Natural Language Processing

Zichao Yang

Zhongnan University of Economics & Law

Date: October 28, 2024

Roadmap

- Basics
  - NLP
  - Tokenization
  - Word Embedding
- Language Models (statistical models)
  - Bag-of-Words (BoW)
  - Term Frequency-Inverse Document Frequency (TF-IDF)
  - N-grams
  - Latent Dirichlet Allocation (LDA)
- Large Language Models (neural network-based models)
  - Attention
  - Transformer
  - GPT
  - BERT

Natural Language Processing

Q: What is Natural Language Processing?

ChatGPT: Natural Language Processing (NLP) is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and human language. NLP aims to enable computers to understand, interpret, and generate human language in a way that is both meaningful and useful.

⋆ ChatGPT is a large language model-based chatbot developed by OpenAI company.
⋆⋆ Guess what? ChatGPT actually contributed a portion of this lecture note. But no worries, I've double-checked it!

NLP Objectives

1. **Language Understanding**

- **Text Classification**: Categorizing text into predefined categories, such as spam detection, sentiment analysis, or topic classification.
- **Named Entity Recognition (NER)**: Identifying and categorizing named entities in text, like names of people, organizations, locations, and more.
- **Part-of-Speech Tagging (POS)**: Assigning grammatical tags to words in a sentence, such as noun, verb, or adjective.

NLP Objectives

2. **Language Generation**

- **Text Generation**: Creating coherent and contextually relevant text (i.e., Chatbots), content generation (i.e., AI News Generators), etc.
- **Machine Translation**: Translating text from one language to another, such as Google Translate.
- **Summarization**: Automatically generating concise and coherent summaries of longer texts, which is valuable in news aggregation and document summarization.

NLP Objectives

3. **Sentiment Analysis**

NLP is widely used in detecting the sentiment or emotional tone in a piece of text, which is valuable for brand monitoring, customer feedback analysis, and market research.

4. **Question Answering**

NLP is used in developing systems that can read and comprehend text and answer questions based on the information in the text. NLP is the cornerstone of building chatbots and virtual assistants to accommodate diverse business needs.

Tokenization

Tokenization is the process of breaking down a text into individual units, often words or subwords. These units are called tokens.

Tokenization serves several functions, including:

- **Text Preprocessing:** Tokenization is the initial step in preparing text data for NLP tasks. It simplifies the text by segmenting it into manageable pieces.
- **Feature Extraction:** Tokens are used as features in NLP models, and they help represent the text data in a numerical format that models can work with.
- **Text Analysis:** Tokenization is essential for text analysis tasks like sentiment analysis, text classification, and named entity recognition.

Types of Tokenization

- **Word Tokenization:** This method breaks text down into individual words. It's the most common approach and is particularly effective for languages with clear word boundaries.
- **Character Tokenization:** The text is segmented into individual characters. This method can save memory but may affect model performance.
- **Subword Tokenization:** Striking a balance between word and character tokenization, this method breaks text into units that might be larger than a single character but smaller than a word.

Subword Tokenization is useful in dealing with unknown words. For example, the model corpus may not include *unlikeliest*, but the model can back out the meaning if the corpus contains *un-*, *likely* and *-est*. This method is commonly used in large language models.

Tokenization Example

BERT model uses subword tokenization:

```
# Load pre-trained BERT model and tokenizer
model = BertModel.from_pretrained('bert-base-uncased').to(device)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Define a sample sentence
text = "BERT uses subword tokenization."

# Tokenize the text
tokens = tokenizer.tokenize(text)
print(tokens)
```

Then we get the following result: [‘bert’, ‘uses’, ‘sub’, ‘##word’, ‘token’, ‘##ization’, ‘.’]
Here, “subword” and “tokenization” are divided into multiple tokens.

Word Embedding

Word embedding is a technique for representing words or tokens in a continuous vector space.

Word embedding serves several functions, including:

- **Semantic Representation:** It captures semantic relationships between words. Words with similar meanings are represented closer in the vector space.
- **Dimensionality Reduction:** Embeddings reduce the high dimensionality of one-hot encoded words, making it easier for models to work with.
- **Improving Model Performance:** Pretrained word embeddings, such as Word2Vec, GloVe, and FastText, are used to initialize models, improving their performance in various NLP tasks.

