NATURAL LANGUAGE PROCESSING

4.1 Deep Learning for Named entity recognition

Named-entity recognition (NER) (also known as **entity identification**, **entity chunking** and **entity extraction**) is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as **person names, organisations, locations, medical codes, time expressions, quantities, monetary values, percentages**, etc.

It adds a wealth of semantic knowledge to your content and helps you to promptly understand the subject of any given text.

The named entity recognition (NER) is one of the most popular data preprocessing task. It involves the identification of key information in the text and classification into a set of predefined categories. An entity is basically the thing that is consistently talked about or refer to in the text.

NER is the form of NLP.

At its core, NLP is just a two-step process, below are the two steps that are involved:

- Detecting the entities from the text
- Classifying them into different categories

Applications :

Few applications of **NER** include: extracting important named entities from **legal, financial, and medical documents,** classifying content for **news providers,** improving the **search algorithms**, and etc.

Ambiguity in NE

For a person, the category definition is intuitively quite clear, but for computers, there is some ambiguity in classification. Let's look at some ambiguous example:

- *England (Organisation)* won the 2019 world cup vs The 2019 world cup happened in *England(Location).*
- *Washington(Location)* is the capital of the US vs The first president of the US was *Washington(Person).*

DIFFERENT NER SYSTEMS:

1. Dictionary-based systems

This is the simplest NER approach. Here we will be having a dictionary that contains a collection of vocabulary. In this approach, basic string matching algorithms are used to check whether the entity is occurring in the given text to the items in vocabulary. The method has limitations as it is required to update and maintain the dictionary used for the system.

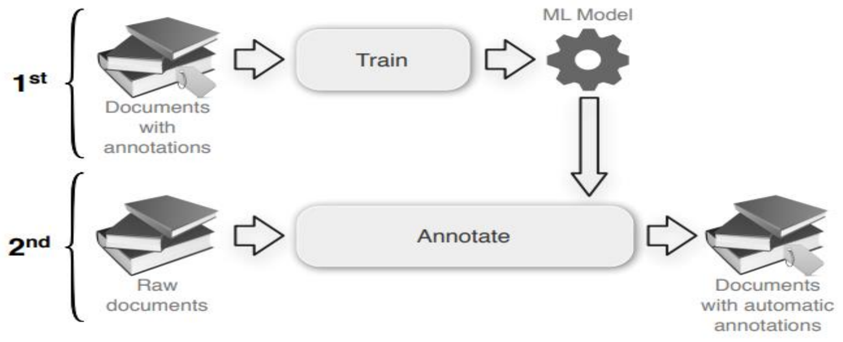
2. Rule-based systems

Here, the model uses a pre-defined set of rules for information extraction. Mainly two types of rules are used, Pattern-based rules, which depend upon the morphological pattern of the words used, and context-based rules, which depend upon the context of the word used in the given text document. A simple example for a context-based rule is "If a person's title is followed by a proper noun, then that proper noun is the name of a person".

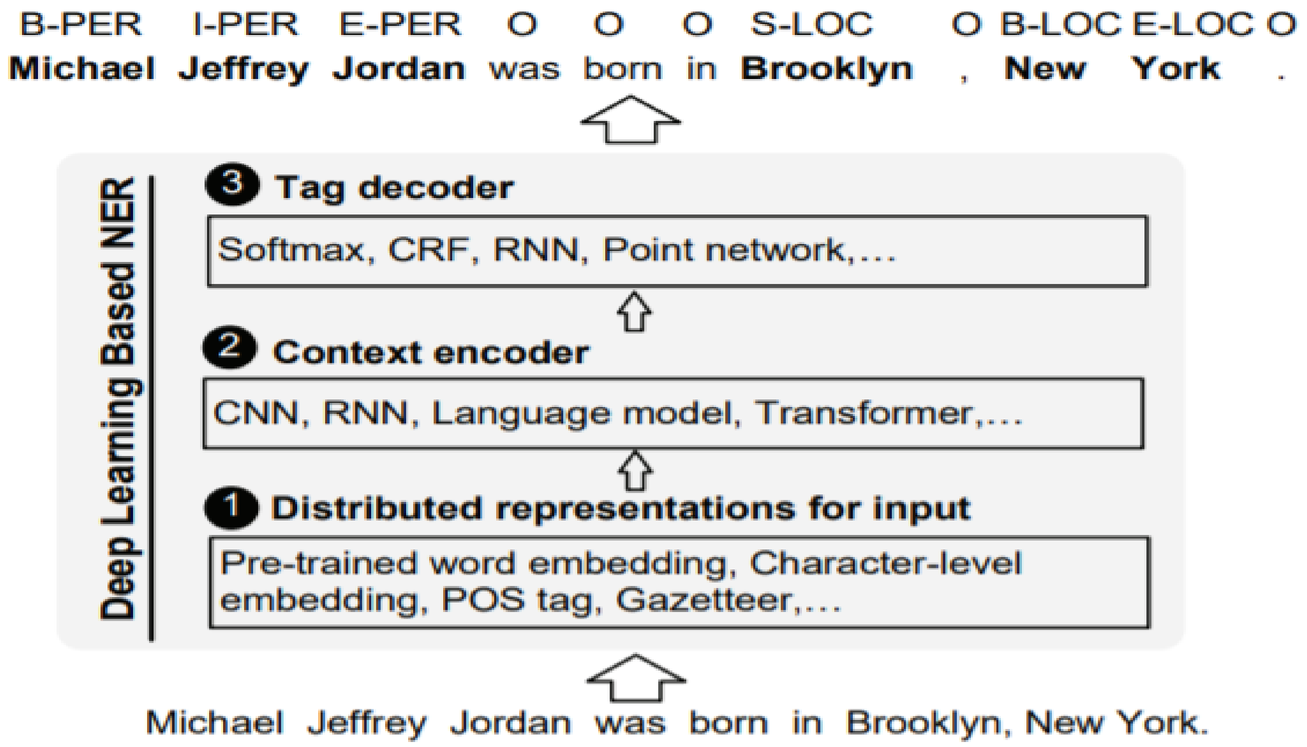
3. Machine learning-based systems

The ML-based systems use statistical-based models for detecting the entity names. These models try to make a feature-based representation of the observed data. By this approach, a lot of limitations of dictionary and rule-based approaches are solved by recognizing an existing entity name even with small spelling variations.

There are mainly two phases while we use an ML-based solution for NER. The first phase involves training the ML model on the annotated documents. The time taken for the model to train will vary depending upon the complexity of the model that we are building. In the next phase, the trained model can be used to annotate the raw documents.


1st
Documents with annotations
Train
ML Model
2nd
Raw documents
Annotate
Documents with automatic annotations

4. Deep Learning approaches


B-PER I-PER E-PER O O O S-LOC O B-LOC E-LOC O
Michael Jeffrey Jordan was born in Brooklyn , New York .
Deep Learning Based NER
3 Tag decoder
Softmax, CRF, RNN, Point network,...
2 Context encoder
CNN, RNN, Language model, Transformer,...
1 Distributed representations for input
Pre-trained word embedding, Character-level embedding, POS tag, Gazetteer,...
Michael Jeffrey Jordan was born in Brooklyn, New York.

In recent years, deep learning-based models are being used for building state-of-the-art systems for NER. There are many advantages of using DL techniques over the previously discussed approaches. Using the DL approach, the input data is mapped to a non-linear representation. This approach helps to learn complex relations that are present in the input data. Another advantage is that we can avoid a lot of time and resources spent on feature engineering which is required for the other traditional approaches.

Next, we will try out some tools for NER extraction.

Standford NER Tagger

It is one of the standard tools that is used for Named Entity Recognition. Mainly there are three types of models for identifying the named entities. They are:

1. Three class model which recognizes the organizations, persons, and locations.

2. Four class model which recognizes persons, organizations, locations, and miscellaneous entities.

3. Seven class model which recognizes persons, organizations, locations, money, time, percents, and dates.

We will be using the four-class model.

1. Downloading the StanfordNER zip file using the following commands

```
!pip3 install nltk==3.2.4
!wget http://nlp.stanford.edu/software/stanford-ner-2015-04-20.zip
!unzip stanford-ner-2015-04-20.zip
```

2. Loading the model

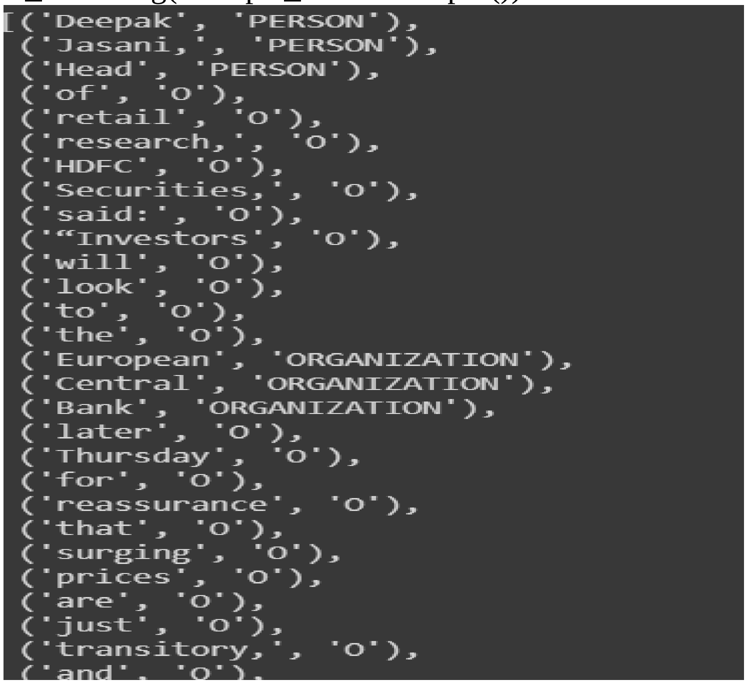
```
from nltk.tag.stanford import StanfordNERTagger
jar = "stanford-ner-2015-04-20/stanford-ner-3.5.2.jar"
model = "stanford-ner-2015-04-20/classifiers/"
st_4class = StanfordNERTagger(model + "english.conll.4class.distsim.crf.ser.gz", jar, encoding='utf8')
```

3. While testing the model, I had taken a news extract from the Indian Express newspaper

```
example_document = '''Deepak Jasani, Head of retail research, HDFC Securities, said: “Investors will look to the European Central Bank later Thursday for reassurance that surging prices are just transitory, and not about to spiral out of control. In addition to the ECB policy meeting, investors are awaiting a report later Thursday on US economic growth, which is likely to show a cooling recovery, as well as weekly jobs data.”.'''
```

4. Providing the news article to the model

```
st_4class.tag(example_document.split())
```



```
[('Deepak', 'PERSON'),
 ('Jasani,', 'PERSON'),
 ('Head', 'PERSON'),
 ('of', 'O'),
 ('retail', 'O'),
 ('research,', 'O'),
 ('HDFC', 'O'),
 ('Securities,', 'O'),
 ('said:', 'O'),
 ('“Investors', 'O'),
 ('will', 'O'),
 ('look', 'O'),
 ('to', 'O'),
 ('the', 'O'),
 ('European', 'ORGANIZATION'),
 ('Central', 'ORGANIZATION'),
 ('Bank', 'ORGANIZATION'),
 ('later', 'O'),
 ('Thursday', 'O'),
 ('for', 'O'),
 ('reassurance', 'O'),
 ('that', 'O'),
 ('surging', 'O'),
 ('prices', 'O'),
 ('are', 'O'),
 ('just', 'O'),
 ('transitory,', 'O'),
 ('and', 'O'),
```

Spacy Pipelines for NER

Spacy has mainly three English pipelines that are optimized for CPU for Named Entity Recognition. They are

a) en_core_web_sm

b) en_core_web_md

c) en_core_web_lg

The above models are listed in ascending order according to their size where SM, MD, and LG denote small, medium, and large models respectively. Let us try out NER using the small model.

1. First let us download the model

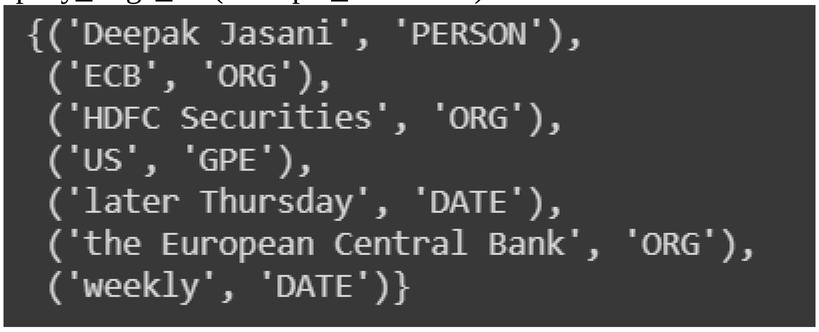
```
import spacy
 import spacy.cli
spacy.cli.download("en_core_web_sm")
```

2. Loading the model

```
sp_sm = spacy.load('en_core_web_sm')
```

3. Creating a function to output the recognized entities by the model.

```
def spacy_large_ner(document):
  return {(ent.text.strip(), ent.label_) for ent in sp_lg(document).ents}
spacy_large_ner(example_document)
```



```
{('Deepak Jasani', 'PERSON'),
 ('ECB', 'ORG'),
 ('HDFC Securities', 'ORG'),
 ('US', 'GPE'),
 ('later Thursday', 'DATE'),
 ('the European Central Bank', 'ORG'),
 ('weekly', 'DATE')}
```

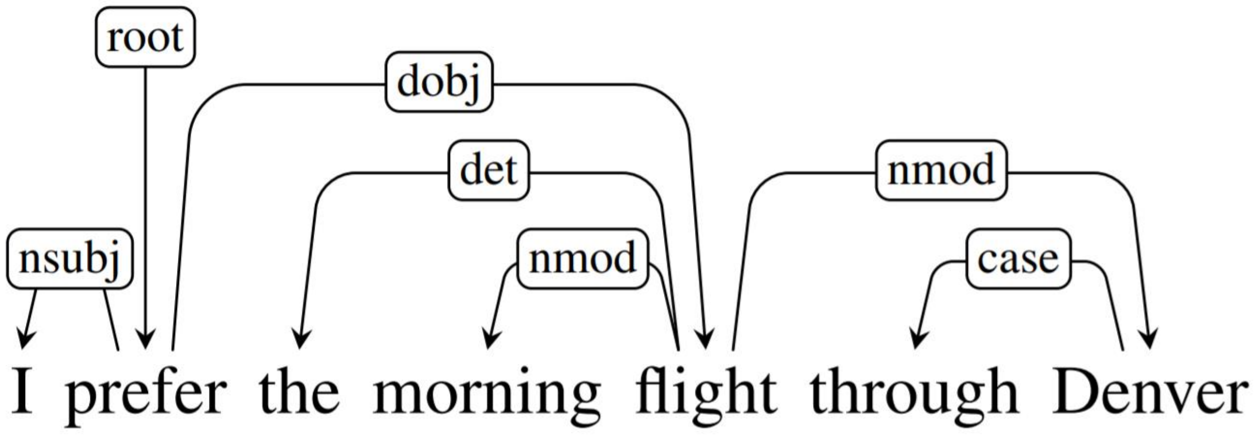
Here GPE means Geopolitical Entity.

