当代人工智能-实验四

摘要

近些年，包括卷积神经网络、循环神经网络和 Transformer 等模型在内的深度神经网络已经广泛地应用于各类应用中。与依赖人工特征的传统机器学习方法相比，深度神经网络能够通过网络层自动学习数据特征，从而摆脱了人工设计特征的局限性，极大地提升了模型的性能。尽管深度神经网络促进各类任务取得了极大的突破，但深度学习模型对数据较高的依赖性也带来了极大的挑战。因为深度学习模型需要学习大量的参数，数据量少必定会造成模型过拟合和泛化能力差。而随着大数据和高性能硬件的发展，自回归语言模型（GPT）和自编码语言模型（BERT）已经取得了巨大的成功，促进了自然语言处理任务性能的大量提升。本文将借助 facebook 预训练的 BART 模型，对于数据进行预训练之后，再使用学习到的模型权重对数据进行预测以及使用 bleu_score 进行计算并且给出模型准确度的评价。

数据预处理

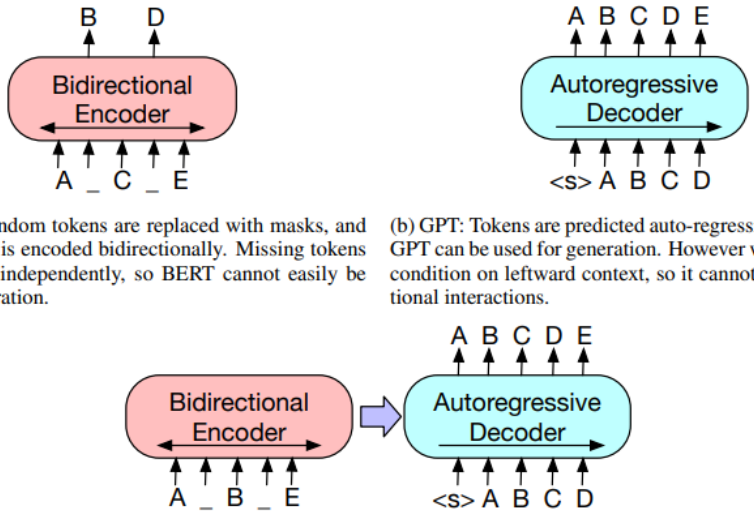
数据集已脱敏，这意味着数据集中的词都被转化为了数字并且无法再重新转化回单词进行语义的学习。数据的训练集有 18000 条，有两个字段，description 和 diagnosis 分别代表输入和输出。测试集有 2000 条数据，description 字段代表输入，我们需要训练模型来预测 diagnosis 输出。

我们先通过 process_data 将 csv 文件读取到列表中，再将其以 80：20 的比例分割成训练集与验证集，然后对数据进行 padding，最后将数据通过 DataLoader 加载成可供预训练的数据。

```
    # Data processing
    data = process_data('./', 'data/train.csv', mode='train')
    test_data = process_data('./', 'data/test.csv', mode='test')

    # split train.csv to train_data and valid_data
    n_split = int(len(data) * 0.8)
    train_data = data[:n_split]
    valid_data = data[n_split:]

    # Load data
    train_loader = build_pre_bart_dataloader(args, train_data, shuffle=True)
    valid_loader = build_pre_bart_dataloader(args, valid_data, shuffle=True)
```

构建 Seq2Seq 模型

我们选择 Bart 作为本次实验的预训练模型。Bart 吸收了 Bert 的 bidirectional encoder 和 GPT 的 left-to right decoder 各自的特点，建立在标准的 Seq2Seq Transformer model 的基础之上，这使得它比 Bert 更适合文本生成的场景；相比 GPT，也多了双向上下文语境信息。在生成任务上获得进步的同时，它也可以在一些文本理解任务上取得很好的成绩。



(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.

(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.

(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.

Figure 1: A schematic comparison of BART with BERT (Devlin et al., 2019) and GPT (Radford et al., 2018).

