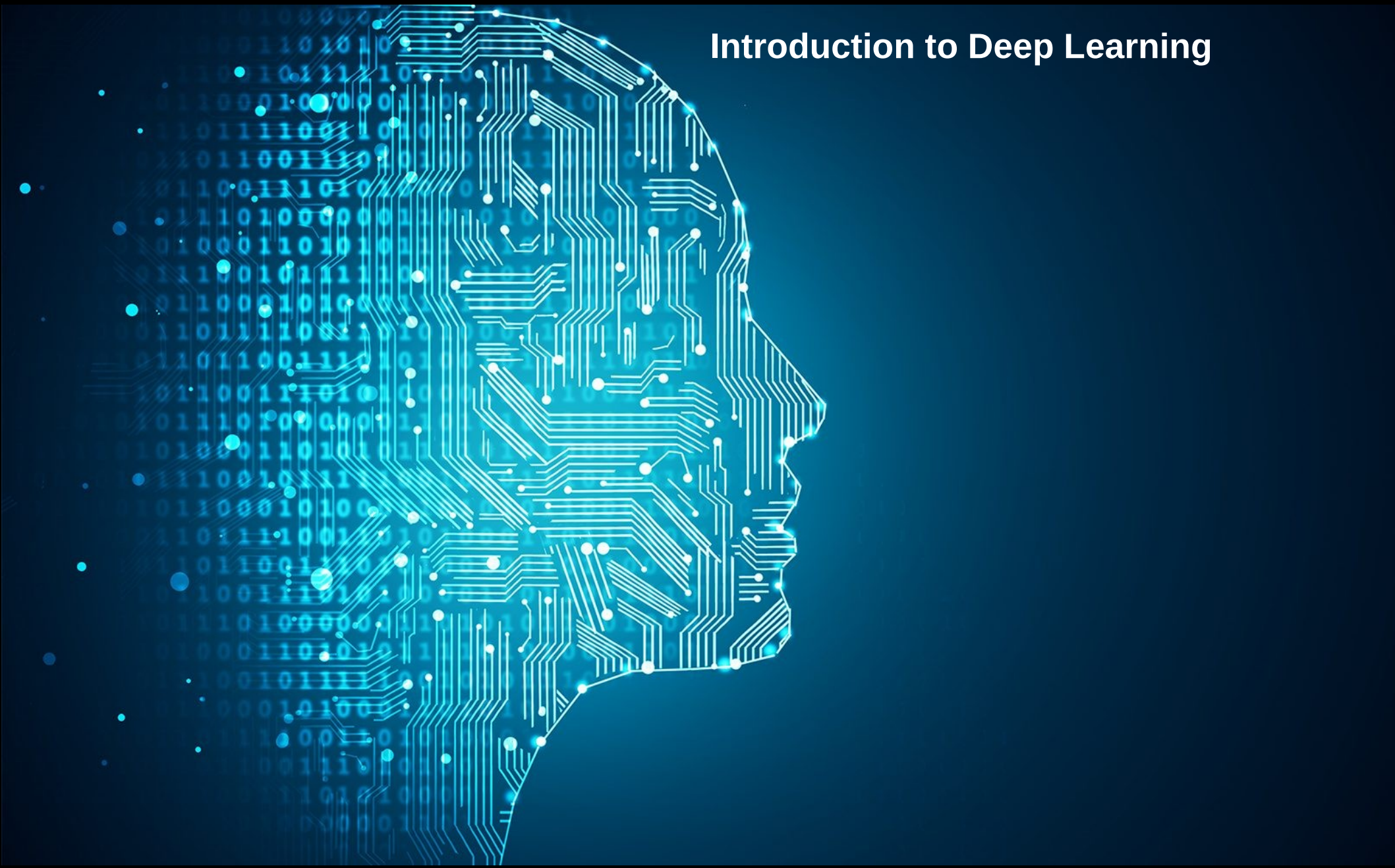

Introduction to Deep Learning

1) Machine Learning Overview

2) Basics of Neural Networks

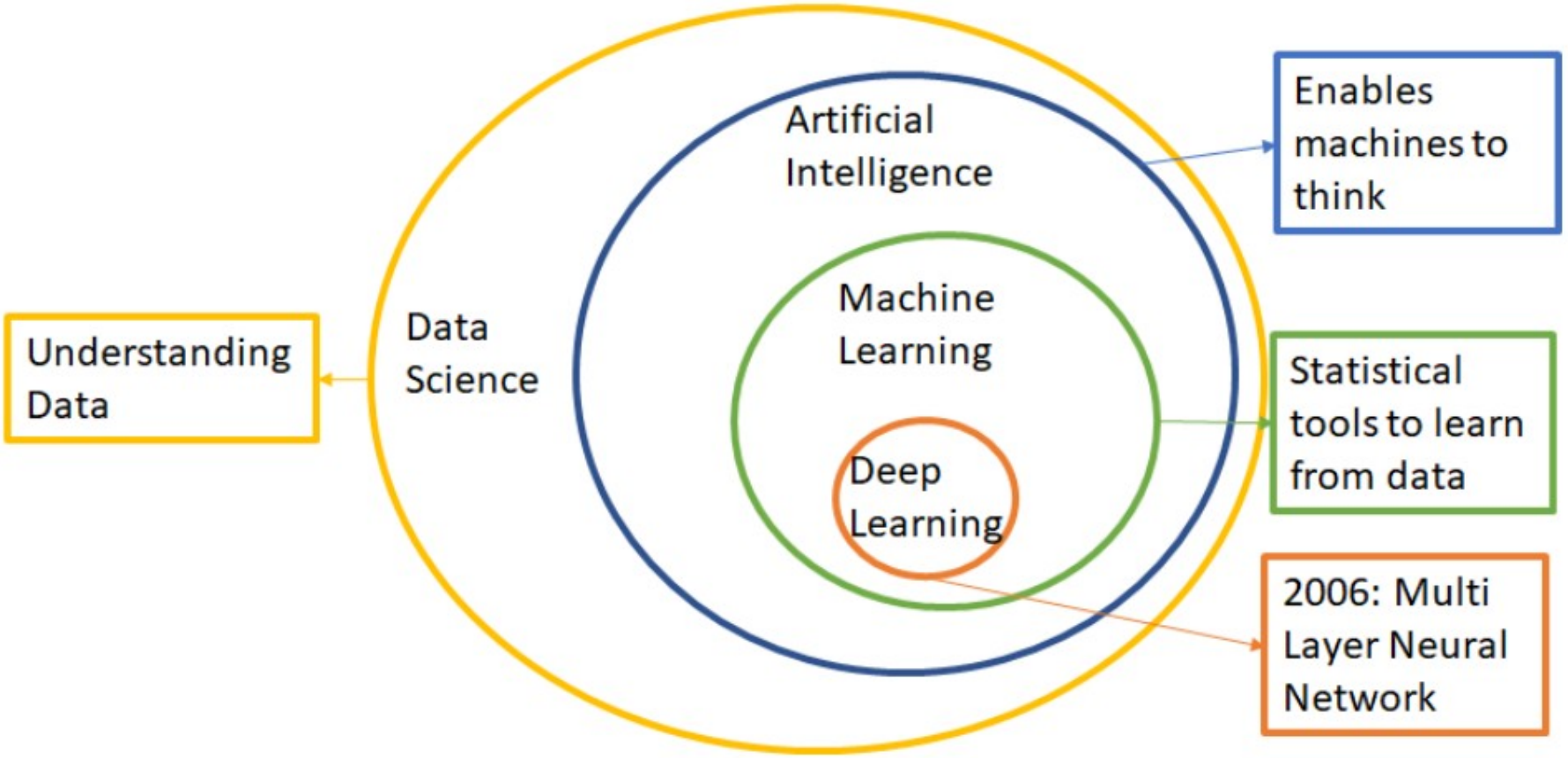
3) TensorFlow Lecture

4) Generative Adversarial Neural Networks

5) NLP and deep learning

6) Final Project Presentation

- 1) Beginning of Basic Machine Learning -


Understanding Data
Data Science
Artificial Intelligence
Machine Learning
Deep Learning
Enables machines to think
Statistical tools to learn from data
2006: Multi Layer Neural Network

What is Machine Learning?

"Learning is any process by which a system improves performance from experience."

- Herbert Simon

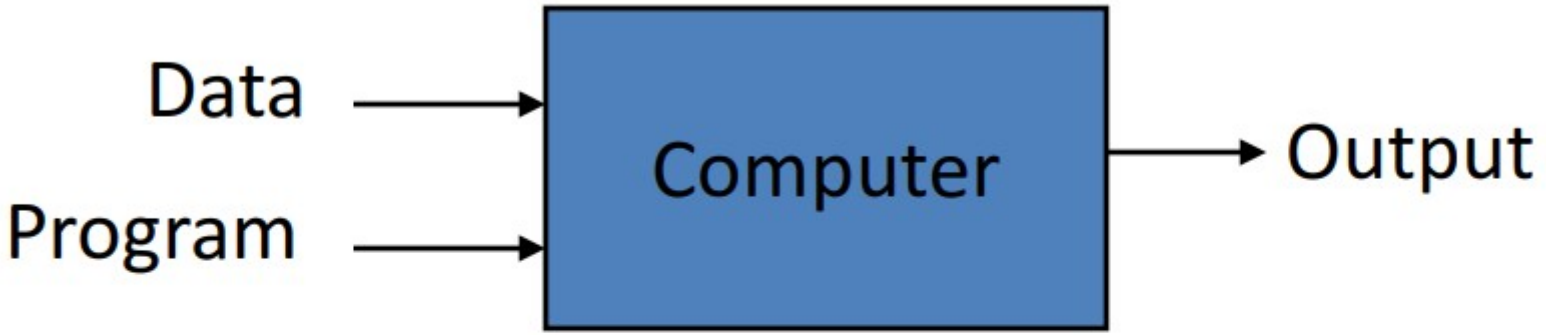
Definition by Tom Mitchell (1998):

Machine Learning is the study of algorithms that

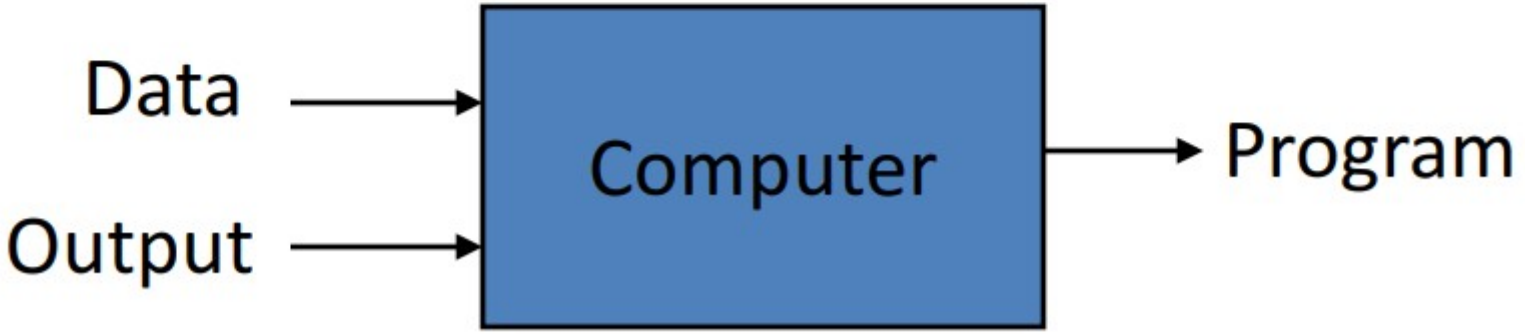
- improve their performance P
- at some task T
- with experience E.

A well-defined learning task is given by <P, T, E>.

Traditional Programming



Machine Learning



Types of Learning

- **Supervised (inductive) learning**
  - Given: training data + desired outputs (labels)
- **Unsupervised learning**
  - Given: training data (without desired outputs)
- **Semi-supervised learning**
  - Given: training data + a few desired outputs
- **Reinforcement learning**
  - Rewards from sequence of actions

- Every machine learning algorithm has three components:
  - **Representation**
  - **Evaluation**
  - **Optimization**

Representation

- Decision trees
- Sets of rules / Logic programs
- Instances
- Graphical models (Bayes/Markov nets)
- Neural networks
- Support vector machines
- Model ensembles
- Logistic regression
- Randomized Forests
- Boosted Decision Trees
- K-nearest neighbor
- Etc.

Evaluation

`

- Accuracy
- Precision and recall
- Mean squared error
- Max Likelihood
- Posterior probability
- Cost / Utility
- Entropy
- Etc.

Optimization

- Combinatorial optimization
  - E.g.: Greedy search
- Convex optimization
  - E.g.: Gradient descent
- Constrained optimization
  - E.g.: Linear programming

Learning techniques

- Supervised learning categories and techniques
  - **Linear classifier** (numerical functions)
    - Works well: output depends on many features
  - **Parametric** (probabilistic functions)
    - Work wells: limited data, but with assumptions about function
    - Naïve Bayes, Gaussian discriminant analysis (GDA), Hidden Markov models (HMM), …
  - **Non-parametric** (Instance-based functions)
    - Works well: Lot of data, no prior knowledge
    - *K*-nearest neighbors, Kernel regression, Kernel density estimation, …

Learning techniques

- Unsupervised learning categories and techniques
  - **Clustering**
    - K-means clustering
    - Spectral clustering
  - **Density Estimation**
    - Gaussian mixture model (GMM)
    - Graphical models
  - **Dimensionality reduction**
    - Principal component analysis (PCA)
    - Factor analysis

Understanding Random Forest

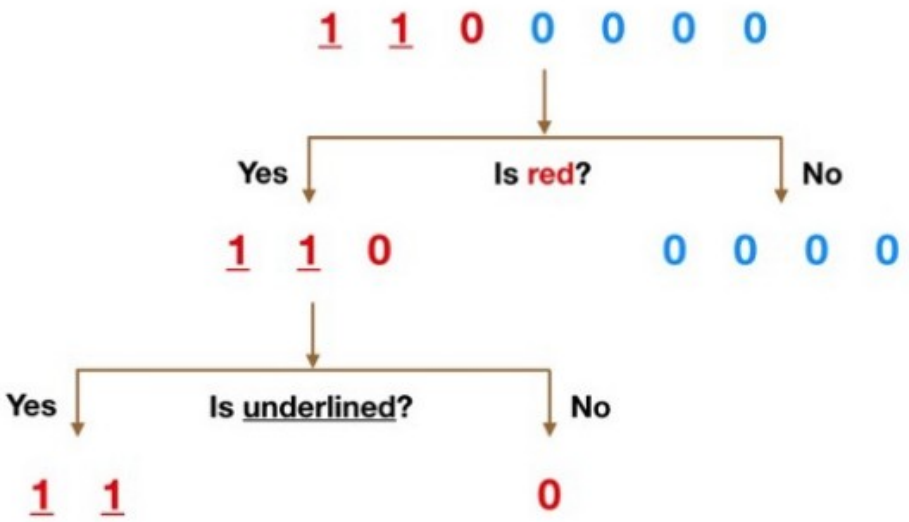
Credit: Tony Yiu

A big part of machine learning is classification — we want to know what class (a.k.a. group) an observation belongs to. The ability to precisely classify observations is extremely valuable for various business applications like predicting whether a particular user will buy a product or forecasting whether a given loan will default or not.

Data science provides a plethora of classification algorithms such as logistic regression, support vector machine, naive Bayes classifier, and decision trees. But near the top of the classifier hierarchy is the random forest classifier (there is also the random forest regressor but that is a topic for another day).

Decision Trees

Let's quickly go over decision trees as they are the building blocks of the random forest model.



Simple Decision Tree Example

Imagine that our dataset consists of the numbers at the top of the figure to the left. We have two 1s and five 0s (1s and 0s are our classes) and desire to separate the classes using their features. The features are color (red vs. blue) and whether the observation is underlined or not. So how can we do this?

Color seems like a pretty obvious feature to split by as all but one of the 0s are blue. So we can use the question, "Is it red?" to split our first node. You can think of a node in a tree as the point where the path splits into two — observations that meet the criteria go down the Yes branch and ones that don't go down the No branch.

The No branch (the blues) is all 0s now so we are done there, but our Yes branch can still be split further. Now we can use the second feature and ask, "Is it underlined?" to make a second split.

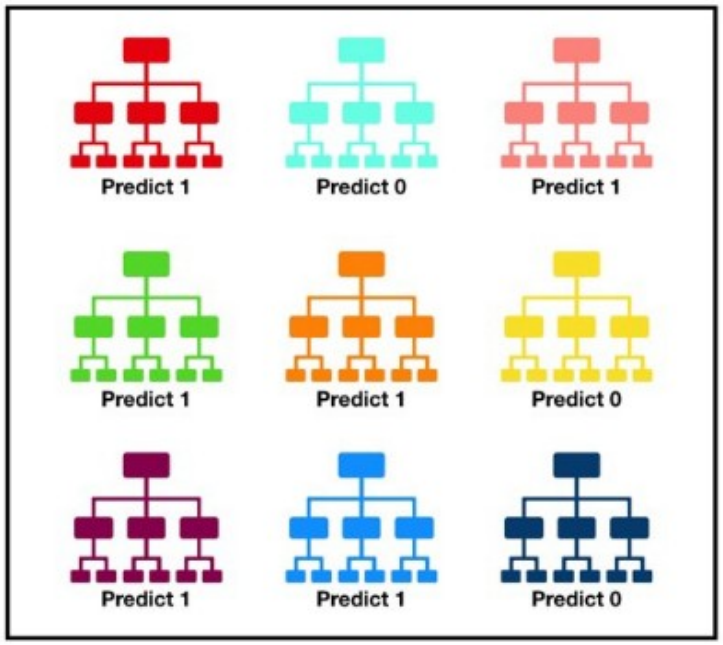
The two 1s that are underlined go down the Yes subbranch and the 0 that is not underlined goes down the right subbranch and we are all done. Our decision tree was able to use the two features to split up the data perfectly. Victory!

> *What feature will allow me to split the observations at hand in a way that the resulting groups are as different from each other as possible (and the members of each resulting subgroup are as similar to each other as possible)?*

. . .

The Random Forest Classifier

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction (see figure below).



Visualization of a Random Forest Model Making a Prediction

The fundamental concept behind random forest is a simple but powerful one — the wisdom of crowds. In data science speak, the reason that the random forest model works so well is:

Top highlight

A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.

The low correlation between models is the key. Just like how investments with low correlations (like stocks and bonds) come together to form a portfolio that is greater than the sum of its parts, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. **The reason for this wonderful effect is that the trees protect each other from their individual errors** (as long as they don't constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction. So the prerequisites for random forest to perform well are:

1. There needs to be some actual signal in our features so that models built using those features do better than random guessing.
2. The predictions (and therefore the errors) made by the individual trees need to have low correlations with each other.

An Example of Why Uncorrelated Outcomes are So

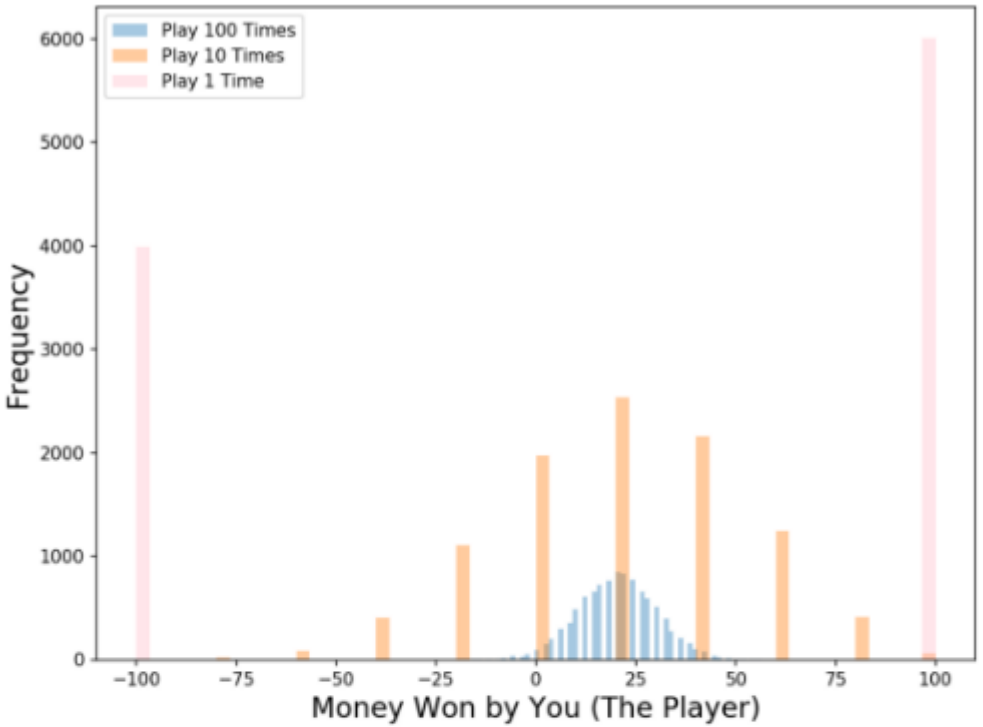
Imagine that we are playing the following game:

- I use a uniformly distributed random number generator to produce a number.
- If the number I generate is greater than or equal to 40, you win (so you have a 60% chance of victory) and I pay you some money. If it is below 40, I win and you pay me the same amount.
- Now I offer you the the following choices. We can either:

1. **Game 1** — play 100 times, betting $1 each time.
2. **Game 2**— play 10 times, betting $10 each time.
3. **Game 3**— play one time, betting $100.

Which would you pick? The expected value of each game is the same:

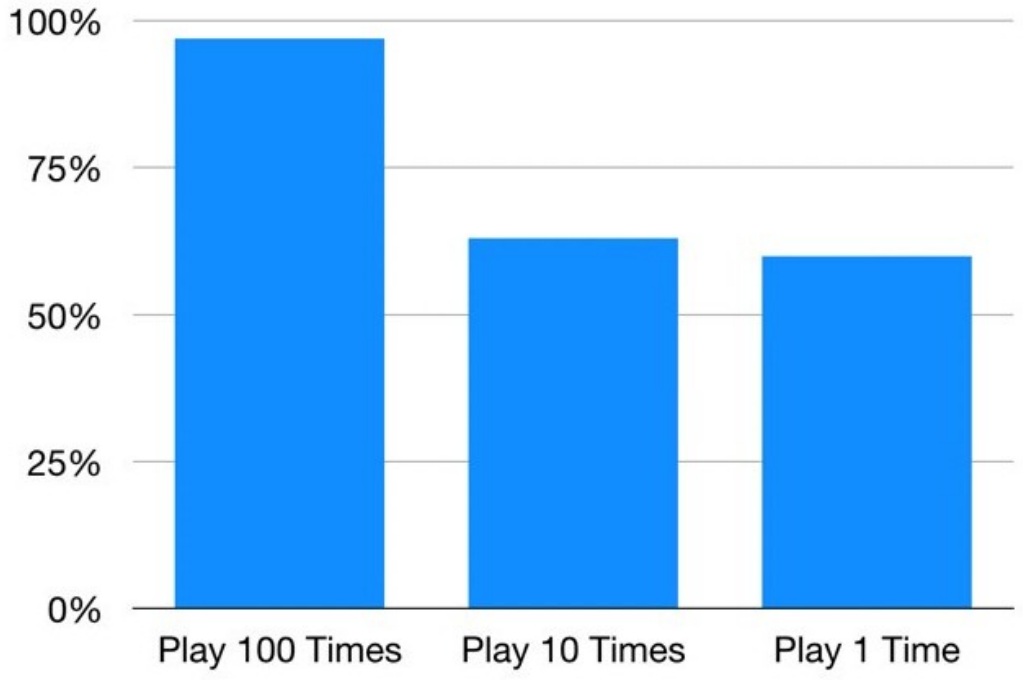
> *Expected Value Game 1 = (0.60*1 + 0.40*-1)*100 = 20*
>
> *Expected Value Game 2= (0.60*10 + 0.40*-10)*10 = 20*
>
> *Expected Value Game 3= 0.60*100 + 0.40*-100 = 20*



Outcome Distribution of 10,000 Simulations for each Game

What about the distributions? Let's visualize the results with a Monte Carlo simulation (we will run 10,000 simulations of each game type; **for example, we will simulate 10,000 times the 100 plays of Game 1**). Take a look at the chart on the left — now which game would you pick? Even though the expected values are the same, **the outcome distributions are vastly different going from positive and narrow (blue) to binary (pink).**

Game 1 (where we play 100 times) offers up the best chance of making some money — **out of the 10,000 simulations that I ran, you make money in 97% of them!** For Game 2 (where we play 10 times) you make money in 63% of the simulations, a drastic decline (and a drastic increase in your probability of losing money). And Game 3 that we only play once, you make money in 60% of the simulations, as expected.



Probability of Making Money for Each Game

So even though the games share the same expected value, their outcome distributions are completely different. The more we split up our $100 bet into different plays, the more confident we can be that we will make money. As mentioned previously, this works because each play is independent of the other ones.

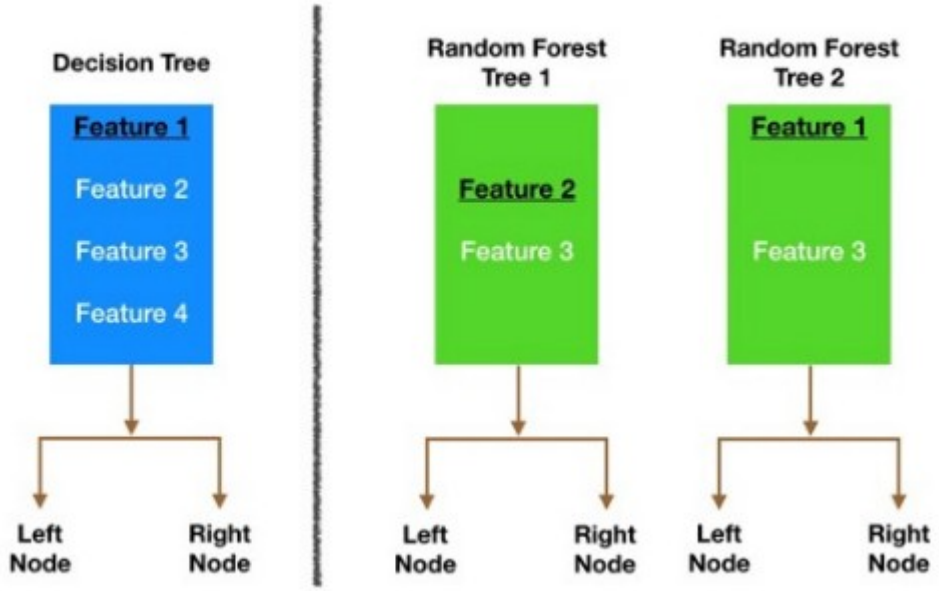
Random forest is the same — each tree is like one play in our game earlier. We just saw how our chances of making money increased the more times we played. Similarly, with a random forest model, our chances of making correct predictions increase with the number of uncorrelated trees in our model.

Ensuring that the Models Diversify Each Other

So how does random forest ensure that the behavior of each individual tree is not too correlated with the behavior of any of the other trees in the model? It uses the following two methods:

Bagging (Bootstrap Aggregation) — Decisions trees are very sensitive to the data they are trained on — small changes to the training set can result in significantly different tree structures. Random forest takes advantage of this by allowing each individual tree to randomly sample from the dataset with replacement, resulting in different trees. This process is known as bagging.

Notice that with bagging we are not subsetting the training data into smaller chunks and training each tree on a different chunk. Rather, if we have a sample of size N, we are still feeding each tree a training set of size N (unless specified otherwise). But instead of the original training data, we take a random sample of size N with replacement. For example, if our training data was [1, 2, 3, 4, 5, 6] then we might give one of our trees the following list [1, 2, 2, 3, 6, 6]. Notice that both lists are of length six and that "2" and "6" are both repeated in the randomly selected training data we give to our tree (because we sample with replacement).



Node splitting in a random forest model is based on a random subset of features for each tree.

Feature Randomness — In a normal decision tree, when it is time to split a node, we consider every possible feature and pick the one that produces the most separation between the observations in the left node vs. those in the right node. In contrast, each tree in a random forest can pick only from a random subset of features. This forces even more variation amongst the trees in the model and ultimately results in lower correlation across trees and more diversification.

Let's go through a visual example — in the picture above, the traditional decision tree (in blue) can select from all four features when deciding how to split the node. It decides to go with Feature 1 (black and underlined) as it splits the data into groups that are as separated as possible.

Now let's take a look at our random forest. We will just examine two of the forest's trees in this example. When we check out random forest Tree 1, we find that it it can only consider Features 2 and 3 (selected randomly) for its node splitting decision. We know from our traditional decision tree (in blue) that Feature 1 is the best feature for splitting, but Tree 1 cannot see Feature 1 so it is forced to go with Feature 2 (black and underlined). Tree 2, on the other hand, can only see Features 1 and 3 so it is able to pick Feature 1.

