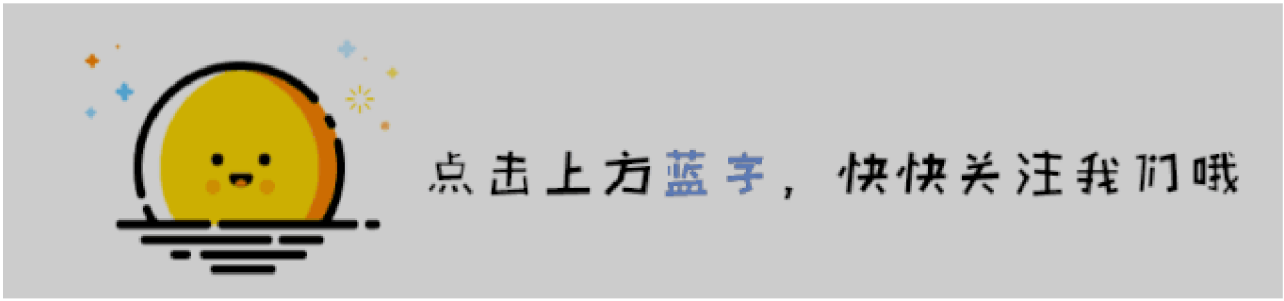
[预训练语言模型专题] Huggingface简介及BERT代码浅析

原创 管扬 朴素人工智能

来自专辑

预训练语言模型

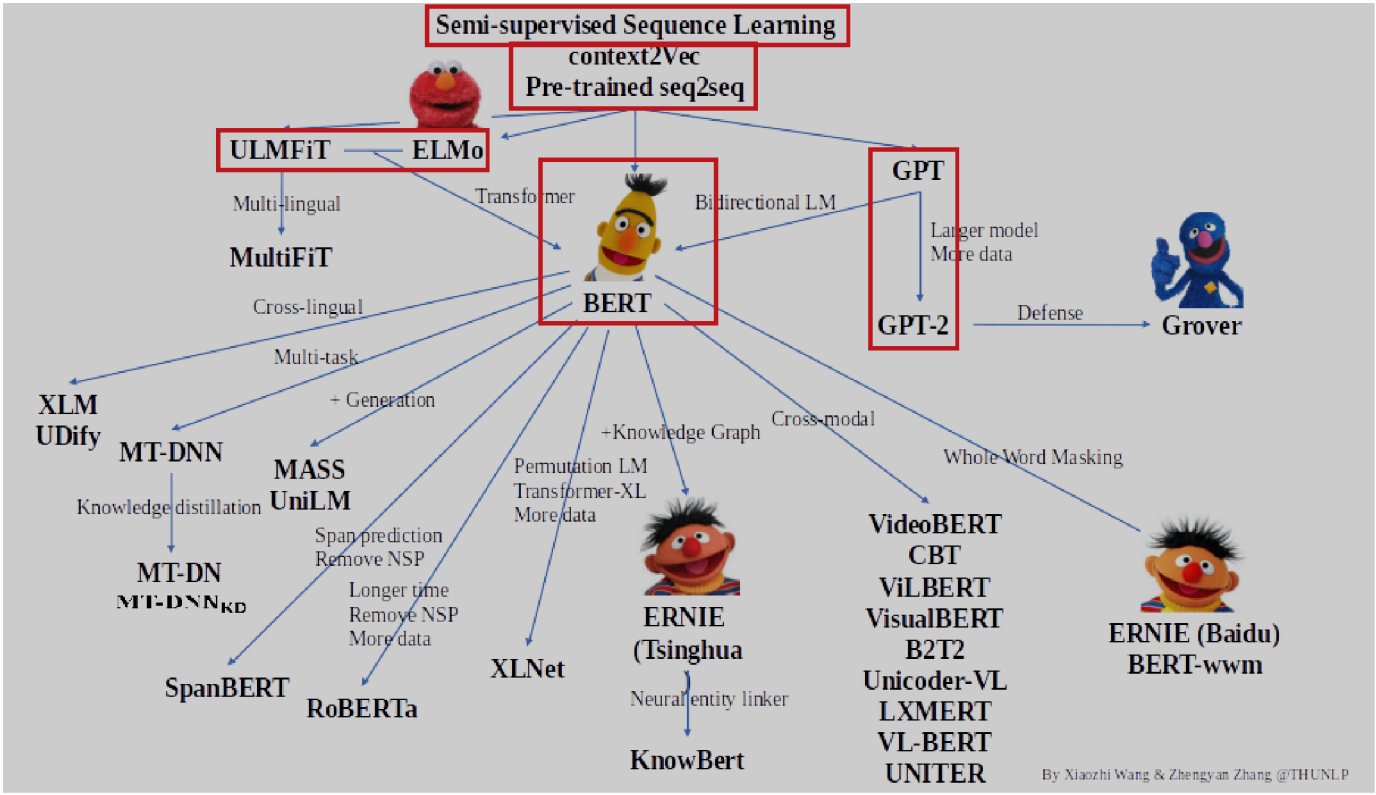


本文为预训练语言模型专题系列第六篇

快速传送门

[萌芽时代]，[风起云涌]，[文本分类通用技巧]，[GPT家族]，[BERT来临]

感谢清华大学自然语言处理实验室对**预训练语言模型架构**的梳理，我们将沿此脉络前行，探索预训练语言模型的前沿技术，红色框为已介绍的文章。本期的内容是结合Huggingface的Transformers代码，来进一步了解下BERT的pytorch实现，欢迎大家留言讨论交流。

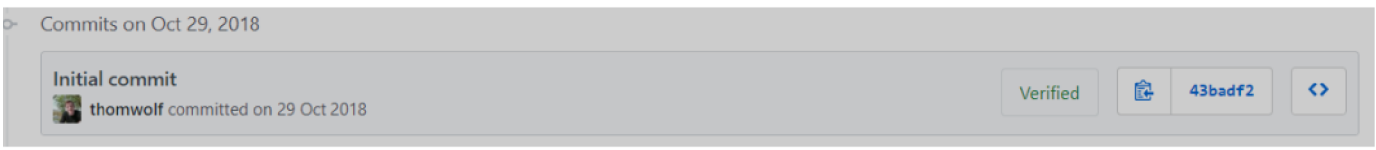


Hugging face 简介

Hugging face🤗 是一家总部位于纽约的聊天机器人初创服务商，开发的应用在青少年中颇受欢迎，相比于其他公司，Hugging Face更加注重产品带来的情感以及环境因素。官网链接在此 https://huggingface.co/ 。

但更令它广为人知的是Hugging Face专注于NLP技术，拥有大型的开源社区。尤其是在github上开源的自然语言处理，预训练模型库 Transformers，已被下载超过一百万次，github上超过**24000**个star。Transformers 提供了NLP领域大量state-of-art的 预训练语言模型结构的模型和调用框架。以下是repo的链接（https://github.com/huggingface/transformers）