4.1.1Dependency Parsing

The term Dependency Parsing (DP) refers to the process of examining the dependencies between the phrases of a sentence in order to determine its grammatical structure. A sentence is divided into many sections based mostly on this. The process is based on the assumption that there is a direct relationship between each linguistic unit in a sentence. These hyperlinks are called dependencies.

Consider the following statement: “I prefer the morning flight through Denver.”

The diagram below explains the sentence’s dependence structure:



In a written dependency structure, the relationships between each linguistic unit, or phrase, in the sentence are expressed by directed arcs. The root of the tree “prefer” varies the pinnacle of the preceding sentence, as labelled within the illustration.

A dependence tag indicates the relationship between two phrases. For example, the word “flight” changes the meaning of the noun “Denver.” As a result, you may identify a dependence from

flight -> Denver, where flight is the pinnacle and Denver is the kid or dependent. It’s represented by nmod, which stands for the nominal modifier.

This distinguishes the scenario for dependency between the two phrases, where one serves as the pinnacle and the other as the dependent. Currently, the Common Dependency V2 taxonomy consists of 37 common syntactic relationships, as shown in the table below:

Dependency Tag	**Description**
acl	clausal modifier of a noun (adnominal clause)
acl:relcl	relative clause modifier
advcl	adverbial clause modifier
advmod	adverbial modifier
advmod:emph	emphasizing phrase, intensifier
advmod:lmod	locative adverbial modifier

amod	adjectival modifier
appos	appositional modifier
aux	auxiliary
aux:move	passive auxiliary
case	case-marking
cc	coordinating conjunction
cc:preconj	preconjunct
ccomp	clausal complement
clf	classifier
compound	compound
compound:lvc	gentle verb building
compound:prt	phrasal verb particle
compound:redup	reduplicated compounds
compound:svc	serial verb compounds
conj	conjunct
cop	copula
csubj	clausal topic
csubj:move	clausal passive topic
dep	unspecified dependency
det	determiner
det:numgov	pronominal quantifier governing the case of the noun
det:nummod	pronominal quantifier agreeing with the case of the noun
det:poss	possessive determiner
discourse	discourse ingredient
dislocated	dislocated parts
expl	expletive
expl:impers	impersonal expletive
expl:move	reflexive pronoun utilized in reflexive passive
expl:pv	reflexive clitic with an inherently reflexive verb
mounted	mounted multiword expression
flat	flat multiword expression
flat:overseas	overseas phrases
flat:title	names

goeswith	goes with
iobj	oblique object
checklist	checklist
mark	marker
nmod	nominal modifier
nmod:poss	possessive nominal modifier
nmod:tmod	temporal modifier
nsubj	nominal topic
nsubj:move	passive nominal topic
nummod	numeric modifier
nummod:gov	numeric modifier governing the case of the noun
obj	object
obl	indirect nominal
obl:agent	agent modifier
obl:arg	indirect argument
obl:lmod	locative modifier
obl:tmod	temporal modifier
orphan	orphan
parataxis	parataxis
punct	punctuation
reparandum	overridden disfluency
root	root
vocative	vocative
xcomp	open clausal complement

Dependency Parsing using NLTK

The Pure Language Toolkit (NLTK) package deal will be used for Dependency Parsing, which is a set of libraries and codes used during statistical Pure Language Processing (NLP) of human language.

We may use NLTK to do dependency parsing in one of several ways:

1. Probabilistic, projective dependency parser: These parsers predict new sentences by using human language data acquired from hand-parsed sentences. They're known to make mistakes and work with a limited collection of coaching information.

2. Stanford parser: It is a Java-based pure language parser. You would want the Stanford CoreNLP parser to perform dependency parsing. The parser supports a number of languages, including English, Chinese, German, and Arabic.

Here's how you should use the parser:

```
from nltk.parse.stanford import StanfordDependencyParser
path_jar = 'path_to/stanford-parser-full-2014-08-27/stanford-parser.jar'
path_models_jar = 'path_to/stanford-parser-full-2014-08-27/stanford-parser-3.4.1-models.jar'
dep_parser = StanfordDependencyParser(
   path_to_jar = path_jar, path_to_models_jar = path_models_jar
)
consequence = dep_parser.raw_parse('I shot an elephant in my sleep')
dependency = consequence.subsequent()
checklist(dependency.triples())
```

The following is the output of the above program:

```
[
   ((u'shot', u'VBD'), u'nsubj', (u'I', u'PRP')),
   ((u'shot', u'VBD'), u'dobj', (u'elephant', u'NN')),
   ((u'elephant', u'NN'), u'det', (u'an', u'DT')),
   ((u'shot', u'VBD'), u'prep', (u'in', u'IN')),
   ((u'in', u'IN'), u'pobj', (u'sleep', u'NN')),
   ((u'sleep', u'NN'), u'poss', (u'my', u'PRP$'))
]
```

Constituency Parsing

Constituency Parsing is based on context-free grammars. Constituency Context-free grammars are used to parse text. Right here the parse tree includes sentences that have been broken down into sub-phrases, each of which belongs to a different grammar class. A terminal node is a linguistic unit or phrase that has a mother or father node and a part-of-speech tag.

For example, "A cat" and "a box beneath the bed", are noun phrases, while "write a letter" and "drive a car" are verb phrases.

Consider the following example sentence: "I shot an elephant in my pajamas." The constituency parse tree is shown graphically as follows:

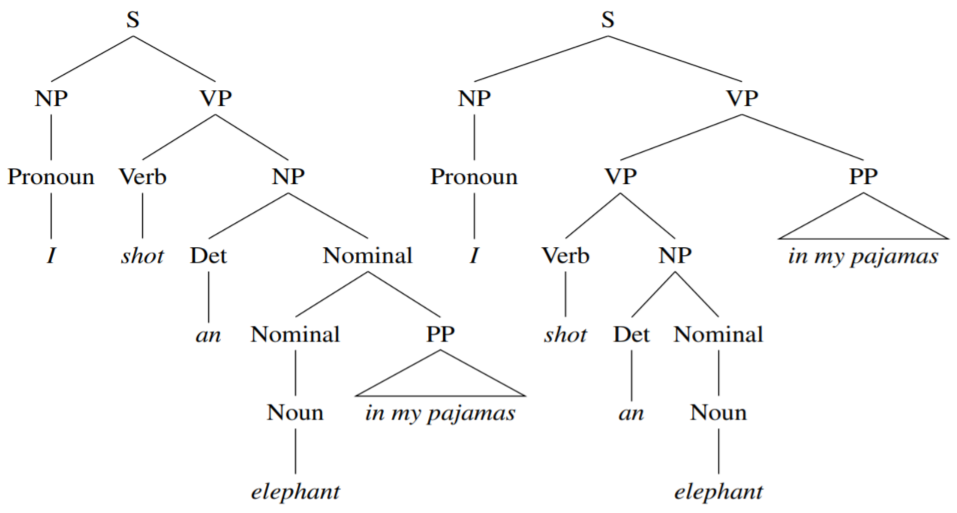


IMAGE – 2

The parse tree on the left represents catching an elephant carrying pyjamas, while the parse tree on the right represents capturing an elephant in his pyjamas.

The entire sentence is broken down into sub-phases till we've got terminal phrases remaining. VP stands for verb phrases, whereas NP stands for noun phrases.

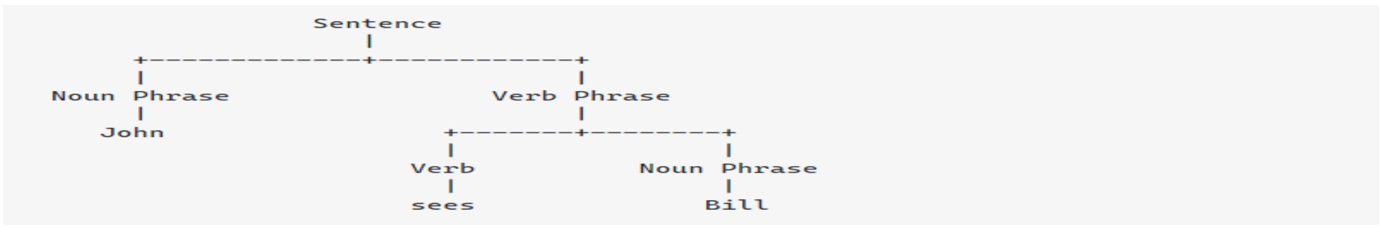
Dependency Parsing vs Constituency Parsing

The Stanford parser will also be used to do constituency parsing. It begins by parsing a phrase using the constituency parser and then transforms the constituency parse tree into a dependency tree.

In case your main objective is to interrupt a sentence into sub-phrases, it is ideal to implement constituency parsing. However, dependency parsing is the best method for discovering the dependencies between phrases in a sentence.

Let's look at an example to see what the difference is:

A constituency parse tree denotes the subdivision of a text into sub-phrases. The tree's non-terminals are different sorts of phrases, the terminals are the sentence's words, and the edges are unlabeled. A constituency parse for the simple statement "John sees Bill" would be:



```
                    Sentence
                       |
        +--------------+--------------+
        |                             |
  Noun Phrase                    Verb Phrase
        |                             |
      John                  +---------+---------+
                            |                   |
                          Verb             Noun Phrase
                            |                   |
                          sees                Bill
```

A dependency parse links words together based on their connections. Each vertex in the tree corresponds to a word, child nodes to words that are reliant on the parent, and edges to relationships. The dependency parse for "John sees Bill" is as follows:

```
             sees
              |
        +--------------+
subject |              | object
        |              |
      John            Bill
```

You should choose the parser type that is most closely related to your objective. If you're looking for sub-phrases inside a sentence, you're definitely interested in the constituency parse. If you're interested in the connection between words, you're probably interested in the dependency parse.

4.1.2Gradient checks, Overfitting

When implementing a neural network from scratch, backpropagation is arguably where it is more prone to mistakes. Therefore, a method to debug this step could potentially save a lot of time and headaches when debugging a neural network.

Here, the method of **gradient checking** will be introduced. Briefly, this methods consists in approximating the gradient using a numerical approach. If it is close to the calculated gradients, then backpropagation was implemented correctly!

A bit of calculus

Assuming that you have some knowledge of calculus, gradient checking will be very easy to understand.

We know that backpropagation calculates the derivatives (or gradient). From your calculus course, you might remember that the definition of a derivative is the following:

$$\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}$$

Definition of the derivative

The definition above can be used as a numerical approximation of the derivative. Taking an *epsilon* small enough, the calculated approximation will have an error in the range of *epsilon* squared.

In other words, if *epsilon* is 0.001, the approximation will be off by 0.00001.

Therefore, we can use this to approximate the gradient, and in turn make sure that backpropagation is implemented properly. This forms the basis of gradient checking!

Vectorized implementation

Now, we need to define a vectorized form of gradient checking before implementing it i Python.

Let's take the weights and bias matrices and reshape them into a big vector *theta*. Similarly, all their respective derivatives will be placed into a vector *d_theta*. Therefore, the approximate gradient can be expressed as:

$$d\theta_{approx} = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon) - J(\theta_1, \theta_2, \ldots, \theta_i - \varepsilon)}{2\varepsilon}$$

Vectorized approximation of the gradient

Notice how the equation above is almost identical to the definition of the limit!

Then, we apply the following formula for gradient check:

$$\frac{\left\| d\theta_{approx} - d\theta \right\|_2}{\left\| d\theta_{approx} \right\|_2 + \left\| d\theta \right\|_2}$$

Gradient check

The equation above is basically the Euclidean distance normalized by the sum of the norm of the vectors. We use normalization in case that one of the vectors is very small.

As a value for *epsilon*, we usually opt for 1e-7. Therefore, if gradient check return a value less than 1e-7, then it means that backpropagation was implemented correctly. Otherwise, there is potentially a mistake in your implementation. If the value exceeds 1e-3, then you are sure that the code is not correct.

What is overfitting?

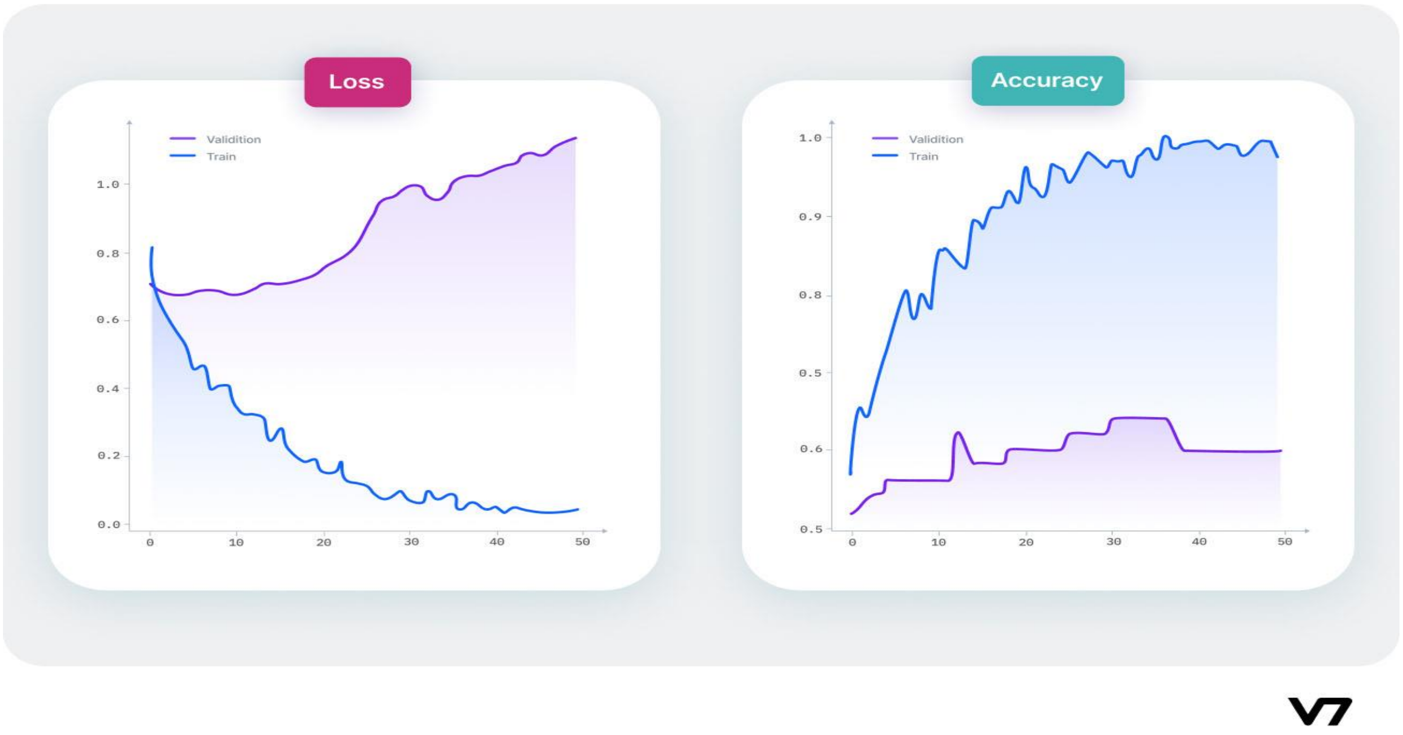
It is a common pitfall in deep learning algorithms in which a model tries to fit the training data entirely and ends up memorizing the data patterns and the noise and random fluctuations.

