

UNIT – VI

Approach & Methodology

World Standard Methodology: CRISP-DM Methodology, SEMMA Methodology

Real Life Work around Multi-Variate Analytics: A few Selected Commonly used

Techniques: Predictive & Classification Models, Regression, Clustering

Real Life Work around Artificial Intelligence, Machine Learning and Deep Learning: A

few Selected Commonly used Techniques & Algorithms: ANN (Artificial Neural Network), CNN (Convolutional Neural Network), RNN (Recurrent Neural Network);

RN Architecture: LSTM, Bidirectional LSTM, Gated Recurrent Unit (GRU), CTRNN (Continuous Time RNN) CNN Architectures: VGG16, Alexnet, InceptionNet, RestNet,

Googlenet

Object Detection Models: R-CNN, Fast R-CNN, Faster R-CNN, cascade R-CNN. Mask RCNN, Single Shot MultiBox Detector (SSD), You Only Look Once (YOLO), Single-Shot Refinement Neural Network for Object Detection (RefineDet), Retina-Net

Autoencoders: Denoising Autoencoder, GAN

Transformers: Attention based Encoder and Decoder: Eg- BERT(Bidirectional Encoder Representations from Transformers), Generative Pretrained Transformers GPT-3, GPT-2, BERT, XLNet, and RoBERTa

WORLD STANDARD METHODOLOGY: CRISP-DM METHODOLOGY, SEMMA METHODOLOGY

Data Science Lifecycle

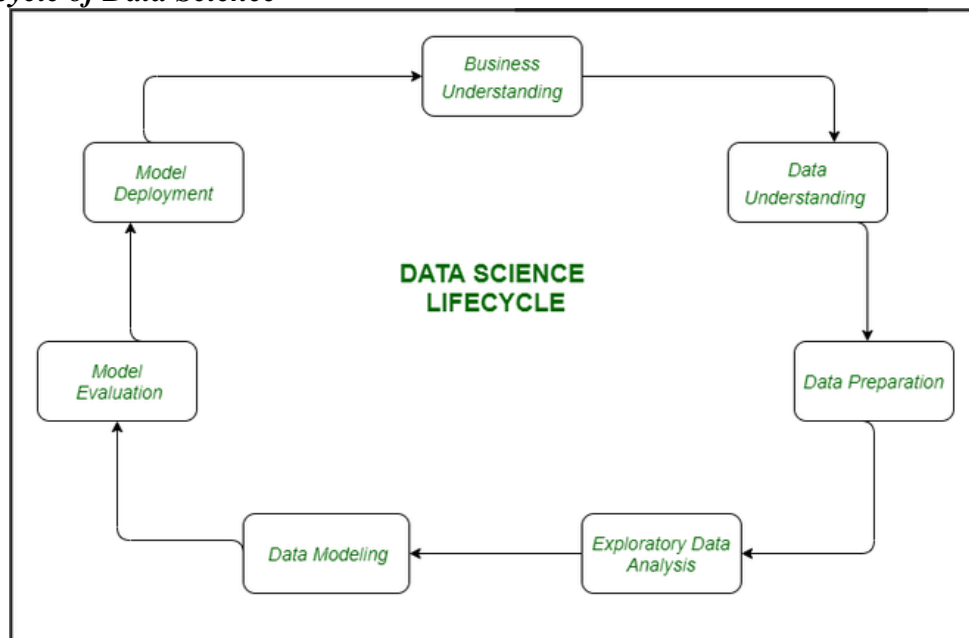
- ✚ Data Science Lifecycle revolves around the use of machine learning and different analytical strategies to produce insights and predictions from information in order to acquire a commercial enterprise objective.
- ✚ The complete method includes a number of steps like data cleaning, preparation, modelling, model evaluation, etc. It is a lengthy procedure and may additionally take quite a few months to complete.
- ✚ So, it is very essential to have a generic structure to observe for each and every hassle at hand. The globally mentioned structure in fixing any analytical problem is referred to as a Cross Industry Standard Process for Data Mining or CRISP-DM framework.

Let us understand what is the need for Data Science?

The following are some primary motives for the use of Data science technology:

1. *It helps to convert the big quantity of uncooked and unstructured records into significant insights.*
2. *It can assist in unique predictions such as a range of surveys, elections, etc.*
3. *It also helps in automating transportation such as growing a self-driving car, we can say which is the future of transportation.*
4. *Companies are shifting towards Data science and opting for this technology. Amazon, Netflix, etc, which cope with the big quantity of data, are the use of information science algorithms for higher consumer experience.*

The lifecycle of Data Science



- ✚ **Business Understanding:** The complete cycle revolves around the enterprise goal. What will you resolve if you do not longer have a specific problem? It is extraordinarily essential to apprehend the commercial enterprise goal sincerely due to the fact that will be your ultimate aim of the analysis.

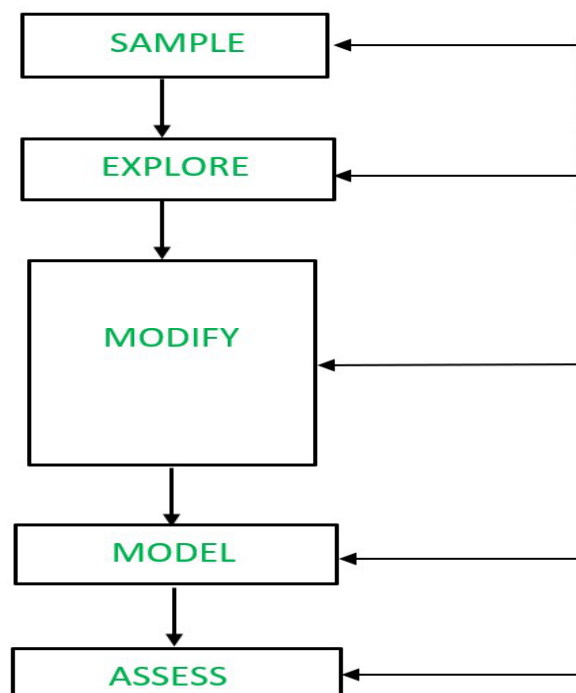
- ✚ *After desirable perception only we can set the precise aim of evaluation that is in sync with the enterprise objective.* You need to understand if the customer desires to minimize savings loss, or if they prefer to predict the rate of a commodity, etc.
- ✚ **2. Data Understanding:** After enterprise understanding, the subsequent step is data understanding. This includes a series of all the reachable data. Here you need to intently work with the commercial enterprise group as they are certainly conscious of what information is present, what facts should be used for this commercial enterprise problem, and different information.
- ✚ This step includes describing the data, their structure, their relevance, their records type. Explore the information using graphical plots. Basically, extracting any data that you can get about the information through simply exploring the data.
- ✚ **3. Preparation of Data:** Next comes the data preparation stage. *This consists of steps like choosing the applicable data, integrating the data by means of merging the data sets, cleaning it,* treating the lacking values through either eliminating them or imputing them, treating inaccurate data through eliminating them, additionally test for outliers the use of box plots and cope with them.
- ✚ Constructing new data, derive new elements from present ones. Format the data into the preferred structure, eliminate undesirable columns and features. *Data preparation is the most time-consuming but arguably the most essential step in the complete existence cycle. Your model will be as accurate as your data.*
- 5. **Exploratory Data Analysis:** This step includes getting some concept about the answer and elements affecting it, earlier than constructing the real model.
- ✚ *Distribution of data inside distinctive variables of a character is explored graphically the usage of bar-graphs, Relations between distinct aspects are captured via graphical representations like scatter plots and warmth maps.*
- ✚ Many data visualization strategies are considerably used to discover each and every characteristic individually and by means of combining them with different features.
- 6. **Data Modeling:** *Data modeling is the coronary heart of data analysis. A model takes the organized data as input and gives the preferred output. This step consists of selecting the suitable kind of model, whether the problem is a classification problem, or a regression problem or a clustering problem.*
- ✚ After deciding on the model family, amongst the number of algorithms amongst that family, we need to cautiously pick out the algorithms to put into effect and enforce them. We need to tune the hyperparameters of every model to obtain the preferred performance.
- ✚ We additionally need to make positive there is the right stability between overall performance and generalizability. We do no longer desire the model to study the data and operate poorly on new data.
- 7. **Model Evaluation:** Here the model is evaluated for checking if it is geared up to be deployed. The model is examined on an unseen data, evaluated on a cautiously thought out set of assessment metrics. We additionally need to make positive that the model conforms to reality.
 - ✚ If we do not acquire a quality end result in the evaluation, we have to re-iterate the complete modelling procedure until the preferred stage of metrics is achieved. Any data science solution, a machine learning model, simply like a

human, must evolve, must be capable to enhance itself with new data, adapt to a new evaluation metric.

- ✚ We can construct more than one model for a certain phenomenon, however, a lot of them may additionally be imperfect. The model assessment helps us select and construct an ideal model.
- ✚ **Model Deployment:** *The model after a rigorous assessment is at the end deployed in the preferred structure and channel. This is the last step in the data science life cycle. Each step in the data science life cycle defined above must be laboured upon carefully.*
- ✚ If any step is performed improperly, and hence, have an effect on the subsequent step and the complete effort goes to waste. For example, if data is no longer accumulated properly, you'll lose records and you will no longer be constructing an ideal model.
- ✚ If information is not cleaned properly, the model will no longer work. If the model is not evaluated properly, it will fail in the actual world. Right from Business perception to model deployment, every step has to be given appropriate attention, time, and effort.

SEMMA Model

- ✚ *SEMMA is the sequential methods to build machine learning models incorporated in 'SAS Enterprise Miner', a product by SAS Institute Inc., one of the largest producers of commercial statistical and business intelligence software. However, the sequential steps guide the development of a machine learning system. Let's look at the five sequential steps to understand it better.*



SEMMA model in Machine Learning

Sample: This step is all about selecting the subset of the right volume dataset from a large dataset provided for building the model. It will help us to build the model very efficiently. Basically in this step, we identify the independent variables(outcome) and dependent variables(factors). The selected subset of data should be actually a representation

of the entire dataset originally collected, which means it should contain sufficient information to retrieve. The data is also divided into training and validation purpose.

Explore: In this phase, activities are carried out to understand the data gaps and relationship with each other. *Two key activities are univariate and multivariate analysis. In univariate analysis, each variable looks individually to understand its distribution, whereas in multivariate analysis the relationship between each variable is explored. Data visualization is heavily used to help understand the data better.* In this step, we do analysis with all the factors which influence our outcome.

Modify: In this phase, *variables are cleaned where required.* New derived features are created by applying business logic to existing features based on the requirement. Variables are transformed if necessary. The outcome of this phase is a clean dataset that can be passed to the machine learning algorithm to build the model. In this step, we check whether the data is completely transformed or not. If we need the transformation of data we use the label encoder or label binarizer.

Model: *In this phase, various modelling or data mining techniques are applied to the pre-processed data to benchmark their performance against desired outcomes.* In this step, we perform all the mathematical which makes our outcome more precise and accurate as well.

Assess: *This is the last phase. Here model performance is evaluated against the test data (not used in model training) to ensure reliability and business usefulness.* Finally, in this step, we perform the evaluation and interpretation of data. We compare our model outcome with the actual outcome and analysis of our model limitation and also try to overcome that limitation.

**Real Life Work around Multi-Variate Analytics: A few Selected Commonly used
Techniques: Predictive & Classification Models, Regression, Clustering
Real Life Work around Artificial Intelligence, Machine Learning and Deep Learning:**

**A few Selected Commonly used
Techniques: Predictive & Classification Models, Regression, Clustering (TOPICS Can
be referred from unit 4)_**

**Real Life Work around Artificial Intelligence, Machine Learning and Deep Learning:
(Refer UNIT 4 COGNITIVE MODEL TOPIC PART D of UNIT 4)**

A few Selected Commonly used Techniques & Algorithms: ANN (Artificial Neural Network), CNN (Convolutional Neural Network), RNN (Recurrent Neural Network); RN Architecture: LSTM, Bidirectional LSTM, Gated Recurrent Unit (GRU), CTRNN (Continuous Time RNN) CNN Architectures: VGG16, Alexnet, InceptionNet, RestNet, Googlenet

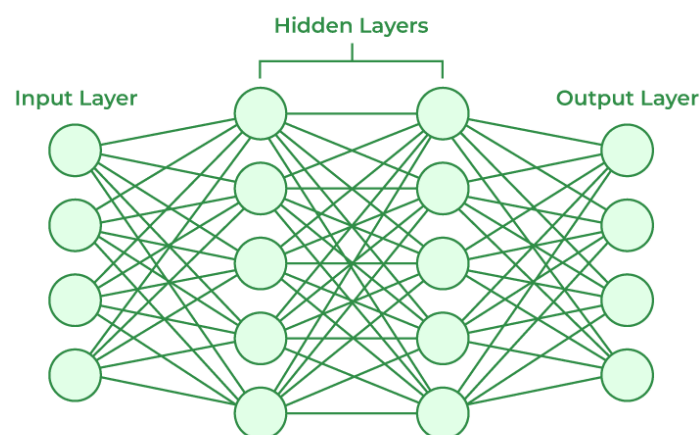
Artificial Neural Networks and its Applications

As you read this article, which organ in your body is thinking about it? It's the brain of course! But do you know how the brain works? Well, it has neurons or nerve cells that are the primary units of both the brain and the nervous system. These neurons receive sensory input from the outside world which they process and then provide the output which might act as the input to the next neuron.

Each of these neurons is connected to other neurons in complex arrangements at synapses. Now, are you wondering how this is related to Artificial Neural Networks? Well, Artificial Neural Networks are modeled after the neurons in the human brain. Let's check out what they are in detail and how they learn information.

Artificial Neural Networks

- ✚ *Artificial Neural Networks contain artificial neurons which are called units. These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers.*
- ✚ *The input layer receives data from the outside world which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is valuable for the output layer. Finally, the output layer provides an output in the form of a response of the Artificial Neural Networks to input data provided.*
- ✚ *In the majority of neural networks, units are interconnected from one layer to another. Each of these connections has weights that determine the influence of one unit on another unit. As the data transfers from one unit to another, the neural network learns more and more about the data which eventually results in an output from the output layer.*



Neural Networks Architecture

The structures and operations of human neurons serve as the basis for artificial neural networks. It is also known as neural networks or neural nets. ***The input layer of an artificial neural network is the first layer, and it receives input from external sources and releases it to the hidden layer, which is the second layer. In the hidden layer, each neuron receives input from the previous layer neurons, computes the weighted sum, and sends it to the neurons in the next layer.*** These connections are weighted means effects of the inputs from the previous layer are optimized more or less by assigning different-different weights to each input and it is adjusted during the training process by optimizing these weights for improved model performance.

Artificial neurons vs Biological neurons

The concept of artificial neural networks comes from biological neurons found in animal brains So they share a lot of similarities in structure and function wise.

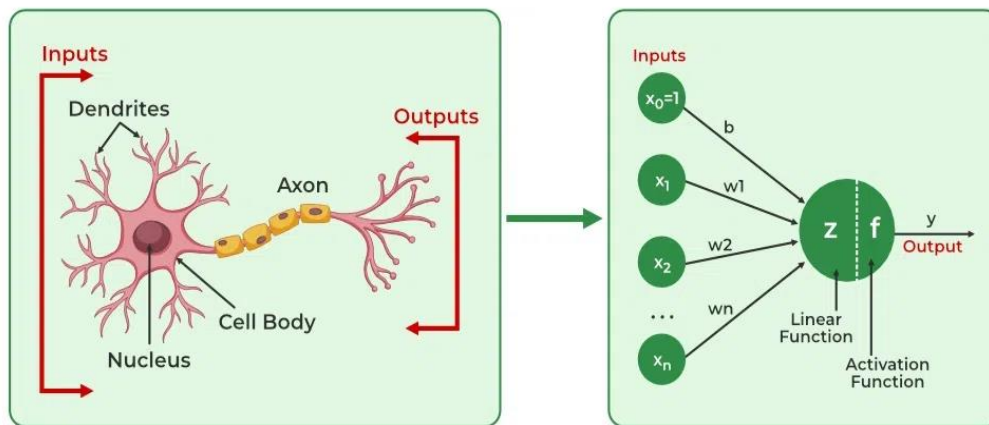
- **Structure:** *The structure of artificial neural networks is inspired by biological neurons. A biological neuron has a cell body or soma to process the impulses, dendrites to receive them, and an axon that transfers them to other neurons. The input nodes of artificial neural networks receive input signals, the hidden layer nodes compute these input signals, and the output layer nodes compute the final output by processing the hidden layer's results using activation functions.*

<i>Biological Neuron</i>	<i>Artificial Neuron</i>
<i>Dendrite</i>	<i>Inputs</i>
<i>Cell nucleus or Soma</i>	<i>Nodes</i>
<i>Synapses</i>	<i>Weights</i>
<i>Axon</i>	<i>Output</i>

- **Synapses:** *Synapses are the links between biological neurons that enable the transmission of impulses from dendrites to the cell body.* Synapses are the weights that join the one-layer nodes to the next-layer nodes in artificial neurons. The strength of the links is determined by the weight value.
- **Learning:** *In biological neurons, learning happens in the cell body nucleus or soma, which has a nucleus that helps to process the impulses.* An action potential is produced and travels through the axons if the impulses are powerful enough to reach the threshold. This becomes possible by synaptic plasticity, which represents the ability of synapses to become stronger or weaker over time in reaction to changes in their activity. In artificial neural networks, backpropagation is a technique used for learning, which adjusts the weights between nodes according to the error or differences between predicted and actual outcomes.

<i>Biological Neuron</i>	<i>Artificial Neuron</i>
<i>Synaptic plasticity</i>	<i>Backpropagations</i>

- **Activation:** *In biological neurons, activation is the firing rate of the neuron which happens when the impulses are strong enough to reach the threshold. In artificial neural networks, A mathematical function known as an activation function maps the input to the output, and executes activations.*



Biological neurons to Artificial neurons

How do Artificial Neural Networks learn?

Artificial neural networks are trained using a training set. For example, suppose you want to teach an ANN to recognize a cat. Then it is shown thousands of different images of cats so that the network can learn to identify a cat. Once the neural network has been trained enough using images of cats, then you need to check if it can identify cat images correctly. This is done by making the ANN classify the images it is provided by deciding whether they are cat images or not. The output obtained by the ANN is corroborated by a human-provided description of whether the image is a cat image or not. If the ANN identifies incorrectly then back-propagation is used to adjust whatever it has learned during training. Backpropagation is done by fine-tuning the weights of the connections in ANN units based on the error rate obtained. This process continues until the artificial neural network can correctly recognize a cat in an image with minimal possible error rates.

What are the types of Artificial Neural Networks?

- **Feedforward Neural Network:** *The feedforward neural network is one of the most basic artificial neural networks. In this ANN, the data or the input provided travels in a single direction.* It enters into the ANN through the input layer and exits through the output layer while hidden layers may or may not exist. So the feedforward neural network has a front-propagated wave only and usually does not have backpropagation.

- **Convolutional Neural Network:** *A Convolutional neural network has some similarities to the feed-forward neural network, where the connections between units have weights that determine the influence of one unit on another unit.* But a CNN has one or more than one convolutional layer that uses a convolution operation on the input and then passes the result obtained in the form of output to the next layer. CNN has applications in speech and image processing which is particularly useful in computer vision.
- **Modular Neural Network:** *A Modular Neural Network contains a collection of different neural networks that work independently towards obtaining the output with no interaction between them.* Each of the different neural networks performs a different sub-task by obtaining unique inputs compared to other networks. The advantage of this modular neural network is that it breaks down a large and complex computational process into smaller components, thus decreasing its complexity while still obtaining the required output.
- **Radial basis function Neural Network:** *Radial basis functions are those functions that consider the distance of a point concerning the center.* RBF functions have two layers. In the first layer, the input is mapped into all the Radial basis functions in the hidden layer and then the output layer computes the output in the next step. Radial basis function nets are normally used to model the data that represents any underlying trend or function.
- **Recurrent Neural Network:** *The Recurrent Neural Network saves the output of a layer and feeds this output back to the input to better predict the outcome of the layer. The first layer in the RNN is quite similar to the feed-forward neural network and the recurrent neural network starts once the output of the first layer is computed.* After this layer, each unit will remember some information from the previous step so that it can act as a memory cell in performing computations.

Applications of Artificial Neural Networks

1. **Social Media:** Artificial Neural Networks are used heavily in Social Media. For example, let's take the '**People you may know**' feature on Facebook that suggests people that you might know in real life so that you can send them friend requests. Well, this magical effect is achieved by using Artificial Neural Networks that analyze your profile, your interests, your current friends, and also their friends and various other factors to calculate the people you might potentially know. Another common application of Machine Learning in social media is **facial recognition**. This is done by finding around 100 reference points on the person's face and then matching them with those already available in the database using convolutional neural networks.
2. **Marketing and Sales:** When you log onto E-commerce sites like Amazon and Flipkart, they will recommend your products to buy based on your previous browsing history. Similarly, suppose you love Pasta, then Zomato, Swiggy, etc. will show you restaurant recommendations based on your tastes and previous order history. This is true across all new-age marketing segments like Book sites, Movie services, Hospitality sites, etc. and it is done by implementing **personalized marketing**. This uses Artificial Neural Networks to identify the customer likes, dislikes, previous shopping history, etc., and then tailor the marketing campaigns accordingly.

3. **Healthcare:** Artificial Neural Networks are used in Oncology to train algorithms that can identify cancerous tissue at the microscopic level at the same accuracy as trained physicians. Various rare diseases may manifest in physical characteristics and can be identified in their premature stages by using **Facial Analysis** on the patient photos. So the full-scale implementation of Artificial Neural Networks in the healthcare environment can only enhance the diagnostic abilities of medical experts and ultimately lead to the overall improvement in the quality of medical care all over the world.
4. **Personal Assistants:** I am sure you all have heard of Siri, Alexa, Cortana, etc., and also heard them based on the phones you have!!! These are personal assistants and an example of speech recognition that uses **Natural Language Processing** to interact with the users and formulate a response accordingly. Natural Language Processing uses artificial neural networks that are made to handle many tasks of these personal assistants such as managing the language syntax, semantics, correct speech, the conversation that is going on, etc.

CONVOLUTIONAL NEURAL NETWORK

- A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

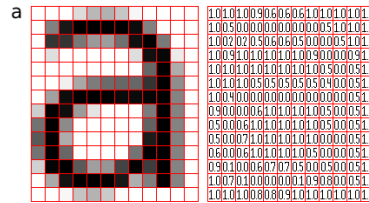


Figure 1: Representation of image as a grid of pixels (Source)

- The human brain processes a huge amount of information the second we see an image. *Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field.* Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well.
- The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. By using a CNN, one can enable sight to computers.

Convolutional Neural Network Architecture

A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.

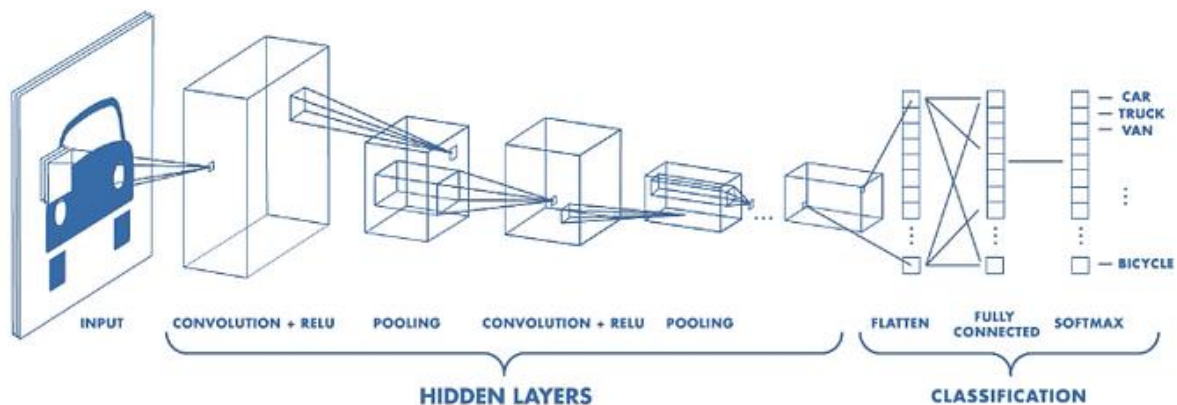


Figure 2: Architecture of a CNN (Source)

Convolution Layer

- The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load.
- This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. *The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels,*

the kernel height and width will be spatially small, but the depth extends up to all three channels.

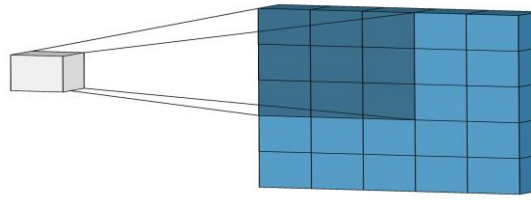


Illustration of Convolution Operation

During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. *The sliding size of the kernel is called a stride.*

If we have an input of size $W \times W \times D$ and D_{out} number of kernels with a spatial size of F with stride S and amount of padding P , then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Formula for Convolution Layer

This will yield an output volume of size $W_{out} \times W_{out} \times D_{out}$.

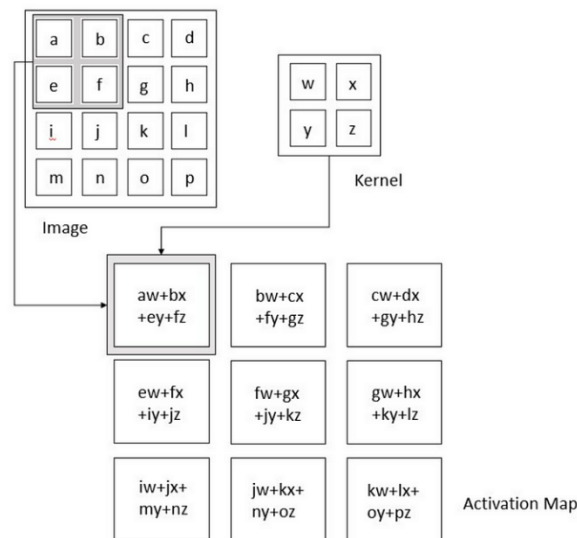


Figure 3: Convolution Operation (Source: Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville)

Pooling Layer

The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

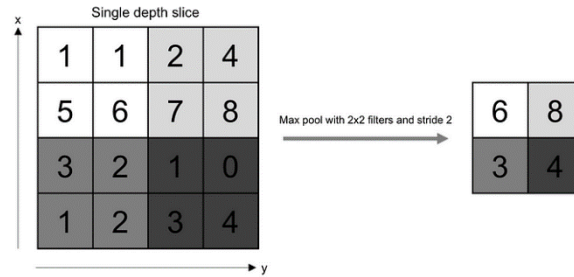


Figure 4: Pooling Operation (Source: O'Reilly Media)

If we have an activation map of size $W \times W \times D$, a pooling kernel of spatial size F , and stride S , then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F}{S} + 1$$

Formula for Padding Layer

This will yield an output volume of size $W_{out} \times W_{out} \times D$.

