

华南理工大学

《神经网络与深度学习》课程实验报告

实验题目：_____ 基于 Encoder-Decoder 架构的神经网络公式识别 _____

姓名：_____ 王嘉泽、黄子洋、杨浩鸿、方子华、王天寅 _____ 学号：_____

_____ 201930343308、 201930341229、 201930344251 、 201930140334、

_____ 201930142499 _____

班级：_____ 19 计科 1 班 _____ 组别：_____ 17 组 _____

合作者：_____ 无 _____

指导教师：_____ 马千里 _____

实验概述

【实验目的及要求】

实验目的：

实现基于 Encoder-Decoder 架构的神经网络公式识别

实验要求：

实现项目书里的要求

【实验环境】

操作系统：Windows XP

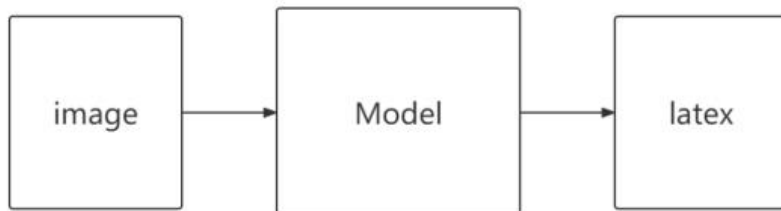
实验内容

一、项目背景

Image to latex 本质上 Image 转化为 seq 问题，类似看图说话。输入 image 后通过 encoder 进行编码获取语境向量，再通过 decoder 进行解码生成语句序列，最后以序列通过词汇表转化为 latex。

Image to Latex Converter

Upload an image of latex math equation

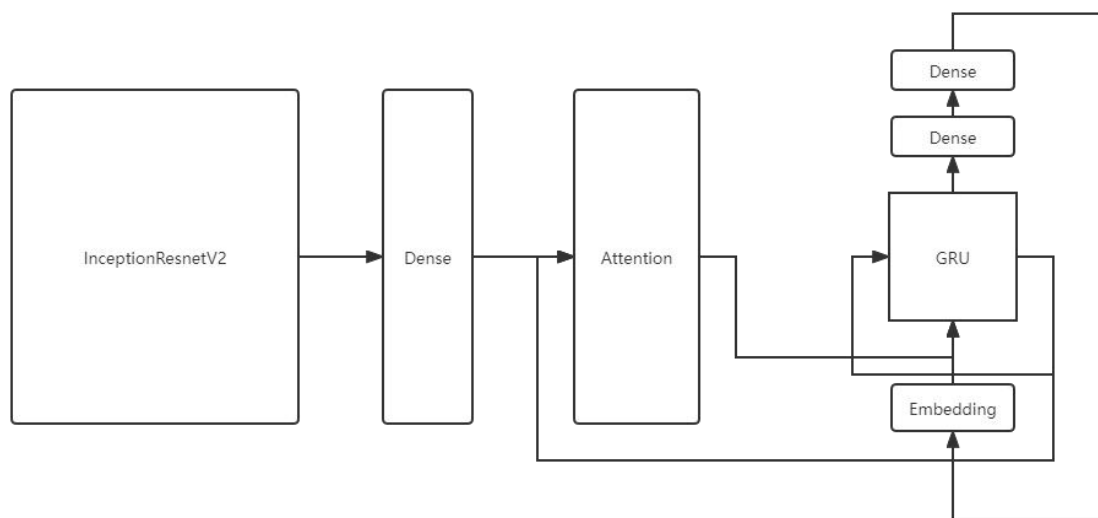


二、小组分工

小组分工为 CNN-RNN 模型两人：黄子洋、方子华；Resnet-Transformer 三人：王嘉泽、杨浩鸿、王天寅。

三、CNN-RNN

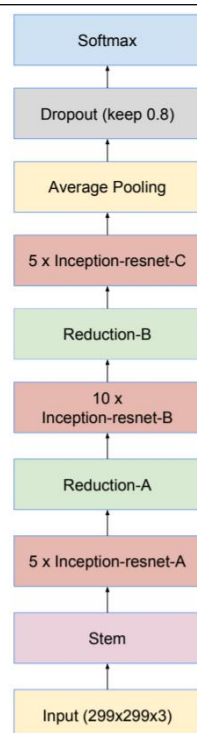
模型大体架构：



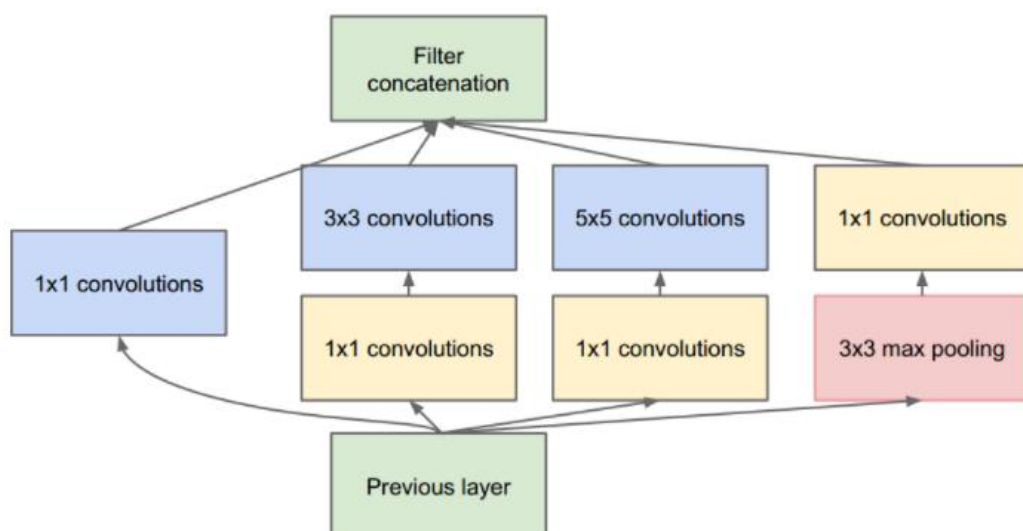
- 模型遵循了 Encoder-Decoder 架构，总体设计参照了实验三模型，但与实验 3 不同的地方有以下几点：
 - 1) 高层 CNN 接口（特征提取器）使用了 InceptionResnetV2 模型，代替了原来的 Resnet 模型
 - 2) 为了提升模型效果，在 CNN 的 dense 层后增加了 attention 层，attention 层接受了 GRU 的隐藏状态信息以及图片的特征信息
 - 3) RNNCell 使用了更直观的 GRU（只有一个门控）
- 接下来我将详细介绍模型架构以及训练中遇到的问题，报告将分为以下几个部分：
 - 1) CNN 相关的问题
 - 2) Attention 的实现方式
 - 3) RNN 的训练方式以及文本的预处理方式
 - 4) 计算效率及结果分析

3.1 CNN 相关问题

- Inception-Resnet 的大体架构如下



- 从名字也可以看出，Inception-resnet 的大体架构融合了 Inception 和 Resnet 这两种出名的 CNN 模型
 - 一般来说，更深或更宽的网络都可以提升网络的性能，但这也会带来一些问题
 - 1) 深度和宽度加大时会更容易发生过拟合
 - 2) 计算量增大
 - 解决上述不足的方法是引入稀疏特性和将全连接层转换成稀疏连接。Inception 作者的思路便是提出了一种既能保持滤波器级别的稀疏特性，又能充分密集矩阵的高计算性能的方法
 - 本模型则是加入了多个 Inception 块来解决这个问题，Inception 块（其中一种）的大致



结构如下

- 采用不同大小的卷积核意味着不同大小的感受野，最后的拼接意味着不同尺寸特征的融合
- 网络到后面，特征越抽象，且每个特征所涉及的感受野也更大

- Lin 等人在《Network in Network》一文中提出，1x1 卷积可以充当投影层，跨通道池化信息，并通过减少滤波器数量的同时保留最重要的特征，即

1) 对数据进行降维

2) 引入更多的非线性（因为卷积后会经过 ReLU 进行激活）

- 对于 Resnet 来说，它引入了 Shortcut Connection 来防止由于网络深度过大而导致的退化问题，将 Resnet 融合进 Inception 中可以搭建更深的网络，获得更好的效果
- 在具体实现的时候，我将 CNN 高层接口中顶层的全连接层摘去，使用模型余下部分构建出新的特征提取器模型
- 但在训练的时候有一个比较有趣的现象发生了，CNN 接口的参数中，weights 部分默认使用的是 'imagenet'，即使用 imagenet 数据集训练出来的参数，理论上来说，完全不同的数据集之间提取特征的方式应该有所不同，但加载默认参数后却仍能获得不错的效果（即使是 imagenet 数据集与我们的公式图片相差如此大的两个数据集之间）

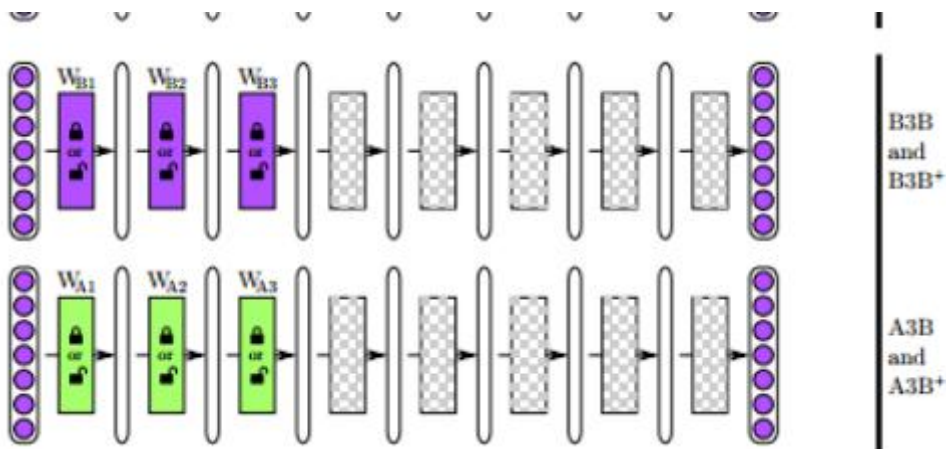
```
(function) InceptionResNetV2: (include_top=True, weights='imagenet', classifier_activation='softmax', **kwargs) -> Any
```

Instantiates the Inception-ResNet v2 architecture.

