Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Сучасні підходи використання нейронних мереж в аналізі текстів**»

Виконав: студент 3-го року навчання,

Освітньої програми «Комп’ютерні науки», 122

Драговоз Даміан Васильович

Керівник Глибовець А.М._____

Доктор технічних наук

Рецензент _____

(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2025

Table of Contents

List of Abbreviations

- API – Application Programming Interface
- BERT – Bidirectional Encoder Representations from Transformers
- BPE – Byte-Pair Encoding
- GPT – Generative Pre-trained Transformer
- JAI – Just a Language
- LLM – Large Language Model
- LSTM – Long Short-Term Memory
- NLP – Natural Language Processing
- NN – Neural Network
- RegEx – Regular Expression
- RNN – Recurrent Neural Network
- TF-IDF – Term Frequency-Inverse Document Frequency

Introduction

In recent years, artificial intelligence (AI) has experienced a significant rise in popularity and usage. One of the breakthroughs responsible for this surge was the invention of GPT language models, which captivated public attention. This work aims to explore fundamental concepts in AI that are utilized in such models while focusing more on NLP-specific tasks. While more advanced topics—such as the entire architecture of transformer networks and other cutting-edge techniques—are not examined in exhaustive detail, the material presented here is sufficient to provide a robust conceptual foundation. The objective is not to cover every aspect exhaustively but to ensure a clear understanding of why these models are effective, how the various components integrate, and what makes NLP systems so successful.

Part 1 (Tokenization)

The first and most basic thing about processing text has to be tokenization.
Tokenization is the task of splitting text to then work with those pieces.
Those pieces are usually called tokens.

The simplest way to tokenize is to split some text into characters or spaces.

1. Character-level tokenization: "flopper" → 'f', 'l', 'o', ...
2. Whitespace-level tokenization: "flopper died" → "flopper", "died"

Unfortunately, these simple types of tokenization techniques do not allow us to work with language efficiently. For example, the character level tokenization does not represent the natural structuring of words. Words like "playing" have some structural meaning that character-level tokenization does not let us capture. Instead, we have seven independent characters.

Whitespace-level tokenization solves this problem since we have words as words. Unfortunately, because words have many word-forms, it might be hard to gather enough text to completely learn a language.

Some techniques like stemming and lemmatization have been developed to deal with this issue.
Stemming is based on removing the variation parts from the word, so words like "playing" would be transformed to "play".
Lemmatization is about mapping word forms to some initial, prime word forms.
So, the same word "playing" would be mapped to "play".
These techniques are rarely used in today's NLP.

A widely used tokenization algorithm is BPE(Byte-Pair encoding).

**The following explanation focuses on raw ASCII characters*.

Byte-Pair encoding algorithm:

1. Count the frequency of adjacent token pairs in the text.
2. Find the most frequent pair.
3. Merge this pair to create a new token.
4. Add the new token to our vocabulary.
5. Update the text by replacing all occurrences of the pair with the new merged token.
6. Repeat until we reach a desired vocabulary size or until no more merges are possible.

The BPE algorithm works well because it captures the relations between words automatically. To be more technical, it has "Byte" in its name since it starts at the character level, and all the characters are 1 byte long. It works well in the real world since ASCII is rarely used, especially for texts outside of English. Usually, UTF-8 is used. UTF-8 is a globally accepted, still active symbol encoding. It supports all 1,112,064 valid Unicode code points using a variable-width encoding of one to four one-byte (8-bit) code units. This means that regular ASCII symbols take 1 byte of memory, and some more fancy symbols, like Chinese letters, might use more (up to 4 bytes), we still represent them as 1-byte value sequences. Since we are working with raw bytes, it allows the BPE algorithm to work with any text, regardless of language or encoding.

Therefore, initially, our vocabulary consists of 256 tokens (Regular ASCII tokens), every other byte pair is learned.

```
text = """Ryan Thomas Gosling"""
encoded = encode(text)
decoded = decode(encoded)
print(f"text         : {text}")
print(f"encoded_ids  : {encoded}")
print(f"encoded_chars: {list(map(lambda x: decode([x]), encoded))}")
print(f"decoded      : {decoded}")
print(f"len(encoded) : {len(encoded)}")
print(f"len(decoded) : {len(text)}")
print(f"original == decoded: {text == decoded}")
```

```
text         : Ryan Thomas Gosling
encoded_ids  : [82, 121, 385, 741, 296, 388, 71, 111, 115, 108, 279]
encoded_chars: ['R', 'y', 'an ', 'Th', 'om', 'as ', 'G', 'o', 's', 'l', 'ing']
decoded      : Ryan Thomas Gosling
len(encoded) : 11
len(decoded) : 19
original == decoded: True
```

Figure 1.1 - The result of using BPE on some arbitrary text.

As we can see, encoding is a list of token IDs. Some of those IDs are less than 256. Those are the original ASCII tokens, and others are created by the BPE algorithm. For example, "ing" is a single token and its id value is 279, "as " is aslo a single token.

For better results, more sophisticated algorithms are used. They preprocess text with regular expressions before tokenization, so, for example, words and punctuation are not combined. Regular BPE might create such tokens: ", he", " There, " when a more sophisticated BPE won't.

OpenAI uses their own version of BPE tokenization with regex pre processing called tiktokenizer [5]. It has an open API and a website that demonstrates how some versions of the public gpt models work. Here are a couple of examples:

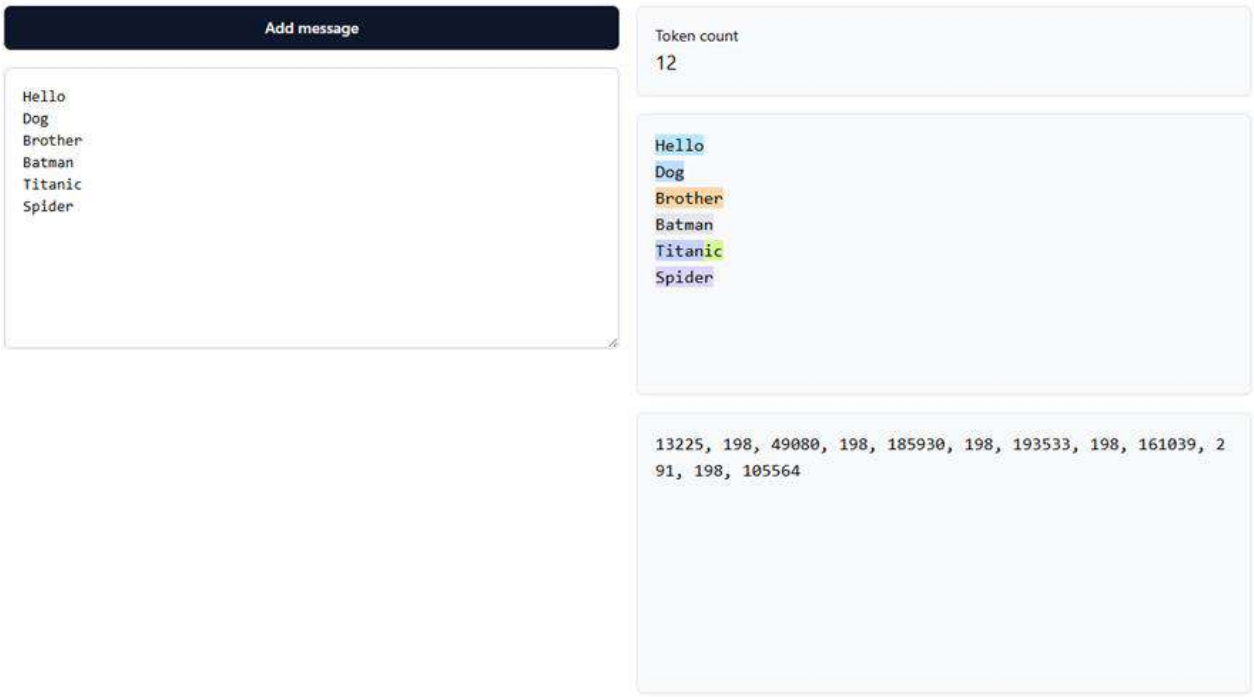

Figure 1.2 – Examples of gpt-4o tokenization of English words [5].

