ViViT: A Video Vision Transformer

```
paper:  https://arxiv.org/abs/2103.15691
accept: ICCV2021
author: Google Research
code(Unoffical):
    model2: https://github.com/rishikksh20/ViViT-pytorch
    model4: https://github.com/noureldien/vivit_pytorch
code(Offical): https://github.com/google-research/scenic/tree/main/scenic/projects/vivit
```

一、前言

Google的这篇paper和FBAI的很类似，都是给出几个VideoTransformer的范式，解决怎么把Transformer结构从Image迁移到Video上。相比于TimeSformer，google的这篇paper不单单给出结构范式，同时也给出了迁移imagenet pretrain的实验分析，并给出了怎么训练小数据集上的策略。实验表明，相比于TimeSformer，ViViT更SOTA。(ps:这两篇paper第一版本相差了1个半月的时间，很有可能是同期工作，也有可能Google填了一点FB的坑)

二、出发点

基本上使用Transformer作为结构都有一个共性，那就是 `self-attention` 的特性：

- 长距离依赖性(long-range dependencies)
- 序列中的每个元素都会与整体进行计算

相比于CNN结构，Transformer具备更少的归纳偏置，所以需要更多的数据或者更强的正则来训练。

对于Video数据，要满足时序和空间特征的计算，所以要改进Transformer结构。

三、算法设计

为了让模型表现力更强，ViViT讨论了两方面的设计和思考(TimeSformer只考虑了模型结构设计)：

- Embedding video clips
- Transformer Models for Video

Overview of Vision Transformers (ViT)

先看一下ViT(Image)的公式定义：
输入为$x_i \in \mathbb{R}^{h \times w}$, 经过线性映射后，变换为一维tokens，$z_i \in \mathbb{R}^d$。输入的编码表达如下:

$$\mathrm{z} = [z_{cls}, \mathrm{E}x_1, \mathrm{E}x_2, ..., \mathrm{E}x_N] + \mathrm{p}$$

这里的E是2d卷积。如下图最左边结构所示，添加一个可学习的token,z_{cls},用于表达最后分类之前的特征。这里，$\mathrm{p} \in \mathbb{R}^{N \times d}$表示为位置编码，加上输入token，用来保留位置信息。接下来，tokens经过L层的Transformer编码层。每层l都包含**Mutil-Headed Self-Attention(MSA)和LayerNorm(LN)** 以及一个**MLP**结构，表示如下:

$$y^l = \mathrm{MSA}(\mathrm{LN}(\mathrm{z}^l)) + \mathrm{z}^l$$
$$\mathrm{z}^{l+1} = \mathrm{MLP}(\mathrm{LN}(\mathrm{y}^l)) + y^l$$

MLP由两个线性层以及GELU构成，整个推理过程中，token的维度d保持不变。最后，用一个线性层来对encoded的$z^l_{cls} \in \mathbb{R}^d$完成最终的分类，当然也可以用除了cls-token以外的所有tokens的的全局池化来进行分类。

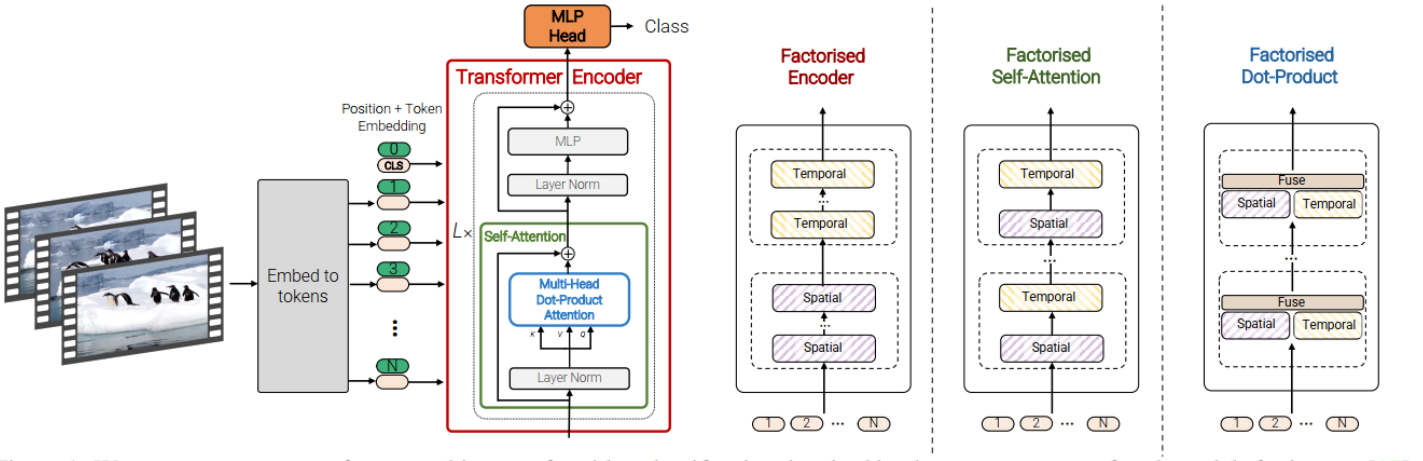


Figure 1: We propose a pure-transformer architecture for video classification, inspired by the recent success of such models for images [17]. To effectively process a large number of spatio-temporal tokens, we develop several model variants which factorise different components of the transformer encoder over the spatial- and temporal-dimensions. As shown on the right, these factorisations correspond to different attention patterns over space and time.

图1. 整体结构

Embedding video clips

文中考虑了**两种**简单的方法来构建从视频$\mathrm{V} \in \mathbb{R}^{T \times H \times W \times C}$到token序列$\tilde{\mathrm{z}} \in \mathbb{R}^{n_t \times n_h \times n_w \times d}$的映射。还记得在TimeSformer里的token序列吗，实际上是先把T与batchsize进行合并，然后用2d卷积进行 `TokenEmbeeding` ，然后通过交换T维度以及额外的 `TimeEmbeeding` 来实现空间和时序的信息特征交互。这里稍微不同在于，本文的一种新的采样方法是直接在 `Embeeding` 阶段就把时序的token引入进来了，看下面提出的两种方法。

