Half Title

Title Page

LOC Page

To my dog
and my cat.

Contents

Part I

This is What a Part Would Look Like

1

Natural Language Processing

Author Name

CONTENTS

In this chapter we introduce modern NLP libraries, techniques and their applications. This chapter will focus on deep learning methods and less on computational linguistics that require nuanced knowledge of linguistics. We explore what it means to represent words and sequences of words with rich numeric representations that are better-suited toward modern computational tasks. We aim to capture some of these modern fine-tuned representations that are specially catered toward a semantic lexicon for medical language. We use these representations and aforementioned tools to showcase a modern reference implementation leveraging PyTorch, PyTorch Lightning and the Huggingface Transformers library. To wrap it all together, we walk through a complete example that highlights best practices that encourage reproducibility and allow for systematic iterative improvements.

This includes:

- An introduction into fundamental NLP concepts
- Bootstrapping techniques to iterate on a dataset in the low-resource setting
- Storing of a reference dataset in a publicly-accessible location
- Downloading, caching, loading, splitting, and preprocessing of the data
- Monitoring the training run:

  Logging and experiment tracking

  Learning curves

  Metrics
- Hyperparameter tuning, some tricks of the trade
- Offline evaluation and sanity checking

We will keep the discussion focused on tasks in epilepsy, using a running example of SUDEP prediction from electronic medical record (EMR) notes. Many of the concepts introduced here are very general and are straightforward translations to domains outside of SUDEP prediction, epilepsy, and even NLP.

1.1 Introduction to Natural Language Processing

Natural language processing (NLP) is a field of computer science that deals with the extraction, processing, and understanding of human language. It is known as the field of computer linguistics, and is a subfield of artificial intelligence. Common NLP tasks include sentence segmentation, tokenization, part-of-speech tagging, named-entity recognition, parsing, question answering, summarization and classification.

How can we teach a computer to perform these tasks? The first challenge is that computers, at their core, only understand numbers. So first we need to think about how words, or more generally text, can be represented with numbers such that we can perform calculations on them. The numbers should allow the computer to assign meaning to words and their context with other words around it.

There is no perfect way to represent language in this numeric fashion. The best we can do is to capture representations that we feel capture the *syntactic* features of text, as well as *semantic* features of text. Syntactic information refers to grammatical constructs and the way that words are put together in a language. Semantic information refers to the meaning of this body of text. Such distinctions are fundamental concepts in computer science and date back to the very foundations of information theory. [**?**]. Much of this chapter, as well as much of the current work in the field of computational linguistics is centered around finding better and better representations that capture both types of this information.

How do we begin to choose these representations? Do we choose to represent words, or perhaps individual letters? Do we choose all the words? What do we do with punctuation and numbers?

1.1.1 Vocabularies and Words

In NLP, the set of unique words in a corpus is called a *vocabulary*. While *the* is the most common word in the English language, it is only one entry in a very large vocabulary table. A rare word like *hemispherectomy* is also one entry in this table. Both *apple* and *apples* might be separate entries in our vocabulary table. Generally the first step in many NLP systems is defining this vocabulary and the rules behind which words are in this table and which words are not.

The process of splitting text into separate smaller units, in this case words, is known as *tokenization*. When we run many NLP tasks, we almost always preprocess our text by putting it into one of these tokenizers. The output tokens, in this case words, are the indexes in our lookup table to different numeric representations that can be understood by a machine. There are a

number of great tokenization libraries that are open source, and you should always use them rather than try to implement this yourself.

```
# Example 1: Spacy Tokenization Using Medical Words

import spacy
sample = "After a temporal lobe resection, the atonic " \
         "and clonic seizure frequency fell by 50%."

nlp = spacy.load("en_core_web_sm")
english_vocab = set(nlp.vocab.strings)
spacy_document = nlp(sample)

for token in spacy_document:
    found = "found" if token.text in english_vocab else "NOT FOUND"
    print("{}: {}".format(token.text, found))
```

The discrete numeric entities created in Example 1 are the indexes for our representations. The tokenizer system did much of the magic under the hood, lowercasing and standardizing text and creating special tokens for punctuation and numbers.

When we build a vocabulary V, we define a preset vocabulary size $|V|$. There is a tradeoff in your choice of $|V|$. If you choose a number that is too small, then your system will only recognize a few words and lose a lot of important task-specific vocaublary. However, if you include the entire english language, your system will be slow, expensive, and not perform as well. Commonly this number is set around 20,000 - 40,000 words in English, though it can be much larger.

Often a vocabulary will include special tokens that make our lives easier when working with NLP tasks. Common special tokens include:

- `[EOS]` - An end of sentence/input marker.
- `[PAD]` - Special inputs to ignore, usually following an EOS marker.
- `[SEP]` - A separator, which can be used in inputs that contain multiple sentences or samples.
- `[OOV]` - Out of vocabulary, also commonly represented as [UNK] (unknown), which is used when we come across a word that does not exist in our vocabulary.

The out-of-vocabulary token is of particular importance in medicine. If we load a vocabulary with 30,000 words, often these are roughly the most common 30,000 words in the English language. We will still have a few [OOV] tokens for rare words. However, just because a general system chooses its vocabulary based on word frequency does not mean that this is the best choice of vocabulary for the task at hand. Medical corpora have a very different

word frequency distribution than non-medical corpora, particulary because of nuanced vocabulary that is absolutely crucial.

In Example 1 above, notice the importance of the words that have been thrown away. If this sentence were to be loaded into a machine as-is, we might be losing far too much information because we have to nearly all of the relevant terminology with OOV tokens. We will address this issue in the ensuing sections. For now, suffice it to say that we should make sure our vocabularies are catered toward our task at hand or are built in such a way that they can still extract signal from these tokens. Furthermore, we should always sanity check our sample token outputs against our raw text inputs.

1.1.2 Word Representations

Given that we have defined a vocabulary V of words, these are the keys to our lookup table of numeric representations. But what do we store in that table as the value?

A simple way to achieve this representation for V is to assign each word a unique integer i. The word is then represented as a vector w of length $|V|$ with all zeros and a one[1]. at index i.

```
have = [1, 0, 0, 0, 0, 0, ... 0]
a    = [0, 1, 0, 0, 0, 0, ... 0]
good = [0, 0, 1, 0, 0, 0, ... 0]
day  = [0, 0, 0, 1, 0, 0, ... 0]
...
```

We could now represent a sentence as the sum of the word vectors $S = \sum_j w_j$. This representation is suitable as input for any classification algorithm (such as logistic regression or a decision tree), to make a prediction about our target variable, e.g. whether the sentence is relevant to our epilepsy task. This simple representation fulfills our requirements but comes with some drawbacks:

- We implicitly assume that each word in the sentence is equally important.
- Each word is equally similar to every other word (e.g. by taking the euclidian distance between word vectors).
- The representation is invariant to a reordering of the words.

