

# 【Github】BERT-train2deploy: BERT模型从训练到部署

xmxoxo AINLP 2019-06-29

周末推荐Github上一个BERT相关的资源：BERT-train2deploy，BERT模型从训练到部署，作者 xmxoxo，Github链接，[点击原文直达](https://github.com/xmxoxo/BERT-train2deploy)：  
<https://github.com/xmxoxo/BERT-train2deploy>

以下来自该项目描述，感兴趣的同学可以试一下。

## BERT模型从训练到部署全流程

Tag: BERT 训练 部署

### 缘起

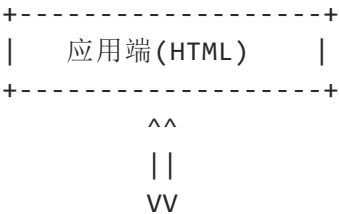
在群里看到许多朋友在使用BERT模型，网上多数文章只提到了模型的训练方法，后面的生产部署及调用并没有说明。这段时间使用BERT模型完成了从数据准备到生产部署的全流程，在这里整理出来，方便大家参考。

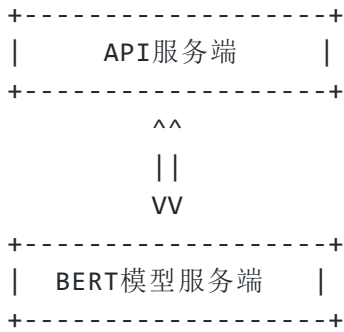
在下面我将以一个“手机评论的情感分类”为例子，简要说明从训练到部署的全部流程。最终完成后可以使用一个网页进行交互，实时地对输入的评论语句进行分类判断。

### 基本架构

基本架构为：

```
graph LR
A(BERT模型服务端) --> B(API服务端)
B --> A
B --> C(应用端)
C --> B
```





架构说明：**BERT模型服务端** 加载模型，进行实时预测的服务；使用的是 BERT-BiLSTM-CRF-NER

**API服务端** 调用实时预测服务，为应用提供API接口的服务，用flask编写；

**应用端** 最终的应用端；我这里使用一个HTML网页来实现；

本项目完整源码地址：BERT从训练到部署git源码 项目博客地址: BERT从训练到部署

附件：本例中训练完成的模型文件.ckpt格式及.pb格式文件，由于比较大，已放到网盘提供下载：

链接：<https://pan.baidu.com/s/1DgVjRK7zicbTlAAkFp7nWw>  
提取码：8iaw

如果你想跳过前面模型的训练过程，可以直接使用训练好的模型，来完成后面的部署。

## 🔗关键节点

主要包括以下关键节点：

- 数据准备
- 模型训练
- 模型格式转化
- 服务端部署与启动
- API服务编写与部署
- 客户端(网页端的编写与部署)

## 🔗数据准备

这里用的数据是手机的评论，数据比较简单,三个分类: -1,0,1 表示负面，中性与正面情感  
数据格式如下：

1 手机很好，漂亮时尚，赠品一般  
 1 手机很好。包装也很完美，赠品也是收到货后马上就发货了  
 1 第一次在第三方买的手机 开始很担心 不过查一下是正品 很满意  
 1 很不错 续航好 系统流畅  
 1 不知道真假，相信店家吧  
 1 快递挺快的，荣耀10手感还是不错的，玩了会王者还不错，就是前后玻璃，  
 1 流很快，手机到手感觉很酷，白色适合女士，很惊艳！常好，运行速度快，流畅！  
 1 用了一天才来评价，都还可以，很满意  
 1 幻影蓝很好看啊，炫彩系列时尚时尚最时尚，速度快，配送运行?做活动优惠买的，开心?  
 1 快递速度快，很赞！软件更新到最新版。安装上软胶保护套拿手上不容易滑落。  
 0 手机出厂贴膜好薄啊，感觉像塑料膜。其他不能发表  
 0 用了一段时间，除了手机续航其它还不错。  
 0 做工一般  
 1 挺好的，赞一个，手机很好，很喜欢  
 0 手机还行，但是手机刚开箱时屏幕和背面有很多指纹痕迹，手机壳跟\*\*在地上磨过似的，好几条  
 -1 手机整体还可以，拍照也很清楚，也很流畅支持华为。给一星是因为有缺陷，送的耳机是坏的！  
 0 前天刚买的，看着还行，指纹解锁反应不错。  
 1 高端大气上档次。  
 -1 各位小主，注意啦，耳机是没有的，需要单独买  
 0 外观不错，感觉很耗电啊，在使用段时间评价  
 1 手机非常好，很好用  
 -1 没有发票，图片与实物不一致  
 1 习惯在京东采购物品，方便快捷，及时开票进行报销，配送员服务也很周到！就是手机收到时没  
 1 高端大气上档次啊！看电影玩游戏估计很爽！屏幕够大！

