混合文本Latex识别

课程设计报告

邵钊明

目录

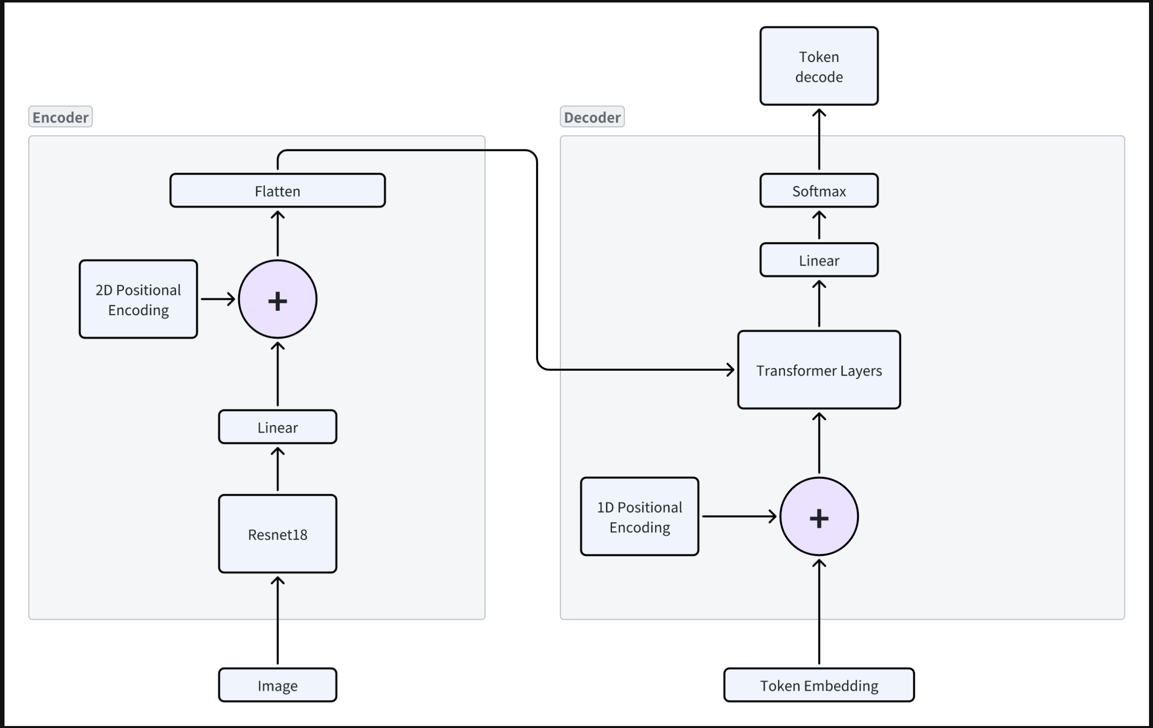
模型架构迭代

1. Resnet - ***LSTM*** (Lab3)
2. Resnet&***PD*** - ***Transformer***[1]
3. ***ViT*** - Transformer[2]
4. 组合 ***Yolov8*** 目标提取能力

[1]https://github.com/kingyiusuen/image-to-latex [2]https://github.com/lukas-blecher/LaTeX-OCR

Resnet&***PD*** - ***Transformer***

Resnet&PD-Transformer - 模型架构图



Resnet&PD - Transformer 模型架构图

Resnet&PD-Transformer - 实现

```python
def encode(self, x: Tensor) -> Tensor:
    x = x.float()
    if x.shape[1] == 1:
        x = x.repeat(1, 3, 1, 1)
    x = self.backbone(x)  # 经过Resnet
    x = self.bottleneck(x)
    x = self.image_positional_encoder(x)  # 图像位置编码器
    x = x.flatten(start_dim=2)  # 展平
    x = x.permute(2, 0, 1)  # 转置维度
    return x

def decode(self, y: Tensor, encoded_x: Tensor) -> Tensor:
    y = y.permute(1, 0)
    y = self.embedding(y) * math.sqrt(self.d_model)  # 词嵌入并缩放
    y = self.word_positional_encoder(y)  # 词位置编码器
    Sy = y.shape[0]
    y_mask = self.y_mask[:Sy, :Sy].type_as(encoded_x)  # 生成目标序列的掩码
    output = self.transformer_decoder(y, encoded_x, y_mask)  # 经过Transformer解码器
    output = self.fc(output)  # 全连接层输出
    return output
```

Resnet&PD-Transformer - 实现

```python
def encode(self, x: Tensor) -> Tensor:
    x = x.float()
    if x.shape[1] == 1:
        x = x.repeat(1, 3, 1, 1)
    x = self.backbone(x)  # 经过Resnet
    x = self.bottleneck(x)
    x = self.image_positional_encoder(x)  # 图像位置编码器
    x = x.flatten(start_dim=2)  # 展平
    x = x.permute(2, 0, 1)  # 转置维度
    return x

def decode(self, y: Tensor, encoded_x: Tensor) -> Tensor:
    y = y.permute(1, 0)
    y = self.embedding(y) * math.sqrt(self.d_model)  # 词嵌入并缩放
    y = self.word_positional_encoder(y)  # 词位置编码器
    Sy = y.shape[0]
    y_mask = self.y_mask[:Sy, :Sy].type_as(encoded_x)  # 生成目标序列的掩码
    output = self.transformer_decoder(y, encoded_x, y_mask)  # 经过Transformer解码器
    output = self.fc(output)  # 全连接层输出
    return output
```

Resnet&PD-Transformer - 实现

```python
def encode(self, x: Tensor) -> Tensor:
    x = x.float()
    if x.shape[1] == 1:
        x = x.repeat(1, 3, 1, 1)
    x = self.backbone(x)  # 经过Resnet
    x = self.bottleneck(x)
    x = self.image_positional_encoder(x)  # 图像位置编码器
    x = x.flatten(start_dim=2)  # 展平
    x = x.permute(2, 0, 1)  # 转置维度
    return x

def decode(self, y: Tensor, encoded_x: Tensor) -> Tensor:
    y = y.permute(1, 0)
    y = self.embedding(y) * math.sqrt(self.d_model)  # 词嵌入并缩放
    y = self.word_positional_encoder(y)  # 词位置编码器
    Sy = y.shape[0]
    y_mask = self.y_mask[:Sy, :Sy].type_as(encoded_x)  # 生成目标序列的掩码
    output = self.transformer_decoder(y, encoded_x, y_mask)  # 经过Transformer解码器
    output = self.fc(output)  # 全连接层输出
    return output
```

Resnet&PD-Transformer - 实现

```python
def encode(self, x: Tensor) -> Tensor:
    x = x.float()
    if x.shape[1] == 1:
        x = x.repeat(1, 3, 1, 1)
    x = self.backbone(x)  # 经过Resnet
    x = self.bottleneck(x)
    x = self.image_positional_encoder(x)  # 图像位置编码器
    x = x.flatten(start_dim=2)  # 展平
    x = x.permute(2, 0, 1)  # 转置维度
    return x

def decode(self, y: Tensor, encoded_x: Tensor) -> Tensor:
    y = y.permute(1, 0)
    y = self.embedding(y) * math.sqrt(self.d_model)  # 词嵌入并缩放
    y = self.word_positional_encoder(y)  # 词位置编码器
    Sy = y.shape[0]
    y_mask = self.y_mask[:Sy, :Sy].type_as(encoded_x)  # 生成目标序列的掩码
    output = self.transformer_decoder(y, encoded_x, y_mask)  # 经过Transformer解码器
    output = self.fc(output)  # 全连接层输出
    return output
```