The above image is for gpt-4o. As we see pretty much all the most used word in the English language are are a combination of a single token. Even batman is a single toke, titanic is 2 tokens long. This is the example of how well BPE can learn words, especially if enough text is given.

But, English is the dominant language for the internet and the training set for open ai models, what about other languages. Here is Ukrainian:

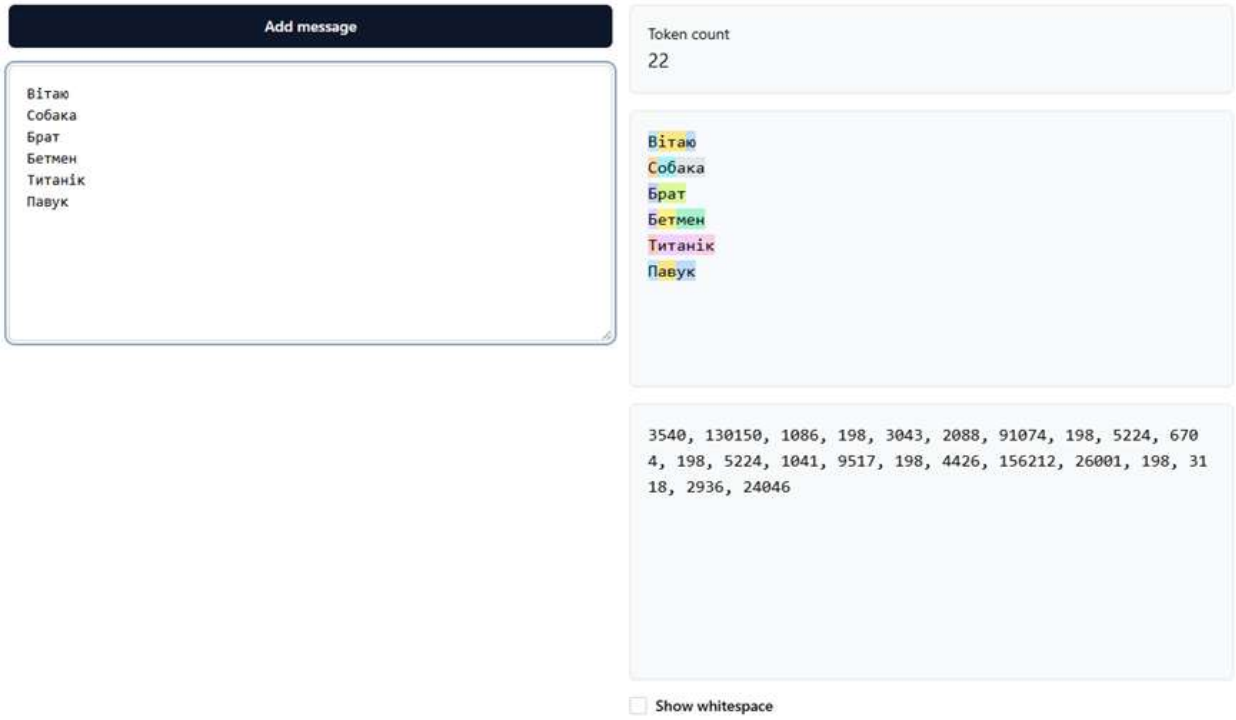

Figure 1.3 – Examples of gpt-4o tokenization of Ukrainian words [5].

As we see, for the same model, but different language, the same widely used word now consist of multiple sub tokens.

This lack of knowledge will directly corelate to how well the model performs generative tasks. Whitch is the selling point behind GPT models. Exactly the reason why sometimes LLMs might perform better in English, especially the older versions. This is not a problem for most use cases now, but its something to think about.

Another example of how tokenization might directly result in validity of generated text is the python code.

GPT 2 model's tokenization is also public on github. For it OpenAI decided to not combine whitespaces into separate multi-white-space tokens. Whitch meant that gpt-2 did not have any sequences of only white spaces. Whitch is fine for most use cases, but not for python code generation.

Figure 1.4 – Examples of gpt-2 tokenization of python code [5].

As we can see all the spaces are separate tokens, whitch might result in worse python code generation for reason that will be talked about later.

GPT-4 fixed this issue by changing the whitepsacing tokenization policy, allowing sequenced of whitespaces to become separate tokens, therefore tabulation can be represented as separate token in python instead of sequences of tokens:

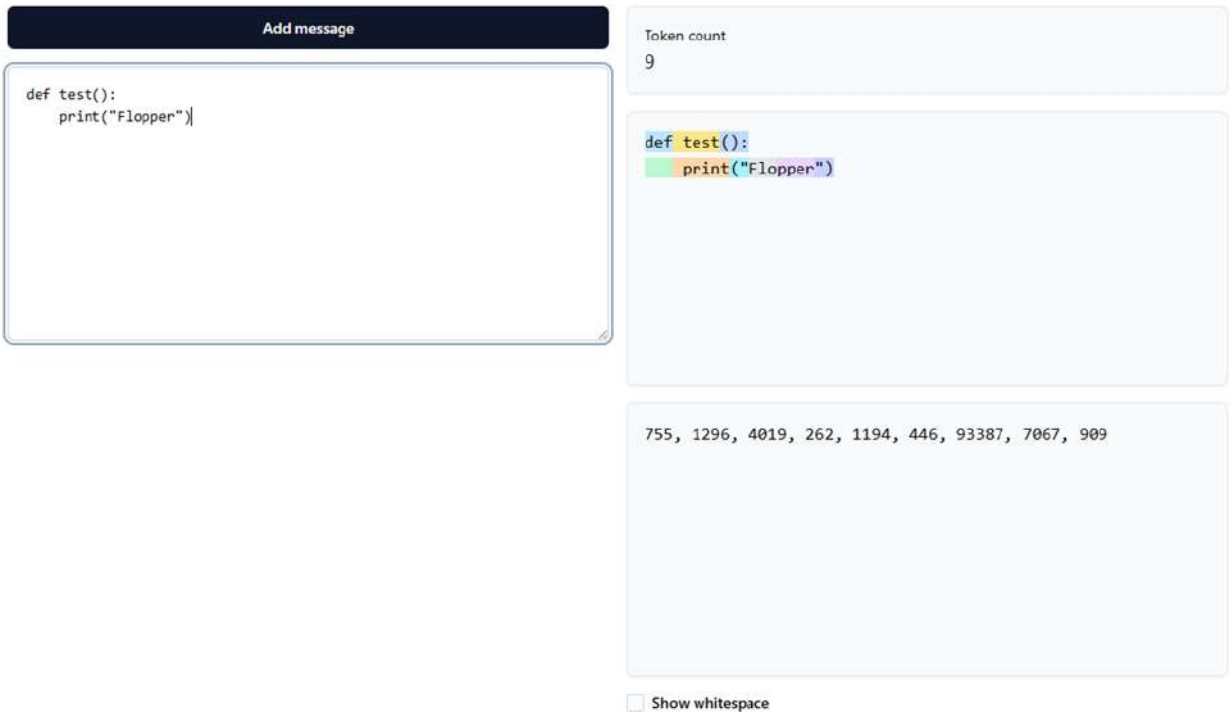

Figure 1.5 – Examples of gpt-4 tokenization of python code [5].

Summary:

This chapter introduces the fundamental process of tokenization – the first important step in processing language. Explained some basic tools like stemming, which reduces words to their root forms; lemmatization, which maps words to their base forms; and byte-pair encoding, which is used in modern state-of-the-art language models.

Also, some concrete examples of how different tokenization approaches might affect the end product were shown, based on the publicly accessible GPT-2 and GPT-4 language model tokenizers from the Tiktokenizer website [5].