These models fail to generalize and perform well in the case of unseen data scenarios, defeating the model's purpose.

When can overfitting occur?

The high variance of the model performance is an indicator of an overfitting problem.

The training time of the model or its architectural complexity may cause the model to overfit. If the model trains for too long on the training data or is too complex, it learns the noise or irrelevant information within the dataset.



Signs of overfitting

Overfitting: Key definitions

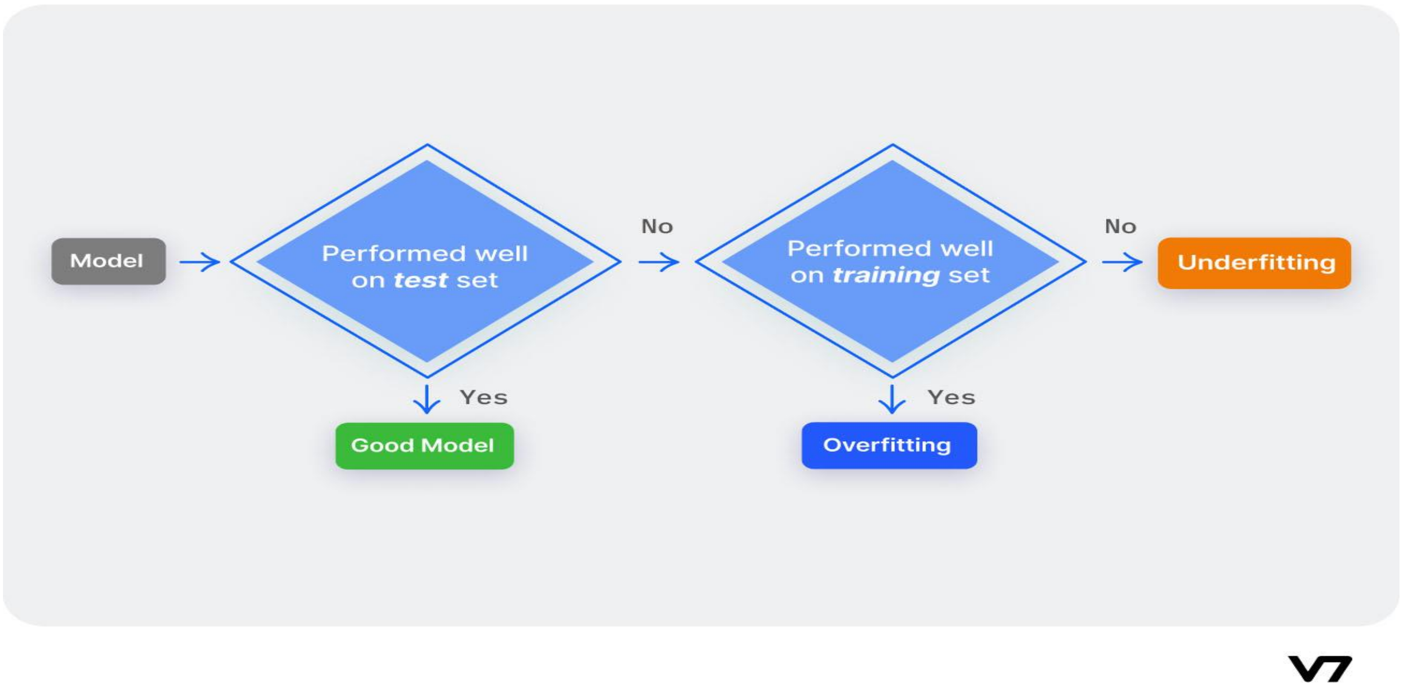
Here are some of the key definitions that'll help you navigate through this guide.

- **Bias:** Bias measures the difference between the model's prediction and the target value. If the model is oversimplified, then the predicted value would be far from the ground truth resulting in more bias.
- **Variance:** Variance is the measure of the inconsistency of different predictions over varied datasets. If the model's performance is tested on different datasets, the closer the prediction, the lesser the variance. Higher variance is an indication of overfitting in which the model loses the ability to generalize.
- **Bias-variance tradeoff:** A simple linear model is expected to have a high bias and low variance due to less complexity of the model and fewer trainable parameters. On the other hand, complex non-linear models tend to observe an opposite behavior. In an ideal scenario, the model would have an optimal balance of bias and variance.
- **Model generalization:** Model generalization means how well the model is trained to extract useful data patterns and classify unseen data samples.
- **Feature selection:** It involves selecting a subset of features from all the extracted features that contribute most towards the model performance. Including all the features unnecessarily increases the model complexity and redundant features can significantly increase the training time.

Overfitting vs. Underfitting

Underfitting occurs when we have a high bias in our data, i.e., we are oversimplifying the problem, and as a result, the model does not work correctly in the training data.

Overfitting occurs when the model has a high variance, i.e., the model performs well on the training data but does not perform accurately in the evaluation set. The model memorizes the data patterns in the training dataset but fails to generalize to unseen examples.


Model
Performed well on *test* set
No
Performed well on *training* set
No
Underfitting
Yes
Good Model
Yes
Overfitting
V7

Overfitting vs. Underfitting vs. Good Model

Overfitting happens when:

1. The data used for training is not cleaned and contains garbage values. The model captures the noise in the training data and fails to generalize the model's learning.
2. The model has a high variance.
3. The training data size is not enough, and the model trains on the limited training data for several epochs.
4. The architecture of the model has several neural layers stacked together. Deep neural networks are complex and require a significant amount of time to train, and often lead to overfitting the training set.

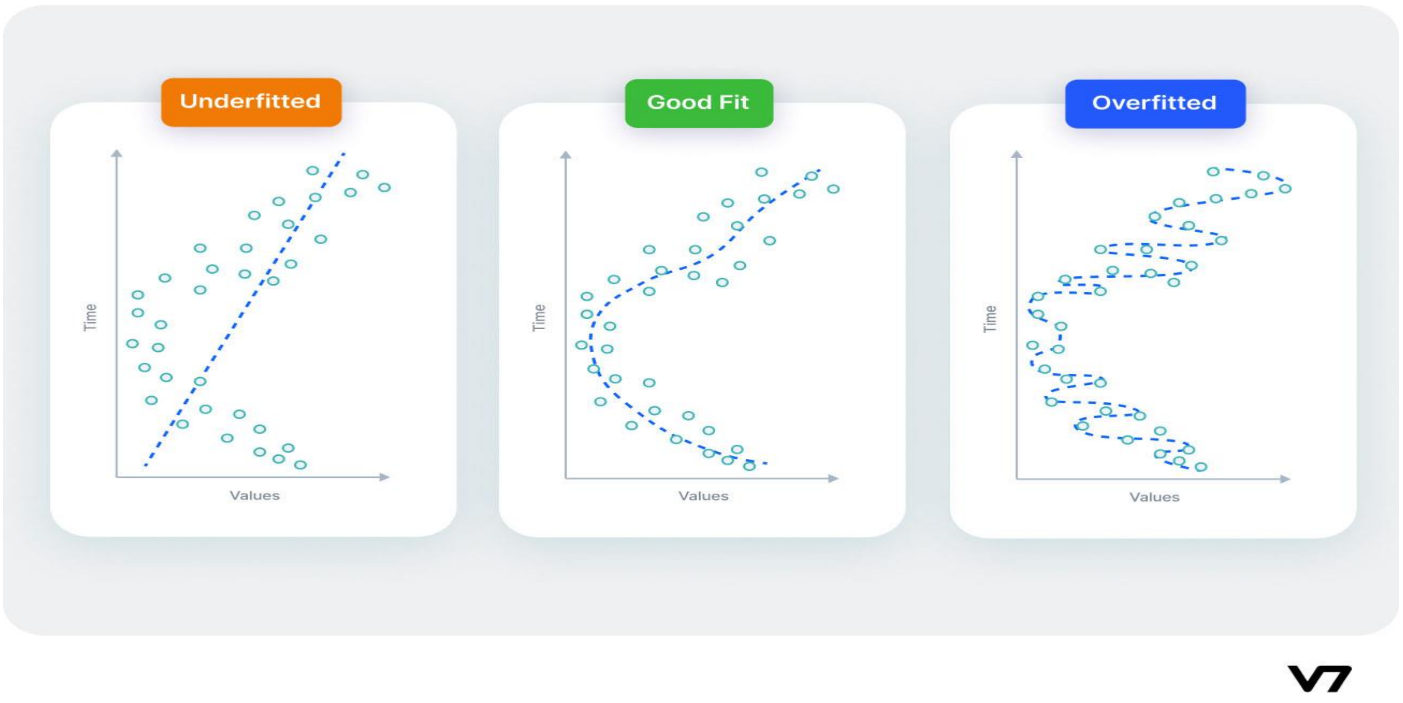
Underfitting happens when:

1. Unclean training data containing noise or outliers can be a reason for the model not being able to derive patterns from the dataset.
2. The model has a high bias due to the inability to capture the relationship between the input examples and the target values.
3. The model is assumed to be too simple. For example, training a linear model in complex scenarios.

The goal is to find a good fit such that the model picks up the patterns from the training data and does not end up memorizing the finer details.

This, in turn, would ensure that the model generalizes and accurately predicts other data samples.

Have a look at this visual comparison to get a better understanding of the differences.



Underfitted vs. Fit vs. Overfitted model

How to detect overfit models?

Here's something you should know—

Detecting overfitting is *technically* not possible unless we test the data.

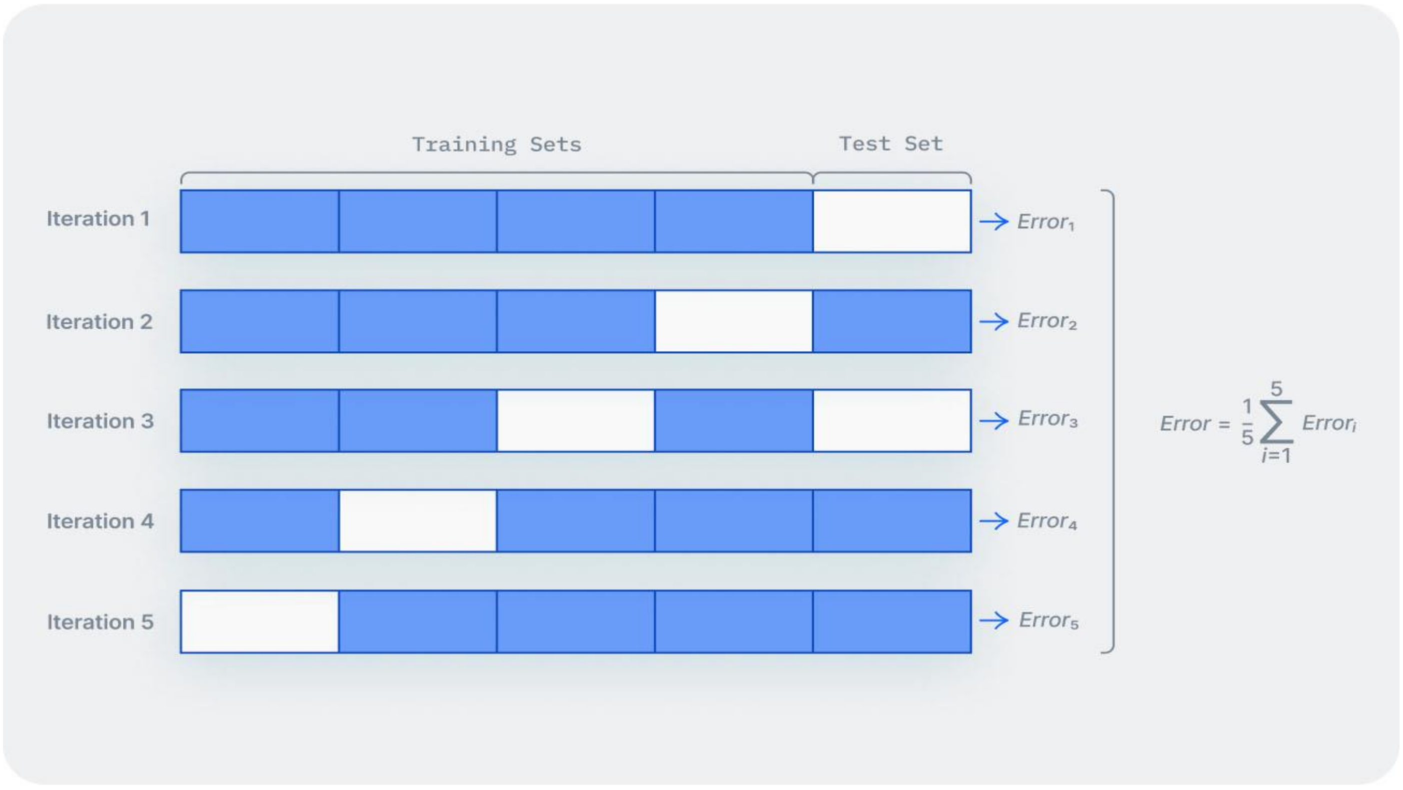
One of the leading indicators of an overfit model is its inability to generalize datasets. The most obvious way to start the process of detecting overfitting machine learning models is to segment the dataset. It's done so that we can examine the model's performance on each set of data to spot overfitting when it occurs and see how the training process works.

K-fold cross-validation is one of the most popular techniques commonly used to detect overfitting.

We split the data points into k equally sized subsets in K-folds cross-validation, called "folds." One split subsets act as the testing set, and the remaining folds will train the model.

The model is trained on a limited sample to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model. One fold acts as a validation set in each turn.

After all the iterations, we average the scores to assess the performance of the overall model.



K-fold cross-validation

10 techniques to avoid overfitting

Here we will discuss possible options to prevent overfitting, which helps improve the model performance.