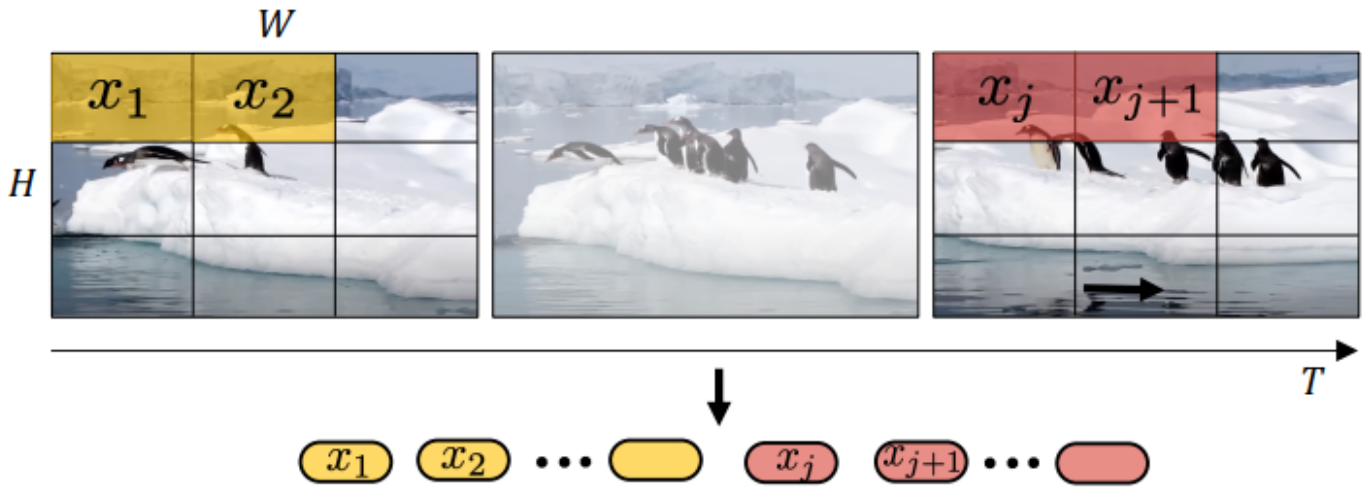


Figure 2: Uniform frame sampling: We simply sample n_t frames, and embed each 2D frame independently following ViT [17].

图2. 均匀帧采样

- **Uniform frame sampling**
  如图2所示，最直接的方法就是均匀采样n_t帧，每帧独立计算 `token-embeeding` ,并把这些token直接concat起来，公式可以表达为:

$$\text{concatenate}(token_i), i \in [1, T]$$

  ViT的token获取是没有重叠的，所以可以直接想象成，我们先把T帧的图像拼接起来成一个大的图像，然后用2d卷积得到token，等价于上述表达。所以可获取token序列为$(n_t \cdot n_h \cdot n_w) \times d$。

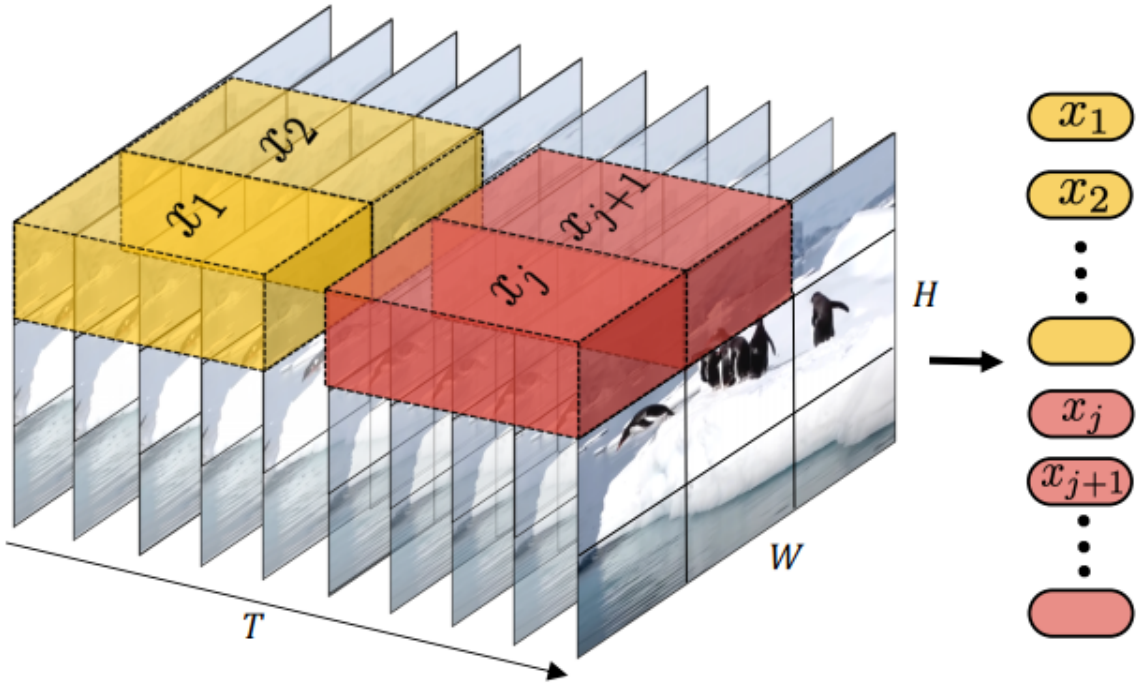


Figure 3: Tubelet embedding. We extract and linearly embed non-overlapping tubelets that span the spatio-temporal input volume.

图3. 管道编码

- **Tubelet embedding**
  图3给出了两一种编码方法，同时获取时序和空间的token，实际可以用3D卷积来实现。对于维度为$t \times h \times w$的tubelet来说，有$n_t = \lfloor \frac{T}{t} \rfloor, n_h = \lfloor \frac{H}{h} \rfloor, n_w = \lfloor \frac{W}{w} \rfloor$。相比于第一种方法需要在encoder阶段融合时序和空间信息，这个方法在生成token阶段就进行了融合，直观上看，没有"割裂"感。

Transformer Models for Video

文中给出了三种模型变体的范式，如图1右边所示，下面给出详细介绍

- **Model 1: Spatio-temporal attention**
  这个其实没有更改模型结构，和第一篇讲的TimeSformer的 `Joint Space-Time` 基本一致，合并token $(b, (t \times h \times w), d)$直接送入ViT，由于 `self-attention` 性质，从第一层开始，时序和空间信息就进行了交互直到最后的分类。这个方法比较暴力也很简单，但是对于 `self-attention` 的计算量会从$O(n^2)$增加到$O((n * t)^2)$，采样帧越多，计算量越大。所以简单的方案不是最优的方案，还需要考虑其他的结构改进。
- **Model 2: Factorised encoder**

