**2**

**An Overview of Natural Language Processing and Neural Networks**

**Learning Objectives**

After completing this chapter, the readers are expected to

* Learn the basics of Natural Language Processing (NLP) necessary for understanding the subsequent chapters.
* Gain knowledge about various semantic and syntactic paradigms in NLP
* Learn the basics of Neural Networks required to understand the subsequent chapters.
* Become familiar with the evaluation metrics employed in neural network modelling.

The natural evolution of languages has enabled humans to communicate and share ideas across geopolitical boundaries. However, the concept of a machine processing and understanding these human languages is a complex challenge. Since machines inherently understand numbers and numerical operations, it is necessary to convert natural language into a format that computers can comprehend. This is where NLP comes into play. NLP is an interdisciplinary field within computer science that encompasses techniques to make human language accessible and interpretable by machines. The foundational concepts in NLP draw from a wide range of disciplines, including theoretical computer science, linguistics, statistics, and artificial intelligence.

Statistical methods and linear machine learning models have long been utilised in NLP tasks. However, these traditional approaches depend heavily on manually crafted dictionaries and features, limiting their capacity to capture complex patterns and semantics in language. As the volume of textual data has increased and computational resources have become more accessible, neural architectures have gained prominence in modern NLP techniques. These neural models excel at capturing *latent knowledge* from data, processing inputs of varying lengths, leveraging information from long sequences of text, and autonomously learning features, reducing the need for extensive manual efforts.

This chapter is divided into two parts. In **Part I**, readers are first introduced to the fields of linguistics and NLP. Section 2.1 discusses the goals of computational linguistics and NLP. Section 2.2 describes various tasks in NLP and introduces the NLP pipeline. Section 2.3 explores the linguistic components of language, such as morphology, lexicon, and text normalisation techniques like stemming and lemmatisation. In Section 2.4, we provide an overview of different tokenisation and semantic analysis techniques. Section 2.5 focuses on syntax and grammar-based parsing methods, while Section 2.6 delves into semantics and semantic parsing. Finally, Section 2.7 presents the task of language modelling, emphasising conditional probability and the frequency of co-occurrence.

In **Part II**, we explore neural networks and related concepts to set the foundation for deep learning techniques discussed in later chapters. Section 2.8 introduces the perceptron and its applications in modelling a linear classifier. Section 2.9 presents multilayer perceptrons and popular non-linear activation functions. Section 2.10 covers the gradient-based training process for neural networks and error backpropagation. In Subsection 2.10.3, we discuss the various hyperparameters that influence neural network training. Section 2.11 focuses on the challenges that affect gradient descent. Finally, in Section 2.12, we examine the performance measures commonly used to evaluate deep learning tasks.

**Part I: Natural Language Processing**

Since the advent of computers, researchers have been captivated by the idea of teaching machines to interact like humans. As early as 1963, Joseph Weizenbaum developed ELIZA (Weizenbaum 1983), a rule-based chatbot designed to converse with humans. Fast forward to 2014, Eugene Goostman, a chatbot, passed the *Turing Test*, with human judges unable to discern that Eugene was, in fact, a bot. In 2017, Google revolutionised the field by introducing a new machine translation architecture, now famously known as *Transformers* (Vaswani et al. 2017). More recently, the success and widespread adoption of language models like ChatGPT, and its successors have generated immense interest in language models, accelerating research in both NLP and deep learning. This book is motivated by the need to enhance our understanding of language models and to demystify the concepts from beginner to advanced levels.

Computation aside, a very nuanced problem with language is its ambiguity, contextualisation, and its dynamic nature relative to the zeitgeist. Consider the sentence ‘*I saw her duck*’. One interpretation could be that I saw a duck (noun) that belonged to her. Another interpretation of this same sentence could be that I saw her perform the act of ducking (verb) to avoid an obstacle. Depending on the context, sentences with similar structures and subjects can still convey very different meanings. Take the following sentences: ‘*I ate rice with a spoon*’, ‘*I ate rice with curd*’and ‘*I ate rice with Rahul*’. All of these sentences convey that I am eating rice. In the first instance, I use a spoon as a utensil, while in the second sentence, I use curd as an accompaniment. The third sentence, however, suggests that I am eating rice in the presence of a person. Teaching computers to understand how to interpret these sentences differently and resolve ambiguities has been a significant motivation behind the development of NLP and understanding techniques.

**2.1 Computational Linguistics and Natural Language Processing**

Like any other medium of information, human languages have undergone various stages of development—they originated at some point in time, propagated far and wide, and have borne witness to the evolution of human society. While we do not know exactly when and how the earliest humans spoke, numerous theories have been proposed about the origin of language. Alister Hardy and Elaine Morgan proposed the *aquatic ape theory* in 1997, highlighting that there are certain traits we do not share with our primate relatives but that we do share communication traits with aquatic animals. For example, whales and dolphins are known to communicate with members of their species using sounds.

Linguistics is the discipline that engages in the scientific study of languages. It is an interdisciplinary system (see Figure 2.1) where linguistics and its branches, such as sociolinguistics, psycholinguistics, and neurolinguistics, seek to answer significant philosophical questions, such as: What rules do languages follow? How do languages evolve? How do we learn and process meanings in our minds? How are different modalities of languages related to each other? Linguists like Noam Chomsky and Steven Pinker hypothesise that language is something innate. From a very early age, we begin to mimic natural sounds and lip movements. We grow up associating these sounds with certain objects, qualities, and attributes of the environment around us. By studying these theories, we can infer what the earliest languages might have been like, but our understanding only extends so far back into the history of human languages (Jurafsky and Martin 2009; Pinker 2010; Vaneechoutte 2014).

**[Insert Figure 2.1]**

Figure 2.1: Language-related Disciplines (Tsujii 2021) – Linguistics, Cognitive Science, Psychology, Natural Language Processing (NLP), Artificial Intelligence (AI), and Computational Linguistics. All of these disciplines study language from different perspectives.

Many other fields, such as neuroscience and psychology, also show great interest in languages. Computational linguistics is a sub-field of both linguistics and computer science that focuses on the interactions between human language and computers, serving as a bridge between the broader field of linguistics and engineering processes. While computational linguistics is more concerned with understanding language structure and developing computational models, NLP emphasises the design and analysis of algorithms and systems for *tasks* that rely on processing human language input.

**2.2 Overview of the Natural Language Processing Pipeline**

The standard *pipeline* used in NLP involves several steps. The natural language input for the pipeline discussed in this book is typically modelled as a collection of machine-readable text documents, known as a *corpus;* a larger collection of these documents is referred to as *corpora.* When employing NLP systems, the standard pipeline consists of a sequence of steps, as illustrated in Figure 2.2, to address one or more NLP-based tasks.

**[Insert Figure]**

Figure 2.2: Stages of the language processing pipeline for textual data input.

**Tasks in Natural Language Processing (NLP).** Depending upon the task, the output from the NLP pipeline could be in the form of a sentence-level or word-level class label, a sequence of words, a piece of text, and even paths of a graph node-to-edge sequence (see Figure 2.3). To better illustrate the tasks, we will consider an example: ‘*I do not support WHO. They underfund Indian diseases’.* Below is a non-exhaustive list of popular tasks in NLP.

1. **Sentiment Analysis:** Detecting the type and intensity of emotional tone or opinion expressed in some text. Here, the output fr the entire sentence is a label such as positive, negative or neutral. Based on the usage of the phrase ‘do not support’, we can label the sentiment of the above example as ‘Negative’.
2. **Part-of-Speech (POS) Tagging:** Figuring out the grammatical class (noun, pronoun, adjective, adverb, etc.) of each word in a sequence. The output is a sequence of class labels tagged for each word in the sentence. The above example can be tagged as: I (*pronoun*)do (*verb*)not (*adverb*)support (*verb*)WHO (*noun*)*.* (*punctuation*).They (*pronoun*)underfund (*verb*)Indian (*adjective*)diseases (*plural noun*) *.* (*punctuation*)*.* Note that in NLP, we use a diverse set of POS tags. For instance, the Penn Treebank project uses 36 POS tags[[1]](#footnote-1).
3. **Named Entity Recognition (NER):** Identifying and classifying noun phrases into real-world entities like organisation, country, groups, nationality, etc. Here, the output is a label for one or more contiguous words. In our example, the terms ‘WHO’ and ‘Indian’ will be tagged as ‘ORG’ for organisation and ‘NORP’ for nationality, respectively.
4. **Text Entailment:** Determining whether the *premise* sentence implies, contradicts, or has nothing to do with the preceding *hypothesis* sentence. The output is a label for the sentence pair. In our example, the premise of ‘underfunding’ *supports* the hypothesis of ‘not supporting’.
5. **Semantic Role Labelling:** Identifying the role of each noun phrase with respect to the predicate of the sentence. The output is a label for a phrase.
6. **Machine Translation:** Conversion of text from one human language to another. Our example, when translated into Hindi, will be मैं WHO का समर्थन नहीं करता. वे भारतीय बीमािरयों के िलए कम फंड देते हैं।.

**[Insert Figure]**

Figure 2.3: Tasks in NLP.

**7. Summarisation:** Producing a shorter version of the larger *reference text* that still retains the information conveyed. The output is a piece of text that is way shorter in length than the reference.

**8. Question Answering:** Providing the correct and concise answer to a user query. The output is a piece of text that mimics human response. Based on our example sentence, we can ask the system, ‘Which country does WHO underfund’? and expect the answer to be ‘India’.

**9. Knowledge Graph Completion:** Filling missing information in a structured knowledge graph by using world knowledge. The output is in the form of an edge that is not in the edge list of a graph. For example, an edge to store or predict can be .

**Data Acquisition.** To enable a machine to learn from data, the primary requirement is the availability of that data, often obtained through a combination of curation strategies such as *web scraping*, *synthetic data generation*, and *manual annotation*. In most cases, the goal is to gather a large collection of unstructured, free-flowing text fragments or documents, which may or may not be annotated by a human expert. When machine-readable text is unavailable, such as when scanning text from PDFs, Optical Character Recognition (OCR) proves useful. Public datasets and text dumps are typically the first sources to explore for *open-domain* text documents.