这个库最初的名称是**pytorch-pretrained-bert**，它随着BERT一起应运而生。Google2018年10月底在 https://github.com/google-research/bert 开源了BERT的tensorflow实现。当时，BERT以其强劲的性能，引起NLPer的广泛关注。几乎与此同时，pytorch-pretrained-bert也开始了它的第一次提交。pytorch-pretrained-bert 用当时已有大量支持者的pytorch框架复现了BERT的性能，并提供预训练模型的下载，使没有足够算力的开发者们也能够在几分钟内就实现state-of-art-fine-tuning。



因为pytorch框架的友好，BERT的强大，以及pytorch-pretrained-bert的简单易用，使这个repo也是受到大家的喜爱，不到10天就突破了1000个star。在2018年11月17日，repo就实现了BERT的基本功能，发布了版本0.1.2。接下来他们也没闲着，又开始将GPT等模型也往repo

上搬。在2019年2月11日release的 0.5.0版本中，已经添加上了OpenAI GPT模型，以及Google的TransformerXL。

直到2019年7月16日，在repo上已经有了包括BERT，GPT，GPT-2，Transformer-XL，XLNET，XLM在内六个预训练语言模型，这时候名字再叫pytorch-pretrained-bert就不合适了，于是改成了pytorch-transformers，势力范围扩大了不少。这还没完！2019年6月Tensorflow2的beta版发布，Huggingface也闻风而动。为了立于不败之地，又实现了TensorFlow 2.0和PyTorch模型之间的深层互操作性，可以在TF2.0/PyTorch框架之间随意迁移模型。在2019年9月也发布了2.0.0版本，同时正式更名为 transformers 。到目前为止，transformers 提供了超过100种语言的，32种预训练语言模型，简单，强大，高性能，是新手入门的不二选择。