Part 2 (N-gram models)

Huge chuck of today's NLP is based on modeling or classifying text. Modeling is a task of predicting upcoming words based on previous words and classification is the task of classifying text to be an instance of some class (e.g.: positive of negative, spam or not, etc.).

For text generation, we want to be able to predict words based on some context. For example if we have a sentence "**People have been waiting for JAI to release for**", what word is most likely to come next? Lets say the word is "ages". This is the soul of text modeling tasks and is usually represented using the following notation.

P("ages" | "People have been waiting for JAI to release for")

Starting to work on text processing requires at least some thinking about the nature of text. All the text is produced by someone, with some idea behind it, at some point in time. Languages change, dialects change, word meanings change, new words get created, some get forgotten. All of those things add some complexity to the way we should think about language processing.

From here, we shift our focus to the most basic type of word prediction, the purely probabilistic model for text, called the n-gram model.

The nature of the n-gram models is based on the Markov assumption. This assumption states that new words are only dependent on some previous set of words we call context and not on the whole prior sequence.

Word probabilities are therefore calculated in the following way:

1. Original idea:

$$P(herewegoagain) = P(\,here \mid \text{<start>}\,) \cdot P(\,we \mid here\,) \cdot P(\,go \mid we\,) \cdot P(\,again \mid go\,)$$

2. Markov assumption with context of 1:

$$P(\,\text{new\_word} \mid here\ we\ go\ again\,) = P(\,\text{new\_word} \mid again\,)$$

It is used as a base for the n-gram models. N-gram models are the most basic type of models for a reason. They work directly with tokens and base the probability of the upcoming words strictly on the text seen in the static context window.

The formula below calculates the probability of a word w_n being the next word for context of a single word w_{n-1}, $C(w_1\ w_2)$ is the count of times words w_1 and w_2 appeared in the same context in the training text:

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}\ w_n)}{\sum_w C(w_{n-1}\ w)}$$

The above logic explains the process of calculating the probabilities of the upcoming word for 2-gram models, also known as bigrams. The same logic can be applied to all the other types of n-grams. The only difference is the size of context windows.

	i	want	to
i	3	640	0
want	1	0	470
to	2	0	4

Figure 2.1 – Example of bigram word-pair table.

In the figure 2.1 each cell represents the number of times the row words were the context words for the column words. So the "i" was the context word for "want" 640 times.

Now it is possible to generate some text, purely on the statistical knowledge of word-pairs from the bigram table.

But, there is a substantial problem. There will be valid word sequences that will never be generated like "to want" because $C(\text{"to" "want"})$ is 0; therefore, any possible sequence that has "to want" as a part of it can never be generated. Obviously, these types of problems might not occur for such trivial and often used word sequences in a model that has enough text to train on, but still, this leaves the model with a hole in generation abilities. There are some techniques that deal with that, such as Smoothing, Interpolation and Backoff.

Smoothing is a type of optimization where values for $C(\text{word}, \text{context})$ are artificially increased to deal with the problem of 0 probability. For example, Laplace smoothing(Add one smoothing) is the most straight forward one.

$$P_{Laplace}(w_n|w_{n-1}) = \frac{C(w_{n-1}\, w_n) + 1}{\sum_w (C(w_{n-1}\, w) + 1)} = \frac{C(w_{n-1}\, w_n) + 1}{C(w_{n-1}) + V}$$

V -> size of vocabulary

Interpolation deals with the problem of 0 probability differently. It focuses on creating a final probability based on probabilities of different contexts. Here is an example for a bigram model:

$$P_{Interpolation}(w_n|w_{n-1}) = \gamma_1 P(w_n) + \gamma_2 P(w_n|w_{n-1})$$

Backoff is similar to interpolation; but, it does not consistently utilize different contexts. Instead, backoff is employed exclusively in cases where a zero probability occurs. In such situations, we revert to an n-gram model with a smaller context until we arrive at one that yields a non-zero probability.

$$P_B(w_i|w_{i-N+1:i-1}) = \begin{cases} \dfrac{C(w_{i-N+1:i})}{C(w_{i-N+1:i-1})} & if\ count(w_{i-N+1:i}) > 0 \\ \gamma S(w_i|w_{i-N+2:i-1}) & otherwise \end{cases}$$

Summary:

This chapter introduces the most basic, purely statistical text-generating model. Explained the concept of context windows. Techniques like smoothing, interpolation, and back-off that help n-gram models generalize better.

Part 3 (Word embeddings)

The main linguistic problem with regular statistical models like n-grams is that they only imitate the most probable text based on the text they were trained on without considering the words' unique meaning. To generalize better, we want to have something that represents word meaning in some way. Let's say our model can generate sequences like "The dog has to be fed," and the word "fed" was generated in the context of "dog has to be". But, if we do not have enough examples for a word like "cat", which would make complete sense in this case, our model might struggle to generate it since it is purely statistical. Having "cat" and "dog" represented as something similar would allow us to generalize and model language better.

So we need to work with word meaning in some way. The easiest way to do it is to evaluate words based on some set of characteristics statically. One of the original ideas was to discriminate between words based on their connotation, using three values (valence, arousal, and dominance) [7].

Even with such a generic system, we can already capture the meaning of words since each word is a 3D vector. (It is not the same meaning as in dictionaries, but by clustering words in space, we can see something; call it meaning here.) Because of this, words can be compared, therefore we can see synonymy and antonymy just based on the Vector position of those words in their 3D vector space.

But we need a better way to represent words than just on some arbitrary, hand-crafted set of 3 values.

Term-Term matrix.

It is a matrix of size V*V (V - vocabulary size), and it stores rows that represent how many times each word from the vocabulary was in some arbitrary static context with every other word in text.

	…	cpu	bread	pie	fortnite	…
banana	…	0	200	400	15	…
sugar	…	0	430	950	3	…
building	…	0	10	17	840	…

Figure 3.2 – Example of a term-term matrix.

We now have vector representations of words of size V. It gives us a picture of words that occur within the context of other words. All of this is built on the premise that surrounding words influence the meaning of other words. Now, having these vectors, we can more accurately judge the similarity between word meanings since the word vectors have more dimensions. Even from Figure 2.2 we can see that the word "banana" is more similar to "sugar" than "building".

The main problem with the term-term matrix is that it is sparse and contains many zero-value columns, which we would rather not to be the case.

Another word vector representation is called tf-idf (Term frequency-inversed document frequency). We will not discuss them here since they are never used for NLP tasks nowadays and have the same sparsity problem as term-term vectors.

P.S. Term-term vectors are also never used, but they present both the nature of word vectors and the sparsity issue, which are important to understand why the final version of word vectors is as cool as it actually is.

So, we want to fix the sparsity issue and create dance vectors that theoretically store word meanings. These vectors are called Embeddings. They are dense, usually 50-1000 dimensions in size.

There are two generally different types of embeddings used: word2vec and BERT. Word2vec is based on static word embeddings, whereas BERT has different embeddings for words based on their surrounding context. This may result in better representation since sometimes words might be used outside their regular context. For example, "fire" in the sequence "LA files destroyed hundreds of homes" has a negative connotation. Whereas the same word "fire" but in a sequence "This Kanye's album is fire" has a very different, even positive connotation. This context-based difference in meaning might not be captured by regular static embeddings, especially if the training texts were not chosen with this kind of example in mind, making "fire" unlikely to appear in both contexts. (Especially since in most corpora "fire" usually means something bad rather than good)

To manually create word2vec-like embeddings, we use a skip-gram embedding algorithm. We first create two embedding tables of size V*d with some initial values, where V is the vocabulary size and d is the chosen word vector size.

Both tables store a single embedding vector for each word from the vocabulary. The first table stores embeddings for words when they are the target words, and the second

table for when they are the context words. Each embedding is initialized somehow, and the values are adjusted to fit the loss function via training. So, both tables (matrices) are of size V*d, where V is the vocabulary size and d is the arbitrarily chosen embedding dimensions.

To adjust the embeddings, we move through text, and, at each point, we work with a single target word and a set of context words. This algorithm assumes that the context words do not have any positional meaning, so it treats them as a set of context words.

