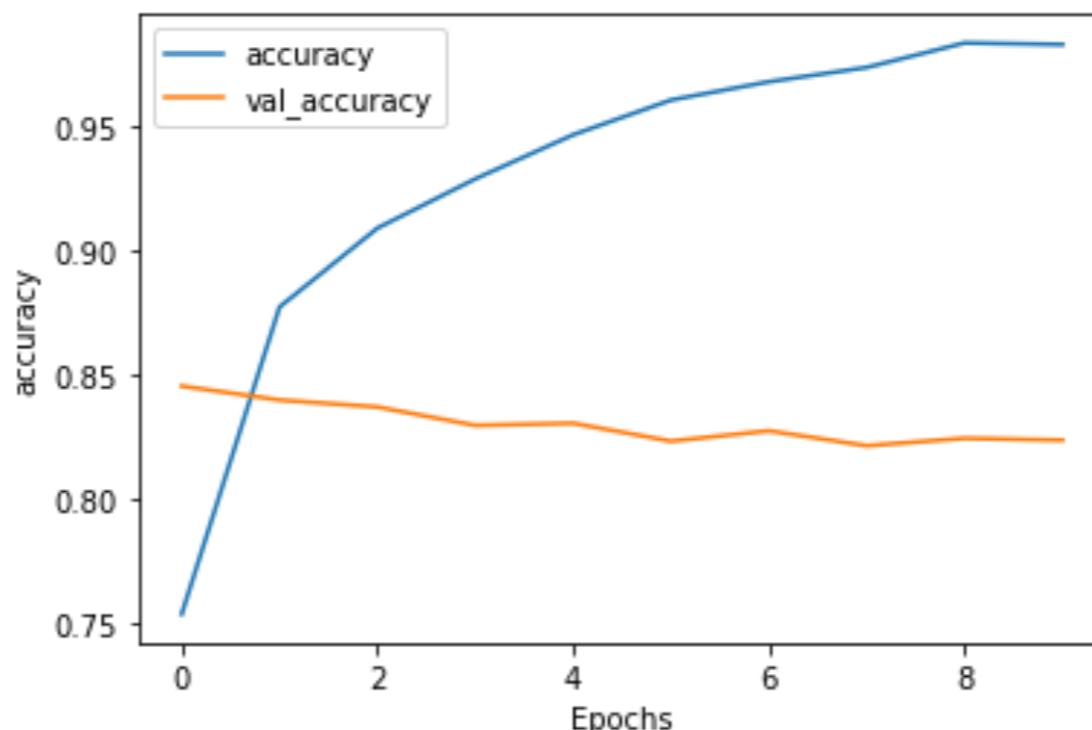
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
embedding (Embedding)	(None, 120, 32)	320000
lstm (LSTM)	(None, 64)	24832
dense (Dense)	(None, 64)	4160
dense_1 (Dense)	(None, 1)	65
=====		
Total params:	349,057	
Trainable params:	349,057	
Non-trainable params:	0	

Step 7 - Training

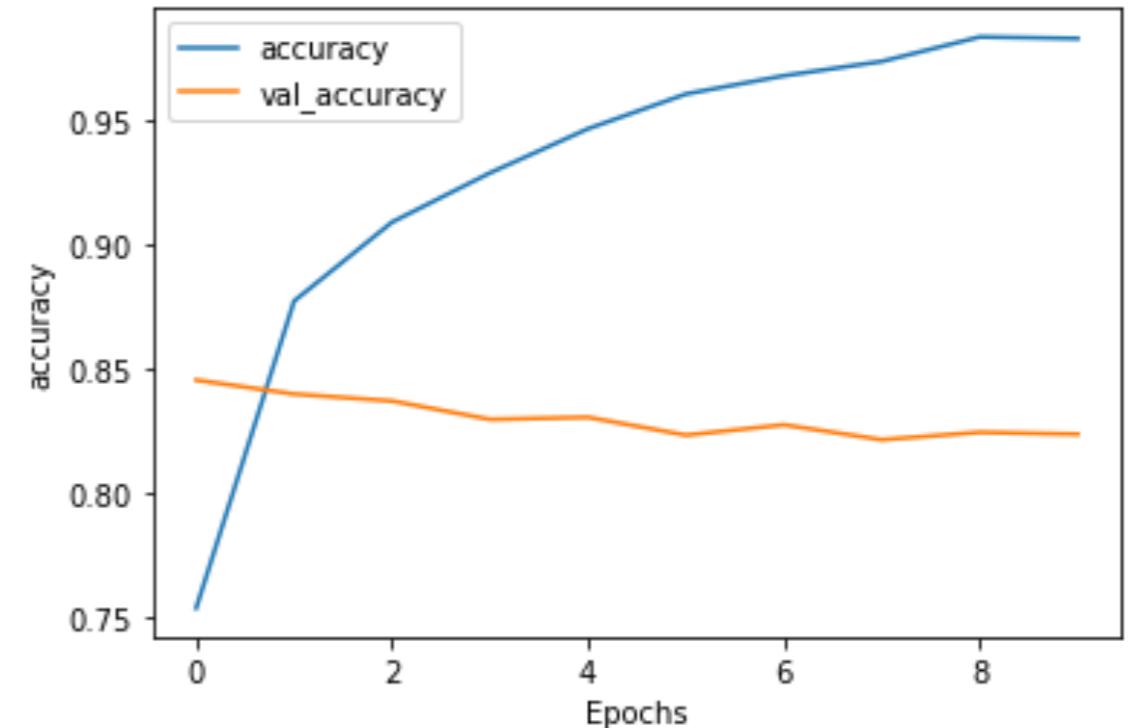
```
num_epochs = 10
history = model.fit(training_sequences_padded, training_labels_final,
                     epochs=num_epochs,
                     validation_data=(testing_sequences_padded, testing_labels_final))
```

```
Epoch 1/10
782/782 [=====] - 10s 13ms/step - loss: 0.4888 - accuracy: 0.7534
                                         - val_loss: 0.3765 - val_accuracy: 0.8452
Epoch 2/10
782/782 [=====] - 10s 13ms/step - loss: 0.3038 - accuracy: 0.8771
                                         - val_loss: 0.3699 - val_accuracy: 0.8396
...
Epoch 10/10
782/782 [=====] - 12s 16ms/step - loss: 0.0520 - accuracy: 0.9828
                                         - val_loss: 0.7797 - val_accuracy: 0.8234
Wall time: 1min 42s
```



Step 8 - Training Review

- We are at validation accuracy of 85%; and training accuracy is approaching 100%
- It is a classic sign of over fitting!



BiDirectional LSTM

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(VOCAB_SIZE, EMBEDDING_DIM, input_length=MAX_SEQ_LENGTH),

    ## BiDirectional LSTM
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),

    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
embedding (Embedding)	(None, 120, 32)	320000
bidirectional (Bidirectional)	(None, 128)	49664
dense (Dense)	(None, 64)	8256
dense_1 (Dense)	(None, 1)	65
<hr/>		
Total params:	377,985	
Trainable params:	377,985	
Non-trainable params:	0	

Two LSTM Layers

- Note, all but the last LSTM layers will have `return_sequences=True`

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(VOCAB_SIZE, EMBEDDING_DIM, input_length=MAX_SEQ_LENGTH),

    ## Two bidirectional LSTM
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),

    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Layer (type)	Output Shape	Param #
=====		
embedding (Embedding)	(None, 120, 32)	320000
bidirectional (Bidirectional)	(None, 120, 128)	49664
bidirectional_1 (Bidirection)	(None, 128)	98816
dense (Dense)	(None, 64)	8256
dense_1 (Dense)	(None, 1)	65
=====		
Total params: 476,801		
Trainable params: 476,801		
Non-trainable params: 0		

Lab: Implementing LSTMs in TensorFlow

- **Overview:**

- Work with LSTM

- **Approximate run time:**

- ~45 - 60 mins

- **Instructions:**

- **LSTM-1** - Basic sine wave
 - **LSTM-2** - Stock prediction
 - **LSTM-3** - IMDB movie review sentiment analysis
 - **(BONUS) LSTM-4** - Tweet sentiment analysis



Review and Q&A

- Let's go over what we have covered so far
- Any questions?



Transfer Learning

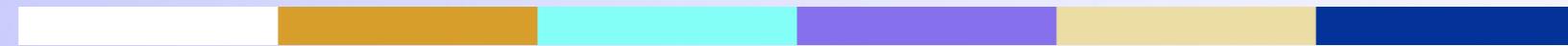
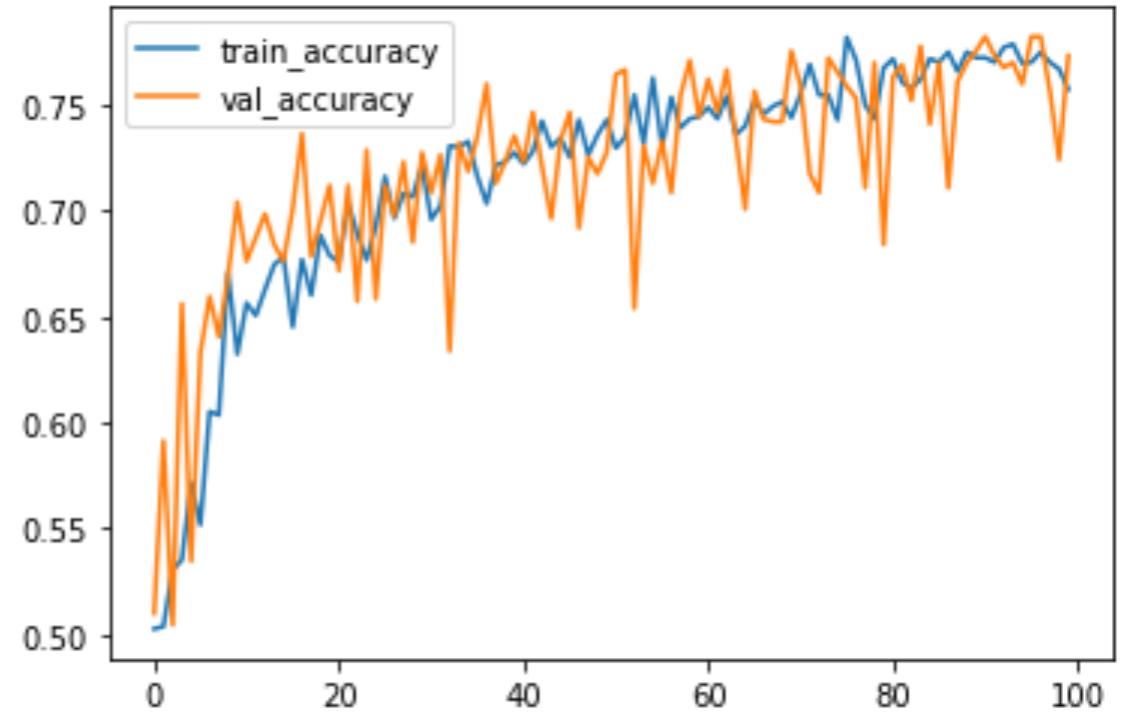


Image Classification

Image Processing is Hard!

- By now we had trained image classifiers on cat-dog / horse-human / flowers datasets
- Our datasets are of modest size (few thousands images, ~50 - 150 MB in size)
- In 'cat-dog-redux' data we had about 2000 training images (size: 45 MB) and 1000 validation images (size 22 MB)
- Our network is pretty small: 3 Convolutional layers (Conv + MaxPool) ; 10 layers total
- We trained this network for 100 epochs
 - Took about 20 minutes on a Ubuntu machine with 16 cores + 64 GB memory + Nvidia GeForce RTX 2070 GPU with 8GB (Using Tensorflow-GPU acceleration)
- Managed to achieve about 75% accuracy



Computer Vision Models

- Our modest model achieved 75% accuracy with a few minutes of training
- State of the art models can achieve 99% accuracy!
- These are trained on much larger datasets and for many hours/days/weeks!
- Can we re-use these models?
- Yes, enter **Transfer Learning!**

Transfer Learning

Transfer Learning Analogy

- Imagine you want to learn how to play the **ukulele**
- If you have no musical background, and you are starting fresh with the ukulele as your very first instrument, it'll take you a few months to get proficient at playing it
- On the other hand, if you are accustomed to playing the **guitar**, it might just take a week, due to how similar the two instruments are
- Taking the learnings from one task and fine-tuning them on a similar task is something we often do in real life.
- The more similar the two tasks are, the easier it is to adapt the learnings from one task to the other.



Training Large Models is Difficult

- Large models have many layers (deep models)
- Deep layers require **LOTS** of time and resources to train
 - Many dozens or hundreds of CPUs / GPUs
- They may require huge amounts of **data** to train
 - Maybe **petabytes** of training data.
- For example, Google translate model trains on 2 billion+ words on 99+ GPUs for week+
- What if we don't have that much compute power or don't have that much data?
- Try to re-use pre-trained models!

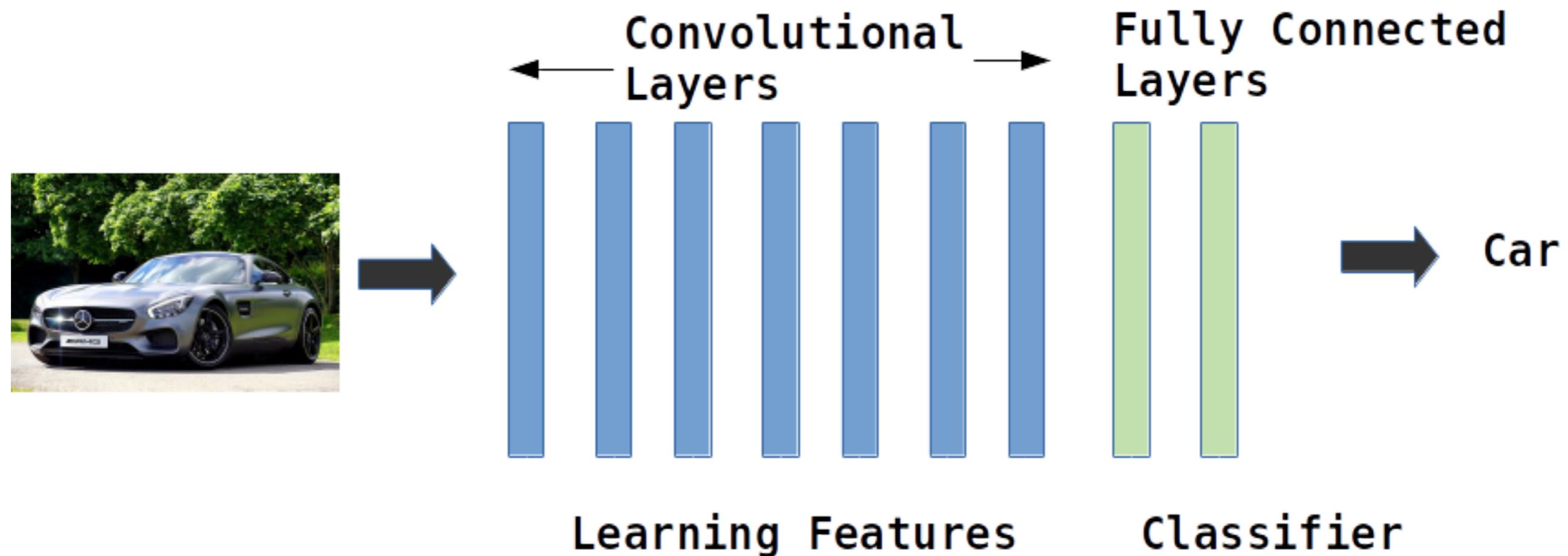
Using a Pre-trained Model

- Despite what you think, your problem is not totally unique
 - Others have worked on it before.
 - Much of their work is useful to you
 - "Stand on the shoulders of giants"
- Instead of starting from scratch, why not start from a trained model?
- But how much of the model is reusable?



Reusability of Pre-trained Models

- In a image classifier neural network, much of the early layers are for extracting features (eyes, ears etc)
- These features are pretty much the same for most real world use cases
- So we can re-use the learned knowledge in these layers

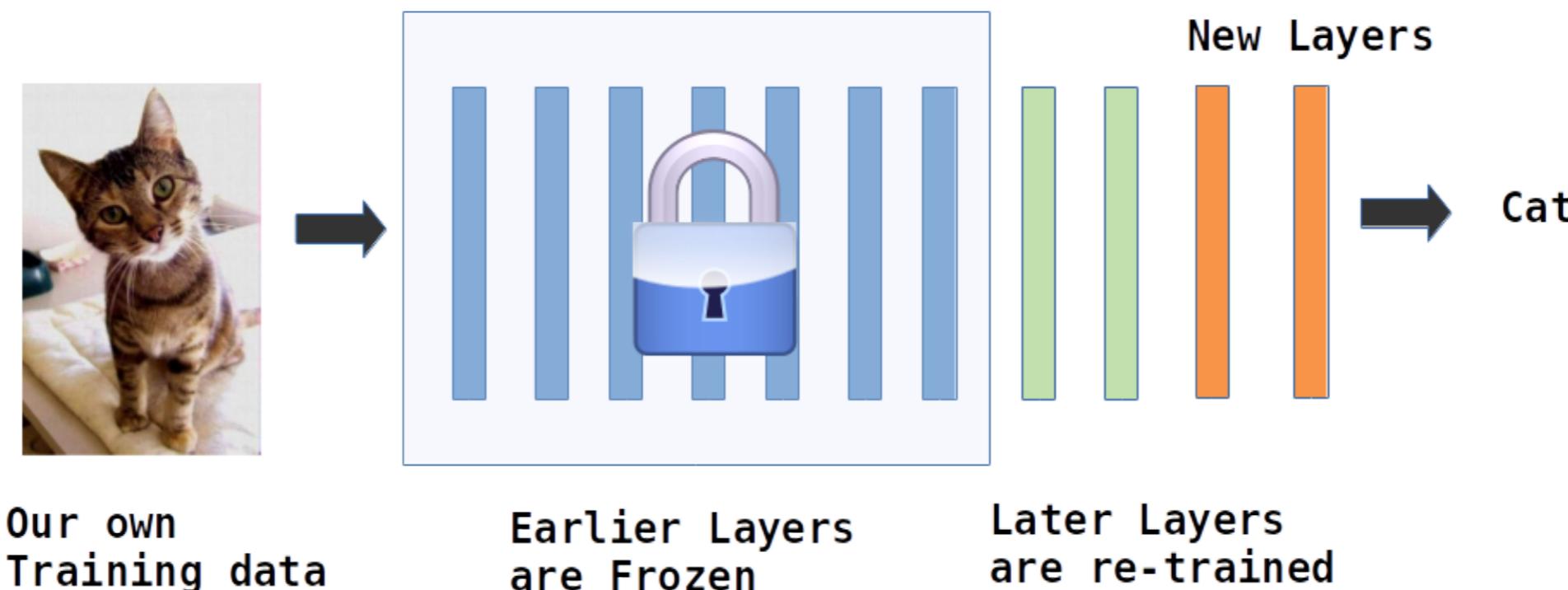


Reusability of Pre-trained Models

- The surprise is that a pre-trained model works pretty well!
- Even if it was trained with data that is totally differently from your data
- Why is this?
 - Because images are images
 - Words are words
 - etc

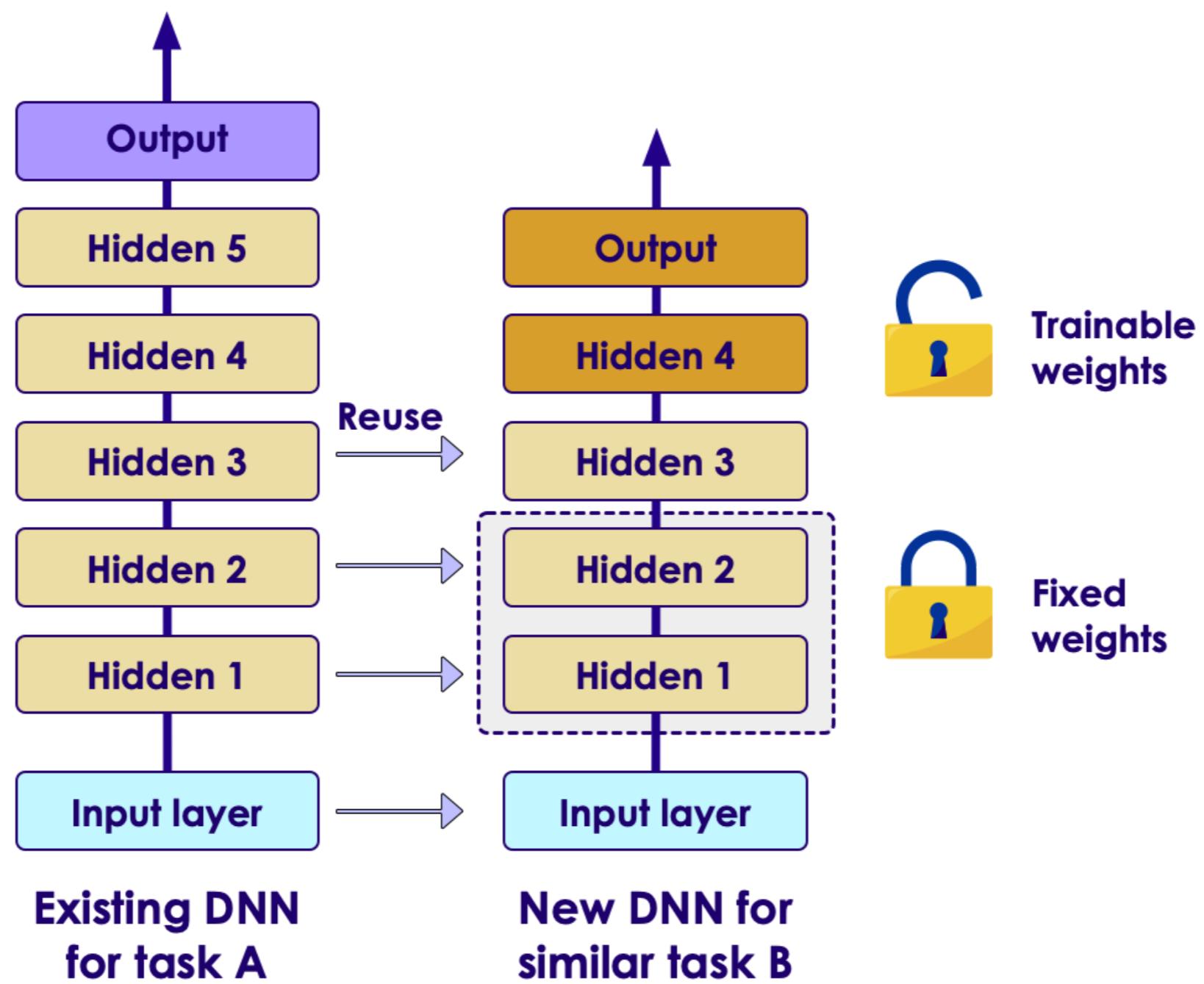
Customizing Pre-Trained Models

- Earlier layers of the pre-trained model are frozen; so their weights do not change during re-training
- The last few layers can be re-trained
- We often add several dense layers to the back end of the network.
- And then these layers are trained on **our own data**
- This allows us to customize the model to our data



Transfer Learning Process

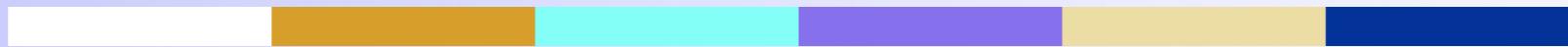
- Earlier layers are frozen; so their weights don't change during re-training
- And later layers are re-trained with our own data



Popular Pre-Trained Models

- Explore available models at modelzoo.co
 - Explore models available for Tensorflow
- **Image Recognition**
 - Inception
 - ResNet
- **Natural Language**
 - Word2Vec
 - BERT

Tensorflow Playground

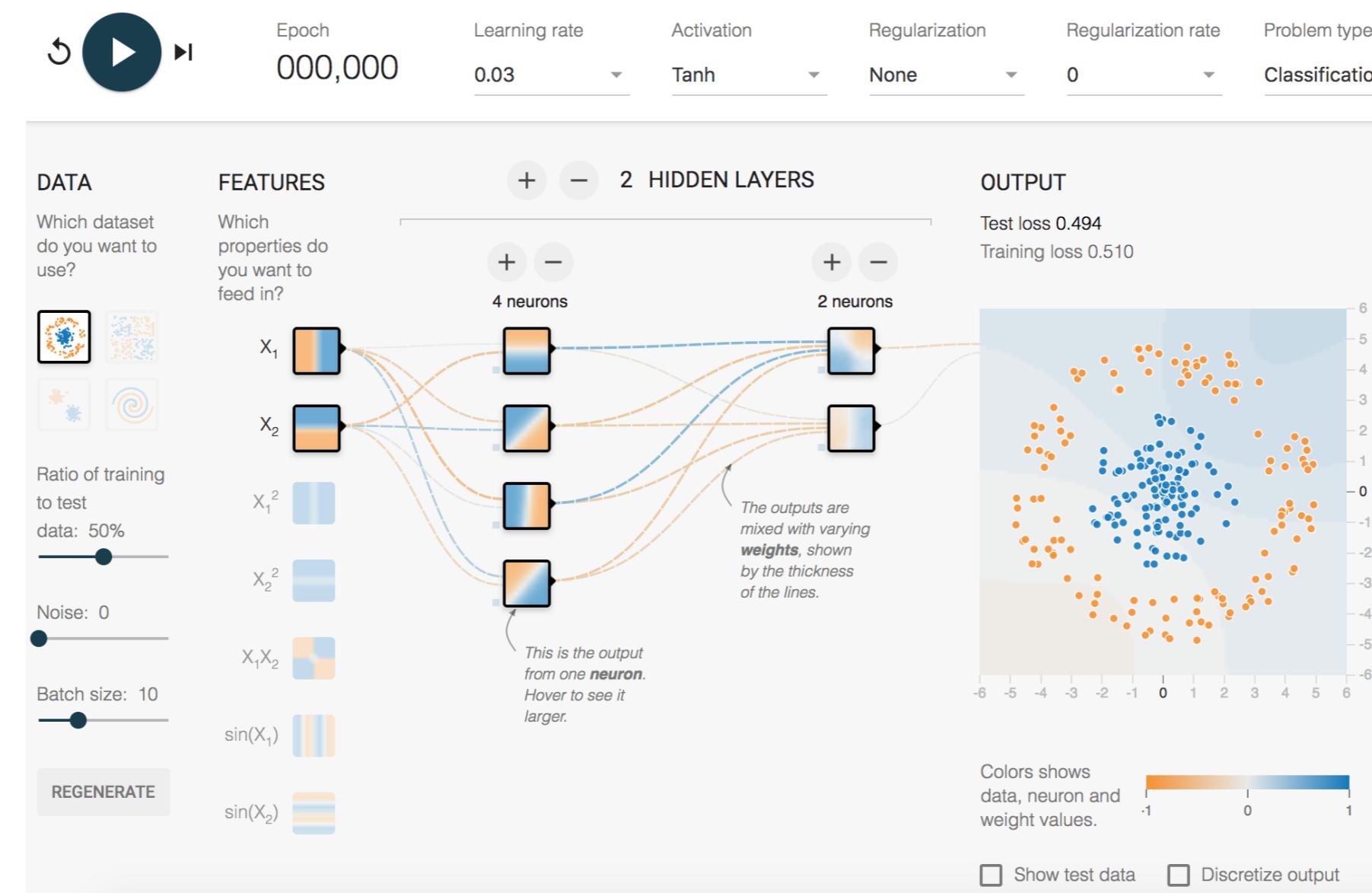


Reference Only.
Specific Sections are covered in other slides.

Introduction to Tensorflow Playground

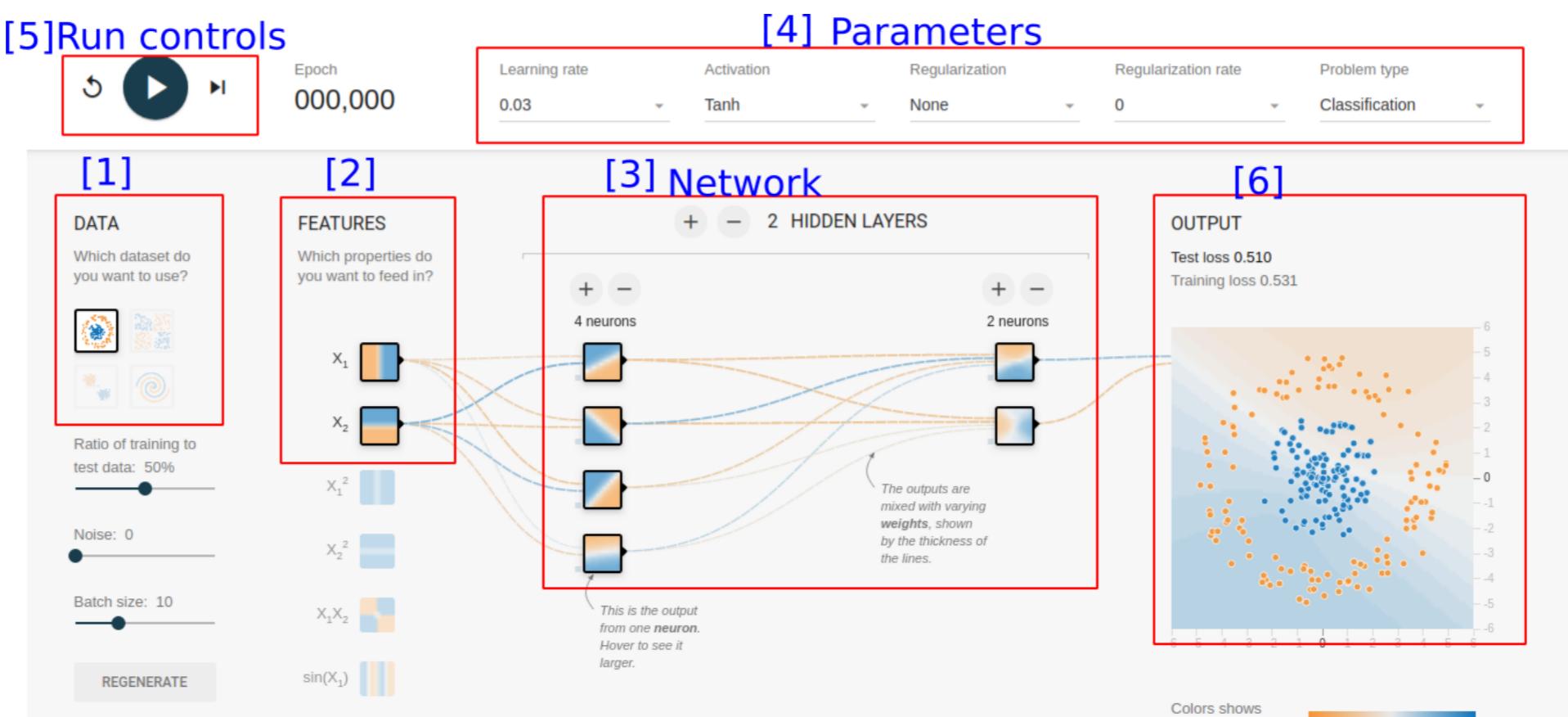
Introducing the Playground

- Navigate in your browser to <http://playground.tensorflow.org>
- This is a playground that we will use to play with some concepts
- It will be fun!
- When you start, you should see this