Reference:

- [Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning]
- This function returns a Keras image classification model, optionally loaded with weights.
- For image classification use cases, see [this page for detailed examples](<https://keras.io/api/applications/#usage-examples-for-image-classification-models>).

- 为了寻找理论依据，我查阅到了《How transferable are features in deep neural networks?》这篇论文，论文以早期的 AlexNet 为例，研究了 CNN 提取特征的模式。文章基于 AlexNet 在两个数据集上的实验，得到了以下结论：
 - 模型前几层的提取出来的特征并不特定于某个数据集或任务，而是一些通用的特征（比如几何特征等）
 - 在较深的几层中，提取出来的则是比较专业的特征（和数据集紧密相关）

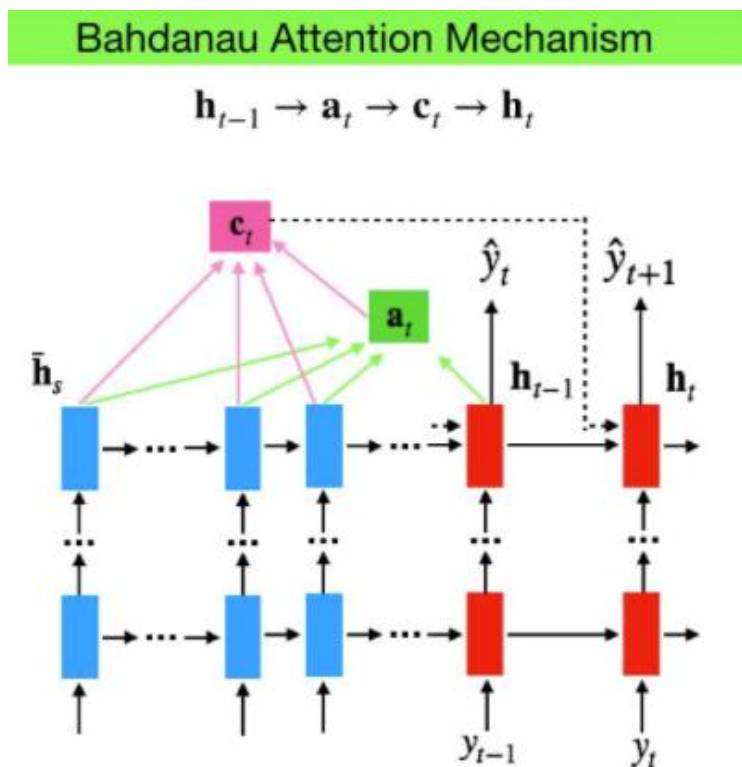


- 这其实是一个典型的迁移学习问题，我们需要冻结哪几层和解冻哪几层以获得更好的特征提取效果，这都是十分值得研究的。奈何在完整数据集上训练一次所需的时间太长，加上

InceptionResnet 也不是 Alexnet 仅有简单的 8 层，所以我这里也取了和实验三一样的比较取巧的做法，在高层接口后再接一个全连接层，将整个高层接口冻结，将最后的全连接层作为训练的目标，这样训练出来的结果也有不错的效果，但我相信，若仔细研究高层接口哪几层是比较 Specified 并将他们的参数也加入可训练权重中会获得更好的效果

3.2 Attention 层

- 若说近几年神经网络最大的几个突破，我想 Attention 层一定榜上有名，他模拟了人类视野中的注视的部分，能通过权重来让模型“关注”某些部分，以获得更好的 Decode 效果
- 本模型采用了 Bahdanau Attention 的实现方式，相较于 Luong 的 Attention 实现，少了一层额外的 RNN decoder 来计算输出，使得模型更加自然和简洁。在原本的实现中，Bahdanau Attention 是被用作 seq2seq 机器翻译的，但也可以被用作类似 caption 的任务，只不过每次输入 attention 层的不是每一时间步的输入单词，而是整个图片特征以及上一个时间步的隐藏状态



- 具体实现来说，我们将上一 GRU 隐藏状态与图片提取的特征分别通过一个全连接层，并将结果加和后使用 \tanh 函数对其进行激活

```

class Attention(Model):
    def __init__(self, units=UNITS, *args, **kwargs):
        super(Attention, self).__init__(*args, **kwargs)
        self.dense1 = layers.Dense(units)
        self.dense2 = layers.Dense(units)
        self.out = layers.Dense(1)

    def call(self, features, hidden):

        hidden_with_time = tf.expand_dims(hidden, axis=1)

        attention_hidden_layer = tf.nn.tanh(self.dense1(features)+self.dense2(hidden_with_time))

        result = self.out(attention_hidden_layer)

        weights = tf.nn.softmax(result, axis=1)

        context_vector = weights * features
        context_vector = tf.reduce_sum(context_vector, axis=1)

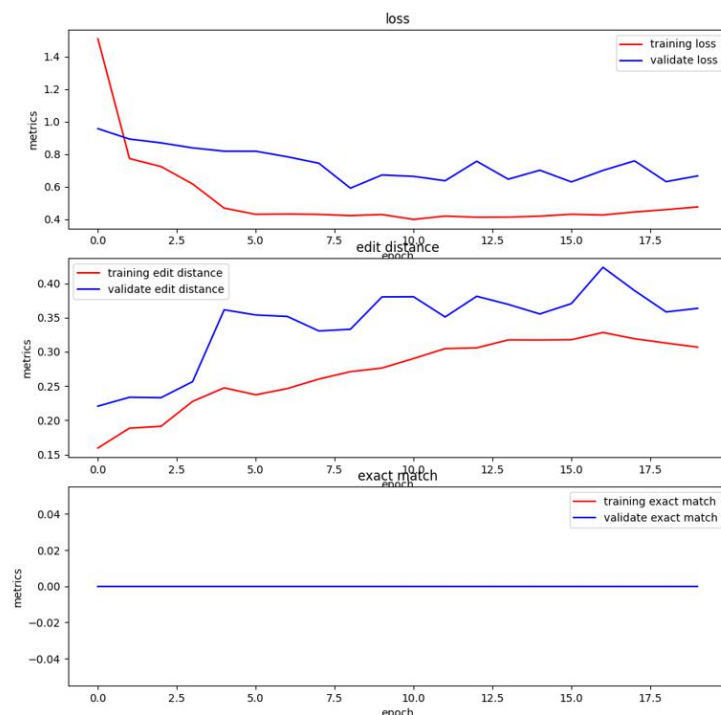
        return context_vector, weights

```

- 由于我们这里是 caption 任务，所以使用了有别于 seq2seq 的输出方式，对于我们的图片特征来说，得到的维度是(batchsize, 64, 1527)，即我们的图片特征维度为 64，所以需要对这些特征的每一维都加一个权重值，而权重值的来源就是使用 tanh 激活后的 attention 层的结果，我们对其进行降维并使用 softmax 激活后得到一个(batchsize, 64, 1)的特征值权重矩阵，与我们的原有的图片特征进行相乘后便可得到一个加上了注意力权重的特征值，让这个权重值与经过 embedding 层的结果进行连接便得到了我们真正的 GRU 输入值