Transfromers中BERT简单运用

前几期里，一直在分享论文的阅读心得，虽然不是第一次看，但不知道大家是不是和我一样又有所收获。本期我们一起来看看如何使用Transformers包实现简单的BERT模型调用。

安装过程不再赘述，比如安装2.2.0版本 pip install transformers==2.2.0 即可，让我们看看如何调用BERT。

```
import torch
from transformers import BertModel, BertTokenizer
# 这里我们调用bert-base模型，同时模型的词典经过小写处理
model_name = 'bert-base-uncased'
# 读取模型对应的tokenizer
tokenizer = BertTokenizer.from_pretrained(model_name)
# 载入模型
model = BertModel.from_pretrained(model_name)
# 输入文本
input_text = "Here is some text to encode"
# 通过tokenizer把文本变成 token_id
input_ids = tokenizer.encode(input_text, add_special_tokens=True)
# input_ids: [101, 2182, 2003, 2070, 3793, 2000, 4372, 16044, 102]
input_ids = torch.tensor([input_ids])
# 获得BERT模型最后一个隐层结果
with torch.no_grad():
    last_hidden_states = model(input_ids)[0]  # Models outputs are now tuples
"""
tensor([[[-0.0549,  0.1053, -0.1065,  ..., -0.3550,  0.0686,  0.6506],
```

```
          [-0.5759, -0.3650, -0.1383,  ..., -0.6782,  0.2092, -0.1639],
          [-0.1641, -0.5597,  0.0150,  ..., -0.1603, -0.1346,  0.6216],
          ...,
          [ 0.2448,  0.1254,  0.1587,  ..., -0.2749, -0.1163,  0.8809],
          [ 0.0481,  0.4950, -0.2827,  ..., -0.6097, -0.1212,  0.2527],
          [ 0.9046,  0.2137, -0.5897,  ...,  0.3040, -0.6172, -0.1950]]])
  shape: (1, 9, 768)
"""
```

可以看到，包括import在内的不到十行代码，我们就实现了读取一个预训练过的BERT模型，来encode我们指定的一个文本，对文本的每一个token生成768维的向量。如果是二分类任务，我们接下来就可以把第一个token也就是[CLS]的768维向量，接一个linear层，预测出分类的logits，或者根据标签进行训练。

如果你想在一些NLP常用数据集上复现BERT的效果，Transformers上也有现成的代码和方法，只要把数据配置好，运行命令即可，而且finetune的任务可以根据你的需要切换，非常方便。

run_glue.py : Fine-tuning on GLUE tasks for sequence classification

The General Language Understanding Evaluation (GLUE) benchmark is a collection of nine sentence- or sentence-pair language understanding tasks for evaluating and analyzing natural language understanding systems.

Before running anyone of these GLUE tasks you should download the GLUE data by running this script and unpack it to some directory `$GLUE_DIR`.

You should also install the additional packages required by the examples:

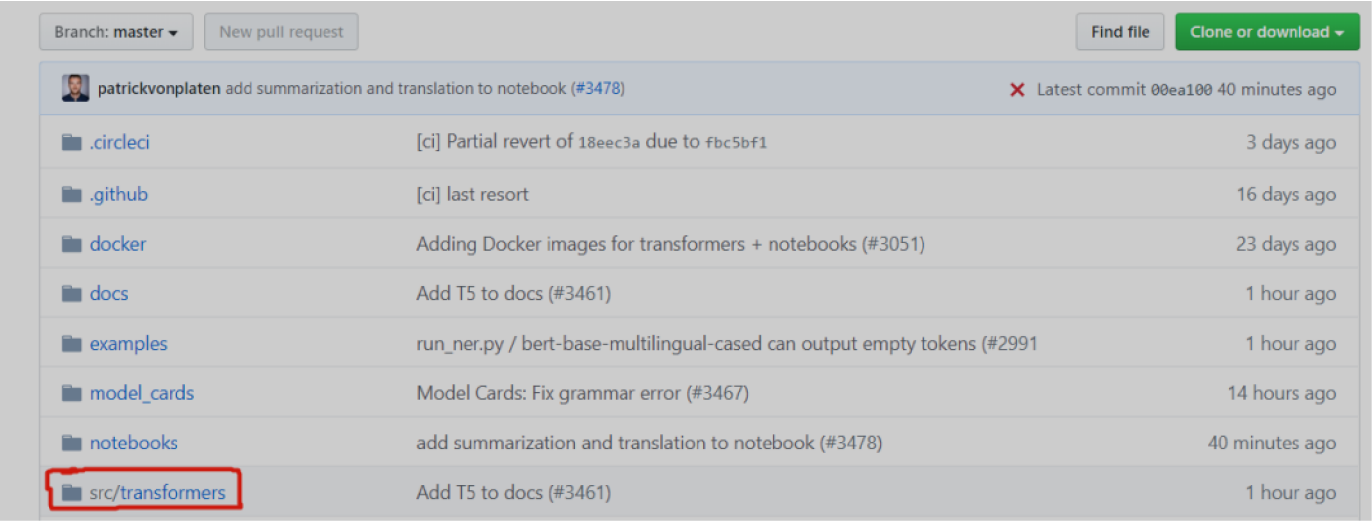
```
pip install -r ./examples/requirements.txt
```

```
export GLUE_DIR=/path/to/glue
export TASK_NAME=MRPC

python ./examples/run_glue.py \
    --model_type bert \
    --model_name_or_path bert-base-uncased \
    --task_name $TASK_NAME \
    --do_train \
    --do_eval \
    --do_lower_case \
    --data_dir $GLUE_DIR/$TASK_NAME \
    --max_seq_length 128 \
    --per_gpu_eval_batch_size=8   \
    --per_gpu_train_batch_size=8   \
    --learning_rate 2e-5 \
    --num_train_epochs 3.0 \
    --output_dir /tmp/$TASK_NAME/
```