Train with more data

With the increase in the training data, the crucial features to be extracted become prominent. The model can recognize the relationship between the input attributes and the output variable. The only assumption in this method is that the data to be fed into the model should be clean; otherwise, it would worsen the problem of overfitting.

Data augmentation

An alternative method to training with more data is data augmentation, which is less expensive and safer than the previous method. Data augmentation makes a sample data look slightly different every time the model processes it.

Addition of noise to the input data

Another similar option as data augmentation is adding noise to the input and output data. Adding noise to the input makes the model stable without affecting data quality and privacy while adding noise to the output makes the data more diverse. Noise addition should be done in limit so that it does not make the data incorrect or too different.

Feature selection

Every model has several parameters or features depending upon the number of layers, number of neurons, etc. The model can detect many redundant features or features determinable from other features leading to unnecessary complexity. We very well know that the more complex the model, the higher the chances of the model to overfit.

Cross-validation

Cross-validation is a robust measure to prevent overfitting. The complete dataset is split into parts. In standard K-fold cross-validation, we need to partition the data into k folds. Then, we iteratively train the algorithm on k-1 folds while using the remaining holdout fold as the test set. This method allows us to tune the hyperparameters of the neural network or machine learning model and test it using completely unseen data.

Simplify data

Till now, we have come across model complexity to be one of the top reasons for overfitting. The data simplification method is used to reduce overfitting by decreasing the complexity of the model to make it simple enough that it does not overfit. Some of the procedures include pruning a decision tree, reducing the number of parameters in a neural network, and using dropout on a neutral network.

Regularization

If overfitting occurs when a model is too complex, reducing the number of features makes sense. Regularization methods like Lasso, L1 can be beneficial if we do not know which features to remove from our model. Regularization applies a "penalty" to the input parameters with the larger coefficients, which subsequently limits the model's variance.

Ensembling

It is a machine learning technique that combines several base models to produce one optimal predictive model. In Ensemble learning, the predictions are aggregated to identify the most popular result. Well-known ensemble methods include bagging and boosting, which prevents overfitting as an ensemble model is made from the aggregation of multiple models.

Early stopping

This method aims to pause the model's training before memorizing noise and random fluctuations from the data. There can be a risk that the model stops training too soon, leading to underfitting. One has to come to an optimum time/iterations the model should train.

Adding dropout layers

Large weights in a neural network signify a more complex network. Probabilistically dropping out nodes in the network is a simple and effective method to prevent overfitting. In regularization, some number of layer outputs are randomly ignored or "*dropped out*" to reduce the complexity of the model.

Our tip: If one has two models with almost equal performance, the only difference being that one model is more complex than the other, one should always go with the less complex model. In data science, it's a thumb rule that one should always start with a less complex model and add complexity over time.

Overfitting: Key Takeaways

Finally, here's a short recap of everything we've learn today.

- A model is trained by hyperparameters tuning using a training dataset and then tested on a separate dataset called the testing set. If a model performs well on training data, it should work well for the testing set.

- The scenario in which the model performs well in the training phase but gives a poor accuracy in the test dataset is called overfitting.
- The machine learning algorithm performs poorly on the training dataset if it cannot derive features from the training set. This condition is called underfitting.

We can solve the problem of overfitting by:

- Increasing the training data by data augmentation
- Feature selection by choosing the best features and remove the useless/unnecessary features
- Early stopping the training of deep learning models where the number of epochs is set high
- Dropout techniques by randomly selecting nodes and removing them from training
- Reducing the complexity of the model

4.1.3Regularization

What is Regularization?

Regularization is one of the most important concepts of machine learning. It is a technique to prevent the model from overfitting by adding extra information to it.

Sometimes the machine learning model performs well with the training data but does not perform well with the test data. It means the model is not able to predict the output when deals with unseen data by introducing noise in the output, and hence the model is called overfitted. This problem can be deal with the help of a regularization technique.

This technique can be used in such a way that it will allow to maintain all variables or features in the model by reducing the magnitude of the variables. Hence, it maintains accuracy as well as a generalization of the model.

It mainly regularizes or reduces the coefficient of features toward zero. In simple words, "*In regularization technique, we reduce the magnitude of the features by keeping the same number of features."*

How does Regularization Work?

Regularization works by adding a penalty or complexity term to the complex model. Let's consider the simple linear regression equation:

$$y= \beta0+\beta1x1+\beta2x2+\beta3x3+\cdots+\beta nxn +b$$

In the above equation, Y represents the value to be predicted

X1, X2, …Xn are the features for Y.

β0,β1,…..βn are the weights or magnitude attached to the features, respectively. Here represents the bias of the model, and b represents the intercept.

Linear regression models try to optimize the β0 and b to minimize the cost function. The equation for the cost function for the linear model is given below:

$$\sum_{i=1}^{M}\left(y_i - y'_i\right)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n} \beta_j * Xij\right)^2$$

Now, we will add a loss function and optimize parameter to make the model that can predict the accurate value of Y. The loss function for the linear regression is called as **RSS or Residual sum of squares.**

Techniques of Regularization

There are mainly two types of regularization techniques, which are given below:

- **Ridge Regression**
- **Lasso Regression**

Ridge Regression

- Ridge regression is one of the types of linear regression in which a small amount of bias is introduced so that we can get better long-term predictions.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as **L2 regularization**.
- In this technique, the cost function is altered by adding the penalty term to it. The amount of bias added to the model is called **Ridge Regression penalty**. We can calculate it by multiplying with the lambda to the squared weight of each individual feature.
- The equation for the cost function in ridge regression will be:

$$\sum_{i=1}^{M}\left(y_i - y'_i\right)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n} \beta_j * x_{ij}\right)^2 + \lambda \sum_{j=0}^{n} \beta_j{}^2$$

- In the above equation, the penalty term regularizes the coefficients of the model, and hence ridge regression reduces the amplitudes of the coefficients that decreases the complexity of the model.
- As we can see from the above equation, if the values of λ **tend to zero, the equation becomes the cost function of the linear regression model.** Hence, for the minimum value of λ, the model will resemble the linear regression model.
- A general linear or polynomial regression will fail if there is high collinearity between the independent variables, so to solve such problems, Ridge regression can be used.
- It helps to solve the problems if we have more parameters than samples.

Lasso Regression:

- Lasso regression is another regularization technique to reduce the complexity of the model. It stands for **Least Absolute and Selection Operator.**
- It is similar to the Ridge Regression except that the penalty term contains only the absolute weights instead of a square of weights.

- Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0.
- It is also called as **L1 regularization.** The equation for the cost function of Lasso regression will be:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}|\beta_j|$$

- Some of the features in this technique are completely neglected for model evaluation.
- Hence, the Lasso regression can help us to reduce the overfitting in the model as well as the feature selection.

Key Difference between Ridge Regression and Lasso Regression

- **Ridge regression** is mostly used to reduce the overfitting in the model, and it includes all the features present in the model. It reduces the complexity of the model by shrinking the coefficients.
- **Lasso regression** helps to reduce the overfitting in the model as well as feature selection.

4.1.4Activation functions

Definition

In artificial neural networks, an activation function is one that outputs a smaller value for tiny inputs and a higher value if its inputs are greater than a threshold. An activation function "fires" if the inputs are big enough; otherwise, nothing happens. An activation function, then, is a gate that verifies how an incoming value is higher than a threshold value.

Because they introduce non-linearities in neural networks and enable the neural networks can learn powerful operations, activation functions are helpful. A feedforward neural network might be refactored into a straightforward linear function or matrix transformation on to its input if indeed the activation functions were taken out.

By generating a weighted total and then including bias with it, the activation function determines whether a neuron should be turned on. The activation function seeks to boost a neuron's output's nonlinearity.

Explanation: As we are aware, neurons in neural networks operate in accordance with weight, bias, and their corresponding activation functions. Based on the mistake, the values of the neurons inside a neural network would be modified. This process is known as back-propagation. Back-propagation is made possible by activation functions since they provide the gradients and error required to change the biases and weights.

Need of Non-linear Activation Functions

An interconnected regression model without an activation function is all that a neural network is. Input is transformed nonlinearly by the activation function, allowing the system to learn and perform more challenging tasks.

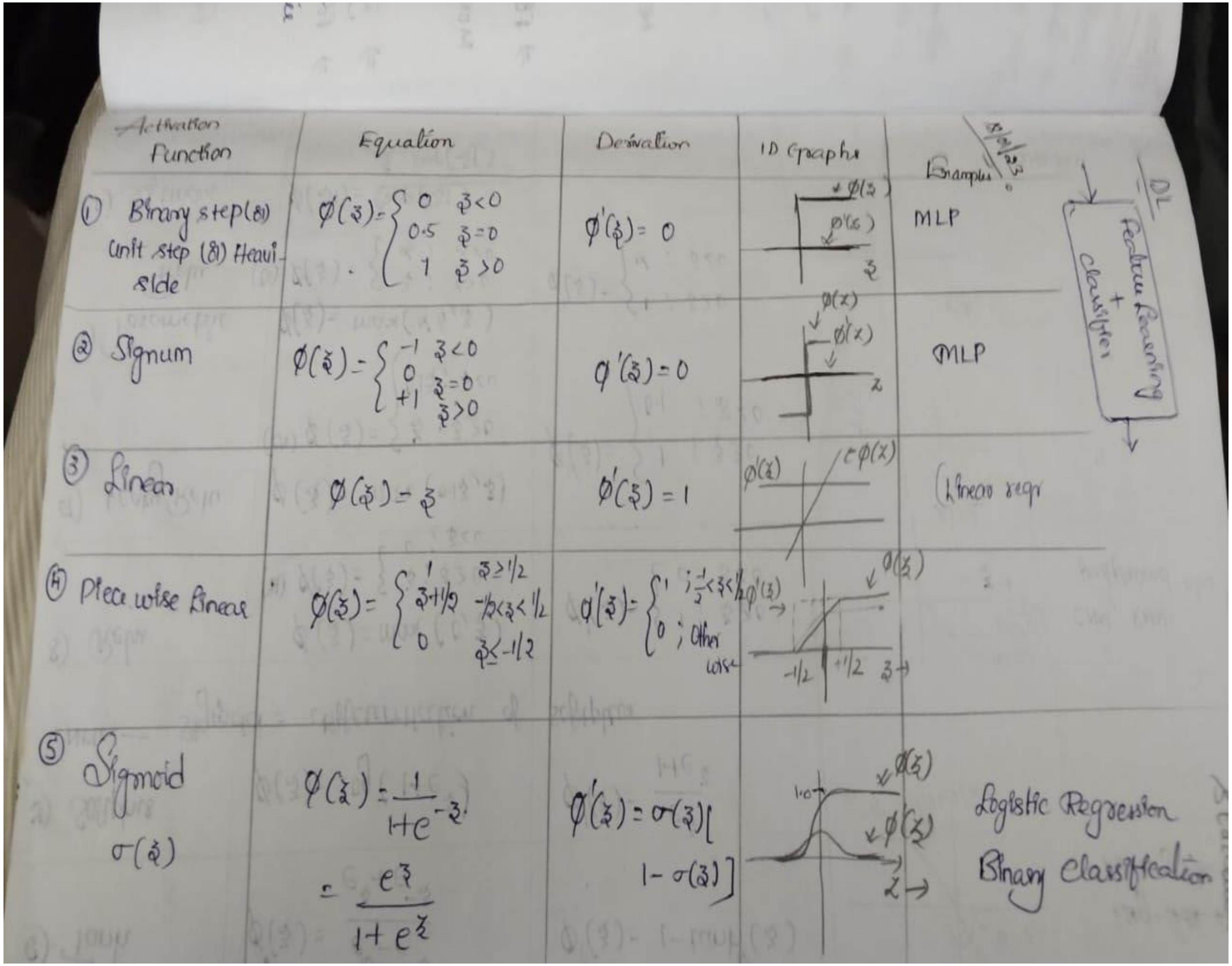
It is merely a thing procedure that is used to obtain a node's output. It also goes by the name Transfer Function.

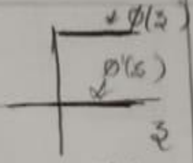
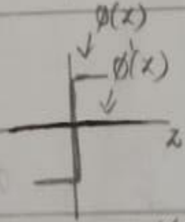
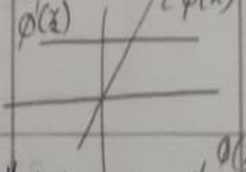
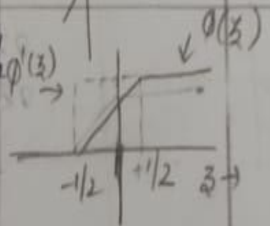
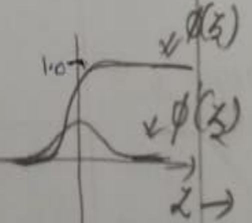
The mixture of two linear functions yields a linear function, so no matter how several hidden layers we add to a neural network, they all will behave in the same way. The neuron cannot learn if all it has is a linear model. It will be able to learn based on the difference with respect to error with a non-linear activation function.

The mixture of two linear functions yields a linear function in itself, so no matter how several hidden layers we add to a neural network, they all will behave in the same way. The neuron cannot learn if all it has is a linear model.