BART 虽然可以理解成是一个 BERT +GPT 的结构，但是相比于 BERT 单一的 noise 类型 (只有简单地用 [MASK] token 进行替换这一种 noise)，BART 在 encoder 端尝试了多种 noise。因为 BERT 的这种简单替换导致的是 encoder 端的输入携带了有关序列结构的一些信息，比如序列长度，而这些信息在文本生成任务中一般是不会提供给模型的。BART 采用多种多样的 noise，意图是破坏掉这些有关序列结构的信息，防止模型去依赖这样的信息。

具体采用了以下几种 noise：

- **Token Masking**: 就是 BERT 的方法 ---- 随机将 token 替换成 [MASK]
- **Token Deletion**: 随机删去 token
- **Text Infilling**: 随机将一段连续的 token（称作 span）替换成一个 [MASK]，span 的长度服从 $\lambda = 3$ 的泊松分布。注意 span 长度为 0 就相当于插入一个 [MASK]。
- **Sentence Permutation**: 将一个 document 的句子打乱
- **Document Rotation**: 从 document 序列中随机选择一个 token，然后使得该 token 作为 document 的开头

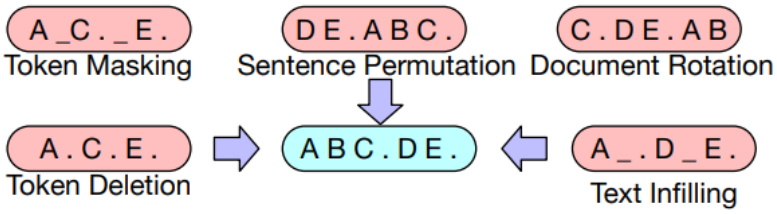


Figure 2: Transformations for noising the input that we experiment with. These transformations can be composed.

由于 BART 吸收了 BERT 双向自编码器和 GPT 自回归解码器的功能，所以它可以应用在多个下游任务上，如 Sequence Classification Task, Token Classification Task, Sequence Generation Task, Machine Translation 等等。本次实验的主题是文本摘要，所以相当于是一个 Sequence Generation Task.

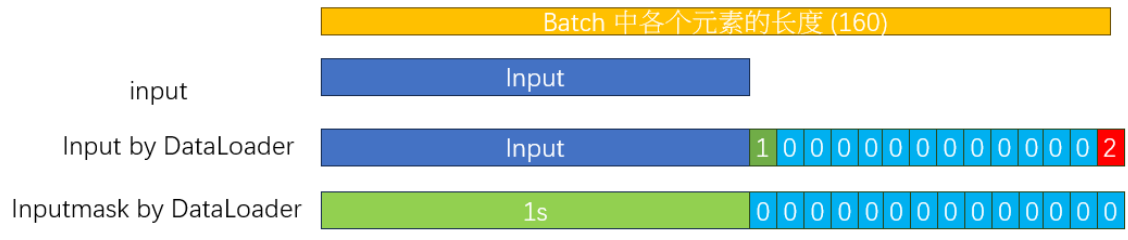
在我们基于 BART 构建的 Seq2Seq 模型中，我们根据 Huggingface 提供的文档 BART (huggingface.co) 使用 BartConfig 设定模型参数，并且使用 BartForConditionalGeneration 指定选取的预训练模型，再将训练数据放在预训练模型中训练参数。

```
class Seq2SeqModel(nn.Module):
```