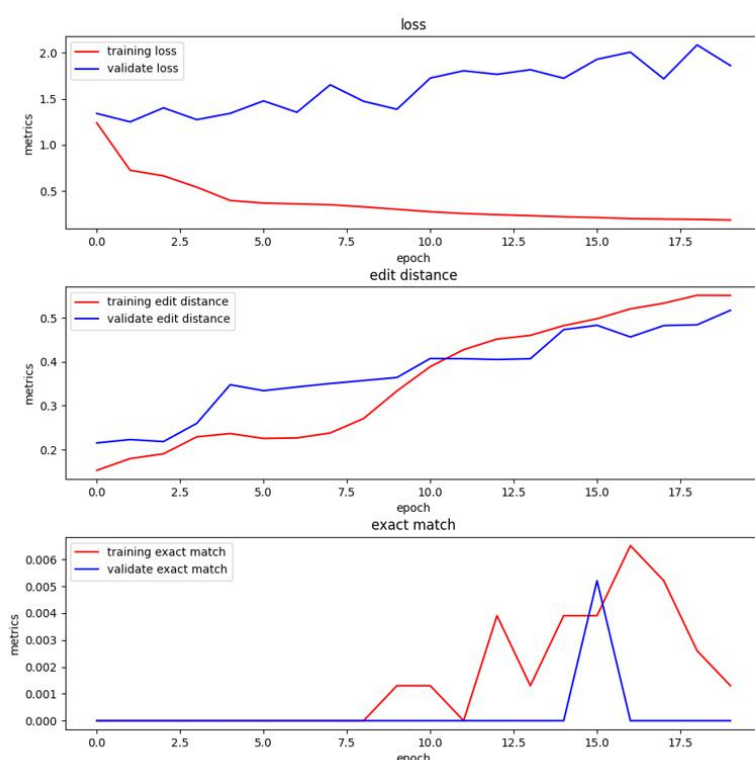
3.3 RNN 的训练方式与文本预处理方式

- 为了保证 RNN 训练的时候能够按照 batch 训练加速训练过程，在对所有的 latex 文本进行序列化后，我简单地再头尾加上<start><end>标识符，并按照最长的 latex 语句来作为基准



padding 所有的语句（使用了 post-padding,其中 padding 的部分用 0 进行填充）

- 在本模型中，RNN 的输入我是按照时间步来输入的，这就意味着每一步喂入的输入可以按照以下两种方式
 - Autoregressive
 - Teacher-Forcing
- 考虑到我们的文本长度不一且有 padding 的部分，而 padding 的部分仅仅是起到占位的作用，对权重更新不应该有贡献，由于目标序列中带着 padding 的信息（即后面的 0），所以我在 embedding 层中设置 mask_zero=true 并使用 teacher forcing 的方式来进行训练，我选了数据集中的 1000 个样本，并绘制出了几个 metrics 的图。
- 可以看出，在小批次样本的训练中，虽然 exactmatch 和 editdistance 都似乎在朝着正确的方向训练，但验证集的 loss 却一直在上升，这表明在小批次中，teacher forcing 的输入方法可能会导致模型快速地收敛，导致了很严重的过拟合
- 但是如果直接使用 Autoregressive，后面 padding 的信息无法提供。所以我在这里添加了一个 Masking 层，让目标序列通过 masking 层后得到一个布尔矩阵，False 位置的值表明了这是一个 padding 位，不应该参与计算，我将上一时间步得到的输出乘以转换为 int 后的 Masking 结果，这就既提取了目标中有关 padding 的信息，又可以充分利用 autoregressive 的优势——泛化能力强，没有那么容易过拟合。然后我在小批次上对其进行了训练，结果如下。
- 可以看出，验证集的 loss 前期虽有下降，但后期的 loss 却上下波动不收敛，且 exact match 始终没有上升，这让我怀疑我可能太小看了非 padding 部分对模型收敛的影响
- 于是我查阅资料，看到了《Scheduled Sampling for Sequence Prediction with Recurrent



Neural Networks》这篇文章，本文介绍了一种用以解决 Exposure Bias 的方法——Scheduled Sampling

- Exposure Bias 用于形容预测分布与实际分布之间的不同而形成的 Gap
- Scheduled Sampling 的思想十分简单，既然 teacher forcing 和 autoregressive 各有缺点，那就设置一个 threshold，每一步在进行输入前，先随机取一个概率，若大于 threshold 则使用 teacherforcing，否则使用 autoregressive，结果如下。可以看出，loss 变化稍微平滑了一点，editdistance 效果也略好，但 exactmatch 仍然没有进展，所以我就决定在完整数据集上跑一下看看能否有更好的表现，但是在大概第 6 个 epoch 的时候，模型近乎收敛，且效果非常不好，我的理解是，模型陷入了局部最优，且越往后取到 groundtruth 用于训练的可能越小，模型就越难跳出局部最优，最终导致模型收敛
- 最后我决定还是使用略有成效的 teacher forcing，虽然 loss 波动但是起码两个 metrics 的效果都是比较好的，最后的效果也是如此，验证集 loss 波动且有上升的趋势，但训练集的 loss 却下降的非常平滑
 - 这里我查阅资料发现有不少相同情况的人，他们将原因归结为即在训练后期，预测的结果趋向于极端，使少数预测错的样本主导了 loss，但同时少数样本不影响整体的验证 acc 情况
 - 除此之外也有一篇文章《Do We Need Zero Training Loss After Achieving Zero Training Error?》描述了这种情况，并提出了一种 flooding 方法来解决，也就是当训练集的 loss 下降到某一个阈值时，会反过来进行梯度上升，并让模型在 flooding 平面进行 random walk，让 loss 控制在某一区间。据文章的实验结果表明，效果十分不错，但本模型中没有实现，这也是优化方向之一
- 总的来说，RNN 的训练还有许多不足之处，比如解决陷入局部最优的方式应该是想办法让他跳出局部最优，而不是改用 teacher forcing 来使其泛化能力变差

3.4 计算效率及结果分析

- 在训练的时候，我在后台监控了显卡的使用效率，上方是高层接口进行特征提取时的 GPU 使用率，下面是我自定义的 RNN 对 GPU 的使用率，可以看到，虽然说 CNN 是计算密集型模型，使用率原生就会高一点，但 RNN 使用的效率也过低了，问题有几个：

- CPU 的 IO 能力和 GPU 处理数据的能力不匹配，这点我是用 tensorflow 中的 prefetch

```
Wed Dec 8 23:36:39 2021
```

NVIDIA-SMI 470.74				Driver Version: 470.74				CUDA Version: 11.4			
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.				
=====											
0	NVIDIA GeForce ...	On	00000000:08:00.0	Off				N/A			
99%	30C	P2	56W / 170W	11540MiB / 12053MiB	75%	Default	N/A				
=====											
Processes:											
GPU	GI	CI	PID	Type	Process name	GPU Memory					
ID	ID					Usage					
=====											

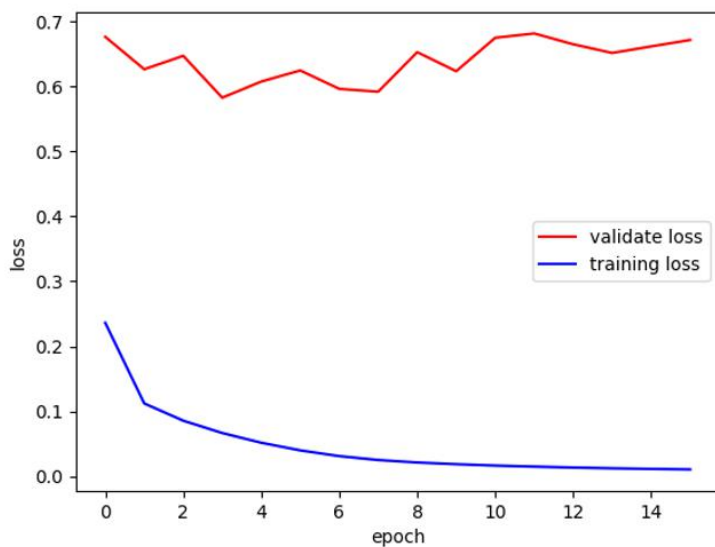
```
root@container-d0331184ac-a603fef6:~# nvidia-smi
```

```
Wed Dec 8 23:36:44 2021
```

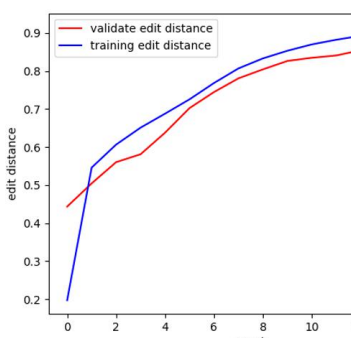
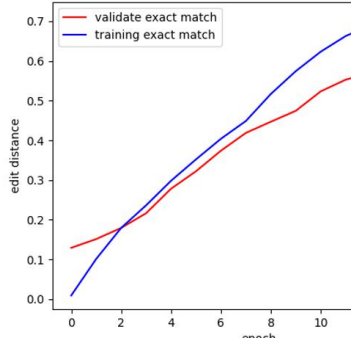
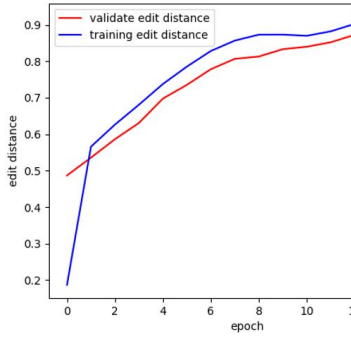
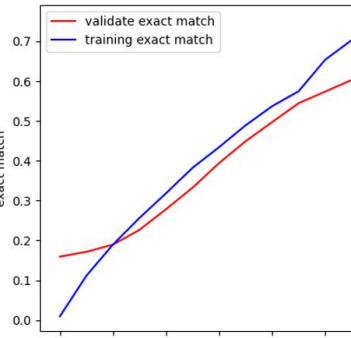
NVIDIA-SMI 470.74				Driver Version: 470.74				CUDA Version: 11.4			
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.				
=====											
0	NVIDIA GeForce ...	On	00000000:08:00.0	Off				N/A			
99%	28C	P2	40W / 170W	11540MiB / 12053MiB	3%	Default	N/A				
=====											
Processes:											
GPU	GI	CI	PID	Type	Process name	GPU Memory					
ID	ID					Usage					
=====											

