Natural Language Processing (NLP)

Master IASD

Pr. Lotfi ELAACHAK
Département Génie Informatique

2023 — 2024

Table des matières

1 Basics of NLP **3**

1.1 Introduction . . . 3
1.2 NLP Tasks . . . 4
1.3 NLP Pipline . . . 4
1.3.1 NLP Libraries . . . 5
1.3.2 Data Acquisition . . . 5
1.3.3 Text Cleaning . . . 6
1.3.4 Text Preprocessing . . . 7

2 Regex and Rule Based Approach in NLP **15**

2.1 Regex . . . 15
2.1.1 How to write regular expressions . . . 15
2.1.2 Regular Expression in Python . . . 17
2.2 Rule Based Approach in NLP . . . 18
2.2.1 Matching Token with Rule-based Approach . . . 20
2.2.2 Matching Phrases with the Rule-based Approach . . . 21
2.2.3 Named Entity Recognization with Spacy . . . 22

3 Text Vectorization / Encoding / Word Embedding **23**

3.1 Hot encoding . . . 23
3.1.1 The advantages of using one hot encoding include . . . 23
3.1.2 The disadvantages of using one hot encoding include . . . 23
3.1.3 One Hot Encoding Examples . . . 23
3.1.4 One-Hot Encoding Using Python . . . 24
3.2 Bag of words . . . 25
3.2.1 Example of the Bag-of-Words Model . . . 25
3.2.2 Python Bag of words . . . 26
3.3 TF/IDF . . . 27
3.3.1 TF-IDF Python . . . 28
3.4 Word Embedding . . . 29
3.4.1 Word Embedding . . . 29
3.4.2 For example . . . 29
3.4.3 Word2Vec . . . 30
3.4.4 FastText . . . 42
3.4.5 Glove . . . 43

4 Language Modeling : Machine Learning **46**

4.1 Naive Bayes . . . 46
4.1.1 Generative Vs Discriminative Machine learning . . . 46

4.1.2 Naive Bayes Algorithm . . . 47
4.1.3 Naive Bayes and NLP . . . 49
4.2 Support Vector Machine . . . 50
4.2.1 Types of SVM . . . 51
4.2.2 Hyperplane . . . 51
4.2.3 How to find the best Hyperplane . . . 52
4.2.4 Linear SVM : Primal . . . 53
4.2.5 Linear SVM : Dual . . . 54
4.2.6 Non Linear SVM : Kernel Methods . . . 62
4.2.7 SVM and NLP . . . 65

5 Language Modeling : Deep Learning 67
5.1 Artificial Neural Network . . . 67
5.1.1 Perceptron . . . 67
5.1.2 Multi layer Perceptron . . . 72
5.1.3 Loss function . . . 73
5.1.4 Activation functions . . . 75
5.1.5 Optimization . . . 78
5.1.6 Regularization . . . 82
5.1.7 L1 and L2 Regularzation . . . 84
5.1.8 Early Stop . . . 85
5.1.9 Dropout . . . 85
5.1.10 DNN based on pytorch . . . 85
5.2 RNN / GRU / LSTM . . . 96
5.2.1 RNN . . . 97
5.2.2 GRU Unit . . . 101
5.2.3 LSTM . . . 101
5.2.4 LSTM/GRU/RNN based on pytorch . . . 102
5.3 Attention / Transformers . . . 110
5.3.1 Attention ALL you need . . . 110
5.3.2 Transformers Architecture . . . 112
5.3.3 Multi-Head Attention . . . 115
5.3.4 Transformers Implementation . . . 119

Chapitre 1

Basics of NLP

1.1 Introduction

Natural Language Processing (NLP) is a subfield of artificial intelligence (AI) that focuses on equipping machines with the ability to understand, analyze, and generate human languages. NLP utilizes various algorithms and techniques to enable computers to interact with and make sense of textual or spoken language data.

NLP is a higher-level term and is the combination of Natural Language Understanding (NLU) and Natural Language Generation (NLG).

While the origins of AI can be traced back to the 1950s, the development of NLP as a distinct field began in the same era. In the 1940s and 1950s, researchers started exploring ways to leverage computers for natural language processing and understanding.

A significant milestone in the early development of NLP was the Georgetown-IBM Experiment in 1954, which marked the first attempt at machine translation. This experiment laid the groundwork for further advancements in automatic language processing and set the stage for NLP's future growth.

As the technology evolved, different approaches have come to deal with NLP tasks :

— Heuristics-Based NLP : This is the initial approach of NLP. It is based on defined rules. Which comes from domain knowledge and expertise. Example : regex

— Machine learning-based NLP : It is based on statistical rules and machine learning algorithms. In this approach, algorithms are applied to the data and learned from the data, and applied to various tasks. Examples : Naive Bayes, support vector machine (SVM), hidden Markov model (HMM), etc.

— Neural Network-based NLP : This is the latest approach that comes with the evaluation of neural network-based learning, known as Deep learning. It provides good accuracy, but it is a very data-hungry and time-consuming approach. It requires high computational power to train the model. Furthermore, it is based on neural network architecture. Examples : Recurrent neural networks (RNNs), Long short-term memory networks (LSTMs), Convolutional neural networks (CNNs), Transformers, etc.

1.2 NLP Tasks

Several tasks related to text/speech and audio can be handled with NLP, the list bellow shows some of the common tasks of NLP.

1. Information retrieval
2. Named entity recognition
3. Relation extraction
4. Text classification/Document Classification
5. Document Ranking
6. Annotation
7. Topic modelling
8. Keyword Extraction
9. Machine translation
10. Parts of speech tagging
11. Semantic Role Labeling
12. Word Sense Disambiguation
13. Grammatical Error Correction
14. Semantic textual similarity
15. Text summarization/Meeting Summarization
16. Reading comprehension
17. Question and answering
18. Question Generation
19. Image captioning
20. Fake News Detection/Hate Speech Detection
21. Text generation
22. Sentiment/emotion analysis
23. Speech-to-text
24. Text-to-speech
25. Dialogue Understanding

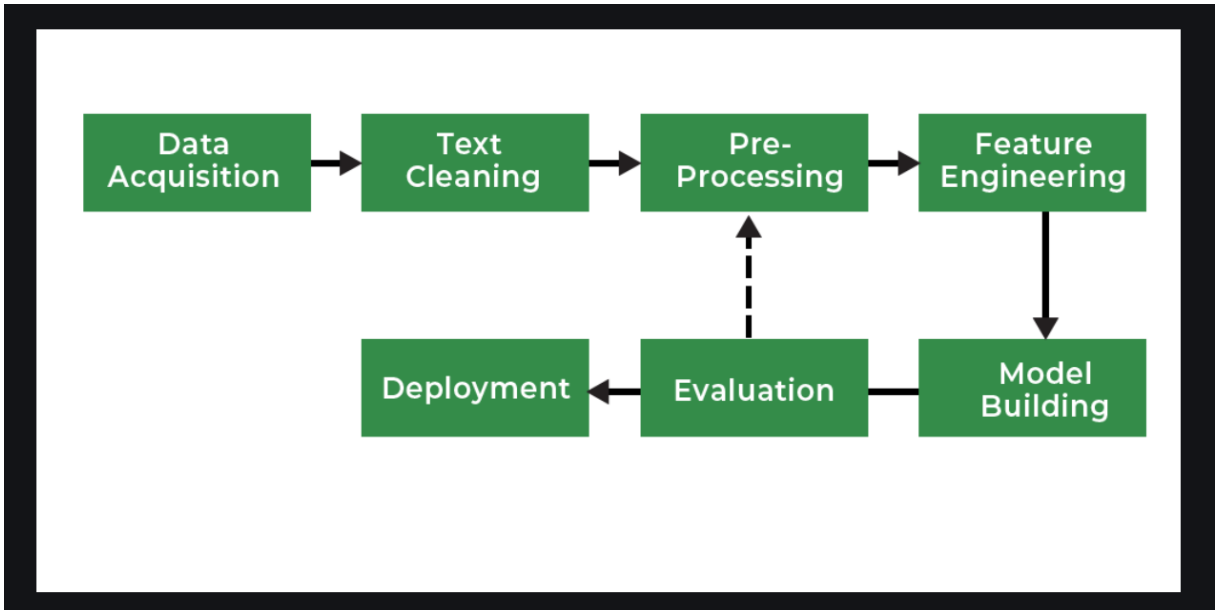
1.3 NLP Pipline

The natural language processing (NLP) pipeline refers to the sequence of processes involved in analyzing and understanding human language. The following is a typical NLP pipeline :

— Text & Speech processing
— Sentiment analysis
— Information Extraction
— Text Summarization
— Text generation
— Automatic Question Answering (chat-bot)
— Language Translation

The basic processes for all the above tasks are the same. In NLP we need to perform some extra processing steps. The region is very simple that machines don't understand the text. Here our biggest problem is How to make the text understandable for machines. Some of the most common problems we face while performing NLP tasks are mentioned below.

1. Data Acquisition
2. Text Cleaning
3. Text Preprocessing
4. Feature Engineering
5. Model Building
6. Evaluation
7. Deployment



1.3.1 NLP Libraries

— NLTK : https://www.nltk.org/
— Spacy : https://spacy.io/
— Gensim : https://github.com/RaRe-Technologies/gensim
— fastText : https://fasttext.cc/
— Stanford toolkit (Glove) : https://nlp.stanford.edu/projects/glove/
— Apache OpenNLP : https://opennlp.apache.org/

1.3.2 Data Acquisition

Text data is available on websites, in emails, in social media, in form of pdf, and many more. But the challenge is. Is it in a machine-readable format ? if in the machine-readable format then will it be relevant to our problem ? So, First thing we need to understand our problem or task then we should search for data. Here we will see some of the ways of collecting data if it is not available in our local machine or database.

— Public Dataset : Search for publicly available data (Kaggle , Data Set Repositories, etc).
— Web Scrapping : Web Scrapping is a technique to scrap data from a website. (we can use Beautiful Soup/ Scrapy to scrape the text data from the web page.)

— Image to Text : Scrap the data from the image files with the help of Optical character recognition (OCR). There is a library Tesseract that uses OCR to convert image to text data.
— pdf to Text : There are multiple Python packages to convert the data into text. With the PyPDF2 library, pdf data can be extracted in the .text file.
— Data augmentation : if the acquired data is not very sufficient for our problem statement then we can generate fake data from the existing data by Synonym replacement, Back Translation, Bigram flipping, or Adding some noise in data. This technique is known as Data augmentation.

1.3.3 Text Cleaning

Sometimes our acquired data is not very clean. it may contain HTML tags, spelling mistakes, or special characters. So, let's see some techniques to clean our text data.

Unicode Normalization

if text data may contain symbols, emojis, graphic characters, or special characters. Either we can remove these characters or we can convert this to machine-readable text.

Unicode Normalization

```
# Unicode Nomalization
text = "GeeksForGeeks ????"
print(text.encode('utf-8'))
```

Regex or Regular Expression

Regular Expression is the tool that is used for searching the string of specific patterns. Suppose our data contain phone number, email-Id, and URL. we can find such text using the regular expression. After that either we can keep or remove such text patterns as per requirements.

Regex

```
import re
text = """<gfg>
#GFG Geeks Learning together
url <https://www.rgex.org/>,
email <acs@sdf.dv>
"""
def clean_text(text):
        # remove HTML TAG
        html = re.compile('[<,#*?>]')
        text = html.sub(r'',text)
        # Remove urls:
        url = re.compile('https?://\S+|www\.S+')
        text = url.sub(r'',text)
```

```
        # Remove email id:
        email = re.compile('[A-Za-z0-2]+@[\w]+.[\w]+')
        text = email.sub(r'',text)
        return text

print(clean_text(text))
```

Spelling corrections

When our data is extracted from social media. Spelling mistakes are very common in that case. To overcome this problem we can create a corpus or dictionary of the most common mistype words and replace these common mistakes with the correct word.

Spelling corrections

```
text = "As far as I am abl to judg, after long attnding to the sbject, the
    condiions of lfe apear to act in two ways-directly on the whle organsaton or on
    certin parts alne and indirectly by afcting the reproducte sstem."

from textblob import TextBlob
textBlb = TextBlob(text)            # Making our first textblob
textCorrected = textBlb.correct()   # Correcting the text
print(textCorrected)
```

1.3.4 Text Preprocessing

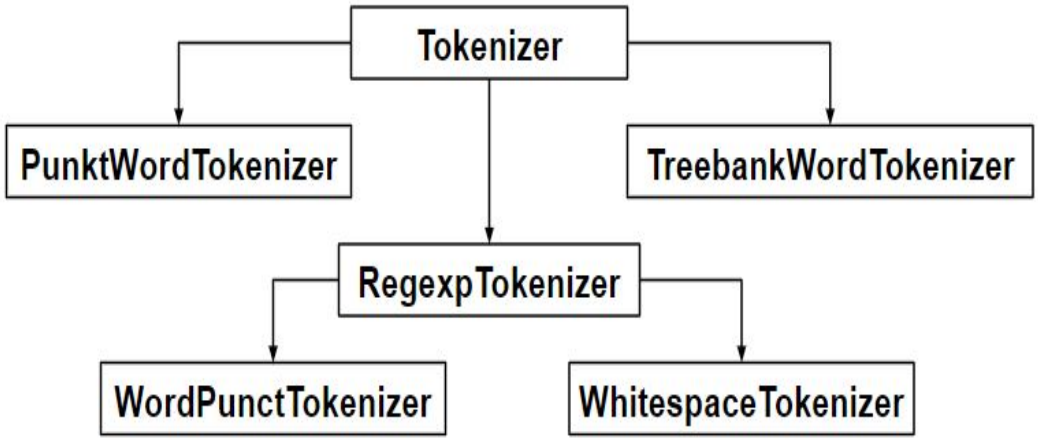
Text preprocessing is an essential step in natural language processing (NLP) that involves cleaning and transforming unstructured text data to prepare it for analysis. It includes tokenization, stemming, lemmatization, stop-word removal, and part-of-speech tagging.

Tokenization

Tokenization is the process of tokenizing or splitting a string, text into a list of tokens. One can think of token as parts like a word is a token in a sentence, and a sentence is a token in a paragraph.

What is a Token ?
As described previously, a token is a piece of data that stands in for another, more valuable piece of information. Tokens have virtually no value on their own they are only useful because they represent something valuable, such as a credit card primary account number (PAN) or Social Security number (SSN).

A tokenizer is in charge of preparing the inputs for a model. The library contains tokenizers for all the models.



Sentence Tokenization – Splitting sentences in the paragraph

Sentence Tokenization

```
from nltk.tokenize import sent_tokenize

text = "Hello everyone. Welcome to Master IASD. You are studying NLP article"
sent_tokenize(text)
```

Word Tokenization – Splitting words in a sentence.

Word Tokenization

```
from nltk.tokenize import word_tokenize

text = "Hello everyone. Welcome to FSTT."
word_tokenize(text)
```

Word Tokenization – Using TreebankWordTokenizer

Word Tokenization (TreebankWordTokenizer)

```
from nltk.tokenize import TreebankWordTokenizer

tokenizer = TreebankWordTokenizer()
tokenizer.tokenize("Let's see how it's working.")
```

These tokenizers work by separating the words using punctuation and spaces.

Word Tokenization – Using PunktWordTokenizer

Word Tokenization (PunktWordTokenizer)

```
from nltk.tokenize import WordPunctTokenizer
```

```
tokenizer = WordPunctTokenizer()
tokenizer.tokenize("Let's see how it's working.")
```

PunktWordTokenizer doesn't separates the punctuation from the words.

Word Tokenization – Using Regular Expression

Word Tokenization (Regular Expression)

```
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer("[\w']+")
text = "Let's see how it's working."
tokenizer.tokenize(text)
```

Lower Casing

Python String lower() method converts all uppercase characters in a string into lowercase characters and returns it.

Lower Casing

```
text = "TeST 09FSTT YUP{"

print("Original String:")
print(text)

# lower() function to convert
# string to lower_case
print("\nConverted String:")
print(text.lower())

# Convert uppercase to lowercase in python using swapcase function.
print(s.swapcase())# change case lower to upper and vice versa

# Convert uppercase to lowercase in python using casefold function.
print(s.casefold())
```

Stop words

Stop Words : A stop word is a commonly used word (such as "the", "a", "an", "in") that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.

Stop words

```
import nltk
from nltk.corpus import stopwords
```

```
nltk.download('stopwords')
print(stopwords.words('english'))

stop_words = set(stopwords.words('english'))

example_sent = """This is a sample sentence
                  showing off the stop words filtration."""

from nltk.tokenize import WordPunctTokenizer

tokenizer = WordPunctTokenizer()
word_tokens = tokenizer.tokenize(example_sent)
filtered_sentence = [w for w in word_tokens if not w.lower() in stop_words]
filtered_sentence = []

for w in word_tokens:
    if w not in stop_words:
        filtered_sentence.append(w)

print(word_tokens)
print(filtered_sentence)
```

Stemming

Stemming is the process of producing morphological variants of a root/base word. Stemming programs are commonly referred to as stemming algorithms or stemmers. A stemming algorithm reduces the words "chocolates", "chocolatey", and "choco" to the root word, "chocolate" and "retrieval", "retrieved", "retrieves" reduce to the stem "retrieve".

Stemming generates the base word from the inflected word by removing the affixes of the word. It has a set of pre-defined rules that govern the dropping of these affixes. It must be noted that stemmers might not always result in semantically meaningful base words. Stemmers are faster and computationally less expensive than lemmatizers.

Stemming

```
nltk.download('wordnet')
rom nltk.stem import PorterStemmer

# create an object of class PorterStemmer
porter = PorterStemmer()
print(porter.stem("play"))
print(porter.stem("playing"))
print(porter.stem("plays"))
print(porter.stem("played"))

# create an object of class PorterStemmer
```

```
porter = PorterStemmer()
print(porter.stem("Communication"))
```

Lemmatization

Lemmatization is the process of reducing a word to its base or dictionary form, known as the lemma. For example, the lemma of the word "cats" is "cat", and the lemma of "running" is "run".

Lemmatization involves grouping together the inflected forms of the same word. This way, we can reach out to the base form of any word which will be meaningful in nature. The base from here is called the Lemma.

Lemmatizers are slower and computationally more expensive than stemmers.

Lemmatization

```
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("plays", 'v'))
print(lemmatizer.lemmatize("played", 'v'))
print(lemmatizer.lemmatize("play", 'v'))
print(lemmatizer.lemmatize("playing", 'v'))


print("rocks :", lemmatizer.lemmatize("rocks"))
print("corpora :", lemmatizer.lemmatize("corpora"))

# a denotes adjective in "pos"
print("better :", lemmatizer.lemmatize("better", pos ="a"))
```

Parts of Speech (POS)

Part-of-speech (POS) tagging is an important Natural Language Processing (NLP) concept that categorizes words in the text corpus with a particular part of speech tag (e.g., Noun, Verb, Adjective, etc.).

Part-of-speech (POS) tagging is the process of labeling words in a text with their corresponding parts of speech in natural language processing (NLP). It helps algorithms understand the grammatical structure and meaning of a text.

POS tagging refers to assigning each word of a sentence to its part of speech. It is significant as it helps give a better syntactic overview of a sentence.

— CC coordinating conjunction
— CD cardinal digit
— DT determiner

— EX existential there (like : 'there is' ... think of it like 'there exists')
— FW foreign word
— IN preposition/subordinating conjunction
— JJ adjective – 'big'
— JJR adjective, comparative - 'bigger'
— JJS adjective, superlative - 'biggest'
— LS list marker 1)
— MD modal – could, will
— NN noun, singular – desk-
— NNS noun plural – 'desks'
— NNP proper noun, singular – 'Harrison'
— NNPS proper noun, plural – 'Americans'
— PDT predeterminer – 'all the kids'
— POS possessive ending parent's
— PRP personal pronoun – I, he, she
— PRP$ possessive pronoun – my, his, hers
— RB adverb – very, silently,
— RBR adverb, comparative – better
— RBS adverb, superlative – best
— RP particle – give up
— TO – to go 'to' the store.
— UH interjection – errrrrrrrm
— VB verb, base form – take
— VBD verb, past tense – took
— VBG verb, gerund/present participle – taking
— VBN verb, past participle – taken
— VBP verb, sing. present, non-3d – take
— VBZ verb, 3rd person sing. present – takes
— WDT wh-determiner – which
— WP wh-pronoun – who, what
— WP$ possessive wh-pronoun, eg- whose
— WRB wh-adverb, eg- where, when

POS Tag

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
stop_words = set(stopwords.words('english'))

// Dummy text
txt = "Sukanya, Rajib and Naba are my good friends. " \
        "Sukanya is getting married next year. " \
        "Marriage is a big step in one's life." \
        "It is both exciting and frightening. " \
        "But friendship is a sacred bond between people." \
        "It is a special kind of love between us. " \
        "Many of you must have tried searching for a friend "\
        "but never found the right one."

# sent_tokenize is one of instances of
```

```
# PunktSentenceTokenizer from the nltk.tokenize.punkt module

tokenized = sent_tokenize(txt)
for i in tokenized:

        # Word tokenizers is used to find the words
        # and punctuation in a string
        wordsList = nltk.word_tokenize(i)

        # removing stop words from wordList
        wordsList = [w for w in wordsList if not w in stop_words]

        # Using a Tagger. Which is part-of-speech
        # tagger or POS-tagger.
        tagged = nltk.pos_tag(wordsList)

        print(tagged)
```

Named Entity Recognition(NER)

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.

Some of the categories that are the most important architecture in NER such that :

- Person
- Organization
- Place/ location

Other common tasks include classifying of the following :

- date/time.
- expression
- Numeral measurement (money, percent, weight, etc)
- E-mail address

Methods of NER :

- One way is to train the model for multi-class classification using different machine learning algorithms, but it requires a lot of labelling. In addition to labelling the model also requires a deep understanding of context to deal with the ambiguity of the sentences. This makes it a challenging task for a simple machine learning algorithm.
- Another way is that Conditional random field that is implemented by both NLP Speech Tagger and NLTK. It is a probabilistic model that can be used to model sequential data such as words. The CRF can capture a deep understanding of the context of the sentence.
- Deep Learning Based NER : deep learning NER is much more accurate than previous method, as it is capable to assemble words. This is due to the fact that it used a method

called word embedding, that is capable of understanding the semantic and syntactic relationship between various words. It is also able to learn analyzes topic-specific as well as high level words automatically.

NER Tag

```
import nltk
nltk.download('words')
sentence = """Born and raised in Madeira, Ronaldo began his senior club career
    playing for Sporting CP,
before signing with Manchester United in 2003, aged 18. After winning the FA Cup in
     his first season,
he helped United win three successive Premier League titles, the UEFA Champions
    League, and the FIFA Club World Cup"""

from nltk import word_tokenize, pos_tag, ne_chunk

tokens = word_tokenize(sentence)
print(tokens)

pos_tags = pos_tag(tokens)
print(pos_tags)

named_entities = ne_chunk(pos_tags)
print(named_entities)
```

Chapitre 2

Regex and Rule Based Approach in NLP

2.1 Regex

A regular expression (regex) is a sequence of characters that define a search pattern. Here's how to write regular expressions :

1. Start by understanding the special characters used in regex, such as ".", "*", "+", " ?", and more.
2. Choose a programming language or tool that supports regex, such as Python, Perl, or grep.
3. Write your pattern using the special characters and literal characters.
4. Use the appropriate function or method to search for the pattern in a string.

Example : Regular expression for an email address :
`^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$`

2.1.1 How to write regular expressions

*Repeaters (*, +, and { })*

These symbols act as repeaters and tell the computer that the preceding character is to be used for more than just one time.

*The asterisk symbol (*) :*

It tells the computer to match the preceding character (or set of characters) for 0 or more times (upto infinite).Example : The regular expression ab*c will give ac, abc, abbc, abbbc. . . .and so on.

The Plus symbol (+) :

It tells the computer to repeat the preceding character (or set of characters) at atleast one or more times(up to infinite). Example : The regular expression ab+c will give abc, abbc,abbc, . . . and so on.

The curly braces { . . . } :

It tells the computer to repeat the preceding character (or set of characters) for as many times as the value inside this bracket. Example : 2 means that the preceding character is to be repeated 2.

Wildcard (.) :

The dot symbol can take the place of any other symbol, that is why it is called the wildcard character. Example : The Regular expression .* will tell the computer that any character.

Optional character (?) :

This symbol tells the computer that the preceding character may or may not be present in the string to be matched. Example : We may write the format for document file as – "docx ?" The ' ?' tells the computer that x may or may not be present in the name of file format.

The caret ($\hat{}$) symbol (Setting position for the match) :

The caret symbol tells the computer that the match must start at the beginning of the string or line. Example : `^\d{3}` will match with patterns like "901" in "901-333-".

The dollar ($) symbol :

It tells the computer that the match must occur at the end of the string or before
n at the end of the line or string.Example : `-\d{3}\$` will match with patterns like "-333" in "-901-333".

Character Classes :

A character class matches any one of a set of characters. It is used to match the most basic element of a language like a letter, a digit, a space, a symbol, etc.

- `\s` : matches any whitespace characters such as space and tab.
- `\S` : matches any non-whitespace characters.
- `\d` :matches any digit character.
- `\D` : matches any non-digit characters.
- `w` : matches any word character (basically alpha-numeric)
- `\W` :matches any non-word character.
- `\b` :matches any word boundary (this would include spaces, dashes, commas, semi-colons, etc.

[$\hat{}$set_of_characters] Negation :

Matches any single character that is not in set_of_characters. By default, the match is case-sensitive. Example : `[^abc]` will match any character except a,b,c .

[first-last] Character range :

Matches any single character in the range from first to last. Example : [a-zA-z] will match any character from a to z or A to Z.

The Escape Symbol (/) :

If you want to match for the actual '+', '.' etc characters, add a backslash( ) before that character. This will tell the computer to treat the following character as a search character and consider it for a matching pattern. Example : `\d+[\+-x\*]\d+` will match patterns like "2+2" and "3*9" in "(2+2) * 3*9".

Grouping Characters ( ) :

A set of different symbols of a regular expression can be grouped together to act as a single unit and behave as a block, for this, you need to wrap the regular expression in the parenthesis(

). Example : `([A-Z]\w+)` contains two different elements of the regular expression combined together. This expression will match any pattern containing uppercase letter followed by any character.

Vertical Bar (|) :

Matches any one element separated by the vertical bar (|) character. Example : th(e|is|at) will match words - the, this and that.

2.1.2 Regular Expression in Python

Regex Python

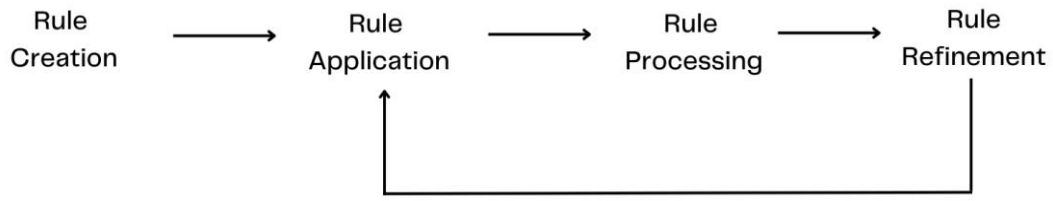
```
import re

s = 'Master: A computer science portal for students'

match = re.search(r'portal', s)

print('Start Index:', match.start())
print('End Index:', match.end())



string = "The quick brown fox jumps over the lazy dog"
pattern = "[a-m]"
result = re.findall(pattern, string)

print(result)

# Match strings starting with "The"
regex = r'^The'
strings = ['The quick brown fox', 'The lazy dog', 'A quick brown fox']
for string in strings:
    if re.match(regex, string):
        print(f'Matched: {string}')
    else:
        print(f'Not matched: {string}')

string = """Hello my Number is 123456789 and
            my friend's number is 987654321"""

# A sample regular expression to find digits.
regex = '\d+'

match = re.findall(regex, string)
print(match)
```

```
# \w is equivalent to [a-zA-Z0-9_].
p = re.compile('\w')
print(p.findall("He said * in some_lang."))

# \w+ matches to group of alphanumeric character.
p = re.compile('\w+')
print(p.findall("I went to him at 11 A.M., he \
said *** in some_language."))

