datasciencetoday.net

Paper Dissected: “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” Explained

Super User

21–26 minutes

One of the major breakthroughs in deep learning in 2018 was the development of effective transfer learning methods in NLP. One method that took the NLP community by storm was BERT (short for “Bidirectional Encoder Representations for Transformers”). Due to its incredibly strong empirical performance, BERT will surely continue to be a staple method in NLP for years to come.

Though the BERT paper is not an extremely difficult read, it can be difficult to grasp for those without the necessary background. This post will cover BERT as well as some necessary background. Concretely, I will discuss the overall idea of BERT, some important details, and how to use BERT through code examples in PyTorch.

TL;DR

- Language modeling is an effective task for using unlabeled data to pre-train neural networks in NLP
- Traditional language models take the previous n tokens and predict the next one. In contrast, BERT trains a language model

that takes **both the previous and next tokens** into account when predicting.

- BERT is also trained on a next sentence prediction task to better handle tasks that require reasoning about the relationship between two sentences (e.g. question answering)
- BERT uses the Transformer architecture for encoding sentences.
- BERT performs better when given more parameters, even on small datasets.

Transfer Learning in NLP

If you’re already familiar with the basics of transfer learning in NLP and just want to learn more about the details of BERT, you can skip ahead to the next section.

Before methods like ELMo and BERT, pretraining in NLP was limited to word embeddings such as word2vec and GloVe. Word embeddings mapped each word to a vector that represented some aspects of its meaning (e.g. the vector for “King” would include information about status, gender, etc.). Word embeddings are generally trained on large, **unlabeled** corpora (such as the Wikipedia dump), and then used to train models on labeled data for downstream tasks such as sentiment analysis. This allows the downstream models to leverage linguistic information that is learned from larger datasets. Word embeddings were shown to be almost universally useful across a wide range of tasks, but there were many limitations to this simple method.

One limitation is that word embedding models are generally not very powerful. Word2vec and fasttext are both trained on **very**

shallow language modeling tasks, so there is a limitation to what the word embeddings can capture. Unlike LSTMs and other complex architectures, the language models of methods like word2vec have trouble capturing the meaning of combinations of words, negation, etc.

Another key limitation is that word embedding models do not take **context** into account. For instance, the word “bank” can have a different meaning depending on the context (e.g. “I stole money from the bank” vs. “The bank of the river overflowed with water”). However, traditional word embedding methods only allocate a single vector for each word, which is forced to represent this wide range of meanings.

These limitations have motivated the use of **deep language models** (language models that use architectures like LSTMs) for transfer learning. Instead of just training a model to map a single vector for each word, these methods train a complex, deep neural network to map a vector to each word based on the entire **sentence/surrounding context**. Though the precise methods are different, ELMo, ULMFiT, and BERT are all examples of this idea in action. The basic idea is to:

1. Train a deep language model
2. Use the representations learned by the language model in downstream tasks

I’ve previously written a blog post on ELMo, so feel free to read it as well if you are interested. Here, I’ll focus on BERT and how it uses deep language models to improve the performance of models on downstream tasks.

The Basic Idea

Language modeling – although it sounds formidable – is essentially just predicting words in a blank. More formally, given a context, a language model predicts the probability of a word occurring in that context. For instance, given the following context

“The _____ sat on the mat”

where _____ is the word we are trying to predict, a language model might tell us that the word “cat” would fill the blank 50% of the time, “dog” would fill the blank 20% of the time, etc.

Language models have generally been trained from “**left to right**“. They are given a sequence of words, then have to predict the next word.

For instance, if the network is given the sequence

“Which Sesame Street”

the network is trained to predict what word comes next. This approach is effective when we actually want to generate sentences. We can predict the next word, append that to the sequence, then predict the next word, etc..

However, this method is not the only way of modeling language. There is no need to train language models from left to right when we are not interested in generating sentences. This is one of the key traits of BERT: Instead of predicting the next word after a sequence of words, **BERT randomly masks words in the sentence and predicts them**.

Differences in the Language Model Architecture between major transfer learning methods

Why is this method effective? Because this method forces the model to learn how to use information from the entire sentence in deducing what words are missing.