> *So in our random forest, we end up with trees that are not only trained on different sets of data (thanks to bagging) but also use different features to make decisions.*

Conclusion

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=4,
                           n_informative=2, n_redundant=0,
                           random_state=0, shuffle=False)
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X, y)

print(clf.predict([[0, 0, 0, 0]]))
```

> *The random forest is a classification algorithm consisting of many decisions trees.* ***It uses bagging and feature randomness when building each individual tree to try to create an uncorrelated forest of trees*** *whose prediction by committee is more accurate than that of any individual tree.*

What do we need in order for our random forest to make accurate class predictions?

1. **We need features that have at least some predictive power.** After all, if we put garbage in then we will get garbage out.
2. **The trees of the forest and more importantly their predictions need to be uncorrelated** (or at least have low correlations with each other). While the algorithm itself via feature randomness tries to engineer these low correlations for us, the features we select and the hyper-parameters we choose will impact the ultimate correlations as well.

Introduction to Unsupervised Learning

Credit: Marco Peixeiro

Unsupervised learning is a set of statistical tools for scenarios in which there is only a set of features and no targets. Therefore, we cannot make predictions, since there are no associated responses to each observation. Instead, we are interested in finding an interesting way to visualize data or in discovering subgroups of similar observations.

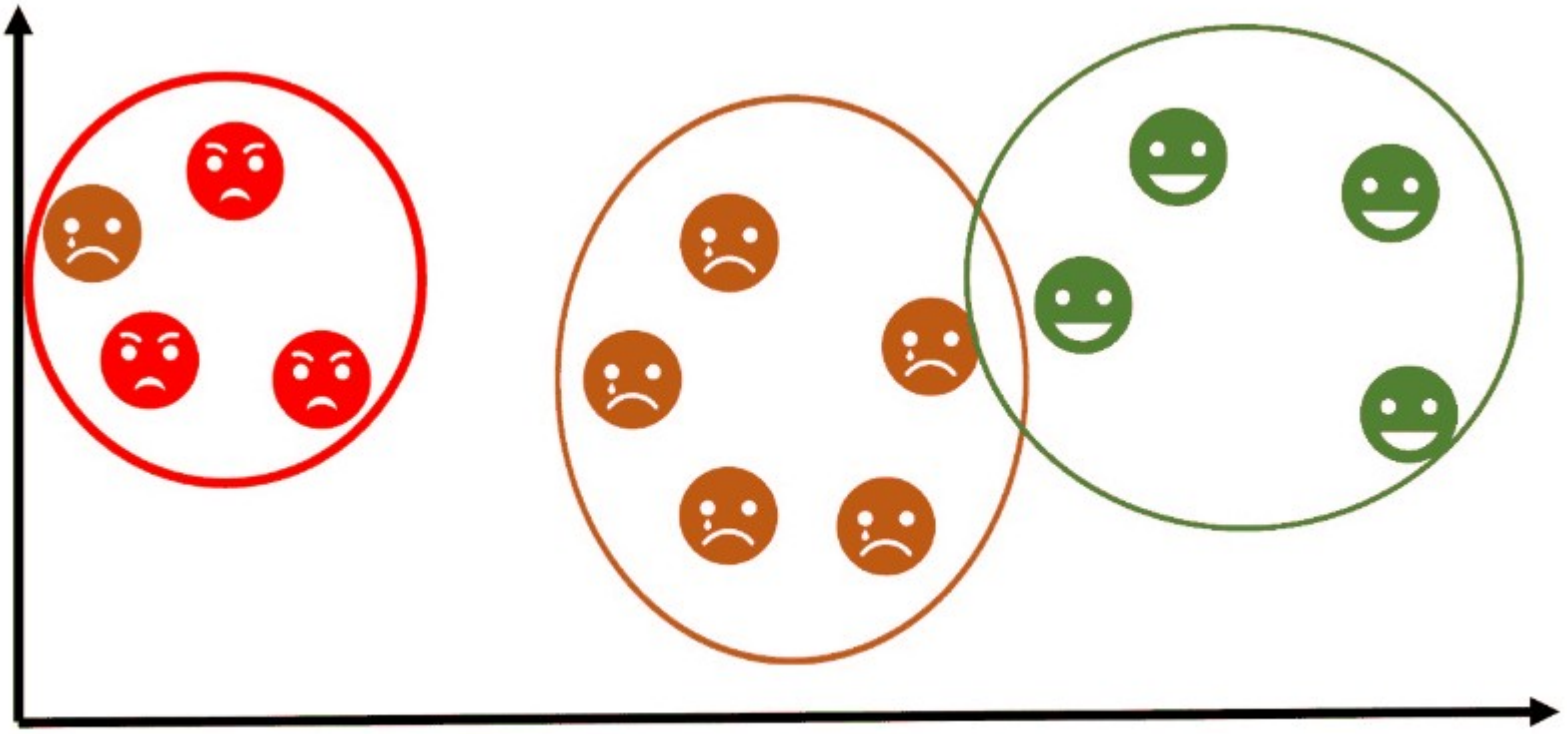
Unsupervised learning tends to be more challenging, because there is no clear objective for the analysis, and it is often subjective. Additionally, it is hard to assess if the obtained results are good, since there is no accepted mechanism for performing cross-validation or validating results on an independent dataset, because we do not know the true answer.

Clustering — Unsupervised Learning

Credit: Anuja Nagpal

What is Clustering?

"Clustering" is the process of grouping similar entities together. The goal of this unsupervised machine learning technique is to find similarities in the data point and group similar data points together.



> ***Why use Clustering?***

Grouping similar entities together help profile the attributes of different groups. In other words, this will give us insight into underlying patterns of different groups. There are many applications of grouping unlabeled data, for example, you can identify different groups/segments of customers and market each group in a different way to maximize the revenue. Another example is grouping documents together which belong to the similar topics etc.

Clustering is also used to reduces the dimensionality of the data when you are dealing with a copious number of variables.

> *How does Clustering algorithms work?*

1. K-mean Clustering

2. Hierarchical Clustering

K-mean Clustering

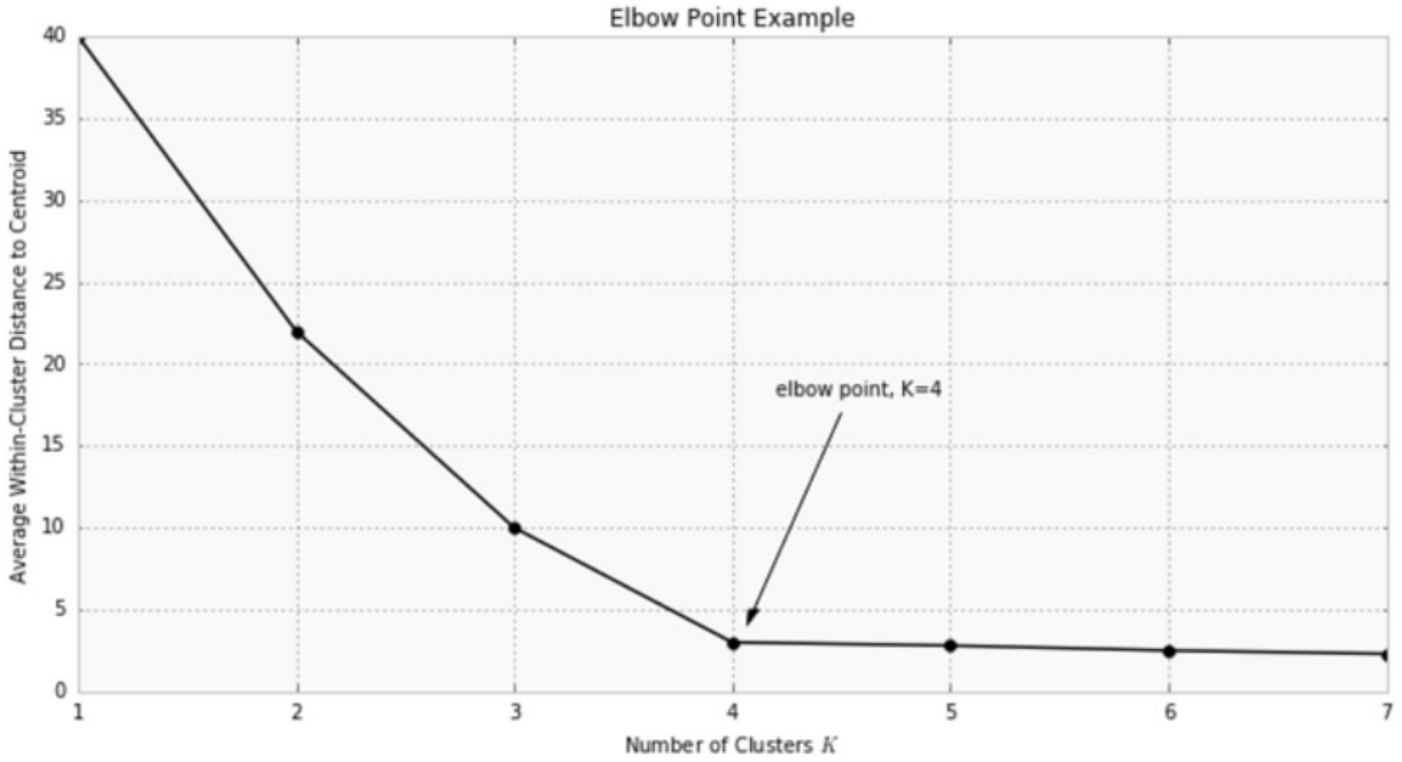
1. It starts with K as the input which is how many clusters you want to find. Place K centroids in random locations in your space.
2. Now, using the euclidean distance between data points and centroids, assign each data point to the cluster which is close to it.
3. Recalculate the cluster centers as a mean of data points assigned to it.
4. Repeat 2 and 3 until no further changes occur.

Now, you might be thinking that how do I decide the value of K in the first step.

One of the methods is called "**Elbow**" method can be used to decide an optimal number of clusters. Here you would run K-mean clustering on a range of K values and plot the "*percentage of variance explained*" on the Y-axis and "*K*" on X-axis.

In the picture below you would notice that as we add more clusters after 3 it doesn't give much better modeling on the data. The first cluster adds much information, but at some point, the marginal gain will start dropping.

Elbow Diagram



Elbow Diagram

Hierarchical Clustering

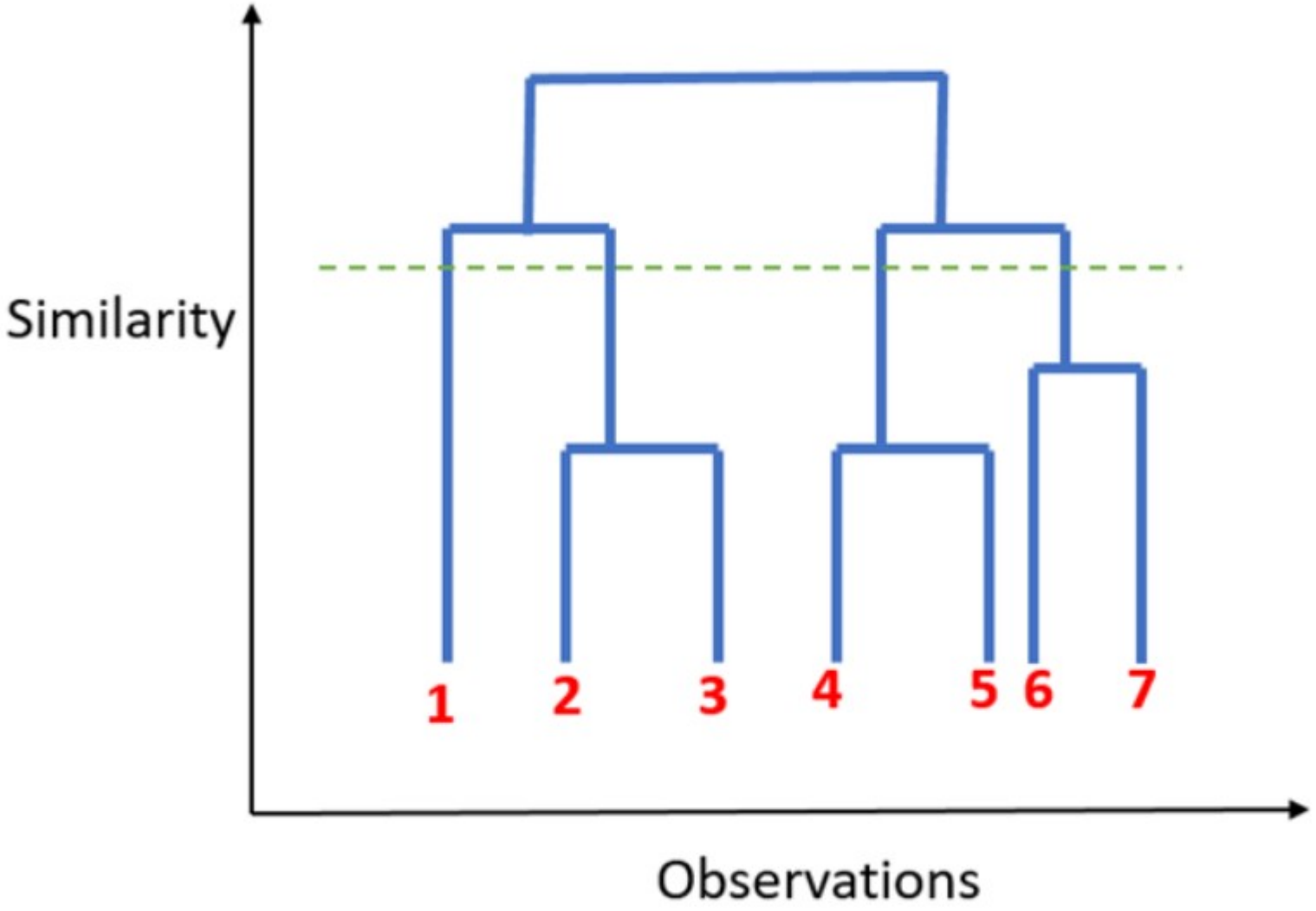
Unlike K-mean clustering *Hierarchical* clustering starts by assigning all data points as their own cluster. As the name suggests it builds the hierarchy and in the next step, it combines the two nearest data point and merges it together to one cluster.

1. Assign each data point to its own cluster.

2. Find closest pair of cluster using euclidean distance and merge them in to single cluster.

3. Calculate distance between two nearest clusters and combine until all items are clustered in to a single cluster.

In this technique, you can decide the optimal number of clusters by noticing which vertical lines can be cut by horizontal line without intersecting a cluster and covers the maximum distance.



Dendogram

Inspiration: The Brain

- Many machine learning methods inspired by biology, e.g., the (human) brain
- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons

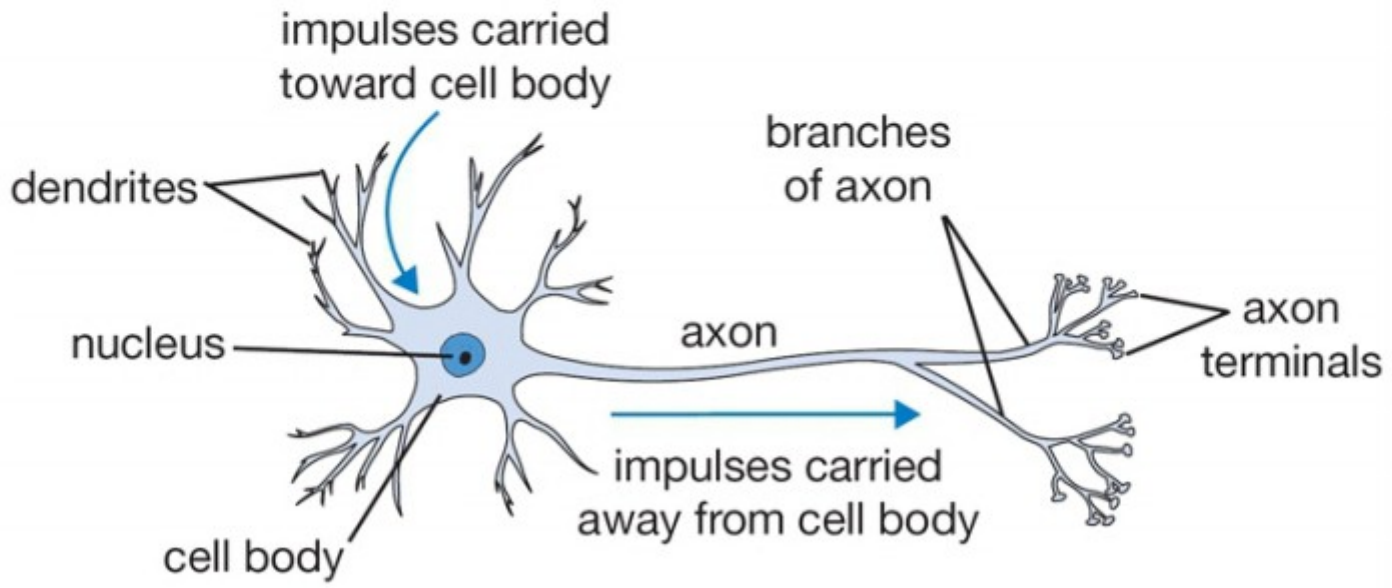


Figure : The basic computational unit of the brain: Neuron

Mathematical Model of a Neuron

- Neural networks define functions of the inputs (hidden features), computed by neurons
- Artificial neurons are called units

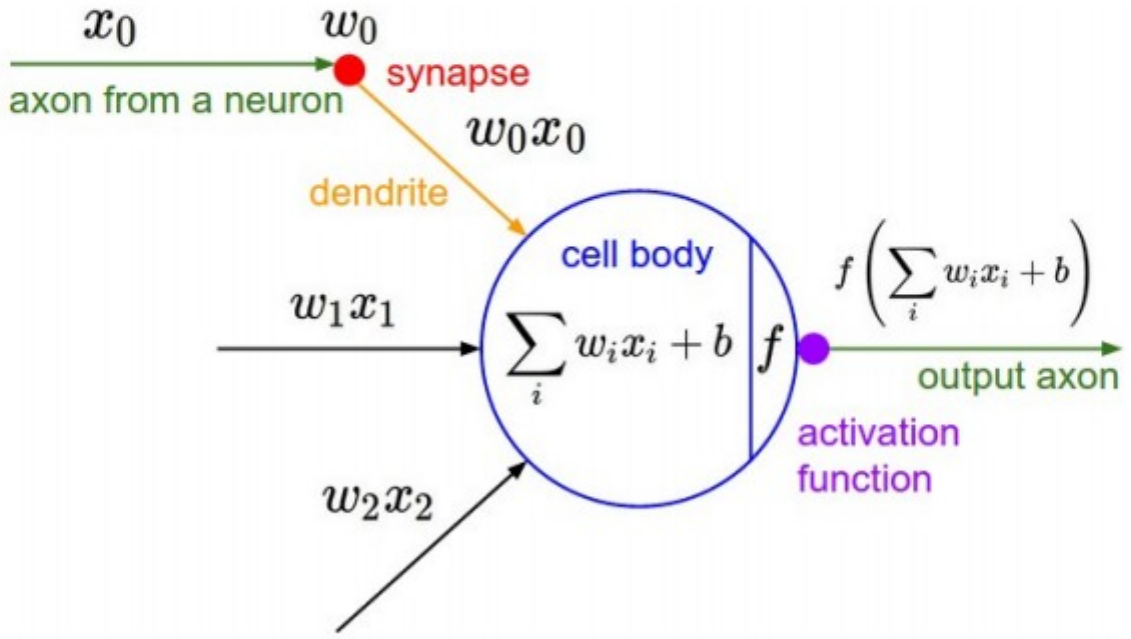
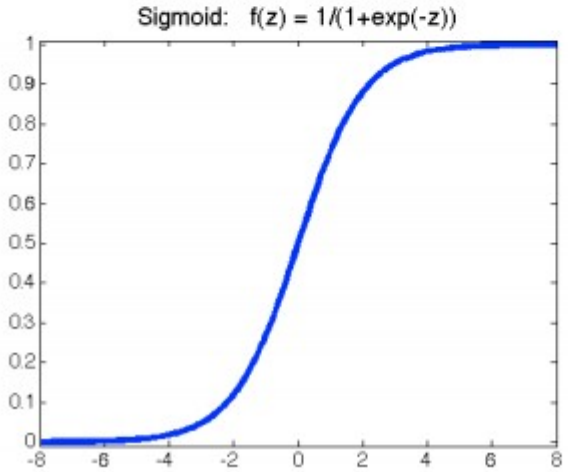


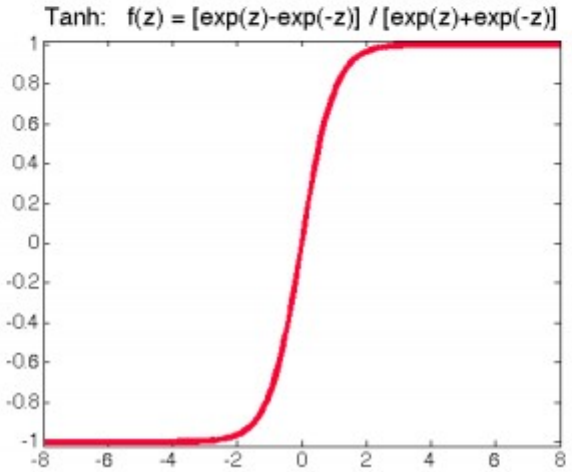
Figure : A mathematical model of the neuron in a neural network

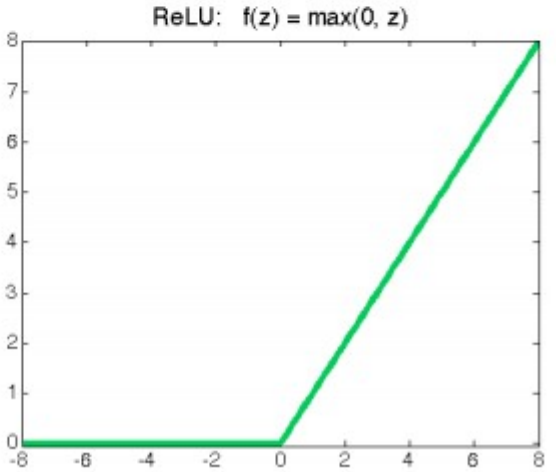
Activation Functions

Most commonly used activation functions:

- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$

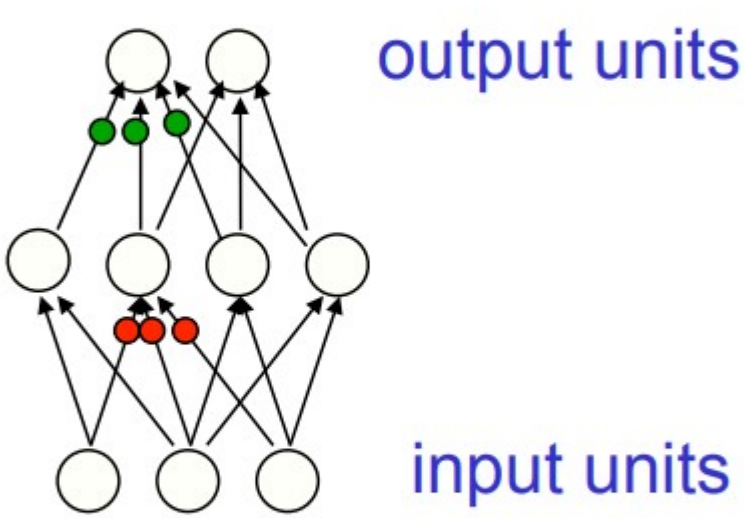






Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:



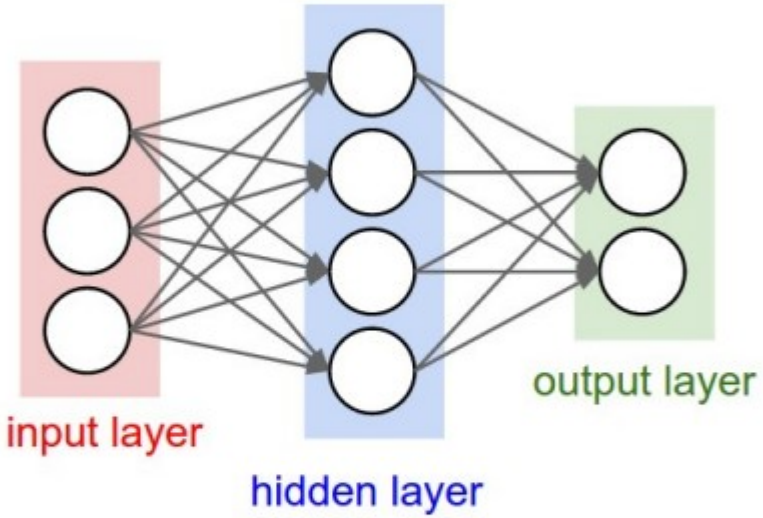


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Each unit computes its value based on linear combination of values of units that point into it, and an activation function

- Network with one layer of four hidden units:

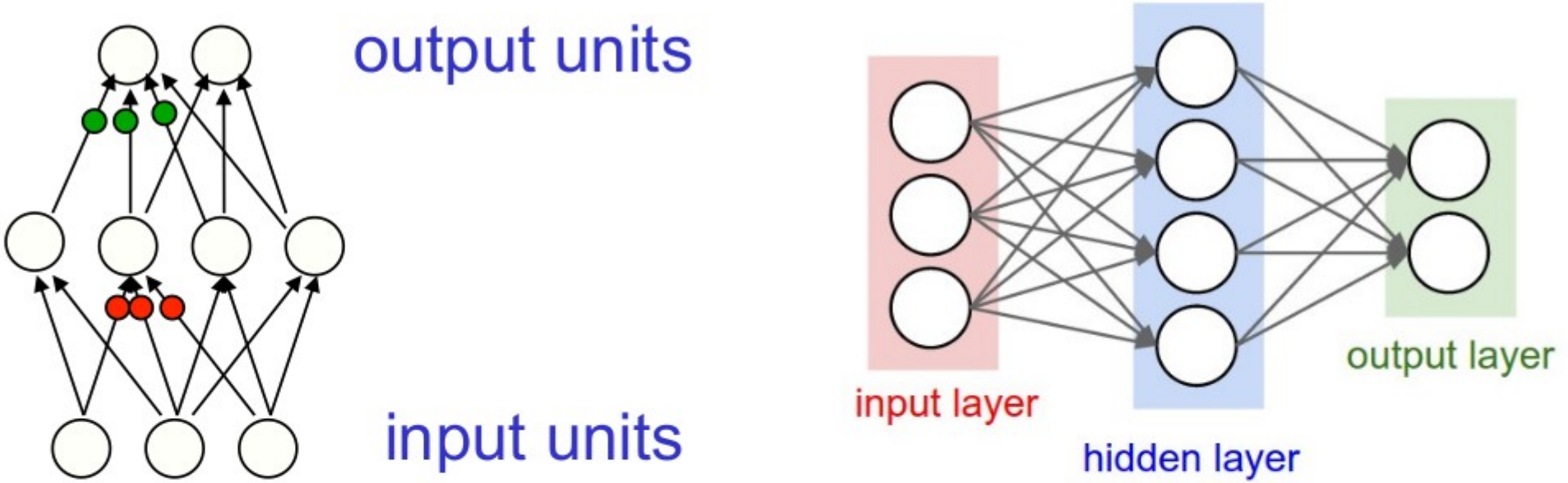


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Naming conventions; a 2-layer neural network:
  - One layer of hidden units
  - One output layer
    (we do not count the inputs as a layer)

Neural Network Architecture (Multi-Layer Perceptron)

- Going deeper: a 3-layer neural network with two layers of hidden units

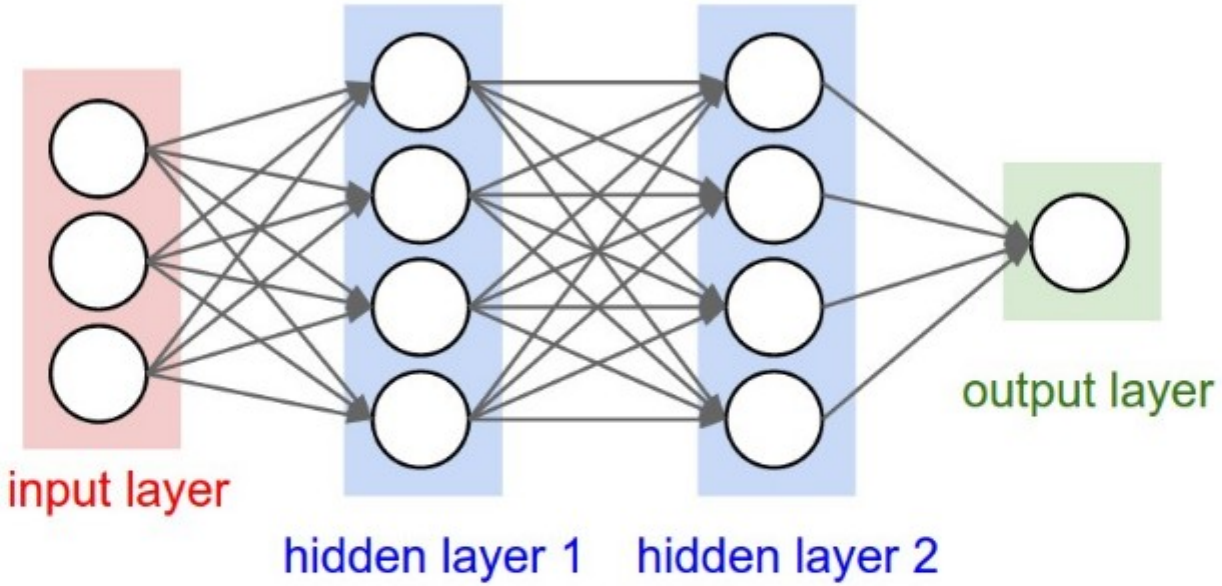


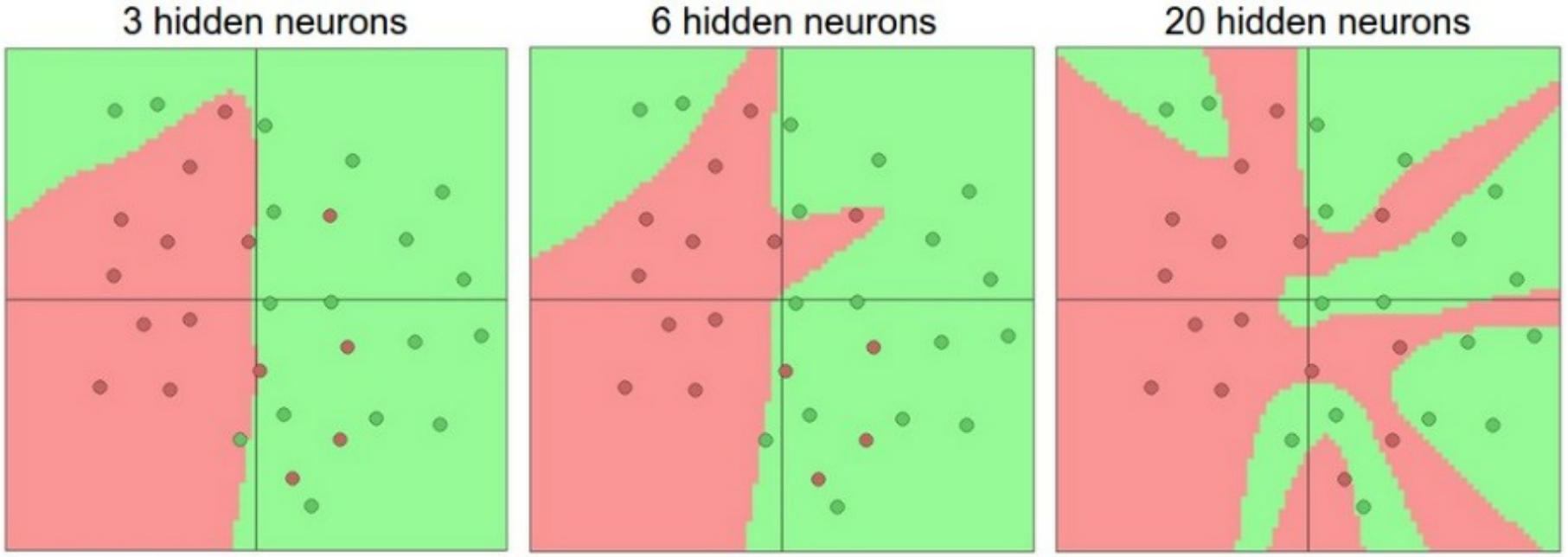
Figure : A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; a N-layer neural network:
  - $N-1$ layers of hidden units
  - One output layer

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper

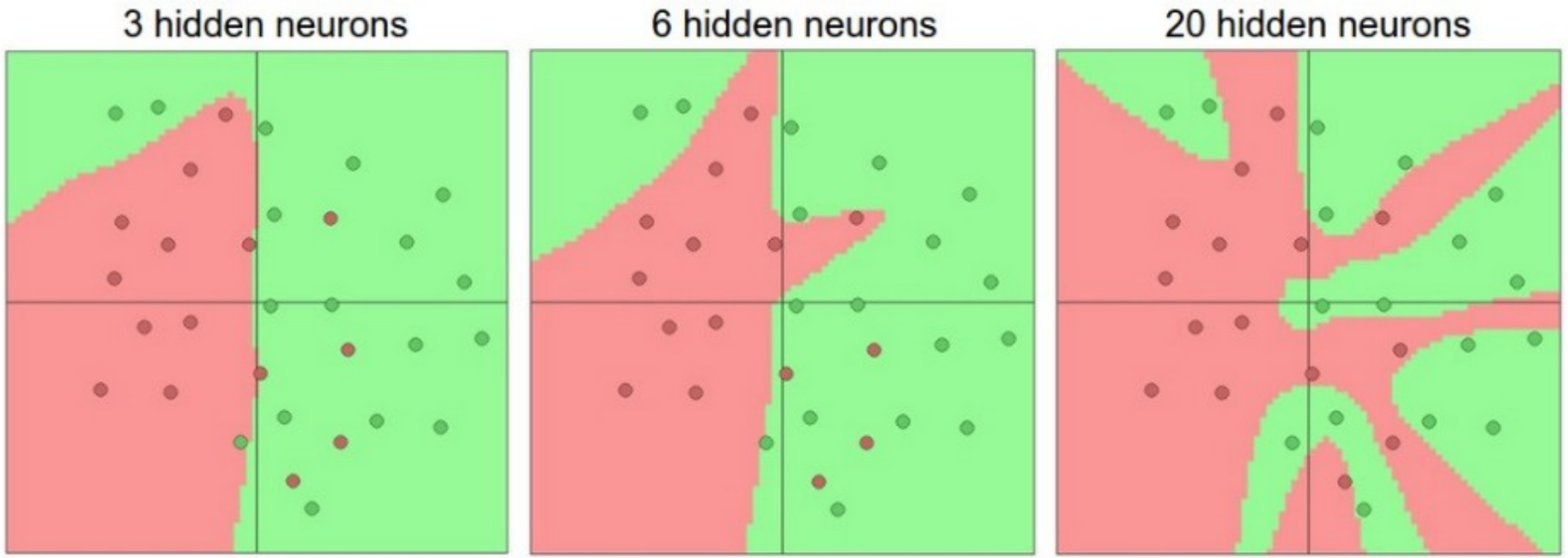


- The capacity of the network increases with more hidden units and more hidden layers

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper

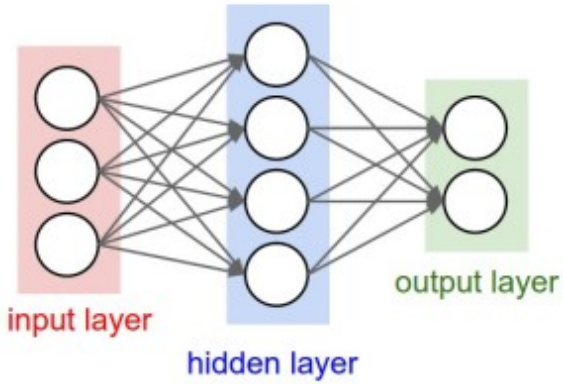


- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper? Read e.g.,: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana, Paper: paper]

Neural Networks

- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning

Forward Pass: What does the Network Compute?



- Output of the network can be written as:

$$h_j(\mathbf{x}) = f\left(v_{j0} + \sum_{i=1}^{D} x_i v_{ji}\right)$$

$$o_k(\mathbf{x}) = g\left(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}\right)$$

(j indexing hidden units, k indexing the output units, D number of inputs)

- Activation functions f, g: sigmoid/logistic, tanh, or rectified linear (ReLU)

$$\sigma(z) = \frac{1}{1+\exp(-z)}, \quad \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}, \quad \mathrm{ReLU}(z) = \max(0, z)$$

Example Application

- Classify image of handwritten digit (32x32 pixels): 4 vs non-4

- How would you build your network?
- For example, use one hidden layer and the sigmoid activation function:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

- How can we train the network, that is, adjust all the parameters $\mathbf{w}$?

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

  where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$
- Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

  where η is the learning rate (and E is error/loss)

Useful Derivatives

name	function	derivative
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

Training Neural Networks: Back-propagation

- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

Key Idea behind Backpropagation

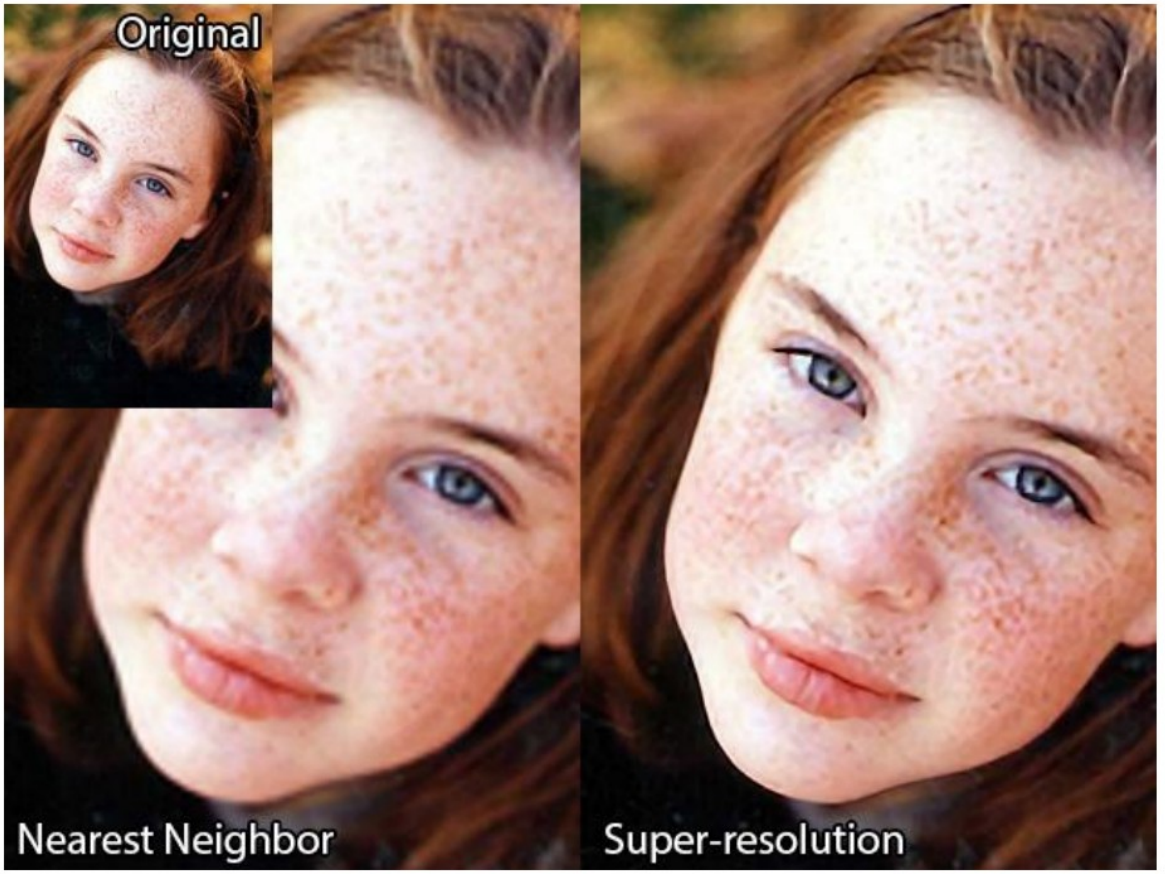
- We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit
- This is just the chain rule!

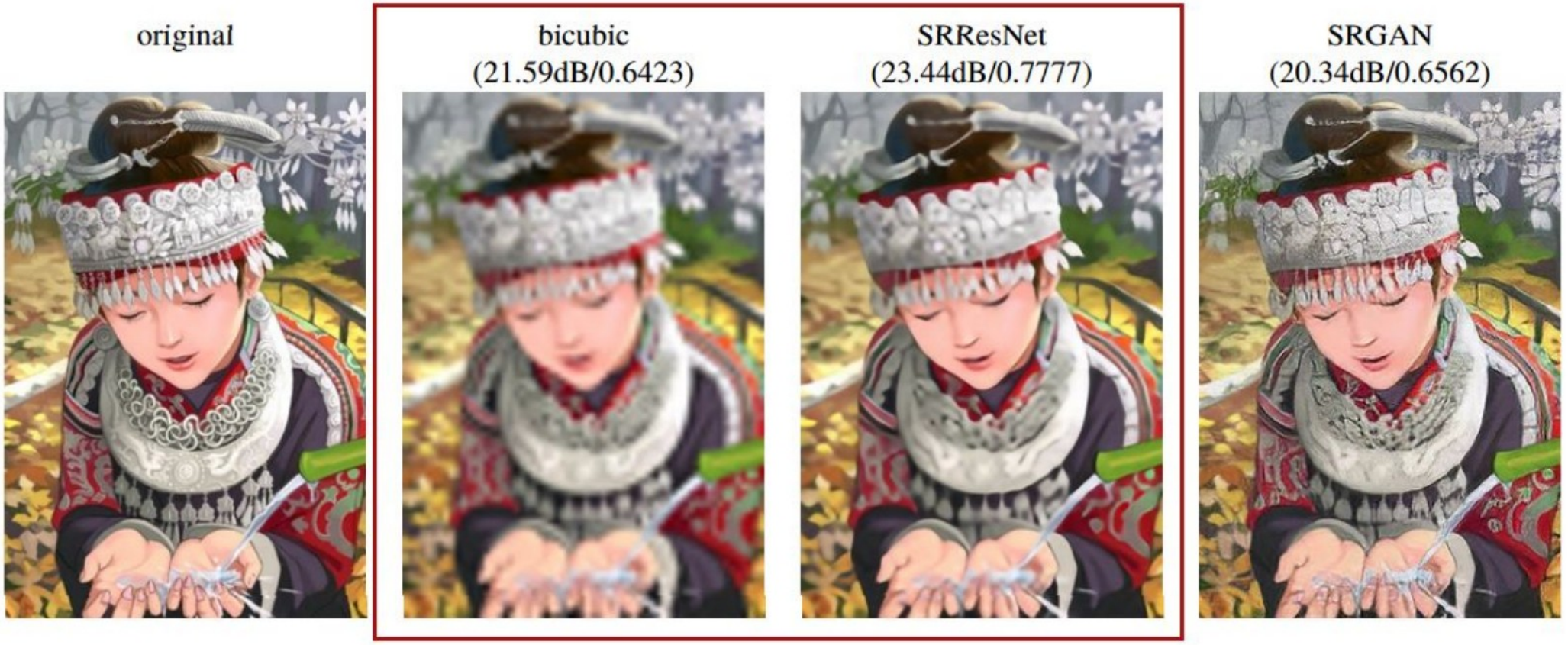
Using GANs for Single Image Super-Resolution

Problem

How do we get a high resolution (HR) image from just one (LR) lower resolution image?

Answer: We use super-resolution (SR) techniques.




original
bicubic
(21.59dB/0.6423)
SRResNet
(23.44dB/0.7777)
SRGAN
(20.34dB/0.6562)

HOMEWORK:

1) Present an advanced machine learning algorithm, e.g. LSTM, RNN, GANs, ...

2) Explain the model in detail

3) What are interesting and state-of-the-art applications of these models

4) prepare a ppt presentation of your results

- End of Basic Machine Learning -

- 2) Beginning of Basics of Neural Networks -

Neural Networks and Deep Learning

First of all, you already know what a neural network is, and you already know how to build such a model. Yes, it's logistic regression! As a matter of fact, the logistic regression model, or rather its generalization for multiclass classification, called the softmax regression model, is a standard unit in a neural network.

Neural Networks

If you understood linear regression, logistic regression, and gradient descent, understanding neural networks would not be a problem.

A neural network (NN), just like a regression or an SVM model, is a mathematical function:

$$y = f_{NN}(\mathbf{x}).$$

The function f_{NN} has a particular form: it's a *nested function*. You have probably already heard of neural network **layers**. So, for a 3-layer neural network that returns a scalar, f_{NN} looks like this:

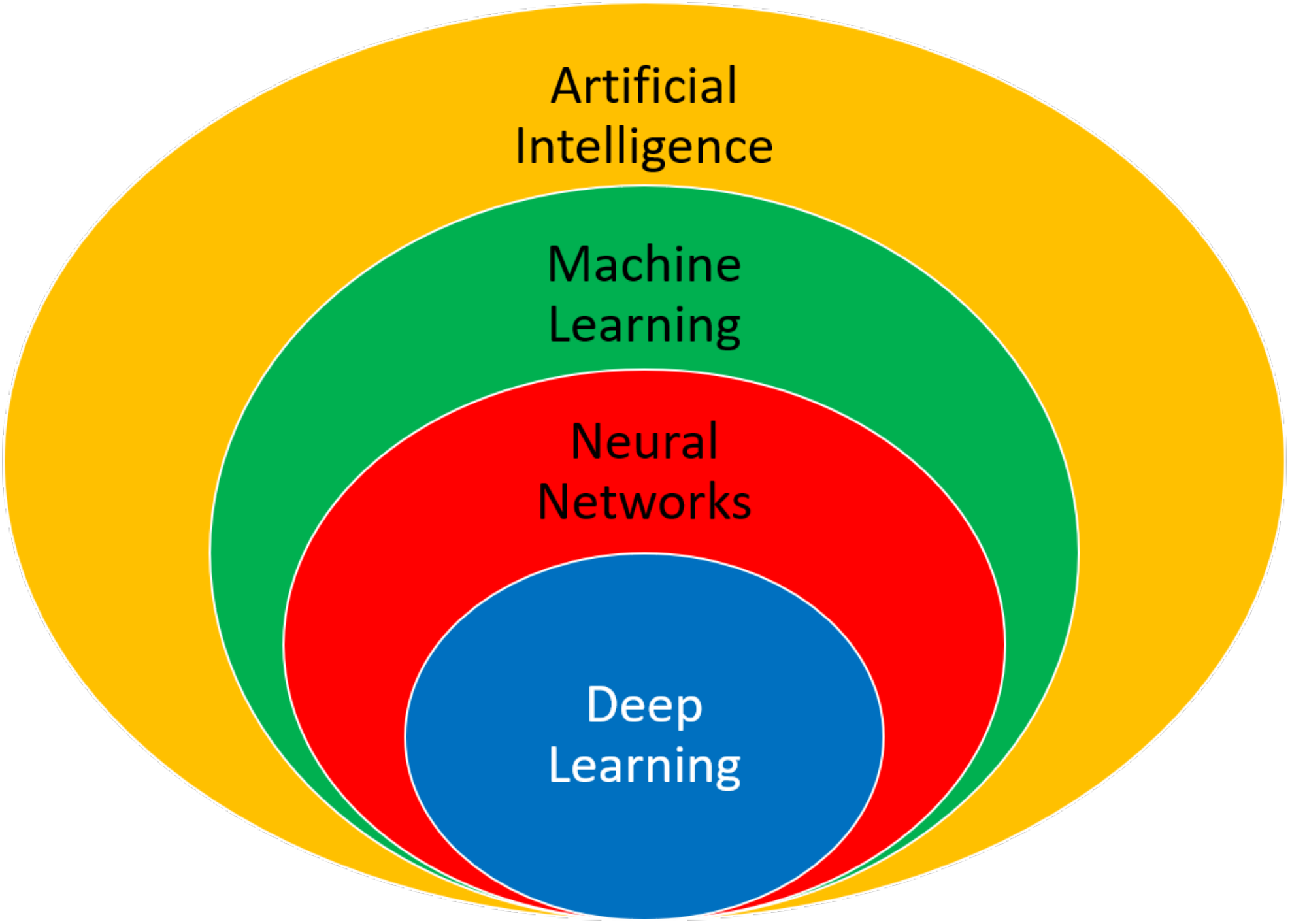
$$y = f_{NN}(\mathbf{x}) = f_3(\boldsymbol{f}_2(\boldsymbol{f}_1(\mathbf{x}))).$$

In the above equation, $\boldsymbol{f}_2$, $\boldsymbol{f}_3$ are vector functions of the following form:

$$\boldsymbol{f}_l(\mathbf{z}) \stackrel{\text{def}}{=} \boldsymbol{g}_l(\mathbf{W}_l\mathbf{z} + \mathbf{b}_l), \tag{1}$$

where l is called the layer index and can span from 1 to any number of layers. The function $\boldsymbol{g}_l$ is called an **activation function**. It is a fixed, usually nonlinear function chosen by the data analyst before the learning is started. The parameters $\mathbf{W}_l$ (a matrix) and $\mathbf{b}_l$ (a vector) for each layer are learned using the familiar gradient descent by optimizing, depending on the task, a particular cost function (such as MSE). Compare eq. 1 with the equation for logistic regression, where you replace $\boldsymbol{g}_l$ by the sigmoid function, and you will not see any difference. The function f_1 is a scalar function for the regression task, but can also be a vector function depending on your problem.

You may probably wonder why a matrix $\mathbf{W}_l$ is used and not a vector $\mathbf{w}_l$. The reason is that $\boldsymbol{g}_l$ is a vector function. Each row $\mathbf{w}_{l,u}$ (u for unit) of the matrix $\mathbf{W}_l$ is a vector of the same dimensionality as $\mathbf{z}$. Let $a_{l,u} = \mathbf{w}_{l,u}\mathbf{z} + b_{l,u}$. The output of $\boldsymbol{f}_l(\mathbf{z})$ is a vector $[g_l(a_{l,1}), g_l(a_{l,2}), \ldots, g_l(a_{l,size_l})]$, where g_l is some scalar function[1], and $size_l$ is the number of units in layer l. To make it more concrete, let's consider one architecture of neural networks called **multilayer perceptron** and often referred to as a **vanilla neural network**.


Artificial
Intelligence
Machine
Learning
Neural
Networks
Deep
Learning

Multilayer Perceptron Example

We have a closer look at one particular configuration of neural networks called **feed-forward neural networks** (FFNN), and more specifically the architecture called a **multilayer perceptron** (MLP). As an illustration, we consider an MLP with three layers. Our network takes a two-dimensional feature vector as input and outputs a number. This FFNN can be a regression or a classification model, depending on the activation function used in the third, output layer.

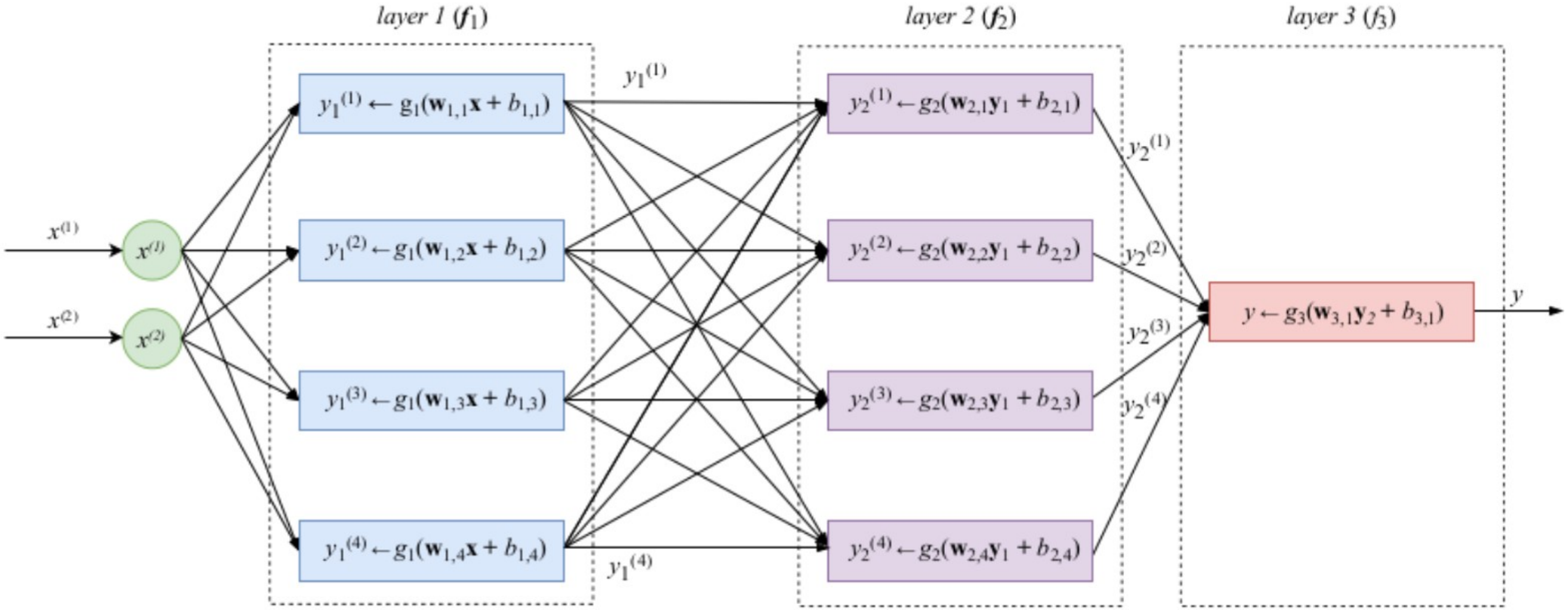
Our MLP is depicted in fig. 1. The neural network is represented graphically as a connected combination of **units** logically organized into one or more **layers**. Each unit is represented by either a circle or a rectangle. The inbound arrow represents an input of a unit and indicates where this input came from. The outbound arrow indicates the output of a unit.

The output of each unit is the result of the mathematical operation written inside the circle or a rectangle. Circle units don't do anything with the input; they just send their input directly to the output.

The following happens in each rectangle unit. Firstly, all inputs of the unit are joined together to form an input vector. Then the unit applies a linear transformation to the input vector, exactly like linear regression model does with its input feature vector. Finally, the unit applies an activation function g to the result of the linear transformation and obtains the output value, a real number. In a vanilla FFNN, the output value of a unit of some layer becomes an input value of each of the units of the subsequent layer.

In fig. 1, the activation function g_l has one index: l, the index of the layer the unit belongs to. Usually, all units of a layer use the same activation function, but it's not strictly necessary. Each layer can have a different number of units. Each unit has its own parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$, where u is the index of the unit, and l is the index of the layer. The vector $\mathbf{y}_{l-1}$ in each unit is defined as $[y_{l-1}^{(1)}, y_{l-1}^{(2)}, y_{l-1}^{(3)}, y_{l-1}^{(4)}]$. The vector $\mathbf{x}$ in the first layer is defined as $[x^{(1)}, \ldots, x^{(D)}]$.

As you can see in fig. 1, in multilayer perceptron all outputs of one layer are connected to each input of the succeeding layer. This architecture is called **fully-connected**. A neural network can contain **fully-connected layers**. Those are the layers whose units receive as inputs the outputs of each of the units of the previous layer.



Feed-Forward Neural Network Architecture

If we want to solve a regression or a classification problem discussed in previous chapters, the last (the rightmost) layer of a neural network usually contains only one unit. If the activation function g_{last} of the last unit is linear, then the neural network is a regression model. If the g_{last} is a logistic function, the neural network is a binary classification model.

The data analyst is free to choose any mathematical function as $g_{l,u}$, assuming it's differentiable[2]. The latter property is essential for gradient descent, which is used to find the values of the parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$ for all l and u. The primary purpose of having nonlinear components in the function f_{NN} is to allow the neural network to approximate nonlinear functions. Without nonlinearities, f_{NN} would be linear, no matter how many layers it has. The reason is that $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$ is a linear function and a linear function of a linear function is also a linear function.

Popular choices of activation functions are the logistic function, already known to you, as well as **TanH** and **ReLU**. The former is the hyperbolic tangent function, similar to the logistic function but ranging from -1 to 1 (without reaching them). The latter is the rectified linear unit function, which equals to zero when its input z is negative and to z otherwise:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

$$relu(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}.$$

As I said above, $\mathbf{W}_l$ in the expression $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$, is a matrix, while $\mathbf{b}_l$ is a vector. That looks different from linear regression's $\mathbf{wz} + b$. In matrix $\mathbf{W}_l$, each row u corresponds to a vector of parameters $\mathbf{w}_{l,u}$. The dimensionality of the vector $\mathbf{w}_{l,u}$ equals to the number of units in the layer $l-1$. The operation $\mathbf{W}_l\mathbf{z}$ results in a vector $\mathbf{a}_l \stackrel{\text{def}}{=} [\mathbf{w}_{l,1}\mathbf{z}, \mathbf{w}_{l,2}\mathbf{z}, \ldots, \mathbf{w}_{l,size_l}\mathbf{z}]$. Then the sum $\mathbf{a}_l + \mathbf{b}_l$ gives a $size_l$-dimensional vector $\mathbf{c}_l$. Finally, the function $\boldsymbol{g}_l(\mathbf{c}_l)$ produces the vector $\mathbf{y}_l \stackrel{\text{def}}{=} [y_l^{(1)}, y_l^{(2)}, \ldots, y_l^{(size_l)}]$ as output.