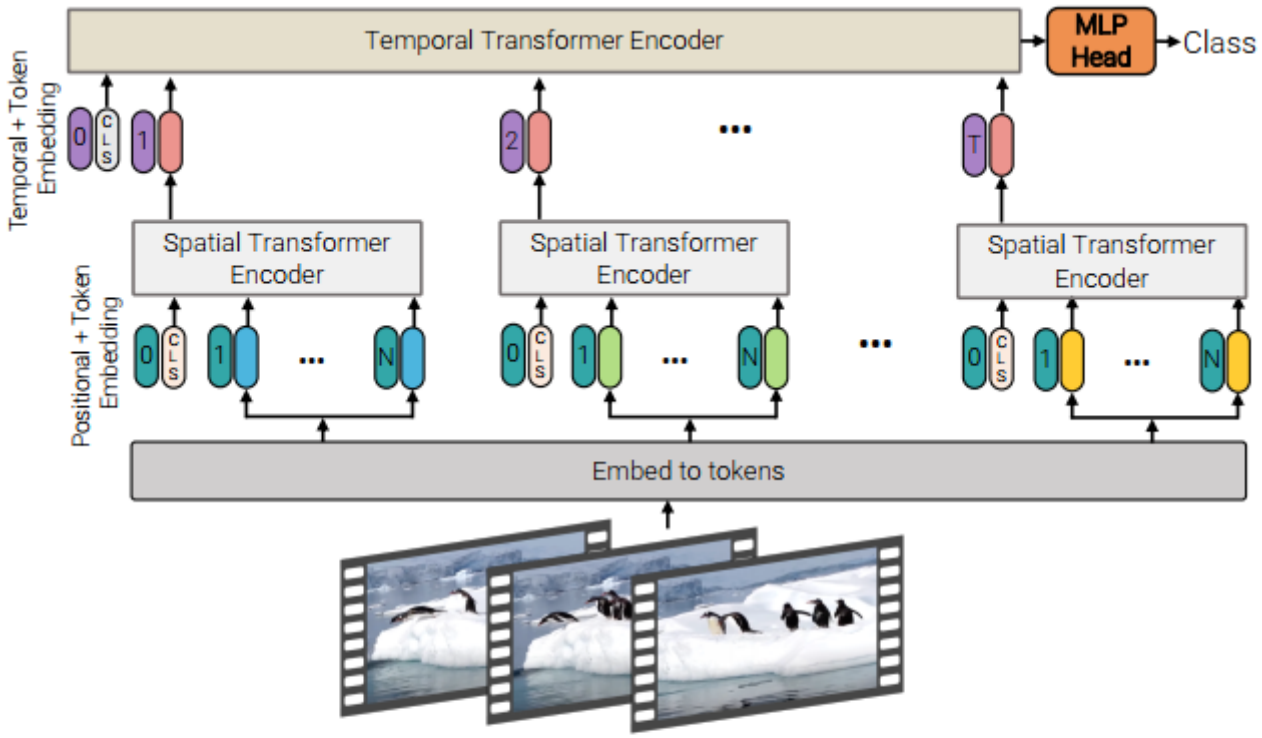


Figure 4: Factorised encoder (Model 2). This model consists of two transformer encoders in series: the first models interactions between tokens extracted from the same temporal index to produce a latent representation per time-index. The second transformer models interactions between time steps. It thus corresponds to a "late fusion" of spatial- and temporal information.

图4. 分离encoder

如上图4所示，这个模型结构包含了两个分离的transformer编码结构。首先是spatial encoder，只计算同一帧下面的spatial-token，经过L_s层后，可以得到每帧的表达,$h_i \in \mathbb{R}^d$。由于spatial-token是有cls-token的，所以这里空间特征表达用$z_{cls}^{L_s}$来表示。把每帧的特征concat起来，$\mathbf{H} \in \mathbb{R}^{n_t \times d}$，输入到$L_t$层的Transformer encoder，用于建模不同时序之间的特征交互。最后的cls-token用于分类，完成整个模型设计。计算的复杂度从$O((n_h \cdot n_w \cdot n_t)^2)$降低到$O((n_h \cdot n_w)^2 + n_t^2)$

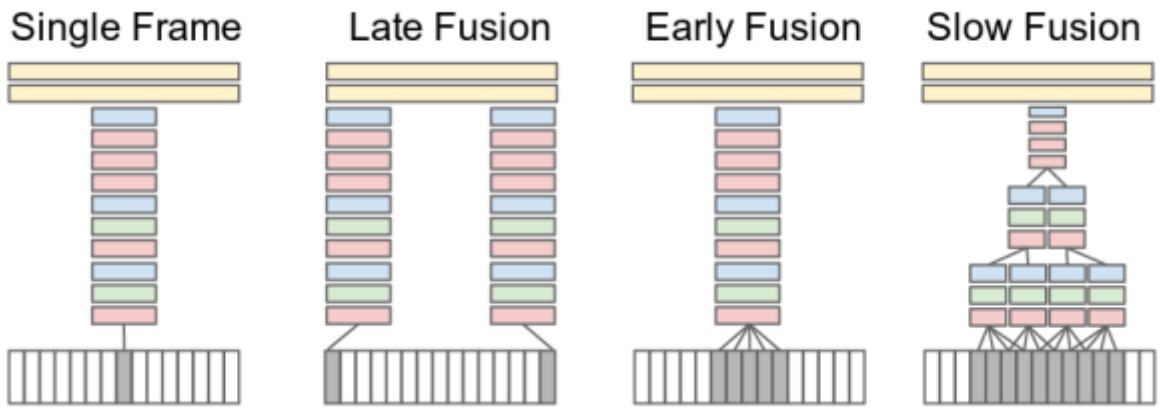


图5. Cnn-base video fusion

这个模型的设计思路与cnn的"late-fusion"很相似，前面用于独立提取特征，后面用于信息交互打分，如上图所示，这个思想也是很多CNN-base的video方法，例如TSN等。