数据总共8097条，按6:2:2的比例拆分成train.tsv,test.tsv,dev.tsv三个数据文件

## 🔗模型训练

训练模型就直接使用BERT的分类方法，把原来的 run\_classifier.py 复制出来并修改为 run\_mobile.py。关于训练的代码网上很多，就不展开说明了，主要有以下方法：

```
#-----
#手机评论情感分类数据处理 2019/3/12
#labels: -1负面 0中性 1正面
class SetimentProcessor(DataProcessor):
    def get_train_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_tsv(os.path.join(data_dir, "train.tsv")), "train")

    def get_dev_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_tsv(os.path.join(data_dir, "dev.tsv")), "dev")

    def get_test_examples(self, data_dir):
        """See base class."""
```

```

return self._create_examples(
    self._read_tsv(os.path.join(data_dir, "test.tsv")), "test")

def get_labels(self):
    """See base class."""

    """
    if not os.path.exists(os.path.join(FLAGS.output_dir, 'label_list.pkl')):
        with codecs.open(os.path.join(FLAGS.output_dir, 'label_list.pkl'), 'wb') as
            pickle.dump(self.labels, fd)
    """
    return ["-1", "0", "1"]

def _create_examples(self, lines, set_type):
    """Creates examples for the training and dev sets."""
    examples = []
    for (i, line) in enumerate(lines):
        if i == 0:
            continue
        guid = "%s-%s" % (set_type, i)

        #debug (by xmxoxo)
        #print("read line: No.%d" % i)

        text_a = tokenization.convert_to_unicode(line[1])
        if set_type == "test":
            label = "0"
        else:
            label = tokenization.convert_to_unicode(line[0])
        examples.append(
            InputExample(guid=guid, text_a=text_a, label=label))
    return examples
#-----

```

然后添加一个方法：

```

processors = {
    "cola": ColaProcessor,
    "mnli": MnliProcessor,
    "mrpc": MrpcProcessor,
    "xnli": XnliProcessor,
    "setiment": SetimentProcessor, #2019/3/27 add by Echo
}

```

**特别说明**，这里有一点要注意，在后期部署的时候，需要一个label2id的字典，所以要在训练的时候就保存起来，在 `convert_single_example` 这个方法里增加一段：

```

#--- save label2id.pkl ---
#在这里输出label2id.pkl , add by xmxoxo 2019/2/27
output_label2id_file = os.path.join(FLAGS.output_dir, "label2id.pkl")
if not os.path.exists(output_label2id_file):

```

```
with open(output_label2id_file, 'wb') as w:
    pickle.dump(label_map, w)
```

```
#--- Add end ---
```

这样训练后就会生成这个文件了。

使用以下命令训练模型，目录参数请根据各自的情况修改：

```
cd /mnt/sda1/transdat/bert-demo/bert/
export BERT_BASE_DIR=/mnt/sda1/transdat/bert-demo/bert/chinese_L-12_H-768_A-12
export GLUE_DIR=/mnt/sda1/transdat/bert-demo/bert/data
export TRAINED_CLASSIFIER=/mnt/sda1/transdat/bert-demo/bert/output
export EXP_NAME=mobile_0

sudo python run_mobile.py \
  --task_name=setiment \
  --do_train=true \
  --do_eval=true \
  --data_dir=$GLUE_DIR/$EXP_NAME \
  --vocab_file=$BERT_BASE_DIR/vocab.txt \
  --bert_config_file=$BERT_BASE_DIR/bert_config.json \
  --init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
  --max_seq_length=128 \
  --train_batch_size=32 \
  --learning_rate=2e-5 \
  --num_train_epochs=5.0 \
  --output_dir=$TRAINED_CLASSIFIER/$EXP_NAME
```