# \W matches to non alphanumeric characters.
p = re.compile('\W')
print(p.findall("he said *** in some_language."))

# '*' replaces the no. of occurrence
# of a character.
p = re.compile('ab*')
print(p.findall("ababbaabbb"))


from re import split

# '\W+' denotes Non-Alphanumeric Characters
# or group of characters Upon finding ','
# or whitespace ' ', the split(), splits the
# string from that point
print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))

# Here ':', ' ' ,',' are not AlphaNumeric thus,
# the point where splitting occurs
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))

# '\d+' denotes Numeric Characters or group of
# characters Splitting occurs at '12', '2016',
# '11', '02' only
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

2.2 Rule Based Approach in NLP

Rule-based approach is one of the oldest NLP methods in which predefined linguistic rules are used to analyze and process textual data. Rule-based approach involves applying a particular set of rules or patterns to capture specific structures, extract information, or perform tasks such as text classification and so on. Some common rule-based techniques include regular expressions and pattern matches.

Steps in Rule-based approach in NLP :



1. Rule Creation : Based on the desired tasks, domain-specific linguistic rules are created such as grammar rules, syntax patterns, semantic rules or regular expressions.
2. Rule Application : The predefined rules are applied to the inputted data to capture matched patterns.
3. Rule Processing : The text data is processed in accordance with the results of the matched rules to extract information, make decisions or other tasks.
4. Rule refinement : The created rules are iteratively refined by repetitive processing to improve accuracy and performance. Based on previous feedback, the rules are modified and updated when needed.

A rule-matching engine in Spacy called the Matcher can work over tokens, entities, and phrases in a manner similar to regular expressions.

Advantages of the Rule-based approach

1. Easily interpretable as rules are explicitly defined
2. Rule-based techniques can help semi-automatically annotate some data in domains where you don't have annotated data (for example, NER(Named Entity Recognization) tasks in a particular domain).
3. Functions even with scant or poor training data
4. Computation time is fast and it offers high precision
5. Many times, deterministic solutions to various issues, such as tokenization, sentence breaking, or morphology, can be achieved through rules (at least in some languages).

Disadvantages of the Rule-based approach

1. Labor-intensive as more rules are needed to generalize
2. Generating rules for complex tasks is time-consuming
3. Needs regular maintenance
4. May not perform well in handling variations and exceptions in language usage
5. May not have a high recall metric.

Why Rule-based Approach with Machine Learning and Neural Network Approaches ?

1. Rule-based NLP usually deals with edge cases when included with other approaches.
2. It helps to speed up the data annotation. For instance, a rule-based technique is used for URL formats, date formats, etc., and a machine learning approach can be used to determine the position of text in a pdf file (including numerical data).
3. Also, in languages other than English annotated data is really scarce even for common tasks which are carried out by Rule-based NLP.
4. By using a rule-based approach, the computation performance of the pipeline is also improved.

Spacy

spaCy is a free open-source library for Natural Language Processing in Python. It features NER, POS tagging, dependency parsing, word vectors and more.

— !pip install - U spacy
— !pip install - U spacy-lookups-data
— !python - m spacy download en_core_web_sm # For English language

2.2.1 Matching Token with Rule-based Approach

Token Rule-based

```
#import modules
import spacy
#import the Matcher
from spacy.matcher import Matcher
#import the Span class
from spacy.tokens import Span


spacy = spacy.load("en_core_web_sm")

#The input text as a Document object
txt ="Natural Language Processing serves as an interrelationship between human
    language and computers. Natural Language Processing is a subfield of Artificial
    Intelligence that helps machines process, understand and generate natural
    language intuitively."

doc = spacy(txt)
Tokens = []
for token in doc:
    Tokens.append(token)

print('Tokens:',Tokens)
print('Number of token :',len(Tokens))
```

```
#Matcher class object instantiation
matcher = Matcher(spacy.vocab)

#pattern to be searched
pattern = [[{'LOWER': 'language'}],[{'LOWER':'human'}]]

#adding the pattern/rule to the matcher object
matcher.add("TokenMatch",pattern)

#Matcher object called
#returns match_id, start and stop indexes of the matched words
matches = matcher(doc)

#Extracting matched results
for m_id, start, end in matches:
        string_id = spacy.vocab.strings[m_id]
        span = doc[start:end]
        print('match_id:{}, string_id:{}, Start:{}, End:{}, Text:{}'.format(
                m_id, string_id, start, end, span.text)
                )
```

2.2.2 Matching Phrases with the Rule-based Approach

Phrases Rule-based

```
#import modules
import spacy
from spacy.matcher import PhraseMatcher

spacy = spacy.load("en_core_web_sm")

#The input text as a Document object
txt ="Natural Language Processing serves as an interrelationship between human
    language and computers. Natural Language Processing is a subfield of Artificial
    Intelligence that helps machines process, understand and generate natural
    language intuitively."
doc = spacy(txt)
print(doc)


# PhraseMatcher object creation
matcher = PhraseMatcher(spacy.vocab, attr='LOWER')


# list of phrases
```

```
term_list = ["Language Processing", "human language"]
# phrases into document object
patterns = [spacy.make_doc(t) for t in term_list]

# patterns added to the matcher object
matcher.add("Phrase Match", None, *patterns)


# Matcher object called. It returns Span objects directly
matches = matcher(doc, as_spans=True)
#Extracting matched results
for span in matches:
        print(span.text,":-", span.label_)
```

2.2.3 Named Entity Recognization with Spacy

NER Rule-based

```
# import spacy
import spacy
#Load the English Language Spacy model
nlp = spacy.load("en_core_web_sm")

#The input text as a Document object
txt = """
My name is Pawan Kumar Gunjan. I live in India
India, officially the Republic of India, is a country in South Asia.
It is the seventh-largest country by area and the second-most populous country.
Bounded by the Indian Ocean on the south, the Arabian Sea on the southwest,
and the Bay of Bengal on the southeast, it shares land borders with Pakistan to the
    west;
China, Nepal, and Bhutan to the north; and Bangladesh and Myanmar to the east.
"""
doc = nlp(txt)
Tokens = []
for entity in doc.ents:
        print('Text:{}, Label:{}'.format(entity.text, entity.label_))
```

Chapitre 3

Text Vectorization / Encoding / Word Embedding

Text vectorization is representing words, sentences, or even larger units of text as vectors(or vector embeddings).
Word Embeddings are the texts converted into numbers and there may be different numerical representations of the same text.

3.1 Hot encoding

In this technique, we represent each unique word in vocabulary by setting a unique token with value 1 and rest 0 at other positions in the vector. In simple words, a vector representation of a one-hot encoded vector represents in the form of 1, and 0 where 1 stands for the position where the word exists and 0 everywhere else.

3.1.1 The advantages of using one hot encoding include

— It allows the use of categorical variables in models that require numerical input.
— It can improve model performance by providing more information to the model about the categorical variable.

3.1.2 The disadvantages of using one hot encoding include

— It can lead to increased dimensionality, as a separate column is created for each category in the variable. This can make the model more complex and slow to train.
— It can lead to sparse data, as most observations will have a value of 0 in most of the one-hot encoded columns.
— It can lead to overfitting, especially if there are many categories in the variable and the sample size is relatively small.
— One-hot-encoding is a powerful technique to treat categorical data, but it can lead to increased dimensionality, sparsity, and overfitting.

3.1.3 One Hot Encoding Examples

In One Hot Encoding, the categorical parameters will prepare separate columns for both Male and Female labels. So, wherever there is a Male, the value will be 1 in the Male column and 0 in the Female column, and vice-versa.

Fruit	Categorical value of fruit	Price
apple	1	5
mango	2	10
apple	1	5
orange	3	20

The output after applying one-hot encoding on the data is given as follows,

apple	mango	orange	Price
1	0	0	5
0	1	0	10
1	0	0	5
0	0	1	20

3.1.4 One-Hot Encoding Using Python

One Hot encoding

```
import pandas as pd


df = pd.DataFrame()

df['ID'] = [1,34,56,80,81 ,100]
df['Gender'] = ["Male","Female","Female","Male","Female" ,"Male"]
df['Remarks'] = ["Nice","Good","Bad","Good","Nice" ,"Bad"]

print(df['Gender'].unique())
print(df['Remarks'].unique())

# get demmies pandas lib

one_hot_encoded_data = pd.get_dummies(df, columns = ['Remarks', 'Gender'])
print(one_hot_encoded_data)

#Hot Encoding using Sci-kit Learn Library

from sklearn.preprocessing import OneHotEncoder


# Converting type of columns to category
df['Gender'] = df['Gender'].astype('category')
df['Remarks'] = df['Remarks'].astype('category')


# Assigning numerical values and storing it in another columns
df['Gen_new'] = df['Gender'].cat.codes
df['Rem_new'] = df['Remarks'].cat.codes


# Create an instance of One-hot-encoder
enc = OneHotEncoder()
```

```
# Passing encoded columns

enc_data = pd.DataFrame(enc.fit_transform(
    df[['Gen_new', 'Rem_new']]).toarray())

# Merge with main
New_df = df.join(enc_data)

print(New_df)
```

3.2 Bag of words

Bag of Words model is used to preprocess the text by converting it into a bag of words, which keeps a count of the total occurrences of most frequently used words.

This model can be visualized using a table, which contains the count of words corresponding to the word itself.

The approach is very simple and flexible, and can be used in a myriad of ways for extracting features from documents.

A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things :

1. A vocabulary of known words.
2. A measure of the presence of known words.

It is called a "bag" of words, because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document.

3.2.1 Example of the Bag-of-Words Model

Below is a snippet of the first few lines of text from the book :

It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,

Step : Design the Vocabulary

The unique words here (ignoring case and punctuation) are :
— "it"
— "was"
— "the"

— "best"
— "of"
— "times"
— "worst"
— "age"
— "wisdom"
— "foolishness"

That is a vocabulary of 10 words from a corpus containing 24 words.

Step : Create Document Vectors

The next step is to score the words in each document. The objective is to turn each document of free text into a vector that we can use as input or output for a machine learning model.

The simplest scoring method is to mark the presence of words as a boolean value, 0 for absent, 1 for present.

"It was the best of times" = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
"it was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
"it was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
"it was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]

Step : Count

To vectorize our documents, all we have to do is count how many times each word appears :

Document	the	cat	sat	in	hat	with
the cat sat	1	1	1	0	0	0
the cat sat in the hat	2	1	1	1	1	0
the cat with the hat	2	1	0	0	1	1

3.2.2 Python Bag of words

Bag of words / Python

```
# Python3 code for preprocessing text
import nltk
nltk.download('punkt')
import re
import numpy as np
from nltk.tokenize import sent_tokenize

# execute the text here as :
text = """ The Master AISD of the FSTT 2024 FSTT """
dataset = sent_tokenize(text)
for i in range(len(dataset)):
    dataset[i] = dataset[i].lower()
    dataset[i] = re.sub(r'\W', ' ', dataset[i])
    dataset[i] = re.sub(r'\s+', ' ', dataset[i])
```

```
# Creating the Bag of Words model
word2count = {}
for data in dataset:
    words = nltk.word_tokenize(data)
    for word in words:
        if word not in word2count.keys():
            word2count[word] = 1
        else:
            word2count[word] += 1

word2count
```

Bag of words / Sklearn

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

vectorizer = CountVectorizer()
doc1 = ['Game of Thrones is an amazing tv series!',
'Game of Thrones is the best tv series!',
 'Game of Thrones is so great']
X = vectorizer.fit_transform(doc1)
df_bow_sklearn = pd.DataFrame(X.toarray(),
              columns=vectorizer.get_feature_names_out())
df_bow_sklearn.head()
```

3.3 TF/IDF

In all the above techniques, Each word is treated equally. TF-IDF tries to quantify the importance of a given word relative to the other word in the corpus. it is mainly used in Information retrieval.

TF-IDF stands for Term Frequency Inverse Document Frequency of records. It can be defined as the calculation of how relevant a word in a series or corpus is to a text. The meaning increases proportionally to the number of times in the text a word appears but is compensated by the word frequency in the corpus (data-set).

TF-IDF stands for Term Frequency Inverse Document Frequency of records. It can be defined as the calculation of how relevant a word in a series or corpus is to a text. The meaning increases proportionally to the number of times in the text a word appears but is compensated by the word frequency in the corpus (data-set).

$$TF(t,d) = \frac{Number\_of\_occurences\_of\_term\_t\_in\_document\_d}{Total\_number\_of\_terms\_in\_the\_document\_d}$$

Inverse document frequency (IDF) : IDF measures the importance of the word across the corpus. it down the weight of the terms, which commonly occur in the corpus, and up the weight of rare

terms.

$$IDF(t) = log_e \frac{Number\_of\_occurences\_of\_documents\_in\_corpus}{Number\_of\_documents\_with\_term\_tin\_corpus}$$

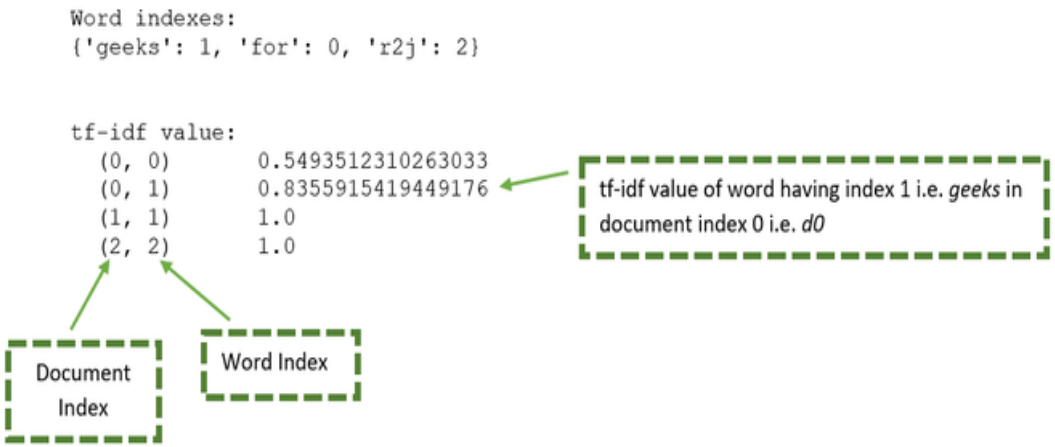
F-IDF score is the product of TF and IDF.

$$TF - IDF = TF * IDF$$

3.3.1 TF-IDF Python

TF-IDF / Sklearn

```
# import required module
from sklearn.feature_extraction.text import TfidfVectorizer
# assign documents
d0 = 'FSTT for geeks'
d1 = 'Master AISD'
d2 = 'Master Computer Sciences'

# merge documents into a single corpus
string = [d0, d1, d2]

# create object
tfidf = TfidfVectorizer()

# get tf-df values
result = tfidf.fit_transform(string)

# get idf values
print('\nidf values:')
for ele1, ele2 in zip(tfidf.get_feature_names_out(), tfidf.idf_):
    print(ele1, ':', ele2)

 # get indexing
print('\nWord indexes:')
print(tfidf.vocabulary_)

# display tf-idf values
print('\ntf-idf value:')
print(result)

# in matrix form
print('\ntf-idf values in matrix form:')
print(result.toarray())
```



3.4 Word Embedding

3.4.1 Word Embedding

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.

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.

The main goal of word embedding :

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

3.4.2 For example

airplane =[0.7, 0.9, 0.9, 0.01, 0.35]
kite =[0.7, 0.9, 0.2, 0.01, 0.2]

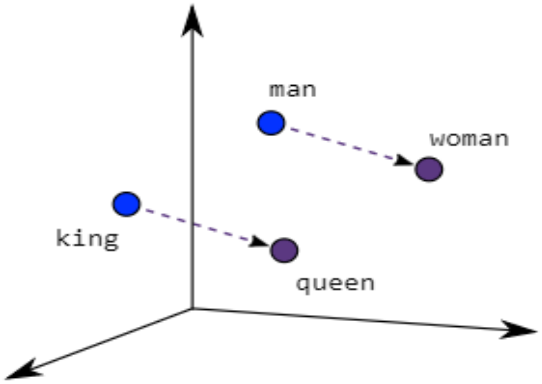
Here each value in the vector represents the measurements of some features or quality of the word which is decided by the model after training on text data.

	Airplane	Kit
Sky	0.7	0.7
Fly	0.9	0.9
Transport	0.9	0.2
Animal	0.1	0.01
Eat	0.3	0.25

3.4.3 Word2Vec

Word2Vec consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer.

Word2Vec creates vectors of the words that are distributed numerical representations of word features these word features could comprise of words that represent the context of the individual words present in our vocabulary. Word embeddings eventually help in establishing the association of a word with another similar meaning word through the created vectors.



Word2Vec utilizes two architectures :

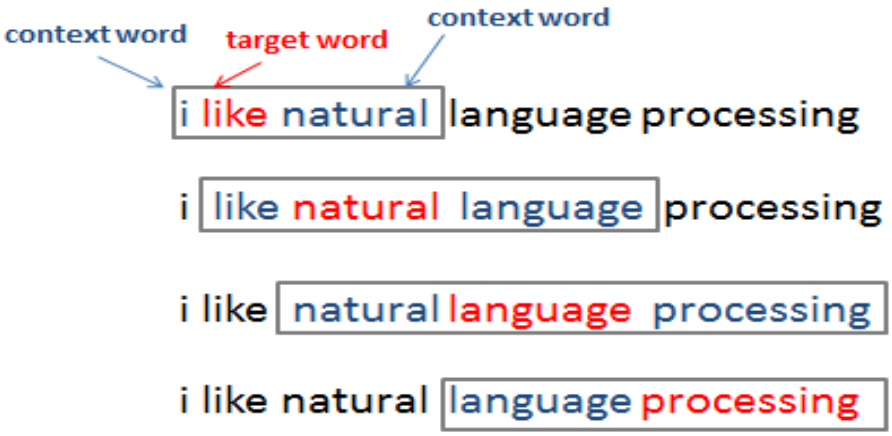
- CBOW (Continuous Bag of Words)
- Skip Gram

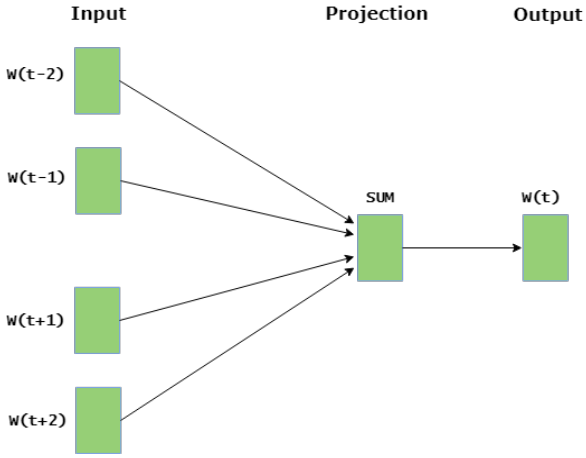
The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space.

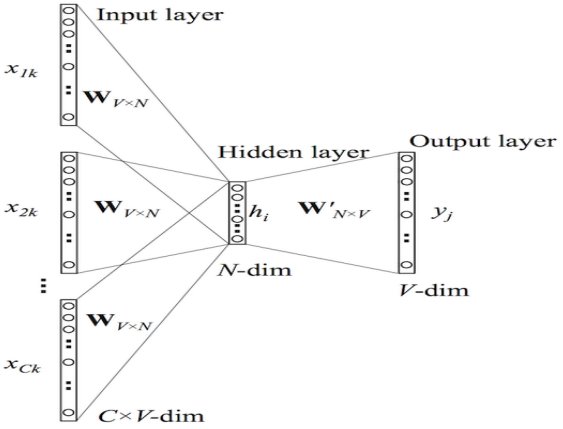
CBOW

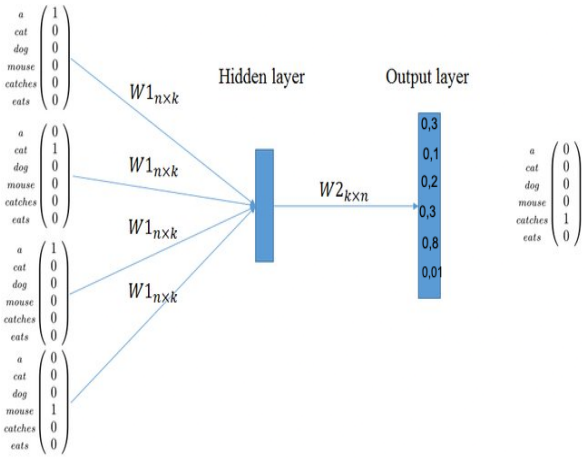
CBOW model predicts the current word given context words within a specific window. The input layer contains the context words and the output layer contains the current word. The hidden layer contains the number of dimensions in which we want to represent the current word present at the output layer.

Considering a simple sentence, "the quick brown fox jumps over the lazy dog", this can be pairs of (context_window, target_word) where if we consider a context window of size 2, we have examples like ([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy) and so on. Thus the model tries to predict the target_word based on the context_window words.









The CBOW architecture is pretty simple contains :

— the word embeddings as inputs (idx)
— the linear model as the hidden layer
— the softmax as the output $p(y_i|w_1, ..., w_c) = y_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
— loss function $E = -log(p(y_i|w_1, ..., w_c))$

CBOW Python

```
import regex as re
import numpy as np
import matplotlib.pyplot as plt
sentences = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells."""

# remove special characters
sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)

# remove 1 letter words
sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()

# lower all characters
sentences = sentences.lower()

words = sentences.split()
vocab = set(words)
vocab_size = len(vocab)
embed_dim = 10
context_size = 2

# dictionary
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for i, word in enumerate(vocab)}

# data - [(context), target]
# data bags
data = []
for i in range(2, len(words) - 2):
    context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]
    target = words[i]
    data.append((context, target))
print(data[:5])

# embedings
embeddings =  np.random.random_sample((vocab_size, embed_dim))
```

```
# linear model

def linear(m, theta):
    w = theta
    return m.dot(w)

# log softmax

def log_softmax(x):
    e_x = np.exp(x - np.max(x))
    return np.log(e_x / e_x.sum())

def NLLLoss(logs, targets):
    out = logs[range(len(targets)), targets]
    return -out.sum()/len(out)

def log_softmax_crossentropy_with_logits(logits,target):

    out = np.zeros_like(logits)
    out[np.arange(len(logits)),target] = 1

    softmax = np.exp(logits) / np.exp(logits).sum(axis=-1,keepdims=True)

    return (- out + softmax) / logits.shape[0]

def forward(context_idxs, theta):
    m = embeddings[context_idxs].reshape(1, -1)
    n = linear(m, theta)
    o = log_softmax(n)

    return m, n, o

def backward(preds, theta, target_idxs):
    m, n, o = preds

    dlog = log_softmax_crossentropy_with_logits(n, target_idxs)
    dw = m.T.dot(dlog)

    return dw

def optimize(theta, grad, lr=0.03):
    theta -= grad * lr
    return theta

# init thetas
```

```
theta = np.random.uniform(-1, 1, (2 * context_size * embed_dim, vocab_size))

epoch_losses = {}

for epoch in range(80):

    losses =  []

    for context, target in data:
        context_idxs = np.array([word_to_ix[w] for w in context])
        preds = forward(context_idxs, theta)

        target_idxs = np.array([word_to_ix[target]])
        loss = NLLLoss(preds[-1], target_idxs)

        losses.append(loss)

        grad = backward(preds, theta, target_idxs)
        theta = optimize(theta, grad, lr=0.03)


    epoch_losses[epoch] = losses

 ix = np.arange(0,80)

fig = plt.figure()
fig.suptitle('Epoch/Losses', fontsize=20)
plt.plot(ix,[epoch_losses[i][0] for i in ix])
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Losses', fontsize=12)

def predict(words):
    context_idxs = np.array([word_to_ix[w] for w in words])
    preds = forward(context_idxs, theta)
    word = ix_to_word[np.argmax(preds[-1])]

    return word , np.argmax(preds[-1])