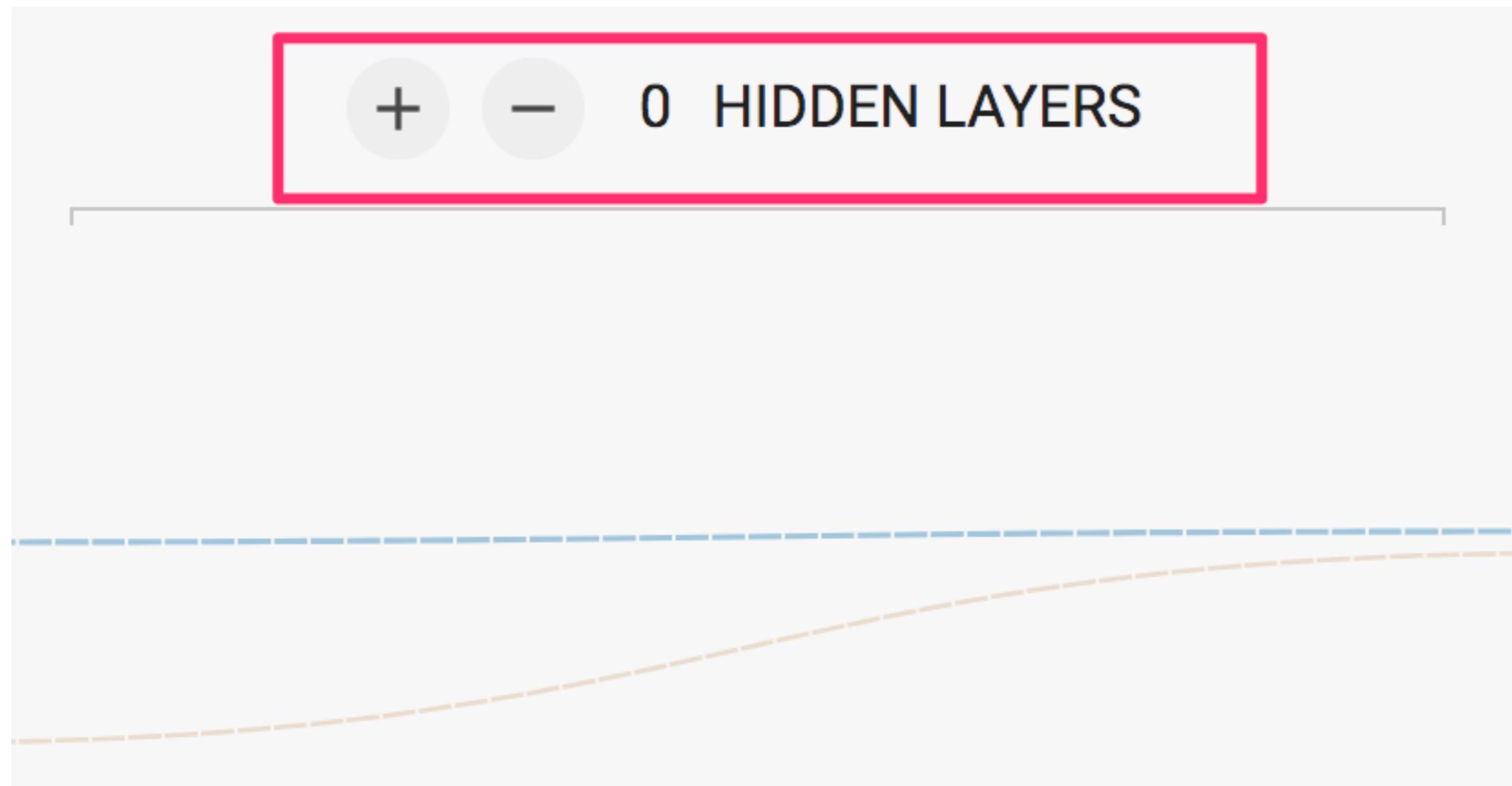
Playground Overview

- Step 1: Select data
- Step 2: Select features
- Step 3: Design neural network
- Step 4: Adjust parameters
- Step 5: Run
- Step 6: Inspect the results



Hidden Layers

- We will start out with **no** hidden layers
- Click the "minus" icon to get to no hidden layers



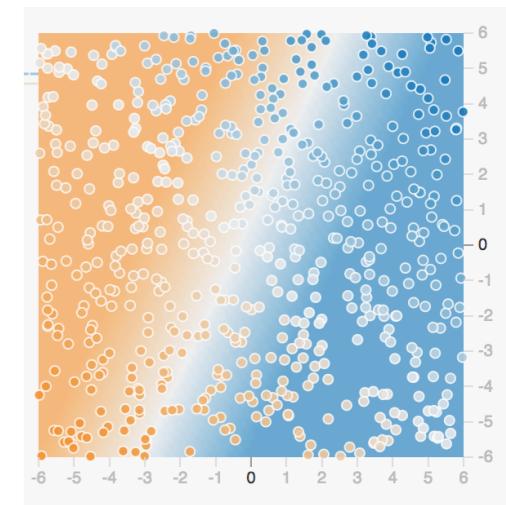
Playground Linear Regression

Linear Regression: Setup

- Click on the dropdown at the upper right, select 'Regression'



- Select the dataset in lower left



- Select the **lowest** setting of Learning Rate



Linear Regression: Parameters

Learning rate	Activation	Regularization	Regularization rate	Problem type
0.00001	Linear	None	0	Regression

- Learning Rate
 - This is the "step size" we use for Gradient Descent
- Activation Function
 - This is what we do to the output of the neuron
 - More on this later.
- Regularization / Regularization Rate
 - L1 / L2 are penalties to help reduce overfitting
 - How much to add

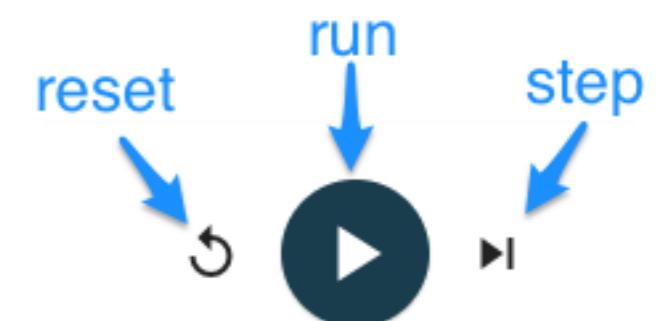
Linear Regression: Run!

- Let's try pressing the PLAY Button
- Look at the "Output" curve:
- **TOO SLOW!!! (Why??)**
 - How long (how many epochs) does it converge?
 - Do you ever get to loss = 0.0?
- What is the meaning of "loss?"
 - It's another way of saying "error"
 - In this case, it's the RMSE (Root Mean Squared Error)
- Is this dataset linearly separable?
 - Is it **possible** to get to zero loss?



Linear Regression: Adjust the Learning Rate

- Hit the reset button to the left of "play"
- Adjust the learning rate dropdown to something higher.
- Try hitting play again.
- What happens if you set a really **big** rate?
 - Note the loss is NaN (Not a Number)
 - The data is only -6.0 to +6.0.
 - A "big" value causes overshoot
- Challenge: What is the "optimal" learning rate?
 - Get to zero loss in the fewest epochs.



OUTPUT

Test loss NaN

Training loss NaN

Lab Review

- What is the impact of 'learning rate'
 - how does it affect convergence



Classification Examples 1

Linear Classification: Setup

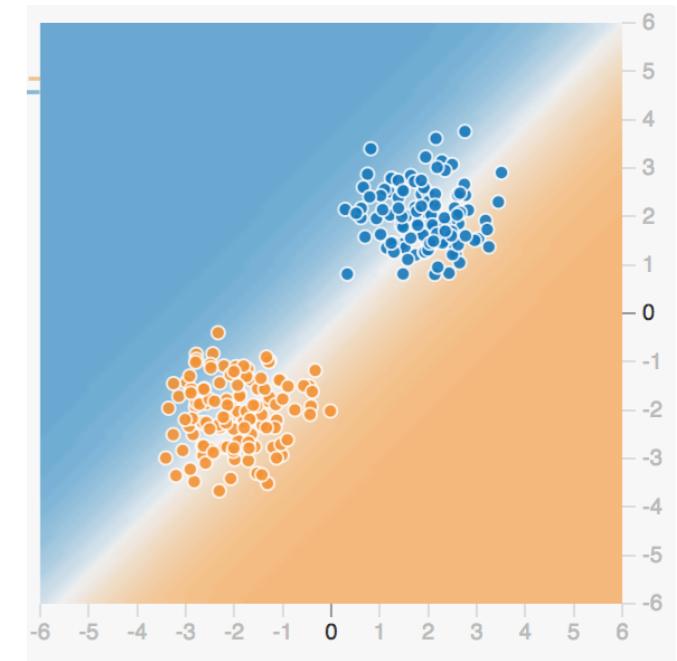
Learning rate	Activation	Regularization	Regularization rate	Problem type
0.03	Tanh	None	0	Classification

- Parameters

- Select 'Classification' on the dropdown at the upper right
- Activation : Tanh
- Learning Rate: 0.01

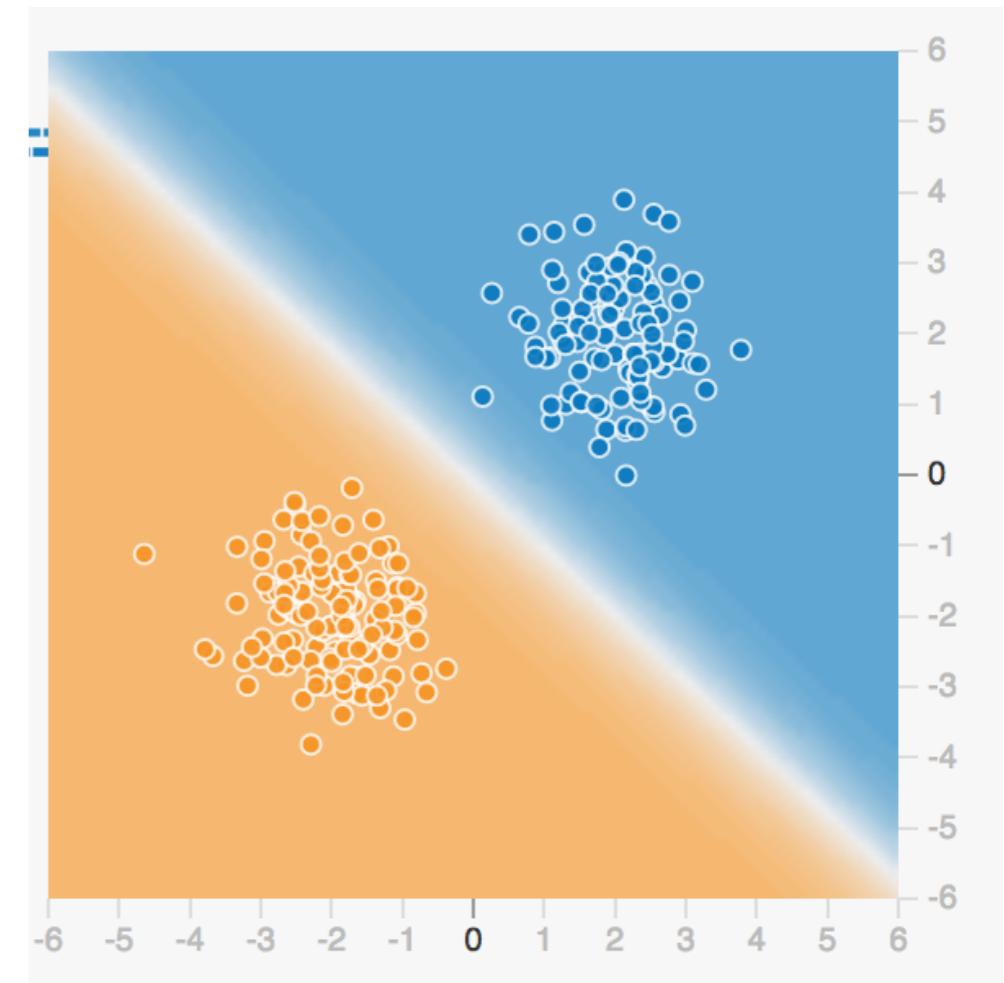
- Select the Two-Blob Datasets

- Is this dataset linearly separable?



Linear Classification: Run

- The separated dataset might look like below
- You may not get zero loss, especially if you introduce noise
- Challenge: Adjust the learning rate to get to minimum loss in as few epochs as possible.



Lab Review

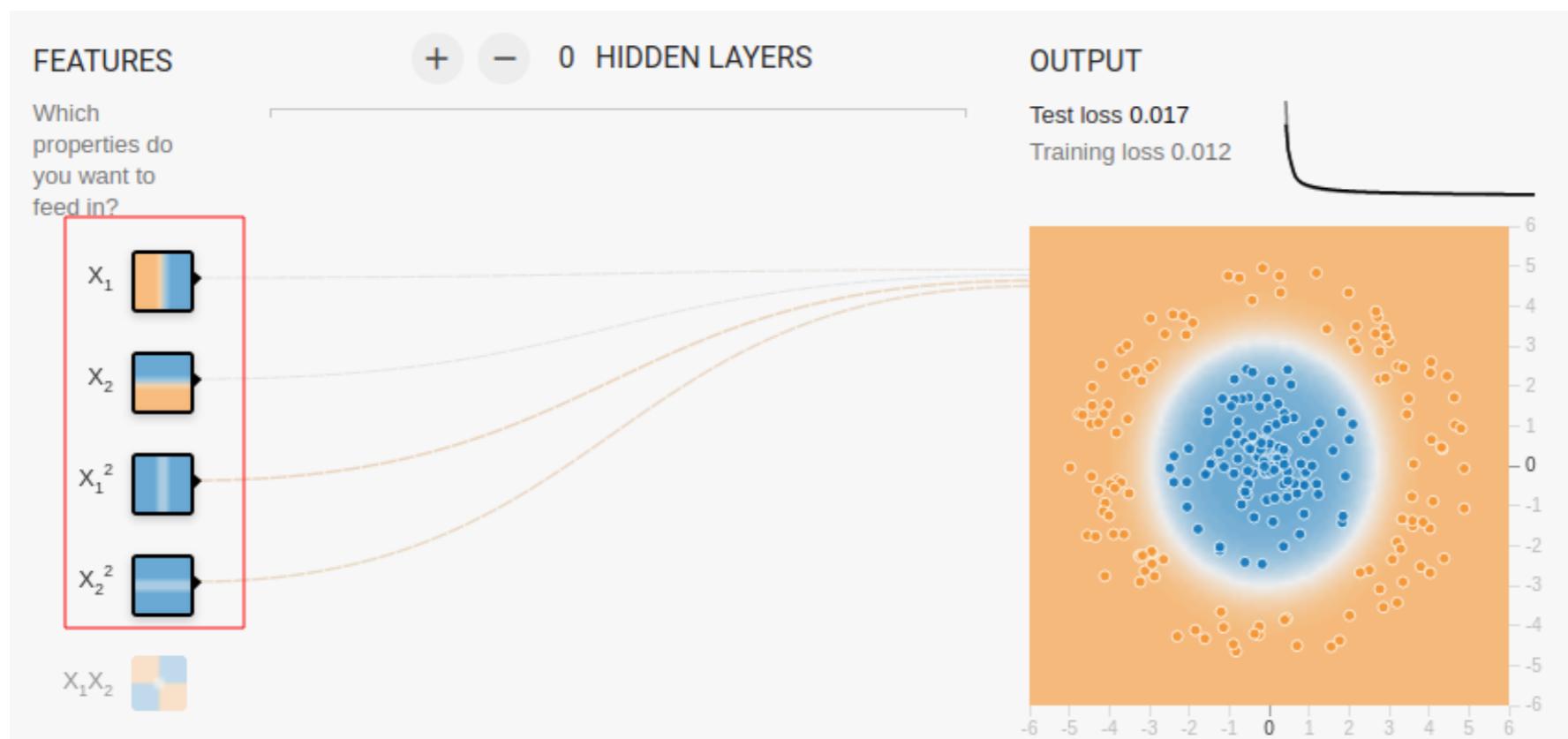
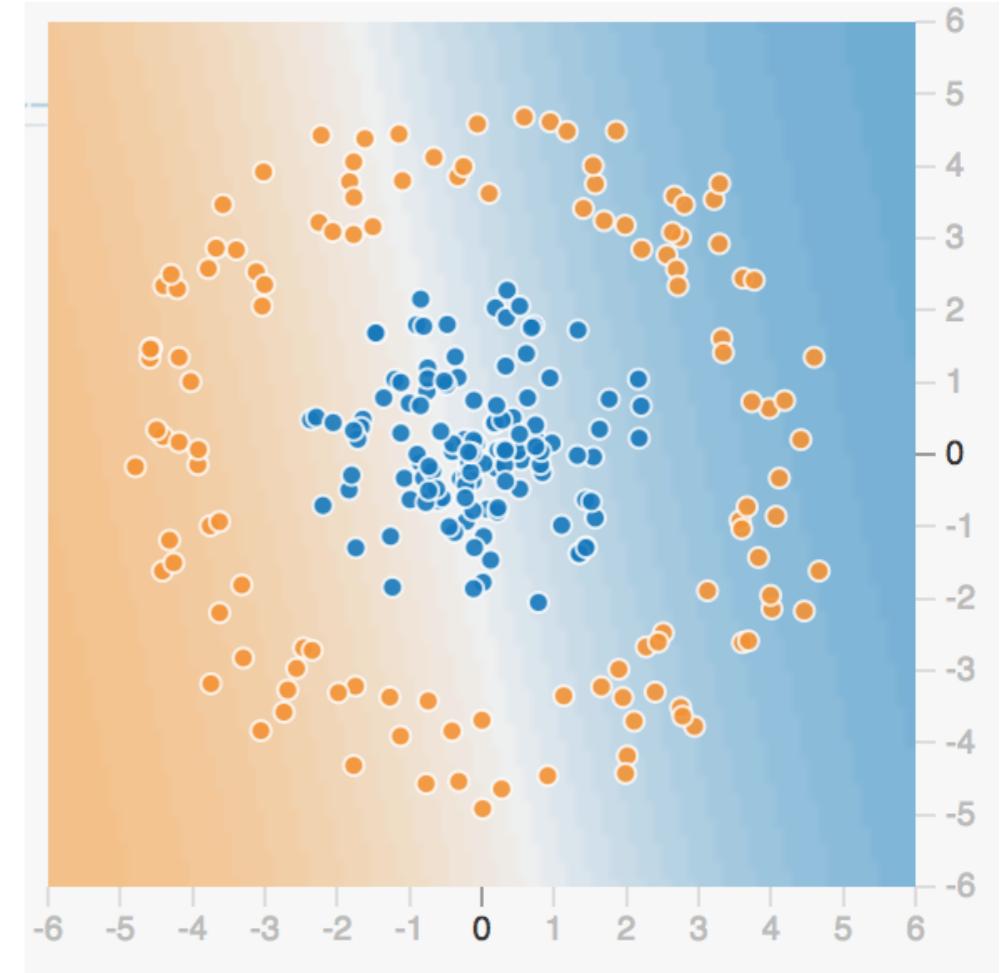
- Why didn't we need hidden layers to converge on a solution?
- What would happen if the dataset wasn't linearly separable?



Classification Examples 2

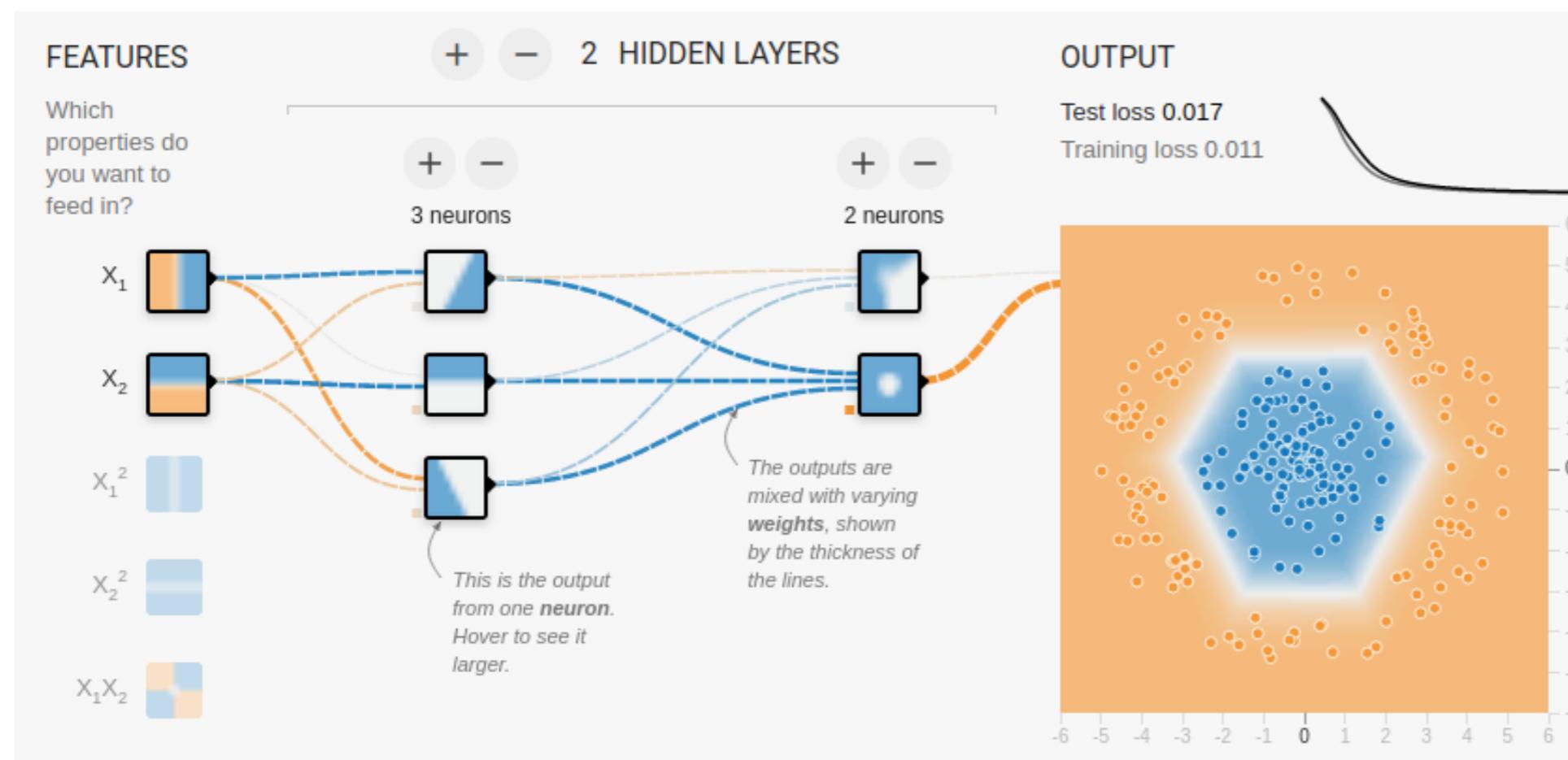
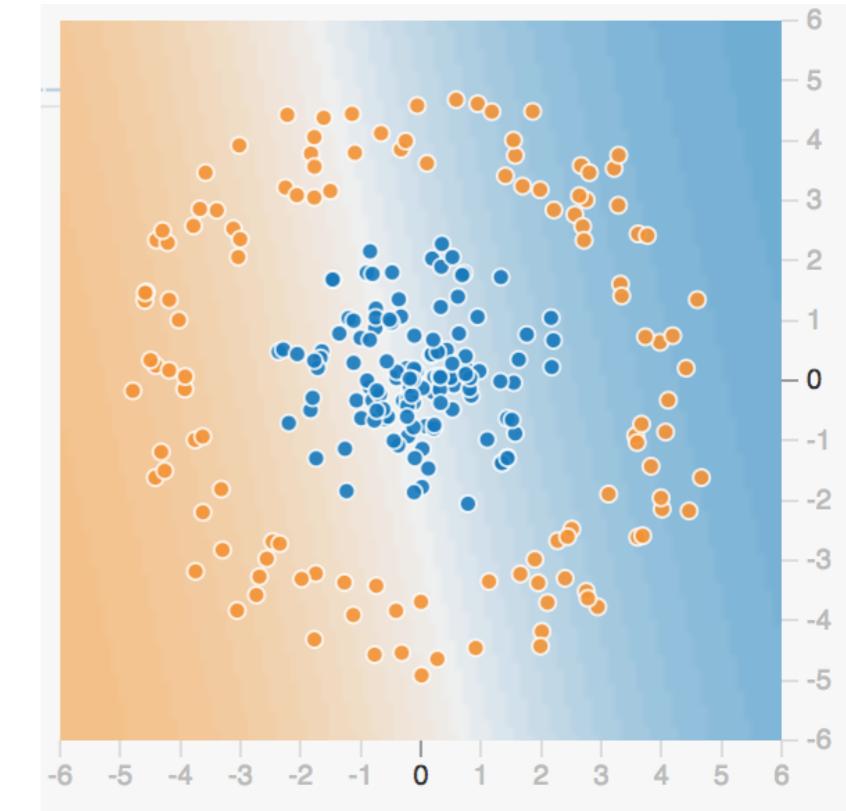
Circle Dataset

- Select the circle dataset
- Can we linearly separate this dataset?
 - No amount of fiddling with learning rate will help!
 - It's not linearly separable.
- Solution-1: Include other features
 - $x_1 + x_2 + x_1^2 + x_2^2$



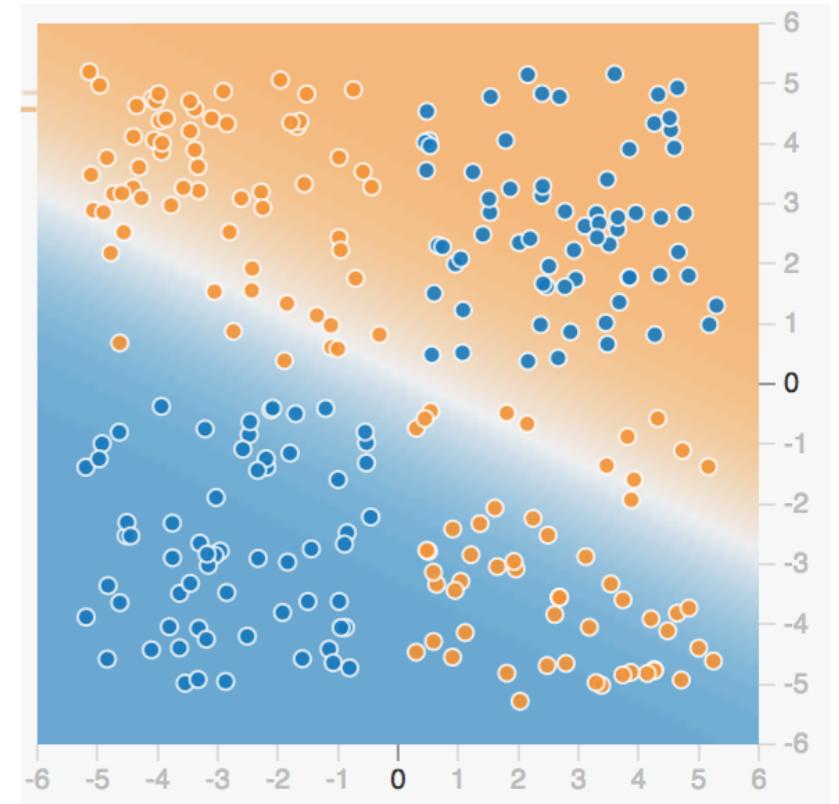
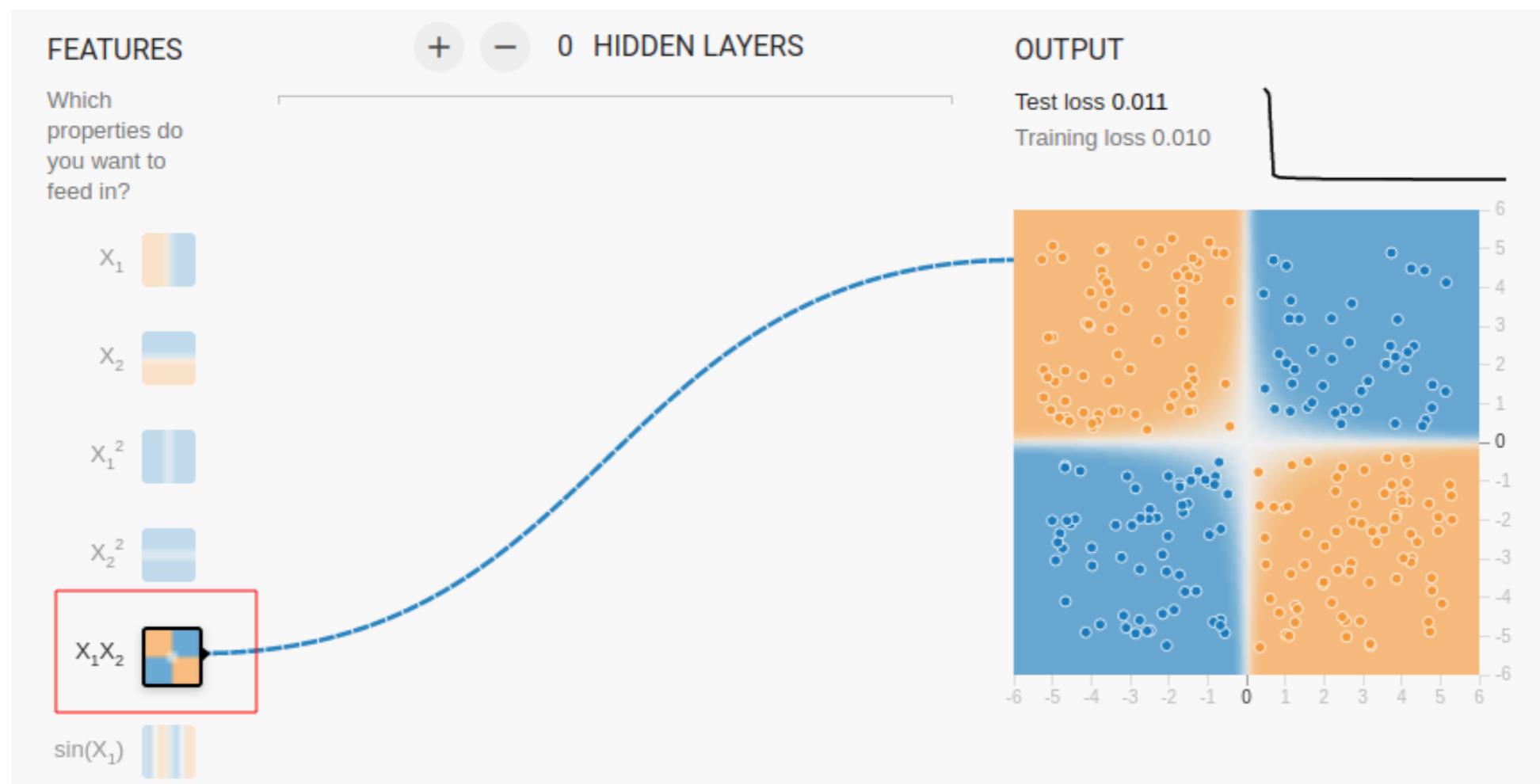
Circle Dataset With Hidden Layers

- Select the circle dataset
- Select only X_1 and X_2 as features
- Add a Hidden Layer
- Can you get a solution with 1 hidden Layer
- You can add more neurons to the hidden layer
- Can you solve it with only one hidden layer?
- If not, add another hidden layer



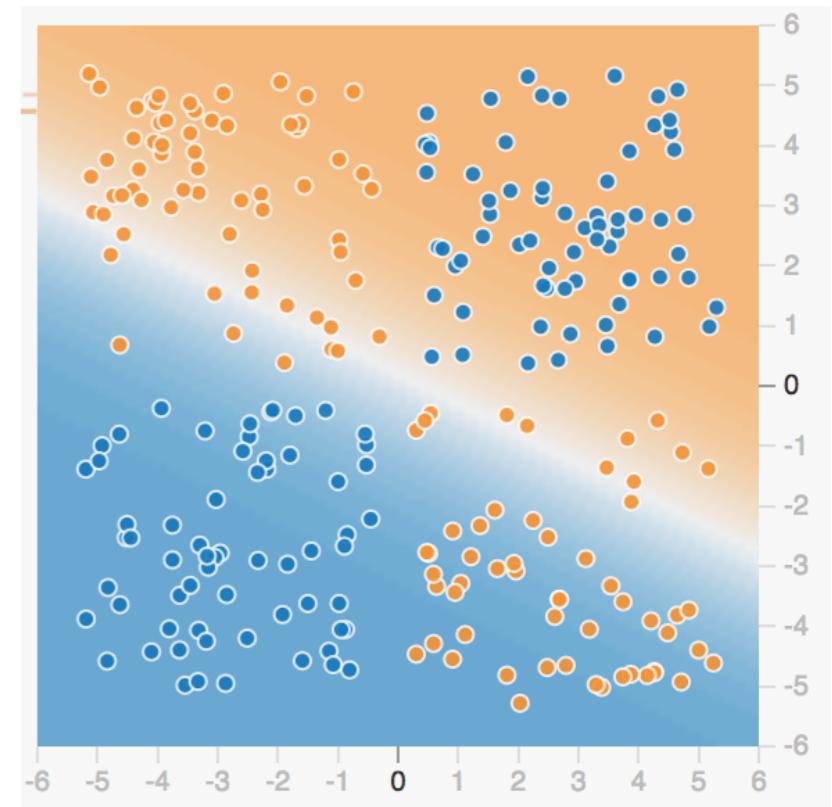
Four Square Dataset

- Set the Four-Square dataset
- Try setting the input to $X_1 . X_2$ with no hidden layers