In all cases, pooling provides some translation invariance which means that an object would be recognizable regardless of where it appears on the frame.

Fully Connected Layer

Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN. This is why it can be computed as usual by a matrix multiplication followed by a bias effect.

The FC layer helps to map the representation between the input and the output.

Non-Linearity Layers

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map.

There are several types of non-linear operations, the popular ones being:

1. Sigmoid

The sigmoid non-linearity has the mathematical form $\sigma(\kappa) = 1/(1+e^{-\kappa})$. It takes a real-valued number and “squashes” it into a range between 0 and 1.

However, a very undesirable property of sigmoid is that when the activation is at either tail, the gradient becomes almost zero. If the local gradient becomes very small, then in backpropagation it will effectively “kill” the gradient. Also, if the data coming into the neuron is always positive, then the output of sigmoid will be either all positives or all negatives, resulting in a zig-zag dynamic of gradient updates for weight.

2. Tanh

Tanh squashes a real-valued number to the range $[-1, 1]$. *Like sigmoid, the activation saturates, but — unlike the sigmoid neurons — its output is zero centered.*

3. ReLU

The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function $f(\kappa) = \max(0, \kappa)$. In other words, the activation is simply threshold at zero.

In comparison to sigmoid and tanh, ReLU is more reliable and accelerates the convergence by six times.

Unfortunately, a con is that ReLU can be fragile during training. A large gradient flowing through it can update it in such a way that the neuron will never get further updated. However, we can work with this by setting a proper learning rate.

Designing a Convolutional Neural Network

Our convolutional neural network has architecture as follows:

[INPUT]

→ [CONV 1] → [BATCH NORM] → [ReLU] → [POOL 1]

→ [CONV 2] → [BATCH NORM] → [ReLU] → [POOL 2]

→ [FC LAYER] → [RESULT]

For both conv layers, we will use kernel of spatial size 5 x 5 with stride size 1 and padding of 2. For both pooling layers, we will use max pool operation with kernel size 2, stride 2, and zero padding.

CONV 1
Input Size ($W_1 \times H_1 \times D_1$) = $28 \times 28 \times 1$
<ul style="list-style-type: none"> Requires four hyperparameter: <ul style="list-style-type: none"> Number of kernels, $k = 16$ Spatial extend of each one, $F = 5$ Stride Size, $S = 1$ Amount of zero padding, $P = 2$
<ul style="list-style-type: none"> Outputting volume of $W_2 \times H_2 \times D_2$ <ul style="list-style-type: none"> $W_2 = (28 - 5 + 2(2)) / 1 + 1 = 28$ $H_2 = (28 - 5 + 2(2)) / 1 + 1 = 28$ $D_2 = k$
Output of Conv 1 ($W_2 \times H_2 \times D_2$) = $28 \times 28 \times 16$

Calculations for Conv 1 Layer

POOL 1
Input Size ($W_2 \times H_2 \times D_2$) = $28 \times 28 \times 16$
<ul style="list-style-type: none"> Requires two hyperparameter: <ul style="list-style-type: none"> Spatial extend of each one, $F = 2$ Stride Size, $S = 2$
<ul style="list-style-type: none"> Outputting volume of $W_3 \times H_3 \times D_3$ <ul style="list-style-type: none"> $W_3 = (28 - 2) / 2 + 1 = 14$ $H_3 = (28 - 2) / 2 + 1 = 14$
Output of Pool 1 ($W_3 \times H_3 \times D_2$) = $14 \times 14 \times 16$

Calculations for Pool1 Layer

CONV 2
Input Size ($W_3 \times H_3 \times D_2$) = $14 \times 14 \times 16$
<ul style="list-style-type: none"> Requires four hyperparameter: <ul style="list-style-type: none"> Number of kernels, $k = 32$ Spatial extend of each one, $F = 5$ Stride Size, $S = 1$ Amount of zero padding, $P = 2$
<ul style="list-style-type: none"> Outputting volume of $W_4 \times H_4 \times D_3$ <ul style="list-style-type: none"> $W_4 = (14 - 5 + 2(2)) / 1 + 1 = 14$ $H_4 = (14 - 5 + 2(2)) / 1 + 1 = 14$ $D_3 = k$
Output of Conv 2 ($W_4 \times H_4 \times D_3$) = $14 \times 14 \times 32$

Calculations for Conv 2 Layer

POOL 2
Input Size ($W_4 \times H_4 \times D_3$) = $14 \times 14 \times 32$
<ul style="list-style-type: none"> • Requires two hyperparameter: <ul style="list-style-type: none"> ◦ Spatial extend of each one, $F = 2$ ◦ Stride Size, $S = 2$
<ul style="list-style-type: none"> • Outputting volume of $W_5 \times H_5 \times D_3$ <ul style="list-style-type: none"> ◦ $W_5 = (14 - 2) / 2 + 1 = 7$ ◦ $H_5 = (14 - 2) / 2 + 1 = 7$
Output of Pool 2 ($W_5 \times H_5 \times D_3$) = $7 \times 7 \times 32$

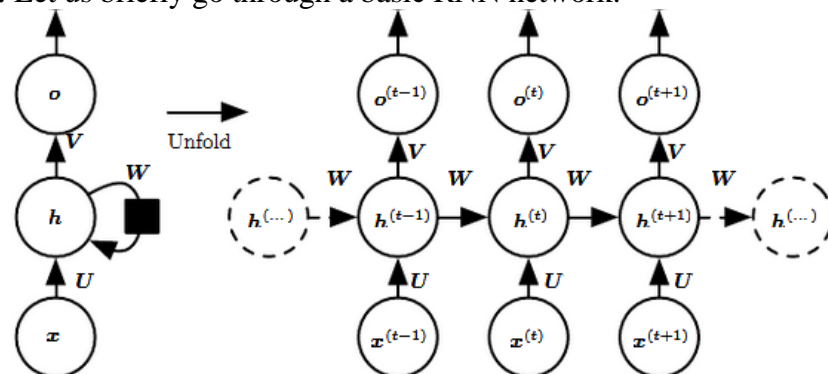
Calculations for Pool2 Layer

FC Layer
Input Size ($W_5 \times H_5 \times D_3$) = $7 \times 7 \times 32$
Output Size (Number of Classes) = 10

RNN

A recurrent neural network is a neural network that is specialized for processing a sequence of data $x(t) = x(1), \dots, x(\tau)$ with the time step index t ranging from 1 to τ . For tasks that involve sequential inputs, such as speech and language, it is often better to use RNNs. In a NLP problem, if you want to predict the next word in a sentence it is important to know the words before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a “memory” which captures information about what has been calculated so far.

Architecture : Let us briefly go through a basic RNN network.

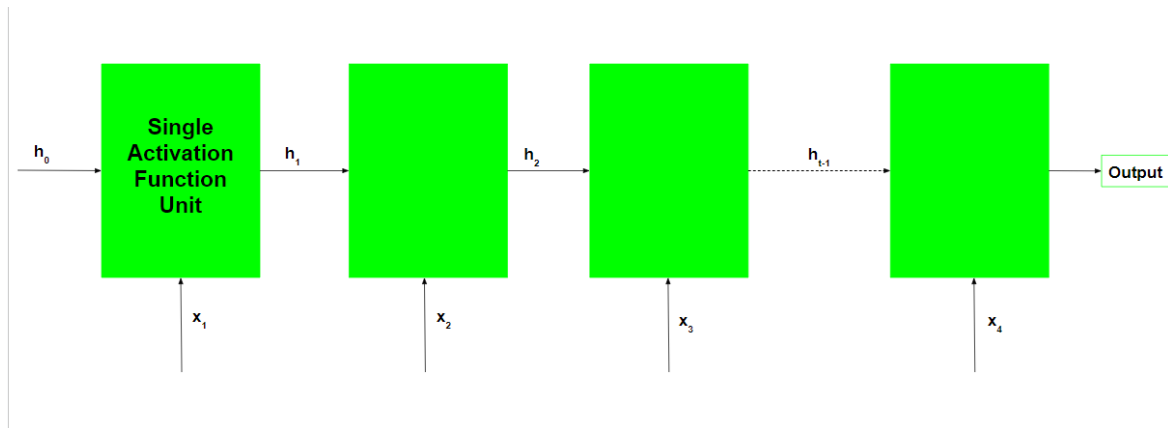


Reference

Recurrent Neural Networks Explanation

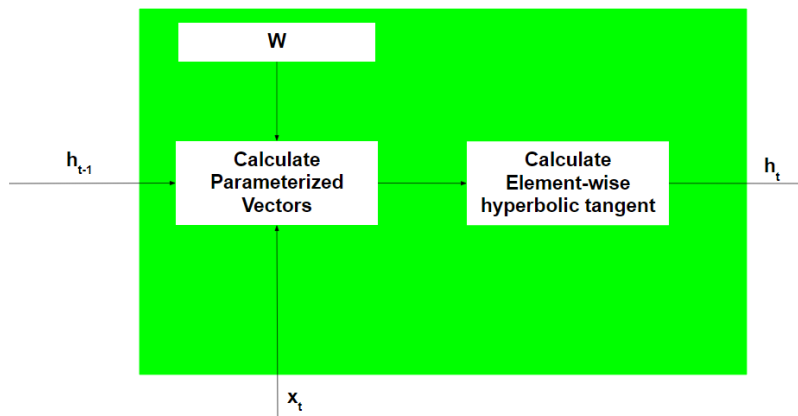
- Today, different Machine Learning techniques are used to handle different types of data. One of the most difficult types of data to handle and the forecast is sequential data. Sequential data is different from other types of data in the sense that while all the features of a typical dataset can be assumed to be order-independent, this cannot be assumed for a sequential dataset.
- To handle such type of data, the concept of **Recurrent Neural Networks** was conceived. It is different from other Artificial Neural Networks in its structure. While other networks “travel” in a linear direction during the feed-forward process or the back-propagation process, the Recurrent Network follows a recurrence relation instead of a feed-forward pass and uses **Back-Propagation through time** to learn.
- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.
- This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation:-

The basic work-flow of a Recurrent Neural Network is as follows:-

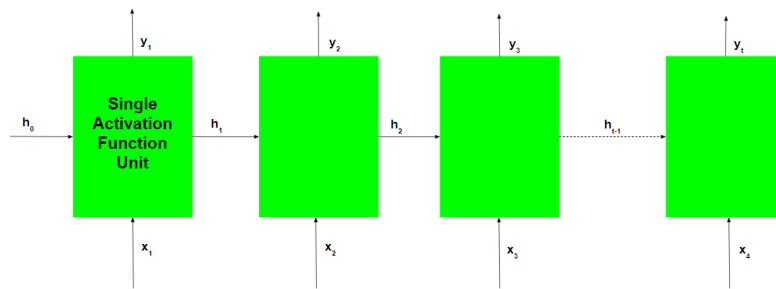


Working of each Recurrent Unit:

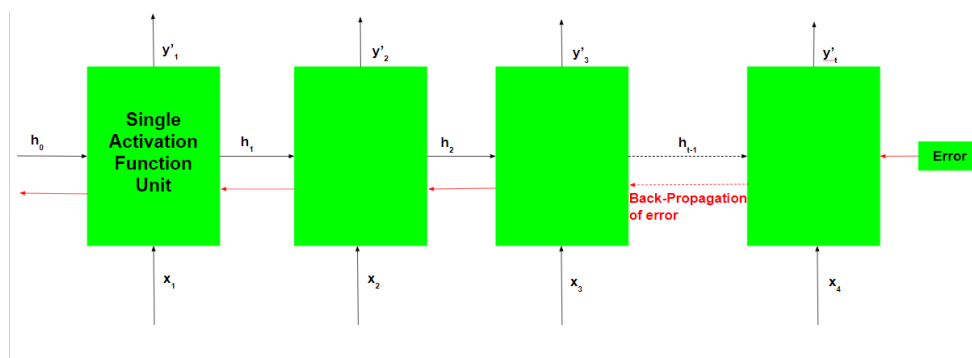
1. Take input the previously hidden state vector and the current input vector.
Note that since the hidden state and current input are treated as vectors, each element in the vector is placed in a different dimension which is orthogonal to the other dimensions. Thus each element when multiplied by another element only gives a non-zero value when the elements involved are non-zero and the elements are in the same dimension.
2. *Element-wise multiplies the hidden state vector by the hidden state weights and similarly performs the element-wise multiplication of the current input vector and the current input weights. This generates the parameterized hidden state vector and the current input vector.*
Note that weights for different vectors are stored in the trainable weight matrix.
3. Perform the vector addition of the two parameterized vectors and then calculate the element-wise hyperbolic tangent to generate the new hidden state vector.



During the training of the recurrent network, the network also generates an output at each time step. This output is used to train the network using gradient descent.



The Back-Propagation involved is similar to the one used in a typical Artificial Neural Network with some minor changes. These changes are noted as:-



- ✚ The problem of Exploding Gradients may be solved by using a hack – By putting a threshold on the gradients being passed back in time. But this solution is not seen as a solution to the problem and may also reduce the efficiency of the network. To deal with such problems, two main variants of Recurrent Neural Networks were developed – **Long Short Term Memory**

Networks and Gated Recurrent Unit Networks.

- ✚ Recurrent Neural Networks (RNNs) are a type of artificial neural network that is designed to process sequential data. Unlike traditional feedforward neural networks, RNNs can take into account the previous state of the sequence while processing the current state, allowing them to model temporal dependencies in data.
- ✚ The key feature of RNNs is the presence of recurrent connections between the hidden units, which allow information to be passed from one time step to the next. This means that the hidden state at each time step is not only a function of the input at that time step, but also a function of the previous hidden state.
- ✚ *In an RNN, the input at each time step is typically a vector representing the current state of the sequence, and the output at each time step is a vector representing the predicted value or classification at that time step. The hidden state is also a vector, which is updated at each time step based on the current input and the previous hidden state.*
- ✚ *The basic RNN architecture suffers from the vanishing gradient problem, which can make it difficult to train on long sequences. To address this issue, several variants of RNNs have been developed, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, which use specialized gates to control the flow of information through the network and address the vanishing gradient problem.*

Applications of RNNs include speech recognition, language modeling, machine translation, sentiment analysis, and stock prediction, among others. Overall, RNNs are a powerful tool for processing sequential data and modeling temporal dependencies, making them an important component of many machine learning applications.

The advantages of Recurrent Neural Networks (RNNs) are:

1. *Ability to Process Sequential Data: RNNs can process sequential data of varying lengths, making them useful in applications such as natural language processing, speech recognition, and time-series analysis.*
2. *Memory: RNNs have the ability to retain information about the previous inputs in the sequence through the use of hidden states. This enables RNNs to perform tasks such as predicting the next word in a sentence or forecasting stock prices.*
3. *Versatility: RNNs can be used for a wide variety of tasks, including classification, regression, and sequence-to-sequence mapping.*
4. *Flexibility: RNNs can be combined with other neural network architectures, such as Convolutional Neural Networks (CNNs) or feedforward neural networks, to create hybrid models for specific tasks.*

However, there are also some disadvantages of RNNs:

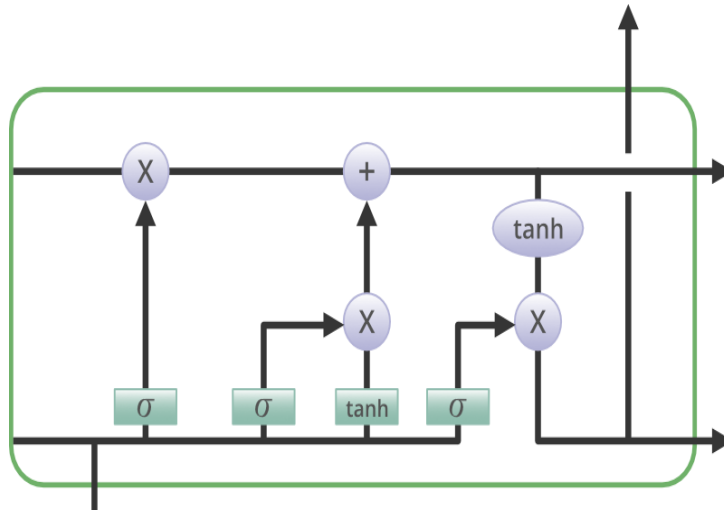
1. **Vanishing Gradient Problem:** The vanishing gradient problem can occur in RNNs, particularly in those with many layers or long sequences, making it difficult to learn long-term dependencies.
2. **Computationally Expensive:** RNNs can be computationally expensive, particularly when processing long sequences or using complex architectures.
3. **Lack of Interpretability:** RNNs can be difficult to interpret, particularly in terms of understanding how the network is making predictions or decisions.
4. Overall, while RNNs have some disadvantages, their ability to process sequential data and retain memory of previous inputs make them a powerful tool for many machine learning applications.

UNDERSTANDING OF LSTM NETWORKS

- ✚ Long Short-Term Memory is an advanced version of recurrent neural network (RNN) architecture that was designed to model chronological sequences and their long-range dependencies more precisely than conventional RNNs.
- ✚ *LSTM networks are an extension of recurrent neural networks (RNNs) mainly introduced to handle situations where RNNs fail.*
- ✚ *It fails to store information for a longer period of time. At times, a reference to certain information stored quite a long time ago is required to predict the current output. But RNNs are absolutely incapable of handling such “long-term dependencies”.*
- ✚ There is no finer control over which part of the context needs to be carried forward and how much of the past needs to be ‘forgotten’.
- ✚ Other issues with RNNs are exploding and vanishing gradients (explained later) which occur during the training process of a network through backtracking.
- ✚ Thus, Long Short-Term Memory (LSTM) was brought into the picture. It has been so designed that the vanishing gradient problem is almost completely removed, while the training model is left unaltered.
- ✚ Long-time lags in certain problems are bridged using LSTMs which also handle noise, distributed representations, and continuous values. With LSTMs, there is no need to keep a finite number of states from beforehand as required in the hidden Markov model (HMM). LSTMs provide us with a large range of parameters such as learning rates, and input and output biases.

Structure of LSTM

- ✚ *The basic difference between the architectures of RNNs and LSTMs is that the hidden layer of LSTM is a gated unit or gated cell. It consists of four layers that interact with one another in a way to produce the output of that cell along with the cell state. These two things are then passed onto the next hidden layer. Unlike RNNs which have got only a single neural net layer of tanh, LSTMs comprise three logistic sigmoid gates and one tanh layer. Gates have been introduced in order to limit the information that is passed through the cell. They determine which part of the information will be needed by the next cell and which part is to be discarded. The output is usually in the range of 0-1 where ‘0’ means ‘reject all’ and ‘1’ means ‘include all’.*

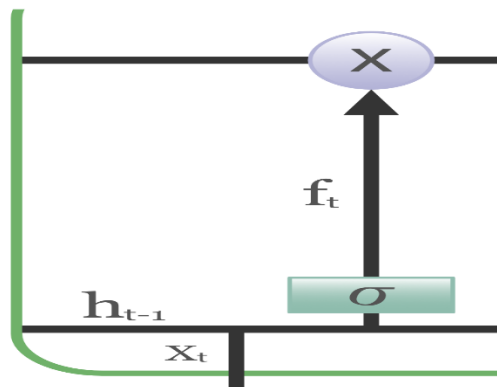


Structure of an LSTM Network

Information is retained by the cells and the memory manipulations are done by the gates. There are three gates which are explained below:

Forget Gate

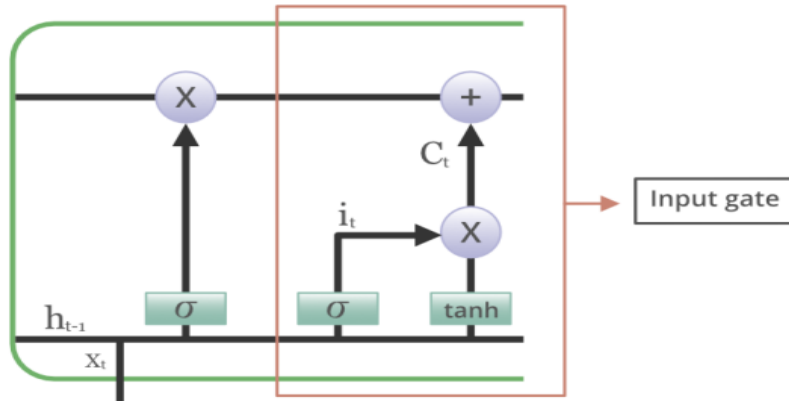
The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x_t (input at the particular time) and h_{t-1} (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state, the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.



Forget Gate in LSTM Cell

Input gate

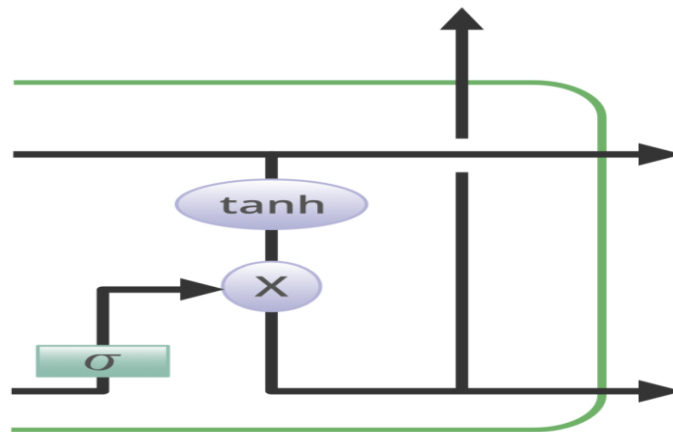
The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs h_{t-1} and x_t . Then, a vector is created using the tanh function that gives an output from -1 to +1, which contains all the possible values from h_{t-1} and x_t . At last, the values of the vector and the regulated values are multiplied to obtain useful information.



Input gate in the LSTM cell

Output gate

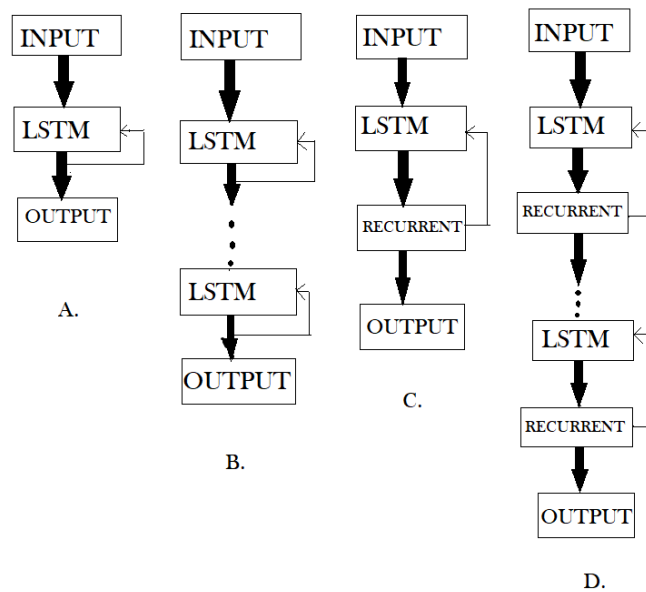
The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying the tanh function on the cell. Then, the information is regulated using the sigmoid function and filtered by the values to be remembered using inputs h_{t-1} and x_t . At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell.



Output gate in the LSTM cell

Variations in LSTM Networks

With the increasing popularity of LSTMs, various alterations have been tried on the conventional ***LSTM architecture to simplify the internal design of cells to make them work in a more efficient way and to reduce computational complexity.*** Gers and Schmidhuber introduced peephole connections which allowed gate layers to have knowledge about the cell state at every instant. Some LSTMs also made use of a coupled input and forget gate instead of two separate gates which helped in making both decisions simultaneously. Another variation was the use of the Gated Recurrent Unit (GRU) which improved the design complexity by reducing the number of gates. It uses a combination of the cell state and hidden state and also an update gate which has forgotten and input gates merged into it.



LSTM(Figure-A), DLSTM(Figure-B), LSTMP(Figure-C) and DLSTMP(Figure-D)

1. *Figure-A represents what a basic LSTM network looks like. Only one layer of LSTM between an input and output layer has been shown here.*
2. *Figure B represents Deep LSTM which includes a number of LSTM layers in between the input and output. The advantage is that the input values fed to the network not only go through several LSTM layers but also propagate through time within one LSTM cell. Hence, parameters are well distributed within multiple layers. This results in a thorough process of inputs in each time step.*
3. *Figure C represents LSTM with the Recurrent Projection layer where the recurrent connections are taken from the projection layer to the LSTM layer input. This architecture was designed to reduce the high learning computational complexity ($O(N)$ for each time step) of the standard LSTM RNN.*
4. *Figure D represents Deep LSTM with a Recurrent Projection Layer consisting of multiple LSTM layers where each layer has its own projection layer. The increased depth is quite useful in the case where the memory size is too large. Having increased depth prevents overfitting in models as the inputs to the network need to go through many nonlinear functions.*

Applications of LSTM Networks

LSTM models need to be trained with a training dataset prior to their employment in real-world applications. Some of the most demanding applications are discussed below:

1. **Language modeling or text generation**, involves the computation of words when a sequence of words is fed as input. Language models can be operated at the character level, n-gram level, sentence level, or even paragraph level.
2. **Image processing involves performing an analysis of a picture and concluding its result into a sentence**. For this, it's required to have a dataset comprising a good amount of pictures with their corresponding descriptive captions. A model that has already been trained is used to predict features of images present in the dataset. This is photo data. The dataset is then processed in such a way that only the words that are most suggestive are

present in it. This is text data. Using these two types of data, we try to fit the model. The work of the model is to generate a descriptive sentence for the picture one word at a time by taking input words that were predicted previously by the model and also the image.