# (['we', 'are', 'to', 'study'], 'about')
predict(['we', 'are', 'to', 'study'])
```

Skip Gram

Skip gram predicts the surrounding context words within specific window given current word. The input layer contains the current word and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word

present at the input layer.

Window Size	Text	Skip-grams
2	[The **wide** road shimmered] in the hot sun.	wide, the wide, road wide, shimmered
	The [wide road **shimmered** in the] hot sun.	shimmered, wide shimmered, road shimmered, in shimmered, the
	The wide road shimmered in [the hot **sun**].	sun, the sun, hot
3	[The **wide** road shimmered in] the hot sun.	wide, the wide, road wide, shimmered wide, in
	[The wide road **shimmered** in the hot] sun.	shimmered, the shimmered, wide shimmered, road shimmered, in shimmered, the shimmered, hot
	The wide road shimmered [in the hot **sun**].	sun, in sun, the sun, hot

Algorithm :
For each word t = 1 ... T, we predict the surrounding words in a window of "radius" m. We train a machine learning model to maximize the probability of any context word given the current centre word.

$$J' = \prod_{t=1}^{T} \prod_{-m \leq j \leq m} P(w_{t+j}|w_t)$$

Like we do for other probabilistic models, we try to minimize the negative log likelihood.

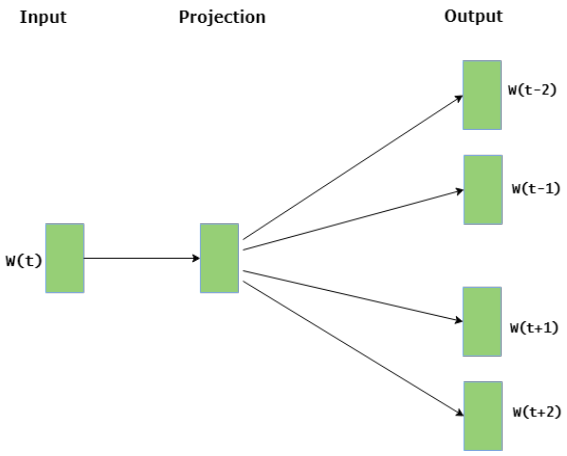
$$J = -\Sigma_{t=1}^{T} \Sigma_{-m \leq j \leq m} log(P(w_{t+j}|w_t))$$

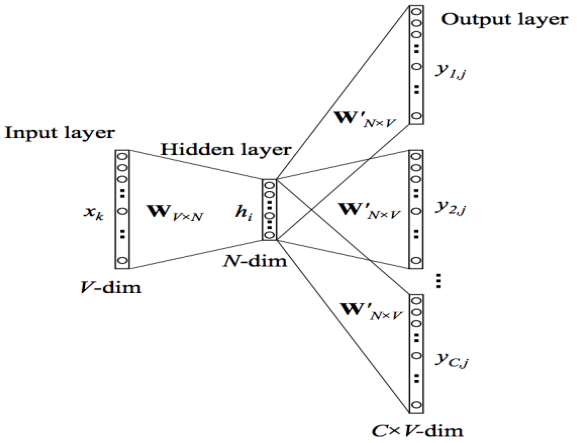
where$P(w_{t+j}|w_t)$ can be formulated as a Softmax function.

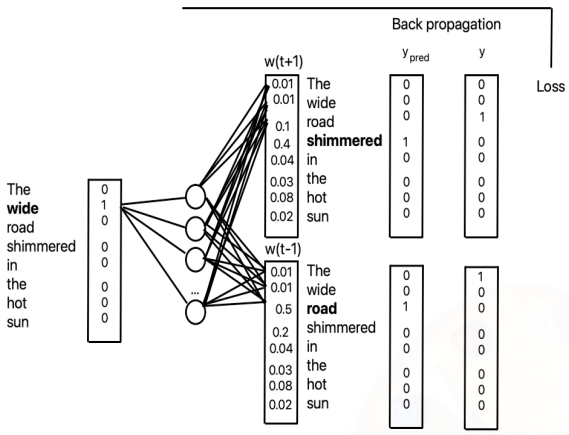
$$P(o|c) = \frac{exp(U_o^T V_c)}{\Sigma_{w=1}^{V} exp(U_{w=1}^T V_c)}$$

where

$$U^T.V = U.V = \Sigma_{i=1}^{n} u_i.v_i$$







Skip gram Python

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

def tokenize(text):
    pattern = re.compile(r'[A-Za-z]+[\w^\']*|[\w^\']*[A-Za-z]+[\w^\']*')
    return pattern.findall(text.lower())

tokens = tokenize("Hello from the FSTT Master AISD")
print(tokens)

id_to_word = {i:x for (i, x) in enumerate(tokens)}
word_to_id = {x:i for (i, x) in enumerate(tokens)}
print(word_to_id)
print(id_to_word)

def generate_training_data(tokens, word_to_id, window_size):
    X, Y = [], []

    for i in range(len(tokens)):
        nbr_inds = list(range(max(0, i - window_size), i)) + \
                   list(range(i + 1, min(len(tokens), i + window_size + 1)))
```

```
        for j in nbr_inds:
            X.append(word_to_id[tokens[i]])
            Y.append(word_to_id[tokens[j]])

    return np.array(X), np.array(Y)

 x, y = generate_training_data(tokens, word_to_id, 3)
print(x)
print(y)

def expand_dims(x, y):
    x = np.expand_dims(x, axis=0)
    y = np.expand_dims(y, axis=0)
    return x, y
x, y = expand_dims(x, y)
print(x)
print(y)

# forward propagation

def init_parameters(vocab_size, emb_size):
    wrd_emb = np.random.randn(vocab_size, emb_size) * 0.01
    w = np.random.randn(vocab_size, emb_size) * 0.01

    return wrd_emb, w

def softmax(z):
    return np.divide(np.exp(z), np.sum(np.exp(z), axis=0, keepdims=True) + 0.001)

def forward(inds, params):
    wrd_emb, w = params
    word_vec = wrd_emb[inds.flatten(), :].T
    z = np.dot(w, word_vec)
    out = softmax(z)

    cache = inds, word_vec, w, z

    return out, cache

# cost funtion
def cross_entropy(y, y_hat):
    m = y.shape[1]
    cost = -(1 / m) * np.sum(np.sum(y_hat * np.log(y + 0.001), axis=0, keepdims=
   True), axis=1)
    return cost
```

```
# dervation of softmax
def dsoftmax(y, out):
    dl_dz = out - y

    return dl_dz

def backward(y, out, cache):
    inds, word_vec, w, z = cache
    wrd_emb, w = params

    dl_dz = dsoftmax(y, out)
    # deviding by the word_vec length to find the average
    dl_dw = (1/word_vec.shape[1]) * np.dot(dl_dz, word_vec.T)
    dl_dword_vec = np.dot(w.T, dl_dz)

    return dl_dz, dl_dw, dl_dword_vec

def update(params, cache, grads, lr=0.03):
    inds, word_vec, w, z = cache
    wrd_emb, w = params
    dl_dz, dl_dw, dl_dword_vec = grads

    wrd_emb[inds.flatten(), :] -= dl_dword_vec.T * lr
    w -= dl_dw * lr

    return wrd_emb, w

# training
vocab_size = len(id_to_word)

m = y.shape[1]
y_one_hot = np.zeros((vocab_size, m))
y_one_hot[y.flatten(), np.arange(m)] = 1

y = y_one_hot

batch_size=256
embed_size = 50

params = init_parameters(vocab_size, 50)

costs = []

for epoch in range(5000):
    epoch_cost = 0
```

```
    batch_inds = list(range(0, x.shape[1], batch_size))
    np.random.shuffle(batch_inds)

    for i in batch_inds:
        x_batch = x[:, i:i+batch_size]
        y_batch = y[:, i:i+batch_size]

        pred, cache = forward(x_batch, params)
        grads = backward(y_batch, pred, cache)
        params = update(params, cache, grads, 0.03)
        cost = cross_entropy(pred, y_batch)

        epoch_cost += np.squeeze(cost)

    costs.append(epoch_cost)

    if(epoch % 250 == 0):
        print("Cost after epoch {}: {}".format(epoch, epoch_cost))
ix = np.arange(0,5000)

fig = plt.figure()
fig.suptitle('Epoch/Losses', fontsize=20)
plt.plot(ix,[costs[i] for i in ix])
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Losses', fontsize=12)

# test
x_test = np.arange(vocab_size)
x_test = np.expand_dims(x_test, axis=0)
softmax_test, _ = forward(x_test, params)
top_sorted_inds = np.argsort(softmax_test, axis=0)[-4:,:]

for input_ind in range(vocab_size):
    input_word = id_to_word[input_ind]
    output_words = [id_to_word[output_ind] for output_ind in top_sorted_inds[::-1,
   input_ind]]
    print("{}'s skip-grams: {}".format(input_word, output_words))
```

There are other libraries like Gensim when we can use a Pre-trained model or train a costume model on a specific corpus :

Word2vec Gensim Python

```
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib.pyplot as plt
```

```
corpus = ['The sky is blue and beautiful.',
          'Love this blue and beautiful sky!',
          'The quick brown fox jumps over the lazy dog.',
          "A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
          'I love green eggs, ham, sausages and bacon!',
          'The brown fox is quick and the blue dog is lazy!',
          'The sky is very blue and the sky is very beautiful today',
          'The dog is lazy but the brown fox is quick!'
]
labels = ['weather', 'weather', 'animals', 'food', 'food', 'animals', 'weather', '
    animals']

corpus = np.array(corpus)
corpus_df = pd.DataFrame({'Document': corpus,
                          'Category': labels})
corpus_df = corpus_df[['Document', 'Category']]
corpus_df

from gensim.models import word2vec
from nltk.tokenize import WordPunctTokenizer
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
import re


# tokenize sentences in corpus
wpt = WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in corpus_df['Document']]

def isValid (word):
  flag = False ;
  #special characters
  x = re.search("[^A-Za-z0-9]+", word)
  # 1 letter words
  y = re.search("(?:^| )\w(?:$| )", word)
  if(x == None and y == None):
    flag = True

  return flag

stop_words = set(stopwords.words('english'))
tokenized_corpus_wst = [[]]
```

```
for line in tokenized_corpus :
  line_tokens = []
  for w in line :
    if w.lower() not in stop_words and isValid(w.lower()) == True:
      line_tokens.append(w.lower())
  tokenized_corpus_wst.append(line_tokens)


tokenized_corpus_wst

# Set values for various parameters
feature_size = 100    # Word vector dimensionality
window_context = 30          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words
# sg = 0 cbow , sg =1 skip gram
w2v_model = word2vec.Word2Vec(tokenized_corpus_wst, vector_size=feature_size,
                          window=window_context, min_count=min_word_count,
                          sample=sample, sg = 1,  epochs=50)

# view similar words based on gensim's model
similar_words = {search_term: [item[0] for item in w2v_model.wv.most_similar([
    search_term], topn=5)]
                 for search_term in ['love', 'fox', 'sky', 'breakfast', 'ham', '
    jumps', 'dog','quick']}
similar_words

from sklearn.manifold import TSNE

words = sum([[k] + v for k, v in similar_words.items()], [])
wvs = w2v_model.wv[words]

tsne = TSNE(n_components=2, random_state=0, n_iter=10000, perplexity=2)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(wvs)
labels = words

plt.figure(figsize=(14, 8))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset points')
```

Word2vec Pre-Trained Python

```
import gensim.downloader as api

# load the pre-trained Word2Vec model
```

```
model = api.load('word2vec-google-news-300')

# define word pairs to compute similarity for
word_pairs = [('learn', 'learning'), ('india', 'indian'), ('fame', 'famous')]

# compute similarity for each pair of words
for pair in word_pairs:
    similarity = model.similarity(pair[0], pair[1])
    print(f"Similarity between '{pair[0]}' and '{pair[1]}' using Word2Vec: {
    similarity:.3f}")
```

3.4.4 FastText

FastText is an open-source, free library from Facebook AI Research(FAIR) for learning word embeddings and word classifications. This model allows creating unsupervised learning or supervised learning algorithm for obtaining vector representations for words. It also evaluates these models. FastText supports both CBOW and Skip-gram models. FastText is a modified version of word2vec.
Uses of FastText :

1. It is used for finding semantic similarities
2. It can also be used for text classification(ex : spam filtering/ sentiments analysis, etc.).
3. It can train large datasets in minutes.

It treats each word as composed of n-grams. In word2vec each word is represented as a bag of words but in FastText each word is represented as a bag of character n-gram.

Character n-gram is the contiguous sequence of n items from a given sample of a character or word. It may be bigram, trigram, etc. For example character trigram (n = 3) of the word "where" will be :

<wh, whe, her, ere, re>

Skip-Gram

Training Example	Context Words	Target Word
#1	(i, natural)	like
#2	(like, language)	natural
#3	(language, processing)	language
#4	(language)	processing

FastText

Training Example	Context Words	Target Word
#1	(<i>, <na, nat, atu, tur, ura, ral, al>)	like
#2	(<li, lik, ike, ke>, <la, lan, ang, ngu, gua, uag, age, ge>)	natural
#3	<la, lan, ang, ngu, gua, uag, age, ge>, <pr, pro, roc, oce, ces, ess, ssi, sin, ing, ng>	language
#4	<la, lan, ang, ngu, gua, uag, age, ge>	processing

— Word2Vec works on the word level, while fastText works on the character n-grams.
— Word2Vec cannot provide embeddings for out-of-vocabulary words, while fastText can provide embeddings for OOV words.
— FastText can provide better embeddings for morphologically rich languages compared to word2vec.
— FastText uses the hierarchical classifier to train the model ; hence it is faster than word2vec.

Example : https://medium.com/@93Kryptonian/word-embedding-using-fasttext-62beb0209db9

FastText Pre-Trained Python

```
import gensim.downloader as api

# load the pre-trained fastText model
fasttext_model = api.load("fasttext-wiki-news-subwords-300")

# define word pairs to compute similarity for
word_pairs = [('learn', 'learning'), ('india', 'indian'), ('fame', 'famous')]

# compute similarity for each pair of words
for pair in word_pairs:
    similarity = fasttext_model.similarity(pair[0], pair[1])
    print(f"Similarity between '{pair[0]}' and '{pair[1]}' using Word2Vec: {
   similarity:.3f}")
```

3.4.5 Glove

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

GloVe (Global Vectors for Word Representation) is an alternate method to create word embeddings. It is based on matrix factorization techniques on the word-context matrix. A large matrix of co-occurrence information is constructed and you count each “word” (the rows), and how frequently we see this word in some “context” (the columns) in a large corpus. Usually, we scan our corpus in the following manner : for each term, we look for context terms within some area defined by a window-size before the term and a window-size after the term. Also, we give less weight for more distant words.

https://nlp.stanford.edu/projects/glove/

Gensim Glove Python

```
# code for Glove word embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

x = {'text', 'the', 'leader', 'prime',
     'natural', 'language'}
```

```
# create the dict.
tokenizer = Tokenizer()
tokenizer.fit_on_texts(x)

# number of unique words in dict.
print("Number of unique words in dictionary=",
      len(tokenizer.word_index))
print("Dictionary is = ", tokenizer.word_index)

# download glove and unzip it in Notebook.
#!wget http://nlp.stanford.edu/data/glove.6B.zip
#!unzip glove*.zip

# vocab: 'the': 1, mapping of words with
# integers in seq. 1,2,3..
# embedding: 1->dense vector
def embedding_for_vocab(filepath, word_index,
                        embedding_dim):
    vocab_size = len(word_index) + 1

    # Adding again 1 because of reserved 0 index
    embedding_matrix_vocab = np.zeros((vocab_size,embedding_dim))

    with open(filepath, encoding="utf8") as f:
        for line in f:
            word, *vector = line.split()
            if word in word_index:
                idx = word_index[word]
                embedding_matrix_vocab[idx] = np.array(vector, dtype=np.float32)[:
    embedding_dim]

    return embedding_matrix_vocab


# matrix for vocab: word_index
embedding_dim = 50
embedding_matrix_vocab = embedding_for_vocab( '../glove.6B.50d.txt', tokenizer.
    word_index,
  embedding_dim)

print("Dense vector for first word is => ",embedding_matrix_vocab[1])
```

Glove Pre-Trained Python

```
import torch
import torchtext.vocab as vocab
```

```
# load the pre-trained GloVe model
glove = vocab.GloVe(name='840B', dim=300)

# define word pairs to compute similarity for
word_pairs = [('learn', 'learning'), ('india', 'indian'), ('fame', 'famous')]

# compute similarity for each pair of words
for pair in word_pairs:
    vec1, vec2 = glove[pair[0]], glove[pair[1]]
    similarity = torch.dot(vec1, vec2) / (torch.norm(vec1) * torch.norm(vec2))
    print(f"Similarity between '{pair[0]}' and '{pair[1]}' using GloVe: {similarity
    :.3f}")
```

Chapitre 4

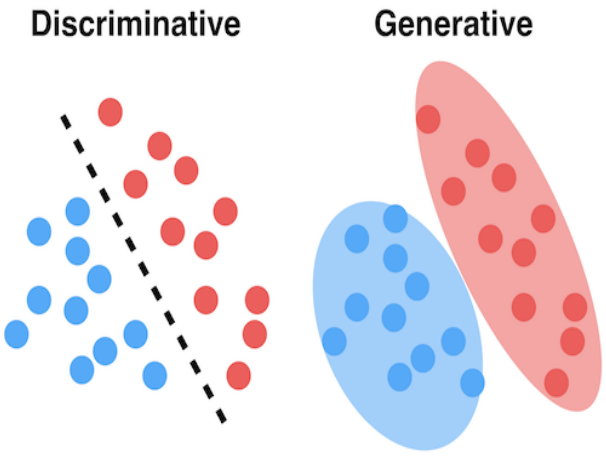
Language Modeling : Machine Learning

Language Modeling is the task of predicting the next word or character in a document. This technique can be used to train language models that can further be applied to a wide range of natural language tasks like text generation, text classification, and question answering.

4.1 Naive Bayes

4.1.1 Generative Vs Discriminative Machine learning

Machine learning models can be classified intotwo types : Discriminative and Generative. In simple words, a discriminative model makes predictions on unseen data based on conditional probability and can be used either for classification or regression problem statements. On the contrary, a generative model focuses on the distribution of a dataset to return a probability for a given example.



n the case of generative models, to find the conditional probability P(Y|X), they estimate the priorprobability P(Y) and likelihood probability P(X|Y) with the help of the training data and use the Bayes Theorem to calculate the posterior probability P(Y |X) :

$$P(Y|X) = \frac{prior * likelihood}{evidence} \rightarrow P(Y|X) = \frac{P(Y) * P(X|Y)}{P(X)}$$

The two definitions for generative models are closely related given the definition of joint probability.

$$P(X, Y) = P(X|Y)P(Y)$$

In the case of discriminative models, to find the probability (Conditional probability), they directly assume some functional form for P(Y|X) andthen estimate the parameters of P(Y|X = x) given an observation x (with the help of the training data).

$$P(X, Y) = \frac{P(X \cap Y)}{P(A)}$$

Some popular discriminative algorithms are :

— k-nearest neighbors (k-NN)
— Logistic regression
— Support Vector Machines (SVMs)
— Decision Trees
— Random Forest
— Artificial Neural Networks (ANNs)

Some popular generative algorithms are :

— Naive Bayes Classifier
— Generative Adversarial Networks
— Gaussian Mixture Model
— Hidden Markov Model
— Probabilistic context-free grammar

4.1.2 Naive Bayes Algorithm

— Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
— It is mainly used in text classification that includes a high-dimensional training dataset.
— Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
— It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
— Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

Naïve : It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features.

Bayes Theorem

Bayes' theorem is also known as Bayes' Rule or Bayes' law, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.

$$P(A|B) = \frac{P(A) * P(B|A)}{P(B)}$$

Where,

— P(A|B) is Posterior probability : Probability of hypothesis A on the observed event B.
— P(B|A) is Likelihood probability : Probability of the evidence given that the probability of a hypothesis is true.
— P(A) is Prior Probability : Probability of hypothesis before observing the evidence.
— P(B) is Marginal Probability : Probability of Evidence.

	Outlook	Play
1	Rainy	yes
2	Sunny	yes
3	Overcast	yes
4	Overcast	yes
5	Sunny	no
6	Rainy	yes
7	Sunny	yes
8	Overcast	yes
9	Rainy	no
10	Sunny	yes
11	Sunny	no
12	Rainy	no
13	Overcast	yes
14	Overcast	yes

Frequency table for the Weather Conditions :

Weather	Yes	No
Overcast	5	0
Sunny	3	2
Rainy	2	2

Likelihood table weather condition :

Weather	Yes	No	
Overcast	5	0	5/14 = 0.35
Sunny	3	2	5/14 = 0.35
Rainy	2	2	4/14 = 0.29

Applying Bayes'theorem :

P(Yes|Sunny)= P(Sunny|Yes)*P(Yes)/P(Sunny)

P(Sunny|Yes)= 3/10= 0.3

P(Sunny)= 0.35

P(Yes)=0.71

So P(Yes|Sunny) = 0.3*0.71/0.35= 0.60

P(No|Sunny)= P(Sunny|No)*P(No)/P(Sunny)

P(Sunny|NO)= 2/4=0.5

P(No)= 0.29

P(Sunny)= 0.35

So P(No|Sunny)= 0.5*0.29/0.35 = 0.41

So as we can see from the above calculation that P(Yes|Sunny)>P(No|Sunny)

Hence on a Sunny day, Player can play the game.

Naive Bayes Sklearn

```
# example of gaussian naive bayes
from sklearn.datasets import make_blobs
from sklearn.naive_bayes import GaussianNB
# generate 2d classification dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# define the model
model = GaussianNB()
# fit the model
model.fit(X, y)
# select a single sample
Xsample, ysample = [X[0]], y[0]
# make a probabilistic prediction
yhat_prob = model.predict_proba(Xsample) print('Predicted Probabilities: ',
    yhat_prob) # make a classification prediction
yhat_class = model.predict(Xsample) print('Predicted Class: ', yhat_class) print('
    Truth: y=%d' % ysample)
```

4.1.3 Naive Bayes and NLP

Our classifier gives “overall liked the movie” the positive tag. Below is the implementation :

Naive Bayes Sklearn

```
# cleaning texts
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer

dataset = [["I liked the movie", "positive"],
                ["It's a good movie. Nice story", "positive"],
                ["Hero's acting is bad but heroine looks good.\
                        Overall nice movie", "positive"],
                        ["Nice songs. But sadly boring ending.", "negative"],
                        ["sad movie, boring movie", "negative"]]

dataset = pd.DataFrame(dataset)
```

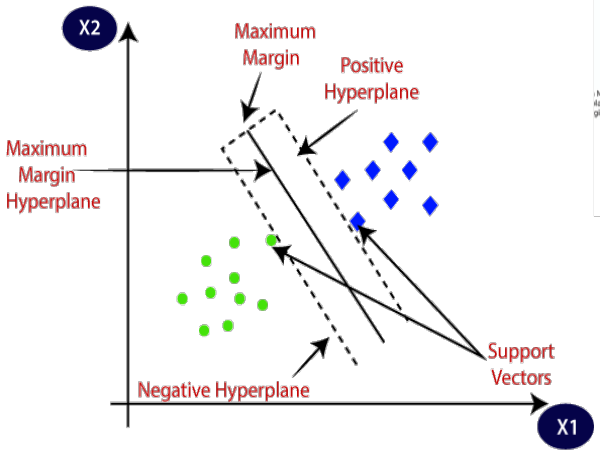
```
dataset.columns = ["Text", "Reviews"]

nltk.download('stopwords')

corpus = []

for i in range(0, 5):
        text = re.sub('[^a-zA-Z]', '', dataset['Text'][i])
        text = text.lower()
        text = text.split()
        ps = PorterStemmer()
        text = ''.join(text)
        corpus.append(text)

# creating bag of words model
cv = CountVectorizer(max_features = 1500)

X = cv.fit_transform(corpus).toarray()
y = dataset.iloc[:, 1].values

# splitting the data set into training set and test set
from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
                X, y, test_size = 0.25, random_state = 0)
# fitting naive bayes to the training set
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix

classifier = GaussianNB();
classifier.fit(X_train, y_train)

# predicting test set results
y_pred = classifier.predict(X_test)

# making the confusion matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

4.2 Support Vector Machine

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

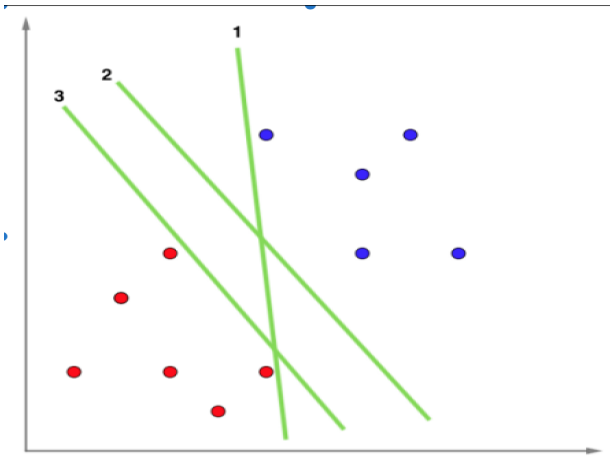
SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane :



4.2.1 Types of SVM

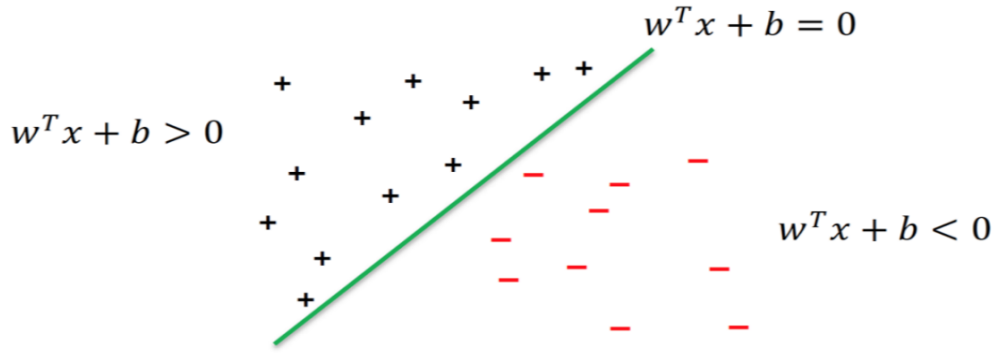
— Linear SVM : Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
— Non-linear SVM : Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

4.2.2 Hyperplane

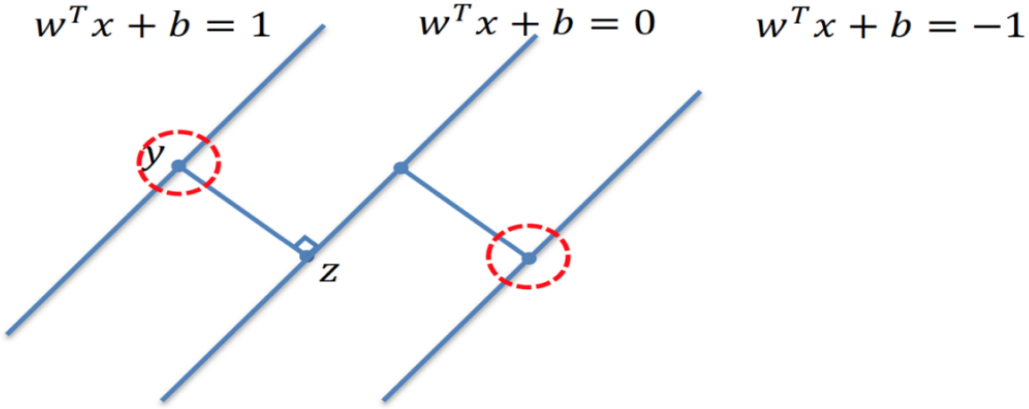


Consider, we want to separate the blue points from the red ones. There are many ways to draw a line between those points. But, let's say we have three different ways as shown above. Our intuition can tell us that line 2 is the best separating line as it's not closer to points like in line 1 but also not very close to just one type of data like in 3. We chose line 2 because it has maximum margin i.e. it is at maximum distance from the both red and blue points.

1. We need a line that should separate the blue and red points
2. The line should have a maximum margin.
3. It should be easy to calculate.



4.2.3 How to find the best Hyperplane



$w^T X + b = 1$ the straight line passing over the support vector of the positive class.
$w^T X + b = -1$ the straight line passing over the support vector of the negative class.

The perpendicular distance d of a data point X from the margin is given by :

$$d = \frac{|w^T X + B_0|}{||X||}$$

$x_1 = w^T X_1 + b = 1$
$x_2 = w^T X_2 + b = -1$
$x_1 - x_2 \rightarrow w^T (X_1 - X_2) = 2$

We will eliminate w^T :

$\frac{w^T (X_1 - X_2)}{||w||} = \frac{2}{||w||} \rightarrow Max(\frac{2}{||w||})$. Or $Min(||w||)$, other writing maximize $2/||w||^2$ or minimize $||w||^2$, (Hard Margine).

$||w|| = \sqrt{(w_1^2 + w_2^2 + ... + w_n^2)}$.

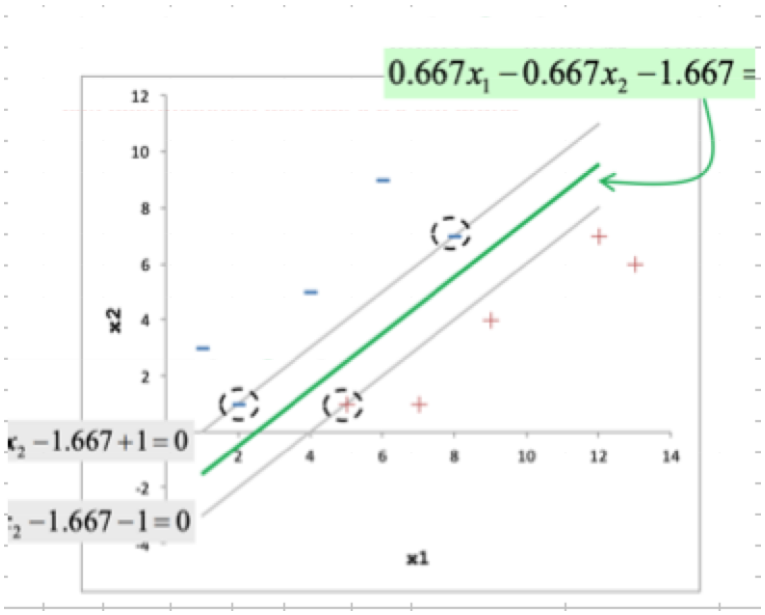
Ou bien un forme généralisée : $y_i * w^T x + b >= 1$

Dans le cas ou on veut éviter le phénomène de sur apprentissage (SOFT Margine) :

$$\min \frac{1}{2}||w||^2 + \Sigma_{k=1}^{p} \epsilon_k, C > 0$$

4.2.4 Linear SVM : Primal

X1	X2	Y
1	3	-1
2	1	-1
4	5	-1
6	9	-1
8	7	-1
5	1	1
7	1	1
9	4	1
12	7	1
13	6	1



L'objectif est de calculer les coef Beta :
Selon le graphique on a trois points qui représentent les support vecteures :

$$S_1 = (2, 1)$$

$$S_2 = (8, 7)$$

$$S_3 = (5, 1)$$

on augmenter les vecteures en ajoutant 1 : Bais/Inception.

$$\tilde{S}_1 = (2, 1, 1)$$

$$\tilde{S}_2 = (8, 7, 1)$$

$$\tilde{S}_3 = (5, 1, 1)$$

Donc on peut construire un systeme d'équation lineaire :

$$\Sigma_i^n \alpha_i < x_i.x >$$

$$\alpha_1 \tilde{S_1}\tilde{S_1} + \alpha_2 \tilde{S_2}\tilde{S_1} + \alpha_3 \tilde{S_3}\tilde{S_1} = -1$$
$$\alpha_1 \tilde{S_1}\tilde{S_2} + \alpha_2 \tilde{S_2}\tilde{S_2} + \alpha_3 \tilde{S_3}\tilde{S_2} = -1$$
$$\alpha_1 \tilde{S_1}\tilde{S_3} + \alpha_2 \tilde{S_2}\tilde{S_3} + \alpha_3 \tilde{S_3}\tilde{S_3} = +1$$

$$\alpha_1(2,1,1)(2,1,1) + \alpha_2(8,7,1)(2,1,1) + \alpha_3(5,1,1)(2,1,1) = -1$$
$$\alpha_1(2,1,1)(8,7,1) + \alpha_2(8,7,1)(8,7,1) + \alpha_3(5,1,1)(8,7,1) = -1$$
$$\alpha_1(2,1,1)(5,1,1) + \alpha_2(8,7,1)(5,1,1) + \alpha_3(5,1,1)(5,1,1) = +1$$

$$\alpha_1 6 + \alpha_2 24 + \alpha_3 12 = -1$$
$$\alpha_1 24 + \alpha_2 114 + \alpha_3 48 = -1$$
$$\alpha_1 12 + \alpha_2 48 + \alpha_3 27 = +1$$

$$\alpha_1 = \frac{17}{6}$$
$$\alpha_2 = \frac{1}{6}$$
$$\alpha_3 = 1$$

$$w = \Sigma_i^n \alpha_i \tilde{S_i}$$

$$w = \frac{17}{6}(2,1,1) + \frac{1}{6}(8,7,1) + (5,1,1)$$

$$w = (\frac{17}{3}, \frac{17}{6}, \frac{17}{6}) + (\frac{4}{3}, \frac{7}{6}, \frac{1}{6}) + (5,1,1)$$

$$w = \begin{pmatrix} -5,66 + 1,33 + 5 \\ -2,83 + 1,16 + 1 \\ -2,83 + 0,166 + 1 \end{pmatrix} \tag{4.1}$$

$$w = \begin{pmatrix} 0,663 \\ -0,663 \\ -1,663 \end{pmatrix} \tag{4.2}$$

4.2.5 Linear SVM : Dual

Support vectors can be defined via Lagrange multipliers, using an optimization method called "Sequential Minimal Optimization".

Lagrange multipliers

To solve the above quadratic programming problem with inequality constraints, we can use the method of Lagrange multipliers. The Lagrange function is therefore :

$$L(w, w_0, \alpha) = \frac{1}{2}||w||^2 + \sum_i \alpha_i \Big(y_i(w^T x_i + w_0) - 1 \Big)$$

To solve the above, we set the following :

$$\frac{\partial L}{\partial w} = 0, \frac{\partial L}{\partial \alpha} = 0, \frac{\partial L}{\partial w_0} = 0$$

Plugging above in the Lagrange function gives us the following optimization problem, (Quadratic Problem) also called the dual :

$$L_d = -\frac{1}{2} \sum_i \sum_k \alpha_i \alpha_k y_i y_k (x_i)^T (x_k) + \sum_i \alpha_i$$

We have to maximize the above subject to the following :

$$w = \sum_i \alpha_i y_i x_i$$

and

$$0 = \sum_i \alpha_i y_i$$

The nice thing about the above is that we have an expression for w in terms of Lagrange multipliers. The objective function involves no term. There is a Lagrange multiplier associated with each data point. The computation of w_0 is also explained later.
The classification of any test point x can be determined using this expression :

$$y(x) = \sum_i \alpha_i y_i x^T x_i + w_0$$

A positive value of y(x)implies x belongs to 1 and a negative value means x belongs to -1.

Karush-Kuhn-Tucker Conditions

Also, Karush-Kuhn-Tucker (KKT) conditions are satisfied by the above constrained optimization problem as given by :

$$\alpha_i \geq 0$$

$$y_i y(x_i) - 1 \geq 0$$

$$\alpha_i (y_i y(x_i) - 1) = 0$$

The KKT conditions dictate that for each data point one of the following is true :

The Lagrange multiplier is zero, i.e., $\alpha_i = 0$. This point, therefore, plays no role in classification OR

$t_i y(x_i) = 1$ and $\alpha_i > 0$: In this case, the data point has a role in deciding the value of w. Such a point is called a support vector.

Computing w_0 :
For w_0, we can select any support vector and solve

$$t_s y(x_s) = 1$$

giving us :

$$t_s(\sum_i \alpha_i y_i x_s^T x_i + w_0) = 1$$

Sequential Minimal Optimization

Sequential minimal optimization (SMO) is an algorithm for solving the quadratic programming (QP) problem that arises during the training of support-vector machines (SVM). SMO is an iterative algorithm for solving the optimization problem described above.

The algorithm proceeds as follows :

The simplified SMO algorithm takes the parameters α, α_i and α_j, and optimizes them. To do this, we iterate over any α_i, i = 1, . . . M. If α_i does not satisfy the Karush-Kuhn-Tucker conditions within a certain numerical tolerance, we select α_j at random from the remaining m - 1 α and optimize α_i and α_j. The following function will help us select j at random :

If none of the α's are modified after a few iterations on all α_i's, then the algorithm terminates. We also need to find the bounds L and H :

— if y(i) != y(j), L = max(0, α_j- α_i), H = min(C, C + α_j- α_i)
— if y(i) = y(j), L = max(0, α_i + α_j - C), H = min(C, α_i + α_j)
Where C is a regularization parameter.

Now we will find that α_j is given by :

$$\alpha_j := \alpha_j - \frac{y^{(i)}(E_i - E_j)}{\eta}$$

where,

$$E_k = f(x^{(k)}) - y^{(k)}$$

$$\eta = 2 < x^{(i)}, x^{(j)} > - <^{(i)}, x^{(i)} > - <^{(j)}, x^{(i)} >$$

If this value ends up outside the L and H bounds, we need to clip the value of α_j to bring it within this range :

$$\alpha_j = \begin{cases} H & if \quad \alpha_j > H \\ \alpha_j & if \quad H \leq \alpha_j \leq L \\ L & if \quad \alpha_j < L \end{cases}$$

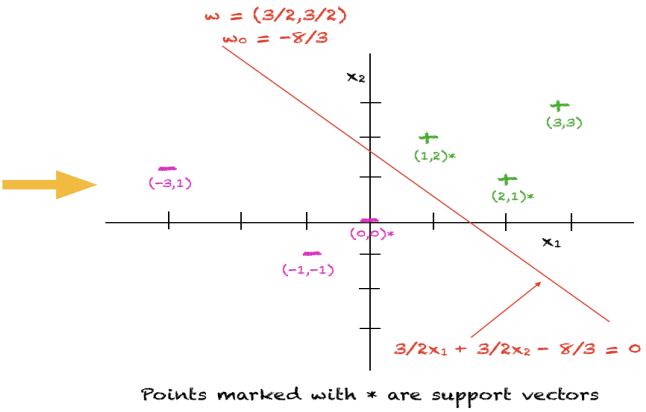
Finally, we can find the value of α_i. This is given by :

$$\alpha_i := \alpha + y^{(i)}y^{(j)}(\alpha_j - \alpha_j^{(old)})$$

where $\alpha_j^{(old)}$ is the value of α_j before optimization.

This example will show you that the model is not as complex as it looks.

i	Data point x	label t	α
0	(1,2)	+1	0.5
1	(2,1)	+1	0.5
2	(3,3)	+1	0
3	(0,0)	-1	1
4	(-1,-1)	-1	0
5	(-3,1)	-1	0



Lagrange multi with SVM