提高他们的并行度解决了

- 除此之外，我还使用 tf.AUTOTUNE 来告诉 gpu 使用多线程运行，这让 RNN 计算时的效率能到 16~18% 左右，但仍不能算是重复利用了 GPU，对于一个落地的项目来说，我认为这这也是一个十分重要的优化方向
- 其次的话，RNN 中文本的 padding 我用了最偷懒的 padding 到最大长度的方法，以方便 vectorization 后并行地算出每一步的输出，但其实仔细想想，大多数文本的长度远远小于这个量，所以其实可以每一步计算一次 batch 的最大长度，仅 padding 到这个长度即可，这剩下了不少空间，也避免了很多无意义的计算
 - 当然，这个方法会导致上面第一点提到的问题，在训练读取数据的时候再 padding 增大了 CPU 和 GPU 之间的效率差异，让 GPU 空闲时间更多，运行时间更慢了
- 最后是结果的部分，之前也有提到，训练集的下降十分平滑，而验证集的 loss 不收敛甚至有上升的趋势。但与之相对的，editdistance 和 exactmatch 却同时在向好的方向优化。这是个十分有趣的问题。文献《Do we Need Zero Training Loss After Achieving Zero Training Error?》解释了这种现象，即训练后期，少数错误的样本主导了 loss 的变化，而这些错误的样本又不影响整体的准确率。这就解释了为什么 loss 一直在上升而模型却仍在被优化。



- 关于指标方面，editdistance 也能够到达基线，甚至能达到 90 多，但 exact match 方面就有点惨了，像数学数据集的 exact match 甚至只能在 60~70 左右浮动，我猜测这与特征提取器没有进行 fine tune 有关，这使得模型的效率达到了一个瓶颈。

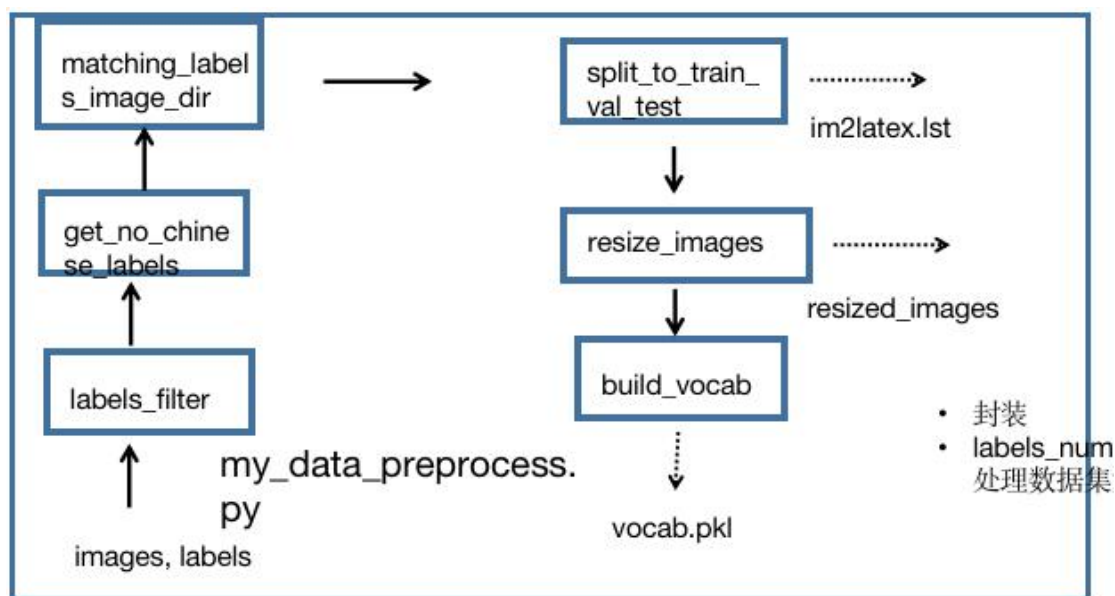
	Edit distance	Exact match
Math		
Physics		

四、Resnet-Transfomer

4.1 数据预处理 (my_data_preprocess.py)

4.1.1 思路: 数据预处理阶段要处理掉不合理的 labels (如含中文的或 `error mathpix`), 然后根据处理后的 labels 找到对应的 images 形成新的 labels 和 images 文件夹。为了方便训练时读出数据, 将 labels 的内容写入 `im2latex_formulas.norm.lst` 文件, 并将 images 的名称和索引写入 `im2latex_train_filter.lst`、`im2latex_validate_filter.lst`、`im2latex_test_filter.lst` 文件

助教之前给的预处理数据函数很分散而且不太规范和完整, 因此此处封装成了函数并且做了规范化处理。封装后的整体模型如下:



4.1.2 总流程:

原始数据集 --> labels_filter (去 `error mathpix` 得到合理的 labels) --> get_no_chinese_labels(去中文处理得到更合理的 labels) --> matching_labels_images_dir (通过 labels 将 images 对齐) --> split_to_train_val_test(原始数据集分割成训练、验证和测试数据集, 并且 shuffle 和形成 lst 文件) --> resize_images (对图片形状进行统一) --> build_vocab(生成词汇表)

4.1.3 函数说明:

(1), labels_filter () 函数

该函数主要功能是过滤掉含 `error mathpix` 的 labels。该函数参数输入: 原始 labels 文件夹路径、指定 labels 数目。函数输出: 无 `error mathpix`、labels 数目为 labels_num 的 labels 文件夹路径。主要原理是字符串的包含。值得注意的是此处新增了 labels_num 参数, 其主要目的是可以指定形成的 labels 数目, 更加个性化, 指定小的数据集, 以方便未来测试跑通模型时可以节约时间, 当然如果不指定 labels_num 表会对文件夹的所有文件进行操作形成完整的数据集 (后文的 labels_num 同理)。

(2), get_no_chinese_labels()

该函数的目的是过滤掉含中文的标签。函数输入参数: vocab.txt 路径、原始 labels 文件夹路径 (一般为 `no_error_mathpix_labels_dir_path`), 指定的 labels 数目 labels_num (不含中文的标签数目+含中文的标签数目), 输出: 不含中文的 labels 文件夹路径、含中文的 labels 文件夹

径、检测到的中文字符串文件 txt 路径。其总的思路: 首先读取原始 labels 文件夹内容为 content, 通过 FMM_func 和 vocab 文件, 对 content 进行最长 vocab 词汇表匹配得到 content 的 token_list, 对 token_list 的每个 token 判断是否在 vocab 中出现, 如果没有出现则该 token 含中文。

(3), matching_labels_images_dir()

该函数的目的在于通过 labels 文件名找到对应的 images 文件从而将 images 文件和 labels 文件对齐 (因为上文的函数处理过后会形成新的 labels 文件夹, 因此必须实现对齐操作)。

函数参数输入: 需要对齐的 labels 文件夹路径、最原始的 images 文件夹, 函数参数输出: 对齐后的 images 文件夹路径。实现原理: 通过遍历 labels 文件名, 以 labels 文件名在最原始的 images 文件夹中查找, 然后粘贴到新的 images 文件夹。

(4), split_to_train_val_test():

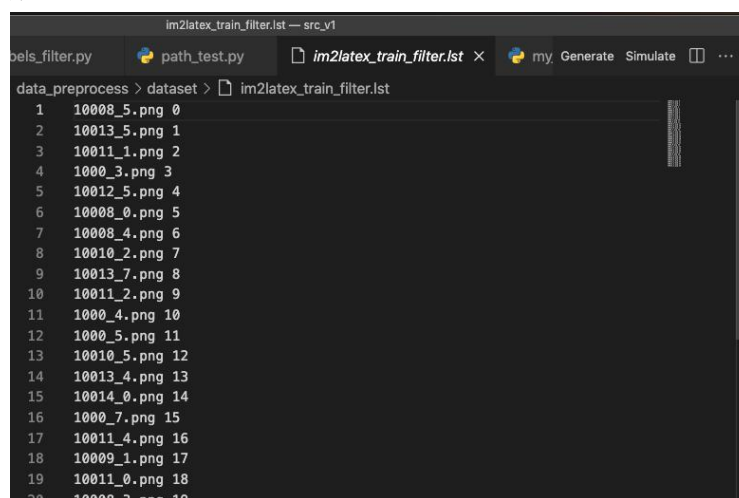
该函数的功能在于 shuffle, 并将原始数据集分成 train、val、test 三个数据集, 并且分别形成 lst 文件。函数参数输入: labels 文件夹路径、images 文件夹、指定 labels 数目 labels_num, 函数输出: lst 文件目录路径、lst 文件路径。实现原理: 原理比较简单, shuffle 读取到的文件 (shuffle 前一定要执行对文件夹 sorted 操作才能将 labels 文件和 images 文件对齐) 后分别将 images 名称和索引写入对应的 lst 文件, 如 0 ~ train_num-1 写入 im2latex_train_filter.lst, train_num~train_num + val_num-1 写入 im2latex_val_filter.lst, 剩下的写入 im2latex_test_filter.lst。全部的 labels 写入 im2latex_formulas.norm.lst。