**Data Cleaning.** Since most textual data is curated from the web, it typically requires cleaning before further processing. This is because the data might contain markup, special characters, personal information, poorly formatted tags, and other unwanted elements. By employing regular expressions, handling stray characters, and using dictionaries to correct misspelt words, we can effectively reduce *noise* and perform *deduplication*. Additionally, the data might be encoded in different Unicode formats, so appropriate logic must be applied to address such encoding issues.

**Pre-processing.** This step involves breaking the text into smaller units and then normalising it using techniques such as lowercasing, stop-word removal, stemming, and lemmatisation, which will be discussed in this chapter. In certain contexts, digits and punctuation may also be removed if they contribute little to the overall information. It is crucial to note that there is no one-size-fits-all preprocessing technique applicable to all NLP tasks.

**Feature Engineering.** Once the text has been preprocessed, we now need to represent the text in a way that a machine can understand. As machines reduce everything into numbers, we build a text representation by encoding it into a numeric vector. In NLP or deep learning, encoding can be considered as a mapping function that takes input in raw human-readable form (text, images, videos) and converts it into numerical vectors for computational methods to be applied to them. However, there can be multiple ways of performing encoding, depending on the task, the datasets, and the computational resources available at hand. This is where feature engineering helps. It helps us analyse the essential features and most informative parts of the input and only use those to encode the input so that we can encode maximum information in as little memory as possible. Encoding can be achieved by simple frequency-based heuristics such as one-hot encoding and bag-of-words representation. NLP practitioners these days use probabilistic, neural approaches to learn *word embeddings*, which are representations of words in the feature space. Parallel to encoding, decoding is a map function for converting numerical vectors into human-readable symbols (texts, pixels, etc.).

**Model Building.** Once we have encoded the textual data, it can be passed to any machine learning or deep learning model to learn from the corpus. Irrespective of whether the task is one of NLU or NLG, learning from textual data boils down to sequence modelling since text can always be represented as a sequence of words/phrases/characters. As NLP-based sequence models aim to learn the ‘hidden/latent language’ (in the vector space) from the input text, they are also called Language Models (LMs). For a long time, neural networks like Recurrent Neural Networks (RNN) (Elman 1990), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRU) (Gers et al. 2000; Tsujii 2021; Cho et al. 2014) showed incredible performance in modelling sequential data, like text. However, with the introduction of transformers in 2017 (Vaswani et al. 2017), there has been a significant leap in the length and complexity of the textual data that can be modelled, with transformers becoming the de facto standard in today’s NLP. In the coming chapters, we will introduce different language models along with the fundamental concepts on which LMs are built.

**Evaluation.** Once the model is designed, we must assess how ‘good’ a language model is. While classification tasks can be evaluated using existing accuracy and F1-score (macro/micro), newer metrics need to be devised for tasks that involve generating text. For machine translation and summarisation tasks, we typically use Bilingual Evaluation Understudy (BLEU) and Recall-Oriented Understudy for Gisting Evaluation (ROUGE) scores, which capture the lexical and syntactic overlap between the expected and predicted text. Meanwhile, newer semantic measures like BERTScore have also been designed. When comparing two LMs themselves, we can employ entropy-based measures like perplexity. We introduce these LM evaluation metrics in Chapter 4.

|  |  |
| --- | --- |
| **Information** | **Meaning** |
| Phonetic | How a word is expressed vocally with a certain sound. |
| Structural | How a word is composed of different linguistic components. |
| Syntactic | How a word fits into the overall structure of a sentence |
| Semantic | What is the meaning of a word in some particular context? |
| Pragmatic | How a word is used in a discourse or conversation. |

Table 2.1: Different kinds of information that can be derived from a word.

**Deployment.** Transforming a trained model into a functional component of a software system in any neural pipeline requires exporting the model and specifying the environment (libraries and versions), hyperparameters (for the model), and the model itself. Language models are often published on open-source forums like the *Hugging Face* platform[[2]](#footnote-2). When a model has to be made available as a service for *inference*, robust monitoring also needs to be set to ensure performance and *model safety* at scale, apart from an optimised input pre-processing pipeline, as discussed above. In many production systems, a feedback loop is also implemented to improve the model over time.

**2.3 Morphology**

Different forms of information involved in the processing language are listed in Table 2.1. In this chapter, we will focus on structural, syntactic, and semantic information processing.

Before we delve into computational methods of developing language models, it is imperative to examine the most fundamental units of linguistic structure – the *word*. Words play an integral role in our ability to use language to express our emotions and creativity, originating from the fundamental question: what do we know when we know a word?

The formal study of the internal structure of words and the relationship among words is called *morphology*. The term itself is derived from the Greek word *morphe*, meaning ‘form’, and *ology*, meaning ‘the branch of knowledge’. Morphology also refers to our internal grammatical knowledge concerning the words and how their usage change based on language, geography, context, and time. Languages like Hindi, Turkish, and Hungarian are considered *morphologically rich*, whereas English and Chinese are *morphologically poor*. In morphologically rich languages, the word forms of some word classes, like verbs, may vary a lot depending on the context. Take the phrase ‘*will go*’ with its usage as described in Table 2.2.

|  |  |  |
| --- | --- | --- |
| **English** | **Hindi** | **Tamil** |
| I will go | मैं जाऊंगा | நான் ேபாேவன |
| We will go | हम जायेंगे | நாம் ேபாேவாம |
| You will go | तुम जाओगे | நீ ேபாவாய |
| He will go | वह जाएगा | அவன் ேபாவான |
| She will go | वो जाएगी | அவள் ேபாவாள |

Table 2.2: Different forms of the token ‘*will go*’ in morphologically-poor (English) and morphologically-rich languages (Hindi and Tamil). Morphologically-rich languages have various forms to represent the same token depending upon the subject in the sentence. Such languages also have additional grammatical classes.

For *morphologically-poor* English, irrespective of whether the action is being performed by a single person, a group of people, or by people of different genders, the phrasing ‘*will go*’ remains the same. Meanwhile, in a *morphologically-rich* language like Hindi, the phrasing will get modified to suit the respective form depending on the preceding noun form (plurality, gender, etc.) and tense form (first person, third person, etc.).

***2.3.1 Morphemes***

Words are composed of *atomic* building blocks called *morphemes*. The words ‘*taking*’ and ‘*courses*’, for instance, are made up of basic units like take and course, and the other blocks like -ing and -s convey additional meanings, such as a sense of the nature of action or plurality, respectively. Some morphemes independently constitute a word by themselves. They are called *free morphemes*. The word *fish*, for example, consists of a singular free morpheme, the word itself, with a predefined meaning. Other morphemes are not words by themselves but are parts of words—these are *bound morphemes*. *Affixes* are the most common type of bound morphemes. They attach to a *base word* or a *stem* and modify its meaning in some way or another. For example, the word ‘*taking*’ consists of the suffix morpheme ‘*ing*’ attached to the base ‘*take*’.

Nouns, verbs, adjectives, and adverbs are put into the bucket of *content words*. Content words are often called *open class words* because we regularly add new words to this bucket. Other classes of words do not have precise lexical meanings or obvious concepts associated with them, including conjunctions (*and*, *or*), prepositions (*to*, *from*, *at*, *with*), articles (*a*, *an*, *the*), quantifiers (*all*, *few*, *many*, *some*), demonstratives (*this*, *that*) and pronouns. These kinds of words are called *function words* because they serve a grammatical function. They are also called *closed class words* as most languages have a small, fixed number of words that fall into this bucket.

A root is the base form of a word that cannot be analysed or reduced further without destroying its meaning. For example, in terms of conserving its meaning, the term ‘*forest*’ cannot be broken down into ‘*for*’ and ‘*est*’. Complex words may consist of a morpheme root and one or more affixes. Affixes like *un*, *dis*, *mis*, *re*, *non*, *sub*, *super*, *anti*, *inter*, and *intra*, that are attached to the beginning of another morpheme are called *prefixes*. Similarly, *suffixes* are morphemes that get added at the end, such as *ing*, *ness*, *ly*, and *able*. A morpheme that is a prefix in one language, in a semantic sense, maybe a suffix in another and vice versa.

|  |  |  |
| --- | --- | --- |
| **Original Word** | **Stemming** | **Lemmatisation** |
| witnessed | wit | witness |
| assignments | assign | assignment |
| considerable | consider | considerable |
| democratisation | democrat | democratisation |
| interpolated | interpol | interpolate |
| effectively | effect | effective |

Table 2.3: Comparing the results of Porter Stemmer and WordNetLemmatizer algorithms for various words.

Historically, we have been following morphological rules that govern how these affixes attach to the base word. For instance, when we add prefixes, the resulting word is formed by putting together the two morphemes as-is (e.g., *pre* + *flight* = *preflight*). In contrast, the resulting word might not be a simple concatenation in many suffixes (e.g., *ready* + *ly* = *readily*). In English, as well as many other languages, apart from attaching affixes, new words can also be formed by *compounding* existing words, where individual words, like ‘*black*’ and ‘*board*’, can be joined together to form a *compound word* like ‘*blackboard*’. In other cases, words like ‘*will*’ and ‘*would*’ are contracted to *-’ll* and *-’d* and attached to the end of words. Identifying the various parts of a word into the morphemes that it is composed of and producing its structured representation is called *morphological parsing* or *stemming*.

***2.3.2 Stemming***

A stemming algorithm or *stemmer* is the one that eliminates affixes and serves as a heuristic to normalise the inflectional (plurals, tenses, etc.) and derivational (turning verbs into nouns) forms of a word. For example, the words *run*, *runs*, *ran*, and *running* all refer to the same underlying concept and can be represented by a single concept instead of four different ones. However, stemming can be tricky as we can lose information by chopping off a few characters of a word indiscriminately. In order to support stemming, a variety of heuristics (rule-based) algorithms have been proposed. NLP packages often include the famous stemming algorithms—the Porter[[3]](#footnote-3) and Snowball[[4]](#footnote-4) Stemmers.