Resnet&PD-Transformer - 实现

```python
def encode(self, x: Tensor) -> Tensor:
    x = x.float()
    if x.shape[1] == 1:
        x = x.repeat(1, 3, 1, 1)
    x = self.backbone(x)  # 经过Resnet
    x = self.bottleneck(x)
    x = self.image_positional_encoder(x)  # 图像位置编码器
    x = x.flatten(start_dim=2)  # 展平
    x = x.permute(2, 0, 1)  # 转置维度
    return x

def decode(self, y: Tensor, encoded_x: Tensor) -> Tensor:
    y = y.permute(1, 0)
    y = self.embedding(y) * math.sqrt(self.d_model)  # 词嵌入并缩放
    y = self.word_positional_encoder(y)  # 词位置编码器
    Sy = y.shape[0]
    y_mask = self.y_mask[:Sy, :Sy].type_as(encoded_x)  # 生成目标序列的掩码
    output = self.transformer_decoder(y, encoded_x, y_mask)  # 经过Transformer解码器
    output = self.fc(output)  # 全连接层输出
    return output
```

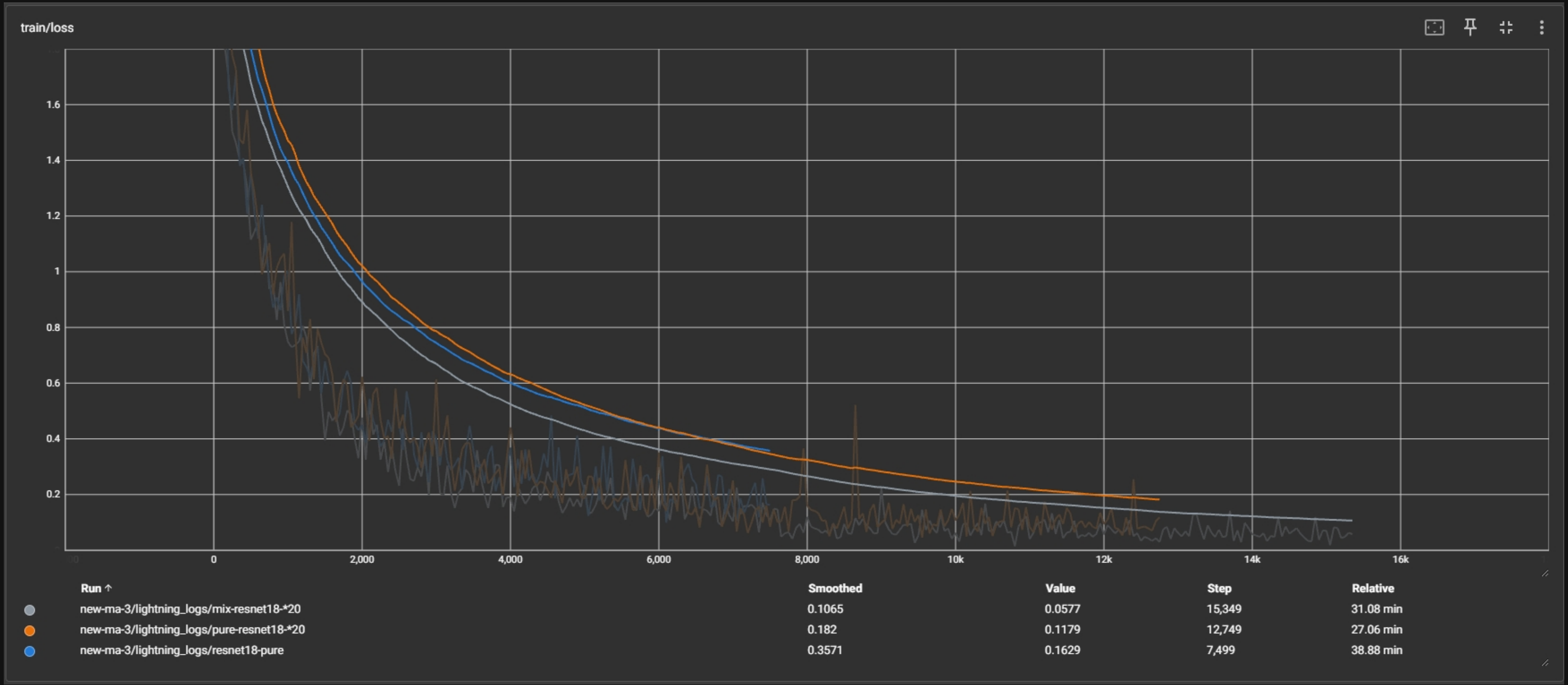
Resnet&PD-Transformer - 实现

```python
def encode(self, x: Tensor) -> Tensor:
    x = x.float()
    if x.shape[1] == 1:
        x = x.repeat(1, 3, 1, 1)
    x = self.backbone(x)  # 经过Resnet
    x = self.bottleneck(x)
    x = self.image_positional_encoder(x)  # 图像位置编码器
    x = x.flatten(start_dim=2)  # 展平
    x = x.permute(2, 0, 1)  # 转置维度
    return x

def decode(self, y: Tensor, encoded_x: Tensor) -> Tensor:
    y = y.permute(1, 0)
    y = self.embedding(y) * math.sqrt(self.d_model)  # 词嵌入并缩放
    y = self.word_positional_encoder(y)  # 词位置编码器
    Sy = y.shape[0]
    y_mask = self.y_mask[:Sy, :Sy].type_as(encoded_x)  # 生成目标序列的掩码
    output = self.transformer_decoder(y, encoded_x, y_mask)  # 经过Transformer解码器
    output = self.fc(output)  # 全连接层输出
    return output

```

Resnet&PD-Transformer - 训练过程



Resnet&PD - Transformer 训练过程

改进

观察`Resnet&PD-Transformer`模型的测试数据，我们发现该模型的训练损失和实际推理能力差距过大，存在严重的过拟合情况。

我们推测可能是因为模型前部Encoder使用CNN实现，尽管已经引入了位置编码能力，仍然存在缺乏上下文能力的问题

我们决定将Encoder部分也引入注意力机制，使用ViT模型实现。

ViT-Transformer

ViT-Transformer - 模型架构

ViT - Transformer模型架构

ViT - Transformer模型架构

ViT-Transformer - Vit实现

```python
def vit_forward(self, img, **kwargs):
    p = self.patch_size
    # 重排输入图像的维度，将其划分为大小为 p x p 的块，并重新排列维度
    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
    x = self.patch_to_embedding(x)
    b, n, _ = x.shape

    # 生成类别令牌，并将其与嵌入的图像块连接起来
    cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
    x = torch.cat((cls_tokens, x), dim=1)

    # 计算位置编码
    h, w = torch.tensor(img.shape[2:])//p
    pos_emb_ind = repeat(torch.arange(h)*(self.max_width//p-w), 'h -> (h w)', w=w)+torch.arange(h*w)
    pos_emb_ind = torch.cat((torch.zeros(1), pos_emb_ind+1), dim=0).long()
    x += self.pos_embedding[:, pos_emb_ind]
    x = self.dropout(x)

    # 经过注意力层
    x = self.attn_layers(x, **kwargs)
    x = self.norm(x)
    return x