The first point is a problem, because as we add more word vectors together, the sentence reperesentation will converge to the global average and drown out any signal relevant to the specific sentence at hand. To address this we can instead write a weighted sum $S = \sum_j \lambda_j w_j$, where each word vector is weighted by its importance λ_j. There are statistical methods we can use to

[1]This is also known as a one-hot encoding

compute an importance value[2]. For example a word like *the* is very common and appears in many different contexts. It is unlikely that there is a lot of signal we can extract from it. On the other hand, a word like *epilepsy* will be much more rare and specific to a given task, and we would like to raise its importance. This has the net effect of allowing us to find good representations for larger word sequences.

To address the second point, we will be touching on a very important concept, not just relevant in NLP, but also in the world of ML in general: Embeddings.

1.1.3 Embeddings

An embedding is a representation of a vector space that captures the similarity between the entities we are encoding, where entities can be words, images, e-commerce products, movies, houses, and many other data modalities. Scientific progress in the field of ML is often directly about finding algorithms that can learn high-quality robust embeddings in an efficient and scalable way. The final performance of a given architecture is often largely determined by the quality of the embeddings it learns.

To illustrate this concept, we will be talking briefly about one of the earliest algorithms that has been used to learn embeddings for words: *word2vec* [**?**], a technique pioneered by a team led by Thomas Mikolov. The idea behind *word2vec* is to set a word into context with the words surrounding it. As it turns out, this idea that similar words appear in similar contexts, is an extremely powerful signal for achieving quality word embeddings. In NLP, the study of techniques that determine the probability of a given sequence of words is known as language modeling.

For example the word *epilepsy* may appear in the context of words like *symptom*, *medication*, or *episode* and vice versa. However, we would not expect that *epilepsy* would appear in the context of words like *volleyball*.

We then translate this idea into a training task: Given the sum of the one-hot encoded word vectors of the words surrounding the target word, map it to the one-hot encoded vector of the target word.

This is a task that can be solved by a neural network with the following setup: Add a layer l_1 that maps from the input dimension (of size $|V|$) to an intermediate dimension of size h and then another layer l_2 that maps from the intermediate dimension to the output dimension (of size $|V|$). Here h is what is called embedding dimension and $h << |V|$. While h is generally a hyperparameter that can be optimized at a later stage, a reasonable choice is to use a value according to the rule of thumb:

$$h \approx 4\sqrt[4]{|V|} \tag{1.1}$$

For a vocabulary size $|V| = 30000$ this comes to a value of about 50. During

[2] TF-IDF

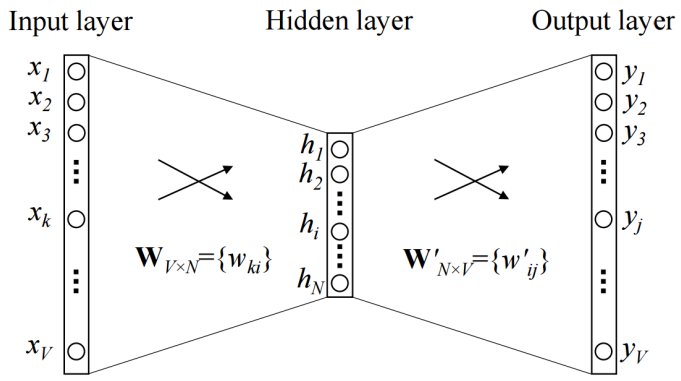

Figure 1: A simple CBOW model with only one word in the context

FIGURE 1.1
Word2vec

training, the network will learn to encode the original sparse indicator vectors in a way that preserves the maximum amount of information necessary to predict the target word, while being forced to squeeze the information through the low dimensional bottleneck. An interesting side effect of this approach is that we can use part of the network to encode a given word in V by using for example the inverse of l_2 as a lookup table. The vectors that we obtain from this lookup table are called word embeddings and they have some very useful properties:

- Similar words tend to be close together (e.g. in terms of euclidian distance).
- Averaging word embeddings of a sentence will give us a more robust representation than our naive approach.
- We can perform some arithmetic, such as adding and subtracting the embedding vectors to traverse the space.

These properties also imply that the represention for two sentences will be close together if they are semantically similar, which will make the job of, say, a downstream classifier much more straightforward - it only has to slice the embedding space. The hard part of understanding the meaning of the sentence is already done. Since the training data can be generated from just raw text, it is sufficient to train the embeddings once on a very large corpus[3] and then reuse them, in effect giving us a headstart when building task-specific models. Instead of also having to learn what words mean, and some of the syntactic

[3] GloVe

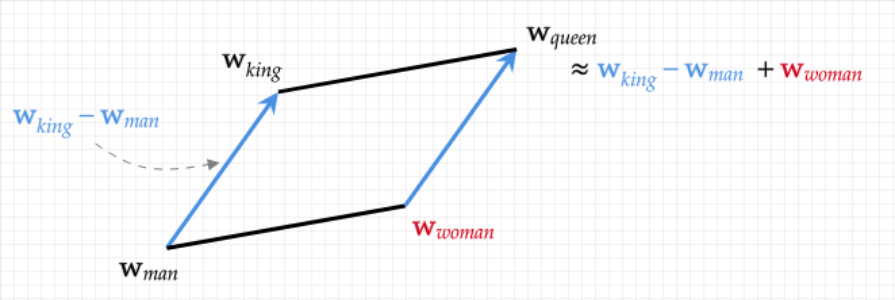

FIGURE 1.2
Word2vec

rules behind written language, we can get a head start on this and focus our computational resources on adapting this knowledge to a specific task.

Using these pre-trained text representations is often critical. Without doing so, the majority of what a machine learns is the basic syntax and semantics of language. By using these pre-trained word embeddings, we allow our models to learn to focus more on the downstream task. By simplifying the learning task with these pre-trained embeddings, we need to use far fewer labeled examples than we would training from scratch.

This underlying assumption that similar neighboring words share a semantic similarity to center words is powerful, and at the heart of many modern training objectives for embeddings that are used in many downstream ML systems. We do not inherently inject a great deal of model bias, but rather rely on having a large enough training set that the word vectors will become meaningful representations with respect to one another. A corpus such as Common Crawl[4], which contains roughly 840 billion tokens of over two million unique words, provides such scale. There are other corpora used in these unsupervised processes, one such common candidate being a collection of all Wikipedia articles.

In practice, one training objective can be to use the distributed representation of a set of context words to guess a center word. This is known as the continuous bag of words (CBOW) model.