```
import numpy as np
# For optimization
from scipy.optimize import Bounds, BFGS
from scipy.optimize import LinearConstraint, minimize
# For plotting
import matplotlib.pyplot as plt
import seaborn as sns
# For generating dataset
import sklearn.datasets as dt

ZERO = 1e-7

def plot_x(x, t, alpha=[], C=0):
    sns.scatterplot(x, markers=['s', 'P'],
    palette=['magenta', 'green'])
    if len(alpha) > 0:
        alpha_str = np.char.mod('%.1f', np.round(alpha, 1))
        ind_sv = np.where(alpha > ZERO)[0]
        for i in ind_sv:
            plt.gca().text(x[i,0], x[i, 1]-.25, alpha_str[i] )

# Objective function
def lagrange_dual(alpha, x, t):
    result = 0
    ind_sv = np.where(alpha > ZERO)[0]
```

```
    for i in ind_sv:
        for k in ind_sv:
            result = result + alpha[i]*alpha[k]*t[i]*t[k]*np.dot(x[i, :], x[k, :])
    result = 0.5*result - sum(alpha)
    return result

def optimize_alpha(x, t, C):
    m, n = x.shape
    np.random.seed(1)
    # Initialize alphas to random values
    alpha_0 = np.random.rand(m)*C
    # Define the constraint
    linear_constraint = LinearConstraint(t, [0], [0])
    # Define the bounds
    bounds_alpha = Bounds(np.zeros(m), np.full(m, C))
    # Find the optimal value of alpha
    result = minimize(lagrange_dual, alpha_0, args = (x, t), method='trust-constr',
                      hess=BFGS(), constraints=[linear_constraint],
                      bounds=bounds_alpha)
    # The optimized value of alpha lies in result.x
    alpha = result.x
    return alpha

def get_w(alpha, t, x):
    m = len(x)
    # Get all support vectors
    w = np.zeros(x.shape[1])
    for i in range(m):
        w = w + alpha[i]*t[i]*x[i, :]
    return w

def get_w0(alpha, t, x, w, C):
    C_numeric = C-ZERO
    # Indices of support vectors with alpha<C
    ind_sv = np.where((alpha > ZERO)&(alpha < C_numeric))[0]
    w0 = 0.0
    for s in ind_sv:
        w0 = w0 + t[s] - np.dot(x[s, :], w)
    # Take the average
    w0 = w0 / len(ind_sv)
    return w0

def classify_points(x_test, w, w0):
    # get y(x_test)
    predicted_labels = np.sum(x_test*w, axis=1) + w0
    predicted_labels = np.sign(predicted_labels)
```

```
    # Assign a label arbitrarily a +1 if it is zero
    predicted_labels[predicted_labels==0] = 1
    return predicted_labels

def misclassification_rate(labels, predictions):
    total = len(labels)
    errors = sum(labels != predictions)
    return errors/total*100

def plot_hyperplane(w, w0):
    x_coord = np.array(plt.gca().get_xlim())
    y_coord = -w0/w[1] - w[0]/w[1] * x_coord
    plt.plot(x_coord, y_coord, color='red')

def plot_margin(w, w0):
    x_coord = np.array(plt.gca().get_xlim())
    ypos_coord = 1/w[1] - w0/w[1] - w[0]/w[1] * x_coord
    plt.plot(x_coord, ypos_coord, '--', color='green')
    yneg_coord = -1/w[1] - w0/w[1] - w[0]/w[1] * x_coord
    plt.plot(x_coord, yneg_coord, '--', color='magenta')

def display_SVM_result(x, t, C):
    # Get the alphas
    alpha = optimize_alpha(x, t, C)
    # Get the weights
    w = get_w(alpha, t, x)
    w0 = get_w0(alpha, t, x, w, C)
    plot_x(x, t, alpha, C)
    xlim = plt.gca().get_xlim()
    ylim = plt.gca().get_ylim()
    plot_hyperplane(w, w0)
    plot_margin(w, w0)
    plt.xlim(xlim)
    plt.ylim(ylim)
    # Get the misclassification error and display it as title
    predictions = classify_points(x, w, w0)
    err = misclassification_rate(t, predictions)
    title = 'C = ' + str(C) + ',  Errors: ' + '{:.1f}'.format(err) + '%'
    title = title + ',  total SV = ' + str(len(alpha[alpha > ZERO]))
    plt.title(title)

dat = np.array([[0, 3], [-1, 0], [1, 2], [2, 1], [3,3], [0, 0], [-1, -1], [-3, 1],
    [3, 1]])
labels = np.array([1, 1, 1, 1, 1, -1, -1, -1, -1])
plot_x(dat, labels)
plt.show()
```

```
display_SVM_result(dat, labels, 100)
plt.show()

dat, labels = dt.make_blobs(n_samples=[20,20],
                            cluster_std=1,
                            random_state=0)
labels[labels==0] = -1
plot_x(dat, labels)

fig = plt.figure(figsize=(8,25))

i=0
C_array = [1e-2, 100, 1e5]

for C in C_array:
    fig.add_subplot(311+i)
    display_SVM_result(dat, labels, C)
    i = i + 1
```

Linear SVM Sklearn

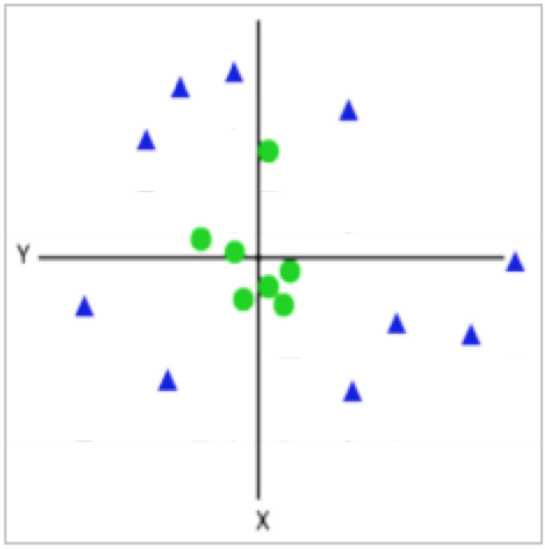
```
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd

#importing datasets
data_set= pd.read_csv('user_data.csv')

#Extracting Independent and dependent Variable
x= data_set.iloc[:, [2,3]].values
y= data_set.iloc[:, 4].values

# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25,
    random_state=0)
#feature Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
x_test= st_x.transform(x_test)



from sklearn.svm import SVC # "Support vector classifier"
classifier = SVC(kernel='linear', random_state=0)
classifier.fit(x_train, y_train)
```

```
alphas = np.abs(classifier.dual_coef_)

#Predicting the test set result
y_pred= classifier.predict(x_test)


#Creating the Confusion matrix
from sklearn.metrics import confusion_matrix
cm= confusion_matrix(y_test, y_pred)


from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].
    max() + 1, step  =0.01),
nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01)
    )
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).
    reshape(x1.shape),
alpha = 0.75, cmap = ListedColormap(('red', 'green')))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
        c = ListedColormap(('red', 'green'))(i), label = j)
mtp.title('SVM classifier (Training set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()


#Visulaizing the test set result
from matplotlib.colors import ListedColormap
x_set, y_set = x_test, y_test
x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].
    max() + 1, step  =0.01),
nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01)
    )
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).
    reshape(x1.shape),
alpha = 0.75, cmap = ListedColormap(('red','green' )))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],  c = ListedColormap(('
```

```
    red', 'green'))(i), label = j)
mtp.title('SVM classifier (Test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

4.2.6 Non Linear SVM : Kernel Methods

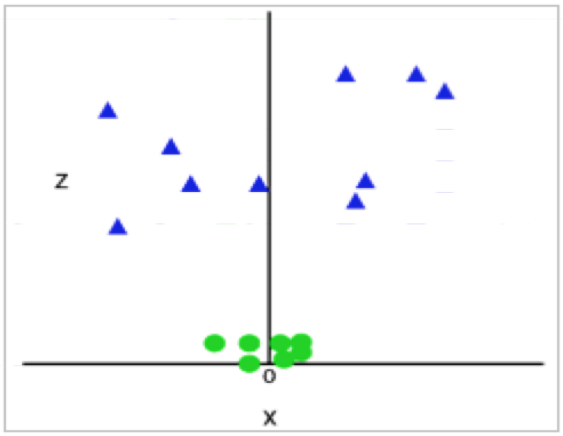
If the data are arranged linearly, we can separate them using a straight line, but for non-linear data, we can't draw a single straight line. Consider the image below :



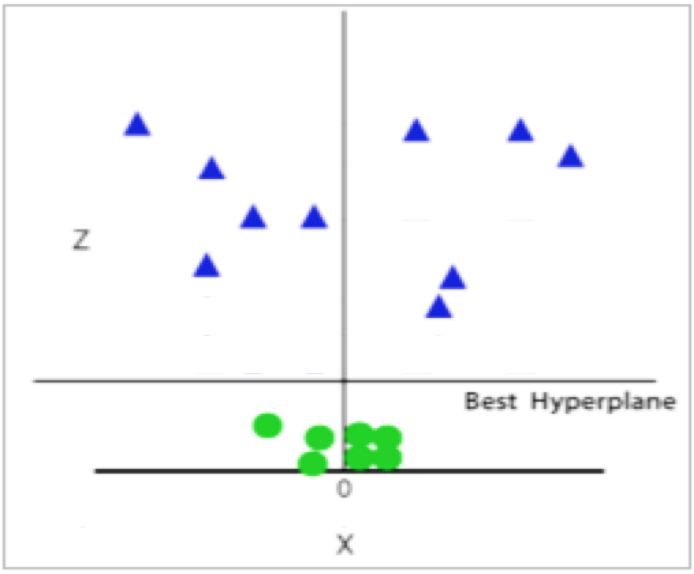
Hence, to separate these data points, we need to add an extra dimension. For linear data, we've used two dimensions x and y, so for non-linear data, we'll add a third dimension z. It can be calculated as follows :

$$z = x^2 + y^2$$

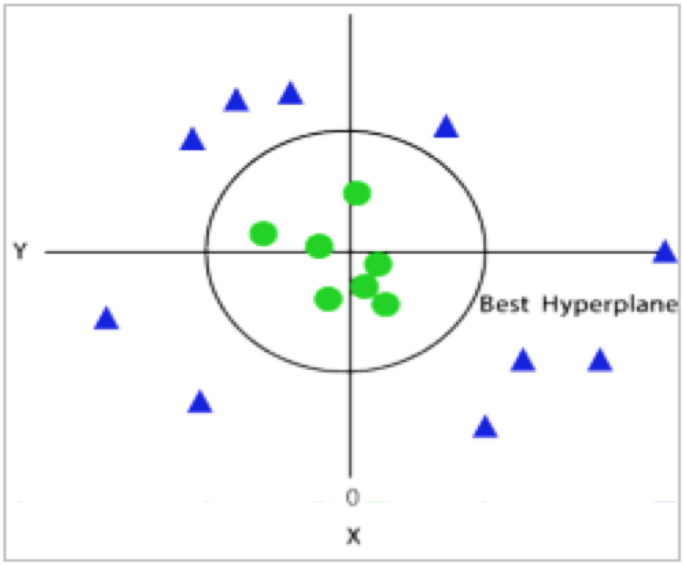
By adding the third dimension, sample space becomes like the image below :



Then now SVM will divide the data sets into classes in the following way. Consider the image below :



Since we're in 3D space, it looks like a plane parallel to the x axis. If we convert it to 2d space with z=1, it becomes :



The simplest example of a kernel function is the linear kernel :

$$K(x_i, x_j) = x_i^T x_j$$

The polynomial kernel :

$$K(x_i, x_j) = (x_i^T x_j + 1)^d$$

The Gaussian kernel :

$$K(x_i, x_j) = exp(\frac{||x - y||^2}{2\sigma^2})$$

The sigmoid kernel :

$$K(x_i, x_j) = tanh(\alpha x^T y + c)$$

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

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data" #
    Assign colum names to the dataset
colnames = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
    # Read dataset to pandas dataframe
irisdata = pd.read_csv(url, names=colnames)

X = irisdata.drop('Class', axis=1)
y = irisdata['Class']

from sklearn.model_selection import train_test_split X_train, X_test, y_train,
    y_test = train_test_split(X, y, test_size = 0.20)


#Polynomial Kernel


from sklearn.svm import SVC
svclassifier = SVC(kernel='poly', degree=8)
svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)


from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

#Guassian Kernel

from sklearn.svm import SVC
svclassifier = SVC(kernel='rbf')
svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))


#Sigmoid Kernel
from sklearn.svm import SVC
svclassifier = SVC(kernel='sigmoid')
```

```
svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

If we compare the performance of the different kernel types, we can clearly see that the sigmoid kernel performs least well. This is because the sigmoid function returns two values, 0 and 1, and is therefore better suited to binary classification problems. However, in our case, we had three output classes.

Among the Gaussian kernel and the polynomial kernel, we can see that the Gaussian kernel achieved a perfect prediction rate of 100%, while the polynomial kernel misclassified one instance. As a result, the Gaussian kernel performed slightly better. However, there is no hard and fast rule as to which kernel works best in each scenario. It's a matter of testing all the kernels and selecting the one with the best results on your test dataset.

4.2.7 SVM and NLP

The nlp task is sentiment analysis via SVM :

SVM Sentiment analysis

```
import pandas as pd
# train Data
trainData = pd.read_csv("https://raw.githubusercontent.com/Vasistareddy/
    sentiment_analysis/master/data/train.csv")
# test Data
testData = pd.read_csv("https://raw.githubusercontent.com/Vasistareddy/
    sentiment_analysis/master/data/test.csv")

from sklearn.feature_extraction.text import TfidfVectorizer
# Create feature vectors
vectorizer = TfidfVectorizer(min_df = 5,
                             max_df = 0.8,
                             sublinear_tf = True,
                             use_idf = True)
train_vectors = vectorizer.fit_transform(trainData['Content'])
test_vectors = vectorizer.transform(testData['Content'])

import time
from sklearn import svm
from sklearn.metrics import classification_report
# Perform classification with SVM, kernel=linear
classifier_linear = svm.SVC(kernel='linear')
t0 = time.time()
classifier_linear.fit(train_vectors, trainData['Label'])
t1 = time.time()
prediction_linear = classifier_linear.predict(test_vectors)
```

```
t2 = time.time()
time_linear_train = t1-t0
time_linear_predict = t2-t1
# results
print("Training time: %fs; Prediction time: %fs" % (time_linear_train,
    time_linear_predict))
report = classification_report(testData['Label'], prediction_linear, output_dict=
    True)
print('positive: ', report['pos'])
print('negative: ', report['neg'])



# exemples

review = """SUPERB, I AM IN LOVE IN THIS PHONE"""
review_vector = vectorizer.transform([review]) # vectorizing
print(classifier_linear.predict(review_vector))


review = """Do not purchase this product. My cell phone blast when I switched the
    charger"""
review_vector = vectorizer.transform([review]) # vectorizing
print(classifier_linear.predict(review_vector))
```

Chapitre 5

Language Modeling : Deep Learning

Deep Learning is a sub-field of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. Deep learning is a procedure of machine learning that teaches computers to do what comes of course to humans : learn by example.

5.1 Artificial Neural Network

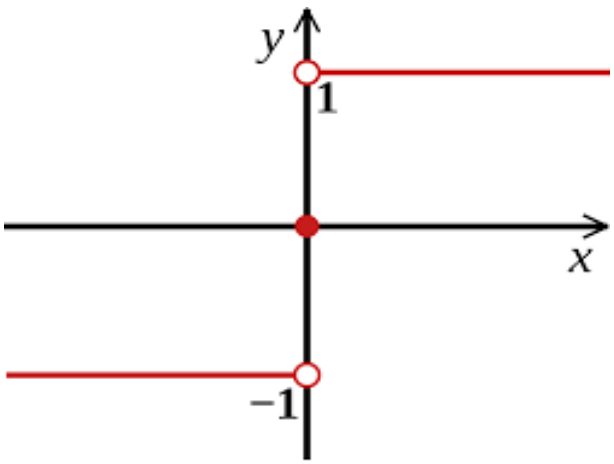
Neural computing is an information processing paradigm, inspired by biological system, composed of a large number of highly interconnected processing elements(neurons) working in unison to solve specific problems.
Artificial neural networks (ANNs), like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.

5.1.1 Perceptron

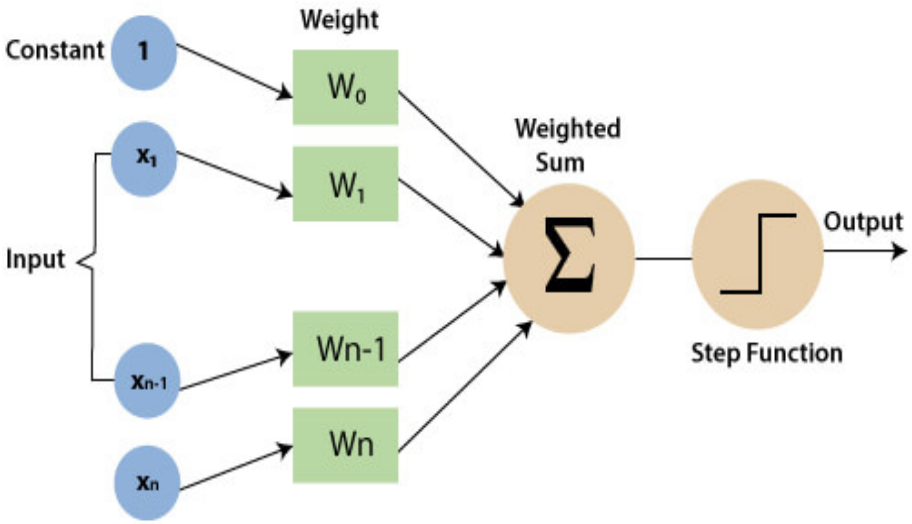
$$x \in R^d, y \in \{0, 1\}$$

$$h_\theta = g(\theta^T X)$$

$$g(z) = \begin{cases} 1 & \text{if z} \geq 0 \\ 0 & \text{if z} < 0 \end{cases} \tag{5.1}$$



Perceptron schema



If we then let $h_\theta(x) = g(\theta^T X)$ as before but using this modified definition of g, and if we use the update rule

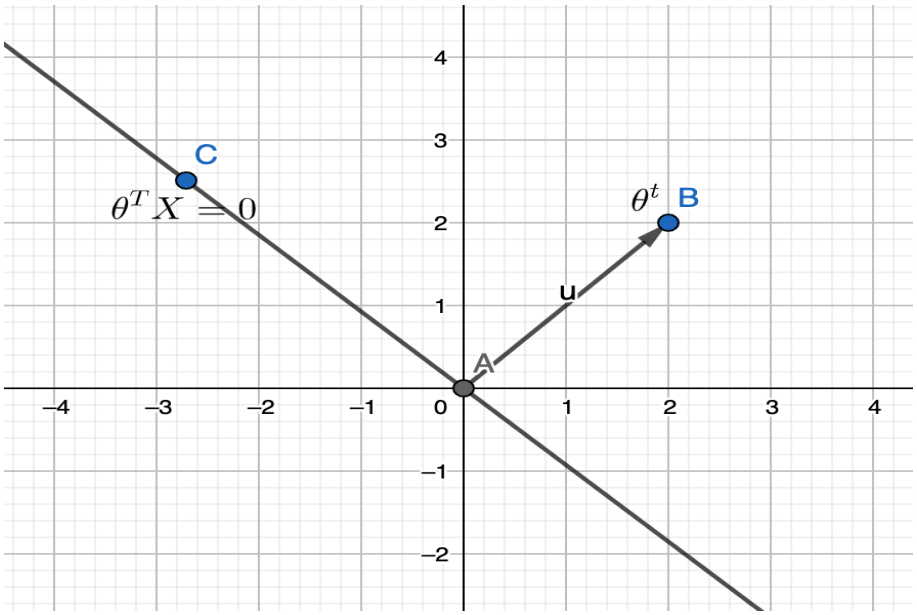
$$\theta_{(j)} = \theta_{(j)} - \alpha(y^{(i)} - h_\theta(x^{(i)}))x^{(i)})_{(j)}$$

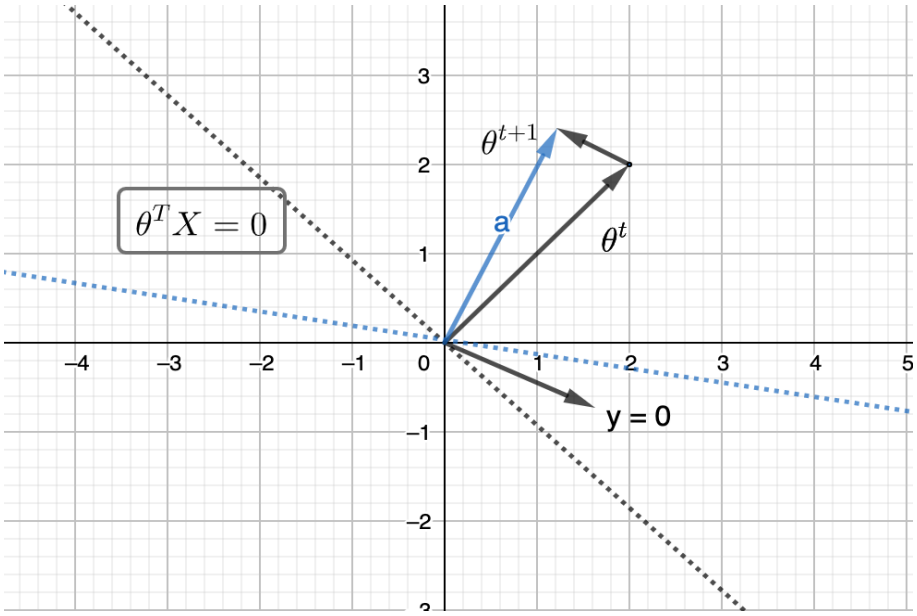
Then we have the perceptron learning algorithm.