Common Word Embedding Methods

1. **Word2Vec**:

   - Word2Vec is developed by Google.
   - Word2Vec utilizes either of two model architectures: Continuous Bag of Words (CBOW) and Skip-gram.
   - Word2Vec represents words as dense vectors, and it's trained to predict context words given a target word (Skip-gram) or predict a target word given context words (CBOW).
   - Word2Vec captures semantic relationships between words, and similar words are represented by vectors that are close in vector space.

Common Word Embedding Methods

2. **GloVe** (Global Vectors for Word Representation):

- GloVe is developed as an open-source project at Stanford.
- GloVe combines the features of two model families, namely the global matrix factorization and local context window methods.
- GloVe captures both syntactic and semantic relationships in text.

Common Word Embedding Methods

3. **fastText**:

- fastText is developed by Facebook's AI Research (FAIR) lab.
- fastText represents words as bags of character n-grams, allowing it to handle out-of-vocabulary words and morphological variations effectively.
- fastText is particularly useful for languages with rich morphology.

Common Word Embedding Methods

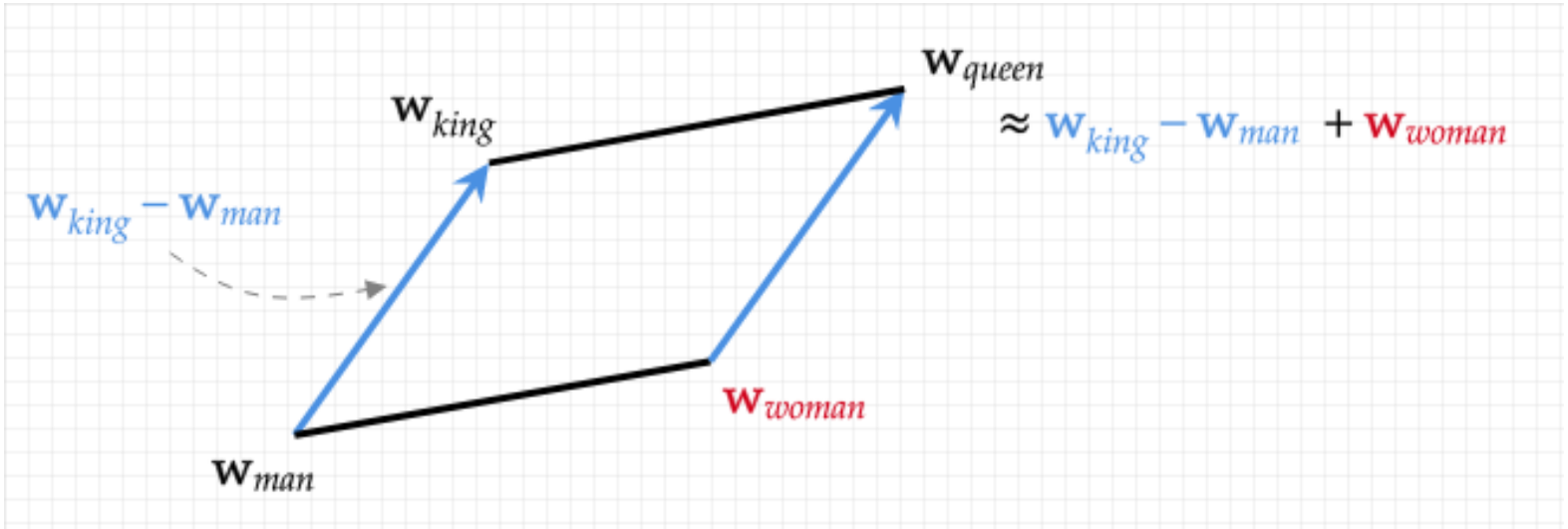
4. **BERT** (Bidirectional Encoder Representations from Transformers):

- BERT is a transformer-based model developed by Google.
- Unlike traditional word embeddings, BERT provides contextual embeddings.
- It is pre-trained on a large corpus and can be fine-tuned for various NLP tasks.
- BERT captures bidirectional context information and has achieved state-of-the-art results in numerous NLP tasks.