由于数据比较小，训练是比较快的，训练完成后，可以在输出目录得到模型文件，这里的模型文件格式是.ckpt的。训练结果：

```
eval_accuracy = 0.861643
eval_f1 = 0.9536328
eval_loss = 0.56324786
eval_precision = 0.9491279
eval_recall = 0.9581805
global_step = 759
loss = 0.5615213
```

可以使用以下语句来进行预测：

```
sudo python run_mobile.py \
  --task_name=setiment \
  --do_predict=true \
  --data_dir=$GLUE_DIR/$EXP_NAME \
  --vocab_file=$BERT_BASE_DIR/vocab.txt \
  --bert_config_file=$BERT_BASE_DIR/bert_config.json \
  --init_checkpoint=$TRAINED_CLASSIFIER/$EXP_NAME \
```

```
--max_seq_length=128 \  
--output_dir=$TRAINED_CLASSIFIER/$EXP_NAME
```

## 🔗模型格式转化

到这里我们已经训练得到了模型，但这个模型是.ckpt的文件格式,文件比较大，并且有三个文件：

```
-rw-r--r-- 1 root root 1227239468 Apr 15 17:46 model.ckpt-759.data-00000-of-00001  
-rw-r--r-- 1 root root      22717 Apr 15 17:46 model.ckpt-759.index  
-rw-r--r-- 1 root root      3948381 Apr 15 17:46 model.ckpt-759.meta
```

可以看到，模板文件非常大，大约有1.17G。后面使用的模型服务端，使用的是.pb格式的模型文件，所以要把生成的ckpt格式模型文件转换成.pb格式的模型文件。我这里提供了一个转换工具: freeze\_graph.py，使用如下：

```
usage: freeze_graph.py [-h] -bert_model_dir BERT_MODEL_DIR -model_dir  
MODEL_DIR [-model_pb_dir MODEL_PB_DIR]  
[-max_seq_len MAX_SEQ_LEN] [-num_labels NUM_LABELS]  
[-verbose]
```

这里要注意的参数是：

- model\_dir 就是训练好的.ckpt文件所在的目录
- max\_seq\_len 要与原来一致；
- num\_labels 是分类标签的个数,本例中是3个

```
python freeze_graph.py \  
-bert_model_dir $BERT_BASE_DIR \  
-model_dir $TRAINED_CLASSIFIER/$EXP_NAME \  
-max_seq_len 128 \  
-num_labels 3
```

执行成功后可以看到在 model\_dir 目录会生成一个 classification\_model.pb 文件。转为.pb格式的模型文件，同时也可以缩小模型文件的大小,可以看到转化后的模型文件大约是390M。

```
-rw-rw-r-- 1 hexi hexi 409326375 Apr 15 17:58 classification_model.pb
```

## 🔗服务端部署与启动

现在可以安装服务端了，使用的是 bert-base, 来自于项目 BERT-BiLSTM-CRF-NER，服务端只是该项目中的一个部分。项目地址：<https://github.com/macanv/BERT-BiLSTM-CRF-NER>，感谢Macanv同学提供这么好的项目。

这里要说明一下，我们经常会看到bert-as-service 这个项目的介绍，它只能加载BERT的预训练模型，输出文本向量化的结果。而如果要加载fine-tuning后的模型，就要用到 bert-base 了，详情请见：基于BERT预训练的中文命名实体识别TensorFlow实现

下载代码并安装：

```
pip install bert-base==0.0.7 -i https://pypi.python.org/simple
```

或者

```
git clone https://github.com/macanv/BERT-BiLSTM-CRF-NER
cd BERT-BiLSTM-CRF-NER/
python3 setup.py install
```