Learning Algorithm

$\vec{\theta} \leftarrow init(\vec{0})$
For in in 1,2.....,n :

$$\theta_{(j)} = \theta_{(j)} - \alpha(y^{(i)} - h_\theta(x^{(i)}))x^{(i)})_{(j)}$$





Perceptron with Pytorch

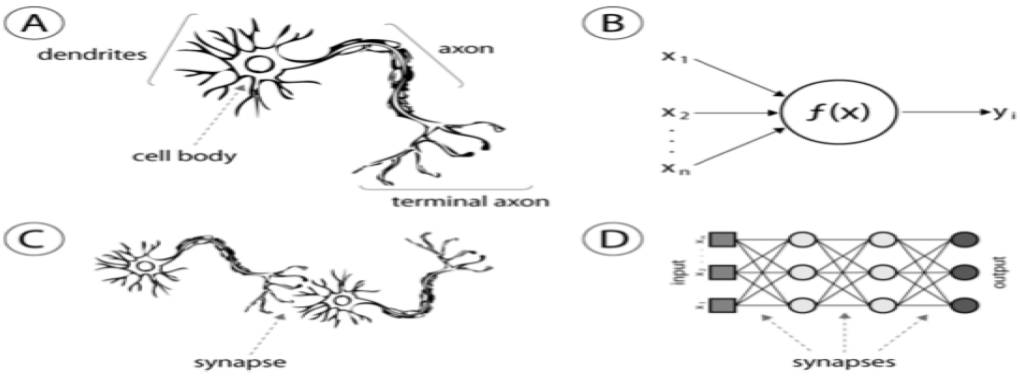
Pytorch Perceptron

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import pandas as pd
%matplotlib inline

def custom_where(cond, x_1, x_2):
    return (cond * x_1) + ((~(cond)) * x_2)


class Perceptron():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = torch.zeros(num_features, 1,dtype=torch.float32, device=
    device)
        self.bias = torch.zeros(1, dtype=torch.float32, device=device)

    def forward(self, x):
        linear = torch.add(torch.mm(x, self.weights), self.bias)
        predictions = custom_where(linear > 0., 1, 0).float()
        return predictions

    def backward(self, x, y):
        predictions = self.forward(x)
        errors = y - predictions
        return errors

    def train(self, x, y, epochs):
```

```
        for e in range(epochs):

            for i in range(y.size()[0]):
                # use view because backward expects a matrix (i.e., 2D tensor)
                errors = self.backward(x[i].view(1, self.num_features), y[i]).view
    (-1)
                self.weights += (errors * x[i]).view(self.num_features, 1)
                self.bias += errors

    def evaluate(self, x, y):
        predictions = self.forward(x).view(-1)
        accuracy = torch.sum(predictions == y).float() / y.size()[0]
        return accuracy

 data = pd.read_csv("datasets/diabetes.csv")
 data

array = data.values
X, y = array[:,0:2] , array[:,8]

from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2)


from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform (X_test)

plt.scatter(X_train[y_train==0, 0], X_train[y_train==0, 1], label='class 0', marker
    ='o')
plt.scatter(X_train[y_train==1, 0], X_train[y_train==1, 1], label='class 1', marker
    ='s')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.legend()
plt.show()

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

ppn = Perceptron(num_features=2)

X_train_tensor = torch.tensor(X_train, dtype=torch.float32, device=device)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32, device=device)

ppn.train(X_train_tensor, y_train_tensor, epochs=5)
```

```
print('Model parameters:')
print('  Weights: %s' % ppn.weights)
print('  Bias: %s' % ppn.bias)

X_test_tensor = torch.tensor(X_test, dtype=torch.float32, device=device)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32, device=device)

test_acc = ppn.evaluate(X_test_tensor, y_test_tensor)
print('Test set accuracy: %.2f%%' % (test_acc*100))

w, b = ppn.weights, ppn.bias

x_min = -2
y_min = ( (-(w[0] * x_min) - b[0])
          / w[1] )

x_max = 2
y_max = ( (-(w[0] * x_max) - b[0])
          / w[1] )


fig, ax = plt.subplots(1, 2, sharex=True, figsize=(7, 3))

ax[0].plot([x_min, x_max], [y_min, y_max])
ax[1].plot([x_min, x_max], [y_min, y_max])

ax[0].scatter(X_train[y_train==0, 0], X_train[y_train==0, 1], label='class 0',
    marker='o')
ax[0].scatter(X_train[y_train==1, 0], X_train[y_train==1, 1], label='class 1',
    marker='s')

ax[1].scatter(X_test[y_test==0, 0], X_test[y_test==0, 1], label='class 0', marker='
    o')
ax[1].scatter(X_test[y_test==1, 0], X_test[y_test==1, 1], label='class 1', marker='
    s')

ax[1].legend(loc='upper left')
plt.show()
```

5.1.2 Multi layer Perceptron



— MLPs are feedforward neural networks (no feedback connections).
— They compose several non-linear functions :

$$f(x) = y(h_3(h_2(h_1(x))))$$

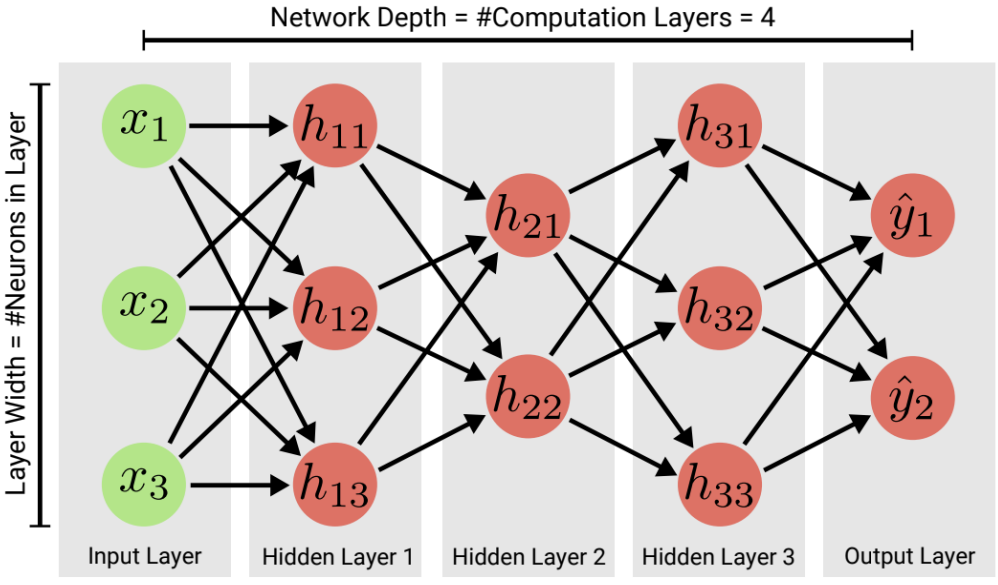
— The data specifies only the behavior of the output layer.
— Each layer i comprises multiple neurons j which are implemented as affine transformations

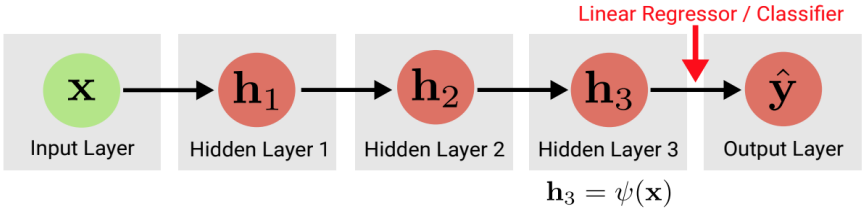
$$(a^T x + b)$$

followed by non-linear activation functions (g) :

$$h_{ij} = g(a_{ij} h_{i-1} + b_{ij})$$

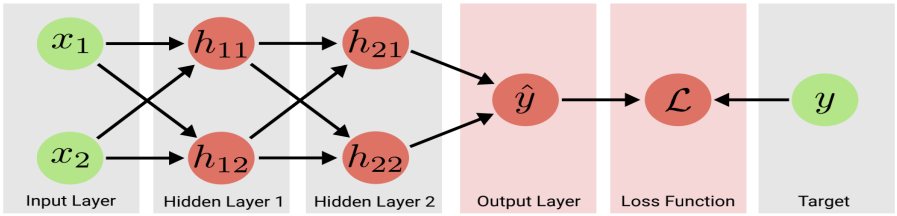
.
— Each neuron in each layer is fully connected to all neurons of the previous layer.
— The overall length of the chain is the depth of the model – "Deep Learning".

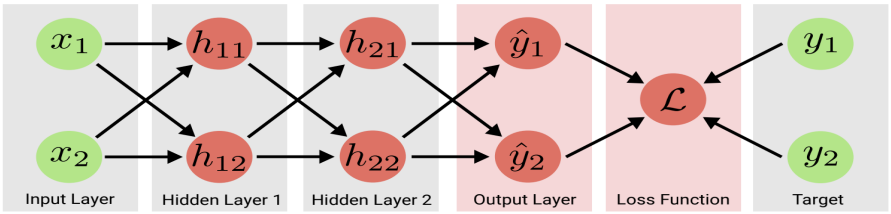




5.1.3 Loss function

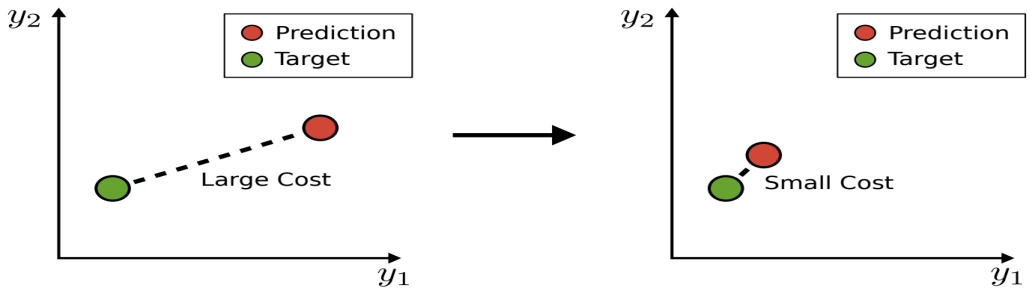
The loss function in a neural network quantifies the difference between the expected outcome and the outcome produced by the machine learning model. From the loss function, we can derive the gradients which are used to update the weights. The average over all losses constitutes the cost.

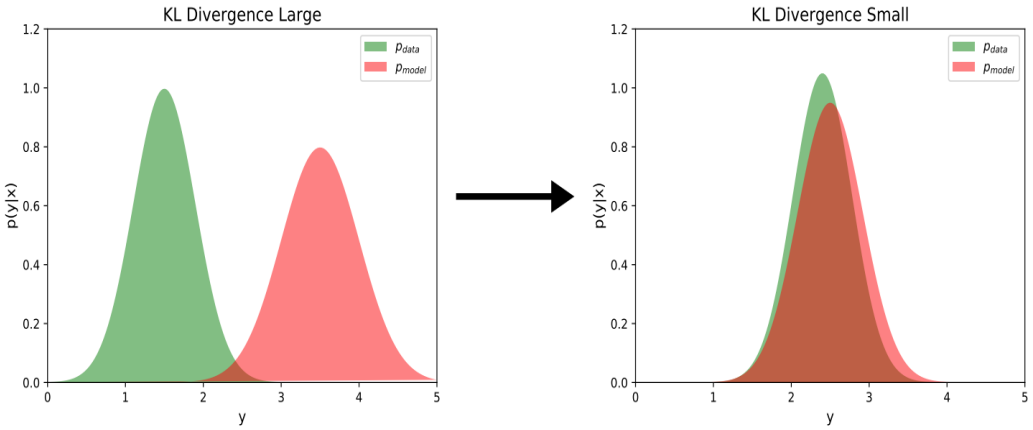




— The output layer is the last layer in a neural network which computes the output
— I The loss function compares the result of the output layer to the target value(s)
— I Choice of output layer and loss function depends on task (discrete, continuous, ..)

What is the goal of optimizing the loss function ?
— I Tries to make the model output (=prediction) similar to the target (=data)
— I Think of the loss function as a measure of cost being paid for a prediction





How to design a good loss function ?

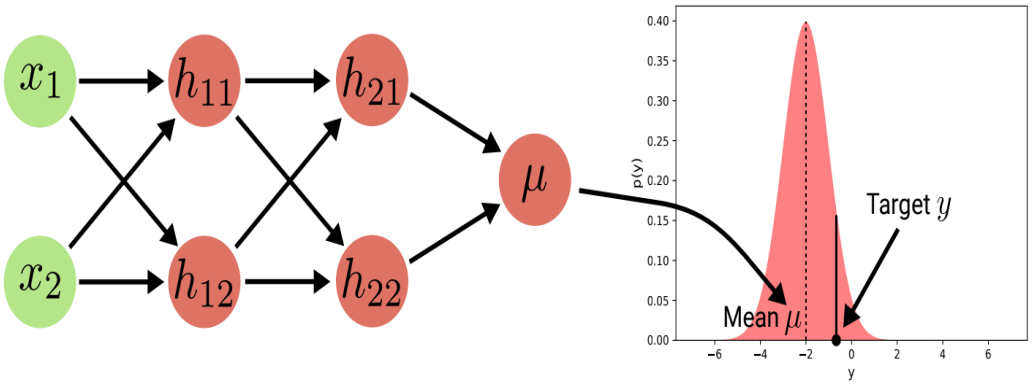
— A loss function can be any differentiable function that we wish to optimize
— Deriving the cost function from the maximum likelihood principle removes the burden of manually designing the cost function for each model
— Consider the output of the neural network as parameters of a distribution over y_i

Log-Likelihood :

$$w_{ML} = argmax\, p_{model}(y|X, w)$$

$$w_{ML} = argmax \prod_{i=1}^{N} p_{model}(y_i|x_i, w)$$

$$w_{ML} = argmax \sum_{i=1}^{N} log\, p_{model}(y_i|x_i, w)$$



— Neural network $f_w(x)$ predicts mean u of Gaussian distribution over y :

$$p(y|x, w) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(y - f_w(x))^2}{2\sigma^2})$$

— We want to maximize the probability of the target y under this distribution

Regression : Mean Squared Error /L_2 Loss

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (f_w(x_i) - y_i)^2$$

We minimize the squared loss (=L_2 loss), affected strongly by outliers.

Regression : Mean Absolute Error /L_1 Loss

$$MAE = \frac{1}{N} \sum_{i=1}^{N} |f_w(x_i) - y_i|$$

We minimize the absolute loss (=L_1 loss) which is more robust than L_2 .

Binary Classification : Binary cross-entropy

$$w_{ML} = argmin_w \sum_{i=1}^{N} -y_i log f_W(X_i) - (1 - y_i) log(1 - f_W(X_i))$$

In other words, we minimize the binary cross-entropy (BCE) loss.
Remark : Last layer of $f_w(x)$ can be a sigmoid function such that $f_w(x)y \in [0, 1]$.

$$w_{ML} = argmin_w \sum_{i=1}^{N} \sum_{c=1}^{C} -y_{i,c} log f_w^{(c)}(x_i)$$

In other words, we minimize the cross-entropy (CE) loss.

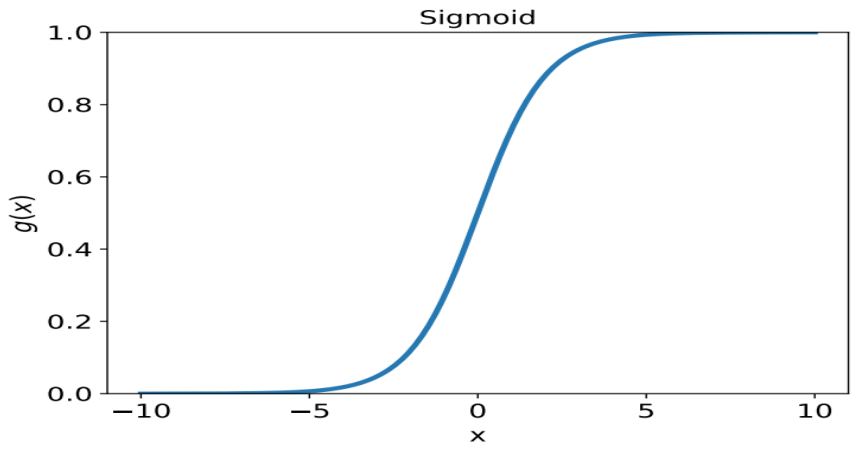
5.1.4 Activation functions

An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

The role of the Activation Function is to derive output from a set of input values fed to a node.

— Hidden layer $h_i = g(A_i h_{i-1} + b_i)$ with activation function g(·) and weights A_i, b_i
— The activation function is frequently applied element-wise to its input
— Activation functions must be non-linear to learn non-linear mappings
— Some of them are not differentiable everywhere (but still ok for training)

Sigmoid



$$g(x) = \frac{1}{1 + e^{-x}}$$

— Maps input to range [0, 1].
— Neuroscience interpretation as saturating "firing rate" of neurons.

Problems :

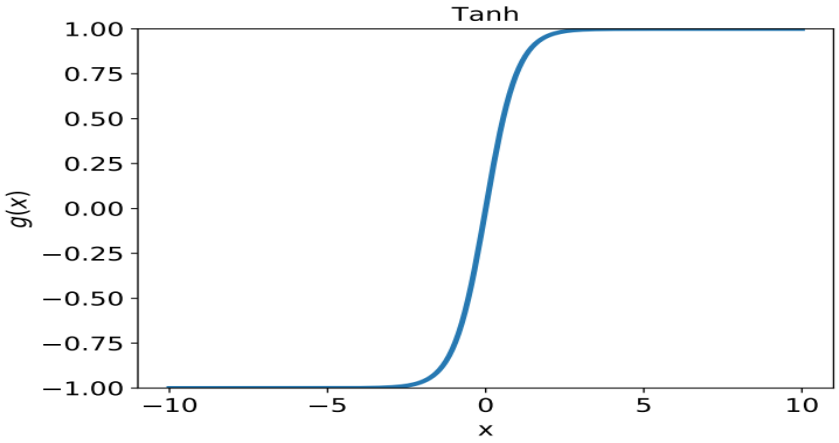
Saturation "kills" gradients (gradients vanishing) :

— Downstream gradient becomes zero when input x is saturated : g'(x) = 0
— No learning if x is very small (<-10)
— No learning if x is very large (>10)

Non zero-centered outputs :

— The output is always between 0 and 1
— The gradient updates go too far in different directions which makes optimization harder.

Tanh

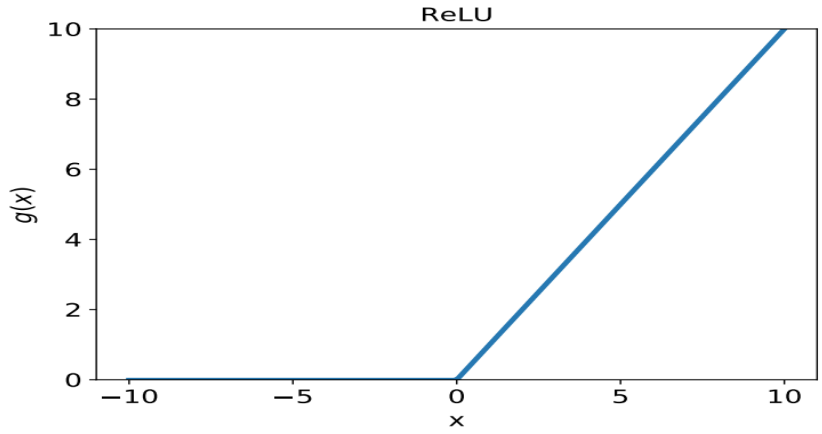


$$g(x) = \frac{2}{1 + e^{-2x}} - 1 = \frac{e^{x} \breve{} e^{-x}}{e^{x} + e^{-x}}$$

— Maps input to range [0, 1].
— Zero-centered.

Problems :
Again, saturation "kills" gradients.

ReLU



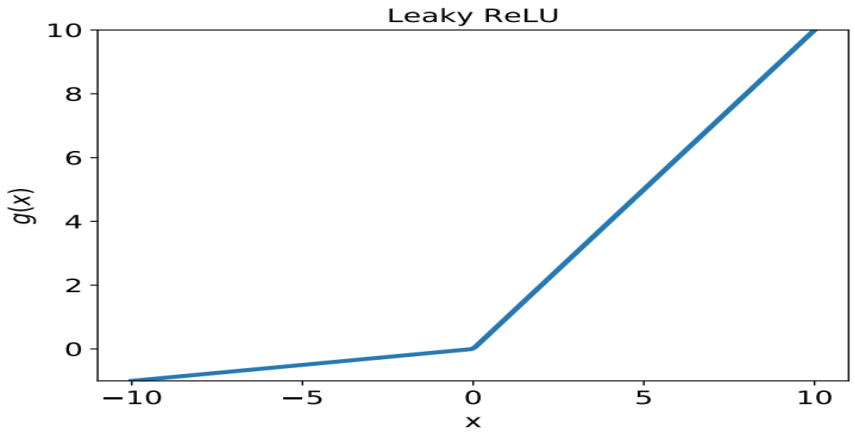
$$g(x) = max(0, x)$$

— Does not saturate (for x $>$ 0).
— Leads to fast convergence.
— Computationally efficient

Problems :

— Not zero-centered
— No learning for x $<$ 0, dead ReLUs

Leaky ReLU

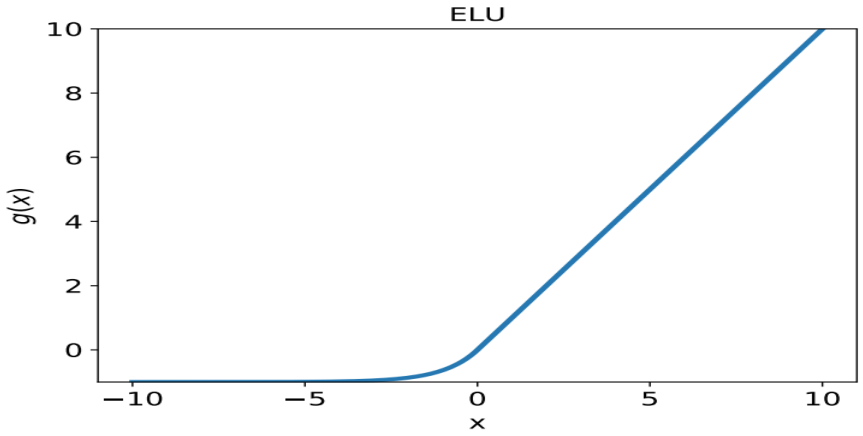


$$g(x) = max(0.01x, x)$$

— Does not saturate (i.e., will not die).
— Closer to zero-centered outputs.
— Leads to fast convergence.
— Computationally efficient.

Note : there is also an alternative Parametric ReLU : $g(x) = max(\alpha x, x)$ with the same advantages as Leaky ReLU and Parameter α learned from data.

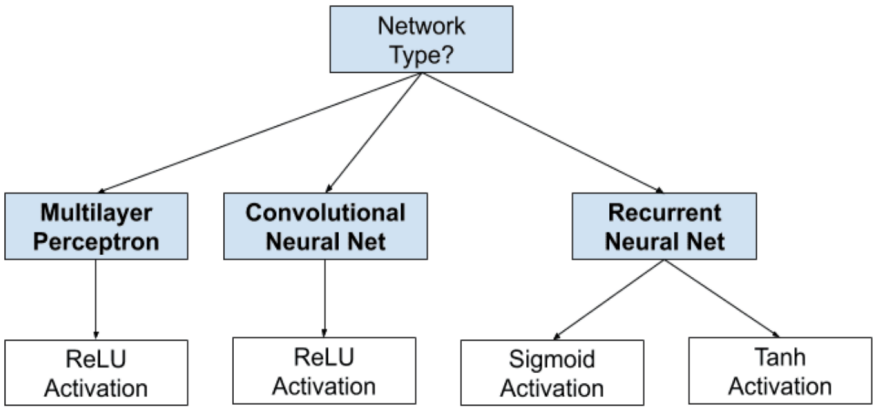
Exponential Linear Units (ELU)



$$g(x) = \begin{cases} \text{x} & \text{if x} > 0 \\ \alpha(e^x - 1) & \text{if x} \leq 0 \end{cases} \tag{5.2}$$

— All benefits of Leaky ReLU
— Adds some robustness to noise
— Default $\alpha = 1$

How to choose an activation function



5.1.5 Optimization

Machine/Deep learning involves using an algorithm to learn and generalize from historical data in order to make predictions on new data.

This problem can be described as approximating a function that maps examples of inputs to examples of outputs. Approximating a function can be solved by framing the problem as function optimization.

Machine/Deep learning optimization is the process of adjusting hyperparameters in order to minimize the cost function by using one of the optimization techniques. It is important to minimize the cost function because it describes the discrepancy between the true value of the estimated parameter and what the model has predicted.

Gradient Descent

Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error.
Algorithm :

1. Initialize weights w_0 and pick learning rate η.
2. For all data points $i \in 1, ..., N$ do :
   (a) Forward propagate x_i through network to calculate prediction $\hat{y}_i$
   (b) Backpropagate to obtain gradient$\nabla_w L_i(w^t) = \nabla_w L_i(\hat{y}_i, y_i, w^t)$
3. Update gradients :$w^{t+1} = w^t - \eta \frac{1}{N} \Sigma_i \nabla_w L_i(w^t)$.
4. If validation error decreases, go to step 2, otherwise stop.

— Typically, millions of parameter dim(w) = 1 million or more
— Typically, millions of training points N = 1 million or more
— Becomes extremely expensive to compute and doesn't fit into memory

Stochastic Gradient Descent

The word 'stochastic' means a system or process linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration.
Algorithm :

1. Initialize weights w_0 and pick learning rate η and minibatch size $|X_{batch}|$.
2. Draw random minibatch $(x_1, y_1), ..., (x_B, y_B) \subseteq X(withB << N)$
3. For all minibatch elements $b \in 1, ..., B$ do :
   (a) Forward propagate x_i through network to calculate prediction $\hat{y}_i$
   (b) Backpropagate to obtain gradient$\nabla_w L_i(w^t) = \nabla_w L_i(\hat{y}_i, y_i, w^t)$
4. Update gradients :$w^{t+1} = w^t - \eta \frac{1}{N} \Sigma_i \nabla_w L_i(w^t)$.
5. If validation error decreases, go to step 2, otherwise stop.

— We call $(x_1, y_1), ..., (x_B, y_B) \subseteq X$ a "minibatch"
— You should choose B as large as your (GPU) memory allows Typically B N, e.g., B = 8, 16, 32, 64, 128
— Smaller batch sizes lead to larger variance in the gradients (noisy updates)
— Batches can be chosen randomly or by partitioning the dataset

$$w^{t+1} = w^t - \eta \Sigma_i \nabla_w L_i(w^t)$$

$$w^1 = w^0 - \eta \Sigma_i \nabla_w L_0$$

$$w^2 = w^1 - \eta \Sigma_i \nabla_w L_1 = w^0 - \eta \Sigma_i \nabla_w L_0 - \eta \Sigma_i \nabla_w L_1$$

$$w^3 = w^2 - \eta \Sigma_i \nabla_w L_2 = w^0 - \eta \Sigma_i \nabla_w L_0 - \eta \Sigma_i \nabla_w L_1 - \eta \Sigma_i \nabla_w L_2$$

Optimization converges if there exists a vector w^* such that for every arbitrarily small positive number ϵ, there exists an integer T such that for all $t \geq T$:

$$|| \ w^t - w^* \ || < \epsilon$$

— Requires conservative learning rate to avoid divergence.
— However, in this case the updates become very small, slow progress.
— Finding a good learning rate is difficult.

SGD with Momentum

— SGD oscillates along w_2 axis, we should dampen, e.g., by averaging over time.
— SGD makes slow progress along w_1 axis, we like to accelerate in this direction.
— Idea of momentum : update weights with exponential moving average of gradients.

$$m^{t+1} = \beta_1 m^t - \eta \nabla_w L_B(w^t)$$

$$w^{t+1} = w^t + m^{t+1}$$

With velocity m and momentumβ_1, typically β_1= 0.9.

Exponential Moving Average :

We can write the expression of m^{t+1} as below :

$$m^{t+1} = \beta_1 m^t - (1-\beta_1)\nabla_w L_B(w^t)$$

$$w^{t+1} = w^t + \eta m^{t+1}$$

Let us abbreviate the gradient at iteration t with $g_t \equiv \nabla_w L_B(w^t)$. We have :

$$m^{t+1} = \beta_1 m^t - (1-\beta_1) g_t$$

with $(m^0 = 0)$.

$$m^1 = \beta_1 m^0 - (1-\beta_1) g_0) = (1-\beta_1) g_0$$

$$m^2 = \beta_1 m^1 - (1-\beta_1) g_1) = \beta_1(1-\beta_1) g_0 + (1-\beta_1) g_1$$

$$m^3 = \beta_2 m^1 - (1-\beta_1) g_2) = \beta_1(1-\beta_1) g_0 + \beta_1(1-\beta_1) g_1 + (1-\beta_1) g_2$$

We see that the weight decays exponentially :

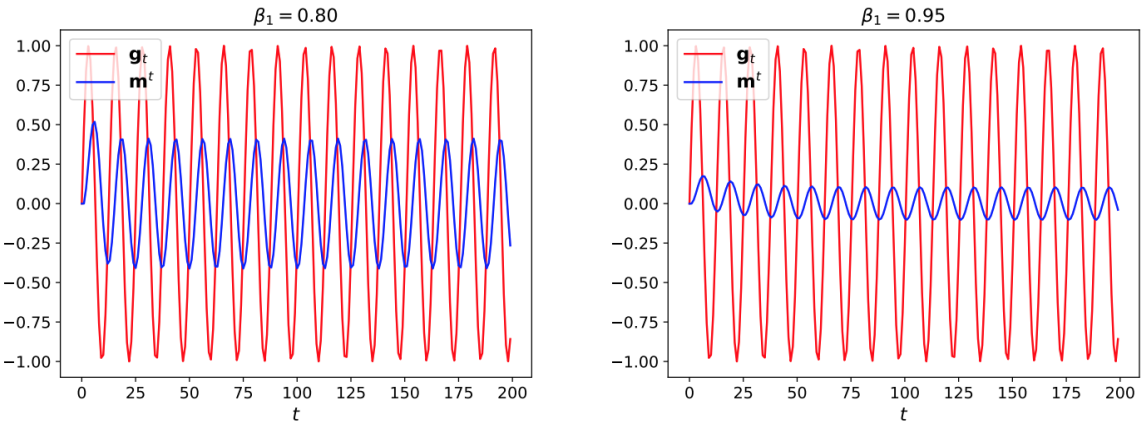
$$m^t = (1-\beta_1)\Sigma_{i=0}^{t-1} \beta_1^{t-i-1} g_i$$

Example :

$t_1, t_2, t_3, ..., t_n$
$b_1, b_2, b_{13}, ..., b_n$

with $0 \geq \gamma \leq 1$.

$V_{t1} = b_1$
$V_{t2} = \gamma V_{t1} + b_2$
$V_{t3} = \gamma^2 V_{t1} + V_{t2}\gamma + b_3$



SGD with Nesterov Momentum : Leads to faster dampening and Has significantly increased the performance of RNNs on a variety of tasks.

RMSprop

Root Mean Squared Propagation, or RMSProp, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter, The main idea is "Divide the gradient by a running average of its recent magnitude"..
Motivation for RMSprop :

- Gradient distribution is very uneven (not equal)and thus requires conservative learning rates.
- In this SGD example, gradients are very large in w_2 but small in w_1.
- Idea of RMSprop : divide learning rate by moving average of squared gradients.

$$v^{t+1} = \beta_2 v^t + (1-\beta_2)\nabla_w^2 L_B(w^t)$$

$$w^{t+1} = w^t - \eta \frac{\nabla_w L_B(w^t)}{\sqrt{v^{t+1}} + \epsilon}$$

With uncentered variance of gradient v, momentum $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$

Adam

Adaptive Moment Estimation (Adam) is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients m_t, similar to momentum and also the decaying average of the past squared gradients v_t, similar to RMSprop and Adadelta. Thus, it combines the advantages of both the methods. Adam is the default choice of the optimizer for any application in general.

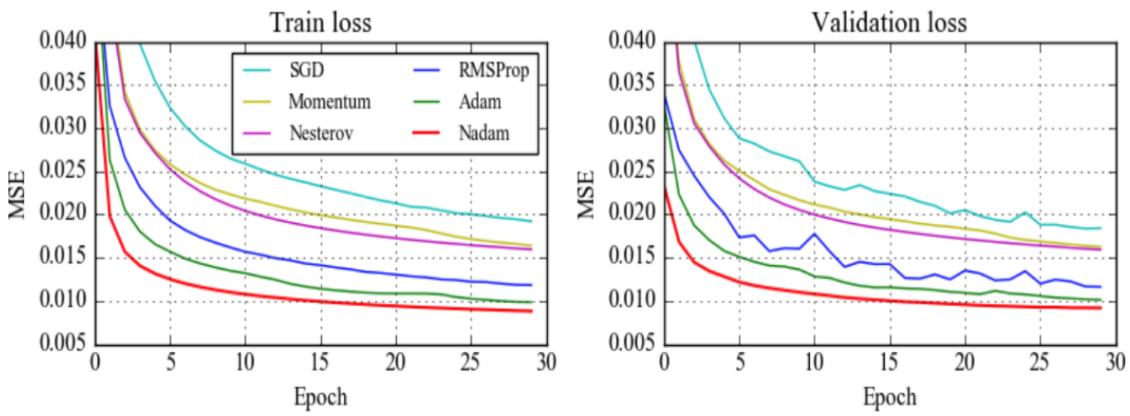
$$m^{t+1} = \beta_1 m^t + (1-\beta_1)\nabla_w L_B(w^t)$$

$$v^{t+1} = \beta_2 v^t + (1-\beta_2)\nabla_w^2 L_B(w^t)$$

$$w^{t+1} = w^t - \alpha \frac{m^{t+1}}{\sqrt{v^{t+1}} + \epsilon}$$

Adam combines the benefits of Momentum and RMSprop.

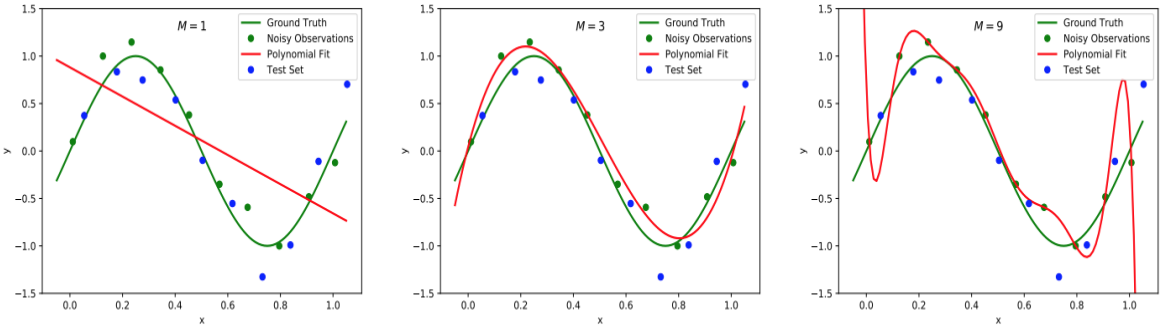
Optimizers Comparison



Visualization : https://github.com/Jaewan-Yun/optimizer-visualization

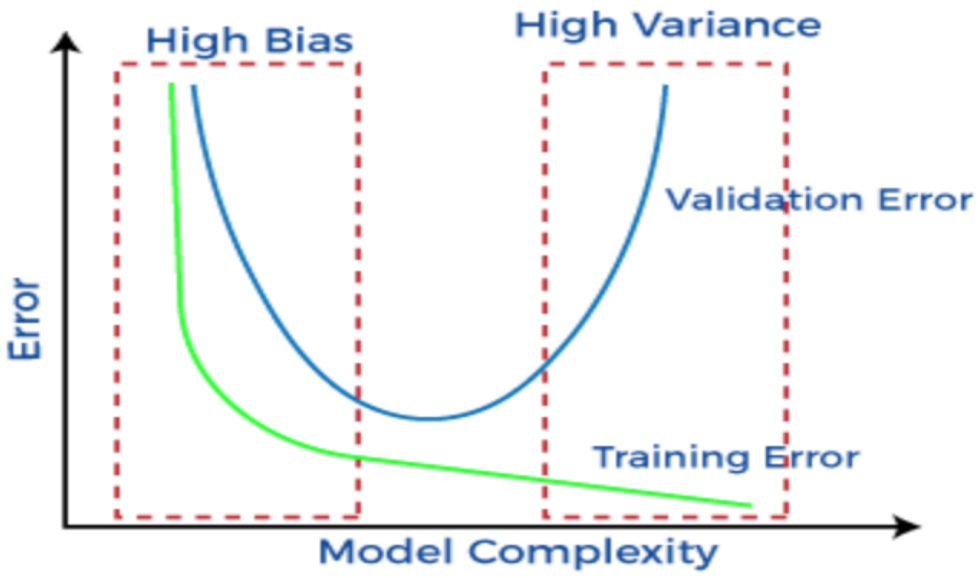
5.1.6 Regularization

Capacity of the model



— Underfitting : Model too simple, does not achieve low error on training set.
— Overfitting : Training error small, but test error (= generalization error) large.

Bias and Variance



Bais : While making predictions, a difference occurs between prediction values made by the model and actual values/expected values, and this difference is known as bias errors or Errors due to bias.

— Low Bias : A low bias model will make fewer assumptions about the form of the target function.
— High Bias : A model with a high bias makes more assumptions, and the model becomes unable to capture the important features of our dataset. A high bias model also cannot perform well on new data.

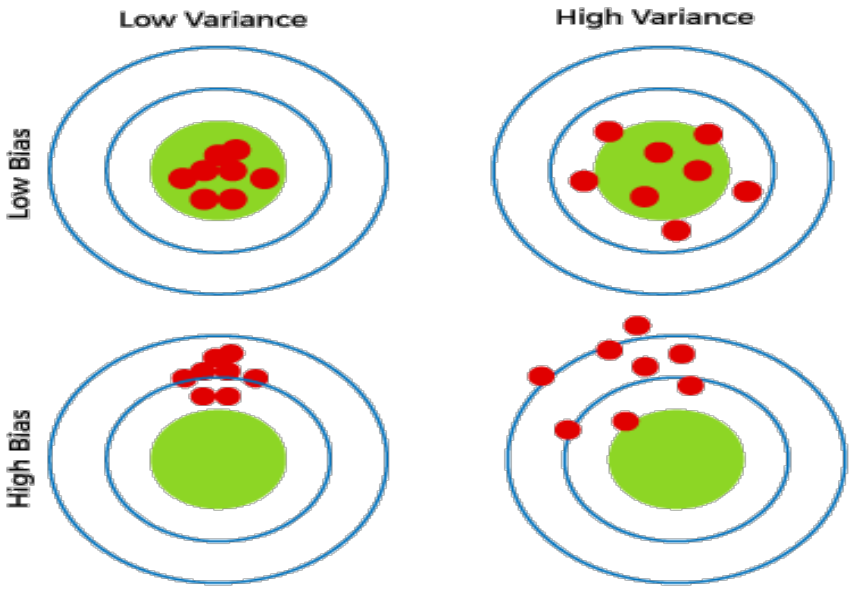
Ways to reduce High Bias :

— Increase the input features as the model is underfitted.
— Decrease the regularization term.

— Use more complex models, such as including some polynomial features.

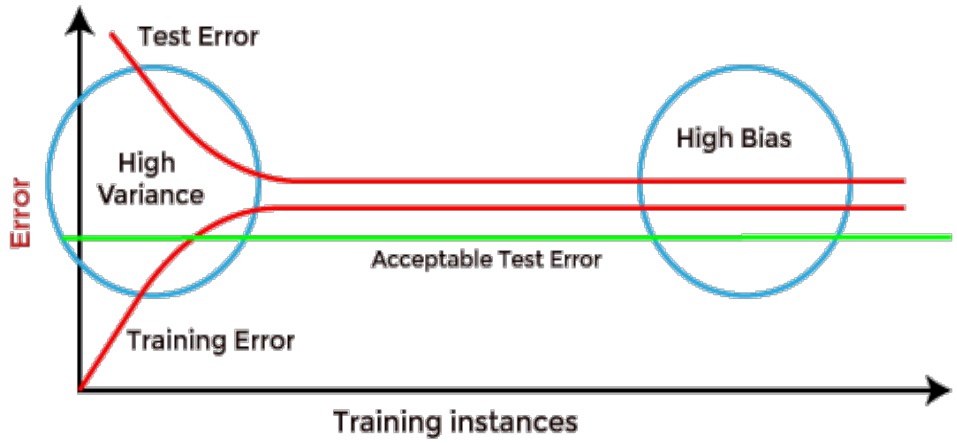
Variance : Variance tells that how much a random variable is different from its expected value. Ideally, a model should not vary too much from one training dataset to another, which means the algorithm should be good in understanding the hidden mapping between inputs and output variables.
Ways to reduce High Variance :

— Reduce the input features or number of parameters as a model is overfitted.
— Do not use a much complex model.
— Increase the training data.
— Increase the Regularization term.



1. Low-Bias, Low-Variance : The combination of low bias and low variance shows an ideal machine learning model. However, it is not possible practically.
2. Low-Bias, High-Variance : With low bias and high variance, model predictions are inconsistent and accurate on average. This case occurs when the model learns with a large number of parameters and hence leads to an overfitting
3. High-Bias, Low-Variance : With High bias and low variance, predictions are consistent but inaccurate on average. This case occurs when a model does not learn well with the training dataset or uses few numbers of the parameter. It leads to underfitting problems in the model.
4. High-Bias, High-Variance : With high bias and high variance, predictions are inconsistent and also inaccurate on average.

How to identify High variance or High Bias ?



5.1.7 L1 and L2 Regularzation

Regularization refers to a set of different techniques that lower the complexity of a neural network model during training, and thus prevent the overfitting.

Let X = (X, y) denote the dataset and w the model parameters. We can limit the model capacity by adding a parameter norm penalty R to the loss L.

$$\hat{L}(X, w) = L(X, w) + \alpha R(w)$$

where $\alpha \in [0, \infty)$ controls the strength of the regularizer.
There are three very popular and efficient regularization techniques called L1, L2, and dropout.

L2 Regularzation

The L2 regularization is the most common type of all regularization techniques and is also commonly known as weight decay or Ridge Regression.

L2 regularization, or the L2 norm, or Ridge (in regression problems), combats overfitting by forcing weights to be small, but not making them exactly 0.

So, if we're predicting house prices again, this means the less significant features for predicting the house price would still have some influence over the final prediction, but it would only be a small influence.

$$\hat{L}(X, w) = L(X, w) + \alpha \frac{1}{2} ||W||_2^2$$

$$\hat{L}(X, w) = L(X, w) + \alpha \Sigma_{i=1}^{n} w_i^2$$

L1 Regularzation

L1 regularization, also known as L1 norm or Lasso (in regression problems), combats overfitting by shrinking the parameters towards 0. This makes some features obsolete.
It's a form of feature selection, because when we assign a feature with a 0 weight, we're multiplying the feature values by 0 which returns 0, eradicating the significance of that feature.

$$\hat{L}(X, w) = L(X, w) + \alpha \frac{1}{2} ||W||_1$$

$$\hat{L}(X, w) = L(X, w) + \alpha \Sigma_{i=1}^{n} |w_i|$$

L1 vs L2

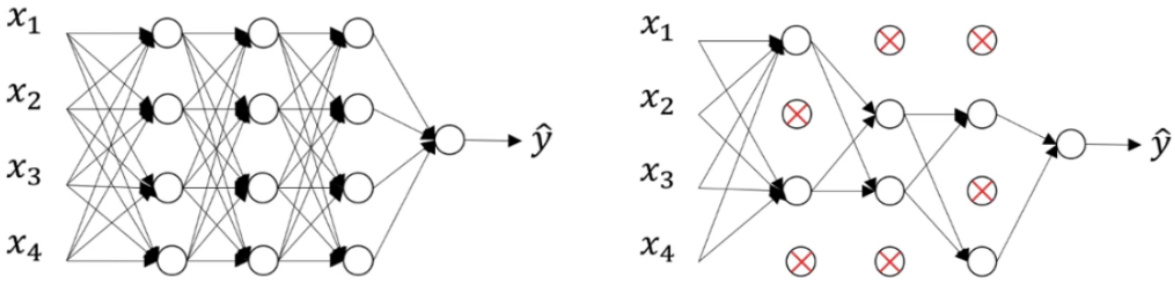
— L1 regularization penalizes the sum of absolute values of the weights, whereas L2 regularization penalizes the sum of squares of the weights.
— The L1 regularization solution is sparse. The L2 regularization solution is non-sparse.
— L2 regularization doesn't perform feature selection, since weights are only reduced to values near 0 instead of 0. L1 regularization has built-in feature selection.
— L1 regularization is robust to outliers, L2 regularization is not.

5.1.8 Early Stop

— Most commonly used form of regularization in deep learning
— Effective, simple and computationally efficient form of regularization
— Training time can be viewed as hyperparameter, model selection problem
— Efficient as a single training run tests all hyperparameters (unlike weight decay)
— Only cost : periodically evaluate validation error on validation set
— Validation set can be small, and evaluation less frequently

5.1.9 Dropout

Dropout means that during training with some probability P a neuron of the neural network gets turned off during training. Let's look at a visual example.



5.1.10 DNN based on pytorch

DNN Pytorch

