Unit-4

Bidirectional Encoder Representations from Transformers (BERT)

BERT (Bidirectional Encoder Representations from Transformers) leverages a transformer-based neural network to understand and generate human-like language. BERT employs an encoder-only architecture. In the original Transformer architecture, there are both encoder and decoder modules. The decision to use an encoder-only architecture in BERT suggests a primary emphasis on understanding input sequences rather than generating output sequences.

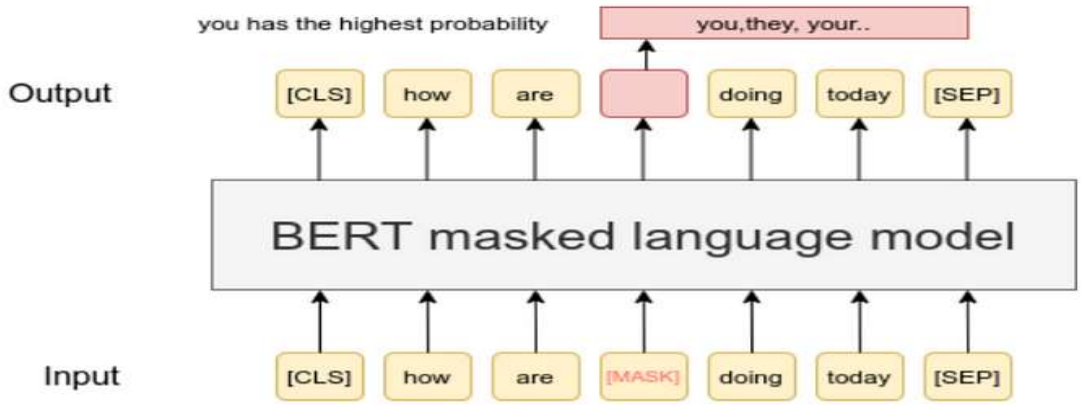
Working

At its core, BERT is powered by a powerful neural network architecture known as Transformers. This architecture incorporates a mechanism called self-attention, allowing BERT to weigh the significance of each word based on its context, both preceding and succeeding. This context-awareness imbues BERT with the ability to generate contextualized word embeddings, which are representations of words considering their meanings within sentences. It's akin to BERT reading and re-reading the sentence to gain a deep understanding of every word's role.

Consider the sentence: "The 'lead' singer will 'lead' the band." Traditional models might struggle with the ambiguity of the word "lead." BERT, however, effortlessly distinguishes that the first "lead" is a noun, while the second is a verb, showcasing its prowess in disambiguating language constructs.

In the chapters to come, we will embark on a journey that demystifies BERT, taking you from its foundational concepts to its advanced applications. You'll explore how BERT is harnessed for various NLP tasks, learn about its attention mechanism, delve into its training process, and witness its impact on reshaping the NLP landscape.

As we delve into the intricacies of BERT, you'll find that it's not just a model; it's a paradigm shift in how machines comprehend the essence of human language. So, fasten your seatbelts as we embark on this enlightening expedition into the world of BERT, where language understanding transcends the ordinary and achieves the extraordinary.



Before BERT can work its magic on text, it needs to be prepared and structured in a way that it can understand. In this chapter, we'll explore the crucial steps of preprocessing text for BERT, including tokenization, input formatting, and the Masked Language Model (MLM) objective.

Tokenization: Breaking Text into Meaningful Chunks

Imagine you're teaching BERT to read a book. You wouldn't hand in the entire book at once; you'd break it into sentences and paragraphs. Similarly, BERT needs text to be broken down into smaller units called tokens. But here's the twist: BERT uses Word Piece tokenization. It splits words into smaller pieces, like turning "running" into "run" and "ning." This helps handle tricky words and ensures that BERT doesn't get lost in unfamiliar words.

Input Formatting: Giving BERT the Context

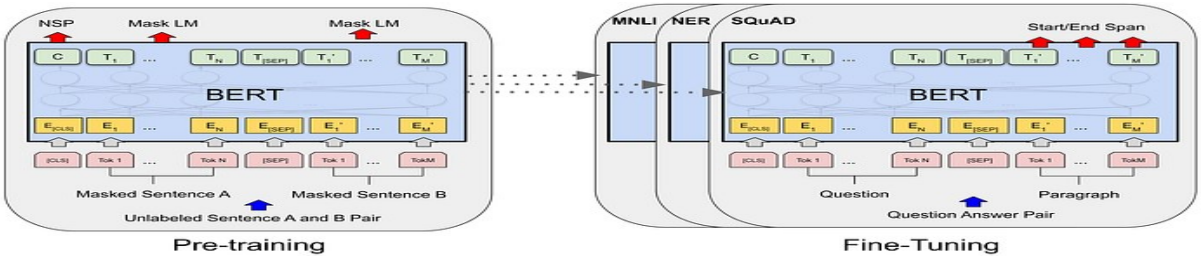
BERT loves context, and we need to serve it to him on a platter. To do that, we format the tokens in a way that BERT understands. We add special tokens like [CLS] (stands for classification) at the beginning and [SEP] (stands for separation) between sentences. As Shown in the Figure (Machine Language Model). We also assign segment embeddings to tell BERT which tokens belong to which sentence.