…Jon Bellion has not [*released an album in more*] *than half a decade …*

Figure 3.3 – Example of a target word (in blue) surrounded by context words (in red) within a fixed window, with non-context words (in black) shown outside the window.

We want the target word to be more like the neighboring context words and less like the words that are not valid context words. We use dot product to calculate the similarity of 2 embeddings. The higher the dot product value, the more similar two words are. The lower the dot product, the more different the meanings of the words. So now we have a base for a loss function. We want to maximize the similarity of the target words relative to their similarity with context words and minimize the similarity between target words and invalid context words. To do that, we use something called negative sampling. We manually sample some noise words as invalid context words.

What do we want to achieve?
If we try to use already trained static word embeddings like "glove-wiki-gigaword-100" [8], we can see words cluster nicely.

```
import gensim.downloader as api

model = api.load("glove-wiki-gigaword-100")
print(type(model))

[==================================================] 100.0%
128.1/128.1MB downloaded
<class 'gensim.models.keyedvectors.KeyedVectors'>
```

```
print(model["bread"][:5])
print(model["croissant"][:5])
print(model["london"][:5])

[-0.66146  0.94335 -0.72214  0.17403 -0.42524]
[-0.25144  0.52157 -0.75452  0.28039 -0.31388]
[ 0.60553  -0.050886 -0.15461  -0.12327   0.6627  ]
```

Figure 3.3 – Example of “glove-wiki-gigaword-100” word embeddings [8].

As we can see, the embedding vectors for words "Bread" and "croissant" are more similar in the first 5 dimensions than "bread" and "Lodnon", which would be expected.

The following figure is an example of word meanings clusterization in space. The word "bread" exists near words related to bread.

```
model.most_similar("bread")

[('flour', 0.7654521465301514),
 ('baked', 0.7607272267341614),
 ('cake', 0.7605516910552979),
 ('loaf', 0.7457114458084106),
 ('toast', 0.7397798895835876),
 ('cheese', 0.737463653087616),
 ('potato', 0.7367483377456665),
 ('butter', 0.7279618382453918),
 ('potatoes', 0.7085272669792175),
 ('pasta', 0.7071877717971802)]
```

Figure 3.4 – Example of "glove-wiki-gigaword-100" word clusterization [8].

The following figure poses the question:
"England to London is the same as Paris to what? " Using word embeddings, we can answer that by doing some simple vector space geometry and looking up similar embeddings from vocabulary via something like cosine similarity.

```
model.most_similar(positive=["paris", "england"], negative="london")

[('france', 0.8000162243843079),
 ('italy', 0.6718133687973022),
 ('spain', 0.6684681177139282),
 ('lyon', 0.6396933197975159),
 ('match', 0.6387163400650024),
 ('french', 0.6371537446975708),
 ('portugal', 0.6350722908973694),
 ('belgium', 0.6339790225028992),
 ('argentina', 0.6298829317092896),
 ('marseille', 0.62220299243927)]
```

Figure 3.5 – Example of "glove-wiki-gigaword-100" word analogy [8].

We would first need a representative textual data set to learn word embeddings from scratch. Preparing text is very important. Manipulation embeddings trained without regex preprocessing might still result in accurate word meaning in space placements, but there will be too many of the same word tokens that are only different due to merged in punctuation. And ever though we can still perform the analogies and synonymy, we might have a token redundancy issue.

We have to figure out what to do with punctuation and such. Usually, we learn embeddings for tokens from vocabulary created by something like BPE. But we should also have a second vocabulary for full words if we want to do some word analogy and comprehensive word clustering. Those full word embeddings could then be calculated by something like a vector mean over the token embeddings. For example, if the word "building" is 2 BPE tokens, "build" and "ing". Both tokens have

some embedding values. To create the embedding for the complete word 'building', we combine these individual token vectors to produce a single vector of the same dimensionality that represents the semantic meaning of the entire word 'building'.

The next few figures show how no regex preprocessing might result in redundant tokens for custom word embeddings.

```
[('ходити', 0.9999999403953552),     [('брат', 0.9999997615814209),
 ('ходити.', 0.9632261991500854),     ('брат.', 0.9686229228973389),
 ('виходити', 0.9412965774536133),    ('братами', 0.9476014971733093),
 ('ходитиме', 0.9189841747283936),    ('братів', 0.9450318813323975),
 ('шкодити', 0.9008939862251282),     ('братові', 0.9435192942619324),
 ('зашкодити', 0.87871253490448),     ('трафить', 0.8901544213294983),
 ('вихопити', 0.8696615695953369),    ('котра', 0.8753548264503479),
 ('ходив', 0.8623732924461365),       ('дратували', 0.8727647066116333),
 ('розбитих', 0.8578888177871704),    ('дратувати', 0.8700396418571472),
 ('ходят', 0.8578352928161621)]       ('завтра', 0.8674534559249878)]
```

Figure 3.5 – Examples of full-word skip-gram embeddings for text with no regex preprocessing and no punctuation issues.

```
[('hobbit', 1.0),
 ('hobbits', 0.902370810508728),
 ('hobbiton', 0.8374344706535339),
 ('hobbit_', 0.8134421706199646),
 ("'hobbits", 0.8121212720870972),
 ("hobbit.'", 0.793027937412262),
 ('bree-hobbits', 0.792302131652832),
 ('hobbit.', 0.7764489650726318),
 ('hobbits.', 0.7608093023300171),
 ('hobbit-holes', 0.7422460913658142)]
```

Figure 3.6 – Example of full-word embeddings for text with no regex preprocessing and a lot of redundant tokens due to punctuation issues.

Summary:

This chapter explores a fundamental concept in NLP – word embeddings. It presents various techniques for representing words as vectors and illustrates the functionality of word2vec-like embeddings, showcasing the powerful capabilities they offer for understanding and manipulating language. The chapter displays examples of self-trained embeddings and highlights issues that arise when dealing with highly punctuated text. Additionally, it discusses techniques to manipulate tokens for better embeddings for those types of texts.

Part 4 (Neural Networks, Feed-Forward)

The last chapter introduced the concept of word embeddings that, as was mentioned, helps to generalize better, by using word meaning representations rather than words themselves. To use those word embeddings effectively and therefore generalize over meaning, we turn to machine learning once again.

Feed-forward networks for NLP classification.

The first thing we can do with a feed-forward network is to use it as a fancy text classifier. For example, we could classify text as positive, negative or neutral. We might handcraft some features that model will learn to classify text on. For example: number of positive words, number of negative words, etc. Here, we should take the same approach as we did with embeddings, and try to move away from handcrafting anything. Instead, we might use already created static embeddings. By using them, we are using a so-called transfer learning technique. Transfer learning is a technique where a model developed for one task is repurposed as the starting point for a model on a second task. Pretrained embeddings are used as the starting point. The model now has the following structure:

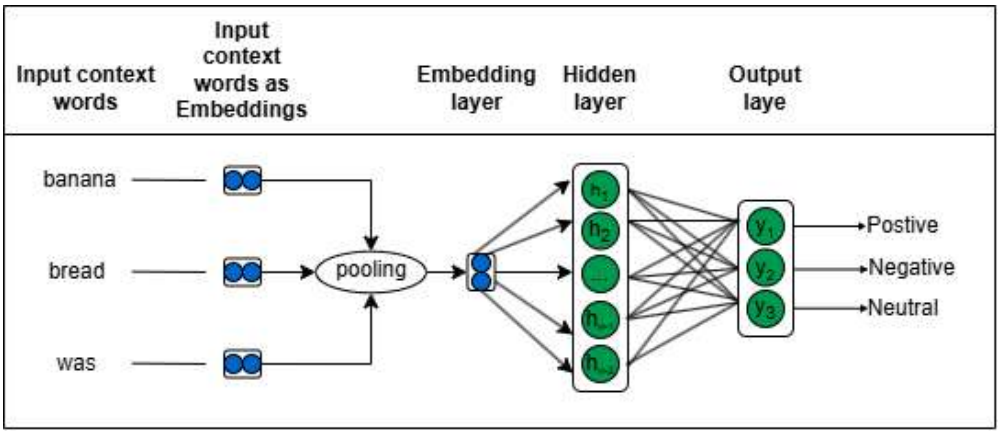

Figure 4.1 – Example of a Feed-Forward word sequence classifier with a single hidden layer.