A stem may not be a valid dictionary word but merely an abstraction that represents all the words that look the same at the character level. For instance, if we have a stemming rule to remove all instances of ‘*s*’ from the end of words in order to normalise plural forms, we might end up with non-meaningful results as well – ‘*lens*’ becomes ‘*len*’, which is not a known English dictionary term, yet will be acceptable as per the stemming rules.

***2.3.3 Lemmatisation***

Instead of normalising the words at the superficial character level, we can group them based on their larger context and usage. *Lemmatisers* are algorithms that normalise words down to the underlying semantic form – the *lemma*. Lemmatisers are usually more accurate than stemmers as they use a knowledge base or thesaurus of words, their synonyms, and forms to ensure that only words that mean the same are clustered together and are represented by a well-defined lemma instead of an arbitrary stem, which may not be a dictionary word. This difference is easier to understand with the examples in Table 2.3. A lemmatiser will be able to group the words ‘*good*’, ‘*better*’, and ‘*best*’ into the same bucket if it knows that these words are adjectives. A table or dictionary lookup is often the way how lemmatisers retrieve information about similar-meaning words. WordNet is a famous database of English words that are linked together by semantic relations.

***2.3.4 Lexicon***

Stemming or lemmatisation helps reduce the signal-to-noise ratio in a text corpus by reducing the redundant concepts present in it. The process allows us to build an optimal vocabulary/lexicon that makes up the language of the corpus. This lexicon defines the input and output space for the language model trained on the corpus. Many classical tasks in NLP, like sentiment analysis, NER, and POS tagging, as well as domain-specific tasks like medical or legal text analysis, depend upon a lexicon for making sense of the input. For many of these tasks, we prefer to use specialised lexicons (e.g., AFINN[[5]](#footnote-5), SentiWordNet[[6]](#footnote-6), EmoLex[[7]](#footnote-7), PropBank[[8]](#footnote-8)) that are built up by manually annotating with the help of human experts, automatic extraction using statistical and machine learning techniques or using a hybrid approach. The intuition behind the lexicon also plays a role in the formation of rules and conventions to incorporate new terms like ‘*tweet*’ and ‘*hangry*’. They can be formed due to the adoption of popular culture, foreign words, compounding, or due to morphological changes.

**2.4 Tokenisation**

In order to build the lexicon, the question is how we define the boundary of breaking the text stream into entities that can be added to the lexicon. Commonly, these informative units of information in NLP are called *tokens*, and the process of obtaining tokens by breaking the text corpus into smaller processable units/chunks is called *tokenisation*.

For example, consider an input corpus consisting of two sentences: S1: ‘*I want the first token from the list of tokens.* and S2: ‘*The tokens are obtained via tokenisation*’.

**Sentence/Word/Character-Level Tokens.** For the above example, sentence-level tokenisation will yield a list [‘I want the first token from the list of tokens.’, ‘The tokens are obtained via tokenisation’.] by splitting the sentences at the punctuation marks. However, this naive splitting at punctuation can also be problematic; the phrase ‘*But, here we are!*’ can be wrongly split at ‘,’ instead of ‘!’. To reduce ambiguity, one can also split the sentence at the word level. In English, word level splitting is easier as whitespace is the default delimiter. Therefore, our text would be broken as the following word tokens: [‘I’, ‘want’, ‘the’, ‘first’, ‘token’, ‘from’, ‘the’, ‘list’, ‘of’ ‘tokens.’, ‘The’, ‘tokens’, ‘are’, ‘obtained’, ‘via’, ‘tokenisation.’]. When splitting by spaces, the tokens ‘tokens.’ and ‘tokenisation.’ have punctuation attached to them. To reduce the number of unique tokens in tokenised output, we can either discard the punctuation altogether or add them separately to the token list as [‘I’, ‘want’, ‘the’, ‘first’, ‘token’, ‘from’, ‘the’, ‘list’, ‘of’ ‘tokens’, ‘The’, ‘tokens’, ‘are’, ‘obtained’, ‘via’, ‘tokenisation’, ‘.’]. Once we have obtained the words, we can make the chunking process even more granular by operating at the character level. In that case, our corpus will be listed as [‘I’, ‘ ’, ‘w’, ‘a’, ‘n’, ‘t’, ‘t’, ‘h’, ‘e’, ‘f’, ‘i’, ‘r’, ‘s’, ‘t’, ‘t’, ‘o’, ‘k’, ‘e’, ‘n’, ‘f’, ‘r’, ‘o’, ‘m’, ‘T’, ‘h’, ‘e’, ‘l’, ‘i’, ‘s’, ‘t’, ‘o’, ‘f’, ‘t’, ‘o’, ‘k’, ‘e’, ‘n’, ‘s’, ‘t’, ‘h’, ‘e’, ‘t’, ‘o’, ‘k’, ‘e’, ‘n’, ‘s’, ‘a’, ‘r’, ‘e’, ‘o’, ‘b’, ‘t’, ‘a’, ‘i’, ‘n’, ‘e’, ‘d’, ‘v’, ‘i’, ‘a’, ‘t’, ‘o’, ‘k’, ‘e’, ‘n’, ‘i’, ‘z’, ‘a’, ‘t’, ‘i’, ‘o’, ‘n’]. In this example of character-level chunking, it becomes difficult to detect word boundaries. In later paragraphs, we will discuss how to overcome this issue.

*N***-grams.** So far, we have observed tokens as one unit at a time. This form of token is also called *uni-gram*, with *uni* being the unit of tokenisation referencing the quantity *one*. However, we can also look at neighbouring tokens, such as *n* tokens, ahead of the current token, leading to *n*-grams instead. For example, when *n* = 2, our word-level tokens will be of the form [‘I want’, ‘want the’, ‘the first’, ..., ‘via tokenisation’, ‘tokenisation <EOS>’], where <EOS> is the unique token indicating we have reached the end of our text stream. Similarly, for *n* = 3, the first token will be ‘I want the’ and so on. The n-gram operation can be performed at the sentence or character level as well. As the window size for *n* increases, we are able to capture more semantic context; however, with a very large value of *n*, we end with the whole stream, defeating the purpose of performing chunking. The task of obtaining the optimal number of *n*-grams is task and data-specific.

***2.4.1 Advanced Techniques: Subword Tokenisation***

On the one hand, character-level tokens provide more resilience against spelling errors. On the other hand, it comes at the cost of semantic information. For example, the subword ‘*ken*’ can be part of semantically diverse terms, ‘*Kendall*’, ‘*token*’, or ‘*broken*’. Here, practitioners have come up with a tokenisation process that is a combination of word and character levels tokens, known as *sub-word tokenisation*, which is primarily based on splitting and merging tokens based on the frequency of occurrence within a corpus. In this section, we discuss the two most widely adopted bottom-up subword tokenisation techniques that take a greedy approach based on the frequency of subword occurrence—Byte Pair Encoding and Wordpiece Tokenisation.

*<H4> Byte Pair Encoding (BPE)*

Byte pair encoding (Gage 1994) was initially developed as an algorithm to encode/compress a text based on the most frequently occurring bytes (a byte or 8 bits refers to a single character token for practical usage). The algorithm merges the most frequently occurring consecutive bytes and replaces them with a new representative token that is not part of the existing lexicon. The process continues until no more merger is possible (see Algorithm 1).

In order to preserve word boundaries, the space token is replaced by a special token, say ‘</w>’, which is not a part of the vocabulary and is concatenated to the last character of each word. Thus, our 2nd and 3rd words in the corpus, [‘w a n t’, ‘t h e’], will be represented as [‘w a n t </w>’, ‘t h e </w>’].

*FCBPE* (*i, j*) = Σ(*i* : *j*) (2.1)

**Example 2.1.** Taking our initial corpus into consideration, let us observe a few iterations of BPE.

* *Iteration* 0 *(pre-tokenisation)*:Our tokens are enlisted as: [‘i’, ‘w a n t’, ‘t h e’, ‘f i r s t’, ‘t o k e n’, ‘f r o m’, ‘t h e’, ‘l i s t’, ‘o f’, ‘t o k e n s’, ‘t h e’, ‘t o k e n s’, ‘a r e’, ‘o b t a i n e d’, ‘v i a’, ‘t o k e n i z a t i o n’]. Our unique vocabulary is enlisted as: {‘i’ ‘w’ ‘a’, ‘n’, ‘t’, ‘h’, ‘e’, ‘f’, ‘r’, ‘s’, ‘o’, ‘k’, ‘m’, ‘l’, ‘b’, ‘d’, ‘v’, ‘z’}.
* *Iteration* 1:Among the possible character combinations, the most frequently occurring character pairs are ‘o’ + ‘k’, occurring 4 times. Thus, all occurrences of ‘o’ + ‘k’ will be replaced by ‘ok‘. The updated tokens thus appear as [‘i’, ‘w a n t’, ‘t h e’, ‘f i r s t’, ‘t ok e n’, ‘f r o m’, ‘t h e’, ‘l i s t’, ‘o f’, ‘t ok e n s’, ‘t h e’, ‘t ok e n s’, ‘a r e’, ‘o b t a i n e d’, ‘v i a’, ‘t ok e n i z a t i o n’] with {‘ok’} added to the unique count.
* *Iteration* 2*:* Now, looking at all paired frequencies with ‘ok‘ considered as a single unit, we observe that ‘t’ + ‘ok’ occurring four times is the next pair to be merged. This updates the word list as [‘i’, ‘w a n t’, ‘t h e’, ‘f i r s t’, ‘tok e n’, ‘f r o m’, ‘t h e’, ‘l i s t’, ‘o f’, ‘tok e n s’, ‘t h e’, ‘tok e n s’, ‘a r e’, ‘o b t a i n e d’, ‘v i a’, ‘tok e n i z a t i o n’] with ‘tok’ added as a vocabulary term.
* *Iteration N*: After *N* merger and replacement steps, our words will be represented as: [‘i’, ‘w a n t’, ‘the’, ‘f i r st’, ‘token’, ‘f r o m’, ‘the’, ‘l i st’, ‘o f’, ‘tokens’, ‘the’, ‘tokens’, ‘a r e’, ‘o b t a i n e d’, ‘v i a’, ‘token i z a t i o n’] and our final vocabulary will be: [‘i’, ‘w’, ‘a’, ‘n’, ‘t’, ‘h’, ‘e’, ‘f’, ‘r’, ‘s’, ‘o’, ‘k’, ‘m’, ‘l’, ‘b’, ‘d’, ‘v’, ‘z’, ’ok’, ‘tok’, ‘en’, ‘token’, ‘th’, ‘the’, ‘st’, ‘tokens’]