Consider a training document describing patients with epilepsy, where we are attempting to train a model with a predefined vocabulary V of the top For a vocabulary V, where we have chosen the top $|V|$ most common words in English to learn. The document being chosen in this training iteration may contain the following excerpt:

```
... recommended administering Lamictal to treat their
focal seizures, the patient continued experiencing
symptoms though reported a 50% reduction in ...
```

[4]Common Crawl Link

In training our word embeddings, we will randomly mask one of the words, in this case *patient*. Our model makes use of an additional parameter, a context window size c, that helps to provide necessary context. If we choose $c = 8$, our training sample would become

```
... lamictal to treat their focal seizures, the [MASK]
continued experiencing symptoms though reported a 50 % ...
```

For a given center word at index i, we consider previous words at indexes $i-1$, $i-2$, ..., $i-c$, and all subsequent words at indexes $i+1$, $i+2$, ..., $i+c$. In our word sequence S, we may then choose to sum the word vectors of each of these context words. Our context input can be written as:

$$\sum_{\substack{j=-c \\ j\neq 0}}^{c} w_{i+j} \tag{1.2}$$

Our task is then to predict a probability distribution over the size of our vocabulary $|V|$, where the target is a vector of all 0's with a 1 at the index in the vocabulary for the word *patient*.

There are a few additional takeaways from this example. Note that we will address many of these in the sections to follow.

- The training process will likely consider this text many times, and mask out different words in each iteration.
- Our processed sentence is slightly modified from our original input. Notice that the number and percent symbol are split, and that Lamictal is lowercased.
- The word *patient* can also be an adjective. Our *patient* embedding will also capture this meaning in our model, dependent upon how often it is used with that meaning.
- If our context window is too small, we lose information. If it is too large, our signal becomes fuzzy as we converge to the global mean vector.

What if we choose to use only a center word, and have our model predict all of the surrounding context words? This is a common training technique for embeddings as well, and is known as the Skip-Gram model.

1.1.3.1 Spacy Example of Word Embeddings

```
import spacy

sample_words = ["epilepsy", "seizure", "patient", "language"]

nlp = spacy.load("en_core_web_md")
docs = [nlp(sample) for sample in sample_words]

#Spacy's Medium English Model Stores Words As Vectors of Dimension 300
#Note each word has this vector size
for doc in docs:
    assert doc.vector.shape[0] == 300

#The cosine distance, ranging from [0,1], of the vectors tells us how sim
print(docs[0].similarity(docs[1])) #epilepsy and seizure -> 0.9999
print(docs[0].similarity(docs[2])) #epilepsy and patient -> 0.4434
print(docs[0].similarity(docs[3])) #epilepsy and language -> 0.1273
```

1.1.3.2 Character and Subword Embeddings

In Example 1, we covered the issue of missing vocabulary that is frequently encountered when using word-level tokenization. However, there are other strategies that are commonly used that help to avoid this issue. One such simple way around out-of-vocabulary tokens is to use character-level tokenization. Instead of having a vocabulary that is tens of thousands of words long, you only have a vocabulary that contains the characters in your language, with the addition of punctuation, digits, and a few special tokens for sentence and word marking. As long as the tokenizer is able to mark which characters start a word and which characters are continuations of a word, then the original sequence of words can always be recovered.

Unfortunately, the tradeoff for a small vocabulary size with no unknown word entries is that the length of tokens to represent a sentence is clearly much larger. Empirically, these models do not perform as well, partially because of this sequence length but also because of the loss of information that is represented in word vectors.

In practice, the best models are often somewhere in the middle, in what is known as word-piece, or subword, vocabularies. These tokenizers and vocabularies contain a rich set of common words, but also have subword units that can be used to piece together out-of-vocabulary words. These subword units behave like normal word vectors and can store semantic and syntactic information.

```
# Example 3 - Wordpiece Tokenizers

from transformers import BertTokenizer
```

```
sample = "After a temporal lobe resection, the atonic " \
         "and clonic seizure frequency fell by 50%."

bert_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

encoded_ids= bert_tokenizer.encode(sample)
encoded_tokens = bert_tokenizer.convert_ids_to_tokens(encoded_ids)

for token, id  in zip(encoded_tokens, encoded_ids):
  print("{}: {}".format(token,id))
```

In example two,

1.1.4 Transformers

There have been significant advancements in ML due to a variety of factors, such as the availability of larger and larger datasets, as well as ever increasing computational power at lower and lower prices, better software tools, libraries and model architectures. Especially the advent of deep learning (DL), fueled by the emergence of GPUs from about 2012, enabled scaling to many orders of magnitude larger datasets and models. The common wisdom before DL was that as the number of parameters in a model increases beyond a certain threshold, it tends to overfit the training data, leading to poor generalization when deployed in the real world. Deep Learning based models empirically do not seem to exhibit this behavior, and instead tend gain performance as they get bigger, albeit more slowly.

Until recently, building strong NLP systems required deep understanding of language and its structure, as well as large amounts of data, even when relying on pre-trained word embeddings - the resulting systems were often still brittle.

In recent years some surprisingly powerful architectures and pre-training regimes have emerged that build on the concepts we have discussed here, most notably under the name of *Transformers*[5], that address the last bullet point of our representation's shortcomings.

The details of the inner workings are beyond the scope of this tutorial, for now it suffices to know that they are able to learn robust sentence embeddings with a fine grained understanding of syntactic structure, semantics, and general knowledge of the world. In fact, they can be considered near the level of a human that just knows the English language (or other languages), along with a broad array of factual knowledge about the world (think Wikipedia). Just like a human, they can be taught specialty knowledge that is relevant to a given task, in ML we would say we *finetune* them. This paradigm shift has led to a step function improvement in terms of performance and sample

[5] BERT, GPT

efficiency in a wide variety of NLP tasks where Transformers are the defacto state of the art.

In the following sections, we will discuss a broad set of problem statements Transformers are suitable for, along with a set of techniques a practitioner can employ to arrive at robust solutions, even with limited data.

1.1.4.1 Pre-training

Modern DL-based systems, such as the aforementioned Transformer architectures, are pre-trained on gigantic datasets[6][?] and can be downloaded for free[7]. Starting with a model pre-trained on a broad set of topics significantly reduces the amount of task specific training data required to achieve a given performance. Furthermore, it may be helpful to start with a model that is pre-trained on more domain-specific datasets such as `BioBERT[?]` or `Bio_ClinicalBERT[?]`, or pre-train your model from scratch.

1.2 NLP Tasks in Epilepsy

Now that we have an understanding of some of the basics of computational representations of language data, let us take a look at some of the modern use cases for NLP in industry, catered toward epileptologists and other medical practitioners.

1.2.1 Unstructured Text - Retrospective Research

Within the medical field, there are many sources of unstructured text that are primarily aimed as a either a communication channel between specialists, or a way for patients to pass information that isn't captured in a structured format. Clinical Notes, Progress Notes, Operative Notes, Discharge Summaries, Pathology Reports, Surgery Reports; all of these sources carry information, but traditionally require experts in the field to extract it. If we can teach systems to read through this unstructured text and perform respectably close to these experts, we can perform large-scale clinical retrospective research much faster and at a much lower cost.