3. ***Speech and Handwriting Recognition.***
4. ***Music generation is quite similar to that of text generation where LSTMs predict musical notes instead of text by analyzing a combination of given notes fed as input.***
5. ***Language Translation involves mapping a sequence in one language to a sequence in another language.*** Similar to image processing, a dataset, containing phrases and their translations, is first cleaned and only a part of it is used to train the model. An encoder-decoder LSTM model is used which first converts the input sequence to its vector representation (encoding) and then outputs it to its translated version.

Drawbacks of Using LSTM Networks

As it is said, everything in this world comes with its own advantages and disadvantages, LSTMs too, have a few drawbacks which are discussed below:

1. LSTMs became popular because they could solve the problem of vanishing gradients. But it turns out, they fail to remove it completely. The problem lies in the fact that the data still has to move from cell to cell for its evaluation. Moreover, the cell has become quite complex now with additional features (such as forget gates) being brought into the picture.
2. They require a lot of resources and time to get trained and become ready for real-world applications. In technical terms, they need high memory bandwidth because of the linear layers present in each cell which the system usually fails to provide. Thus, hardware-wise, LSTMs become quite inefficient.
3. With the rise of data mining, developers are looking for a model that can remember past information for a longer time than LSTMs. The source of inspiration for such kind of model is the human habit of dividing a given piece of information into small parts for easy remembrance.
4. LSTMs get affected by different random weight initialization and hence behave quite similarly to that of a feed-forward neural net. They prefer small-weight initialization instead.
5. LSTMs are prone to overfitting and it is difficult to apply the dropout algorithm to curb this issue. Dropout is a regularization method where input and recurrent connections to LSTM units are probabilistically excluded from activation and weight updates while training a network.

GATED RECURRENT UNIT NETWORKS

Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) that was introduced by Cho et al. in 2014 as a simpler alternative to Long Short-Term Memory (LSTM) networks. Like LSTM, GRU can process sequential data such as text, speech, and time-series data.

The basic idea behind GRU is to use gating mechanisms to selectively update the hidden state of the network at each time step. **The gating mechanisms are used to control the flow of information in and out of the network.** The GRU has two gating mechanisms, called the reset gate and the update gate.

The reset gate determines how much of the previous hidden state should be forgotten, while the update gate determines how much of the new input should be used to update the hidden state. The output of the GRU is calculated based on the updated hidden state.

The equations used to calculate the reset gate, update gate, and hidden state of a GRU are as follows:

Reset gate: $r_t = \text{sigmoid}(W_r * [h_{t-1}, x_t])$

Update gate: $z_t = \text{sigmoid}(W_z * [h_{t-1}, x_t])$

Candidate hidden state: $h_t' = \tanh(W_h * [r_t * h_{t-1}, x_t])$

Hidden state: $h_t = (1 - z_t) * h_{t-1} + z_t * h_t'$

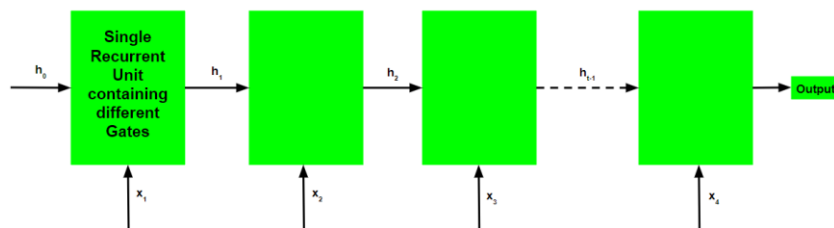
where W_r , W_z , and W_h are learnable weight matrices, x_t is the input at time step t , h_{t-1} is the previous hidden state, and h_t is the current hidden state.

In summary, GRU networks are a type of RNN that use gating mechanisms to selectively update the hidden state at each time step, allowing them to effectively model sequential data. They have been shown to be effective in various natural language processing tasks, such as language modeling, machine translation, and speech recognition

Prerequisites: Recurrent Neural Networks, Long Short Term Memory Networks

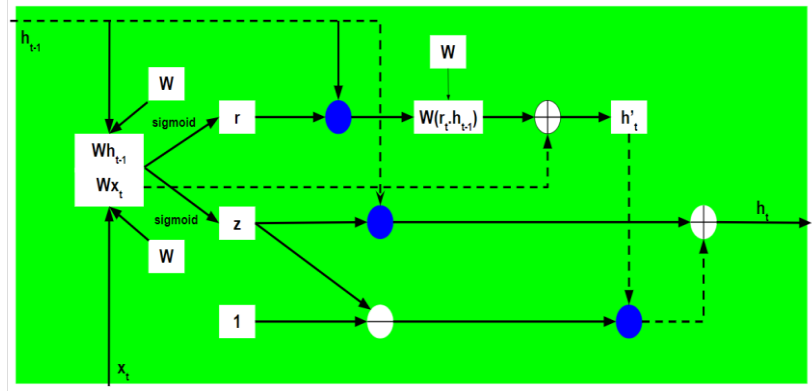
To solve the Vanishing-Exploding gradients problem often encountered during the operation of a basic Recurrent Neural Network, many variations were developed. One of the most famous variations is the **Long Short Term Memory Network(LSTM)**. One of the lesser-known but equally effective variations is the **Gated Recurrent Unit Network(GRU)**.

The basic work-flow of a Gated Recurrent Unit Network is similar to that of a basic Recurrent Neural Network when illustrated, the main difference between the two is in the internal working within each recurrent unit as Gated Recurrent Unit networks consist of gates which modulate the current input and the previous hidden state.



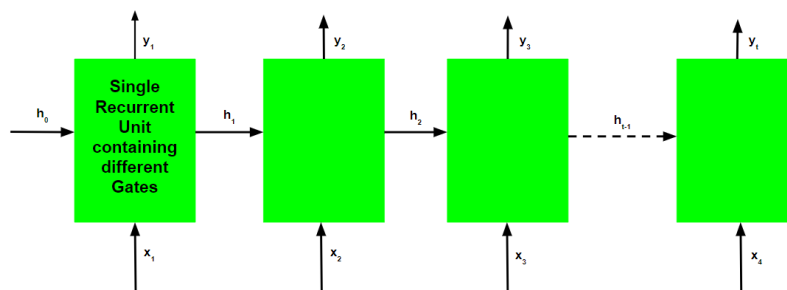
Working of a Gated Recurrent Unit:

- Take input the current input and the previous hidden state as vectors.
- Calculate the values of the three different gates by following the steps given below:-
 1. *For each gate, calculate the parameterized current input and previously hidden state vectors by performing element-wise multiplication (Hadamard Product) between the concerned vector and the respective weights for each gate.*
 2. *Apply the respective activation function for each gate element-wise on the parameterized vectors. Below given is the list of the gates with the activation function to be applied for the gate.*



Note that the blue circles denote element-wise multiplication. The positive sign in the circle denotes vector addition while the negative sign denotes vector subtraction(vector addition with negative value). The weight matrix W contains different weights for the current input vector and the previous hidden state for each gate.

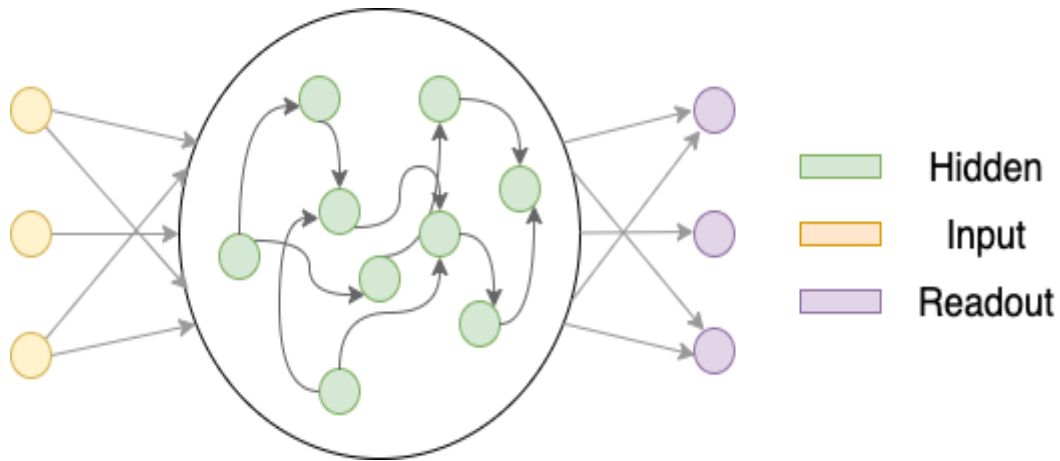
Just like Recurrent Neural Networks, a GRU network also generates an output at each time step and this output is used to train the network using gradient descent.



Note that just like the workflow, the training process for a GRU network is also diagrammatically similar to that of a basic Recurrent Neural Network and differs only in the internal working of each recurrent unit.

The Back-Propagation Through Time Algorithm for a Gated Recurrent Unit Network is similar to that of a Long Short Term Memory Network and differs only in the differential chain formation.

Developing and Understanding Continuous-Time RNNs (CTRNNs) for Neuroscience Applications



neural network (ANN) to simulate brains?

Despite the emergence of numerous state-of-the-art technologies like fMRI, Neuropixels, and fluorescent imaging that allow us to some extent see what's going on in the in-vivo brain, there are still considerable limitations. For instance, there is a trade-off between temporal resolution and spatial resolution when choosing these techniques. And none of the technology can fully capture how all neurons in the area of interest spike across time.

However, it is readily possible to examine an ANN to the finest detail, from how each artificial neuron spikes at a given time to how each synapse grows across trials. Therefore, it might not be such a bad idea to trade off some biological realism for a much finer examination by using ANN to simulate biological neural networks (BNN). Additionally, despite in-silico simulations requiring many assumptions on the BNN, if we actually were able to find similar patterns between both networks, these assumptions may help us narrow down the most relevant factors that drive a certain network dynamic and exclude other irrelevant factors.

What would be a reasonable model for BNN?

Biological brains can be considered as neural networks that continuously receive external stimuli and perform synaptic adaptations. At any given time, the neuron could be either spiking or not spiking. Modeling BNN involves two critical parts: modeling a single neuron and modeling the entire network.

- ✚ To model a single neuron, we primarily want to know a neuron's firing rate and action potential at any given time so that we will know what kind of signal it will send to its downstream neurons.
- ✚ To this end, a two stages process is typically considered: the neuron first linearly integrates its inputs (from multiple synapses) as well as its current action potential,

then applies a non-linear transformation to this sum to generate an output (the firing rate). This comes from the notion when we are dealing with actual experimental data, where we fit a linear model followed by a non-linear model to predict the firing rate of a neuron. Lastly, as we characterize the neuron states as firing rates, we will consider signals propagated in synapses and neuron states to be analog values instead of binary spike trains. Then, once we can model a single neuron, we can put everything together into a network by assembling neurons with synapses. This is no doubt a simplification of what's really happening in the brain, but it is already powerful enough to capture a vast amount of brain dynamics.

To be more specific about what it meant by ‘assembling neurons with synapses’, a single neuron linearly integrates signals from its presynaptic neurons and non-linearly produces some outputs to its postsynaptic neurons. As actual neurons generate and receive spikes, this process happens asynchronously in the actual brain. However, in our modeling, we can treat spike trains as discretized firing rates. And therefore, at any given time t , a neuron's firing rate will only be determined by its current action potential v and the firing rates of its presynaptic neurons at time $t-1$, as shown in the equation below.

$$fr_j^t = f(\alpha \sum_{i=0}^N w_{ij} fr_i^{t-1} + (1 - \alpha)v_j^{t-1})$$

✚ In this equation, to calculate the firing rate of neuron i at time t , we first consider the firing rates of all of neuron i 's presynaptic neurons at time $t-1$. Each of these firing rates is multiplied by the corresponding synaptic strength, w_{ij} , indicating the influence of each presynaptic neuron on neuron i .

✚ We also take into account the action potential of neuron i at time $t-1$. The parameter α is used to balance the contribution of the neuron's own action potential versus the inputs from its presynaptic neurons. In essence, α determines the rate at which our neuron ‘forgets’ its previous state. We then apply an activation function, $f(\cdot)$, to the sum of these inputs, capturing the non-linearity inherent in real neuronal responses. If we extend this equation to encompass the entire network, it can be written as follows:

$$\vec{fr}^t = f(\alpha W^T \vec{fr}^{t-1} + (1 - \alpha)v_j^{t-1})$$

Where \vec{fr} is the firing rate vector of all the neurons in the network, and W is the connectivity matrix that describes the synaptic strengths between all neurons.

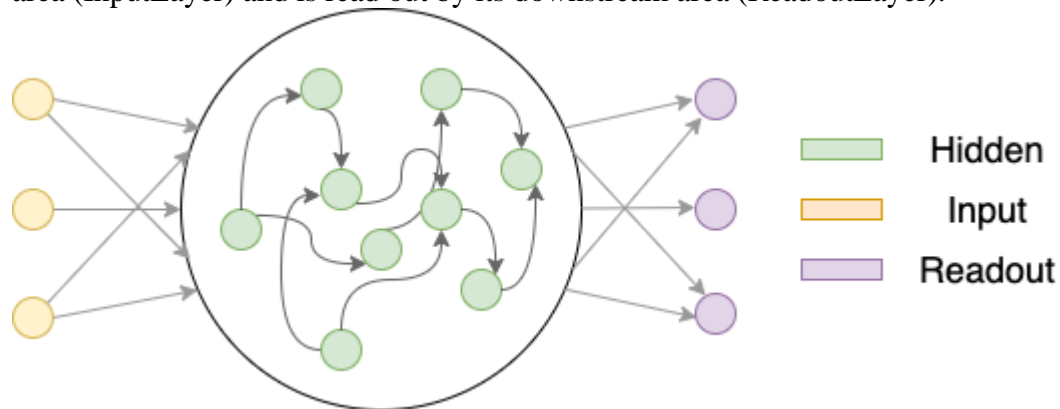
Convert it to a computational model (neural network)

✚ The *human central nervous system (CNS)* is typically estimated to have 80–100 billion neurons. With little doubt, we don't want to model all of them at the same time. We typically want to model only a small fraction of neurons in a certain area due to the sheer complexity and size of neuronal networks. If this small fraction of

neurons can accurately represent the behavior of the larger network, then it's a useful simplification. With this being said, to model this small fraction of the network, we consider this area of interest will receive some input from another area and output signals to its downstream area.

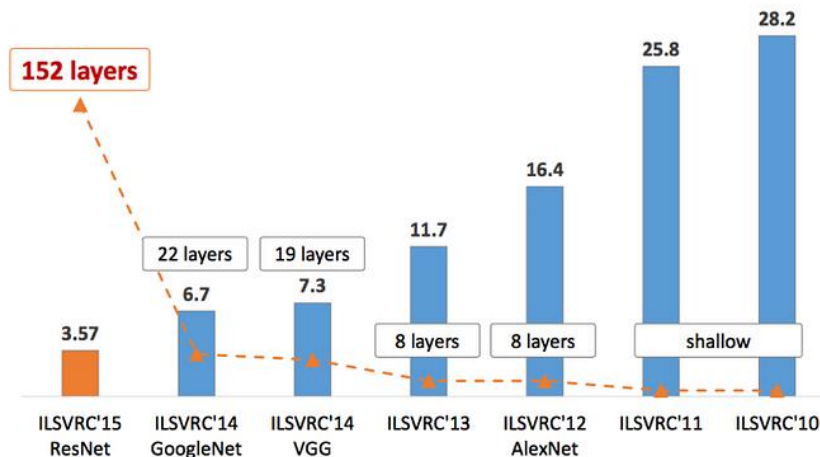
How to implement the RNN

Now, once we have the model and the task established, we can implement it in Python. As described previously, the area of concern (HiddenLayer) receives input for its upper stream area (InputLayer) and is read out by its downstream area (ReadoutLayer).



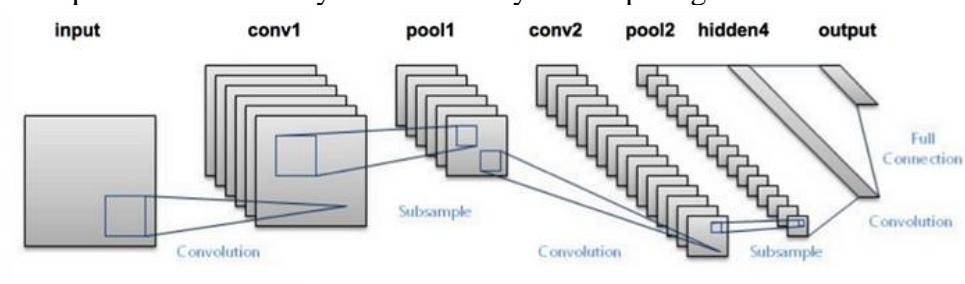
CNN ARCHITECTURES: LENET, ALEXNET, VGG, GOOGLNET, RESNET

A Convolutional Neural Network (CNN, or ConvNet) are a special kind of multi-layer neural networks, designed to recognize visual patterns directly from pixel images with minimal preprocessing.. The ImageNet project is a large visual database designed for use in visual object recognition software research. The ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes. Here I will talk about CNN architectures of ILSVRC top competitors .



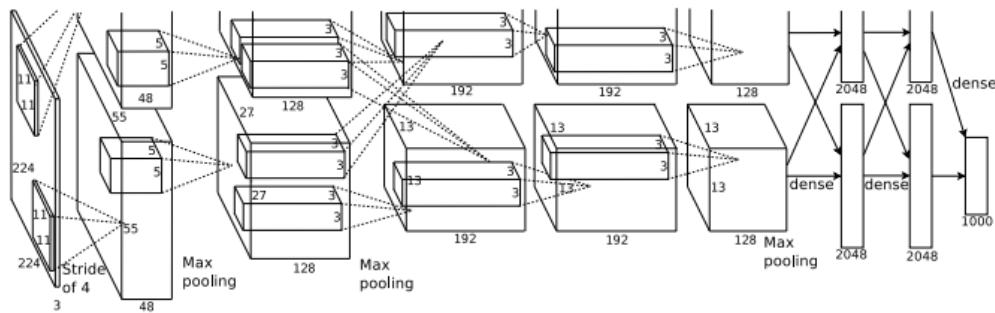
LeNet-5 (1998)

LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel greyscale input-images. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources.



AlexNet (2012)

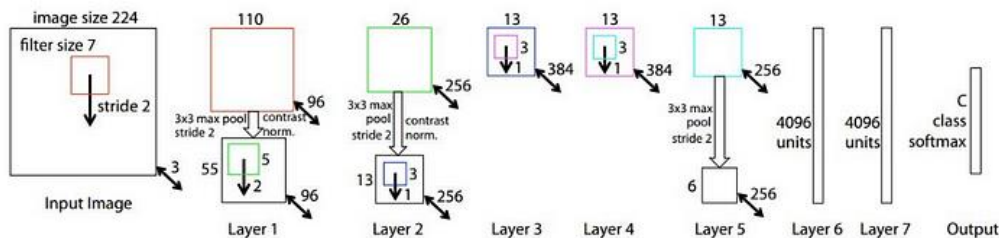
In 2012, AlexNet significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error from 26% to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%.



The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5, 3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer. AlexNet was trained for 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines. AlexNet was designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

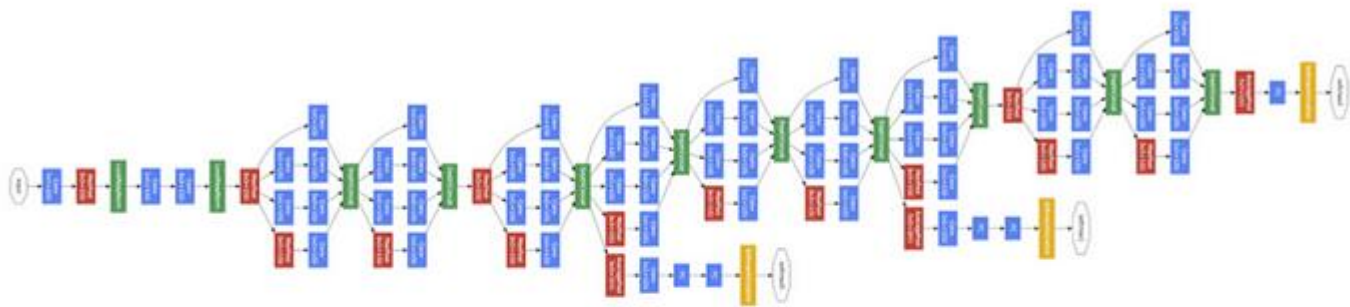
ZFNet(2013)

Not surprisingly, the ILSVRC 2013 winner was also a CNN which became known as ZFNet. It achieved a top-5 error rate of 14.8% which is now already half of the prior mentioned non-neural error rate. *It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional Deep Learning elements* as discussed earlier in this essay.



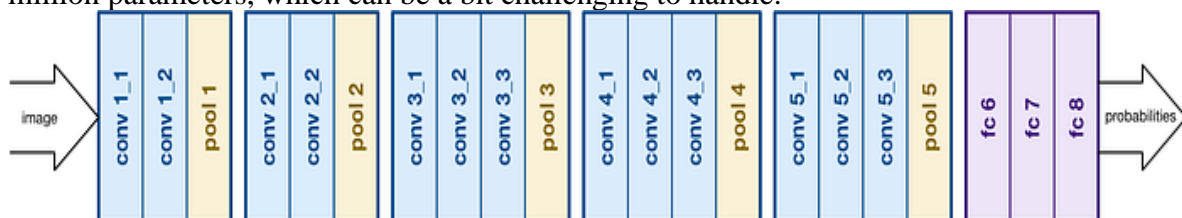
GoogLeNet/Inception(2014)

The winner of the ILSVRC 2014 competition was GoogLeNet(a.k.a. Inception V1) from Google. It achieved a top-5 error rate of 6.67%! This was very close to human level performance which the organisers of the challenge were now forced to evaluate. As it turns out, this was actually rather hard to do and required some human training in order to beat GoogLeNets accuracy. After a few days of training, the human expert (Andrej Karpathy) was able to achieve a top-5 error rate of 5.1%(single model) and 3.6%(ensemble). *The network used a CNN inspired by LeNet but implemented a novel element which is dubbed an inception module. It used batch normalization, image distortions and RMSprop. This module is based on several very small convolutions in order to drastically reduce the number of parameters. Their architecture consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million.*



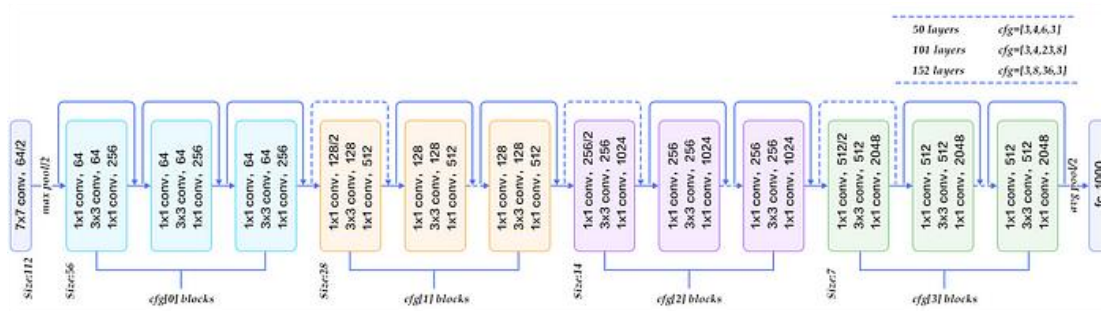
VGGNet (2014)

The runner-up at the ILSVRC 2014 competition is dubbed VGGNet by the community and was developed by Simonyan and Zisserman. VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture. Similar to AlexNet, only 3x3 convolutions, but lots of filters. Trained on 4 GPUs for 2–3 weeks. It is currently the most preferred choice in the community for extracting features from images. The weight configuration of the VGGNet is publicly available and has been used in many other applications and challenges as a baseline feature extractor. However, VGGNet consists of 138 million parameters, which can be a bit challenging to handle.

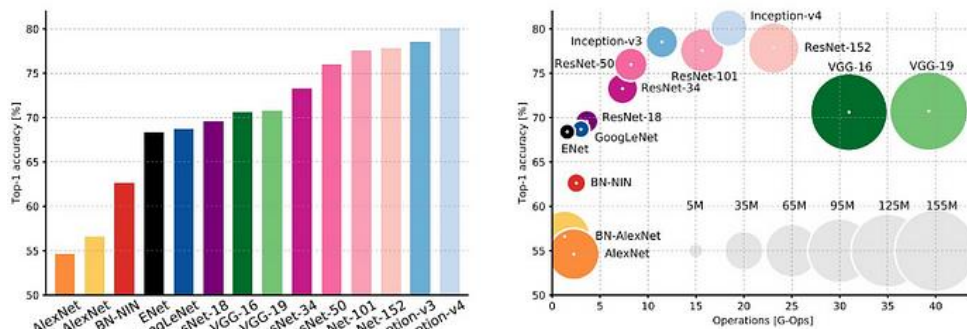


ResNet(2015)

At last, at the ILSVRC 2015, the so-called Residual Neural Network (ResNet) by Kaiming He et al introduced anovel architecture with “skip connections” and features heavy batch normalization. Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNNs. Thanks to this technique they were able to train a NN with 152 layers while still having lower complexity than VGGNet. It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset.



AlexNet has parallel two CNN line trained on two GPUs with cross-connections, GoogleNet has inception modules ,ResNet has residual connections.



