1. **Conformer模型**

在深度学习中，CNN通过卷积操作分层收集局部特征，并保留局部线索作为特征。Vision Transformer被认为可以通过级联的Self-Attention模块以一种soft的方式在压缩的patch embedding之间聚合全局表示。为了利用局部特征和全局表示，作者设计了一个并发网络的结构Conformer，如下图所示。考虑两种特征的互补性，作者将来自Vision Transformer分支的全局特征送入CNN中，以增强CNN分支的全局感知能力。主要提出了FCU结构将卷积后的特征向量送入Transformer，然后将经过Transformer后的特征送入卷积网络，实现二者的交互融合。

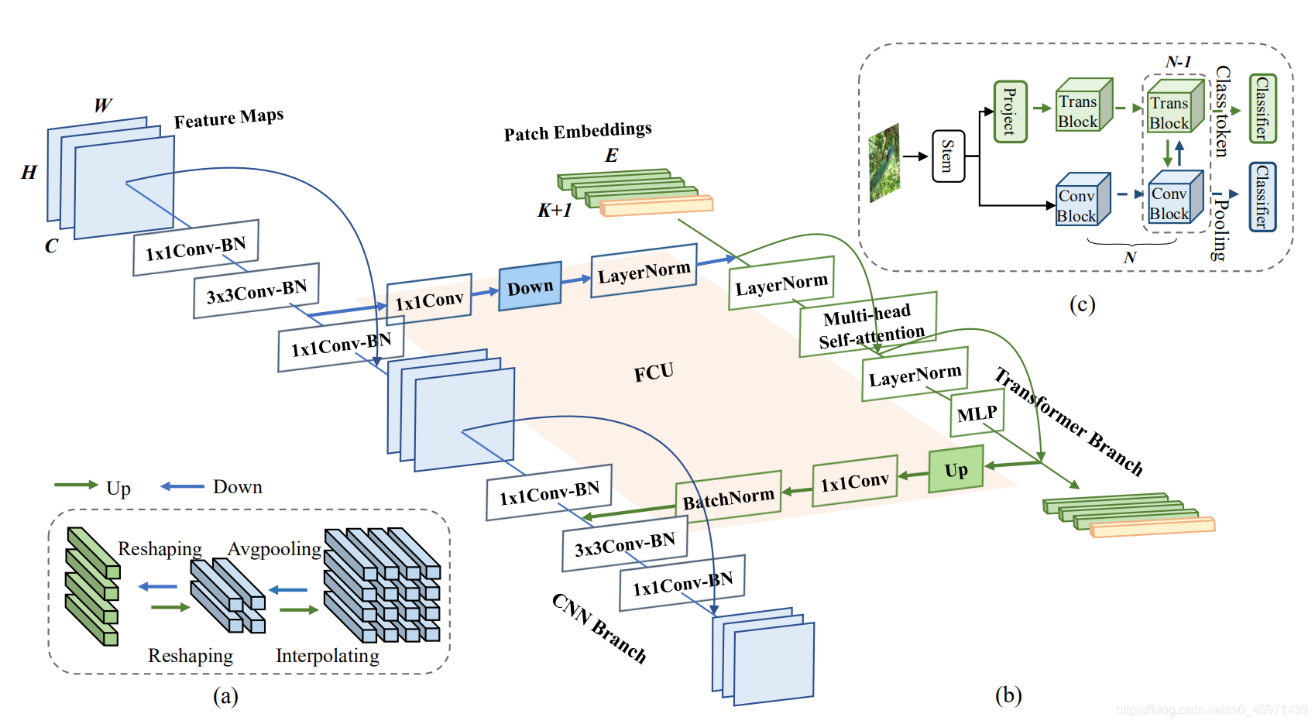


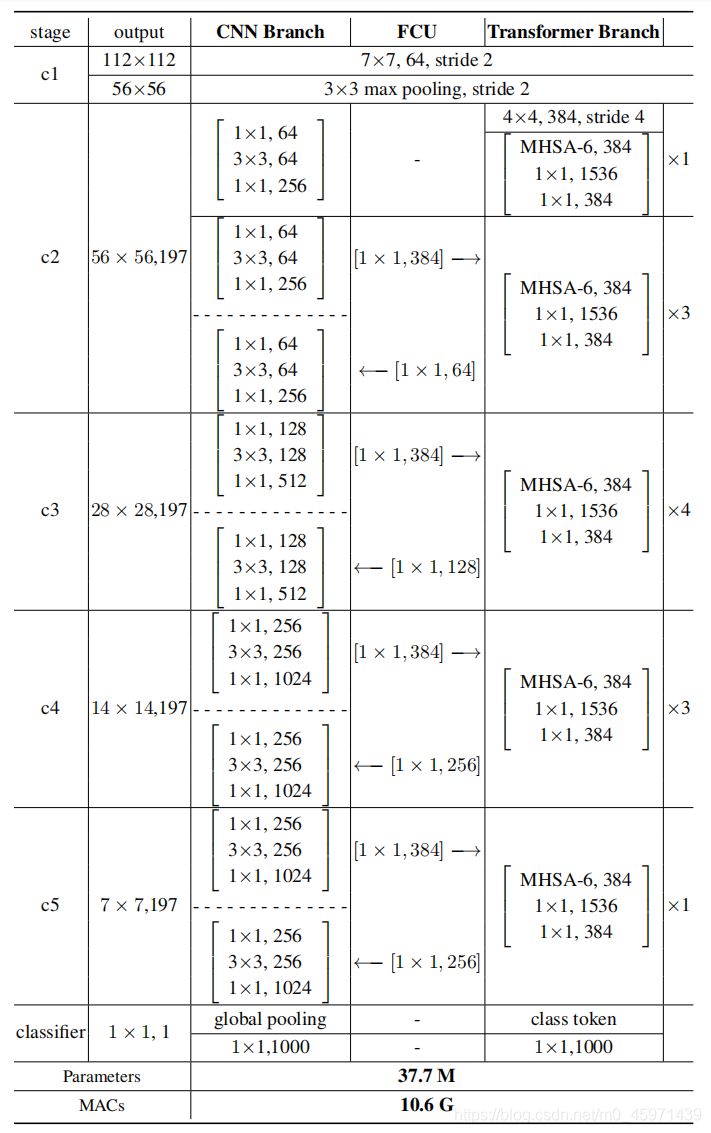
图1 Conformer结构图

网络的整体流程图：a）对于conv feature maps和Transformer中的patch embeddings在空间维度上的转换(H, W)， conv feature map通过下采样得到patch embeddings，反之通过上采样；b）Conformer的具体细节，conv分支中每一个conv block包括:1x1 conv, 3x3conv, 1x1 conv(这里的每一个卷积都有BN+GELU高斯误差的Relu)；Transformer分支中每一个Trans block包括：layer norm -> Multi-head Self-attention -> short cut -> Layer norm -> MLP -> short cut 。

c) 网络整体流程，输入图像经过一个stem block(s=2的7x7conv，卷积核个数为64；3x3，s=2的最大池化)，进行两个并行分支网络：conv分支，Transformer分支，两个分支的最后都会有一个class head，得到两个分支的classifier。在训练时，使用两个CE loss进行BP，两个分支同时训练，并且loss的重要性程度相同；在推理时，两个分类器的预测结果进行加和。

* 1. **CNN Branch**

表1 CNN与Transformer分支结构（Conformer-s）



将整个分支分为4个阶段(c2-c5)，如表1(CNN Branch)所述。每个阶段由多个卷积块组成，每个卷积块包含nc个瓶颈。根据ResNet中的定义，瓶颈包含1×1

下投影卷积，3×3空间卷积，1×1上投影卷积，以及瓶颈的输入和输出之间的残差连接。在实验中，在第一个卷积块中nc设为1，在随后的N−1个卷积块中nc满足≥2。

* 1. **Transformer Branch**

基于ViT模型来设计Transformer分支，包括N个Transformer blocks。如4.1中的图b，每一个Transformer blocks都包含：Multi-head Self-Attention以及MLP block(一个升维的FC层和一个降维的FC层)，其中还使用Layer Norm和short cut。