Masked Language Model (MLM) Objective: Teaching BERT Context

BERT's secret sauce lies in its ability to understand the bidirectional context. During its training, some words are masked (replaced with [MASK]) in sentences, and BERT learns to predict those words from their context. This helps BERT grasp how words relate to each other, both before and after. As Shown in the Figure (Machine Language Model)

Example: Original Sentence: "The cat is on the mat." Masked Sentence: "The [MASK] is on the mat."

Fine-Tuning BERT for Specific Tasks



After understanding how BERT works, it's time to put its magic to practical use. In this chapter, we'll explore how to fine-tune BERT for specific language tasks. This involves adapting the pre-trained BERT model to perform tasks like text classification.

Fine Tuning for downstream tasks

Fine-tuning in NLP is a process where a pre-trained language model (like BERT, GPT, T5, etc.) is adapted for a specific downstream task using a labeled dataset. This approach leverages the knowledge captured during pre-training on a large corpus and refines it for specialized tasks.

Steps for Fine-Tuning NLP Models

1. Select a Pre-trained Model

- Choose a Transformer-based model like **BERT, RoBERTa, GPT, T5, or LLaMA** depending on your task.
- Example:
  - **BERT/RoBERTa** → Classification, Named Entity Recognition (NER)
  - **GPT** → Text Generation
  - **T5/BART** → Summarization, Translation

2. Prepare the Dataset

- Collect a labeled dataset suited for your task.
- Preprocess the text:
  - Tokenization using the model's tokenizer (e.g., `BertTokenizer`, `T5Tokenizer`).
  - Padding and truncation to maintain a fixed input size.

3. Define the Task

- Common NLP downstream tasks:
  - **Text Classification** (e.g., sentiment analysis, spam detection)
  - **Named Entity Recognition (NER)** (e.g., recognizing entities like names, locations)
  - **Question Answering** (e.g., SQuAD-style datasets)
  - **Text Generation** (e.g., summarization, chatbot responses)
  - **Machine Translation** (e.g., English to French)

4. Load and Fine-Tune the Model

- Use Hugging Face's `transformers` library or TensorFlow/PyTorch to fine-tune the model.

```
{ ***
from transformers import BertForSequenceClassification, Trainer, TrainingArguments
from transformers import BertTokenizer
from datasets import load_dataset
model_name = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(model_name)
dataset = load_dataset("imdb")
tokenized_dataset = dataset.map(lambda x: tokenizer(x["text"], padding="max_length",
truncation=True), batched=True)
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)
training_args = TrainingArguments(output_dir="./results", num_train_epochs=3,
per_device_train_batch_size=8)
trainer = Trainer(model=model, args=training_args, train_dataset=tokenized_dataset["train"])
trainer.train()
***}
```

5. Evaluate the Model

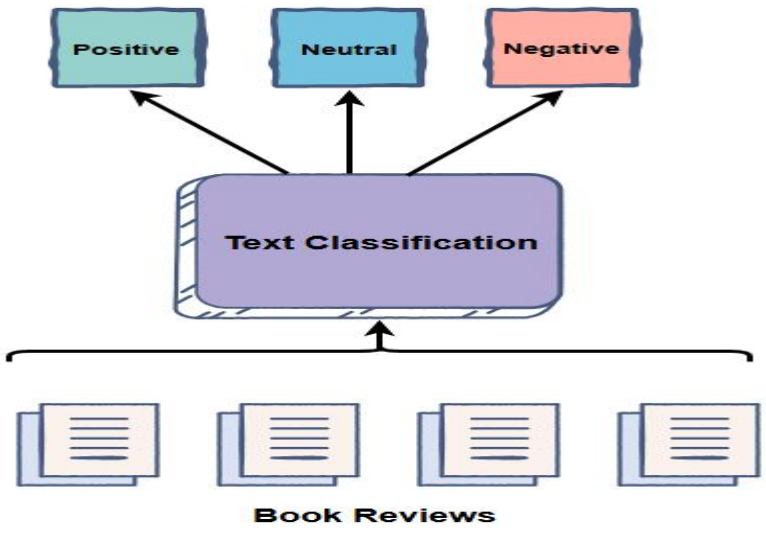
- Use metrics like accuracy, F1-score, BLEU score (for translation), ROUGE (for summarization)

6. Deploy the Model

- Convert the fine-tuned model to **ONNX** or **TorchScript** for optimization.
- Deploy using FastAPI, Flask, or Hugging Face's Inference API.