The two main categories of activation functions are:

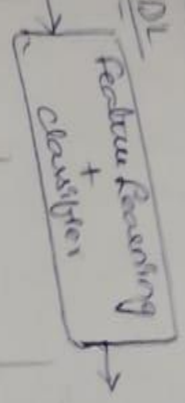
- Linear Activation Function
- Non-linear Activation Functions

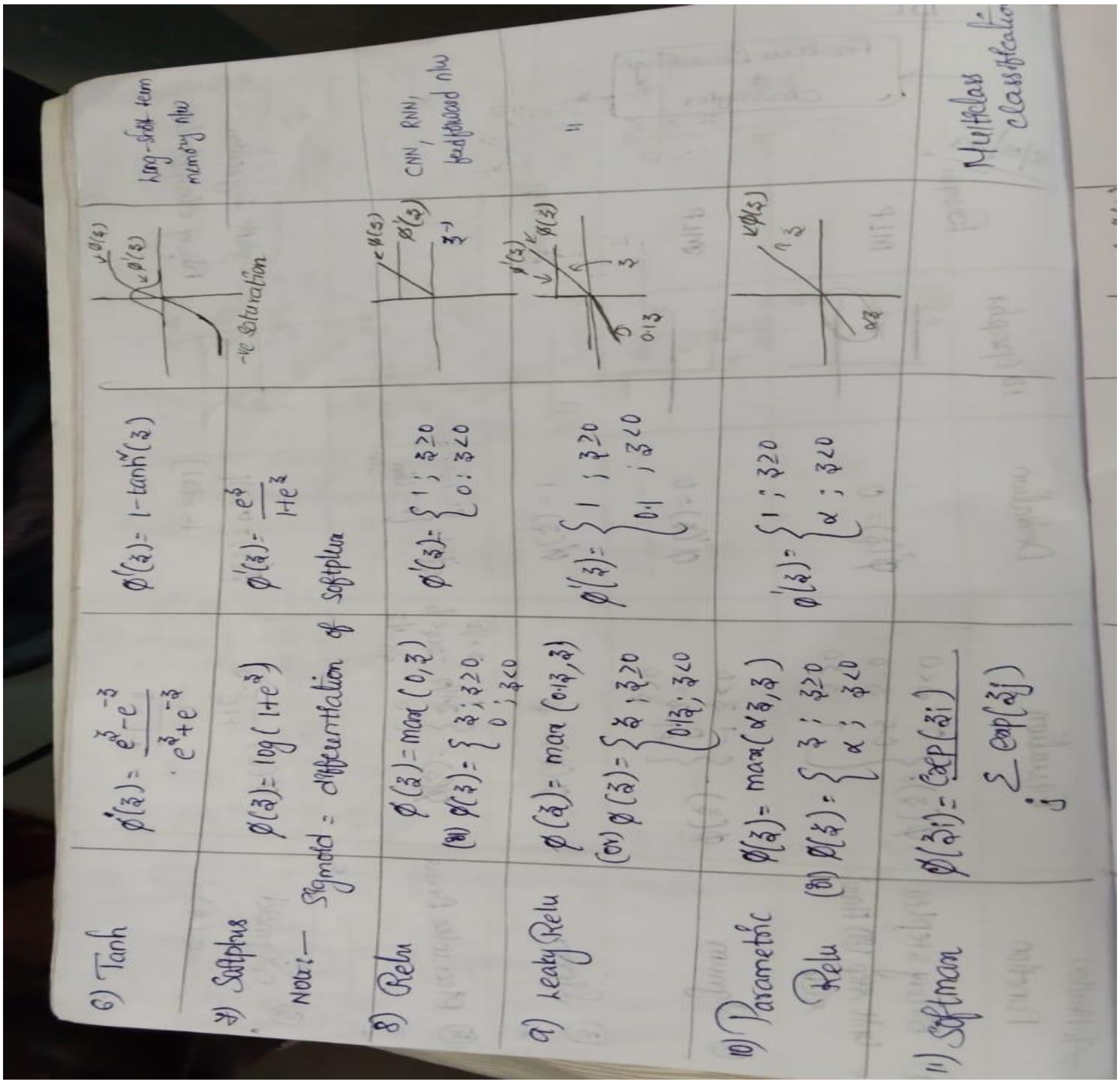


Activation Function	Equation	Derivation	1D Graphs	Examples
① Binary step (or) Unit step (or) Heavi-side	$\phi(z) = \begin{cases} 0 & z<0 \\ 0.5 & z=0 \\ 1 & z>0 \end{cases}$	$\phi'(z) = 0$		MLP
② Signum	$\phi(z) = \begin{cases} -1 & z<0 \\ 0 & z=0 \\ +1 & z>0 \end{cases}$	$\phi'(z) = 0$		MLP
③ Linear	$\phi(z) = z$	$\phi'(z) = 1$		(Linear regr
④ Piece wise Linear	$\phi(z) = \begin{cases} 1 & z \geq 1/2 \\ z+1/2 & -1/2<z<1/2 \\ 0 & z \leq -1/2 \end{cases}$	$\phi'(z) = \begin{cases} 1 & ; -\frac{1}{2}<z<1/2 \\ 0 & ; \text{Otherwise} \end{cases}$		
⑤ Sigmoid $\sigma(z)$	$\phi(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$	$\phi'(z) = \sigma(z)[1-\sigma(z)]$		Logistic Regression Binary Classification

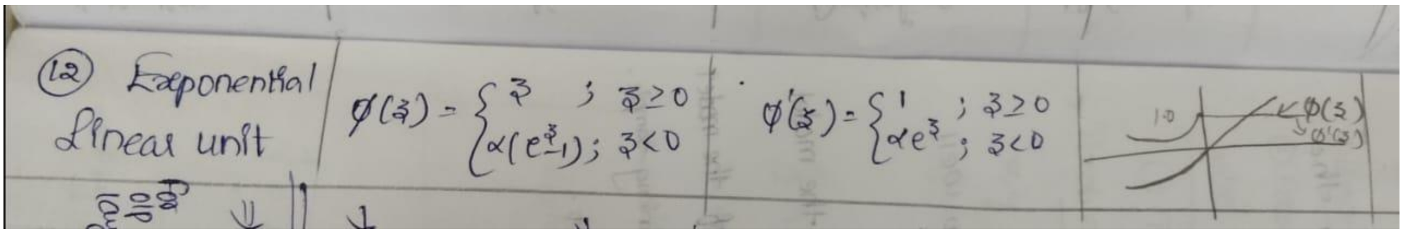
8/12/23

DL





6) Tanh	$\phi(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$\phi'(z) = 1-\tanh^2(z)$	-ve Saturation	Long-short term memory n/w
7) Softplus	$\phi(z) = \log(1+e^{z})$	$\phi'(z) = \frac{e^{z}}{1+e^{z}}$		
Note:— Sigmoid = differentiation of Softplus				
8) Relu	$\phi(z) = \max(0,z)$ (or) $\phi(z) = \begin{cases} z ; z\geq 0 \\ 0 ; z<0 \end{cases}$	$\phi'(z) = \begin{cases} 1 ; z\geq 0 \\ 0 : z<0 \end{cases}$		CNN, RNN, feedforward n/w
9) Leaky Relu	$\phi(z) = \max(0.1z, z)$ (or) $\phi(z) = \begin{cases} z ; z\geq 0 \\ 0.1z ; z<0 \end{cases}$	$\phi'(z) = \begin{cases} 1 ; z\geq 0 \\ 0.1 ; z<0 \end{cases}$	0.1z	"
10) Parametric Relu	$\phi(z) = \max(\alpha z, z)$ (or) $\phi(z) = \begin{cases} z ; z\geq 0 \\ \alpha ; z<0 \end{cases}$	$\phi'(z) = \begin{cases} 1 ; z\geq 0 \\ \alpha ; z<0 \end{cases}$	αz	
11) Softmax	$\phi(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$			Multiclass classification



(12) Exponential Linear unit	$\phi(z) = \begin{cases} z ; z\geq 0 \\ \alpha(e^{z}-1) ; z<0 \end{cases}$	$\phi'(z) = \begin{cases} 1 ; z\geq 0 \\ \alpha e^{z} ; z<0 \end{cases}$	

4.1.5Multitask and Semi-supervised Learning

Multi-Task Learning (MTL) is a type of machine learning technique where a model is trained to perform multiple tasks simultaneously. In deep learning, MTL refers to training a neural network to perform multiple tasks by sharing some of the network's layers and parameters across tasks.

In MTL, the goal is to improve the generalization performance of the model by leveraging the information shared across tasks. By sharing some of the network's parameters, the model can learn a more efficient and compact representation of the data, which can be beneficial when the tasks are related or have some commonalities.

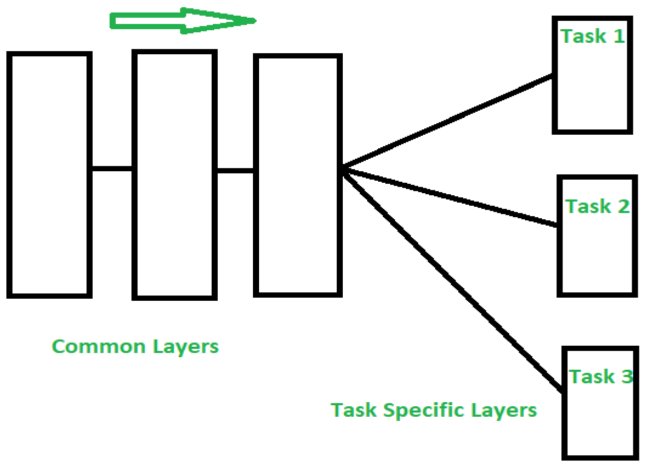
There are different ways to implement MTL in deep learning, but the most common approach is to use a shared feature extractor and multiple task-specific heads. The shared feature extractor is a part of the network that is shared across tasks and is used to extract features from the input data. The task-specific heads are used to make predictions for each task and are typically connected to the shared feature extractor.

Another approach is to use a shared decision-making layer, where the decision-making layer is shared across tasks, and the task-specific layers are connected to the shared decision-making layer.

MTL can be useful in many applications such as natural language processing, computer vision, and healthcare, where multiple tasks are related or have some commonalities. It is also useful when the data is limited, MTL can help to improve the generalization performance of the model by leveraging the information shared across tasks.

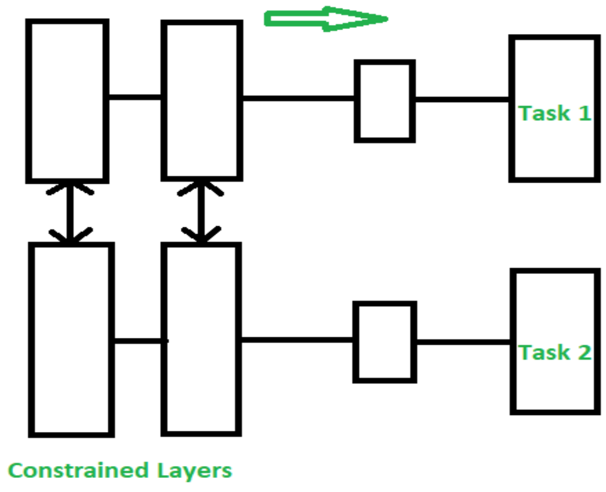
However, MTL also has its own limitations, such as when the tasks are very different

Multi-Task Learning is a sub-field of Deep Learning. It is recommended that you familiarize yourself with the concepts of neural networks to understand what multi-task learning means. **What is Multi-Task Learning?** Multi-Task learning is a sub-field of Machine Learning that aims to solve multiple different tasks at the same time, by taking advantage of the similarities between different tasks. This can improve the learning efficiency and also act as a regularizer which we will discuss in a while. Formally, if there are **n** tasks (conventional deep learning approaches aim to solve just 1 task using 1 particular model), where these **n** tasks or a subset of them are related to each other but not exactly identical, Multi-Task Learning **(MTL)** will help in improving the learning of a particular model by using the knowledge contained in all the n tasks. **Intuition behind Multi-Task Learning (MTL):** By using Deep learning models, we usually aim to learn a good representation of the features or attributes of the input data to predict a specific value. Formally, we aim to optimize for a particular function by training a model and fine-tuning the hyperparameters till the performance can't be increased further. By using MTL, it might be possible to increase performance even further by forcing the model to learn a more generalized representation as it learns (updates its weights) not just for one specific task but a bunch of tasks. Biologically, humans learn in the same way. We learn better if we learn multiple related tasks instead of focusing on one specific task for a long time. **MTL as a regularizer:** In the lingo of Machine Learning, MTL can also be looked at as a way of inducing bias. It is a form of inductive transfer, using multiple tasks induces a bias that prefers hypotheses that can explain all the **n** tasks. MTL acts as a regularizer by introducing inductive bias as stated above. It significantly reduces the risk of overfitting and also reduces the model's ability to accommodate random noise during training. Now, let's discuss the major and prevalent techniques to use MTL. **Hard Parameter Sharing** – A common hidden layer is used for all tasks but several task specific layers are kept intact towards the end of the model. This technique is very useful as by learning a representation for various tasks by a common hidden layer, we reduce the risk of overfitting.



Hard Parameter Sharing

Soft Parameter Sharing – Each model has their own sets of weights and biases and the distance between these parameters in different models is regularized so that the parameters become similar and can represent all the tasks.



Soft Parameter Sharing

Assumptions and Considerations – Using MTL to share knowledge among tasks are very useful only when the tasks are very similar, but when this assumption is violated, the performance will significantly decline. **Applications:** MTL techniques have found various uses, some of the major applications are-

- Object detection and Facial recognition
- Self Driving Cars: Pedestrians, stop signs and other obstacles can be detected together
- Multi-domain collaborative filtering for web applications
- Stock Prediction
- Language Modelling and other NLP applications

Important points:

Here are some important points to consider when implementing Multi-Task Learning (MTL) for deep learning:

1. Task relatedness: MTL is most effective when the tasks are related or have some commonalities, such as natural language processing, computer vision, and healthcare.
2. Data limitation: MTL can be useful when the data is limited, as it allows the model to leverage the information shared across tasks to improve the generalization performance.
3. Shared feature extractor: A common approach in MTL is to use a shared feature extractor, which is a part of the network that is shared across tasks and is used to extract features from the input data.
4. Task-specific heads: Task-specific heads are used to make predictions for each task and are typically connected to the shared feature extractor.
5. Shared decision-making layer: another approach is to use a shared decision-making layer, where the decision-making layer is shared across tasks, and the task-specific layers are connected to the shared decision-making layer.
6. Careful architecture design: The architecture of MTL should be carefully designed to accommodate the different tasks and to make sure that the shared features are useful for all tasks.
7. Overfitting: MTL models can be prone to overfitting if the model is not regularized properly.
8. Avoiding negative transfer: when the tasks are very different or independent, MTL can lead to suboptimal performance compared to training a single-task model. Therefore, it is important to make sure that the shared features are useful for all tasks to avoid negative transfer.

Semi-Supervised learning is a type of Machine Learning algorithm that represents the intermediate ground between Supervised and Unsupervised learning algorithms. It uses the combination of labeled and unlabeled datasets during the training period.

Before understanding the Semi-Supervised learning, you should know the main categories of Machine Learning algorithms. Machine Learning consists of three main categories: **Supervised Learning**, **Unsupervised Learning**, **and** **Reinforcement Learning.** Further, the basic difference between Supervised and unsupervised learning is that *supervised learning datasets consist of an output label training data associated with each tuple*, and *unsupervised datasets do not consist the same.* ***Semi-supervised learning is an important category that lies between the Supervised and Unsupervised machine learning.*** Although Semi-supervised learning is the middle ground between supervised and unsupervised learning and operates on the data that consists of a few labels, it mostly consists of unlabeled data. As labels are costly, but for the corporate purpose, it may have few labels.

The basic disadvantage of supervised learning is that it requires hand-labeling by ML specialists or data scientists, and it also requires a high cost to process. Further unsupervised learning also has a limited spectrum for its applications. **To overcome these drawbacks of supervised learning and unsupervised learning algorithms, the concept of Semi-supervised learning is introduced**. In this algorithm, training data is a combination of both labeled and unlabeled data. However, labeled data exists with a very small amount while it consists of a huge amount of unlabeled data. Initially, similar data is clustered along with an unsupervised learning algorithm, and further, it helps to label the unlabeled data into labeled data. It is why label data is a comparatively, more expensive acquisition than unlabeled data.