- **Model 3: Factorised self-attention**

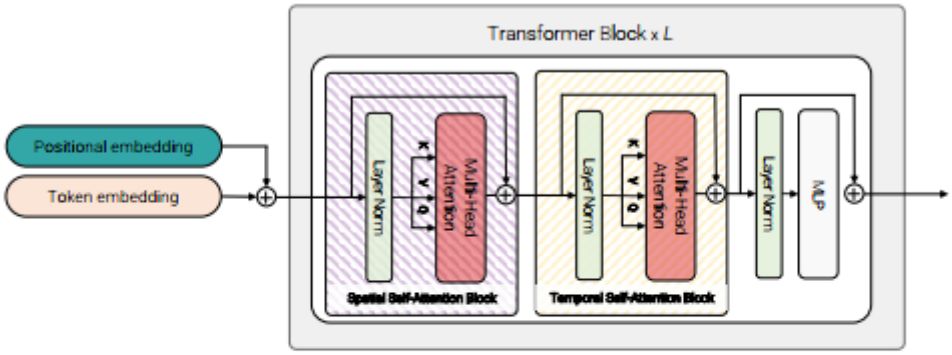


Figure 5: Factorised self-attention (Model 3). Within each transformer block, the multi-headed self-attention operation is factorised into two operations (indicated by striped boxes) that first only compute self-attention spatially, and then temporally.

图6. 分离self-attention

如图6所示，这个结构和TimeSformer设计的 `Divided Space-Time` 基本一样的，在一个transformer blocl里面，先进行spatial-attention再做temporal-attention，相比于Model1有效性更高，同时和Model2一样的计算复杂度。可以通过重排token的shape来实现计算空间attention，$\mathbb{R}^{1\times n_t\cdot n_h\cdot n_w\times d}$到$\mathbb{R}^{n_t\times n_h\cdot n_w\times d}$。计算时序attention的时候，再进行重排，$\mathbb{R}^{n_h\cdot n_w\times n_t\times d}$,这里的batchsize默认为1。公式可以表达为:

$$
\begin{aligned}
\mathrm{y}_s^l &= \mathrm{MSA}(\mathrm{LN}(\mathrm{z}_s^l)) + \mathrm{z}_s^l \\
\mathrm{y}_t^l &= \mathrm{MSA}(\mathrm{LN}(\mathrm{y}_s^l)) + \mathrm{y}_s^l \\
\mathrm{z}^{l+1} &= \mathrm{MLP}(\mathrm{LN}(\mathrm{y}_t^l)) + \mathrm{y}_t^l
\end{aligned}
$$

这里有个很有意思的点，TimeSformer的结论是T-S的顺序会有提升，S-T的顺序会有下降，但是

ViViT的结论是T-S和S-T指标没区别。TimeSformer在实现的时候考虑了cls-token的信息变化，ViViT直接弃用了，以免信息混淆。

- **Model 4: Factorised dot-product attention**

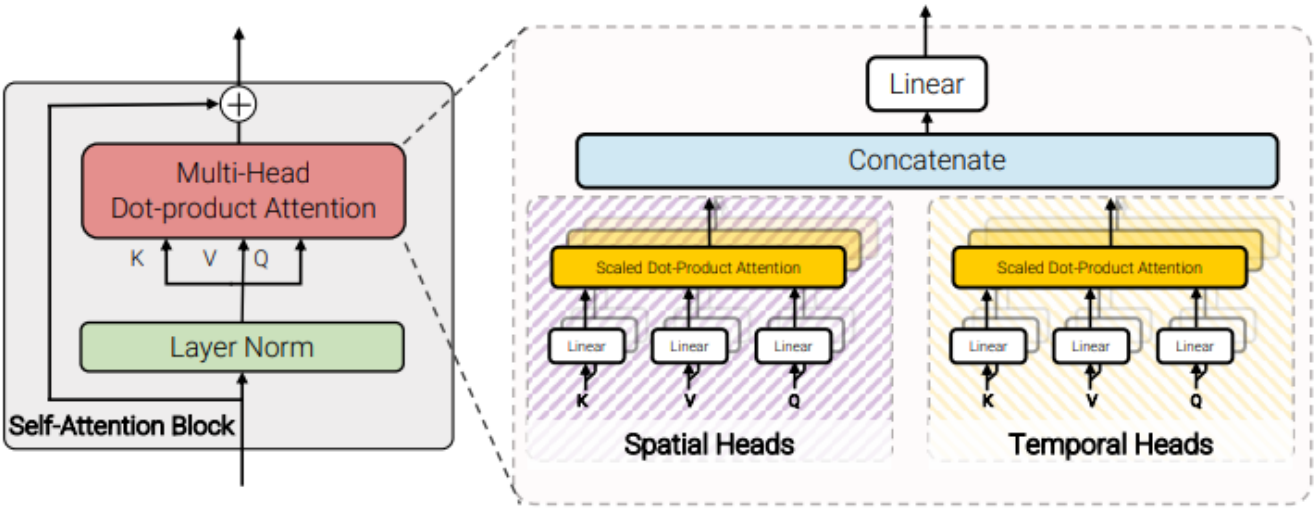


Figure 6: Factorised dot-product attention (Model 4). For half of the heads, we compute dot-product attention over only the spatial axes, and for the other half, over only the temporal axis.

图7. Factorised dot-product attention

如图所示，模型采用分离的MSA，使用不同的heads来分别计算spatial和temporal。我们定义attention公式为:

$$\text{Attention}(\mathrm{Q},\mathrm{K},\mathrm{V}) = \text{Softmax}\left(\frac{\mathrm{QK}^T}{\sqrt{d_k}}\right)\mathrm{V}$$

其中，$\mathrm{Q} = \mathrm{XW}_q, \mathrm{K} = \mathrm{XW}_k, \mathrm{V} = \mathrm{XW}_v, \mathrm{X}, \mathrm{Q}, \mathrm{K}, \mathrm{V} \in \mathbb{R}^{N\times d}$，这里维度表示为$N = n_t \cdot n_w \cdot n_h$。

此结构的核心idea是构建空间$\mathrm{K}_s, \mathrm{V}_s \in \mathbb{R}^{n_h \cdot n_w \times d}$和时序$\mathrm{K}_t, \mathrm{V}_t \in \mathbb{R}^{n_t \times d}$,表示各自维度的key和value信息。用一半的heads来计算空间特征，$\mathrm{Y}_s = \text{Attention}(\mathrm{Q}, \mathrm{K}_s, \mathrm{V}_s)$，余下的heads用于计算时序特征$\mathrm{Y}_t = \text{Attention}(\mathrm{Q}, \mathrm{K}_t, \mathrm{V}_t)$最后把时序和空间特征concateate起来，并经过线性映射用于特征交互,$\mathrm{Y} = Concat(\mathrm{Y}_s, \mathrm{Y}_t)\mathrm{W}_o$。