Deep Learning

- Complex models with large number of parameters
  - Hierarchical representations
  - More parameters = more accurate on training data
  - Simple learning rule for training (gradient-based).
- Lots of data
  - Needed to get better generalization performance.
  - High-dimensional input need exponentially many inputs (curse of dimensionality).
- Lots of computing power: GPGPU, etc.
  - Training large networks can be time consuming.

The Rise of Deep Learning

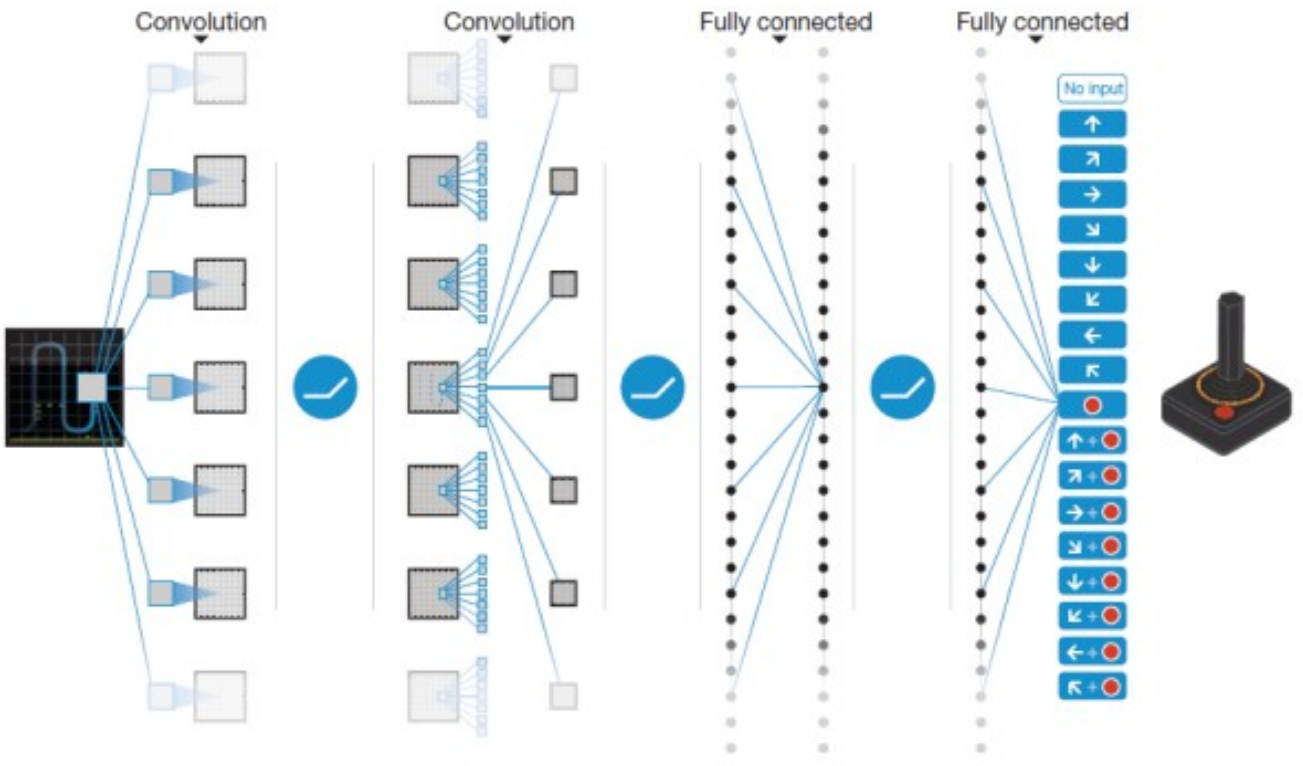
Made popular in recent years

- Geoffrey Hinton et al. (2006).
- Andrew Ng & Jeff Dean (Google Brain team, 2012).
- Schmidhuber et al.'s deep neural networks (won many competitions and in some cases showed super human performance; 2011–). Recurrent neural networks using LSTM (Long Short-Term Memory).
- Google Deep Mind: Atari 2600 games (2015), AlphaGo (2016).
- ICLR, International Conference on Learning Representations: First meeting in 2013.

Current Trends

- Deep belief networks (based on Boltzmann machine)
- Convolutional neural networks
- Deep Q-learning Network (extensions to reinforcement learning)
- Deep recurrent neural networks using (LSTM)
- Applications to diverse domains.
  - Vision, speech, video, NLP, etc.
- Lots of open source tools available.

Deep Q-Network (DQN)



Google Deep Mind (Mnih et al. *Nature* 2015).

- Latest application of deep learning to a *reinforcement learning* domain (Q as in Q-learning).
- Applied to *Atari 2600* video game playing.

Deep Learning

Deep learning refers to training neural networks with more than two non-output layers. In the past, it became more difficult to train such networks as the number of layers grew. The two biggest challenges were referred to as the problems of **exploding gradient** and **vanishing gradient** as gradient descent was used to train the network parameters.

While the problem of exploding gradient was easier to deal with by applying simple techniques like **gradient clipping** and L1 or L2 regularization, the problem of vanishing gradient remained intractable for decades.

What is vanishing gradient and why does it arise? To update the values of parameters in neural networks the algorithm called **backpropagation** is typically used. Backpropagation is an efficient algorithm for computing gradients on neural networks using the chain rule. In Chapter 4, we have already seen how the chain rule is used to calculate partial derivatives of a complex function. During gradient descent, the neural network's parameters receive an update proportional to the partial derivative of the cost function with respect to the current parameter in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing some parameters from changing their value. In the worst case, this may completely stop the neural network from further training.

Traditional activation functions, such as the hyperbolic tangent function I mentioned above, have gradients in the range $(0, 1)$, and backpropagation computes gradients by the chain rule. That has the effect of multiplying n of these small numbers to compute gradients of the earlier (leftmost) layers in an n-layer network, meaning that the gradient decreases exponentially with n. That results in the effect that the earlier layers train very slowly, if at all.

However, the modern implementations of neural network learning algorithms allow you to effectively train very deep neural networks (up to hundreds of layers). The ReLU activation function suffers much less from the problem of vanishing gradient. Also, **long short-term memory** (LSTM) networks, which we consider below, as well as such techniques as **skip connections** used in **residual neural networks** allow you to train even deeper neural networks, with thousands of layers.

Therefore, today, since the problems of vanishing and exploding gradient are mostly solved (or their effect diminished) to a great extent, the term "deep learning" refers to training neural networks using the modern algorithmic and mathematical toolkit independently of how deep the neural network is. In practice, many business problems can be solved with neural networks having 2-3 layers between the input and output layers. The layers that are neither input nor output are often called **hidden layers**.

Convolutional Neural Network

You may have noticed that the number of parameters an MLP can have grows very fast as you make your network bigger. More specifically, as you add one layer, you add $size_l(size_{l-1}+1)$ parameters (our matrix $\mathbf{W}_l$ plus the vector $\mathbf{b}_l$). That means that if you add another 1000-unit layer to an existing neural network, then you add more than 1 million additional parameters to your model. Optimizing such big models is a very computationally intensive problem.

When our training examples are images, the input is very high-dimensional. If you want to learn to classify images using an MLP, the optimization problem is likely to become intractable.

A **convolutional neural network** (CNN) is a special kind of FFNN that significantly reduces the number of parameters in a deep neural network with many units without losing too much in the quality of the model. CNNs have found applications in image and text processing where they beat many previously established benchmarks.

Because CNNs were invented with image processing in mind, I explain them on the image classification example.

You may have noticed that in images, pixels that are close to one another usually represent the same type of information: sky, water, leaves, fur, bricks, and so on. The exception from the rule are the edges: the parts of an image where two different objects "touch" one another.

So, if we can train the neural network to recognize regions of the same information as well as the edges, then this knowledge would allow the neural network to predict the object represented in the image. For example, if the neural network detected multiple skin regions and edges that look like parts of an oval with skin-like tone on the inside and bluish tone on the outside, then it is very likely that there's a face on the sky background. If our goal is to detect people on pictures, the neural network will most likely succeed in predicting a person in this picture.

Having in mind that the most important information in the image is local, we can split the image into square patches using a moving window approach[3]. We can then train multiple smaller regression models at once, each small regression model receiving a square patch as input. The goal of each small regression model is to learn to detect a specific kind of pattern in the input patch. For example, one small regression model will learn to detect the sky; another one will detect the grass, the third one will detect edges of a building, and so on.

In CNNs, a small regression model looks like the one in fig. 1, but it only has the layer 1 and doesn't have layers 2 and 3. To detect some pattern, a small regression model has to learn the parameters of a matrix F (for "filter") of size $p \times p$, where p is the size of a patch. Let's assume, for simplicity, that the input image is back and white, with 1 representing black and 0 representing white pixels. Assume also that our patches are 3 by 3 pixels ($p = 3$). Some patch could then look like the following matrix P (for "patch"):

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

The above patch represents a pattern that looks like a cross. The small regression model that will detect such patterns (and only them) would need to learn a 3 by 3 parameter matrix F where parameters at positions corresponding to the 1s in the input patch would be positive numbers, while the parameters in positions corresponding to 0s would be close to zero. If we calculate the dot-product between matrices P and F and then sum all values from the resulting vector, the value we obtain is higher the more similar F is to P. For instance, assume that F looks like this:

$$F = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 4 & 1 \\ 0 & 3 & 0 \end{bmatrix}.$$

Then,

$$P \cdot F = [0 \cdot 0 + 2 \cdot 1 + 0 \cdot 0, 2 \cdot 1 + 4 \cdot 1 + 3 \cdot 1, 3 \cdot 0 + 1 \cdot 1 + 0 \cdot 1] = [2, 9, 1].$$

Then the sum of all elements of the above vector is $2 + 9 + 1 = 12$. This operation — the dot product between a patch and a filter and then summing the values — is called **convolution**.

If our input patch P had a different patten, for example, that of a letter T,

$$P = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

then the convolution would give a lower result: $0 + 9 + 0 = 9$. So, you can see the more the patch "looks" like the filter, the higher the value of the convolution operation is. For convenience, there's also a bias parameter b associated with each filter F which is added to the result of a convolution before applying the nonlinearity.

One layer of a CNN consists of multiple convolution filters (each with its own bias parameter), just like one layer in a vanilla FFNN consists of multiple units. Each filter of the first (leftmost) layer slides — or *convolves* — across the input image, left to right, top to bottom, and convolution is computed at each iteration.

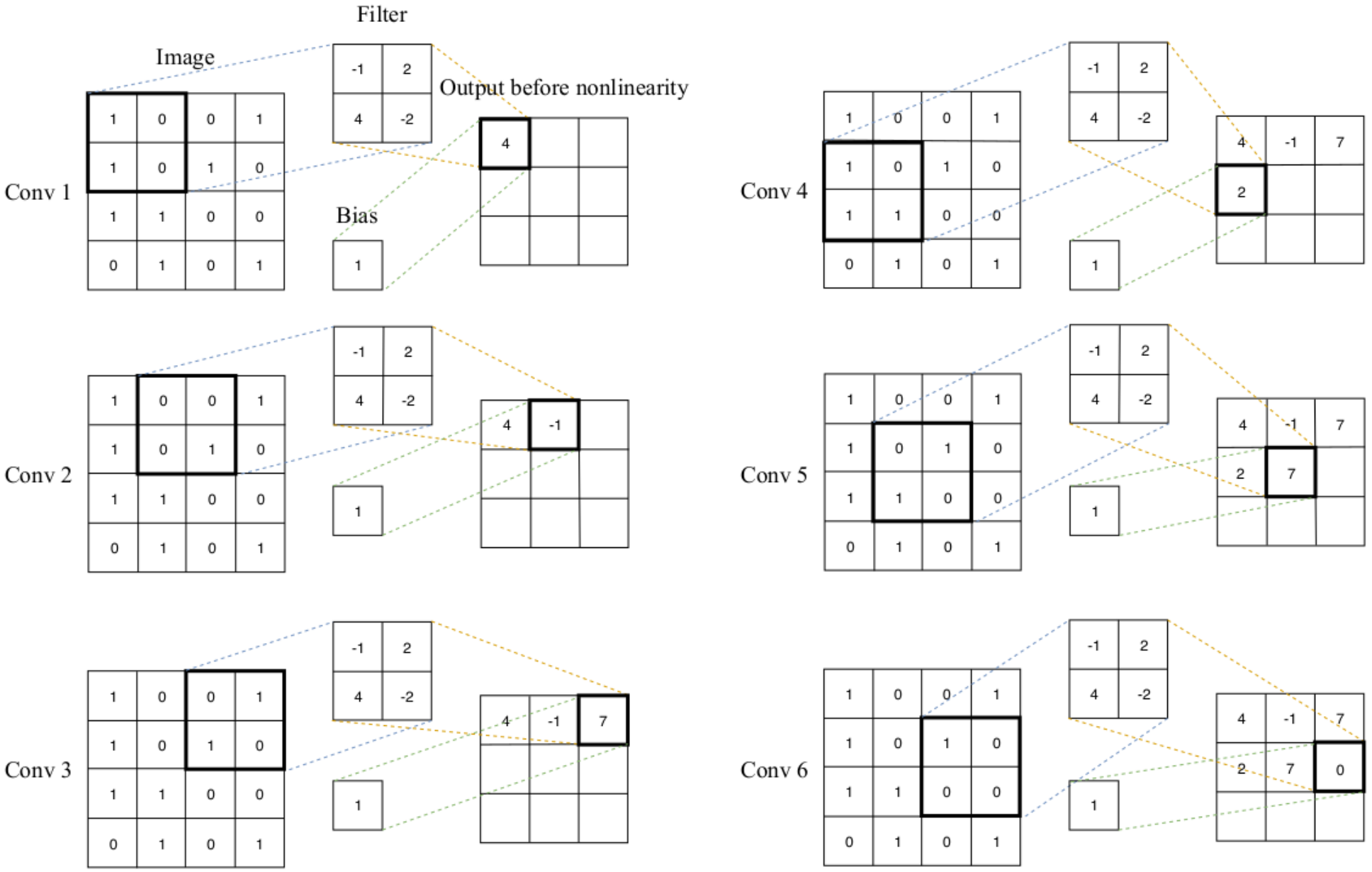


Figure 2: A filter convolving across an image.

An illustration of the process is given in fig. 2 where 6 steps of a filter convolving across an image are shown.

The numbers in the filter matrix, for each filter F in each layer, as well as the value of the bias term b, are found by the gradient descent with backpropagation, based on data by minimizing the cost function.

A nonlinearity is applied to the sum of the convolution and the bias term. Typically, the ReLU activation function is used in all hidden layers. The activation function of the output layer depends on the task.

Since we can have $size_l$ filters in each layer l, the output of the convolution layer l would consist of $size_l$ matrices, one for each filter.

If the CNN has one convolution layer following another convolution layer, then the subsequent layer $l+1$ treats the output of the preceding layer l as a collection of $size_l$ image matrices. Such a collection is called a *volume*. Each filter of layer $l+1$ convolves the whole volume. The convolution of a patch of a volume is simply the sum of convolutions of the corresponding patches of individual matrices the volume consists of.

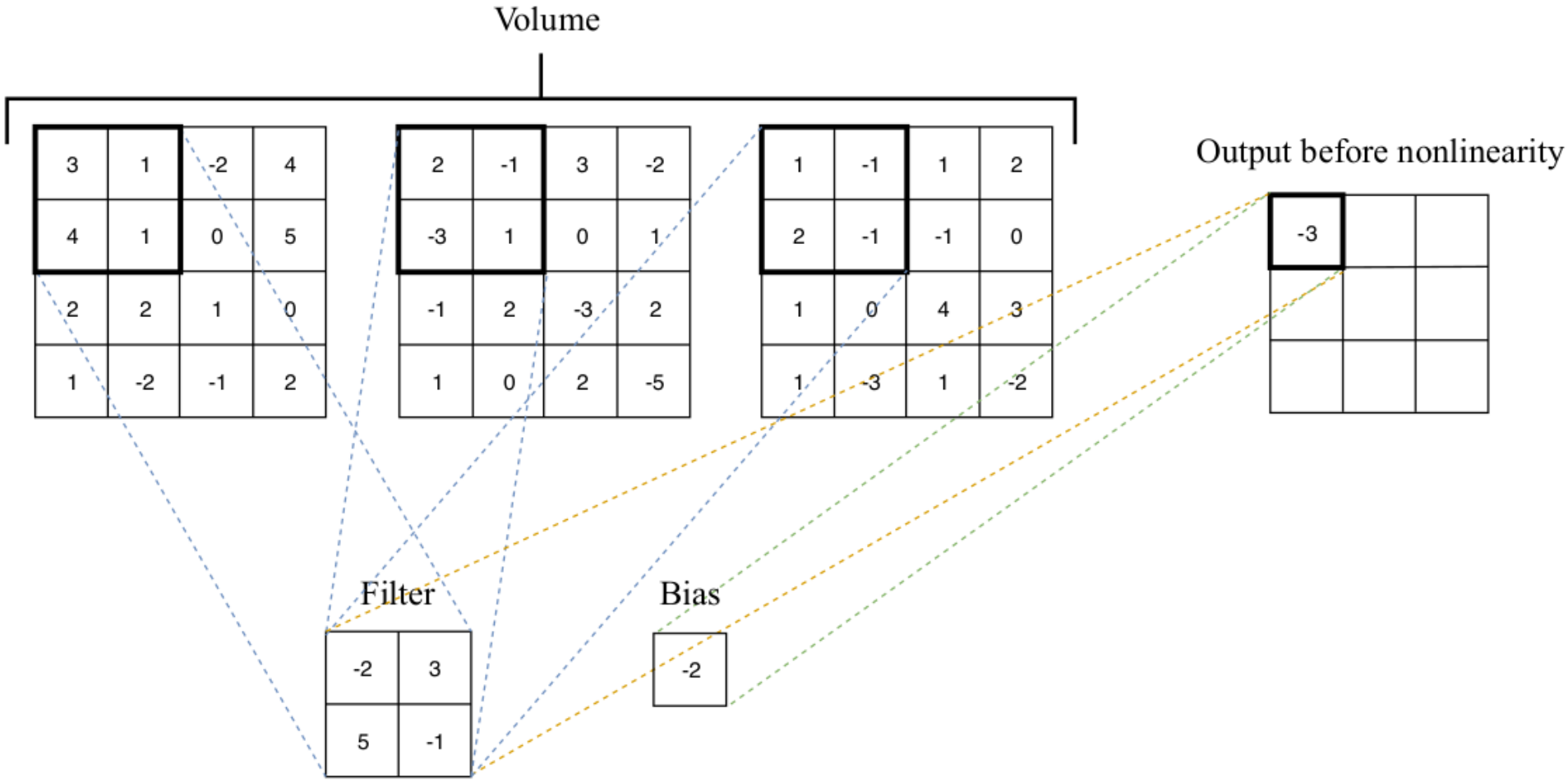


Figure 3: Convolution of a volume consisting of three matrices.

An example of a convolution of a patch of a volume consisting of three matrices is shown in fig. 3. The value of the convolution, -3, was obtained as $(-2 \cdot 3 + 3 \cdot 1 + 5 \cdot 4 + -1 \cdot 1) + (-2 \cdot 2 + 3 \cdot (-1) + 5 \cdot (-3) + -1 \cdot 1) + (-2 \cdot 1 + 3 \cdot (-1) + 5 \cdot 2 + -1 \cdot (-1)) + (-2)$.

In computer vision, CNNs often get volumes as input, since an image is usually represented by three channels: R, G, and B, each channel being a monochrome picture.

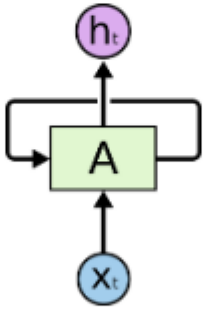
By now, you should have a good high-level understanding of the CNN architecture. We didn't discuss some essential features of CNNs though, such as strides, padding, and pooling. Strides and padding are two important hyperparameters of the convolution filter and the sliding window, while pooling is a technique that works very well in practice by reducing the number of parameters of a CNN even more.

Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

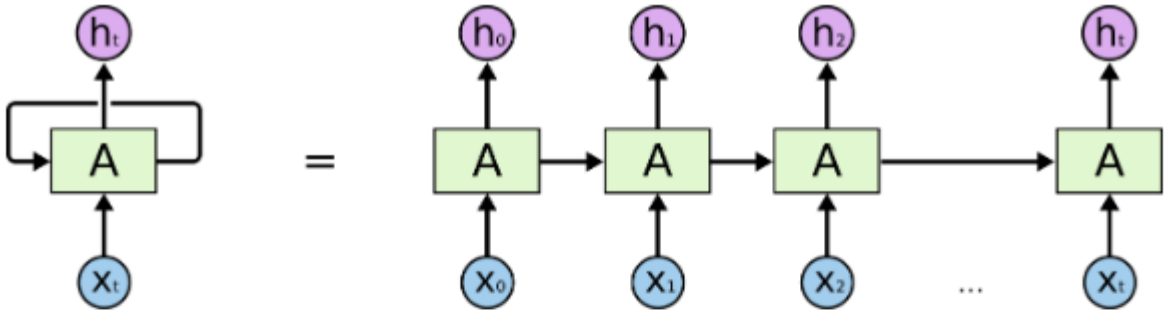
Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



An unrolled recurrent neural network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

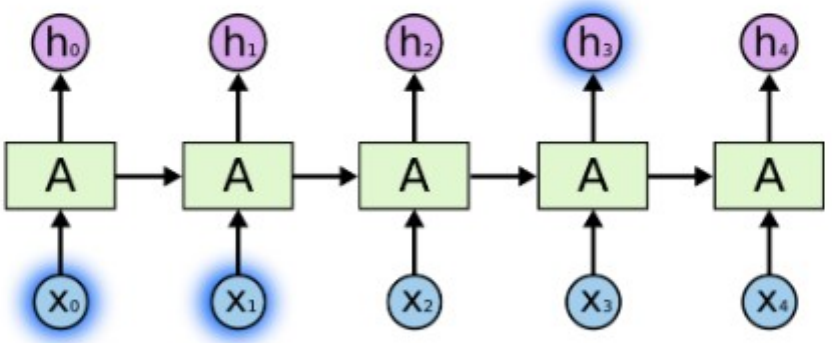
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning... The list goes on. I'll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy's excellent blog post, The Unreasonable Effectiveness of Recurrent Neural Networks. But they really are pretty amazing.

Essential to these successes is the use of "LSTMs," a very special kind of recurrent neural network which works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It's these LSTMs that this essay will explore.

The Problem of Long-Term Dependencies

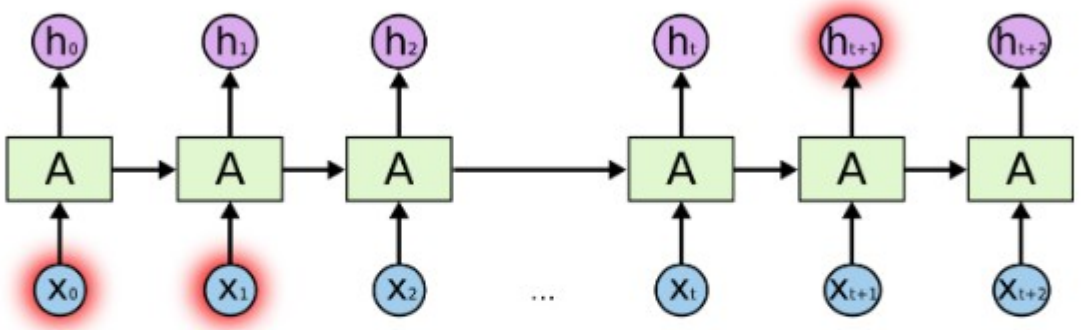
One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the *sky*," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

HOMEWORK:

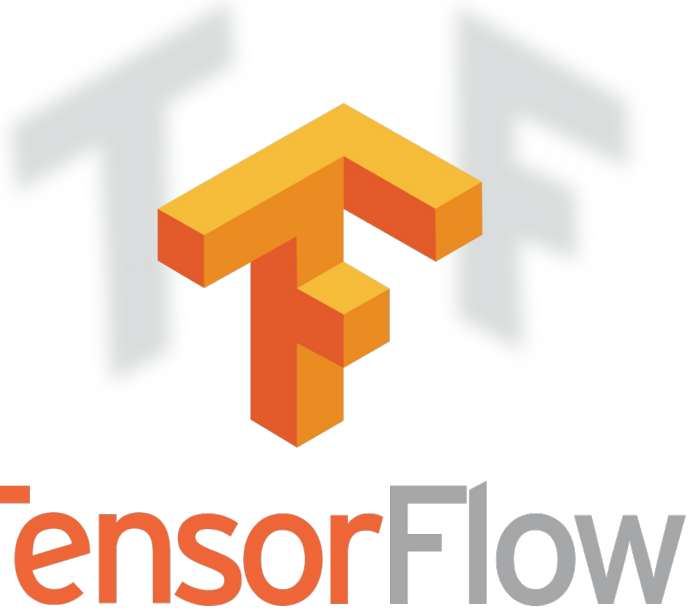
1) Implement a CNN, RNN in Python.

2) Apply the algorithms to real world problems of your choice.

3) prepare a ppt presentation of your results.

- End of Neural Networks -

- 3) Beginning of TensorFlow Lecture -



Google Machine Learning Tools

1st Generation : *DistBelief*

- *Dean et al. 2011*
- *Major Output Products*
  - *Inception (Image Categorization)*
  - *Google Search*
  - *Google Translate*
  - *Google Photos*

2nd Generation : *TensorFlow*

- *Dean et al. 2015 (November, 1st)*
- *Most of DistBelief users at Google have already switched to TensorFlow*

An open-source library by Google:

TensorFlow:

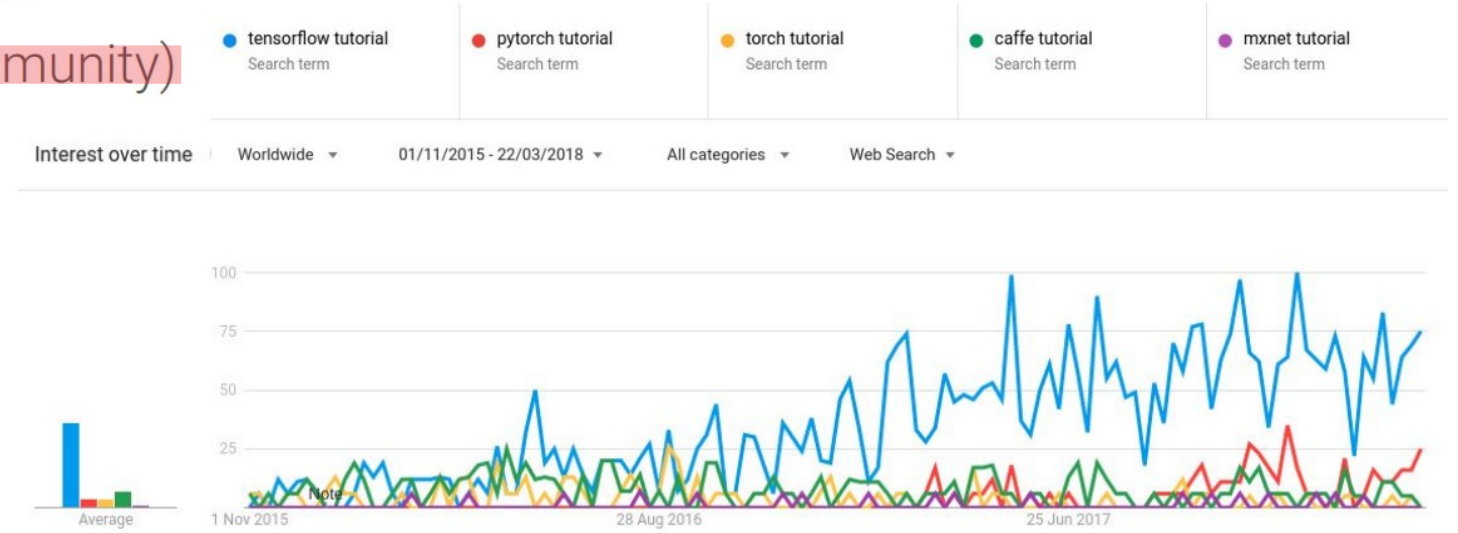
Large-Scale Machine Learning on Heterogeneous Distributed Systems

(Preliminary White Paper, November 9, 2015)

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,
Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,
Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser,
Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray,
Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar,
Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals,
Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng
Google Research*

Why TensorFlow

- “TensorFlow™ is an open source software library for numerical computation using data flow graphs.”
- One of many frameworks for deep learning computations
- Scalable and flexible
- Popular (= big community)



- Official website: https://www.tensorflow.org/

If want to master every details.

- *Deep Learning with Python* by Francois Chollet

Focus on Keras

- Hands-On Machine Learning with Scikit-Learn and TensorFlow

Tensorflow part is somewhat outdated.

Core Functionalities:

- Augmented tensor operations (nearly identical to numpy)

Seamless interfaces with existing programs.

- Automatic differentiation

The very core of Optimization based algorithms.

- Parallel(CPU/GPU/TPU) and Distributed(multi-machine) Computation

Essential for large(industrial level) applications.

Implemented in C++. Highly Efficient.

3 levels of tensorflow:

- Primitive tensorflow: lowest, finest control and most flexible

Suitable for most machine learning and deep learning algorithms.

- Keras(Mostly for deep learning):highest, most convenient to use, lack flexibility

- Tensorflow layers (Mostly for deep learning): somewhere at the middle.

General pipeline:

- Define inputs and variable tensors(weights/parameters).

*Keras will take care of these for you.

- Define computation graphs from inputs tensors to output tensors.

- Define loss function and optimizer

Once the loss is defined, the optimizer will compute the gradient for you!

- Execute the graphs.

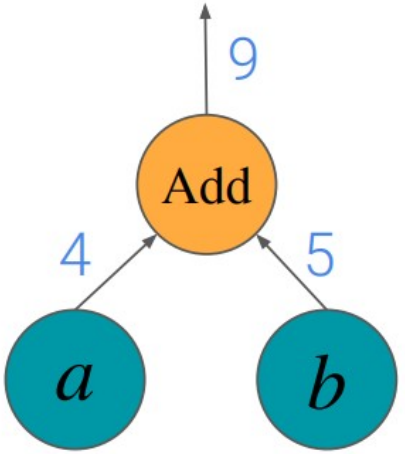
*Keras will take care of this for you as well

Basic Code Structure

- View functions as computational graphs
- First build a computational **graph**, and then use a **session** to execute operations in the graph
- This is the basic approach, there is also a dynamic approach implemented in the recently introduced eager mode

Basic Code Structure - Graphs

- Nodes are operators (ops), variables, and constants
- Edges are tensors
  - 0-d is a scalar
  - 1-d is a vector
  - 2-d is a matrix
  - Etc.
- TensorFlow = Tensor + Flow = Data + Flow



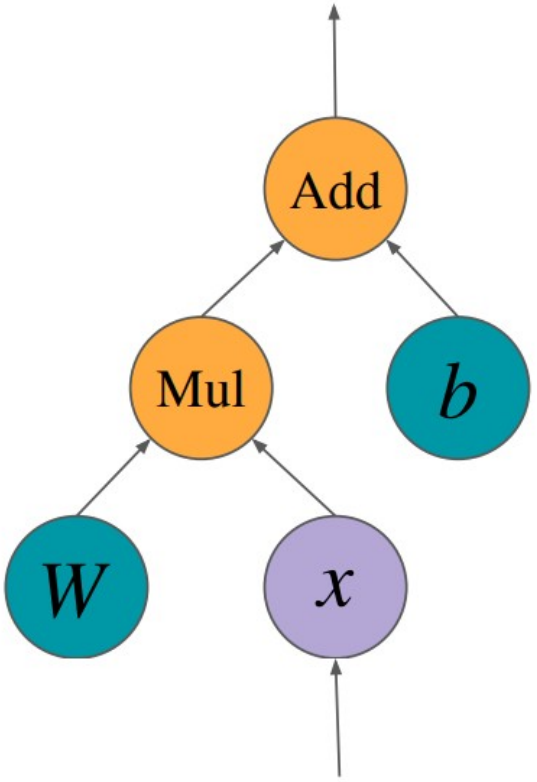
Basic Code Structure - Graphs

- **Constants** are fixed value tensors - not trainable
- **Variables** are tensors initialized in a session - trainable
- **Placeholders** are tensors of values that are unknown during the graph construction, but passed as input during a session
- **Ops** are functions on tensors

Basic Code Structure - Graphs

$$\hat{y} = Wx + b$$

variable ops placeholder variable



Basic Code Structure - Sessions

- Session is the runtime environment of a graph, where operations are executed, and tensors are evaluated