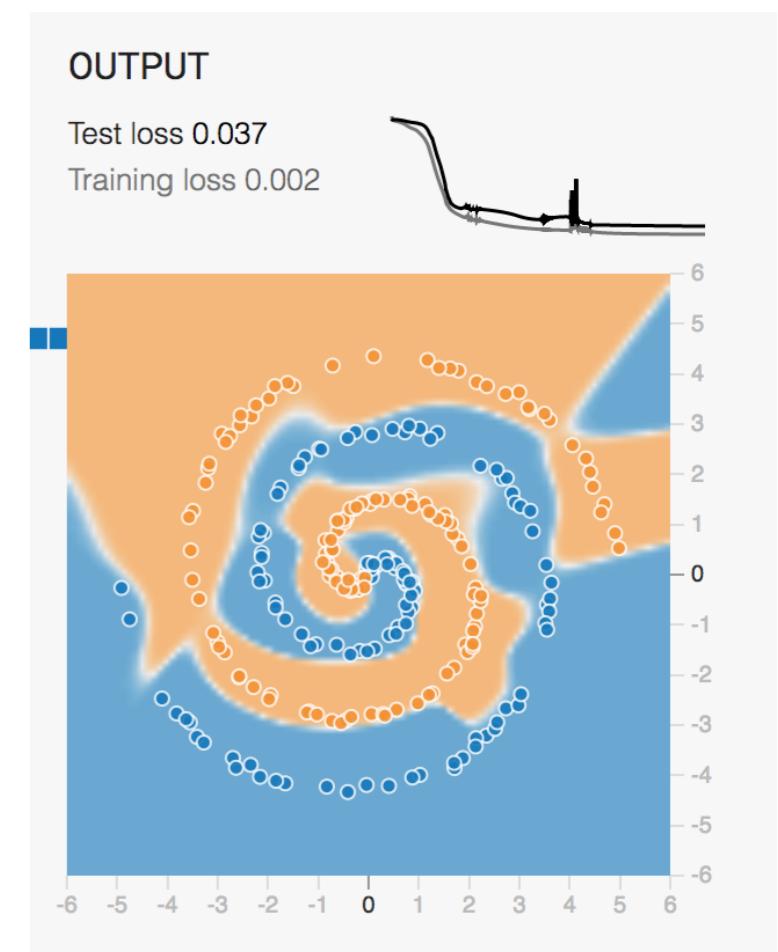
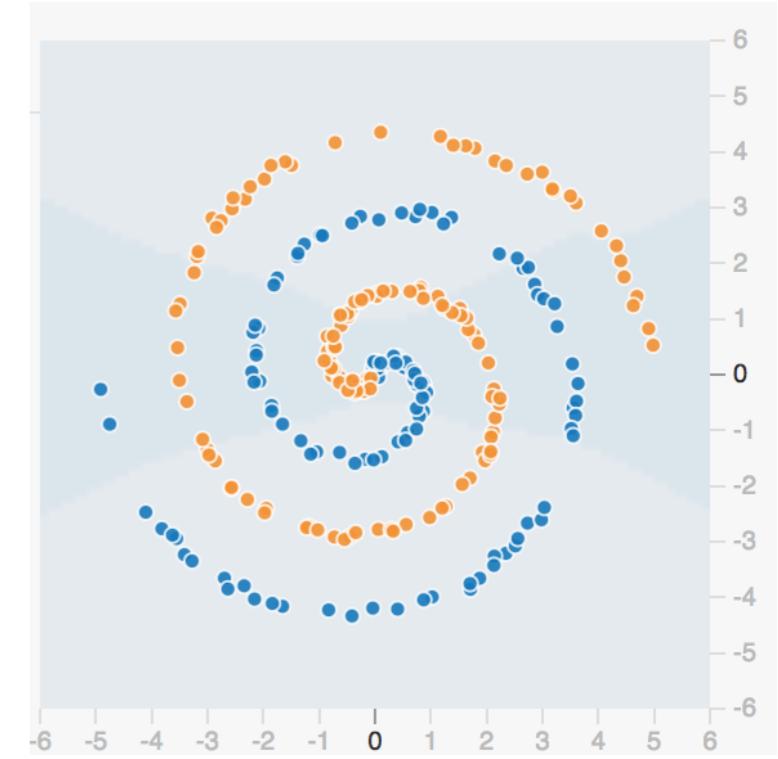
Four Square Dataset With Hidden Layers

- Solve the **Four Square** using hidden layers
- Set the inputs to **X1** and **X2**
- Try adding 1 or 2 hidden layers
- **Instructor** : Offer hints from the notes section



Spiral Dataset

- Set the Spiral dataset
- This is a **challenging** dataset
- Try
 - multiple features
 - multiple hidden layers
- Can you get this result?
- **Instructor** : Offer hints from the notes section

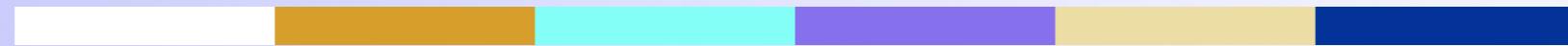


Lab Review

- What's the minimum number of hidden layers required to correctly classify all the test data?
- Does adding any additional features help at all?
- Do we necessarily get better results with more neurons and/or hidden layers?



Machine Learning Concepts

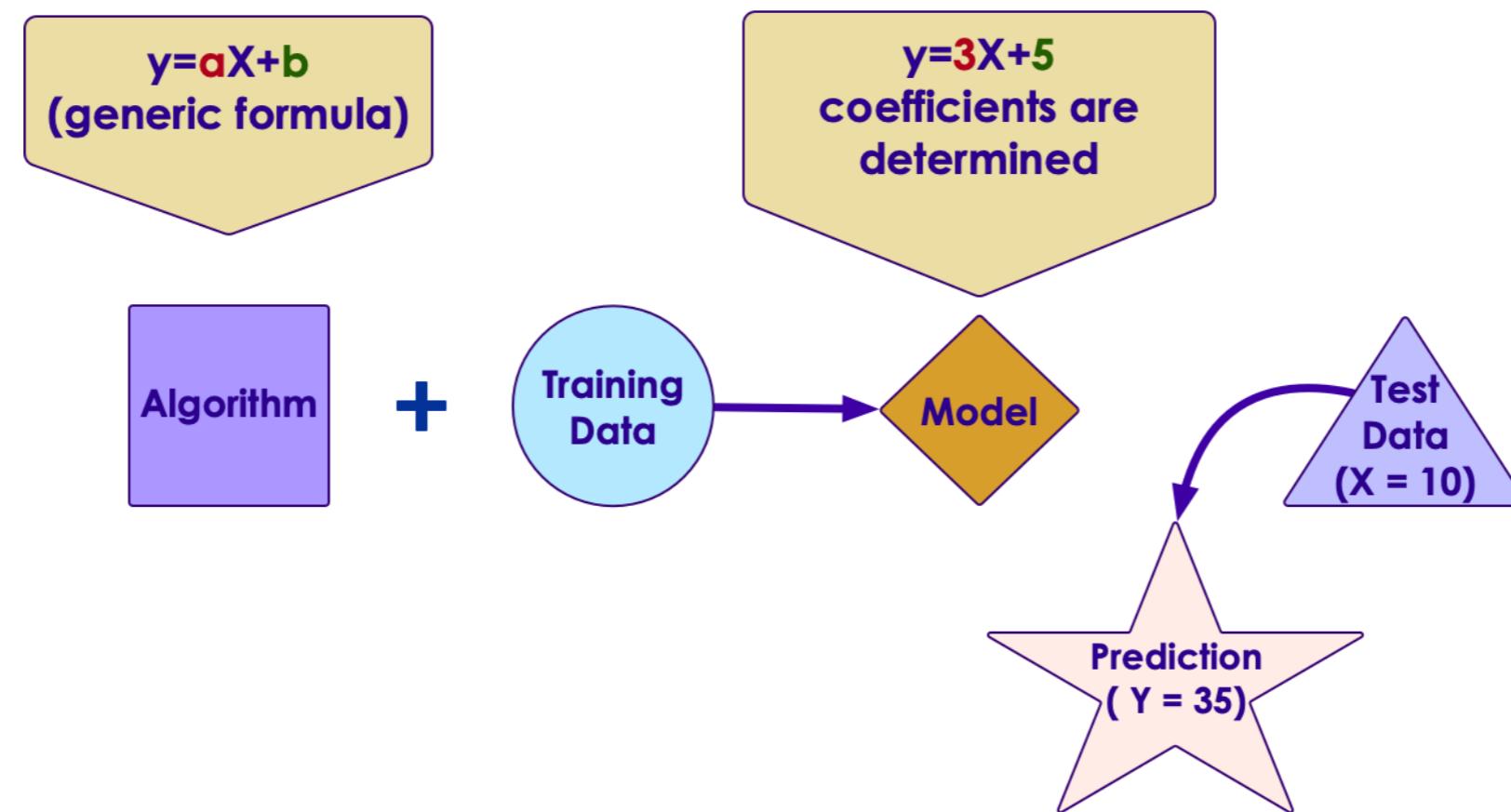


ML Concepts Part 1

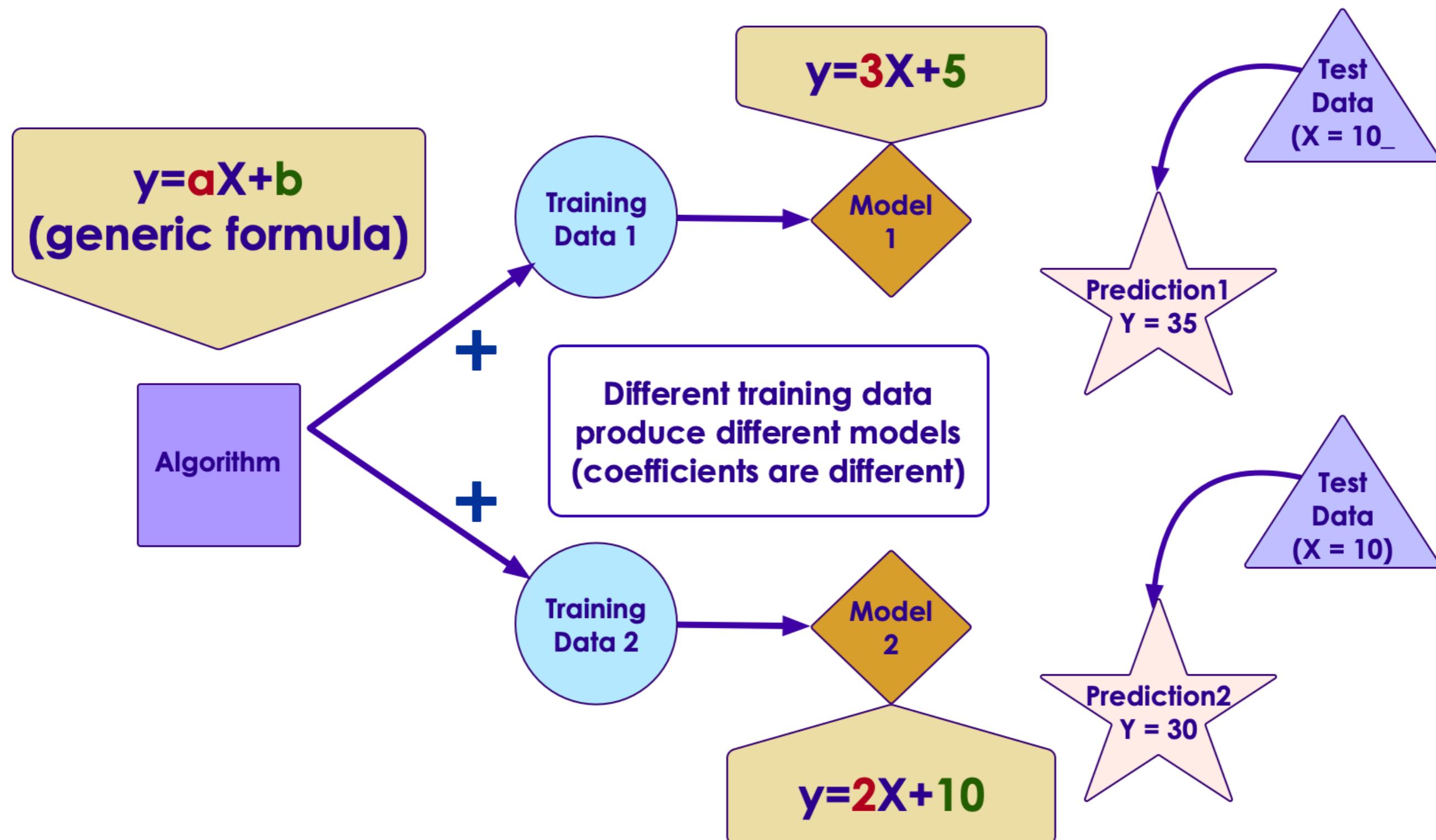
Machine Learning Terminology

Algorithm vs. Model

- Often Algorithm and Model are used interchangeably
- Algorithm: Mathematical / statistical formula or methodology
- Training the algorithm with data yields a model
 - Algorithm + training data -> model
 - Model = Algorithm(data)



Algorithm and Model



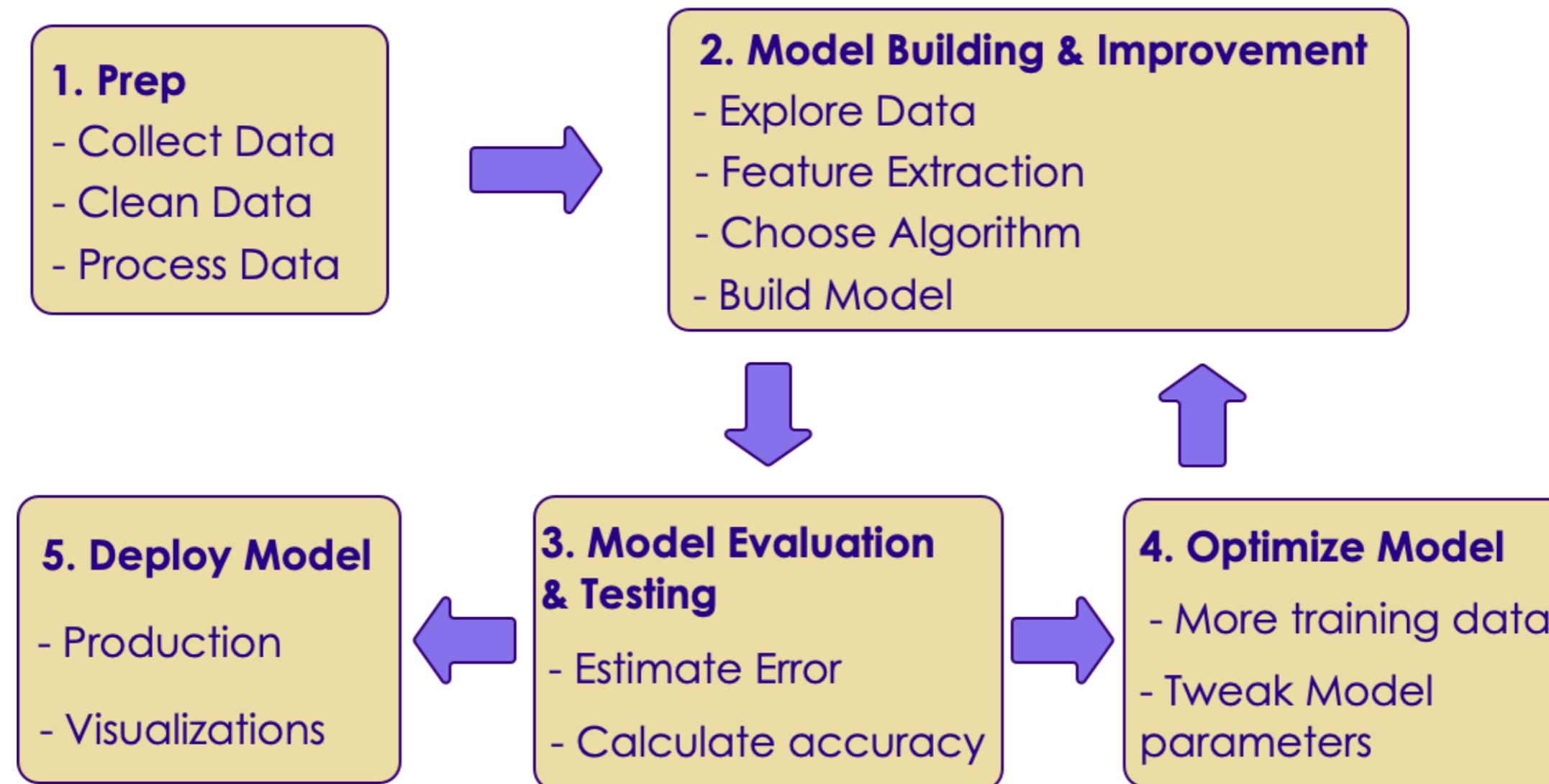
Model

- 'Model': Mathematical object describing the relationship between input and output
- We can treat ML model as a 'black box'
- Input goes in, model produces an output



Machine Learning Workflow / Process

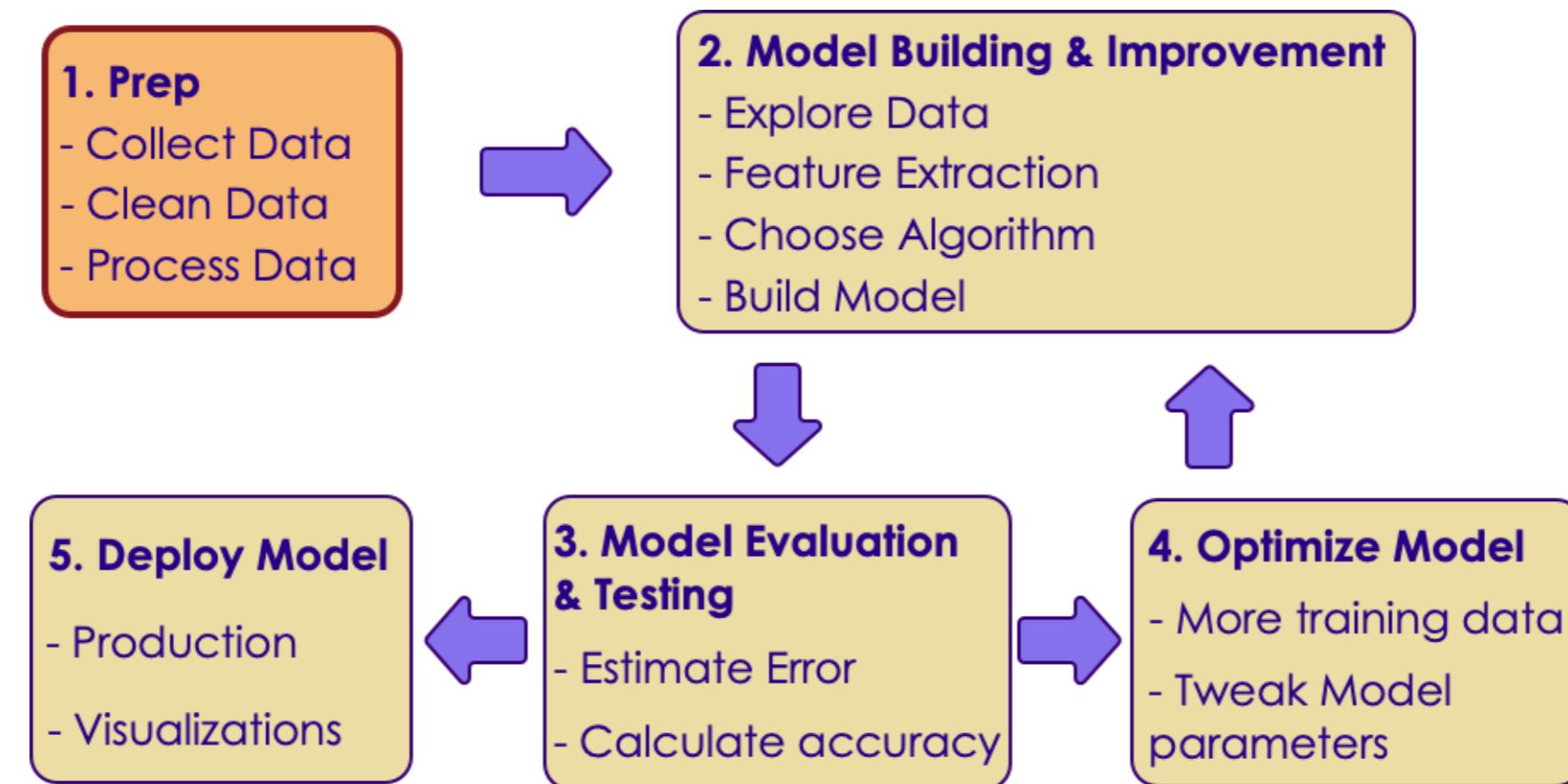
Machine Learning Process



Machine Learning Process

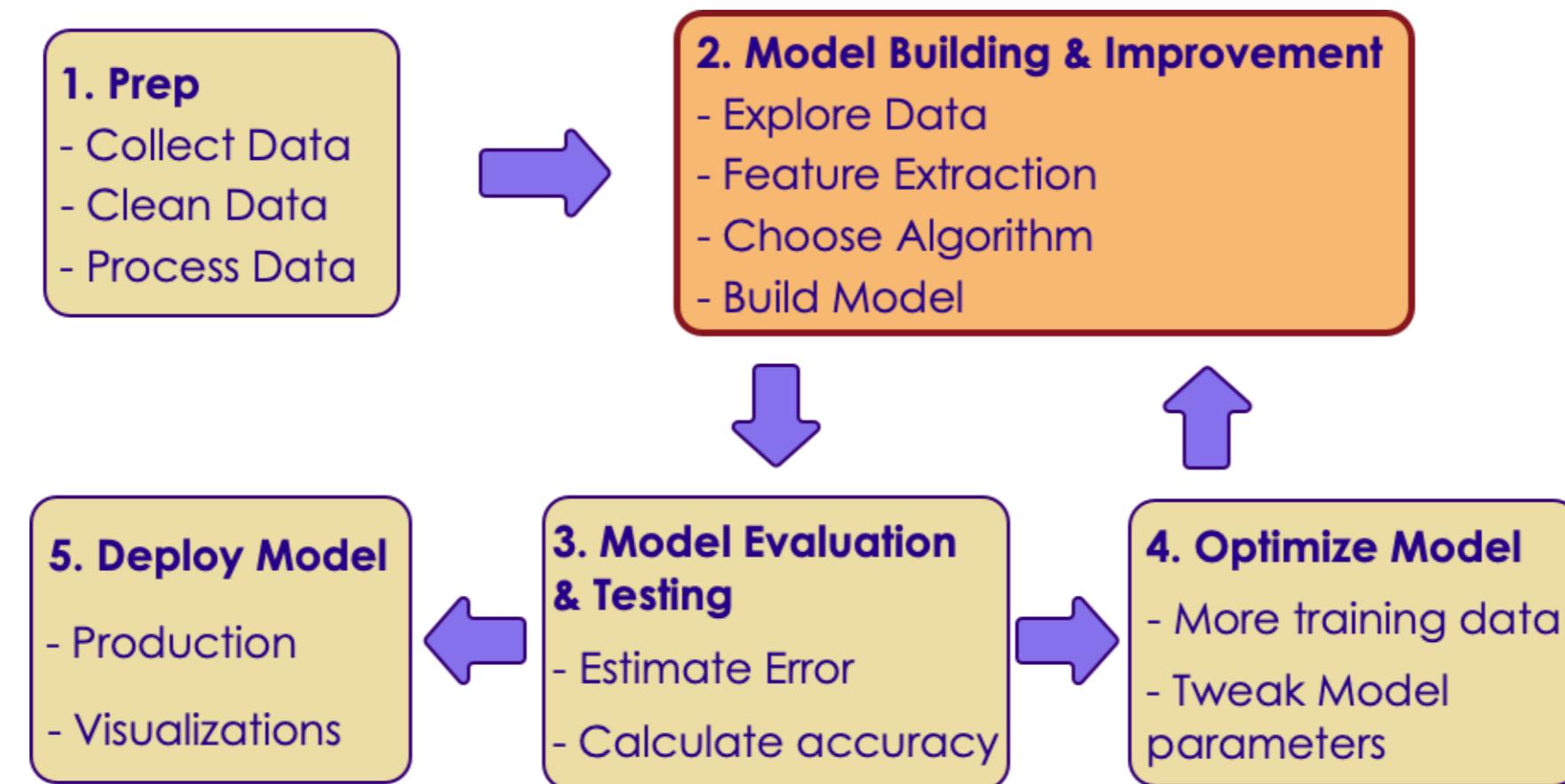
- Step 1: Get Data
- Step 2: Explore Data and build model
- Step 3: Evaluate model
- Step 4: Optimize model
- Step 5: Deploy and monitor

Machine Learning Process: Step 1: Data Exploration



Developing A Model

Machine Learning Process: Step 2: Developing a Model



Sample Dataset: Cars

- We want to predict MPG of a car
- What attributes to consider?

	row.names	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
13	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4

Sample Model for Predicting MPG

- Designate inputs as X
 - X_1 : first input (e.g. number of cylinders)
 - X_2 : second input (e.g. weight of car)
 - X_i : ith input (e.g. horsepower)
- Output / target variable is denoted as Y
- $Y = f(X) + E$
 - Y : Target:
 - Inputs ($X_1, X_2 \dots$ etc)
 - E : error / noise

Goal: Find $f()$

$$Y = f(x_1, x_2, x_3) + E$$

row.names	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1 Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2 Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3 Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4 Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5 Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6 Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7 Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8 Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9 Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10 Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
11 Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
12 Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
13 Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
14 Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
15 Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4

Let's Play a Guessing Game!

- Look at the data below. Come up with a formula linking X and Y

X	Y
1	2
2	5

- So what is the formula?
- $Y = ???$
- Answer next slide



Guessing Game

X	Y
1	2
2	5

- I have 2 possible formulas (there may be more)
- $Y = 3X - 1$
- $Y = X^2 + 1$



Guessing Game

- Let me provide more data

X	Y
1	2
2	5
3	10
4	17



- Now, what would be the formula?
- Answer next slide

Guessing Game

X	Y
1	2
2	5
3	10
4	17

- With more data, we can finalize on a formula
- $Y = X^2 + 1$
- Lesson: More (quality) data we have, we can come up with a more precise formula
- **This is the essence of machine learning!**



Modeling Techniques (Little Math!)

- ML model has two types: Parametric / Non-Parametric
- Parametric models assume a strong 'f'
 - Tend to be simple models
 - May not be very accurate
- Non-parametric models don't assume a rigid 'f'
 - Adopt to data very well
 - More accurate
 - More difficult to understand than parametric models

Parametric vs. Non Parametric

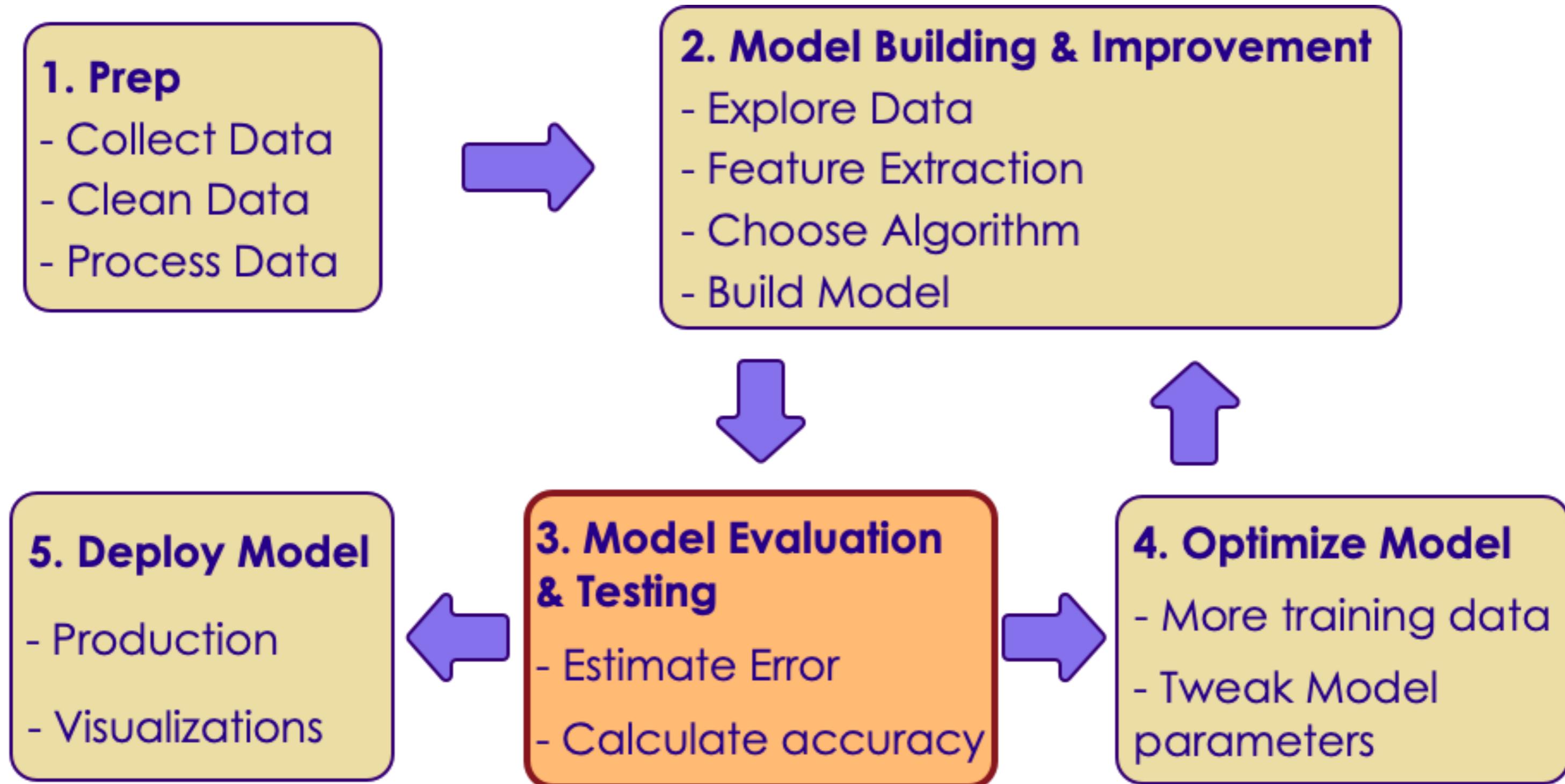
	Parametric	Non Parametric
Advantages	<ul style="list-style-type: none">- Simpler- Very fast to learn from data- Don't required, a lot of training data	<ul style="list-style-type: none">- Flexible: can adopt to complex data,- No assumptions about underlying function,- good prediction performance
Disadvantages	<ul style="list-style-type: none">- limited by function- Can not adopt to complex data- Can underfit	<ul style="list-style-type: none">- Complex to understand and explain,- Require more data for learning,- Slower to train as they have more parameters to tweak,- Can over-fit
Algorithms	<ul style="list-style-type: none">- Linear Regression- Logistic Regression- Linear Discriminant Analysis	<ul style="list-style-type: none">- Decision Trees,- Support Vector Machines,- Naïve Bayes
Best for	<ul style="list-style-type: none">- small size data with previous knowledge of features	when having lots of data and no prior knowledge of features

Parametric vs. Non Parametric

	Parametric	Non Parametric
Model complexity	Simple	More complex
Training speed	Fast	Slow
Amount of training data	Doesn't need a lot	Needs more data
Explainability	Simple to explain	Harder to explain
Fit	under-fit	over-fit
Adopting to data	simple data	complex data
Prediction accuracy	good	better