之所以这其中有如下 sorted 代码是为了使指定图片数量时我们能够将图片名称和文字名称成功对上, 将 sorted 后的两个 list 打印出来, 打印结果:

```
['0_0.png', '0_1.png', '0_2.png', '0_3.png', '0_4.png', '0_5.png', '0_6.png', '10000_0.png', '10000_1.png', '10000_2.png', '10000_3.png', '10000_4.png', '10000_5.png', '10000_6.png', '10001_0.png', '10001_1.png', '10001_2.png', '10002_0.png', '10002_1.png', '10002_2.png']  
['0_0.txt', '0_1.txt', '0_2.txt', '0_3.txt', '0_4.txt', '0_5.txt', '0_6.txt', '10000_0.txt', '10000_1.txt', '10000_2.txt', '10000_3.txt', '10000_4.txt', '10000_5.txt', '10000_6.txt', '10001_0.txt', '10001_1.txt', '10001_2.txt', '10002_0.txt', '10002_1.txt', '10002_2.txt']
```

可见文件名称成功对上, 能够指定文件数方便自身 cpu 电脑进行模型跑通测试。

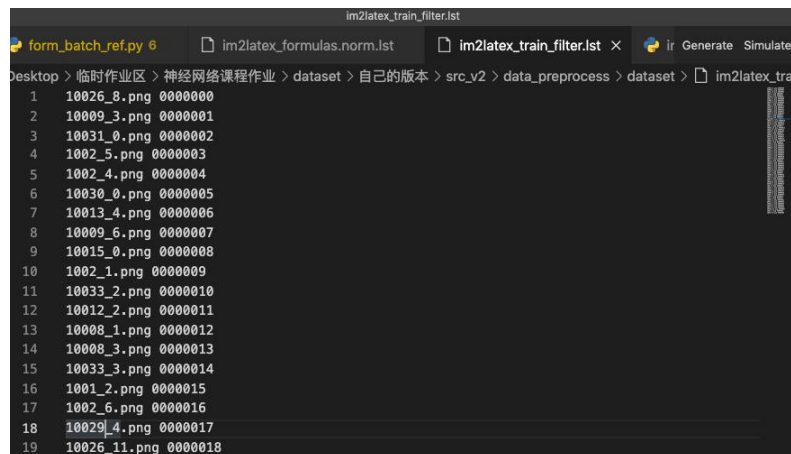
lst 文件夹部分内容展示如下:



```
im2latex_train_filter.lst -- src_v1  
data_preprocess > dataset > im2latex_train_filter.lst  
1 10008_5.png 0  
2 10013_5.png 1  
3 10011_1.png 2  
4 1000_3.png 3  
5 10012_5.png 4  
6 10008_0.png 5  
7 10008_4.png 6  
8 10010_2.png 7  
9 10013_7.png 8  
10 10011_2.png 9  
11 1000_4.png 10  
12 1000_5.png 11  
13 10010_5.png 12  
14 10013_4.png 13  
15 10014_0.png 14  
16 1000_7.png 15  
17 10011_4.png 16  
18 10009_1.png 17  
19 10011_0.png 18  
20 10008_3.png 19
```

但是这里有些问题因为后面的代码需要从 im2latex_train_filter.lst 读取图片文字名, 但是后面的

数字长度并不统一会形成读取图片文字名的困难,因此我们可以将其该进成后面的索引名是统一的 7 长度情况如 0000001 等。改进的方法主要是用到 `zfill` 函数。效果图如下:



```
im2latex_train_filter.lst
form_batch_ref.py 6  im2latex_formulas.norm.lst  im2latex_train_filter.lst  Generate  Simulate
Desktop > 临时作业区 > 神经网络课程作业 > dataset > 自己的版本 > src_v2 > data_preprocess > dataset > im2latex_tra
1 10026_8.png 0000000
2 10009_3.png 0000001
3 10031_0.png 0000002
4 1002_5.png 0000003
5 1002_4.png 0000004
6 10030_0.png 0000005
7 10013_4.png 0000006
8 10009_6.png 0000007
9 10015_0.png 0000008
10 1002_1.png 0000009
11 10033_2.png 0000010
12 10012_2.png 0000011
13 10008_1.png 0000012
14 10008_3.png 0000013
15 10033_3.png 0000014
16 1001_2.png 0000015
17 1002_6.png 0000016
18 10029_4.png 0000017
19 10026_11.png 0000018
```

(5) `build_vocab()`:

由于输入模型的不可能是直接的 `label`(因为 `label` 是字符串),因此我们可以参照 `image-caption` 将 `label` 转化成 `label_ids`,此时 `label_ids` 可以作为 `Tensor` 输入模型。模块输入参数: `vocab.txt` 的路径,指定的 `vocab.pkl` 路径,模块函数输出: `vocab.pkl` 文件。原理: 该函数的关键在于 `word` 和 `id` 的相互映射,该映射写成 `Vocabulary` 类,每添加一个词就增添一对 `id` 和 `word` 的映射。`Vocabulary` 类最初含有的默认词汇有`<pad>`、`<start>`、`<end>`、`<unk>`。其中 `start` 和 `end` 标记语句的开始和结束,当语句中出现 `vocab.txt` 没有的词汇时会返回 `unk`, `pad` 是 0 号词汇作用是对短的 `label_ids` 进行填充使 `label_ids` 的形状统一。再加入默认词汇后逐一将 `vocab.txt` 中的词汇添加入词典便可。值得注意的是去掉 `vocab.txt` 每个词后面的`'\n'`。在形成正确的 `Vocabulary` 类后以二进制的方式写成文件 `vocab.pkl` 方便未来读取。

4.1.4 数据加载(my_data_loader.py)

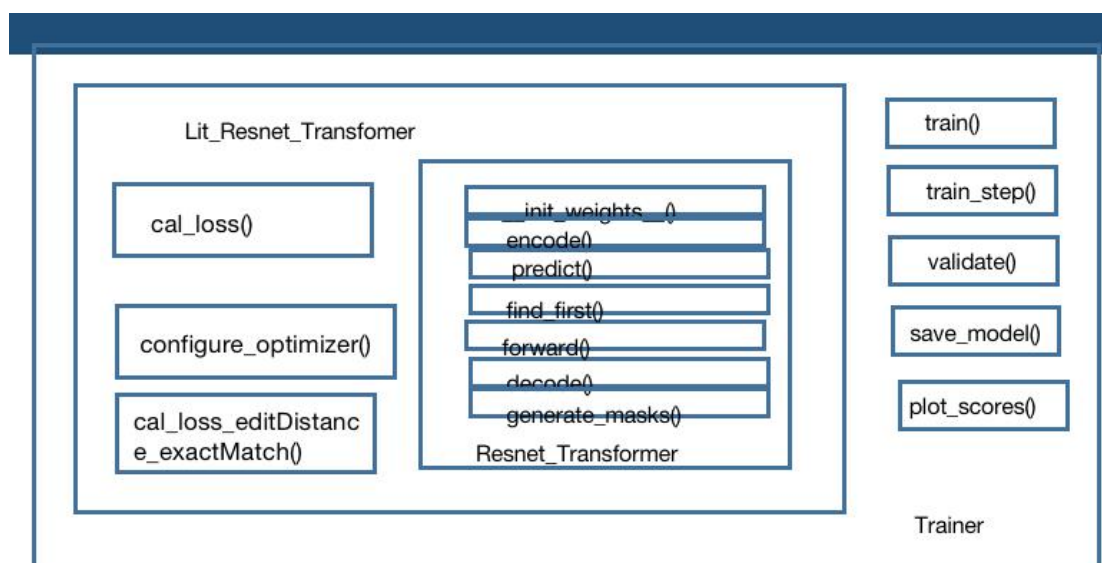
参考 `image-caption` 模型可知模型的输入常常是 `batch` 加载,输入形状常常是`[batch_size, channel_num, height, width]`。因此我们需要写一个能够以 `batch` 加载数据的函数。该函数被封装成 `get_loader`,函数参数输入: `labels_lst` 文件路径(即上文的 `im2latex_formulas_norm.lst`), `images` 目录(即上文对齐后且 `resized` 后的 `images` 文件夹)、`images_lst` 文件路径(即上文的 `im2latex_train_filter.lst` 等三个文件)、批大小 `batch_size`、`transform` (`Tensor` 转化指定)。函数返回: 按批组织好的 `images Tensor`, `labels`。函数原理为: 首先读取 `images_lst` 内容,形成 `images` 路径和 `images` 对应的 `labels` 索引,然后通过 `images` 路径找到对应的图片,使用 `PIL Image` 类转化成 `Tensor`,再通过 `labels` 索引和 `labels lst` 文件形成 `labels` 内容,同时我们要利用好上文的 `Vocabulary` 类将 `labels` 内容正确转化成 `label_ids`。将以上 `images` 和 `label_ids` 封装入 `pytorch data.Dataset` 内部,封装好后便可通过 `torch.utils.data.DataLoader` 来批次加载。