```

ViT-Transformer - Vit实现

```python
def vit_forward(self, img, **kwargs):
    p = self.patch_size
    # 重排输入图像的维度，将其划分为大小为 p x p 的块，并重新排列维度
    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
    x = self.patch_to_embedding(x)
    b, n, _ = x.shape

    # 生成类别令牌，并将其与嵌入的图像块连接起来
    cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
    x = torch.cat((cls_tokens, x), dim=1)

    # 计算位置编码
    h, w = torch.tensor(img.shape[2:])//p
    pos_emb_ind = repeat(torch.arange(h)*(self.max_width//p-w), 'h -> (h w)', w=w)+torch.arange(h*w)
    pos_emb_ind = torch.cat((torch.zeros(1), pos_emb_ind+1), dim=0).long()
    x += self.pos_embedding[:, pos_emb_ind]
    x = self.dropout(x)

    # 经过注意力层
    x = self.attn_layers(x, **kwargs)
    x = self.norm(x)
    return x
```

ViT-Transformer - Vit实现

```python
def vit_forward(self, img, **kwargs):
    p = self.patch_size
    # 重排输入图像的维度，将其划分为大小为 p x p 的块，并重新排列维度
    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
    x = self.patch_to_embedding(x)
    b, n, _ = x.shape

    # 生成类别令牌，并将其与嵌入的图像块连接起来
    cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
    x = torch.cat((cls_tokens, x), dim=1)

    # 计算位置编码
    h, w = torch.tensor(img.shape[2:])//p
    pos_emb_ind = repeat(torch.arange(h)*(self.max_width//p-w), 'h -> (h w)', w=w)+torch.arange(h*w)
    pos_emb_ind = torch.cat((torch.zeros(1), pos_emb_ind+1), dim=0).long()
    x += self.pos_embedding[:, pos_emb_ind]
    x = self.dropout(x)

    # 经过注意力层
    x = self.attn_layers(x, **kwargs)
    x = self.norm(x)
    return x
```

ViT-Transformer - Vit实现

```
def vit_forward(self, img, **kwargs):
    p = self.patch_size
    # 重排输入图像的维度，将其划分为大小为 p x p 的块，并重新排列维度
    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
    x = self.patch_to_embedding(x)
    b, n, _ = x.shape

    # 生成类别令牌，并将其与嵌入的图像块连接起来
    cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
    x = torch.cat((cls_tokens, x), dim=1)

    # 计算位置编码
    h, w = torch.tensor(img.shape[2:])//p
    pos_emb_ind = repeat(torch.arange(h)*(self.max_width//p-w), 'h -> (h w)', w=w)+torch.arange(h*w)
    pos_emb_ind = torch.cat((torch.zeros(1), pos_emb_ind+1), dim=0).long()
    x += self.pos_embedding[:, pos_emb_ind]
    x = self.dropout(x)

    # 经过注意力层
    x = self.attn_layers(x, **kwargs)
    x = self.norm(x)
    return x
```

ViT-Transformer - Vit实现

```python
def vit_forward(self, img, **kwargs):
    p = self.patch_size
    # 重排输入图像的维度，将其划分为大小为 p x p 的块，并重新排列维度
    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
    x = self.patch_to_embedding(x)
    b, n, _ = x.shape

    # 生成类别令牌，并将其与嵌入的图像块连接起来
    cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
    x = torch.cat((cls_tokens, x), dim=1)

    # 计算位置编码
    h, w = torch.tensor(img.shape[2:])//p
    pos_emb_ind = repeat(torch.arange(h)*(self.max_width//p-w), 'h -> (h w)', w=w)+torch.arange(h*w)
    pos_emb_ind = torch.cat((torch.zeros(1), pos_emb_ind+1), dim=0).long()
    x += self.pos_embedding[:, pos_emb_ind]
    x = self.dropout(x)

    # 经过注意力层
    x = self.attn_layers(x, **kwargs)
    x = self.norm(x)
    return x
```

ViT-Transformer - Vit实现

```python
def vit_forward(self, img, **kwargs):
    p = self.patch_size
    # 重排输入图像的维度，将其划分为大小为 p x p 的块，并重新排列维度
    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
    x = self.patch_to_embedding(x)
    b, n, _ = x.shape

    # 生成类别令牌，并将其与嵌入的图像块连接起来
    cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
    x = torch.cat((cls_tokens, x), dim=1)

    # 计算位置编码
    h, w = torch.tensor(img.shape[2:])//p
    pos_emb_ind = repeat(torch.arange(h)*(self.max_width//p-w), 'h -> (h w)', w=w)+torch.arange(h*w)
    pos_emb_ind = torch.cat((torch.zeros(1), pos_emb_ind+1), dim=0).long()
    x += self.pos_embedding[:, pos_emb_ind]
    x = self.dropout(x)

    # 经过注意力层
    x = self.attn_layers(x, **kwargs)
    x = self.norm(x)
    return x

```

纯Latex识别任务

By ViT-Transformer

纯Latex识别模型训练 - 数据预处理

当我们在构造纯Latex数据集的词表时，在词表中发现了大量的中文。

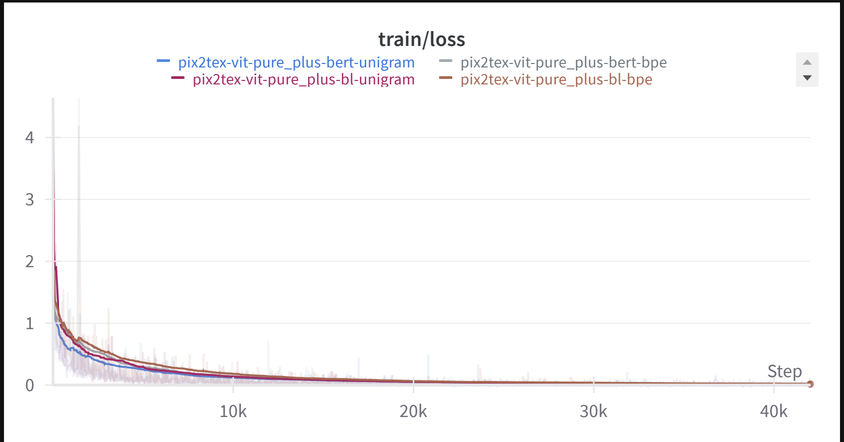

```
多":7956,"服装":7957,"朝上的":7958,"期为":7959,"期的":7960,"期播
":7968,"村庄":7969,"条鱼":7970,"条数为":7971,"条棱长均为":7972,"
某次":7981,"某班":7982,"某项":7983,"某产品":7984,"某女生":7985,"
根式":7994,"格子":7995,"格中的":7996,"棱数为":7997,"楼房":7998,"
"le ft","i n","a n","s q","r t","sq rt","t h","m a","a r","ma
"right arrow","o v","ov er","over rightarrow","g le","an gle",
pha","et a","ft y","in fty","d s","m bol","y mbol","bol ds","b
"ra y","ar ray","l n","p ri","pri me","c i","r c","ci rc","平
分 别","b eg","en d","beg in","方 程","b d","l am","bd a","lam
"v ar","s u","t imes","值 为","的 取","n eq","实 数","的取 值",
"s i","抛 物","两 点","分别 为","交 于","m u","c ap","的最 小","
```