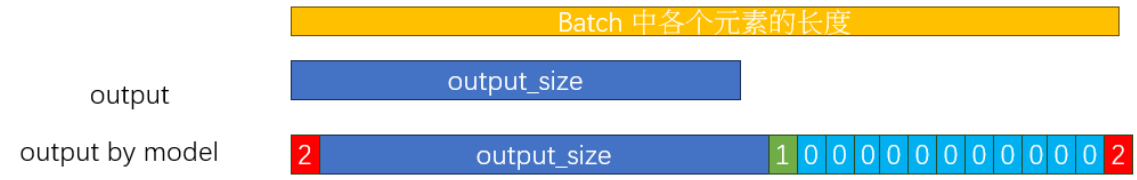
```
    def __init__(self, model_name, vocab_size, label_smoothing=0.01):
        super(Seq2SeqModel, self).__init__()
        self.new_config = BartConfig(
            vocab_size=vocab_size,
            max_position_embeddings=1024,
            encoder_layers=6,
            encoder_ffn_dim=1024,
            encoder_attention_heads=8,
            decoder_layers=6,
            decoder_ffn_dim=1024,
            decoder_attention_heads=8,
            activation_function="gelu",
            d_model=256,
            dropout=0.5,
            attention_dropout=0.0,
            scale_embedding=False,
            use_cache=True,
            pad_token_id=0,
            bos_token_id=1,
            eos_token_id=2,
            mask_token_id=4,
            decoder_start_token_id=2,
            forced_eos_token_id=2,
        )
        # 从配置项中拉取预训练的 模型
        self.bart_model =
BartForConditionalGeneration.from_pretrained(model_name, config=self.new_config,

ignore_mismatched_sizes=True)
```

其中，对于几个关键的参数进行解释：

encoder_layers 以及 decoder_layers 定义为 6 是为了符合 bart-base 的 encoder 以及 decoder 层数。pad_token_id 是给每个句子加上 padding 让输入进的 batch 内的每一个句子形状都相同，但是对于 padding 部分不予以 attention 机制。bos_token_id 代表句子有效长度的结束，标记为 1. eos_token_id 代表整个句子，包括 padding 部分结束，标记为 2. 这些在文件 my_dataset 中都已经实现，把这些参数让 bart 预训练模型知道，让模型"读懂"数据并且给出更好的 attention。



decoder_start_token_id 代表生成的每一个句子开头都是 2，forced_eos_token_id 代表生成的每一个句子结尾都是 2，这些是对于模型生成的语句，加上的一些字符。对于生成的句子，有效部分末尾都会有一个 bos_token_id, 即是代表句子内容的结束。



我们定义的 model_name 和 vocab_size 分别为 `facebook/bart-base` 以及 1300

```
parser.add_argument('--model_name', default='facebook/bart-base', type=str)
parser.add_argument('--vocab_size', default=1300, help='')
```

选择 `bart-base` 是因为认为选择更大的模型会导致模型过拟合，让预测准确率下降。同时加载预训练之后的模型权重会变得更慢，在后续训练与预测的时候也会更花时间与计算资源。对于一个训练+验证数据总共 18000 条的数据集，并且其中只包含 1300 个词，不需要使用太大的预训练模型。

对于为何选择 vocab_size 值为 1300，因为通过将所有的训练数据放到一个 set 中，发现其取值是 9-1300 之间的整数，那么就干脆让模型生成 1300 以内的整数好了。

训练和验证-观察 loss 值和 bleu score

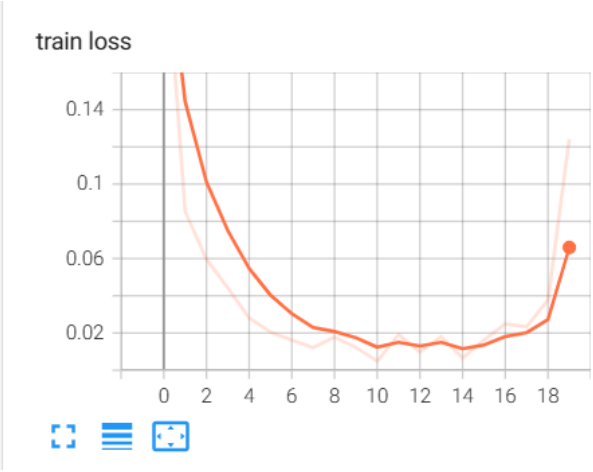
设定 num_epochs 为 20，定义 scheduler 以及 optimizer 进行混合精度训练，提升训练性能。每 5 个 epoch 保存一下模型参数，以便后面的验证和预测。我们通过 torch.utils.tensorboard 中的 SummaryWriter 工具将每次运行之后的结果保存在 runs/ 目录下，terminal run `tensorboard --logdir='path/to/dir'` 就可以查看对应训练中每个轮次的 loss 变化。

```
from torch.utils.tensorboard import SummaryWriter

tb_writer = SummaryWriter()

# Define optimizer and scheduler
optimizer = build_optimizer(args.optimizer_name, pg, args.lr, args.weight_decay)
print('Optimizer configured successfully.')
scheduler = build_scheduler(args.lr_decay_func, optimizer,
args.num_warmup_steps, args.epochs, args.lrf)

scaler = torch.cuda.amp.GradScaler()
# 混合精度训练
for epoch in range(args.epochs):
    cur_lr = optimizer.state_dict()['param_groups'][0]['lr']
    print('lr:', cur_lr)
    print('weight decay:', optimizer.state_dict()['param_groups'][0]
['weight_decay'])
    train_loss = train_one_epoch(train_loader, model, optimizer, epoch,
device=device, is_adversial=False, scaler=scaler)

    tb_writer.add_scalar('train loss', train_loss, epoch)
    tb_writer.add_scalar('lr', cur_lr, epoch)
    if (epoch + 1) % 5 == 0:
        save_model(model, model_dir, args.model_name, epoch)
    scheduler.step()
```