Once the vocabulary is learned from the initial corpus, the algorithm can break any word it has seen (in the corpus) or not seen before (an on-the-fly word) based on the subword token it has learned. For example, the new word ‘*mist*’ will be tokenised into ‘m i st’, with ‘m’, ‘i’, and ‘st’ forming the sub-words. Including the word boundary, we can represent ‘mist’ as ‘m i st </w>’. Note that the subwords do not have to be actual dictionary terms with a meaning attached to them.

|  |  |  |
| --- | --- | --- |
| **Algorithm 1** The steps for subword tokenisation as adopted by BPE and WordPiece. The actual formula in Step 4 is realised by Equations (2.1) and (2.2), respectively, for BPE and WordPiece. |  | **Algorithm 2** Algorithm for obtaining unique tokens in the corpus via splitting at the word level. |
| **Input:** Vocabulary size *k*, Corpus *D*, Maximum Iteration *maxiter*  **Output:** Vocabulary V  1: *V* ***←*** *PREPROCESS(D)*  2: *i* ***←*** 0  3: **while |***V****|*** *< k* or *i* <*maxiter* **do**  4: *tl* : *tr* ***←*** *max (FC (t l, t r))*  5: *tlr* ***←*** *tl : tr*  6: *tl*:*tr* ***←*** *tlr*  7: *V* ***←*** *V* **∩** *tlr*  8: *i* ***←*** *i* + 1  9: **end while**  **return** V |  | **Input:** Corpus *D*  **Output:** Vocabulary V  1: *V* ***←***{}  2: **for** *w* ∈*split(D, delimiter=" ")* **do**  3: **for** *ch* ∈*w* **do**  4: *V* **∩** *ch*  5: **end for**  6: **end for**  **return** V |

*<H4> WordPiece Tokeniser*

The processing of merging characters in BPE depends solely on the frequency count of the characters at each iteration. Instead of maximising information gain purely based on frequency, we can maximise the likelihood of improving the subword’s coverage within the corpus. A famous probabilistic variant of BPE is the WordPiece Tokeniser. Keeping the rest of the BPE process the same, the primary modification that WordPiece introduces is by replacing the exact frequency count in Algorithm 1 at step 4 with Equation (2.2).

 (2.2)

By dividing the frequency count Σ of the co-occurring pair by the product of individual frequency counts (Equation 2.2), WordPiece penalises those pairs that are highly frequent in the corpus. In other words, if particular terms are by themselves high frequency, then their combination provides lesser information gain compared to combining less frequent terms.

*<H4> SentencePiece Tokeniser*

So far, both the tokenisation methods we have examined require the corpus to be split at a word level and be preprocessed at an individual word level, for which the assumption is the language of the corpus contains spaces as delimiters. However, there are some languages, like Chinese or Japanese, in which space delimitation is not available. Thus, a language-agnostic/space-agnostic approach is required. Here, the SentencePiece tokeniser comes into play. SentencePiece incorporates a number of techniques to improve upon the existing tokenisation setup. SentencePiece employs Unicode Normalization to work with raw texts. It employs heap sort to keep track of the vocabulary size. But most importantly, unlike BPE and WordPiece, which employ a pre-tokenisation step (Step 2, Algorithm 1), SentencePiece is capable of working with raw texts.

**2.5 Syntactics**

As per the Oxford Dictionary, the term ‘syntax’ refers to the rules/grammar that state how words are placed and used in a language to form sentences. The syntax is based on the grouping of words in a natural order. An English sentence is composed of a group of words that form the Noun Phrases (NP) and the Verb Phrases (VP). For instance, in the sentence ‘*The old house in the neighbourhood is being demolished’,* the noun phrases *‘the old house’* and *‘neighbourhood’* can be combined as a single noun phrase *‘The old house in the neighbourhood*’. Meanwhile, the phrase ‘*is being demolished*’ is the verb phrase. The whole sentence can syntactically be represented as *S* → *N P* + *V P,* with N P further composed of *N P* → *N P* + *N P*. This process of mapping words and groups of words (phrases) into their grammatical units is called *syntax parsing*. While linear representation in terms of rules is a way to decompose the sentence, it can also be represented in the form of a hierarchy or a tree, with the words forming the leave nodes, the grammatical constituents forming the intermediate nodes, and the sentence forming the root node. In this section, we provide an overview of three ways in which the syntax tree can be parsed. The parse trees act as an abstraction of the sentence.

**[Insert Figure]**

Figure 2.4: Constituent Parsing for the Sentence, ‘*The mouse ate the cheese that was kept in the drawer*’.

**Dependency Parsing.** While performing POS tagging and constituency parsing, we implicitly looked at the relation among the words to assign adequate tags and phrases. Still, the information was insufficient to answer questions such as *What did the mouse eat*? or *Where was the cheese kept*? In such cases, we need to mark the relation between *mouseate–cheese-drawer* explicitly. Being able to state the subjects and objects in a sentence, along with the relationship among them, is known as dependency parsing. The dependency grammar describes the structure of a sentence in terms of the words and the grammatical relationship that holds between words. The dependency relations thus act as a proxy to the semantic relations in text. These binary relations consist of a *head* and a *dependent*. The head is the central word in a constituent (e.g., noun in a noun phrase, verb in a verb phrase). All other words are dependent on the head. In a dependency parse tree, the heads are linked to words that are immediately dependent on them. The main verb of the sentence is the root node from which one can follow a unique directed path to each word in the sentence. Such a parse tree is flexible with word order and is helpful in parsing morphologically rich languages as well. Figure 2.5 shows the parse tree for an example sentence ‘*The mouse ate the cheese that was kept in the drawer*’. The actual parsing is realised through *transition-based state spaces* that use stacks to create dependency structures and graph-based methods that use maximum spanning trees.

**[Insert Figure]**

**Figure 2.5:** The dependency parse tree for the sentence, ‘*The mouse ate the cheese that was kept in the drawer*.’ The labels on the arcs are according to Universal Dependency nomenclature for grammatical relations.

**2.6 Semantics**

In the last section, we saw how grammatical abstractions can help answer simple questions within a sentence. Instead of the question *What did the mouse eat*? if we were to ask *Which furniture is being referred to*? then the notion of ‘furniture’ and ‘drawer’ being concepts that are close to each other needs to be established. This idea of establishing closeness of concepts that may linguistically or grammatically not appear close to each other is known as semantic similarity. *Semantics*, in turn, can be defined as the underlying meaning associated with the entity under consideration. Semantics help access what is the relation that different words have with each other when present together in a sentence. For example, when presented with a *stimulus word*, ‘*bank*’, we think of other *response words* like ‘*money*’, ‘*river*’, and ‘*blood*’ depending on the context in which the *stimulus word* is used. The way a language evolves plays a central role in explaining these relations. Here, word association can be defined as a relationship between words in a language based on their meaning. Semantics is not just concerned with the meaning of words but also how to combine words into meaningful phrases and sentences. For example, the phrases *‘not honest’* and the word ‘*dishonest’* carry the same connotation/semantics even though the terms are lexically different.

Semantic parsing involves mapping the natural language input to a logical form that connects the language to real-world concepts. Unlike syntactic parsing methods, which focus solely on structure and grammar, semantic parsing methods try to extract the meaning and context of a sentence. A semantic parser consists of a formal knowledge representation technique and an inference mechanism. One of the ways to represent language formally is by translating a sentence to first-order logic, where the predicates are the words in the sentence. In order to represent the words as predicates, we need to be sure of the *sense* in which the word has been used in the sentence. Even if we can represent words and relationships as predicates, these do not make sense on their own. An external knowledge source is required to help us define rules that use these predicates and learn the semantic logic. In this section, we briefly describe three techniques for semantic parsing—decomposition, ontology, and distributional statistics.

**Decompositional Semantics.** We can derive the meaning of a word by dividing it into various semantic components or qualities. For instance, in the sentence ‘*The mouse ate the cheese that was kept in the drawer’,* the word ‘*mouse’* implies that the *subject* of the sentence is a *mammal* and a *terrestrial* but not a *human.* However, if the word ‘*mouse’* was replaced with the word *‘boy’,* it would imply that the subject has all of the three qualities—being a *mammal,* a *terrestrial* and a *human.* These decomposed semantics can also be mapped to first-order logic, such as mouse ⇒ mammal ∧ terrestrial ∧ ¬ human and boy ⇒ mammal ∧ terrestrial ∧ human.

**Ontological Semantics.** Another way of decomposing the meaning of a word is by studying its relationship to other words. Take the classic example of the word ‘*bank’.* ‘Bank’ itself means a collection or storage. However, what that collection is about – ‘water’, ‘blood’, or ‘money’—dictates the exact definition that word will semantically adopt. This process of defining the existence/usage of a term with respect to a sentence is called *ontology. WordNet* (Miller 1995)is a famous lexical and ontological resource in English. It contains various kinds of relations that exist between English words. For example, the word *‘small’* might be synonymous with *‘little’,* while it conveys the opposite meaning to *‘large’.* The notion of a *‘mouse’* implies that it is a type of *animal.* Capturing these relations in the text is crucial to understanding world knowledge.

**Distributional Semantics.** So far, in our discussion of semantics, we have assumed the computational methods to carry the same level of contextualisation as humans. While machines lack subconscious contextualisation, they can approximate the same by analysing large corpora of text and deriving a sense of words based on their distributional properties (e.g., co-occurrence, frequency). This maps to the law of association that *words with similar distributions might have similar meanings.* For instance, the meaning of the word *‘mouse’* may be complex for the machine to grasp, yet it can be inferred from the contexts it appears in, i.e., sentences where it co-occurs with words like ‘*rodent*’, ‘*animal*’, ‘*food*’, etc. Distributional Semantics forms the core of the modern-day NLP.