1. **源代码及注释**

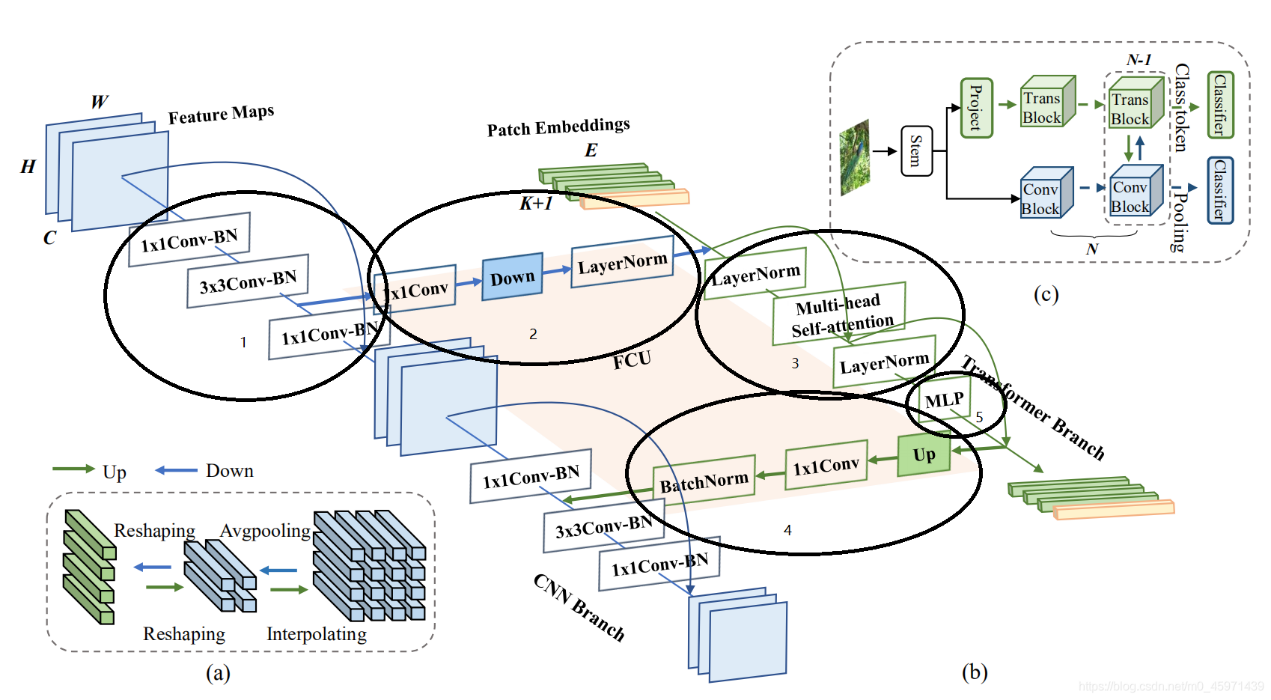
****

图6 代码与总体对应关系

**3.1 ConvBlock（对应图6中标注1）**

**#瓶颈层 包含 1X1， 3X3，1X1卷积，每一层都进行批量归一化**

**class** ConvBlock(nn.Module):  
#采用RELU激活，在BN层将分母不为零的参数设置为10^-6  
 **def** \_\_init\_\_(self, inplanes, outplanes, stride=1, res\_conv=**False**, act\_layer=nn.ReLU, groups=1,  
 norm\_layer=partial(nn.BatchNorm2d, eps=1e-6), drop\_block=**None**, drop\_path=**None**):  
 super(ConvBlock, self).\_\_init\_\_()

#将中间层的输出设置为整个瓶颈层输出的四分之一维度  
 expansion = 4  
 med\_planes = outplanes // expansion

#1X1  
 self.conv1 = nn.Conv2d(inplanes, med\_planes, kernel\_size=1, stride=1, padding=0, bias=**False**)  
 self.bn1 = norm\_layer(med\_planes)  
 self.act1 = act\_layer(inplace=**True**)

#3X3  
 self.conv2 = nn.Conv2d(med\_planes, med\_planes, kernel\_size=3, stride=stride, groups=groups, padding=1, bias=**False**)  
 self.bn2 = norm\_layer(med\_planes)  
 self.act2 = act\_layer(inplace=**True**)

#1X1  
 self.conv3 = nn.Conv2d(med\_planes, outplanes, kernel\_size=1, stride=1, padding=0, bias=**False**)  
 self.bn3 = norm\_layer(outplanes)  
 self.act3 = act\_layer(inplace=**True**)  
#就是只用一个卷积，输入为整个层的输入，输出为整个瓶颈层的输出，最后与瓶颈层输出进行拼接