Because these pre-trained language models can draw upon information from so many domains and have a sound understanding of language, they need relatively few labeled examples to begin to perform tasks such as classifying text. All they have to learn is the task at hand, not the nuances of languages. If a team of experts is able to identify a set of labels in which they are interested, they can label this data in a single shot. The next step is to fine-tune a pre-

[6] BooksCorpus (800M words, Wikipedia 2,500M words)
[7] Huggingface

trained language model, often by just attaching a logistic regression layer to the outputs of the language model, to recognize each of these labels.

Instead of having to wait for, not to mention pay for, experts to do large-scale retrospective research, fine-tuned pre-trained language models can perform admirably on text classification with, in many cases, only a few hundred labeled examples. The trained model can then read through and extract targeted text much faster and much cheaper. Studies have already been done applying this strategy[**?**], using three pre-trained neural language models, $\text{Bio}_{C}linicalBERT, to extract seizure frequency from epilepsy clinical notes.$

1.2.2 Unstructured Text - Quality of Life

When treating patients with epilepsy, often our principal aim is to target improving quality of life. Seizure frequency is a numeric represenation that has a measurement that is well-understood and directly impacts quality of life. However, many measurements of quality of life come directly from patient surveys and other more subjective metrics that can be more difficult to interpret at scale. While often these surveys call for responses on numeric scales, many offer unstructured text as a method for patients to freely communicate other information relevant to quality of life, or further explain their responses. This information is of exceptional importance, but it can be difficult to parse through it at scale without modern natural language processing techniques.

Sentiment analysis, the task of extracting and studying subjective sentiment in text and speech data, is yet another field of NLP that has greatly benefited from the widespread adoption of transformer architectures, particularly in specialized domains. Freeform responses that can be evaluated as positive or negative, or sometimes into more granular sentiments, can be learned with machine learning methods. It's worth noting that naive methods on sentiment tend to capture some cases well. The words love and hate are strong signals, but even their simple negation becomes a diluted signal with possible interpretations of sarcasm that become subjective. Add in the many colorful ways that people write, and we start to see why rules-based text-mining systems for sentiment are so difficult.

Sentiment analysis models using transformer architectures such as RoBERTa aim to project the large-dimensional output vector of the text, and project this semantic representation into a basic sentiment of two class, positive or negative. Very often a third class is used to represent neutral sentiment. This method should be used when analyzing free-form responses in patient surveys using NLP models trained on sentiment. While a model trained on medical data is ideal, it's quite impressive how well a model trained on movie reviews will transfer over to other sentiment tasks.

The same methods for analyzing basic sentiment can be used to label tones, more granular emotional classifications, or other task-specific points of interest. Similar to the case with the retrospective research in section 1.2.1, it is important to have the survey designers, and/or other domain experts, label

these targeted semantic expressions of their patients. Once a model is trained with a set of positive examples of these targets, it can be used on any text response.

The datasets provided by SeizureTracker, as part of their It's Not Just Seizures (INJS) initiative, provides an excellent example of unstructured text responses being used as a guide to a suggested set of useful labels that can be learned with NLP techniques. This is outlined in section 1.5

1.2.3 Text Preprocessing and Tokenization

Placeholder.

1.2.4 Named Entity Recognition and PII

Named Entity Recognition (NER) is a subtask of information extraction that classifies unstructured text into personal names, locations, time indications, organizations, and other tags. NER is an important problem in medicine, if nothing else to provide an ability to mask out Personally-Identifiable Information. If a model, probably a transformer architecture in practice, is able to identify text as a first name, a home address, or a credit card number, then we can anonymize this information to mask information that is PII-specific.

It should be noted that, although some of this text is easy to catch with rule-based approaches and regular expressions, such as a credit card number, other information requires contextual understanding and an ML solution to achieve the necessary performance.

Spacy, NLTK, and Stanford NLP all provide out-of-the-box NER models that perform well on unstructured text. Let's look at a Spacy example in practice.

```
# Example 4 - Simple PII Anonymizer

import spacy

sample = "The patient, John Smith, began feeling symptoms at his home in
nlp = spacy.load("en_core_web_md")

doc = nlp(sample)

for word in doc.ents:
    sample = sample.replace(word.text, word.label_)

print(sample)

#The patient, PERSON, began feeling symptoms at his home in GPE, and was
```

1.2.5 Mining Unstructured Text and Clinical Documentation

This is placeholder text. Talk about clinical documentation and EHR notes, and other forms of unstructred text, being used for downstream classification tasks. Then, give some examples of them, and what these people did, and possibly how transformers could probably improve this work greatly.

1.2.6 Clinical Trial Matching and Eligibility Prescreening

This is a placeholder text. https://www.nature.com/articles/d41586-019-02871-3

1.2.7 Surgical Candidacy

This is placeholder text

1.2.8 Avoiding Epilepsy Misdiagnosis

This is placeholder text. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4419916/

1.3 Research Process

Part of any research process is rigorous record keeping to ensure experiments can be independently and reliably reproduced. While researchers keep meticulous journals, when dealing with software this bookkeeping can be largely automated.

1.3.1 Version Control

Software engineering primarily relies on version control systems, such as GitHub or BitBucket, to keep track of code changes.

....

1.3.2 Experiment Tracking

An experiment is defined by the training code, data, and configuration. When ML and Data Science are involved, the data generating process has to be documented. In software engineering we say that code doesn't exist until it is "checked into" version control. Similarly, data doesn't exist until it is stored in the cloud. In addition to the raw data, we should also store documentation about how it was created, by whom, when, and any additional processing steps. This should be done in a way that is easy to replicate and understand – ideally directly in code.

Version control system are specialized to keep track changes in plain text files, but they are not well suited for tracking the other aspects of an experiment. The ML community has developed specialized tools[8], which we have integrated into Sheepy. These tools are excellent at visualizing and comparing of

- Hyperparameters
- Model metrics
- (Interactive) plots
- Logs
- Hardware configuration
- Hardware metrics
- Code and dependency versions
- Artifacts (raw data, processed data, models)

and facilitate collaboration and sharing of results by directly linking to the experiments.

At the beginning of a new training run an experiment is initialized and data related to the run streamed to a web based platform. The next step is obtaining data artifacts from a cloud storage provider[9], preprocessing, training and finally storing and tracking of output artifacts and metadata. Metadata can include information about how many samples the data contains, the size and type of fields, input features and outputs of a model, when it was created etc.. This facilitates discovery, analysis, iteration, and collaboration.

