Member-only story

The Ultimate Guide to Training BERT from Scratch: Introduction

Demystifying BERT: The definition and various applications of the model that changed the NLP landscape.

Dimitris Poulopoulos · Following

Published in Towards Data Science · 10 min read · Sep 2

Photo by Ryan Wallace on Unsplash

> *Part II and Part III of this story are now live.*

A few weeks ago, I trained and deployed my very own question-answering system using Retrieval Augmented Generation (RAG). The goal was to introduce such a system over my study notes and create an agent to help me connect the dots. LangChain truly shines in these specific types of applications:

As the system's quality blew me away, I couldn't help but dig deeper to understand the wizardry under the hood. One of the features of the RAG pipeline is its ability to sift through mountains of information and find the context most relevant to a user's query. It sounds complex but starts with a simple yet powerful process: encoding sentences into information-dense vectors.

The most popular way to create these sentence embeddings for free is none other than SBERT, a sentence transformer built upon the legendary BERT encoder. And finally, that brings us to the main object of this series: understanding the fascinating world of BERT. What is it? What can you do with it? And the million-dollar question: How can you train your very own BERT model from scratch?

We'll kick things off by demystifying what BERT actually is, delve into its objectives and wide-ranging applications, and then move on to the nitty-

gritty — like preparing datasets, mastering tokenization, understanding key metrics, and, finally, the ins and outs of training and evaluating your model.

This series will be highly detailed and technical, featuring code snippets as well as links to GitHub repositories. By the end, I'm confident you'll gain a deeper understanding of why BERT is regarded as a legendary model in the field of NLP. So, if you share my excitement, grab a colab Notebook, and let's dive in!

> *Learning Rate is a newsletter for those who are curious about the world of ML and MLOps. If you want to learn more about topics like this subscribe here. You'll hear from me on the last Sunday of every month with updates and thoughts on the latest MLOps news and articles!*

The Definition

BERT, which stands for Bidirectional Encoder Representations from Transformers, is a revolutionary Natural Language Processing (NLP) model developed by Google in 2018 (Michael Rupe, How the Google BERT Update Changed Keyword Research). Its introduction marked a significant advancement in the field, setting new state-of-the-art benchmarks across various NLP tasks. For many, this is regarded as the ImageNet moment for the field.