```
>>> import tensorflow as tf
>>> a = tf.constant(4)
>>> b = tf.constant(5)
>>> add_op = tf.add(a, b)
>>> print(add_op)
Tensor("Add:0", shape=(), dtype=int32)
```

```
>>> import tensorflow as tf
>>> a = tf.constant(4)
>>> b = tf.constant(5)
>>> add_op = tf.add(a, b)
>>> with tf.Session() as session:
...     print(session.run(add_op))
...
9
```

- `a.eval()` is equivalent to `session.run(a)`, but in general, “eval” is limited to executions of a single op and ops that returns a value
- Upon op execution, only the subgraph required for calculating its value is evaluated

Linear Regression with TensorFlow

Credit: Derek Chia

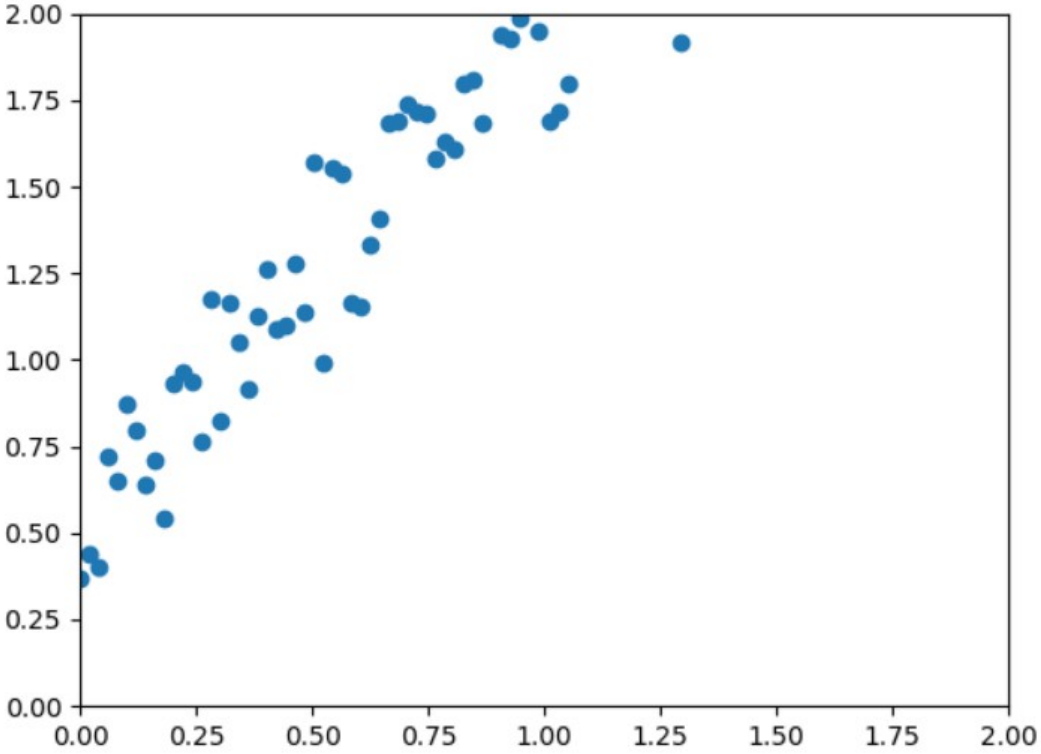
$$y = Wx + b$$

linear regression equation

Looking at the equation of linear regression above, we begin by constructing a graph that learns the gradient of the slope (**W**) and bias (**b**) through multiple iterations. In each iteration, we aim to close up the gap (loss) by comparing input **y** to the **predicted y**. This means to say, we want to modify **W** and **b** such that inputs of **x** will give us the **y** we want. Solving the linear regression is also known as finding the line of best fit or trend line.

Generating Dataset

```
1  import numpy as np
2  import tensorflow as tf
3  import matplotlib.pyplot as plt
4
5  def generate_dataset():
6   x_batch = np.linspace(0, 2, 100)
7   y_batch = 1.5 * x_batch + np.random.randn(*x_batch.shape) * 0.2 + 0.5
8   return x_batch, y_batch
```



Constructing the Graph

```
def linear_regression():
  x = tf.placeholder(tf.float32, shape=(None, ), name='x')
  y = tf.placeholder(tf.float32, shape=(None, ), name='y')

  with tf.variable_scope('lreg') as scope:
    w = tf.Variable(np.random.normal(), name='W')
    b = tf.Variable(np.random.normal(), name='b')

    y_pred = tf.add(tf.multiply(w, x), b)

    loss = tf.reduce_mean(tf.square(y_pred - y))

  return x, y, y_pred, loss
```

```
x = tf.placeholder(tf.float32, shape=(None, ), name='x')
y = tf.placeholder(tf.float32, shape=(None, ), name='y')
```

Next, we construct the TensorFlow graph that helps us compute **W** and **b**. This is done in the function `linear_regression()`. In our formula `y = Wx + b`, the **x** and **y** are nodes represented as TensorFlow's placeholder. Declaring **x** and **y** as placeholders mean that we need to pass in values at a later time — we will revisit this in the following section. Note that we are now merely constructing the graph and not running it (TensorFlow has lazy evaluation).

In the first argument of `tf.placeholder`, we define the data type as float32 — a common data type in placeholder. The second argument is the shape of the placeholder set to `None` as we want it to be determined during training time. The third argument lets us set the name for the placeholder.

> *tf.placeholder - A placeholder is simply a variable that we will assign data to at a later date. It allows us to create our operations and build our computation graph, without needing the data. In TensorFlow terminology, we then feed data into the graph through these placeholders.*

[line 5] `with tf.variable_scope('lreg') as scope:`

This line defines the variable scope for our variables in line 6 and 7. In short, Variable scope allows naming of variable in a hierarchy way to avoid name clashes. To elaborate, it is a mechanism in TensorFlow that allows variables to be shared in different parts of the graph without passing references to the variable around. Note that even though we do not reuse variables here, it is a good practice to name them appropriately.

[line 6] `w = tf.Variable(np.random.normal(), name='W')`

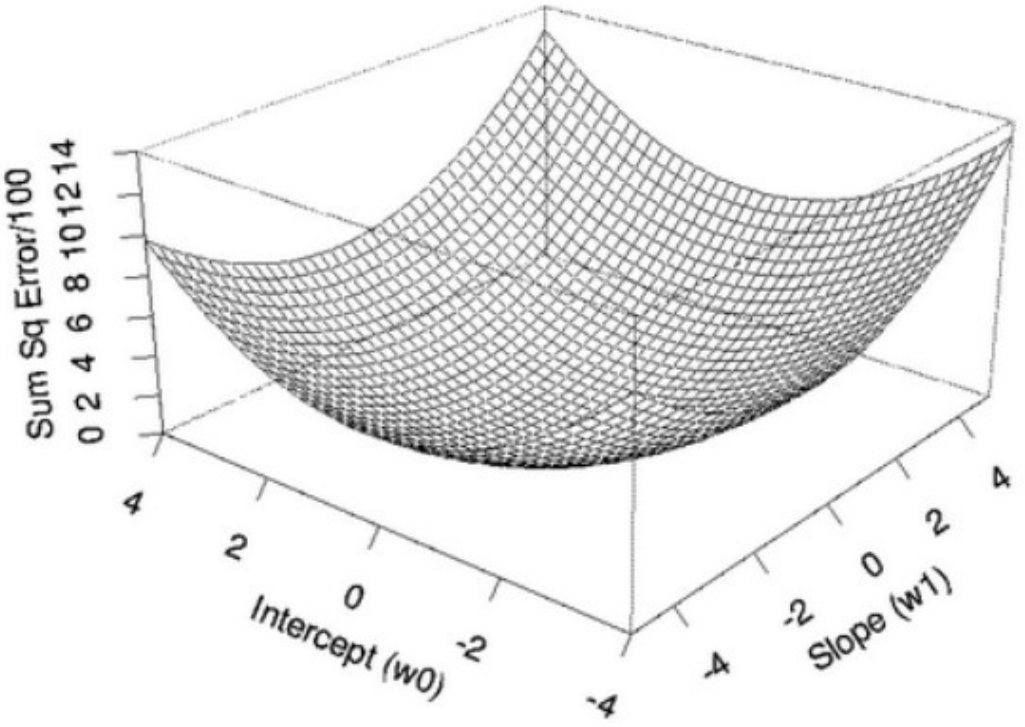
Different from a placeholder, **W** is defined as a `tf.Variable` where the value changes as we train the model, each time ending with lower loss. In line 10, we will explain what "loss" means. For now, we set the variable using `np.random.normal()` so that it draw a sample from the normal (Gaussian) distribution.

> *tf.Variable — A variable maintains state in the graph across calls to* `run()`*. You add a variable to the graph by constructing an instance of the class* `Variable`*.*
>
> *The* `Variable()` *constructor requires an initial value for the variable, which can be a* `Tensor` *of any type and shape. The initial value defines the type and shape of the variable. After construction, the type and shape of the variable are fixed. The value can be changed using one of the assign methods.*

Note that even though the variable is now defined, it has to be explicitly initialised before you can run operation using that value. This a feature of lazy evaluation and we will do the actual initialisation later.

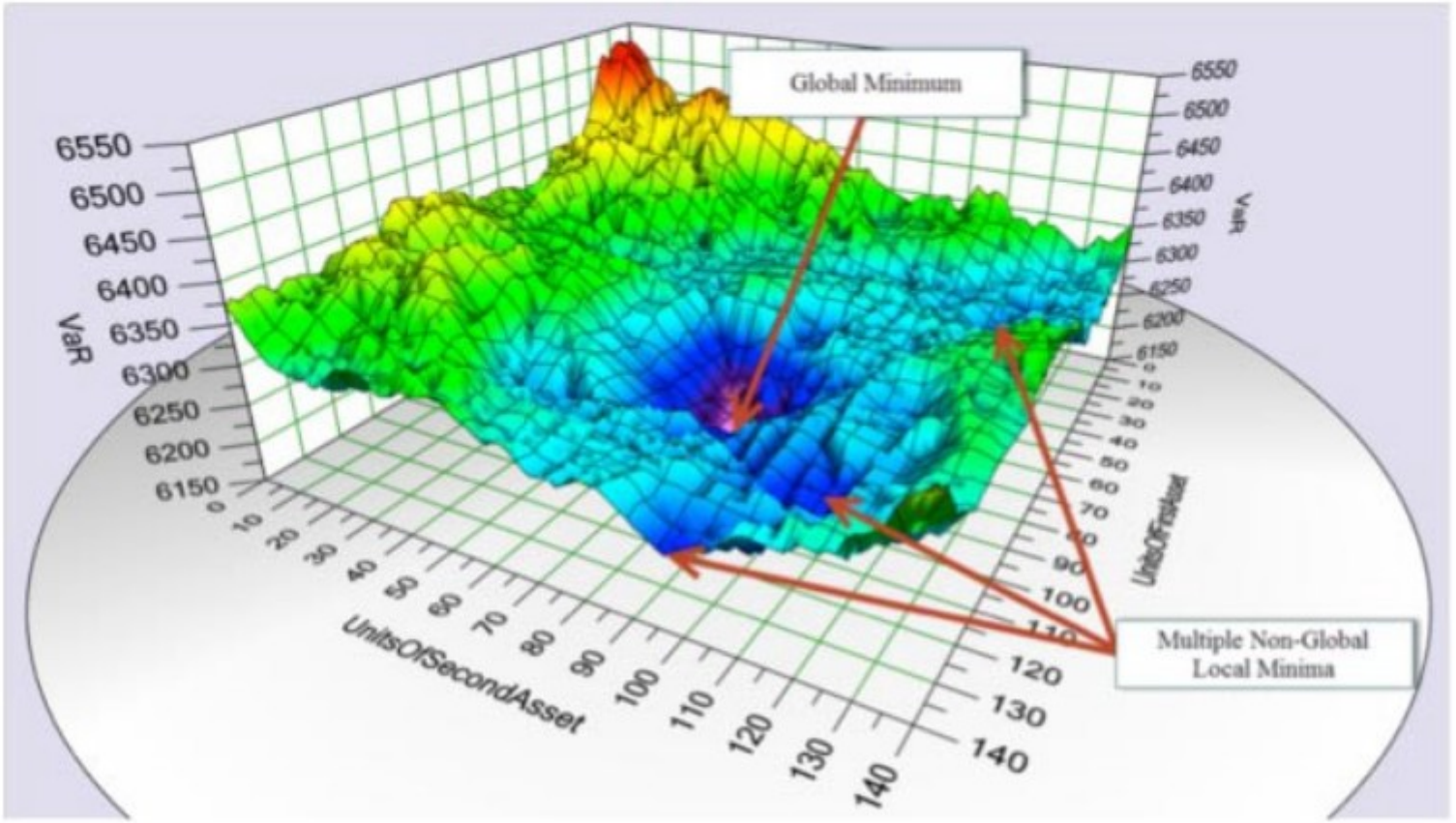
What **W** is really doing here is to find the gradient of our line of best fit. Previously, we generated the dataset using a gradient of 1.5 so we should expect the trained **W** to be close to this number. Selecting the starting number for **W** is somewhat important — imagine the work we save if we could "randomly" select 1.5, job's done isn't it? About right…

Since we are on this topic of searching for the optimal gradient in linear regression, I need to point out that our loss function will always result in one minimum loss value regardless of where we initialise **W**. This is due to the convexity of our loss function, **W** and **b** when we plot them in a chart like this. In other words, this bowl shape figure allows us to identify the lowest point, regardless of where we start.



One global minimum

However, this is not the case for more complex problems where there are multiple local minima like the one shown below. Choosing a bad number to initialise your variables could result in your gradient search being stuck at one of the local minima. This prevents you from reaching the global minimum which has a lower loss.



Multiple local minima with one global minimum

Researchers have come up with alternate methods of initialisation such as Xavier initialisation in attempt to avoid this problem. If you feel like using it, feel free to do so with:

`tf.get_variable(…, initializer=tf.contrib.layers.xavier_initializer())`.

[line 7] `b = tf.Variable(np.random.normal(), name='b')`

Other than **W**, we also want to train our bias **b**. Without **b**, our line of best fit will always cut through the origin and not learn the y-intercept. Remember the 0.5? We need to learn that as well.

[line 9] `y_pred = tf.add(tf.multiply(w, x), b)`

After defining **x**, **y** and **W** individually, we are now ready to put them together. To implement the formula `y = Wx + b`, we start off by multiplying `w` and `x` using `tf.multiply` before adding the variable `b` using `tf.add`. This will perform an element-wise multiplication and then addition which results in a tensor `y_pred`. `y_pred` represents the predicted **y** value and as you might be suspecting, the **predicted y** will be terrible at first and is far off from the **generated y.** Similar to a placeholder or variable, you are free to put a name to it.

[line 11] `loss = tf.reduce_mean(tf.square(y_pred - y))`

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y\_\text{pred} - y)^2$$

Mean Squared Error (MSE)

After calculating `y_pred`, we want to know how far the **predicted y** is away from our **generated y**. To do this, we need to design a method to calculate the "gap". This design is known as the loss function. Here, we selected the Mean Squared Error (MSE) a.k.a. L2 loss function as our "scoring mechanism". There are other popular loss functions but we are not covering them.

To understand our implementation of MSE, we first find the difference between each of the 100 points for `y_pred` and `y` using `y_pred - y`. Next, we amplify their difference by squaring them (`tf.square`), thereby making the difference (a lot) larger.

With a vector size of 100, we now have a problem — how can we know if these 100 values represent a good score or not? Usually a score is a single number that determines how well you perform (just like your exams). So to get to a single value, we make use of `tf.reduce_mean` to find the mean of all the 100 values and set it as our `loss`.

[line 13] `return x, y, y_pred, loss`

Last but not least, we return all the 4 values after constructing them.

Computing the Graph