为了获得更好的训练效果，我们对数据集进行了二次处理，通过使用正则表达式`[\u4e00-\u9fa5]+`识别label中是否含有中文来判断该图片是否为纯Latex图片。

纯Latex识别模型训练

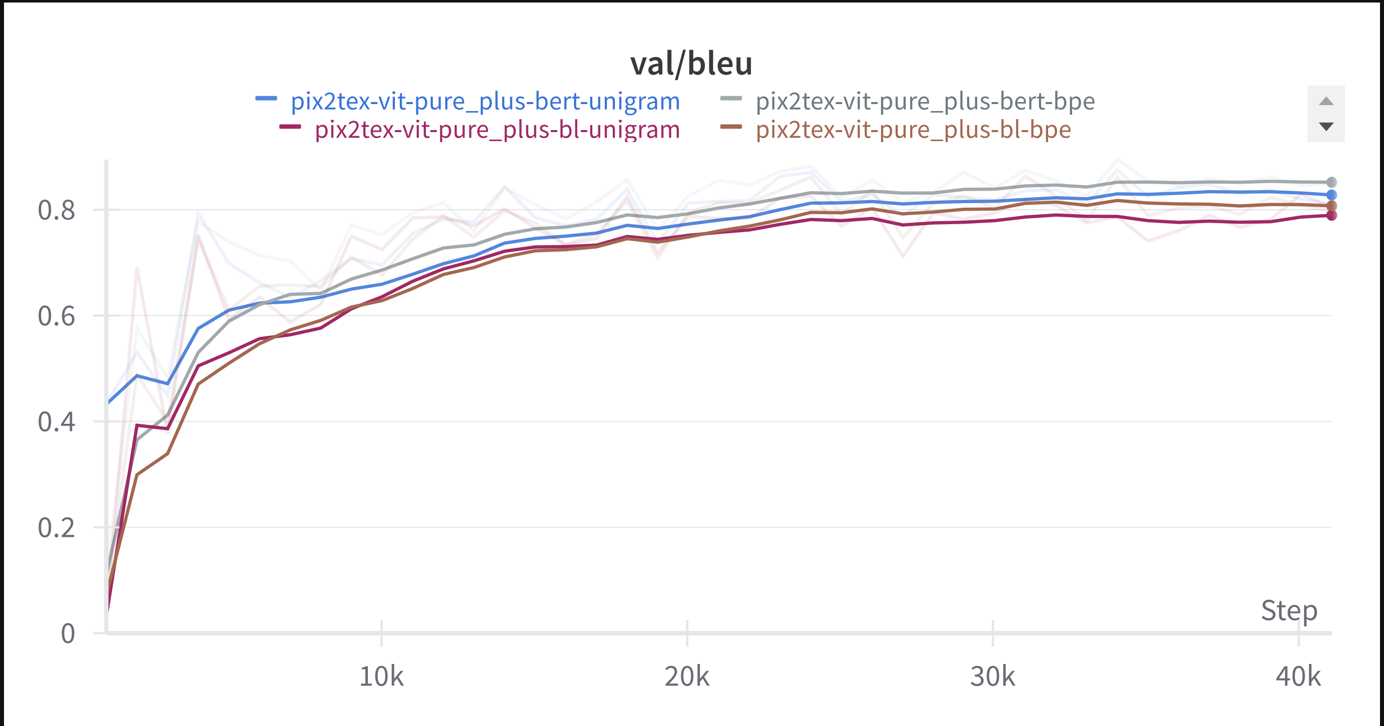
对二次处理过的数据集，使用以下组合进行四次训练

- ByteLevel - Unigram
- BertPreTokenizer - Unigram
- ByteLevel - BPE
- BertPreTokenizer - BPE