使用 bert-base 有三种运行模式，分别支持三种模型，使用参数 -mode 来指定：

- NER 序列标注类型，比如命名实体识别；
- CLASS 分类模型，就是本文中使用的模型
- BERT 这个就是跟bert-as-service 一样的模式了

之所以要分成不同的运行模式，是因为不同模型对输入内容的预处理是不同的，命名实体识别NER是要进行序列标注；而分类模型只要返回label就可以了。

安装完后运行服务，同时指定监听 HTTP 8091端口，并使用GPU 1来跑；

```
cd /mnt/sda1/transdat/bert-demo/bert/bert_svr

export BERT_BASE_DIR=/mnt/sda1/transdat/bert-demo/bert/chinese_L-12_H-768_A-12
export TRAINED_CLASSIFIER=/mnt/sda1/transdat/bert-demo/bert/output
export EXP_NAME=mobile_0

bert-base-serving-start \
  -model_dir $TRAINED_CLASSIFIER/$EXP_NAME \
  -bert_model_dir $BERT_BASE_DIR \
  -model_pb_dir $TRAINED_CLASSIFIER/$EXP_NAME \
  -mode CLASS \
  -max_seq_len 128 \
  -http_port 8091 \
  -port 5575 \
  -port_out 5576 \
  -device_map 1
```

**注意：** port 和 port\_out 这两个参数是API调用的端口号，默认是5555和5556,如果你准备部署多个模型服务实例，那一定要指定自己的端口号，避免冲突。我这里是改为：5575 和 5576

如果报错没运行起来，可能是有些模块没装上,都是 bert\_base/server/http.py里引用的，装上就好了：

```
sudo pip install flask
sudo pip install flask_compress
sudo pip install flask_cors
sudo pip install flask_json
```

我这里的配置是2个GTX 1080 Ti，这个时候双卡的优势终于发挥出来了，GPU 1用于预测，GPU 0还可以继续训练模型。

运行服务后会自动生成很多临时的目录和文件，为了方便管理与启动，可建立一个工作目录，并把启动命令写成一个shell脚本。这里创建的是 mobile\_svr\bertsvr.sh，这样可以比较方便地设置服务器启动时自动启动服务，另外增加了每次启动时自动清除临时文件

代码如下：

```
#!/bin/bash
#chkconfig: 2345 80 90
#description: 启动BERT分类模型

echo '正在启动 BERT mobile svr...'
cd /mnt/sda1/transdat/bert-demo/bert/mobile_svr
sudo rm -rf tmp*

export BERT_BASE_DIR=/mnt/sda1/transdat/bert-demo/bert/chinese_L-12_H-768_A-12
export TRAINED_CLASSIFIER=/mnt/sda1/transdat/bert-demo/bert/output
export EXP_NAME=mobile_0

bert-base-serving-start \
    -model_dir $TRAINED_CLASSIFIER/$EXP_NAME \
    -bert_model_dir $BERT_BASE_DIR \
    -model_pb_dir $TRAINED_CLASSIFIER/$EXP_NAME \
    -mode CLASS \
    -max_seq_len 128 \
    -http_port 8091 \
    -port 5575 \
    -port_out 5576 \
    -device_map 1
```

补充说明一下内存的使用情况：BERT在训练时需要加载完整的模型数据，要用的内存是比较多的，差不多要10G，我这里用的是GTX 1080 Ti 11G。但在训练完后，按上面的方式部署加载pb模型文件时，就不需要那么大了，上面也可以看到pb模型文件就是