```
def run():
  x_batch, y_batch = generate_dataset()
  x, y, y_pred, loss = linear_regression()

  optimizer = tf.train.GradientDescentOptimizer(0.1)
  train_op = optimizer.minimize(loss)

  with tf.Session() as session:
    session.run(tf.global_variables_initializer())
    feed_dict = {x: x_batch, y: y_batch}

    for i in range(30):
      session.run(train_op, feed_dict)
      print(i, "loss:", loss.eval(feed_dict))

    print('Predicting')
    y_pred_batch = session.run(y_pred, {x : x_batch})

  plt.scatter(x_batch, y_batch)
  plt.plot(x_batch, y_pred_batch, color='red')
  plt.xlim(0, 2)
  plt.ylim(0, 2)
  plt.savefig('plot.png')

if __name__ == "__main__":
  run()
```

With `generate_dataset()` and `linear_regression()`, we are now ready to run the program and begin finding our optimal gradient **W** and bias **b**!

[line 2, 3]

```
x_batch, y_batch = generate_dataset()
x, y, y_pred, loss = linear_regression()
```

In this `run()` function, we start off by calling `generate_dataset()` and `linear_regression()` to get `x_batch`, `y_batch`, `x`, `y`, `y_pred` and `loss`.

[line 5, 6]

```
optimizer = tf.train.GradientDescentOptimizer(0.1)
train_op = optimizer.minimize(loss)
```

Then, we define the optimiser and ask it to minimise the loss in the graph. There are several optimisers to choose from and we conveniently selected the Gradient Descent algorithm and set the learning rate to 0.1.

We will not dive into the world of optimisation algorithms but in short, the job of an optimiser is to minimise (or maximise) your loss (objective) function. It does so by updating the trainable variables (**W** and **b**) in the direction of the optimal solution everytime it runs.

Calling the minimize function computes the gradients and applying them to the variables — this is the behaviour by default and you are free to change it using the argument `var_list`.

[line 8] `with tf.Session() as session:`

In the earlier part where we construct the graph, we said that TensorFlow uses lazy evaluation. This really means that the graph is only computed when a session starts. Here, we name the session object as `session`.

[line 9] `session.run(tf.global_variables_initializer())`

Then we kickstart our first session by initialising all the values we ask the variables to hold. Due to lazy evaluation, variables e.g. **W** (`w = tf.Variable(np.random.normal(), name='W')`) are not initialised when the graph is first constructed, until we run this line. See this for further explanation.

[line 10] `feed_dict = {x: x_batch, y: y_batch}`

Next, we need to come up with feed_dict which is essentially an argument for `session.run()`. `feed_dict` is a dictionary with its key being a `tf.Tensor`, `tf.placeholder` or `tf.SparseTensor`. The `feed_dict` argument allows the caller to override the value of the tensors (scalar, string, list, numpy array o tf.placeholder e.g. **x** and **y**) in the graph.

In this line, the **x** and **y** are the placeholders and **x_batch** and **y_batch** are the values generated, ready to fill up the placeholders during `session.run()`.

[line 12] `for i in range(30):`

After initialising the variables and preparing values for placeholders using `feed_dict`, we now come to the core of the script which is to define how many times we want to "adjust" / "train" the weight (**W**) and bias (**b**). The number of times we go through the training data (**x** and **y**) in one full cycle is also known as **epoch / training step**. One full cycle is also defined as a one feedforward and one backpropagation.

During feedforward, we pass in the value of **x**, **w** and **b** to get the **predicted y**. This computes the loss which is represented by a number. As the objective of this graph is to minimise the loss, the optimiser will then perform a backpropagation to "adjust" the trainable variables (**W** and **b**) so that the next time we perform the feedforward (in another epoch), the loss will be lowered.

We do this forward and backward cycle for 30 times. Note that 30 is a hyperparameter and you are free to change it. Also note that more epochs = longer training time.

[line 13]

```
session.run(train_op, feed_dict)
```

Now we are ready to run our first epoch by calling `session.run()` with fetches and `feed_dict`. Over here, `session.run()` evaluates every tensor in fetches (`train_op`) and substitutes the values in `feed_dict` for the corresponding input values.

> ***`fetches`***: *A single graph element, a list of graph elements, or a dictionary whose values are graph elements or lists of graph elements (see documentation for `run`).*

What happens behind the scene when the `run()` method is called by `session` object is that your code will run through the necessary part (nodes) of the graph to calculate every tensor in the fetches. Since `train_op` refers to the `optimizer` calling the method `minimize(loss)`, it will being to evaluate `loss` by calling the loss function which in turn trigger `y_pred`, `y`, `W`, `x` and `b` to be computed.

```
0 loss: 2.5949948
1 loss: 0.9005146
2 loss: 0.35990283
3 loss: 0.18539226
4 loss: 0.12716371
5 loss: 0.10598618
6 loss: 0.09672409
7 loss: 0.09139658
8 loss: 0.08746332
9 loss: 0.08410974
10 loss: 0.0810691
11 loss: 0.07824889
12 loss: 0.07561237
13 loss: 0.07314089
14 loss: 0.07082201
15 loss: 0.06864563
16 loss: 0.06660278
17 loss: 0.06468518
18 loss: 0.062885165
19 loss: 0.061195504
20 loss: 0.059609417
21 loss: 0.058120556
22 loss: 0.056722976
23 loss: 0.055411052
24 loss: 0.05417957
25 loss: 0.053023595
26 loss: 0.05193847
27 loss: 0.050919887
28 loss: 0.04996371
29 loss: 0.049066164
```

print(i, "loss:", loss.eval(feed_dict))

[Line 14] `print(i, "loss:", loss.eval(feed_dict))`

This line prints out the loss at each epoch. On the left, you can see the value for loss is decreasing for every epoch.

The loss value is calculated using `loss.eval()` and `feed_dict` as argument.

[line 16, 17]

```
print('Predicting')
y_pred_batch =
session.run(y_pred, {x :
x_batch})
```

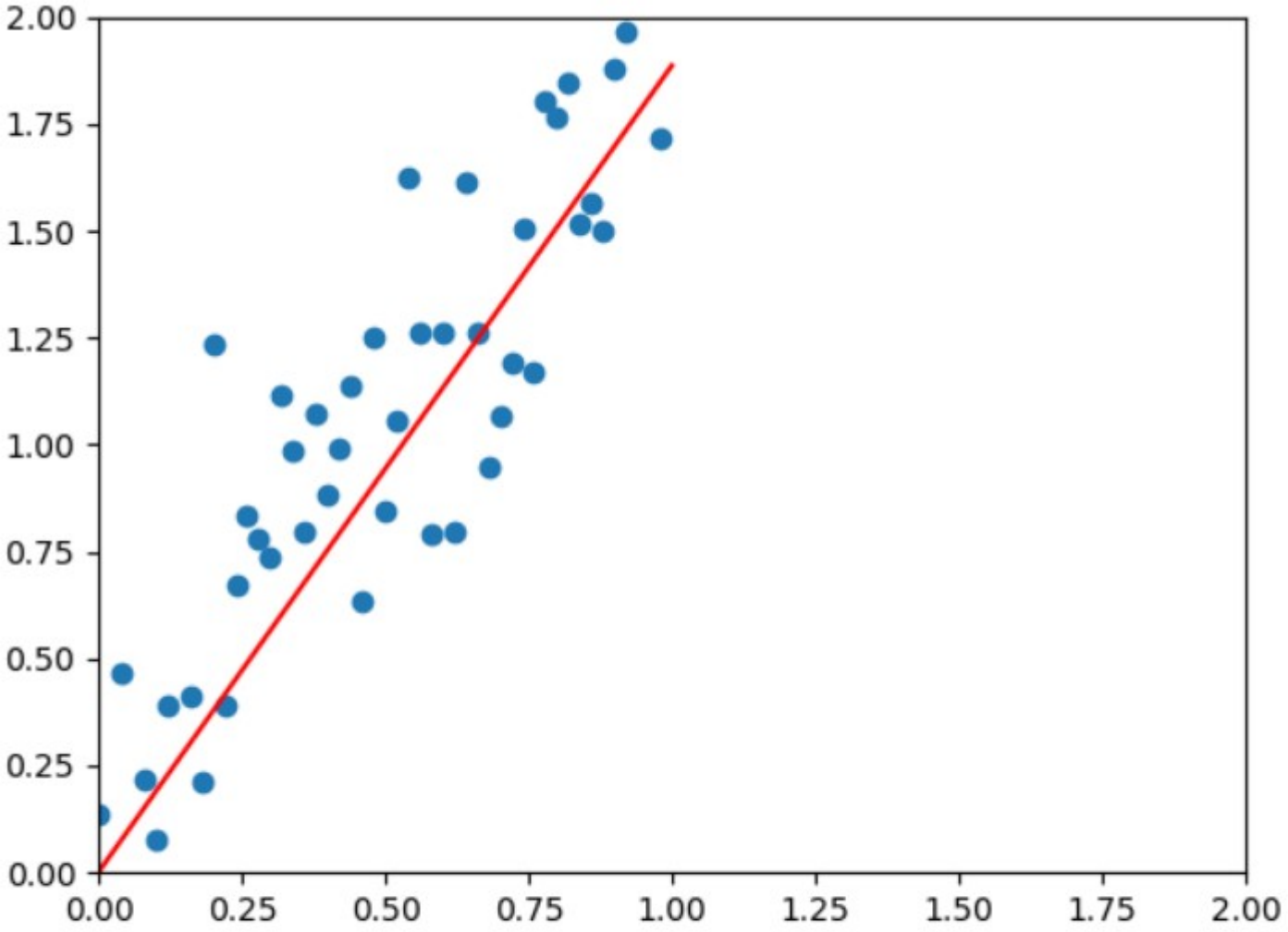
After 30 epochs, we now have a trained **W** and **b** for us to perform inference. Similar to training, inference can be done with the same graph using `session.run()` but this time, the fetches will be **y_pred** instead of **train_op** and we only need to feed in **x**. We do this because **W** and **b** are already trained and the **predicted y** can be computed with just **x**. Notice that in `tf.add(tf.multiply(w, x), b)`, there isn't **y**.

By now we have already declared 3 `session.run()`, so let's recap their usage since `session.run()` is our command to run operations and evaluate tenors in our graph. The first time we did was to initialise our variables, second time during training to pass in our feed_dict and third time to run prediction.

[line 19–23]

```
plt.scatter(x_batch, y_batch)
plt.plot(x_batch, y_pred_batch, color='red')
plt.xlim(0, 2)
plt.ylim(0, 2)
plt.savefig('plot.png')
```

We plot the chart with both the generated `x_batch` and `y_batch`, together with our predicted line (with `x_batch` and `y_pred_batch`).



plt.plot(x_batch, y_pred_batch) — we drew the line of best fit

Deep Learning with TensorFlow and Keras

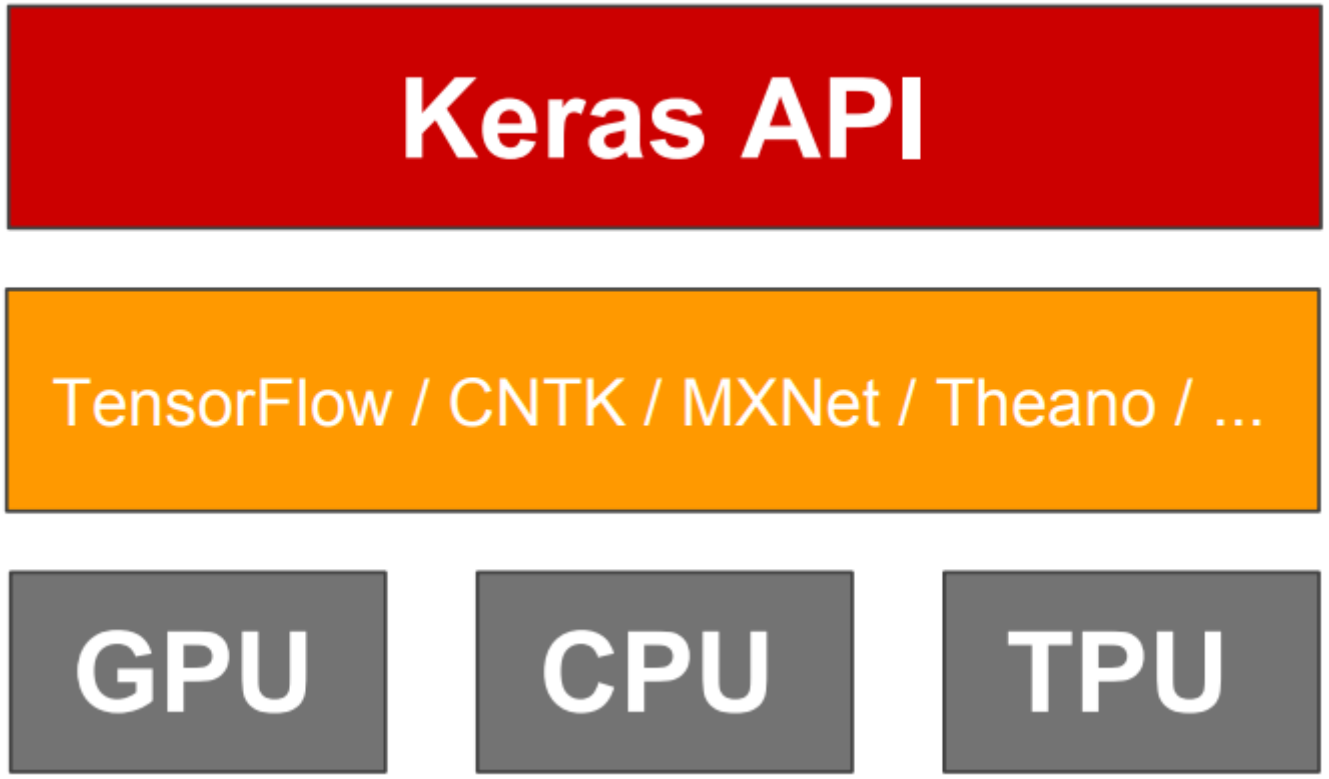
Credit: Jordi Torres

1- Load and Preprocess the Data

2- Define the Model

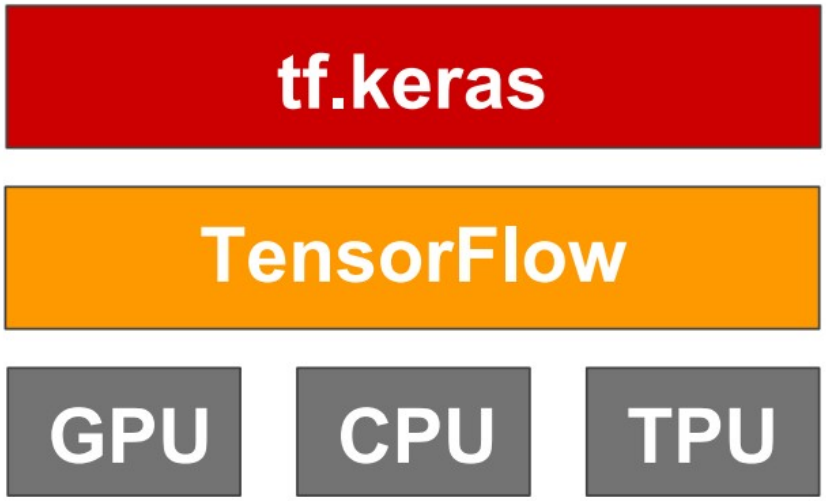
3- Train the Model

To do this, we will use the TensorFlow Keras API, the most popular library currently in the Deep Learning community.


Keras API
TensorFlow / CNTK / MXNet / Theano / ...
GPU
CPU
TPU

Keras is the official high-level API of TensorFlow

- tensorflow.keras (tf.keras) module
- Part of core TensorFlow since v1.4
- Full Keras API
- Better optimized for TF
- Better integration with TF-specific features
  - Estimator API
  - Eager execution
  - etc.



- A focus on user experience.
- Large adoption in the industry and research community.
- Multi-backend, multi-platform.
- Easy productization of models.


NETFLIX


UBER


Google


instacart


HUAWEI


NVIDIA


Square


Expedia


Zocdoc


yelp

etc...

Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

This makes Keras easy to learn and easy to use. As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn helps you win machine learning competitions.

This ease of use does not come at the cost of reduced flexibility: because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as tf.keras, the Keras API integrates seamlessly with your TensorFlow workflows.

Keras is multi-backend, multi-platform

- Develop in Python, R
  - On Unix, Windows, OSX
- Run the same code with...
  - TensorFlow
  - CNTK
  - Theano
  - MXNet
  - PlaidML
  - ??
- CPU, NVIDIA GPU, AMD GPU, TPU...

Three API styles

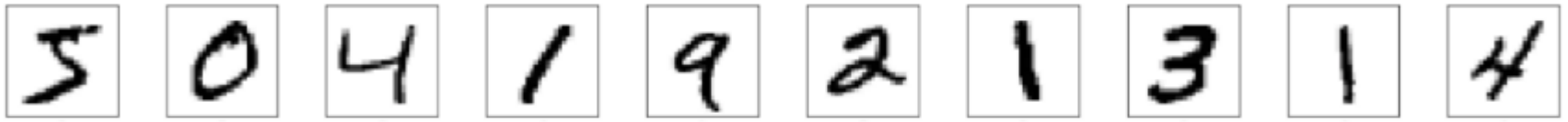
- The Sequential Model
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks
  - Good for 70+% of use cases
- The functional API
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies
  - Good for 95% of use cases
- Model subclassing
  - Maximum flexibility
  - Larger potential error surface

Recognizing Handwritten Digits Using Tensorflow/Keras

Credit: Jordi Torres

Handwritten digits

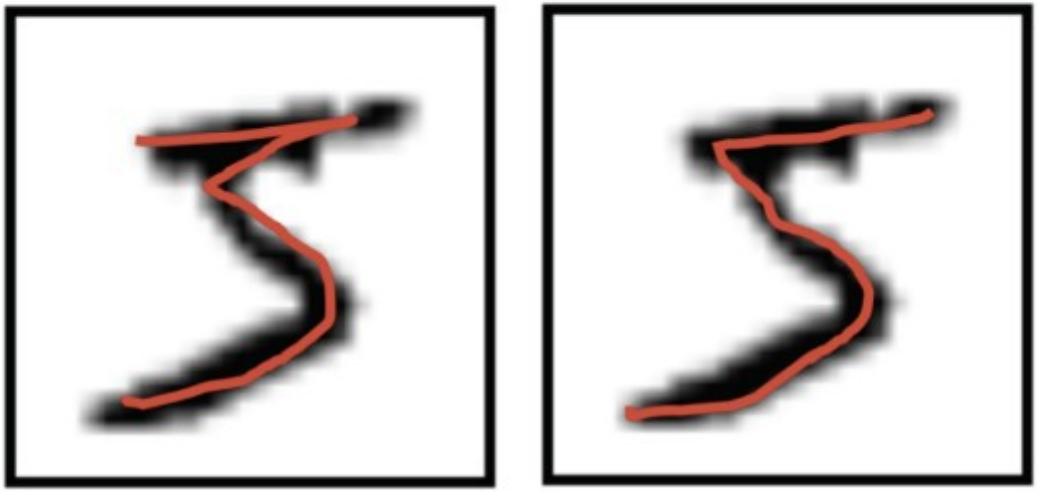
As a case study, we will create a model that allows us to identify handwritten digits such as te following ones:



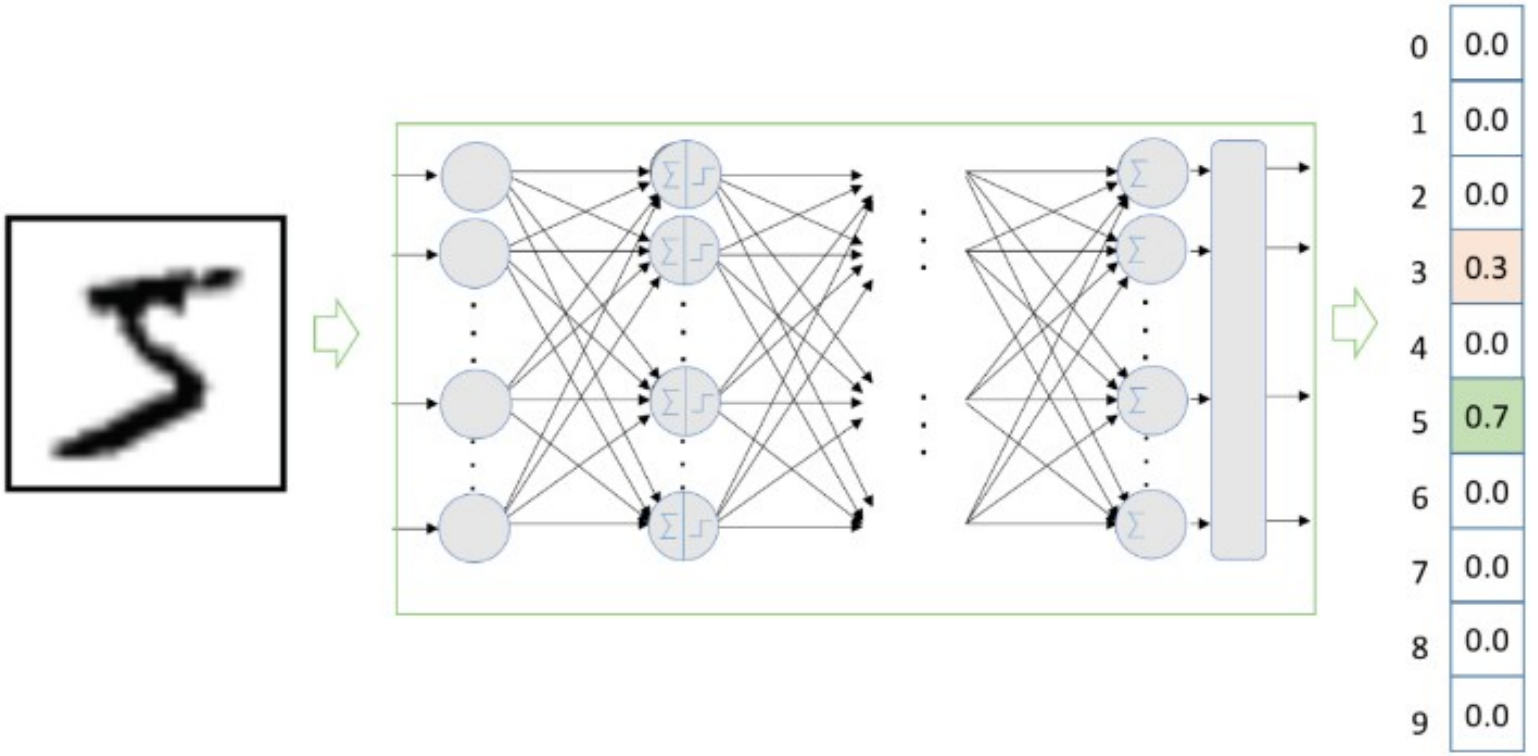
The goal is to create a mathematical model that, given an image, the model identify the number it represents. For example, if we feed to the model the first image, we would expect it to answer that it is a 5. The next one a 0, next one a 4, an so on.

Classification problem

Actually, we are dealing with a classification problem, which given an image, the model classifies it between 0 and 9. But sometimes, even we can find ourselves with certain doubts, for example, the first image represents a 5 or a 3?



For this purpose, the neural network that we will create returns a vector with 10 positions indicating the likelihood of each of the ten possible digits:



```
1: import tensorflow as tf
2: from tensorflow.keras.utils import to_categorical


3:(x_train, y_train), _ = tf.keras.datasets.mnist.load_data()


4: x_train = x_train.reshape(60000, 784).astype('float32')/255
5: y_train = to_categorical(y_train, num_classes=10)


6: model = tf.keras.Sequential()
7: model.add(tf.keras.layers.Dense(10, activation='sigmoid',
           input_shape=(784,)))
8: model.add(tf.keras.layers.Dense(10, activation='softmax'))


9: model.compile(loss="categorical_crossentropy", optimizer="sgd",
                metrics = ['accuracy'])


10: model.fit(x_train, y_train, epochs=10, verbose=0)
```

1. Load and Preprocessing Data

First of all we need to import some Python libraries that we need in order to program our neural network in TensorFlow:

```
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
```

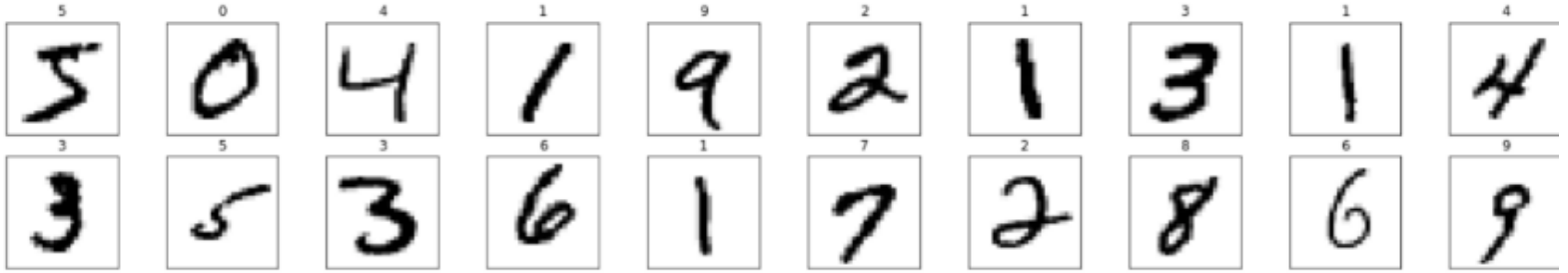
Next step is to loading data that will be used to train our neural network. We will use the MNIST dataset, which can be downloaded from *The MNIST database* page. This dataset contains 60,000 images of hand-made digits to train the model and it is ideal for entering pattern recognition techniques for the first time without having to spend much time preprocessing and formatting data, both very important and expensive steps in the analysis of data and of special complexity when working with images.

With TensorFlow this can be done using this line of code (line 3):

```
(x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
```

Optional step: If you want, you can verify the data loaded using this code:

```
import numpy as np
import matplotlib.pyplot as plt


fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
   ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
   ax.imshow(x_train[idx], cmap=plt.cm.binary)
   ax.set_title(str(y_train[idx]))
```



This dataset of black and white images (images contain gray levels) has been normalized to 20×20 pixels while retaining their aspect ratio. Subsequently, the images were centered, calculating the center of mass of these and moving the image in order to position this point in the center of the 28×28 field.

These MNIST images of 28×28 pixels are represented as an array of numbers whose values range from [0, 255] of type `uint8` . But it is usual to scale the input values of neural networks to certain ranges. In the example of this post the input values should be scaled to values of type `float32` within the interval [0, 1].

On the other hand, to facilitate the entry of data into our neural network we must make a transformation of the input (image) from 2 dimensions (2D) to a vector of 1 dimension (1D). That is, the matrix of 28×28 numbers can be represented by a vector (array) of 784 numbers (concatenating row by row), which is the format that accepts as input a densely connected neural network like the one we will see in this post.

We can achieve these transformations with the following line of code (line 4):

```
x_train = x_train.reshape(60000, 784).astype('float32')/255
```

Furthermore, the dataset has a label for each of the images that indicates what digit it represents (downloaded in `y_train`). In our case are numbers between 0 and 9 that indicate which digit the image represents,that is, to which class it is associated.

We need to represent each label with a vector of 10 positions as we presented before, where the position corresponding to the digit that represents the image contains a 1 and the rest contains 0s. This process of transforming the labels into a vector of as many zeros as the number of different labels, and putting a 1 in the index corresponding to the label, is known as *one-hot encoding*. For example, the number 7 will be encoded as:

We can achieve this transformations with the following line of code (line 5):