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Summary Table

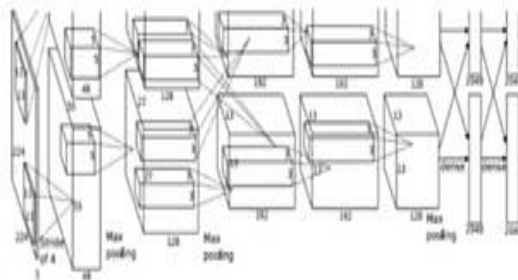
Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

OBJECT DETECTION MODELS: R-CNN, FAST R-CNN, FASTER R-CNN, CASCADE R-CNN, MASK RCNN, SINGLE SHOT MULTIBOX DETECTOR (SSD), YOU ONLY LOOK ONCE (YOLO), SINGLE-SHOT REFINEMENT NEURAL NETWORK FOR OBJECT DETECTION (REFINEDET), RETINA-NET

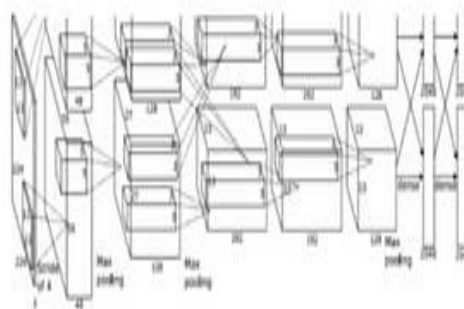
R-CNN, FAST R-CNN, FASTER R-CNN, YOLO — OBJECT DETECTION ALGORITHMS

UNDERSTANDING OBJECT DETECTION ALGORITHMS

Computer vision is an interdisciplinary field that has been gaining huge amounts of traction in the recent years(since CNN) and self-driving cars have taken centre stage. Another integral part of computer vision is object detection. Object detection aids in pose estimation, vehicle detection, surveillance etc. The difference between object detection algorithms and classification algorithms is that in detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image. Also, you might not necessarily draw just one bounding box in an object detection case, there could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.



CAT: (x, y, w, h)



DUCK: (x, y, w, h)

DUCK: (x, y, w, h)

....

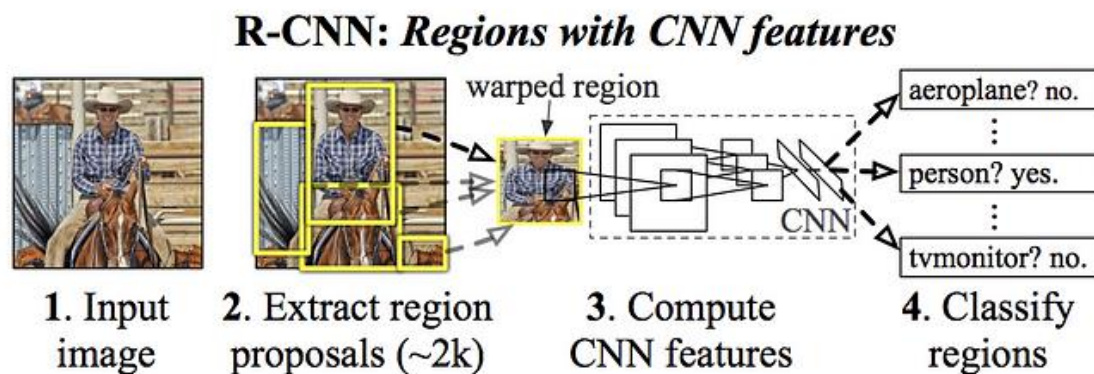
The major reason why you cannot proceed with this problem by building a standard convolutional network followed by a fully connected layer is that, the length of the output layer is variable — not constant, this is because the number of occurrences of the objects of interest is not fixed. A naive approach to solve this problem would be to take different regions of interest from the image, and use a CNN to classify the presence of the object within that region. The problem with this approach is that the objects of interest might have different

spatial locations within the image and different aspect ratios. Hence, you would have to select a huge number of regions and this could computationally blow up. Therefore, algorithms like R-CNN, YOLO etc have been developed to find these occurrences and find them fast.

R-CNN

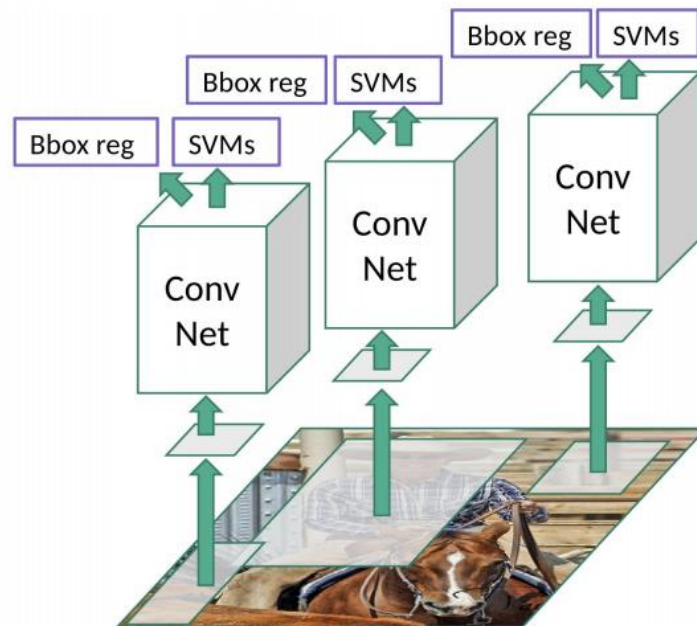
To bypass the problem of selecting a huge number of regions, [Ross Girshick et al.](#) proposed a method where we use selective search to extract just 2000 regions from the image and he called them region proposals. Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions. These 2000 region proposals are generated using the selective search algorithm which is written below.

Selective	Search:
1. Generate initial sub-segmentation, we generate many candidate	regions
2. Use greedy algorithm to recursively combine similar regions into larger ones	
3. Use the generated regions to produce the final candidate region proposals	



R-CNN

To know more about the selective search algorithm, follow this [link](#). These 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output. *The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an SVM to classify the presence of the object within that candidate region proposal. In addition to predicting the presence of an object within the region proposals, the algorithm also predicts four values which are offset values to increase the precision of the bounding box.* For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half. Therefore, the offset values help in adjusting the bounding box of the region proposal.

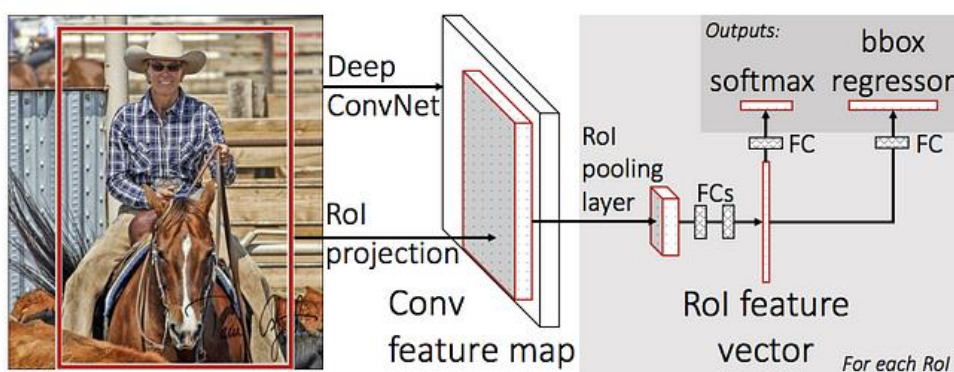


R-CNN

Problems with R-CNN

- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

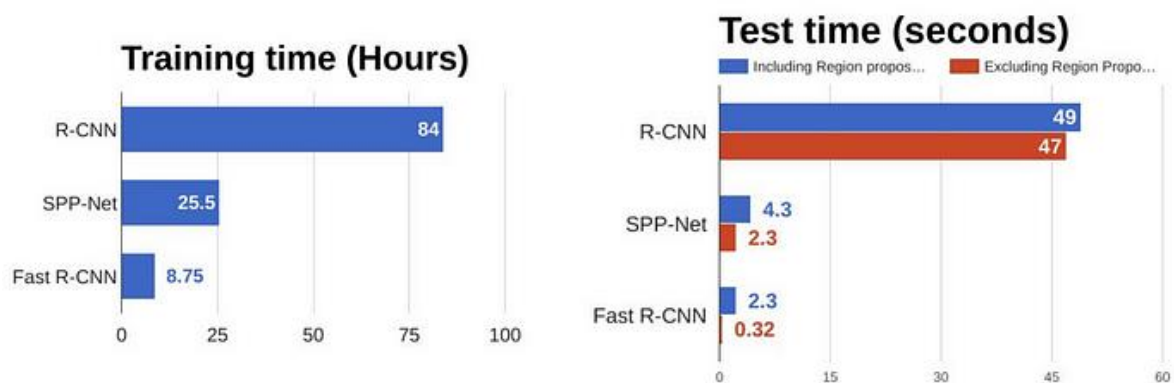
FAST R-CNN



Fast R-CNN

The same author of the previous paper(R-CNN) solved some of the drawbacks of R-CNN to build a faster object detection algorithm and it was called Fast R-CNN. The approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. *From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a RoI pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.*

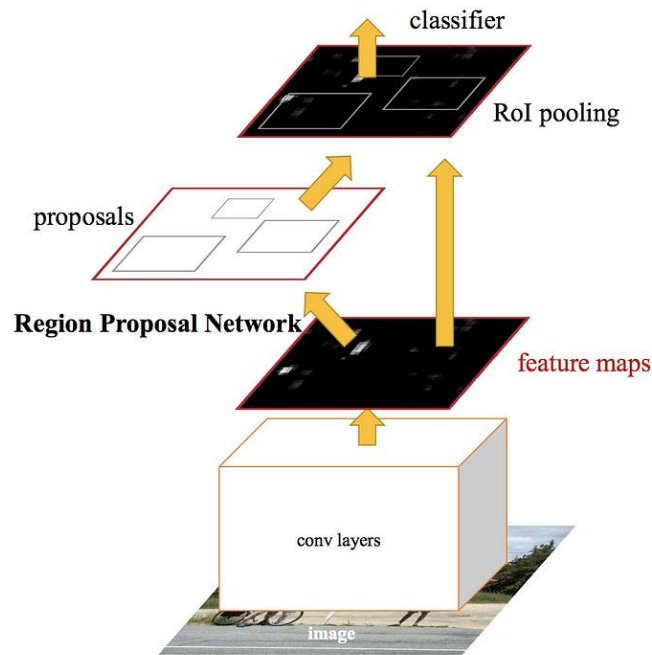
The reason “Fast R-CNN” is faster than R-CNN is because you don’t have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.



COMPARISON OF OBJECT DETECTION ALGORITHMS

From the above graphs, you can infer that Fast R-CNN is significantly faster in training and testing sessions over R-CNN. When you look at the performance of Fast R-CNN during testing time, including region proposals slows down the algorithm significantly when compared to not using region proposals. Therefore, region proposals become bottlenecks in Fast R-CNN algorithm affecting its performance.

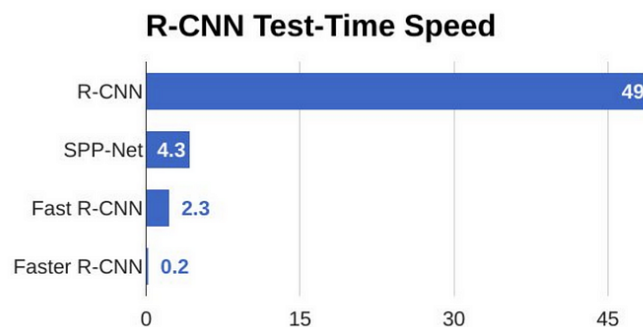
FASTER R-CNN



FASTER R-CNN

Both of the above algorithms(R-CNN & Fast R-CNN) uses selective search to find out the region proposals. Selective search is a slow and time-consuming process affecting the performance of the network. Therefore, Shaoqing Ren et al. came up with an object detection algorithm that eliminates the selective search algorithm and lets the network learn the region proposals.

Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map. Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

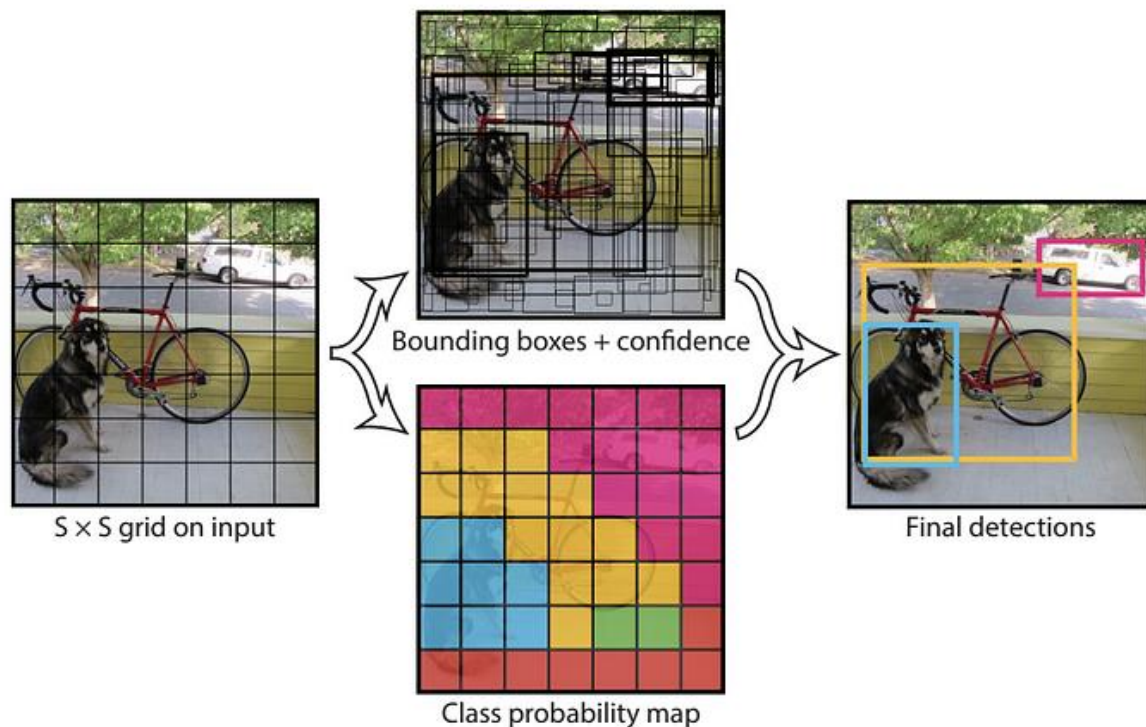


COMPARISON OF TEST-TIME SPEED OF OBJECT DETECTION ALGORITHMS

From the above graph, you can see that Faster R-CNN is much faster than its predecessors. Therefore, it can even be used for real-time object detection.

YOLO — You Only Look Once

All of the previous object detection algorithms use regions to localize the object within the image. The network does not look at the complete image. Instead, parts of the image which have high probabilities of containing the object. YOLO or You Only Look Once is an object detection algorithm much different from the region based algorithms seen above. In YOLO a single convolutional network predicts the bounding boxes and the class probabilities for these boxes.



YOLO

How YOLO works is that we take an image and split it into an $S \times S$ grid, within each of the grid we take m bounding boxes. For each of the bounding box, the network outputs a class probability and offset values for the bounding box. The bounding boxes having the class probability above a threshold value is selected and used to locate the object within the image.

YOLO is orders of magnitude faster(45 frames per second) than other object detection algorithms. The limitation of YOLO algorithm is that it struggles with small objects within the image, for example it might have difficulties in detecting a flock of birds. This is due to the spatial constraints of the algorithm.

REVIEW: RETINANET — FOCAL LOSS (OBJECT DETECTION)

One-Stage Detector, With Focal Loss and RetinaNet Using ResNet+FPN, Surpass the Accuracy of Two-Stage Detectors, Faster R-CNN

Review — RefineDet: Single-Shot Refinement Neural Network for Object Detection (Object Detection)

Outperforms CoupleNet, DCN, RetinaNet, G-RMI, FPN, TDM, Faster R-CNN, Fast R-CNN, OHEM, YOLOv2, YOLOv1, SSD, DSSD

In this story, **Single-Shot Refinement Neural Network for Object Detection**, (RefineDet), by Chinese Academy of Sciences, University of Chinese Academy of Sciences, and GE Global Research, is reviewed. In this paper:

- RefineDet is proposed, which consists of two inter-connected modules, namely, the **anchor refinement module (ARM)** and the **object detection module (ODM)**.
- **ARM filters out negative anchors** to reduce search space for the classifier, and **coarsely adjust the locations and sizes of anchors**.
- **ODM** takes the refined anchors as the input from ARM to further **improve the regression** and **predict multi-class label**.

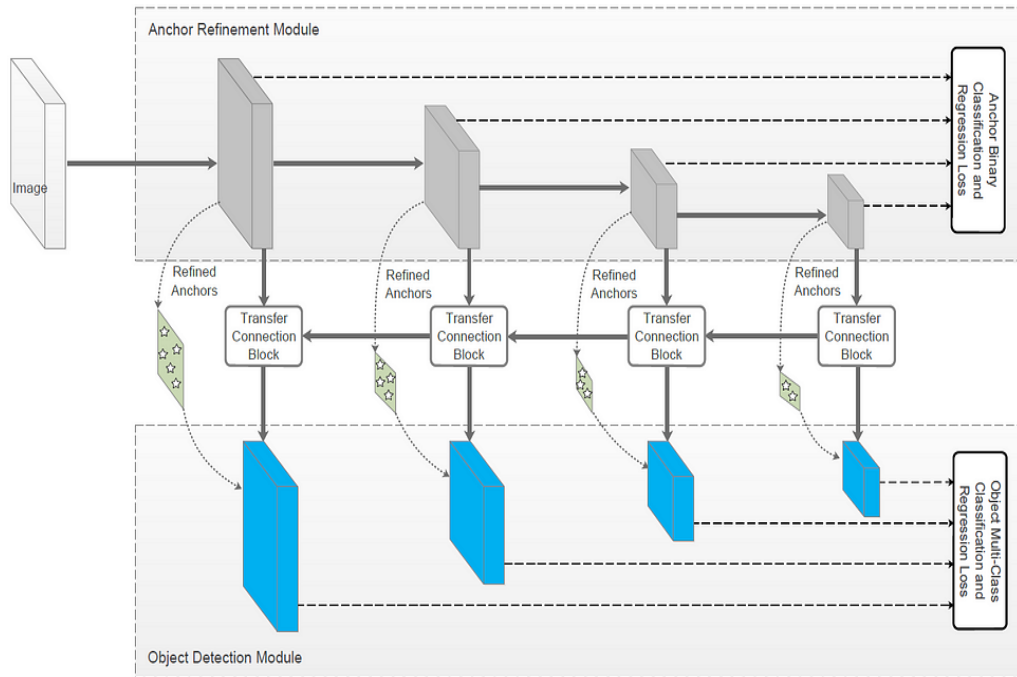
This is a paper in **2018 CVPR** with over **670 citations**. (
Sik-Ho Tsang

@ Medium)

Outline

1. **RefineDet: Network Architecture**
2. **Anchor Refinement Module (ARM)**
3. **Object detection module (ODM)**
4. **Transfer connection block (TCB)**
5. **Loss Function & Inference**
6. **Experimental Results**

1. RefineDet: Network Architecture



RefineDet: Network Architecture

- Similar to SSD, RefineDet **produces a fixed number of bounding boxes and the scores** indicating the presence of different classes of objects in those boxes, **followed by the non-maximum suppression (NMS)** to produce the final result.
- RefineDet is formed by two inter-connected modules, i.e., **anchor refinement module (ARM)** and the **object detection module (ODM)**.
- ILSVRC CLS-LOC pretrained VGG-16 and ResNet-101 are used as backbone.
- (There are some small modifications/refinement at the end of the backbones, please feel free to read the paper.)

1.1. Two-Step Cascaded Regression

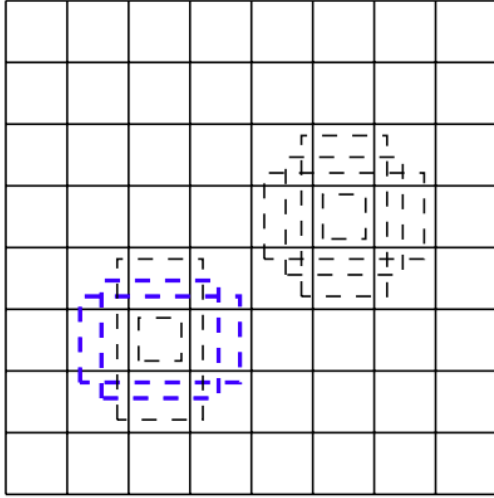
- As mentioned, **ARM filters out negative anchors** to reduce search space for the classifier, and **coarsely adjust the locations and sizes of anchors**.
- **ODM** takes the refined anchors as the input from ARM to further **improve the regression and predict multi-class label**.

1.2. Anchors Design and Matching

- **4 feature layers with the total stride sizes 8, 16, 32, and 64 pixels** are used to handle different scales of objects.
- Each feature layer is associated with one specific scale of anchors and 3 aspect ratios.

- (This paper is highly related to SSD, please read SSD if interested.)

2. Anchor Refinement Module (ARM)



Pre-defined Anchor Boxes in SSD

- There are pre-defined anchor boxes in SSD with fixed locations, ratios and sizes. (Please feel free to read SSD if interested.)

*ARM aims to **remove negative anchors** so as to **reduce search space** for the classifier and also **coarsely adjust the locations and sizes of anchors** to provide better initialization for the subsequent regressor.*

- Specifically, **n anchor boxes** are associated with each regularly divided cell on the feature map.
- At each feature map cell, **four offsets of the refined anchor boxes** are predicted.
- **Two confidence scores** are used to indicate the presence of foreground objects in those boxes.

2.1. Negative Anchor Filtering

- **If its negative confidence is larger than a preset threshold (i.e. 0.99 empirically), the anchor box is discarded in training the ODM.** It is quite sure that it is a background.
- Thus, only the refined hard negative anchor boxes and refined positive anchor boxes are passed to train the ODM.

3. Object detection module (ODM)

- After obtaining the refined anchor boxes, the refined anchor boxes are passed to the corresponding feature maps in the ODM.

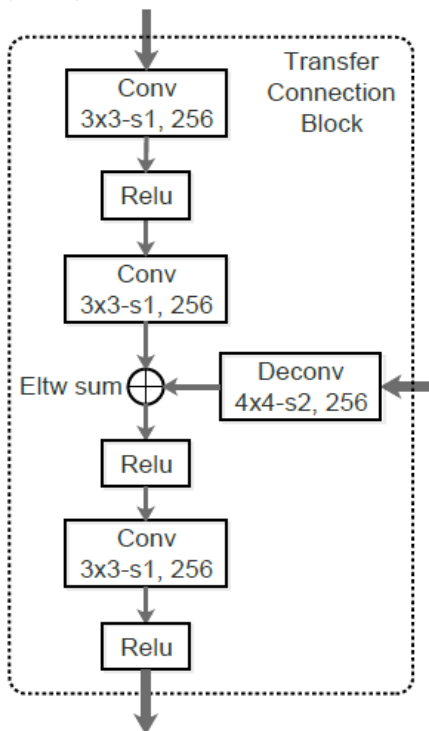
*ODM aims to **regress accurate object locations and predict multi-class labels** based on the refined anchors.*

- Specifically, **c class scores** and the **4 accurate offsets** of objects relative to the refined anchor boxes are calculated, yielding **$c + 4$ outputs for each refined anchor boxes** to complete the detection task.

3.1. Hard Negative Mining

- Hard negative mining is used to mitigate the extreme foreground-background class imbalance.
- Some negative anchor boxes with top loss values are selected to make the ratio between the negatives and positives below 3 : 1**, instead of using all negative anchors or randomly selecting the negative anchors in training.

4. Transfer connection block (TCB)



Transfer connection block (TCB)

- TCB converts the features from the ARM to the ODM for detection.
- TCBs is to **integrate large-scale context by adding the high-level features** to the transferred features to improve detection accuracy.
- To match the dimensions between them, **the deconvolution operation is used to enlarge the high-level feature maps and sum them in the element-wise way.**

5. Loss Function & Inference

5.1. Loss Function

- Therefore, the loss function for RefineDet consists of two parts, i.e., **the loss in the ARM** and **the loss in the ODM**.
- For the ARM, a binary class label (of being an object or not) is assigned to each anchor and regress its location and size simultaneously to get the refined anchor.
- After that, the refined anchors with the negative confidence less than the threshold are passed to the ODM to further predict object categories and accurate object locations and sizes.
- The loss function is:

$$\begin{aligned} \mathcal{L}(\{p_i\}, \{x_i\}, \{c_i\}, \{t_i\}) = & \frac{1}{N_{\text{arm}}} \left(\sum_i \mathcal{L}_b(p_i, [l_i^* \geq 1]) \right) \\ & + \sum_i [l_i^* \geq 1] \mathcal{L}_r(x_i, g_i^*) + \frac{1}{N_{\text{odm}}} \left(\sum_i \mathcal{L}_m(c_i, l_i^*) \right) \\ & + \sum_i [l_i^* \geq 1] \mathcal{L}_r(t_i, g_i^*) \end{aligned}$$

- where N_{arm} and N_{odm} are the **numbers of positive anchors in the ARM and ODM**, respectively.
- The **binary classification loss L_b** is the cross-entropy/log loss over two classes (object vs. not object)
- The **multi-class classification loss L_m** is the softmax loss over multiple classes confidences.
- Similar to Fast R-CNN, smooth L1 loss is used as the **regression loss L_r** .
- $[l_i^* \geq 1]$ means the regression loss is ignored for negative anchors.

5.2. Inference

- ARM first filters out the regularly tiled anchors with the negative confidence scores larger than the threshold 0.99.
- ODM takes over these refined anchors, and outputs top 400 high confident detections per image.
- NMS with jaccard overlap of 0.45 per class is applied.
- The top 200 high confident detections per image are retained to produce the final detection results.

6. Experimental Results

6.1. Ablation Study

Component	RefineDet320			
negative anchor filtering?	✓			
two-step cascaded regression?	✓	✓		
transfer connection block?	✓	✓	✓	
mAP (%)	80.0	79.5	77.3	76.2

Effectiveness of various designs on VOC 2007 test set

- All models are trained on VOC 2007 and VOC 2012 trainval sets, and tested on VOC 2007 test set.

With low dimension input (i.e., 320×320), *RefineDet* produces **80.0% mAP** with all above techniques, which is *the first method achieving above 80% mAP with such small input images*.