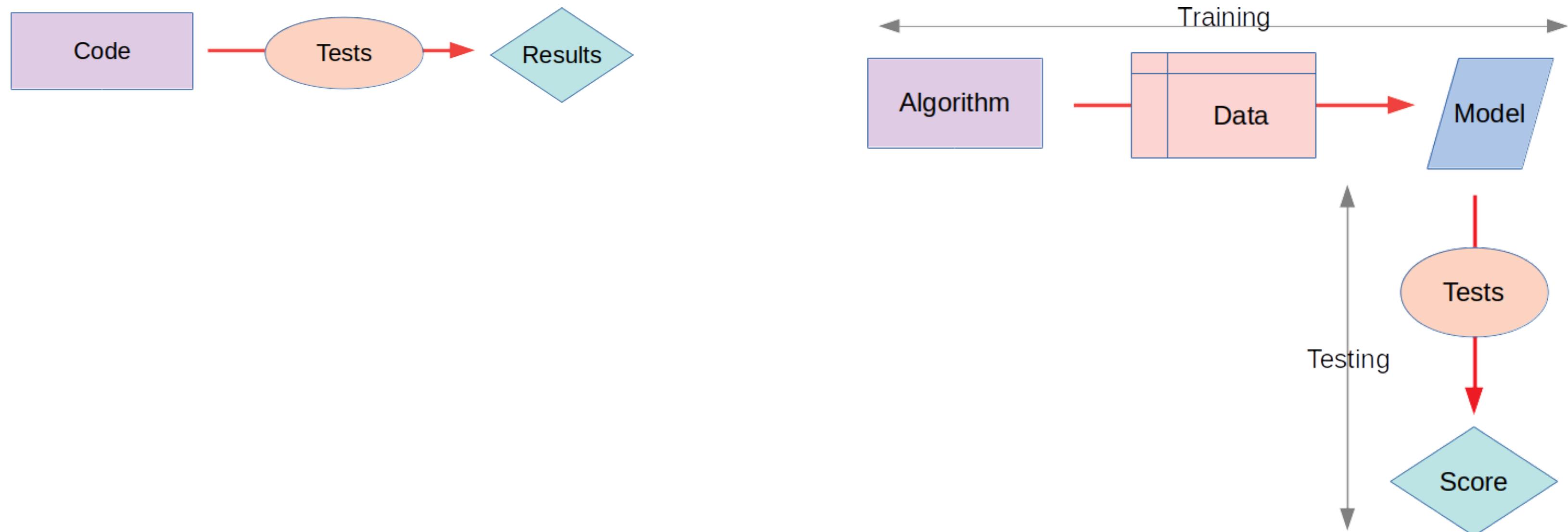
Model Validation

Model Evaluation



Evaluating A Model

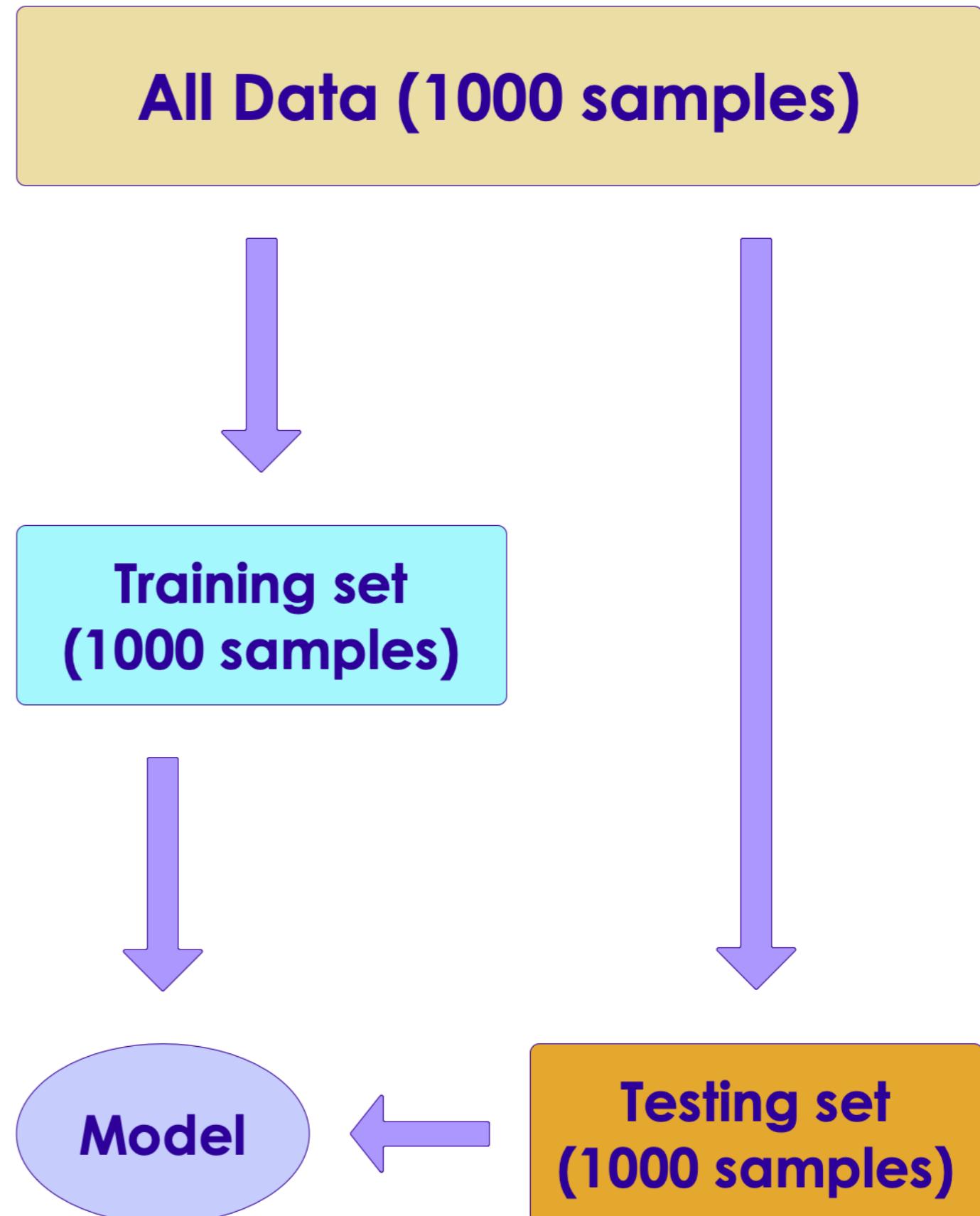
- How do we know our model is 'good'?
- Just like we test our software, we test our model
- Model is trained with 'training data'
- Its performance is measured on 'test data' (the model hasn't seen 'test data')



Model Validation Methods

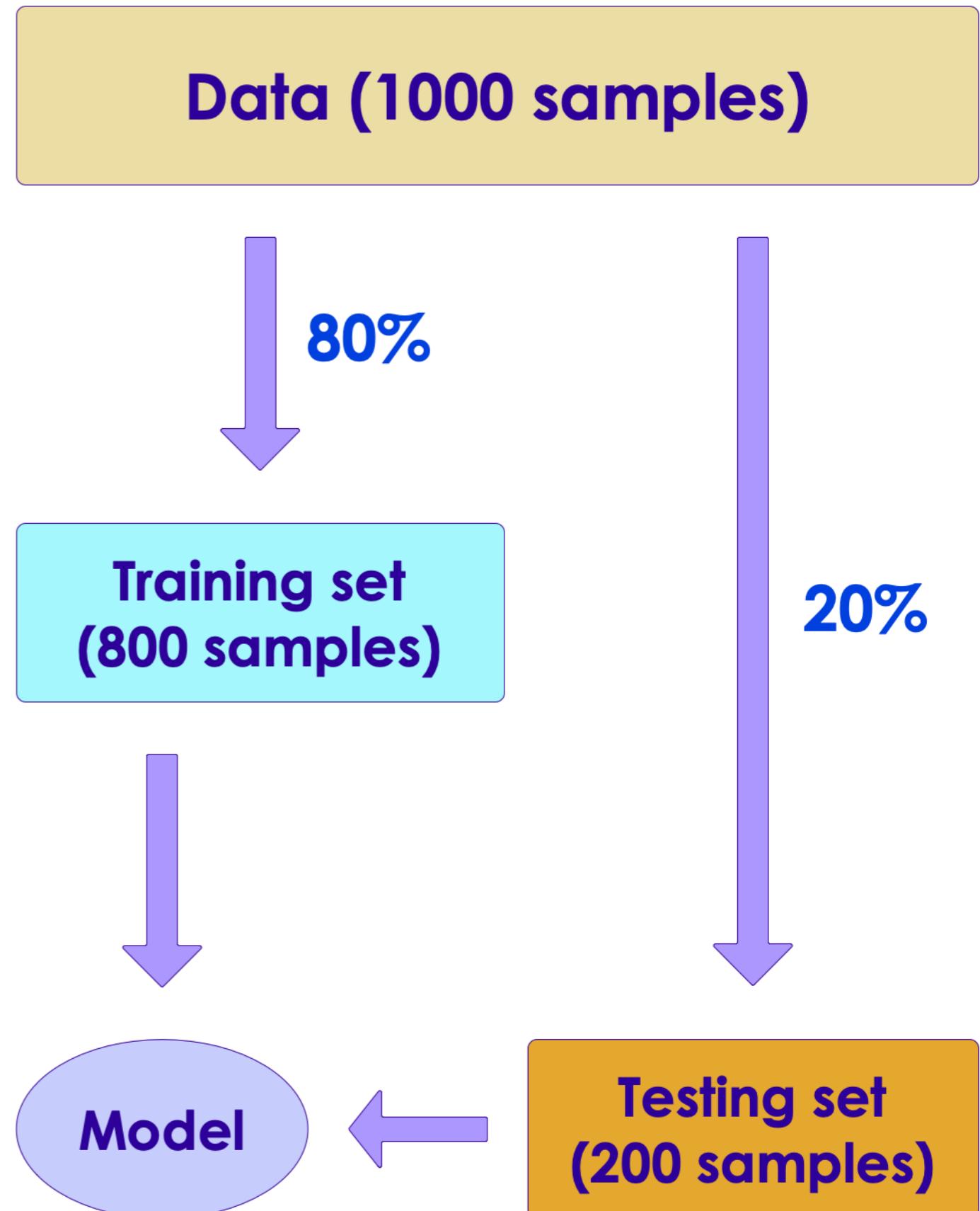
Model Validation

- **Mistake: Re-using 'training data' as 'testing data'**
- Here we are using the same data for training and testing
- Model can predict well on testing (because it has 'seen' the data before during training)
- This gives us 'false confidence'
- But the model will do badly on new data
- **Solution: We need to use separate datasets for training and testing**



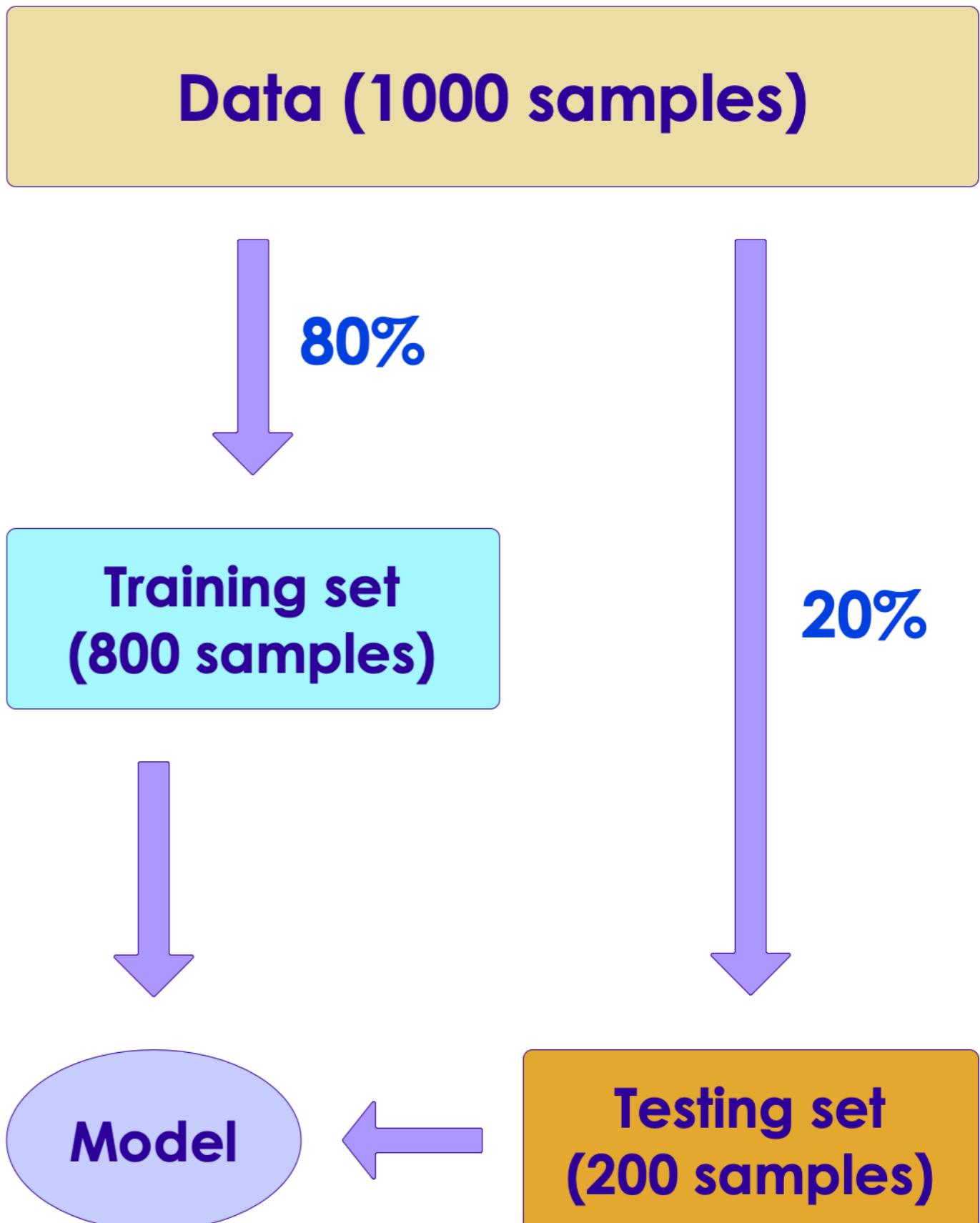
Hold Out Method

- Here we split the data into
 - Training set (80%)
 - Testing set (20%)
- The split is done **randomly**
- The split is done so majority of data goes to training set
 - 70% training + 30% testing
 - 80% training + 20% testing
 - No hard rule, adjust as needed
- The following are not great splits:
 - 50% training + 50% testing : too little training data
 - 95% training + 5% testing : may not be enough data for testing



Hold Out Method Drawbacks

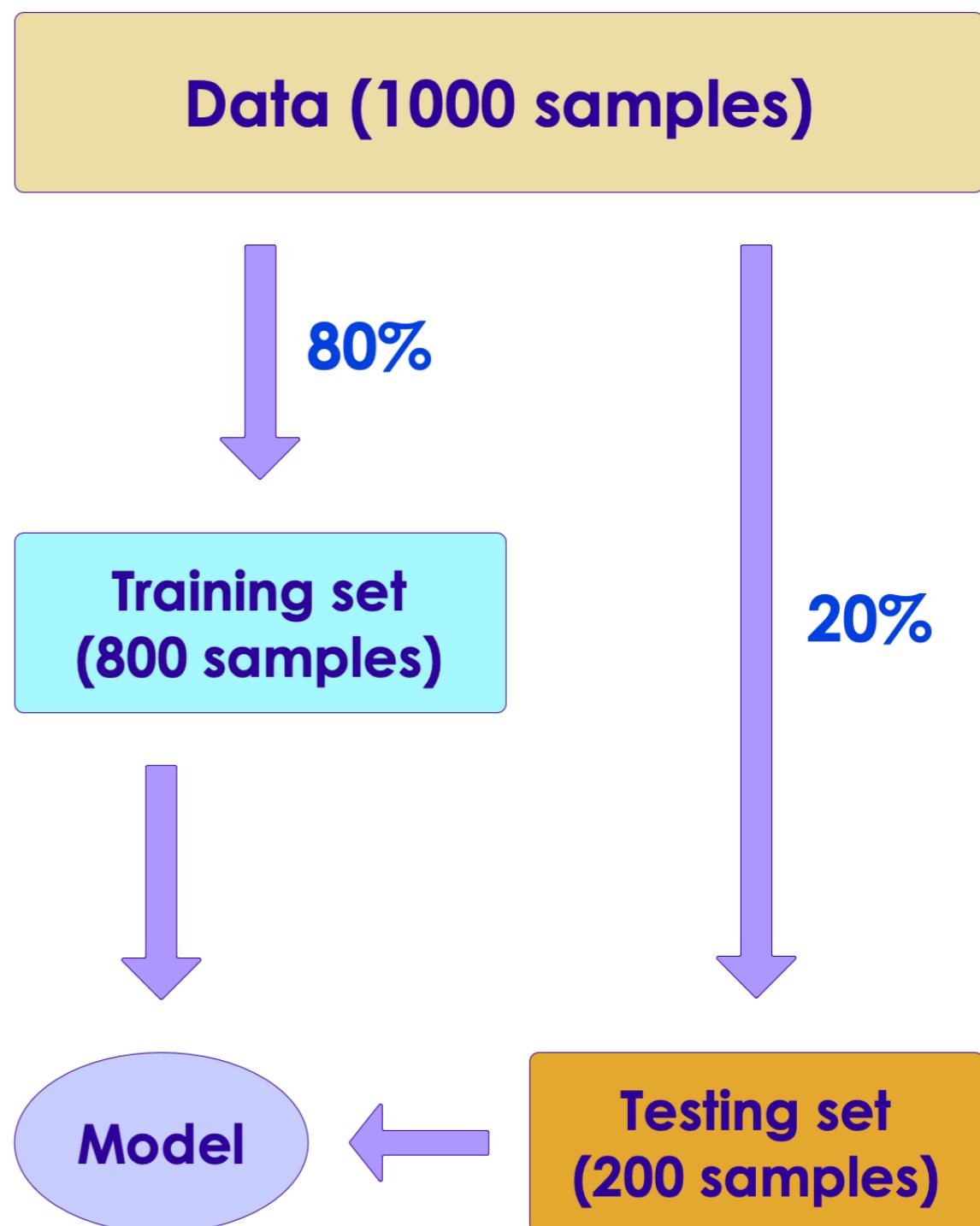
- Training/Test split is done randomly
- If we are 'lucky', we can get a well rounded training set and the model can learn well
- Also we can get an easy test set, resulting in higher than usual accuracy
- Or we could get a 'weak' training set, the model doesn't learn much;
And get a 'hard' test set, where model does badly
- So model accuracy (performance) can significantly fluctuate based on how data is divided (randomly)
- See next slide for an example



Hold Out Method Drawback

Example

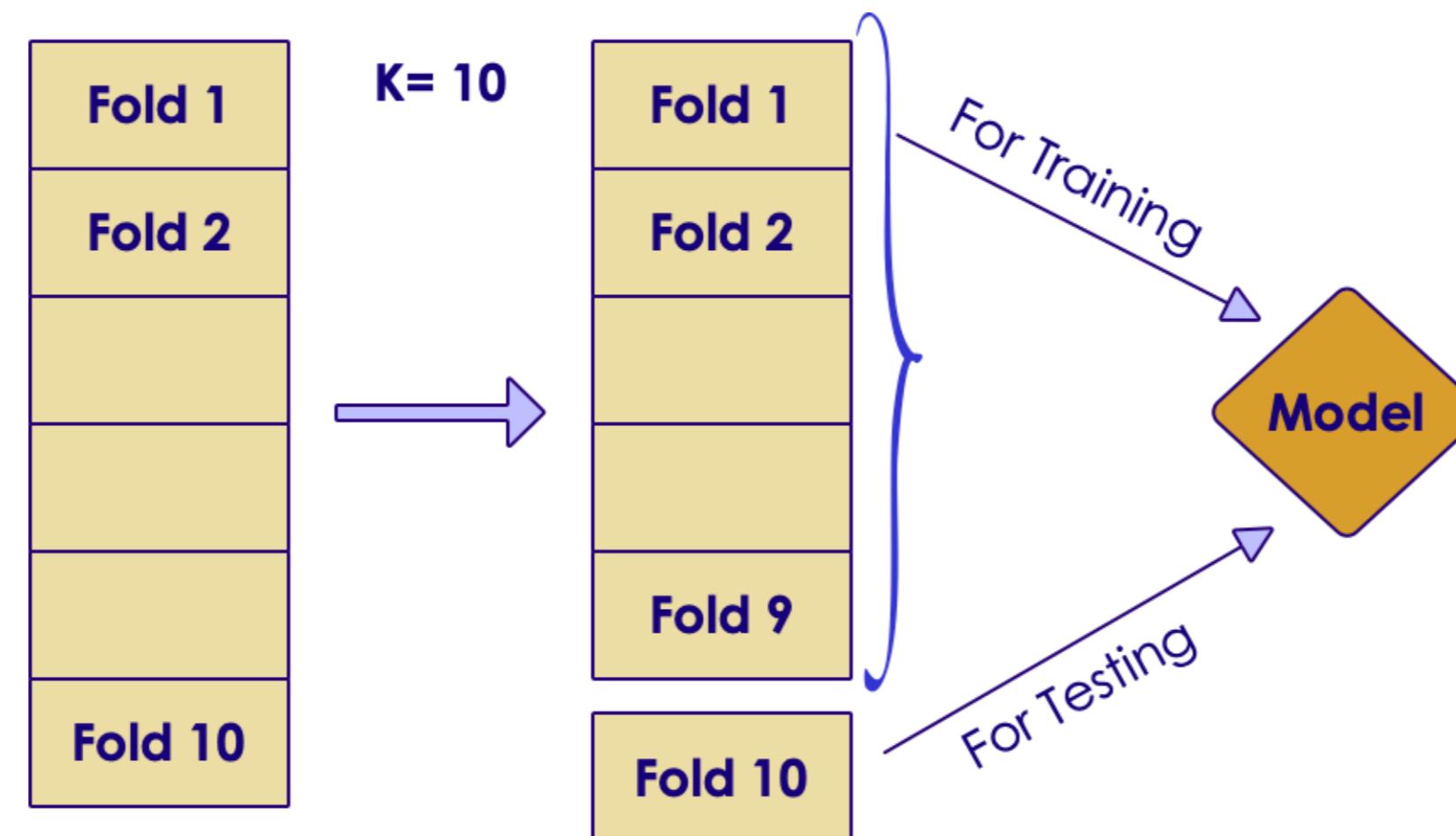
- Assume we want to test a student's knowledge in a subject
- We have a pool of 20 questions
- Out of 20, we randomly choose 15 questions And the student scores 60%
- Is this the final score? No.
 - This is just one score in a random test
- We need to do more tests and average out the score
- Solution: **k-fold Cross validation**



Cross Validation

K-Fold Cross Validation

- Divide the data into equal k sections (k folds, usually 10 to 20)
- Reserve one fold for testing (say fold-i)
- Use others folds for training
- Then test with fold-I
- After we have cycled through all k folds, prediction accuracies are aggregated and compared



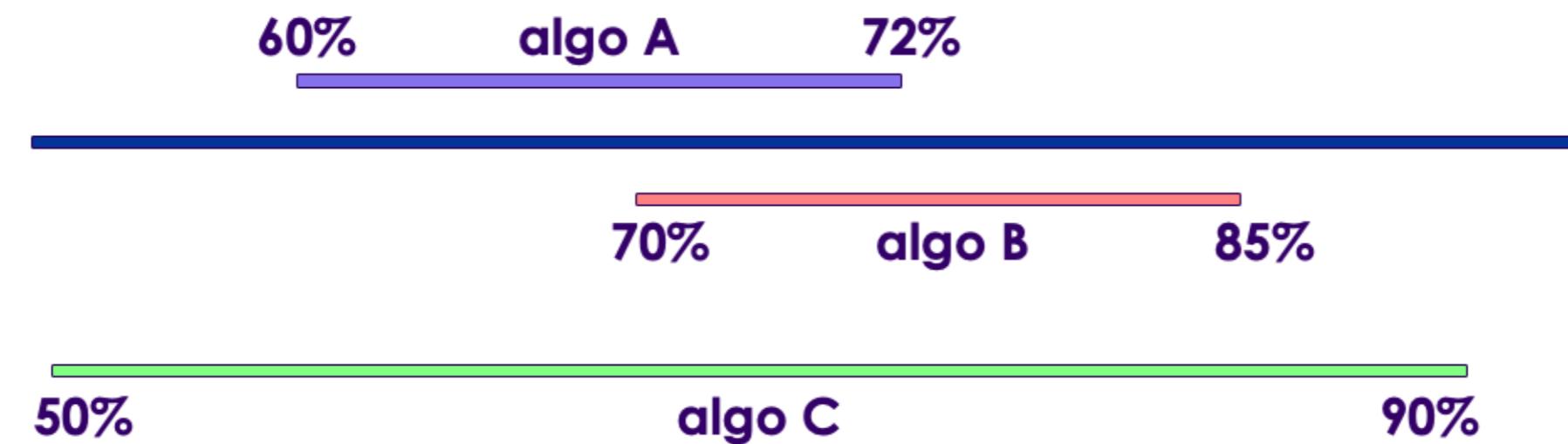
Cross-Validation Example

Runs	Splits						Model	Accuracy
Run 1	Test	Train	Train	Train	Train	Model 1	80%	
Run 2	Train	Test	Train	Train	Train	Model 2	84%	
Run 3	Train	Train	Test	Train	Train	Model 3	90%	
Run 4	Train	Train	Train	Test	Train	Model 4	86%	
Run 5	Train	Train	Train	Train	Test	Model 5	82%	

- Here we are doing a 5-fold cross validation
- Data is split into 5 splits - one held for testing, remaining 4 used for training
- Accuracy varies from 80% to 90%
- Average accuracy is $\text{AVG}(80, 84, 90, 86, 82) = 85\%$

Cross Validation

- Cross Validation is used to evaluate different algorithms
- See the following CV runs of 3 different algorithms (A,B,C)
 - Algorithm A accuracy is : 60% to 72%
 - Algorithm B accuracy is : 70% to 85%
 - Algorithm C accuracy is : 50% to 90%
- We might select algorithm B, as it seems to produce decent range
- Algorithm C is not desirable as its accuracy varies so much (high variance)



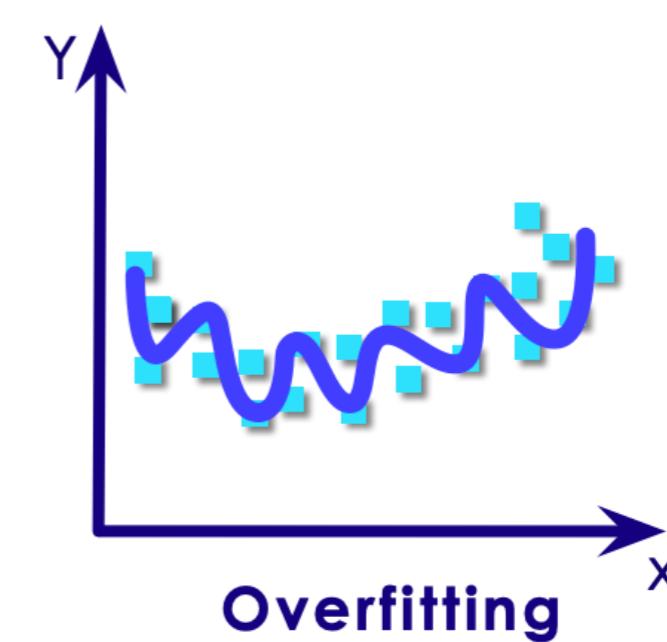
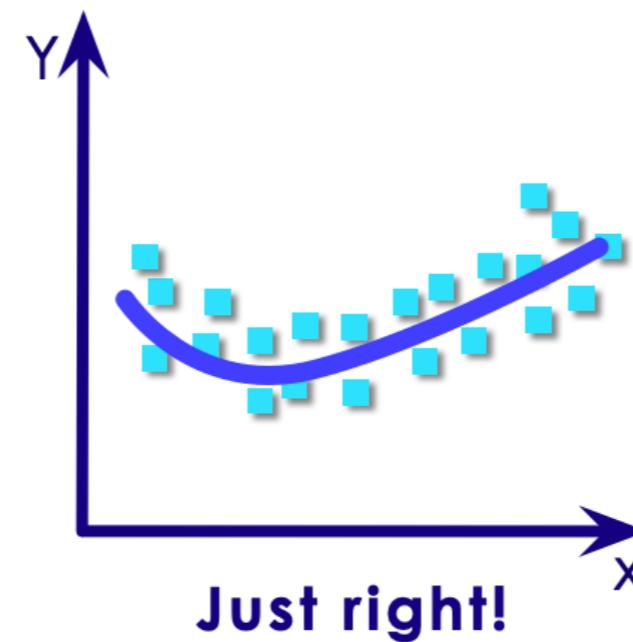
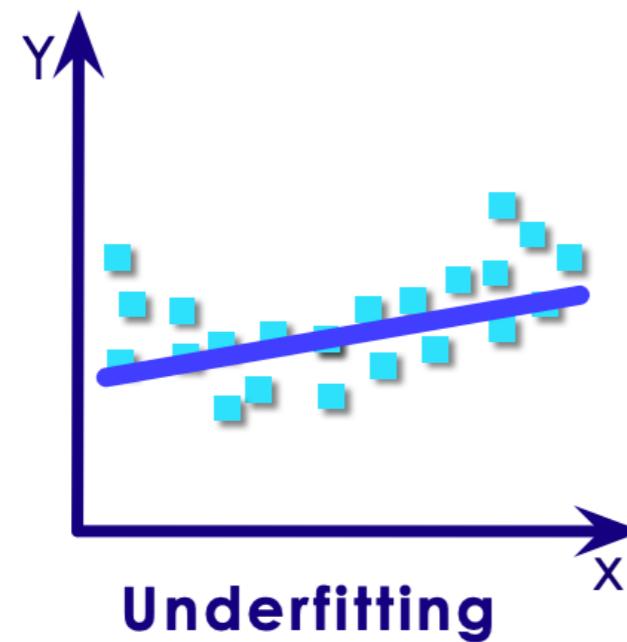
Cross Validation Takeaways

- We don't choose the 'best performing model' from CV
 - CV is used to understand a particular algorithm's performance for the given data
 - And how well it can generalize to new data
- Pros
 - Helps us systematically tests a model through the data
 - Can identify high-variance / over-fitting models
- Cons
 - Increased compute time to create multiple models and test
 - Solution: run CV tasks in parallel across multiple CPU-cores or on a cluster (embarrassingly parallelizable problem)

Underfitting / Overfitting

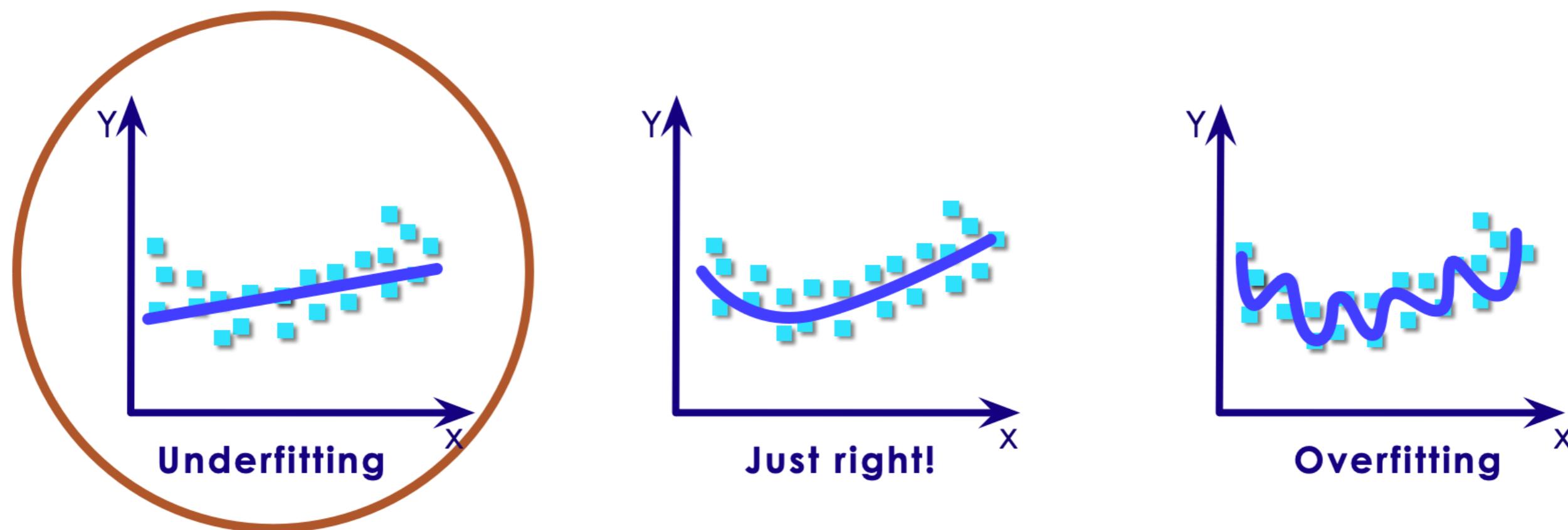
Under-fitting / Over-fitting

- Here we have 3 models
- One on left: is not really capturing the essence of the data
 - Underfitting
- One on right: following every small variation of data, not really generalizing
 - Overfitting
- One in the middle is just right



Under-fitting

- Model is 'too simple' to capture the trends in input data
- How to detect under-fitting?
 - We will get poor performance in both training & testing data
 - E.g.:
 - Training accuracy : 45%
 - Testing accuracy : 42%
- Resolution:
 - Try a different algorithm / model, that better fits the data



Overfitting

- Imagine we are teaching addition to young kids
- Rather than learning the 'carry concept' what if the kids just memorized all the examples
- They can do well in problems, they have seen before, but not any thing new
 - will do well: $(17+25)$, $(28+37)$
 - will not so well (they haven't seen it before): $(45+48)$
- This is basically **overfitting**
- When model is **memorizing** training data instead of learning from it.