```
from numpy import vstack
from numpy import sqrt
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
```

```
from torch.nn import Linear
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD
from torch.nn import MSELoss
from torch.nn.init import xavier_uniform_
from tqdm import tqdm

# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path, header=None)
        # store the inputs and outputs
        self.X = df.values[:, :-1].astype('float32')
        self.y = df.values[:, -1].astype('float32')
        # ensure target has the right shape
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.33):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])
# model definition
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 10)
        xavier_uniform_(self.hidden1.weight)
        self.act1 = Sigmoid()
        # second hidden layer
        self.hidden2 = Linear(10, 8)
```

```
        xavier_uniform_(self.hidden2.weight)
        self.act2 = Sigmoid()
        # third hidden layer and output
        self.hidden3 = Linear(8, 1)
        xavier_uniform_(self.hidden3.weight)

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
         # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # third hidden layer and output
        X = self.hidden3(X)
        return X
# prepare the dataset
def prepare_data(path):
    # load the dataset
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl

# train the model
def train_model(train_dl, model):
    size = len(train_dl.dataset)
    # define the optimization
    criterion = MSELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    # enumerate epochs
    for epoch in tqdm(range(100),desc='Training Epochs'):
        print(f"Epoch {epoch+1}\n-------------------------------")
        # enumerate mini batches
        for batch, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
```

```
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()

            #if batch % 100 == 0:
        loss, current = loss.item(), batch * len(inputs)
        print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

 # evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate mse
    mse = mean_squared_error(actuals, predictions)
    return mse

# make a class prediction for one row of data
def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat

# prepare the data
path = 'datasets/housing.csv'
train_dl, test_dl = prepare_data(path)
print(len(train_dl.dataset), len(test_dl.dataset))
# define the network
model = MLP(13)
# train the model
train_model(train_dl, model)
# evaluate the model
```

```
mse = evaluate_model(test_dl, model)
print('MSE: %.3f, RMSE: %.3f' % (mse, sqrt(mse)))
# make a single prediction (expect class=1)
row = [0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0,15.30,396.90,4.98]
yhat = predict(row, model)
print('Predicted: %.3f' % yhat)
```

DNN Pytorch

```
# pytorch mlp for binary classification
from numpy import vstack
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD
from torch.nn import BCELoss
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_
from tqdm import tqdm

# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path, header=None)
        # store the inputs and outputs
        self.X = df.values[:, :-1]
        self.y = df.values[:, -1]
        # ensure input data is floats
        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)
        self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)
```

```
    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.33):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])

# model definition
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        # second hidden layer
        self.hidden2 = Linear(10, 8)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # third hidden layer and output
        self.hidden3 = Linear(8, 1)
        xavier_uniform_(self.hidden3.weight)
        self.act3 = Sigmoid()

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
         # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # third hidden layer and output
        X = self.hidden3(X)
        X = self.act3(X)
        return X

# prepare the dataset
def prepare_data(path):
```

```
    # load the dataset
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl

# train the model
def train_model(train_dl, model):
    size = len(train_dl.dataset)
    # define the optimization
    criterion = BCELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    for epoch in tqdm(range(100),desc='Training Epochs'):
        print(f"Epoch {epoch+1}\n-------------------------------")
        # enumerate mini batches
        for batch, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()

            #if batch % 100 == 0:
        loss, current = loss.item(), batch * len(inputs)
        print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

 # evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # round to class values
```

```
        yhat = yhat.round()
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return acc

# make a class prediction for one row of data
def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat

 #prepare the data
path = 'datasets/ionosphere.csv'
train_dl, test_dl = prepare_data(path)
print(len(train_dl.dataset), len(test_dl.dataset))
# define the network
model = MLP(34)
# train the model
train_model(train_dl, model)
# evaluate the model
acc = evaluate_model(test_dl, model)
print('Accuracy: %.3f' % acc)
# make a single prediction (expect class=1)
row = [1,0,0.99539,-0.05889,0.85243,0.02306,0.83398,-0.37708,1,0.03760,0.85243,-0.1
    7755,0.59755,-0.44945,0.60536,-0.38223,0.84356,-0.38542,0.58212,-0.32192,0.56971
    ,-0.29674,0.36946,-0.47357,0.56811,-0.51171,0.41078,-0.46168,0.21266,-0.34090,0.
    42267,-0.54487,0.18641,-0.45300]
yhat = predict(row, model)
print('Predicted: %.3f (class=%d)' % (yhat, yhat.round()))
```

DNN Pytorch

```
# pytorch mlp for multiclass classification
from numpy import vstack
from numpy import argmax
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from torch import Tensor
```

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import Softmax
from torch.nn import Module
from torch.optim import SGD
from torch.nn import CrossEntropyLoss
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_
from tqdm import tqdm

# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path, header=None)
        # store the inputs and outputs
        self.X = df.values[:, :-1]
        self.y = df.values[:, -1]
        # ensure input data is floats
        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.33):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])


# model definition
class MLP(Module):
```

```
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        # second hidden layer
        self.hidden2 = Linear(10, 8)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # third hidden layer and output
        self.hidden3 = Linear(8, 3)
        xavier_uniform_(self.hidden3.weight)
        self.act3 = Softmax(dim=1)

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
        # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # output layer
        X = self.hidden3(X)
        X = self.act3(X)
        return X


# prepare the dataset
def prepare_data(path):
    # load the dataset
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl


# train the model
def train_model(train_dl, model):
    size = len(train_dl.dataset)
    # define the optimization
```

```
    criterion = CrossEntropyLoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    for epoch in tqdm(range(500),desc='Training Epochs'):
        print(f"Epoch {epoch+1}\n-------------------------------")
        # enumerate mini batches
        for batch, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()

            #if batch % 100 == 0:
        loss, current = loss.item(), batch * len(inputs)
        print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

# evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        # convert to class labels
        yhat = argmax(yhat, axis=1)
        # reshape for stacking
        actual = actual.reshape((len(actual), 1))
        yhat = yhat.reshape((len(yhat), 1))
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return acc


# make a class prediction for one row of data
```

```
def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat


# prepare the data
path = 'datasets/Iris.csv'
train_dl, test_dl = prepare_data(path)
print(len(train_dl.dataset), len(test_dl.dataset))
# define the network
model = MLP(4)
# train the model
train_model(train_dl, model)
# evaluate the model
acc = evaluate_model(test_dl, model)
print('Accuracy: %.3f' % acc)
# make a single prediction
row = [5.1,3.5,1.4,0.2]
yhat = predict(row, model)
print('Predicted: %s (class=%d)' % (yhat, argmax(yhat)))
```

5.2 RNN / GRU / LSTM

Sequence Models have been motivated by the analysis of sequential data such text sentences, time-series and other discrete sequences data. These models are especially designed to handle sequential information while Convolutional Neural Network are more adapted for process spatial information.

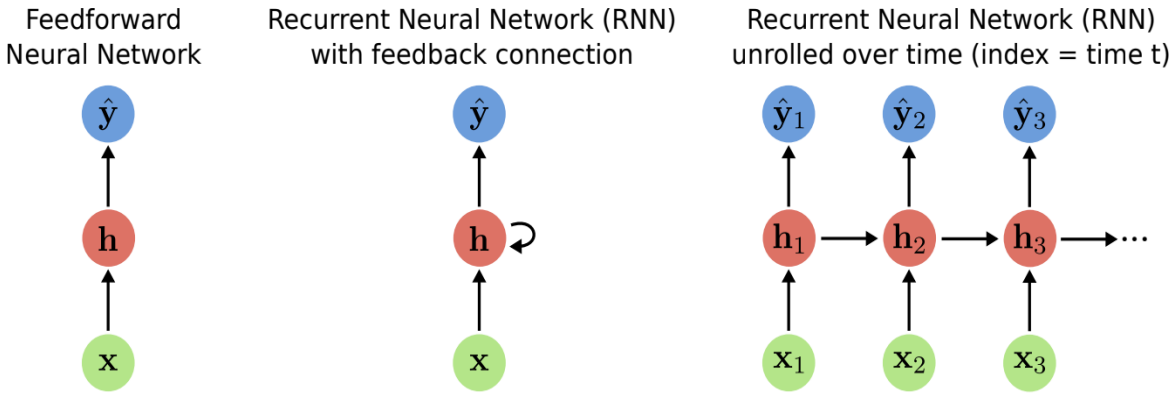
Why sequence models. Examples of sequence data :

- Speech recognition
- Music generation
- Sentiment classification
- DNA sequence analysis
- Machine translation
- Video activity recognition
- Named entity recognition
- Image captioning

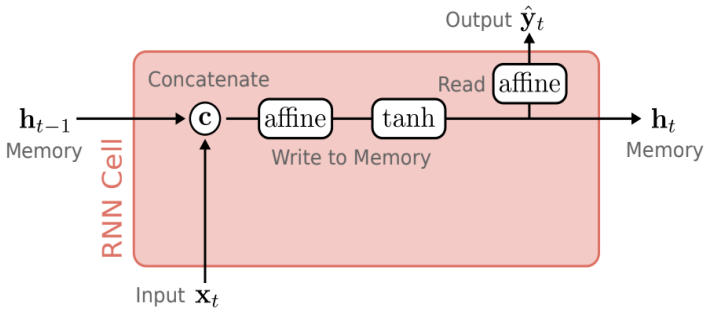
5.2.1 RNN

A recurrent neural network (RNN) is a special type of an artificial neural network adapted to work for time series data or data that involves sequences.

Ordinary feed forward neural networks are only meant for data points, which are independent of each other. However, if we have data in a sequence such that one data point depends upon the previous data point, we need to modify the neural network to incorporate the dependencies between these data points. RNNs have the concept of 'memory' that helps them store the states or information of previous inputs to generate the next output of the sequence.