**2.7 Introduction to Language Modelling**

According to Herbert Clark, whenever two words occur together or in close proximity, an *associative link* is formed between them in our mind over time, and the more frequently they appear together, the stronger the association (Clark 1970).

Building up word association and logic of distributional semantics, we can describe a Language Model (LM) as a model that learns the probability distribution over the words in the corpus. This probability is learned based on the frequency co-occurrence of words in a large training corpus. Once trained/learned, the LM attempts to predict the next token in a sequence of tokens. For a sequence of *m* tokens, *x*1*, x*2*, . . ., xm*, the LM predicts the (*m* + 1)*th*token, *xm*+1 based on the language learned from its training corpus of words and phrases. The output space, i.e., the set of all possible words that can be the (*m* + 1)*th* token in a sequence, is the whole vocabulary/lexicon learned over the language. If the LM is learned over *N* unique tokens, then in the worst case, each of *N* tokens has an equal and independent probability of 1*/N* for being the (*m* + 1)*th* token.

However, from our semantic and syntactic parsing, we know that for a given sentence, not all words have an equal probability of occurrence. Instead, the words that can appear next are *conditioned* on the words that are present so far in the sentence. It forms the basis of language modelling in NLP. In layman’s terms, a language model predicts the probability of the (*m* + 1)*th* token given a sequence of *m* tokens seen before. Going back to our example sentence, if you are asked to predict the next word in the sequence of ‘*Hello Sam. How are*’, of all the words we know in English (i.e., our vocabulary), the most likely next word should be ‘*you*’. This likelihood is the probability spread over the whole vocabulary of which ‘*you*’ has the highest probability score. We will introduce the formal concepts of conditional probability and language modelling in detail in Chapter 4.

**Bag-of-Word Based Representation.** Forgoing the notion of conditional probability, one can still obtain a crude form of language modelling that depends solely on the constituted tokens present in the sentence. Let us consider the task of sentiment analysis. A simple method for determining whether a sentence expresses positive sentiment would be to count the favourable and negatively connotated lexical terms that occur in the sentence. The process is solely based on the *occurrence* of individual words and not where and how they appear in the sentence, i.e., the notion of semantics or syntax is overlooked. Such setups are called the *bag-of-word*approach, where we know the words in the bag but not the order in which they are placed in the bag.

**Example 2.2.** Let us understand the bag-of-words modelling via a simple example of sentiment classification.

Consider three sentences that represent three samples of sentiment analysis: S1: ‘*The movie is bad.*’, S2: ‘*The movie is good.*’, S3: ‘*I liked the movie*’.

After preprocessing (lowercasing, punctuation removal), lemmatisation (liked → like), and tokenisation, we end up with a unigram vocabulary set [*the*, *movie*, *is*, *bad*, *good*, *I*, *like*]. Based on the unique vocabulary, the sentences can then be represented as vectors of length 7, indicating whether the *i*thindex vocabulary term is present in the sentence or not.

Thus, S1 = [yes, yes, yes, yes, no, no, no] and mapped numerically as [1, 1, 1, 1, 0, 0, 0] where 1 means ‘yes’—the token is present, and 0 means ‘no’—the token is not present in the given sentence. In a similar way, S2 and S3 become [1, 1, 1, 0, 1, 0, 0] and [1, 1, 0, 0, 0, 1, 1], respectively.

Further, each sentence has a sentiment label associated with it where –1 means negative sentiment, 0 means neutral, and 1 means positive. Our example sentences have a sentiment score of S1: –1, S2:v1, and S3: 1, respectively. From the crude analysis of the sentence vectors, we see that tokens ‘*the*’ and ‘*movie*’ occur in all three sentences and do not lead to any differentiation for the sentiment classification, i.e., we cannot tell by looking at only these two terms if the movie is good or bad. Meanwhile, the presence of ‘*bad*’ in S1 and its subsequent absence in S2 and S3 is an indicator of associating the presence of ‘*bad*’ with the label –1. Language models build on bag-of-word representation and try to learn such heuristics between tokens and labels based on the frequency of occurrence of the tokens in different class labels.

The notion of building representations from term frequency is detailed in Chapter 3.

**Part II: Neural Networks**

So far, the algorithms we have discussed for parsing and understanding language are based on simple heuristics and probabilities. To develop a more advanced and nuanced understanding of language, we must work with neural networks. This part of the book will help readers establish a basic understanding of computational neural networks. The theoretical foundation for these networks can be traced back to the independent works of Alexander Bain in 1873 and William James in 1890. Both hypothesised that human thoughts and decisions emerge from interactions among billions of *neurons* in the human brain. This biological network of nerve cells is responsible for all human reasoning and decision-making. Warren McCulloch, a neuroscientist, and Walter Pitts, a logician, laid out a theoretical model for a biological nerve cell in 1943. They called it a *perceptron*. In 1957, Frank Rosenblatt, a psychologist, provided an early hardware implementation of a perceptron. Rosenblatt took a linear combination of different input variables and gave a response of 1 or 0, depending on whether the linear combination of input variables was positive or negative. The version of the perceptron we use today was introduced by Minsky and Papert in 1969. They introduced the concept of an *activation function*, an essential component of all artificial neural networks used today.

**2.8 The Perceptron**

The architecture of computational neural networks is inspired by the nervous system in humans, where a network of neurons is responsible for processing, relaying, storing, and recalling information. A biological neuron receives signals from other neurons and chooses to transfer the processed signals to neighbouring neurons, depending on the outcome of processing. We can replicate the same in software via the *perceptron*, which is the most straightforward software implementation of a biological neural cell.

***2.8.1 Definition***

Given a *N*-dimensional input vector x = (*x*1, *x*2, …, *xN*), the perceptron computes a linear combination *w*1*x*1 + *w*2*x*2 + …+ *wnxn*, adds a term *β* and decides to output among the values {–1*,*0*,*1} depending on the computation. Formally, a perceptron can be represented by Equation (2.3), where **w** = (*w*1, *w*2, …, *wN*) is called the *weight vector, β* is called the *bias.*

*y* = sgn (**w***T***x** + *β*)(2.3)

where sgn(·) is the signum function defined as:



Note that the sgn(·) is a step function. We will slightly modify this function in the next section to model some elementary boolean functions.

***2.8.2 Implementing AND, OR, and XOR Logic***

Given that any computational task can be decomposed into a combination of Boolean operations, exploring the scope of modelling such functions using the perceptron is highly motivated. We will attempt to model some elementary boolean functions using the perceptron defined in the previous section (with a slightly modified definition of sgn(·) function). In particular, we will model AND, OR, and XOR Boolean functions; their function definitions are shown in Tables 2.4, 2.5 and 2.6, respectively. These are binary functions as they take two input variables, denoted by *x*1 and *x*2*,* and the output, denoted by *y,* within one of the possible values {0, 1}.

|  |  |  |
| --- | --- | --- |
| *x*1 | *x*2 | *x*1 AND *x*2 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Table 2.4: The AND Function.

|  |  |  |
| --- | --- | --- |
| *x*1 | *x*2 | *x*1 OR *x*2 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 2.5: The OR Function.

|  |  |  |
| --- | --- | --- |
| *x*1 | *x*2 | *x*1 XOR *x*2 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 2.6: The XOR Function.

The *AND*function (or gate) implements logical conjunction. It takes two Boolean inputs (either 1 or 0) and produces an output according to the Truth Table 2.4. We can model the AND function using a perceptron, where we have to assign such values to *w*1*, w*2and *β* so that Equation (2.4) below satisfies Table 2.4 with *y = x*1AND *x*2*.*

*y* = sgn′(*w*1*x*1 + *w*2*x*2 + *β*) (2.4)

where *x*1*, x*2∈ {0,1} and sgn’(·) is defined as:



−

**[Insert Figure]**

Figure 2.6: 2D plots showing different boolean logic functions and the corresponding line (dotted) separating the input coordinates with different output values for AND (Left), OR (Centre), and XOR (Right). Note that no separating line exists for the XOR function.

The 2D plot for the AND function in Figure 2.6 (Left) shows the line *x*2= *x*1+ 1*.*5 linearly separating the input coordinate points associated with 1 and 0 output values. If we let *w*1= 1, *w*2= 1, and *β* = – 1*.*5, then the perceptron model *y* = *sgn'(x*1+ *x*2 *–*1*.*5) emulates the AND gate as *y* attains 1 if and only if both the inputs *x*1and *x*2assume value 1. We can verify this from Table 2.7.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *x*1 | *x*2 | *x*1 + *x*2 – 1*.*5 | *y* = *sgn'*(*x*1+ *x*2***–*** 1*.*5) | *x*1 AND *x*2 |
| 0 | 0 | –1.5 | **0** | **0** |
| 0 | 1 | –0.5 | **0** | **0** |
| 1 | 0 | –0.5 | **0** | **0** |
| 1 | 1 | 0.5 | **1** | **1** |

Table 2.7: The perceptron model *y* = *sgn'*(*w*1*x*1+ *w*2*x*2+ *β*)with *w*1= 1, *w*2= 1 and *β* = –1*.*5 correctly models the Boolean AND function.

The *OR*gate (or function) implements logical disjunction. It receives two Boolean inputs (either 1 or 0) and produces an output according to the Truth Table 2.5. Following the same perceptron model defined in Equation (2.4), what values should the weight and bias be assigned? The 2D plot for the OR function in Figure 2.6 (Centre) shows the line *x*2= -*x*1+ 0*.*5 linearly separating the input coordinate points. Let *w*1= 1, *w*2= 1, and *β* = –0*.*5. Then, the perceptron model *y* = sgn*′*(*x*1+ *x*2 *–*0*.*5) emulates the OR gate as *y* attains 0 if only if both the inputs *x*1and *x*2assume value 0.

**What is happening with XOR?** If we observe Figure 2.6, we see that the AND and OR functions possess a linear boundary separating the points labelled with output values. For XOR, no such boundary exists. If we try to model the XOR function using the perceptron definition in Equation (2.4), we will fail to model it.

**2.9 Multilayer Perceptron**

To be able to model more complex functions, we need to generalise the perceptron architecture. Let us define a more general neuron-like processing unit where we replace *sign(·)* with a generic function *ϕ*(·)termed as the *activation function* (or *transfer function*)*.* A *neural network* is realised as a combination of such neuron-like processing units as formulated in Equation (2.5).