Word Embedding Example

The famous Word2Vec embedding example:

$$King – Man + Woman = Queen$$



Credit: School of Informatics, University of Edinburgh

Reference: Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems 26 (2013).

Word Embedding Example

BERT word embedding:

```python
# Load pre-trained BERT model and tokenizer
model = BertModel.from_pretrained('bert-base-uncased').to(device)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Define a sample sentence
text = "BERT uses subword tokenization."

# Tokenize and Encode Text
inputs = tokenizer(text, return_tensors="pt")
print(inputs)

# Obtain Word Embeddings
outputs = model(**inputs)
embeddings = outputs.last_hidden_state
# to access the word embedding for word 'BERT'
embeddings[0][1].shape
```

Language Models

After mapping text sequences into tokens, $x_1, x_2, ..., x_T$, the goal of language models (statistical models) is to estimate the join probability of the whole token sequence:

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t | x_1, ..., x_{t-1})$$

N-grams Model

However, if we apply Markov models to language modeling, we do not need all the history information to predict each token. This leads to a number of approximations that we could apply to model a sequence:

(1) Stochastic Model

$$P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2)P(x_3)P(x_4)$$

(2) First-order Markov Model

$$P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2|x_1)P(x_3|x_2)P(x_4|x_3)$$

(3) Second-order Markov Model

$$P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)P(x_4|x_2, x_3)$$

They are also called *unigram*, *bigram* and *trigram* models.

Bigram Model Example

```python
from collections import defaultdict

# Sample text corpus
text = "i want to eat chinese food for lunch or for dinner"

# Function to create a bigram model
def create_bigram_model(text):
    # Split the text into words
    words = text.split()

    # Initialize a dictionary to store bigram frequencies
    bigram_model = defaultdict(list)

    # Create bigrams and count their occurrences
    for i in range(len(words) - 1):
        current_word, next_word = words[i], words[i + 1]
        bigram_model[current_word].append(next_word)

    return bigram_model

# Function to calculate conditional probabilities
def calculate_conditional_probabilities(bigram_model):
    conditional_probabilities = {}

    for current_word, next_words in bigram_model.items():
        next_word_counts = defaultdict(int)

        for word in next_words:
            next_word_counts[word] += 1

        total_next_words = len(next_words)

        probabilities = {
            word: count / total_next_words
            for word, count in next_word_counts.items()
        }

        conditional_probabilities[current_word] = probabilities

    return conditional_probabilities

# Create the bigram model
bigram_model = create_bigram_model(text)

# Calculate conditional probabilities
conditional_probabilities = calculate_conditional_probabilities(bigram_model)

# Example usage
# The algorithem thinks that 'for' is followed by either 'lunch' or 'dinner'.
print("Conditional probabilities for 'for':", conditional_probabilities['for'])
```

Bigram Model Example

In the above code sample, we only use one sentence to train the model, so the result is boring. The Berkeley Restaurant Project corpus contains 9332 sentences, let's see what Bigram model can learn from it.

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Source: Jurafsky, D. & Martin, J.H., 2023. Speech and Language Processing (draft)

We can see, *i* is most likely followed by *want* (prob = 0.33), *food* is not commonly followed by any word in this table.

Latent Dirichlet Allocation (LDA)

LDA is primarily employed for topic modeling, a process that aims to identify topics within a collection of documents.

Documents are mixtures of topics: Each document in a corpus is assumed to be a mixture of various topics. The number of topics is a hyperparameter and is predefined by you.

Topics are mixtures of words: Each topic is also represented as a mixture of words.

LDA aims to find the optimal set of topic mixtures and word mixtures by examining the entire corpus.