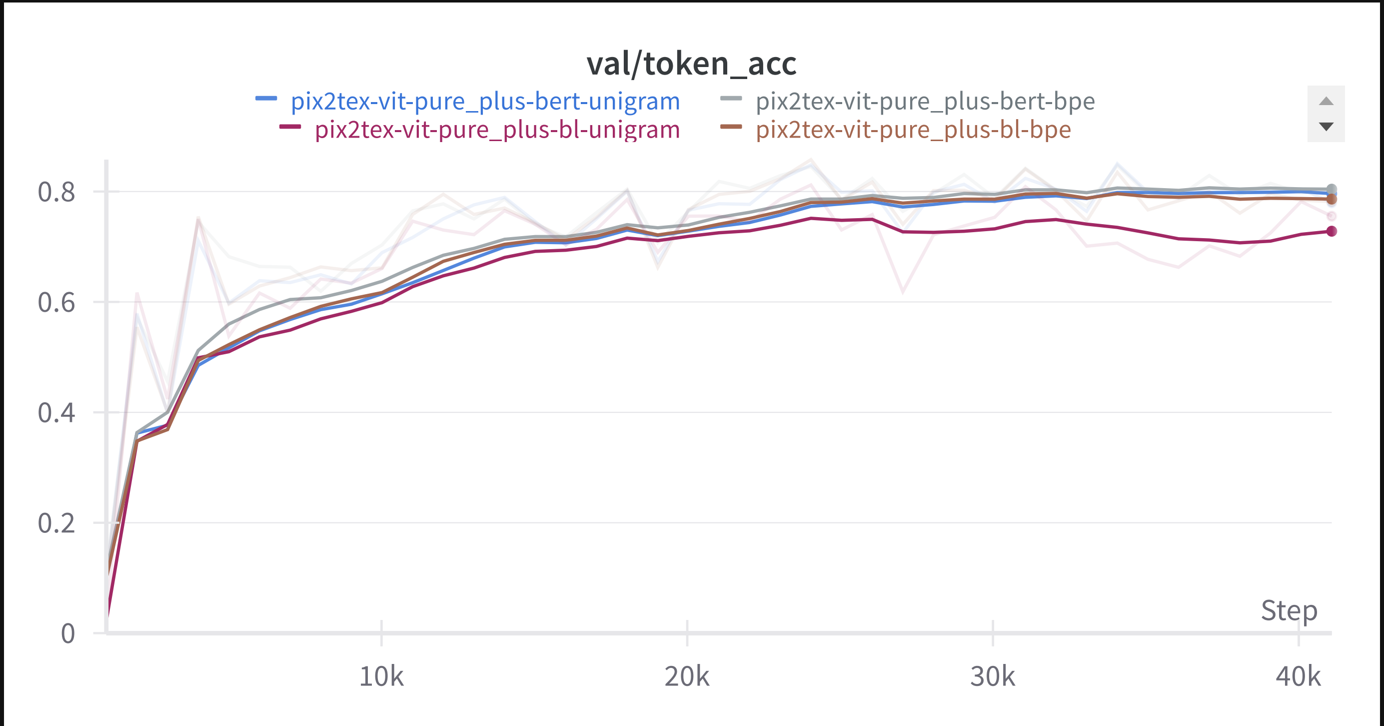
纯latex模型训练曲线

纯Latex识别模型 - `BLEU`



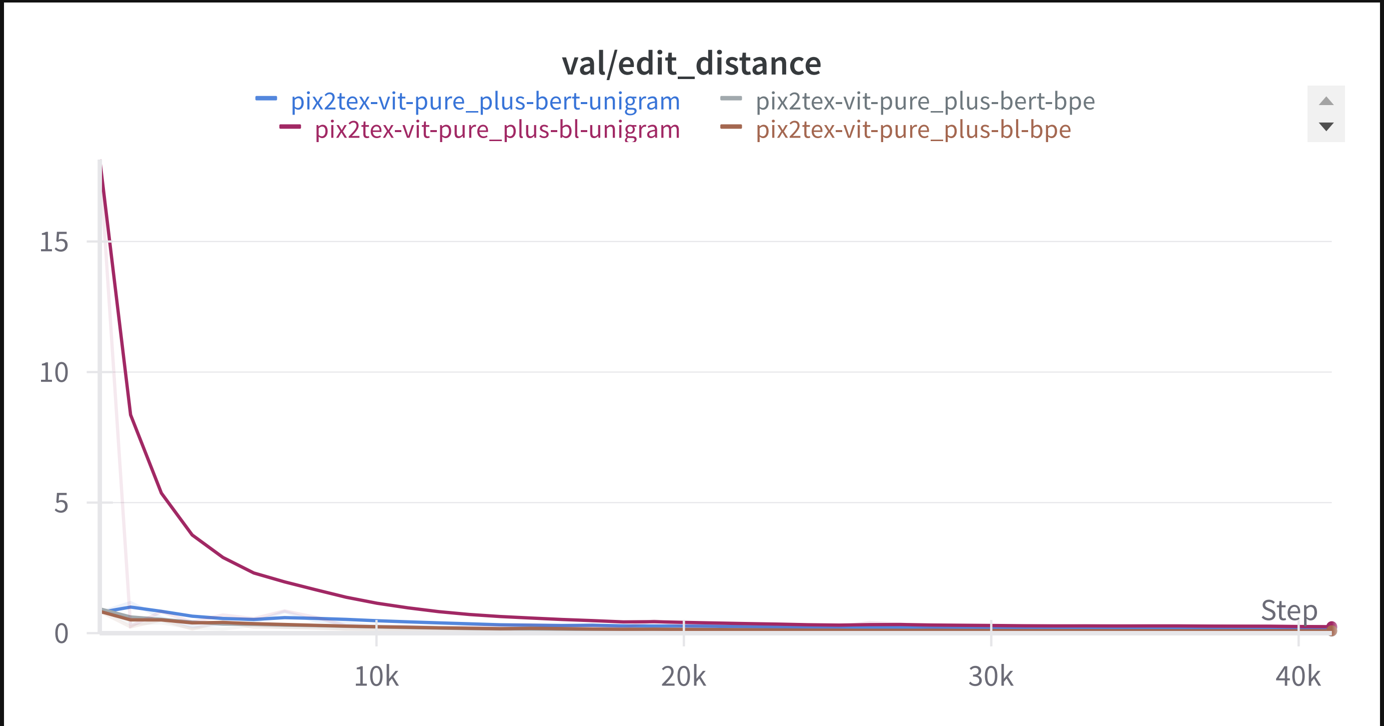
BLEU score

纯Latex识别模型 - `Token Acc`



Token Acc score

纯Latex识别模型 - `Edit Distance`



Edit Distance score

纯Latex识别模型 - 结果

综合上述四种Tokenizer组合的实验结果：

	bert-unigram	bert-bpe	bl-unigram	bl-bpe
BLEU	0.875011	***0.894623***	0.829015	0.863552
Token Acc	0.850422	0.847384	0.8119	***0.857799***
Edit Distance	0.846109	***0.932043***	0.808049	0.927207
Overall	0.857181	***0.89135***	0.816321	0.882853

最终选择`BertPreTokenizer`和`BPE`的搭配供纯Latex识别。

混合文本识别任务

By ViT-Transformer

混合文本任务`Tokenizer`实验

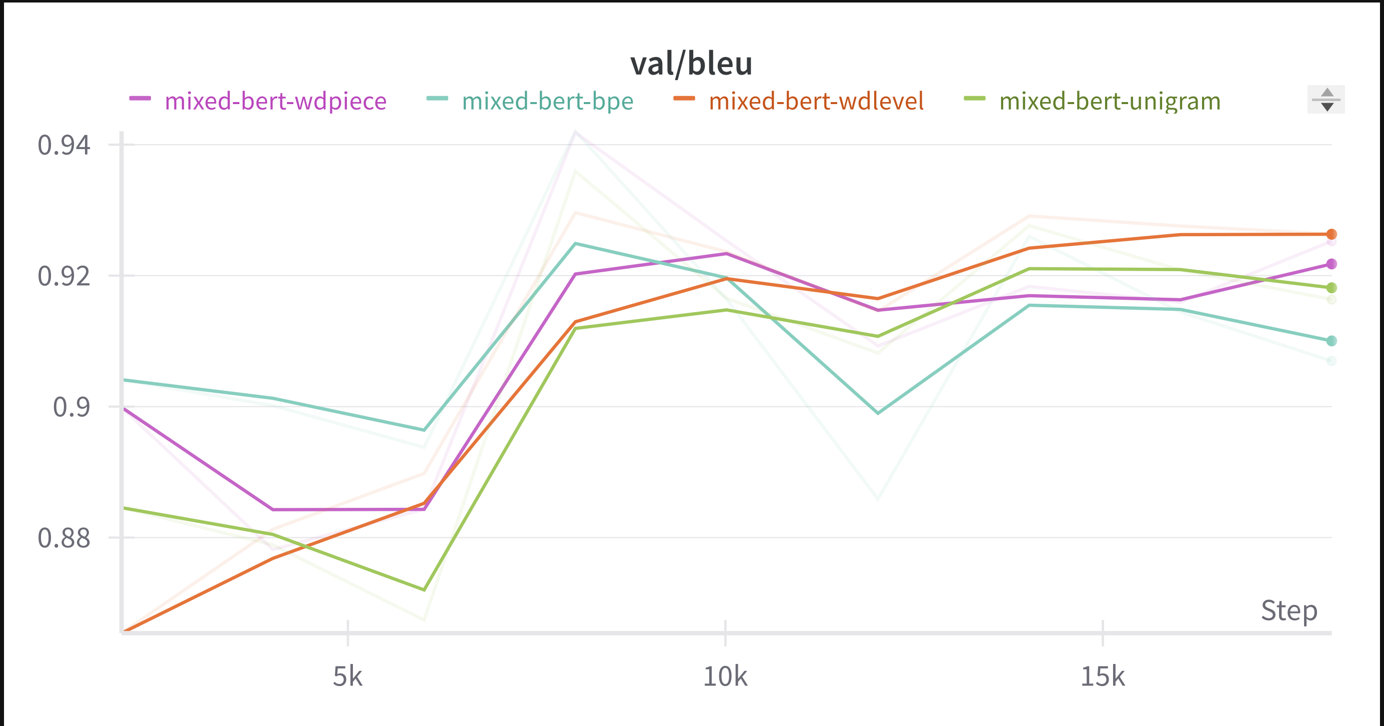
基于预训练的Latex识别模型, 分别使用以下组合进行四次训练

- BertPreTokenizer - WordPiece
- BertPreTokenizer - WordLevel
- BertPreTokenizer - Unigram
- BertPreTokenizer - BPE