1.3.3 Data Preparation

We will go over each step of data preparation in the next sections:

- Data download
- Pre-processing

[8] Weights&Biases, Neptune
[9] Amazon S3, Google Cloud Storage

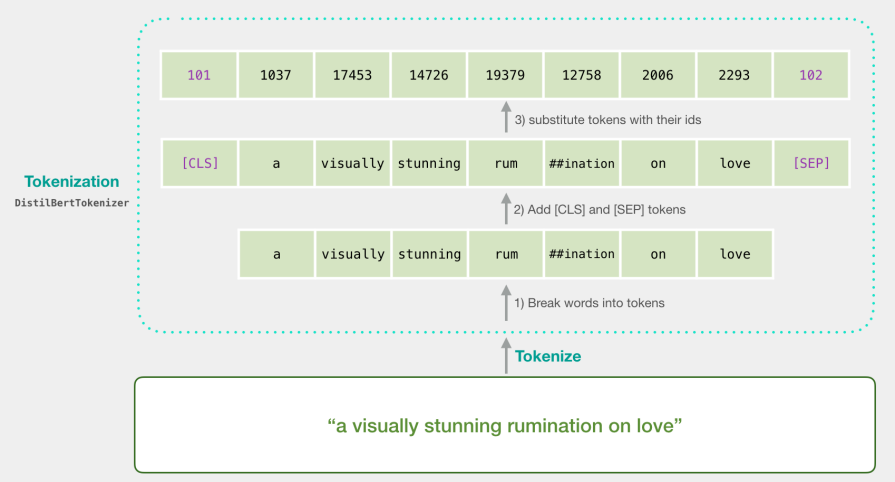

FIGURE 1.3
Tokenization

- Splitting into training, validation, and test sets
- Feeding data to the model during training

The goal is to rerun an experiment from scratch with a single command and we build on top of PyTorch Lightning's `DataModule` class to handle these steps.

1.3.3.1 Data Download

Downloading the dataset from cloud storage and caching are handled in the `prepare_data` method and only executed once per run. We will first check if the data is already cached, and if not, download it.

1.3.3.2 Data Preprocessing

Pre-processing refers to transformation steps we want to apply to the raw data to make it suitable to input into the model. For example in NLP, raw text needs to be cleaned, tokenized, trimmed and padded (see Section 1.1.1) (Fig. 1.3). These steps can be slow and in principle we need to do them only once per sample and cache the results.

1.3.3.3 Data Splitting

To evaluate the generalization performance of a model we need to test it on data it hasn't seen during training. The dataset is therefore split into training and validation sets, usually at random (Fig. 1.4). It should be ensured the randomness is deterministic to reproduce results exactly, which can be achieved

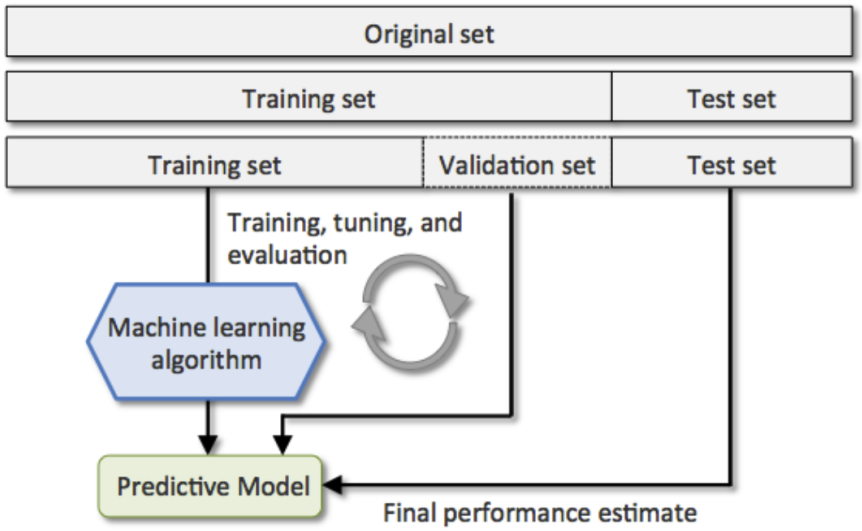

FIGURE 1.4
Data Splitting

by setting a "seed" for the random number generator. The validation set is used during training to monitor progress and tune hyperparameters. In addition, it is good practice to hold out another portion of the data, the test set, which is never used during training, but only to report final results. We otherwise run the risk of tuning hyperparameters to the validation data, thereby overestimating real performance. While datasets are continuously evolving, we should keep a frozen version of the data to get fair comparisons.

- Training dataset: Tune model parameters θ (80% of the data)
- Validation dataset: Tune hyperparameters (10% of the data)
- Testing dataset: Evaluate and report the performance of the model (10% of the data)

1.3.3.4 Feeding Data to the Model

Model parameters are stored in GPU memory (known as VRAM); in order to update them, the GPU needs to process data from the training set. After downloading the data from cloud storage it resides on local disk, from where it needs to be moved to RAM and finally VRAM. Loading data into the model needs to be fast, or we run the risk of starving the GPU, meaning the GPU processes each batch of data faster than the CPU can deliver the next one, leading to low resource utilization and slow training. We use PyTorch's

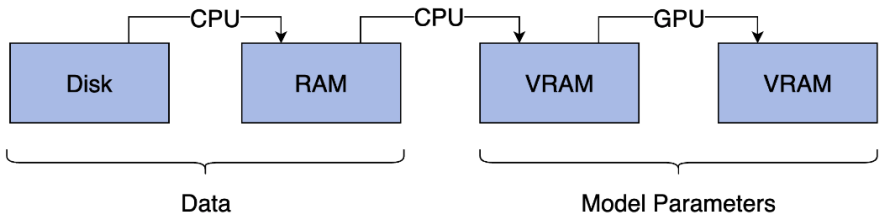

FIGURE 1.5
Data needs to be moved from disk to RAM to VRAM so that it can be used to update the model parameters.

Loss Name	Function	Use
Mean Squared Error	$\frac{1}{N}\sum_{i=1}^{N}\|\hat{y}_i - y_i\|^2$	regression
Binary Cross Entropy	$-\frac{1}{N}\sum_{i=1}^{N}(y_i \cdot log(\hat{y}_i) + (1-y_i) \cdot log(1-\hat{y}_i))$	binary classification
Cross Entropy	$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(\hat{y}_i)$	multiclass classification

TABLE 1.1
Loss functions

`DataLoader` class to parallelize this process across multiple CPU workers. A `DataLoader` takes as argument a `Dataset` and provides a stream of batches of data. The *batch size* is an important parameter that needs to be tuned on a case by case basis, as we will explain in the next section.