If you are familiar with the NLP literature, you might know about bidirectional LSTM based language models and wonder why they are insufficient. Bidirectional LSTM based language models train a standard left-to-right language model and also train a right-to-left (reverse) language model that predicts previous words from subsequent words. Actually, this is what methods like ELMo and ULMFiT did. In ELMo, there is a single LSTM for the forward language model and backward language model **each**. The crucial difference is this: neither LSTM takes both the previous and subsequent tokens into account at the **same time**.

Forward, Backward, and Masked Language Modeling

This is crucial since this forces the model to use information from the entire sentence **simulatenously** – regardless of the position – to make good predictions.

In addition to this, the authors of BERT use the Transformer instead of the LSTM, a powerful state-of-the-art architecture (for more details on the Transformer, see my blog post here). I’ll give a very high-level overview of the model in this post so don’t worry.

This simple idea combined with a powerful model and effective execution lead to state-of-the-art results across a wide range of NLP tasks.

Now that the overall idea of BERT is clear, let’s delve into the details.

The Details

Though BERT is simple in concept, as always, the devil is in the details. Here, I will explain various design decisions in BERT while showing code examples whenever possible to make the points clearer (code examples are taken from this repo, thanks to the amazing authors!).

GENERAL

One important detail is that BERT uses wordpieces (e.g. playing -> play + ##ing) instead of words. This is effective in reducing the size of the vocabulary and increases the amount of data that is available for each word.

THE MODEL

As I mentioned earlier, BERT uses the Transformer architecture for its underlying model. The Transformer architecture is a model that does not use recurrent connections at all and uses attention over

the sequence instead. The Transformer is composed of multiple attention blocks. Each block transforms the input using linear layers and applies attention to the sequence. If you want to know more in depth, please refer to the blog post I mentioned earlier.

In essence, all you need to know for the rest of this post is that the Transformer basically **stacks** a layer that maps sequences to sequences, just like the LSTM except it is not recurrent and uses a different set of transformations. Therefore, if you input a sequence of n words, the output will be a sequence of n tensors.

Here is the high-level code for the encoder (written in PyTorch):

```
class BertLayer(nn.Module):
    def __init__(self, config):
        super(BertLayer, self).__init__()
        self.attention = BertAttention(config)
        self.intermediate = BertIntermediate(config)
        self.output = BertOutput(config)

    def forward(self, hidden_states, attention_mask):
        attention_output =
self.attention(hidden_states, attention_mask)
        intermediate_output =
self.intermediate(attention_output)
        layer_output =
self.output(intermediate_output, attention_output)
        return layer_output


class BertEncoder(nn.Module):
    def __init__(self, config):
```

```
        super(BertEncoder, self).__init__()
        layer = BertLayer(config)
        self.layer =
nn.ModuleList([copy.deepcopy(layer) for _ in
range(config.num_hidden_layers)])

    def forward(self, hidden_states, attention_mask,
output_all_encoded_layers=True):
        all_encoder_layers = []
        for layer_module in self.layer:
            hidden_states =
layer_module(hidden_states, attention_mask)
            if output_all_encoded_layers:

all_encoder_layers.append(hidden_states)
        if not output_all_encoded_layers:
            all_encoder_layers.append(hidden_states)
        return all_encoder_layers
```

As input, BERT takes token embeddings as well as a couple of additional embeddings that provide some crucial metadata. One of these embeddings is the **positional embedding**. One limitation of the Transformer architecture is that – unlike RNNs – it cannot take the order of the inputs into account (i.e. it will treat the first and last tokens of the inputs exactly the same if they are the same word). To overcome this problem, BERT learns and uses positional embeddings to express the position of words in a sentence. These embeddings are added to the token embeddings before feeding them into the model.

BERT also takes **segment embeddings** as input. BERT can be

trained on sentence **pairs** for tasks that take sentence pairs as input (e.g. question answering and natural language inference). It learns a unique embedding for the first and second sentences to help the model distinguish between the sentences.

The input schema for BERT is summarized below:

The input schema for BERT

MASKED LANGUAGE MODEL TRAINING

Though masked language modeling seems like a relatively simply task, there are a couple of subtleties to doing it right.

The most naive way of training a model on masked language modeling is to randomly replace a set percentage of words with a special [MASK] token and to require the model to predict the masked token. Indeed, in the majority of cases, this is what BERT is trained to do. For each example, 15% of the tokens are selected uniformly at random to be masked.