LDA Example

In this simple example, we create 5 documents and try to find out 2 topics from these documents.

```python
# Import necessary libraries
from gensim import corpora
from gensim.models import LdaModel
from gensim.test.utils import common_texts

# Create a sample corpus of documents
documents = [
    ["apple", "banana", "cherry", "date", "elderberry"],
    ["apple", "banana", "cherry", "date"],
    ["cherry", "date", "elderberry"],
    ["apple", "banana", "cherry"],
    ["apple", "banana"],
]

# Create a dictionary from the documents
dictionary = corpora.Dictionary(documents)

# Create a bag-of-words representation of the documents
corpus = [dictionary.doc2bow(doc) for doc in documents]

# Create an LDA model
lda_model = LdaModel(corpus, num_topics=2, id2word=dictionary, passes=10)

# Print the topics and their word distributions
for topic in lda_model.print_topics():
    print(topic)

# Iterate through the corpus and print the topic distribution for each document
for i, doc in enumerate(corpus):
    doc_topic_distribution = lda_model.get_document_topics(doc)
    print(f"Document {i + 1} - Topic Distribution: {doc_topic_distribution}")
```

Large language Models

Large Language Models (LLMs) are neural networks and are pre-trained using self-supervised learning or semi-supervised learning.

LLMs use massive amount of data to learn billions of parameters and can achieve the ability of general-purpose language understanding and generation.

Recent years, the deep learning architecture known as the **transformer** has come to dominate the LLM research field, superseding other deep learning structures such as CNNs, RNNs, and LSTMs.

Online Large Language Models

Large Language Models (LLMs) are complex. Hence, deploying them for real-time applications requires substantial computing power to maintain low-latency responses, and often requires dedicated servers or cloud resources.

Online LLMs becomes the default choice for most individual users:

- ChatGPT
- Claude
- Perplexity
- Kimi-Moonshot
- DouBao-ByteDance
- YiYan-Baidu

Local Large Language Models

Some users may prefer to process all their data locally rather than uploading it to the cloud due to various concerns.

Recently, various local LLM applications have been developed to cater to this need.

How to deploy an LLM on a local PC?

(1) Ollama

(2) Local LLMs

(3) Optional: Anything LLM

(4) Optional: Ollama Python Library

Large Language Model Fine-tuning: DSPy Framework

We write prompts to interact with language models. To tackle complex problems, we usually need to:

- break the problem down into steps
- prompt your LM well until each step works well in isolation
- tweak the steps to work well together
- generate synthetic examples to tune each step
- use these examples to finetune smaller LMs to cut costs

DSPy provides an algorithmical way to optimize LM prompts and weights, especially when LMs are used multiple times within a pipeline.

DSPy User Guide: https://dspy-docs.vercel.app

Important Concepts in DSPy

(1) DSPy Signatures

A signature is a declarative specification of input/output behavior of a DSPy module. Signatures allow you to tell the LM what it needs to do, rather than specify how we should ask the LM to do it.

(2) DSPy Modules

Here DSPy provides different modules for different tasks.

(3) DSPy Optimizers

A DSPy optimizer is an algorithm that can tune the parameters of a DSPy program to maximize the metrics you specify, like accuracy.

Let's code it out!

Attention Mechanism

Attention mechanism is the cornerstone of the **transformer** architecture.

The central idea of attention, was first developed by Graves (2013) in the context of handwriting recognition. Bahdanau et al. (2015) extended the idea, named it "attention" and applied it to machine translation.

In 2017, in their influential paper, *Attention Is All You Need*, Ashish Vaswani et al., introduced the transformer architecture built on the **self-attention** mechanism.

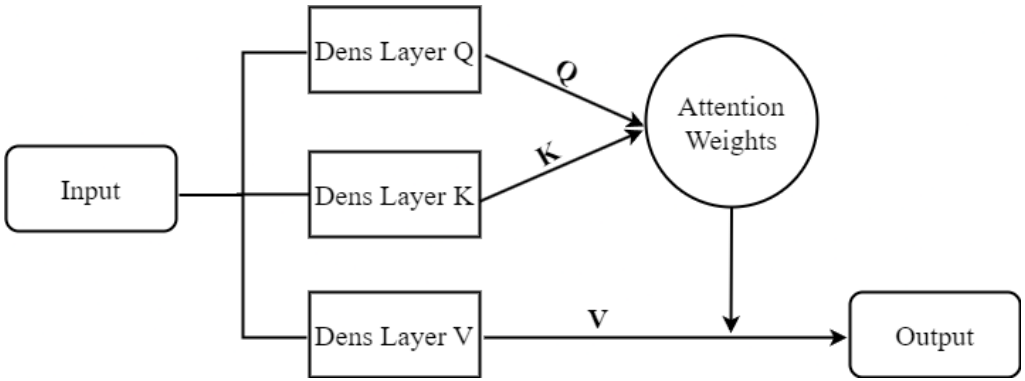
Attention: The Intuition

Consider a sentence: *I would like to have a sandwich for lunch.*

If we ask humans to pick out the words that they should pay most attention to, most likely they will pick "I", "sandwich" and "lunch".