**if** res\_conv:  
 self.residual\_conv = nn.Conv2d(inplanes, outplanes, kernel\_size=1, stride=stride, padding=0, bias=**False**)  
 self.residual\_bn = norm\_layer(outplanes)  
 self.res\_conv = res\_conv  
 self.drop\_block = drop\_block  
 self.drop\_path = drop\_path  
 **def** zero\_init\_last\_bn(self):  
 nn.init.zeros\_(self.bn3.weight)  
 **def** forward(self, x, x\_t=**None**, return\_x\_2=**True**):  
 residual = x  
 x = self.conv1(x)  
 x = self.bn1(x)  
 **if** self.drop\_block **is not None**:  
 x = self.drop\_block(x)  
 x = self.act1(x)  
 x = self.conv2(x) **if** x\_t **is None else** self.conv2(x + x\_t)  
 x = self.bn2(x)  
 **if** self.drop\_block **is not None**:  
 x = self.drop\_block(x)  
 x2 = self.act2(x)  
 x = self.conv3(x2)  
 x = self.bn3(x)  
 **if** self.drop\_block **is not None**:  
 x = self.drop\_block(x)  
 **if** self.drop\_path **is not None**:  
 x = self.drop\_path(x)  
 **if** self.res\_conv:  
 residual = self.residual\_conv(residual)  
 residual = self.residual\_bn(residual)  
 #直接将输入与经过瓶颈层处理的特征向量进行连接，实现残差

x += residual  
 x = self.act3(x)  
 **if** return\_x\_2:  
 **return** x, x2  
 **else**:  
 **return** x

**3.2 FCUDown（对应图6中标注2）**

#经过卷积的特征向量送入transformer前的处理

**class** FCUDown(nn.Module):  
 *""" CNN feature maps -> Transformer patch embeddings  
 """* **def** \_\_init\_\_(self, inplanes, outplanes, dw\_stride, act\_layer=nn.GELU,  
 norm\_layer=partial(nn.LayerNorm, eps=1e-6)):  
 super(FCUDown, self).\_\_init\_\_()  
 self.dw\_stride = dw\_stride  
 self.conv\_project = nn.Conv2d(inplanes, outplanes, kernel\_size=1, stride=1, padding=0)

#主要通过平均池化来降维达到与patchsize同维度进行拼接  
 self.sample\_pooling = nn.AvgPool2d(kernel\_size=dw\_stride, stride=dw\_stride)  
 self.ln = norm\_layer(outplanes)  
 self.act = act\_layer()

#X代表卷积后的特征向量，X\_t代表enbedding后的向量  
 **def** forward(self, x, x\_t):  
 x = self.conv\_project(x) *# [N, C, H, W]* x = self.sample\_pooling(x).flatten(2).transpose(1, 2)  
 x = self.ln(x)  
 x = self.act(x)  
 x = torch.cat([x\_t[:, 0][:, **None**, :], x], dim=1)  
 **return** x

**3.3 Block（对应图6中标注3和5，类Transformer,内部包含了Attention机制）**

**#自注意力机制，Q,K,V等，与VisionTransformer模型中的Attention代码同**

**class** Attention(nn.Module):  
 **def** \_\_init\_\_(self, dim, num\_heads=8, qkv\_bias=**False**, qk\_scale=**None**, attn\_drop=0., proj\_drop=0.):  
 super().\_\_init\_\_()  
 self.num\_heads = num\_heads  
 head\_dim = dim // num\_heads  
 *# NOTE scale factor was wrong in my original version, can set manually to be compat with prev weights* self.scale = qk\_scale **or** head\_dim \*\* -0.5  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias)  
 self.attn\_drop = nn.Dropout(attn\_drop)  
 self.proj = nn.Linear(dim, dim)  
 self.proj\_drop = nn.Dropout(proj\_drop)  
 **def** forward(self, x):  
 B, N, C = x.shape  
 qkv = self.qkv(x).reshape(B, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4)  
 q, k, v = qkv[0], qkv[1], qkv[2] *# make torchscript happy (cannot use tensor as tuple)* attn = (q @ k.transpose(-2, -1)) \* self.scale  
 attn = attn.softmax(dim=-1)  
 attn = self.attn\_drop(attn)  
 x = (attn @ v).transpose(1, 2).reshape(B, N, C)  
 x = self.proj(x)  
 x = self.proj\_drop(x)  
 **return** x