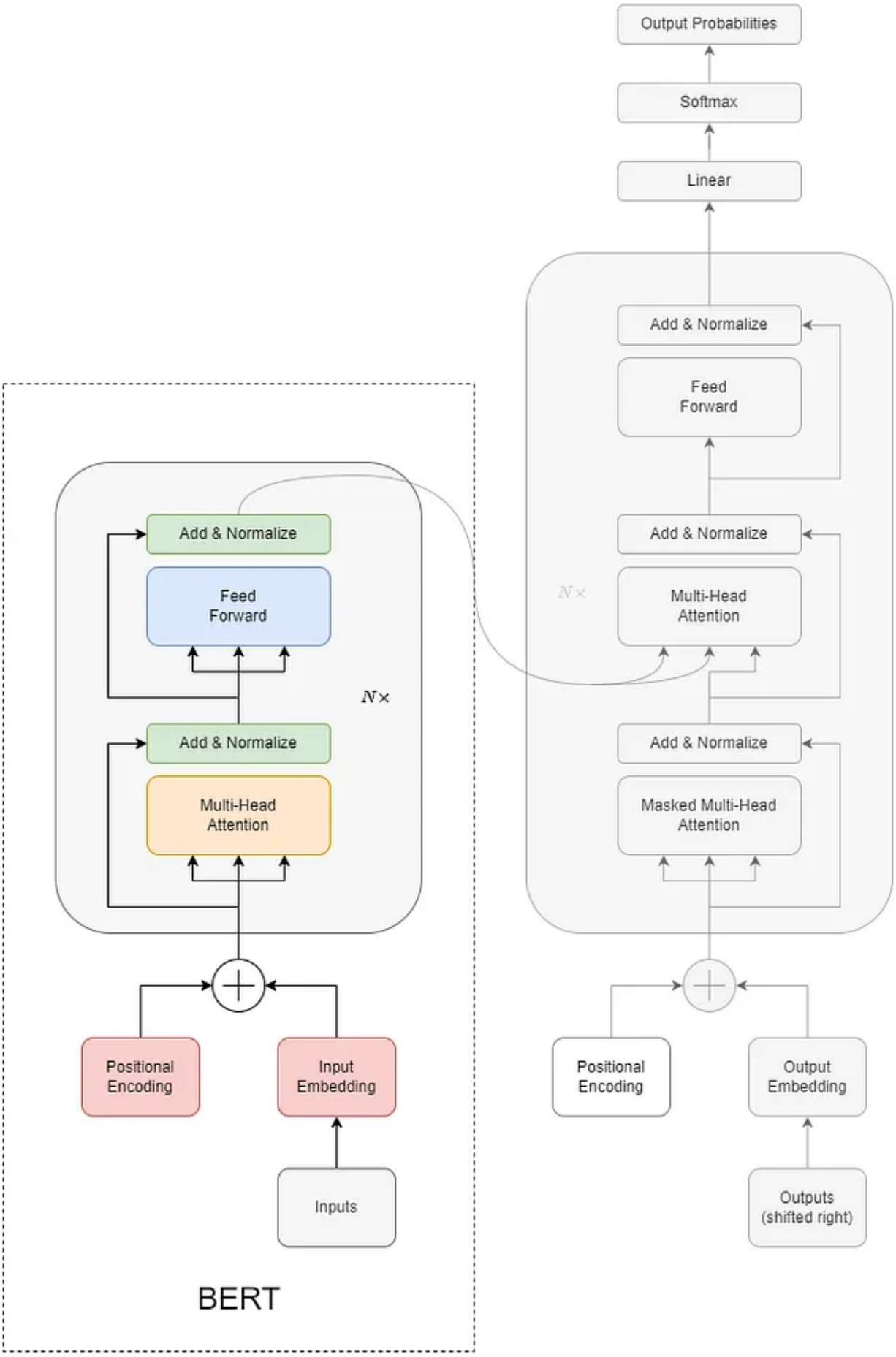
BERT is pre-trained on a massive amount of data, with one goal: to understand what language is and what's the meaning of context in a document. As a result, this pre-trained model can be fine-tuned for specific tasks such as question-answering or sentiment analysis.

So far, so good, but too much theory won't get us anywhere. Thus, let's briefly go through the architecture of BERT over the next section and then take a

closer look at how you pre-train such a model and, more importantly, how you can put it to good use, using the question-answering use case as a concrete example.

The Scaffold

BERT's architecture is based on the Transformer model, which has been particularly influential in the realm of deep learning for NLP tasks. The Transformer model itself is made of an encoder-decoder stack, but BERT only uses the encoder:


Output Probabilities
Softmax
Linear
Add & Normalize
Feed Forward
Add & Normalize
Multi-Head Attention
Add & Normalize
Masked Multi-Head Attention
N×
Positional Encoding
Output Embedding
Outputs (shifted right)
Add & Normalize
Feed Forward
Add & Normalize
Multi-Head Attention
N×
Positional Encoding
Input Embedding
Inputs
BERT

Bert Architecture — Image by Author

Let's see what each colored block in the architecture signifies:

- **Input Embeddings:** Input tokens (words or subwords) are transformed into embeddings, which are then fed into the model. BERT combines both token and positional embeddings as input.
- **Positional Encodings:** Since BERT and the underlying Transformer architecture do not have any built-in sense of word order (as recurrent models like LSTMs do), they incorporate positional encodings to give the model information about the position of words in a sequence. In the original Transformer paper, the positional encodings were fixed beforehand; however, nowadays, it's much more common to learn these encodings during training.
- **Attention Mechanism:** One of the primary innovations in the Transformer architecture is the "self-attention mechanism", which allows the model to weigh the importance of different words in a sentence relative to a given word, thereby capturing context. This is crucial for BERT's bidirectionality.
- **Feed-Forward Neural Network:** Each Transformer block contains a feed-forward neural network that operates independently on each position.
- **Layer Normalization & Residual Connections:** Each sub-layer (like self-attention or feed-forward neural network) in the model includes a residual connection around it followed by layer normalization. This helps in training deep networks by mitigating the vanishing gradient problem.
- **Multiple Stacks:** BERT's depth is one of its defining characteristics. BERT's "base" version uses 12 stacked Transformer encoders, while the

"large" version uses 24.

Now that we have a mental image in our cache let's proceed to examine more closely how we can train such a model.

Going to School

We usually split the training of BERT into two phases: in the first phase — called pre-training — the goal is to teach the model what language is and how context changes the meaning of words. In the second phase — called fine-tuning — we make it do something actually useful.

Let's examine the two phases separately, using concrete examples and visuals.

Pre-training

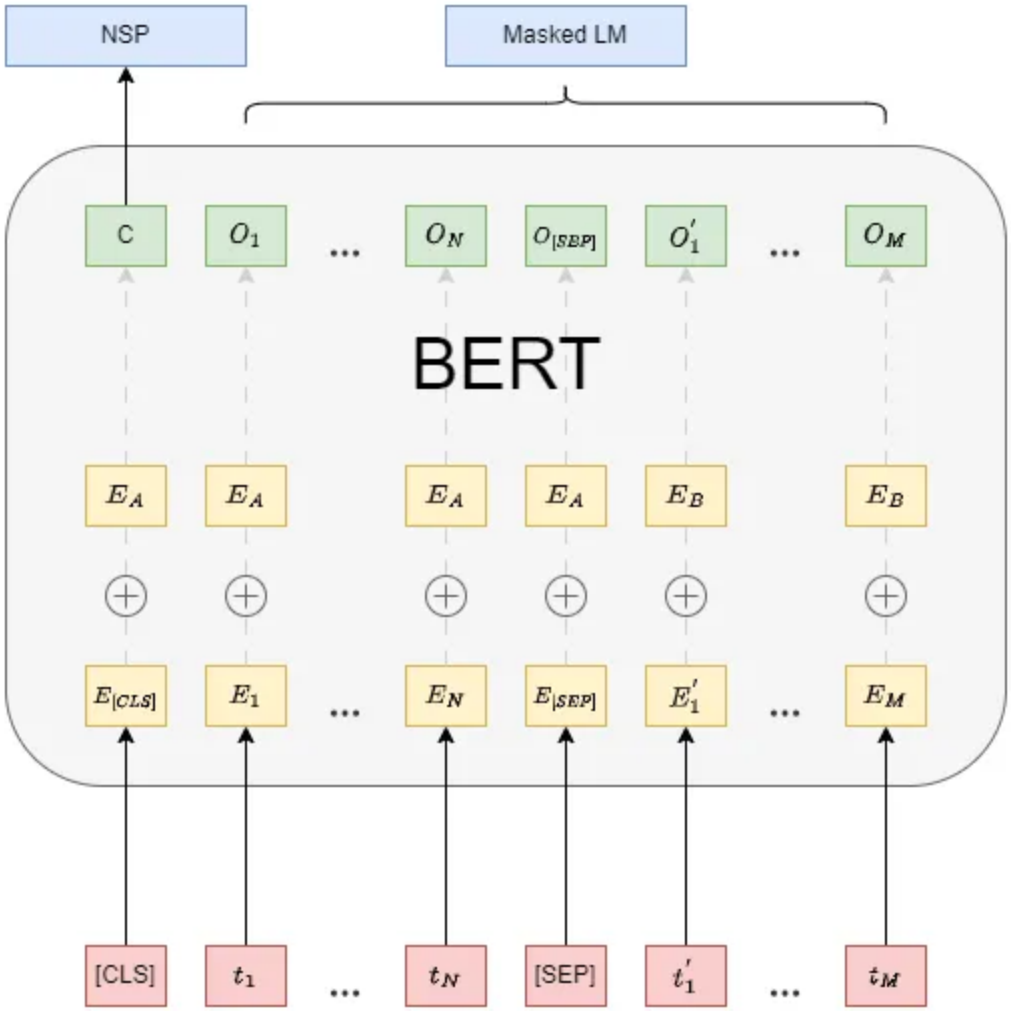
In pre-training, BERT tries to solve two tasks simultaneously: i) masked language model (MLM or "cloze" test) and ii) next-sentence prediction (NSP).

> *The word cloze is derived from closure in Gestalt theory ("Gestalt psychology").*

In the first paradigm, random words in a sentence are replaced with a `[MASK]` token, and BERT tries to predict the original word from the context. This differs from traditional language models, which predict words in a sequence. This is crucial for BERT since it is trying to encode every word in a sequence using information from both directions, left and right, hence "Bidirectional".

For the second task, BERT takes in two sentences, determining if the second sentence follows the first. This helps BERT understand context across sentences. Also, this is where segment embeddings become critical, as they allow the model to differentiate between the two sentences. A pair of sentences fed into BERT for NSP will be assigned different segment embeddings to indicate to the model which sentence each token belongs to.

How does BERT learn these two tasks simultaneously? The following figure will clarify everything:



BERT pre-training — Image by Author

First, we add two special tokens to our sequence: the `[CLS]` token — for classification — and the `[SEP]` token to separate the two sentences. Next, we pass the sequence through BERT and obtain one contextualized representation (i.e., embedding) for each token. If you're paying attention, you'll see two new embeddings: `E_A` and `E_B`. These are two new embeddings we learn during training to inform BERT that there are two separate sequences in the example. We'll see the role of these embeddings later in the fine-tuning phase.

Here comes the final part: First, we take the embedding of the `[CLS]` token and pass it through a new linear layer with two output units to classify it either as a positive label (the sentences are related) or a negative one. Take a look at the HuggingFace `transformers` library source code on GitHub:

```
class BertOnlyNSPHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.seq_relationship = nn.Linear(config.hidden_size, 2)

    def forward(self, pooled_output):
        seq_relationship_score = self.seq_relationship(pooled_output)
        return seq_relationship_score
```

Let's first examine the model's prediction shape for the Masked ML task. After some transformations, the predictions for each token provide a score for every word in the vocabulary. So, the first output will be of shape `[1, 50,000]` if we assume that we have 50,000 words in our vocabulary. Let's see that in the code:

```
class BertPredictionHeadTransform(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        if isinstance(config.hidden_act, str):
            self.transform_act_fn = ACT2FN[config.hidden_act]
        else:
            self.transform_act_fn = config.hidden_act
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_

    def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
        hidden_states = self.dense(hidden_states)
        hidden_states = self.transform_act_fn(hidden_states)
        hidden_states = self.LayerNorm(hidden_states)
        return hidden_states


class BertLMPredictionHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.transform = BertPredictionHeadTransform(config)

        # The output weights are the same as the input embeddings, but there is
        # an output-only bias for each token.
        self.decoder = nn.Linear(config.hidden_size, config.vocab_size, bias=Fal

        self.bias = nn.Parameter(torch.zeros(config.vocab_size))

        # Need a link between the two variables so that the bias is correctly re
        self.decoder.bias = self.bias

    def forward(self, hidden_states):
        hidden_states = self.transform(hidden_states)
        hidden_states = self.decoder(hidden_states)
        return hidden_states


class BertOnlyMLMHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.predictions = BertLMPredictionHead(config)

    def forward(self, sequence_output: torch.Tensor) -> torch.Tensor:
        prediction_scores = self.predictions(sequence_output)
        return prediction_scores
```

Finally, to compute the total loss, we use cross entropy to compute the loss of each task separately, and then we add them together:

```
if labels is not None and next_sentence_label is not None:
    loss_fct = CrossEntropyLoss()
    masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size)
    next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_sente
    total_loss = masked_lm_loss + next_sentence_loss
```

In the final section, let's see how we can fine-tune BERT to produce a question-answering model.