Text classification

Text Classification is the processing of labeling or organizing text data into groups. It forms a fundamental part of Natural Language Processing. In the digital age that we live in we are surrounded by text on our social media accounts, in commercials, on websites, Ebooks, etc. The majority of this text data is unstructured, so classifying this data can be extremely useful.



Applications

Text Classification has a wide array of applications. Some popular uses are:

- Spam detection in emails
- Sentiment analysis of online reviews
- Topic labeling documents like research papers
- Language detection like in Google Translate
- Age/gender identification of anonymous users
- Tagging online content
- Speech recognition used in virtual assistants like Siri and Alexa

Approaches

Text Classification can be achieved through three main approaches:

1. **Rule-based approaches**
   These approaches make use of handcrafted linguistic rules to classify text. One way to group text is to create a list of words related to a certain column and then judge the text based on the occurrences of these words. For example, words like “fur”, “feathers”, “claws”, and “scales” could help a zoologist identify texts talking about animals online. These approaches require a lot of domain knowledge to be extensive, take a lot of time to compile, and are difficult to scale.
2. **Machine learning approaches**
   We can use machine learning to train models on large sets of text data to predict categories of new text. To train models, we need to transform text data into numerical data – this is known as feature extraction. Important feature extraction techniques include bag of words and n-grams. There are several useful machine learning algorithms we can use for text classification. The most popular ones are:
   - Naive Bayes classifiers
   - Support vector machines
   - Deep learning algorithms
3. **Hybrid approaches**
   These approaches are a combination of the two algorithms above. They make use of both rule-based and machine learning techniques to model a classifier that can be fine-tuned in certain scenarios.

Text Generation

Text generation is a subfield of natural language processing (NLP) that deals with generating text automatically. It has a wide range of applications, including machine translation, content creation, and conversational agents.

Popular applications of text generation in NLP

One popular application of text generation is machine translation, where the model is trained to translate text from one language to another. Another application is content creation, where the model can generate articles, summaries, or social media posts.

Conversational agents, such as chatbots or virtual assistants, also use text generation to produce responses to user inputs. These models are trained on a large dataset of conversational exchanges and can generate appropriate responses based on the context of the conversation.

Text generation has the potential to improve many aspects of our lives, from making it easier to communicate with people who speak different languages to helping businesses generate content more efficiently. However, there are also concerns about the potential for text generation models to produce biased or inaccurate content, so it is essential to carefully consider the ethical implications of these models.

Overall, text generation is a rapidly growing and important area of NLP with numerous applications. As technology advances, we can expect to see even more exciting developments in this field.

Text generation in Python

There are several libraries and frameworks available for text generation in Python. One popular library for natural language processing (NLP) tasks, including text generation, is NLTK (Natural Language Toolkit). NLTK provides a range of tools for preprocessing, tokenization, and stemming, as well as language models and text generation algorithms.

Another popular library for text generation in Python is GPT-3 (Generative Pre-trained Transformer 3), which is a state-of-the-art language model developed by OpenAI. GPT-3 can generate coherent and natural-sounding text in a variety of languages and can be used for tasks such as translation, summarization, and content creation.

Other Python libraries and frameworks that can be used for text generation include TensorFlow, Keras, and PyTorch. These libraries provide tools for building and training neural network models, which can be used for text-generation tasks.

NLTK for text generation

One way to generate text using NLTK is to use a statistical language model, such as an n-gram model. An n-gram model is a language model that predicts the likelihood of a word or sequence of words based on the previous n-1 words in the sequence. To generate text using an n-gram model, you can sample from the distribution of words predicted by the model and select the most likely words based on the context.

Another approach to text generation using NLTK is a Hidden Markov Model. A Markov model is a statistical model that predicts the likelihood of a sequence of words based on the previous words in the sequence. To generate text using a Markov model, you can sample from the distribution of words predicted by the model and select the most likely words based on the context.

To get started with text generation using NLTK, you will need to install the library and familiarize yourself with its language modelling and text generation functions. You will also need a text dataset for your model to use as training data. Once you have these resources, you can build and train your text generation model using NLTK.

Tensorflow, Keras, and PyTorch

Tensorflow, Keras, and PyTorch are popular open-source software libraries for machine learning that can be used to develop and train generative models for text generation. There are several approaches to using these libraries for generative text, including:

Sequence-to-sequence models: These models are trained to map input sequences to output sequences and can be used to generate text by feeding in a seed phrase and generating the next word or phrase in the sequence.

Language models: These models are trained to predict the next word in a sequence based on the context of the previous words. Language models can generate text by sampling from the model's predictions at each time step.

Variational autoencoders (VAEs): VAEs are a generative model that can generate text by learning to reconstruct a given input sequence and then sampling from the latent space to generate new sequences.