混合识别模型tokenizer比较

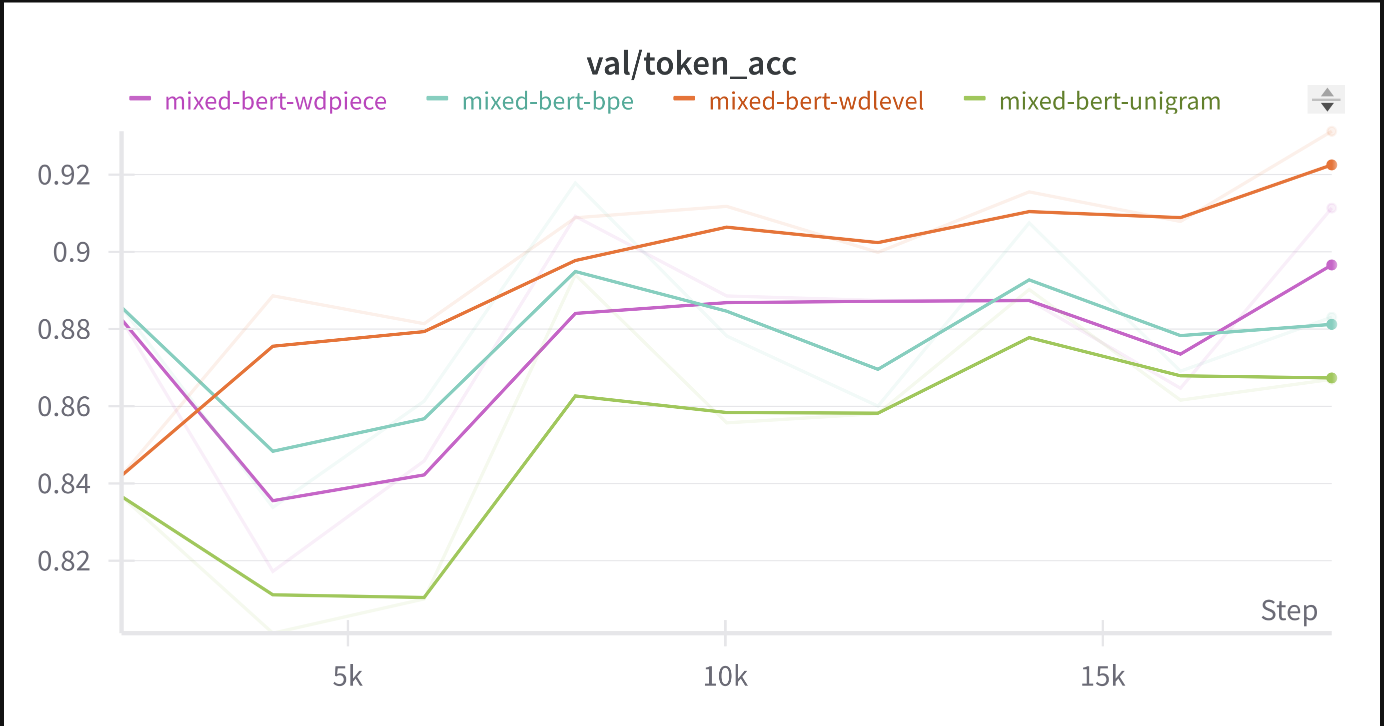
混合识别模型tokenizer比较

混合文本任务`Tokenizer`实验结果 - `BLEU`



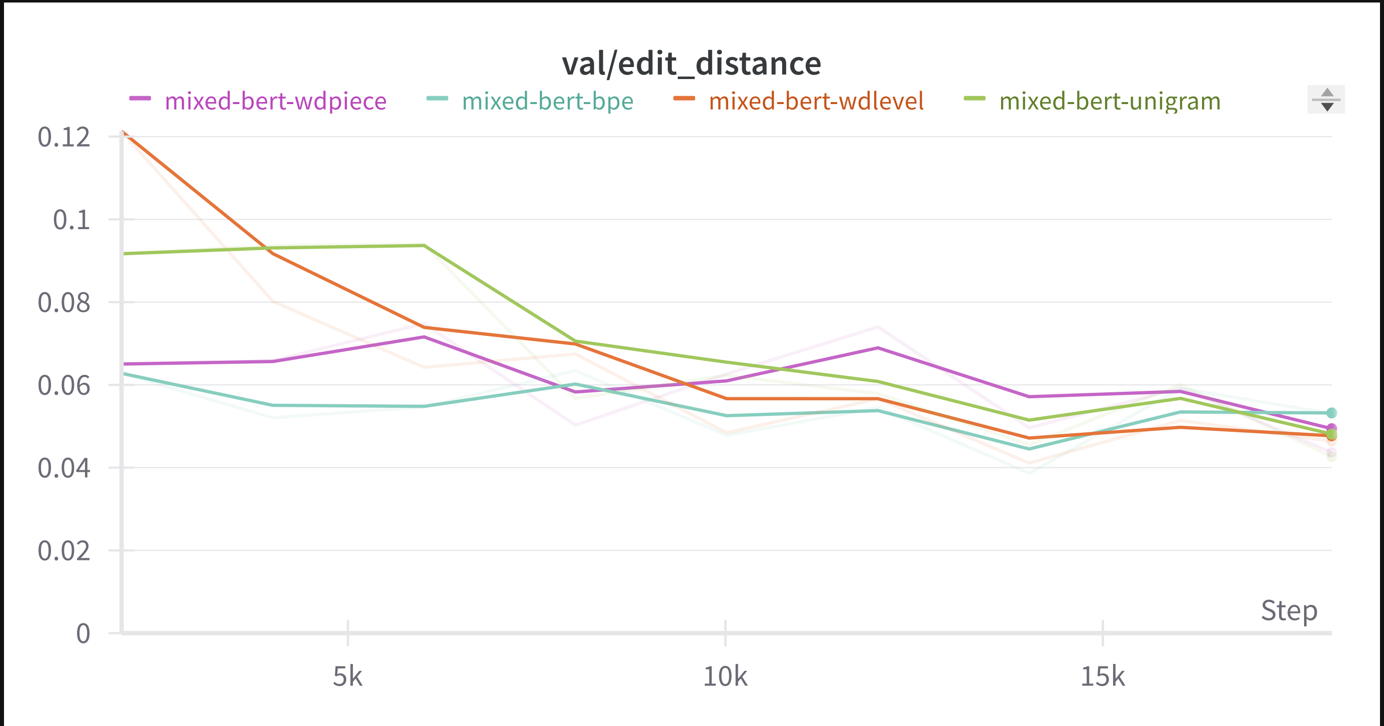
BLEU score

混合文本任务`Tokenizer`实验结果 - `Token Acc`



Token Acc score

混合文本任务`Tokenizer`实验结果 - `Edit Distance`



Edit Distance score

混合文本任务`Tokenizer`实验结果 - 汇总

综合上述四种Tokenizer组合的实验结果：

	bert-wdpiece	*bert-wdlevel*	bert-bpe	bert-unigram
BLEU	0.941852	***0.94205***	0.929618	0.93595
Token Acc	0.911341	***0.917836***	0.931232	0.894052
Edit Distance	0.956329	***0.961388***	0.958903	0.957396
Overall	0.936507	***0.940425***	0.939918	0.929133