Learning addition

$$\begin{array}{r} 17 \\ + 25 \\ \hline 2 \end{array} \qquad \begin{array}{r} 28 \\ + 37 \\ \hline 5 \end{array}$$

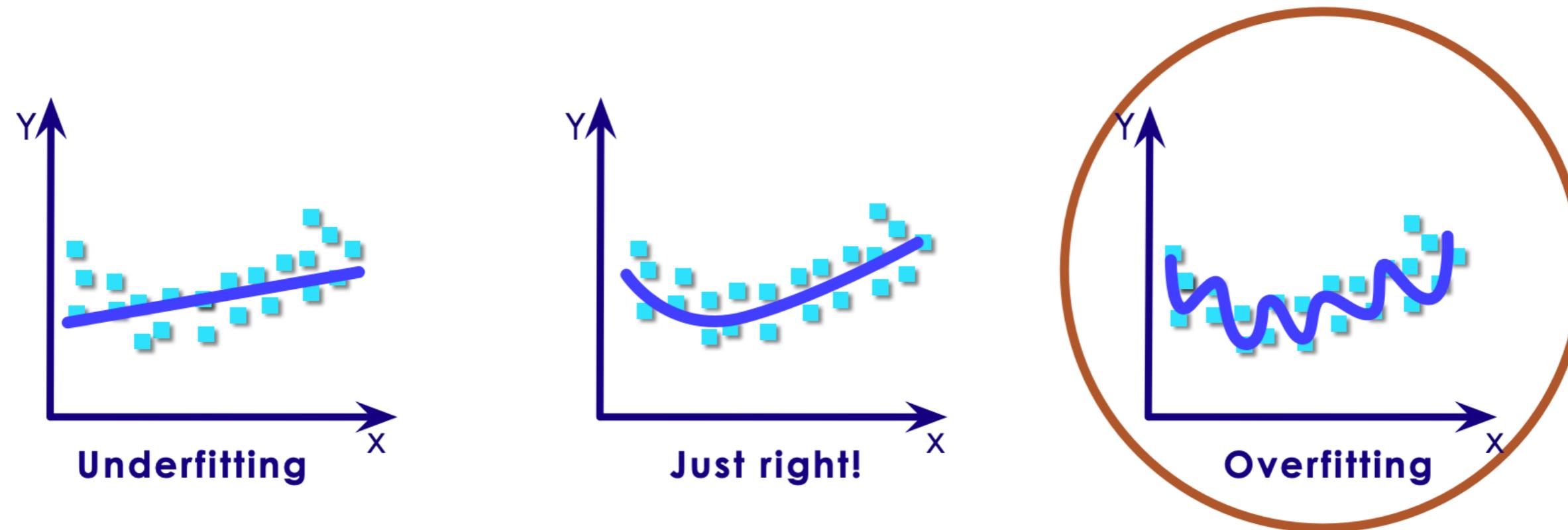
Carry 1 **Carry 1**

$1 + 2 + 1 = 4$ **$2 + 3 + 1 = 6$**

Final = 42 **Final = 65**

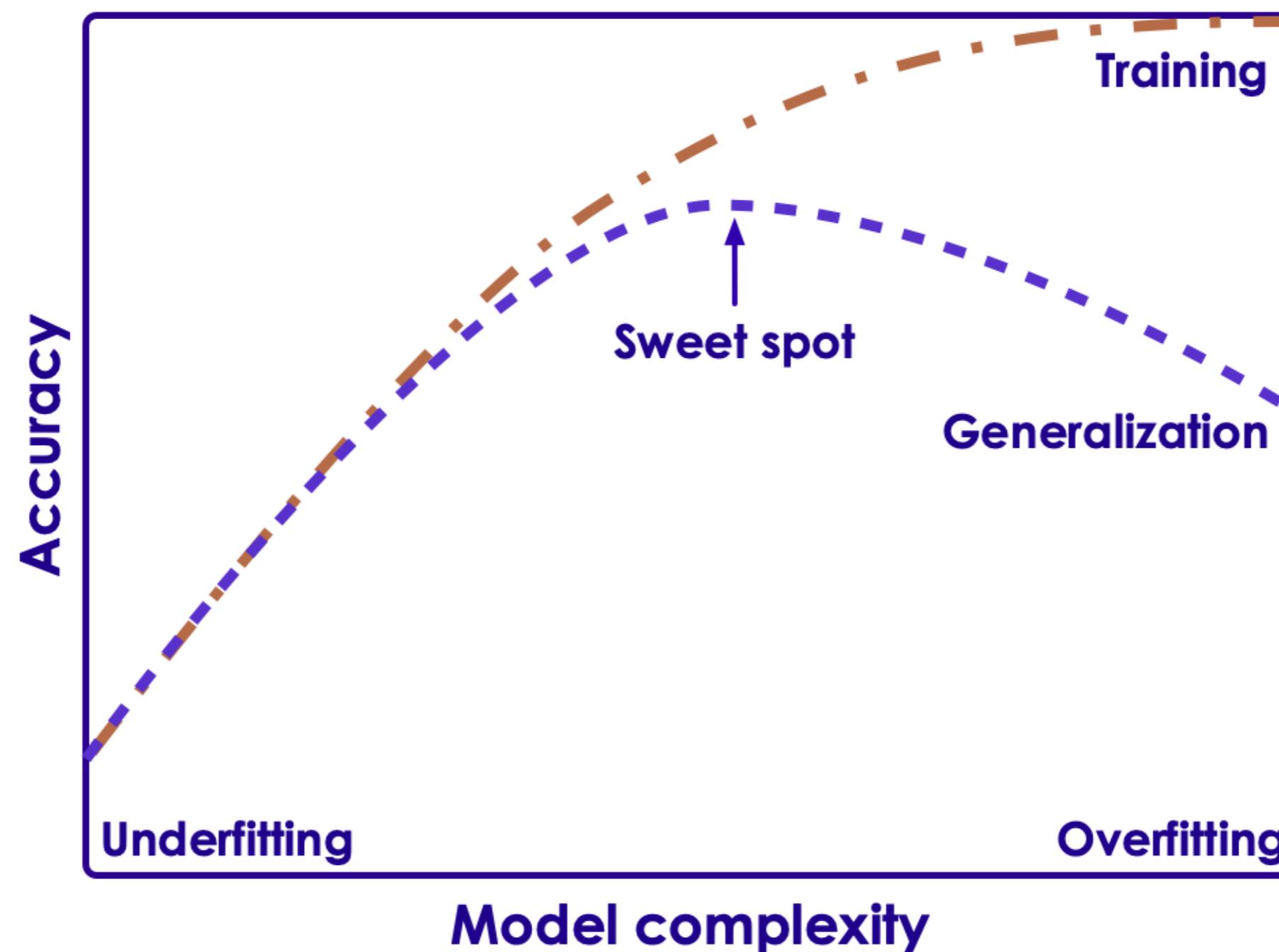
Over-fitting

- How to detect over-fitting?
 - Excellent performance on training data, but poor performance on testing (new) data
 - E.g. :Training accuracy : 95%
 - Testing accuracy : 62%
- Resolution:
 - Try a different algorithm / model, that better fits the data
 - Simplify inputs



Achieving a Good Fit

- In ML we strive to find the 'sweet spot' between under-fitting models and over-fitting models



Achieving a Good Fit

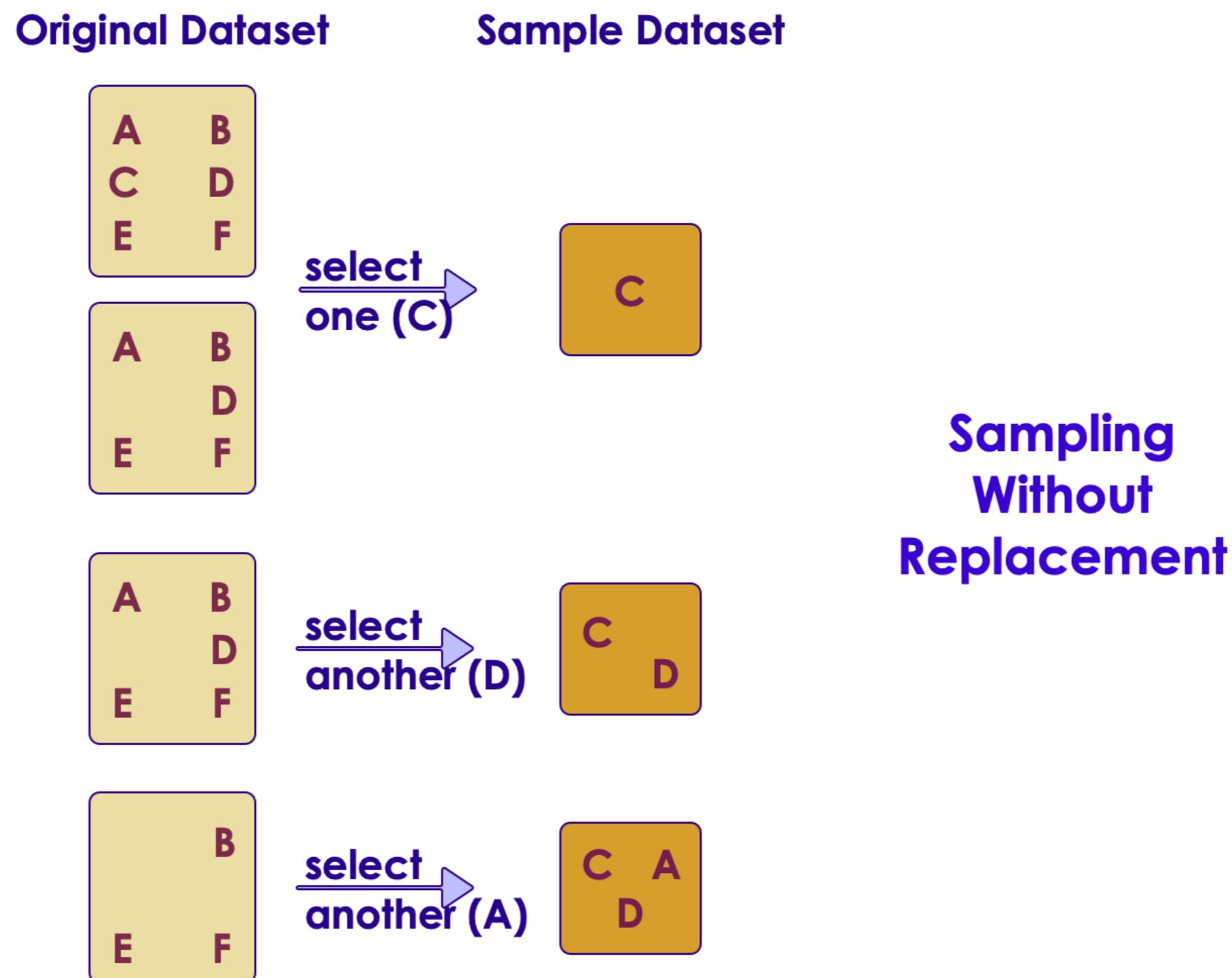
- Both overfitting and underfitting can lead to poor model performance
- underfitting is easier to spot
 - Bad performance on training data
 - Bad performance on test data
- Overfitting can be hard to spot
 - because it performs well on training data
 - But doesn't do well on 'unseen' test data
- Avoiding overfitting
 - Resampling technique
 - Hold back a validation dataset
 - Most popular method is: k-fold validation (more on this later)
- For **decision trees**, we **prune** the tree to limit overfitting

Bootstrapping

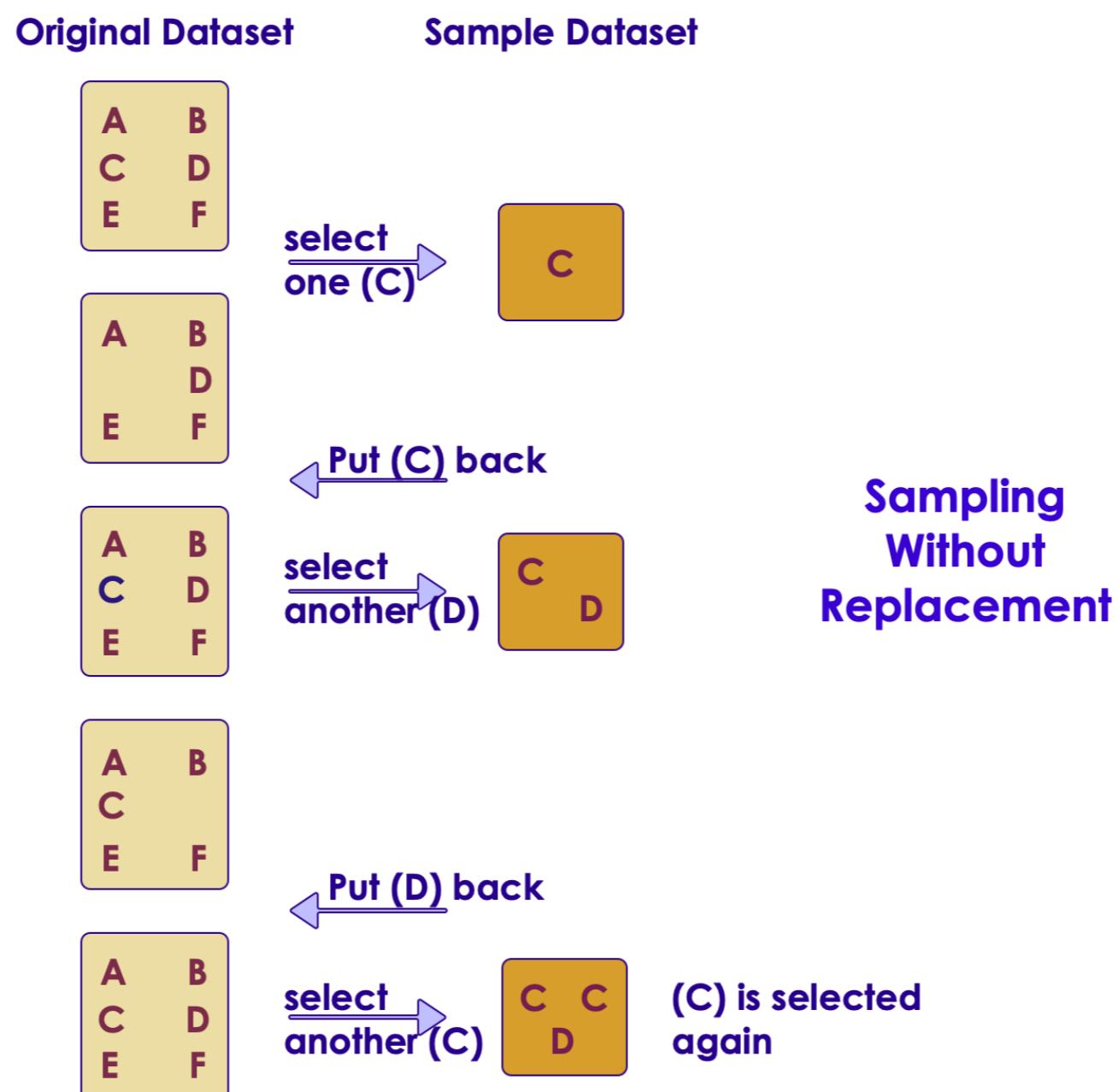
Bootstrapping

- Randomly selecting data for training with replacement
- Data points: [a, b, c, d, e]
 - Bootstrap selection 1: [b, d, d, c]
 - Bootstrap selection 2: [d, a, d, a]
- It may seem counter-intuitive to draw the same data again and again
- But in some scenarios, bootstrapping really helps to train the model
- See next slides to understand sampling with and without replacement

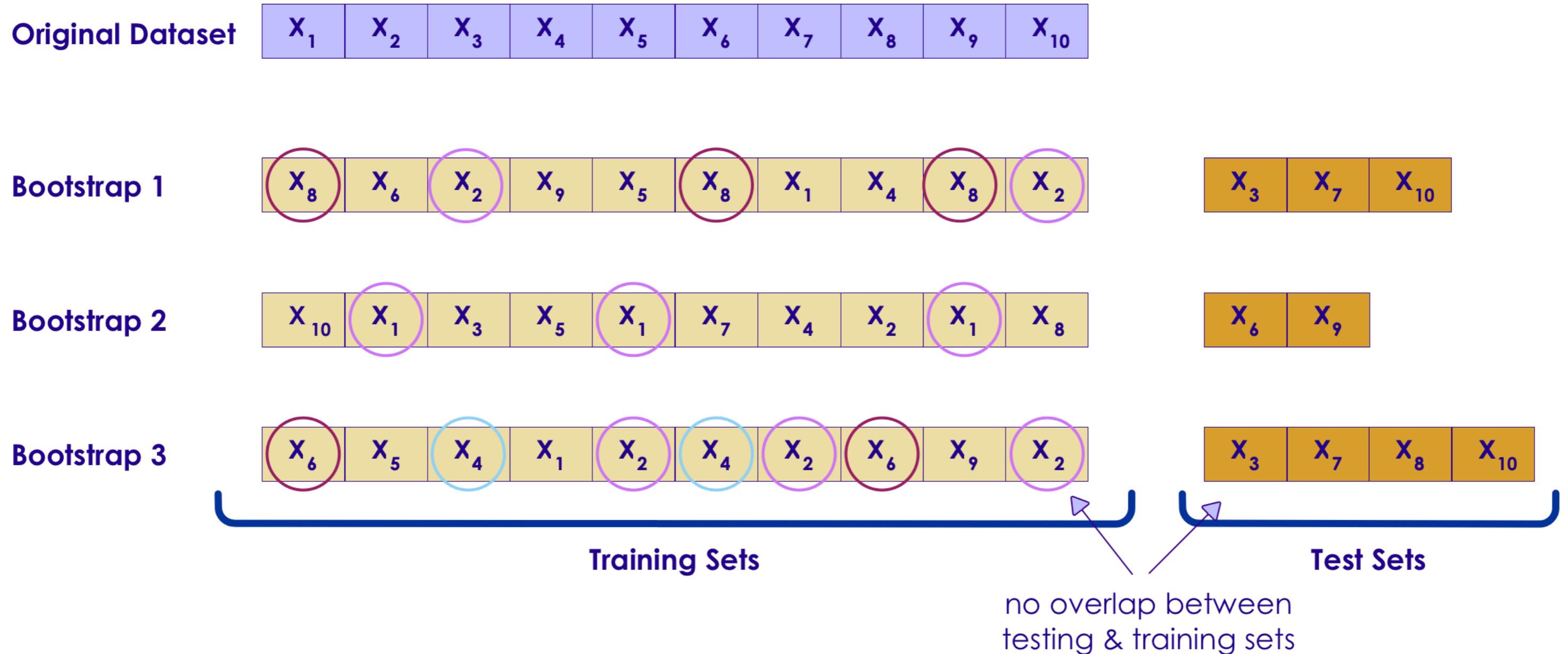
Sampling Without Replacement



Sampling With Replacement (aka Bootstrapping)



Bootstrapping Example 2



End Part 1

- To Instructor:
Pause here, and go to next section

ML Concepts: Part 2

- To Instructor:
These sub sections are included in other sections

Errors and Loss Functions

Error/Loss Functions for Regressions

Error / Loss Function

- The function that is used to compute the error is known as **Loss Function J()**
- Different loss functions will calculate different values for the same prediction errors
- In next few slides, we are going to examine some of the loss functions

Estimating Tips for Meals

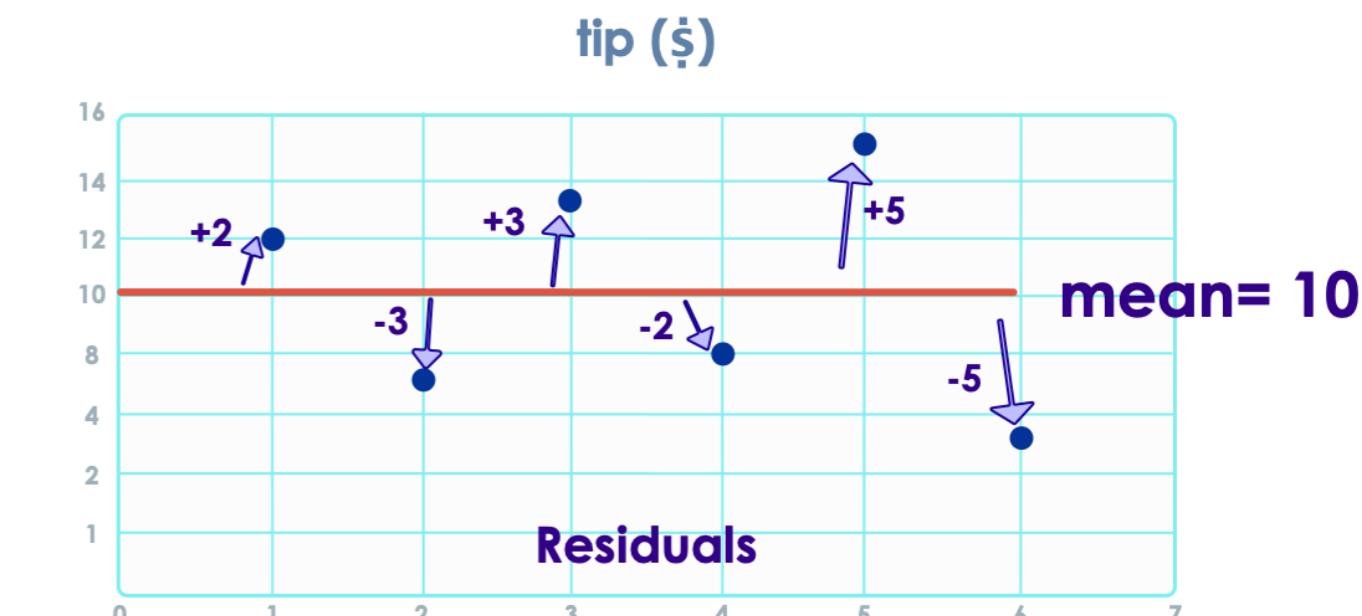
- Let's consider tips at a restaurant

Meal #	Tip (\$)
1	12
2	7
3	13
4	8
5	15
7	5

Understanding Residuals / Errors

- Let's say, our *very naive* model **always predicts tip as \$10 :-)**
- From the table, we can see none of the tip amounts are exactly \$10
 - This difference (delta) is called **Error or Residual**
 - Residual = actual tip - predicted tip**
- Sum of all residuals = **ZERO** (positive and negative errors are canceling each other)

Actual Tip	Predicted Tip	Error or Residual = (Actual - Predicted)
12	10	+2 = (12 - 10)
7	10	-3 = (7 - 10)
13	10	+3 = (13 - 10)
8	10	-2 = (8 - 10)
15	10	+5 = (15 - 10)
5	10	-5 = (5 - 10)
		SUM = 0 (+2 -3 +3 -2 +5 -5)



Sum of Squared Errors (SSE)

- From the previous table, errors can cancel each other out
- Let's square the error:
 - To make them all positive (so negative and positive don't cancel each other out)
 - To amplify 'outliers' (large deviations)
- **Question for the class: Can SSE be zero? :-)**

Actual Tip	Predicted Tip	Error = (Actual - Predicted)	Error Squared
12	10	+2 = (12 - 10)	4
7	10	-3 = (7 - 10)	9
13	10	+3 = (13 - 10)	9
8	10	-2 = (8 - 10)	4
15	10	+5 = (15 - 10)	25
5	10	-5 = (5 - 10)	25
	Total ==>	0	76

Sum of Squared Errors (SSE)

- Also known as
 - **Residual Sum of Squares (RSS)**
 - **Sum of Squared Residuals (SSR)**
- In this formula
 - y_i : actual value
 - \hat{y}_i : predicted value
- Properties
 - A good all purpose error metric that is widely used
 - SSE also 'amplifies' the outliers (because of squaring)
- For example, if SSE for model-A = 75 and SSE for model-B = 50
 - Model-B might be better fit

$$SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Mean Squared Error (MSE) (L2)

Actual Tip	Predicted Tip	Error = (Actual - Predicted)	Error Squared
12	10	+2 = (12 - 10)	4
7	10	-3 = (7 - 10)	9
13	10	+3 = (13 - 10)	9
8	10	-2 = (8 - 10)	4
15	10	+5 = (15 - 10)	25
5	10	-5 = (5 - 10)	25
Total ==>		0	76

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

■ $MSE = (4 + 9 + 9 + 4 + 25 + 25)/6 = 76 / 6 = 12.6$

■ Properties

- Can be sensitive to outliers; predictions that deviate a lot from actual values are penalized heavily
- Easy to calculate gradients (fast)

Mean Absolute Error (MAE)

Actual Tip	Predicted Tip	Error = (Actual - Predicted)	Absolute Error	Error Squared
12	10	+2 = (12 - 10)	2	4
7	10	-3 = (7 - 10)	3	9
13	10	+3 = (13 - 10)	3	9
8	10	-2 = (8 - 10)	2	4
15	10	+5 = (15 - 10)	5	25
5	10	-5 = (5 - 10)	5	25
Total ==>		0	20	76

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

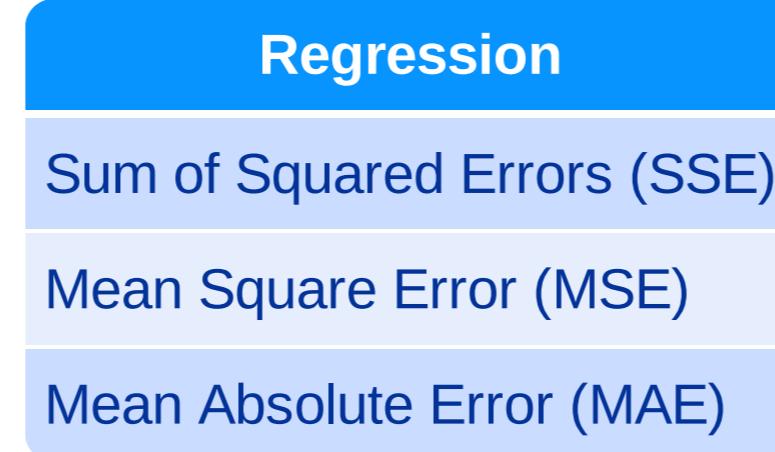
- MAE = $(2 + 3 + 3 + 2 + 5 + 5) = 20 / 6 = 3.33$

- Properties:

- More robust and is generally not affected by outliers
- Use if 'outliers' are considered 'corrupt data' (or not critical part of data)

Regression Error Functions - Summary

- Error functions tell us 'how far off' our prediction from actual value is.
- We have seen 3 popular error functions for regression
- Which one to use?
 - No 'hard' rules!, follow some practical guide lines
 - Try them all and see which one gives better results! :-)
(most ML libraries allow us to configure the error function very easily)



Error/Loss Functions for Classifications

Loss Functions for Classification

- Binary Class Entropy
- Categorical Crossentropy / Sparse Categorical Crossentropy
- Negative Log Likelihood
- Margin Classifier
- Soft Margin Classifier

Binary Classifications: Binary Class Entropy

- Cross Entropy is used in binary classification scenarios (0 / 1)
- Measures the divergence of probability distributions between actual and predicted values

Income (input 1)	Credit Score (input 2)	Current Debt (input 3)	Loan Approved (output)
40,000	620	0	0
80,000	750	100,000	1
100,000	800	50,000	1

$$E = -\frac{1}{n} \sum_{i=1}^n [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Multi Class Classifications: Sparse Categorical Crossentropy

- Here we predict one of many labels (1,2,3)
- Our labels are integers
- Choice: **sparse_categorical_crossentropy**

a	b	c	d	label
6.4	2.8	5.6	2.2	1
5.0	2.3	3.3	1.0	2
4.9	3.1	1.5	0.1	3

Multi Class Classifications: Categorical Crossentropy

- Here we predict one of many labels (1,2,3)
- Our labels are **one-hot-encoded**
- Choice: **categorical_crossentropy**

a	b	c	d	label
6.4	2.8	5.6	2.2	[1,0,0]
5.0	2.3	3.3	1.0	[0,1,0]
4.9	3.1	1.5	0.1	[0,0,1]

Summary of Errors / Loss Functions

Regression	Classification	Embedding
Sum of Squared Errors (SSE)	Binary Class Entropy	Cosine Error
Mean Square Error (MSE)	Categorical Crossentropy	L1 Hinge Error
Mean Absolute Error (MAE)	Margin Classifier	
	Soft Margin Classifier	
	Negative Log Likelihood	

Loss Functions Based on Task

Problem Type	Prediction	Loss Function
Regression	a number	mse, sse, mae
Classification	binary (0/1)	binary_crossentropy

Cosine Similarity

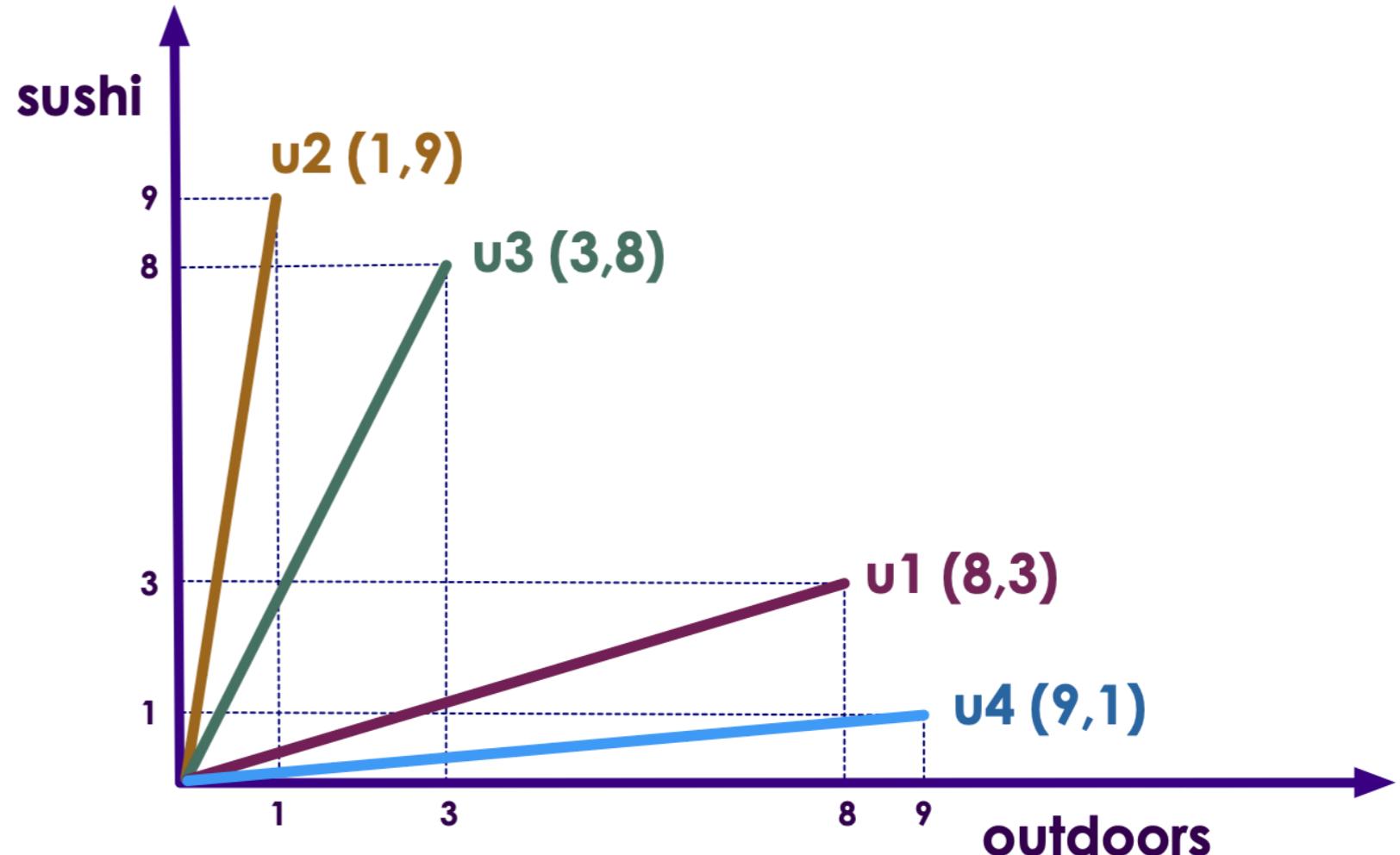
Use Case: Find Similar Users in a Dating Website

User	Likes Outdoors	Likes Sushi
u1	8	3
u2	1	9
u3	3	8
u4	9	1

- Users are rating each item on a 1-10 scale (1 being least, 10 being most)
- We can represent these as vectors
 - u1: [8,3]
 - u2: [1,9]
- **Question for class:**
 - Looking at the matrix, which users **have similar tastes** ?