Attention seeks to mimic this mechanism by adding **attention weights** to a model as trainable parameters to pick out important parts of our input.

An Illustration of the Self-attention Mechanism



To generate the Q, K and V vectors, we apply linear transformations (Dens Layer Q, K and V) to the embedded representations of the input.

The linear transformations are defined by learnable weight matrices. These weight matrices are part of the model's parameters and are updated during training.

Attention VS. Self-attention: in self-attention, Q, K and V are generated from the same input. In general attention, they are not.

Attention Function

To generate the output, we need to run the following function:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V$$

Where Q, K and V are different linear transformations of the same input, usually they have the same shape (batch size, n, d). n: number of tokens, d: embedding size.

$softmax(.)$: is a mathematical function that transforms a vector of real numbers into a probability distribution.

Attention Function

Q1: why the dot product QK^T is used to measure the similarity?

We know that $v_i u_j = cos(v_i, u_j)||v_i||_2||u_j||_2$ is a measure of how similar vectors v_i and u_j are. The closer the direction and the larger the length, the greater the dot product.

Similarly, here Q_i and K_i are different projections of the token i into a d dimensional space. We can think about the dot product of these projections as a measure of similarity between token projections.

Attention Function

Q2: why do we need to scale the dot product QK^T by $\sqrt{d}$?
Basically it is used to reduce the variance of the dot product. It is a common technique to ensure numerical stability during training and reduce the risk of vanishing or exploding gradients.

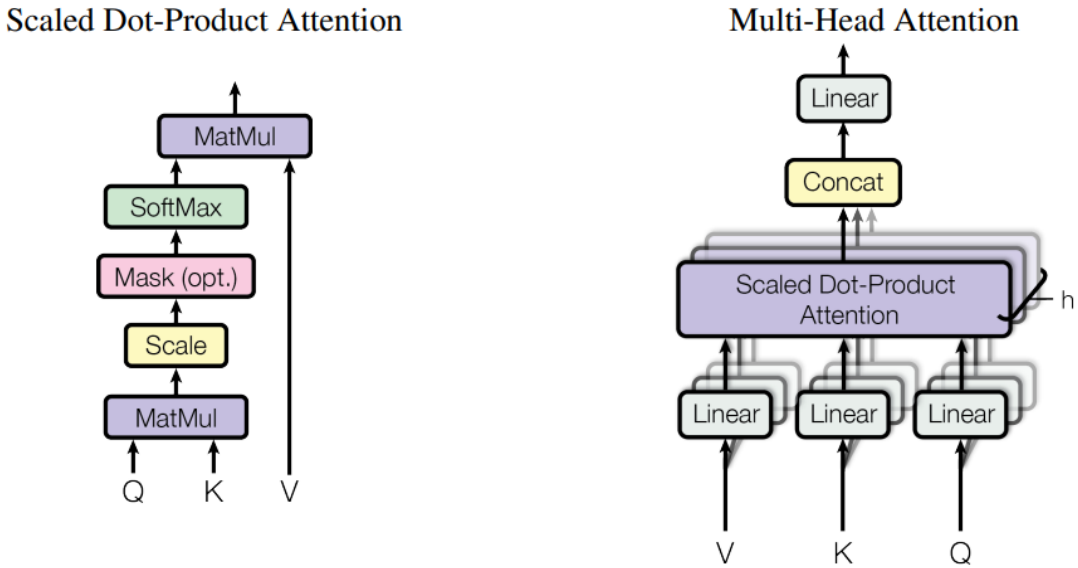
```
# generate some random q, k, v for illustration purpose
n, d = 4, 8   # here we ignore the batch size
q = np.random.randn(n, d)
k = np.random.randn(n, d)
v = np.random.randn(n, d)
np.matmul(q, k.T)

# if we do not scale the dot product
q.var(), k.var(), np.matmul(q, k.T).var()

# if we scale the dot product
scaled = np.matmul(q, k.T) / math.sqrt(d)
q.var(), k.var(), scaled.var()
```

Multi-head Attention

The transformer does not use attention directly. Instead, it employs a mechanism called multi-head attention.