1.3.4 Training

While glossing over some details, model training can summarized in the following steps:

- Given data pairs (x_i, y_i), where x_i is an input sample, and y_i is the desired output
- Determine a function f, such that $f(\theta, x_i) = \hat{y}_i \approx y_i$, where θ are model parameters and f is the model architecture
- Define a loss function $L(\theta) = \sum_i L(\hat{y}_i, y_i)$ that measures how close the model's output is to the desired output
- Optimize θ to minimize $L(\theta)$

Here the loss function can take on many forms, depending on the training task, we summarize some of the most common ones in Table 1.1. Parameters

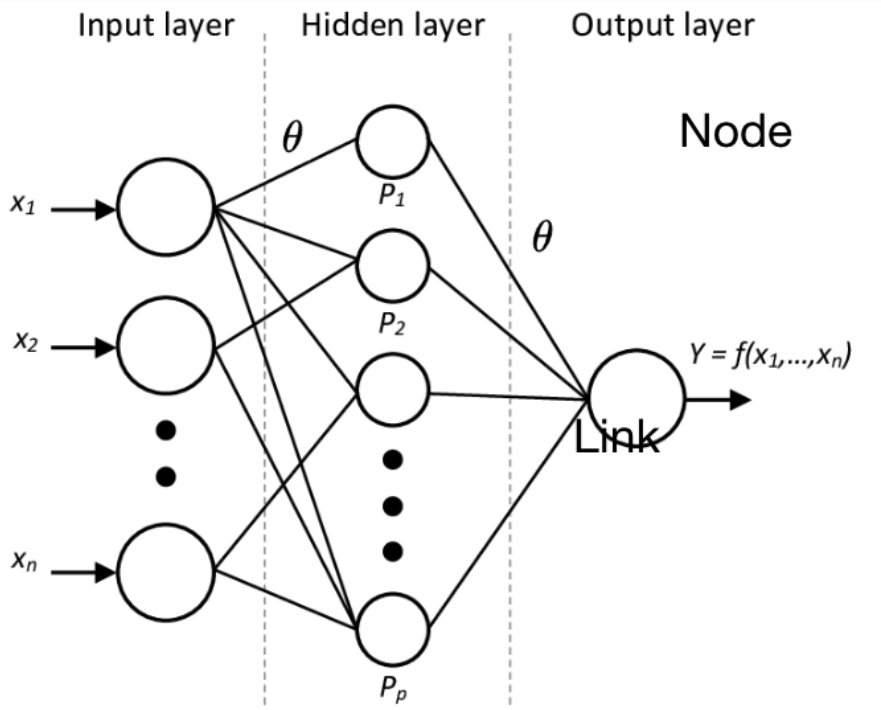

FIGURE 1.6
Deep neural network architecture

in the last layer are then optimized according to the rule

$$\theta \to \theta - \alpha \frac{\partial L}{\partial \theta} \tag{1.3}$$

where α is called the *learning rate* and determines the step size, the partial derivative of the loss function with respect to the parameters is called the *gradient*. For other parameters the rule is similar, but involves the application of the chain rule from calculus. The algorithm is called *gradient descent* and is the workhorse of all modern deep learning. In contemporary architectures there are millions to hundreds of billions of parameters, and updating them efficiently is crucial.

To optimize the parameters as outlined in Eq. 1.3 we need to evaluate the gradient of the loss function given the data (x_i, y_i). To compute the optimial update step we need to sum over all samples in the dataset, compute and store all the gradients, update the parameters once, and repeat the process until convergence. This, however is not practical, given the size of typical datasets and the number of model parameters – each step would be very computationally expensive and the training slow.

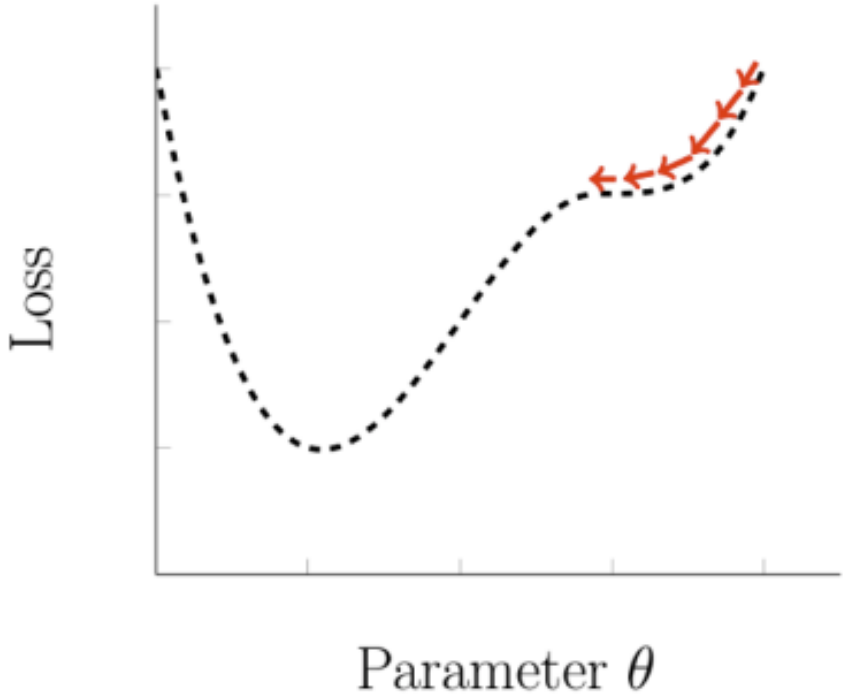

FIGURE 1.7
Minimizing the loss function

On the other end of the spectrum we could use only a single sample to compute approximations of the optimal gradients, and they will take us close to the global minimum. This method is called *stochastic gradient descent* and generally works well, but it comes with some tradeoffs:

- Loading single samples from the dataset is slow, due to CPU, RAM and bus overhead when copying data to VRAM
- There is overhead in the GPU related to scheduling of compute operations
- The gradients are much more noisy than the optimal ones, and the model will learn more slowly (see Fig. 1.8)

There is a happy middle ground called *mini-batch gradient descent* which is a combination of stochastic gradient descent with mini-batches. Using mini-batches allows for a more accurate estimate of the gradient, and reduces the overhead per sample, as we can take advantage of hardware accelerated vectorized compute operations in CPUs and GPUs. In practice, we will choose a batch size that is much smaller than the number of samples in the dataset, and

Metric Name	Function	Abbreviation
Mean Squared Error	$\frac{1}{N}\sum_{i=1}^{N}\|\hat{y}_i - y_i\|^2$	MSE
Mean Absolute Error	$\frac{1}{N}\sum_{i=1}^{N}\|\hat{y}_i - y_i\|$	MAE
Root Mean Squared Error	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}\|\hat{y}_i - y_i\|^2}$	RMSE
Mean Absolute Percentage Error	$\frac{1}{N}\sum_{i=1}^{N}\left\|\frac{\hat{y}_i - y_i}{y_i}\right\|$	MAPE
Symmetric Mean Absolute Percentage Error	$\frac{1}{N}\sum_{i=1}^{N}\frac{\|\hat{y}_i - y_i\|}{\|\hat{y}_i\| + \|y_i\|}$	SMAPE

TABLE 1.2
Regression metrics

as big as possible without running out of VRAM, where the latter condition is typically the more relevant constraint.