> 此时选用的评测方法与最终评测标准不同，仅做定性分析

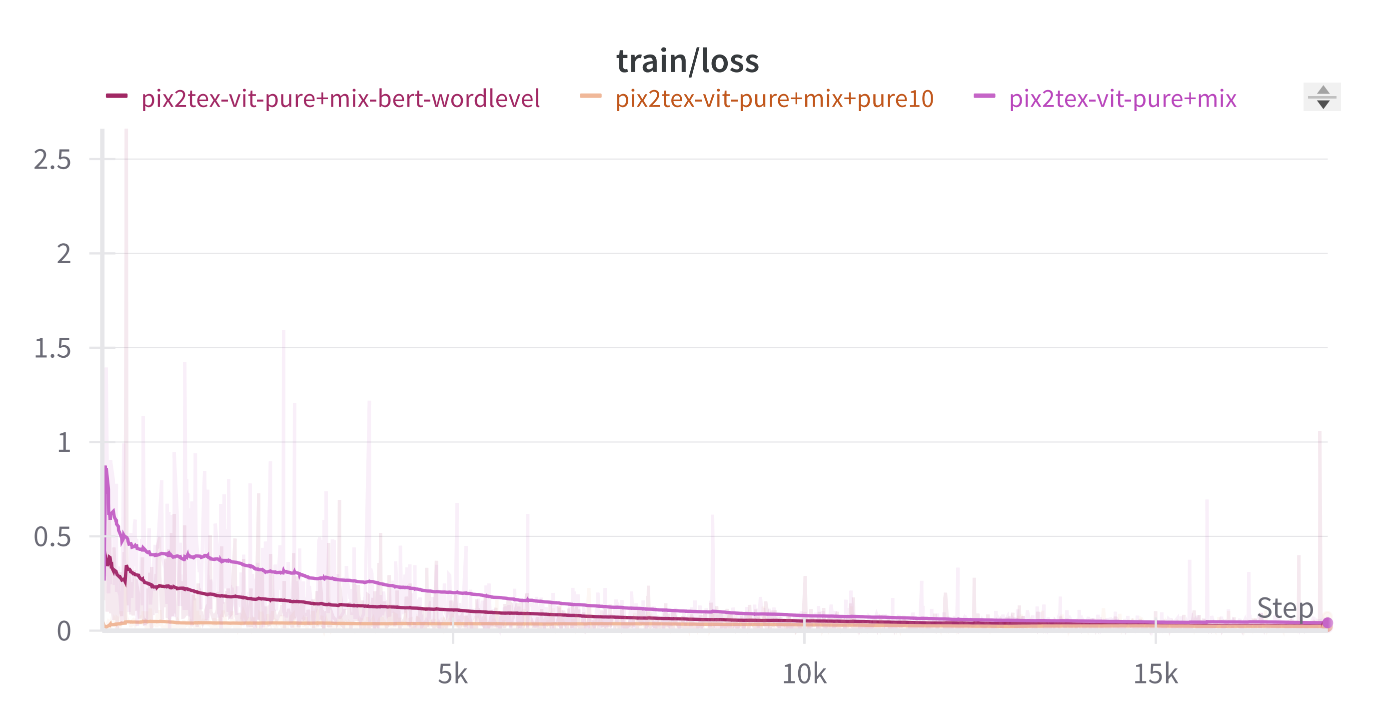
最终选择`BertPreTokenizer`和`WordLevel`的搭配供混合文本识别。

训练混合识别模型

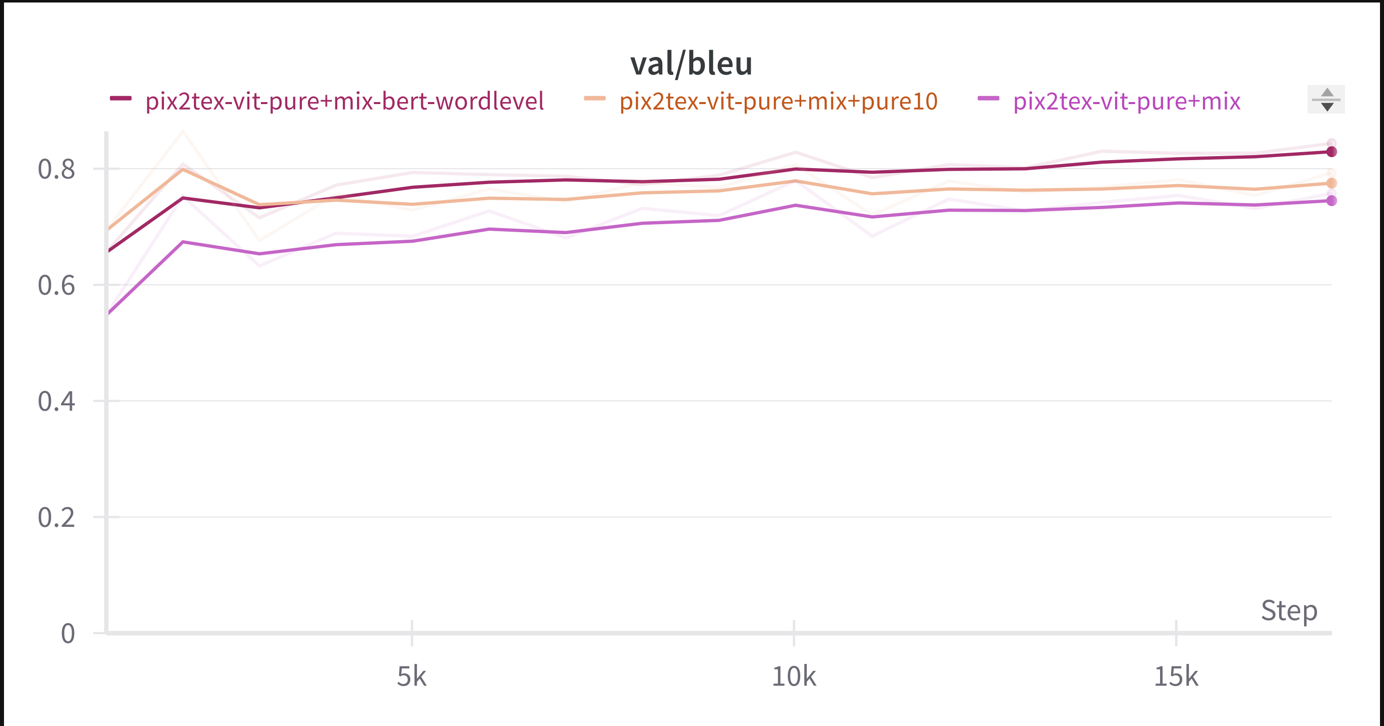
根据上述实验选出的Tokenizer组合，使用之前训练的纯Latex识别模型作为预训练参数