Words are passed in and encoded into their embeddings representation. Polling is something that, in this case, maps from the list of embeddings to a vector that represents the input. Something like an average over embedding dimensions might be used. That averaged vector is now the input to the feed-forward classification model.

Feed-forward Language Modeling

Language modeling is a process of predicting upcoming words. The network will be able to predict words based on the context of previous words. The same way as with n-grams it is using a static context window of size N, so the probability of word w_t after some context can be calculates with the formula below:

$$P(w_t|w_1, \dots, w_{t-1}) \approx P(w_t|w_{t-N+1}, \dots, w_{t-1})$$

Neural language models represent words as embeddings; this positively affects generalization since the model can work better with pairs $(\text{word}, \text{context})$ that it might not have seen while training. The simplest example would be the sentence, "Cat has to be fed". If we use this sentence in training, but then in testing have "Dog has to be fed", the regular n-gram model will be able to predict "fed" after "Cat has to be" (Context window of size four is used in this example), but might have a problem predicting the exact word "fed" for "Dog has to be". A neural network, on the other hand, might do a better job predicting "fed" since it works with word meanings via embeddings and, therefore, will be able to generalize from "cat" to "dog" since these two words have similar embedding vectors.

The following figure shows the process of predicting a single word via a simple Feed-forward neural network using pre-trained static embeddings with a context window of three words.

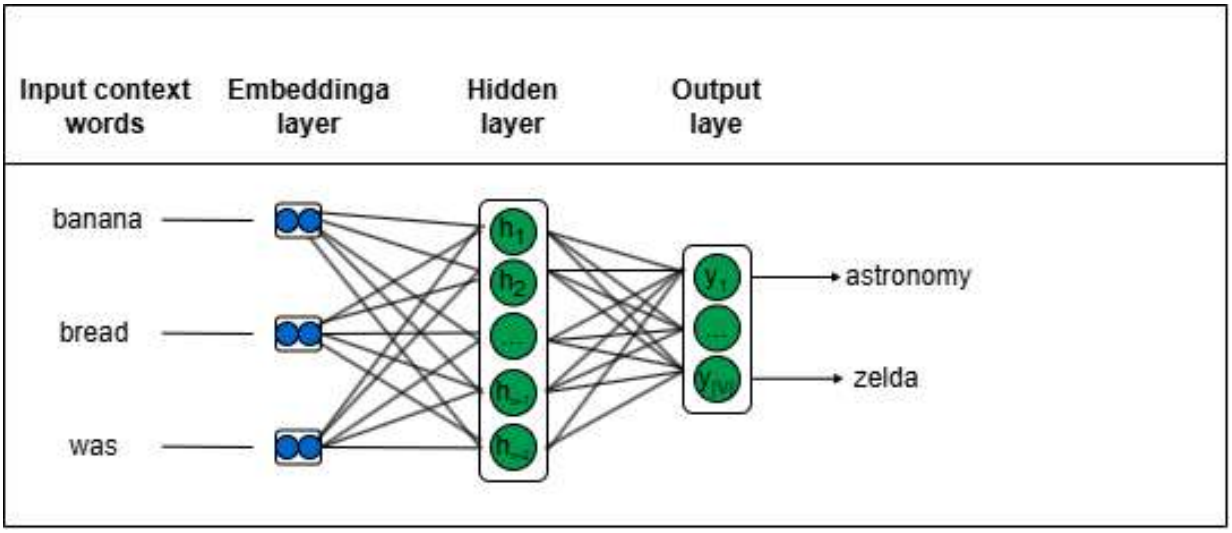

Figure 4.2 – Example of a Feed-Forward text generator with a single hidden layer.

Summary

This chapter explores the application of machine learning in NLP tasks, highlighting classification and generation tasks that utilize feed-forward networks with pretrained word embeddings as a key starting point.

Part 5 (Recurrent Neural Networks)

The problem we are facing now is the fixed context window we have for our generative models. Recurrent neural networks perform better in tasks where context from far back might be crucial, by using the learned knowledge about the input from previous hidden states to produce the output of the current hidden state.

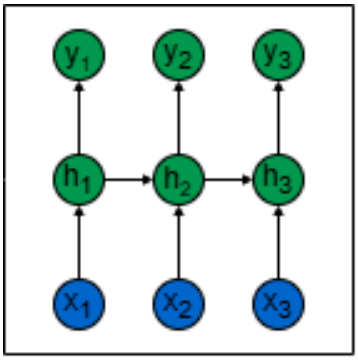

Figure 5.1 – Recurrent network architecture.

RNNs for text classification:

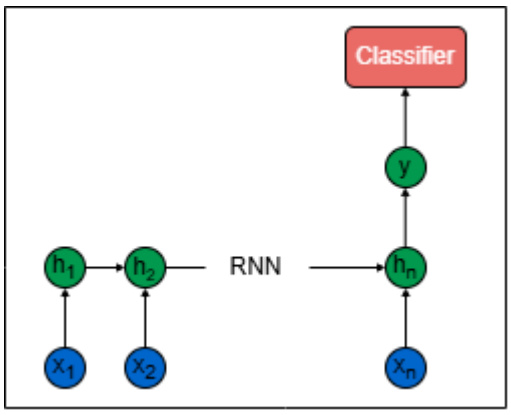

Figure 5.2 – RNN sequence classification architecture.

In this approach, the final hidden state h_n of the RNN serves a purpose of a better summary representation of the input sequence and passed to a feedforward network

for classification. While this captures information from the entire sequence, relying solely on h_n can be limiting, as it may be biased toward more recent inputs in the sequence.

Alternative strategies such as mean pooling or max pooling over all hidden states (h_t for all t) are commonly used to address this. These methods aggregate information from all time steps, producing a more balanced representation that captures early and late sequence features. This is particularly useful since RNNs struggle to retain long-range dependencies due to vanishing gradients, making it challenging to backpropagate effectively through long sequences.

RNN for text modeling:

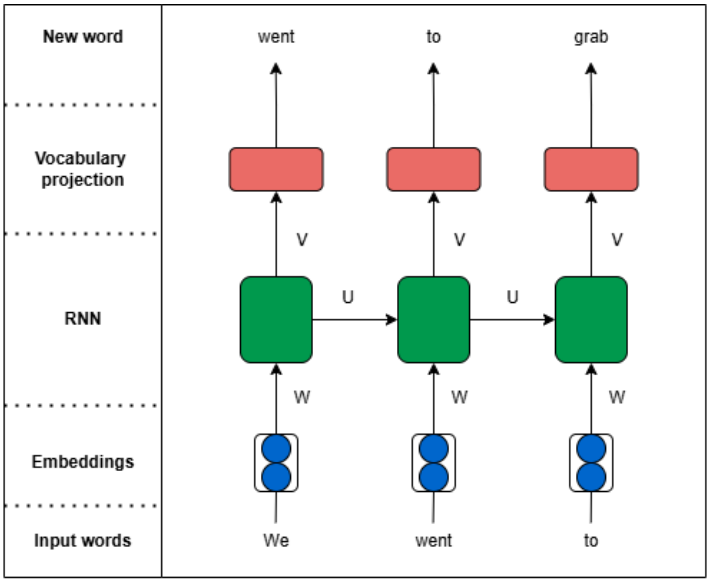

Figure 5.3 – RNN text modeling architecture.

RNNs perform better in text generation tasks for the same reason they perform better in the classification tasks, they store long range context better than regular feed-forward models with static context window.

For text generation RNNs take in word embeddings as inputs and utilize both them and the result of the previous hidden state. The output is a vector of size V (vocabulary size) that represents the most probable next words.