```
y_train = to_categorical(y_train, num_classes=10)
```

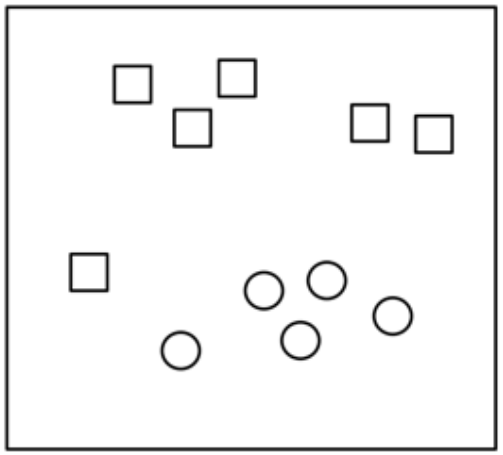
2. Define the Model

In order to define the model with the Keras's API we only need these code lines (lines 6–8):

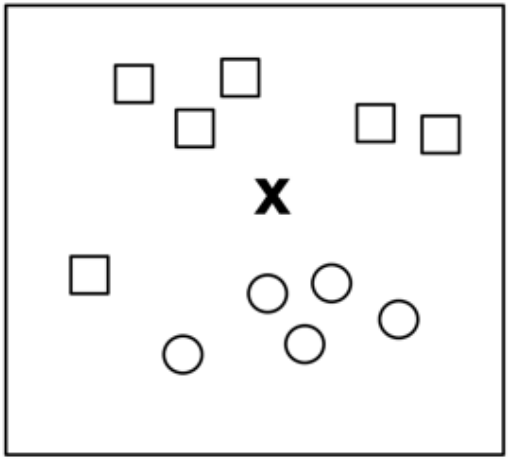
```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(10,activation='sigmoid',
          input_shape=(784,)))
model.add(tf.keras.layers.Dense(10,activation='softmax'))
```

A plain artificial neuron

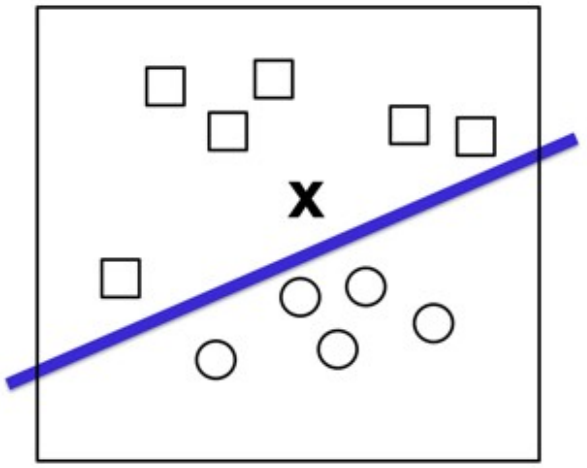
In order to show how a basic neuronal is, let's suppose a simple example where we have a set of points in a two-dimensional plane and each point is already labeled "square" or "circle":



Given a new point "*X*", we want to know what label corresponds to it:



A common approach is to draw a line that separates the two groups and use this line as a classifier:



In this case, the input data will be represented by vectors of the form (*x1*, *x2*) that indicate their coordinates in this two-dimensional space, and our function will return '0' or '1' (above or below the line) to know if it should be classified as "square" or "circle". It can be defined by:

$$y = w_1x_1 + w_2x_2 + b$$

More generally, we can express the line as:

$$y = W * X + b$$

To classify input elements X, which in our case are two-dimensional, we must learn a vector of weight W of the same dimension as the input vectors,that is, the vector (*w1, w2*) and a *b* bias.

With these calculated values, we can now construct an artificial neuron to classify a new element *X*. Basically, the neuron applies this vector *W* of calculated weights on the values in each dimension of the input element *X*, and at the end adds the bias *b*. The result of this will be passed through a non-linear "activation" function to produce a result of '0' or '1'. The function of this artificial neuron that we have just defined can be expressed in a more formal way such as:

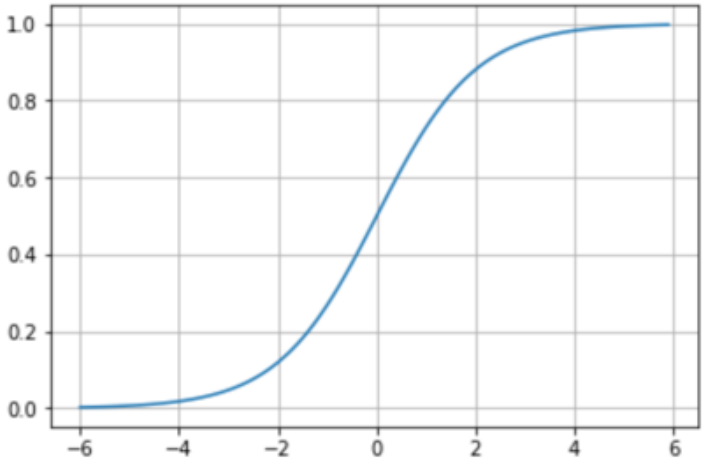
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 \text{ si } z \geq 0 \\ 0 \text{ si } z < 0 \end{cases}$$

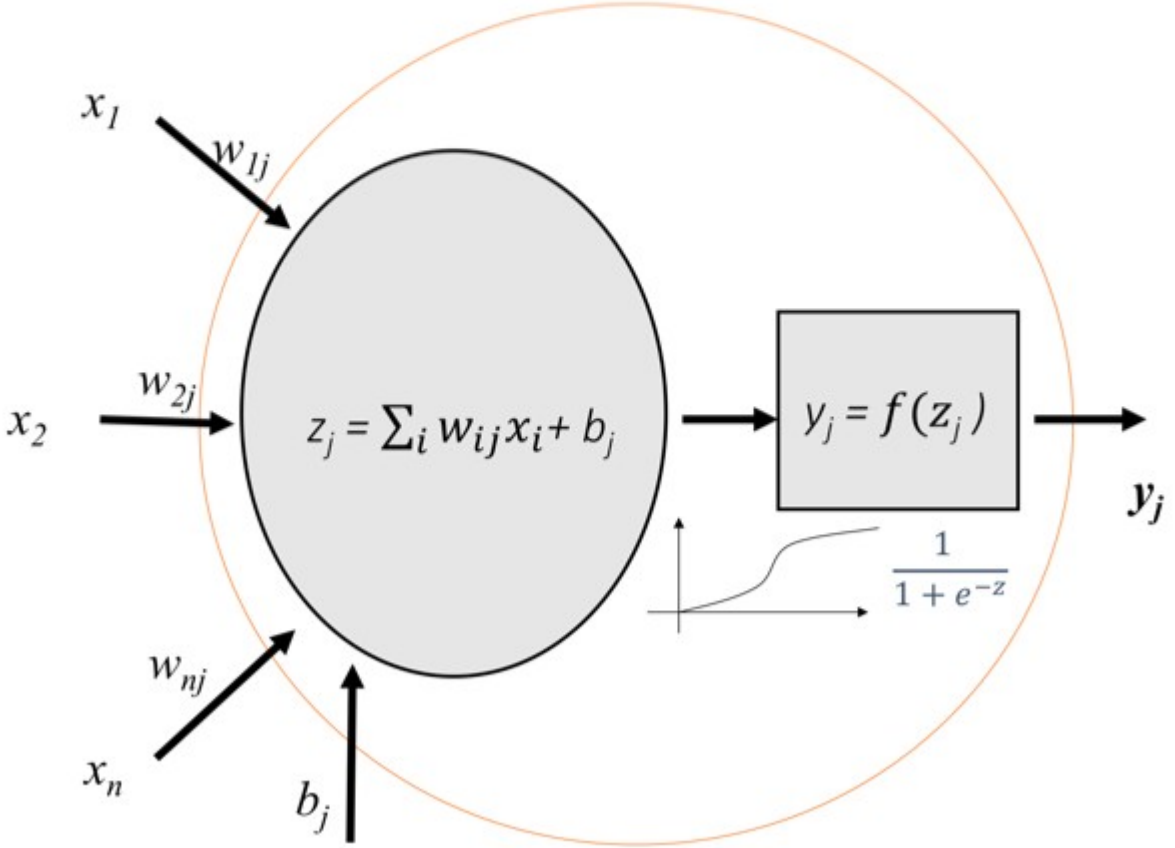
Now, we will need a function that applies a transformation to variable z so that it becomes '0' or '1'. Although there are several functions ("activation functions"), for this example we will use one known as a *sigmoid* function that returns an actual output value between 0 and 1 for any input value:

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

If we analyze the previous formula, we can see that it always tends to give values close to 0 or 1. If the input z is reasonably large and positive, "e" at minus z is zero and, therefore, the y takes the value of 1. If z has a large and negative value, it turns out that for "e" raised to a large positive number, the denominator of the formula will turn out to be a large number and therefore the value of y will be close to 0. Graphically, the sigmoid function presents this form:

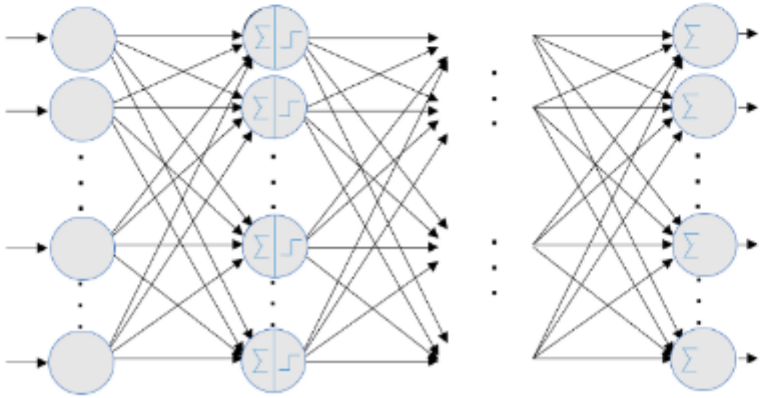


So far we have presented how to define an artificial neuron, the simplest architecture that a neural network can have. In particular this architecture is named in the literature of the subject as Perceptron (also called *linear threshold unit* (LTU)), invented in 1957 by Frank Rosenblatt, and visually summarized in a general way with the following scheme:

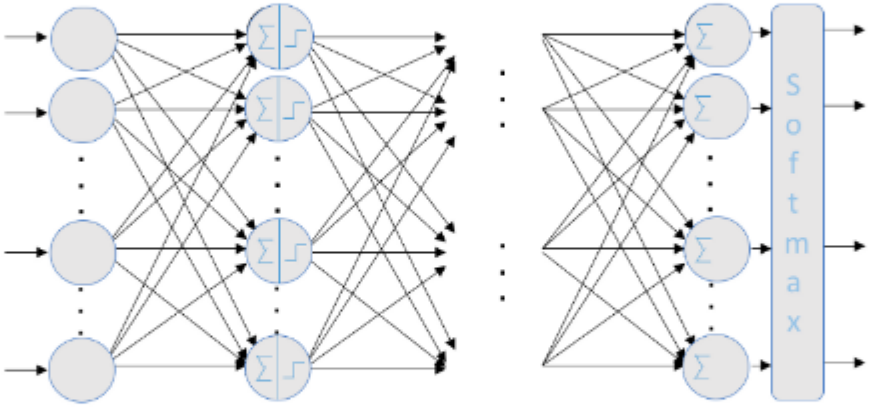


Multi-Layer Perceptron

In the literature of the area we refer to a Multi-Layer Perceptron (MLP) when we find neural networks that have an *input layer*, one or more layers composed of perceptrons, called *hidden layers* and a final layer with several perceptrons called the *output layer*. In general we refer to *Deep Learning* when the model based on neural networks is composed of multiple hidden layers. Visually it can be presented with the following scheme:



MLPs are often used for classification, and specifically when classes are exclusive, as in the case of the classification of digit images (in classes from 0 to 9). In this case, the output layer returns the probability of belonging to each one of the classes, thanks to a function called softmax. Visually we could represent it in the following way:



As we mentioned, there are several activation functions in addition to the *sigmoid*, each with different properties. One of them is the one we just mentioned, the *softmax* activation function, which will be useful to present an example of simple neural network to classify in more than two classes. For the moment we can consider the *softmax* function as a generalization of the *sigmoid* function that allows us to classify more than two classes.

Softmax activation function

We will solve the problem in a way that, given an input image, we will obtain the probabilities that it is each of the 10 possible digits. In this way, we will have a model that, for example, could predict a five in an image, but only being sure in 70% that it is a five. Due to the stroke of the upper part of the number in this image, it seems that it could become an three in a 20% chance and it could even give a certain probability to any other number. Although in this particular case we will consider that the prediction of our model is a five since it is the one with the highest probability, this approach of using a probability distribution can give us a better idea of how confident we are of our prediction. This is good in this case, where the numbers are made by hand, and surely in many of them, we cannot recognize the digits with 100% certainty.

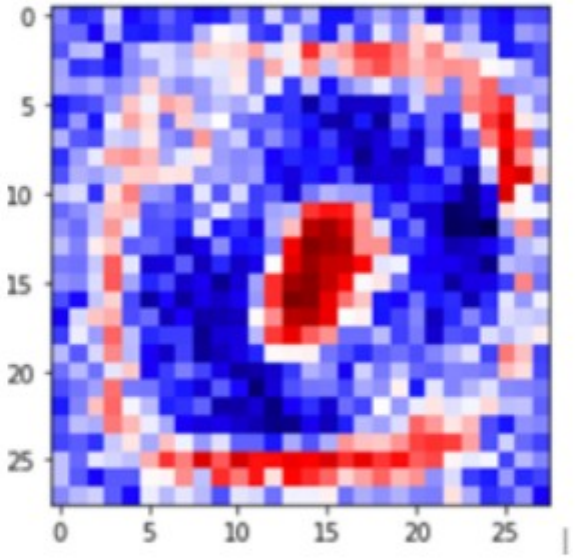
Therefore, for this example of classification we will obtain, for each input example, an output vector with the probability distribution over a set of mutually exclusive labels. That is, a vector of 10 probabilities each corresponding to a digit and also the sum of all these 10 probabilities results in the value of 1 (the probabilities will be expressed between 0 and 1).

As we have already advanced, this is achieved through the use of an output layer in our neural network with the *softmax* activation function, in which each neuron in this *softmax* layer depends on the outputs of all the other neurons in the layer, since that the sum of the output of all of them must be 1.

But how does the *softmax* activation function work? The *softmax* function is based on calculating "the evidence" that a certain image belongs to a particular class and then these evidences are converted into probabilities that it belongs to each of the possible classes.

An approach to measure the evidence that a certain image belongs to a particular class is to make a weighted sum of the evidence of belonging to each of its pixels to that class. To explain the idea I will use a visual example.

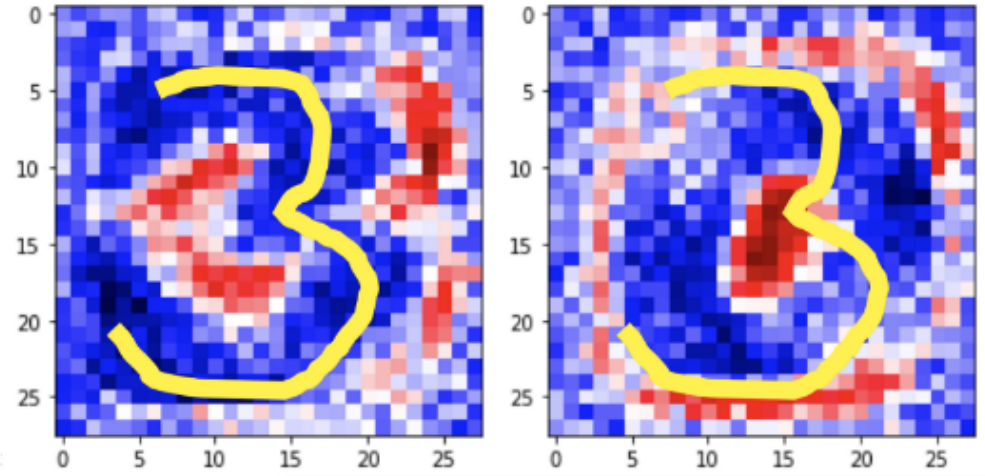
Let's suppose that we already have the model learned for the number zero. For the moment, we can consider a model as "something" that contains information to know if a number is of a certain class. In this case, for the number zero, suppose we have a model like the one presented below:



In this case, with a matrix of 28×28 pixels, where the pixels in red represent negative weights (i.e., reduce the evidence that it belongs), while that the pixels in blue represent positive weights (the evidence of which is greater increases). The white color represents the neutral value.

Let's imagine that we trace a zero over it. In general, the trace of our zero would fall on the blue zone (remember that we are talking about images that have been normalized to 20×20 pixels and later centered on a 28×28 image). It is quite evident that if our stroke goes over the red zone, it is most likely that we are not writing a zero; therefore, using a metric based on adding if we pass through the blue zone and subtracting if we pass through the red zone seems reasonable.

To confirm that it is a good metric, let's imagine now that we draw a three; it is clear that the red zone of the center of the previous model that we used for the zero will penalize the aforementioned metric since, as we can see in the left part of the following figure, when writing a three we pass over:



But on the other hand, if the reference model is the one corresponding to number 3 as shown in the right part of the previous figure, we can see that, in general, the different possible traces that represent a three are mostly maintained in the blue zone.

Once the evidence of belonging to each of the 10 classes has been calculated, these must be converted into probabilities whose sum of all their components add 1. For this, softmax uses the exponential value of the calculated evidence and then normalizes them so that the sum equates to one, forming a probability distribution. The probability of belonging to class i is:

$$Softmax_i = \frac{e^{evidencia_i}}{\sum_j e^{evidencia_j}}$$

Intuitively, the effect obtained with the use of exponentials is that one more unit of evidence has a multiplier effect and one unit less has the inverse effect. The interesting thing about this function is that a good prediction will have a single entry in the vector with a value close to 1, while the remaining entries will be close to 0. In a weak prediction, there will be several possible labels, which will have more or less the same probability.

Sequential class in Keras

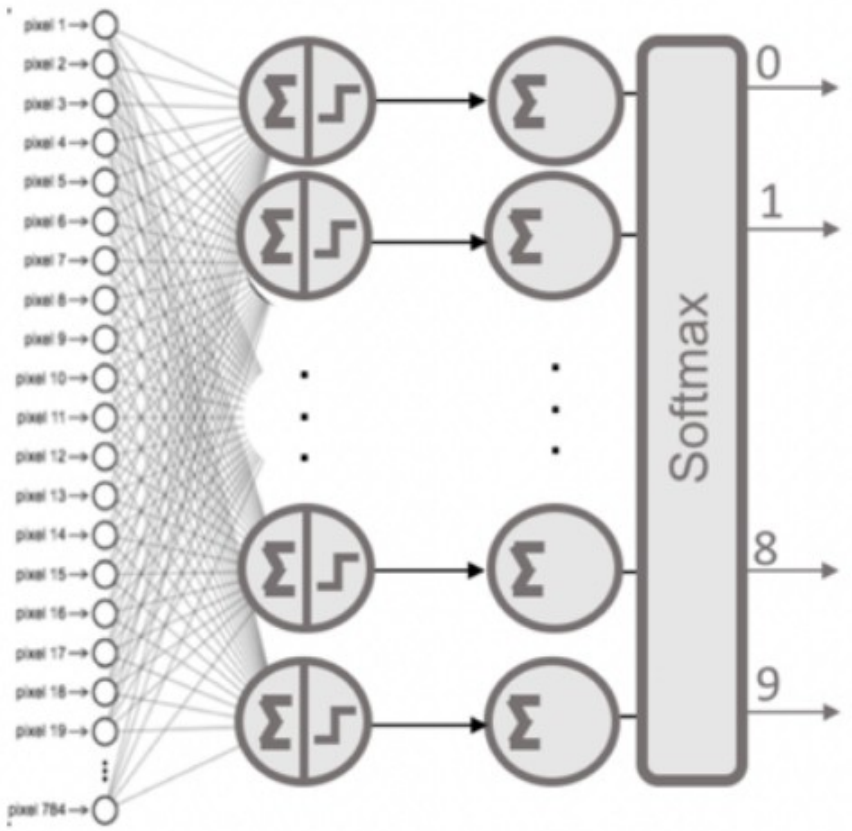
The main data structure in Keras is the *Sequential* class, which allows the creation of a basic neural network. Keras also offers an API that allows implementing more complex models in the form of a graph that can have multiple inputs, multiple outputs, with arbitrary connections in between, but it is beyond the scope of this post.

The *Sequential* class of the Keras library is a wrapper for the sequential neural network model that Keras offers and can be created in the following way:

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(10,activation='sigmoid',
          input_shape=(784,)))
model.add(tf.keras.layers.Dense(10,activation='softmax'))
```

In this case, the model in Keras is considered as a sequence of layers and each of them gradually "distills" the input data to obtain the desired output.In Keras we can find all the required types of layers that can be easily added to the model through the `add()` method.

Here, the neural network has been defined as a sequence of two layers that are densely connected (or fully connected), meaning that all the neurons in each layer are connected to all the neurons in the next layer. Visually we could represent it in the following way:



In the previous code we explicitly express in the *input_shape* argument of the first layer what the input data is like: a tensor that indicates that we have 784 features of the model.

A very interesting characteristic of the Keras library is that it will automatically deduce the shape of the tensors between layers after the first one. This means that the programmer only has to establish this information for the first of them. Also, for each layer we indicate the number of nodes that it has and the activation function that we will apply in it (in this example, *sigmoid*).

The second layer in this example is a *softmax* layer of 10 neurons, which means that it will return a matrix of 10 probability values representing the 10 possible digits (in general, the output layer of a classification network will have as many neurons as classes, except in a binary classification, where only one neuron is needed). Each value will be the probability that the image of the current digit belongs to each one of them.

Optional step: A very useful method that Keras provides to check the architecture of our model is `summary()`:

```
model.summary()


_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
dense_1 (Dense) (None, 10) 7850
_________________________________________________________________
dense_2 (Dense) (None, 10) 110
=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
```

For our simple example, we see that it indicates that 7,960 parameters are required (column *Param #*), which correspond to 7,850 parameters to the first layer and 110 to the second.

In the first layer, for each neuron i (between 0 and 9) we require 784 parameters for the weights wij and therefore 10×784 parameters to store the weights of the 10 neurons. In addition to the 10 additional parameters for the 10 bj biases corresponding to each one of them. In the second layer, being a *softmax* function, it is required to connect all 10 neurons with the 10 neurons of the previous layer. Therefore 10×10 wi parameters are required and in addition 10 bj biases corresponding to each node.

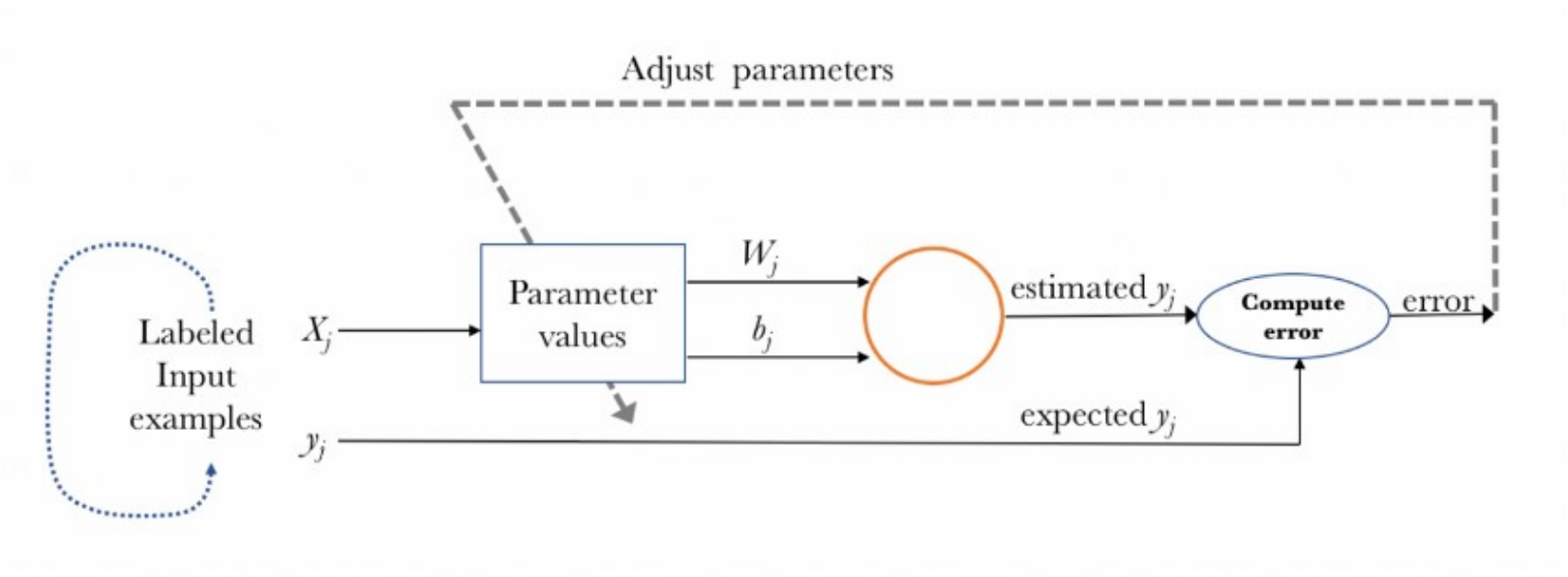
The details of the arguments that we can indicate for the `dense` layer can be found in the Keras manual. In our example, the most relevant ones appear in the example. The first argument indicates the number of neurons in the layer; the following is the activation function that we will use in it.

3. Train the Model

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

model.fit(x_train, y_train, epochs=10, verbose=0)
```

Learning process

The way how the neural network can learn the weights W and the biases b of the neurons is an iterative process for all the known labeled input examples, comparing the value of its label estimated through the model, with the expected value of the label of each element. After each iteration, the parameter values are adjusted in such a way that the discordance (error) between the estimated value for the image and the actual value, is becoming smaller. The following scheme wants to visually summarize the learning process of one perceptron in a general way:



Configuration of the learning process

We can configure how this learning process will be with the `compile()` method, with which we can specify some properties through method arguments.

The first of these arguments is the *loss function* that we will use to evaluate the degree of error between calculated outputs and the desired outputs of the training data. On the other hand, we specify an *optimizer* that is the way we have to specify the optimization algorithm that allows the neural network to calculate the weights of the parameters from the input data and the defined loss function.

And finally we must indicate the metric that we will use to monitor the learning process of our neural network. In this first example we will only consider the *accuracy* (fraction of images that are correctly classified). For example, in our case we can specify the following arguments in *compile()* method to test it:

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

In this example we specify that the loss function is *categorical_crossentropy*, the optimizer used is the *stocastic gradient descent (sgd)* and the metric is *accuracy*, with which we will evaluate the percentage of correct guesses.ç

Model training

Once our model has been defined and the learning method configured, it is ready to be trained. For this we can train or "adjust" the model to the training data available by invoking the *fit()* method of the model:

```
model.fit(x_train, y_train, epochs=10, verbose=0)
```

In the first two arguments we have indicated the data with which we will train the model in the form of Numpy arrays. The *batch_size* argument indicates the number of data that we will use for each update of the model parameters and with *epochs* we are indicating the number of times we will use all the data in the learning process.

This method finds the value of the parameters of the network through the iterative training algorithm that we mentioned. Roughly, in each iteration of this algorithm, this algorith takes training data from *x_train*, passes them through the neural network (with the values that their parameters have at that moment), compares the obtained result with the expected one(indicated in *y_train*) and calculates the *loss* to guide the adjustment process of the model parameters, which intuitively consists of applying the optimizer specified above in the *compile()* method to calculate a new value of each one of the model parameters (weights and biases)in each iteration in such a way that the loss is reduced.

This is the method that, as we will see, may take longer and Keras allows us to see its progress using the *verbose* argument (by default, equal to 1), in addition to indicating an estimate of how long each *epoch* takes:

```
Epoch 1/5
60000/60000 [========] – 1s 15us/step – loss: 2.1822 – acc: 0.2916
Epoch 2/5
60000/60000 [========] – 1s 12us/step – loss: 1.9180 – acc: 0.5283
Epoch 3/5
60000/60000 [========] – 1s 13us/step – loss: 1.6978 – acc: 0.5937
Epoch 4/5
60000/60000 [========] – 1s 14us/step – loss: 1.5102 – acc: 0.6537
Epoch 5/5
60000/60000 [========] – 1s 13us/step – loss: 1.3526 – acc: 0.7034
10000/10000 [========] – 0s 22us/step
```

Using the model

In order to use the model we can download another set of images (different o the training images) with the following code:

```
_, (x_test_, y_test_)= tf.keras.datasets.mnist.load_data()
x_test = x_test_.reshape(10000, 784).astype('float32')/255
y_test = to_categorical(y_test_, num_classes=10)
```

Optional step: Model evaluation

At this point, the neural network has been trained and its behavior with new test data can now be evaluated using the `evaluation()` method. This method returns two values:

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

These values indicate how well or badly our model behaves with new data that it has never seen. These data have been stored in *x_test* and *y_test* when we have performed the *mnist.load_data()* and we pass them to the method as arguments. In the scope of this post we will only look at one of them, the accuracy:

```
print('Test accuracy:', test_acc)
Test accuracy: 0.9018
```

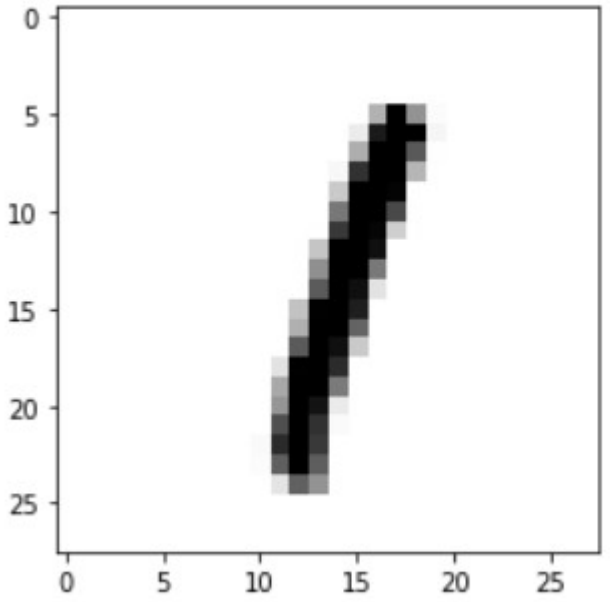
The accuracy is telling us that the model we have created in this post, applied to data that the model has never seen before, classifies 90% of them correctly.

Generate predictions

Finally, readers need to know how we can use the model trained in the previous section to make predictions. In our example, it consists in predict which digit represents an image. In order to do this, Keras supply the `predict()` method.

Let's choose one image (and plot it) in order to predict the number:

```
image = 5

_ = plt.imshow(x_test_[image], cmap=plt.cm.binary)
```



and in order to predict the number we can use the following code:

```
import numpy as np

prediction = model.predict(x_test_)
print("Model prediction: ", np.argmax(prediction[image]) )
```

HOMEWORK:

1) Implement an LSTM network in TensorFlow

2) Explain the models in detail

3) Come up with an interesting use-case application, ie solve a practical problem

4) prepare a ppt presentation of your results

- End of TensorFlow Lecture -

- 4) Beginning of Adversarial Neural Networks -

Generative Models

- Collect large amount of data in some domain
- Train generative model to generate data like it

Generative Models

- Given training data generate samples from same distribution (density estimation)

Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

Source: Fei-Fei Li et al.

Why Generative Models?

- Some Current applications
  - Semi-supervised learning (pretraining) in cases where labeled data is expensive
  - dataset augmentation
  - image denoising, super-resolution
  - Other thoughts?
- Fairness applications (today)

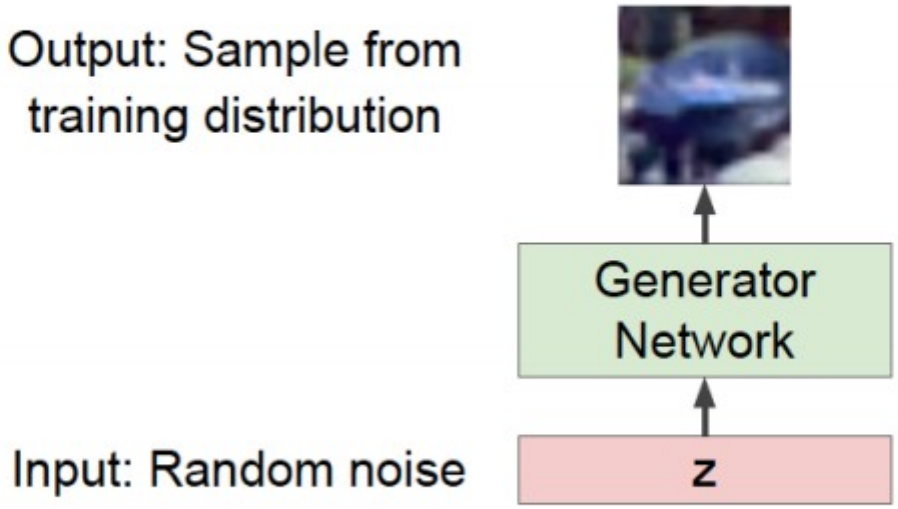
GANs

- Goal: Sample from complex, high-dimensional training distribution

- Approach
  - Sample from a simple distribution (e.g., random noise)
  - Learn transformation from noise to training distribution

- Question
  - How to represent this complex transformation?
  - A neural network!

GANs


Output: Sample from training distribution
Generator Network
Input: Random noise
z

How do we Learn a Good Generator?

- The generator's goal is to map from random noise to an instance from the training distribution.
- How do we learn this mapping?
  - Assign random index to each training point and try to learn the mapping from index to point?
  - Idea: Learn to distinguish between realistic (true) and non-realistic (generated) points to find the space of realistic points.

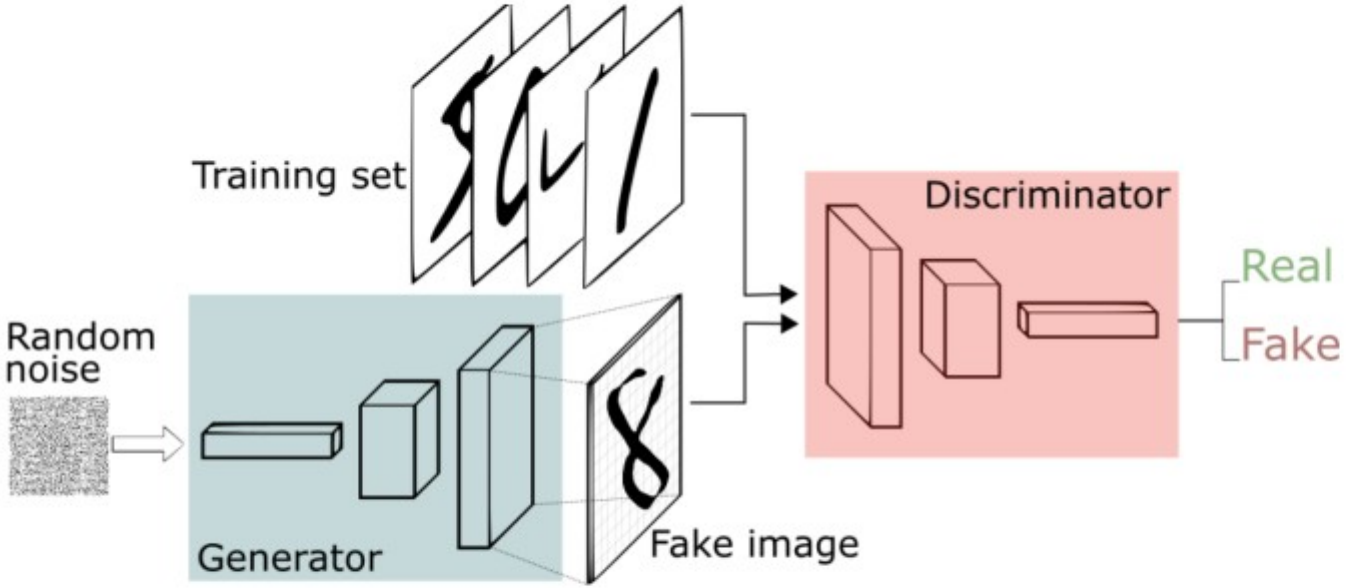
we could think of the noise as an index into the training distribution

probably too constrained – we don't care about any particular mapping

How do we Learn a Good Generator?

- The generator's goal is to map from random noise to an instance from the training distribution.
- How do we learn this mapping?
  - Idea: Learn to distinguish between realistic (true) and non-realistic (generated) points to find the space of realistic points.

GAN Training: Two Player Game

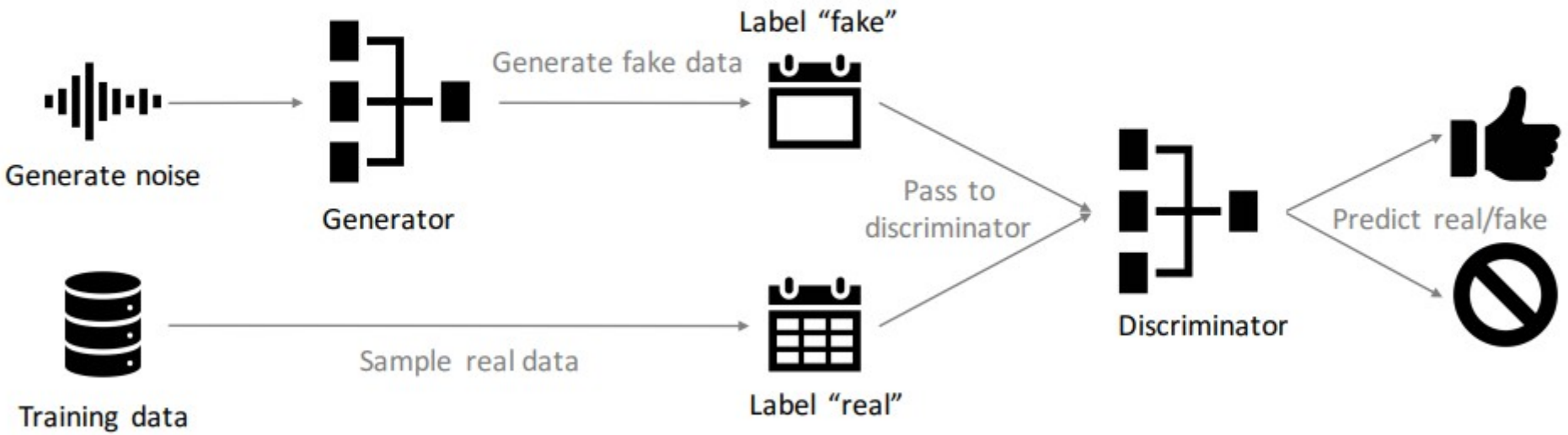


Generative Adversarial Network framework.

Discriminator's job: decipher which of its inputs are real data and which are from the generator (fake)

Generator's job: fool the discriminator by creating images that look like the real images from the training set

Illustration



Training GANs

we can think of this objective as capturing the discriminator's ability to distinguish between real examples and generated examples

Train jointly in **minimax game**

Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

generator is in control of these parameters only

discriminator is in control of these parameters only

- Discriminator (θ_d) wants to **maximize objective** such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)

- Generator (θ_g) wants to **minimize objective** such that D(G(z)) is close to 1 (discriminator is fooled into thinking generated G(z) is real)

Training GANs

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

here, consider that we are "freezing" the generator's weights

2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

here, consider that we are "freezing" the discriminator's weights

Training GANs

Putting it together: GAN training algorithm

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

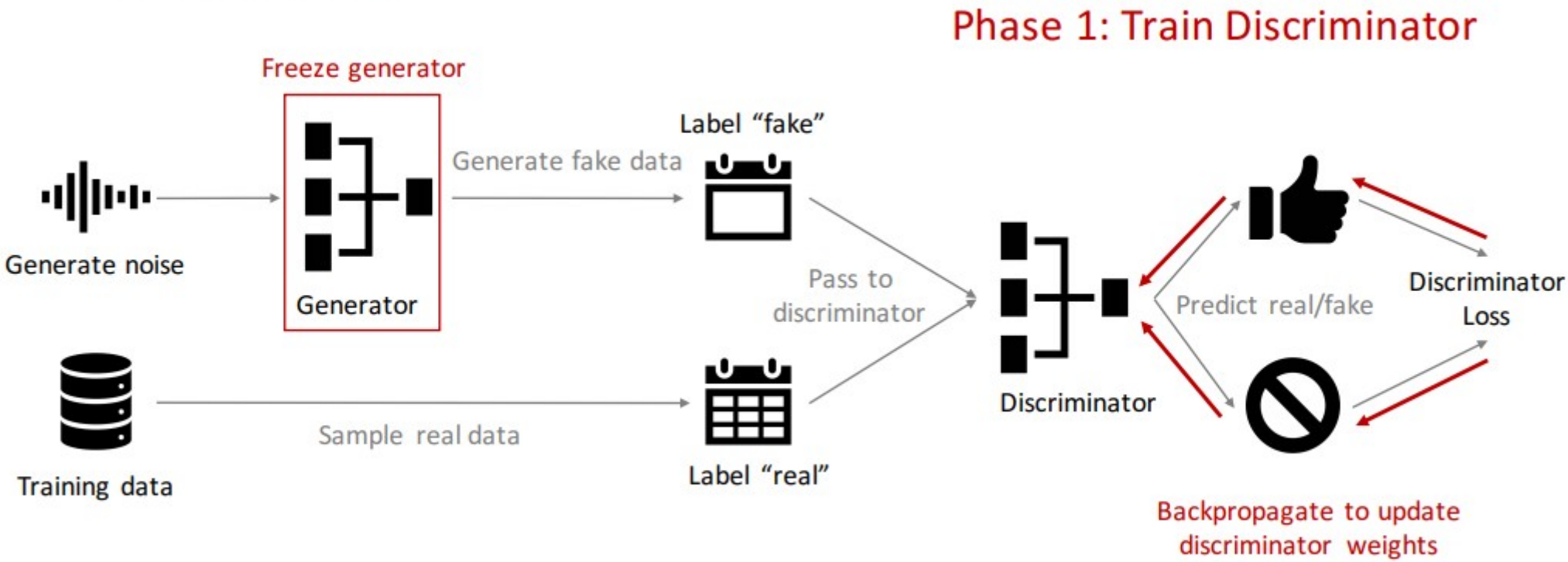
  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by ascending its stochastic gradient (improved objective):

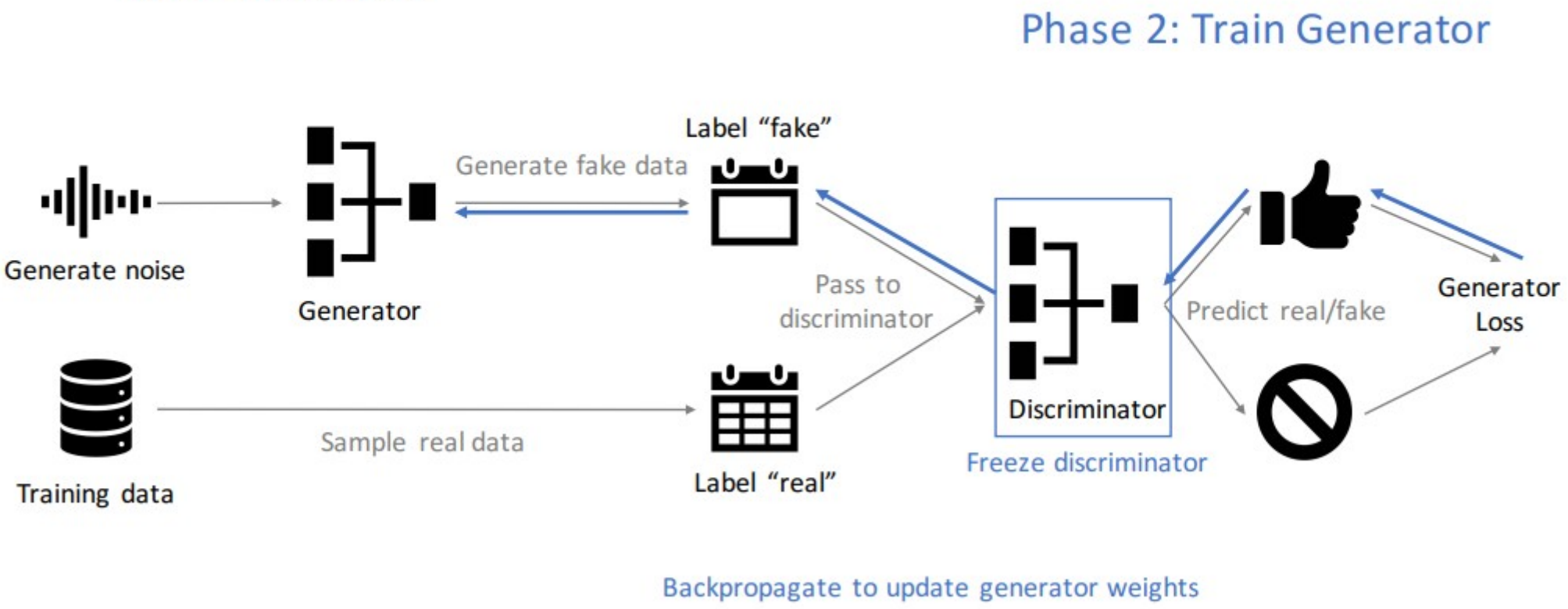
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

Illustration



Illustration



Training GANs

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

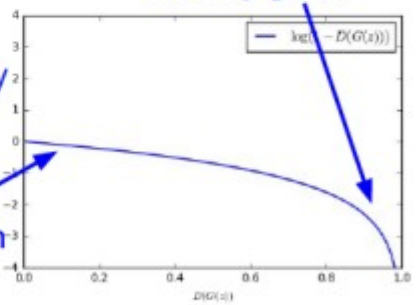
2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

In practice, optimizing this generator objective does not work well!

Gradient signal dominated by region where sample is already good

When sample is likely fake, want to learn from it to improve generator. But gradient in this region is relatively flat!



Training GANs

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

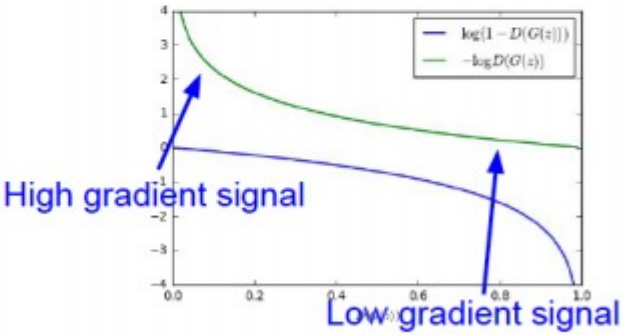
1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Instead: Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.
Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.



HOMEWORK:

1) Implement a GAN in TensorFlow

2) Explain the models in detail

3) Come up with an interesting use-case application, ie solve a practical problem

4) prepare a ppt presentation of your results

Final Project:

1) Implement a medical doctor application that can diagnose cancer

2) Use a deep CNN for image classification

3) Add a user interface for upload functionality

4) Implement a NLP speech synthesis interface to communicate the diagnoses result

5) Prepare a ppt on your work

Requirements:

Session 5:
each group presents 5 minutes of their idea on the project

Session 6:
each group presents 10 minutes on the implementation

Session 7:
each group gives a 15 minutes presentation on the final project + 5 page written summary

- diagnoses of skin cancer using medical images and CNN
- diagnoses of brain cancer using medical images and CNN
- diagnoses of lung cancer using medical images and CNN
- diagnose blood disease
- diagnose Alzheimer
- diagnose heart disease based on frequency analysis of heart beats and LSTM

- End of Adversarial Neural Networks -

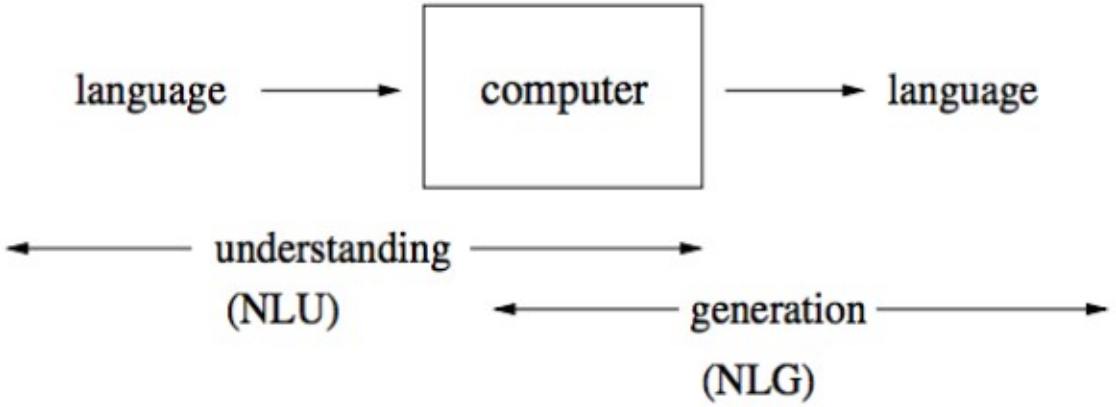
- 5) Beginning of NLP and Deep Learning -

What is Natural Language Processing

- Natural Language Processing is related to the area of human-computer interaction.
- Natural language understanding
- Natural language generation

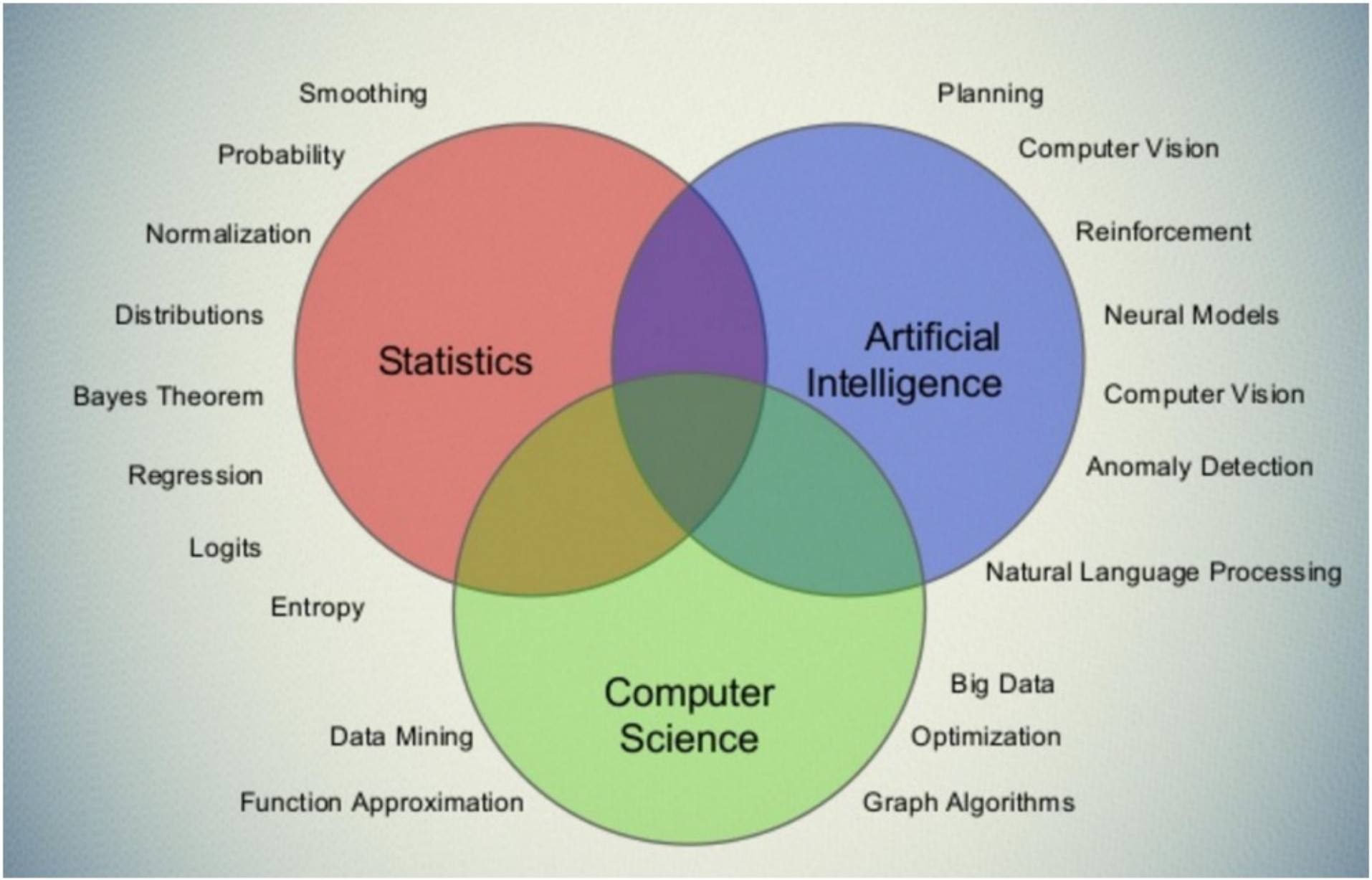
Natural Language Processing

computers using natural language as input and/or output



https://d396qusza40orc.cloudfront.net/nlangp/lectures/intro.pdf
http://www.cs.nyu.edu/~petrov/lecture1.pdf

Natural Language Processing



http://www.slideshare.net/BenjaminBengfort/introduction-to-machine-learning-with-scikitlearn

NLP Applications

- Information Extraction
- Name Entity Recognition
- Machine Translation
- Question Answering
- Topic Model
- Summarization

Information Extraction

10TH DEGREE is a full service advertising agency specializing in direct and interactive marketing. Located in Irvine CA, 10TH DEGREE is looking for an Assistant Account Manager to help manage and coordinate interactive marketing initiatives for a marquee automative account. Experience in online marketing, automative and/or the advertising field is a plus. Assistant Account Manager Responsibilities Ensures smooth implementation of programs and initiatives Helps manage the delivery of projects and key client deliverables . . . Compensation: $50,000-$80,000 Hiring Organization: 10TH DEGREE

⇓

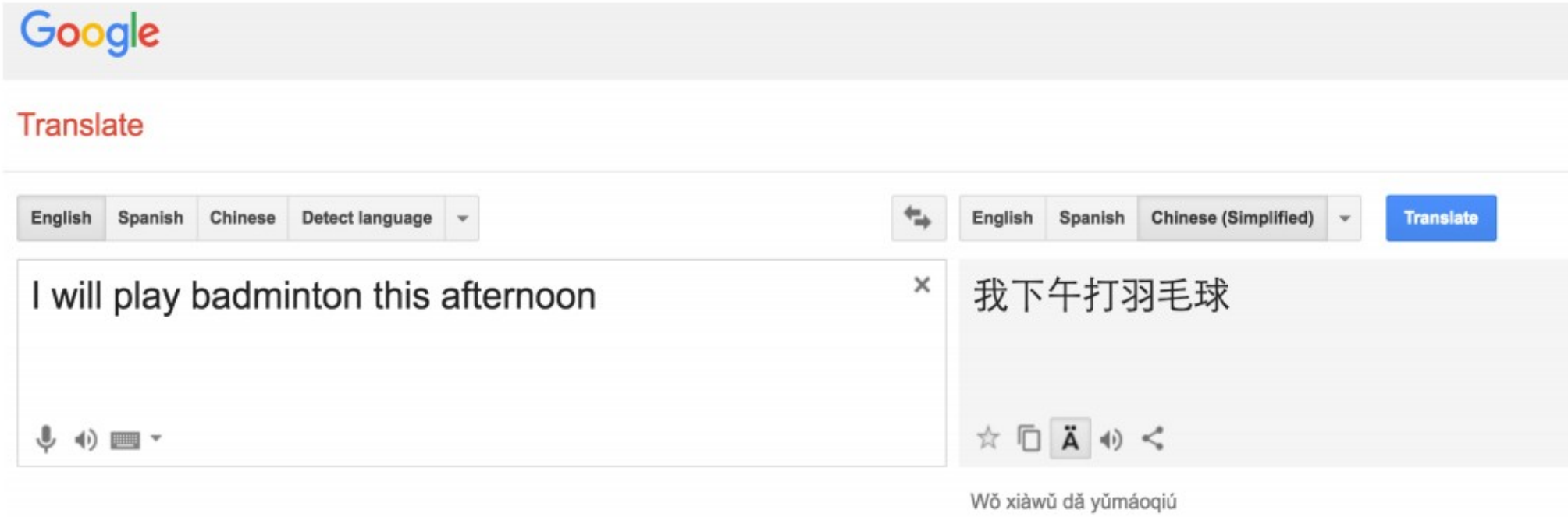
INDUSTRY	Advertising
POSITION	Assistant Account Manager
LOCATION	Irvine, CA
COMPANY	10TH DEGREE
SALARY	$50,000-$80,000

https://d396qusza40orc.cloudfront.net/nlangp/lectures/intro.pdf

Name Entity Recognition

- Classify elements in text into categories such as location, time, name of person, organization.
- Jim worked in Google corp. in 2012
- (Jim)[person] worked in (Google corp.)[organization] in (2012)[time]

Machine Translation



Machine Translation Difficulties

- Words together are more than the sum of their parts.
- Can not translated word by word
  - E.g, Fast food, Light rain
- Need a big dictionary with grammar rules in both languages, large start-up cost
- Require computer to understand

Question Answering

- IBM Watson won Jeopardy on 02/16/2011

Question Answering

Google

what is the birthdate of barack obama

All News Images Videos Shopping More Search tools

About 287,000 results (0.54 seconds)

Barack Obama / Date of birth

August 4, 1961 (age 54 years)

Michelle Obama
Spouse
January 17, 1964

Malia Ann Obama
Daughter
July 4, 1998

Ann Dunham
Mother
November 29, 1942

Bara
44th U.S. P
Barack Hu
President
office. Wiki
Born: Aug
Height: 6'
Spouse: M
Parents: A
Children:
Education
Profiles

Question Answering

Google

who is the author of game of thrones

All News Images Videos Shopping More Search tools

About 6,030,000 results (0.60 seconds)

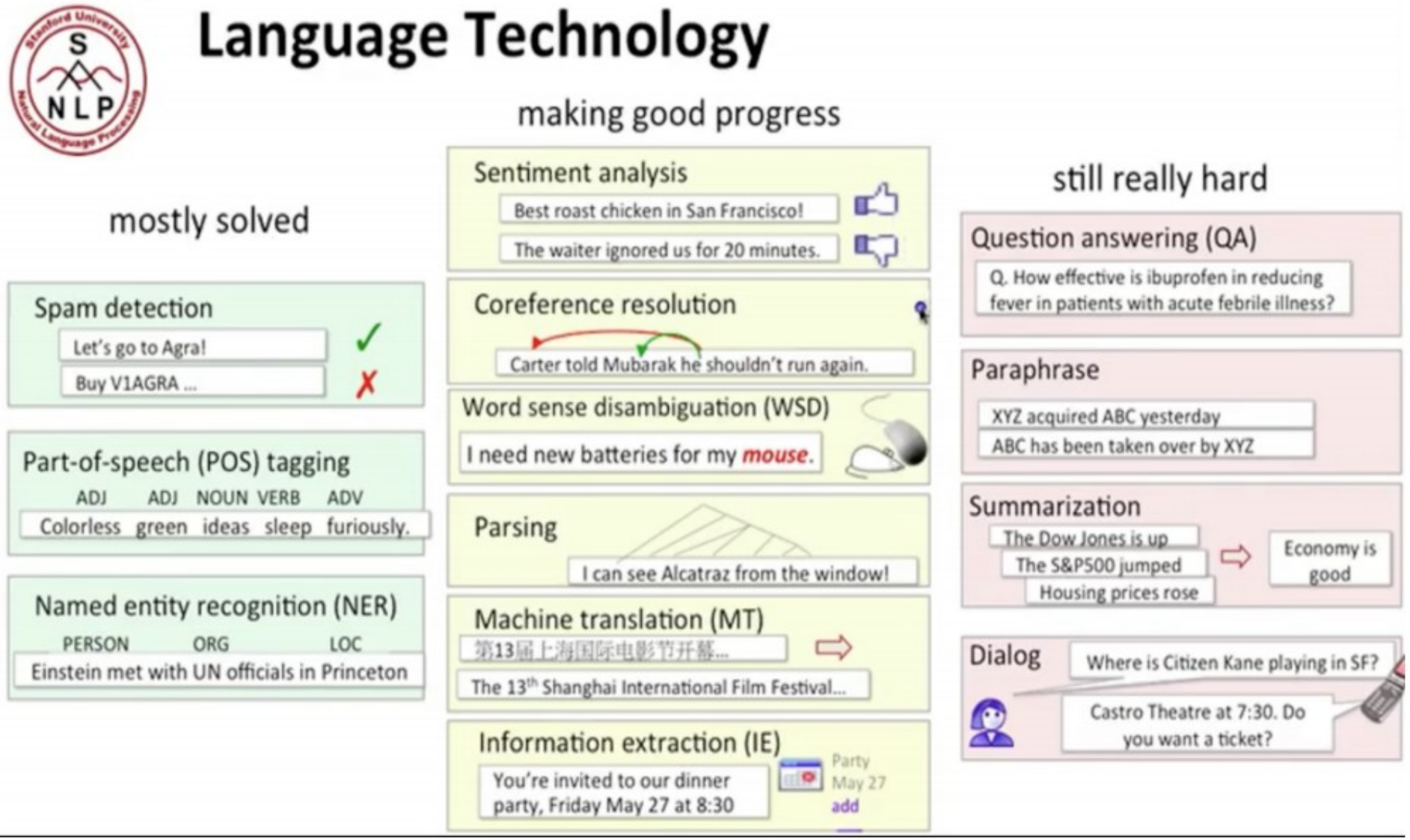
A Game of Thrones / Author

George R. R. Martin

Quotes, books, and overview

Feedback

NLP Tasks



https://class.coursera.org/nlp/lecture/124

Why NLP is hard

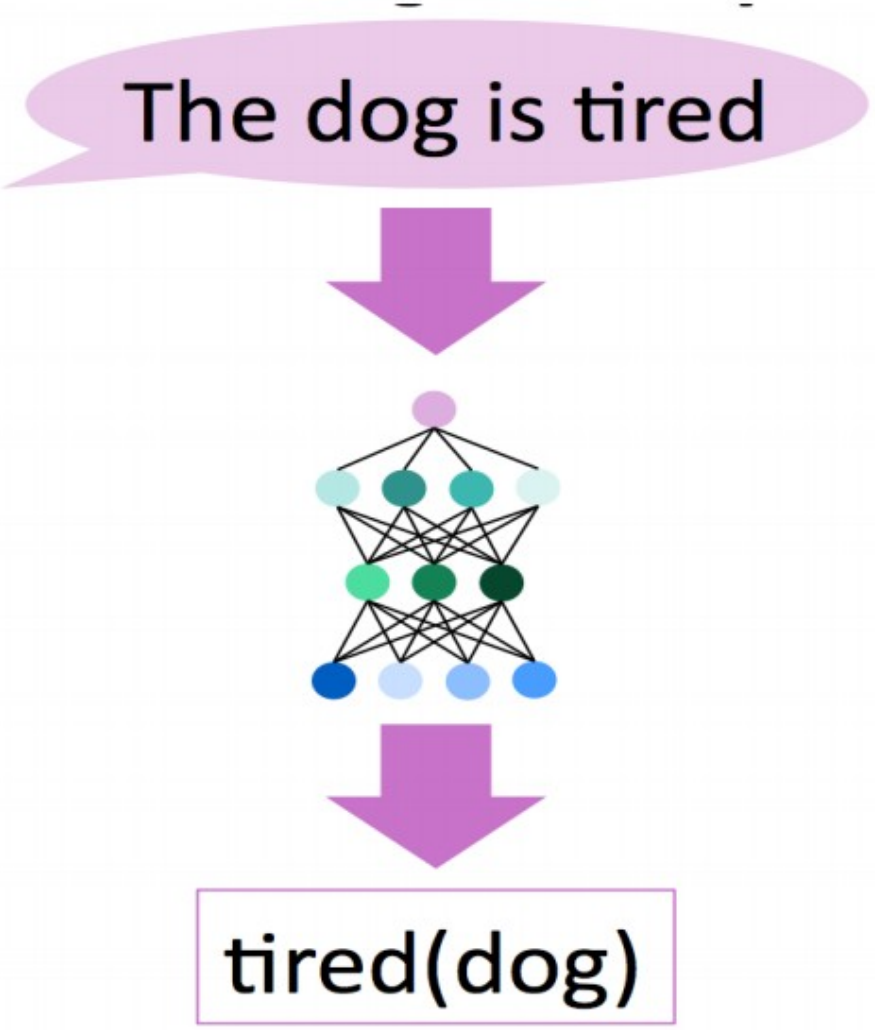
- Basically text is not computer-friendly
- Many different ways to represent the same thing
- Order and context are extremely important
- Language is very high dimensional and sparse. Tons of rare words.
  - B4 (before), IC (I see), cre8(create)
- Ambiguity

Ambiguity

- "At last, a computer understands you like your mother"
  - It understands you as well as your mother understands you
  - It understands (that) you like your mother
  - It understands you as well as it understands your mother

DEEP LEARNING IN NATURAL LANGUAGE PROCESSING

Deep Learning (Representation learning) in NLP



http://www.iro.umontreal.ca/~memisevr/dlss2015/DLSS2015-NLP-1.pdf

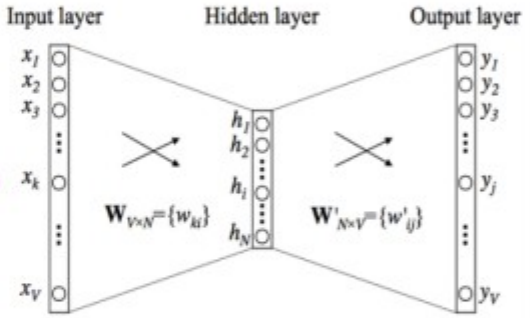
Deep Learning in NLP

- Word Level Application: Word Embedding, word2vec
- Sentence/paragraph Level Application: Neural Machine Translation, doc2vec, etc.

Word Representation

- The majority of rule-based and statistical NLP work regarded words as atomic symbols
- In vector space terms, this is a vector with one 1 and many zeros, it is called "one-hot" representation
  - Condo: [0,0,0,0,1,0,0,…0]
  - Apartment: [0,1,0,0,0,0,0,…0]
- These two vectors are orthogonal, no similarity

Word2vec



word2vec

Supply text

Wait a few hours

Get a $d \times |V|$ matrix

A vector for each word

What Are Word Embeddings?

A word embedding is a learned representation for text where words that have the same meaning have a similar representation.

It is this approach to representing words and documents that may be considered one of the key breakthroughs of deep learning on challenging natural language processing problems.

> *One of the benefits of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors. … The main benefit of the dense representations is generalization power: if we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities.*

— Page 92, Neural Network Methods in Natural Language Processing, 2017.

Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning.

Key to the approach is the idea of using a dense distributed representation for each word.

Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding.

> *associate with each word in the vocabulary a distributed word feature vector … The feature vector represents different aspects of the word: each word is associated with a point in a vector space. The number of features … is much smaller than the size of the vocabulary*

— A Neural Probabilistic Language Model, 2003.

The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag of words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

There is deeper linguistic theory behind the approach, namely the "*distributional hypothesis*" by Zellig Harris that could be summarized as: words that have similar context will have similar meanings. For more depth see Harris' 1956 paper "Distributional structure".

This notion of letting the usage of the word define its meaning can be summarized by an oft repeated quip by John Firth:

> *You shall know a word by the company it keeps!*

Word Embedding Algorithms

Word embedding methods learn a real-valued vector representation for a predefined fixed sized vocabulary from a corpus of text.

The learning process is either joint with the neural network model on some task, such as document classification, or is an unsupervised process, using document statistics.

This section reviews three techniques that can be used to learn a word embedding from text data.

1. Embedding Layer

An embedding layer, for lack of a better name, is a word embedding that is learned jointly with a neural network model on a specific natural language processing task, such as language modeling or document classification.

It requires that document text be cleaned and prepared such that each word is one-hot encoded. The size of the vector space is specified as part of the model, such as 50, 100, or 300 dimensions. The vectors are initialized with small random numbers. The embedding layer is used on the front end of a neural network and is fit in a supervised way using the Backpropagation algorithm.

> *… when the input to a neural network contains symbolic categorical features (e.g. features that take one of k distinct symbols, such as words from a closed vocabulary), it is common to associate each possible feature value (i.e., each word in the vocabulary) with a d-dimensional vector for some d. These vectors are then considered parameters of the model, and are trained jointly with the other parameters.*

— Page 49, Neural Network Methods in Natural Language Processing, 2017.

The one-hot encoded words are mapped to the word vectors. If a multilayer Perceptron model is used, then the word vectors are concatenated before being fed as input to the model. If a recurrent neural network is used, then each word may be taken as one input in a sequence.

This approach of learning an embedding layer requires a lot of training data and can be slow, but will learn an embedding both targeted to the specific text data and the NLP task.

2. Word2Vec

Word2Vec is a statistical method for efficiently learning a standalone word embedding from a text corpus.

It was developed by Tomas Mikolov, et al. at Google in 2013 as a response to make the neural-network-based training of the embedding more efficient and since then has become the de facto standard for developing pre-trained word embedding.

Additionally, the work involved analysis of the learned vectors and the exploration of vector math on the representations of words. For example, that subtracting the "*man-ness*" from "*King*" and adding "*women-ness*" results in the word "*Queen*", capturing the analogy "*king is to queen as man is to woman*".

> *We find that these representations are surprisingly good at capturing syntactic and semantic regularities in language, and that each relationship is characterized by a relation-specific vector offset. This allows vector-oriented reasoning based on the offsets between words. For example, the male/female relationship is automatically learned, and with the induced vector representations, "King – Man + Woman" results in a vector very close to "Queen."*

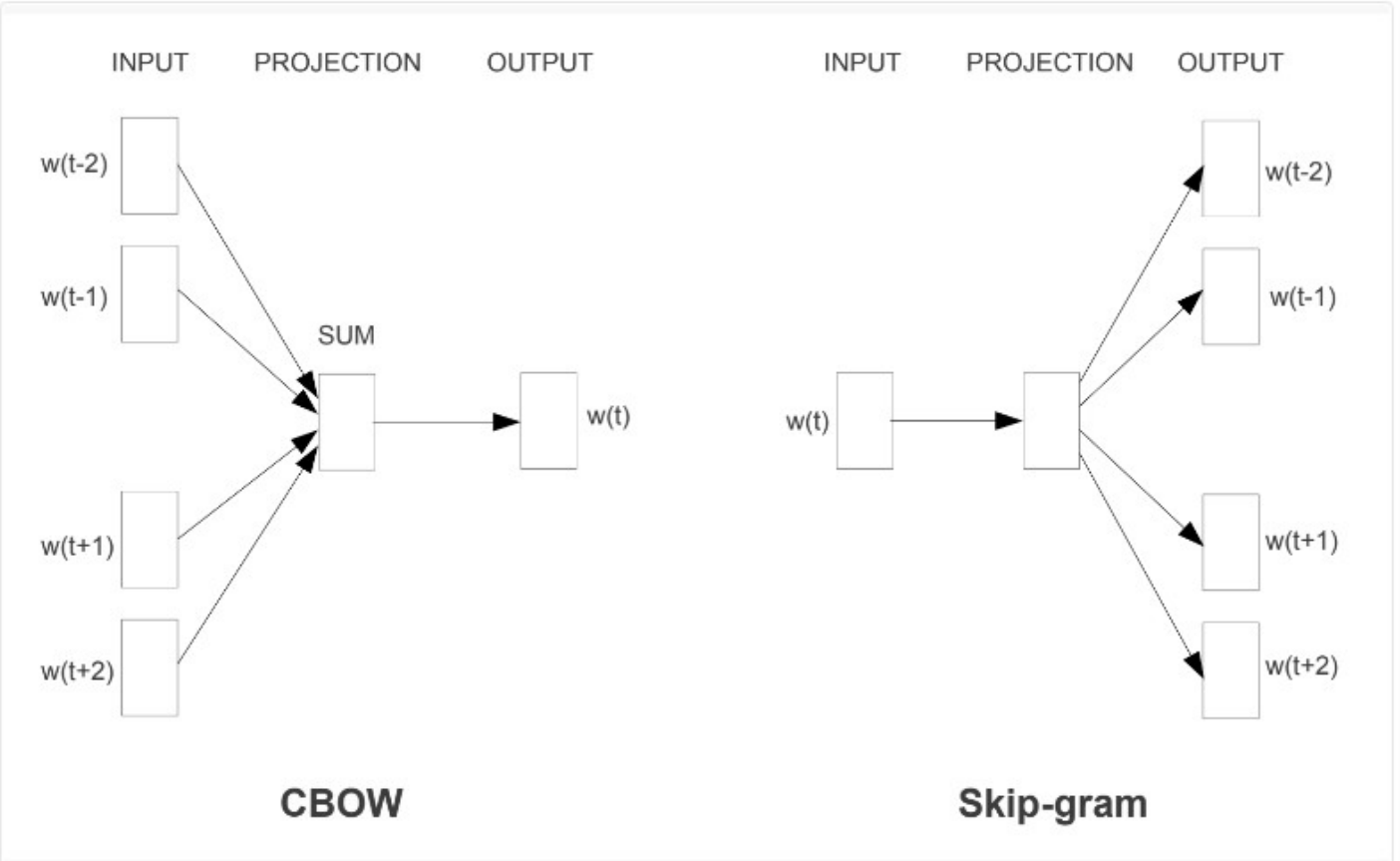
— Linguistic Regularities in Continuous Space Word Representations, 2013.

Two different learning models were introduced that can be used as part of the word2vec approach to learn the word embedding; they are:

- Continuous Bag-of-Words, or CBOW model.
- Continuous Skip-Gram Model.

The CBOW model learns the embedding by predicting the current word based on its context. The continuous skip-gram model learns by predicting the surrounding words given a current word.

The continuous skip-gram model learns by predicting the surrounding words given a current word.



Word2Vec Training Models
Taken from “Efficient Estimation of Word Representations in Vector Space”, 2013

Both models are focused on learning about words given their local usage context, where the context is defined by a window of neighboring words. This window is a configurable parameter of the model.

> *The size of the sliding window has a strong effect on the resulting vector similarities. Large windows tend to produce more topical similarities […], while smaller windows tend to produce more functional and syntactic similarities.*

— Page 128, Neural Network Methods in Natural Language Processing, 2017.

The key benefit of the approach is that high-quality word embeddings can be learned efficiently (low space and time complexity), allowing larger embeddings to be learned (more dimensions) from much larger corpora of text (billions of words).

Using Word Embeddings

You have some options when it comes time to using word embeddings on your natural language processing project.

This section outlines those options.

1. Learn an Embedding

You may choose to learn a word embedding for your problem.

This will require a large amount of text data to ensure that useful embeddings are learned, such as millions or billions of words.

You have two main options when training your word embedding:

1. **Learn it Standalone**, where a model is trained to learn the embedding, which is saved and used as a part of another model for your task later. This is a good approach if you would like to use the same embedding in multiple models.
2. **Learn Jointly**, where the embedding is learned as part of a large task-specific model. This is a good approach if you only intend to use the embedding on one task.

2. Reuse an Embedding

It is common for researchers to make pre-trained word embeddings available for free, often under a permissive license so that you can use them on your own academic or commercial projects.

For example, both word2vec and GloVe word embeddings are available for free download.

These can be used on your project instead of training your own embeddings from scratch.

You have two main options when it comes to using pre-trained embeddings:

1. **Static**, where the embedding is kept static and is used as a component of your model. This is a suitable approach if the embedding is a good fit for your problem and gives good results.
2. **Updated**, where the pre-trained embedding is used to seed the model, but the embedding is updated jointly during the training of the model. This may be a good option if you are looking to get the most out of the model and embedding on your task.

How do I Start with NLP using Python?

Natural language toolkit (NLTK) is the most popular library for natural language processing (NLP) which was written in Python and has a big community behind it.

NLTK also is very easy to learn, actually, it's the easiest natural language processing (NLP) library that you'll use.

In this NLP Tutorial, we will use Python NLTK library.

Before I start installing NLTK, I assume that you know some **Python basics** to get started.

Install nltk

Here we will learn how to identify what the web page is about using NLTK in Python

First, we will grab a webpage and analyze the text to see what the page is about.

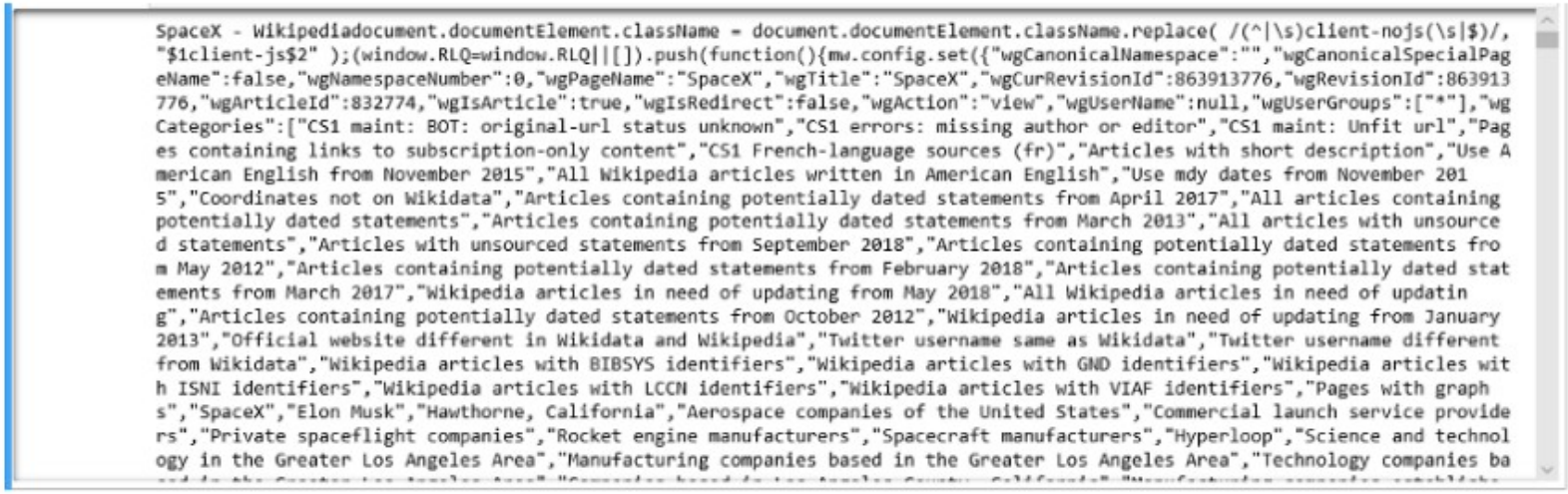
urllib module will help us to crawl the webpage