各对混合文本数据集进行20个epoch的训练

训练混合识别模型

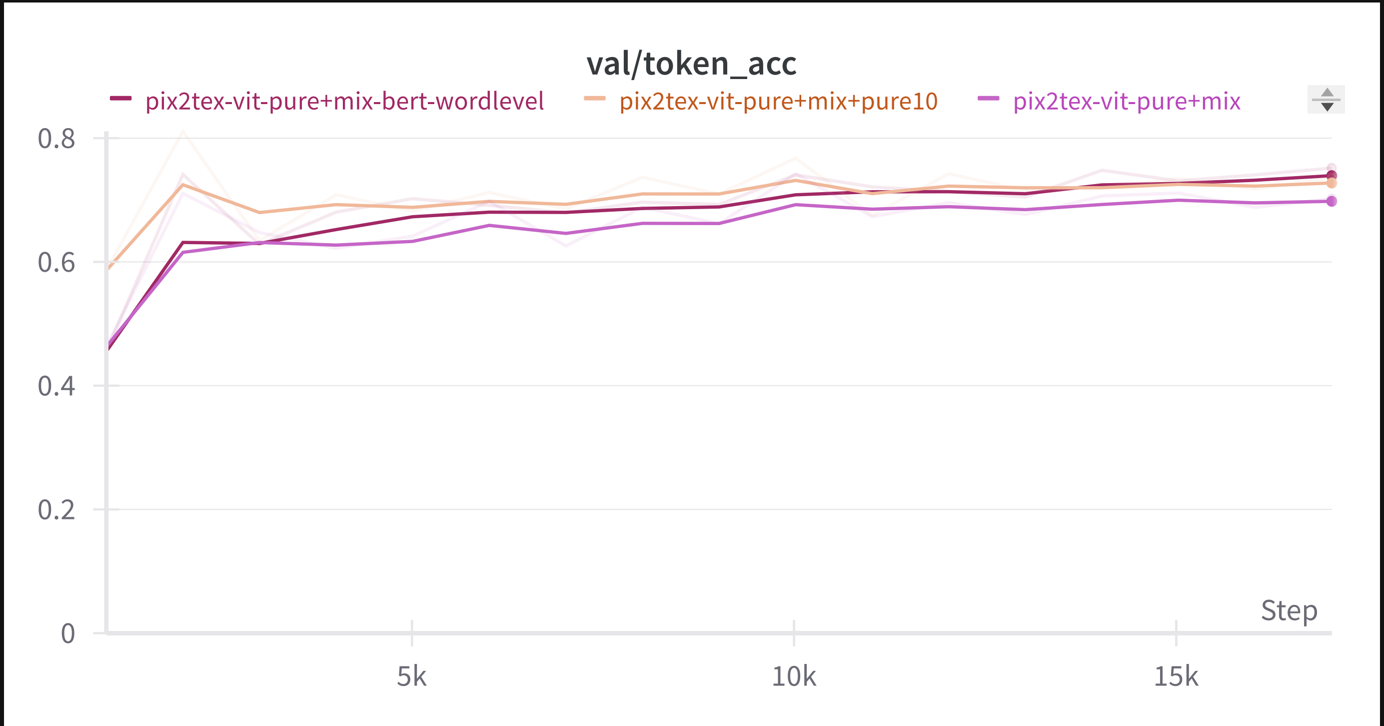

train/loss
pix2tex-vit-pure+mix-bert-wordlevel
pix2tex-vit-pure+mix+pure10
pix2tex-vit-pure+mix
2.5
2
1.5
1
0.5
0
5k
10k
15k
Step

训练混合识别模型 - `BLEU`



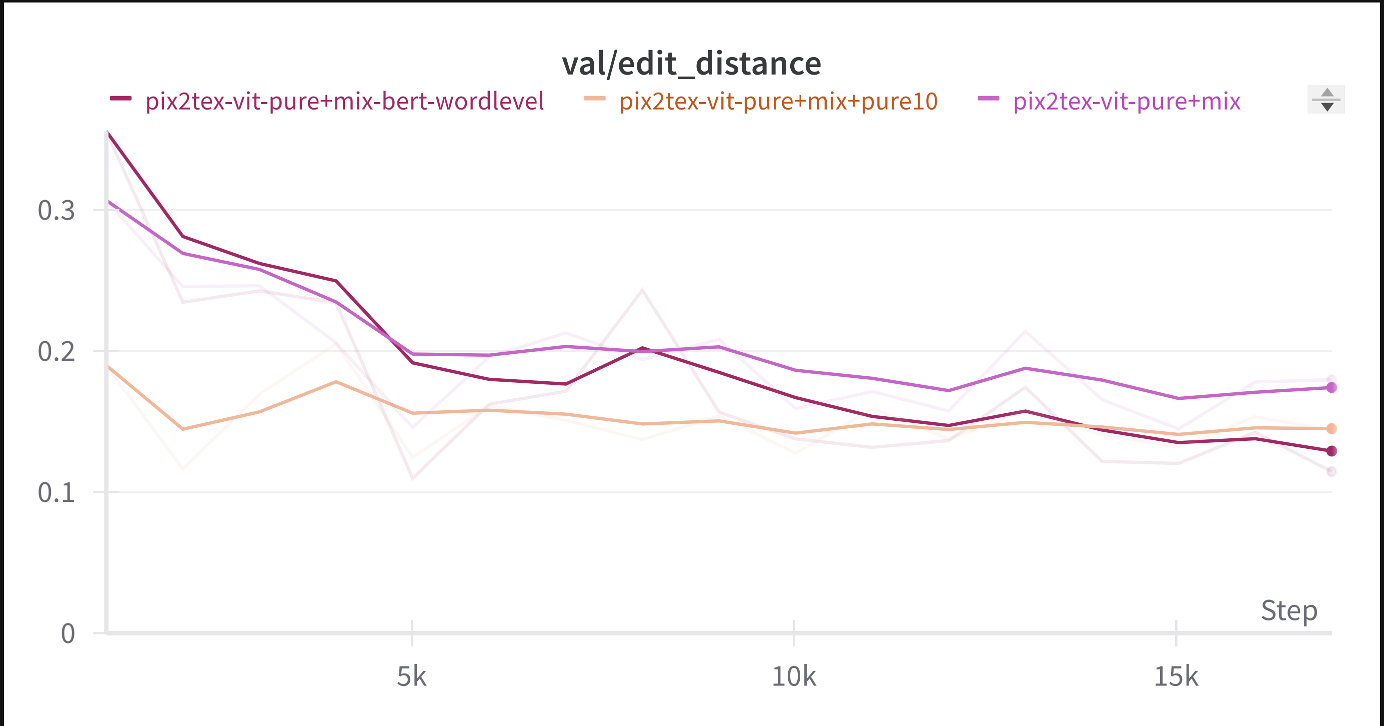
BLEU score

训练混合识别模型 - `Token Acc`



Token Acc score

训练混合识别模型 - `Edit Distance`



Edit Distance score

改进

Yolov8