Initialisation by leveraging pretrained models

广为人知的是，Transformer只有在大规模的数据集上表现很好，因为相比于CNN，有更少的归纳偏置。虽然目前很多视频数据集量足够大，但是标注级别相比于图像来说还是略少，所以想要从零开始训练一个高精度的大模型还是具有挑战性的。为了规避问题，需要使用image的pretrain来做video的初始

化，但是video模型和image模型还是存在部分差异的，不能完全迁移权重，本文提出了几个有效的策略来解决这个问题。

- **Positional embeddings**
  由于图像模型的 `positonal-embeeding` 维度为$\mathbb{R}^{n_w \cdot n_h \times d}$, 视频具有一个时序的维度$n_t$，所以为了维度匹配，直接按t维度进行repeat为$\mathbb{R}^{n_w \cdot n_h \cdot n_t \times d}$。这样初始化阶段，每帧都具备相同的编码信息。
- **Embedding weights, E**
  对于 `tubelet embedding` 来说，用的是3d卷积，图像经常使用的是2d卷积，那么就需要考虑如何把2d迁移到3d上去。第一种方法是"inflate"，简单来说，按t维度上进行复制，然后再求个平均，表达式如下：

$$\mathrm{E} = \frac{1}{t}[\mathrm{E}_{\text{image}}, ..., \mathrm{E}_{\text{image}}, ..., \mathrm{E}_{\text{image}}]$$

  另一种方法是"central frame initialisation", 除了中间帧初始化以外，其他的帧都设置为0，表达如下:

$$\mathrm{E} = [0, ..., textup\mathrm{E}_{\text{image}}, ..., 0]$$

- **Transformer weights for Model 3**
  模型3的结构设计，是独立的空间attention和时序attention，空间attention可以直接使用图像模型的pretrain，时序attention初始化为0。

四、代码分析

- **Model-1**
  model1就是标准的VIT结构，除了patchembeeding以外没有任何的改变，直接看vit代码就可以了。
- **Model-2**
  有spatial-attention和temporal-attention，所以需要注意cls-token的初始化和变化情况，代码如下:

```
self.to_patch_embedding = nn.Sequential(
        Rearrange('b t c (h p1) (w p2) -> b t (h w) (p1 p2 c)', p1 = patch_size, p2 = patch_
        nn.Linear(patch_dim, dim),
    )# 这部分可以用conv2d直接替换掉
self.pos_embedding = nn.Parameter(torch.randn(1, num_frames, num_patches + 1, dim))
self.space_token = nn.Parameter(torch.randn(1, 1, dim))
self.space_transformer = Transformer(dim, depth, heads, dim_head, dim*scale_dim, dropout)

self.temporal_token = nn.Parameter(torch.randn(1, 1, dim))
self.temporal_transformer = Transformer(dim, depth, heads, dim_head, dim*scale_dim, dropout)
```

  上述代码两个token， `space_token` 用来表示spatial的clstoken， `temporal_token` 用来表示temporal的clstoken。其实，cls-token的目的是最终的分类，model2的模型最后的token输出是来自于

temporal-transformer的，所以space的cls-token其实是不需要的，直接用avgpool来获取spatial-transformer的结果就可以了，下面看一下forward部分。

```
x = self.to_patch_embedding(x)
b, t, n, _ = x.shape

cls_space_tokens = repeat(self.space_token, '() n d -> b t n d', b = b, t=t)
x = torch.cat((cls_space_tokens, x), dim=2)
x += self.pos_embedding[:, :, :(n + 1)]
x = self.dropout(x)

x = rearrange(x, 'b t n d -> (b t) n d')
x = self.space_transformer(x)
x = rearrange(x[:, 0], '(b t) ... -> b t ...', b=b)

cls_temporal_tokens = repeat(self.temporal_token, '() n d -> b n d', b=b)
x = torch.cat((cls_temporal_tokens, x), dim=1)

x = self.temporal_transformer(x)

x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]
```

这里用了 `einops` 库来实现，简单易懂，做了三个操作，第一个就是获取 `patch_embeeding`，先进行spatial-attention的计算，把space_token按的维度 `patch_embeeding` 进行扩展后concat，reshape为((bt),(n+1),d)后送入space_transformer。输出和输入维度一致，需要计算temporal的attention，重排并concat上cls-token，shape为((b,n),(t+1),d)，送入temporal-attention后，用cls-token进行分类计算，over。

- **Model-3**
  model3的实现和TimeSformer的实现是一样的，去掉cls-token即可，可以参考TimeSformer的文章。
- **Model-4**
  model4的实现与model1不同之处在于，transformer是有两个不同维度的attention来进行计算的，代码如下：

```
class Attention(nn.Module):
def __init__(self, dim, heads=8, dim_head=64, dropout=0.0, num_patches_space=None, num_patch
    super().__init__()

    assert attn_type in ['space', 'time'], 'Attention type should be one of the following: s

    self.attn_type = attn_type
    self.num_patches_space = num_patches_space
    self.num_patches_time = num_patches_time

    inner_dim = dim_head * heads
    self.scale = dim_head ** -0.5
    self.heads = heads

    self.attend = nn.Softmax(dim=-1)
    self.to_qkv = nn.Linear(dim, inner_dim * 3, bias=False)

def forward(self, x):

    t = self.num_patches_time
    n = self.num_patches_space

    # reshape to reveal dimensions of space and time
    x = rearrange(x, 'b (t n) d -> b t n d', t=t, n=n)  #  (b, t, n, d)

    if self.attn_type == 'space':
        out = self.forward_space(x) # (b, tn, d)
    elif self.attn_type == 'time':
        out = self.forward_time(x) # (b, tn, d)
    else:
        raise Exception('Unknown attention type: %s' % (self.attn_type))

    return out

def forward_space(self, x):
    """
    x: (b, t, n, d)
    """

    t = self.num_patches_time
    n = self.num_patches_space

    # hide time dimension into batch dimension
    x = rearrange(x, 'b t n d -> (b t) n d')  # (bt, n, d)

    # apply self-attention
    out = self.forward_attention(x)  # (bt, n, d)

    # recover time dimension and merge it into space
    out = rearrange(out, '(b t) n d -> b (t n) d', t=t, n=n)  # (b, tn, d)
```

```
        return out

    def forward_time(self, x):
        """
        x: (b, t, n, d)
        """

        t = self.num_patches_time
        n = self.num_patches_space

        # hide time dimension into batch dimension
        x = x.permute(0, 2, 1, 3)  # (b, n, t, d)
        x = rearrange(x, 'b n t d -> (b n) t d')  # (bn, t, d)

        # apply self-attention
        out = self.forward_attention(x)  # (bn, t, d)

        # recover time dimension and merge it into space
        out = rearrange(out, '(b n) t d -> b (t n) d', t=t, n=n)  # (b, tn, d)

        return out

    def forward_attention(self, x):
        h = self.heads
        qkv = self.to_qkv(x).chunk(3, dim=-1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h=h), qkv)

        dots = einsum('b h i d, b h j d -> b h i j', q, k) * self.scale
        attn = self.attend(dots)

        out = einsum('b h i j, b h j d -> b h i d', attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')

        return out
```