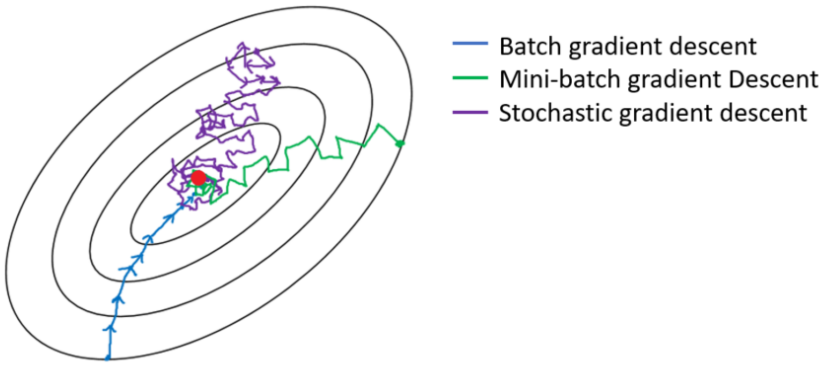

FIGURE 1.8
Batch vs mini-batch vs stochastic gradient descent

1.3.5 Evaluation and Metrics

To know how well our model is doing we need to define how to measure its performance. When the target is a real number (regression) we can simply compare the model output using one of the metrics in Table 1.2.

When the target is a categorical variable (classification) we will still get a distribution of real numbers as output, i.e. the confidence of the model that a sample belongs to a certain class or set of classes. The model output will be close to one if the sample belongs to the class, and close to zero if it does not. To compute classification metrics, we critically need to define a *threshold*

Metric Name	Function	Interpretation
Accuracy	$\frac{TP+TN}{TP+FP+FN+TN}$	Fraction of samples classified correctly
Precision	$\frac{TP}{TP+FP}$	Of all samples predicted as $True$, fraction of $True$ samples
Recall	$\frac{TP}{TP+FN}$	Of all $True$ samples, fraction of samples predicted as $True$
F_1	$2\frac{Precision \times Recall}{Precision+Recall}$	Harmonic mean between Precision and Recall

TABLE 1.3
Classification metrics

t. By default we could choose $t = 0.5$, as that corresponds to how the loss function determines whether to push a sample in one direction or the other. Later we will discuss reasons and tips for how and why you might want to change this value.

Assuming we have only have two classes, i.e. $True$ or $False$, once a threshold has been chosen, every sample will be in one of four categories:

- The model output is greater than the threshold and the sample belongs to the $True$ class, we call this a True Positive (TP)
- The model output is greater than the threshold and the sample belongs to the $False$ class, we call this a False Positive (FP)
- The model output is less than the threshold and the sample belongs to the $True$ class, we call this a False Negative (FN)
- The model output is less than the threshold and the sample belongs to the $False$ class, we call this a True Negative (TN)

With this we can define the metrics in Table 1.3. Most classification problems are strongly imbalanced and we typically assign the $True$ label to the less common class. This also means that *Accuracy* is rarely the best metric to consider. For example, assume we have an imbalance of $1:100$ of $True:False$, if we have an algorithm that always predicts $False$ we will get $Accuracy = 0.99$, $Precision = undefined$, $Recall = 0.00$, $F_1 = undefined$. By adjusting the threshold we can change the tradeoff between $Precision$ and *Recall* according to our downstream needs, reflecting whether it is more acceptable to have have a higher rate of FP or FN predictions. For example, it may be acceptable to lower the threshold if the goal is finding very rare events from a large dataset for manual inspection, resulting in lower $Precision$ but higher *Recall*. The F_1 score is balanced measure of $Precision$ and *Recall* and will be low if any of them is low, and there is a threshold for which it will be maximal. As a rule of thumb, you should aim to achieve $F_1 \gtrsim 0.7$.

There is another metric that is often used in classifcation tasks that is independent of class imbalance and threshold called the area under the *Receiver Operating Charateristic* curve, or *ROC AUC* (Figure 1.9). For

this we determine the *False Positive Rate* (FPR) and *True Positive Rate* (TPR) for each threshold t

$$FPR(t) = \frac{FP(t)}{FP(t) + TN(t)}; \;\; TPR(t) = \frac{TP(t)}{TP(t) + FN(t)} \tag{1.4}$$

and take the integral

$$ROC\ AUC = \int_0^1 TPR\big(FPR(t)\big)\, dt, \tag{1.5}$$

corresponding to the weighted mean over all possible thresholds. We automat-

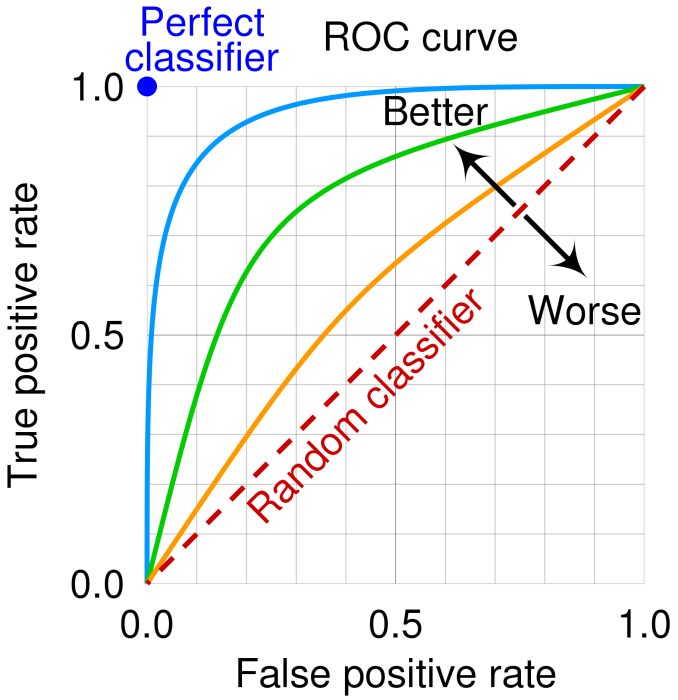

FIGURE 1.9
ROC curve, a random classifier would achieve $AUC = 0.5$, a perfect one 1.0.

ically generate metrics and plots relevant for text classification tasks in the code accompanying this chapter.