#线性层转换维度->激活->线性层转换维度

**class** Mlp(nn.Module):  
 **def** \_\_init\_\_(self, in\_features, hidden\_features=**None**, out\_features=**None**, act\_layer=nn.GELU, drop=0.):  
 super().\_\_init\_\_()  
 out\_features = out\_features **or** in\_features  
 hidden\_features = hidden\_features **or** in\_features  
 self.fc1 = nn.Linear(in\_features, hidden\_features)  
 self.act = act\_layer()  
 self.fc2 = nn.Linear(hidden\_features, out\_features)  
 self.drop = nn.Dropout(drop)  
 **def** forward(self, x):  
 x = self.fc1(x)  
 x = self.act(x)  
 x = self.drop(x)  
 x = self.fc2(x)  
 x = self.drop(x)  
 **return** x

#封装成的模块，归一化->多头注意力->归一化->mlp  
**class** Block(nn.Module):  
 **def** \_\_init\_\_(self, dim, num\_heads, mlp\_ratio=4., qkv\_bias=**False**, qk\_scale=**None**, drop=0., attn\_drop=0.,  
 drop\_path=0., act\_layer=nn.GELU, norm\_layer=partial(nn.LayerNorm, eps=1e-6)):  
 super().\_\_init\_\_()  
 self.norm1 = norm\_layer(dim)  
 self.attn = Attention(  
 dim, num\_heads=num\_heads, qkv\_bias=qkv\_bias, qk\_scale=qk\_scale, attn\_drop=attn\_drop, proj\_drop=drop)  
 *# NOTE: drop path for stochastic depth, we shall see if this is better than dropout here*

#没有用到随机裁剪，具体实现不了解，drop\_path=0.时不起作用self.drop\_path = DropPath(drop\_path) **if** drop\_path > 0. **else** nn.Identity()  
 self.norm2 = norm\_layer(dim)  
 mlp\_hidden\_dim = int(dim \* mlp\_ratio)  
 self.mlp = Mlp(in\_features=dim, hidden\_features=mlp\_hidden\_dim, act\_layer=act\_layer, drop=drop)  
 **def** forward(self, x):

#此处drop\_path为空值，相当于直接返回x  
 x = x + self.drop\_path(self.attn(self.norm1(x)))  
 x = x + self.drop\_path(self.mlp(self.norm2(x)))  
 **return** x

**3.4 FCUUp（对应图6中标注4）**

**#升维使经过transformer的输出能够与卷积分支的向量维度匹配**

**class** FCUUp(nn.Module):  
 *""" Transformer patch embeddings -> CNN feature maps  
 """* **def** \_\_init\_\_(self, inplanes, outplanes, up\_stride, act\_layer=nn.ReLU,  
 norm\_layer=partial(nn.BatchNorm2d, eps=1e-6),):  
 super(FCUUp, self).\_\_init\_\_()

#扩大倍数  
 self.up\_stride = up\_stride  
 self.conv\_project = nn.Conv2d(inplanes, outplanes, kernel\_size=1, stride=1, padding=0)  
 self.bn = norm\_layer(outplanes)  
 self.act = act\_layer()  
 **def** forward(self, x, H, W):  
 B, \_, C = x.shape  
 *# [N, 197, 384] -> [N, 196, 384] -> [N, 384, 196] -> [N, 384, 14, 14]* x\_r = x[:, 1:].transpose(1, 2).reshape(B, C, H, W)  
 x\_r = self.act(self.bn(self.conv\_project(x\_r)))  
#填充过渡数值完成转换  
 **return** F.interpolate(x\_r, size=(H \* self.up\_stride, W \* self.up\_stride))

1. **训练曲线（Loss-Epoch）**

训练轮次为300轮，每五轮训练完成后进行一次对验证集（10000张图片）进行测试，得出每五轮的训练损失与测试损失如图2。由表中可以得出在测试近30轮（实际训练150轮后），测试集上的损失略微上升，模型采取了清空权重重新训练防止陷入局部最优的困境，再接下来的训练中，测试集的损失能够到达比之前更低，说明该策略的有效性。

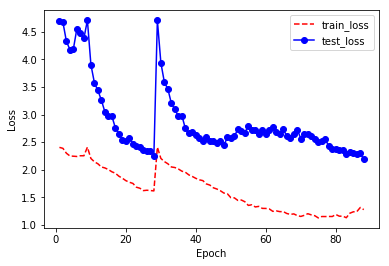


图2 训练损失与测试损失

较在相同数据集上取得良好结果的SAM模型，进行参数调整，将batch-size设置为128，学习率设置为0.1，权重衰减设置为0.0005。训练轮次设置为与之相同的200，以便于比较模型的性能。