6.2. PASCAL VOC 2007 & 2012

Method	Backbone	Input size	#Boxes	FPS	mAP (%)	
					VOC 2007	VOC 2012
<i>two-stage:</i>						
Fast R-CNN[15]	VGG-16	$\sim 1000 \times 600$	~ 2000	0.5	70.0	68.4
Faster R-CNN[36]	VGG-16	$\sim 1000 \times 600$	300	7	73.2	70.4
OHEM[41]	VGG-16	$\sim 1000 \times 600$	300	7	74.6	71.9
HyperNet[25]	VGG-16	$\sim 1000 \times 600$	100	0.88	76.3	71.4
Faster R-CNN[36]	ResNet-101	$\sim 1000 \times 600$	300	2.4	76.4	73.8
ION[11]	VGG-16	$\sim 1000 \times 600$	4000	1.25	76.5	76.4
MR-CNN[14]	VGG-16	$\sim 1000 \times 600$	250	0.03	78.2	73.9
R-FCN[5]	ResNet-101	$\sim 1000 \times 600$	300	9	80.5	77.6
CoupleNet[54]	ResNet-101	$\sim 1000 \times 600$	300	8.2	82.7	80.4
<i>one-stage:</i>						
YOLO[34]	GoogleNet [45]	448 \times 448	98	45	63.4	57.9
RON384[24]	VGG-16	384 \times 384	30600	15	75.4	73.0
SSD321[13]	ResNet-101	321 \times 321	17080	11.2	77.1	75.4
SSD300*[30]	VGG-16	300 \times 300	8732	46	77.2	75.8
DSOD300[39]	DS/64-192-48-1	300 \times 300	8732	17.4	77.7	76.3
YOLOv2[35]	Darknet-19	544 \times 544	845	40	78.6	73.4
DSSD321[13]	ResNet-101	321 \times 321	17080	9.5	78.6	76.3
SSD512*[30]	VGG-16	512 \times 512	24564	19	79.8	78.5
SSD513[13]	ResNet-101	513 \times 513	43688	6.8	80.6	79.4
DSSD513[13]	ResNet-101	513 \times 513	43688	5.5	81.5	80.0
RefineDet320	VGG-16	320 \times 320	6375	40.3	80.0	78.1
RefineDet512	VGG-16	512 \times 512	16320	24.1	81.8	80.1
RefineDet320+	VGG-16	-	-	-	83.1	82.7
RefineDet512+	VGG-16	-	-	-	83.8	83.5

Detection results on PASCAL VOC 2007 & 2012

6.2.1. VOC 2007

- On VOC 2007, by using larger input size 512×512 , **RefineDet512** achieves **81.8% mAP**, surpassing SSD and DSSD.
- Comparing to the two-stage methods, RefineDet512 performs better than most of them except CoupleNet.

- **With multi-scale testing strategy, RefineDet achieves 83.1% (RefineDet320+) and 83.8% (RefineDet512+) mAPs**, which are much better than the state-of-the-art methods.
- RefineDet processes an image in **24.8ms (40.3 FPS)** and **41.5ms (24.1 FPS)** with input sizes **320×320** and **512×512**, respectively.

RefineDet is the first real-time method to achieve detection accuracy above 80% mAP on PASCAL VOC 2007.

- **RefineDet associates fewer anchor boxes** on the feature maps (e.g., 24564 anchor boxes in SSD512 vs. 16320 anchor boxes in RefineDet512).
- Only YOLOv1 and SSD300 are slightly faster than RefineDet320, but their accuracy are 16.6% and 2.5% worse than RefineDet.

In this story, **RetinaNet**, by **Facebook AI Research (FAIR)**, is reviewed. It is discovered that **there is extreme foreground-background class imbalance problem in one-stage detector**. And it is believed that this is the central cause which makes the performance of one-stage detectors **inferior to two-stage detectors**.

In RetinaNet, an one-stage detector, **by using focal loss, lower loss is contributed by “easy” negative samples so that the loss is focusing on “hard” samples**, which improves the prediction accuracy. With **ResNet+FPN as backbone** for feature extraction, **plus two task-specific subnetworks for classification and bounding box regression**, forming the **RetinaNet**, which achieves state-of-the-art performance, **outperforms Faster R-CNN**, the well-known two-stage detectors. It is a **2017 ICCV Best Student Paper Award** paper with more than **500 citations**. (The first author, Tsung-Yi Lin, has become Research Scientist at Google Brain when he was presenting RetinaNet in 2017 ICCV.) (

Sik-Ho Tsang
@ Medium)

A Demo of RetinaNet on Parking Lot Entrance Video
(<https://www.youtube.com/watch?v=51ujDJ-01oc>)
Another Demo of RetinaNet on Car Camera Video

Outline

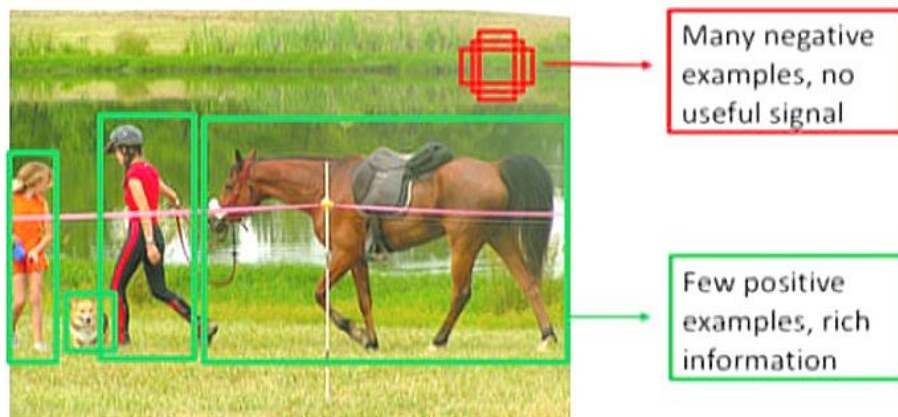
1. **Class Imbalance Problem of One-Stage Detector**
2. **Focal Loss**
3. **RetinaNet Detector**
4. **Ablation Study**
5. **Comparison with State-of-the-art Approaches**

1. Class Imbalance Problem of One-Stage Detector

1.1. Two-Stage Detectors

- In two-stage detectors such as Faster R-CNN, **the first stage, region proposal network (RPN)** narrows down the **number of candidate object locations to a small number (e.g. 1–2k)**, filtering out most background samples.
- At the **second stage, classification** is performed for each candidate object location. **Sampling heuristics** using fixed foreground-to-background ratio (1:3), **or online hard example mining (OHEM)** to select a small set of anchors (e.g., 256) for each minibatch.
- Thus, there is manageable class balance between foreground and background.

1.2. One-Stage Detectors



Many negative background examples, Few positive foreground examples

- A much larger set of candidate object locations is regularly sampled across an image (~100k locations), which densely cover spatial positions, scales and aspect ratios.
- The training procedure is still dominated by easily classified background examples. It is typically addressed via bootstrapping or hard example mining. But they are not efficient enough.

1.3. Number of Boxes Comparison

- YOLOv1: 98 boxes
- YOLOv2: ~1k
- OverFeat: ~1–2k
- SSD: ~8–26k

- **RetinaNet: ~100k.** RetinaNet can have ~100k boxes with the resolve of class imbalance problem using focal loss.

2. Focal Loss

2.1. Cross Entropy (CE) Loss

$$CE(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise.} \end{cases}$$

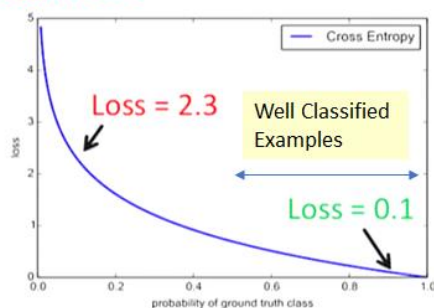
- The above equation is the CE loss for binary classification. $y \in \{\pm 1\}$ which is the ground-truth class and $p \in [0, 1]$ which is the model's estimated probability. It is straightforward to extend it to multi-class case. For notation convenience, p_t is defined and CE is rewritten as below:

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise,} \end{cases}$$

$$CE(p, y) = CE(p_t) = -\log(p_t).$$

- **When summed over a large number of easy examples, these small loss values can overwhelm the rare class.** Below is the example:

- 100000 easy : 100 hard examples
- 40x bigger loss from easy examples



Example

- Let treat the above figure as an example. If we have 100000 easy examples (0.1 each) and 100 hard examples (2.3 each). When we need to sum over to estimate the CE loss.
- The loss from easy examples = $100000 \times 0.1 = 10000$
- The loss from hard examples = $100 \times 2.3 = 230$
- $10000 / 230 = 43$. It is about 40x bigger loss from easy examples.
- Thus, CE loss is not a good choice when there is extreme class imbalance.

2.2. α -Balanced CE Loss

$$\text{CE}(p_t) = -\alpha_t \log(p_t).$$

- To address the class imbalance, one method is to add a weighting factor α for class 1 and $1 - \alpha$ for class -1.
- α may be set by inverse class frequency or treated as a hyperparameter to set by cross validation.
- As seen at the two-stage detectors, α is implicitly implemented by selecting the foreground-to-background ratio of 1:3.

2.3. Focal Loss (FL)

$$\text{FL}(p_t) = -(1 - p_t)^\gamma \log(p_t).$$

- The loss function is reshaped to down-weight easy examples and thus focus training on hard negatives. A modulating factor $(1-p_t)^\gamma$ is added to the cross entropy loss where γ is tested from [0,5] in the experiment.
- There are two properties of the FL:
 1. When an example is misclassified and p_t is small, the modulating factor is near 1 and the loss is unaffected. As $p_t \rightarrow 1$, the factor goes to 0 and the loss for well-classified examples is down-weighted.
 2. The focusing parameter γ smoothly adjusts the rate at which easy examples are down-weighted. When $\gamma = 0$, FL is equivalent to CE. When γ is increased, the effect of the modulating factor is likewise increased. ($\gamma=2$ works best in experiment.)
- For instance, with $\gamma = 2$, an example classified with $p_t = 0.9$ would have 100 lower loss compared with CE and with $p_t = 0.968$ it would have 1000 lower loss. This in turn increases the importance of correcting misclassified examples.
- The loss is scaled down by at most $4\times$ for $p_t \leq 0.5$ and $\gamma = 2$.

2.4. α -Balanced Variant of FL

$$\text{FL}(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t).$$

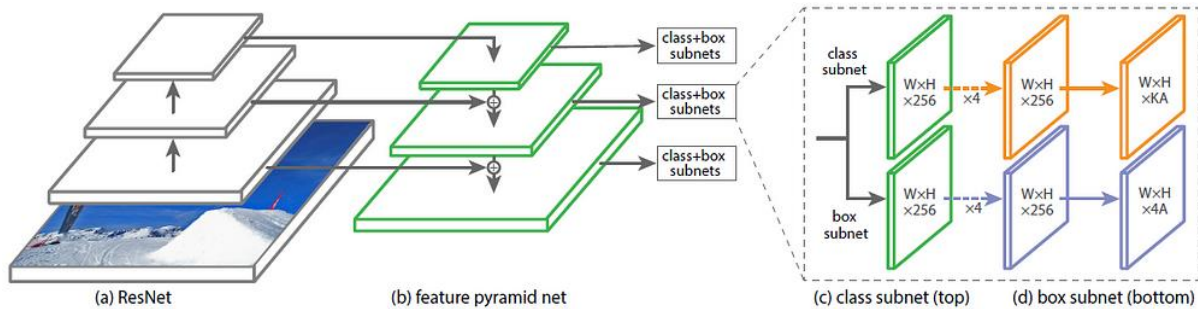
- The above form is used in experiment in practice where α is added into the equation, which yields slightly improved accuracy over the one without α . And using **sigmoid activation function for computing p** resulting in **greater numerical stability**.

- γ : Focus more on hard examples.
- α : Offset class imbalance of number of examples.

2.5. Model Initialization

- A prior π is set for the value of p at the start of training, so that the model's estimated p for examples of the rare class is low, e.g. **0.01**, in order to improve the training stability in the case of heavy class imbalance.
- It is found that training RetinaNet uses standard CE loss **WITHOUT** using prior π for initialization leads to **network divergence during training** and eventually failed.
- And results are insensitive to the exact value of π . And $\pi = 0.01$ is used for all experiments.

3. RetinaNet Detector



RetinaNet Detector Architecture

3.1. (a) and (b) Backbone

- **ResNet** is used for deep feature extraction.
- **Feature Pyramid Network (FPN)** is used on top of ResNet for constructing a rich multi-scale feature pyramid from one single resolution input image. (Originally, FPN is a two-stage detector which has state-of-the-art results. Please read my review about FPN if interested.)
- FPN is multiscale, semantically strong at all scales, and fast to compute.
- There are some modest changes for the FPN here. A pyramid is generated from P3 to P7. Some major changes are: P2 is not used now due to computational reasons. (ii) P6 is computed by strided convolution instead of downsampling. (iii) P7 is included additionally to improve the accuracy of large object detection.

3.2. Anchors

- The anchors have the areas of 32^2 to 512^2 on pyramid levels from P3 to P7 respectively.

- **Three aspect ratios {1:2, 1:1, 2:1}** are used.
- For denser scale coverage, **anchors of sizes $\{2^0, 2^{(1/3)}, 2^{(2/3)}\}$** are added at each pyramid level.
- In total, **9 anchors per level**.
- **Across levels, scale is covered from 32 to 813 pixels.**
- **Each anchor**, there is a **length K one-hot vector of classification targets** (K : number of classes), and a **4-vector of box regression targets**.
- **Anchors are assigned to ground-truth object boxes using IoU threshold of 0.5** and to **background if IoU is in $[0,0.4)$** . Each anchor is assigned at most one object box, and set the corresponding class entry to one and all other entries to 0 in that K one-hot vector. If anchor is **unassigned if IoU is in $[0.4,0.5)$** and ignored during training.
- Box regression is computed as the offset between anchor and assigned object box, or omitted if there is no assignment.

3.3. (c) Classification Subnet

- This classification subnet **predicts the probability of object presence at each spatial position** for each of the A anchors and K object classes.
- The subnet is a **FCN** which applies four 3×3 conv layers, each with C filters and each followed by ReLU activations, followed by a 3×3 conv layer with KA filters. (K classes, $A=9$ anchors, and $C = 256$ filters)

3.4. (d) Box Regression Subnet

- This subnet is a **FCN** to each pyramid level for the purpose of regressing the offset from each anchor box to a nearby ground-truth object, if one exists.
- It is identical to the classification subnet except that it terminates in **4A linear outputs per spatial location**.
- It is a **class-agnostic bounding box regressor** which uses fewer parameters, which is found to be equally effective.

3.5. Inference

- The network only decodes box predictions from **at most 1k top-scoring predictions per FPN level, after thresholding detector confidence at 0.05**.
- **The top predictions from all levels are merged and non-maximum suppression (NMS) with a threshold of 0.5 is applied** to yield the final detections.

3.6. Training

- Thus, during training, the **total focal loss** of an image is computed as **the sum of the focal loss over all 100k anchors, normalized by the number of anchors assigned to a ground-truth box.**
- ImageNet1K pre-trained ResNet-50-FPN and ResNet-101-FPN are used.

4. Ablation Study

- **COCO dataset** is used. **COCO trainval35k** split is used for **training**. And **minival (5k)** split is used for **validation**.

α	AP	AP ₅₀	AP ₇₅	γ	α	AP	AP ₅₀	AP ₇₅
.10	0.0	0.0	0.0	0	.75	31.1	49.4	33.0
.25	10.8	16.0	11.7	0.1	.75	31.4	49.9	33.1
.50	30.2	46.7	32.8	0.2	.75	31.9	50.7	33.4
.75	31.1	49.4	33.0	0.5	.50	32.9	51.7	35.2
.90	30.8	49.7	32.3	1.0	.25	33.7	52.0	36.2
.99	28.7	47.4	29.9	2.0	.25	34.0	52.5	36.5
.999	25.1	41.7	26.1	5.0	.25	32.2	49.6	34.8

(a) Varying α for CE loss ($\gamma = 0$)

(b) Varying γ for FL (w. optimal α)

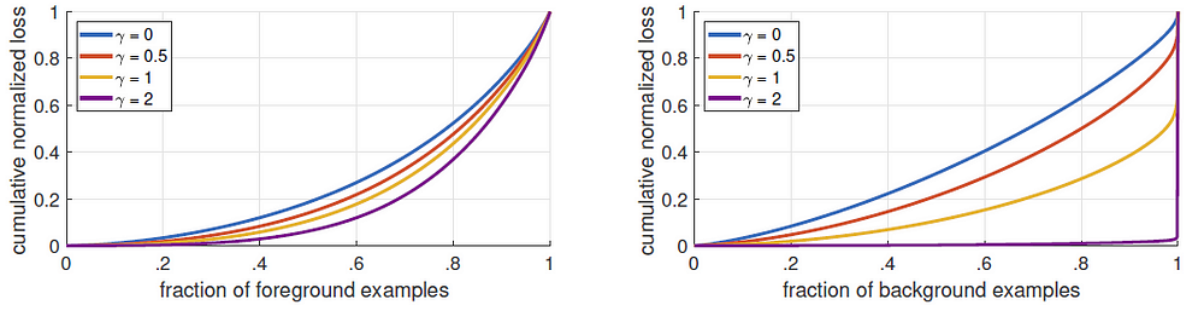
α for CE loss (Left), γ for FL (Right)

4.1. α for α -Balanced CE loss

- ResNet-50 is used.
- First, α -Balanced CE loss with different α is tested.
- $\alpha = 0.75$ gives a gain of 0.9 AP.

4.2. γ for FL

- $\gamma=0$ is α -Balanced CE loss.
- When γ increases, easy examples get discounted to the loss.
- $\gamma=2$ and $\alpha=0.25$ yields a 2.9 AP improvement over α -Balanced CE loss ($\alpha=0.75$).
- It is observed that lower α 's are selected for higher γ 's.
- The benefit of changing is much larger, and indeed the best α 's ranged in just $[0.25, 0.75]$ with $\alpha \in [0.01; 0.999]$ tested.



Cumulative distribution functions of the normalized loss for positive and negative samples

4.3. Foreground and Background Samples Analysis

Foreground samples

- The loss from lowest to highest is sorted and plot its cumulative distribution function (CDF) for both positive and negative samples and for different settings for γ .
- Approximately 20% of the hardest positive samples account for roughly half of the positive loss.
- As γ increases more of the loss gets concentrated in the top 20% of examples, but the effect is minor.

Background samples

- As γ increases, substantially more weight becomes concentrated on the hard negative examples.
- The vast majority of the loss comes from a small fraction of samples.
- **FL can effectively discount the effect of easy negatives, focusing all attention on the hard negative examples.**

4.4. Anchor Density

#sc	#ar	AP	AP ₅₀	AP ₇₅
1	1	30.3	49.0	31.8
2	1	31.9	50.0	34.0
3	1	31.8	49.4	33.7
1	3	32.4	52.3	33.9
2	3	34.2	53.1	36.5
3	3	34.0	52.5	36.5
4	3	33.8	52.1	36.2

Different Number of Scales (#sc) and Aspect Ratios (#ar)

- Using one square anchor (#sc=1, #ar=1) achieves 30.3% AP which is not bad.
- AP can be improved by nearly 4 points (34.0) using 3 scales and 3 aspect ratios.
- Increasing beyond 6–9 anchors did not shown further gains.

4.5. FL vs OHEM (Online Hard Example Mining)

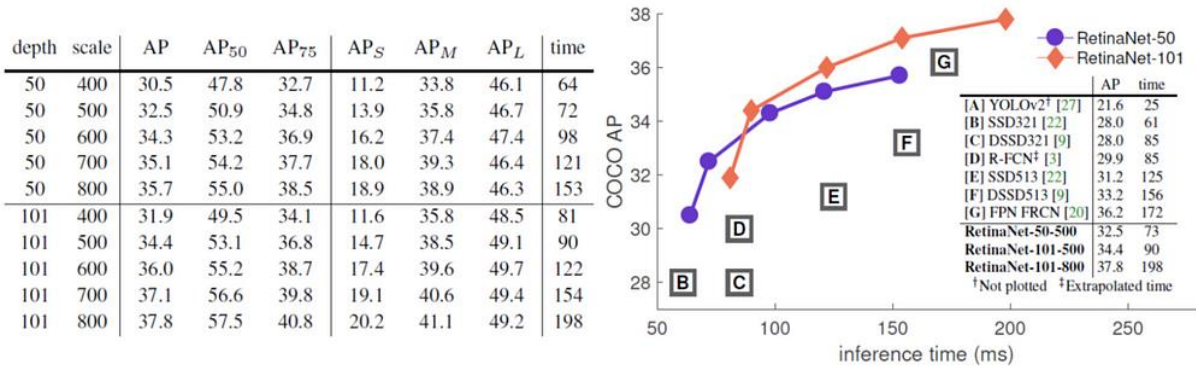
method	batch size	nms thr	AP	AP ₅₀	AP ₇₅
OHEM	128	.7	31.1	47.2	33.2
OHEM	256	.7	31.8	48.8	33.9
OHEM	512	.7	30.6	47.0	32.6
OHEM	128	.5	32.8	50.3	35.1
OHEM	256	.5	31.0	47.4	33.0
OHEM	512	.5	27.6	42.0	29.2
OHEM 1:3	128	.5	31.1	47.2	33.2
OHEM 1:3	256	.5	28.3	42.4	30.3
OHEM 1:3	512	.5	24.0	35.5	25.8
FL	n/a	n/a	36.0	54.9	38.7

FL vs OHEM (Online Hard Example Mining)

- Here, ResNet-101 is used.
- In OHEM, each example is scored by its loss, non-maximum suppression (NMS) is then applied, and a minibatch is constructed with the highest-loss examples.
- Like the focal loss, OHEM puts more emphasis on misclassified examples.
- But unlike FL, OHEM completely discards easy examples.
- After applying nms to all examples, the minibatch is constructed to enforce a 1:3 ratio between positives and negatives.
- The best setting for OHEM (no 1:3 ratio, batch size 128, NMS of 0.5) achieves 32.8% AP.
- And FL obtains 36.0% AP, i.e. a gap 3.2 AP, which proves the effectiveness of FL.
- Note: Authors also tested Hinge Loss, where loss is set to 0 above a certain value of pt. However, training is unstable.

Comparison with State-of-the-art Approaches

Speed versus Accuracy Tradeoff



Speed versus Accuracy

- **RetinaNet-101-600:** RetinaNet with ResNet-101-FPN and a 600 pixel image scale, matches the accuracy of the recently published ResNet-101-FPN Faster R-CNN (FPN) while running in 122 ms per image compared to 172 ms (both measured on an Nvidia M40 GPU).
- Larger backbone networks yield higher accuracy, but also slower inference speeds.
- Training time ranges from 10 to 35 hours.
- Using larger scales allows RetinaNet to surpass the accuracy of all two-stage approaches, while still being faster.
- Except YOLOv2 (which targets on extremely high frame rate), RetinaNet outperforms SSD, DSSD, R-FCN and FPN.
- For faster runtimes, there is only one operating point (500 pixel input) at which RetinaNet using ResNet-50-FPN improves over the one using ResNet-101-FPN.

5.2. State-of-the-art Accuracy

	backbone	AP	AP ₅₀	AP ₇₅	AP _S	AP _M	AP _L
<i>Two-stage methods</i>							
Faster R-CNN+++ [16]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [20]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [17]	Inception-ResNet-v2 [34]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [32]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	52.1
<i>One-stage methods</i>							
YOLOv2 [27]	DarkNet-19 [27]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [22, 9]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [9]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet (ours)	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet (ours)	ResNeXt-101-FPN	40.8	61.1	44.1	24.1	44.2	51.2

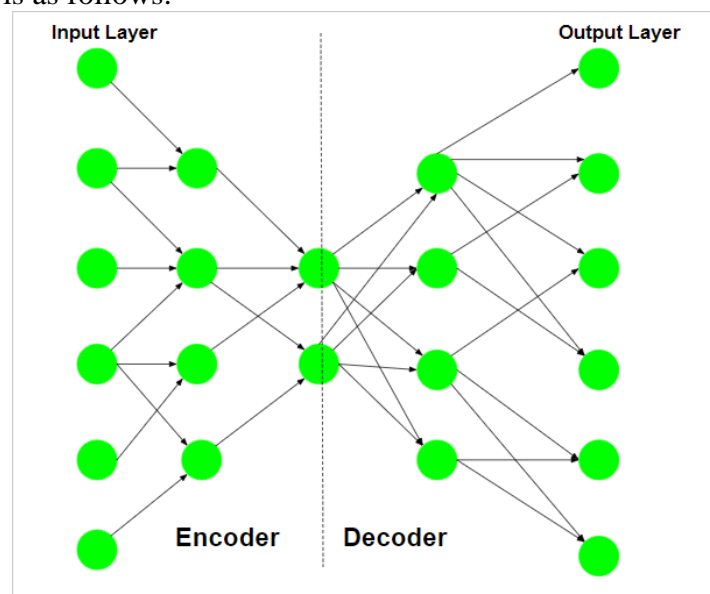
Object detection single-model results (bounding box AP), vs. state-of-the-art on COCO test-dev

- **RetinaNet Using ResNet-101-FPN:** **RetinaNet-101-800** model trained using **scale jitter** and for $1.5\times$ longer than the models in Table (5.1).
- Compared to existing one-stage detectors, it achieves a healthy 5.9 point AP gap (39.1 vs. 33.2) with the closest competitor, DSSD.
- Compared to recent two-stage methods, RetinaNet achieves a 2.3 point gap above the top-performing Faster R-CNN model based on Inception-ResNet-v2-TDM. (If interested, please read my review about Inception-ResNet-v2 and TDM.)
- RetinaNet Using ResNeXt-101-FPN: Plugging in ResNeXt-32x8d-101-FPN [38] as the RetinaNet backbone further improves results another 1.7 AP, surpassing 40 AP on COCO. (If interested, please read my review about ResNeXt.)

By using focal loss, the total loss can be balanced adaptively between easy samples and hard samples.