BERT configuration

接下来，我们进一步看下Transformers的源码，我们首先进入代码的路径src/transformers下，其中有很多的python代码文件。



以 **configuration** 开头的都是各个模型的配置代码，比如 configuration_bert.py。在这个文件里我们能够看到，主要是一个继承自 PretrainedConfig 的类 BertConfig的定义，以及不同BERT模型的config文件的下载路径，下方显示前三个。

```
BERT_PRETRAINED_CONFIG_ARCHIVE_MAP = {
    "bert-base-uncased": "https://s3.amazonaws.com/models.huggingface.co/bert/b
    "bert-large-uncased": "https://s3.amazonaws.com/models.huggingface.co/bert/
    "bert-base-cased": "https://s3.amazonaws.com/models.huggingface.co/bert/ber
}
```

我们打开第一个的链接，就能下载到bert-base-uncased的模型的配置，其中包括dropout, hidden_size, num_hidden_layers, vocab_size 等等。比如bert-base-uncased的配置它是12层的，词典大小30522等等，甚至可以在config里利用output_hidden_states配置是否输出所有hidden_state。

```
{
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
```

```
  "type_vocab_size": 2,
  "vocab_size": 30522
}
```

BERT tokenization

以**tokenization**开头的都是跟vocab有关的代码，比如在 tokenization_bert.py 中有函数如whitespace_tokenize，还有不同的tokenizer的类。同时也有各个模型对应的vocab.txt。从第一个链接进去就是bert-base-uncased的词典，这里面有30522个词，对应着config里面的vocab_size。

其中，第0个token是[pad]，第101个token是[CLS]，第102个token是[SEP]，所以之前我们encode得到的 [101, 2182, 2003, 2070, 3793, 2000, 4372, 16044, 102] ，其实tokenize后convert前的token就是 ['[CLS]', 'here', 'is', 'some', 'text', 'to', 'en', '##code', '[SEP]']，经过之前BERT论文的介绍，大家应该都比较熟悉了。其中值得一提的是，BERT的vocab预留了不少unused token，如果我们会在文本中使用特殊字符，在vocab中没有，这时候就可以通过替换vacab中的unused token，实现对新的token的embedding进行训练。

```
PRETRAINED_VOCAB_FILES_MAP = {
    "vocab_file": {
        "bert-base-uncased": "https://s3.amazonaws.com/models.huggingface.co/be
        "bert-large-uncased": "https://s3.amazonaws.com/models.huggingface.co/b
        "bert-base-cased": "https://s3.amazonaws.com/models.huggingface.co/bert
    }
}
```