过程中多次出现loss is nan而中断训练。尝试改变激活函数以减少梯度消失或爆炸的现象。

1. **运行或运行结果截图**

由于image-net数据集过大，使用作者训练好的模型对image-net数据集进行验证，复现论文中所展示的准确率，83.4%。准确度如图3所示。

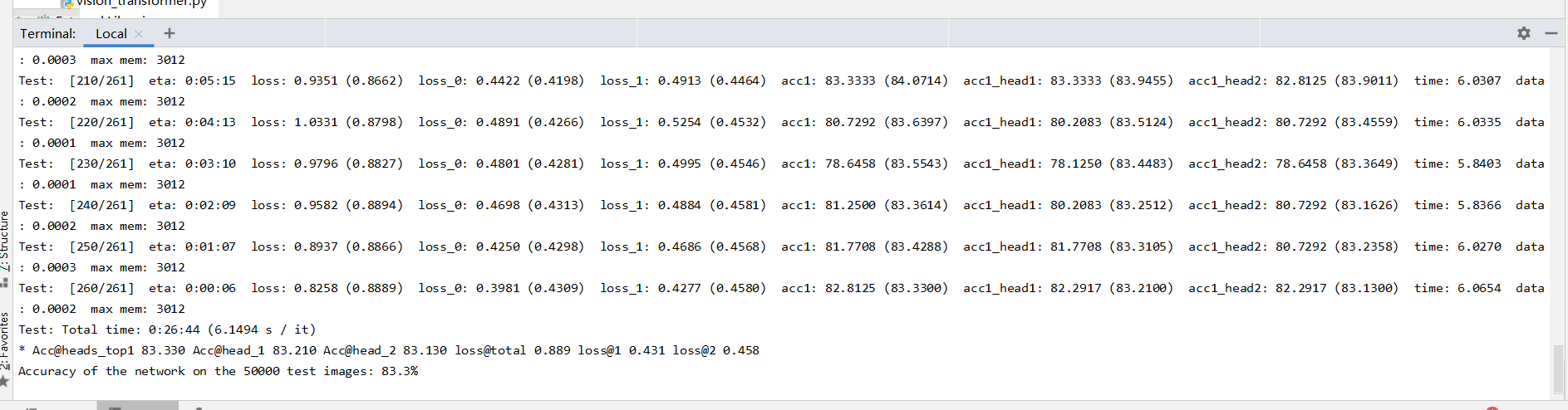


图3 运行结果