Stacked RNNs

Stacked RNNs perform better than single-layer RNNs. This is the same reason regular NNs use more than one hidden layer. This allows the model to learn more high-level features over time and therefore have better results. Since a single RNN layer is just a more sophisticated hidden layer, using more of them allows for better results.

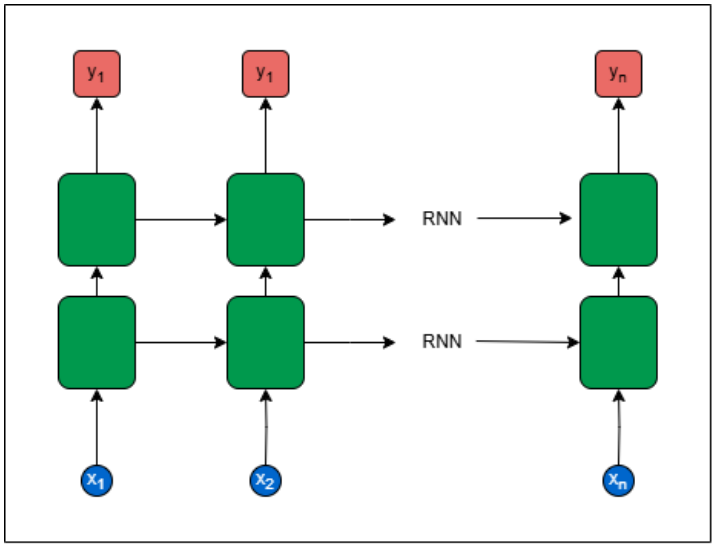

Figure 5.4 – Stacked RNN architecture

Bidirectional RNNs

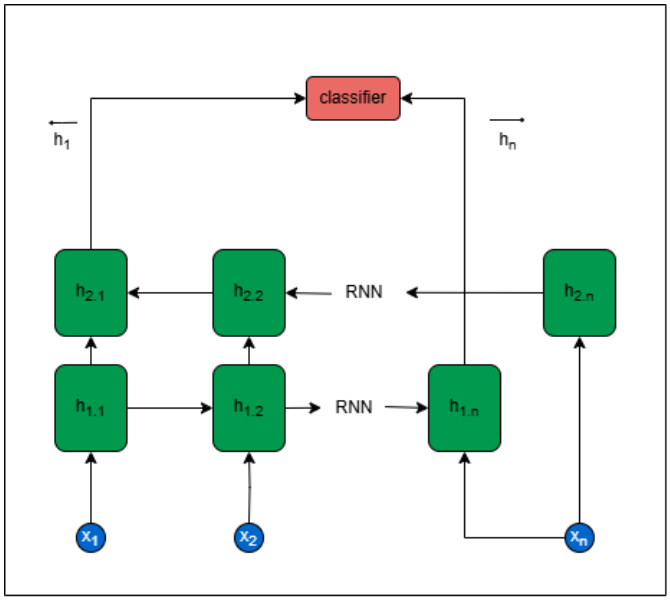

Figure 5.5 – Bidirectional RNN architecture

Bidirectional RNNs can also be used for something like text classification. Different directions of the layers allow for better long-sequence representation, since we are working with 2 directions of learning, so both long range and short-range features get more representation in the hidden layers.

LSTM

LSTM – Long Short-Term Memory is another, more sophisticated version of RNNs. They consist of cells consisting of three different gates: the forget gate, the input gate, and the output gate. LSTMs deal with the problem of vanishing gradient in long-range sequences, by removing unimportant context. The forget gate serves as the eraser of important information from the cell state. The add gate chooses what information to add to the cell state for future calculations. The output state chooses what information is needed for the calculation of the current state.

These cells serve as a more sophisticated hidden layer inside a regular RNN.

Referring to Figure 5.4, the green squares are the hidden states of a stacked RNN. In an LSTS, these would be LSTM layer cells.

Summary:

This chapter introduces Recurrent Neural Networks (RNNs) and demonstrates their applications in NLP tasks. Several RNN architectures are examined, including standard RNNs, Bidirectional RNNs, and Stacked RNNs, LSTMs. The chapter highlights how these specialized neural networks effectively process sequential data by maintaining internal memory states, making them particularly suitable for NLP problems.

Part 6 (Transformers – Attention is all you need)

Nowadays, every known text-generative AI is based on the transformer architecture first presented in a paper published by Google in 2017. The paper "Attention is all you need" introduces an architectural concept called "Attention." In language processing, attention can be thought of as a way to build up contextual representation of meaning over time, which is why it was used for text translation in the paper.

Why is attention useful? Let’s look at a sentence: "The dog didn’t eat the food because it was too old. " As humans, we can easily tell what "it" refers to. But earlier language models often had trouble understanding these kinds of references. Simple embeddings like Word2Vec aren’t enough to handle this level of meaning — that's where attention mechanism makes a big difference.

These examples demonstrate that words that help us understand the meaning of other words can appear quite far apart in sentences or paragraphs. Transformers excel at

creating contextual representations of word meaning by incorporating these helpful contextual words, even when they're distant.

In a transformer model, we build increasingly rich contextual word embeddings layer by layer. At each stage, the representation of a token combines its information from the previous layer with information from surrounding tokens. This process creates progressively more sophisticated contextual representations for each word at each position.

Attention is a mechanism that figures out the importance of context words to the target word.

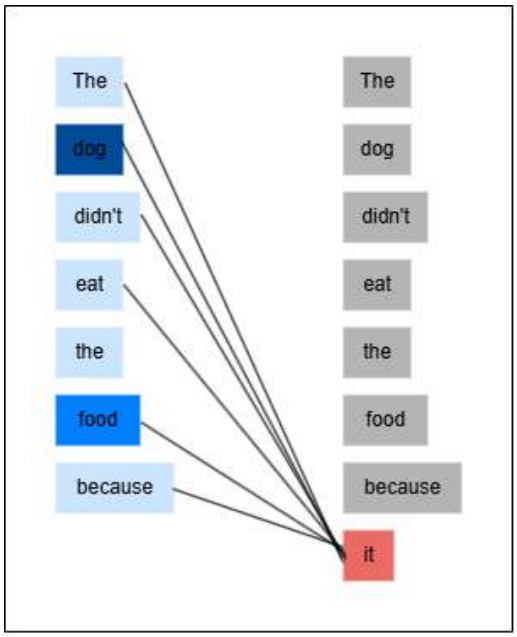

Figure 6.1 – Illustration of what attentions tries to achieve.

This figure illustrates how a transformer builds up a contextual representation for the token 'it' basing it of the whole prior context. The process draws on representations from all previous tokens.

The color intensity in the figure represents the attention distribution: "dog" and "food" appear in darker colors, indicating they receive higher attention weights when computing the representation for "it". This concentration of attention is crucial because "it" ultimately refers to either "dog" or "food", and these strong connections help the model resolve this coreference relationship correctly.

The process uses three important matrices:

1. Query (Q): Represents what the current token is "looking for" in the context
2. Key (K): Represents what each token in the sequence "offers" as context
3. Value (V): Represents the actual information the word contributes to the overall representation.

For any input vector x_i, these are computed as:

1. $Q = x_i * W^Q$ $(where\ W^Q\ is\ the\ query\ weight\ matrix)$
2. $K = x_i * W^K$ $(where\ W^K\ is\ the\ key\ weight\ matrix)$
3. $V = x_i * W^V$ $(where\ W^V\ is\ the\ value\ weight\ matrix)$

A single attention head process:

1. x_i – current element, x_j – some prior context.
2. Calculate Q, K, V matrices.
3. Get dot product between Q_i and K_j.
4. Normalize by dividing by the square root of the dimensionality of the query and key vectors (dk).
5. Softmax.
6. Output $head_i$ is based on a dot product between α_{ij} and V_j.