The problem with this approach is that the model only tries to predict when the [MASK] token is present in the input. This means the model can “slack-off” when the input token is not the [MASK] token, meaning the hidden state for the input token might not be as rich as it could be. What we really want the model to do is to **try to predict the correct tokens regardless of what token is**

present in the input.

To solve this problem, the authors sometimes replace words in the sentence with random words instead of the [MASK] token. This must be done with caution though since swapping words at random is a very strong form of noise that can potentially confuse the model and degrade results. This is why BERT only swaps 10% of the 15% tokens selected for masking (in total 1.5% of all tokens) and leaves 10% of the tokens intact (it does not mask or swap them). The remaining 80% are actually replaced with the [MASK] token.

NEXT SENTENCE PREDICTION TRAINING

In addition to masked language modeling, BERT also uses a **next sentence prediction** task to pre-train the model for tasks that require an understanding of the *relationship* between two sentences (e.g. question answering and natural language inference).

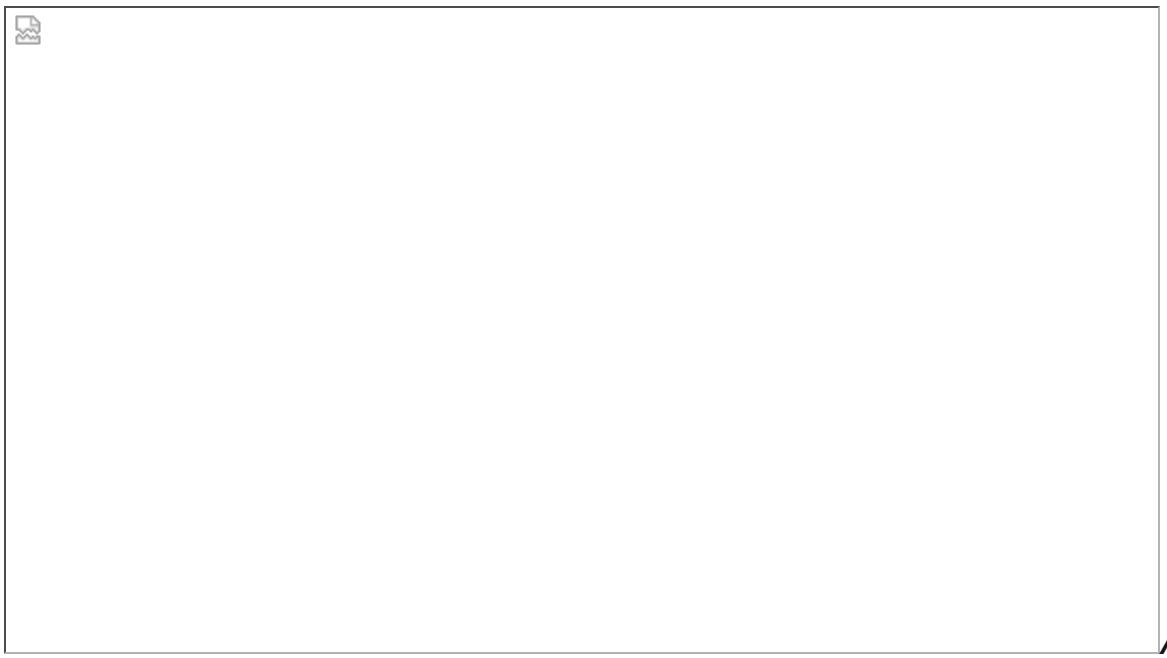
When taking two sentences as input, BERT separates the sentences with a special [SEP] token. During training, BERT is fed two sentences and 50% of the time the second sentence comes after the first one and 50% of the time it is a randomly sampled sentence. BERT is then required to predict whether the second sentence is random or not.

Example inputs for the next sentence prediction task

FINE-TUNING

As I mentioned earlier, the BERT encoder produces a sequence of hidden states. For classification tasks, this sequence ultimately needs to be reduced to a single vector. There are multiple ways of converting this sequence to a single vector representation of a sentence. One is max/mean pooling. Another is applying attention. The authors, however, opt to go with a much simpler method: simply taking the hidden state corresponding to the first token.

To make this pooling scheme work, BERT prepends a [CLS] token (short for “classification”) to the start of each sentence (this is essentially like a start-of-sentence token).