We can imagine these algorithms with an example. Supervised learning is where a student is under the supervision of an instructor at home and college. Further, if that student is self-analyzing the same concept without any help from the instructor, it comes under unsupervised learning. Under semi-supervised learning, the student has to revise itself after analyzing the same concept under the guidance of an instructor at college.

Play Video

Assumptions followed by Semi-Supervised Learning

To work with the unlabeled dataset, there must be a relationship between the objects. To understand this, semi-supervised learning uses any of the following assumptions:

- **Continuity Assumption:**
  As per the continuity assumption, the objects near each other tend to share the same group or label. This assumption is also used in supervised learning, and the datasets are separated by the decision boundaries. But in semi-supervised, the decision boundaries are added with the smoothness assumption in low-density boundaries.
- **Cluster assumptions-** In this assumption, data are divided into different discrete clusters. Further, the points in the same cluster share the output label.

- **Manifold assumptions-** This assumption helps to use distances and densities, and this data lie on a manifold of fewer dimensions than input space.
- The dimensional data are created by a process that has less degree of freedom and may be hard to model directly. **(This assumption becomes practical if high).**

Working of Semi-Supervised Learning

Semi-supervised learning uses pseudo labeling to train the model with less labeled training data than supervised learning. The process can combine various neural network models and training ways. The whole working of semi-supervised learning is explained in the below points:

- Firstly, it trains the model with less amount of training data similar to the supervised learning models. The training continues until the model gives accurate results.
- The algorithms use the unlabeled dataset with pseudo labels in the next step, and now the result may not be accurate.
- Now, the labels from labeled training data and pseudo labels data are linked together.
- The input data in labeled training data and unlabeled training data are also linked.
- In the end, again train the model with the new combined input as did in the first step. It will reduce errors and improve the accuracy of the model.

Difference between Semi-supervised and Reinforcement Learning.

Reinforcement learning is different from semi-supervised learning, as it works with rewards and feedback. ***Reinforcement learning aims to maximize the rewards by their hit and trial actions, whereas in semi-supervised learning, we train the model with a less labeled dataset.***

Real-world applications of Semi-supervised Learning-

Semi-supervised learning models are becoming more popular in the industries. Some of the main applications are as follows.

- **Speech Analysis-** It is the most classic example of semi-supervised learning applications. Since, labeling the audio data is the most impassable task that requires many human resources, this problem can be naturally overcome with the help of applying SSL in a Semi-supervised learning model.
- **Web content classification-** However, this is very critical and impossible to label each page on the internet because it needs mode human intervention. Still, this problem can be reduced through Semi-Supervised learning algorithms.
  Further, Google also uses semi-supervised learning algorithms to rank a webpage for a given query.
- **Protein sequence classification-** DNA strands are larger, they require active human intervention. So, the rise of the Semi-supervised model has been proximate in this field.
- **Text document classifier-** As we know, it would be very unfeasible to find a large amount of labeled text data, so semi-supervised learning is an ideal model to overcome this.

4.2Text Embedding

4.2.1 Word Vector representations: word2vec, GloVe

What are Word Embeddings or Text Embeddings?

It is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meaning to have a similar representation. They can also approximate meaning. A word vector with 50 values can represent 50 unique features.

Features: Anything that relates words to one another. Eg: Age, Sports, Fitness, Employed etc. Each word vector has values corresponding to these features.

Goal of Word Embeddings

- To reduce dimensionality
- To use a word to predict the words around it
- Inter word semantics must be captured

How are Word Embeddings used?

- They are used as input to machine learning models.
  Take the words —-> Give their numeric representation —-> Use in training or inference
- To represent or visualize any underlying patterns of usage in the corpus that was used to train them.

Implementations of Word Embeddings:

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words(BOW), CountVectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems.

Let's take an example to understand how word vector is generated by taking emoticons which are most frequently used in certain conditions and transform each emoji into a vector and the conditions will be our features.

Happy	????	????	????
Sad	????	????	????
Excited	????	????	????
Sick	????	????	????

The emoji vectors for the emojis will be:

[happy,sad,excited,sick]

???? =[1,0,1,0]

???? =[0,1,0,1]

???? =[0,0,1,1]

.....

In a similar way, we can create word vectors for different words as well on the basis of given features. The words with similar vectors are most likely to have the same meaning or are used to convey the same sentiment.

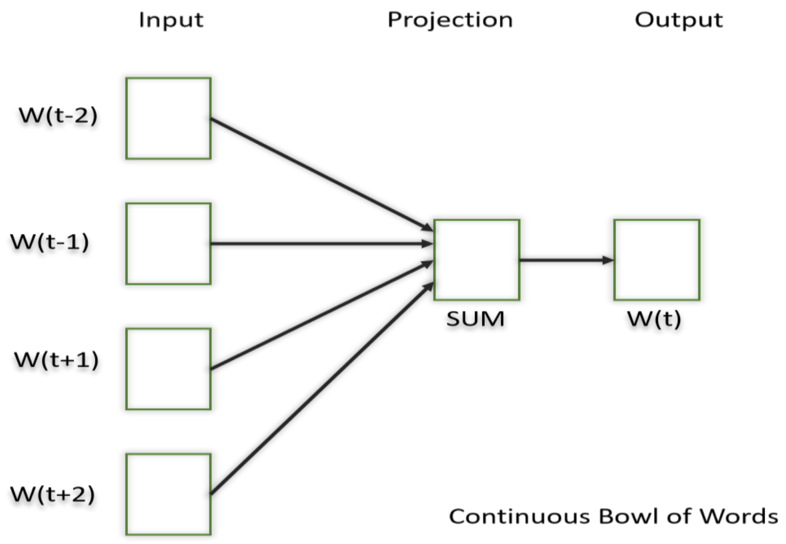
In this article we will be discussing two different approaches to get Word Embeddings:

1) Word2Vec:

In Word2Vec every word is assigned a vector. We start with either a random vector or **one-hot vector**. **One-Hot vector:** A representation where only one bit in a vector is 1.If there are 500 words in the corpus then the vector length will be 500. After assigning vectors to each word we take a window size and iterate through the entire corpus. While we do this there are two **neural embedding methods** which are used:

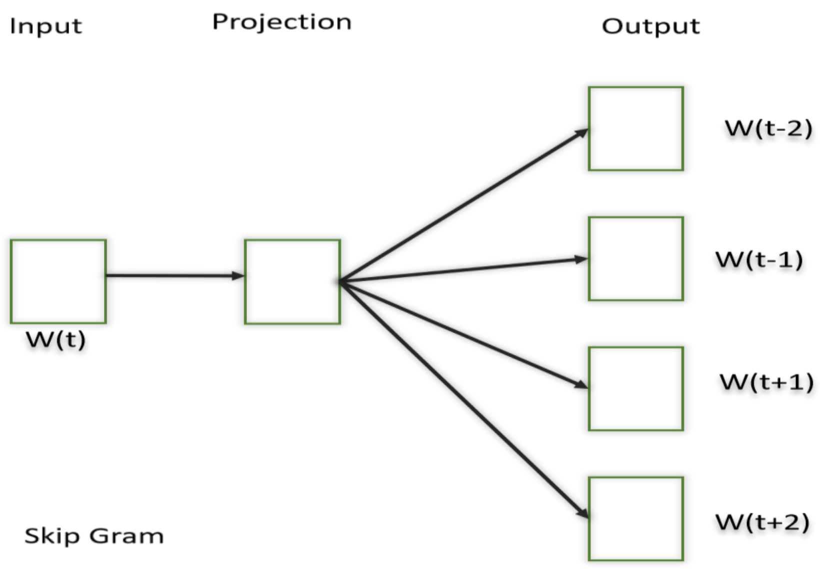
1.1) Continuous Bowl of Words(CBOW)

In this model what we do is we try to fit the neighboring words in the window to the central word.



1.2) Skip Gram

In this model, we try to make the central word closer to the neighboring words. It is the complete opposite of the CBOW model. It is shown that this method produces more meaningful embeddings.



After applying the above neural embedding methods we get trained vectors of each word after many iterations through the corpus. These trained vectors preserve syntactical or semantic information and are converted to lower dimensions. The vectors with similar meaning or semantic information are placed close to each other in space.

2) GloVe:

This is another method for creating word embeddings. In this method, we take the corpus and iterate through it and get the co-occurrence of each word with other words in the corpus. We get a co-occurrence matrix through this. The words which occur next to each other get a value of 1, if they are one word apart then 1/2, if two words apart then 1/3 and so on.

Let us take an example to understand how the matrix is created. We have a small corpus:

Corpus:

It is a nice evening.

Good Evening!

Is it a nice evening?

	it	**is**	**a**	**nice**	**evening**	**good**
it	0					
is	1+1	0				
a	1/2+1	1+1/2	0			
nice	1/3+1/2	1/2+1/3	1+1	0		
evening	1/4+1/3	1/3+1/4	1/2+1/2	1+1	0	
good	0	0	0	0	1	0

The upper half of the matrix will be a reflection of the lower half. We can consider a window frame as well to calculate the co-occurrences by shifting the frame till the end of the corpus. This helps gather information about the context in which the word is used.

Initially, the vectors for each word is assigned randomly. Then we take two pairs of vectors and see how close they are to each other in space. If they occur together more often or have a higher value in the co-occurrence matrix and are far apart in space then they are brought close to each other. If they are close to each other but are rarely or not frequently used together then they are moved further apart in space.

After many iterations of the above process, we'll get a vector space representation that approximates the information from the co-occurrence matrix. The performance of GloVe is better than Word2Vec in terms of both semantic and syntactic capturing.

Pre-trained Word Embedding Models:

People generally use pre-trained models for word embeddings. Few of them are:

- SpaCy
- fastText
- Flair etc.

Common Errors made:

- You need to use the exact same pipeline during deploying your model as were used to create the training data for the word embedding. If you use a different tokenizer or different method of handling white space, punctuation etc. you might end up with incompatible inputs.
- Words in your input that doesn't have a pre-trained vector. Such words are known as **Out of Vocabulary Word(oov). What** you can do is replace those words with "UNK" which means unknown and then handle them separately.
- Dimension mis-match: Vectors can be of many lengths. If you train a model with vectors of length say 400 and then try to apply vectors of length 1000 at inference time, you will run into errors. So make sure to use the same dimensions throughout.

Benefits of using Word Embeddings:

- It is much faster to train than hand build models like WordNet(which uses ***graph embeddings***)
- Almost all modern NLP applications start with an embedding layer
- It Stores an approximation of meaning

Drawbacks of Word Embeddings:

- It can be memory intensive
- It is corpus dependent. Any underlying bias will have an effect on your model
- It cannot distinguish between homophones. Eg: brake/break, cell/sell, weather/whether etc.

4.2.2Advanced word vector representations

Expressing power of notations used to represent a vocabulary of a language has been a great deal of interest in the field of linguistics. Languages in practice have semantic ambiguity.

"John wished his wife, and so did Sam".

Sam wished John's wife or his own? These ambiguities must be handled in order to represent information in true form.

So, how do we develop a machine level understanding for language modelling task. Classical count based or similarity parameter based methods have existed for quite a long time in Computer Science.

But, now with the advent of deep learning models like RNNs this expressing power of language modelling is of great interest for creating efficient system to store information and finding relations between vocabulary terms. This will act as fundamental block in encoder-decoder models like seq-to-seq model.

The fundamental understanding of language will be different for machines.

Machines are better at understanding numbers that actual text passed on as tokens. This process of converting text to numbers is called **vectorization**. Vectors then combine to form vector space which is continuous in nature, an algebraic model where rules of vector addition and similarity measures apply. Different approaches of vectorization exists let's move from the most primitive ones to most advanced ones.

The representation models that will be discussed in this article are :

1. One-hot representations

2. Distributed Representations

3. Singular Value Decomposition

4. Continuous bag of words model

5. Skip-Gram model

6. Glove Representations

One Hot Representation

For defining representation power of a system we first will look into workings of different representation systems. These systems represent each and every word of a vocabulary in the form a vector and create a finite vector space.

Let's see an example of ***one-hot*** *representation* of words. Each word is represented with a large vector of size |V| i.e. vocabulary's size for the given corpus.

V = [human,machine, interface, for, computer, applications, user, opinion, of, system, response, time, interface, management, engineering, improved]

machine:

0	1	0	...	0	0	0

The representation of the i-th word will have a 1 in the i-th position and a 0 in the remaining |V | − 1 positions.

It is a very simple form of representation with very easy implementation. But, many of the faults would have become clear even from such small example. Like, huge memory required for storing and processing such vectors. Along, with sparse nature of these vectors.For example, the size of |V| is very large like 3M for Google 1T corpus. This notation will fail in terms of computation overhead caused by representation power of this system.

Also, no notion of similarity is captured. Cosine similarity b/w unique words is zero and Euclidean distance is always sqrt(2). Meaning, no semantic information is getting expressed with this representation system.

Now, clearly we should move to a representation which saves space and hold some semantic power to express relationships among words. Let’s see a representation based on similarity.

You shall know a word by the company it keeps

— Firth, J.R.

Distributed Representation Of Words

The idea is to quantify co-occurrence of terms in a corpus. This co-occurrence is measured with window size of ‘k’ around the terms which signifies the context being distributed in that window size. With this method in mind, a *co-occurrence matrix of* **terms × terms** which captures the number of times a term appears in the context of another term is created. Also, it is a good practice to remove stopwords as these are high frequency words providing least amount of meaningful insight. Also, we can set an upper threshold for ***t*** for words.

$$X_{ij} = min(count(w_i, c_j), t),$$

Now, consider the following corpus:

Human machine interface for computer applications.User opinion of computer system response time.User interface management system.System engineering for improved response time

	human	machine	system	for	...	user
human	0	1	0	1	...	0
machine	1	0	0	1	...	0
system	0	0	0	1	...	2
for	1	1	1	0	...	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
user	0	0	2	0	...	0

Co-occurrence Matrix with window of size k=2. Each row[column] of the co-occurrence matrix gives a vectorial representation of the corresponding word's context.

Here, in above case if stopwords are not handled properly they will create problem with their relative high frequency. So, a new quantity known as **Positive Pointwise Mutual Information** will be used that takes into account relative and individual occurrences along with size of vocabulary.

$$\begin{aligned} PPMI(w,c) &= PMI(w,c) \quad if\ PMI(w,c) > 0 \\ &= 0 \qquad\qquad\qquad otherwise \end{aligned}$$

$$\begin{aligned} PMI(w,c) &= \log \frac{p(c|w)}{p(c)} \\ &= \log \frac{count(w,c) * N}{count(c) * count(w)} \end{aligned}$$

	human	machine	system	for	...	user
human	0	2.944	0	2.25	...	0
machine	2.944	0	0	2.25	...	0
system	0	0	0	1.15	...	1.84
for	2.25	2.25	1.15	0	...	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
user	0	0	1.84	0	...	0

New transformed table with PPMI values

With this method we were able to get some idea about context but still this won't be an ideal approach. Consider example 'cat' and 'dog' in corpus but they don't lie within the window gap. Clearly there is relation b/w cat and dog like both being pets, mammal etc. this method won't be able to give any insight about their relation.