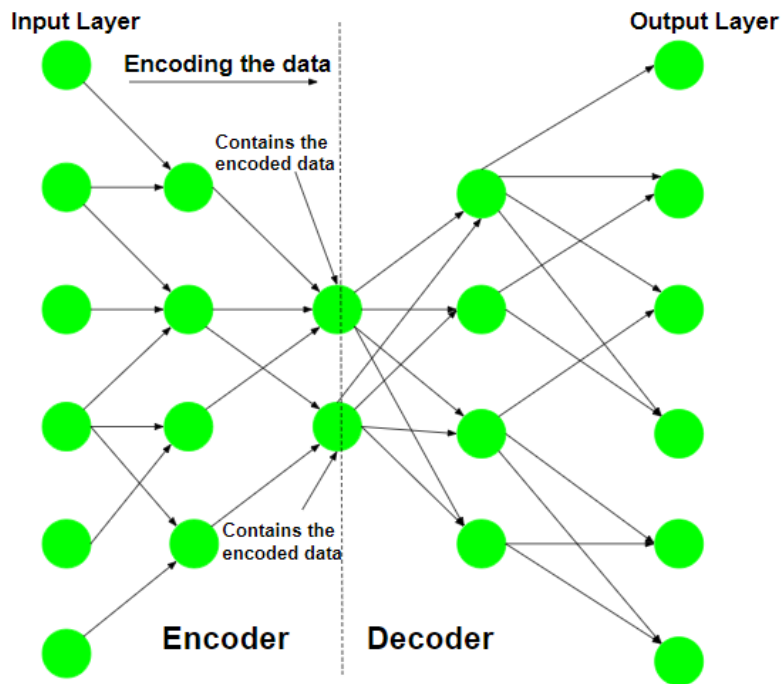
ML | AUTO-ENCODERS

A typical use of a **Neural Network** is a case of supervised learning. It involves training data that **contains an output label**. The neural network tries to learn the mapping from the given input to the given output label. But what if the output label is replaced by the input vector itself? Then the network will try to find the mapping from the input to itself. This would be the identity function which is a trivial mapping. **But if the network is not allowed to simply copy the input, then the network will be forced to capture only the salient features.** This constraint opens up a different field of applications for Neural Networks which was unknown. The primary applications are dimensionality reduction and specific data compression. The network is first trained on the given input. The network tries to reconstruct the given input from the features it picked up and gives an approximation to the input as the output. The training step involves the computation of the error and backpropagating the error. The typical architecture of an Auto-encoder resembles a bottleneck. The schematic structure of an autoencoder is as follows:

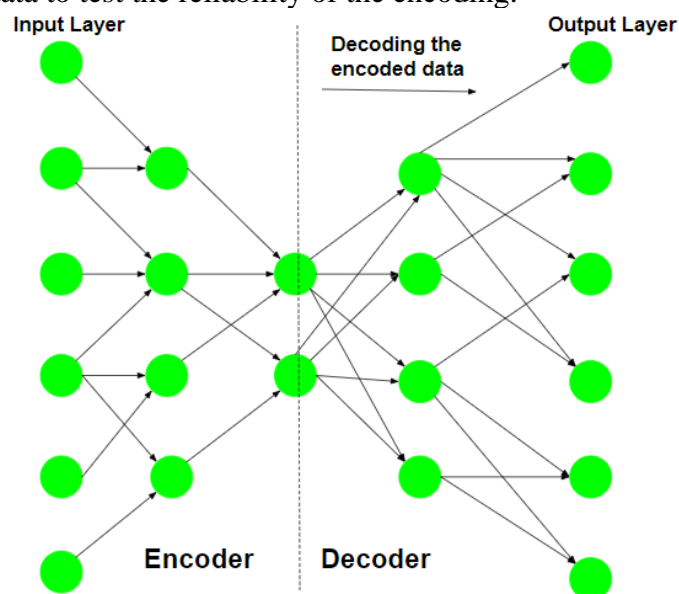


The encoder part of the network is used for **encoding** and sometimes even for **data compression** purposes although it is **not very effective as compared to other general compression techniques like JPEG**. Encoding is achieved by the encoder part of the network which has a **decreasing number of hidden units** in each layer. Thus this part is forced to pick up only the most significant and representative features of the data. The second half of the network performs the **Decoding function**. This part has an **increasing number of hidden units** in each layer and thus tries to reconstruct the original input from the encoded data. Thus Auto-encoders are an **unsupervised learning technique**.

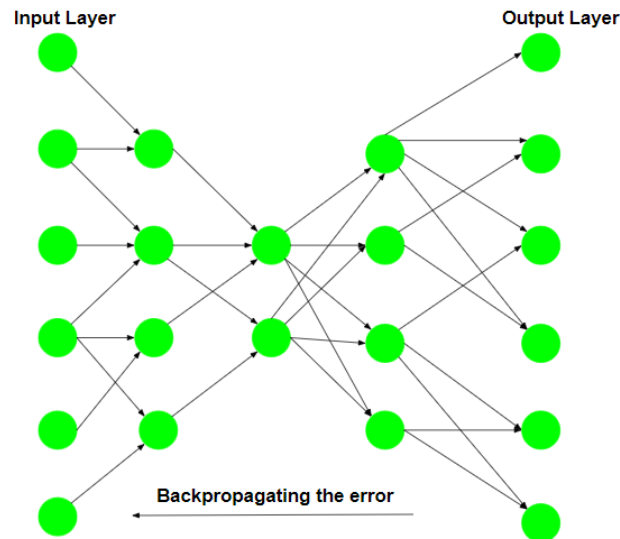
Example: See the below code, in autoencoder training data, is fitted to itself. That's why instead of fitting `X_train` to `Y_train` we have used `X_train` in both places. **Training of an Auto-encoder for data compression:** For a data compression procedure, the most important aspect of the compression is the reliability of the reconstruction of the compressed data. This requirement dictates the structure of the Auto-encoder as a bottleneck. **Step 1: Encoding the input data** The Auto-encoder first tries to encode the data using the initialized weights and biases.



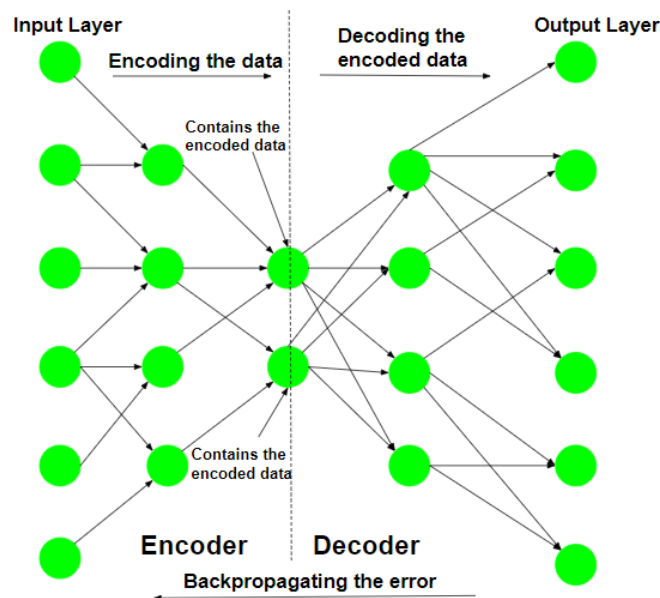
Step 2: Decoding the input data The Auto-encoder tries to reconstruct the original input from the encoded data to test the reliability of the encoding.



Step 3: Backpropagating the error After the reconstruction, the loss function is computed to determine the reliability of the encoding. The error generated is backpropagated.



The above-described training process is reiterated several times until an acceptable level of reconstruction is reached.



After the training process, only the encoder part of the Auto-encoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are:-

- **Keep small Hidden Layers:** If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- **Regularization:** In this method, a **loss term is added** to the cost function which encourages the network to train in ways other than copying the input.
- **Denoising:** Another way of constraining the network is to **add noise to the input** and teach the network how to remove the noise from the data.
- **Tuning the Activation Functions:** This method involves **changing the activation functions of various nodes so that a majority of the nodes are dormant** thus effectively reducing the size of the hidden layers.

The different variations of Auto-encoders and the advantages and disadvantages of using them are:-

- **Denoising Auto-encoder:** This type of auto-encoder works on a partially corrupted input and trains to recover the original undistorted image. As mentioned above, this method is an effective way to constrain the network from simply copying the input and thus learn the underlying structure and important features of the data.

Advantages:

1. This type of autoencoder can extract important features and reduce the noise or the useless features.
2. Denoising autoencoders can be used as a form of data augmentation, the restored images can be used as augmented data thus generating additional training samples.

Disadvantages:

1. Selecting the right type and level of noise to introduce can be challenging and may require domain knowledge.
 2. Denoising process can result into loss of some information that is needed from the original input. This loss can impact accuracy of the output.
- **Sparse Auto-encoder:** This type of auto-encoder typically contains more hidden units than the input but only a few are allowed to be active at once. This property is called the sparsity of the network. The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.

Advantages:

1. The sparsity constraint in sparse autoencoders helps in filtering out noise and irrelevant features during the encoding process.
2. These auto-encoders often learn important and meaningful features due to their emphasis on sparse activations.

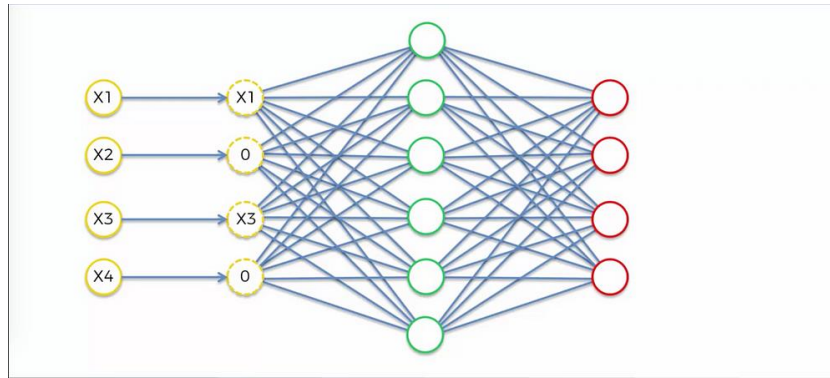
Disadvantages:

1. The choice of hyperparameters play a significant role in the performance of this autoencoder. Different inputs should result in the activation of different nodes of the network.
2. The application of sparsity constraint increases computational complexity.

DENOISING AUTOENCODERS EXPLAINED

Autoencoders are Neural Networks which are commonly used for feature selection and extraction. However, when there are more nodes in the hidden layer than there are inputs, the Network is risking to learn the so-called “Identity Function”, also called “Null Function”, meaning that the output equals the input, marking the Autoencoder useless.

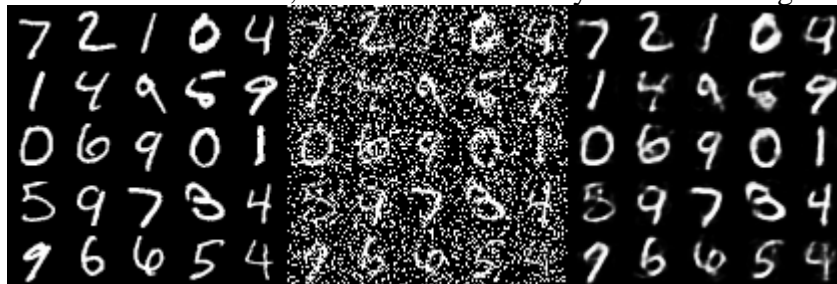
Denoising Autoencoders solve this problem by corrupting the data on purpose by randomly turning some of the input values to zero. In general, the percentage of input nodes which are being set to zero is about 50%. Other sources suggest a lower count, such as 30%. It depends on the amount of data and input nodes you have.



Architecture of a DAE. Copyright by Kirill Eremenko ([Deep Learning A-Z™: Hands-On Artificial Neural Networks](#))

When calculating the Loss function, it is important to compare the output values with the original input, not with the corrupted input. That way, the risk of learning the identity function instead of extracting features is eliminated.

A great implementation has been posted by [opendeep.org](#) where they use Theano to build a very basic Denoising Autoencoder and train it on the MNIST dataset. The OpenDeep articles are very basics and are made for beginners. So even if you don't have too much experience with Neural Networks, the article is definitely worth checking out!



A Generative Adversarial Network (GAN) is a deep learning architecture that consists of two neural networks competing against each other in a zero-sum game framework. The goal of GANs is to generate new, synthetic data that resembles some known data distribution.

What is a Generative Adversarial Network?

Generative Adversarial Networks (GANs) are a powerful class of neural networks that are used for unsupervised learning. It was developed and introduced by Ian J. Goodfellow in 2014. GANs are basically made up of a system of two competing neural network models which compete with each other and are able to analyze, capture and copy the variations within a dataset.

Why were GANs developed in the first place?

It has been noticed most of the mainstream neural nets can be easily fooled into misclassifying things by adding only a small amount of noise into the original data. Surprisingly, the model after adding noise has higher confidence in the wrong prediction than when it predicted correctly. ***The reason for such an adversary is that most machine learning models learn from a limited amount of data, which is a huge drawback, as it is prone to overfitting. Also, the mapping between the input and the output is almost linear.***

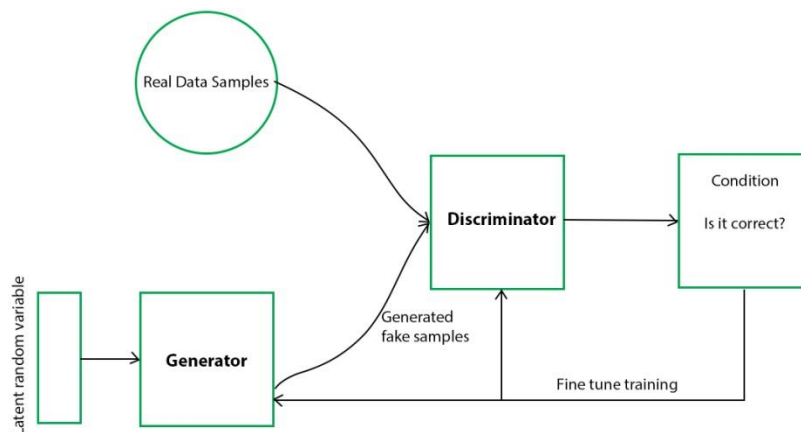
Although, it may seem that the boundaries of separation between the various classes are linear, but in reality, they are composed of linearities, and even a small change in a point in the feature space might lead to the misclassification of data.

How do GANs work?

Generative Adversarial Networks (GANs) can be broken down into three parts:

- **Generative:** To learn a generative model, which describes how data is generated in terms of a probabilistic model.
- **Adversarial:** The training of a model is done in an adversarial setting.
- **Networks:** Use deep neural networks as artificial intelligence (AI) algorithms for training purposes.

In GANs, there is a **Generator** and a **Discriminator**. The Generator generates fake samples of data (be it an image, audio, etc.) and tries to fool the Discriminator. The Discriminator, on the other hand, tries to distinguish between the real and fake samples. The Generator and the Discriminator are both Neural Networks and they both run in competition with each other in the training phase. The steps are repeated several times and in this, the Generator and Discriminator get better and better in their respective jobs after each repetition. The work can be visualized by the diagram given below:



Generative Adversarial Network Architecture and its Components

Here, the generative model captures the distribution of data and is trained in such a manner that it tries to maximize the probability of the Discriminator making a mistake. The Discriminator, on the other hand, is based on a model that estimates the probability that the sample that it got is received from the training data and not from the Generator. The GANs are formulated as a minimax game, where the Discriminator is trying to minimize its reward $V(D, G)$ and the Generator is trying to minimize the Discriminator's reward or in other words, maximize its loss. It can be mathematically described by the formula below:

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

Loss function for a GAN Model

where,

- G = Generator
- D = Discriminator
- $P_{data}(x)$ = distribution of real data
- $P(z)$ = distribution of generator
- x = sample from $P_{data}(x)$
- z = sample from $P(z)$
- $D(x)$ = Discriminator network
- $G(z)$ = Generator network

Generator Model

The Generator is trained while the Discriminator is idle. After the Discriminator is trained by the generated fake data of the Generator, we can get its predictions and use the results for training the Generator and get better from the previous state to try and fool the Discriminator.

Discriminator Model

The Discriminator is trained while the Generator is idle. In this phase, the network is only forward propagated and no back-propagation is done. The Discriminator is trained on real data for n epochs and sees if it can correctly predict them as real. Also, in this phase, the Discriminator is also trained on the fake generated data from the Generator and see if it can correctly predict them as fake.

Different Types of GAN Models

1. **Vanilla GAN:** This is the simplest type of GAN. Here, the Generator and the Discriminator are simple multi-layer perceptrons. In vanilla GAN, the algorithm is really simple, it tries to optimize the mathematical equation using stochastic gradient descent.
2. **Conditional GAN (CGAN):** CGAN can be described as a deep learning method in which some conditional parameters are put into place. In CGAN, an additional parameter 'y' is added to the Generator for generating the corresponding data. Labels are also put into the input to the Discriminator in order for the Discriminator to help distinguish the real data from the fake generated data.
3. **Deep Convolutional GAN (DCGAN):** DCGAN is one of the most popular and also the most successful implementations of GAN. It is composed of ConvNets in place of multi-layer perceptrons. The ConvNets are implemented without max pooling, which is in fact replaced by convolutional stride. Also, the layers are not fully connected.
4. **Laplacian Pyramid GAN (LAPGAN):** The Laplacian pyramid is a linear invertible image representation consisting of a set of band-pass images, spaced an octave apart, plus a low-frequency residual. This approach uses multiple numbers of Generator and Discriminator networks and different levels of the Laplacian Pyramid. This approach is mainly used because it produces very high-quality images. The image is down-sampled at first at each layer of the pyramid and then it is again up-scaled at each layer in a backward pass where the image acquires some noise from the Conditional GAN at these layers until it reaches its original size.

5. **Super Resolution GAN (SRGAN):** SRGAN as the name suggests is a way of designing a GAN in which a deep neural network is used along with an adversarial network in order to produce higher-resolution images. This type of GAN is particularly useful in optimally up-scaling native low-resolution images to enhance their details minimizing errors while doing so.

TRANSFORMERS: ATTENTION BASED ENCODER AND DECODER: EG- BERT(BIDIRECTIONAL ENCODER REPRESENTATIONS FROM TRANSFORMERS), GENERATIVE PRETRAINED TRANSFORMERS GPT-3, GPT-2, BERT, XLNET, AND ROBERTA

The Transformer is an architecture that uses Attention to significantly improve the performance of deep learning NLP translation models. It was first introduced in the paper Attention is all you need and was quickly established as the leading architecture for most text data applications.

Since then, numerous projects including Google's BERT and OpenAI's GPT series have built on this foundation and published performance results that handily beat existing state-of-the-art benchmarks.

Over a series of articles, I'll go over the basics of Transformers, its architecture, and how it works internally. We will cover the Transformer functionality in a top-down manner. In later articles, we will look under the covers to understand the operation of the system in detail. We will also do a deep dive into the workings of the multi-head attention, which is the heart of the Transformer.

Here's a quick summary of the previous and following articles in the series. My goal throughout will be to understand not just how something works but why it works that way.

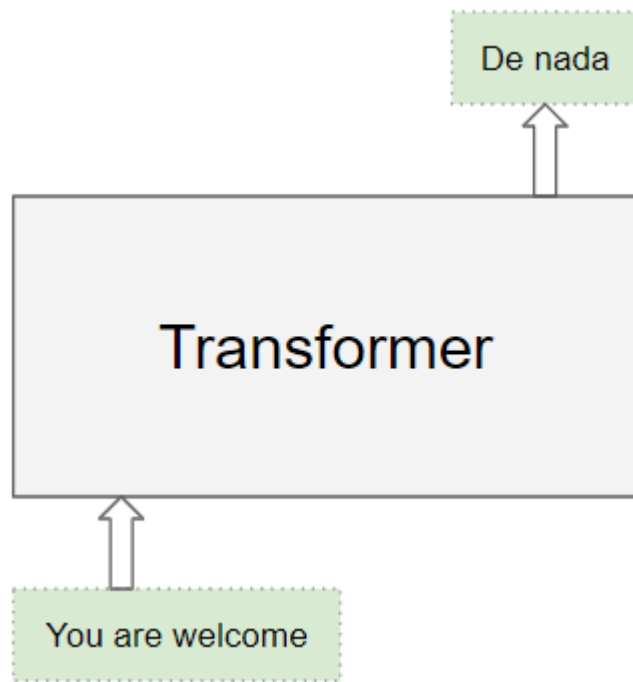
1. **Overview of functionality — this article** (*How Transformers are used, and why they are better than RNNs. Components of the architecture, and behavior during Training and Inference*)
2. How it works (*Internal operation end-to-end. How data flows and what computations are performed, including matrix representations*)
3. Multi-head Attention (*Inner workings of the Attention module throughout the Transformer*)
4. Why Attention Boosts Performance (*Not just what Attention does but why it works so well. How does Attention capture the relationships between words in a sentence*)

And if you're interested in NLP applications in general, I have some other articles you might like.

1. Beam Search (*Algorithm commonly used by Speech-to-Text and NLP applications to enhance predictions*)
2. Bleu Score (*Bleu Score and Word Error Rate are two essential metrics for NLP models*)

What is a Transformer

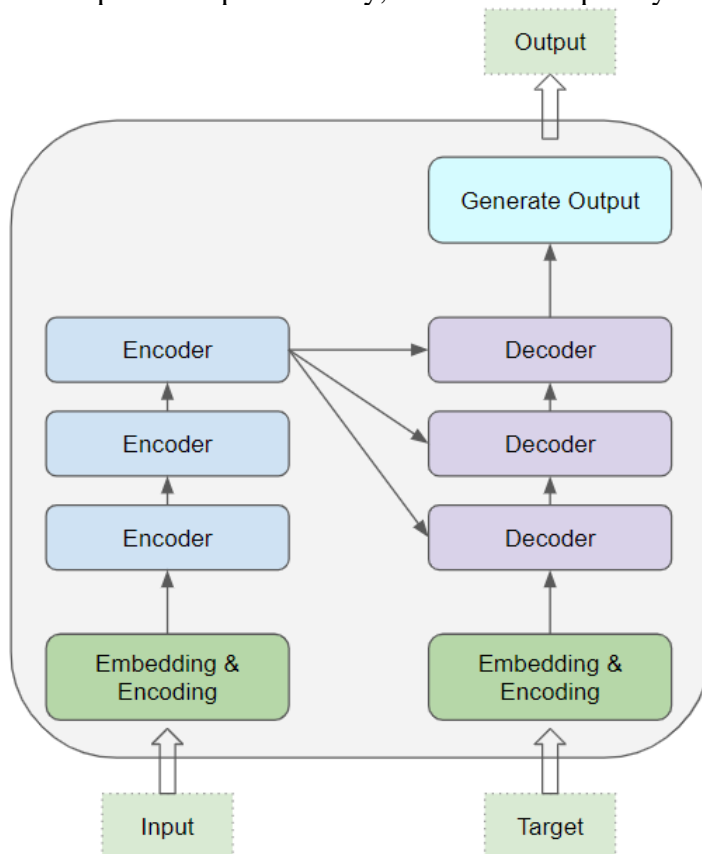
The Transformer architecture excels at handling text data which is inherently sequential. They take a text sequence as input and produce another text sequence as output. eg. to translate an input English sentence to Spanish.



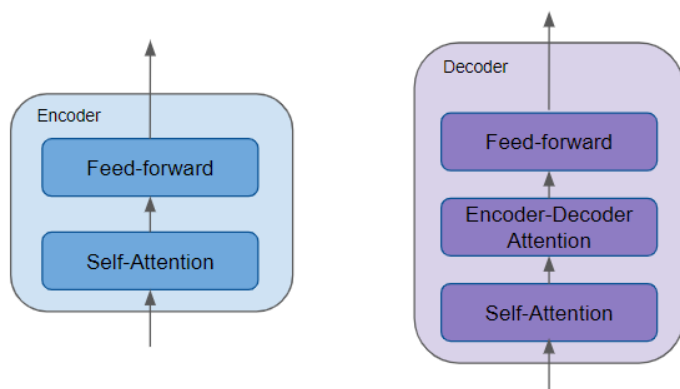
(Image by Author)

At its core, it contains a stack of Encoder layers and Decoder layers. To avoid confusion we will refer to the individual layer as an Encoder or a Decoder and will use Encoder stack or Decoder stack for a group of Encoder layers.

The Encoder stack and the Decoder stack each have their corresponding Embedding layers for their respective inputs. Finally, there is an Output layer to generate the final output.

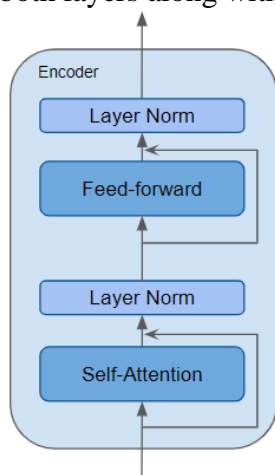


All the Encoders are identical to one another. Similarly, all the Decoders are identical.



- The Encoder contains the all-important Self-attention layer that computes the relationship between different words in the sequence, as well as a Feed-forward layer.
- The Decoder contains the Self-attention layer and the Feed-forward layer, as well as a second Encoder-Decoder attention layer.
- Each Encoder and Decoder has its own set of weights.

The Encoder is a reusable module that is the defining component of all Transformer architectures. In addition to the above two layers, it also has Residual skip connections around both layers along with two LayerNorm layers.



(Image by Author)

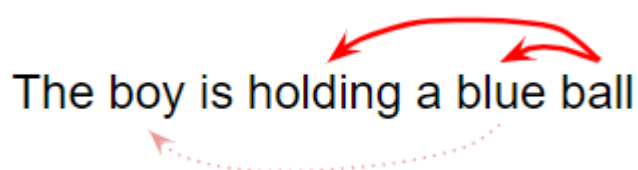
There are many variations of the Transformer architecture. Some Transformer architectures have no Decoder at all and rely only on the Encoder.

What does Attention Do?

The key to the Transformer's ground-breaking performance is its use of Attention.

While processing a word, Attention enables the model to focus on other words in the input that are closely related to that word.

eg. 'Ball' is closely related to 'blue' and 'holding'. On the other hand, 'blue' is not related to 'boy'.



The Transformer architecture uses self-attention by relating every word in the input sequence to every other word.

eg. Consider two sentences:

- The *cat* drank the milk because **it** was hungry.
- The cat drank the *milk* because **it** was sweet.

In the first sentence, the word 'it' refers to 'cat', while in the second it refers to 'milk'. When the model processes the word 'it', self-attention gives the model more information about its meaning so that it can associate 'it' with the correct word.