A basic conceptual diagram representing how BERT works

Code:

```
class BertPooler(nn.Module):
    def __init__(self, config):
        super(BertPooler, self).__init__()
        self.dense = nn.Linear(config.hidden_size,
config.hidden_size)
        self.activation = nn.Tanh()

    def forward(self, hidden_states):
        # We "pool" the model by simply taking the
hidden state corresponding
        # to the first token.
        first_token_tensor = hidden_states[:, 0]
        pooled_output =
self.dense(first_token_tensor)
        pooled_output =
self.activation(pooled_output)
```

```
        return pooled_output
```

This sentence representation can then be fed into any arbitrary classifier. The classifier and BERT can be fine-tuned jointly or the classifier can be tuned on top of fixed features extracted from BERT.

HYPERPARAMETERS

As with any deep learning model, hyperparameter settings can make or break the results. For fine-tuning, the authors found the following settings to work well across a wide range of tasks:

- Dropout: 0.1
- Batch size: 32, 16
- Optimizer: Adam
- Learning rate: 5e-5, 3e-5, 2e-5
- Number of epochs: 3, 4

For pretraining, BERT uses the following hyperparameters:

- Sequence length (single example): 256
- Batch size: 512
- Training steps: 1,000,000 (Approximately 40 epochs)
- Optimizer: Adam
- Learning rate: 1e-4
- Learning rate schedule: Warmup for 10,000 steps, then linear decay
- Dropout: 0.1

- Activation function: gelu (Gaussian Error Linear Unit)

The training took 4 days on 16 cloud TPUs (64 TPU chips).

Results

BERT outperformed the state-of-the-art across the following tasks:

- Language understanding
- Natural language inference
- Paraphrase detection
- Sentiment analysis
- Linguistic acceptability analysis
- Semantic similarity analysis
- Textual entailment

BERT not only outperformed traditional word-embedding based approaches but also outperformed new methods such as ELMo. For a detailed breakdown of the results and preprocessing for each task, the original paper is the best source.

Findings

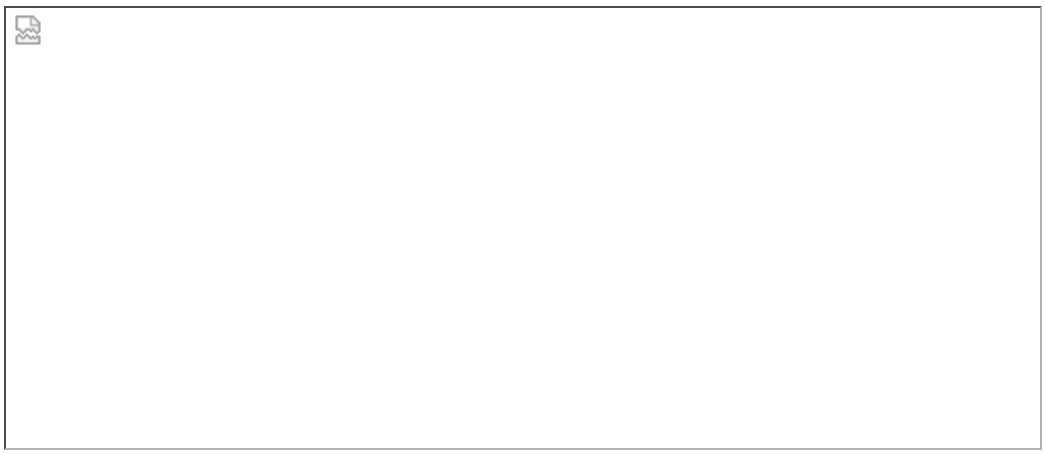
Aside from showing strong empirical results, the authors of BERT also performed ablation studies to further uncover what makes BERT effective. Here are some questions the authors asked as well as their answers.

1. IS MASKED LANGUAGE MODELING REALLY MORE EFFECTIVE THAN SEQUENTIAL LANGUAGE MODELING?

Short answer: Yes.

Long answer:

The authors tried training the Transformer on a left-to-right (LTR) language modeling task instead of the masked language modeling task. The results for this setup can be seen in the third row of the table below (“LTR & No NSP”). The baseline (standard BERT) is on the first row. Across all tasks, the performance is significantly lower when left-to-right language modeling is used instead of masked language modeling.



The results for the ablation study for various pretraining *tasks*

2. IS THE NEXT SENTENCE PREDICTION TASK NECESSARY?

Short answer: Yes.

Long answer:

The authors also tried pretraining without the next sentence prediction task. The results are shown in the table above in the second row (“No NSP”). Unlike masked language modeling, the performance drops for only a subset of the tasks. Concretely, for natural language inference and question answering (the MNLI-m, QNLI, and SQuAD datasets), the next sentence prediction seems

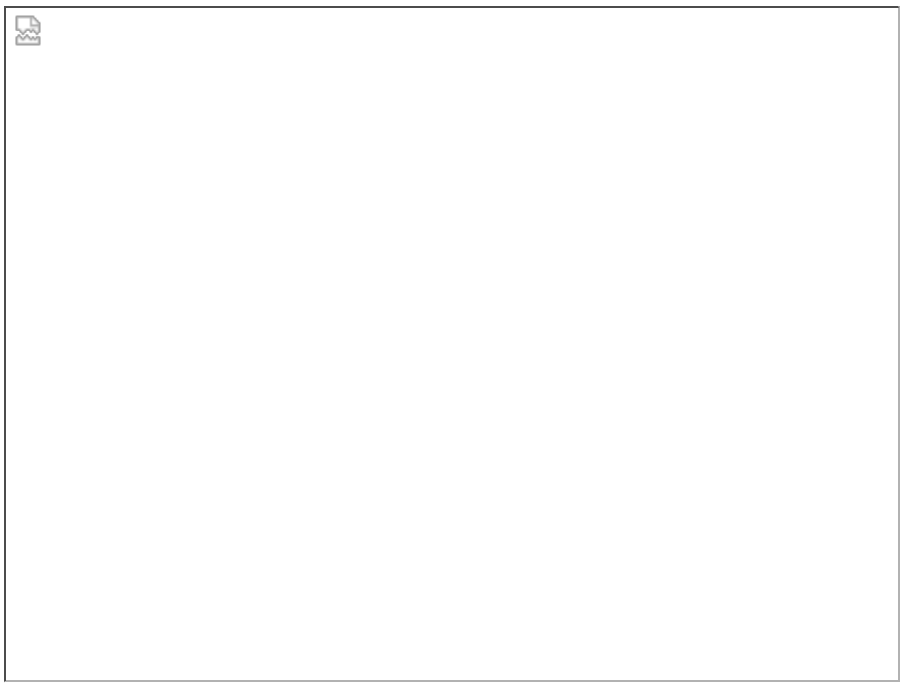
to help a lot. For paraphrase detection (MRPC), the performance change is much smaller, and for sentiment analysis (SST-2) the results are virtually the same.

3. SHOULD I USE A LARGER BERT MODEL (A BERT MODEL WITH MORE PARAMETERS) WHENEVER POSSIBLE?

Short answer: Yes.

Long answer:

The authors experimented with various numbers of layers, hidden activation sizes, and numbers of attention heads. Surprisingly, across all tasks the performance of larger models was better, even for very small datasets like the MRPC dataset.



4. DOES BERT NEED TO BE TRAINED FOR SUCH A LARGE NUMBER OF STEPS?

Short answer: Yes.

Long answer:

For the MNLI dataset, the accuracy changes by nearly an entire percentage point when the number of training steps is reduced to 500,000 (instead of 1,000,000).

5. DOES MASKED LANGUAGE MODELING CONVERGE MORE SLOWLY THAN LEFT-TO-RIGHT LANGUAGE MODELING PRETRAINING (SINCE MASKED LANGUAGE MODELING ONLY PREDICTS 15% OF THE INPUT TOKENS WHEREAS LEFT-TO-RIGHT LANGUAGE MODELING PREDICTS ALL OF THE TOKENS)?

Short answer: Yes and No.

Long answer:

Left-to-right language modeling does converge faster, but masked language modeling achieves a much higher accuracy with the same number of steps. Results are shown below.

The accuracy on the MNLI dataset vs *the number of* pretraining *steps for left-to-right and masked language modeling.*

6. DO I HAVE TO FINE-TUNE THE ENTIRE BERT MODEL? CAN’T I JUST USE BERT AS A FIXED FEATURE EXTRACTOR?

Short answer: Yes, if you want the best performance. But the feature extractor approach isn’t bad either.

Long answer:

The authors tested how a BiLSTM model that used fixed embeddings extracted from BERT would perform on the CoNLL-NER dataset. The results are shown in the table below. It turns out that using a **concatenation of the hidden activations from the last four layers** provides very strong performance, only 0.3 behind finetuning the entire model. For those on a strict computational budget, this feature extraction approach is a good option.