Graduation

Once pre-trained, we can specialize BERT on a specific task using a smaller labeled dataset. For example, let's take a closer look at the Q&A task:

Fine-tuning BERT for question-answering (Q&A) tasks, such as the Stanford Question Answering Dataset (SQuAD), involves adjusting the model to predict the start and end positions of the answer in a given passage for a provided question.

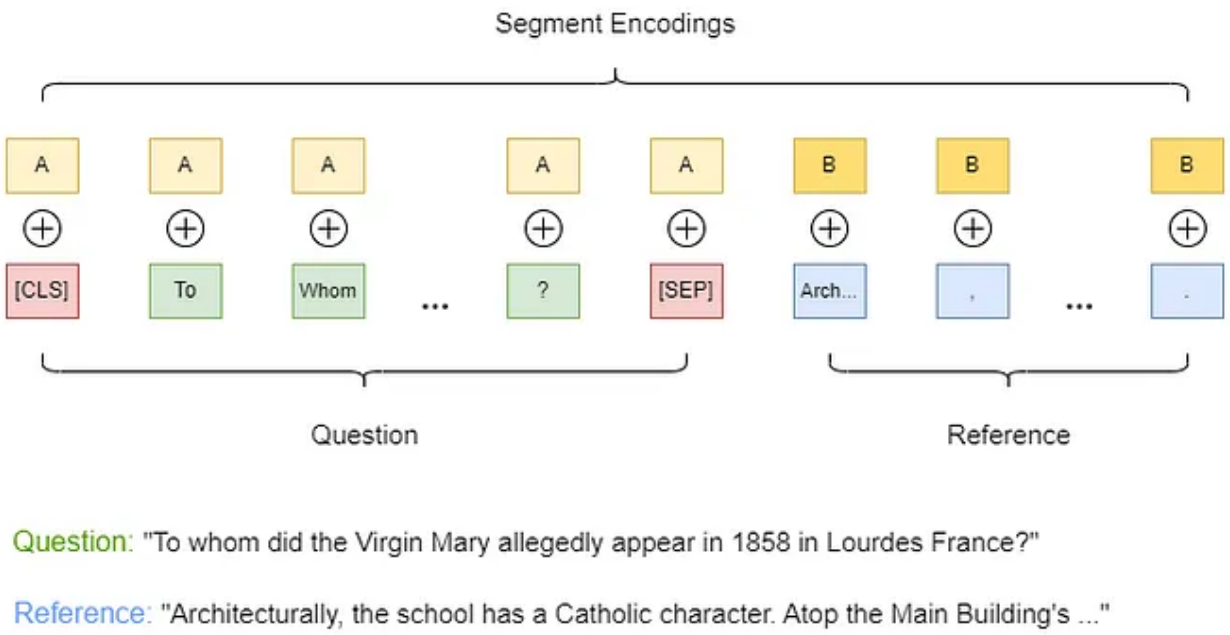
Let's go through the steps to fine-tune BERT for such tasks.

Dataset preparation

Each item in the dataset will typically have a question, a passage (or reference), and the start and end positions of the answer within the passage as the label.

We tokenize the question and the passage into subwords using BERT's tokenizer, separate the question from the passage using the `[SEP]` special token, and start the input sequence using the `[CLS]` special token.

Finally, we create a new array, marking the question as segment `A` and the reference as segment `B`. We will use this information to add the learned segment embeddings later on.



BERT QnA Dataset Preparation — Image by Author

Model modification

Although the pre-trained BERT model can output contextualized embeddings for each token in a sequence, for Q&A tasks, you'll need to derive start and end position predictions from these embeddings.

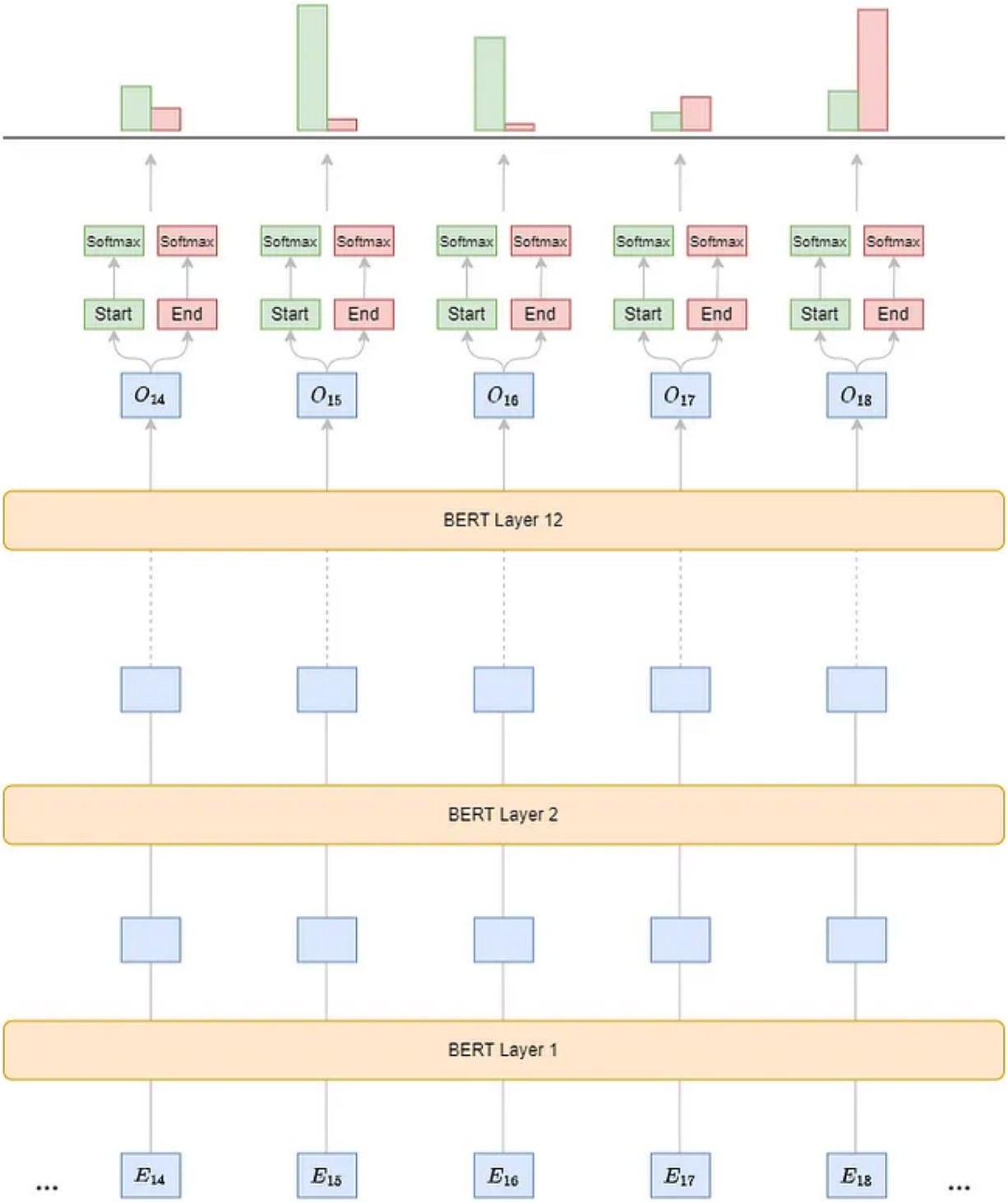
To this end, we add a dense (fully connected) layer on top of BERT, with two output nodes: one for predicting the start position and one for predicting the end position of the answer in the passage.

```
class BertForQuestionAnswering(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
```

Training

For each token in the passage, the model will output a score indicating how likely that token is the starting point of the answer and another score for how likely it is the ending point of the answer.



BERT QnA Prediction — Image by Author

In the figure above, it seems that the model predicts that the answer starts with token `15`, because the model assigned the highest probability to its output and ends with token `18`, for the same reason.