另外值的注意的是 `torch.utils.data.DataLoader` 中一定要有 `collate_fn` 函数,因为原始的 `label_ids` 的 `Tensor` 长度并不统一,因此 `collate_fn` 的作用是找到 `label_ids` 的最大长度,使短的 `label_ids` 被正确地 `pad` 成相同的长度。

4.2 基本模型复现

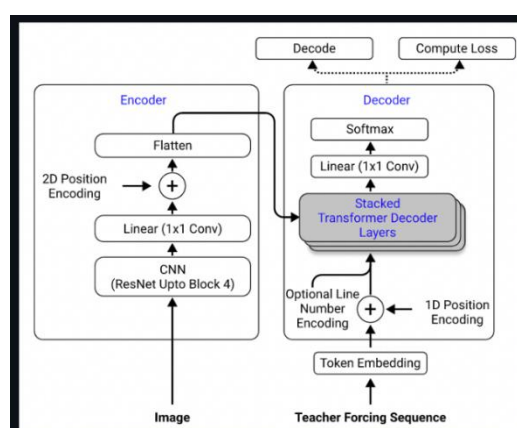
4.2.1 体系结构与模块

基本模型复现中我们参考 <https://github.com/kingyiusuen/image-to-latex> 项目进行代码重构，体系结构如下图：



(1) models.py

本文的模型搭建基于 github image-to-latex 模型。模型结构图如下：



模型保存在 models.py 文件中具体模型 models.py 主要有 ResNet_Transformer 类, 主要函数有 encode()、decode()、forward()、predict()、generate_square_subsequent_mask()、find_first()。其中 encode()、decode()是具体模型的搭建, forward()将 encode()、decode()正确连接, predict()作用是输入图像得到预测的 label_ids 序列 (有了序列才能方便转化成对应的词汇)。

值得注意的是 generate_square_subsequent_mask()生成掩码主要是在搭建模型中使用到, find_first()找到句子里出现的第一个指定词汇 (用于找到结束符 '<end>'), 该函数主要用于 predict()时找到语句结束符。ignore_index = [eos_index, sos_index, pad_index, unk_index]指的是 '<end>', '<start>', '<pad>', '<unk>' 所对应的 id。

(2) score.py

实现了 Edit_distance 和 Exact_match 的两个评价指标, 具体实现为 exact_match_score()、edit_distance()。exact_match_score() 用小数表示, 表示完全匹配的数量占总数据量的比值; edit_distance() 为取反表示, 越高越好。

均为对一个批次数据对指标衡量。对一个 epoch 的衡量即为批次衡量的平均值。

(3) lit_models.py

封装的是对 models.py 的调用类 Lit_Resnet_Transformer，其主要函数包含 cal_loss()、cal_loss_editDistance_exactMatch()、configure_optimizer()。主要作用是对训练步骤和验证步骤进行封装。

(4) training.py

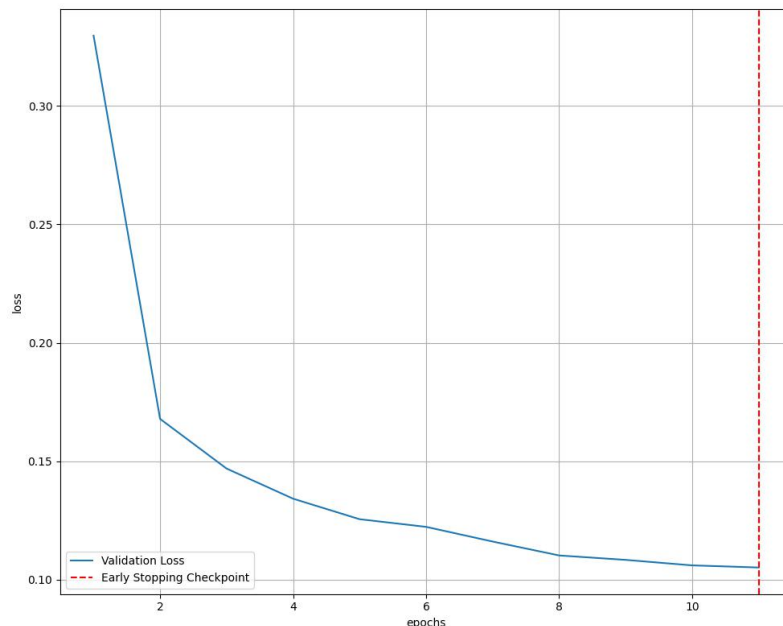
training.py 中包含着类 Trainer,该类对整个训练流程（包含数据加载、模型初始化、训练步骤、验证步骤、保存训练模型、绘制训练结果）进行完整封装，使代码可读性更强、更加简洁。关键代码包含 train()、train_step()、validation()、save_model()、plot_losses()、plot_scores(),

(5) utils.py

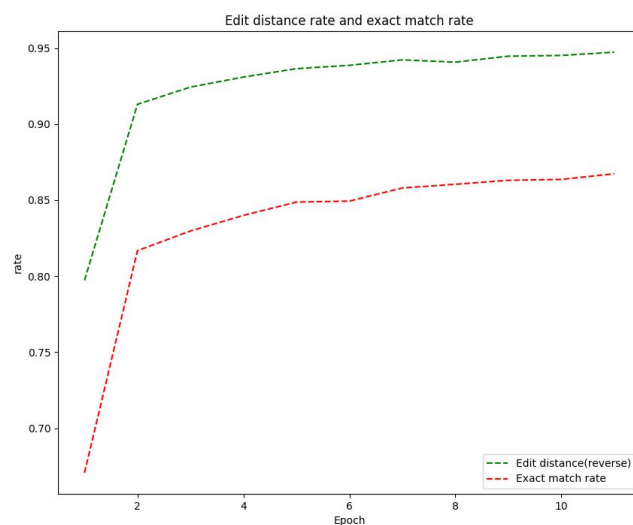
包含了项目所使用的工具函数。分别为与 checkPoint 相关的 get_checkpoint ()、get_best_checkpoint ()，用于获取之前保存的模型；与过拟合处理有关的 EarlyStopping 类，实现了当验证损失在一定区间不再提升后的早停；与部署相关的 decode ()、resize_image ()，实现了对输入图片的预处理以及对模型预测输出的解码。

4.2.2 基本模型复现结果:

epoch 为 10 损失情况为:



此时最优 edit distance 和 exact match rate 情况为:

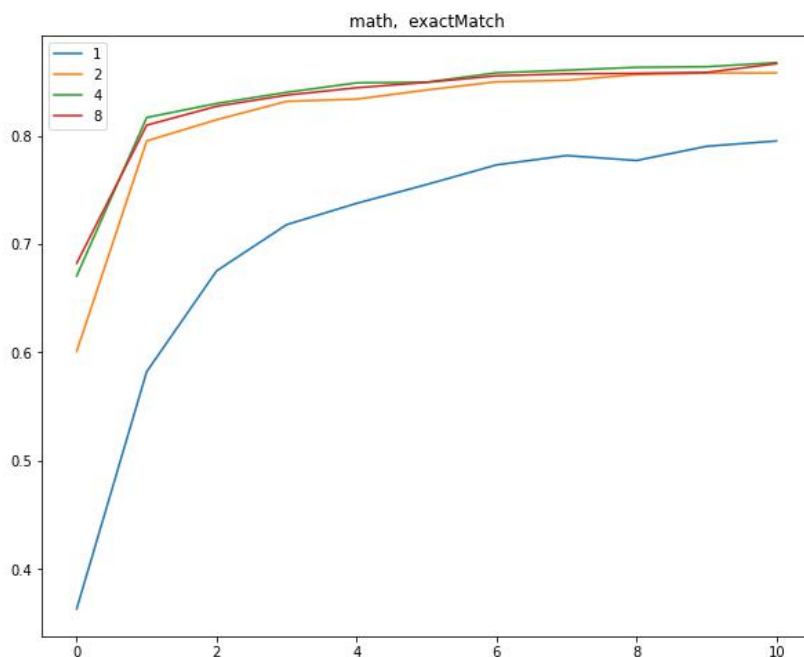


可以看到，两者均超过了 0.85 的基线，指标效果良好。

4.3 优化:

(1) 训练时优化

a) 使用了梯度累积方法，变相扩大 batch size，加速训练时梯度更新。



分别在数学模型上测试了不同的梯度累积步数。当 Batch size 为 8 时，可以看到，当步数由 1 到至其他时，exactMatch 指标的提升速度与效果最好；其他情况下则差距不明显；当 batch size 为 64 时，该实验效果不佳，当梯度累积步数增加时误差下降减缓；因此可以大致推断出，由于实际 batch size = 设定 batch size * 梯度累积步数，该实际 batch size 存在一个阈值，阈值之下梯度累积效果能取得正效果，超过阈值则效果不佳。

b) 添加了梯度截断，防止梯度爆炸。(实际上没有用到，因为没有梯度爆炸)