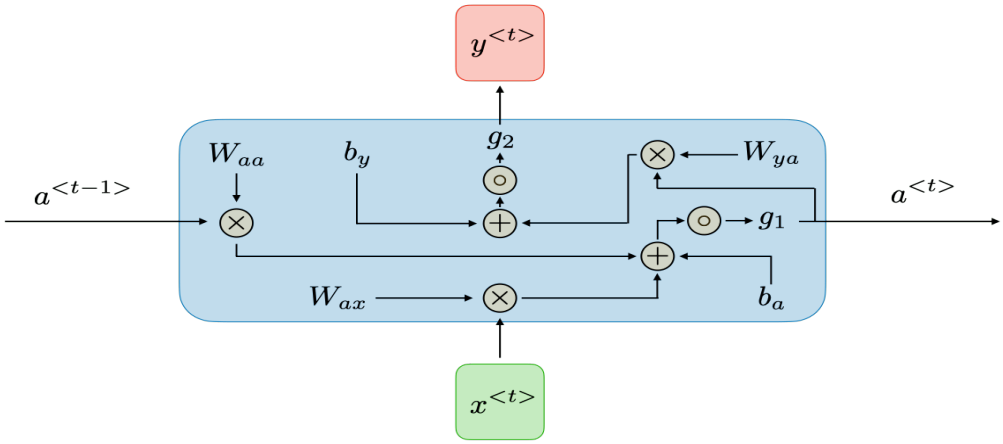
— Core idea : update hidden state h based on input and previous hidden state using same update rule (same/shared parameters) at each time step.
— Allows for processing sequences of variable length, not only fixed-sized vectors.
— Infinite memory : h is function of all previous inputs (long-term dependencies).



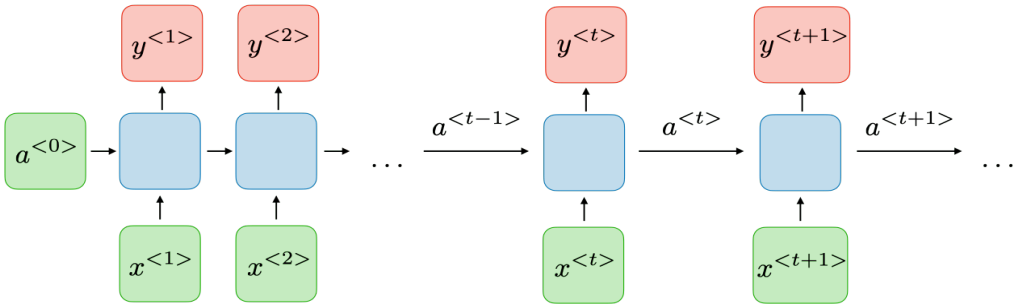
$$h_t = tanh(A_h h_{t-1} + A_x x_t + b)$$

$$\hat{y}_t = A_y h_t$$

— Hidden state h_t = linear combination of input x_t and previous hidden state h_{t-1}.
— Output $\hat{y}_t$ = linear prediction based on current hidden state h_t.
— tanh(·) is commonly used as activation function (data is in the range [-1, 1]).
— Parameters A_h, A_x, A_y, b are constant over time (sequences may vary in length).



Architecture of a traditional RNN



For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows :

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}a^{<t>} + b_a)$$

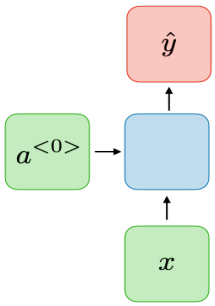
$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions.

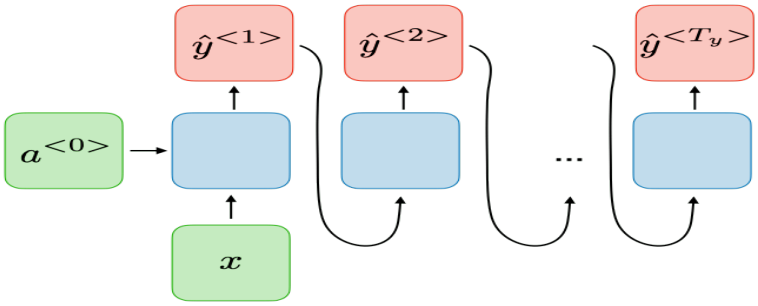
Different types of RNNs

RNN models are mostly used in the fields of natural language processing and speech recognition.

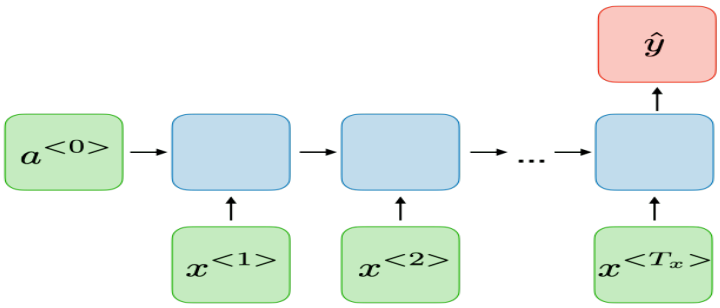
One-to-one $T_x = T_y = 1$ **:**



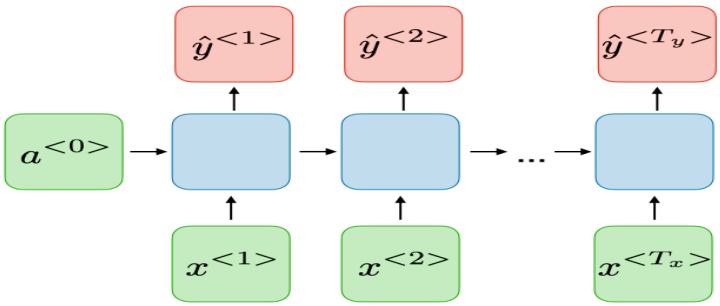
One-to-many $T_x = 1, T_y > 1$ **:**



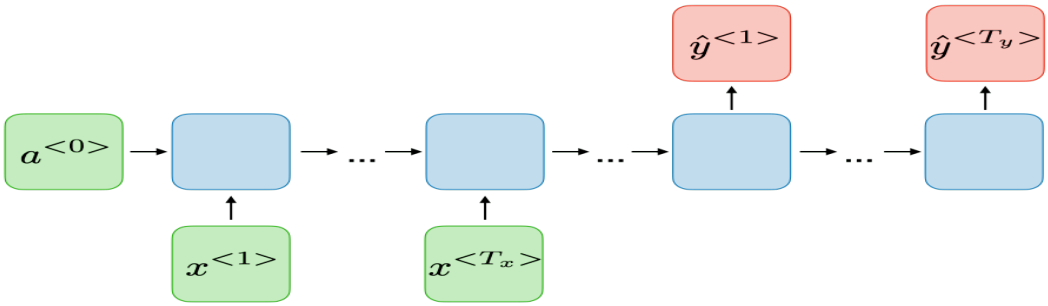
Many-to-one $T_x > 1, T_y = 1$ **:**



Many-to-many $T_x > 1, T_y > 1, T_x = T_y$ **:**



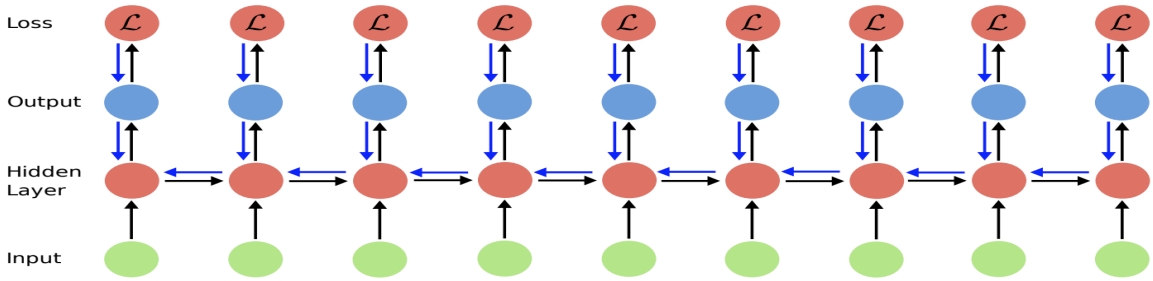
Many-to-many $T_x > 1, T_y > 1, T_x \neq T_y$ **:**



Backpropagation through time

Backpropagation is done at each point in time. At timestep T, the derivative of the loss $\mathcal{L}$ with respect to weight matrix W is expressed as follows :

$$\frac{\delta L^t}{\delta W} = \Sigma_{t=1}^{T} \frac{\delta L^t}{\delta W} \mid_t$$

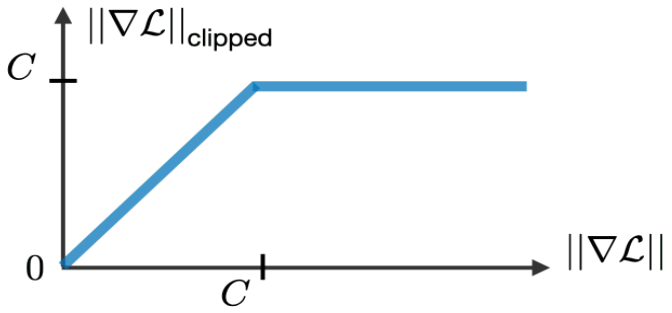


— To train RNNs, we backpropagate gradients through time.
— As all hidden RNN cells share their parameters, gradients get accumulated.
— However, very quickly intractable (memory) for larger sequences.

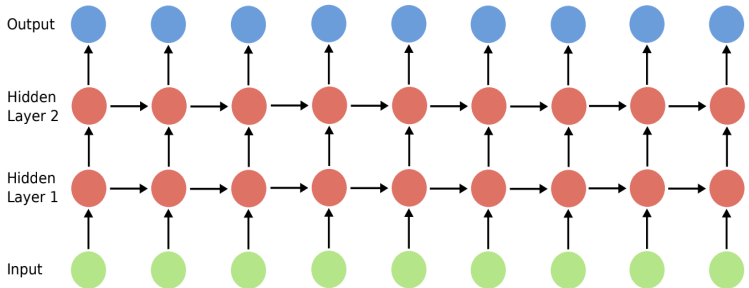
RNN Problems

Vanishing/exploding gradient : The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

Gradient clipping : it is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled in practice.



Multi-Layer RNNs



$$h_t^1 = tanh(A_h^1 h_{t-1}^1 + A_x^1 x_t + b^1)$$
$$h_t^2 = tanh(A_h^2 h_{t-1}^2 + A_x^2 h_t^1 + b^2)$$
$$\hat{y}_t = A_y h_t^2$$

— Deeper multi-layer RNNs can be constructed by stacking RNN layers.
— An alternative is to make each individual computation (=RNN cell) deeper.
— Today, often combined with residual connections in vertical direction.

GRU

Gate Recurrent Unit is one of the ideas that has enabled RNN to become much better at capturing very long range dependencies and has made RNN much more effective, it deals with the vanishing gradient problem encountered by traditional RNNs.

Gates

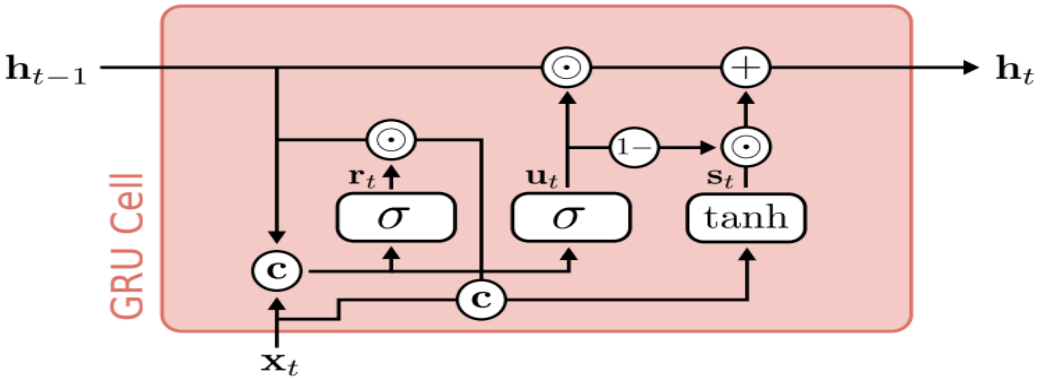
In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted Γ "Gamma" and are equal to :

$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

where W, U, b are coefficients specific to the gate and σ is the sigmoid function.,The main ones are summed up as below :

— Update gate Γ_u , How much past should matter now ? , GRU , LSTM.
— Relevance gate Γ_r , Drop previous information ? , GRU , LSTM.
— Forget gate Γ_f , Erase a cell or not ? , LSTM.
— Output gate Γ_o , How much to reveal of a cell ? , LSTM.

5.2.2 GRU Unit



$$r_t = \Gamma_r = \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r)$$

$$u_t = \Gamma_u = \sigma(W_{uh}h_{t-1} + W_{ux}x_t + bu)$$

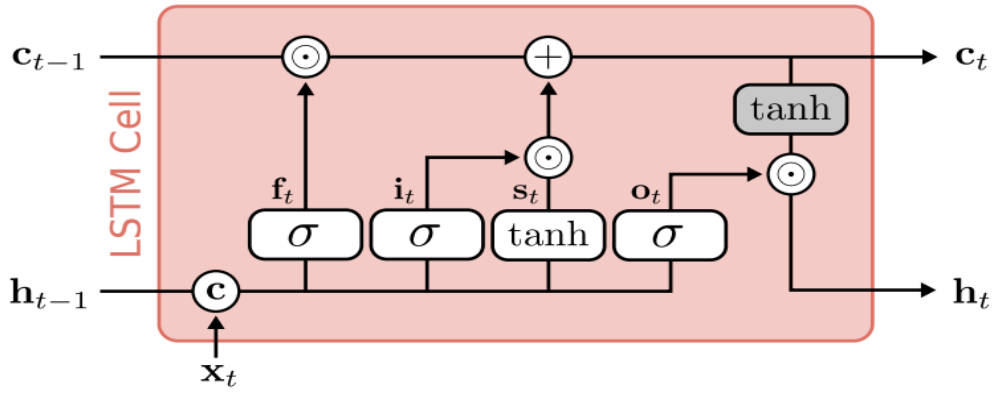
$$s_t = tanh(W_{sh}(r_t \odot h_{t-1}) + W_{sx}x_t + b_s)$$

$$h_t = u_t \odot h_{t-1} + (1 - u_t) \odot s_t$$

— Reset gate controls which parts of the state are used to compute next target state.
— Update gate controls how much information to pass from previous time step.

5.2.3 LSTM

Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU.



$$f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f)$$

$$i_t = \sigma(W_{ih}h_{t-1} + W_{ix}xt + b_i)$$

$$o_t = \sigma(Wohh_{t-1} + W_{ox}x_t + b_o)$$

$$s_t = tanh(W_{sh}h_{t-1} + W_{sx}x_t + b_s)$$

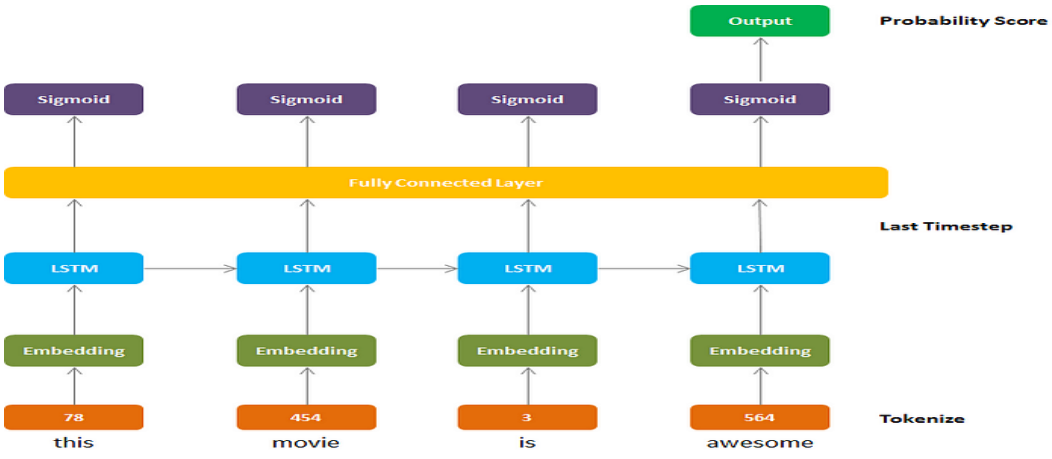
$$c_t = f_t \odot c_{t-1} + i_t \odot s_t$$

$$h_t = o_t \odot tanh(c_t)$$

Passes along an additional cell state c in addition to the hidden state h. Has 3 gates :

— Forget gate determines information to erase from cell state.
— Input gate determines which values of cell state to update.
— Output gate determines which elements of cell state to reveal at time t.

5.2.4 LSTM/GRU/RNN based on pytorch



https ://www.kaggle.com/code/bhaveshkumar2806/sentiment-analysis-pytorch-lstm/notebook

RNN/LSTM Pytorch

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
```

```
import seaborn as sns

import re
import string
from collections import Counter
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))

from sklearn.model_selection import train_test_split

import torch
from torch.utils.data import DataLoader, TensorDataset
import torch.nn as nn

# load data from kaggle repo
df = pd.read_csv('../input/imdb-dataset-of-50k-movie-reviews/IMDB Dataset.csv')
df.head()

# Data Processing - convert to lower case, Remove punctuation etc

def data_preprocessing(text):
    text = text.lower()
    text = re.sub('<.*?>', '', text) # Remove HTML from text
    text = ''.join([c for c in text if c not in string.punctuation])# Remove
   punctuation
    text = [word for word in text.split() if word not in stop_words]
    text = ' '.join(text)
    return text

df['cleaned_reviews'] = df['review'].apply(data_preprocessing)
df.head()

# Tokenize - Create Vocab to Int mapping dictionary

corpus = [word for text in df['cleaned_reviews'] for word in text.split()]
count_words = Counter(corpus)
sorted_words = count_words.most_common()

keys = []
values = []
for key, value in sorted_words[:20]:
    keys.append(key)
    values.append(value)

plt.figure(figsize=(12, 5))
plt.bar(keys, values)
```

```
plt.title('Top 20 most common words', size=15)
plt.show()

vocab_to_int = {w:i+1 for i, (w,c) in enumerate(sorted_words)}

reviews_int = []
for text in df['cleaned_reviews']:
    r = [vocab_to_int[word] for word in text.split()]
    reviews_int.append(r)

print(reviews_int[:1])
df['Review int'] = reviews_int

# Tokenize - Encode the labels

df['sentiment'] = df['sentiment'].apply(lambda x: 1 if x == 'positive' else 0)
df.head()

# Padding / Truncating the remaining data
# This sequence length is same as number of time steps for LSTM layer.
def Padding(review_int, seq_len):
    '''
    Return features of review_ints, where each review is padded with 0's or
    truncated to the input seq_length.
    '''
    features = np.zeros((len(reviews_int), seq_len), dtype = int)
    for i, review in enumerate(review_int):
        if len(review) <= seq_len:
            zeros = list(np.zeros(seq_len - len(review)))
            new = zeros + review
        else:
            new = review[: seq_len]
        features[i, :] = np.array(new)

    return features

features = Padding(reviews_int, 200)
print(features[0, :])


X_train, X_remain, y_train, y_remain = train_test_split(features, df['sentiment'].
    to_numpy(), test_size=0.2, random_state=1)
X_valid, X_test, y_valid, y_test = train_test_split(X_remain, y_remain, test_size=0
    .5, random_state=1)

# create tensor dataset
```

```
train_data = TensorDataset(torch.from_numpy(X_train), torch.from_numpy(y_train))
test_data = TensorDataset(torch.from_numpy(X_test), torch.from_numpy(y_test))
valid_data = TensorDataset(torch.from_numpy(X_valid), torch.from_numpy(y_valid))

# dataloaders
batch_size = 50

train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)


# obtain one batch of training data
dataiter = iter(train_loader)
sample_x, sample_y = dataiter.next()

print('Sample input size: ', sample_x.size()) # batch_size, seq_length
print('Sample input: \n', sample_x)
print('Sample input: \n', sample_y)

class sentimentLSTM(nn.Module):
    """
    The RNN model that will be used to perform Sentiment analysis.
    """

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers
    , drop_prob=0.5):
        """
        Initialize the model by setting up the layers.
        """
        super().__init__()

        self.output_size = output_size
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        # Embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=drop_prob,
    batch_first=True)

        # Dropout layer
        self.dropout = nn.Dropout(0.3)

        # Linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
```

```
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hidden state.
        """
        batch_size = x.size(0)

        #embedding and lstm_out
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        #stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        # Dropout and fully connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)

        #sigmoid function
        sig_out = self.sigmoid(out)

        # reshape to be batch size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        return sig_out, hidden

    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        h0 = torch.zeros((self.n_layers,batch_size,self.hidden_dim)).to(device)
        c0 = torch.zeros((self.n_layers,batch_size,self.hidden_dim)).to(device)
        hidden = (h0,c0)
        return hidden

 is_cuda = torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'll use this device
    variable later in our code.
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
```

```
    print("GPU not available, CPU used")

# Instantiate the model w/ hyperparams
vocab_size = len(vocab_to_int) + 1
output_size = 1
embedding_dim = 64
hidden_dim = 256
n_layers = 2

model = sentimentLSTM(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
model = model.to(device)

print(model)

lr=0.001

criterion = nn.BCELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# function to predict accuracy
def acc(pred,label):
    pred = torch.round(pred.squeeze())
    return torch.sum(pred == label.squeeze()).item()

clip = 5
epochs = 2
valid_loss_min = np.Inf
# train for some number of epochs
epoch_tr_loss,epoch_vl_loss = [],[]
epoch_tr_acc,epoch_vl_acc = [],[]

for epoch in range(epochs):
    train_losses = []
    train_acc = 0.0
    model.train()
    # initialize hidden state
    h = model.init_hidden(batch_size)
    for inputs, labels in train_loader:

        inputs, labels = inputs.to(device), labels.to(device)
        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        model.zero_grad()
```

```
        output,h = model(inputs,h)

        # calculate the loss and perform backprop
        loss = criterion(output.squeeze(), labels.float())
        loss.backward()
        train_losses.append(loss.item())
        # calculating accuracy
        accuracy = acc(output,labels)
        train_acc += accuracy
        #`clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()



    val_h = model.init_hidden(batch_size)
    val_losses = []
    val_acc = 0.0
    model.eval()
    for inputs, labels in valid_loader:
        val_h = tuple([each.data for each in val_h])

        inputs, labels = inputs.to(device), labels.to(device)

        output, val_h = model(inputs, val_h)
        val_loss = criterion(output.squeeze(), labels.float())

        val_losses.append(val_loss.item())

        accuracy = acc(output,labels)
        val_acc += accuracy

    epoch_train_loss = np.mean(train_losses)
    epoch_val_loss = np.mean(val_losses)
    epoch_train_acc = train_acc/len(train_loader.dataset)
    epoch_val_acc = val_acc/len(valid_loader.dataset)
    epoch_tr_loss.append(epoch_train_loss)
    epoch_vl_loss.append(epoch_val_loss)
    epoch_tr_acc.append(epoch_train_acc)
    epoch_vl_acc.append(epoch_val_acc)
    print(f'Epoch {epoch+1}')
    print(f'train_loss : {epoch_train_loss} val_loss : {epoch_val_loss}')
    print(f'train_accuracy : {epoch_train_acc*100} val_accuracy : {epoch_val_acc*100}')
    if epoch_val_loss <= valid_loss_min:
```

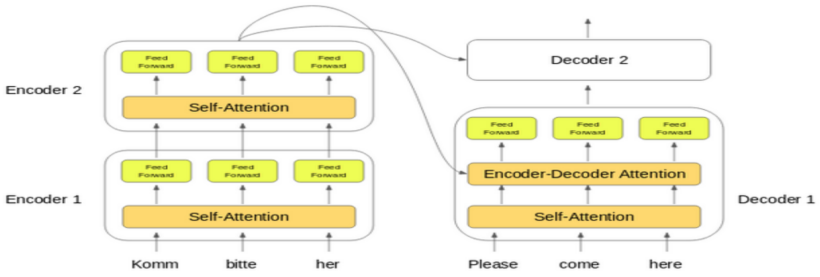
```
        torch.save(model.state_dict(), '../working/state_dict.pt')
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.
    format(valid_loss_min,epoch_val_loss))
        valid_loss_min = epoch_val_loss
    print(25*'==')

    # Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0

# init hidden state
test_h = model.init_hidden(batch_size)

model.eval()
# iterate over test data
for inputs, labels in test_loader:

    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    test_h = tuple([each.data for each in test_h])

    inputs, labels = inputs.to(device), labels.to(device)

    output, test_h = model(inputs, test_h)

    # calculate loss
    test_loss = criterion(output.squeeze(), labels.float())
    test_losses.append(test_loss.item())

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())  # rounds to the nearest integer

    # compare predictions to true label
    correct_tensor = pred.eq(labels.float().view_as(pred))
    correct = np.squeeze(correct_tensor.cpu().numpy())
    num_correct += np.sum(correct)


# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}".format(test_acc))
```

5.3 Attention / Transformers

NLP's Transformer is a new architecture that aims to solve tasks sequence-to-sequence while easily handling long-distance dependencies. Computing the input and output representations without using sequence-aligned RNNs or convolutions and it relies entirely on self-attention.
The Transformer architecture follows an encoder-decoder structure, See Paper.

Encoder : The encoder is responsible for stepping through the input time steps and encoding the entire sequence into a fixed-length vector called a context vector.
Decoder : The decoder is responsible for stepping through the output time steps while reading from the context vector.

Let's see how this setup of the encoder and the decoder stack works :

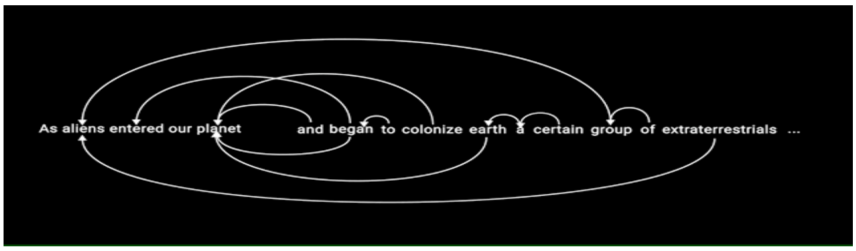


5.3.1 Attention ALL you need

Attention

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of "attention" in the literature, but the one we will use here is the following : the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys.

— The Attention mechanism enables the transformers to have extremely long term memory.
— A transformer model can attend or focus on all previous tokens that have been generated.



Attention mechanism focusing on different tokens while generating words 1 by 1

The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to "attend" more than others. In particular, an attention mechanism has usually four parts we need to specify :

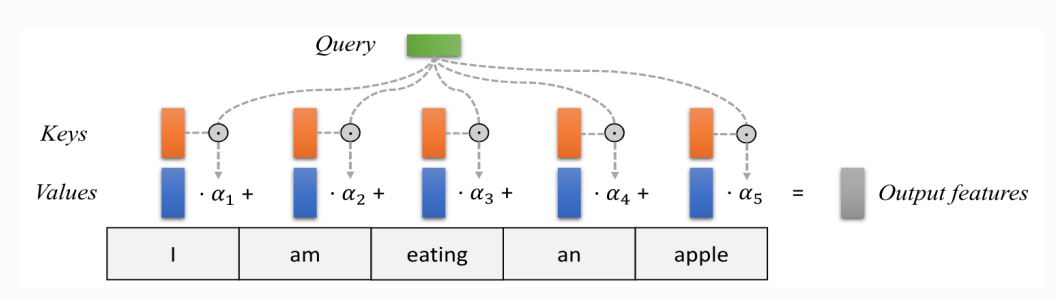
— Query : The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.

— Keys : For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is "offering", or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
— Values : For each input element, we also have a value vector. This feature vector is the one we want to average over.
— Score function : To rate which elements we want to pay attention to, we need to specify a score function . The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write :

$$\alpha_i = \left(\frac{e^{f_{attn}(Key_i, Query)}}{\sum_j e^{f_{attn}(Key_i, Query)}} \right), out = \sum \alpha_i value_i \tag{5.3}$$

Visually, we can show the attention over a sequence of words as follows :



5.3.2 Transformers Architecture

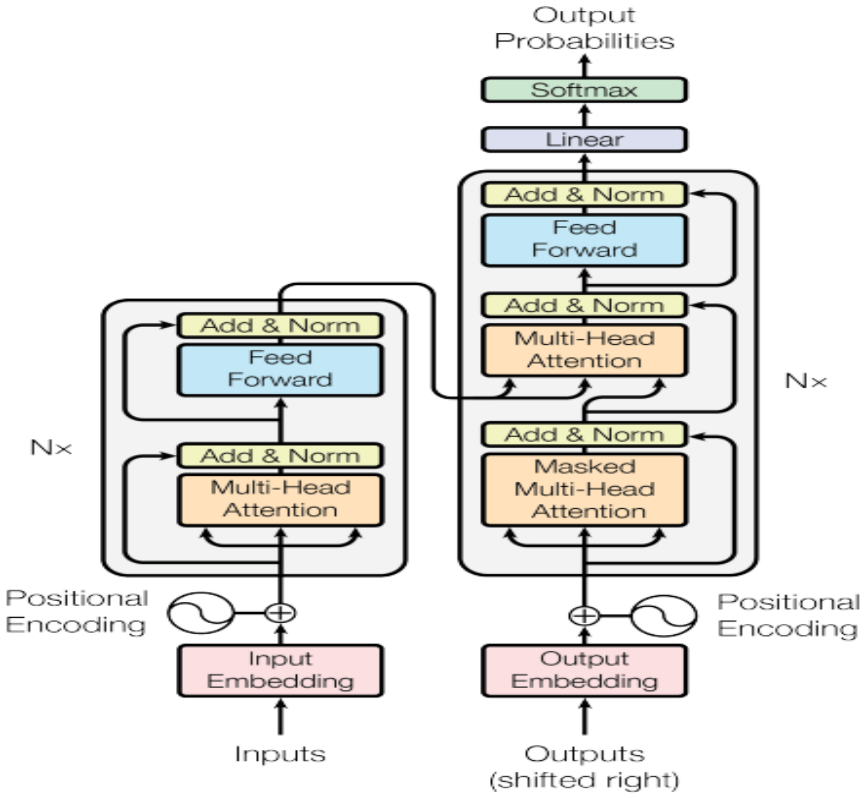


Figure 1: The Transformer - model architecture.

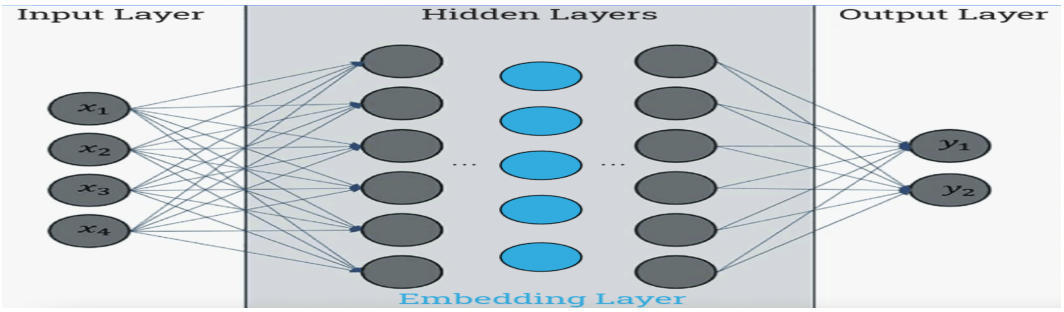
Embedding & Positional Encoding

Embedding :
In NLP, word embedding is a projection of a word, consisting of characters into meaningful vectors of real numbers. Conceptually it involves a mathematical embedding from a dimension N (all words in a given corpus)-often a simple one-hot encoding is used- to a continuous vector space with a much lower dimensionality, typically 128 or 256 dimensions are used. Word embedding is a crucial preprocessing step for training a neural network.

- one hot encoding
- embedding layer
- word2vec
- Glove
- FastText
- ELMo

Embedding Layer :
An embedding layer is a type of hidden layer in a neural network. In one sentence, this layer maps input information from a high-dimensional to a lower-dimensional space, allowing the network to learn more about the relationship between inputs and to process the data more efficiently.

For example, in natural language processing (NLP), we often represent words and phrases as one-hot vectors, where each dimension corresponds to a different word in the vocabulary. These vectors are high-dimensional and sparse, which makes them difficult to work with.



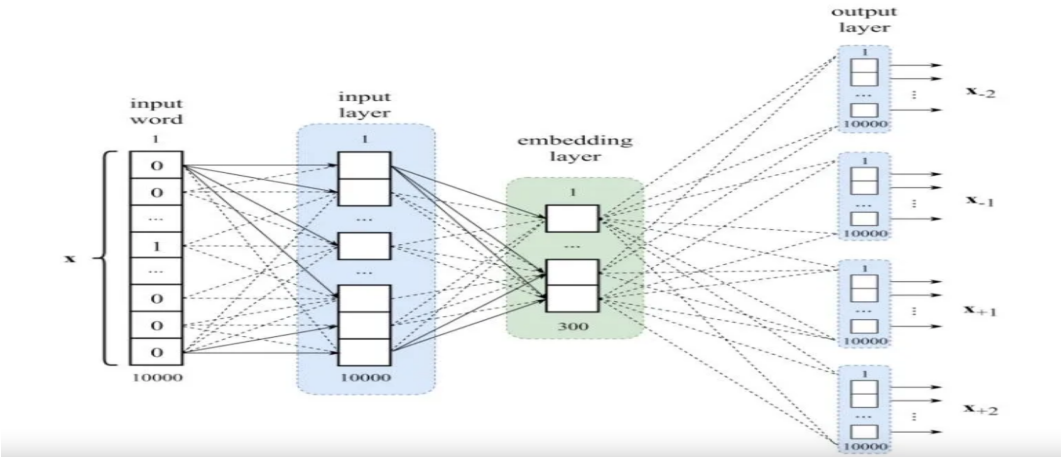
The type of embedding layer depends on the neural network and the embedding process. There are several types of embedding that exist :

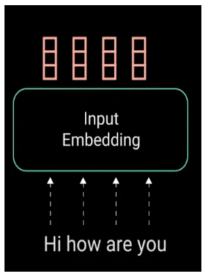
— text embedding
— Image embedding
— Graph embedding and others

Text embedding :

A standard approach is, to feed the one-hot encoded tokens (mostly words, or sentence) into a embedding layer. During training the model tries to find a suitable embedding (lower dimensionality as the input layer). The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. In some cases it could be useful to use a pretrained embedding, which was trained on a hugh corpus.

— Input : one-hot encoding of the word in a vocabulary
— Output : one vector of N dimensions (given by the user, probably tuned with hyperparameter tuning)





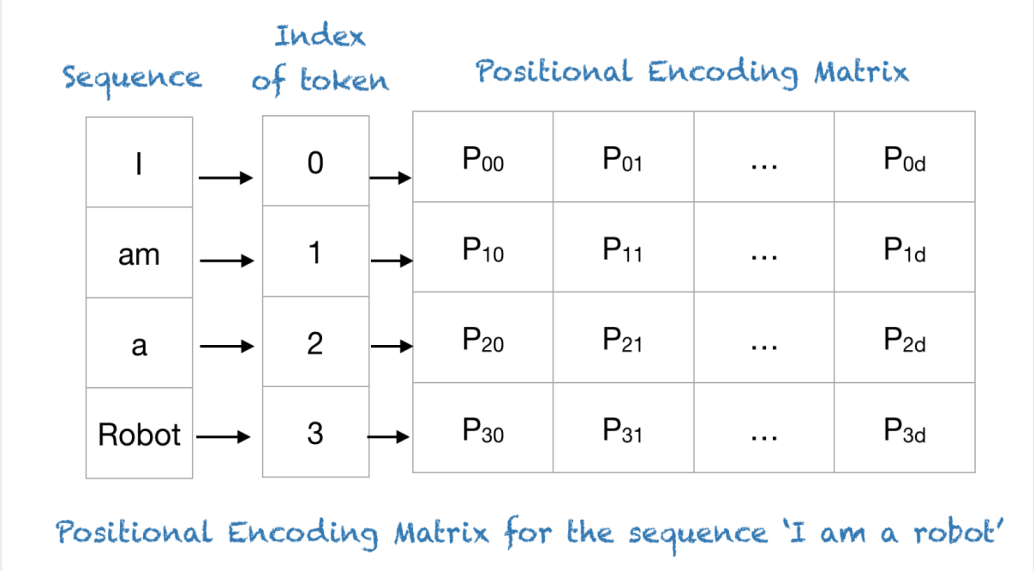
converting Words to Input Embeddings

Positional Encoding

Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models. For long

sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently.

Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information. An example of the matrix that encodes only the positional information is shown in the figure below.



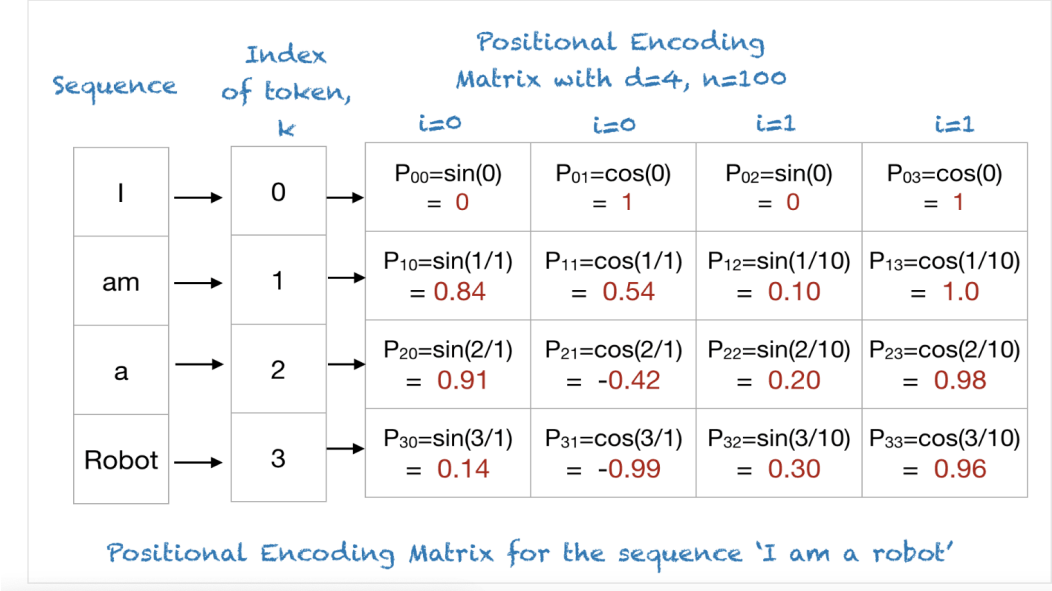
Positional Encoding : is to inject positional information into the embeddings (information about the positions).

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

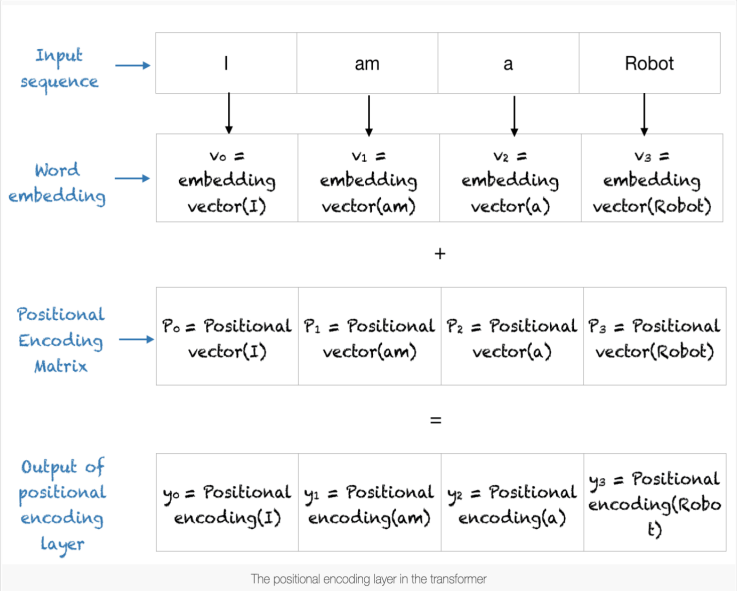
$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Here :

- K : Position of an object in the input sequence $0 =< k < L/2$
- d : Dimension of the output embedding space
- P(K,j) : Position function for mapping a position in the input k sequence to(K,j) index of the positional matrix
- n : User-defined scalar, set to 10,000 by the authors of Attention Is All You Need.
- i : Used for mapping to column indices $0 =< i < d/2$, with a single value of maps to both sine and cosine functions.

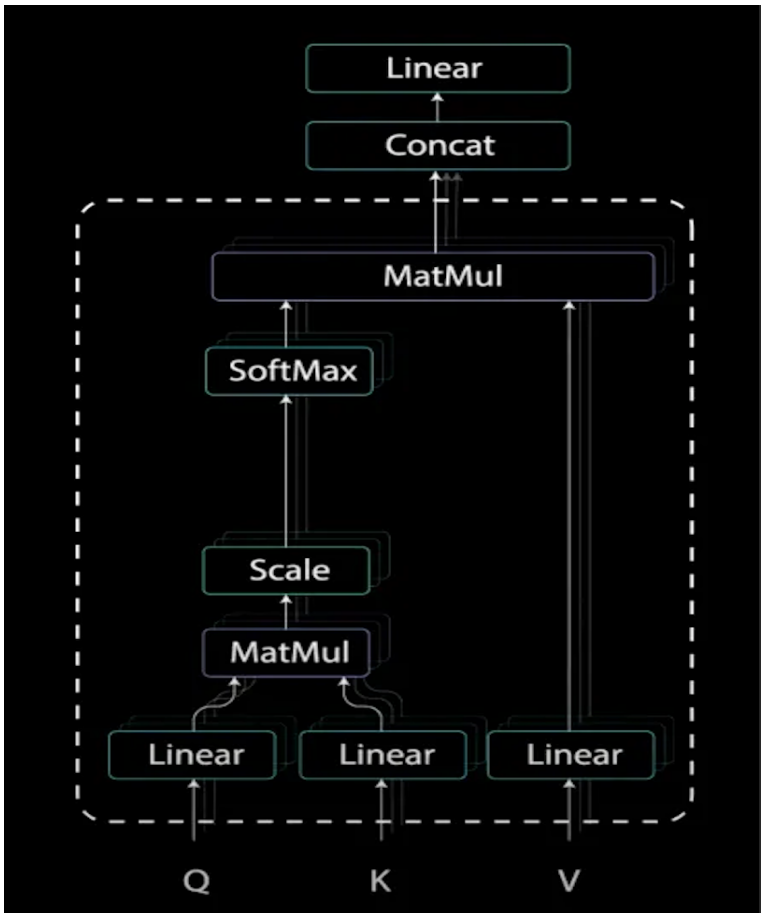


What Is the Final Output of the Positional Encoding Layer ?



5.3.3 Multi-Head Attention

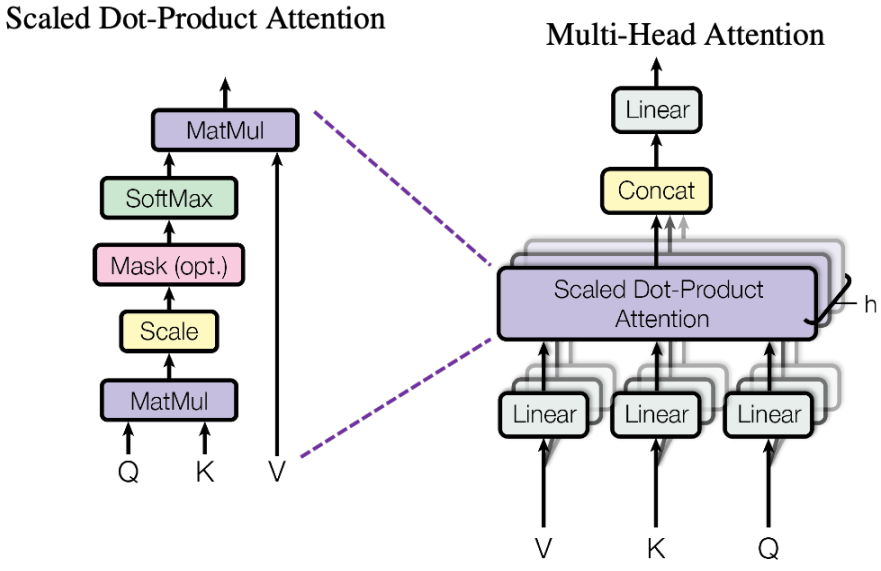
— Attention mechanism calculates attentions, or relevance between "queries" and "keys.
— The encoder applies a specific attention mechanism called self-attention (Understanding the context).
— Self-attention allows the models to associate each word in the input, to other words. (in the context)



Multi-Head Attention (Query, Key, and Value Vectors)

The idea behind (Q, K, V) is similar to the search engine that will map the query against a set of keys (video title, description etc.) associated with candidate videos in the database, then present the best matched videos (values).

— Query : what i am looking for
— key : what i can offer
— Value : what i actually offer

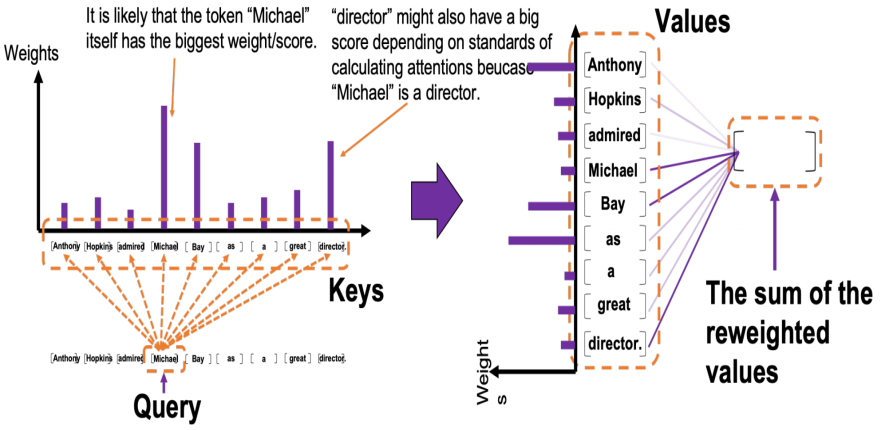


Multi-Head Attention (Where are Q and K from)

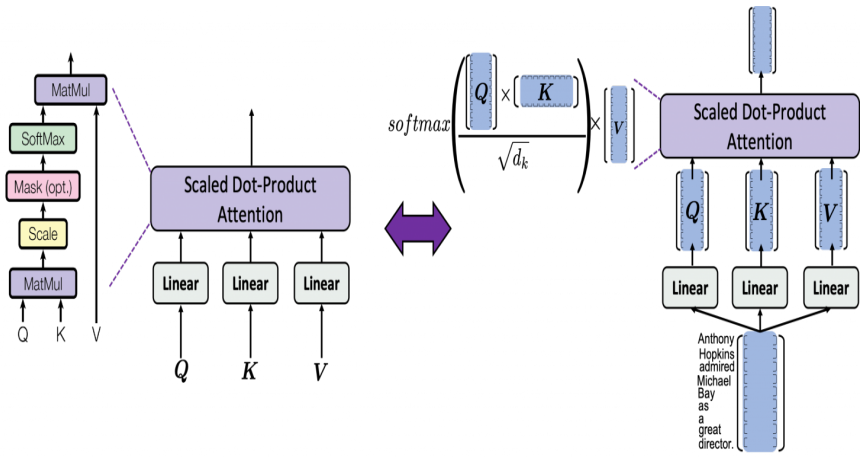
The transformer encoder training builds the weight parameter matricesW_Q and W_k.

The calculation goes like below where x is a sequence of position-encoded word embedding vectors that represents an input sentence.

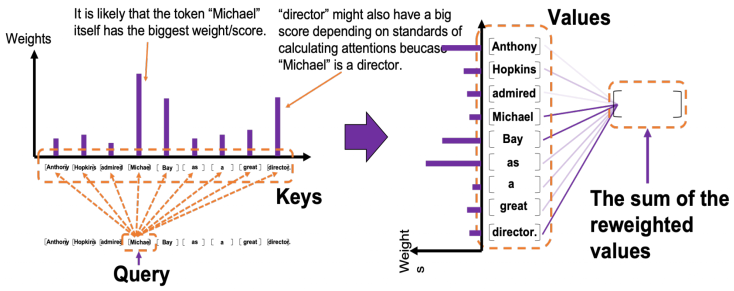
1. $Q = X \cdot W_Q^T$
2. $K = X \cdot W_K^T$
3. For each (q, k) pair, their relation strength is calculated using dot product : $q\_to\_k\_similarity\_sco$ $matmul(Q, K^T)$
4. Weight matrices W_Q and W_K are trained via the back propagations during the Transformer training.



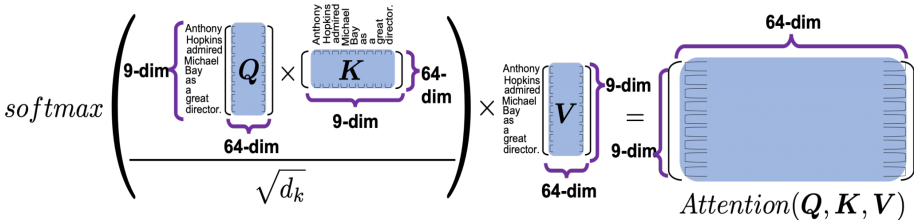
Multi-Head Attention (V is created using Q and K)



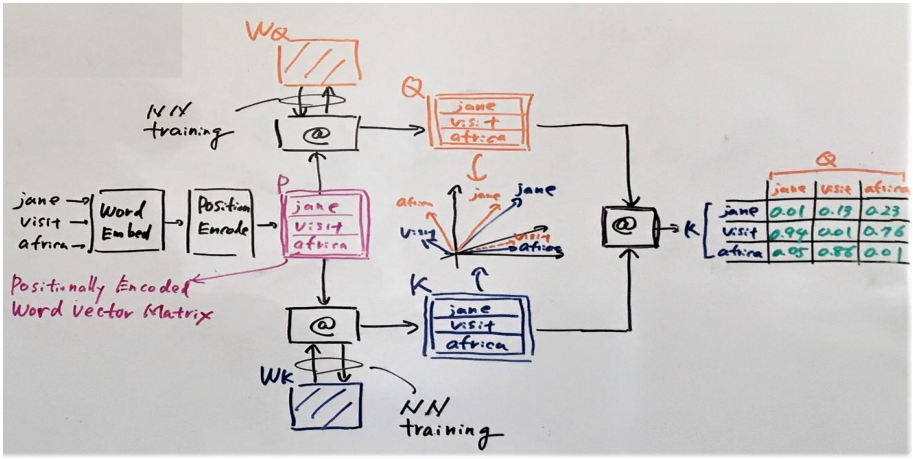
Multi-Head Attention (Scaled product)



$\sqrt{d_k}$the square root of the dimension of query and key.

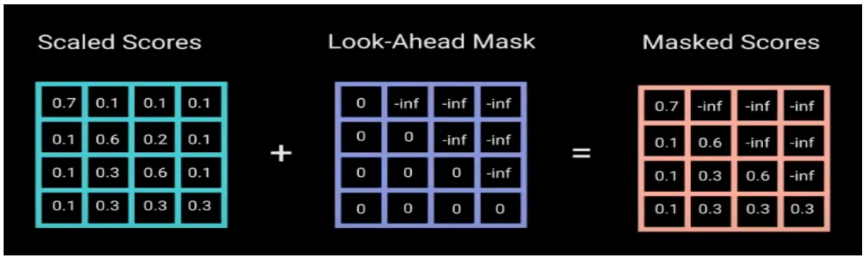


Multi-Head Attention (Full process)



Mask Mutli head attention

Mask is used because while creating attention of target words, we donot need a word to look in to the future words to check the dependency. ie, we already learned that why we create attention because we need to know contribution of each word with the other word. Since we are creating attention for words in target sequnce, we donot need a particular word to see the future words. For eg : in word "I am a strudent", we donot need the word "a" to look word "student".

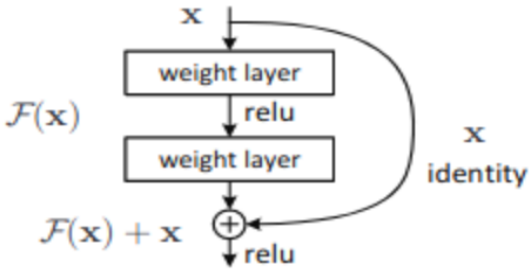


Adding a look-ahead mask to the scaled scores

The reason for the mask is because once you take the softmax of the masked scores, the negative infinities get zeroed out, leaving zero attention scores for future tokens.

Add & Norm

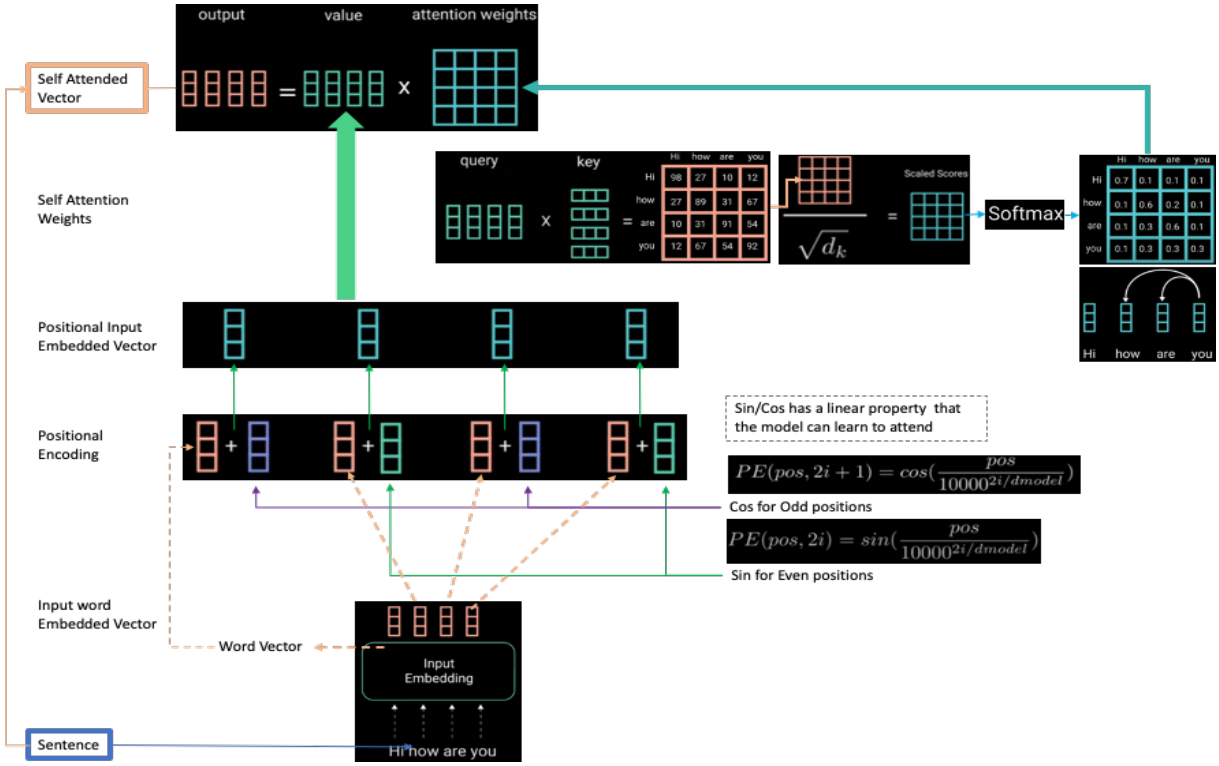
Add & Norm are in fact two separate steps. The add step is a residual connection



It means that we take sum together the output of a layer with the input F(x)+x.
The idea was introduced with the ResNet model. It is one of the solutions for vanishing gradient problem.

The norm step is about layer normalization , it is another way of normalization and it is one of the many computational tricks to make life easier for training the model, hence improve the performance and training time.

Global Architecture



5.3.4 Transformers Implementation

- Attention (Colab)
- Transformers (Kaggle)
- Transformers/ASAG (Kaggle)

References

https://www.geeksforgeeks.org/natural-language-processing-nlp-pipeline/
https://www.geeksforgeeks.org/natural-language-processing-nlp-tutorial/
https://www.geeksforgeeks.org/rule-based-approach-in-nlp/
https://www.kaggle.com/code/alincijov/nlp-starter-continuous-bag-of-words-cbow
https://www.kaggle.com/code/alincijov/word2vec-skip-gram-numpy
https://machinelearningmastery.com/method-of-lagrange-multipliers-the-theory-behind-support-vector-n