`forward_attention` 和 `forward_time` 分别用于计算((b, t), n, d)和((b, n), t, d)维度的token的attention。

```
class Transformer(nn.Module):
def __init__(self, dim, heads, dim_head, mlp_dim, dropout, num_patches_space, num_patches_ti
    super().__init__()

    self.num_patches_space = num_patches_space
    self.num_patches_time = num_patches_time
    heads_half = int(heads / 2.0)

    assert dim % 2 == 0

    self.attention_space = PreNorm(dim, Attention(dim, heads=heads_half, dim_head=dim_head,
    self.attention_time = PreNorm(dim, Attention(dim, heads=heads_half, dim_head=dim_head, d

    inner_dim = dim_head * heads_half * 2
    self.linear = nn.Sequential(nn.Linear(inner_dim, dim), nn.Dropout(dropout))
    self.mlp = PreNorm(dim, FeedForward(dim, mlp_dim, dim, dropout=dropout))

def forward(self, x):

    # self-attention
    xs = self.attention_space(x)
    xt = self.attention_time(x)
    out_att = torch.cat([xs, xt], dim=2)

    # linear after self-attention
    out_att = self.linear(out_att)

    # residual connection for self-attention
    out_att += x

    # mlp after attention
    out_mlp = self.mlp(out_att)

    # residual for mlp
    out_mlp += out_att

    return out_mlp
```

得到的不同维度的attention进行concat，做一个线性映射后接MLP，其余和model1保持一致。这份代码实现里面没有用到cls-token，直接求了mean。

五、实验结果

Input encoding

Table 1: Comparison of input encoding methods using ViViT-B and spatio-temporal attention on Kinetics. Further details in text.

	Top-1 accuracy
Uniform frame sampling	78.5
Tubelet embedding	
Random initialisation [24]	73.2
Filter inflation [8]	77.6
Central frame	79.2

为了比较embeeding方法，这里模型采用的是model1的结构，输入为32帧的video, 采样8帧，使用tubelet embeeding的话，设置$t = 4$, 这样token的数量可以保持一致。实验结果表明，使用tubelet embeeding并且使用central frame的初始化的方法可以达到最好的精度，后续所有实验均采用此方法。

Model variants

Table 2: Comparison of model architectures using ViViT-B as the backbone, and tubelet size of 16×2. We report Top-1 accuracy on Kinetics 400 (K400) and action accuracy on Epic Kitchens (EK). Runtime is during inference on a TPU-v3.

	K400	EK	FLOPs ($\times 10^9$)	Params ($\times 10^6$)	Runtime (ms)
Model 1: Spatio-temporal	80.0	43.1	455.2	88.9	58.9
Model 2: Fact. encoder	78.8	43.7	284.4	115.1	17.4
Model 3: Fact. self-attention	77.4	39.1	372.3	117.3	31.7
Model 4: Fact. dot product	76.3	39.5	277.1	88.9	22.9
Model 2: Ave. pool baseline	75.8	38.8	283.9	86.7	17.3

比较模型性能，这里Model2的temporal-transformer设置4层。model1的性能最好，但是FLOPs最大，运行时间最长。Model4没有额外的参数量，计算量比model1少很多，但是性能不高。Model3相比与其他的变体，需要更高的计算量和参数量。Model2表现最佳，精度尚可，计算量和运行时比较低。最后一

行是单独做的实验，去掉了Model2的temporal transformer，直接在帧上做了pooling，EK上的精度下降很多，对于时序强的数据集需要用temporal transformer来做时序信息交互。

Table 3: The effect of varying the number of temporal transformers, L_t, in the Factorised encoder model (Model 2). We report the Top-1 accuracy on Kinetics 400. Note that $L_t = 0$ corresponds to the "average pooling baseline".

L_t	0	1	4	8	12
Top-1	75.8	78.6	78.8	78.8	78.9

可以看到Model2取temporal-transorformer层数为4的时候可以达到最优效果和最佳参数量。

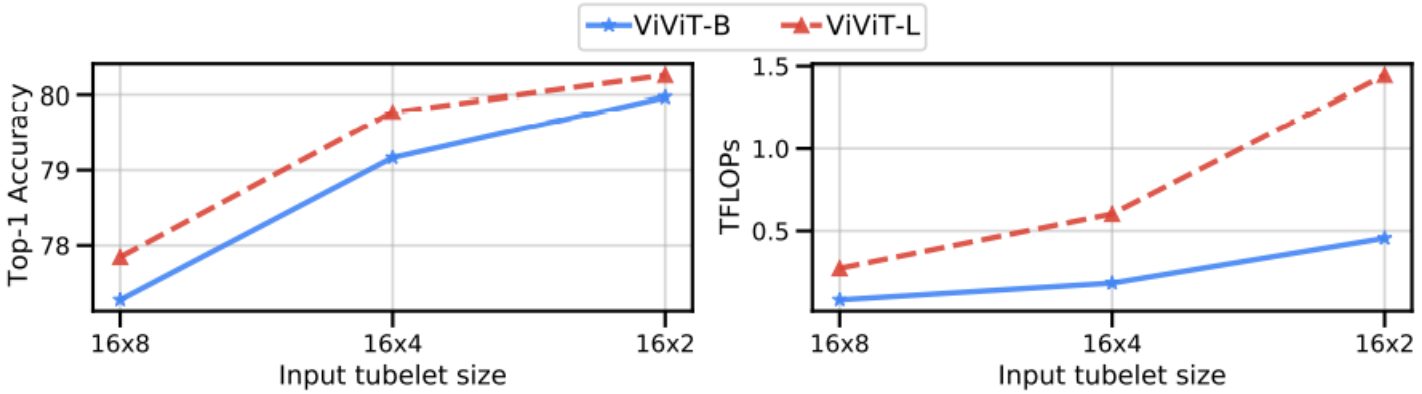
Model regularisation

Table 3: The effect of progressively adding regularisation (each row includes all methods above it) on Top-1 action accuracy on Epic Kitchens. We use ViViT-B/16x2 Factorised Encoder.