Finding Similar Users

- Now let's plot our users' interest
 - X-axis: outdoors
 - Y-axis: sushi
- It is much easier to identify users with 'similar tastes' from this plot
 - u1 and u4 have very similar tastes
 - u2 and u3 have very similar tastes
- Cosine similarity** measures how close the vectors are to each other (the angle between them)



Finding Similar Users: Adding More Dimensions

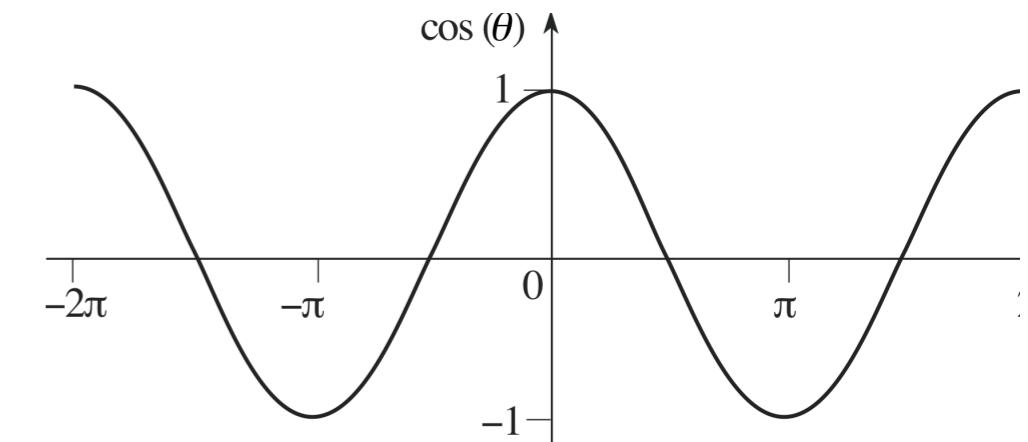
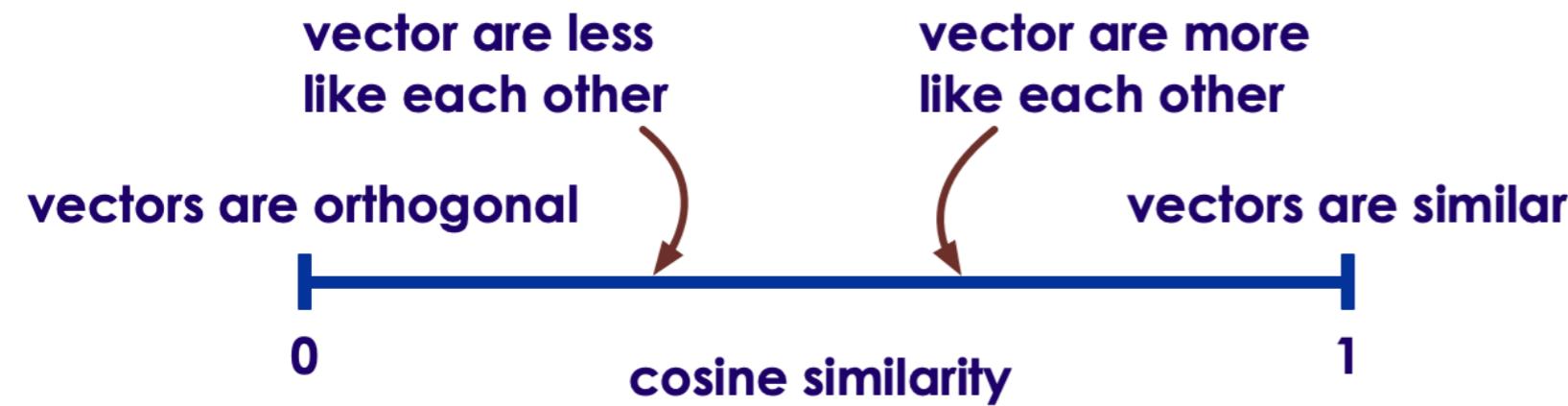
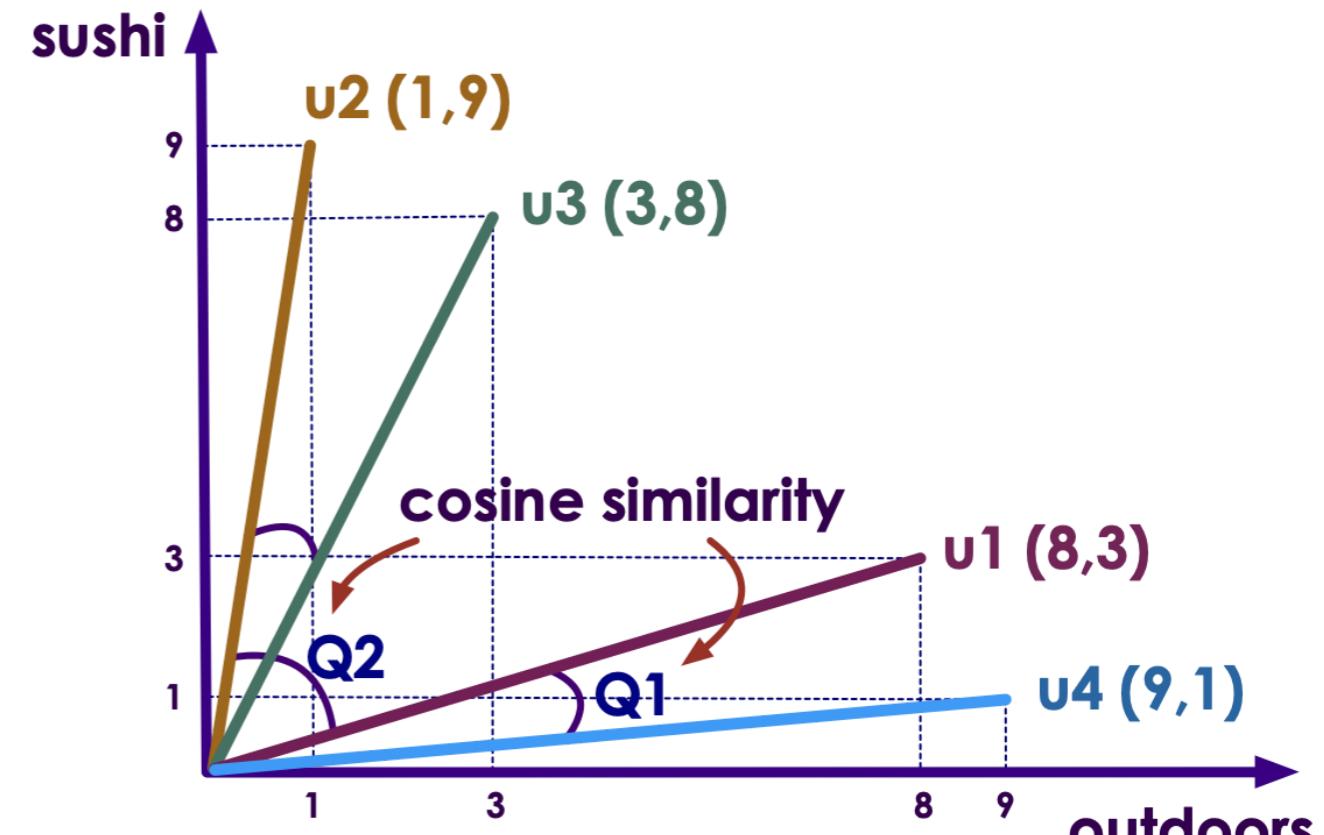
- Now let's add more attributes to our user profiles

User	Likes Outdoors	Likes Sushi	Likes Cats	Likes to Read	Likes Movies	Likes Rock Music	Likes XYZ
u1	8	3	10	8	4	3	7
u2	1	9	1	9	9	8	6
u3	3	8	5	7	1	1	1
u4	9	1	8	1	10	8	9

- Now our vectors have more dimension than 2
 - u1: [8,3,10,8,4,3,7]
 - u2: [1,9,1,9,9,8,6]
- Question for class:** Can we visualize this?
 - More than 3 dimensions is hard to visualize
- We need a way to measure 'similarity' of vectors in regardless of dimensions
 - cosine similarity**

Cosine Similarity

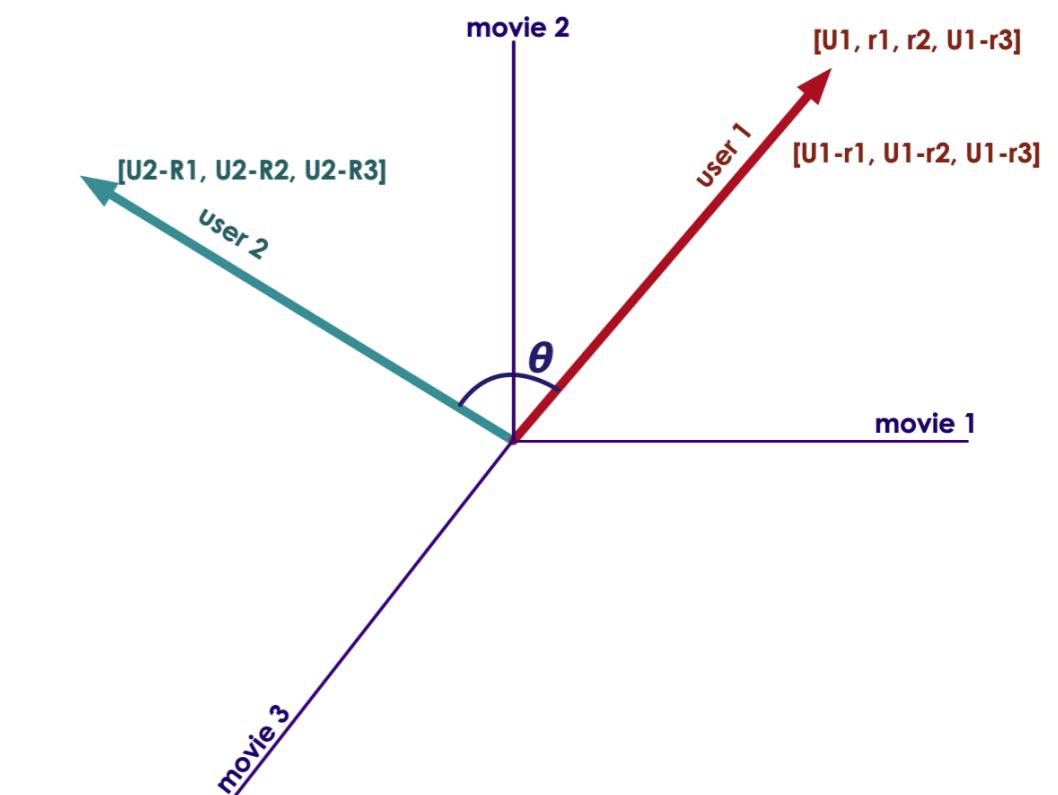
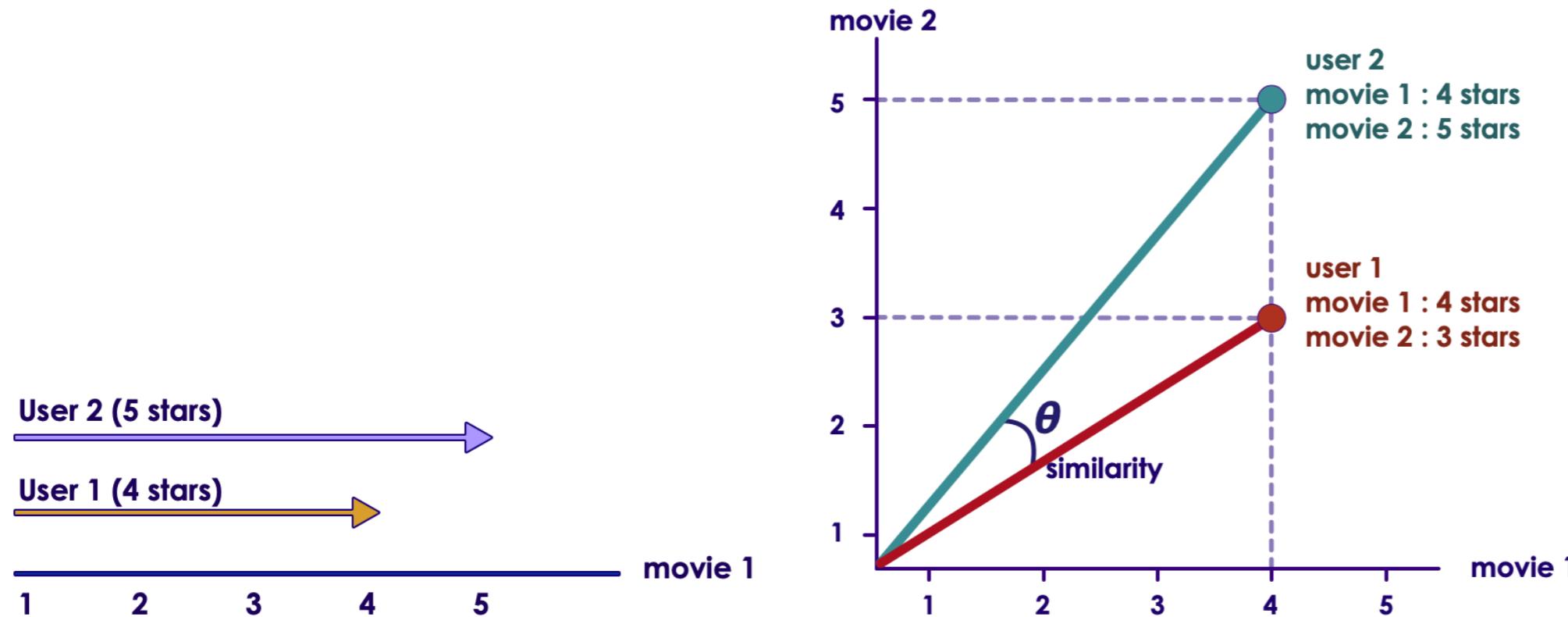
- **Cosine Similarity** measures 'angle' between vectors
- Even though cosine values are between -1 and +1, cosine similarity is normalized between 0 and 1
 - 1: vectors are perfect alignment
 - 0: vectors are orthogonal (not like each other)
- So here an example would be:
 - $\text{cosine_similarity}(u_1, u_4) = 0.85$ (very close)
 - $\text{cosine_similarity}(u_1, u_2) = 0.1$ (not close)



Cosine Similarity for Ratings

- Here we are representing ratings as vectors
- (Left) Start with ratings for 'movie-1'
- (Middle) Add ratings for 'movie-2'
- (Right) Generalize it to any number of movies

User	m1	m2	m3
u1	4	3	5
u2	4	5	5



Finding Similar Users Using Cosine Similarity

User	Likes Outdoors	Likes Sushi	Likes Cats	Likes to Read	Likes Movies	Likes Rock Music	Likes XYZ
u1	8	3	10	8	4	3	7
u2	1	9	1	9	9	8	6
u3	3	8	5	7	1	1	1
u4	9	1	8	1	10	8	9

- In this data, once we represent each user's interest as a vector, we can use cosine similarity to find similar tastes
- cosine_similarity(u1, others) gives us closest matches
 - u4 (0.8): closest
 - u3 (0.5)
 - u2 (0.1): farthest

Cosine Proximity

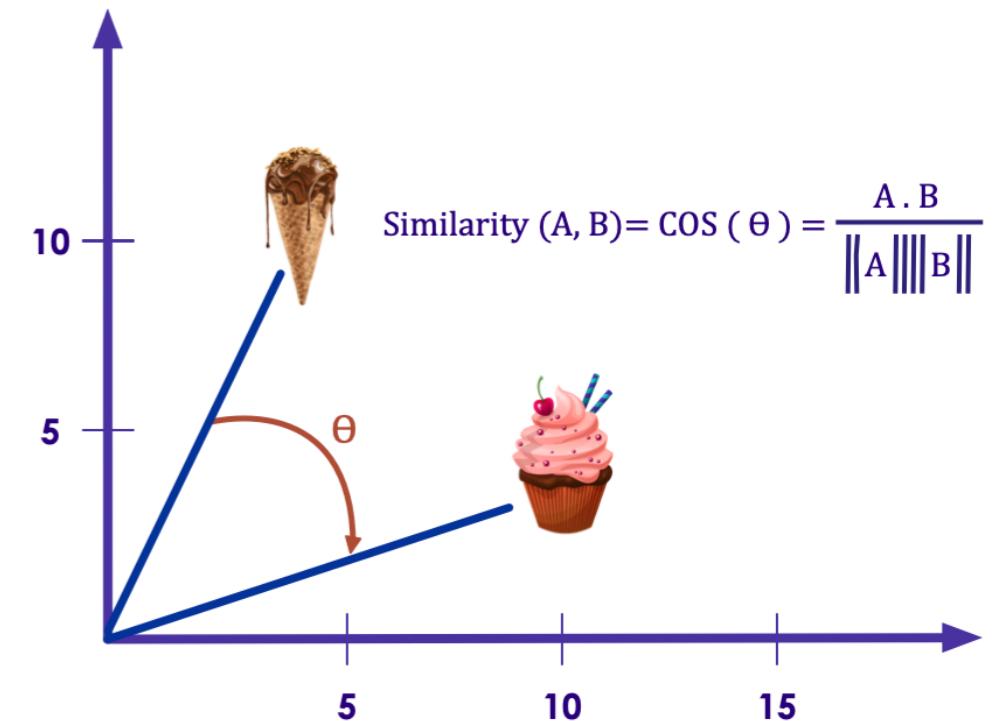
- Computes the cosine proximity between predicted value and actual value
- Based on Cosine similarity
 - Vectors are 'similar' if they are parallel
 - Vectors are 'not similar' if they are perpendicular / orthogonal

$$\mathbf{y}(\text{actual}) = \{y^{(1)}, y^{(2)}, \dots, y^{(n)}\} \in \mathbb{R}^n$$

$$\hat{\mathbf{y}}(\text{predicted}) = \{\hat{y}^{(1)}, \hat{y}^{(2)}, \dots, \hat{y}^{(n)}\} \in \mathbb{R}^n$$

$$E = -\frac{\mathbf{y} \cdot \hat{\mathbf{y}}}{\|\mathbf{y}\|_2 \cdot \|\hat{\mathbf{y}}\|_2} = -\frac{\sum_{i=1}^n y^{(i)} \cdot \hat{y}^{(i)}}{\sqrt{\sum_{i=1}^n (y^{(i)})^2} \cdot \sqrt{\sum_{i=1}^n (\hat{y}^{(i)})^2}}$$

Cosine Similarity



Summary of Errors / Loss Functions

Summary of Errors / Loss Functions

Regression	Classification	Embedding
Sum of Squared Errors (SSE)	Binary Class Entropy	Cosine Error
Mean Square Error (MSE)	Categorical Crossentropy	L1 Hinge Error
Mean Absolute Error (MAE)	Margin Classifier	
	Soft Margin Classifier	
	Negative Log Likelihood	

Loss Functions: Resources

- <https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0>
- <https://rishy.github.io/ml/2015/07/28/l1-vs-l2-loss/>
- https://isaacchanghau.github.io/post/loss_functions/
- https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
- <https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23>

Backup Slides

Negative Logarithmic Likelihood

- Used when model outputs probability of each class
(digit-1 : 10%, digit-9 : 90% ..etc)

$$E = -\frac{1}{n} \sum_{i=1}^n \log(\hat{y}^{(i)})$$

Poisson Loss Function

- Derived from Poisson distribution which is used for counting data

$$E = \frac{1}{n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)} \cdot \log(\hat{y}^{(i)}))$$

Hinge Loss / Max Margin Loss

- From Support Vector Machines (SVM)
- For binary output

$$E = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y^{(i)} \cdot \hat{y}^{(i)})$$

- For multi-class classifier

$$E = \frac{1}{n} \sum_{i=1}^n \max(0, m - y^{(i)} \cdot \hat{y}^{(i)})$$

Bias Variance Trade-off

Estimating Target Function

- In supervised algorithms try to estimate target function 'f'

$$Y = f(X)$$

Y: output, X: input

- The error can be broken down to

- Bias error
- Variance error
- Irreducible error

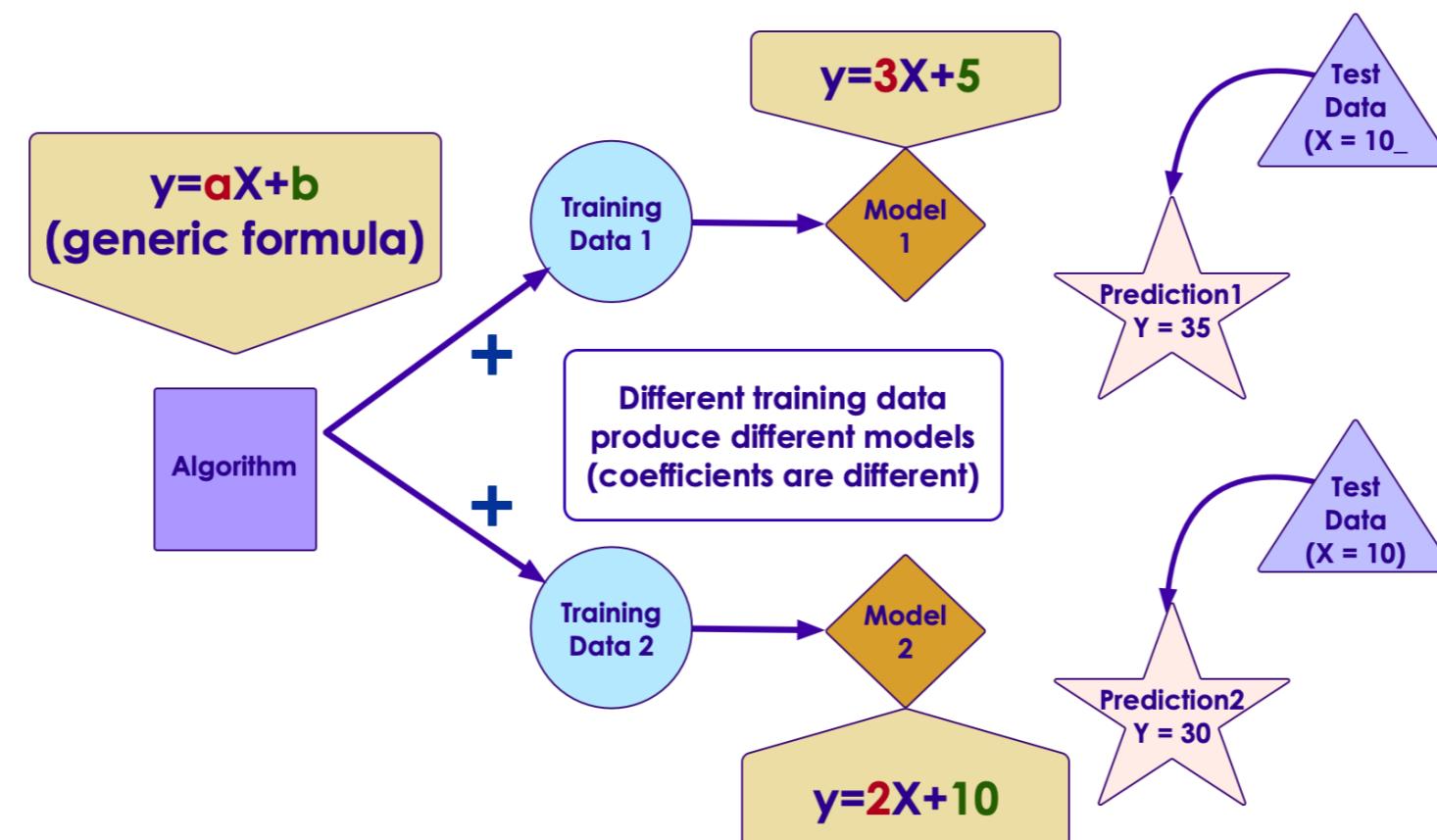
- Irreducible error can not be minimized. May be caused by unknown variables, noise ..etc

Bias Error

- Bias are the **simplifying assumptions** made by a model to make target function easier to learn
- Generally **parametric algorithms have high bias**
 - Fast learning
 - Easier to understand
 - But less flexible
 - Lower predicting performance on complex data (That doesn't fit the simplifying assumptions of algorithm)
- **Low Bias (good!):**
 - Less assumptions made about target function
 - E.g. Decision Trees, KNN, SVM
- **High Bias (not good):**
 - more assumptions of target function
 - E.g. Linear regression, Logistic regression

Variance

- Target function is estimated from training data
 - So it is influenced by training data
- **Variance is the amount that the estimate of the target function will change if different training data was used**
- Ideally target function should not change drastically from one training set to next
 - meaning that the algorithm is good at picking out the hidden underlying mapping between the inputs and the output variables



Variance

- **Low Variance:**

- Suggests small changes to the estimate of the target function with changes to the training dataset
- E.g. parametric algorithms: Linear Regression, Logistic Regression

- **High Variance:**

- Suggests large changes to the estimate of the target function with changes to the training dataset.
- Generally nonparametric machine learning algorithms that have a lot of flexibility have a high variance
- E.g. decision trees have a high variance, that is even higher if the trees are not pruned before use

Bias - Variance Tradeoff

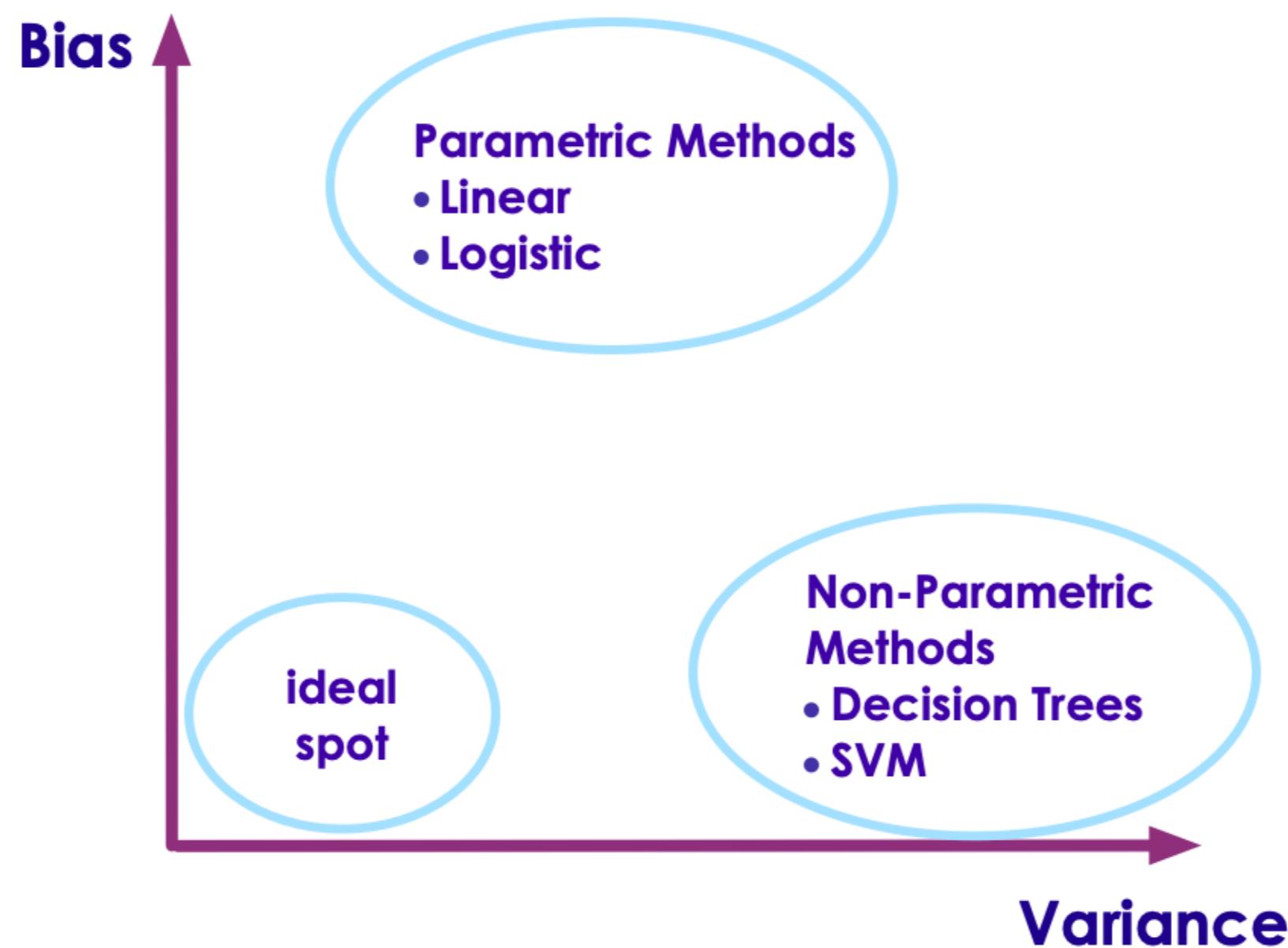
- Goal of supervised algorithm is to achieve **low bias and low variance**
- Low bias: less assumptions of target function form --> more flexibility
- Low variance: less swings in target function for changes in training data --> stable algorithm
- **Parametric or linear algorithms** often have a high bias but a low variance
- **Nonparametric or nonlinear algorithms** often have a low bias but a high variance
- There is no escaping the relationship between bias and variance in machine learning
 - Increasing the bias will decrease the variance
 - Increasing the variance will decrease the bias
 - Bias-Variance can be adjusted for particular algorithms

Bias-Variance Tradeoff

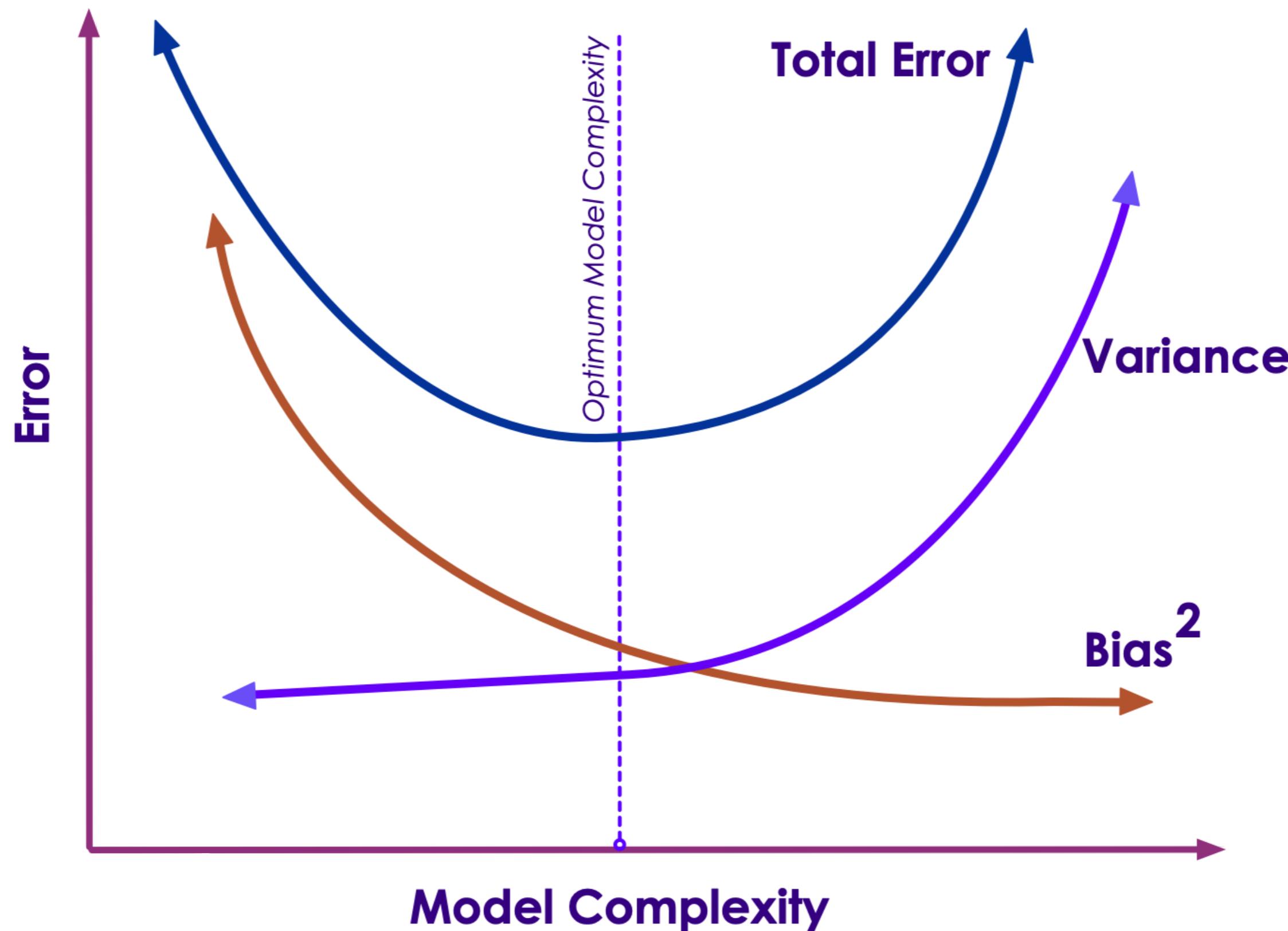
Low Bias (good)	High Bias,(not good)
Decision Trees, k-Nearest Neighbors and Support Vector Machines	Linear Regression, Linear Discriminant Analysis and Logistic Regression
More able to adopt to complex data	May not be able to adopt to complex data

Low Variance (good)	High Variance (not good)
Modestly influenced by change of data	Strongly influenced by change of data
Parametric methods usually have low variance	nonparametric machine learning algorithms that have a lot of flexibility have a high variance
Linear Regression, Linear Discriminant Analysis and Logistic Regression	Decision Trees, k-Nearest Neighbors and Support Vector Machines.