390M。其实只要你使用的是BERT base 预训练模型，最终的得到的pb文件大小都是差不多的。

还有同学问到能不能用CPU来部署，我这里没尝试过，但我想肯定是可以的，只是在计算速度上跟GPU会有差别。

我这里使用GPU 1来实时预测，同时加载了2个BERT模型,截图如下：

```

192.168.15.111 - PuTTY
+-----+
| NVIDIA-SMI 384.81                               | Driver Version: 384.81 |
+-----+-----+
| GPU  Name      Persistence-M | Bus-Id      Disp.A    | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage      | GPU-Util  Compute M. |
+-----+-----+-----+
|  0  GeForce GTX 108...  Off   | 00000000:17:00.0 Off   |          0%      N/A |
|  2%   45C    P8      20W / 250W| 181MiB / 11172MiB      |             Default  |
+-----+-----+-----+
|  1  GeForce GTX 108...  Off   | 00000000:65:00.0 Off   |          0%      N/A |
|  2%   46C    P8      19W / 250W| 4608MiB / 11170MiB     |             Default  |
+-----+-----+-----+
+-----+
| Processes:                                         | GPU Memory |
| GPU      PID    Type    Process name      |      Usage  |
+-----+-----+-----+
|  0      92189    C      /usr/bin/python    |      171MiB |
|  1      20077    C      /usr/bin/python3   |     1275MiB |
|  1      20381    C      /usr/bin/python3   |     3323MiB |
+-----+

```

## 端口测试

模型服务端部署完成了，可以使用curl命令来测试一下它的运行情况。

```
curl -X POST http://192.168.15.111:8091/encode \
-H 'content-type: application/json' \
-d '{"id": 111,"texts": ["总的来说，这款手机性价比是特别高的。","糟糕的售后服务!!! 店
```

执行结果：

```
> -H 'content-type: application/json' \
> -d '{"id": 111,"texts": ["总的来说，这款手机性价比是特别高的。","糟糕的售后服务!!!
{"id":111,"result":[{"pred_label":["1","-1"],"score":[0.9974544644355774,0.996142206
```

可以看到对应的两个评论，预测结果一个是1，另一个是-1，计算的速度还是非常很快的。通过这种方式来调用还是不太方便，知道了这个通讯方式，我们可以用flask编写一个API服务，为所有的应用统一提供服务。

## 🔗API服务编写与部署

为了方便客户端的调用，同时也为了可以对多个语句进行预测，我们用flask编写一个API服务端，使用更简洁的方式来与客户端（应用）来通讯。整个API服务端放在独立目录 /mobile\_apisvr/ 目录下。

用flask创建服务端并调用主方法,命令行参数如下：

```
def main_cli ():
    pass
    parser = argparse.ArgumentParser(description='API demo server')
    parser.add_argument('-ip', type=str, default="0.0.0.0",
                        help='chinese google bert model serving')
    parser.add_argument('-port', type=int, default=8910,
                        help='listen port,default:8910')

    args = parser.parse_args()

    flask_server(args)
```

主方法里创建APP对象：

```
app.run(
    host = args.ip,      #'0.0.0.0',
    port = args.port,    #8910,
    debug = True
)
```

这里的接口简单规划为 /api/v0.1/query，使用POST方法，参数名为'text'，使用JSON返回结果；路由配置：

```
@app.route('/api/v0.1/query', methods=['POST'])
```

API服务端的核心方法，是与BERT-Serving进行通讯，需要创建一个客户端 BertClient：

```
#对句子进行预测识别
def class_pred(list_text):
    #文本拆分成句子
    #list_text = cut_sent(text)
    print("total setance: %d" % (len(list_text)))
    with BertClient(ip='192.168.15.111', port=5575, port_out=5576, show_server_conf=True):
        start_t = time.perf_counter()
        rst = bc.encode(list_text)
        print('result:', rst)
        print('time used:{}'.format(time.perf_counter() - start_t))
    #返回结构为:
    # rst: [{'pred_label': ['0', '1', '0'], 'score': [0.9983683228492737, 0.99889934
```

#抽取出标注结果

```
pred_label = rst[0]["pred_label"]
```

```
result_txt = [ [pred_label[i],list_text[i] ] for i in range(len(pred_label))]
```