 (2.5)

where *wi* is the *ith* component of the weight vector, *xi* is the *ith* component of the input vector, and *β* is the bias term. Note that the output *z* is also termed as the *hidden unit*.More specifically, in this chapter, we will learn about *feed-forward neural networks,* where these units are combined in a tree-like fashion without any cycles.

The most straightforward feed-forward neural network is the *Multilayer Perceptron* (MLP), as shown in Figure 2.7. Here, the neuron-like units are arranged in a set of layers, with each layer having some number of these identical units. The first layer is called the input layer, and the units in this layer receive the input features. The last layer is called the output layer, and the number of units in this layer can vary depending on the output required from the feed-forward model. All the layers in between are called the *hidden layers.* The number of layers is known as the *depth,* and the number of units in a layer is known as the *width* of that layer. As you might have guessed, *deep learning* refers to training neural networks with many hidden layers.

**[Insert Figure]**

Figure 2.7: Architecture of a Multilayer Perceptron.

**[Insert Figure]**

Figure 2.8: Implementing XOR Boolean function using an MLP with a single hidden layer and sgn’(·) as the activation function as defined in Section 2.8.2.

**Can an MLP model XOR function?** Can combining multiple perceptrons help in modelling the XOR function? Figure 2.8 shows the required MLP architecture. The XOR function returns 1 when exactly one of the inputs is 1. We can use hidden units to capture this information. Let *h*1(*=* sgn'(*x*1+ *x*2***–*** 0*.*5)) detect if at least one of the input features is 1 and let *h*2(= *sgn'*(*x*1+ *x*2***–*** 1*.*5)) detect if both the input features are 1. The output *y* will then be one if and only if *h*1 = 1 AND *h*2 = 0. From Table 2.8, we can see the values the hidden and output units attain at various values of input features.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Units** | **Hidden Units** | | **Output** | ***x*1 XOR *x*2** |
| ***x*1** | ***x*2** | ***h*1 *=* sgn'(*x*1+ *x*2 *–* 0.5)** | ***h*2 *=* sgn'(*x*1+ *x*2 *–* 1*.*5)** | ***y =* sgn'(*h*1 *– h*2 *–* 0*.*5)** |
| 0  0  1  1 | 0  1  0  1 | sgn*'*(0 · 1 + 0 · 1 – 0*.*5) = 0  sgn*'*(0 · 1 + 1 · 1 – 0*.*5) = 1  sgn*'*(1 · 1 + 0 · 1 – 0*.*5) = 1  sgn'(1 · 1 + 1 · 1 – 0*.*5) = 1 | sgn'(0 · 1 + 0 · 1 – 1*.*5) = 0  sgn'(0 · 1 + 1 · 1 – 1*.*5) = 0  sgn'(1 · 1 + 0 · 1 – 1*.*5) = 0  sgn'(1 · 1 + 1 · 1 – 1*.*5) = 1 | sgn*'*(0 · 1 + 0 · – 1 – 0*.*5) = **0**  sgn*'*(1 · 1 + 0 · – 1 – 0*.*5) = **1**  sgn*'*(1 · 1 + 0 · – 1 – 0*.*5) = **1**  sgn*'*(1 · 1 + 1 · – 1 – 0*.*5) = **0** | **0**  **1**  **1**  **0** |

Table 2.8: Modelling the boolean XOR function using the Multilayer Perceptron in Figure 2.8.

***2.9.1 Neural Networks***

We will now formally define a neural network. Assume a neural network that takes an *N*-dimensional input vector, {*x*1*, x*2*, ..., xN*}) and outputs a *K*-dimensional vector {*y*1*, y*2*, ..., yK*}. The neural network contains a single hidden layer (depth = 1) with *M* neurons in the layer (width = *M*) realised as a linear combination via Equation (2.6), where *j* = 1*,* 2*, . .., M*.

 (2.6)

We refer to the parametersas *weights* and  as *bias*, following the terminology we laid out in the definition of the perceptron in Equation (2.3). The quantities  are known as *activations*. Each of them is transformed using an activation function *h*(*.*) as . In the context of neural networks,  is termed as the *hidden units*. Following Equation (2.6), the hidden units  are again linearly combined with suitable weights and biases to give *output activations* in Equation (2.7) :

 (2.7)

where *k =* 1, 2, ..., *K,* and *K* is the total number of output variables. Finally, the output activations are transformed using *activation function h'*(·)(which may be the same as *h*(·))to produce the output values *yk* as . We can combine Equations (2.6) and (2.7) to write the overall neural network function as follows:

 (2.8)

Succinctly, as Equation (2.8) describes, a neural network is simply a function that maps a set of input variables *x*1, *x*2,…, *xN* to a set of output variables *y*1, *y*2,…, *yK* using a series of controllable parameters: *weights* and *biases*. The forward processing of input variables in order to generate the set of output variables is termed the **forward propagation** of the neural network.

1. **Sigmoid**: The sigmoid/logistic activation *σ*(·) is defined as:

 (2.9)

The Sigmoid activation function (Equation 2.9) is generally used when we have to model probability as the output since the range of this nonlinear function is between 0 and 1. The derivative of the sigmoid activation function *f'* (·) can be written in terms of the function of itself as *σ'*(*x*) *= σ*(*x*)(1– *σ*(*x*)).

2. **tanh**: The tanh activation function tanh(·) is defined as:

 (2.10)

The output of the tanh activation function (Equation 2.10) is zero-centred within the range of –1 to 1. Hence, we can easily map output values as strongly negative, neutral, or strongly positive. The derivative can be written as tanh'(*x*) = 1 – tanh2(*x*).

3. **Softmax**: So far, we have looked at the activation function, which models only a single output at a time. If we want to classify a data point into one of many categories or classes, then we employ the softmax activation function. It outputs a well-defined probability distribution over all the output classes and is generally used in the output layer of the neural network. The Softmax activation is formally defined as:

 (2.11)

**Example 2.3.** Consider a list with frequency count as [1*,* 5*,* 2]. Converting this list into probabilities represented by softmax will require the denominator to be *D* = *e*1 + *e*5 + *e*2, and each frequency in the list can then be represented as [*e*1*/D, e*5*/D, e*2*/D*]or [0.02, 0.94, 0.04] whose sum approaches unity.

4. **ReLU**: ReLU stands for Rectified Linear Unit and is defined as:

*ReLU(x)* = max(0*, x*) (2.12)

The ReLU activation function (Equation 2.12) is far more computationally efficient than the sigmoid or tanh activation functions. This is because only a selective set of neurons are activated when ReLU is employed.

5. **GELU:** GELU stands for Gaussian Linear Unit and is defined by Equation (2.13), where Φ(*x*)is the cumulative standard normal distribution.

*GELU*(*x*) *= x* ***·*** Φ(*x*)(2.13)

The GELU activation function weights the incoming signal by their percentile rather than their sign. Consequently, GELU can be thought of as a smoother ReLU.

6. **GLU:** GLU stands for Gated Linear Unit. The *linear* gating is parameterised via a sigmodi activation (Equation 2.9) on weight *w* and bias *b,* as defined in Equation (2.14).

*GLU*(***x***)= **x** ⊙*σ*(**wx** + *b*)(2.14)

The operation ⊙represents the component-wise multiplication, and this allows the GLU to control the flow of information from the incoming vector **x** by learning which parts to emphasise or de-emphasise.

7. **Swish:** The Swish activation function also employs sigmoid activation to induce smoothness and differentiability. The Swish activation function *Swish*(·)is defined as:

*Swish*(*x*) *= x* ***·*** *σ(βx)* (2.15)

where *β* is a learnable parameter.

8. **SwiGLU:** SwiGLU stands for Swish-Gated Linear Unit Activation Function. As the name suggests, it combines Swish and GLU via Equation (2.16) and allows for better optimisation of weights.

*SwiGLU*(***x***)= **x** ⊙*Swishβ(****w*x** + *b)* (2.16)

**2.10 Training Neural Networks**

Now that we have established neural networks to be parametric nonlinear mapping functions, the question remains: how do we assign values to the network parameters, i.e., ***weights***and *biases?* We will elaborate on this in this section.

 (2.17)

After each update, the gradient is re-evaluated for the new weight vector, and the process is repeated. The error function is defined with respect to a training set, so each step/iteration requires that the entire training set be processed in order to evaluate **E**(**w**). This algorithm of training captured by Equation (2.17) is known as the *gradient descent*.

***2.10.1 Backpropagation***

Error functions usually comprise a summation of the error over each data point of the training set, . This reduces our problem to evaluating the gradient with respect to only one training sample (here *jth* training sample): ∇**E***j* (**w**).

**[Insert Figure]**

Figure 2.9: A basic neural network architecture with linear activation function and no hidden layers.

Referring to the neural network shown in Figure 2.9, the *kth* output unit *yk* (where *k* **∈** {1*,* 2*, ..., K*})is a linear combination of the input variables, *xi* (*i* **∈** {1*,* 2*, ..., N*})such that . As the process of gradient descent involves obtaining the derivative of the error function w.r.t to the weights, we prefer an error function that is continuous and differentiable, and the squared error is one of the most straightforward functions that fit this criterion. The squared error term for the *nth* training sample can be written as follows:

 (2.18)

The gradient of this error term with respect to weight *wij* is given by Equation (2.19). Note in the above example, *wij* connects the activation node associated with the *ith* input feature to the *jth* output value. In general, for a hidden layer, *wij* connects the *ith* unit of the previous layer to the *jth* unit of the next layer. The weight gradients in Equation 19 can then be used to update the weight *wij* according to Equation (2.17).

 (2.19)

This process of calculating error gradients by using the chain rule starting from the output layer to the hidden layers is termed as *backpropagation*.

**Example 2.4.** Let us understand more about backpropagation using a more general example involving a neural network with a single layer of hidden units and the tanh activation function, which we saw in Section 2.9.2. It receives *N*-dimensional input, **x** = (*x*1*, x*2*, . . . , xN*) and outputs a *K*-dimensional vector, **y** = (*y*1*, y*2*, ..., yK*). The hidden layer has *M* hidden units. Refer to Figure 2.10 for the neural network described above; in this figure, the output units have a linear activation function, *h*(*x*)= *x*, and the units in the hidden layer have *tanh*(*·*)activation function. The final output *yk* can be expressed as , with and . At the input level, .