Dark colors represent higher attention (Image by Author)

To enable it to handle more nuances about the intent and semantics of the sentence, Transformers include multiple attention scores for each word.

eg. While processing the word 'it', the first score highlights 'cat', while the second score highlights 'hungry'. So when it decodes the word 'it', by translating it into a different language, for instance, it will incorporate some aspect of both 'cat' and 'hungry' into the translated word.



(Image by Author)

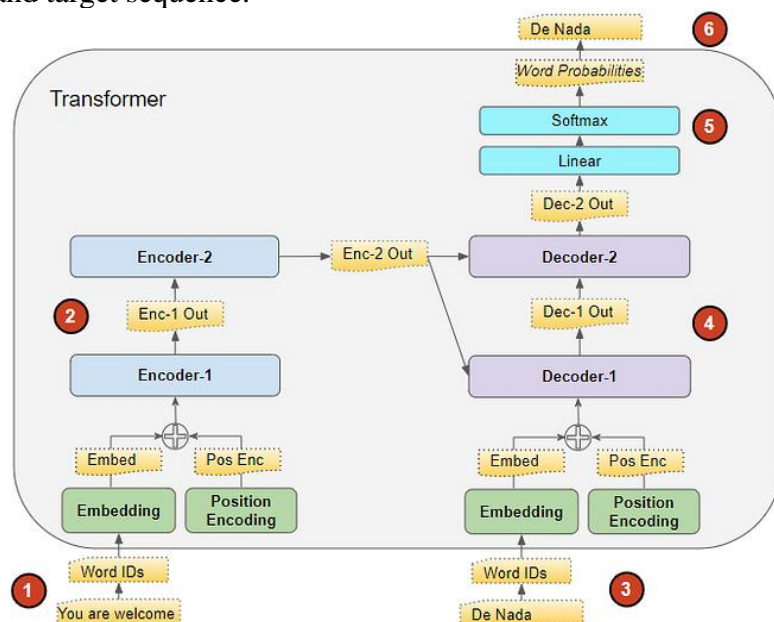
Training the Transformer

The Transformer works slightly differently during Training and while doing Inference.

Let's first look at the flow of data during Training. Training data consists of two parts:

- The source or input sequence (eg. "You are welcome" in English, for a translation problem)
- The destination or target sequence (eg. "De nada" in Spanish)

The Transformer's goal is to learn how to output the target sequence, by using both the input and target sequence.



(Image by Author)

The Transformer processes the data like this:

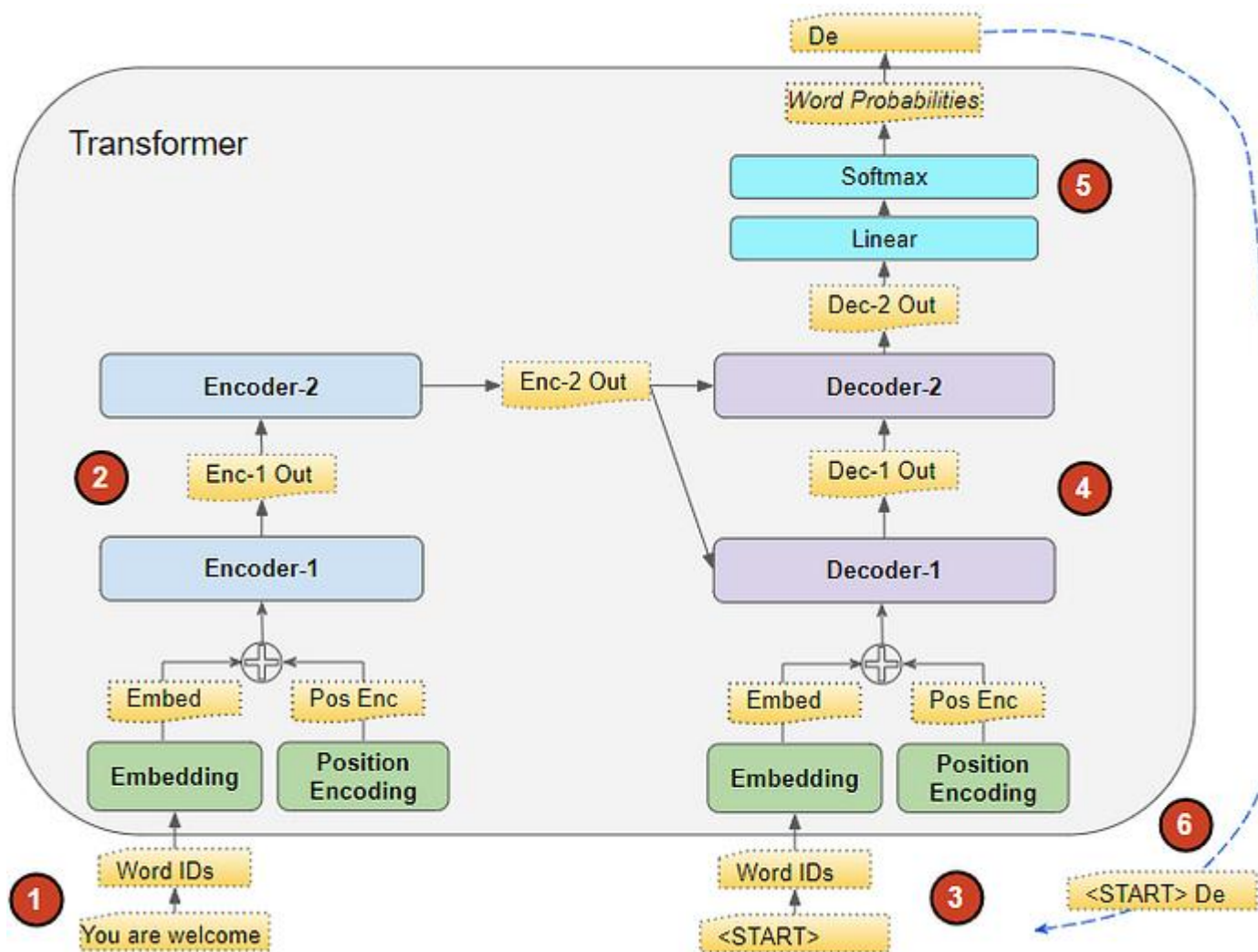
1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
3. The target sequence is prepended with a start-of-sentence token, converted into Embeddings (with Position Encoding), and fed to the Decoder.
4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
5. The Output layer converts it into word probabilities and the final output sequence.
6. The Transformer's Loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients to train the Transformer during back-propagation.

Inference

During Inference, we have only the input sequence and don't have the target sequence to pass as input to the Decoder. The goal of the Transformer is to produce the target sequence from the input sequence alone.

So, like in a Seq2Seq model, we generate the output in a loop and feed the output sequence from the previous timestep to the Decoder in the next timestep until we come across an end-of-sentence token.

The difference from the Seq2Seq model is that, at each timestep, we re-feed the entire output sequence generated thus far, rather than just the last word.



Inference flow, after first timestep (Image by Author)

The flow of data during Inference is:

1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
3. Instead of the target sequence, we use an empty sequence with only a start-of-sentence token. This is converted into Embeddings (with Position Encoding) and fed to the Decoder.
4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
5. The Output layer converts it into word probabilities and produces an output sequence.
6. We take the last word of the output sequence as the predicted word. That word is now filled into the second position of our Decoder input sequence, which now contains a start-of-sentence token and the first word.
7. Go back to step #3. As before, feed the new Decoder sequence into the model. Then take the second word of the output and append it to the Decoder sequence. Repeat this until it predicts an end-of-sentence token. Note that since the Encoder sequence does not change for each iteration, we do not have to repeat steps #1 and #2 each time (*Thanks to Michal Kučírka for pointing this out*).

Teacher Forcing

The approach of feeding the target sequence to the Decoder during training is known as Teacher Forcing. Why do we do this and what does that term mean?

During training, we could have used the same approach that is used during inference. In other words, run the Transformer in a loop, take the last word from the output sequence, append it to the Decoder input and feed it to the Decoder for the next iteration. Finally, when the end-of-sentence token is predicted, the Loss function would compare the generated output sequence to the target sequence in order to train the network.

Not only would this looping cause training to take much longer, but it also makes it harder to train the model. The model would have to predict the second word based on a potentially erroneous first predicted word, and so on.

Instead, by feeding the target sequence to the Decoder, we are giving it a hint, so to speak, just like a Teacher would. Even though it predicted an erroneous first word, it can instead use the correct first word to predict the second word so that those errors don't keep compounding.

In addition, the Transformer is able to output all the words in parallel without looping, which greatly speeds up training.

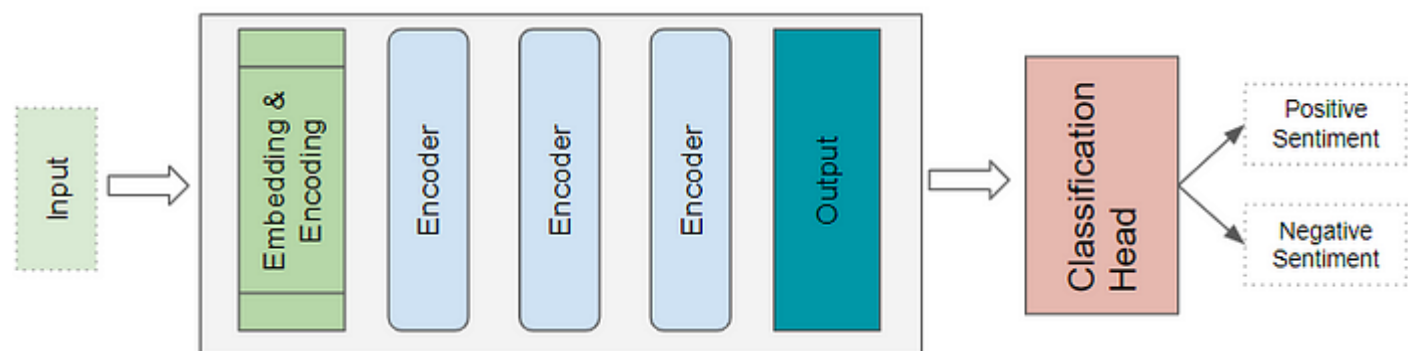
What are Transformers used for?

Transformers are very versatile and are used for most NLP tasks such as language models and text classification. They are frequently used in sequence-to-sequence models for applications such as Machine Translation, Text Summarization, Question-Answering, Named Entity Recognition, and Speech Recognition.

There are different flavors of the Transformer architecture for different problems. The basic Encoder Layer is used as a common building block for these architectures, with different application-specific 'heads' depending on the problem being solved.

Transformer Classification architecture

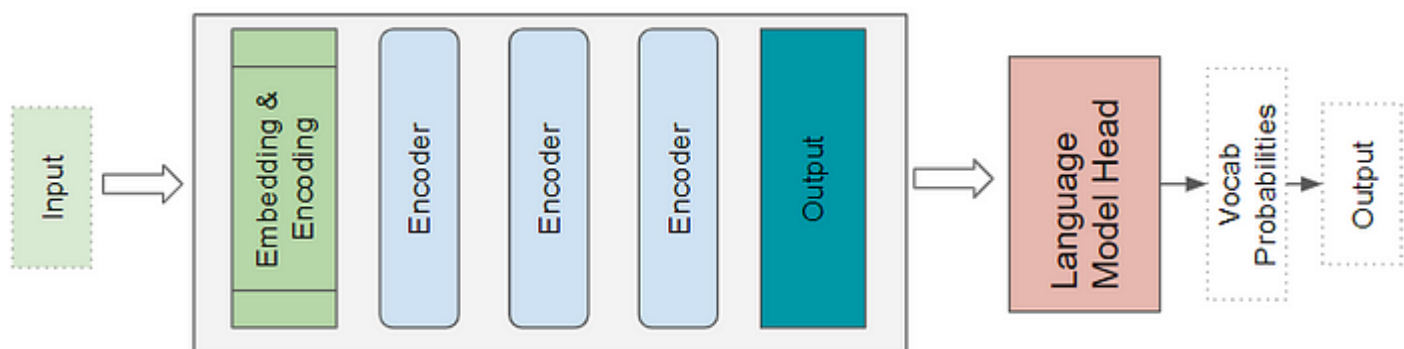
A Sentiment Analysis application, for instance, would take a text document as input. A Classification head takes the Transformer's output and generates predictions of the class labels such as a positive or negative sentiment.



(Image by Author)

Transformer Language Model architecture

A Language Model architecture would take the initial part of an input sequence such as a text sentence as input, and generate new text by predicting sentences that would follow. A Language Model head takes the Transformer's output and generates a probability for every word in the vocabulary. The highest probability word becomes the predicted output for the next word in the sentence.



(Image by Author)

How are they better than RNNs?

RNNs and their cousins, LSTMs and GRUs, were the de facto architecture for all NLP applications until Transformers came along and dethroned them.

RNN-based sequence-to-sequence models performed well, and when the Attention mechanism was first introduced, it was used to enhance their performance.

However, they had two limitations:

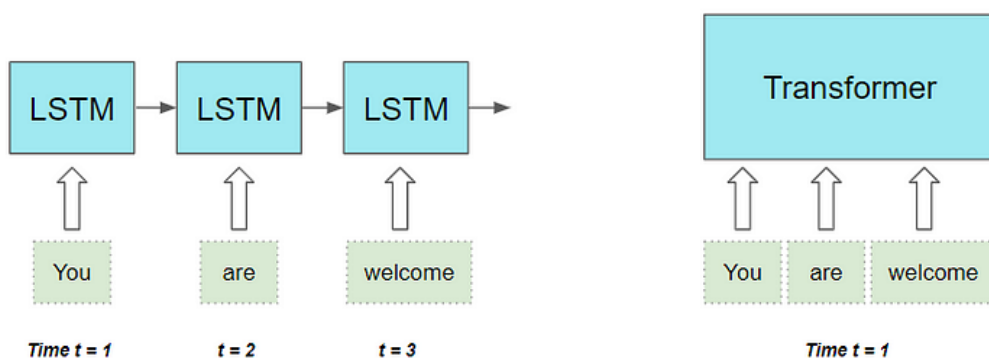
- It was challenging to deal with long-range dependencies between words that were spread far apart in a long sentence.
- They process the input sequence sequentially one word at a time, which means that it cannot do the computation for time-step t until it has completed the computation for time-step $t - 1$. This slows down training and inference.

As an aside, with CNNs, all of the outputs can be computed in parallel, which makes convolutions much faster. However, they also have limitations in dealing with long-range dependencies:

- In a convolutional layer, only parts of the image (or words if applied to text data) that are close enough to fit within the kernel size can interact with each other. For items that are further apart, you need a much deeper network with many layers.

The Transformer architecture addresses both of these limitations. It got rid of RNNs altogether and relied exclusively on the benefits of Attention.

- They process all the words in the sequence in parallel, thus greatly speeding up computation.



(Image by Author)

- The distance between words in the input sequence does not matter. It is equally good at computing dependencies between adjacent words and words that are far apart.

Now that we have a high-level idea of what a Transformer is, we can go deeper into its internal functionality in the [next](#) article to understand the details of how it works.

And finally, if you liked this article, you might also enjoy my other series on Audio Deep Learning, Geolocation Machine Learning, and Image Caption architectures.

BERT Explained: A Complete Guide with Theory and Tutorial

At the end of 2018 researchers at Google AI Language open-sourced a new technique for Natural Language Processing (NLP) called **BERT** (Bidirectional Encoder Representations from Transformers) — a major breakthrough which took the Deep Learning community by storm because of its incredible performance. Since BERT is likely to stay around for quite some time, in this blog post, we are going to understand it by attempting to answer these 5 questions:

1. *Why was BERT needed?*
2. *What is the core idea behind it?*
3. *How does it work?*
4. *When can we use it and how to fine-tune it?*
5. *How can we use it? Using BERT for Text Classification — Tutorial*

1. Why was BERT needed?

One of the biggest challenges in NLP is the lack of enough training data. Overall there is *enormous amount of text data available, but if we want to create task-specific datasets, we need to split that pile into the very many diverse fields*. And when we do this, we end up with only a few thousand or a few hundred thousand human-labeled training examples. Unfortunately, in order to perform well, deep learning based NLP models require much larger amounts of data — they see major improvements when trained on millions, or billions, of annotated training examples. To help bridge this gap in data, researchers have developed various techniques for training general purpose language representation models using the enormous piles of unannotated text on the web (this is known as *pre-training*). These general purpose pre-trained models can then be *fine-tuned* on smaller task-specific datasets, e.g., when working with problems like question answering and sentiment analysis. This approach results in great accuracy improvements compared to training on the smaller task-specific datasets from scratch. *BERT is a recent addition to these techniques for NLP pre-training; it caused a stir in the deep learning community because it presented state-of-the-art results in a wide variety of NLP tasks, like question answering.*

The best part about BERT is that it can be download and used for free — we can either use the BERT models to extract high quality language features from our text data, or we can fine-tune these models on a specific task, like sentiment analysis and question answering, with our own data to produce state-of-the-art predictions.

2. What is the core idea behind it?

What is language modeling really about? Which problem are language models trying to solve? Basically, their task is to “fill in the blank” based on context. For example, given

“The woman went to the store and bought a _____ of shoes.”

a language model might complete this sentence by saying that the word “cart” would fill the blank 20% of the time and the word “pair” 80% of the time.

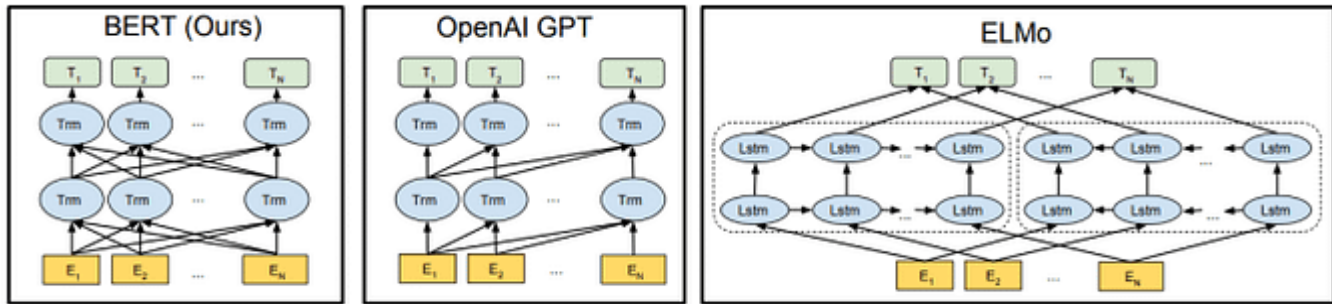
In the pre-BERT world, a language model would have looked at this text sequence during training from either left-to-right or combined left-to-right and right-to-left. This one-directional approach works well for generating sentences — we can predict the next word, append that to the sequence, then predict the next to next word until we have a complete sentence.

Now enters BERT, a language model which is **bidirectionally trained** (this is also its key technical innovation). This means we can now have a deeper sense of language context and flow compared to the single-direction language models.

Instead of predicting the next word in a sequence, BERT makes use of a novel technique called **Masked LM** (MLM): it randomly masks words in the sentence and then it tries to predict them. Masking means that the model looks in both directions and it uses the full context of the sentence, both left and right surroundings, in order to predict the masked word. Unlike the previous language models, it takes both the previous and next tokens into account at the **same time**. The existing combined left-to-right and right-to-left LSTM based models were missing this “same-time part”. (It might be more accurate to say that BERT is non-directional though.)

But why is this non-directional approach so powerful?

Pre-trained language representations can either be *context-free* or *context-based*. *Context-based* representations can then be *unidirectional* or *bidirectional*. Context-free models like word2vec generate a single word embedding representation (a vector of numbers) for each word in the vocabulary. For example, the word “bank” would have the same context-free representation in “bank account” and “bank of the river.” On the other hand, context-based models generate a representation of each word that is based on the other words in the sentence. For example, in the sentence “I accessed the bank account,” a unidirectional contextual model would represent “bank” based on “I accessed the” but not “account.” However, BERT represents “bank” using both its previous and next context — “I accessed the ... account” — starting from the very bottom of a deep neural network, making it deeply bidirectional.



Moreover, BERT is based on the Transformer model architecture, instead of LSTMs. We will very soon see the model details of BERT, but in general:

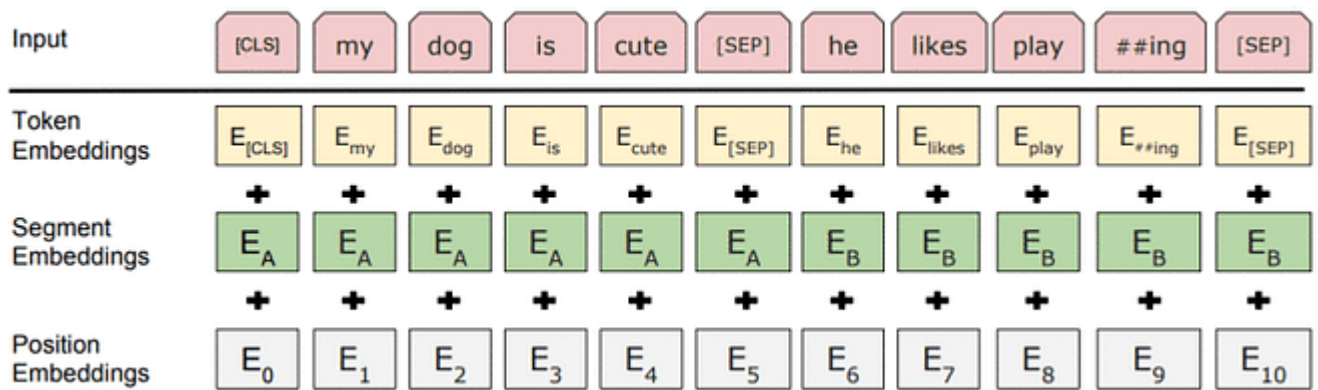
A Transformer works by performing a small, constant number of steps. In each step, it applies an attention mechanism to understand relationships between all words in a sentence, regardless of their respective position. For example, given the sentence, “I arrived at the bank after crossing the river”, to determine that the word “bank” refers to the shore of a river and not a financial institution, the Transformer can learn to immediately pay attention to the word “river” and make this decision in just one step.

Now that we understand the key idea of BERT, let’s dive into the details.

3. How does it work?

BERT relies on a Transformer (the attention mechanism that learns contextual relationships between words in a text). A basic Transformer consists of an encoder to read the text input and a decoder to produce a prediction for the task. Since BERT’s goal is to generate a language representation model, it only needs the encoder part. The input to the encoder for BERT is a sequence of tokens, which are first converted into vectors and then processed in the neural network. But before processing can start, BERT needs the input to be massaged and decorated with some extra metadata:

1. **Token embeddings:** A [CLS] token is added to the input word tokens at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. **Segment embeddings:** A marker indicating Sentence A or Sentence B is added to each token. This allows the encoder to distinguish between sentences.
3. **Positional embeddings:** A positional embedding is added to each token to indicate its position in the sentence.



Essentially, the Transformer stacks a layer that maps sequences to sequences, so the output is also a sequence of vectors with a 1:1 correspondence between input and output tokens at the same index. And as we learnt earlier, BERT does not try to predict the next word in the sentence. Training makes use of the following two strategies:

1. Masked LM (MLM)

The idea here is “simple”: Randomly mask out 15% of the words in the input — replacing them with a [MASK] token — run the entire sequence through the BERT attention based encoder and then predict only the masked words, based on the context provided by the other non-masked words in the sequence. However, there is a problem with this naive masking approach — the model only tries to predict when the [MASK] token is present in the input, while we want the model to try to predict the correct tokens regardless of what token is present in the input. To deal with this issue, out of the 15% of the tokens selected for masking:

- 80% of the tokens are actually replaced with the token [MASK].
- 10% of the time tokens are replaced with a random token.
- 10% of the time tokens are left unchanged.

While training the BERT loss function considers only the prediction of the masked tokens and ignores the prediction of the non-masked ones. This results in a model that converges much more slowly than left-to-right or right-to-left models.

2. Next Sentence Prediction (NSP)

In order to understand *relationship* between two sentences, BERT training process also uses next sentence prediction. A pre-trained model with this kind of understanding is relevant for tasks like question answering. During training the model gets as input pairs of sentences and it learns to predict if the second sentence is the next sentence in the original text as well.

As we have seen earlier, BERT separates sentences with a special [SEP] token. During training the model is fed with two input sentences at a time such that:

- 50% of the time the second sentence comes after the first one.
- 50% of the time it is a random sentence from the full corpus.

BERT is then required to predict whether the second sentence is random or not, with the assumption that the random sentence will be disconnected from the first sentence:

Input = [CLS] the man went to [MASK] store [SEP]
 he bought a gallon [MASK] milk [SEP]

Label = IsNext

Input = [CLS] the man [MASK] to the store [SEP]
 penguin [MASK] are flight ##less birds [SEP]

Label = NotNext

To predict if the second sentence is connected to the first one or not, basically the complete input sequence goes through the Transformer based model, the output of the [CLS] token is transformed into a 2×1 shaped vector using a simple classification layer, and the IsNext-Label is assigned using softmax.

The model is trained with both Masked LM and Next Sentence Prediction together. This is to minimize the combined loss function of the two strategies — “*together is better*”.

Architecture

There are four types of pre-trained versions of BERT depending on the scale of the model architecture:

BERT-Base: 12-layer, 768-hidden-nodes, 12-attention-heads, 110M parameters
BERT-Large: 24-layer, 1024-hidden-nodes, 16-attention-heads, 340M parameters

Fun fact: BERT-Base was trained on 4 cloud TPUs for 4 days and BERT-Large was trained on 16 TPUs for 4 days!

For details on the hyperparameter and more on the architecture and results breakdown, I recommend you to go through the original paper.

4. When can we use it and how to fine-tune it?

BERT outperformed the state-of-the-art across a wide variety of tasks under general language understanding like natural language inference, sentiment analysis, question answering, paraphrase detection and linguistic acceptability.

Now, how can we fine-tune it for a specific task? BERT can be used for a wide variety of language tasks. If we want to fine-tune the original model based on our own dataset, we can do so by just adding a single layer on top of the core model.

For example, say we are creating a **question answering application**. In essence question answering is just a prediction task — on receiving a question as input, the goal of the application is to identify the right answer from some corpus. So, given a question and a context paragraph, the model predicts a start and an end token from the paragraph that most likely answers the question. This means that using BERT a model for our application can be trained by learning two extra vectors that mark the beginning and the end of the answer.