Source: Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

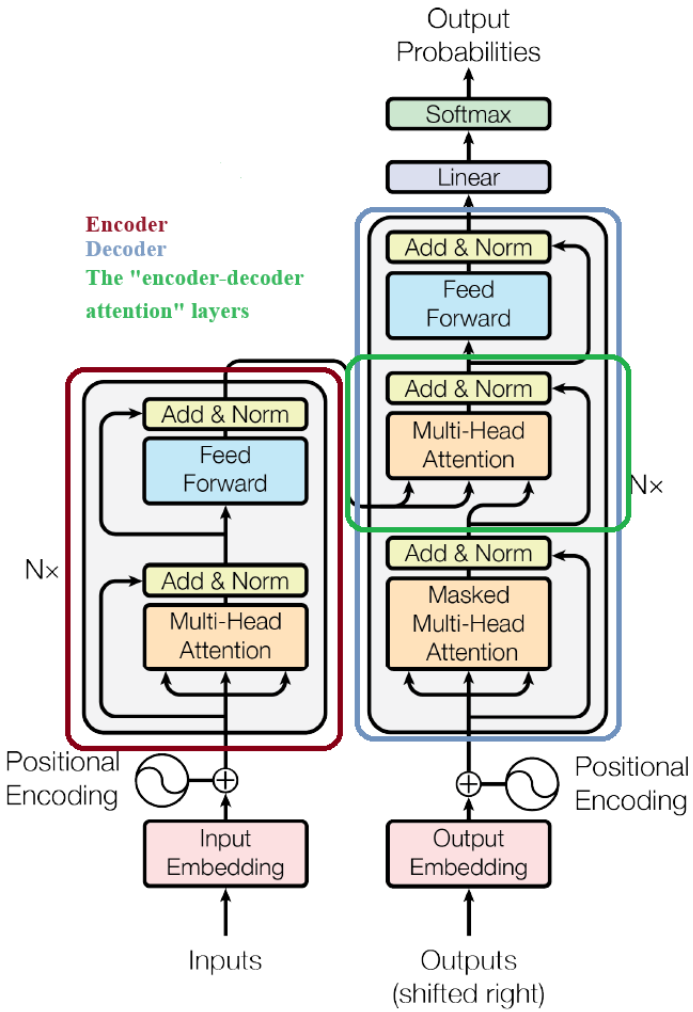
The Multi-Head Self-Attention cuts the Q, K, V into multiple chunks and implement Self-Attention on each chunk, then combines the results together and runs it through a dense layer to get the final output.

Advantages of Multi-head Attention

1. **Improve Model Capacity**: Multi-head attention allows the model to focus on different parts of the input sequence simultaneously. Each head can learn different relationships and patterns in the data, which makes the model more robust and capable of handling complex relationships within the data.

2. **Parallelization**: Multi-head attention can be parallelized, which can lead to faster training and inference times. This is imperative for training a large model with billions of parameters.

Transformer Structure



Source: Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

Encoder-Decoder

- **Encoder**:
  - The encoder is the first part of a sequence-to-sequence model, and its primary function is to process and represent the input data.
  - Besides using Attention, the encoder can also be one or more layers of RNN or LSTM, etc.
- **Decoder**:
  - The decoder is the second part of a sequence-to-sequence model, and its primary role is to generate an output sequence based on the encoded input data.
  - The decoder is typically designed to be autoregressive. It generates one element of the output sequence at a time while conditioning on the previously generated elements. (i.e., text generation tasks)
  - Besides using Attention, the decoder can also be one or more layers of RNN or LSTM, etc.