We use a SoftMax function over the entire sequence to get a probability distribution for the start and end positions, and the loss is calculated using the cross entropy between the predictions and the correct start and end positions.

```
loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
start_loss = loss_fct(start_logits, start_positions)
end_loss = loss_fct(end_logits, end_positions)
total_loss = (start_loss + end_loss) / 2
```

Finally, we start with the pre-trained BERT weights and use a smaller learning rate (e.g., `2e-5` or `3e-5`) since BERT is already pre-trained. Too large a learning rate might cause the model to diverge. We fine-tune the model on the Q&A dataset for several epochs until the validation performance plateaus or starts decreasing.

Conclusion

We've covered a lot of ground in this comprehensive guide on training BERT from scratch. Starting from understanding what BERT is, we delved into its architectural intricacies, the logic behind its pre-training and fine-tuning, and even explored how to adapt it for a question-answering task. Along the way, we touched on the key metrics, the importance of tokenization, and the code snippets essential for anyone looking to get their hands dirty in the field of Natural Language Processing.

The story of BERT, however, doesn't end here. In the next story, we'll take things from the start and dive deeper into the BERT tokenizer and how it learns to split words into subwords. Stay tuned!

About the Author

My name is Dimitris Poulopoulos, and I'm a machine learning engineer working for HPE. I have designed and implemented AI and software solutions for major clients such as the European Commission, IMF, the European Central Bank, IKEA, Roblox and others.

If you are interested in reading more posts about Machine Learning, Deep Learning, Data Science, and DataOps, follow me on Medium, LinkedIn, or @james2pl on Twitter.

Opinions expressed are solely my own and do not express the views or opinions of my employer.

Technology | Artificial Intelligence | Deep Learning | Towards Data Science | Hands On Tutorials

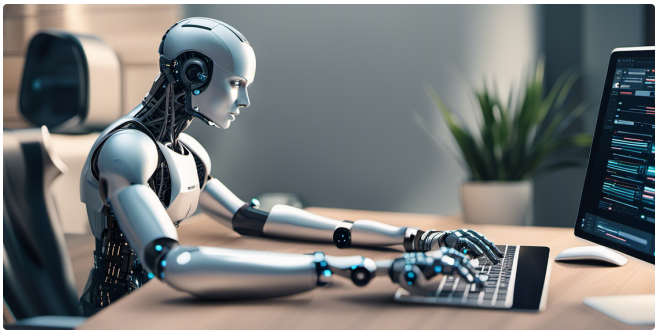
Written by Dimitris Poulopoulos

12.8K Followers · Writer for Towards Data Science

Following

Machine Learning Engineer. I talk about AI, MLOps, and Python programming.
More about me: www.dimpo.me

More from Dimitris Poulopoulos and Towards Data Science



Dimitris Poulopoulos in Towards Data Science

Breaking Boundaries: Exploring Function Calling for LLMs

How function calling paves the way for seamless integration of Large Language...

✨ · 8 min read · Aug 10

80

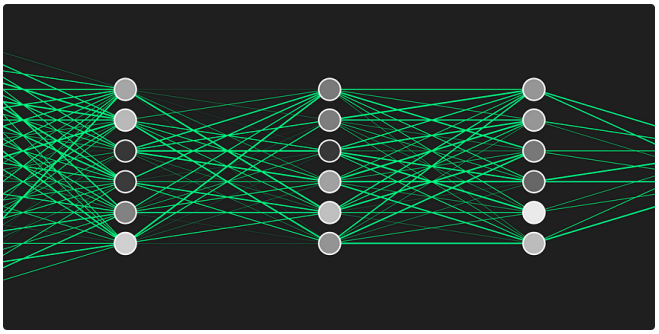
Mike Shakhomirov in Towards Data Science

How to Become a Data Engineer

A shortcut for beginners in 2024

✨ · 17 min read · Oct 7

746 11



Callum Bruce in Towards Data Science

How to Program a Neural Network

A step-by-step guide to implementing a neural network from scratch

Dimitris Poulopoulos in Towards Data Science

The Power of Linux Cgroups: How Containers Take Control of Their...