7. Matrix W^O is used to reshape the output of the head to have the same dimensionality as the inputs (x_i).

The final set of equations for a single input vector x_i:

$$Q_i = x_i * W^Q \quad K_j = x_j * W^K \quad V_j = x_j * W^V$$

$$score(x_i, x_j) = \frac{Q * K}{\sqrt{d_k}} \quad where\ d_k\ is\ the\ dimensionality\ of\ Q\ and\ K$$

$$\alpha_{ij} = softmax\left(score(w_i, w_j)\right) \quad \forall j \leq i$$

$$head_i = \sum_{j \leq i} \alpha_{ij} V_j$$

$$a_i = head_i W^O$$

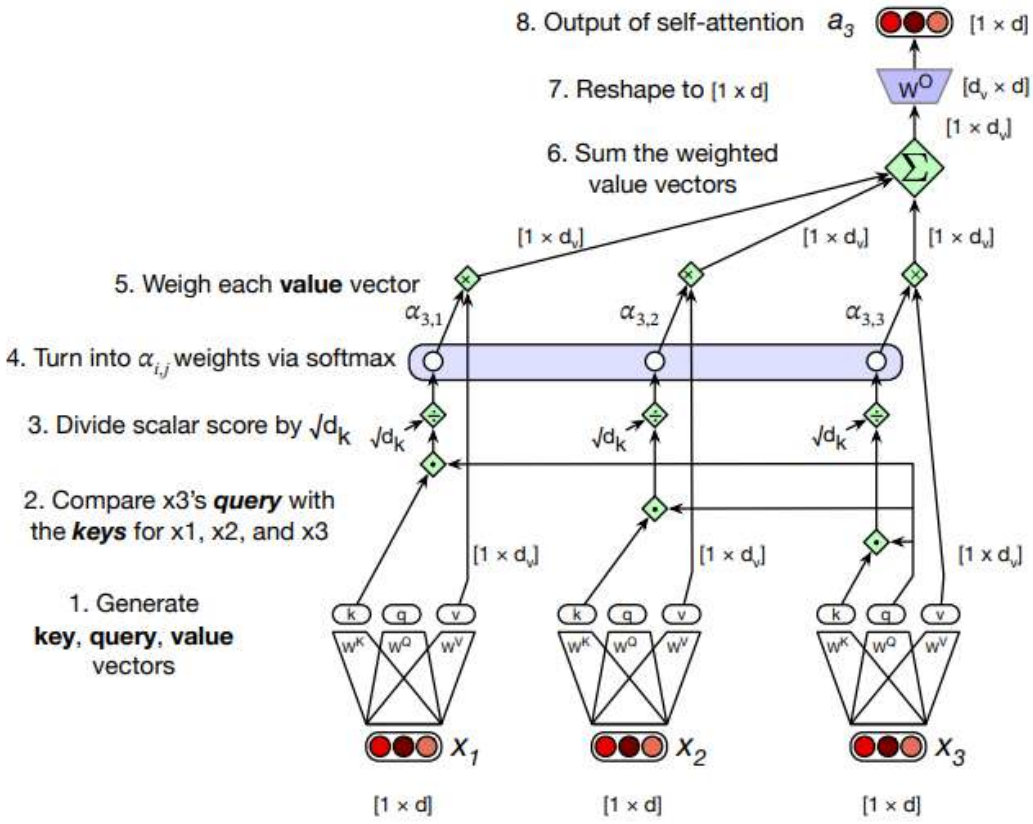

Figure 6.2 – Process of calculating the third element using attention [10].

Transformers utilize the attention mechanism to build contextual representations; everything else is similar to regular NNs. Therefore, a regular transformer Layer has the following architecture:

1. Fist layer norm is used to normalize data to deal with better gradient calculation later.
2. Attention.
3. Second layer norm is used to normalize data to deal with better gradient calculation later.
4. Feed-forward layer.

By stacking multiple transformer layers and increasing the model's depth, we enable it to learn more sophisticated features about the text. This approach produced the following language modeling transformer design:

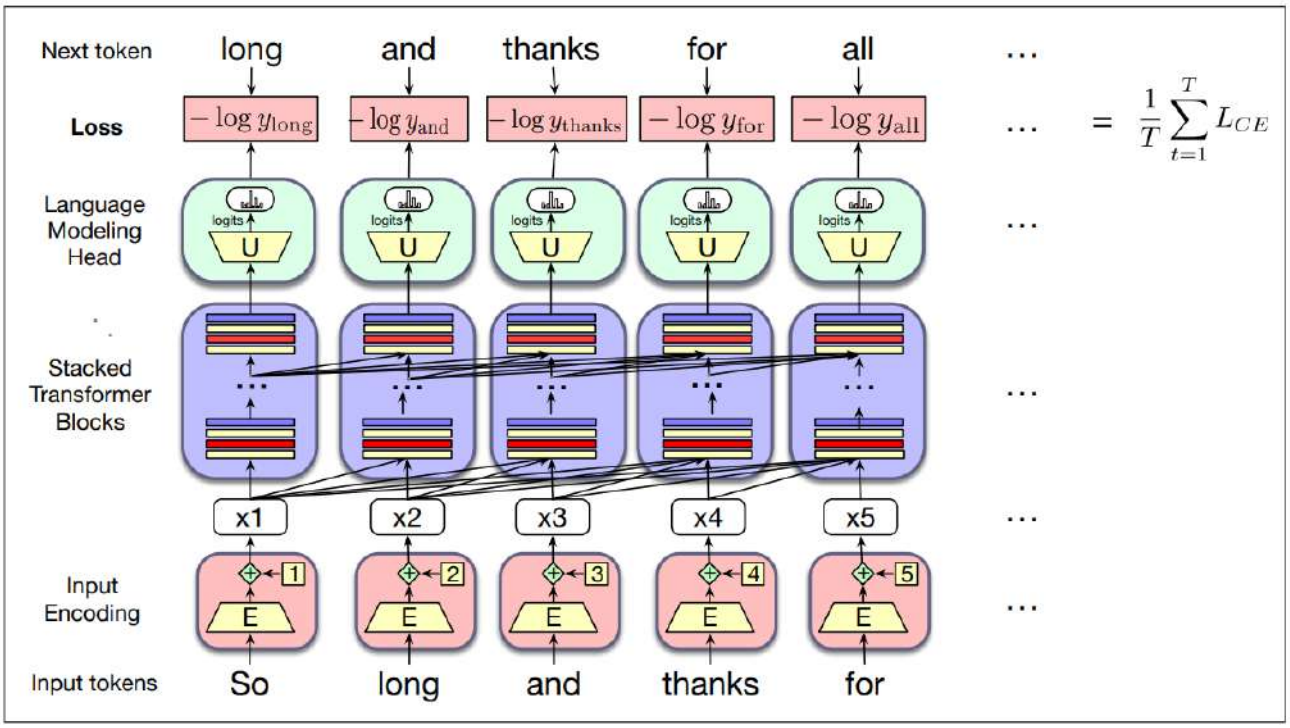

Figure 6.3 – Transformer neural network architecture for language modeling [11]

Transformers can be used for way more than just language modeling. "Atten is all you need" paper focused specifically on text language translation tasks and introduced two different transformer blocks: Encoder and Decoder. The idea behind having two different blocks is to first translate text's meaing into some mathematical vector representation, therefore encoder block. The decoder block then decodes the collected knowledge into some other language.

For language modeling tasks, we only use the encoder part.

Summary:

This chapter introduced the revolutionary Transformer architecture that changed modern Machine Learning and quickly became the go-to state-of-the-art design. It explained the nature of attention and the key features that go into it. Some examples of systems designed for producing new words based on learned information from text using transformers were shown.

Part 7 (Personal Attempts and Results)

This part is dedicated to my personal attempts to recreate some of the architecture previously explained, mostly focusing on text generation.