Usage Example

Finally, we’ll go through a concrete example of how to use BERT in practice.

Google has released a Colab notebook detailing how to fine-tune a BERT model in TensorFlow using TPUs. Here, I’ll go through a minimal example of using BERT in PyTorch to train a classifier for the CoLa dataset. For the full code with all options, please refer to this link.

```
# parameters
MAX_SEQ_LEN = 128
NUM_LABELS = 2
NUM_EPOCHS = 3
BS = 32
BERT_MODEL = "bert-large-uncased"
DATA_DIR = "."
PYTORCH_PRETRAINED_BERT_CACHE = "."
```

Our first step will be initializing a tokenizer so that we handle the input in the same way the BERT model did while pretraining.

```
# initialize tokenizer
tokenizer = BertTokenizer.from_pretrained(BERT_MODEL,
do_lower_case=True)
```

Next, we construct the dataset and dataloader (if you’re unfamiliar with datasets and dataloaders, you can refer to my tutorial on torchtext; this is a general pattern in PyTorch so is worth remembering).

```
# construct dataset
processor = processors["cola"]()
train_examples =
processor.get_train_examples(DATA_DIR)
train_features = convert_examples_to_features(
            train_examples, label_list, MAX_SEQ_LEN,
```

```
tokenizer)
all_input_ids = torch.tensor([f.input_ids for f in
train_features], dtype=torch.long)
        all_input_mask = torch.tensor([f.input_mask
for f in train_features], dtype=torch.long)
        all_segment_ids = torch.tensor([f.segment_ids
for f in train_features], dtype=torch.long)
        all_label_ids = torch.tensor([f.label_id for
f in train_features], dtype=torch.long)
        train_data = TensorDataset(all_input_ids,
all_input_mask, all_segment_ids, all_label_ids)

# construct data loader
train_dataloader = DataLoader(train_data,
sampler=RandomSampler(train_data),
batch_size=args.train_batch_size)
```

Next, we load the pretrained model. This is where the magic really happens. You will need to specify where to download the data and prepare the weights.

```
# read pretrained model
model =
BertForSequenceClassification.from_pretrained(BERT_MOD

cache_dir=PYTORCH_PRETRAINED_BERT_CACHE,
              num_labels=NUM_LABELS)
```

Now all we have to do is prepare the optimizer and start training!

```
# Prepare optimizer
param_optimizer = list(model.named_parameters())
```

```
no_decay = ['bias', 'LayerNorm.bias',
'LayerNorm.weight']
optimizer_grouped_parameters = [
        {'params': [p for n, p in param_optimizer if
not any(nd in n for nd in no_decay)], 'weight_decay':
0.01},
        {'params': [p for n, p in param_optimizer if
any(nd in n for nd in no_decay)], 'weight_decay':
0.0}
]
t_total = NUM_EPOCHS * math.ceil(len(train_examples /
BS)
optimizer = BertAdam(optimizer_grouped_parameters,
                     lr=5e-5, warmup=0.1,
                     t_total=t_total)

# train
global_step = 0
model.train()
for _ in trange(int(args.num_train_epochs),
desc="Epoch"):
            tr_loss = 0
            nb_tr_examples, nb_tr_steps = 0, 0
            for step, batch in
enumerate(tqdm(train_dataloader, desc="Iteration")):
                input_ids, input_mask, segment_ids,
label_ids = batch
                loss = model(input_ids, segment_ids,
input_mask, label_ids)
                loss.backward()
```

```
                tr_loss += loss.item()
                nb_tr_examples += input_ids.size(0)
                nb_tr_steps += 1

                # modify learning rate with special
warm up BERT uses
                lr_this_step = 5e-5 *
warmup_linear(global_step/t_total,
args.warmup_proportion)
                for param_group in
optimizer.param_groups:
                    param_group['lr'] = lr_this_step
                optimizer.step()
                optimizer.zero_grad()
```

Further Readings

For those who are interested in learning more about BERT, the best resources are the original paper and the Github repo that open sources the code behind BERT. The blog post by Google Research is also a nice resource.

There is also an implementation of BERT in PyTorch and Chainer which I highly recommend going through.

As always, if you have any questions or find anything wrong with this post, please leave them in the comments! Thanks for reading, and happy transfer learning!

The original blog can be found here.

Author: keitakurita.