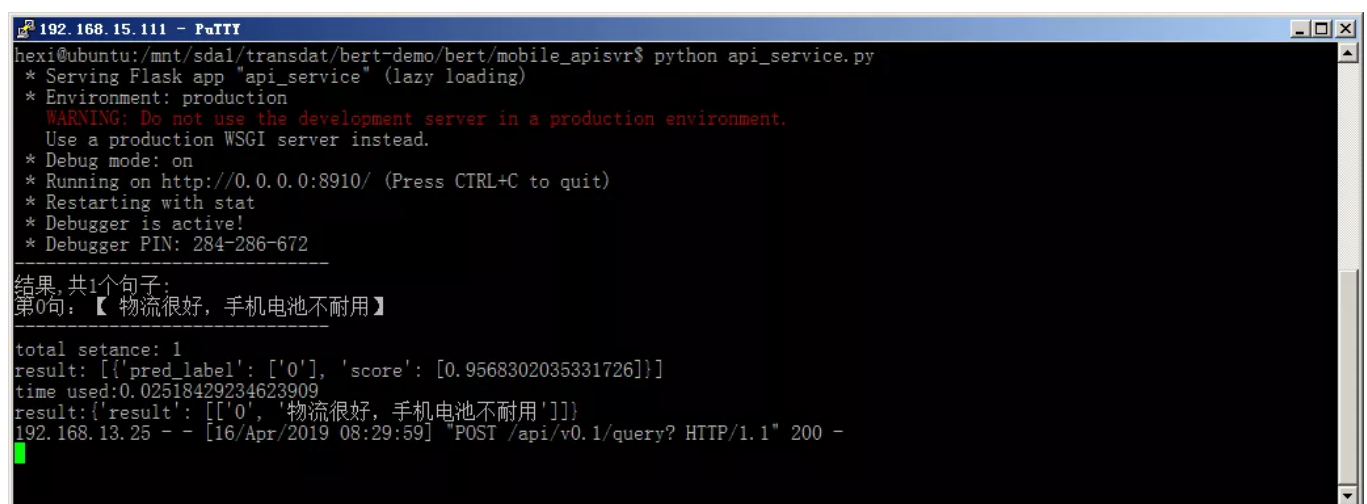
```
return result_txt
```

注意：这里的IP，端口要与服务端的对应。

运行API 服务端：

```
python api_service.py
```

在代码中的debug设置为True，这样只要更新文件，服务就会自动重新启动，比较方便调试。运行截图如下：

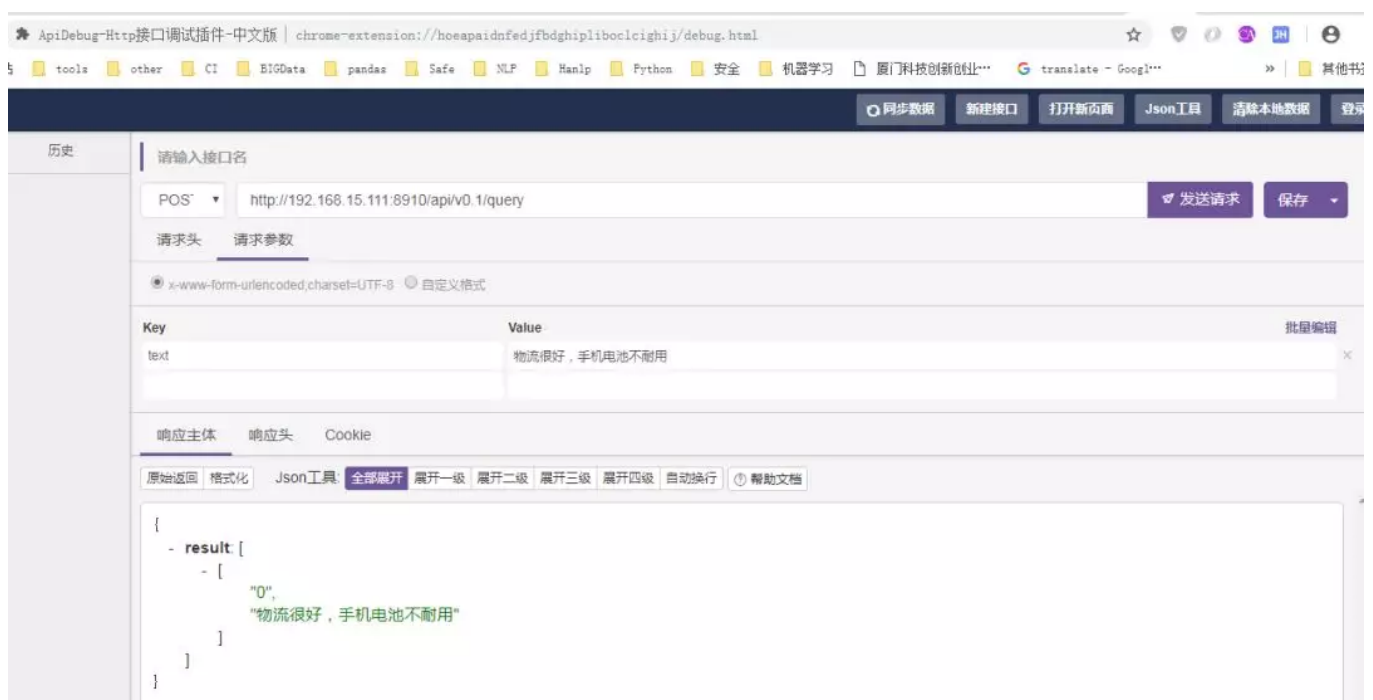


```
192.168.15.111 - PuTTY
hexi@ubuntu:/mnt/sdal/transdat/bert-demo/bert/mobile_apisvr$ python api_service.py
* Serving Flask app "api_service" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:8910/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 284-286-672

结果,共1个句子:
第0句: 【 物流很好,手机电池不耐用】

total setence: 1
result: [{'pred_label': ['0'], 'score': [0.9568302035331726]}]
time used:0.02518429234623909
result: {'result': [['0', '物流很好,手机电池不耐用']]}
192.168.13.25 -- [16/Apr/2019 08:29:59] "POST /api/v0.1/query? HTTP/1.1" 200 -
```

到这一步也可以使用curl或者其它工具进行测试，也可以等完成网页客户端后一并调试。我这里使用chrome插件 API-debug来进行测试，如下图：



## 🔗客户端(网页端)

这里使用一个HTML页面来模拟客户端，在实际项目中可能是具体的应用。为了方便演示就把网页模板与API服务端合并在一起了，在网页端使用AJAX来与API服务端通讯。