从终端查看训练过程中 loss 的变化，发现在第六个 epoch 以后 loss 值就基本不太变化，降到一个比较低的数值。

```
[train epoch 3] loss: 0.044 lm loss: 0.008 mlm loss: 0.362: 100%|
lr: 0.00025
weight decay: 0.0
[train epoch 4] loss: 0.028 lm loss: 0.008 mlm loss: 0.202: 100%|
lr: 0.0003
weight decay: 0.0
[train epoch 5] loss: 0.021 lm loss: 0.009 mlm loss: 0.114: 100%|
lr: 0.00035
weight decay: 0.0
[train epoch 6] loss: 0.016 lm loss: 0.009 mlm loss: 0.073: 100%|
lr: 0.0004
weight decay: 0.0
[train epoch 7] loss: 0.012 lm loss: 0.007 mlm loss: 0.049: 100%|
lr: 0.00045000000000000004
weight decay: 0.0
[train epoch 8] loss: 0.018 lm loss: 0.011 mlm loss: 0.067: 100%|
lr: 0.0005
weight decay: 0.0
[train epoch 9] loss: 0.012 lm loss: 0.007 mlm loss: 0.048: 100%|
lr: 0.00055
weight decay: 0.0
[train epoch 10] loss: 0.005 lm loss: 0.003 mlm loss: 0.021: 100%|
```

在本地查看 tensorboard 上面 summarywriter 给出的信息，在 localhost 的 6006 端口：



在这里我将 lr 每过一个 epoch 就加上 5e-5 是为了防止训练 loss 一直下降，因为如果 lr 一直保持在一个值就会让训练 loss 一直下降，我们不知道什么时候是过拟合。可以看到，9-14 个 epoch 附近训练的模型是比较合适的。

使用 `load_state_dict` 方法加载预训练后的模型权重以后，在验证集上进行 bleu score 的计算：

```
[valid epoch 0] loss: 0.038 bleu_score: 0.0000:   4%|
bleu score is: 7.128302885413644e-232
[valid epoch 0] loss: 0.038 bleu_score: 0.0000:   4%|
bleu score is: 7.7131359025025e-232
[valid epoch 0] loss: 0.038 bleu_score: 0.0000:   4%|
bleu score is: 7.403557708825715e-232
[valid epoch 0] loss: 0.038 bleu_score: 0.0000:   4%|
bleu score is: 7.128302885413644e-232
[valid epoch 0] loss: 0.037 bleu_score: 0.0000:   4%|
bleu score is: 7.755884093473518e-232
```

可以发现 bleu score 出奇的低。一开始我还以为是在 batch 内部使用 corpus bleu score 的问题，不过后来查阅资料发现，在存在多个参考答案和多个候选答案的时候，计算机制使用双重循环对每个候选答案和参考答案进行配对，并计算每个配对的 BLEU 分数。然后，将每个候选答案的 BLEU 分数取平均，得到最终的平均 BLEU 分数。