Applications of Attention in Transformer

The Transformer uses multi-head attention in three different ways:

- The encoder contains self-attention layers. In a self-attention layer all of the K, V and Q come from the same input (i.e., the output of the previous layer).
- The decoder contains self-attention layers. In a self-attention layer all of the K, V and Q come from the same input with a mask layer to prevent leftward inf ormation flow in the decoder. Clearly, we do not want the decoder to read the right answer before it makes a prediction.
- In "encoder-decoder attention" layer, Q comes from the previous decoder layer, K and V come from the output of the encoder.

Miscellaneous

- **Positional Encoding**: enable the model to make use of the order of the sequence, and improve the model performance.
- **Residual connection & layer normalization**: improve the model performance.
- How to code out the Transformer architecture is beyond the scope of this lecture, but you can find numerous replications online. A good starting point is the Paper With Code Website.

Next, we will discuss two important models built on the Transformer architecture: BERT and GPT.

Bidirectional Encoder Representations from Transformers (BERT)

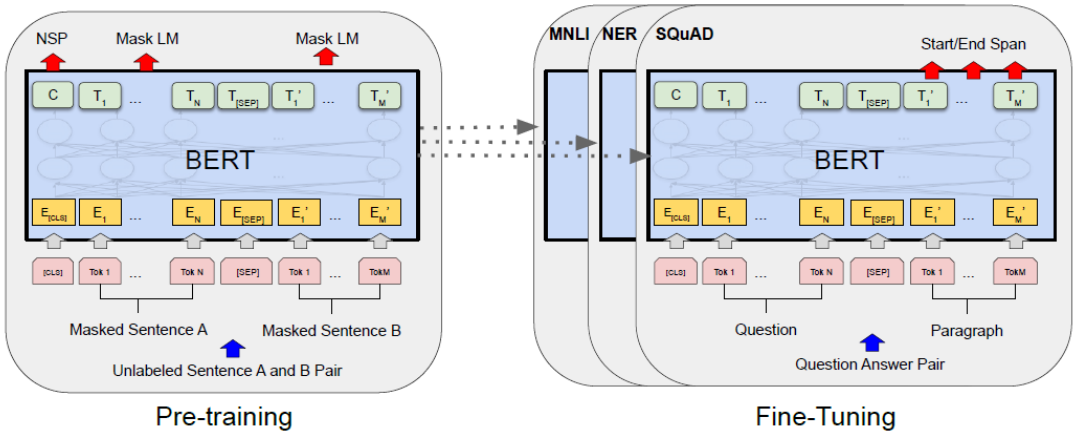
- BERT was developed by Google AI in 2018.
- BERT only utilizes the encoder part of the Transformer.
- BERT's key innovation lies in its bidirectional training approach by using a "masked language model" pre-training objective, inspired by the Cloze task (Taylor, 1953).

BERT Architecture

There are two steps in the BERT framework:

Pre-training: The model is first trained on unlabeled data over two unsupervised pre-training tasks (Masked Language Modeling and Next Sequence Prediction)

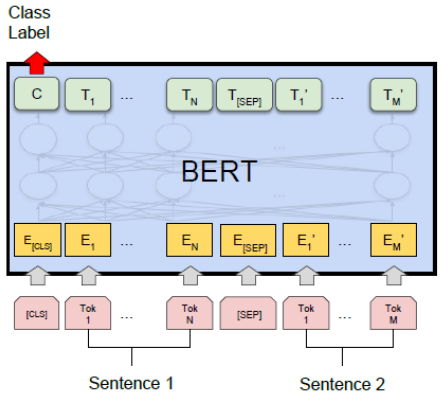
Fine-tuning: Then the parameters are fine-tuned using labeled data from downstream tasks like Text Similarity, Classification, etc.