In above case co-occurrence between {system, machine} and {human, user} is not visible. But, they are related to each other in above corpus's context.

Also, high dimensionality, sparse nature of matrix & redundancy(*as symmetric matrix*) still persists along with increase in size along with vocabulary.

Singular Value Decomposition

High dimensional problem are solved by PCA or by its generalized version SVD. SVD is a generalization of the eigen-decomposition of a positive semi-definite normal matrix to any matrix via an extension of the polar decomposition. It gives the best rank-k approximation of the original data. Let original data **X** be of dimension *m x n.* The singular value decomposition will break this into best rank approximation capturing information from most relevant to least relevant ones.

$$\begin{bmatrix} \uparrow & \cdots & \uparrow \\ & & \\ u_1 & \cdots & u_k \\ \downarrow & \cdots & \downarrow \end{bmatrix}_{m \times k} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{bmatrix}_{k \times k} \begin{bmatrix} \leftarrow & v_1^T & \rightarrow \\ & \vdots & \\ \leftarrow & v_k^T & \rightarrow \end{bmatrix}_{k \times n}$$

$$= \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_k u_k v_k^T$$

SVD theorem tells us that u1 ,v 1 and σ1 store the most important information in X. Subsequent terms stores less and less important information.

An analogy to this can be seen with with case of colors being represented with 8-bits. These 8-bits will provide more resolution to us. But, now we want to compress this into 4-bits only. Which bits should we retain, lower or higher ones?

$$\overbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{1}}^{verylight}\overbrace{\boxed{1}\boxed{0}\boxed{1}\boxed{1}}^{green}$$

$$\overbrace{\boxed{0}\boxed{0}\boxed{1}\boxed{0}}^{light}\overbrace{\boxed{1}\boxed{0}\boxed{1}\boxed{1}}^{green}$$

$$\overbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{0}}^{dark}\overbrace{\boxed{1}\boxed{0}\boxed{1}\boxed{1}}^{green}$$

$$\overbrace{\boxed{1}\boxed{0}\boxed{0}\boxed{0}}^{verydark}\overbrace{\boxed{1}\boxed{0}\boxed{1}\boxed{1}}^{green}$$

When the bits are reduced, most important information is to identify the color not the shades of different colors. Similar resolution will exist for different colors like Red, Blue, Yellow etc.

As by capturing information about different shades only there would be no meaning for colors because all information related to colors was lost. Hence, lower bits are the most important ones. Now, in this case SVD would have captured those lower bits.

With SVD, the latent co-occurrence between {system, machine} and {human, user} will become visible. We take the matrix product of **X** and **X***t*, whose ij-th entry is the dot product between the representation of word i (X[i :]) and word j (X[j :]). The ij-th entry of **X**, **X***t* roughly captures the cosine similarity between word i , word j.

	human	machine	system	for	...	user
human	0	2.944	0	2.25	...	0
machine	2.944	0	0	2.25	...	0
system	0	0	0	1.15	...	1.84
for	2.25	2.25	1.15	0	...	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
user	0	0	1.84	0	...	0

Co-occurrence Matrix (X)

	human	machine	system	for	...	user
human	2.01	2.01	0.23	2.14	...	0.43
machine	2.01	2.01	0.23	2.14	...	0.43
system	0.23	0.23	1.17	0.96	...	1.29
for	2.14	2.14	0.96	1.87	...	-0.13
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
user	0.43	0.43	1.29	-0.13	...	1.71

Low rank $X \rightarrow$ Low rank $\hat{X}$

From product of [XX*t] matrix a low rank matrix capturing important features is constructed.* the latent co-occurrence between {system, machine} and {human, user} has become visible. See, the red and blue parts in given figure.

We would want representations of *words(i, j)* to be of smaller dimensions but still have the same similarity(dot product) as the corresponding rows of **X***hat*. Hence, our search for finding more powerful representations must go on.

Also, notice that the dot product between the rows of the the matrix **W***word*=**UΣ** is the same as the dot product between the rows of **X***hat*.

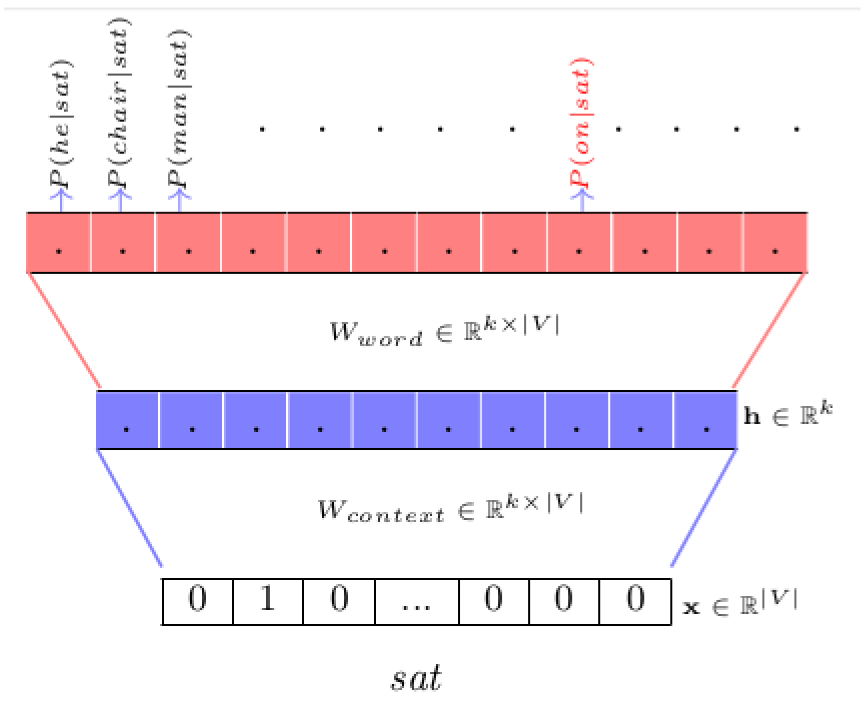
$$\begin{aligned} \hat{X}\hat{X}^T &= (U\Sigma V^T)(U\Sigma V^T)^T \\ &= (U\Sigma V^T)(V\Sigma U^T) \\ &= U\Sigma\Sigma^T U^T \quad (\because V^T V = I) \\ &= U\Sigma(U\Sigma)^T = W_{word}W_{word}^T \end{aligned}$$

W*word* = UΣ ∈ R m×k is taken as the representation of the m words in the vocabulary and Wcontext = V is taken as the representation of the context words.

Continuous Bag of Words

The methods we have seen are count based models like SVD as it uses co-occurrence count which uses the classical statistic based NLP principles. Now, we will move onto prediction based model which directly learn word representations. Consider a **task**, predict the n*th* word given previous *(n-1)* words. For training data all n-word window in training corpus can be used and corpus can be obtained from scraping any webpage.

Now, How to **model this task** of predicting *nth* word? & What's the connection between this task and learning word representations? For modelling this problem we will use feed-forward neural network as shown below.



$P(he|sat)$
$P(chair|sat)$
$P(man|sat)$
$P(on|sat)$
$W_{word} \in \mathbb{R}^{k \times |V|}$
$\mathbf{h} \in \mathbb{R}^{k}$
$W_{context} \in \mathbb{R}^{k \times |V|}$
0 1 0 … 0 0 0
$\mathbf{x} \in \mathbb{R}^{|V|}$
sat

One-hot representation as input. |V| words possible as output classes.

Our aim is to predict a probability distribution over these |V| classes as a multi-class classification problem. But this looks very complex and neural networks have very high number parameters. How is this a simpler approach?

For this, we need to look little bit behind the mathematics behind the vector multiplication happening in behind the scenes with this neural network. The product **W***context****x** given that **x** is a one hot vector.

$$\begin{bmatrix} -1 & 0.5 & 2 \\ 3 & -1 & -2 \\ -2 & 1.7 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -1 \\ 1.7 \end{bmatrix}$$

Is simply, the i-th column of Wcontext . A ont-to-one mapping exists b/w words & Wcontext’s columns.

We can treat the i-th column of **W***context* as the representation of context *i*. For P*(on|sat)* is proportional to the dot product between *jth column* of **W***context* and *ith column* of **W***word*. P*(word = i|sat)* thus depends on the *ith*

column of **W***word*. We thus treat the *ith column* of **W***word* as the representation of word *i*. This clearly shows the weight parameters are being represented as word vector representation in a neural network architecture.

Having understood the simplicity of interpretation behind the parameters, our aim now is to learn these parameters. For multi-class classification problem use **softmax** as output activation function and **cross-entropy** as loss function.

$$\mathscr{L}(\theta) = -\log \hat{y}_w = -\log P(w|c)$$

$$h = W_{context} \cdot x_c = u_c$$

$$\hat{y}_w = \frac{exp(u_c \cdot v_w)}{\sum_{w' \in V} exp(u_c \cdot v_{w'})}$$

uc is the column of Wcontext corresponding to context word c and vw is the column of Wword corresponding to the word w.

Let's see what we can interpret from update rule from above loss-function. Put value of *yhat* in loss-function and differentiate it to derive gradient update rule.

$$\begin{aligned} v_w &= v_w - \eta \nabla_{v_w} \\ &= v_w + \eta u_c (1 - \hat{y}_w) \end{aligned}$$

When yhat is 1, already corrected word predicted. Hence, no update. & when it is 0, vw gets updated by fraction of uc added.

This increases the cosine similarity between *vw* and *uc*. As, training objective ensures that the cosine similarity between word *(vw)* and context word *(uc)* is maximized. Hence, similarity measure is also captured by this representation. Also, neural network helps in learning much simpler and abstract vector representations of words.

In practice, more than one words are used in the window size, it common to use 'd' window size depending on the use-case. That would simply mean we have to stack copies of **Wcontext** in the bottom layers as two one-hot rod bits will be high.

$$\begin{bmatrix} -1 & 0.5 & 2 & -1 & 0.5 & 2 \\ 3 & -1.0 & -2 & 3 & -1.0 & -2 \\ -2 & 1.7 & 3 & -2 & 1.7 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{array}{l} \\ \} \, sat \\ \\ \\ \\ \} \, he \end{array}$$

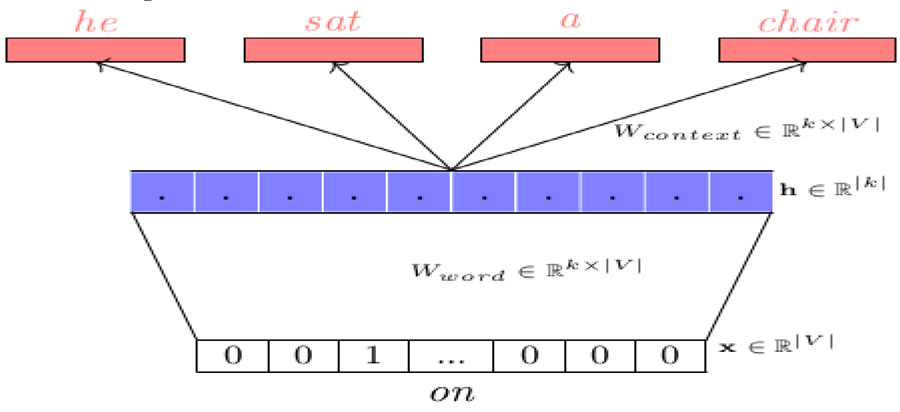
[W context , W context] just means that we are stacking 2 copies of W context matrix

Here, still computation bottleneck of softmax is present with denominator term involving summation over the entire vocabulary. We must explore some other model mitigating this bottleneck step.

Skip-Gram Model

This model predicts context words with respect to given input words. The role of context and word has changed to an almost opposite sense. Now, with given input words as one hot representations our aim is to predict context word related to it. This opposite relation b/w CBOW model & Skip-Gram model will become more clear below.

Given a corpus the model loops on the words of each sentence and either tries to use the current word of to predict its neighbors*(its context)*, in which case the model is called "Skip-Gram", or it uses each of these contexts to predict the current word, in which case the model is called "Continuous Bag Of Words" (CBOW).



he
sat
a
chair
$W_{context} \in \mathbb{R}^{k\times|V|}$
$\mathbf{h} \in \mathbb{R}^{|k|}$
$W_{word} \in \mathbb{R}^{k\times|V|}$
0 0 1 ... 0 0 0
$\mathbf{x} \in \mathbb{R}^{|V|}$
on

'on' as input word, probabilities of context words related to it are predicted by this network.

Train a simple neural network with a single hidden layer to perform a certain task, but then we're not actually going to use that neural network for the task we trained it on every single time for new task of modeling! Instead, the goal is actually just to learn the weights of the hidden layer which will be word vectors as stated by mathematics shown in above section.

In the simple case when there is only one context word, we will arrive at

the same update rule for **uc** as we did for **vw** earlier. If we have multiple context words the loss function would just be a summation of many cross-entropy errors.

$$\mathscr{L}(\theta) = -\sum_{i=1}^{d-1} \log \hat{y}_{w_i}$$

Again same issue as with CBOW, the problem of softmax being computationally expensive exists for this model also.

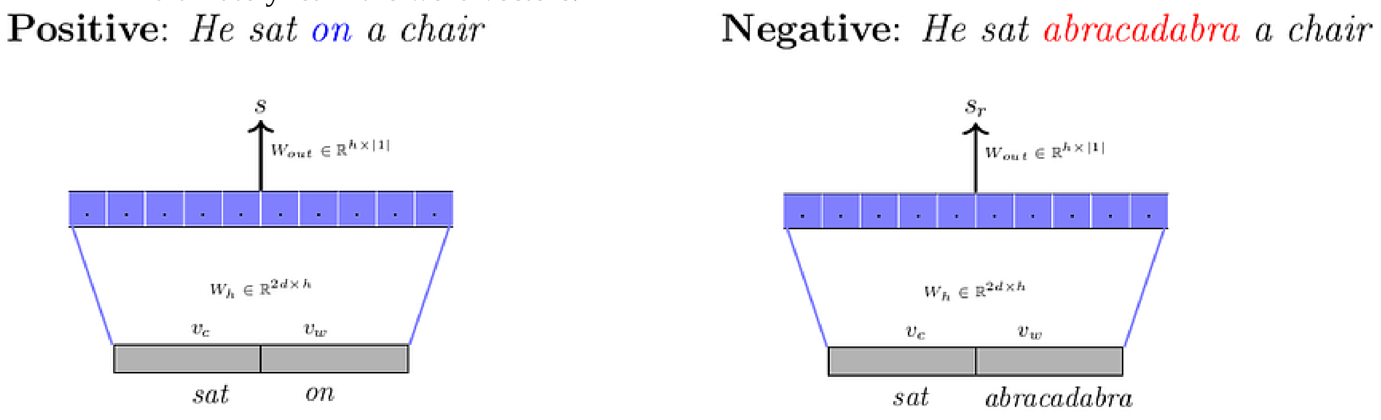
Three strategies namely negative sampling, contrastive estimation and hierarchical softmax can be used to eliminate this softmax problem of summation over entire vocabulary.