**[Insert Figure]**

Figure 2.10: A neural network architecture with a single hidden layer and a nonlinear activation function.

We consider the standard sum of error squares as the error function. For *n*th training sample, let ***y*** *= yk* denote the predicted output and **t** = *tk* denote the target for. The error term for the *nth* training sample can thus be represented as follows:

 (2.20)

We can calculate the required gradients for backpropagation via Equations (2.21) and (2.22).

 (2.21)

 (2.22)

As,



and,



 (2.23)

***2.10.2 Batching***

We have already described in the gradient descent algorithm in Equation (2.17) that prior to weight update, a full pass through the training dataset is required. In the case of a large number of training samples, it becomes computationally expensive and slow to iterate over all the data samples at once. In order to improve the training process, we use two variants of *gradient descent.*

*<H4> Stochastic Gradient Descent*

This variant of gradient descent allows for the updation of the model parameters after processing a single training example.

 (2.24)

This variant allows for faster convergence towards the minima and is less memory-intensive (loads only a single sample to memory at a time) than vanilla gradient descent. However, by optimising after each sample, the model is more likely to overfit.

In the Stochastic Gradient Descent (SGD) algorithm, *stochastic* or *randomness* comes into play when we randomly select a data point to be optimised. While in each epoch, all the data points are processed, the order in which they can be processed can be randomly shuffled.

*Mini-Batch Gradient Descent*

In the case of the vanilla gradient descent, for *N* number of samples and *T* epochs, the weight optimisation operation happens only *T* times. In SGD, the optimisation operation happens *N* **×***T* times. Between optimising one sample at a time vs optimising all samples aggregated, we can update the gradient over a group of samples instead. Let *N* samples be grouped into a set of *n* smaller samples. The optimisation step is performed *n* **×***T* times such that *T < n* **×** *T < N* **×** *T.* This optimisation technique is called *mini-batching,* and each of the *n* sets is called a *batch,* denoted as *B.*

 (2.25)

**Example 2.5.** Consider the neural network in Figure 2.11. The network takes two input variables *x*1 and *x*2, outputs two continuous variables *y*1 and *y*2, and utilises the Sigmoid activation function at each hidden unit. At current training checkpoint, the weights have following values:  . The bias terms, *b*1 = 0*.*25 and *b*2 = 0*.*35.

Given a new training input vector **x** = (*x*1, *x*2) = (0.1, 0.5) and the expected output **t** = (*t*1, *t*2) = (0.05, 0.95), let us calculate the update for using stochastic gradient descent and *η* = 0*.*1.

We will first forward propagate through the neural network to store values of hidden units and predicted outputs.

**Forward Propagation:**

**[Insert Figure]**

Figure 2.11: The Neural Network Architecture for Example 2.5.



We will now calculate the error contribution due to this new training input vector.

***2.10.3 Hyperparameters***

As explained before, the training of a neural network involves processing all the samples in the training dataset for which the model is optimised. Once trained (i.e., no more weights are updated), it is imperative to determine how well the model will predict on *unseen* samples. The dataset on which we evaluate the generalisability of a trained neural network is called the *test dataset*. Note we assume that both training and testing samples are drawn from the same underlying distribution.

A neural network model is said to *underfit* if it fails to perform well even in the training stage. It can be a result of the smaller number of training samples from which to learn any meaningful patterns or the smaller complexity of the neural network that prevents it from learning more complex patterns within the training dataset, or both. On the contrary, a neural network is said to *overfit* if it performs well on the training dataset but fails to perform on the test set. In such cases, a neural network learns the noisy patterns in the training set, which leads to a lack of generalisability.

Thus, by controlling how complex the network is and configuring the learning rate *η*, we can, in turn, impact the learning process. Such configurable variables (explicitly declared before training) whose value controls the learning process are termed *hyperparameters*.

*<H4> Breadth and Depth*

Based on our understanding of overfitting and underfitting, it appears that the model’s complexity plays a vital role in the learning process. But how do we define the complexity of a neural network? In terms of the number of weight multiplication operations that form the basic building block of a neural network, we can control the complexity of the network by capping the number of activation units. Recall the concept of depth and breadth of an MLP in Figure 2.7. By increasing the depth of the network, we allow the system to model more complex functions. Meanwhile, by increasing the breadth of the network, we can accommodate more feature vectors. Both will enable us to reduce underfitting. Note that while theoretically, one can have infinite depth and breadth, such a system will overfit.

*<H4> Number of Epochs*

The ideal number of training iterations/steps is such that any further training provides little to no boost in test accuracy. The number of iterations is also known by the term number of *epochs*, where each epoch is complete when all the training samples have been processed.

*<H4>Learning Rate*

The learning rate *η* determines the magnitude of steps taken in the direction of decreasing gradient (Equation 2.17). A large learning rate implies taking larger strides, which may lead to scenarios where we keep hovering around the local minima without reaching it. In contrast, with a smaller learning rate, it takes too long to reach the optima. There are various strategies that one can use to manage the learning rate during the training of a neural network.

**1. Fixed Learning Rate**: In this training strategy, the learning rate remains constant throughout the training process.

**2. Time-Based Decay**: In this training strategy, the learning rate decreases proportionally to training steps. It is based on the idea that initially, the model will begin by predicting randomly and have a higher error rate. However, as the training progressed, the error would have reduced. , where *decay* is a factor by which the learning rate decreases, and *epoch* is the training iteration *t.*

***2.10.4 Regularisation***

Regularisation is another set of techniques that can help avoid overfitting during training.

1. **Early Stopping**: One of the most straightforward techniques to prevent overfitting is to limit the number of updates made to the weight parameters. Heuristically, if we can avoid the training loss from becoming arbitrarily low, the model will be less likely to overfit.
2. **L1 and L2 Regularisation**: By penalising larger weights while training, we can further reduce overfitting. Let us first look at the *Lp* norm of a vector **x** in an *n-*dimensional space, defined by .When *p* = 1, we call this the *L*1norm or *Manhattan distance* given by and when *p* = 2, we refer to it as the *L*2 norm, given by .[[9]](#footnote-9)

Employing the penalty term, we can minimise the error term **E**(**w**) via Equation (2.26) with *α* the *regularisation* constant.

By replacing *p* with 1 or 2, we obtain the *L*1or *L2* regularisation, respectively. *L*1regularisation allows for more sparse weight parameters. Unlike *L*1regularisation that forces weights to zero, *L*2 regularisation shrinks weights while ensuring that important components of the weight vector are larger than the others.

1. **Dropout**: As the name suggests, we randomly drop or freeze a fraction (dropout probability) of neurons from being updated. Suppose we are using mini-batch gradient descent; using a dropout regularisation would amount to training different weight parameters for various subsets of training data to avoid overfitting the entire training dataset. During test time, no neurons are dropped.

**2.11 Vanishing and Exploding Gradients**

When obtaining the derivative of the loss with respect to weights, the derivate value may be extremely small or large, leading to the problem of vanishing or exploding gradients.

**Vanishing Gradients.** This refers to the situation when the gradient information cannot be transferred from the output layers to the hidden layers due to the gradients assuming very small values. Following our previous notation, let *L* denote the index of the output layer. Then, we calculate the gradient of error term *E(w)* with respect to weights in different layers of the networks. For the *ith* hidden layer, let us denote the weight parameters as **w**(*i*), the hidden units as *z*(*i*) and activations as *a*(*i*) such that *z*(*i*) = *h(a*(*i*)),where *h* is the activation function. Note that each hidden layer would have multiple hidden units, but we do not label such hidden units to avoid complications.

As we note from Equation (2.27), the further the hidden layer *i* is from the output layer (deeper the neural network), the more terms of the form incorporating the partial derivative of the hidden unit with respect to the activation appear.

It so happens that these derivatives assume very low values for activation functions like sigmoid and tanh. ReLU activation is usually employed when there is a risk of a vanishing gradient problem.

**Exploding Gradients.** On the opposite spectrum is the problem where large error gradients accumulate and result in huge updates to neural network model weights during training. These may occur due to lousy initialisation of weights or some combinations of activation functions.

**2.12 Evaluation Metrics**

Once we have optimally trained our neural network, we need to be able to report how *well* the model is performing. Additionally, given that for a given set of input and target values, multiple *optimal* weights can be obtained. How do we determine which set of weights are the best for an unseen dataset? To perform this assessment, we utilise evaluation metrics.

Let us go back to the task of sentiment analysis. Suppose we have ten sentences that are labelled as either positive (1) or negative (–1). Out of these, seven samples are labelled as positive. Let us assume an arbitrary target label list for the ten samples as *y* = [1, 1, –1, 1, –1, –1, 1, 1, 1, 1], with the *ith* element of the list providing a sentiment label for the *ith* sentence.

* Case 1:How many times did we correctly predict the positive sentiment?
* Case 2:How many times did we incorrectly predict positive sentiments as negative?
* Case 3:How many times did we incorrectly predict negative sentiments as positive?
* Case 4:How many times did we correctly predict the negative sentiment?

**True Positive/Negative.** Case 1 of the confusion matrix can also be termed as *true positive* (TP) as we are truly/correctly predicting the positive class as positive. Consequently, case 4 is termed as *true negative* (TN) as we truthfully predict the negative class as negative.

**False Negative.** Case 2 can be understood as the number of times we erroneously/falsely produce a negative output (sentiment in our case) when the actual output is positive, i.e., *false negative* (FN).

**False Positive.** Reverse of FN is when we falsely predict the output to be positive while it should have been negative, leading to the case of *false positive* (FP).

**Example 2.6.** Let us map true positives, true negatives, false positives, and false negatives when y = [1, 1, –1, 1, –1, –1, 1, 1, 1, 1] and *ŷ* = [1, –1, 1, 1, –1, 1, 1, 1, 1, –1]. Further, based on these counts, we can produce a confusion matrix.

In Table 2.9, we enlist the type of correct/incorrect information captured by the *ith* index. We can see that TP occurs when *yi = ŷi* = 1 and TN at *yi = ŷi = –*1. Meanwhile, at indices 2 and 10, we observe the case of *yi* = 1 but *ŷi =* –1, causing false negatives. Finally, at indices 3 and 6, we note *ŷi =* –1 but *ŷi* = 1, leading to false positives.

Now, mapping the type count in Table 2.9, we can construct the confusion matrix for the four cases as accounted in Table 2.10.

**Precision.** Looking only at the predictions that are marked as positive, precision measures the number of times the predictions were actually correct, as actualised by Equation (2.28).

 (2.28)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **Expected** *y* | 1 | 1 | -1 | 1 | -1 | -1 | 1 | 1 | 1 | 1 |
| **Predicted** *ŷ* | 1 | -1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | -1 |
| **Type** | TP | FN | FP | TP | TN | FP | TP | TP | TP | FN |

Table 2.9: Mapping True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) for Expected Labels y = [1, 1, –1, 1, –1, –1, 1, 1, 1, 1] and Predicted Labels *ŷ* = [1, –1, 1, 1, –1, 1, 1, 1, 1, –1].

|  |  |  |
| --- | --- | --- |
| Actual | Predicted | |
| Positive | Negative |
| Positive | 5 (TP) | 2 (FN) |
| Negative | 2 (FP) | 1 (TN) |

Table 2.10: Confusion matrix for sentiment classification of positive (1) and negative (-1) sentiments for ten sentences. We construct this from expected labels *y* = [1, 1, –1, 1, –1, –1, 1, 1, 1, 1] and predicted labels *ŷ* = [1, –1, 1, 1, –1, 1, 1, 1, 1, –1]. The tabulations follow from mapping in Table 2.9.

**Recall.** On the other hand, looking at the actual/expected positive samples, recall measures the number of times we correctly predicted the positive class. The confusion matrix can be mapped using Equation (2.29).

 (2.29)

**Precision vs Recall.** We note from Equations (2.28) and (2.29) that the main difference in precision and recall is dictated by the type of erroneous outputs that are accounted for. In the case of precision, we place higher importance on FPs. Consider the case of spam detection. If the emails keep getting falsely classified as safe/positive, then the user will be inundated with spam instead of useful information. Meanwhile, in the case of a recall, we place higher importance on FNs. Consider the case of medical testing, where a positive test means a disease is detected. Failing to detect the disease (FN when it should have been positive) can cost human life. In any given experimental setup, precision and recall will be a tug-of-war, as reducing FN can impact FP and vice-versa, and which metric is prioritised depends on the task at hand.

**F1 Score.** For most use cases, we rather prefer to look at a single metric that considers both precision and recall. Here, the F1 score comes into play. It is simply a harmonic mean of precision and recall as follows:

Another advantage of the F1 score over other metrics is its ability to account for class imbalance, therefore providing a more holistic measure of model performance.

**2.13 Summary**

In this chapter, we explored some of the fundamental concepts of NLP and neural networks necessary for understanding the more advanced topics covered later in the book. We began by discussing the motivation behind processing information conveyed through natural language, focusing on how a word is structured using morphological knowledge. We then reviewed the essential steps of the NLP pipeline and examined various preprocessing techniques such as stemming, lemmatisation, and tokenisation. Additionally, we explored the syntax and semantics of language before introducing core ideas related to language models and word/sentence representation techniques.

To motivate the use of *n*-dimensional feature vectors for sentiment analysis, we introduced the concept of neural networks. Given that neural networks are the foundation of modern NLP, this chapter provided an overview of the fundamental aspects of neural networks. We discussed perceptrons and their limitations, which led to the development of multi-layer perceptrons and the concept of deep neural networks. The chapter also covered training neural networks via backpropagation, the basics of activation functions, and the role of various hyperparameters that can impact the training process. Furthermore, we outlined scenarios where a model might encounter vanishing or exploding gradient problems and how these issues can be mitigated. The chapter concluded with an introduction to evaluation metrics commonly used in classification tasks.

In the following chapters, we will build upon the concepts of word associations, neural networks, and *n*-grams to develop more sophisticated representations and language models that go beyond the bag-of-words approach.

**Additional Resources**

**Important Articles**

* A Survey of Surveys (NLP & ML): <https://github.com/NiuTrans/ABigSurvey>.
* Awesome NLP: https://github.com/keon/awesome-nlp.
* Introduction to Linguistics: Akmajian et al. (2001)
* Mielke, Sabrina J., et al. “Between Words and Characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP.” arXiv preprint arXiv:2112.10508 (2021).
* Min, Bonan, et al. “Recent advances in natural language processing via large pre-trained language models: A survey.” ACM Computing Surveys 56.2 (2023): 1-40.
* Otter, Daniel **W**., et al. “A survey of the usages of deep learning for natural language processing.” IEEE transactions on neural networks and learning systems 32.2 (2020): 604-624.
* Pattern Recognition and Machine Learning: Bishop (2006)

**Visual Summary**

* Dependency Parsing, Named Entity Recognition, Tokenization and Token Similarity  
  https://huggingface.co/spaces/spacy/pipeline-visualizer#en\_core\_web\_lg
* Deep Neural Network Architecture: <https://playground.tensorflow.org/>
* Optimization with Gradient Descent  
  https://uclaacm.github.io/gradient-descent-visualiser/#playground

**Exercises**

***True/False Questions***

1. Lemmatisation is more computationally expensive than stemming. (True/False)
2. The sigmoid activation function outputs a value between -1 and 1. (True/False)
3. SentencePiece does not require the input sequence to be pre-tokenised. (True/False)
4. Multiplying the output of a linear unit with a scalar can introduce non-linearity. (True/False)
5. Dependency parsing focuses on identifying relationships between words based on the order in which they appear in a sentence. (True/False)

***Multiple Choice Questions***

1. In dependency parsing, the \_\_\_\_\_\_\_\_\_\_ is the main verb of the sentence.

(a) Prime (b) Root (c) Lemma (d) Stem

2. The study of the internal structure of words is called \_\_\_\_\_\_\_\_\_\_.

(a) Etymology (b) Sociolinguistics (c) Morphology (d) Phonology

3. In gradient descent, what is updated out of the following?

(a) Parameters (b) Inputs (c) Architecture (d) Activations

4. Dependency parsing helps in understanding the \_\_\_\_\_\_\_\_\_\_ structure of a sentence.

(a) Syntactic (b) Semantic (c) Pragmatic (d) Morphological

5. Which of the following introduces non-linearity into a neural model?

(a) Weight Sharing (b) Gradient Descent (c) Convolution (d) GELU Activation

***Short Questions***

1. How does stemming reduce the dimensionality of textual data?
2. What is the difference between stemming and lemmatisation?
3. Compare the ReLU and sigmoid activation functions. Which one of them is used in (i) the hidden layers and (ii) the output layer? Why?
4. Consider the following sentences. Try to trace an NLP pipeline that consists of tokenisation, POS tagging, lemmatisation and dependency parsing on each sentence:
5. I am eating pizza with cheese and corn.
6. My mother cooked my favourite dish for me on my birthday!
7. I am at the airport, and my flight departs in an hour.
8. He loves to bake cookies for his friends and family.

***Long Questions***

1. Compare and contrast various tokenisation strategies discussed in this chapter.
2. Explain how text preprocessing techniques impact the performance of NLP models.
3. Calculate the output of a three-input neuron where the weights [*w1,w2,w3, b*]are [0*.*3*,* –0*.*1*,* 0*.*2*,* 0*.*5]. The input to this network is [0*.*3*,* 0*.*2*,* 0*.*6]. Assume the sigmoid activation function.
4. What is WordNet? Explain the structure and applications of WordNet.
5. Why do we need subword tokenisation? Give an example where word tokenisation fails.
6. Describe the typical stages involved in a natural language processing pipeline. Explain the significance of each stage by taking some NLP tasks as an example.

Consider a simple neural network with one layer and sigmoid activation where **ŷ** = *σ*(**w***T***x** + *b*) and *σ(z) =* 1*/*(1 + exp(–*z*))*.* Compute the gradients of the loss function with respect to the parameters and derive the weight update rule for gradient descent.

**Bibliography**

Akmajian, A., Demers, R. A., Farmer, A. K., and Harnish, R. M. (2001). *Linguistics: An Introduction to Language and Communication*. MIT Press.  
URL https://doi.org/10.7551/mitpress/4252.001.0001

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (pp. 1724–1734).  
URL https://aclanthology.org/D14-1179

Clark, H. H. (1970). Word Associations and Linguistic Theory. *New Horizons in Linguistics*, *1*, 271–286.

Eisenstein, J. (2019). *Introduction to Natural Language Processing*. MIT Press.

Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, *14*(2), 179–211.  
URL https://www.sciencedirect.com/science/article/pii/036402139090002E

Gage, P. (1994). A New Algorithm for Data Compression. *The C Users Journal*, *12*(2), 23–38.

Gers, F. A., Schmidhuber, J. A., and Cummins, F. A. (2000). Learning to forget: Continual prediction with lstm. *Neural Comput.*, *12*(10), 2451–2471.

1. https://www.ling.upenn.edu/courses/Fall\_2003/ling001/penn\_treebank\_pos.html [↑](#footnote-ref-1)
2. https://huggingface.co/ [↑](#footnote-ref-2)
3. https://tartarus.org/martin/PorterStemmer/ [↑](#footnote-ref-3)
4. https://snowballstem.org/ [↑](#footnote-ref-4)
5. https://darenr.github.io/afinn/ [↑](#footnote-ref-5)
6. https://www.nltk.org/api/nltk.corpus.reader.sentiwordnet.html [↑](#footnote-ref-6)
7. https://saifmohammad.com/WebPages/NRC-Emotion-Lexicon.htm [↑](#footnote-ref-7)
8. https://propbank.github.io/ [↑](#footnote-ref-8)
9. The squared L2 norm (i.e.,  is often used in optimisation problems because it avoids the computational cost of the square root while still preserving the essential properties of the norm, like magnitude comparison. [↑](#footnote-ref-9)