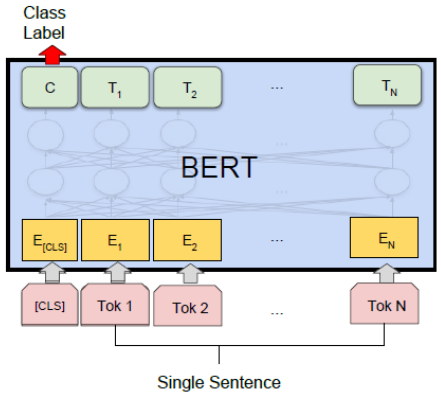
Source: Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

BERT Applications

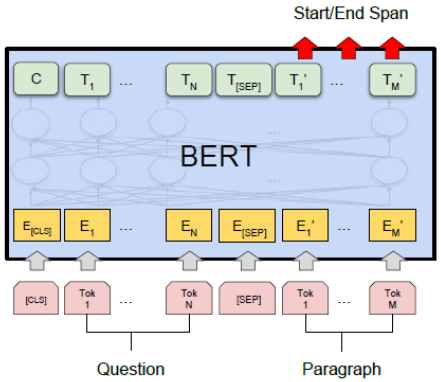
BERT is specifically trained for the following tasks:



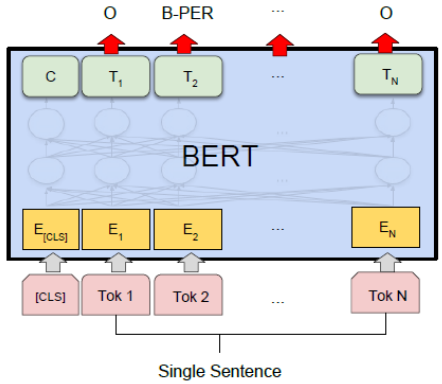
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA



(c) Question Answering Tasks: SQuAD v1.1



(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Source: Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

Generative Pre-trained Transformer (GPT)

AB

Write me a short introduction (under 200 words) to GPT.

Generative Pre-trained Transformer, commonly known as GPT, represents a groundbreaking advancement in natural language processing (NLP). GPT is a family of deep learning models developed by OpenAI that has revolutionized how we interact with and understand text. These models are pre-trained on vast amounts of text data, enabling them to generate human-like text and perform a wide range of NLP tasks.

GPT's key innovation lies in its ability to predict the next word in a sentence, which results in a model that can understand context, grammar, and semantics in a given text. It uses a transformer architecture, which excels at handling sequential data and capturing long-range dependencies in language.

Applications of GPT span various domains, from automated content generation and language translation to chatbots and text summarization. In academia, GPT has catalyzed research in areas such as language understanding, sentiment analysis, and text generation. It has also played a significant role in advancing the field of machine learning and has spurred numerous research papers and developments.

For more comprehensive information on GPT, you can visit the official OpenAI website: OpenAI GPT. Confidence level: High.

GPT Introduction (Human Version)

- The first GPT model was developed by OpenAI in 2018. Now the latest GPT models are GPT-4o and OpenAI o1.
- In the business realm, GPT has truly excelled as the most successful large language model. ChatGPT, the chatbot built on the GPT model, is a standout example of this achievement.
- In contrast to BERT, GPT only uses the decoder part of the Transformer.
- The latest GPT-4o model is a closely guarded business secret. Given the model contains more than 200 billion parameters, it is beyond the scope of this lecture to discuss it in details.

Application of GPT

- **Text Generation:** GPT can generate coherent and contextually relevant text, making it valuable for creative writing and content creation.
- **Language Translation:** GPT can be fine-tuned for language translation tasks, aiding in machine translation.
- **Question Answering:** GPT can understand and answer questions based on provided text, used in chatbots and virtual assistants.
- **Text Summarization:** GPT can generate concise summaries of longer texts, useful in news aggregation and document summarization.
- **Sentiment Analysis:** GPT can analyze and understand sentiment in text, benefiting market research and customer feedback analysis.

Transformer, BERT & GPT

The relationships between Transformer, BERT and GPT

- BERT and GPT are both built on the Transformer architecture.
- BERT focuses on bidirectional contextual understanding of language, as it trains on both the left and right contexts of a word. This is particularly useful for tasks like text classification.
- GPT, on the other hand, is designed for autoregressive text generation. It predicts the next word in a sentence based on the preceding context and is known for its strong language generation capabilities.

Extra Learning Material

3Blue1Brown has produced some amazing educational videos on the transformer framework, check them out:

(1) 3Blue1Brown-But what is a GPT? Visual intro to transformers

(2) 3Blue1Brown-Attention in transformers, visually explained