```
import urllib.request
response =
urllib.request.urlopen('https://en.wikipedia.org/wiki/SpaceX')
html = response.read()
print(html)
```


It's pretty clear from the link that page is about SpaceX now let us see whether our code is able to correctly identify the page's context.

We will use **Beautiful Soup** which is a Python library for pulling data out of HTML and XML files. We will use beautiful soup to clean our webpage text of HTML tags.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html,'html5lib')
text = soup.get_text(strip = True)
print(text)
```



You will get an output somewhat like this

Now we have clean text from the crawled web page, let's convert the text into tokens.

```
tokens = [t for t in text.split()]
print(tokens)
```

your output text is now converted into tokens

```
['SpaceX', '-', 'Wikipediadocument.documentElement.className', '=', 'document.documentElement.className.replace(', '/(^|\\s)c
lient-nojs(\\s|$)/,', '"$1client-js$2"', ');(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({"wgCanonicalNamespac
e":"","wgCanonicalSpecialPageName":false,"wgNamespaceNumber":0,"wgPageName":"SpaceX","wgTitle":"SpaceX","wgCurRevisionId":863
913776,"wgRevisionId":863913776,"wgArticleId":832774,"wgIsArticle":true,"wgIsRedirect":false,"wgAction":"view","wgUserName":n
ull,"wgUserGroups":["*"],"wgCategories":["CS1', 'maint:', 'BOT:', 'original-url', 'status', 'unknown","CS1', 'errors:', 'miss
ing', 'author', 'or', 'editor","CS1', 'maint:', 'Unfit', 'url","Pages', 'containing', 'links', 'to', 'subscription-only', 'co
ntent","CS1', 'French-language', 'sources', '(fr)","Articles', 'with', 'short', 'description","Use', 'American', 'English',
'from', 'November', '2015","All', 'Wikipedia', 'articles', 'written', 'in', 'American', 'English","Use', 'mdy', 'dates', 'fro
m', 'November', '2015","Coordinates', 'not', 'on', 'Wikidata","Articles', 'containing', 'potentially', 'dated', 'statement
s', 'from', 'April', '2017","All', 'articles', 'containing', 'potentially', 'dated', 'statements","Articles', 'containing',
'potentially', 'dated', 'statements', 'from', 'March', '2013","All', 'articles', 'with', 'unsourced', 'statements","Article
s', 'with', 'unsourced', 'statements', 'from', 'September', '2018","Articles', 'containing', 'potentially', 'dated', 'stateme
nts', 'from', 'May', '2012","Articles', 'containing', 'potentially', 'dated', 'statements', 'from', 'February', '2018","Artic
les', 'containing', 'potentially', 'dated', 'statements', 'from', 'March', '2017","Wikipedia', 'articles', 'in', 'need', 'o
f', 'updating', 'from', 'May', '2018","All', 'Wikipedia', 'articles', 'in', 'need', 'of', 'updating","Articles', 'containin
g', 'potentially', 'dated', 'statements', 'from', 'October', '2012","Wikipedia', 'articles', 'in', 'need', 'of', 'updating',
'from', 'January', '2013","Official', 'website', 'different', 'in', 'Wikidata', 'and', 'Wikipedia","Twitter', 'username', 'sa
me', 'as', 'Wikidata","Twitter', 'username', 'different', 'from', 'Wikidata","Wikipedia', 'articles', 'with', 'BIBSYS', 'iden
tifiers","Wikipedia', 'articles', 'with', 'GND', 'identifiers","Wikipedia', 'articles', 'with', 'ISNI', 'identifiers","Wikipe
```

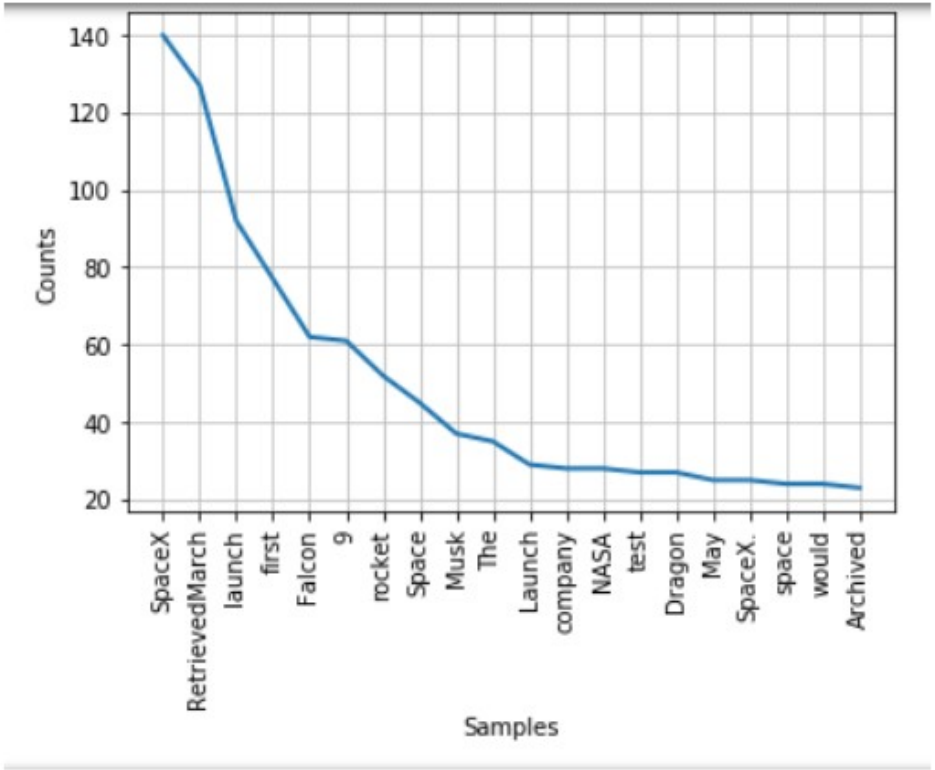
Count word Frequency

nltk offers a function **FreqDist()** which will do the job for us. Also, we will remove stop words (a, at, the, for etc) from our web page as we don't need them to hamper our word frequency count. We will plot the graph for most frequently occurring words in the webpage in order to get the clear picture of the context of the web page

```
from nltk.corpus import stopwords
sr= stopwords.words('english')
clean_tokens = tokens[:]
for token in tokens:
    if token in stopwords.words('english'):

        clean_tokens.remove(token)


freq = nltk.FreqDist(clean_tokens)
for key,val in freq.items():
    print(str(key) + ':' + str(val))


freq.plot(20, cumulative=False)
```

```
SpaceX:140
-:5
Wikipediadocument.documentElement.className:1
-:1
document.documentElement.className.replace(:1
/(^|\s)client-nojs(\s|$)/,:1
"$1client-js$2":1
);(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":false,"wg
NamespaceNumber":0,"wgPageName":"SpaceX","wgTitle":"SpaceX","wgCurRevisionId":863913776,"wgRevisionId":863913776,"wgArticleI
d":832774,"wgIsArticle":true,"wgIsRedirect":false,"wgAction":"view","wgUserName":null,"wgUserGroups":["*"],"wgCategories":["C
S1:1
maint::15
BOT::12
original-url:12
status:12
unknown","CS1:1
errors::2
missing:2
author:2
```

frequency word count output



graph of 20 most frequent words.

Great!!! the code has correctly identified that the web page speaks about **SpaceX**.

It was so simple and interesting right !!! you can similarly identify the news articles, blogs etc.

I have done my best to make the article simple and interesting for you, hope you found it useful and interesting too.

You have successfully taken your first step towards NLP, there is an ocean to explore for you…

Build your first chatbot using Python NLTK

"A **chatbot** (also known as a **talkbot**, **chatterbot**, **Bot**, **IM bot**, **interactive agent**, or **Artificial Conversational Entity**) is a computer program or an artificial intelligence which conducts a conversation via auditory or textual methods. Such programs are often designed to convincingly simulate how a human would behave as a conversational partner, thereby passing the Turing test. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatterbots use sophisticated natural language processing systems, but many simpler systems scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database."

Chatbots are not very new, one of the foremost of this kind is ELIZA, which was created in the early 1960s and is worth exploring. In order to successfully build a conversational engine, it should take care of the following things:

1. Understand who is the target audience
2. Understand the Natural Language of the communication.
3. Understand the intent or desire of the user
4. provide responses that can answer the user

Today we will learn to create a simple chat assistant or chatbot using Python's NLTK library.

NLTK has a module, nltk.chat, which simplifies building these engines by providing a generic framework.

Chat: This is a class that has all the logic that is used by the chatbot.

Reflections: This is a dictionary that contains a set of input values and its corresponding output values. It is an optional dictionary that you can use. You can also create your own dictionary in the same format as below and use it in your code. If you check nltk.chat.util, you will see its values as below:

```
reflections = {
  "i am"       : "you are",
  "i was"      : "you were",
  "i"          : "you",
  "i'm"        : "you are",
  "i'd"        : "you would",
  "i've"       : "you have",
  "i'll"       : "you will",
  "my"         : "your",
  "you are"    : "I am",
  "you were"   : "I was",
  "you've"     : "I have",
  "you'll"     : "I will",
  "your"       : "my",
  "yours"      : "mine",
  "you"        : "me",
  "me"         : "you"
}
```

You can also create your own reflections dictionary in the same format as above and use it in your code. Here is an example for this:

```
my_dummy_reflections= {
    "go"     : "gone",
    "hello"    : "hey there"
}
```

and use it as :

```
chat = Chat(pairs, my_dummy_reflections)
```

Using above concept from python's NLTK library, lets build a simple chatbot without using any of the Machine Learning or Deep Learning Algorithms. So obviously our chatbot will be a decent one but not an intelligent one.

Source Code :

```
from nltk.chat.util import Chat, reflections

pairs = [
    [
        r"my name is (.*)",
        ["Hello %1, How are you today ?",]
    ],
     [
        r"what is your name ?",
        ["My name is Chatty and I'm a chatbot ?",]
    ],
    [
        r"how are you ?",
        ["I'm doing good\nHow about You ?",]
    ],
    [
        r"sorry (.*)",
        ["Its alright","Its OK, never mind",]
    ],
    [
        r"i'm (.*) doing good",
        ["Nice to hear that","Alright :)",]
    ],
    [
        r"hi|hey|hello",
        ["Hello", "Hey there",]
    ],
```

```
    [
        r"(.*) age?",
        ["I'm a computer program dude\nSeriously you are asking me
this?",]

    ],
    [
        r"what (.*) want ?",
        ["Make me an offer I can't refuse",]

    ],
    [
        r"(.*) created ?",
        ["Nagesh created me using Python's NLTK library ","top secret
;)",]
    ],
    [
        r"(.*) (location|city) ?",
        ['Chennai, Tamil Nadu',]
    ],
    [
        r"how is weather in (.*)?",
        ["Weather in %1 is awesome like always","Too hot man here in
%1","Too cold man here in %1","Never even heard about %1"]
    ],
    [
        r"i work in (.*)?",
        ["%1 is an Amazing company, I have heard about it. But they
are in huge loss these days.",]
    ],
```

.....

```
def chatty():
        print("Hi, I'm Chatty and I chat alot ;)\nPlease type
lowercase English language to start a conversation. Type quit to
leave ") #default message at the start


    chat = Chat(pairs, reflections)
    chat.converse()
if __name__ == "__main__":
    chatty()
```

The code is quite simple, still lets understand it.

Once the function chatty() is invoked , a default message will be displayed:

```
In [23]: runfile('D:/KJ/Nagesh/Downloads/chatbot/chatbot_1.py', wdir='D:/KJ/Nagesh/Downloads/chatbot')
Hi, I'm Chatty and I chat alot ;)
Please type lowercase English language. Type quit to leave

>
```

Next I've created an instance of Chat class containing **pairs**(list of tuples containing set of question and answers) and **reflections**(discussed above).

Next step is to trigger the conversation:

```
chat.converse()
```

A simple conversation :

```
In [27]: runfile('D:/KJ/Nagesh/Downloads/chatbot/chatbot_1.py', wdir='D:/KJ/Nagesh/Downloads/chatbot')
Hi, I'm Chatty and I chat alot ;)
Please type lowercase English language to start a conversation. Type quit to leave

>hi
Hey there

>how are you ?
I'm doing good
How about You ?

>i'm also doing good
Nice to hear that

>what is your name ?
My name is Chatty and I'm a chatbot

>what is your age ?
I'm a computer program dude
Seriously you are asking me this?

>sorry Chatty
Its OK, never mind

>
```

a simple conversation with Chatty

As you can see we have just hardcoded the probable question and answers in the list pairs.

As you can see we have just hardcoded the probable question and answers in the list pairs.

Lets interact more with Chatty :

```
In [29]: runfile('D:/KJ/Nagesh/Downloads/chatbot/chatbot_1.py', wdir='D:/KJ/Nagesh/Downloads/chatbot')
Hi, I'm Chatty and I chat alot ;)
Please type lowercase English language to start a conversation. Type quit to leave

>hi
Hello

>what is your location ?
Chennai, Tamil Nadu

>how is weather in Chennai
Too cold man here in chennai

>i work in xyz company
xyz company is an Amazing company, I have heard about it. But they are in huge loss these days.

>is it raining in chennai
Damn its raining too much here in chennai

>what is your fav game ?
I'm a very big fan of Football

>who is your fav moviestar ?
Brad Pitt

>quit
BBye take care. See you soon :)

In [30]:
```

a simple conversation with Chatty

A short introduction to NLP in Python with spaCy

spaCy is a relatively new package for "Industrial strength NLP in Python" developed by Matt Honnibal at Explosion AI. It is designed with the applied data scientist in mind, meaning it does not weigh the user down with decisions over what esoteric algorithms to use for common tasks and it's fast. Incredibly fast (it's implemented in Cython). If you are familiar with the Python data science stack, spaCy is your `numpy` for NLP – it's reasonably low-level, but very intuitive and performant.

So, what can it do?

spacy provides a one-stop-shop for tasks commonly used in any NLP project, including:

- Tokenisation
- Lemmatisation
- Part-of-speech tagging
- Entity recognition
- Dependency parsing
- Sentence recognition
- Word-to-vector transformations
- Many convenience methods for cleaning and normalising text

I'll provide a high level overview of some of these features and show how to access them using spaCy.

First, we load spaCy's pipeline, which by convention is stored in a variable named `nlp`. declaring this variable will take a couple of seconds as spaCy loads its models and data to it up-front to save time later. In effect, this gets some heavy lifting out of the way early, so that the cost is not incurred upon each application of the `nlp` parser to your data. Note that here I am using the English language model, but there is also a fully featured German model, with tokenisation (discussed below) implemented across several languages.

We invoke nlp on the sample text to create a `Doc` object. The `Doc` object is now a vessel for NLP tasks on the text itself, slices of the text (`Span` objects) and elements (`Token` objects) of the text. It is worth noting that `Token` and `Span` objects actually hold no data. Instead they contain pointers to data contained in the `Doc` object and are evaluated lazily (i.e. upon request). Much of spaCy's core functionality is accessed through the methods on `Doc` (n=33), `Span` (n=29) and `Token` (n=78) objects.

```
In[1]: import spacy
...: nlp = spacy.load("en")
...: doc = nlp("The big grey dog ate all of the chocolate, but
fortunately he wasn't sick!")
```

Tokenization

Tokenisation is a foundational step in many NLP tasks. Tokenising text is the process of splitting a piece of text into words, symbols, punctuation, spaces and other elements, thereby creating "tokens". A naive way to do this is to simply split the string on white space:

```
In[2]: doc.text.split()
...: Out[2]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the',
'chocolate,', 'but', 'fortunately', 'he', "wasn't", 'sick!']
```

On the surface, this looks fine. But, note that `a)` it disregards the punctuation and, `b)` it does not split the verb and adverb ("was", "n't"). Put differently, it is naive, it fails to recognise elements of the text that help us (and a machine) to understand its structure and meaning. Let's see how SpaCy handles this:

```
In[3]: [token.orth_ for token in doc]
...: Out[3]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the',
'chocolate', ',', 'but', 'fortunately', 'he', 'was', "n't", ' ',
'sick', '!']
```

Here we access the each token's `.orth_` method, which returns a string representation of the token rather than a SpaCy token object, this might not always be desirable, but worth noting. SpaCy recognises punctuation and is able to split these punctuation tokens from word tokens. Many of SpaCy's token method offer both string and integer representations of processed text – methods with an underscore suffix return strings, methods without an underscore suffix return integers. For example:

```
In[4]: [(token, token.orth_, token.orth) for token in doc]
...: Out[4]: [
(The, 'The', 517),
(big, 'big', 742),
(grey, 'grey', 4623),
(dog, 'dog', 1175),
(ate, 'ate', 3469),
(all, 'all', 516),
(of, 'of', 471),
(the, 'the', 466),
(chocolate, 'chocolate', 3593),
(,, ',', 416),
(but, 'but', 494),
(fortunately, 'fortunately', 15520),
 (he, 'he', 514),
 (was, 'was', 491),
 (n't, "n't", 479),
 ( , ' ', 483),
(sick, 'sick', 1698),
(!, '!', 495)]
```

Here, we return the SpaCy token, the string representation of the token and the integer representation of the token in a list of tuples.

If you want to avoid returning tokens that are punctuation or white space, SpaCy provides convienence methods for this (as well as many other common text cleaning tasks — for example, to remove stop words you can call the `.is_stop` method.

```
In[5]: [token.orth_ for token in doc if not token.is_punct |
token.is_space]
...: Out[5]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the',
'chocolate', 'but', 'fortunately', 'he', 'was', "n't", 'sick']
```

Lemmatization

A related task to tokenisation is lemmatisation. Lemmatisation is the process of reducing a word to its base form, its mother word if you like. Different uses of a word often have the same root meaning. For example, `practice, practised and practising` all essentially refer to the same thing. It is often desirable to standardise words with similar meaning to their base form. With SpaCy we can access each word's base form with a token's `.lemma_` method:

```
In[6]: practice = "practice practiced practicing"
...: nlp_practice = nlp(practice)
...: [word.lemma_ for word in nlp_practice]
...: Out[6]: ['practice', 'practice', 'practice']
```

Why is this useful? An immediate use case is in machine learning, specifically text classification. Lemmatising the text prior to, for example, creating a "bag-of-words" avoids word duplication and, therefore, allows for the model to build a clearer picture of patterns of word usage across multiple documents.

POS Tagging

Part-of-speech tagging is the process of assigning grammatical properties (e.g. noun, verb, adverb, adjective etc.) to words. Words that share the same POS tag tend to follow a similar syntactic structure and are useful in rule-based processes.

For example, in a given description of an event we may wish to determine who owns what. By exploiting possessives, we can do this (providing the text is grammatically sound!). SpaCy uses the popular Penn Treebank POS tags, see https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html. With SpaCy you can access coarse and fine-grained POS tags with the `.pos_` and `.tag_` methods, respectively. Here, I access the fine grained POS tag:

```
In[7]: doc2 = nlp("Conor's dog's toy was hidden under the man's sofa
in the woman's house")
...:
pos_tags = [(i, i.tag_) for i in doc2] ...:
pos_tags
...: Out[7]: [(Conor, 'NNP'), ('s, 'POS'), (dog, 'NN'), ('s, 'POS'),
(toy, 'NN'), (was, 'VBD'), (hidden, 'VBN'), (under, 'IN'), (the,
'DT'), (man, 'NN'), ('s, 'POS'), (sofa, 'NN'), (in, 'IN'), (the,
'DT'), (woman, 'NN'), ('s, 'POS'), (house, 'NN')]
```

We can see that the " `'s` " tokens are labelled as `POS`. We can exploit this tag to extract the owner and the thing that they own:

```
In[8]: owners_possessions = []
...: for i in pos_tags:
    ...: if i[1] == "POS": ...: owner = i[0].nbor(-1)
...:         possession = i[0].nbor(1)
...:         owners_possessions.append((owner, possession)) ...: ...:
owners_possessions
...: Out[8]: [(Conor, dog), (dog, toy), (man, sofa), (woman, house)]
```

This returns a list of owner-possession tuples. If you want to be super Pythonic about it, you can do this in a list comprehenion (which, I think is preferable!):

```
In[9]: [(i[0].nbor(-1), i[0].nbor(+1)) for i in pos_tags if i[1] ==
"POS"]
...: Out[9]: [(Conor, dog), (dog, toy), (man, sofa), (woman, house)]
```

Here we are using each token's `.nbor` method which returns a token's neighbouring tokens.

Top

Entity recognition

Entity recognition is the process of classifying named entities found in a text into pre-defined categories, such as persons, places, organizations, dates, etc. spaCy uses a statistical model to classify a broad range of entities, including persons, events, works-of-art and nationalities / religions (see the documentation for the full list https://spacy.io/docs/usage/entity-recognition).

For example, let's take the first two sentences from Barack Obama's wikipedia entry. We will parse this text, then access the identified entities using the `Doc` object's `.ents` method. With this method called on the `Doc` we can access additional `Token` methods, specifically `.label_` and `.label`:

```
In[10]: wiki_obama = """Barack Obama is an American politician who
served as ...: the 44th President of the United States from 2009 to
2017. He is the first ...: African American to have served as
president, ...: as well as the first born outside the contiguous
United States."""
...:
...: nlp_obama = nlp(wiki_obama) ...: [(i, i.label_, i.label) for i
in nlp_obama.ents]
...: Out[10]: [(Barack Obama, 'PERSON', 346), (American, 'NORP',
347), (the United States, 'GPE', 350), (2009 to 2017, 'DATE', 356),
(first, 'ORDINAL', 361), (African, 'NORP', 347), (American, 'NORP',
347), (first, 'ORDINAL', 361), (United States, 'GPE', 350)]
```

You can see the entities that the model has identified and how accurate they are (in this instance). PERSON is self explanatory, NORP is natianalities or religuos groups, GPE identifies locations (cities, countries, etc.), DATE recognises a specific date or date-range and ORDINAL identifies a word or number representing some type of order.

While we are on the topic of `Doc` methods, it is worth mentioning spaCy's sentence identifier. It is not uncommon in NLP tasks to want to split a document into sentences. It is simple to do this with SpaCy by accessing a `Doc's` `.sents` method:

```
In[11]: for ix, sent in enumerate(nlp_obama.sents, 1):
...: print("Sentence number {}: {}".format(ix, sent))
...:
Sentence number 1: Barack Obama is an American politician who served
as the 44th President of the United States from 2009 to 2017.
Sentence number 2: He is the first African American to have served as
president, as well as the first born outside the contiguous United
States.
```

- End of NLP and Deep Learning -

Final Project:

1) Implement a medical doctor application that can diagnose cancer

2) Use a deep CNN for image classification

3) Add a user interface for upload functionality

4) Implement a NLP speech synthesis interface to communicate the diagnoses result

5) Prepare a ppt on your work

Requirements:

Week 5:
each group presents 5 minutes of their idea on the project

Week 6:
each group presents 10 minutes on the implementation

Week 7:
each group gives a 15 minutes presentation on the final project + 5 page written summary