Byte-Pair Encoding:

It started with regular byte pair encoding, which is explained in Part 1. There was not much to see there since tokenization does not provide any value outside of the context for which it is done. At first, a regular BPE tokenization with no regex preprocessing was implemented. A couple of iterations were implemented to see how the original text, from which the vocabulary is formed, influences the later tokenization of unseen words, predominantly from other languages.

```
text = """瑞恩·汤玛士·高斯林（英语：1980年11月12号-）"""
encoded = encode(text)
decoded = decode(encoded)
print(f"text[:50]:    {text[:50]}")
print(f"len(test):    {len(text.encode(encoding='utf-8'))}")
print(f"len(encoded): {len(encoded)}")
print(f"len(decoded): {len(decoded)}")
print(f"decoded:      {decoded}")
print(f"original == decoded: {text == decoded}")

# NOTE: as we see, the BPE is able to encode and decode text that it was not trained on.
#       chinese symbols are utf characters and take up more than 1 byte os space
#       bpe was not able to cut down the length,
#           because it can only merge pairs that it has isnide vocab
#       but it have no byte pairs for chinese symbols,
#           since it was trained on english text, therefore different byte pairs
```

Figure 7.1 – Example of BPE tokenization trained on English text and then applied to text from another language. As a result, the byte length of the starting text and the number of tokens produced remained the same.

Embeddings:

To start of with embeddings, some pretrained publicly available embeddings were used to understand the basic idea of them. In particular, "Glove-wiki-gigaword-100" were used. After, a couple of iterations of training, embeddings were tested and later utilized in generation tasks. First iteration was using tokens produced by the regular BPE.

```
[('померти', 1.0),
 ('померти.', 0.9784828424453735),
 ('помер', 0.9567384123802185),
 ('померла', 0.9477607607841492),
 ('померли', 0.9427328109741211),
 ('помер.', 0.9414197206497192),
 ('помітити', 0.9154691696166992),
 ('помітити.', 0.9083929657936096),
 ('міксером.', 0.8963026404380798),
 ('помахом', 0.8949304223060608)]
```

```
[('мантія', 1.000000238418579),
 ('мантіями', 0.9708421230316162),
 ('мантіях', 0.9686458110809326),
 ('мантіях.', 0.956716001033783),
 ('мантіями.', 0.9547590017318726),
 ('мантіях…', 0.9496756792068481),
 ('мантій', 0.9296981692314148),
 ('мантії', 0.9233766794204712),
 ('мантію', 0.9206645488739014),
 ('мантію.', 0.9128550887107849)]
```

Figure 7.2 – Examples of word similarity, produced by custom word embeddings trained on a Ukrainian version of Harry Potter books using skip-gram technique. Tokens were created using vanilla BLE. Embeddings of the full words, were everaged over the subtoken embeddings.

```
[('seeing', 0.9520323),
 ('unseeing', 0.890099),
 ('seething', 0.85602164),
 ('_see', 0.85052687),
 ('seeks', 0.8464993)]
```

Figure 7.3 – Example of word analogy ("play" → "playing" == "see" → ?), produced by the custom word embeddings trained on an English version of Harry Potter books, using the same algorithms as in Figure 7.2.

A later version of word embeddings utilized a sophisticated regex preprocessing using the pretrained Tiktoken tokenizer "cl100k_base" [6] used for the GPT-4 model. Better results came with better tokenization, the most satisfying aspect is how the vectors were able to capture the existence of prepositions nearby in space:

```
[('and', 1.0000001192092896),
 ('but', 0.8101518154144287),
 ('so', 0.7278461456298828),
 ('so.', 0.7171671390533447),
 ('yeah.', 0.7127646207809448),
 ('you…', 0.7123894095420837),
 ('nowhere', 0.7083984613418579),
 ('vand', 0.7060924768447876),
 ('you', 0.6926586031913757),
 ('well.', 0.692141056060791)]
```

Figure 7.4 – Example of prepositions being positioned closely in multidimensional embedding vector space.

We can also see some flaws, like two different versions of "you" being close to "and". This is not an issue since this was not a large dataset relative to the datasets used in NLP.

Feed-Forward:

Feed-forward models were manually tested but mostly skipped, just like n-grams, since they serve more as a path to better architectures; therefore, more time was spent on them.

RNN:

RNNs showed much promise. Unfortunately, I did not have enough computational resources to conveniently train on sufficiently large training sets, so tiny datasets were used. For this section, I used pretrained embeddings for tokens from a larger set of Harry Potter books (735_000 words, 185_000 tokens, 100-dimensional embeddings). The model used a regular stacked RNN architecture and a feed-forward layer at the end to map into the vocabulary. (Refer to figures 5.3 and 5.4). Via training, I was able to capture the development of model's text generation abilities. The model went from generating such text.

==========

harry looked to a head and and harape's and" from the base seed "harry" to generating "harry woke them. He loocked down, down at his.

==========

To generating such text:

==========

harryappedly on a floor and his knees trembling and saw a bowl.

thely split exactly it, he was still too in a funny when he had a plan to stop to the dumbledore's eyes's arms, which he looked at last, he

==========

As we can see, many words make sense and even make some sense in combination with other words around them. The model was also able to mimic the structure of text. Since most of the text had an empty line between sentences, the model learned to insert new lines as well.

Also, I ran some testing using different generation hyperparameters. Since the model is trained on a tiny dataset, if we only use the most probable next word, we will end up with a sometimes repetitive, but pretty coherent text:

==========

‘i’m ’ npt tp try, potter?

‘oh, i’m glad to be chess of the teachers championshit the gamekeeper. And i’ll lose you loads,” said hagrid, patting his crossbow.

==========

If we sample a random word from the most probable list, we can see how the model starts to generate increasingly incoherent text.

A lot of these issues can be fixed if a sufficiently large data set is used, which would allow the model to adjust the weights better, and would also make the model have more words to use. Better tokenization could also be used to deal with capital letters.

LSTM:

As was expected, LSTM produced better text generation results than regular RNNs. This time I used a larger data set. The same data set used for embedding training.

The model produced "harry had to a to be, and hariddle was a very room to a wand. " after just a single epoch of training. With time, it started mimicking the text structure more and getting complex grammar correct, for example, a sentence: "hagrid was still looking at him. she had just forgotten her. ". Sometimes the model would still generate weird words, like "harrycases", but this was more due to tokenization and the minor errors inside the dataset. There is still obviously a long way to go, since generated sequences are sometimes not 100% valid, but the subsequences look a lot like real text. Some examples of great sequence generations:

1. harry had to see it. But she had been in one of a very deal, and he was sure, but the dambledore was going for the ministry
2. “well, you know,” he asked her voice to the gargoes

Transformers:

Transformers were not implemented, but some transformer architectures have been reviewed, such as the character-level transformer from Andrew Karpathy [4]. It used character-level tokens and broad context to learn how to model text. It showed that transformers are so powerful that even character level is enough to generate valid text sequences. If pretrained token embeddings are used, it should lead to legitimate results.

References

1. Jurafsky, D., & Martin, J. H. (n.d.). Speech and language processing (3rd ed. draft).
2. Karpathy, A. (2015, May 21). The unreasonable effectiveness of recurrent neural networks. GitHub Blog. https://karpathy.github.io/2015/05/21/rnn-effectiveness/
3. Karpathy, A. (2025). Minbpe [GitHub repository]. GitHub. https://github.com/karpathy/minbpe
4. Karpathy, A. (2025). Nanogpt-lecture [GitHub repository]. GitHub. https://github.com/karpathy/ng-video-lecture
5. OpenAI. (2025). Tiktokenizer API. https://tiktokenizer.vercel.app/
6. OpenAI. (2025). Tiktoken [GitHub repository]. GitHub. https://github.com/openai/tiktoken
7. Osgood, C. E., Suci, G. J., & Tannenbaum, P. H. (1957). The measurement of meaning. University of Illinois Press.
8. Stanford NLP Group. (n.d.). GloVe: Global vectors for word representation (Wiki Gigaword 100). Stanford University.
9. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In Advances in Attention is all you need (2017). https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
10. Figure 6.2: Process of calculating the third element using attention. [Adapted from Jurafsky & Martin (2021, p. 189)]

11. Figure 6.3: Transformer neural network architecture for language modeling. [Adapted from Jurafsky & Martin (2021, p. 211)]