Optimizing Container Resource Allocation with Linux Control Groups

· 14 min read · Sep 24

489 4

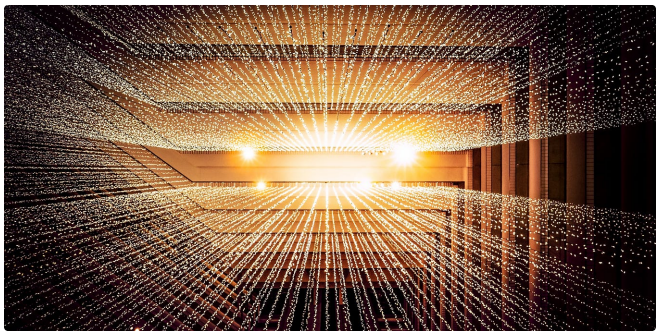
· 8 min read · Jan 10

111 1

See all from Dimitris Poulopoulos

See all from Towards Data Science

Recommended from Medium



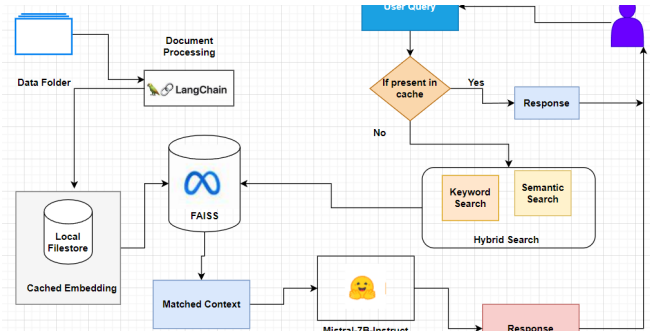
Beatriz Stollnitz in Towards Data Science

Add Your Own Data to an LLM Using Retrieval-Augmented...

Learn how to add your own proprietary data to a pre-trained LLM using a prompt-based...

21 min read · Sep 30

305 3



Plaban Nayak in AI Planet

Advanced RAG Implementation on Custom Data Using Hybrid Searc...

What is RAG ?

40 min read · Oct 8

96 2

Lists

AI Regulation
6 stories · 153 saves

ChatGPT prompts
27 stories · 515 saves

ChatGPT
21 stories · 203 saves

Generative AI Recommended Reading
52 stories · 310 saves

Gathnex

Mistral-7B Fine-Tuning: A Step-by-Step Guide

Introducing Mistral 7B: The Powerhouse of Language Models

5 min read · Oct 4

26 2

Qendel AI in GoPenAI

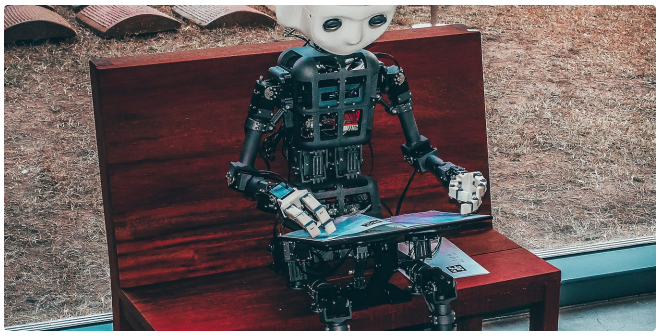
Bye Bye Llama-2, Mistral 7B is Taking Over: Get Started With...

Getting started with Mistral 7B and Langchain integration: A step-by-step guide.

· 7 min read · Oct 1

265 3

Analytics at Meta



Yanli Liu in Level Up Coding

Data engineering at Meta: High-Level Overview of the internal tec...

This article provides an overview of the internal tech stack that we use on a daily bas...

12 min read · Oct 10

1.4K 16

A Step-by-Step Guide to Runing Mistral-7b AI on a Single GPU wit...

How to run your AI efficiently through 4-bit Quantization (with Colab notebook...

✦ · 6 min read · Oct 9

317 3

See more recommendations