(2) 过拟合处理

a) Dropout

```
lit_model = Lit_Resnet_Transformer(d_model=128, dim_feedforward=256,  
                                   nhead=4, dropout=0.3, num_decoder_layers=3,  
                                   max_output_len=300, sos_index=vocab('<start>'),  
                                   eos_index=vocab('<end>'), pad_index=vocab('<pad>'),  
                                   unk_index=vocab('<unk>'), num_classes=num_classes)
```

dropout 设置为 0.3, 随机删除 30% 神经元, 获得不同的神经网络。对很多个不同的神经网络取平均, 通过不同的网络产生不同的过拟合, 来一定程度抵消整体的过拟合。

b) EarlyStopping

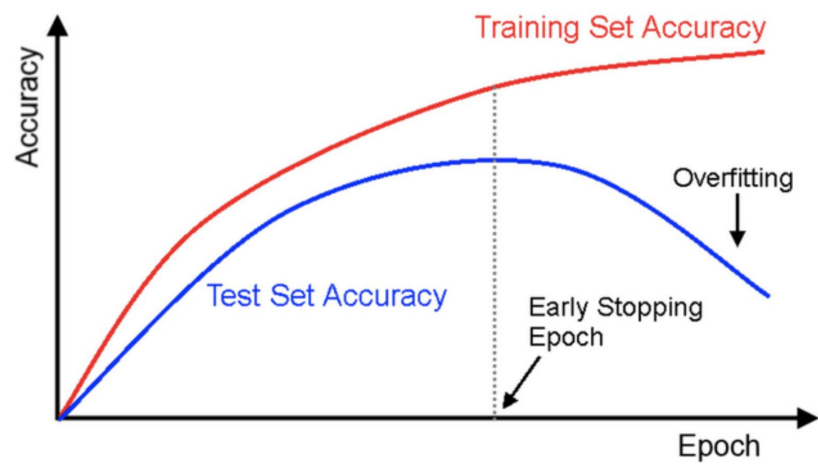
可以达到当训练集上的 loss 减小的程度小于某个阈值 (设置为 20) 的时候停止继续训练。由于过拟合会导致继续训练导致测试准确率下降, 该方法通过对准确率的检测对过拟合作出了限制。

```
# early stopping patience; how long to wait after last time validation loss improved.  
patience = 20  
early_stopping = EarlyStopping(patience=patience, verbose=True)
```

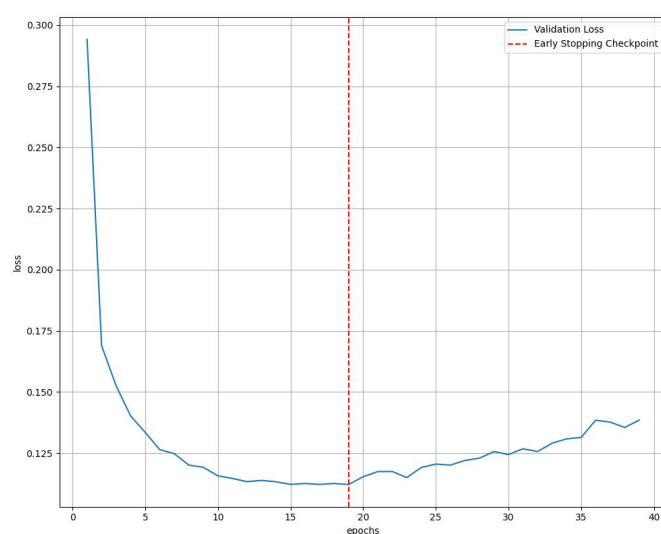
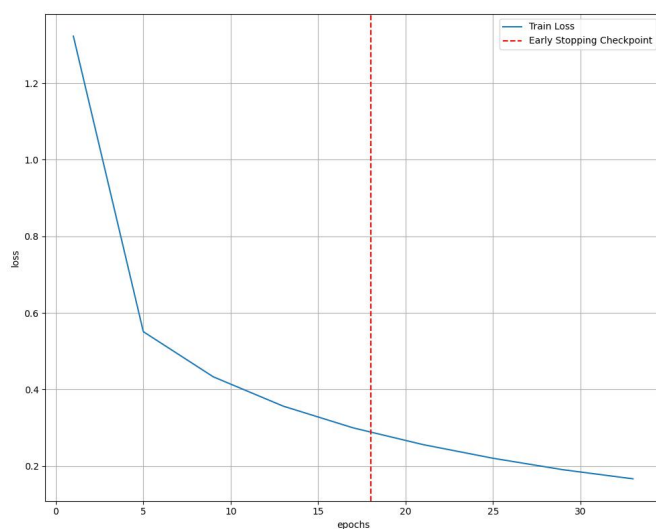
```
# early_stopping needs the validation loss to check if it has decreased,  
# and if it has, it will make a checkpoint of the current model  
early_stopping(valid_loss, self.model.models)  
  
if early_stopping.early_stop:  
    print("Early stopping")  
    # load the last checkpoint with the best model  
    self.save_model('checkpoint_ckpt')  
    break
```

实际上也没有使用到该正则化方法, 因为我们的模型在 epoch 为 10 的时候就超过了 85% 的基线, 并未出现过拟合情况。

当 epoch 继续增加, 当 max_epoch=40 时在 epoch 为 38 时早停, 由于 patience 值为 20, 可推出在 epoch=18 时测试损失不再下降, 此时有目前为止最好的验证集精度, 而随着 epoch 的增加, 在验证集上发现测试误差上升, 此时为过拟合, 停止训练。



Early Stopping



(3) Padding Mask

在此模型中, LaTeX 表达式会被 padding 成每个 batch 最大的长度, 为了不让 transformer 把注意力放在 padding token '0' 上, 因此考虑加入 padding mask, 把用于 padding 的'0' mask 掉.

具体做法:找出某个 batch 中的最大序列的长度, 然后产生一个[batch_size, max_length], 类型为 bool 的矩阵, 然后根据每个 LaTeX 表达式的长度, 非 padding_token 对应 False, padding_token 对应 True.

```
def generate_padding_mask(length) -> Tensor:
    """
    给定一个batch数据的length, 返回padding_mask.
    返回的Tensor为True的地方表示是padding(0)

    length: list
    return: torch.Tensor

    Usage:
    >>> generate_padding_mask([3,1,2])
    tensor([[False, False, False],
           [False, True, True],
           [False, False, True]])
    """
    bs = len(length)
    max_len = max(length)
    mask = torch.full((bs, max_len), 1)
    for i, l in enumerate(length):
        mask[i, :l] = 0
    mask = mask.bool()

    return mask
```

(4) Mask Language Model Pretrain

原因:这个任务算是半个语言模型。就比如说 LaTeX 它和自然语言一样, 也要遵循一定的语法规则, 比如说括号匹配。Google 在训练 BERT (由 transformer 堆叠) 的时候用 MLM (Mask Language Model) 来预训练, 所以这里也试一下

做法: 对于一个 LaTeX 表达式, 随机选择 15% 的 token, 其中 80% 换成 '[MASK]' 10% 随机换成别的 token, 10% 不变。在 train 和 validation 的时候都这样做

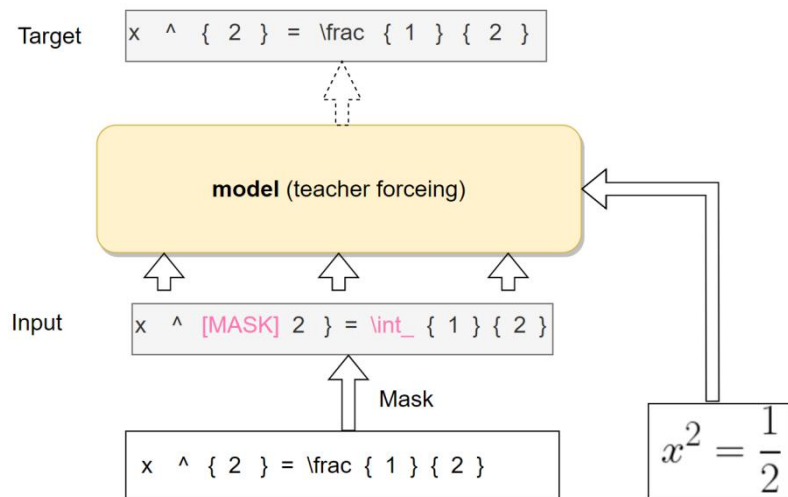
(详情在 BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding 这篇论文的 3.1 节 Task #1:

Masked LM 的最后一段)