- **Input Question:**

```
Where do water droplets collide with ice  
crystals to form precipitation?
```

- **Input Paragraph:**

```
... Precipitation forms as smaller droplets  
coalesce via collision with other rain drops  
or ice crystals within a cloud. ...
```

- **Output Answer:**

```
within a cloud
```

Just like sentence pair tasks, the question becomes the first sentence and paragraph the second sentence in the input sequence. However, this time there are two new parameters learned during fine-tuning: a **start vector** and an **end vector**.

In the fine-tuning training, most hyper-parameters stay the same as in BERT training; the paper gives specific guidance on the hyper-parameters that require tuning.

Note that in case we want to do fine-tuning, we need to transform our input into the specific format that was used for pre-training the core BERT models, e.g., we would need to add special tokens to mark the beginning ([CLS]) and separation/end of sentences ([SEP]) and segment IDs used to distinguish different sentences — convert the data into features that BERT uses.

Before we discuss how GPT-3 outsmarts GPT-2 let's take a look at the **similarities** between the two.

1. Both GPT-2 and GPT-3 are developed by **OpenAI** and are **open-source**, which allows researchers and developers to access the code and use it for their own projects.
2. Both are **pre-trained transformer-based neural network** models that are used for natural language processing (NLP) tasks.
3. Both models use a technique called **unsupervised learning** which allows them to *learn patterns in text data* without the need for labeled examples.
4. Both can **generate text** and **complete text prompts**, they can be fine-tuned for specific tasks and applied to various domains.
5. Both models use a technique called **attention mechanism** which allows them to focus on *specific parts of the input* when generating text.
6. Both can be accessed via the **OpenAI API** which allows developers to easily *use their capabilities* in their applications.
7. Both models are considered as *state of the art models* for language modeling and they can be used for a wide range of NLP tasks.
8. Both models use a technique called **masked language modeling** which is a type of unsupervised *pre-training* where a portion of the input text is masked and the model is trained *to predict the masked tokens*.
9. Both models use pre-training and fine-tuning techniques to improve performance, where **pre-training** is used to obtain a general *understanding* of the language and **fine-tuning** is used to *adapt* the model to specific tasks.
10. Both models can be used to **translate languages**.
11. Both models have the ability to generate *coherent and fluent text*.
12. Both can be fine-tuned for specific tasks and **applications such as** text generation, text completion, question answering, summarization, text classification etc.
13. Both models have the ability to handle *out-of-vocabulary words* and *grammatical errors*.

Differences between GPT-2 and GPT-3

GPT-3 is considered to be more advanced and capable than GPT-2 due to its larger model size, more diverse training data and ability to perform a wider range of language tasks.

1. **Model size:** GPT-3 is significantly larger than GPT-2, with *175 billion parameters* compared to GPT-2's *1.5 billion parameters*. This means that GPT-3 has the ability to learn more *complex relationships* between words and sentences.
2. **Training data:** GPT-3 was trained on a much larger dataset than GPT-2, consisting of *570GB* of text data compared to GPT-2's *40GB* of text data. This allows GPT-3 to have a more diverse set of knowledge and better generalization capabilities.
3. **Language tasks:** GPT-3 is capable of performing a wider range of language tasks than GPT-2 such as question answering, summarization, and text classification. It can also perform tasks that require common-sense reasoning and understanding of context.
4. **Fine-tuning:** GPT-3 can be fine-tuned on a *smaller dataset* than GPT-2, which makes it more accessible for practical applications.
5. **Accessibility:** GPT-3 is available *via the OpenAI API*, which allows developers to easily use its capabilities in their applications.
6. **Speed:** GPT-3 is *slower* than GPT-2 due to its larger model size and the complexity of the tasks it can perform.
7. **Cost:** GPT-3 is more expensive than GPT-2 as it requires more computational resources and is only available through the OpenAI API, which charges based on usage.
8. **Quality of output:** GPT-3 is considered to produce outputs of higher quality than GPT-2 as it can generate more human-like text and respond to prompts more accurately.
9. **Control over the output:** GPT-3 has **more control features** than GPT-2, for example, it can control the level of creativity, formality, coherence, and consistency of the output.
10. **Flexibility:** GPT-3 is *more versatile* than GPT-2, as it can be fine-tuned for a variety of tasks and applied to various domains.
11. **Language support:** GPT-3 supports *multiple languages*, while GPT-2 only supports English.
12. **Dependency on pre-training data:** GPT-3 is more dependent on the pre-training data than GPT-2, meaning that it could generate biased or false information if the pre-training data has errors or biases.
13. **Text generation:** GPT-3 is known for its ability to generate human-like text, and it can produce coherent, fluent, and contextually appropriate text. GPT-2 is also able to generate text but with a lower level of coherence and fluency.
14. **Text completion:** GPT-3 can complete text with higher accuracy and fluency than GPT-2 because it has a better understanding of the context and language patterns.
15. **Summarization:** GPT-3 can summarize text with higher accuracy and fluency than GPT-2 because it has a better understanding of the context and language patterns.

16. **Robustness:** GPT-3 is considered to be more robust than GPT-2 as it can handle *a wide range of inputs*, including out-of-vocabulary words, misspellings, and grammatical errors.
17. **Explainability:** GPT-3 is considered to be less explainable than GPT-2, *as it is a black-box model* and it's hard to understand how it generates its outputs.

Evolution of NLP — Part 4 — Transformers — BERT, XLNet, RoBERTa

Issues with LSTMs

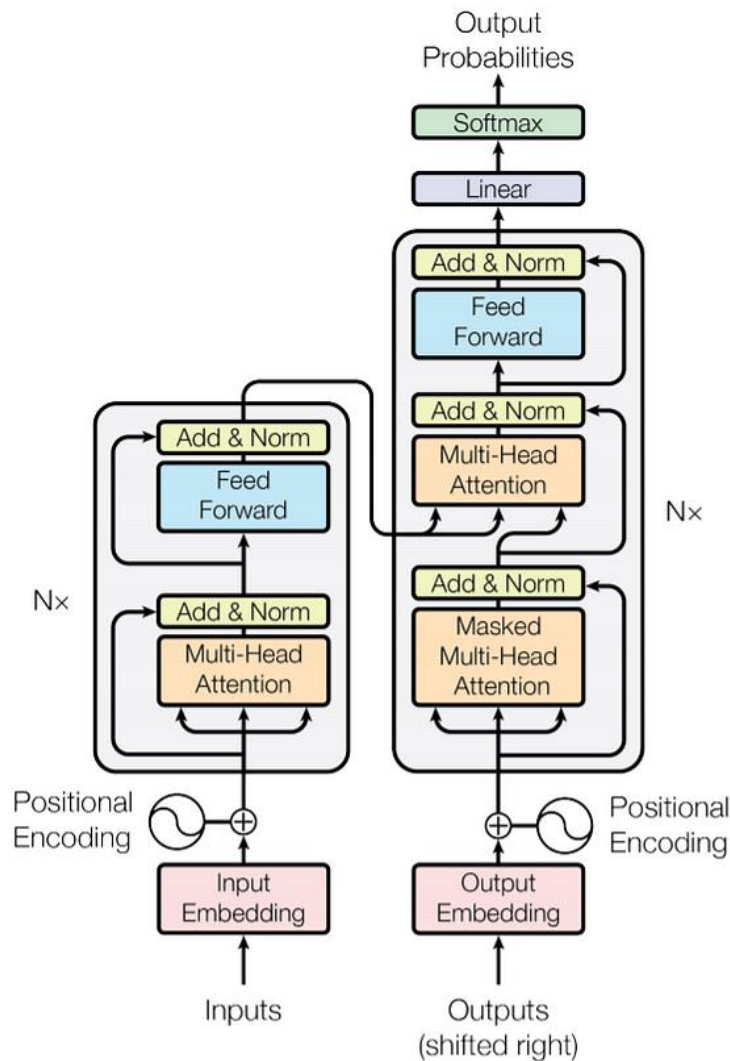
LSTMs have been very critical in the evolution of NLP. These were some of the most pivotal architectures that resolved issues within RNNs and made deep learning more wide-spread. But they pose some issues which make them difficult to work with.

1. **Slow to train!** RNNs are already slower to train because we need to provide data **sequentially** while training, instead of **parallelly**. A hidden state would require inputs from all the previous words to make any progress. This kind of architecture doesn't take advantage of today's GPUs which are optimized for parallel processing. Couple this with the added complexity in an LSTM, in the form of multiple gates, they are even slower to train.
2. **Better Contextual Awareness!** Normal LSTMs process the words only in one direction, and this limits the contextual awareness of the network. Even the Bi-Directional LSTMs learn the context in forward & backward direction and concatenate them, while a better approach would be to look at both the words ahead and back together!
3. **Long Sequences!** We already know that LSTMs can perform better than RNNs due to their gates, but even with this, the improvement in the case of LSTMs is not significant when we are trying tasks that crunch a large number of sentences, for example — Text Summarization and Questions & Answers.

With **Transformers**, let's see how we can address these issues.

Transformers

This architecture was first show-cased in Attention is All You Need! (2017) paper. The complete architecture can be split into two parts — Encoder & Decoder. In the attached image, the structure on the left is Encoder and on the right is Decoder



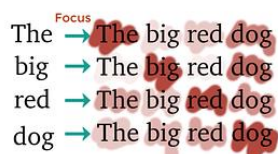
Encoder-Decoder architecture from Attention is All You Need! (2017) Paper

1. Encoder

In a nutshell, an Encoder tries to understand the context of an input sentence, and in-process learn what the language is! Let's see how data moves in an encoder.

- The input to an Encoder is a sentence — say The big red dog. These individual words are first tokenized and replaced by pre-trained word embeddings in the **Input Embeddings** layer.
- In the next step, we add a **Position Encoding** to individual word vectors. Essentially a function that maps to the position of the word in a sentence. We add this to our initial input embeddings to make sure that apart from the word, we also have a position aspect of it embedded in, while we pass the sentence into the encoder.
- Next, we pass this to a **Self-Attention Layer**. The idea here to create a matrix of importance each word has while defining context for other words in the same sentence.

Each word would obviously have significant importance with itself, and then with other words.



The darker the red, the higher the attention each individual word has with all the other words.

— Image from Transformers Explained —
<https://www.youtube.com/watch?v=TQQIZhbC5ps>

- And, finally, we pass it through a **Feed-Forward** layer, which is basically a dense neural network layer.

The biggest advantage of this network is that we don't need to pass the data sequentially but in parallel! This greatly improves the training time. And since, we are looking at the words in an overall manner, without any directional sense, the overall understanding of language or context is much better than other architectures.

To understand BERT and XLNet, you only need to know Encoders. So, if that is your goal, please jump to the section on BERT. I'll give explaining the Decoders a shot next.

2. Decoder

In a nutshell, a Decoder takes the output sentence as the input and tries to output the next word. Visualize the whole Transformer architecture as a Language Translation Model — input as “The big red dog” in English and output as “Le Gros Chien rogue” in French. Now let's try to see how this output flows in a decoder network.

- The first step is similar to Encoder. Embedding, coupled with Positional Encoding for each output word.
- Next, we compute **Masked Self-Attention**. This is slightly different from before, as here we intentionally mask the words that are yet to come. Essentially we don't provide the words that are supposed to come next while calculating the importance. Now, this is interesting because this is different from what we did in Encoder. Intuitively, while predicting the next French word, we can use the context of all English words, but we can't use the **next** french word, because then the model would just output that french word. I know it's a little confusing, but it'll become clear as we see how these models are used in practice.
- Then, the input from the Decoder network is looked at alongside Encoder's input. Here, we try to establish the context or relationship between English and French words together, but again with masking for the French words

- Finally, as the word travels through the Feed-Forward and Linear layer, we reach the Softmax. The softmax is essentially on all the words in Vocabulary, which gives us a probability score for all the words in the dictionary for the next word. The highest probability word is chosen for each position.

Do note that, again this process happens in parallel, similar to Encoder.

I hope this gives you some insight into the architecture of Transformers, and how they can be an improvement over LSTMs. Let's try and look at two such architectures & their implementation in detail.

BERT

Bert stands for **Bi-directional Encoder Representation from Transformers**. As the name implies, this architecture uses the Encoder part of the Transformers network, but the difference is that multiple encoder networks are stacked one after the other. Another important aspect that makes BERT stand out is its training methodology. Let's try to understand that.

1. Pre-Training

This is the phase where the model understands what is language and context. For this part, BERT uses two simultaneous tasks to train —

- **Masked Language Modeling** — Intuitively, this is like a “fill in the blanks” learning task. The model randomly masks some portion of sentences and its job is to predict those masked words. For example — With input being, “The *[MASK]* brown fox *[MASK]* over the lazy dog.”, the output, would be [“quick”, “jumped”].
- **Next Sentence Prediction** — Here, BERT takes in two sentences and determines if the second sentence follows the first one, basically like a Classification Task.

In a nutshell, these two tasks give BERT an understanding of language and context both within a sentence and over multiple sentences.

2. Fine-Tuning

Now the model basically understands what the language and context are, next is to train it for our specific task. In our case, this would be Sentiment Classification. We simply add Dense layers at the end of the network to get output, as per our specific task.

Let's start with implementing BERT. We'll look at the familiar fast.ai construct, however since these models are not directly available in fast.ai, we'll have to create classes for Tokenizers, Vocabulary, and Numericalizers for fast.ai. This is exceptionally confusing, especially if one is not familiar with these libraries.

Note that the implementation below is inspired by [this excellent Kernel Tutorial](#) and [Medium Blog](#) by the same author. This not only allows you to implement transformers quickly, but also gives tremendously flexibility in trying out other powerful architectures.

That said, I'll try to share my interpretation of this tutorial below. Feel free to check out [this kernel](#) for better clarity. There are other tutorials regarding the same, which I'll link below, but if you want to settle on a general solution to load multiple types of Transformers Models, I would recommend reading [this implementation](#).

Apart from the **fast.ai** library, we learned about in the last tutorial, here we'll additionally use the **HuggingFace Transformers** library. This library has almost all the major SOTA NLP models, with specific models for NLP tasks, like Question-Answers, Text Summarization, Masked LM, etc. We'll be using the **Sequence Classification** models here, but feel free to try out models for other tasks.

Now, to run any model using HuggingFace library requires you to load 3 components —

1. **Model Class** — This will help us load the architecture and pre-trained weights for our specific model.
2. **Tokenizer Class** — This will help pre-process the data into tokens. Additionally, the padding, start and ending of sentences, and handling missing words in vocabulary is somewhat different across different models.
3. **Config Class** — This is the configuration class to store the configuration of the chosen model. It is used to instantiate the model according to the specified arguments, defining the model architecture.

Note that, we need to load these three classes for the **same model**. For example, in our initial trial, if I'm running BERT, we load **BertForSequenceClassification** for the model class, **BertTokenizer** for tokenizer class, and **BertConfig** for config class.

The next step is Tokenization!

Tokenization

Note that BERT has its own vocabulary and tokenizer. Hence, we need to create a wrapper around the BERT's internal implementation, so that it is compatible with fast.ai's implementation. This can be done in 3 steps shown below.

1. First, we create a tokenizer object, by loading the default tokenizer for our specific model.
`transformer_tokenizer = tokenizer_class.from_pretrained(pretrained_model_name)`

```
[26]: transformer_tokenizer.tokenize("Hey Abram, your dog is cute ", return_tensors="pt")
```

```
Out[26]: ['hey', 'ab', '##ram', ',', 'your', 'dog', 'is', 'cute']
```

Tokenizer output — Image from Author

2. We then create our own **BertBaseTokenizer** Class, where we update the **tokenizer** function, incorporating functions that help process our specific set of transformers models.

```
class BertBaseTokenizer(BaseTokenizer):
    """Wrapper around PreTrainedTokenizer to be compatible with fast.ai"""
    def __init__(self, pretrained_tokenizer: PreTrainedTokenizer, model_type = 'bert', **kwargs):
        self._pretrained_tokenizer = pretrained_tokenizer
        self.max_seq_len = pretrained_tokenizer.max_len
        self.model_type = model_type
    def __call__(self, *args, **kwargs):
        return self.tokenize(self, t:str) -> List[str]:
    """Limits the maximum sequence length and add the special tokens"""
    CLS = self._pretrained_tokenizer.cls_token
    SEP = self._pretrained_tokenizer.sep_token
    tokens = self._pretrained_tokenizer.tokenize(t)[:self.max_seq_len - 2]
    tokens = [CLS] + tokens + [SEP]
    return tokens
```

And we initialize our base tokenizer using this —

```
bert_base_tokenizer = BertBaseTokenizer(pretrained_tokenizer = transformer_tokenizer,
model_type = model_type)
```

3. We are not done yet, and this is the most confusing part. We pass our **bert_base_tokenizer** into the **Tokenizer** function which is then processed by fast.ai. This additional step is important so do make sure you do this in your implementation as well.

```
fastai_tokenizer = Tokenizer(tok_func = bert_base_tokenizer)
```

4. To use this while loading data, it is recommended to convert this into a TokenizerProcessor. We can call this during DataBunch call, as we had seen in the earlier tutorial

```
tokenize_processor = TokenizeProcessor(tokenizer=fastai_tokenizer, include_bos=False,
include_eos=False)
```

Numericalization

This step involves encoding the tokens into numerical encodings. Once again, each transformer model has its own version of Vocabulary, and hence, it's very own encoding. Let's load them as well into the familiar fast.ai framework.

1. First, let's create our version of **Vocab** library in fast.ai, updating its functions — **numericalize** (converts tokens to encodings) and **textify** (converts encodings to tokens). In the definitions of these functions, we use — **convert_tokens_to_ids** and **convert_ids_to_tokens** functions respectively that work with HuggingFace's pre-trained transformer models.

11:

```
class TransformersVocab(Vocab):
    def __init__(self, tokenizer: PreTrainedTokenizer):
        super(TransformersVocab, self).__init__(itos = [])
        self.tokenizer = tokenizer

    def numericalize(self, t:Collection[str]) -> List[int]:
        "Convert a list of tokens 't' to their ids."
        return self.tokenizer.convert_tokens_to_ids(t)
        #return self.tokenizer.encode(t)

    def textify(self, nums:Collection[int], sep=' ') -> List[str]:
        "Convert a list of 'nums' to their tokens."
        nums = np.array(nums).tolist()
        return sep.join(self.tokenizer.convert_ids_to_tokens(nums)) if sep is not None else self.tokenizer.convert

    def __getstate__(self):
        return {'itos':self.itos, 'tokenizer':self.tokenizer}

    def __setstate__(self, state:dict):
        self.itos = state['itos']
        self.tokenizer = state['tokenizer']
        self.stoi = collections.defaultdict(int, {v:k for k,v in enumerate(self.itos)})
```

TransformersVocab class definition — Image from Author

2. Finally, we pass this into a Numericalize Processor class, similar to Tokenize Processor, which we will call during our DataBunch creation.

```
transformer_vocab = TransformersVocab(tokenizer = transformer_tokenizer)
numericalize_processor = NumericalizeProcessor(vocab=transformer_vocab)
```

Loading the Data

Next, we load the data using, DataBlock API.

```
databunch = (TextList.from_df(train, cols='user_review', processor=transformer_processor)
              .split_by_rand_pct(0.1,seed=seed)
              .label_from_df(cols=                                'user_suggestion')
              .add_test(test)
              .databunch(bs=bs, pad_first=pad_first, pad_idx=pad_idx))
```

Modeling

Before we start the modeling process, we create our own model using the pre-trained model as the base, only extracting the logits needed for our specific prediction.

```

11: # Create our own Classification Model
class ClssificationModel(nn.Module):
    def __init__(self, transformer_model: PreTrainedModel):
        super(ClssificationModel, self).__init__()
        self.transformer = transformer_model

    def forward(self, input_ids, attention_mask=None):
        attention_mask = (input_ids!=pad_idx).type(input_ids.type())
        logits = self.transformer(input_ids, attention_mask = attention_mask)[0]
        return logits

```

Custom Classification Model — Image from Author

And finally, initialize the **AdamW** optimizer that comes packed with HuggingFace Transformers library.

```

11: from fastai.callbacks import *
from transformers import AdamW
from functools import partial

CustomAdamW = partial(AdamW, correct_bias=False)

learner = Learner(databunch,
                  custom_transformer_model,
                  opt_func = CustomAdamW,
                  metrics=[accuracy, error_rate])

# Show graph of learner stats and metrics after each epoch.
learner.callbacks.append>ShowGraph(learner))

```

+ Code

+ Markdown

Learner — Image from Author

After this, we can directly run the model, but it might be helpful to introduce another helpful tool while using the fast.ai library.

Gradual Unfreezing

In our previous tutorials, we covered Discriminative Fine Tuning and Slated Triangular Learning Rate. Gradual Unfreezing is also something you can explore while making models.

The idea is quite simple — most of the earlier layers of models like BERT are for the understanding of language — and just like Pre-Trained CNN models, they can be left untouched, and simply tuning the last few layers can lead to good results, pretty quickly.

For models available within fast.ai, like AWD-LSTM, the layers are already present in groups, which can be put as un-trainable. For BERT, we'll have to create the groups on our own.

```
list_layers = [learner.model.transformer.bert.embeddings,
               learner.model.transformer.bert.encoder.layer[0],
               learner.model.transformer.bert.encoder.layer[1],
               learner.model.transformer.bert.encoder.layer[2],
               learner.model.transformer.bert.encoder.layer[3],
               learner.model.transformer.bert.encoder.layer[4],
               learner.model.transformer.bert.encoder.layer[5],
               learner.model.transformer.bert.encoder.layer[6],
               learner.model.transformer.bert.encoder.layer[7],
               learner.model.transformer.bert.encoder.layer[8],
               learner.model.transformer.bert.encoder.layer[9],
               learner.model.transformer.bert.encoder.layer[10],
               learner.model.transformer.bert.encoder.layer[11],
               learner.model.transformer.bert.pooler]
```

These layers can be split into individual groups -

```
learner.split(list_layers)
```

Now, to freeze all the layers, except for the last one, we use -

```
learner.freeze_to(-1)
```

And, let the model train for 1 epoch.

After this, we sequentially unfreeze another layer —

```
learner.freeze_to(-2)
```

And, let the model train for another epoch.

And finally, we unfreeze all the layers and let it run for 5 epochs.

Accuracy — 92%

This gave you a glimpse of using BERT for Sentiment Analysis. But, the HuggingFace Library is not limited to BERT. There have been several new models, which have come up and shown improvements over BERT.

XLNet

XLNet shot up to fame after it beat BERT in roughly 20 NLP tasks, sometimes with quite substantial margins. So, what is XLNet and how is it different from BERT? XLNet has a similar architecture to BERT. However, the major difference comes in it's approach to pre-training.

- BERT is an Autoencoding (AE) based model, while XLNet is an Auto-Regressive (AR). This difference materializes in the MLM task, where randomly masked language tokens are to be predicted by the model. To better understand the difference, let's consider a concrete example [New, York, is, a, city].
- Suppose both BERT and XLNet select the two tokens [New, York] as the prediction targets and maximize $\log(\text{New, York} \mid \text{is, a, city})$. Also suppose that XLNet samples the factorization order [is, a, city, New, York]. In this case, BERT and XLNet respectively reduce to the following objective functions:

$J\{\text{BERT}\} = \log(\text{New} \mid \text{is, a, city}) + \log(\text{York} \mid \text{is, a, city})$ and

$J\{\text{XLNet}\} = \log(\text{New} \mid \text{is, a, city}) + \log(\text{York} \mid \text{New, is, a, city})$

- Notice that XLNet is able to capture the dependency between the pair (New, York), which is omitted by BERT. Although in this example, BERT learns some dependency pairs such as (New, city) and (York, city), it is obvious that XLNet always learns more dependency pairs given the same target and contains “denser” effective training signals.

Let's see for our specific task if XLNet offers any significant improvement.

Overall, the implementation is fairly similar, with minor changes in padding, starting, and ending of the sentences. You can study the code at this kernel

Accuracy — 93%

RoBERTa

Robustly optimized BERT approach — RoBERTa, is a retraining of BERT with improved training methodology, 1000% more data, and compute power.

To improve the training procedure, RoBERTa removes the Next Sentence Prediction (NSP) task from BERT's pre-training and introduces dynamic masking so that the masked token changes during the training epochs. Larger batch-training sizes were also found to be more useful in the training procedure.

Importantly, RoBERTa uses 160 GB of text for pre-training, including 16GB of Books Corpus and English Wikipedia used in BERT.

Overall, the implementation is fairly similar, with minor changes in padding, starting, and ending of the sentences. You can study the code at this kernel.

Accuracy — 94%

With a few lines of code, we were able to implement and study SOTA Transformer Models. I hope this series served as a good starter for you to learn these techniques.

With these new innovations, the possibilities of what could be done in NLP have changed significantly over the last few years. These Transformer models are on their way to potentially replace all the existing LSTM/RNN based models! And they are not slowing down — just recently — OpenAI's GPT-3 was released, which is also based on Transformer architecture, but built using Decoders. It's important to stay up-to-date with these new models/architectures, and continue learning along the way!

See you in the next blog! Happy Learning :)

	BERT	RoBERTa	DistilBERT	XLNet
Size (millions)	Base: 110 Large: 340	Base: 110 Large: 340	Base: 66	Base: ~110 Large: ~340
Training Time	Base: 8 x V100 x 12 days* Large: 64 TPU Chips x 4 days (or 280 x V100 x 1 days*)	Large: 1024 x V100 x 1 day; 4-5 times more than BERT.	Base: 8 x V100 x 3.5 days; 4 times less than BERT.	Large: 512 TPU Chips x 2.5 days; 5 times more than BERT.
Performance	Outperforms state-of-the-art in Oct 2018	2-20% improvement over BERT	3% degradation from BERT	2-15% improvement over BERT
Data	16 GB BERT data (Books Corpus + Wikipedia). 3.3 Billion words.	160 GB (16 GB BERT data + 144 GB additional)	16 GB BERT data. 3.3 Billion words.	Base: 16 GB BERT data Large: 113 GB (16 GB BERT data + 97 GB additional). 33 Billion words.
Method	BERT (Bidirectional Transformer with MLM and NSP)	BERT without NSP**	BERT Distillation	Bidirectional Transformer with Permutation based modeling