Bias Variance Trade Off



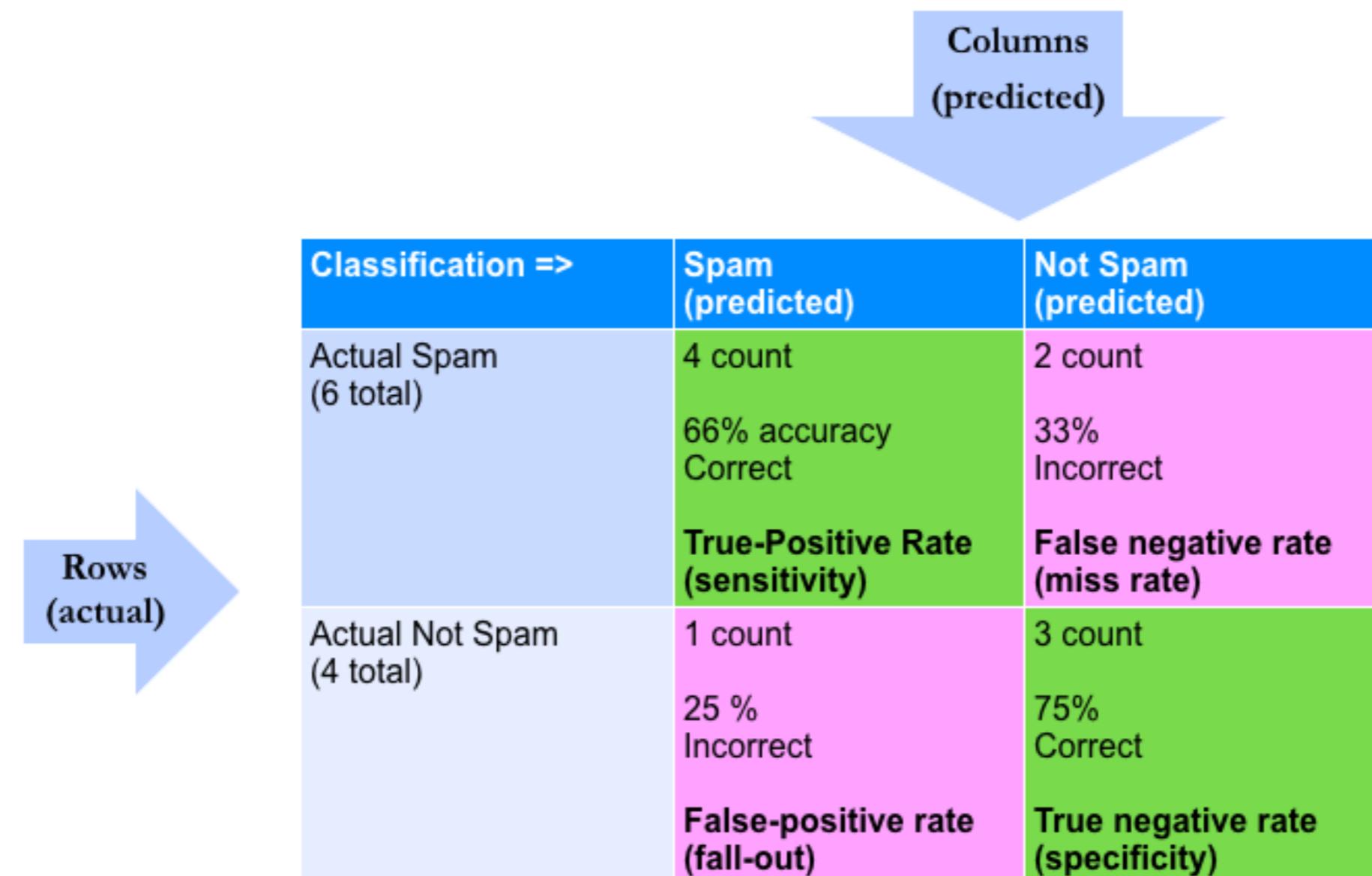
Bias-Variance Trade Off



Confusion Matrix and ROC Curve

Confusion Matrix / Error Matrix

- Let's consider a binary classifier
 - Picks one of two outcomes (spam / not-spam)
- Say we are classifying 10 emails (6 spam, 4 not-spam)



The diagram illustrates a confusion matrix for a binary classifier. A blue arrow labeled "Rows (actual)" points to the left side of the matrix, which lists "Actual Spam (6 total)" and "Actual Not Spam (4 total)". Another blue arrow labeled "Columns (predicted)" points down to the top of the matrix, which lists "Spam (predicted)" and "Not Spam (predicted)". The matrix itself is a 2x2 grid with the following data:

Classification =>	Spam (predicted)	Not Spam (predicted)
Actual Spam (6 total)	4 count 66% accuracy Correct True-Positive Rate (sensitivity)	2 count 33% Incorrect False negative rate (miss rate)
Actual Not Spam (4 total)	1 count 25 % Incorrect False-positive rate (fall-out)	3 count 75% Correct True negative rate (specificity)

Class Quiz: A Perfect Confusion Matrix

- What will a perfect confusion matrix will look like?
- Here we have two classes: A & B
- And we have 10 As and 20 Bs
- What will the confusion matrix will look like?
 - Answer next slide!



	Predicted A	Predicted B
Actual A (10)	?	?
Actual B (20)	?	?

Perfect Confusion Matrix

	Predicted A	Predicted B
Actual A (10)	10	0
Actual B (20)	0	20

Confusion Matrix: More Than 2 Outcomes

		Predicted		
		Cat	Dog	Rabbit
Actual	Cat (8)	5	3	0
	Dog (6)	2	3	1
	Rabbit (13)	0	2	11

- Which animal the algorithm has trouble classifying? (too many misclassifications)
- Which animal the algorithm is good at classifying?

Interpreting Confusion Matrix

(True/False Positives/Negatives)

		Predicted Condition	
		Predicted Positive	Predicted Negative
Actual condition (a cancer diagnostic)	Positive (has cancer)	True positive - Patients who have cancer are correctly identified	False negative <u>Miss rate</u> - A cancer patient is missed - Guilty prisoner was not convicted
	Negative (doesn't have cancer)	False Positive <u>Sensitivity</u> - A healthy patient is flagged incorrectly - False alarm - 'Crying wolf' - Hiring someone who is not qualified	True negative - Patients who do not have cancer are correctly identified

Confusion Matrix: Accuracy / Error Rate

■ Accuracy

Overall how accurate is the model?

$$= (TP + TN) / \text{total}$$

$$= (90 + 70) / 200$$

$$= 0.8 \text{ or } 80\%$$

■ Misclassifications / Error rate

How wrong is the model?

$$= (FP + FN) / \text{total}$$

$$= (10 + 30) / 200$$

$$= 0.2 \text{ or } 20\%$$

$$= 1 - \text{accuracy}$$

		Predicted Condition	
		Predicted Positive	Predicted Negative
Actual condition (n = 200)	Positive (n = 120)	True positive (n = 90)	False negative (n = 30)
	Negative (n = 80)	False Positive (n = 10)	True negative (n = 70)

Confusion Matrix: Accuracy May Not Be Enough

- Let's say our classifier is used to diagnose cancer patients.
- We have total 100 patients
 - 98 healthy
 - 2 have cancer
- Accuracy**
$$= (TP + TN) / \text{total}$$
$$= (1 + 98) / 100$$
$$= 99\% \text{ (very good!)}$$
- Misclassifications / Error rate**
$$= (FP + FN) / \text{total}$$
$$= (0 + 1) / 100$$
$$= 1\% (1 - \text{accuracy})$$

		Predicted Condition	
		Predicted Positive	Predicted Negative
Actual condition (n = 100)	Positive (n = 2)	True positive (n = 1)	False negative (n = 1)
	Negative (n = 98)	False Positive (n = 0)	True negative (n = 98)

Confusion Matrix: Accuracy May Not Be Enough

- **Question for class:**
 - What is the implication of 'False Positive'
 - What is the implication of 'False Negative' ?
 - Which is more serious?
- Since accuracy may not be enough of a metric, there are other metrics
 - Precision
 - Recall
- Next few slides will explain these



Confusion Matrix: TPR / FPR

- **True Positive Rate (TPR)**

/Sensitivity / Hit Rate / Recall

How often model predicts 'positive' as 'positive' (correctly) ? -- actual positive

$$= \text{TP} / (\text{TP} + \text{FN})$$

$$= 90 / 120$$

$$= 0.75 \text{ or } 75\%$$

- **False Positive Rate (FPR)**

How often model predicts 'negative' as 'positive' (incorrectly) -- actual negative

$$\text{negative} = \text{FP} / (\text{FP} + \text{TN})$$

$$= 10 / 80$$

$$= 0.125 \text{ or } 12.5\%$$

		Predicted Condition	
		Predicted Positive	Predicted Negative
Actual condition (n = 200)	Positive (n = 120)	True positive (n = 90) TPR = 75%	False negative (n = 30)
	Negative (n = 80)	False Positive (n = 10) FPR = 12.5%	True negative (n = 70)

Confusion Matrix: Specificity / Precision / Prevalence

■ Specificity

How often model predicts negative' as negative' (correctly)? -- actual no

$$= \text{TN} / (\text{TN} + \text{FP})$$

$$= 70 / (70 + 10)$$

$$= 0.875 \text{ or } 87.5 \%$$

$$= 1 - \text{FPR}$$

		Predicted Condition	
		Predicted Positive	Predicted Negative
Actual condition (n = 200)	Positive (n = 120)	True positive (n = 90)	False negative (n = 30)
	Negative (n = 80)	TPR = 75%	
		False Positive (n = 10)	True negative (n = 70)
		FPR = 12.5%	Specificity = 87.5%

■ Precision / Positive Predictive Value (PPV)

When model predicts 'positive' how often it is right? -- true / predicted positive

$$= \text{TP} / (\text{TP} + \text{FP})$$

$$= 90 / (90 + 10)$$

$$= 0.9 \text{ or } 90\%$$

Confusion Matrix: PPV / Null Error Rate

■ Prevalence

How often does 'positive' occurs in our sample

$$= \text{actual positive} / \text{total}$$

$$= 120 / 200$$

$$= 0.6 \text{ or } 60\%$$

■ Null Error Rate

How often would the model be wrong if it always predicted the majority class?

Here our majority = Positive

If we always predicted 'positive' we would be wrong 80 times (negative)

$$= 80/200$$

$$= 40\% \text{ of time}$$

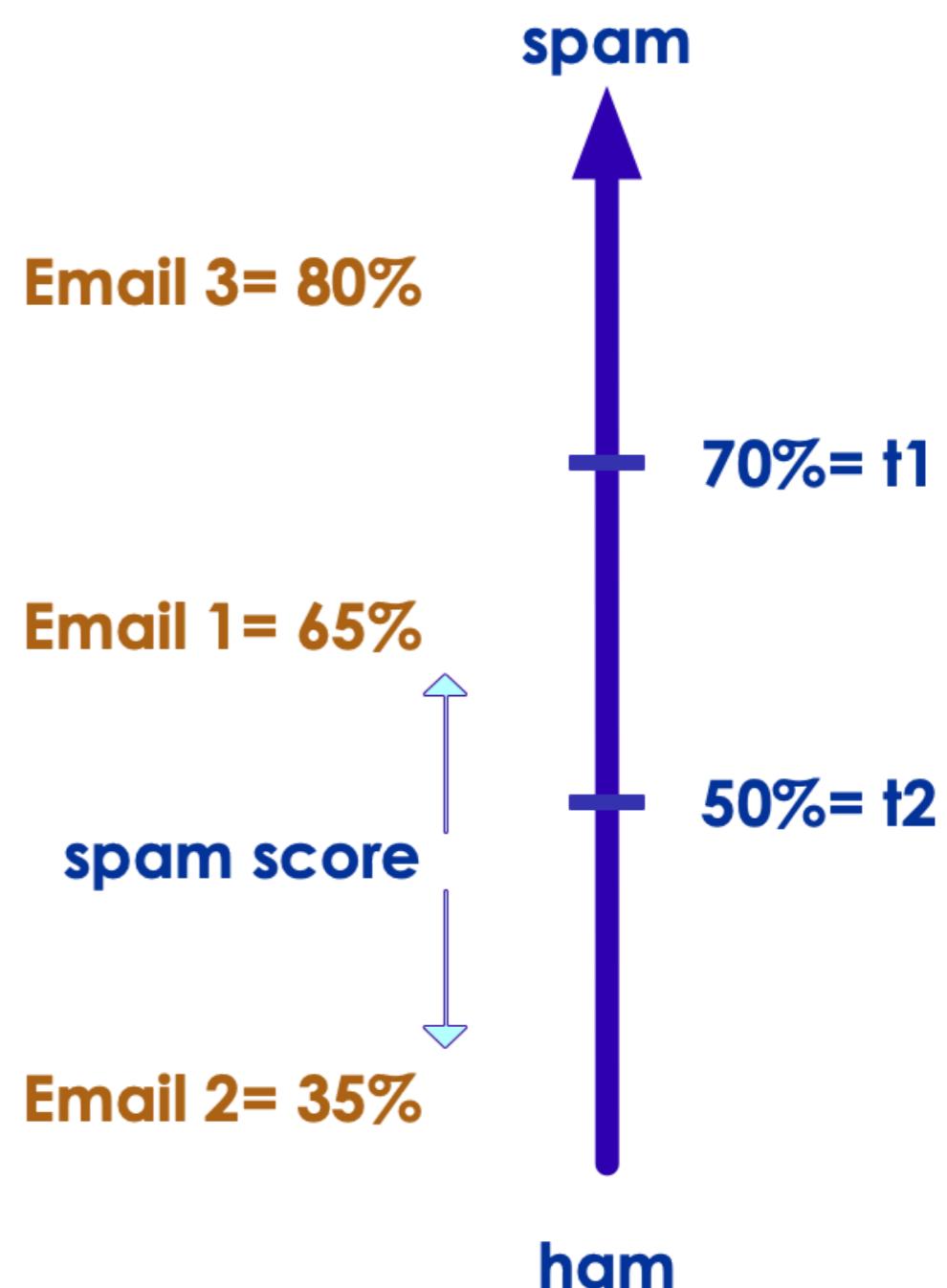
		Predicted Condition	
		Predicted Positive	Predicted Negative
Actual condition (n = 200)	Positive (n = 120)	True positive (n = 90)	False negative (n = 30)
	Negative (n = 80)	TPR = 75%	
		False Positive (n = 10)	True negative (n = 70)
		FPR = 12.5%	Specificity = 87.5%

Confusion Matrix : F-Score

- So, while precision and recall are very important measures, looking at only one of them will not provide us with the full picture.
- One way to summarize them is the f-score or f-measure, which is with the harmonic mean of precision and recall
- $F = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$

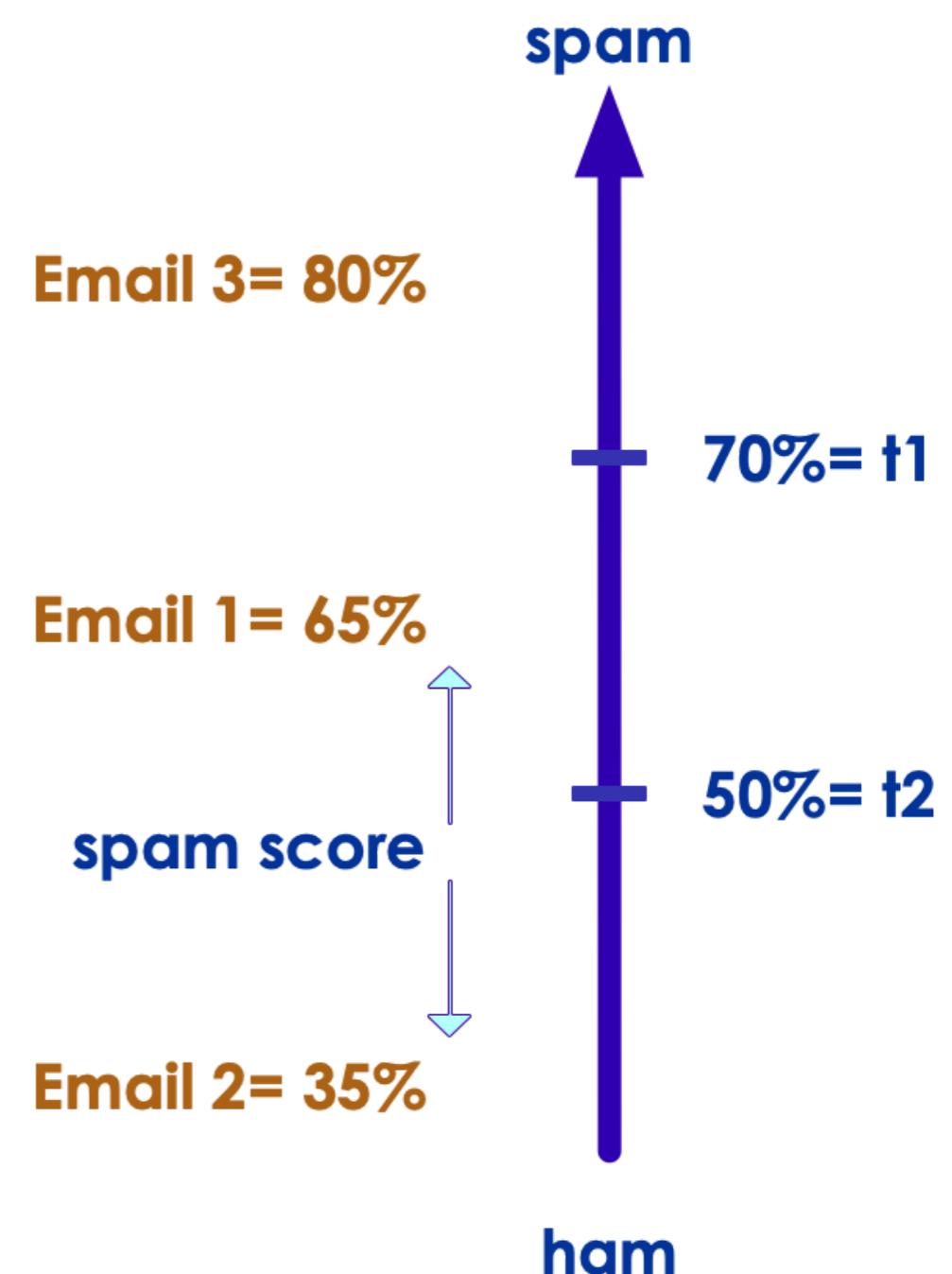
Threshold

- Our spam classifier provides a 'spam probability' for each email
 - Probability is between 0.0. and 1.0 (or 0 to 100%)
 - 1.0 definitely spam
 - 0.0 definitely not spam
- When an email's 'spam score' is above a certain number we mark it as spam
 - This is called 'threshold'



Threshold

- If spam threshold is lower (say 50%)
 - more emails will be classified as spam (email1, email3)
 - Users will miss emails (as they are in Spam folder)
- If spam threshold is higher (70%)
 - Fewer emails will be classified as spam (email3)
 - Users will see more spam emails be in Inbox
- We need to find the sweet spot for threshold



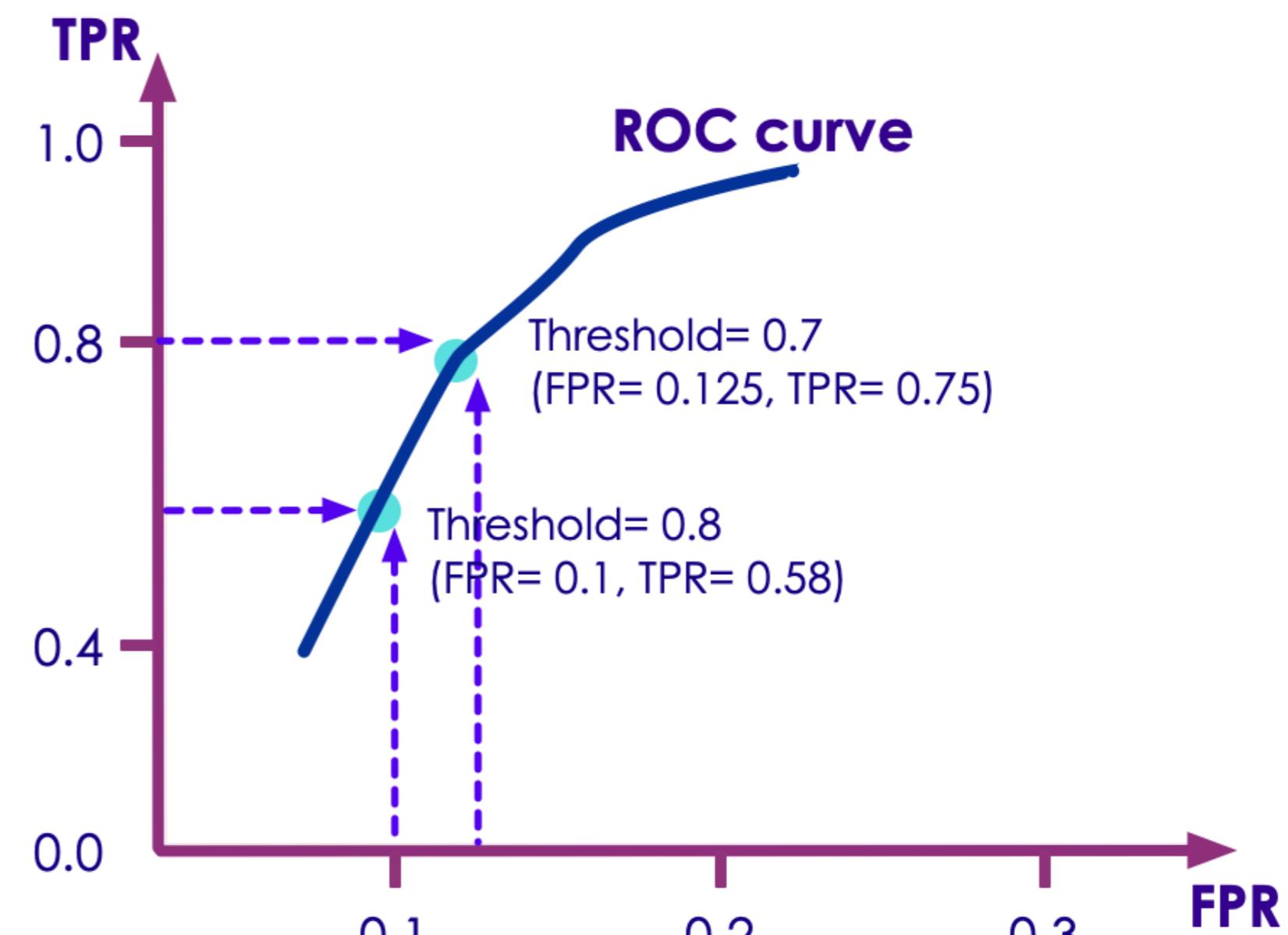
Threshold

- In first table our threshold is 0.7
 - 90 emails are correctly predicted as spam
- Next table, our threshold is higher 0.8
 - Only 70 emails are classified as spam Lower TPR

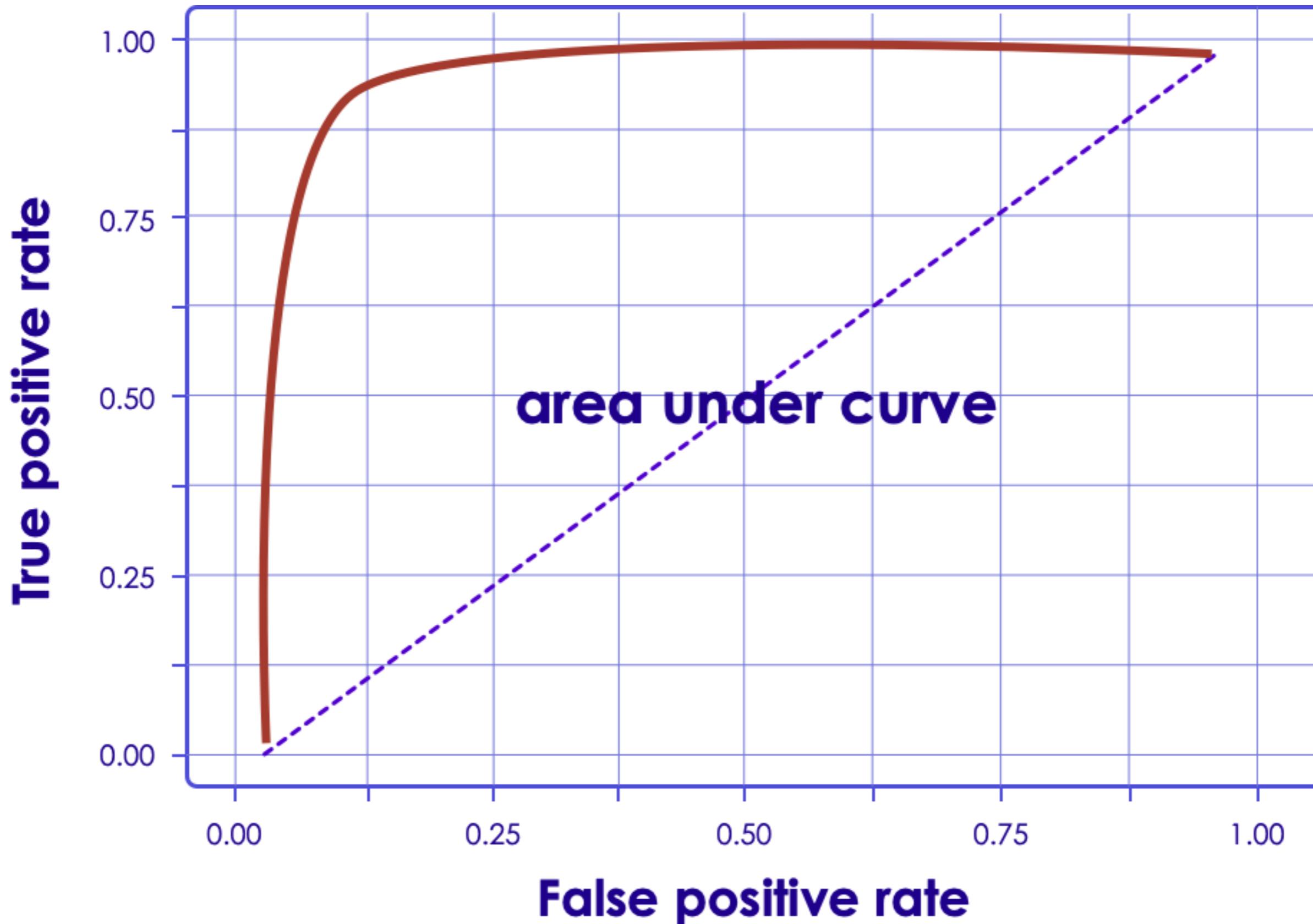
		Predicted Condition				Predicted Condition	
Threshold = 0.7		Predicted Spam	Predicted Not Spam	Threshold = 0.8 (higher)		Predicted Spam	Predicted Not Spam
Actual condition (total = 200)	Spam (n = 120)	True positive (n = 90) TPR = TP / positive = 90/120 = 75%	False negative (n = 30)	Actual condition (total = 200)	Spam (n = 120)	True positive (n = 70) TPR = TP / positive = 70 / 120 = 58.33%	False negative (n = 50)
	Not Spam (n = 80)	False Positive (n = 10) FPR = FP / negative = 10/80 = 12.5%	True negative (n = 70)		Not Spam (n = 80)	False Positive (n = 8) FPR = FP / negative = 8 / 80 = 10%	True negative (n = 72)

How is ROC Curve Generated

- Y-axis: True Positive Rate (TPR)
 - Actual=positive, predicted=positive
 - Correct!
- X-axis: False Positive Rate (FPR)
 - Actual=negative, predicted=positive
 - Incorrect!
- $0.0 \leq TPR \text{ & } FPR \leq 1.0$
- Plot TPR / FPR while varying 'threshold'