1.3.6 Over- and Underfitting

A common problem in ML is the balance between over- and underfitting (Figure 1.10). Underfitting can happen when the model is too simple, i.e. it is not

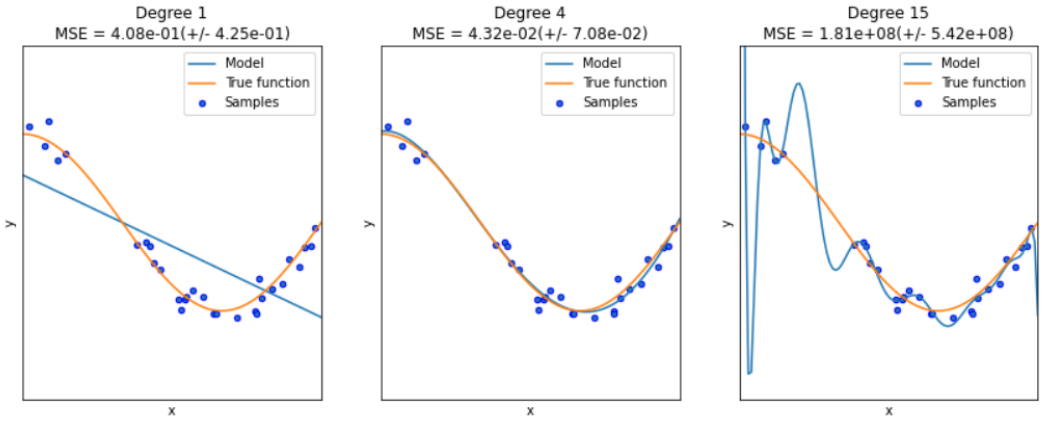

FIGURE 1.10
Left: Underfitting, a straight line doesn't fit the data well. Right: Overfitting, a complex curve fits the (training) data perfectly, but will likely not generalize well. Center: The model and true underlying function match closely.

able to capture the underlying function (rarely a problem in deep learning). Overfitting happens when the model is too complex or has too many degrees of freedom, such that it can fit the training data well, but does not capture the essence of the true data generating process, leading to poor generalization performance when applied to new data.

An easy way to detect these issues is by observing training and validation metrics as a function training steps or epochs, called learning curves. We want to see that both, training and validation metrics continue to improve as training progresses until they both taper off at a certain point. Note that it is normal to see a difference in the absolute value and rate of change between the training and validation metrics. This is called the generalization error, and is nothing to be too concerned about, ultimately only the validation and test metrics are relevant. What *is* problematic is if we observe that the validation metrics start to get worse, while the training metrics continue to improve – this is a clear sign of overfitting and we should stop the training early (Figure 1.11). To address underfitting, the easiest remedy is to increase the complexity of the model (e.g. add more layers) or to increase the number of training steps. We can also try to add more features to the dataset, such as adding more words to the vocabulary. When dealing with overfitting in deep learning, we usually don't want to reduce the model complexity, but instead add *regularization* and more data. Regularization is a technique that restricts the freedom of the model, for example by introducing a penalty on the magnitude of the model

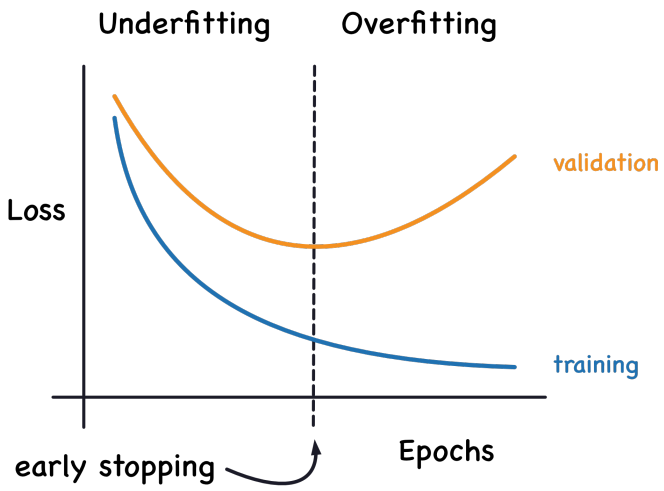

FIGURE 1.11
Learning curves, the training and validation loss are plotted as a function of the number of training steps / epochs.

parameters. For this, the loss functions in Table 1.1 are modified as

$$\tilde{L}(\theta) = L(\theta) + \frac{1}{2}\sum_{i} |\theta_i|^2. \tag{1.6}$$

Another common technique is to add *dropout*, where a fraction of neurons is randomly turned off for each training step. This ensures that each neuron has a unique purpose and learns to detect a specific set of features in the data, without co-adapting to the output of other neurons. Another interpretation of dropout is that it can be seen as an approximation to a bagging technique, where the outputs of different models at each iteration are averaged, leading to an overall reduction in errors if they are uncorrelated. This tends to reduce generalization error, prevent the model from fitting to irrelevant details, and make it more robust to noise.

1.4 Full Walkthrough - SUDEP Detection

This is placeholder text

1.5 Full Walkthrough - Brain Surgery and Financial Impact of Epilepsy

In this section we will demonstrate how the techniques discussed above can be applied to a practical example within the field of Epilepsy. Modern NLP techniques are a powerful tool to analyze large amounts of unstructred or semi structured text data, and we can employ them to develop a coding scheme and significantly reduce the time required to apply it to new data.

We are providing self contained code examples in the acompanying GitHub repository[10] that makes use of a general purpose NLP library[11] we have developed to make current industry standard tools and libraries more accessible to researchers and practitioners.

1.5.1 Data Analysis

For this section we started out with fairly small dataset ($\sim$ 150 rows) derived from survey responses regarding impact and treatment of epilepsy[12]:

> Do you have any comments on the financial impact of epilepsy?
>
> Do you have any comments about brain surgery in general?

This dataset is small enough to manually inspect, but we the techniques discussed here scale well beyond.

1.5.1.1 Embeddings

“nobreak

1.5.1.2 Dimensionality Reduction and Clustering

“nobreak

1.5.1.3 Cluster Labeling

“nobreak

1.5.1.4 Annotation

We can use the cluster labels as a starting point to decide on a coding scheme. For the general comments we decided on the following labels:

> `"Not eligible"`, `"Last resort"`, `"Would never do it"`,
>
> `"Considering it"`, `"Was Unsuccessful"`, `"Was partially successful"`,

[10] https://github.com/chris-boson/epilepsy
[11] https://github.com/robmsylvester/sheepy
[12] Seizure Tracker

`"Was successful"`, `"Side effects"`, `"Risk"`,

`"Too expensive"`, `"Complications"`, `"Unknown outcome"`,

`"Unnecessary"`, `"Cannot find origin"`

We treat this as a so called multilabel classification problem, where a given sample can have multiple true labels simultaneously (i.e. a surgery can be successful and have side effects).

The labeling scheme should cover all the aspects of the data of interest. It also needs to be unambiguous, meaning that two experts independently generating the annotations should largely agree on what the correct labels are for each sample. If human experts cannot agree, the model will most likely perform poorly as well, as it gets inconsistent signals during training. If this is the case, the coding scheme or annotation instructions should be revised. We will discuss in Section **??** how to go about analyzing model performance and uncover issues in data and annotations.