- **Negative Sampling:** We sample k negative (w, r) pairs with no context for every positive (w, c) context pairs. The size of D*dash* is thus k times the size of D. In our neural network we will define loss functions and train on both these sets. These corrupted pairs are drawn from a specially designed distribution, which favours less frequent words to be drawn more often.

$$\underset{\theta}{maximize} \prod_{(w,c)\in D} p(z=1|w,c) \prod_{(w,r)\in D'} p(z=0|w,r)$$

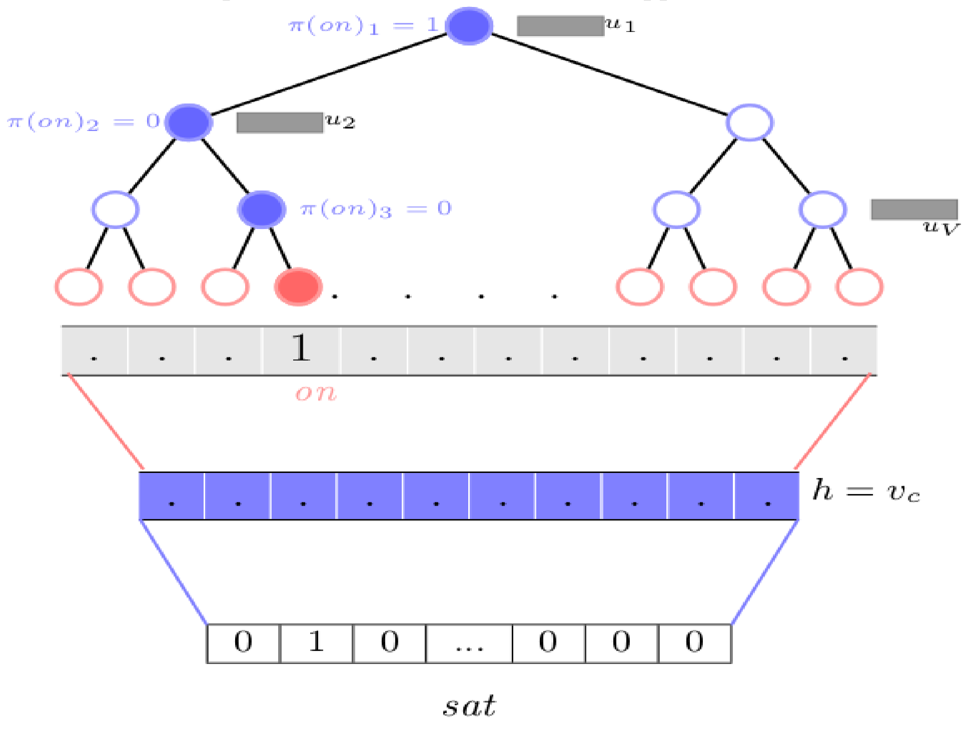
Summation over entire vocabulary get reduces as two different sets get created.

- **Contrastive Estimation:** The basic idea is to convert a multinomial classification problem(*as it is the problem of predicting the next word*) to a binary classification problem. Instead of using softmax to estimate a true probability distribution of the output word, a binary logistic regression (*binary classification*) is used instead i.e. the optimized classifier simply predicts whether a pair of words is good or bad. For each training sample, the enhanced (optimized) classifier is fed a true pair (*a center word and another word that appears in its context*) and a number of **k** randomly corrupted pairs (*consisting of the center word and a randomly chosen word from the vocabulary*). By learning to distinguish the true pairs from corrupted ones, the classifier will ultimately learn the word vectors.


Positive: He sat on a chair
s
$W_{out} \in \mathbb{R}^{h\times|1|}$
$W_h \in \mathbb{R}^{2d\times h}$
v_c
v_w
sat
on
Negative: He sat abracadabra a chair
s_r
$W_{out} \in \mathbb{R}^{h\times|1|}$
$W_h \in \mathbb{R}^{2d\times h}$
v_c
v_w
sat
abracadabra

maximize max(0, s − (s r + m))

- **Hierarchical softmax:** In this technique a binary tree is constructed such that there are |V| leaf nodes each corresponding to one word in the vocabulary. There exists a unique path from the root node to a leaf node. In effect, the task gets formulated to the probability of predicting a word is the same as predicting the correct unique path from the root node to that word. The above model ensures that the representation of a context word **v***c* will have a high(low) similarity with the representation of the node **u***i* if **u***i* appears on the path and the path branches to the left(right) at **u***i*. Representations of contexts which appear with the same words will have high similarity.



p(w|vc) can now be computed using |π(w)| computations instead of |V| required by softmax. Also, random arrangement of the words on leaf nodes does well in practice

GloVe Representations

Count based methods (SVD) rely on global co-occurrence counts from the

corpus for computing word representations. Predict based methods **learn** word representations using co-occurrence information. Why not combine the two **count** and **learn** mechanisms ?

Let’s formulate this idea mathematically and then develop an intuition for it. Let X*ij* encodes important global information about the co-occurrence between *i* and *j*.

$$P(j|i) = \frac{X_{ij}}{\sum X_{ij}} = \frac{X_{ij}}{X_i}$$

$$X_{ij} = X_{ji}$$

Our aim will be to learn word vectors which complies with computed probability on entire corpus. Essentially we are saying that we want word

vectors **v***i* and **v***j* such that **v***i*^T **v***j* is faithful to the globally computed P (j|i).

$$\begin{aligned} v_i^T v_j &= \log P(j|i) \\ &= \log X_{ij} - \log(X_i) \end{aligned}$$

Similar equation for Xj will be there. Also, Xij = Xji.

Add the two equations for **X***i* and **X***j* respectively. Also, log(**X***i*) and log(**X***j*) depend only on the words *i* & *j* and we can think of them as word specific biases which will be learned. Formulate this problem in following way.

$$v_i^T v_j = \log X_{ij} - b_i - b_j$$

$$v_i^T v_j + b_i + b_j = \log X_{ij}$$

$$\underset{v_i, v_j, b_i, b_j}{min} \sum_{i,j} (\underbrace{v_i^T v_j + b_i + b_j}_{\substack{\text{predicted value} \\ \text{using model} \\ \text{parameters}}} - \underbrace{\log X_{ij}}_{\substack{\text{actual value} \\ \text{computed from} \\ \text{the given corpus}}})^2$$

Now, the problem is weights of all the co-occurrences are equal. Weight should be defined in such a manner that neither rare or frequent words are over-weighted.

$$\underset{v_i, v_j, b_i, b_j}{min} \sum_{i,j} f(X_{ij})(v_i^T v_j + b_i + b_j - \log X_{ij})^2$$

$$f(x) = \left\{ \begin{array}{ll} (\frac{x}{x_{max}})^\alpha, & \text{if } x < x_{max} \\ 1, & \text{otherwise} \end{array} \right\}$$

Words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space. It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word. Those representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.

Conclusion: What to Chose ?

Considering the recent popular research papers. *Boroni et.al [2014]* showed that predict models consistently outperform count models in all tasks. *Levy et.al [2015]* do a much more through analysis (IMO) and show that good
old SVD does better than prediction based models on similarity tasks but not
on analogy tasks. Levy showed that word2vec also implicitly does a matrix factorization. It turns out that we can also show that

$$M = W_{context} * W_{word}$$

where

$$M_{ij} = PMI(w_i, c_i) - log(k)$$
$$k = \text{number of negative samples}$$

word2vec factorizes a matrix M which is related to the PMI based co-occurrence matrix. Very similar to what SVD does.

Good old SVD will just do fine. But, currently working with pre-trained Glove embeddings on huge corpuses results in much better results and once trained, can be reused again. Libraries like Keras, Gensim does provide embedding layers which can be used with ease for modelling tasks.

4.2.3Sequence-to-sequence model

What is seq2seq Model in Machine Learning?

Seq2seq was first introduced for machine translation, by Google. Before that, the translation worked in a very naïve way. Each word that you used to type was converted to its target language giving no regard to its grammar and sentence structure. Seq2seq revolutionized the process of translation by making use of deep learning. It not only takes the current word/input into account while translating but also its neighborhood.

Seq2Seq (Sequence-to-Sequence) is a type of model in machine learning that is used for tasks such as machine translation, text summarization, and image captioning. The model consists of two main components:

- Encoder
- Decoder

Seq2Seq models are trained using a dataset of input-output pairs, where the input is a sequence of tokens and the output is also a sequence of tokens. The model is trained to maximize the likelihood of the correct output sequence given the input sequence.

Seq2Seq models have been widely used in NLP tasks such as machine translation, text summarization, and image captioning, due to their ability to handle variable-length input and output sequences. Additionally, the Attention mechanism is often used in Seq2Seq models to improve performance and it allows the decoder to focus on specific parts of the input sequence when generating the output.

Nowadays, it is used for a variety of different applications such as image captioning, conversational models, text summarization, etc.

Encoder-Decoder Stack

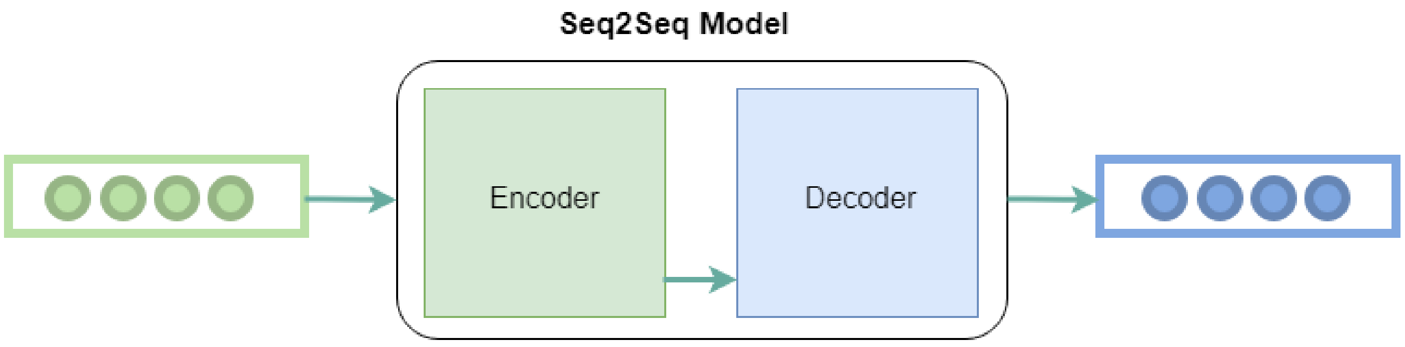
As the name suggests, seq2seq takes as input a sequence of words(sentence or sentences) and generates an output sequence of words. It does so by use of the recurrent neural network (RNN). Although the vanilla version of RNN is rarely used, its more advanced version i.e. LSTM or GRU is used. This is because RNN suffers from the problem of vanishing gradient. LSTM is used in the version proposed by Google. It develops the context of the word by taking 2 inputs at each point in time. One from the user and the other from its previous output, hence the name recurrent (output goes as input).
The encoder and decoder are typically implemented as Recurrent Neural Networks (RNNs) or Transformers.

Encoder Stack

It uses deep neural network layers and converts the input words to corresponding hidden vectors. Each vector represents the current word and the context of the word. The encoder takes the input sequence, one token at a time, and uses an RNN or transformer to update its hidden state, which summarizes the information in the input sequence. The final hidden state of the encoder is then passed as the context vector to the decoder.

Decoder Stack

It is similar to the encoder. It takes as input the hidden vector generated by the encoder, its own hidden states, and the current word to produce the next hidden vector and finally predict the next word. The decoder uses the context vector and an initial hidden state to generate the output sequence, one token at a time. At each time step, the decoder uses the current hidden state, the context vector, and the previous output token to generate a probability distribution over the possible next tokens. The token with the highest probability is then chosen as the output, and the process continues until the end of the output sequence is reached.



Encoder and Decoder Stack in seq2seq model

Components of seq2seq Model in Machine Learning

Apart from these two, many optimizations have led to other components of seq2seq:

- **Attention:** The input to the decoder is a single vector that has to store all the information about the context. This becomes a problem with large sequences. Hence the attention mechanism is applied which allows the decoder to look at the input sequence selectively.
- **Beam Search:** The highest probability word is selected as the output by the decoder. But this does not always yield the best results, because of the basic problem of greedy algorithms. Hence beam search is applied which suggests possible translations at each step. This is done by making a tree of top k-results.

- **Bucketing:** Variable-length sequences are possible in a seq2seq model because of the padding of 0's which is done to both input and output. However, if the max length set by us is 100 and the sentence is just 3 words long it causes a huge waste of space. So we use the concept of bucketing. We make buckets of different sizes like (4, 8) (8, 15), and so on, where 4 is the max input length defined by us and 8 is the max output length defined.

Advantages of seq2seq Models:

- **Flexibility**: Seq2Seq models can handle a wide range of tasks such as machine translation, text summarization, and image captioning, as well as variable-length input and output sequences.
- **Handling Sequential Data:** Seq2Seq models are well-suited for tasks that involve sequential data such as natural language, speech, and time series data.
- **Handling Context:** The encoder-decoder architecture of Seq2Seq models allows the model to capture the context of the input sequence and use it to generate the output sequence.
- **Attention Mechanism:** Using attention mechanisms allows the model to focus on specific parts of the input sequence when generating the output, which can improve performance for long input sequences.

Disadvantages of seq2seq Models:

- **Computationally Expensive:** Seq2Seq models require significant computational resources to train and can be difficult to optimize.
- **Limited Interpretability:** The internal workings of Seq2Seq models can be difficult to interpret, which can make it challenging to understand why the model is making certain decisions.
- **Overfitting**: Seq2Seq models can overfit the training data if they are not properly regularized, which can lead to poor performance on new data.
- **Handling Rare Words:** Seq2Seq models can have difficulty handling rare words that are not present in the training data.
- **Handling Long input Sequences:** Seq2Seq models can have difficulty handling input sequences that are very long, as the context vector may not be able to capture all the information in the input sequence.