后来想到，预训练模型的选取不是瞎选的，主要还是需要 tokenizer 匹配。因为中文的数据集不能用英文的预训练模型，反之亦然。因为它们的 tokenizer 要么是无法对应到相应的 token，要么是有对应的 token 但是没有语义可以学习，学习效果很差。而输入数字和输出数字也是这样，首先我不能按照官方文档所给的那样，使用 Bart 的 tokenizer，因为根本没办法进行 tokenizer，1300 个数字能够都好好地进行 tokenize 就基本不可能。只能通过数字来进行 seq2seq。这样就完全学不到语义，甚至是乱学。

那么为什么 loss 值不高，0.03 的水平，bleu score 还那么低呢？因为 loss 值是关于 ground truth 的，输出的数据可能是一条一条像模像样的，在那 1300 个词中产生，长度也差不多对应，总体符合分布的情况。但是对于 bleu score 这个计算指标，reference 和 generate 的句子可能有几个 1-gram 的对应，但是更高 gram 的对应基本没有的情况下，bleu score 的值就会是很低的。就相当于在一个 1300 个数里面盲猜。看到几乎所有的句子 bleu score 都是7.几e-232 ，可以知道这个数就是0. 所以盲猜得到的结果 bleu score 就是 0 也不奇怪了。

所以得出结论：在没有任何上下文语义的数据，或者用错了 tokenizer 的数据，在预训练模型上面训练，即使优秀如 BART，也依然很难学习到什么。我认为我的结论还是比较合理的。既然如此，也没有什么 fine-tuning 的必要了，因为我将 dropout 值分别设置为 0.1, 0.2, 0.5, 训了几个 epoch 之后，bleuscore 的表现依旧相当地稳定，我也就不想说什么了。通过简单概率论的论证，可以知道，本次数据不适合在预训练语言模型上进行学习和后续的微调。

遇到的问题

1. torchtext 的安装

直接输入 `pip install torchtext` 会使自己的 pytorch 自动更换版本，成为 cpu 版本，因为 torchtext 的版本严格对应 torch 的版本，只能卸载重装

```
In[3]: print(torchtext.__version__)
0.15.2
In[4]: import torch
In[5]: print(torch.__version__)
2.0.1+cpu
```

解决方法：对于 torch 版本，输入 `pip install torchtext==0.xx.y` 安装，在我的 torch 版本 1.13.0 中，xx = 13 + 1, y = 0. 所以输入命令 `pip install torchtext==0.14.0` 安装成功，没有报错。

输入 print(torch.__version) 与 print(torchtext.__version) 之后，都显示正常了。

```
In[3]: print(torch.__version__)
1.13.0+cu116
In[4]: import torchtext
In[5]: print(torchtext.__version__)
0.14.0
```

可以接下来开展进一步的实验。

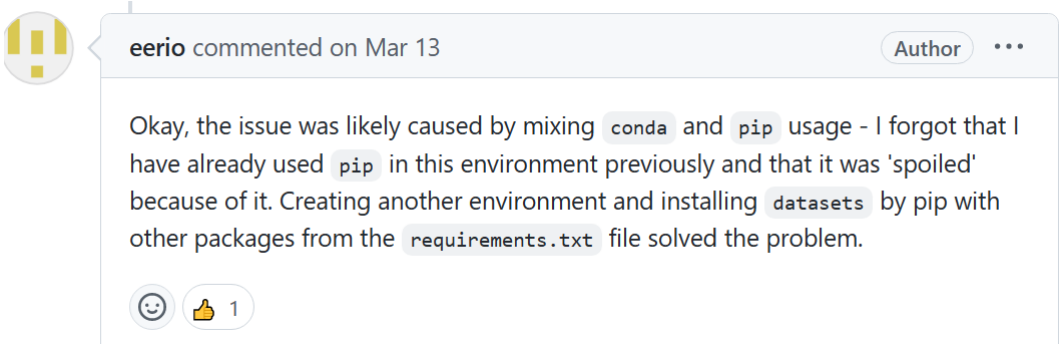
reference：N1、Pytorch 文本分类入门 - CASTWJ - 博客园 (cnblogs.com)