	Top-1 accuracy
Random crop, flip, colour jitter	38.4
+ Kinetics 400 initialisation	39.6
+ Stochastic depth [30]	40.2
+ Random augment [12]	41.1
+ Label smoothing [60]	43.1
+ Mixup [81]	43.7

做了一些正则化实验，可以看到使用部分CNN的正则化训练手段也可以有效的提升ViViT在小规模数据集上的性能。在ssv2数据集上也同样提升了近5%的性能。对于Kinetics和Moments in Time数据集，除了第一行以外的正则都没有使用就已经SOTA了。

Varying the backbone

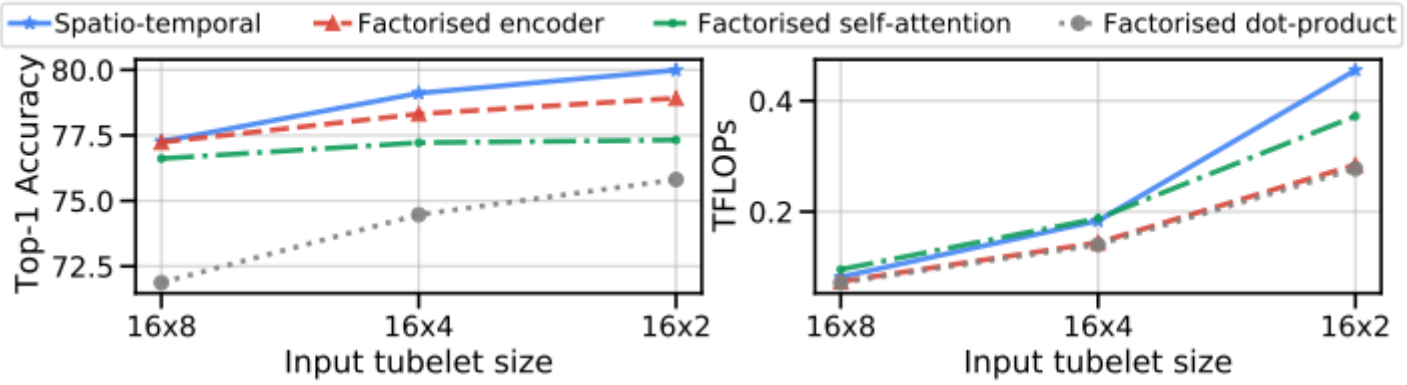


(a) Accuracy (b) Compute

Figure 7: The effect of the backbone architecture on (a) accuracy and (b) computation on Kinetics 400, for the spatio-temporal attention model (Model 1).

使用更大的模型会有更高的精度，不过计算量的增长速度远超精度，收益不高。

Varying the number of tokens



(a) Accuracy (b) Compute

Figure 6: The effect of varying the number of temporal tokens on (a) accuracy and (b) computation on Kinetics 400, for different variants of our model with a ViViT-B backbone.

使用更小的tubelet size可以提升所有模型的精度，当然计算量也是跟着增长，Model1的影响最大。

Table 4: The effect of spatial resolution on the performance of ViViT-L/16x2 and spatio-temporal attention on Kinetics 400.

Crop size	224	288	320
Accuracy	80.3	80.7	81.0
GFLOPs	1446	2919	3992
Runtime	58.9	147.6	238.8

图像尺寸从224提升到了320，精度和计算量也都随着增长，不过仅使用224的尺寸就可以SOTA了。

Varying the number of input frames

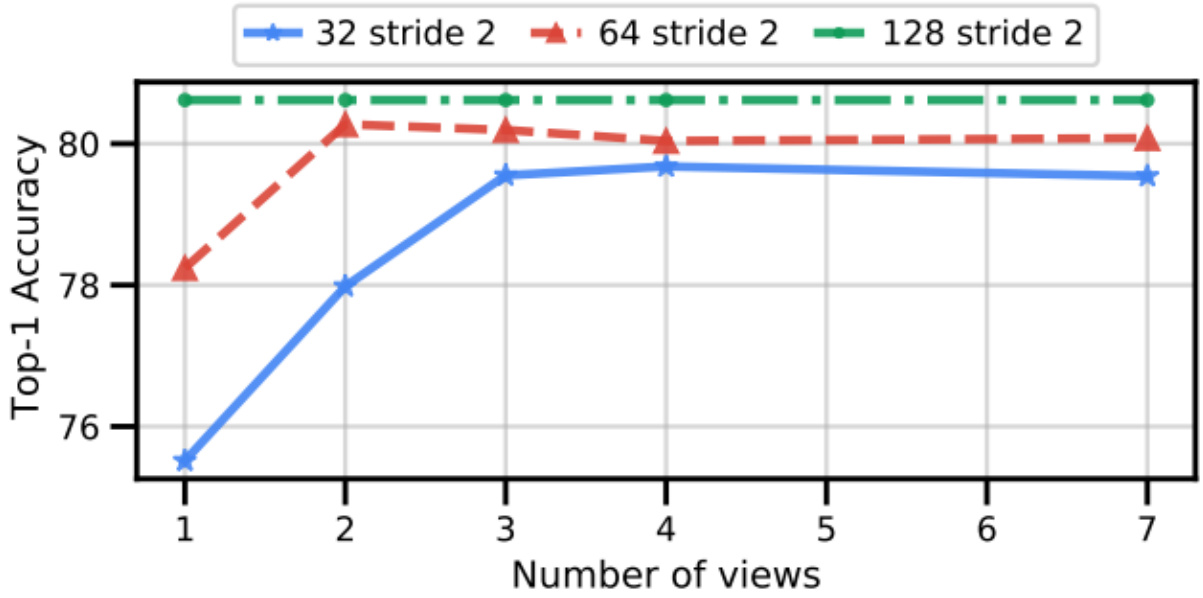


Figure 7: The effect of varying the number of frames input to the network and increasing the number of tokens proportionally. A Kinetics video contains 250 frames (10 seconds sampled at 25 fps) and the accuracy for each model saturates once the number of equidistant temporal views is sufficient to "see" the whole video clip. Observe how models processing more frames (and thus more tokens) achieve higher single- and multi-view accuracy.