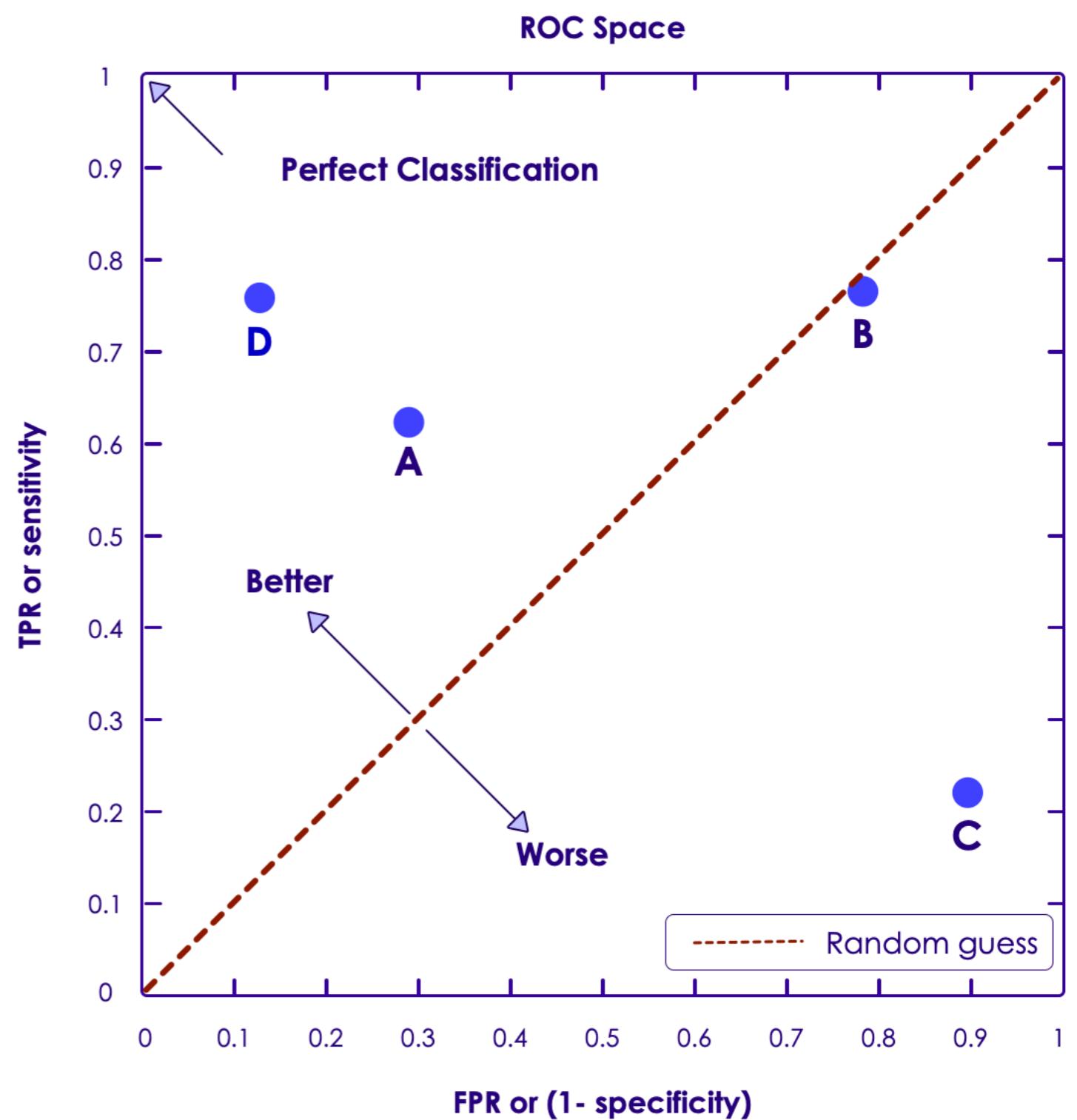


ROC Curve Example



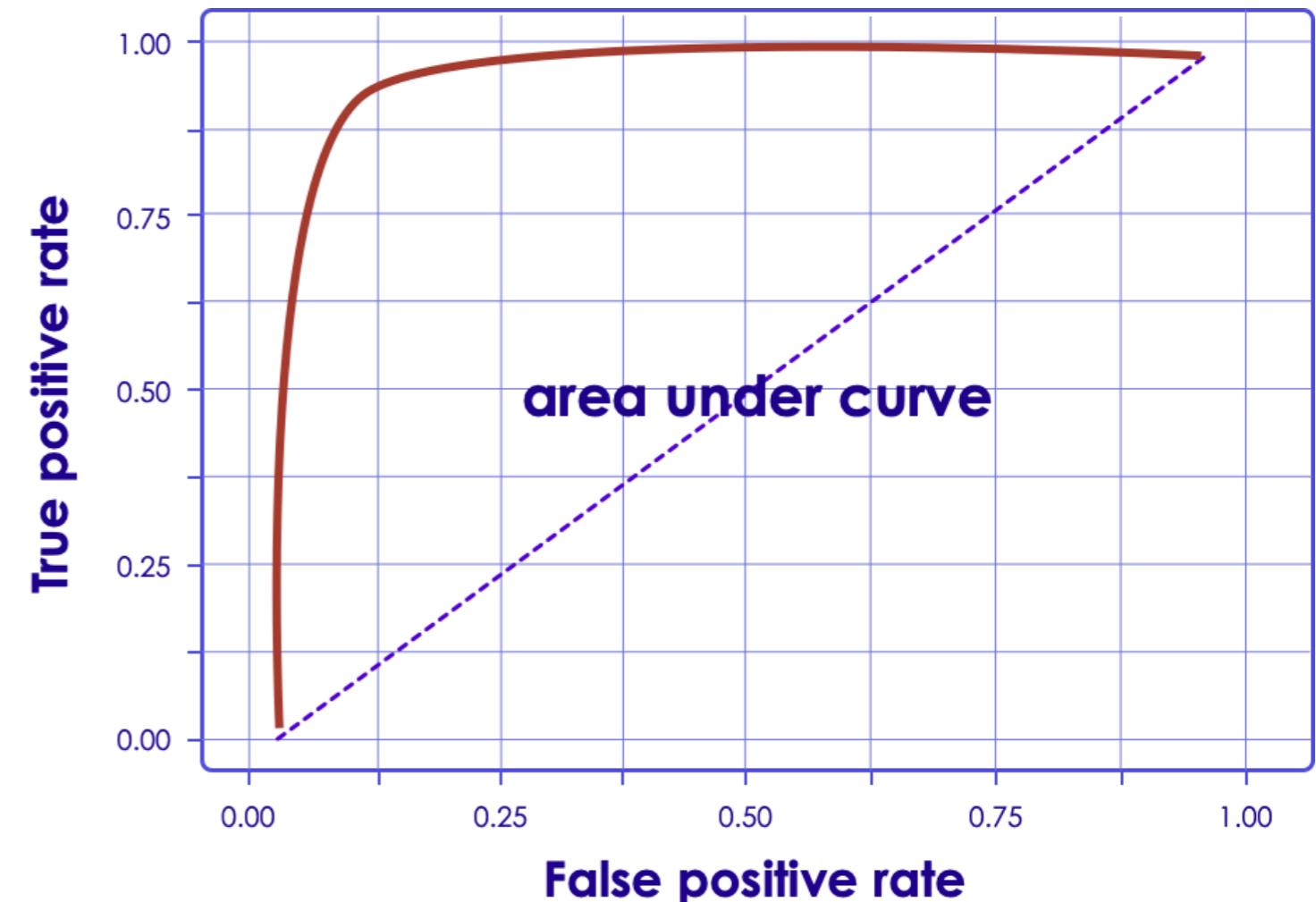
Interpreting ROC Curve

- The red line plots 'random guess' = B
- Approaching 'top left' corner would be a perfect classifier!
 - So D is better A
- C performs worse than random --> bad



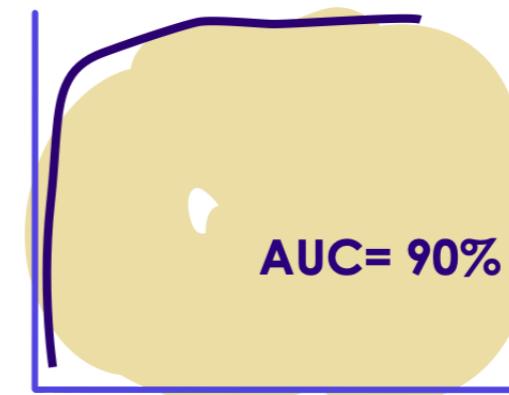
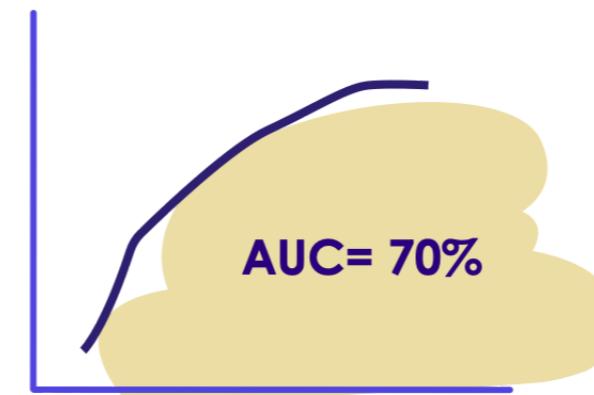
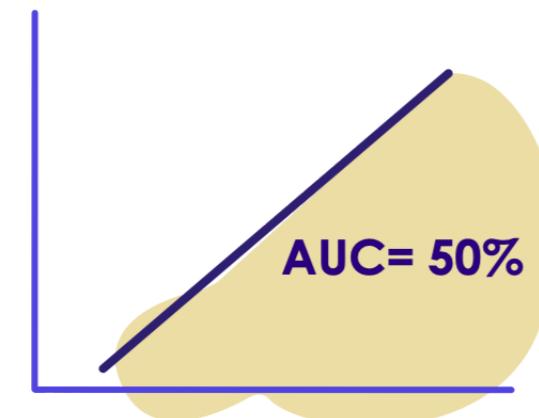
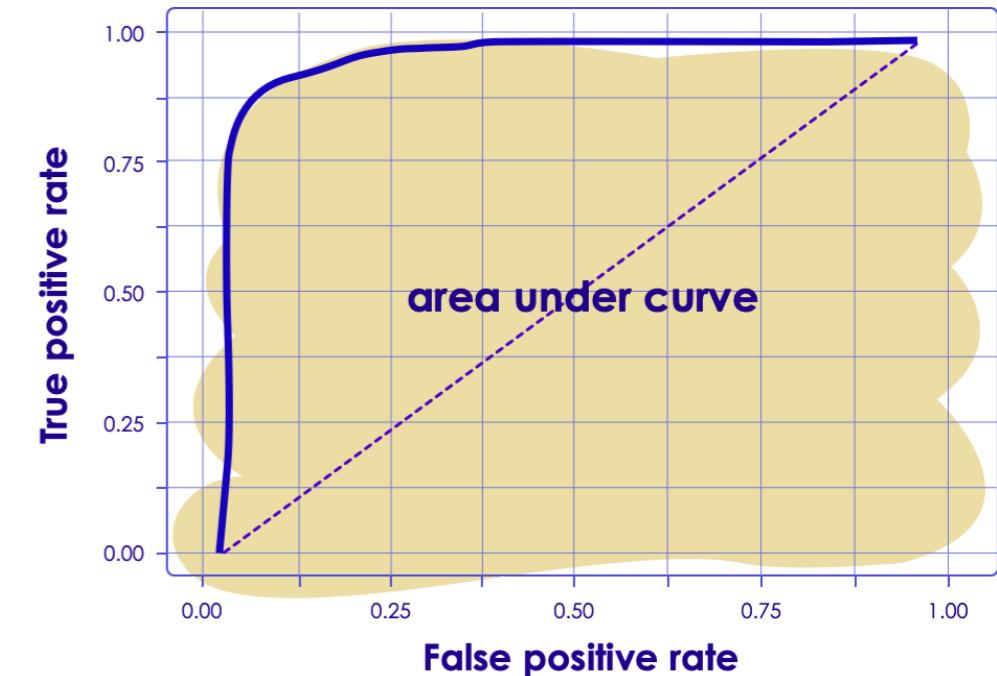
Interpreting ROC Curve

- Shows tradeoff of TPR (sensitivity) vs. FPR (1 - specificity)
- The closer to top-left , the more accurate the model
- Upper left corner (0,1) = perfect classification!
- The closer to middle line (45 degree) the less accurate the test
 - Middle line represents: random classification (50%)



Area Under Curve - AUC

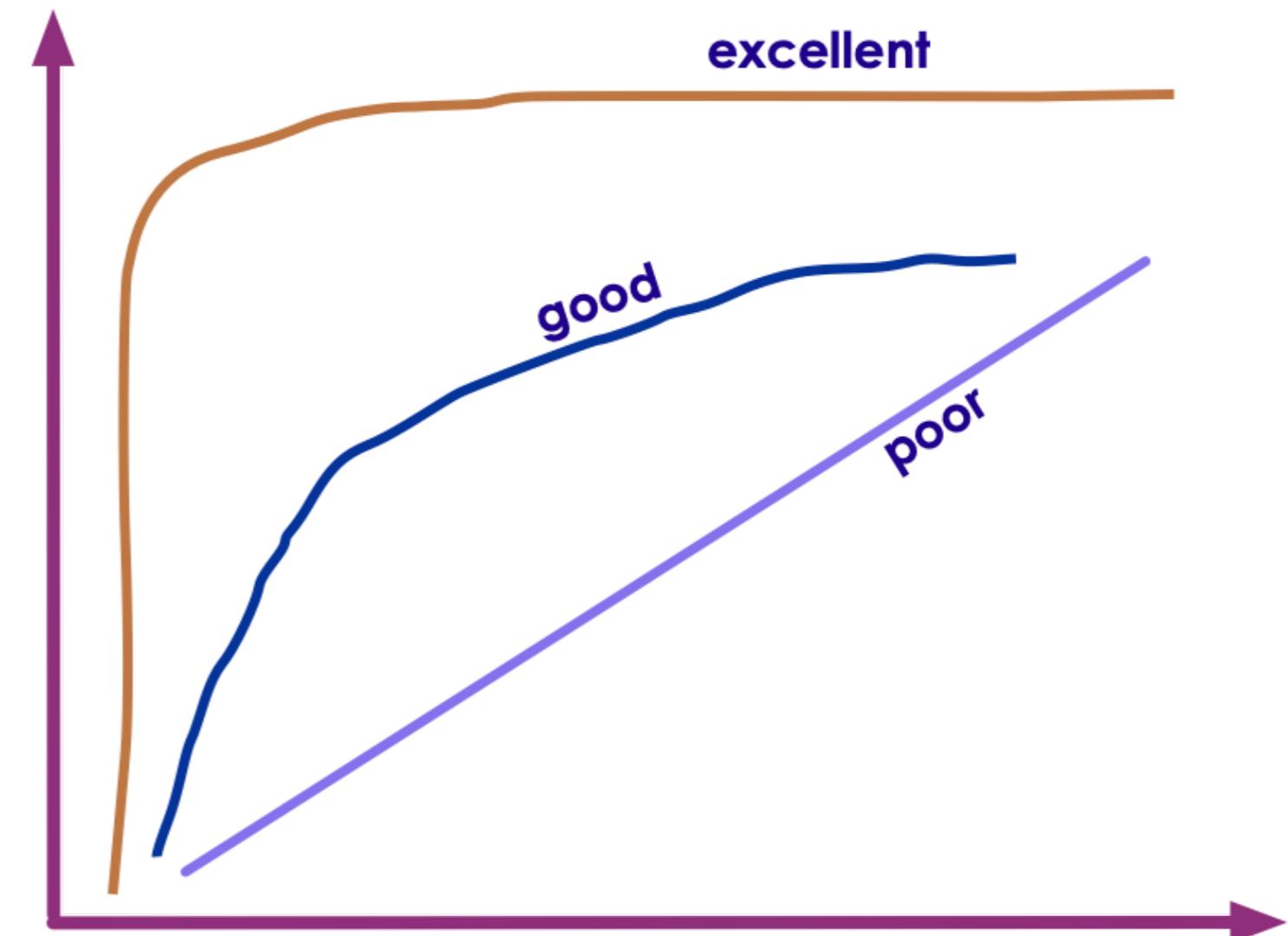
- Measures the percentage of area 'under the curve'
- AUC is between 0 and 1.0
- Higher AUC --> more accurate the model
- See 3 scenarios below
 - Leftmost is bad (50%)
 - Middle: OK (70%)
 - Rightmost: very good (90%)



Using AUC to Measure Accuracy

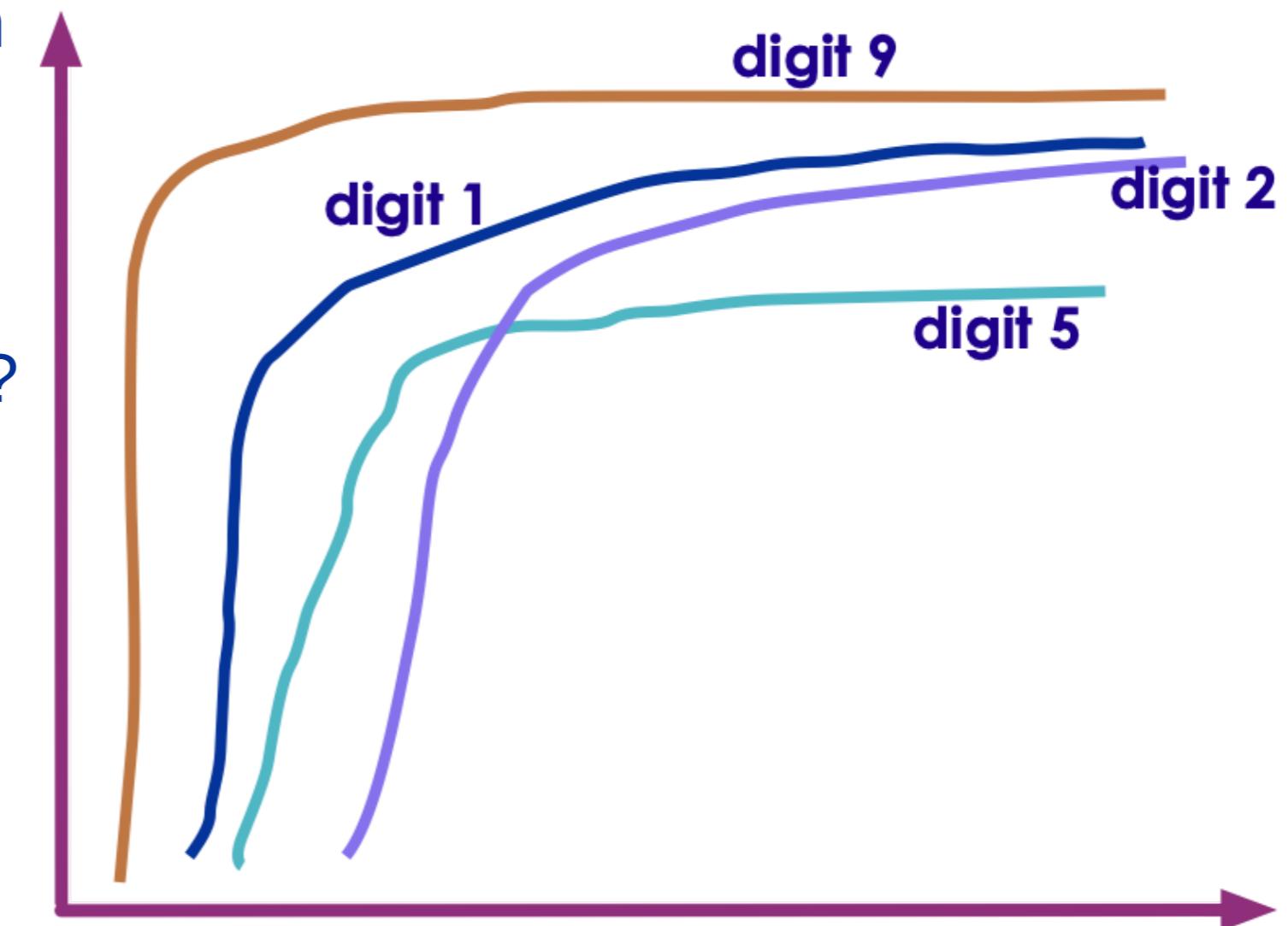
- Accuracy can be specified using a grading system

AUC	Grade
0.9 - 1.00	A - Excellent
0.80 - 0.90	B - good
0.70 - 0.80	C - fair
0.60 - 0.70	D - poor
0.50 - 0.60	F - Fail



ROC / AUC For Multiclass Classifiers

- Say our algorithm recognizes hand-written digits (postal code) into numbers.
- Its ROC can be drawn as follows
- **Question for class:**
 - Which digit the classifier is doing well?
 - Which digit the classifier is not doing well?



Review Questions

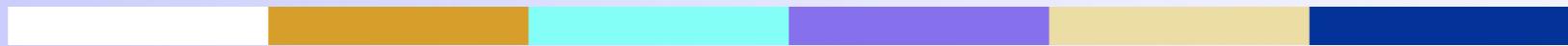
Review Questions

- Define the following:
 - Supervised learning
 - Model / Algorithm
 - Feature
 - Target / Label

Review Questions

- Explain the following
- Confusion Matrix
- ROC curve
- Area under Curve (AUC)
- Over / under fitting
- Cross validation / Boosting
- Feature Engineering

Feature Engineering



Feature Engineering

- **Feature Engineering:**

"Using transformations of raw input data to create new features to be used in ML model"

- Feature Engineering examples

- Data cleanup
- Convert to same units of measurements (imperial to metric)
- Enriching data by combining with other data sources (e.g. combining house sales prices with census data)

Features / Output

- Features are inputs to the algorithm
- Output is what we are trying to predict
- The following is an example
 - Inputs: Bedrooms, Bathrooms, Size
 - Output: Sale Price

Bedrooms (input 1)	Bathrooms (input 2)	Size sqft (input 3)	Sale Price (in thousands)(we are trying to predict)
3	1	1500	230
3	2	1800	320
5	3	2400	600
4	2	2000	500
4	3.5	2200	550

Feature Selection

Feature Selection

- We could have a lot of features to choose from
- Let's say there are 100 features in our dataset
- Not all of them would be important in predicting the outcome
- We don't want to input all the possible features into the algorithm
 - More features will take more compute power / resources
 - Can result in more noise than signal
 - Can distort the results

Class Quiz: Feature Selection

- Assume we are evaluating a credit card application
- **Q: What features we might consider?**

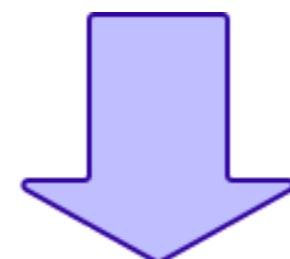


Customer_id	Name	Zipcode	Have Direct Deposit	Age	Income	Marital Status	Owns a Home
1	Joe	11111	Yes	24	45,000	Single	No
2	Jane	22222	No	34	84,000	Married	Yes

Feature Extraction

- Here is a sample data for credit card applications
- Our algorithm only uses selected input (features) to determine credit worthiness
- Here 'name' and 'zipcode' aren't considered

Customer_id	Name	Zipcode	Have Direct Deposit	Age	Income	Marital Status	Owns a Home
1	Joe	11111	Yes	24	45,000	Single	No
2	Jane	22222	No	34	84,000	Married	Yes



Feature Extraction

	Age	Income	Marital Status	Owns a Home
	24	45,000	Single	No
	34	84,000	Married	Yes

Class Quiz: Predicting Credit Card Fraud

- Assume we are evaluating if a credit card transaction is fraud or not
- **Q: What features we might consider? (open ended)**



How do We Select Features?

- **Using Domain Knowledge**

- In the previous example how did we figure out the features to consider?

Probably 'common sense' :-)

- In practice we use our **domain knowledge** to identify important features
- For example if you work in finance domain, you know what attributes are good signals
 - e.g How did Alan Greenspan predict labor market ?
- **Some algorithms can help**
Some ML algorithms can take in all features and provide '*feature importance*'

Feature Selection

- **Question:What makes a good feature?**
- Known at model building time (during training)
- Has to be meaningful to the objective
- Has enough examples in data
- Numeric features are **preferred but not required**
- Must be legal to use (See next slide)

Legal Implications

- Some features can not be used legally !
- Some examples:
 - **Race** of a person
 - **Age** of a person
 - **Address** may not be used in some instances ('red lining')
 - Can you think of any thing else that may not be used?

'Curse of Dimensionality'

- 'Curse of Dimensionality' says more features, required more *observations* (rows)
- This is not a linear relationship;
 - More features --> **many** more rows
- For example, let's say we have only 100 rows/samples of data.
 - Say each row has lots of features / columns (100+),
 - then we'd need more samples for ML algorithm to learn effectively

Non-numeric Features

Numeric Features

- Most ML algorithms deal in numbers (vectors)
- So numeric features are preferred
- Which of these is numeric?

Feature	Sample Value	Numeric?
Number of Bedrooms	3	?
House Type	- Single Family - Townhome - Apartment	?
Discount	10%	?
Owns a home	Yes / No	?
Item Category	- Jewelry - Groceries - Electronics	?

Categorical Variables

- Some of the variables are non-numeric
- Example: Marital Status (Married / Divorced / Single) / Owns a Home (Yes / No)
- We have to convert the variable to a numeric value
- Example:
Owes A Home -> 0 = No, 1 = Yes
- Categorical Variables are essentially structured data, despite being strings
- (Unstructured data would include things like: documents, emails, tweets)

			Age	Income	Marital Status	Owes a Home
			24	45,000	Single	No
			34	84,000	Married	Yes

Encoding Categorical Variables

- We have to convert our categorical variables into numbers
- 3 Strategies:
 - Factorization / Indexing
 - One-Hot-Encoding/Dummy Variables
 - Quantization

Example of Factorization / Indexing

- We can convert our string variables into factors / numbers
- This means we assign a number to each unique value of the column
- Added benefits
 - Numbers are more efficient to store
 - And compute!

The diagram illustrates the process of factorization. On the left, there is a table with two columns: 'id' and 'status'. The 'status' column contains categorical values: 'married', 'single', 'married', 'Divorced', and 'single'. An arrow points from this table to another table on the right. Inside the arrow, the mappings for the factorization are shown: 'Married = 0', 'Single = 1', and 'Divorced = 2'. The right-hand table has two columns: 'id' and 'Status idx'. The 'Status idx' column contains numerical values corresponding to the factorized categories: 0, 1, 0, 2, and 1 respectively.

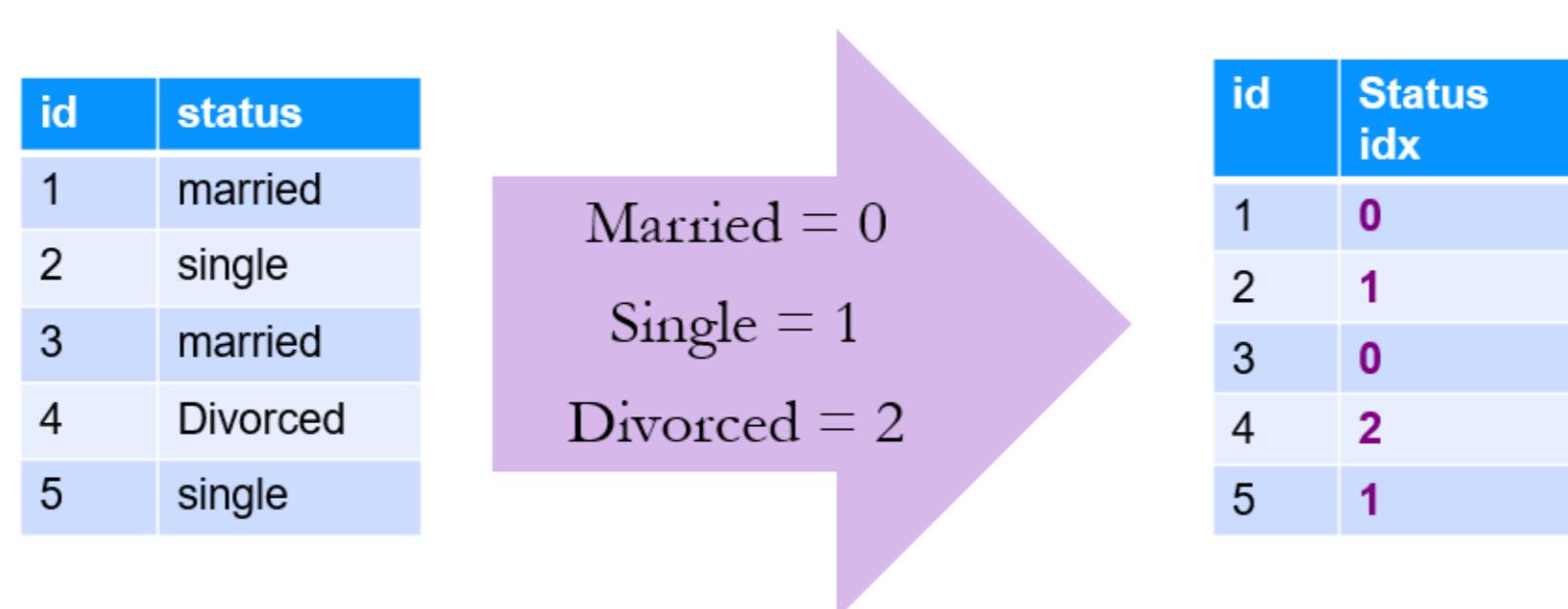
id	status
1	married
2	single
3	married
4	Divorced
5	single

Married = 0
Single = 1
Divorced = 2

id	Status idx
1	0
2	1
3	0
4	2
5	1

Potential Problems With Factorization / Indexing

- Some ML algorithms can start interpreting the numbers!
- In the example below, an ML algorithm can think
 - 2 (Divorced) > 1 (Single) > 0 (Married)
- This can lead to surprising outcomes
- We can fix this by 'one-hot-encoding' method



Dummy Variables / One-Hot-Encoding

- Dummy variables can help us treat the different values separately
 - Without trying to infer some relationship between values.
- 'dummy variables' assigns true / false to each.
 - Note, only one bit is on
 - This is called **ONE-HOT-Encoding**

The diagram illustrates the process of One-Hot Encoding. It starts with a table of status values:

id	status
1	married
2	single
3	married
4	Divorced
5	single

This is followed by a mapping table:

id	Status idx
1	0
2	1
3	0
4	2
5	1

Two purple arrows point from the original status table to the mapping table, indicating the transformation. Finally, the resulting One-Hot Encoded matrix is shown:

id	is married	is single	is divorced
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	1	0

Generating New Dimensions

- Problem: Comparing house prices
- Can we say Mountain View is most expensive city?
- On first table, there is no data point for 'size of the house'
- May be an 'apples-to-apples' comparison would be 'price per sq. foot'

City	House Price
San Jose	800k
Mountain View	1,200 k (1.2M)
San Francisco	1,000 k (1 M)
Gilroy	700 k

City	House Price	Sq. ft	Price / sq. feet
San Jose	\$ 800k	2000	\$ 400 / sqft
Mountain View	\$ 1,200 k (1.2M)	1500	\$ 800 / sqft
San Francisco	\$ 1,000 k (1 M)	1000	\$ 1000 / sqft
Gilroy	\$ 700 k	4000	\$175 / sqft

Converting Word to Vectors

Document 1
Cat dog cow

Document 2
Cat dog

Document 3
Cat Cow

Document 4
Cow Cow Dog

**Document/ Term
Matrix**

word frequency	cat	cow	dog	vector
doc 1	1	1	1	[1, 1, 1]
doc 2	1	0	1	[1, 0, 1]
doc 3	1	1	0	[1, 1, 0]
doc 4	0	2	1	[0, 2, 1]

Scaling and Normalization

Scaling

- Usually data needs to be cleaned up and transformed before creating features
- In the data below, we see **age** and **income** are in two different scales
 - age: ranges from 33 - 60
 - income ranges from 32,000 to 120,000
- Some algorithms will yield better results if these different ranges can be scaled to a uniform range
 - Remove high magnitude data

	age	income	home_owner	marital_status	approved
0	33	40000	no	single	no
1	45	80000	yes	married	yes
2	42	120000	no	divorced	yes
3	35	32000	yes	single	no
4	60	110000	yes	married	yes

Scaling Approaches

- Z-Scoring:

- Subtract mean and divide standard deviation

$$z = \frac{x - \mu}{\sigma}$$

μ – mean

σ – std deviation

- Min-Max Scaling

- Scale between a range 0 to 1 typically (or other ranges like 1 to 100)

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Standardized with zero mean and standard deviation of one

Scaling Using MinMaxScaler

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

data = pd.DataFrame ( { 'age' : [33, 45, 42, 35, 60],
                        'income' : [40000, 80000, 120000, 32000, 110000]
                      })

# create scaler, between 0 and 1
scaler = MinMaxScaler(feature_range=(0, 1))

# scale data
normalized = scaler.fit_transform(data)

# inverse transform
inverse = scaler.inverse_transform(normalized)
```

	age	income
0	33	40000
1	45	80000
2	42	120000
3	35	32000
4	60	110000

	age	income
0	0.000000	0.090909
1	0.444444	0.545455
2	0.333333	1.000000
3	0.074074	0.000000
4	1.000000	0.886364

Scaling Using StandardScaler

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

data = pd.DataFrame ( { 'age' : [33, 45, 42, 35, 60],
                        'income' : [40000, 80000, 120000, 32000, 110000]
                      })

scaler = StandardScaler()

# scale data
normalized = scaler.fit_transform(data)

# inverse transform
inverse = scaler.inverse_transform(normalized)
```

	age	income		age	income
0	33	40000	0	-0.934539	-0.914354
1	45	80000	1	0.186908	0.090431
2	42	120000	2	-0.093454	1.095215
3	35	32000	3	-0.747631	-1.115310
4	60	110000	4	1.588716	0.844019

Comparing Scalers

- Here our original data (left) , z-scaling / standard scaling (middle) is on a uniform distribution; and min-max scale (right) is between 0 to 1.0
- After standard scaling, some values would be negative! That is OK!

	age	income		age	income		age	income
0	33	40000	0	-0.934539	-0.914354	0	0.000000	0.090909
1	45	80000	1	0.186908	0.090431	1	0.444444	0.545455
2	42	120000	2	-0.093454	1.095215	2	0.333333	1.000000
3	35	32000	3	-0.747631	-1.115310	3	0.074074	0.000000
4	60	110000	4	1.588716	0.844019	4	1.000000	0.886364

Lab: Exploratory Data Analysis (EDA)

- **Overview:**
 - Analyze house sales data
- **Approximate Time:**
 - 20 - 25 mins
- **Instructions:**
 - '**exploration/explore-house-sales**' lab for Python / R / Spark

Bonus Lab: Feature Engineering

- **Overview:**

- Feature engineering exercises

- **Approximate Time:**

- 20 - 30 mins

- **Instructions:**

- **'feature-eng' lab for Python / R / Spark**