使用CIFAR100数据集进行训练和预测，最大准确率如图4所示。运行时长如图5所示。

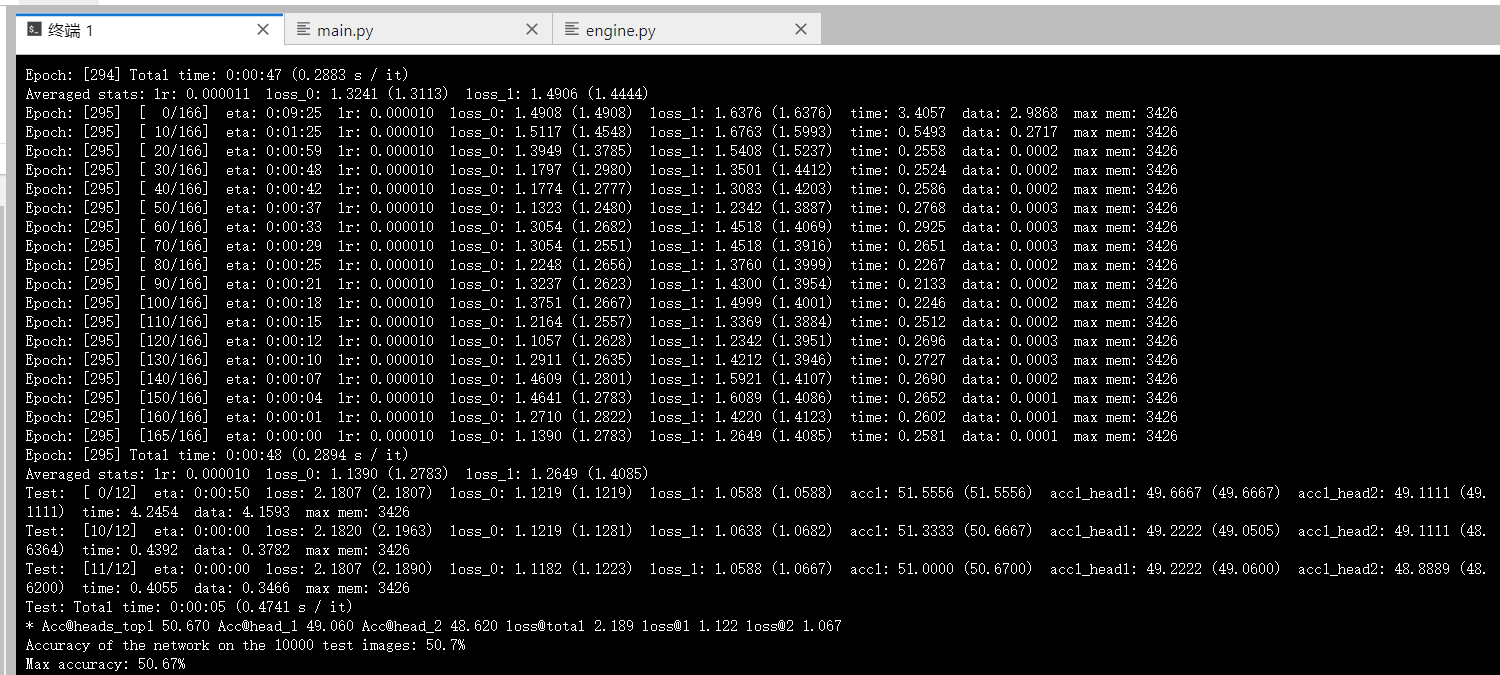


图4 准确率

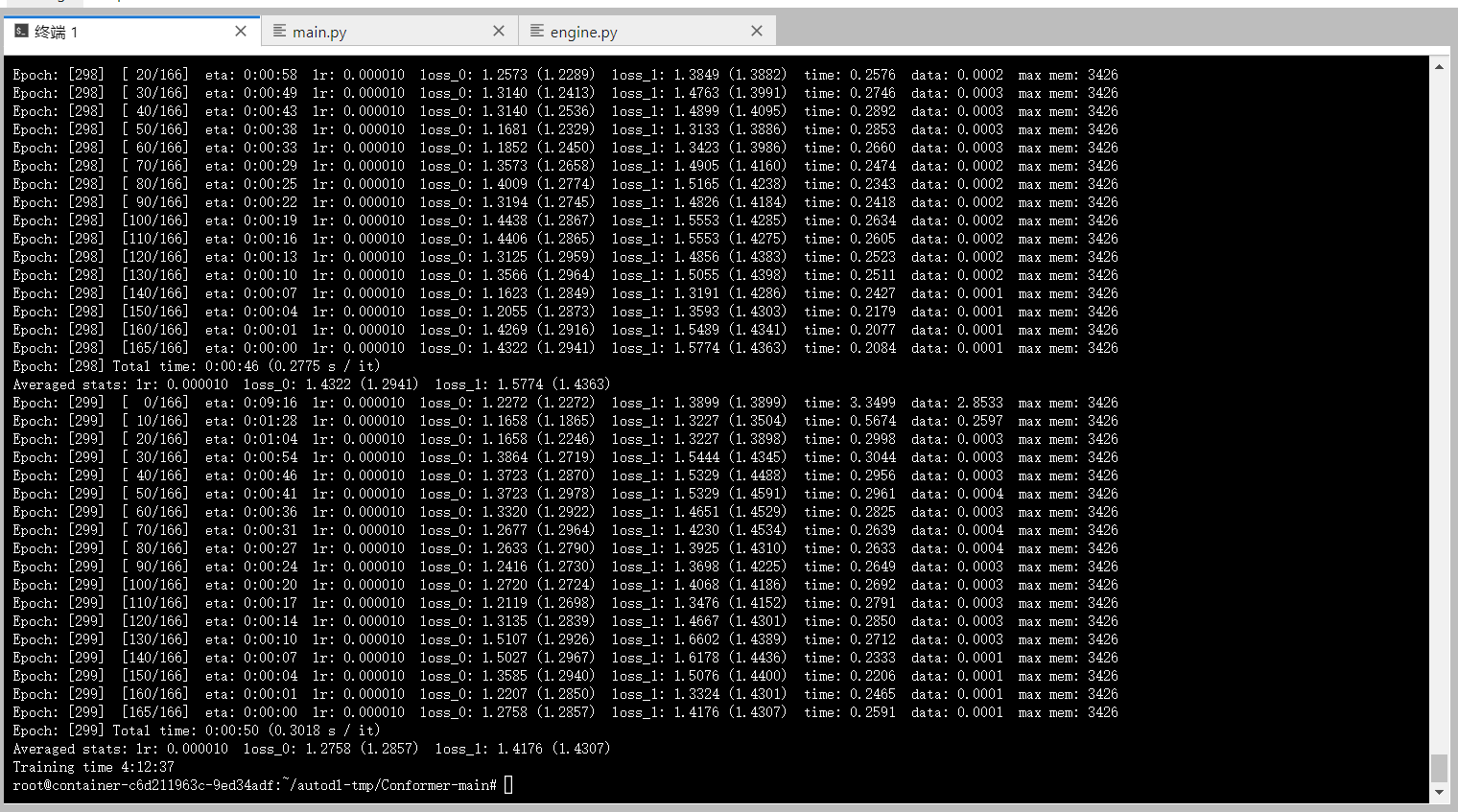


图5 训练时长

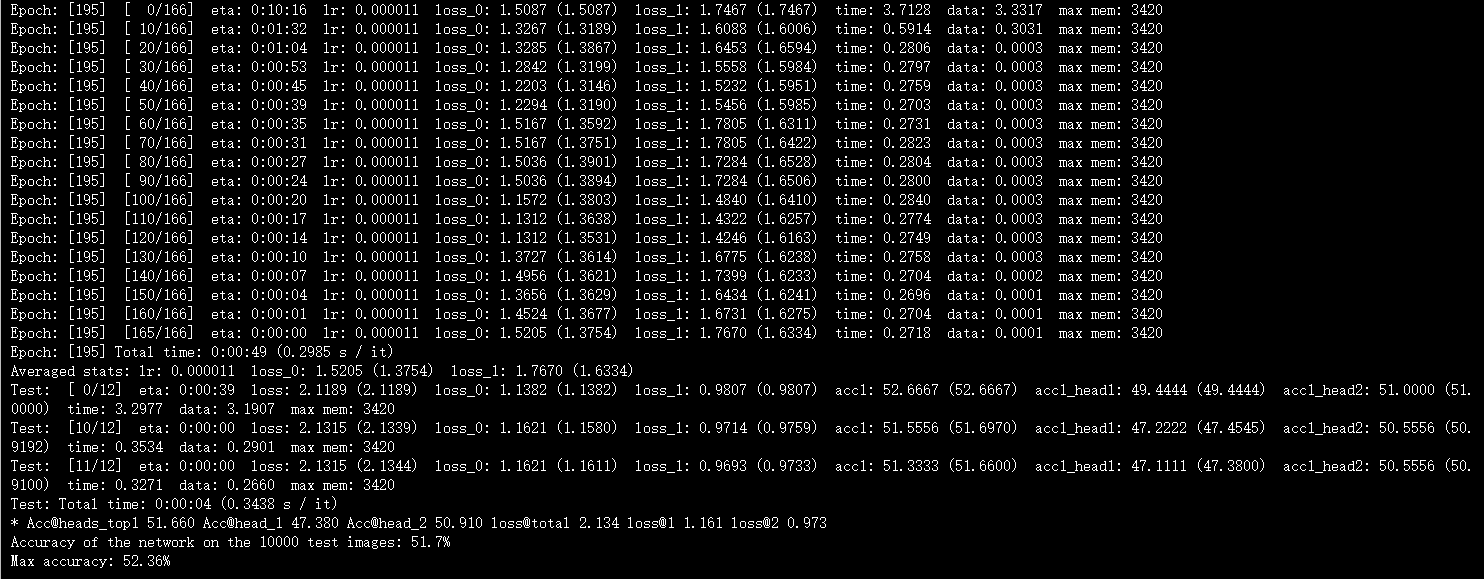


图6 准确率提升

通过调整学习率和drop-out的数值，使得准确率提升2个百分点，其他调优的方式，例如缩减训练模型的大小，采用不同batch-size等等。最为关键的应该是数据集本身，如果类别间存在某种特殊的联系，将更能够发挥出模型中的自注意力机制。Cifar100中识别类别间没有特殊关系都是独立的物体。如果将汽车，公路，红绿灯，指示牌等等相关物体作为类别和标签进行训练，相信该模型会取得更加良好的效果。