对于K400这种短视频来说，总计250帧，使用stride2采样128帧足够遍历一整个视频了，不管用多少个view的片段都不会影响精度。其他的clips只要views满足遍历一整个视频的时候，精度都不会在提升。

Comparison to state-of-the-art

Table 5: Comparisons to state-of-the-art across multiple datasets. For "views", $x \times y$ denotes x temporal crops and y spatial crops. We report the TFLOPs to process all spatio-temporal views. "FE" denotes our Factorised Encoder model.

(a) Kinetics 400

Method	Top 1	Top 5	Views	TFLOPs
blVNet [18]	73.5	91.2	–	–
STM [32]	73.7	91.6	–	–
TEA [41]	76.1	92.5	10 × 3	2.10
TSM-ResNeXt-101 [42]	76.3	–	–	–
I3D NL [74]	77.7	93.3	10 × 3	10.77
CorrNet-101 [69]	79.2	–	10 × 3	6.72
ip-CSN-152 [65]	79.2	93.8	10 × 3	3.27
LGD-3D R101 [50]	79.4	94.4	–	–
SlowFast R101-NL [20]	79.8	93.9	10 × 3	7.02
X3D-XXL [19]	80.4	94.6	10 × 3	5.82
TimeSformer-L [4]	80.7	**94.7**	1 × 3	7.14
ViViT-L/16x2 FE	80.6	92.7	1 × 1	3.98
ViViT-L/16x2 FE	**81.7**	93.8	1 × 3	11.94
Methods with large-scale pretraining				
ip-CSN-152 [65] (IG [43])	82.5	95.3	10 × 3	3.27
ViViT-L/16x2 FE (JFT)	83.5	94.3	1 × 3	11.94
ViViT-H/16x2 (JFT)	**84.9**	**95.8**	4 × 3	47.77

(b) Kinetics 600

Method	Top 1	Top 5
AttentionNAS [75]	79.8	94.4
LGD-3D R101 [50]	81.5	95.6
SlowFast R101-NL [20]	81.8	95.1
X3D-XL [19]	81.9	95.5
TimeSformer-L [4]	82.2	**95.6**
ViViT-L/16x2 FE	**82.9**	94.6
ViViT-L/16x2 FE (JFT)	84.3	94.9
ViViT-H/16x2 (JFT)	**85.8**	**96.5**

(c) Moments in Time

	Top 1	Top 5
TSN [71]	25.3	50.1
TRN [85]	28.3	53.4
I3D [8]	29.5	56.1
blVNet [18]	31.4	59.3
AssembleNet-101 [53]	34.3	62.7
ViViT-L/16x2 FE	**38.5**	**64.1**

(d) Epic Kitchens 100 Top 1 accuracy

Method	Action	Verb	Noun
TSN [71]	33.2	60.2	46.0
TRN [85]	35.3	65.9	45.4
TBN [35]	36.7	66.0	47.2
TSM [42]	38.3	**67.9**	49.0
SlowFast [20]	38.5	65.6	50.0
ViViT-L/16x2 FE	**44.0**	66.4	**56.8**

(e) Something-Something v2

Method	Top 1	Top 5
TRN [85]	48.8	77.6
SlowFast [19, 79]	61.7	–
TimeSformer-HR [4]	62.5	–
TSM [42]	63.4	88.5
STM [32]	64.2	89.8
TEA [41]	65.1	–
blVNet [18]	65.2	**90.3**
ViVIT-L/16x2 FE	**65.9**	89.9

- **Kinetics**
  K400上，使用JFT做pretrain相比于imagenet21k高了3.2%，取得了84.9%的高精度，pretrain对于模型的影响还是至关重要的。当然只使用imagenet21k的ViViT-L就已经超过了TimeSformer-L的效果了。推理使用的是1个clips，3个crop(left,center,right)。
  K600上，ViViT同样SOTA, JFT pretrian高出了imagenet21k pretrian 2.9%，取得了85.8%的高精度。
- **Moments in Time**
  由于数据集庞杂，标注不精细，整体准确率都偏低，ViViT取得了38.5%的SOTA。
- **Epic Kitchens 100**
  除了"verb"以外均取得了大幅度的领先SOTA，"verb"其他模型更高的结果是因为引入了光流(PS:我查了一下TSM的dataset貌似没有EK的对比结果)。
- **Something-Something v2 (SSv2)**
  ssv2上的结果虽然SOTA，但是没有很大幅度的领先，不过超过了TimeSformer3个点多，说明了ViViT模型结构设计的更合理。ssv2的不同类别的背景和物体非常相似，这意味着识别细粒度的运动模式对于区分不同类别是必要的。

Training config

训练超参数设置详情

Table 7: Training hyperparamters for experiments in the main paper. "–" indicates that the regularisation method was not used at all. Values which are constant across all columns are listed once. Datasets are denoted as follows: K400: Kinetics 400. K600: Kinetics 600. MiT: Moments in Time. EK: Epic Kitchens. SSv2: Something-Something v2.

	K400	K600	MiT	EK	SSv2
Optimisation					
Optimiser			Synchronous SGD		
Momentum			0.9		
Batch size			64		
Learning rate schedule			cosine with linear warmup		
Linear warmup epochs			2.5		
Base learning rate	0.1	0.1	0.25	0.5	0.5
Epochs	30	30	10	50	35
Data augmentation					
Random crop probability			1.0		
Random flip probability			0.5		
Scale jitter probability			1.0		
Maximum scale			1.33		
Minimum scale			0.9		
Colour jitter probability	0.8	0.8	0.8	–	–
Rand augment number of layers [12]	–	–	–	2	2
Rand augment magnitude [12]	–	–	–	15	20
Other regularisation					
Stochastic droplayer rate, p_{drop} [31]	–	–	–	0.2	0.3
Label smoothing λ [61]	–	–	–	0.2	0.3
Mixup α [82]	–	–	–	0.1	0.3

六、结论

- 设计了几种不同的VideoViT范式，提出了最有效的Factorised encoder结构。
- 相比TimeSformer，思考的更全面，结构设计上更简单有效。
- 提出了更好的迁移imagenet pretrain到video模型的方法，central frame。
- 提出了tubelet embeeding的方法，更好的获取temporal embeeding信息。
- 多个数据集上取得了SOTA的效果并且大幅度领先。