2. 使用 huggingface 模块，安装遇到的问题

我在 pycharm 的 UI 上面直接点了 install package datasets, 随后在 console 中输入 `from datasets import load_dataset` 之后，出现报错 ImportError & ModuleNotFoundError. 由于自己的报错截图已经无法找到，所以从 huggingface 官方项目的 github 仓库中找到相关 issue 的报错信息，大致如下：

```
$ python3
Python 3.8.15 (default, Nov 24 2022, 15:19:38)
[GCC 11.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import datasets
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/datasets/
    from .arrow_dataset import Dataset, concatenate_datasets
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/datasets/
    from .arrow_reader import ArrowReader
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/datasets/
    import pyarrow.parquet as pq
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/pyarrow/p
    from .core import *
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/pyarrow/p
    from pyarrow._parquet import (ParquetReader, Statistics,  # noqa
ImportError: cannot import name 'FileEncryptionProperties' from 'pyarrow._par
```

在此 issue 的时间线中，提供的方法：



在这之后，我通过 `conda remove datasets` 卸载之后，重新使用 pip 安装，即输入命令 `pip install datasets`. 不过在 console 中输入 `from datasets import load_dataset` 之后，出现报错 DLL Not Found, 而对于这一类问题，所给出的方法是卸载重装。于是我又卸载了之后重装，结果还是不行。。

后来受到文章 详解 Python 虚拟环境的原理及使用 - 腾讯云开发者社区 - 腾讯云 (tencent.com) 影响，运行 `python -m venv venv/` 创建了一个虚拟环境，然后 `venv/Scripts/activate` 以后，进入 `/venv/Scripts` 目录，输入 python 命令进入交互页面。在此处输入 `import datasets` `from datasets import load_dataset` 命令。没有报错。所以这一类错误需要在项目目录下自行新建一个 venv 虚拟环境。

```
WARNING: You are using pip version 22.0.4; however, version 23.1.2 is available.
You should consider upgrading via the 'E:\2023 Spring\ContemporaryAI\labs\Lab4\venv\Scripts
(venv) PS E:\2023 Spring\ContemporaryAI\labs\Lab4> cd venv/Scripts
(venv) PS E:\2023 Spring\ContemporaryAI\labs\Lab4\venv\Scripts> python
Python 3.9.16 (main, Jan 11 2023, 16:16:36) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import datasets
>>> from datasets import load_dataset
>>>
```

3. 使用 bleu_score 时候遇到的问题

我将 model.generate 的数字直接与 diagnosis 的数字作对比，发现了如下报错：

```
  File "/usr/local/miniconda3/lib/python3.8/site-packages/nltk/translate/bleu_score.py", line 210,
    p_i = modified_precision(references, hypothesis, i)
  File "/usr/local/miniconda3/lib/python3.8/site-packages/nltk/translate/bleu_score.py", line 353,
    Counter(ngrams(reference, n)) if len(reference) >= n else Counter()
TypeError: object of type 'int' has no len()
```

报错的意思一开始没有读懂，后来发现我的输入和输出都是int类型的数字。于是把标准输出与候选输出的内容全部由整数转换为字符串。可以计算 bleu_score.

References

[1] Xu H, Zhengyan Z, Ning D, et al. Pre-Trained Models: Past, Present and Future[J]. arXiv preprint arXiv:2106.07139, 2021.

[2] Mike Lewis, Yinhan Liu, Naman Goyal, et al. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension arXiv preprint arXiv:1910.13461, 2019.

预训练模型的过去、现在和将来之一-预训练模型和训练模型 (51cto.com)

Pipenv vs Virtualenv vs Conda environment - 知乎 (zhihu.com)

详解 Python 虚拟环境的原理及使用 - 腾讯云开发者社区 - 腾讯云 (tencent.com)

浅谈 BLEU 评分 - Jiayue Cai's Blog (coladrill.github.io)

BART (huggingface.co)