创建模板目录 `templates`，使用模板来加载一个HTML,模板文件名为 `index.html`。在HTML页面里使用AJAX来调用接口,由于是在同一个服务器，同一个端口，地址直接用 `/api/v0.1/query` 就可以了，在实际项目中，客户应用端与API是分开的，则需要指定接口URL地址，同时还要注意数据安全性。代码如下：

```
function UrlPOST(txt,myfun){
    if (txt=="")
    {
        return "error parm";
    }
    var httpurl = "/api/v0.1/query";
    $.ajax({
        type: "POST",
        data: "text="+txt,
        url: httpurl,
        //async:false,
        success: function(data)
        {
            myfun(data);
        }
    });
}
```

启动API服务端后，可以使用 IP+端口 来访问了，这里的地址是

`http://192.168.15.111:8910/`

运行界面截图如下：



可以看到请求的用时时间为37ms，速度还是很快的，当然这个速度跟硬件配置有关。

🔗参考资料:

- <https://github.com/google-research/bert>
- <https://github.com/hanxiao/bert-as-service>
- <https://github.com/macanv/BERT-BiLSTM-CRF-NER>

欢迎批评指正，联系邮箱(xmxoxo@qq.com)

阅读原文

喜欢此内容的人还喜欢

如何通俗易懂地让女朋友明白什么是语言模型？

AINLP

“出人命，他把人捅了！”悉尼Chatswood深夜出现流血事件！华人送餐员挥刀刺伤同胞工友！事故原因曝光.....

看看土澳

央视记者调查脱贫县摘帽掺假被抢手机，当地干部反问：“你是谁家的亲戚？”

新华每日电讯