BERT modeling

以modeling开头的就是我们最关心的模型代码，比如 modeling_bert.py。同样的，文件中有许多不同的预训练模型以供下载，我们可以按需获取。

代码中我们可以重点关注BertModel类，它就是BERT模型的基本代码。我们可以看到它的类定义中，由embedding，encoder，pooler组成，forward时顺序经过三个模块，输出output。

```
class BertModel(BertPreTrainedModel):
```

```
    def __init__(self, config):
        super().__init__(config)
        self.config = config

        self.embeddings = BertEmbeddings(config)
        self.encoder = BertEncoder(config)
        self.pooler = BertPooler(config)

        self.init_weights()

def forward(
        self, input_ids=None, attention_mask=None, token_type_ids=None,
        position_ids=None, head_mask=None, inputs_embeds=None,
        encoder_hidden_states=None, encoder_attention_mask=None,
    ):
    """ 省略部分代码 """

        embedding_output = self.embeddings(
            input_ids=input_ids, position_ids=position_ids, token_type_ids=tok
        )
        encoder_outputs = self.encoder(
            embedding_output,
            attention_mask=extended_attention_mask,
            head_mask=head_mask,
            encoder_hidden_states=encoder_hidden_states,
            encoder_attention_mask=encoder_extended_attention_mask,
        )
        sequence_output = encoder_outputs[0]
        pooled_output = self.pooler(sequence_output)

        outputs = (sequence_output, pooled_output,) + encoder_outputs[
            1:
        ]  # add hidden_states and attentions if they are here
        return outputs  # sequence_output, pooled_output, (hidden_states), (at
```

BertEmbeddings这个类中可以清楚的看到，embedding由三种embedding相加得到，经过layernorm 和 dropout后输出。

```
def __init__(self, config):
```

```
        super().__init__()
        self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_s
        self.position_embeddings = nn.Embedding(config.max_position_embeddings
        self.token_type_embeddings = nn.Embedding(config.type_vocab_size, conf
        # self.LayerNorm is not snake-cased to stick with TensorFlow model var
        # any TensorFlow checkpoint file
        self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_no
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

def forward(self, input_ids=None, token_type_ids=None, position_ids=None, inpu
        """ 省略 embedding生成过程 """

        embeddings = inputs_embeds + position_embeddings + token_type_embeddin
        embeddings = self.LayerNorm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

BertEncoder主要将embedding的输出，逐个经过每一层Bertlayer的处理，得到各层hidden_state，再根据config的参数，来决定最后是否所有的hidden_state都要输出，BertLayer的内容展开的话，篇幅过长，读者感兴趣可以自己一探究竟。

```
class BertEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.output_attentions = config.output_attentions
        self.output_hidden_states = config.output_hidden_states
        self.layer = nn.ModuleList([BertLayer(config) for _ in range(config.nu

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
    ):
        all_hidden_states = ()
        all_attentions = ()
        for i, layer_module in enumerate(self.layer):
```

```
            if self.output_hidden_states:
                all_hidden_states = all_hidden_states + (hidden_states,)

            layer_outputs = layer_module(
                hidden_states, attention_mask, head_mask[i], encoder_hidden_st
            )
            hidden_states = layer_outputs[0]

            if self.output_attentions:
                all_attentions = all_attentions + (layer_outputs[1],)

        # Add last layer
        if self.output_hidden_states:
            all_hidden_states = all_hidden_states + (hidden_states,)

        outputs = (hidden_states,)
        if self.output_hidden_states:
            outputs = outputs + (all_hidden_states,)
        if self.output_attentions:
            outputs = outputs + (all_attentions,)
        return outputs  # last-layer hidden state, (all hidden states), (all a
```

Bertpooler 其实就是将BERT的[CLS]的hidden_state 取出，经过一层DNN和Tanh计算后输出。

```
class BertPooler(nn.Module):
    def __init__(self, config):
        super().__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
```

在这个文件中还有上述基础的BertModel的进一步的变化，比如`BertForMaskedLM`，`BertForNextSentencePrediction`这些是Bert加了预训练头的模型，还有`BertForSequenceClassification`，`BertForQuestionAnswering`这些加上了特定任务头的模型。

未完待续

本期的代码浅析就给大家分享到这里，感谢大家的阅读和支持，下期我们会继续给大家带来预训练语言模型相关的论文阅读，敬请大家期待！

欢迎关注朴素人工智能，这里有很多最新最热的论文阅读分享，有问题或建议可以在公众号下留言。

往期推荐

- 性能媲美BERT却只有其1/10参数量？| 近期最火模型ELECTRA解析
- 微软统一预训练语言模型UniLM 2.0解读
- BERT，开启NLP新时代的王者
- 十分钟了解文本分类通用技巧