简单来说就是扰乱数据集, 换成 '[MASK]' 或者别的词, 但 target 仍然是没有被扰乱的数据,

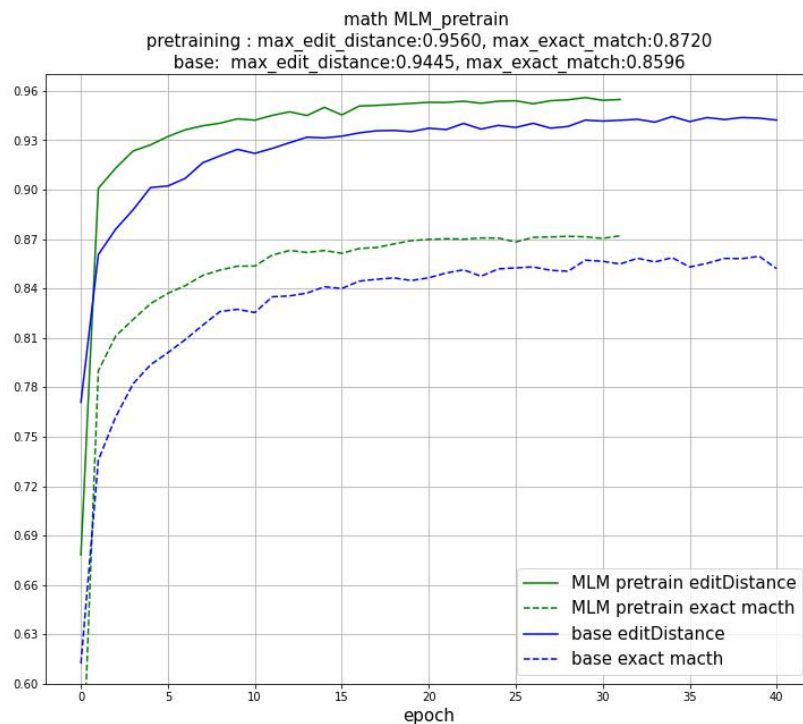
也就是说希望模型在数据集被扰乱的情况下还能正确预测。

具体图示如下：



其中值得注意的是，虽然 input 被 Mask 掉了，但是 target 仍然为标准的数据。

效果：



上图，预训练过程中评价指标的变化。Pretraining 表示使用 MLM 预训练的评价指标变化，base 表示按照 github 复现的基本模型的评价指标的变化

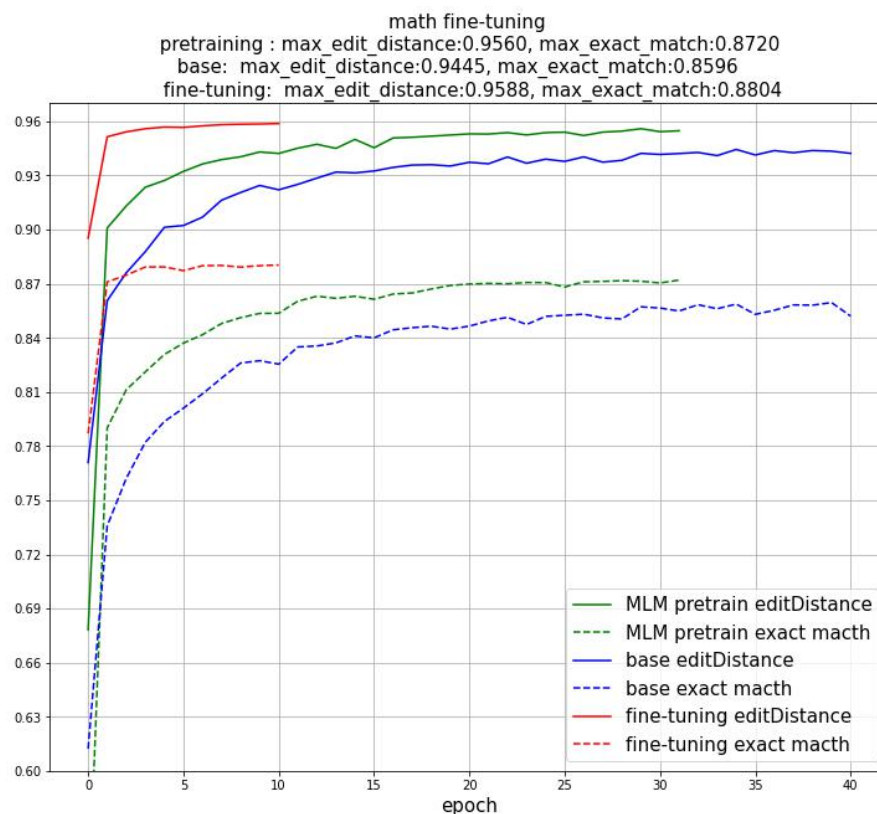
可以看到，在预训练过程中评价指标已经比 base 模型的高了，再在预训练模型的基础上微

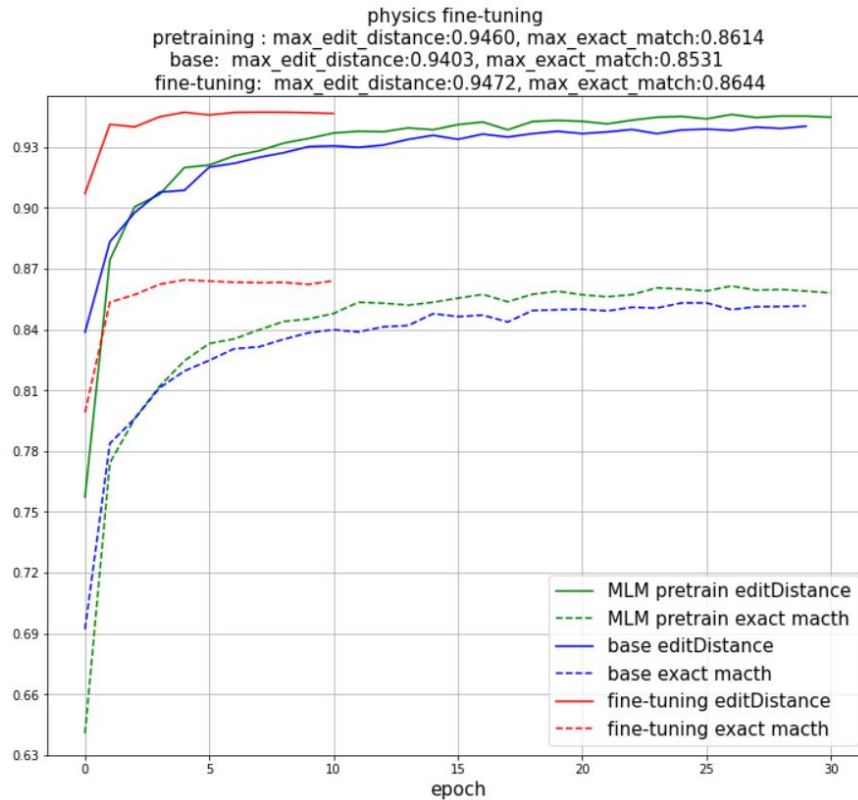
调(fine-tuning) 应该会有更好的结果。

4.4 Fine-tuning

在用 MLM 大概预训练 45K 个 step (≈ 31 个 epoch) (因为原文的最低预训练 step 大概就在 45K steps 左右), 然后用保存下来的模型在标准数据集上训练 10 个 epoch:

下面两幅图是在数学和物理数据集上分布对预训练模型进行 fine-tuning 的评价指标随 epoch 的变化, 可以看到模型收敛得非常快, 10 个 epoch 基本上就已经收敛完了。





4.5 最终结果

Tasks	Dev Set			
	Math		Physics	
	Edit Distance	Exact Match	Edit Distance	Exact Match
King	0.9445	0.8596	0.9398	0.8531
ACU	0.9426	0.8664	0.9437	0.8586
MLM	0.9588	0.8804	0.9472	0.8644

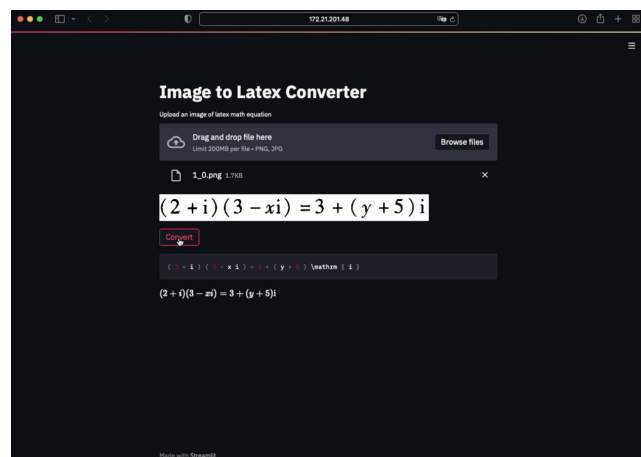
King 为按照 github 作者 King Yiu Suen 模型,不用任何优化的结果

ACU 为使用梯度累计(accumulation_step=8) 的结果

MLM 为使用 Mask Language Model 预训练模型微调结果

五、部署

后端 **Api**:使用 **fastapi**, 实现了对最佳模型、词典的读取以及对输入图像的预处理、预测与解码。
 前端 **streamlit**: 对生成对 **api** 进行调用获取信息, 实现了简洁的前端页面, 支持拖拽图片上传与结果展示。



小结

小组成员经过搭建 Encoder-Decoder 模型小组成功解决 Image-to-Latex 问题, 并且对其原理有了更深层次的体会。

指导教师评语及成绩

评语:

成绩: 指导教师签名:
批阅日期: