PyTorchVideo

PyTorchVideo contributors

Sep 15, 2021

MODELS

1 Overview 1

2 Model Zoo and Benchmarks 5

3 Models API 7

4 Overview 59

5 Data Preparation 61

6 Data API 65

7 Overview 77

8 Transforms API 79

9 Overview 95

10 Layers API 97

11 Overview 107

Python Module Index 109

Index 111

CHAPTER

ONE

OVERVIEW

PyTorchVideo is an open source video understanding library that provides up to date builders for state of the art video understanding backbones, layers, heads, and losses addressing different tasks, including acoustic event detection, action recognition (video classification), action detection (video detection), multimodal understanding (acoustic visual classification), self-supervised learning.

The models subpackage contains definitions for the following model architectures and layers:

- Acoustic Backbone
  - Acoustic ResNet
- Visual Backbone
  - I3D
  - C2D
  - Squeeze-and-Excitation Networks
  - Nonlocal Networks
  - R2+1D
  - CSN
  - SlowFast
  - Audiovisual SlowFast
  - X3D
- Self-Supervised Learning
  - SimCLR
  - Bootstrap Your Own Latent
  - Non-Parametric Instance Discrimination

1.1 Build standard models

PyTorchVideo provide default builders to construct state-of-the-art video understanding models, layers, heads, and losses.

1.1.1 Models

You can construct a model with random weights by calling its constructor:

```
import pytorchvideo.models as models

resnet = models.create_resnet()
acoustic_resnet = models.create_acoustic_resnet()
slowfast = models.create_slowfast()
x3d = models.create_x3d()
r2plus1d = models.create_r2plus1d()
csn = models.create_csn()
```

You can verify whether you have built the model successfully by:

```
import pytorchvideo.models as models

resnet = models.create_resnet()
B, C, T, H, W = 2, 3, 8, 224, 224
input_tensor = torch.zeros(B, C, T, H, W)
output = resnet(input_tensor)
```

1.1.2 Layers

You can construct a layer with random weights by calling its constructor:

```
import pytorchvideo.layers as layers

nonlocal = layers.create_nonlocal(dim_in=256, dim_inner=128)
swish = layers.Swish()
conv_2plus1d = layers.create_conv_2plus1d(in_channels=256, out_channels=512)
```

You can verify whether you have built the model successfully by:

```
import pytorchvideo.layers as layers

nonlocal = layers.create_nonlocal(dim_in=256, dim_inner=128)
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = nonlocal(input_tensor)

swish = layers.Swish()
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = swish(input_tensor)

conv_2plus1d = layers.create_conv_2plus1d(in_channels=256, out_channels=512)
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = conv_2plus1d(input_tensor)
```

1.1.3 Heads

You can construct a head with random weights by calling its constructor:

```
import pytorchvideo.models as models

res_head = models.head.create_res_basic_head(in_features, out_features)
x3d_head = models.x3d.create_x3d_head(dim_in=1024, dim_inner=512, dim_out=2048, num_
↪classes=400)
```

You can verify whether you have built the head successfully by:

```
import pytorchvideo.models as models

res_head = models.head.create_res_basic_head(in_features, out_features)
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = res_head(input_tensor)

x3d_head = models.x3d.create_x3d_head(dim_in=1024, dim_inner=512, dim_out=2048, num_
↪classes=400)
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = x3d_head(input_tensor)
```

1.1.4 Losses

You can construct a loss by calling its constructor:

```
import pytorchvideo.models as models

simclr_loss = models.SimCLR()
```

You can verify whether you have built the loss successfully by:

```
import pytorchvideo.models as models
import pytorchvideo.layers as layers

resnet = models.create_resnet()
mlp = layers.make_multilayer_perceptron(fully_connected_dims=(2048, 1024, 2048))
simclr_loss = models.SimCLR(mlp=mlp, backbone=resnet)
B, C, T, H, W = 2, 256, 4, 14, 14
view1, view2 = torch.zeros(B, C, T, H, W), torch.zeros(B, C, T, H, W)
loss = simclr_loss(view1, view2)
```

1.2 Build customized models

PyTorchVideo also supports building models with customized components, which is an important feature for video understanding research. Here we take a standard stem model as an example, show how to build each resnet components (head, backbone, stem) separately, and how to use your customized components to replace standard components.

```
from pytorchvideo.models.stem import create_res_basic_stem


# Create standard stem layer.
stem = create_res_basic_stem(in_channels=3, out_channels=64)

# Create customized stem layer with YourFancyNorm
stem = create_res_basic_stem(
    in_channels=3,
    out_channels=64,
    norm=YourFancyNorm,  # GhostNorm for example
)

# Create customized stem layer with YourFancyConv
stem = create_res_basic_stem(
    in_channels=3,
    out_channels=64,
    conv=YourFancyConv,  # OctConv for example
)

# Create customized stem layer with YourFancyAct
stem = create_res_basic_stem(
    in_channels=3,
    out_channels=64,
    activation=YourFancyAct,  # Swish for example
)

# Create customized stem layer with YourFancyPool
stem = create_res_basic_stem(
    in_channels=3,
    out_channels=64,
    pool=YourFancyPool,  # MinPool for example
)
```

CHAPTER

TWO

MODEL ZOO AND BENCHMARKS

PyTorchVideo provides reference implementation of a large number of video understanding approaches. In this document, we also provide comprehensive benchmarks to evaluate the supported models on different datasets using standard evaluation setup. All the models can be downloaded from the provided links.

2.1 Kinetics-400

2.2 Something-Something V2

2.3 Charades

2.4 AVA (V2.2)

2.5 Using PyTorchVideo model zoo

We provide several different ways to use PyTorchVideo model zoo.

- The models have been integrated into TorchHub, so could be loaded with TorchHub with or without pre-trained models. Additionally, we provide a tutorial which goes over the steps needed to load models from TorchHub and perform inference.
- PyTorchVideo models/datasets are also supported in PySlowFast. You can use PySlowFast workflow to train or test PyTorchVideo models/datasets.
- You can also use PyTorch Lightning to build training/test pipeline for PyTorchVideo models and datasets. Please check this tutorial for more information.

Notes:

- The above benchmarks are conducted by PySlowFast workflow using PyTorchVideo datasets and models.
- For more details on the data preparation, you can refer to *PyTorchVideo Data Preparation*.
- For `Flops x views` column, we report the inference cost with a single "view" × the number of views (FLOPs × space_views × time_views). For example, we take 3 spatial crops for 10 temporal clips on Kinetics.

2.6 PytorchVideo Accelerator Model Zoo

Accelerator model zoo provides a set of efficient models on target device with pretrained checkpoints. To learn more about how to build model, load checkpoint and deploy, please refer to Use PyTorchVideo/Accelerator Model Zoo.

Efficient Models for mobile CPU All top1/top5 accuracies are measured with 10-clip evaluation. Latency is benchmarked on Samsung S8 phone with 1s input clip length.

2.7 TorchHub models

We provide a large set of TorchHub models for the above video models with pre-trained weights. So it's easy to construct the networks and load pre-trained weights. Please refer to PytorchVideo TorchHub models for more details.

CHAPTER

THREE

MODELS API

3.1 pytorchvideo.models.resnet

Building blocks for Resnet and resnet-like models

pytorchvideo.models.resnet.**create_bottleneck_block**(*, *dim_in*, *dim_inner*, *dim_out*, *conv_a_kernel_size=(3, 1, 1)*, *conv_a_stride=(2, 1, 1)*, *conv_a_padding=(1, 0, 0)*, *conv_a=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_b_kernel_size=(1, 3, 3)*, *conv_b_stride=(1, 2, 2)*, *conv_b_padding=(0, 1, 1)*, *conv_b_num_groups=1*, *conv_b_dilation=(1, 1, 1)*, *conv_b=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_c=<class 'torch.nn.modules.conv.Conv3d'>*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Bottleneck block: a sequence of spatiotemporal Convolution, Normalization, and Activations repeated in the following order:

```
   Conv3d (conv_a)
         ↓
Normalization (norm_a)
         ↓
  Activation (act_a)
         ↓
   Conv3d (conv_b)
         ↓
Normalization (norm_b)
         ↓
  Activation (act_b)
         ↓
   Conv3d (conv_c)
```

(continues on next page)

(continued from previous page)

```
          ↓
Normalization (norm_c)
```

Normalization examples include: BatchNorm3d and None (no normalization). Activation examples include: ReLU, Softmax, Sigmoid, and None (no activation).

Parameters

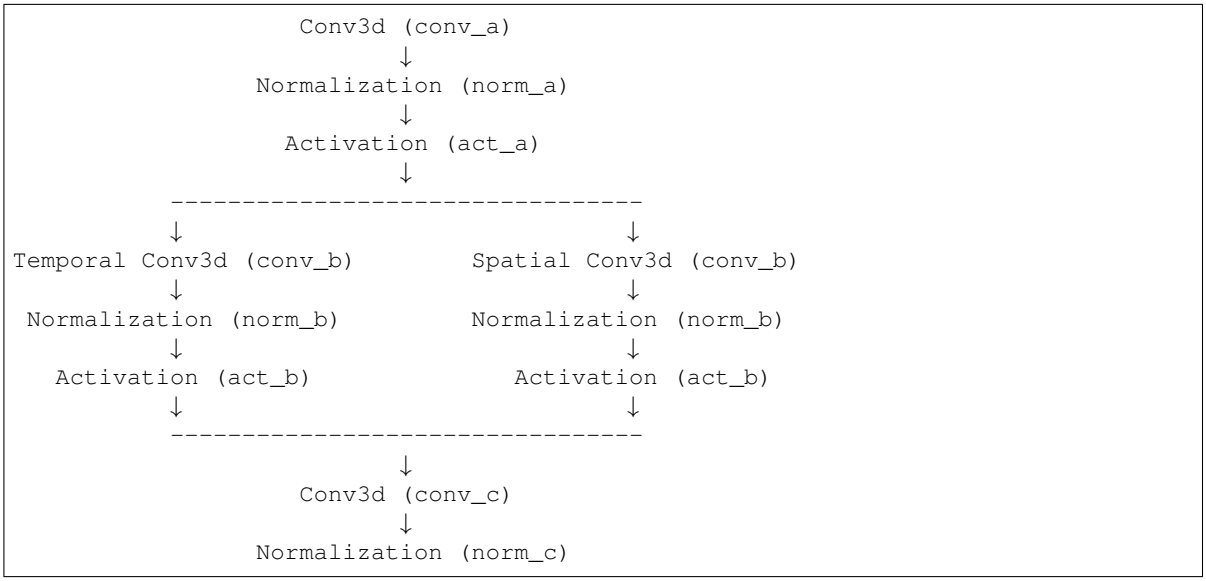
- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **conv_a_stride** (*tuple*) – convolutional stride size(s) for conv_a.
- **conv_a_padding** (*tuple*) – convolutional padding(s) for conv_a.
- **conv_a** (*callable*) – a callable that constructs the conv_a conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_b_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **conv_b_padding** (*tuple*) – convolutional padding(s) for conv_b.
- **conv_b_num_groups** (*int*) – number of groups for groupwise convolution for conv_b.
- **conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **conv_b** (*callable*) – a callable that constructs the conv_b conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_c** (*callable*) – a callable that constructs the conv_c conv layer, examples include nn.Conv3d, OctaveConv, etc
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet bottleneck block.

Return type torch.nn.modules.module.Module

pytorchvideo.models.resnet.**create_acoustic_bottleneck_block**(*, *dim_in*, *dim_inner*, *dim_out*, *conv_a_kernel_size=(3, 1, 1)*, *conv_a_stride=(2, 1, 1)*, *conv_a_padding=(1, 0, 0)*, *conv_a=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_b_kernel_size=(1, 1, 1)*, *conv_b_stride=(1, 1, 1)*, *conv_b_padding=(0, 0, 0)*, *conv_b_num_groups=1*, *conv_b_dilation=(1, 1, 1)*, *conv_b=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_c=<class 'torch.nn.modules.conv.Conv3d'>*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Acoustic Bottleneck block: a sequence of spatiotemporal Convolution, Normalization, and Activations repeated in the following order:



Normalization examples include: BatchNorm3d and None (no normalization). Activation examples include: ReLU, Softmax, Sigmoid, and None (no activation).

Parameters

- **dim_in** (*int*) – input channel size to the bottleneck block.

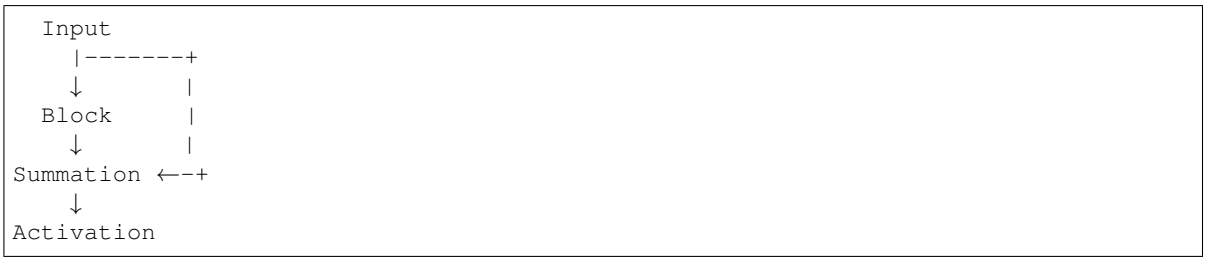
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **conv_a_stride** (*tuple*) – convolutional stride size(s) for conv_a.
- **conv_a_padding** (*tuple*) – convolutional padding(s) for conv_a.
- **conv_a** (*callable*) – a callable that constructs the conv_a conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_b_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **conv_b_padding** (*tuple*) – convolutional padding(s) for conv_b.
- **conv_b_num_groups** (*int*) – number of groups for groupwise convolution for conv_b.
- **conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **conv_b** (*callable*) – a callable that constructs the conv_b conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_c** (*callable*) – a callable that constructs the conv_c conv layer, examples include nn.Conv3d, OctaveConv, etc
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet acoustic bottleneck block.

Return type torch.nn.modules.module.Module

pytorchvideo.models.resnet.**create_res_block**(*, *dim_in*, *dim_inner*, *dim_out*, *bottleneck*, *use_shortcut=False*, *branch_fusion=<function <lambda>>*, *conv_a_kernel_size=(3, 1, 1)*, *conv_a_stride=(2, 1, 1)*, *conv_a_padding=(1, 0, 0)*, *conv_a=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_b_kernel_size=(1, 3, 3)*, *conv_b_stride=(1, 2, 2)*, *conv_b_padding=(0, 1, 1)*, *conv_b_num_groups=1*, *conv_b_dilation=(1, 1, 1)*, *conv_b=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_c=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_skip=<class 'torch.nn.modules.conv.Conv3d'>*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation_bottleneck=<class 'torch.nn.modules.activation.ReLU'>*, *activation_block=<class 'torch.nn.modules.activation.ReLU'>*)

Residual block. Performs a summation between an identity shortcut in branch1 and a main block in branch2. When the input and output dimensions are different, a convolution followed by a normalization will be performed.



Normalization examples include: BatchNorm3d and None (no normalization). Activation examples include: ReLU, Softmax, Sigmoid, and None (no activation). Transform examples include: BottleneckBlock.

Parameters

- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **bottleneck** (*callable*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block.
- **use_shortcut** (*bool*) – If true, use conv and norm layers in skip connection.
- **branch_fusion** (*callable*) – a callable that constructs summation layer. Examples include: lambda x, y: x + y, OctaveSum.
- **conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **conv_a_stride** (*tuple*) – convolutional stride size(s) for conv_a.

- **conv_a_padding** (*tuple*) – convolutional padding(s) for conv_a.
- **conv_a** (*callable*) – a callable that constructs the conv_a conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_b_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **conv_b_padding** (*tuple*) – convolutional padding(s) for conv_b.
- **conv_b_num_groups** (*int*) – number of groups for groupwise convolution for conv_b.
- **conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **conv_b** (*callable*) – a callable that constructs the conv_b conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_c** (*callable*) – a callable that constructs the conv_c conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_skip** (*callable*) – a callable that constructs the conv_skip conv layer,
- **include nn.Conv3d** (*examples*) –
- **OctaveConv** –
- **etc** –
- **norm** (*callable*) – a callable that constructs normalization layer. Examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation_bottleneck** (*callable*) – a callable that constructs activation layer in bottleneck. Examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).
- **activation_block** (*callable*) – a callable that constructs activation layer used at the end of the block. Examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet basic block layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.resnet.**create_res_stage**(*, *depth*, *dim_in*, *dim_inner*, *dim_out*, *bottleneck*, *conv_a_kernel_size=(3, 1, 1)*, *conv_a_stride=(2, 1, 1)*, *conv_a_padding=(1, 0, 0)*, *conv_a=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_b_kernel_size=(1, 3, 3)*, *conv_b_stride=(1, 2, 2)*, *conv_b_padding=(0, 1, 1)*, *conv_b_num_groups=1*, *conv_b_dilation=(1, 1, 1)*, *conv_b=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_c=<class 'torch.nn.modules.conv.Conv3d'>*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Create Residual Stage, which composes sequential blocks that make up a ResNet. These blocks could be, for example, Residual blocks, Non-Local layers, or Squeeze-Excitation layers.

```
 Input
    ↓
ResBlock
    ↓
    .
    .
    .
    ↓
ResBlock
```

Normalization examples include: BatchNorm3d and None (no normalization). Activation examples include: ReLU, Softmax, Sigmoid, and None (no activation). Bottleneck examples include: create_bottleneck_block.

Parameters

- **depth** (*int*) – number of blocks to create.
- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **bottleneck** (*callable*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block.
- **conv_a_kernel_size** (*tuple or list of tuple*) – convolutional kernel size(s) for conv_a. If conv_a_kernel_size is a tuple, use it for all blocks in the stage. If conv_a_kernel_size is a list of tuple, the kernel sizes will be repeated until having same length of depth in the stage. For example, for conv_a_kernel_size = [(3, 1, 1), (1, 1, 1)], the kernel size for the first 6 blocks would be [(3, 1, 1), (1, 1, 1), (3, 1, 1), (1, 1, 1), (3, 1, 1)].
- **conv_a_stride** (*tuple*) – convolutional stride size(s) for conv_a.
- **conv_a_padding** (*tuple or list of tuple*) – convolutional padding(s) for conv_a. If conv_a_padding is a tuple, use it for all blocks in the stage. If conv_a_padding is a list of tuple, the padding sizes will be repeated until having same length of depth in the stage.

- **conv_a** (*callable*) – a callable that constructs the conv_a conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_b_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **conv_b_padding** (*tuple*) – convolutional padding(s) for conv_b.
- **conv_b_num_groups** (*int*) – number of groups for groupwise convolution for conv_b.
- **conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **conv_b** (*callable*) – a callable that constructs the conv_b conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_c** (*callable*) – a callable that constructs the conv_c conv layer, examples include nn.Conv3d, OctaveConv, etc
- **norm** (*callable*) – a callable that constructs normalization layer. Examples include nn.BatchNorm3d, and None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer. Examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet basic stage layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.resnet.**create_resnet**(*, *input_channel=3*, *model_depth=50*, *model_num_class=400*, *dropout_rate=0.5*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_dim_out=64*, *stem_conv_kernel_size=(3, 7, 7)*, *stem_conv_stride=(1, 2, 2)*, *stem_pool=<class 'torch.nn.modules.pooling.MaxPool3d'>*, *stem_pool_kernel_size=(1, 3, 3)*, *stem_pool_stride=(1, 2, 2)*, *stem=<function create_res_basic_stem>*, *stage1_pool=None*, *stage1_pool_kernel_size=(2, 1, 1)*, *stage_conv_a_kernel_size=((1, 1, 1), (1, 1, 1), (3, 1, 1), (3, 1, 1))*, *stage_conv_b_kernel_size=((1, 3, 3), (1, 3, 3), (1, 3, 3), (1, 3, 3))*, *stage_conv_b_num_groups=(1, 1, 1, 1)*, *stage_conv_b_dilation=((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 1, 1))*, *stage_spatial_h_stride=(1, 2, 2, 2)*, *stage_spatial_w_stride=(1, 2, 2, 2)*, *stage_temporal_stride=(1, 1, 1, 1)*, *bottleneck=<function create_bottleneck_block>*, *head=<function create_res_basic_head>*, *head_pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *head_pool_kernel_size=(4, 7, 7)*, *head_output_size=(1, 1, 1)*, *head_activation=None*, *head_output_with_global_average=True*)

Build ResNet style models for video recognition. ResNet has three parts: Stem, Stages and Head. Stem is the first Convolution layer (Conv1) with an optional pooling layer. Stages are grouped residual blocks. There are usually multiple stages and each stage may include multiple residual blocks. Head may include pooling, dropout, a fully-connected layer and global spatial temporal averaging. The three parts are assembled in the following order:

```
Input
  ↓
Stem
  ↓
Stage 1
  ↓
  .
  .
  .
  ↓
Stage N
  ↓
Head
```

Parameters

- **input_channel** (*int*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet. Options include: 50, 101, 152.

- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_dim_out** (*int*) – output channel size to stem.
- **stem_conv_kernel_size** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_stride** (*tuple*) – convolutional stride size(s) of stem.
- **stem_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **stem_pool_kernel_size** (*tuple*) – pooling kernel size(s).
- **stem_pool_stride** (*tuple*) – pooling stride size(s).
- **stem** (*callable*) – a callable that constructs stem layer. Examples include: create_res_video_stem.
- **stage_conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_num_groups** (*tuple*) – number of groups for groupwise convolution for conv_b. 1 for ResNet, and larger than 1 for ResNeXt.
- **stage_conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **stage_spatial_h_stride** (*tuple*) – the spatial height stride for each stage.
- **stage_spatial_w_stride** (*tuple*) – the spatial width stride for each stage.
- **stage_temporal_stride** (*tuple*) – the temporal stride for each stage.
- **bottleneck** (*callable*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block.
- **head** (*callable*) – a callable that constructs the resnet-style head. Ex: create_res_basic_head
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_pool_kernel_size** (*tuple*) – the pooling kernel size.
- **head_output_size** (*tuple*) – the size of output tensor for head.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **stage1_pool** (*Callable*) –
- **stage1_pool_kernel_size** (*Tuple[int]*) –

Returns *(nn.Module)* – basic resnet.

Return type torch.nn.modules.module.Module

pytorchvideo.models.resnet.**create_resnet_with_roi_head**(*, *input_channel=3*, *model_depth=50*, *model_num_class=80*, *dropout_rate=0.5*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_dim_out=64*, *stem_conv_kernel_size=(1, 7, 7)*, *stem_conv_stride=(1, 2, 2)*, *stem_pool=<class 'torch.nn.modules.pooling.MaxPool3d'>*, *stem_pool_kernel_size=(1, 3, 3)*, *stem_pool_stride=(1, 2, 2)*, *stem=<function create_res_basic_stem>*, *stage1_pool=None*, *stage1_pool_kernel_size=(2, 1, 1)*, *stage_conv_a_kernel_size=((1, 1, 1), (1, 1, 1), (3, 1, 1), (3, 1, 1))*, *stage_conv_b_kernel_size=((1, 3, 3), (1, 3, 3), (1, 3, 3), (1, 3, 3))*, *stage_conv_b_num_groups=(1, 1, 1, 1)*, *stage_conv_b_dilation=((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 2, 2))*, *stage_spatial_h_stride=(1, 2, 2, 1)*, *stage_spatial_w_stride=(1, 2, 2, 1)*, *stage_temporal_stride=(1, 1, 1, 1)*, *bottleneck=<function create_bottleneck_block>*, *head=<function create_res_roi_pooling_head>*, *head_pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *head_pool_kernel_size=(4, 1, 1)*, *head_output_size=(1, 1, 1)*, *head_activation=<class 'torch.nn.modules.activation.Sigmoid'>*, *head_output_with_global_average=False*, *head_spatial_resolution=(7, 7)*, *head_spatial_scale=0.0625*, *head_sampling_ratio=0*)

Build ResNet style models for video detection. ResNet has three parts: Stem, Stages and Head. Stem is the

first Convolution layer (Conv1) with an optional pooling layer. Stages are grouped residual blocks. There are usually multiple stages and each stage may include multiple residual blocks. Head may include pooling, dropout, a fully-connected layer and global spatial temporal averaging. The three parts are assembled in the following order:

Parameters

- **input_channel** (*int*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet. Options include: 50, 101, 152.
- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_dim_out** (*int*) – output channel size to stem.
- **stem_conv_kernel_size** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_stride** (*tuple*) – convolutional stride size(s) of stem.
- **stem_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **stem_pool_kernel_size** (*tuple*) – pooling kernel size(s).
- **stem_pool_stride** (*tuple*) – pooling stride size(s).
- **stem** (*callable*) – a callable that constructs stem layer. Examples include: create_res_video_stem.
- **stage_conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_num_groups** (*tuple*) – number of groups for groupwise convolution for conv_b. 1 for ResNet, and larger than 1 for ResNeXt.
- **stage_conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **stage_spatial_h_stride** (*tuple*) – the spatial height stride for each stage.
- **stage_spatial_w_stride** (*tuple*) – the spatial width stride for each stage.
- **stage_temporal_stride** (*tuple*) – the temporal stride for each stage.
- **bottleneck** (*callable*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block.

- **head** (*callable*) – a callable that constructs the detection head which can take in the additional input of bounding boxes. Ex: create_res_roi_pooling_head
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_pool_kernel_size** (*tuple*) – the pooling kernel size.
- **head_output_size** (*tuple*) – the size of output tensor for head.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **head_spatial_resolution** (*tuple*) – h, w sizes of the RoI interpolation.
- **head_spatial_scale** (*float*) – scale the input boxes by this number.
- **head_sampling_ratio** (*int*) – number of inputs samples to take for each output sample interpolation. 0 to take samples densely.
- **stage1_pool** (*Callable*) –
- **stage1_pool_kernel_size** (*Tuple[int]*) –

Returns *(nn.Module)* – basic resnet.

Return type torch.nn.modules.module.Module

pytorchvideo.models.resnet.**create_acoustic_resnet**(*, *input_channel=1*, *model_depth=50*, *model_num_class=400*, *dropout_rate=0.5*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_dim_out=64*, *stem_conv_kernel_size=(9, 1, 9)*, *stem_conv_stride=(1, 1, 3)*, *stem_pool=None*, *stem_pool_kernel_size=(3, 1, 3)*, *stem_pool_stride=(2, 1, 2)*, *stem=<function create_acoustic_res_basic_stem>*, *stage1_pool=None*, *stage1_pool_kernel_size=(2, 1, 1)*, *stage_conv_a_kernel_size=(3, 1, 1)*, *stage_conv_b_kernel_size=(3, 1, 3)*, *stage_conv_b_num_groups=(1, 1, 1, 1)*, *stage_conv_b_dilation=(1, 1, 1)*, *stage_spatial_h_stride=(1, 1, 1, 1)*, *stage_spatial_w_stride=(1, 2, 2, 2)*, *stage_temporal_stride=(1, 2, 2, 2)*, *bottleneck=(<function create_acoustic_bottleneck_block>, <function create_acoustic_bottleneck_block>, <function create_bottleneck_block>, <function create_bottleneck_block>)*, *head_pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *head_pool_kernel_size=(4, 1, 2)*, *head_output_size=(1, 1, 1)*, *head_activation=None*, *head_output_with_global_average=True*)

Build ResNet style models for acoustic recognition. ResNet has three parts: Stem, Stages and Head. Stem is the first Convolution layer (Conv1) with an optional pooling layer. Stages are grouped residual blocks. There are usually multiple stages and each stage may include multiple residual blocks. Head may include pooling, dropout, a fully-connected layer and global spatial temporal averaging. The three parts are assembled in the following order:

```
Input
  ↓
Stem
  ↓
Stage 1
  ↓
  .
  .
  .
  ↓
Stage N
```

(continues on next page)

(continued from previous page)

```
  ↓
Head
```

Parameters

- **input_channel** (*int*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet. Options include: 50, 101, 152.
- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_dim_out** (*int*) – output channel size to stem.
- **stem_conv_kernel_size** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_stride** (*tuple*) – convolutional stride size(s) of stem.
- **stem_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **stem_pool_kernel_size** (*tuple*) – pooling kernel size(s).
- **stem_pool_stride** (*tuple*) – pooling stride size(s).
- **stem** (*callable*) – a callable that constructs stem layer. Examples include: create_res_video_stem.
- **stage_conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_num_groups** (*tuple*) – number of groups for groupwise convolution for conv_b. 1 for ResNet, and larger than 1 for ResNeXt.
- **stage_conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **stage_spatial_h_stride** (*tuple*) – the spatial height stride for each stage.
- **stage_spatial_w_stride** (*tuple*) – the spatial width stride for each stage.
- **stage_temporal_stride** (*tuple*) – the temporal stride for each stage.
- **bottleneck** (*callable*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block.
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_pool_kernel_size** (*tuple*) – the pooling kernel size.
- **head_output_size** (*tuple*) – the size of output tensor for head.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **stage1_pool** (*Callable*) –
- **stage1_pool_kernel_size** (*Tuple[int]*) –

Returns

(nn.Module) –

audio resnet, that takes spectragram image input with shape: (B, C, T, 1, F), where T is the time dimension and F is the frequency dimension.

Return type torch.nn.modules.module.Module

class pytorchvideo.models.resnet.**ResBlock**(*branch1_conv=None*, *branch1_norm=None*, *branch2=None*, *activation=None*, *branch_fusion=None*)

Residual block. Performs a summation between an identity shortcut in branch1 and a main block in branch2. When the input and output dimensions are different, a convolution followed by a normalization will be performed.

The builder can be found in *create_res_block*.

__init__(*branch1_conv=None*, *branch1_norm=None*, *branch2=None*, *activation=None*, *branch_fusion=None*)

Parameters

- **branch1_conv** (*torch.nn.modules*) – convolutional module in branch1.
- **branch1_norm** (*torch.nn.modules*) – normalization module in branch1.
- **branch2** (*torch.nn.modules*) – bottleneck block module in branch2.
- **activation** (*torch.nn.modules*) – activation module.
- **branch_fusion** (*Callable*) – (Callable): A callable or layer that combines branch1 and branch2.

Return type torch.nn.modules.module.Module

class pytorchvideo.models.resnet.**SeparableBottleneckBlock**(*, *conv_a*, *norm_a*, *act_a*, *conv_b*, *norm_b*, *act_b*, *conv_c*, *norm_c*, *reduce_method='sum'*)

Separable Bottleneck block: a sequence of spatiotemporal Convolution, Normalization, and Activations repeated in the following order. Requires a tuple of models to be provided to conv_b, norm_b, act_b to perform Convolution, Normalization, and Activations in parallel Separably.

(continues on next page)

(continued from previous page)

```
Normalization(s) (norm_b), ...
          ↓ (↓)
   Activation(s) (act_b), ...
          ↓ (↓)
    Reduce (sum or cat)
            ↓
      Conv3d (conv_c)
            ↓
   Normalization (norm_c)
```

__init__(*, *conv_a*, *norm_a*, *act_a*, *conv_b*, *norm_b*, *act_b*, *conv_c*, *norm_c*, *reduce_method='sum'*)

Parameters

- **conv_a** (*torch.nn.modules*) – convolutional module.
- **norm_a** (*torch.nn.modules*) – normalization module.
- **act_a** (*torch.nn.modules*) – activation module.
- **conv_b** (*torch.nn.modules_list*) – convolutional module(s).
- **norm_b** (*torch.nn.modules_list*) – normalization module(s).
- **act_b** (*torch.nn.modules_list*) – activation module(s).
- **conv_c** (*torch.nn.modules*) – convolutional module.
- **norm_c** (*torch.nn.modules*) – normalization module.
- **reduce_method** (*str*) – if multiple conv_b is used, reduce the output with *sum*, or *cat*.

Return type None

class pytorchvideo.models.resnet.**BottleneckBlock**(*, *conv_a=None*, *norm_a=None*, *act_a=None*, *conv_b=None*, *norm_b=None*, *act_b=None*, *conv_c=None*, *norm_c=None*)

Bottleneck block: a sequence of spatiotemporal Convolution, Normalization, and Activations repeated in the following order:

```
   Conv3d (conv_a)
          ↓
Normalization (norm_a)
          ↓
  Activation (act_a)
          ↓
   Conv3d (conv_b)
          ↓
Normalization (norm_b)
          ↓
  Activation (act_b)
          ↓
   Conv3d (conv_c)
          ↓
Normalization (norm_c)
```

The builder can be found in *create_bottleneck_block*.

__init__(*, *conv_a=None*, *norm_a=None*, *act_a=None*, *conv_b=None*, *norm_b=None*, *act_b=None*, *conv_c=None*, *norm_c=None*)

Parameters

- **conv_a** (*torch.nn.modules*) – convolutional module.
- **norm_a** (*torch.nn.modules*) – normalization module.
- **act_a** (*torch.nn.modules*) – activation module.
- **conv_b** (*torch.nn.modules*) – convolutional module.
- **norm_b** (*torch.nn.modules*) – normalization module.
- **act_b** (*torch.nn.modules*) – activation module.
- **conv_c** (*torch.nn.modules*) – convolutional module.
- **norm_c** (*torch.nn.modules*) – normalization module.

Return type None

class pytorchvideo.models.resnet.**ResStage**(*res_blocks*)

ResStage composes sequential blocks that make up a ResNet. These blocks could be, for example, Residual blocks, Non-Local layers, or Squeeze-Excitation layers.

```
 Input
    ↓
ResBlock
    ↓
    .
    .
    .
    ↓
ResBlock
```

The builder can be found in *create_res_stage*.

__init__(*res_blocks*)

Parameters **res_blocks** (*torch.nn.module_list*) – ResBlock module(s).

Return type torch.nn.modules.module.Module

3.2 pytorchvideo.models.net

class pytorchvideo.models.net.**Net**(*, *blocks*)

Build a general Net models with a list of blocks for video recognition.

```
Input
  ↓
Block 1
  ↓
  .
  .
  .
  ↓
Block N
  ↓
```

The ResNet builder can be found in *create_resnet*.

__init__(*, *blocks*)

Parameters **blocks** (*torch.nn.module_list*) – the list of block modules.

Return type None

class pytorchvideo.models.net.**DetectionBBoxNetwork**(*model*, *detection_head*)
A general purpose model that handles bounding boxes as part of input.

__init__(*model*, *detection_head*)

Parameters

- **model** (*nn.Module*) – a model that preceeds the head. Ex: stem + stages.
- **detection_head** (*nn.Module*) – a network head. that can take in input bounding boxes and the outputs from the model.

forward(*x*, *bboxes*)

Parameters

- **x** (*torch.tensor*) – input tensor
- **bboxes** (*torch.tensor*) – accociated bounding boxes. The format is N*5 (Index, X_1,Y_1,X_2,Y_2) if using RoIAlign and N*6 (Index, x_ctr, y_ctr, width, height, angle_degrees) if using RoIAlignRotated.

class pytorchvideo.models.net.**MultiPathWayWithFuse**(*, *multipathway_blocks*, *multipathway_fusion*, *inplace=True*)
Build multi-pathway block with fusion for video recognition, each of the pathway contains its own Blocks and Fusion layers across different pathways.

```
Pathway 1  ... Pathway N
    ↓              ↓
 Block 1        Block N
    ↓ --Fusion----↓
```

__init__(*, *multipathway_blocks*, *multipathway_fusion*, *inplace=True*)

Parameters

- **multipathway_blocks** (*nn.module_list*) – list of models from all pathways.
- **multipathway_fusion** (*nn.module*) – fusion model.
- **inplace** (*bool*) – If inplace, directly update the input list without making a copy.

Return type None

3.3 pytorchvideo.models.head

class pytorchvideo.models.head.**SequencePool**(*mode*)
Sequence pool produces a single embedding from a sequence of embeddings. Currently it supports “mean” and “cls”.

__init__(*mode*)

Parameters **mode** (*str*) – Optionals include “cls” and “mean”. If set to “cls”, it assumes the first element in the input is the cls token and returns it. If set to “mean”, it returns the mean of the entire sequence.

Return type None

pytorchvideo.models.head.**create_res_basic_head**(*, *in_features*, *out_features*, *pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *output_size=(1, 1, 1)*, *pool_kernel_size=(1, 7, 7)*, *pool_stride=(1, 1, 1)*, *pool_padding=(0, 0, 0)*, *dropout_rate=0.5*, *activation=None*, *output_with_global_average=True*)

Creates ResNet basic head. This layer performs an optional pooling operation followed by an optional dropout, a fully-connected projection, an activation layer and a global spatiotemporal averaging.

```
 Pooling
    ↓
 Dropout
    ↓
Projection
    ↓
Activation
    ↓
Averaging
```

Activation examples include: ReLU, Softmax, Sigmoid, and None. Pool3d examples include: AvgPool3d, MaxPool3d, AdaptiveAvgPool3d, and None.

Parameters

- **in_features** (*int*) – input channel size of the resnet head.
- **out_features** (*int*) – output channel size of the resnet head.
- **pool** (*callable*) – a callable that constructs resnet head pooling layer, examples include: nn.AvgPool3d, nn.MaxPool3d, nn.AdaptiveAvgPool3d, and None (not applying pooling).
- **pool_kernel_size** (*tuple*) – pooling kernel size(s) when not using adaptive pooling.
- **pool_stride** (*tuple*) – pooling stride size(s) when not using adaptive pooling.
- **pool_padding** (*tuple*) – pooling padding size(s) when not using adaptive pooling.
- **output_size** (*tuple*) – spatial temporal output size when using adaptive pooling.
- **activation** (*callable*) – a callable that constructs resnet head activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not applying activation).
- **dropout_rate** (*float*) – dropout rate.
- **output_with_global_average** (*bool*) – if True, perform global averaging on temporal and spatial dimensions and reshape output to batch_size x out_features.

Return type torch.nn.modules.module.Module

pytorchvideo.models.head.**create_vit_basic_head**(*, *in_features*, *out_features*, *seq_pool_type='cls'*, *dropout_rate=0.5*, *activation=None*)

Creates vision transformer basic head.

```
 Pooling
    ↓
 Dropout
    ↓
Projection
```

(continues on next page)

(continued from previous page)

```
    ↓
Activation
```

Activation examples include: ReLU, Softmax, Sigmoid, and None. Pool type examples include: cls, mean and none.

Parameters

- **in_features** (*int*) – input channel size of the resnet head.
- **out_features** (*int*) – output channel size of the resnet head.
- **pool_type** (*str*) – Pooling type. It supports "cls", "mean " and "none". If set to "cls", it assumes the first element in the input is the cls token and returns it. If set to "mean", it returns the mean of the entire sequence.
- **activation** (*callable*) – a callable that constructs vision transformer head activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not applying activation).
- **dropout_rate** (*float*) – dropout rate.
- **seq_pool_type** (*str*) –

Return type torch.nn.modules.module.Module

pytorchvideo.models.head.**create_res_roi_pooling_head**(*, *in_features*, *out_features*, *resolution*, *spatial_scale*, *sampling_ratio=0*, *roi=<class 'torchvision.ops.roi_align.RoIAlign'>*, *pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *output_size=(1, 1, 1)*, *pool_kernel_size=(1, 7, 7)*, *pool_stride=(1, 1, 1)*, *pool_padding=(0, 0, 0)*, *pool_spatial=<class 'torch.nn.modules.pooling.MaxPool2d'>*, *dropout_rate=0.5*, *activation=None*, *output_with_global_average=True*)

Creates ResNet RoI head. This layer performs an optional pooling operation followed by an RoI projection, an optional 2D spatial pool, an optional dropout, a fully-connected projection, an activation layer and a global spatiotemporal averaging.

Pool3d ↓

RoI Align

↓

Pool2d ↓

Dropout ↓

Projection ↓

Activation ↓

Averaging

Activation examples include: ReLU, Softmax, Sigmoid, and None. Pool3d examples include: AvgPool3d, MaxPool3d, AdaptiveAvgPool3d, and None. RoI examples include: detectron2.layers.ROIAlign, detectron2.layers.ROIAlignRotated,

tochvision.ops.RoIAlign and None

Pool2d examples include: MaxPool2e, AvgPool2d, and None.

Parameters

- **related configs** (*Output*) – in_features: input channel size of the resnet head. out_features: output channel size of the resnet head.
- **layer related configs** (*RoI*) – resolution (tuple): h, w sizes of the RoI interpolation. spatial_scale (float): scale the input boxes by this number sampling_ratio (int): number of inputs samples to take for each output

  sample interpolation. 0 to take samples densely.

  roi (callable): a callable that constructs the roi interpolation layer, examples include detectron2.layers.ROIAlign, detectron2.layers.ROIAlignRotated, and None.

- **related configs** –

  pool (callable): a callable that constructs resnet head pooling layer, examples include: nn.AvgPool3d, nn.MaxPool3d, nn.AdaptiveAvgPool3d, and None (not applying pooling).

  pool_kernel_size (tuple): pooling kernel size(s) when not using adaptive pooling.

  pool_stride (tuple): pooling stride size(s) when not using adaptive pooling. pool_padding (tuple): pooling padding size(s) when not using adaptive

  pooling.

  output_size (tuple): spatial temporal output size when using adaptive pooling.

  pool_spatial (callable): a callable that constructs the 2d pooling layer which follows the RoI layer, examples include: nn.AvgPool2d, nn.MaxPool2d, and None (not applying spatial pooling).

- **related configs** –

  activation (callable): a callable that constructs resnet head activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not applying activation).

- **related configs** – dropout_rate (float): dropout rate.
- **related configs** –

  output_with_global_average (bool): if True, perform global averaging on temporal and spatial dimensions and reshape output to batch_size x out_features.

- **in_features** (*int*) –
- **out_features** (*int*) –
- **resolution** (*Tuple*) –
- **spatial_scale** (*float*) –
- **sampling_ratio** (*int*) –
- **roi** (*Callable*) –

- **pool** (*Callable*) –
- **output_size** (*Tuple[int]*) –
- **pool_kernel_size** (*Tuple[int]*) –
- **pool_stride** (*Tuple[int]*) –
- **pool_padding** (*Tuple[int]*) –
- **pool_spatial** (*Callable*) –
- **dropout_rate** (*float*) –
- **activation** (*Callable*) –
- **output_with_global_average** (*bool*) –

Return type torch.nn.modules.module.Module

class pytorchvideo.models.head.**ResNetBasicHead**(*pool=None*, *dropout=None*, *proj=None*, *activation=None*, *output_pool=None*)

ResNet basic head. This layer performs an optional pooling operation followed by an optional dropout, a fully-connected projection, an optional activation layer and a global spatiotemporal averaging.

```
 Pool3d
    ↓
 Dropout
    ↓
Projection
    ↓
Activation
    ↓
Averaging
```

The builder can be found in *create_res_basic_head*.

__init__(*pool=None*, *dropout=None*, *proj=None*, *activation=None*, *output_pool=None*)

Parameters

- **pool** (*torch.nn.modules*) – pooling module.
- **dropout** (*torch.nn.modules*) – dropout module.
- **proj** (*torch.nn.modules*) – project module.
- **activation** (*torch.nn.modules*) – activation module.
- **output_pool** (*torch.nn.Module*) – pooling module for output.

Return type None

class pytorchvideo.models.head.**ResNetRoIHead**(*pool=None*, *pool_spatial=None*, *roi_layer=None*, *dropout=None*, *proj=None*, *activation=None*, *output_pool=None*)

ResNet RoI head. This layer performs an optional pooling operation followed by an RoI projection, an optional 2D spatial pool, an optional dropout, a fully-connected projection, an activation layer and a global spatiotemporal averaging.

Pool3d ↓

RoI Align

↓

Pool2d ↓

Dropout ↓

Projection ↓

Activation ↓

Averaging

The builder can be found in *create_res_roi_pooling_head*.

__init__(*pool=None*, *pool_spatial=None*, *roi_layer=None*, *dropout=None*, *proj=None*, *activation=None*, *output_pool=None*)

Parameters

- **pool** (*torch.nn.modules*) – pooling module.
- **pool_spatial** (*torch.nn.modules*) – pooling module.
- **roi_spatial** (*torch.nn.modules*) – RoI (Ex: Align, pool) module.
- **dropout** (*torch.nn.modules*) – dropout module.
- **proj** (*torch.nn.modules*) – project module.
- **activation** (*torch.nn.modules*) – activation module.
- **output_pool** (*torch.nn.Module*) – pooling module for output.
- **roi_layer** (*torch.nn.modules.module.Module*) –

Return type None

forward(*x*, *bboxes*)

Parameters

- **x** (*torch.tensor*) – input tensor
- **bboxes** (*torch.tensor*) – Accociated bounding boxes. The format is N*5 (Index, X_1,Y_1,X_2,Y_2) if using RoIAlign and N*6 (Index, x_ctr, y_ctr, width, height, angle_degrees) if using RoIAlignRotated.

Return type torch.Tensor

class pytorchvideo.models.head.**VisionTransformerBasicHead**(*sequence_pool=None*, *dropout=None*, *proj=None*, *activation=None*)

Vision transformer basic head.

```
SequencePool
     ↓
  Dropout
     ↓
 Projection
     ↓
 Activation
```

The builder can be found in *create_vit_basic_head*.

__init__(*sequence_pool=None*, *dropout=None*, *proj=None*, *activation=None*)

Parameters

- **sequence_pool** (*torch.nn.modules*) – pooling module.
- **dropout** (*torch.nn.modules*) – dropout module.
- **proj** (*torch.nn.modules*) – project module.
- **activation** (*torch.nn.modules*) – activation module.

Return type None

3.4 pytorchvideo.models.stem

pytorchvideo.models.stem.**create_res_basic_stem**(*, *in_channels*, *out_channels*, *conv_kernel_size=(3, 7, 7)*, *conv_stride=(1, 2, 2)*, *conv_padding=(1, 3, 3)*, *conv_bias=False*, *conv=<class 'torch.nn.modules.conv.Conv3d'>*, *pool=<class 'torch.nn.modules.pooling.MaxPool3d'>*, *pool_kernel_size=(1, 3, 3)*, *pool_stride=(1, 2, 2)*, *pool_padding=(0, 1, 1)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Creates the basic resnet stem layer. It performs spatiotemporal Convolution, BN, and Relu following by a spatiotemporal pooling.

```
   Conv3d
     ↓
Normalization
     ↓
  Activation
     ↓
   Pool3d
```

Normalization options include: BatchNorm3d and None (no normalization). Activation options include: ReLU, Softmax, Sigmoid, and None (no activation). Pool3d options include: AvgPool3d, MaxPool3d, and None (no pooling).

Parameters

- **in_channels** (*int*) – input channel size of the convolution.
- **out_channels** (*int*) – output channel size of the convolution.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s).
- **conv_stride** (*tuple*) – convolutional stride size(s).
- **conv_padding** (*tuple*) – convolutional padding size(s).
- **conv_bias** (*bool*) – convolutional bias. If true, adds a learnable bias to the output.
- **conv** (*callable*) – Callable used to build the convolution layer.
- **pool** (*callable*) – a callable that constructs pooling layer, options include: nn.AvgPool3d, nn.MaxPool3d, and None (not performing pooling).

- **pool_kernel_size** (*tuple*) – pooling kernel size(s).
- **pool_stride** (*tuple*) – pooling stride size(s).
- **pool_padding** (*tuple*) – pooling padding size(s).
- **norm** (*callable*) – a callable that constructs normalization layer, options include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, options include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet basic stem layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.stem.**create_acoustic_res_basic_stem**(*, *in_channels*, *out_channels*, *conv_kernel_size=(3, 7, 7)*, *conv_stride=(1, 1, 1)*, *conv_padding=(1, 3, 3)*, *conv_bias=False*, *pool=<class 'torch.nn.modules.pooling.MaxPool3d'>*, *pool_kernel_size=(1, 3, 3)*, *pool_stride=(1, 2, 2)*, *pool_padding=(0, 1, 1)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Creates the acoustic resnet stem layer. It performs a spatial and a temporal Convolution in parallel, then performs, BN, and Relu following by a spatiotemporal pooling.

```
Conv3d   Conv3d
       ↓
 Normalization
       ↓
   Activation
       ↓
    Pool3d
```

Normalization options include: BatchNorm3d and None (no normalization). Activation options include: ReLU, Softmax, Sigmoid, and None (no activation). Pool3d options include: AvgPool3d, MaxPool3d, and None (no pooling).

Parameters

- **in_channels** (*int*) – input channel size of the convolution.
- **out_channels** (*int*) – output channel size of the convolution.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s).
- **conv_stride** (*tuple*) – convolutional stride size(s), it will be performed as temporal and spatial convolution in parallel.

- **conv_padding** (*tuple*) – convolutional padding size(s), it will be performed as temporal and spatial convolution in parallel.
- **conv_bias** (*bool*) – convolutional bias. If true, adds a learnable bias to the output.
- **pool** (*callable*) – a callable that constructs pooling layer, options include: nn.AvgPool3d, nn.MaxPool3d, and None (not performing pooling).
- **pool_kernel_size** (*tuple*) – pooling kernel size(s).
- **pool_stride** (*tuple*) – pooling stride size(s).
- **pool_padding** (*tuple*) – pooling padding size(s).
- **norm** (*callable*) – a callable that constructs normalization layer, options include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, options include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet basic stem layer.

Return type torch.nn.modules.module.Module

class pytorchvideo.models.stem.**ResNetBasicStem**(*, *conv=None*, *norm=None*, *activation=None*, *pool=None*)

ResNet basic 3D stem module. Performs spatiotemporal Convolution, BN, and activation following by a spatiotemporal pooling.

```
   Conv3d
     ↓
Normalization
     ↓
  Activation
     ↓
   Pool3d
```

The builder can be found in *create_res_basic_stem*.

__init__(*, *conv=None*, *norm=None*, *activation=None*, *pool=None*)

Parameters

- **conv** (*torch.nn.modules*) – convolutional module.
- **norm** (*torch.nn.modules*) – normalization module.
- **activation** (*torch.nn.modules*) – activation module.
- **pool** (*torch.nn.modules*) – pooling module.

Return type None

class pytorchvideo.models.stem.**PatchEmbed**(*, *patch_model=None*)

Transformer basic patch embedding module. Performs patchifying input, flatten and and transpose.

```
PatchModel
    ↓
 flatten
    ↓
transpose
```

The builder can be found in *create_patch_embed*.

pytorchvideo.models.stem.**create_conv_patch_embed**(*, *in_channels*, *out_channels*, *conv_kernel_size=(1, 16, 16)*, *conv_stride=(1, 4, 4)*, *conv_padding=(1, 7, 7)*, *conv_bias=True*, *conv=<class 'torch.nn.modules.conv.Conv3d'>*)

Creates the transformer basic patch embedding. It performs Convolution, flatten and transpose.

```
 Conv3d
    ↓
 flatten
    ↓
transpose
```

Parameters

- **in_channels** (*int*) – input channel size of the convolution.
- **out_channels** (*int*) – output channel size of the convolution.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s).
- **conv_stride** (*tuple*) – convolutional stride size(s).
- **conv_padding** (*tuple*) – convolutional padding size(s).
- **conv_bias** (*bool*) – convolutional bias. If true, adds a learnable bias to the output.
- **conv** (*callable*) – Callable used to build the convolution layer.

Returns *(nn.Module)* – transformer patch embedding layer.

Return type torch.nn.modules.module.Module

3.5 pytorchvideo.models.csn

pytorchvideo.models.csn.**create_csn**(*, *input_channel=3*, *model_depth=50*, *model_num_class=400*, *dropout_rate=0*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_dim_out=64*, *stem_conv_kernel_size=(3, 7, 7)*, *stem_conv_stride=(1, 2, 2)*, *stem_pool=None*, *stem_pool_kernel_size=(1, 3, 3)*, *stem_pool_stride=(1, 2, 2)*, *stage_conv_a_kernel_size=(1, 1, 1)*, *stage_conv_b_kernel_size=(3, 3, 3)*, *stage_conv_b_width_per_group=1*, *stage_spatial_stride=(1, 2, 2, 2)*, *stage_temporal_stride=(1, 2, 2, 2)*, *bottleneck=<function create_bottleneck_block>*, *bottleneck_ratio=4*, *head_pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *head_pool_kernel_size=(1, 7, 7)*, *head_output_size=(1, 1, 1)*, *head_activation=None*, *head_output_with_global_average=True*)

Build Channel-Separated Convolutional Networks (CSN): Video classification with channel-separated convolutional networks. Du Tran, Heng Wang, Lorenzo Torresani, Matt Feiszli. ICCV 2019.

CSN follows the ResNet style architecture including three parts: Stem, Stages and Head. The three parts are assembled in the following order:

```
Input
  ↓
Stem
  ↓
Stage 1
  ↓
  .
  .
  .
  ↓
Stage N
  ↓
Head
```

CSN uses depthwise convolution. To further reduce the computational cost, it uses low resolution (112x112), short clips (4 frames), different striding and kernel size, etc.

Parameters

- **input_channel** (*int*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet. Options include: 50, 101, 152. model_num_class (int): the number of classes for the video dataset. dropout_rate (float): dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_dim_out** (*int*) – output channel size to stem.
- **stem_conv_kernel_size** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_stride** (*tuple*) – convolutional stride size(s) of stem.
- **stem_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **stem_pool_kernel_size** (*tuple*) – pooling kernel size(s).
- **stem_pool_stride** (*tuple*) – pooling stride size(s).
- **stage_conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_width_per_group** (*int*) – the width of each group for conv_b. Set it to 1 for depthwise convolution.
- **stage_spatial_stride** (*tuple*) – the spatial stride for each stage.
- **stage_temporal_stride** (*tuple*) – the temporal stride for each stage.
- **bottleneck** (*callable*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block.
- **bottleneck_ratio** (*int*) – the ratio between inner and outer dimensions for the bottleneck block.
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_pool_kernel_size** (*tuple*) – the pooling kernel size.
- **head_output_size** (*tuple*) – the size of output tensor for head.

- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **model_num_class** (*int*) –
- **dropout_rate** (*float*) –

Returns *(nn.Module)* – the csn model.

Return type torch.nn.modules.module.Module

3.6 pytorchvideo.models.x3d

pytorchvideo.models.x3d.**create_x3d_stem**(*, *in_channels*, *out_channels*, *conv_kernel_size=(5, 3, 3)*, *conv_stride=(1, 2, 2)*, *conv_padding=(2, 1, 1)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Creates the stem layer for X3D. It performs spatial Conv, temporal Conv, BN, and Relu.

```
   Conv_xy
      ↓
   Conv_t
      ↓
Normalization
      ↓
  Activation
```

Parameters

- **in_channels** (*int*) – input channel size of the convolution.
- **out_channels** (*int*) – output channel size of the convolution.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s).
- **conv_stride** (*tuple*) – convolutional stride size(s).
- **conv_padding** (*tuple*) – convolutional padding size(s).
- **norm** (*callable*) – a callable that constructs normalization layer, options include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, options include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – X3D stem layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.x3d.**create_x3d_bottleneck_block**(*, *dim_in*, *dim_inner*, *dim_out*, *conv_kernel_size=(3, 3, 3)*, *conv_stride=(1, 2, 2)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *se_ratio=0.0625*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *inner_act=<class 'pytorchvideo.layers.swish.Swish'>*)

Bottleneck block for X3D: a sequence of Conv, Normalization with optional SE block, and Activations repeated in the following order:

```
   Conv3d (conv_a)
          ↓
Normalization (norm_a)
          ↓
  Activation (act_a)
          ↓
   Conv3d (conv_b)
          ↓
Normalization (norm_b)
          ↓
Squeeze-and-Excitation
          ↓
  Activation (act_b)
          ↓
   Conv3d (conv_c)
          ↓
Normalization (norm_c)
```

Parameters

- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **se_ratio** (*float*) – if > 0, apply SE to the 3x3x3 conv, with the SE channel dimensionality being se_ratio times the 3x3x3 conv dim.
- **activation** (*callable*) – a callable that constructs activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).
- **inner_act** (*callable*) – whether use Swish activation for act_b or not.

Returns *(nn.Module)* – X3D bottleneck block.

Return type torch.nn.modules.module.Module

pytorchvideo.models.x3d.**create_x3d_res_block**(*, *dim_in*, *dim_inner*, *dim_out*, *bottleneck=<function create_x3d_bottleneck_block>*, *use_shortcut=True*, *conv_kernel_size=(3, 3, 3)*, *conv_stride=(1, 2, 2)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *se_ratio=0.0625*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *inner_act=<class 'pytorchvideo.layers.swish.Swish'>*)

Residual block for X3D. Performs a summation between an identity shortcut in branch1 and a main block in branch2. When the input and output dimensions are different, a convolution followed by a normalization will be performed.

```
  Input
    |-------+
    ↓       |
  Block     |
    ↓       |
Summation ←-+
    ↓
Activation
```

Parameters

- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **bottleneck** (*callable*) – a callable for create_x3d_bottleneck_block.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **se_ratio** (*float*) – if > 0, apply SE to the 3x3x3 conv, with the SE channel dimensionality being se_ratio times the 3x3x3 conv dim.
- **activation** (*callable*) – a callable that constructs activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).
- **inner_act** (*callable*) – whether use Swish activation for act_b or not.
- **use_shortcut** (*bool*) –

Returns *(nn.Module)* – X3D block layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.x3d.**create_x3d_res_stage**(*, *depth*, *dim_in*, *dim_inner*, *dim_out*, *bottleneck=<function create_x3d_bottleneck_block>*, *conv_kernel_size=(3, 3, 3)*, *conv_stride=(1, 2, 2)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *se_ratio=0.0625*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *inner_act=<class 'pytorchvideo.layers.swish.Swish'>*)

Create Residual Stage, which composes sequential blocks that make up X3D.

```
 Input
    ↓
ResBlock
    ↓
    .
    .
    .
    ↓
ResBlock
```

Parameters

- **depth** (*int*) – number of blocks to create.
- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **bottleneck** (*callable*) – a callable for create_x3d_bottleneck_block.
- **conv_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **se_ratio** (*float*) – if > 0, apply SE to the 3x3x3 conv, with the SE channel dimensionality being se_ratio times the 3x3x3 conv dim.
- **activation** (*callable*) – a callable that constructs activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).
- **inner_act** (*callable*) – whether use Swish activation for act_b or not.

Returns *(nn.Module)* – X3D stage layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.x3d.**create_x3d_head**(*, *dim_in*, *dim_inner*, *dim_out*, *num_classes*, *pool_act=<class 'torch.nn.modules.activation.ReLU'>*, *pool_kernel_size=(13, 5, 5)*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *bn_lin5_on=False*, *dropout_rate=0.5*, *activation=<class 'torch.nn.modules.activation.Softmax'>*, *output_with_global_average=True*)

Creates X3D head. This layer performs an projected pooling operation followed by an dropout, a fully-connected projection, an activation layer and a global spatiotemporal averaging.

```
ProjectedPool
     ↓
   Dropout
     ↓
  Projection
     ↓
  Activation
     ↓
  Averaging
```

Parameters

- **dim_in** (*int*) – input channel size of the X3D head.
- **dim_inner** (*int*) – intermediate channel size of the X3D head.
- **dim_out** (*int*) – output channel size of the X3D head.
- **num_classes** (*int*) – the number of classes for the video dataset.
- **pool_act** (*callable*) – a callable that constructs resnet pool activation layer such as nn.ReLU.
- **pool_kernel_size** (*tuple*) – pooling kernel size(s) when not using adaptive pooling.
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **bn_lin5_on** (*bool*) – if True, perform normalization on the features before the classifier.
- **dropout_rate** (*float*) – dropout rate.
- **activation** (*callable*) – a callable that constructs resnet head activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not applying activation).
- **output_with_global_average** (*bool*) – if True, perform global averaging on temporal and spatial dimensions and reshape output to batch_size x out_features.

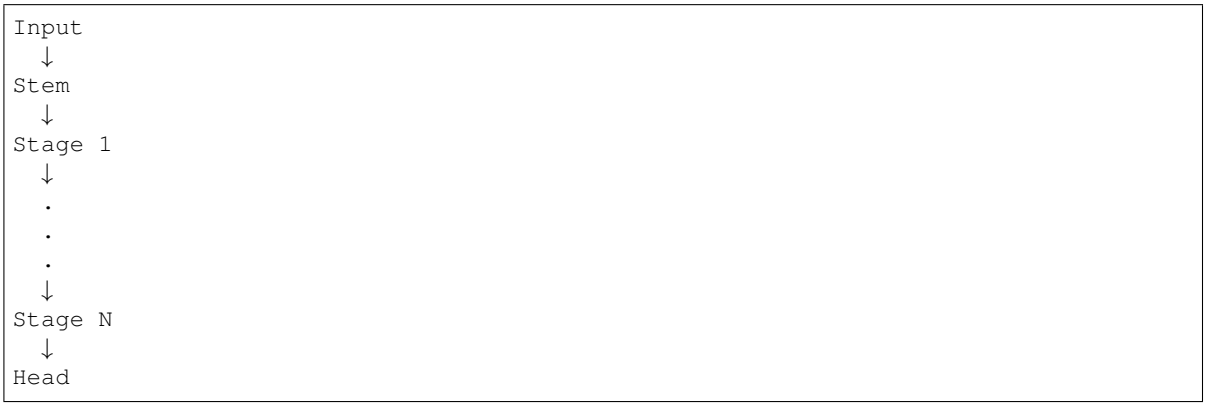
Returns *(nn.Module)* – X3D head layer.

Return type torch.nn.modules.module.Module

pytorchvideo.models.x3d.**create_x3d**(*, *input_channel=3*, *input_clip_length=13*, *input_crop_size=160*, *model_num_class=400*, *dropout_rate=0.5*, *width_factor=2.0*, *depth_factor=2.2*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_dim_in=12*, *stem_conv_kernel_size=(5, 3, 3)*, *stem_conv_stride=(1, 2, 2)*, *stage_conv_kernel_size=((3, 3, 3), (3, 3, 3), (3, 3, 3), (3, 3, 3))*, *stage_spatial_stride=(2, 2, 2, 2)*, *stage_temporal_stride=(1, 1, 1, 1)*, *bottleneck=<function create_x3d_bottleneck_block>*, *bottleneck_factor=2.25*, *se_ratio=0.0625*, *inner_act=<class 'pytorchvideo.layers.swish.Swish'>*, *head_dim_out=2048*, *head_pool_act=<class 'torch.nn.modules.activation.ReLU'>*, *head_bn_lin5_on=False*, *head_activation=<class 'torch.nn.modules.activation.Softmax'>*, *head_output_with_global_average=True*)

X3D model builder. It builds a X3D network backbone, which is a ResNet.

Christoph Feichtenhofer. "X3D: Expanding Architectures for Efficient Video Recognition." https://arxiv.org/abs/2004.04730



Parameters

- **input_channel** (*int*) – number of channels for the input video clip.
- **input_clip_length** (*int*) – length of the input video clip. Value for different models: X3D-XS: 4; X3D-S: 13; X3D-M: 16; X3D-L: 16.
- **input_crop_size** (*int*) – spatial resolution of the input video clip. Value for different models: X3D-XS: 160; X3D-S: 160; X3D-M: 224; X3D-L: 312.
- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **width_factor** (*float*) – width expansion factor.
- **depth_factor** (*float*) – depth expansion factor. Value for different models: X3D-XS: 2.2; X3D-S: 2.2; X3D-M: 2.2; X3D-L: 5.0.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **norm_eps** (*float*) – normalization epsilon.

- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_dim_in** (*int*) – input channel size for stem before expansion.
- **stem_conv_kernel_size** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_stride** (*tuple*) – convolutional stride size(s) of stem.
- **stage_conv_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_spatial_stride** (*tuple*) – the spatial stride for each stage.
- **stage_temporal_stride** (*tuple*) – the temporal stride for each stage.
- **bottleneck_factor** (*float*) – bottleneck expansion factor for the 3x3x3 conv.
- **se_ratio** (*float*) – if > 0, apply SE to the 3x3x3 conv, with the SE channel dimensionality being se_ratio times the 3x3x3 conv dim.
- **inner_act** (*callable*) – whether use Swish activation for act_b or not.
- **head_dim_out** (*int*) – output channel size of the X3D head.
- **head_pool_act** (*callable*) – a callable that constructs resnet pool activation layer such as nn.ReLU.
- **head_bn_lin5_on** (*bool*) – if True, perform normalization on the features before the classifier.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **bottleneck** (*Callable*) –

Returns *(nn.Module)* – the X3D network.

Return type torch.nn.modules.module.Module

class pytorchvideo.models.x3d.**ProjectedPool**(*, *pre_conv=None*, *pre_norm=None*, *pre_act=None*, *pool=None*, *post_conv=None*, *post_norm=None*, *post_act=None*)

A pooling module augmented with Conv, Normalization and Activation both before and after pooling for the head layer of X3D.

```
   Conv3d (pre_conv)
          ↓
Normalization (pre_norm)
          ↓
  Activation (pre_act)
          ↓
       Pool3d
          ↓
   Conv3d (post_conv)
          ↓
Normalization (post_norm)
          ↓
  Activation (post_act)
```

__init__(*, *pre_conv=None*, *pre_norm=None*, *pre_act=None*, *pool=None*, *post_conv=None*, *post_norm=None*, *post_act=None*)

Parameters

- **pre_conv** (*torch.nn.modules*) – convolutional module.
- **pre_norm** (*torch.nn.modules*) – normalization module.
- **pre_act** (*torch.nn.modules*) – activation module.
- **pool** (*torch.nn.modules*) – pooling module.
- **post_conv** (*torch.nn.modules*) – convolutional module.
- **post_norm** (*torch.nn.modules*) – normalization module.
- **post_act** (*torch.nn.modules*) – activation module.

Return type None

3.7 pytorchvideo.models.slowfast

pytorchvideo.models.slowfast.**create_slowfast**(*, *slowfast_channel_reduction_ratio=(8,), slowfast_conv_channel_fusion_ratio=2, slowfast_fusion_conv_kernel_size=(7, 1, 1), slowfast_fusion_conv_stride=(4, 1, 1), fusion_builder=None, input_channels=(3, 3), model_depth=50, model_num_class=400, dropout_rate=0.5, norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>, activation=<class 'torch.nn.modules.activation.ReLU'>, stem_function=(<function create_res_basic_stem>, <function create_res_basic_stem>), stem_dim_outs=(64, 8), stem_conv_kernel_sizes=((1, 7, 7), (5, 7, 7)), stem_conv_strides=((1, 2, 2), (1, 2, 2)), stem_pool=(<class 'torch.nn.modules.pooling.MaxPool3d'>, <class 'torch.nn.modules.pooling.MaxPool3d'>), stem_pool_kernel_sizes=((1, 3, 3), (1, 3, 3)), stem_pool_strides=((1, 2, 2), (1, 2, 2)), stage_conv_a_kernel_sizes=(((1, 1, 1), (1, 1, 1), (3, 1, 1), (3, 1, 1)), ((3, 1, 1), (3, 1, 1), (3, 1, 1), (3, 1, 1))), stage_conv_b_kernel_sizes=(((1, 3, 3), (1, 3, 3), (1, 3, 3), (1, 3, 3)), ((1, 3, 3), (1, 3, 3), (1, 3, 3), (1, 3, 3))), stage_conv_b_num_groups=((1, 1, 1, 1), (1, 1, 1, 1)), stage_conv_b_dilations=(((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 1, 1)), ((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 1, 1))), stage_spatial_strides=((1, 2, 2, 2), (1, 2, 2, 2)), stage_temporal_strides=((1, 1, 1, 1), (1, 1, 1, 1)), bottleneck=((<function create_bottleneck_block>, <function create_bottleneck_block>, <function create_bottleneck_block>, <function create_bottleneck_block>), (<function create_bottleneck_block>, <function create_bottleneck_block>, <function create_bottleneck_block>, <function create_bottleneck_block>)), head=<function create_res_basic_head>, head_pool=<class 'torch.nn.modules.pooling.AvgPool3d'>, head_pool_kernel_sizes=((8, 7, 7), (32, 7, 7)), head_output_size=(1, 1, 1), head_activation=None, head_output_with_global_average=True*)

Build SlowFast model for video recognition, SlowFast model involves a Slow pathway, operating at low frame

rate, to capture spatial semantics, and a Fast pathway, operating at high frame rate, to capture motion at fine temporal resolution. The Fast pathway can be made very lightweight by reducing its channel capacity, yet can learn useful temporal information for video recognition. Details can be found from the paper:

Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast networks for video recognition." https://arxiv.org/pdf/1812.03982.pdf

```
Slow Input  Fast Input
     ↓           ↓
    Stem        Stem
     ↓  Fusion- ↓
  Stage 1     Stage 1
     ↓  Fusion- ↓
     .           .
     ↓           ↓
  Stage N     Stage N
     ↓  Fusion- ↓
           ↓
         Head
```

Parameters

- **slowfast_channel_reduction_ratio** (*int*) – Corresponds to the inverse of the channel reduction ratio, eta between the Slow and Fast pathways.
- **slowfast_conv_channel_fusion_ratio** (*int*) – Ratio of channel dimensions between the Slow and Fast pathways.
- **slowfast_fusion_conv_kernel_size** (*DEPRECATED*) – the convolutional kernel size used for fusion.
- **slowfast_fusion_conv_stride** (*DEPRECATED*) – the convolutional stride size used for fusion.
- **fusion_builder** (*Callable[[int, int], nn.Module]*) – Builder function for generating the fusion modules based on stage dimension and index
- **input_channels** (*tuple*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet.
- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_function** (*Tuple[Callable]*) – a callable that constructs stem layer. Examples include create_res_basic_stem. Indexed by pathway
- **stem_dim_outs** (*tuple*) – output channel size to stem.
- **stem_conv_kernel_sizes** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_strides** (*tuple*) – convolutional stride size(s) of stem.
- **stem_pool** (*Tuple[Callable]*) – a callable that constructs resnet head pooling layer. Indexed by pathway
- **stem_pool_kernel_sizes** (*tuple*) – pooling kernel size(s).
- **stem_pool_strides** (*tuple*) – pooling stride size(s).

- **stage_conv_a_kernel_sizes** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_sizes** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_num_groups** (*tuple*) – number of groups for groupwise convolution for conv_b. 1 for ResNet, and larger than 1 for ResNeXt.
- **stage_conv_b_dilations** (*tuple*) – dilation for 3D convolution for conv_b.
- **stage_spatial_strides** (*tuple*) – the spatial stride for each stage.
- **stage_temporal_strides** (*tuple*) – the temporal stride for each stage.
- **bottleneck** (*Tuple[Tuple[Callable]]*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block. Indexed by pathway and stage index
- **head** (*callable*) – a callable that constructs the resnet-style head. Ex: create_res_basic_head
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_output_sizes** (*tuple*) – the size of output tensor for head.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **head_pool_kernel_sizes** (*Tuple[Tuple[int]]*) –
- **head_output_size** (*Tuple[int]*) –

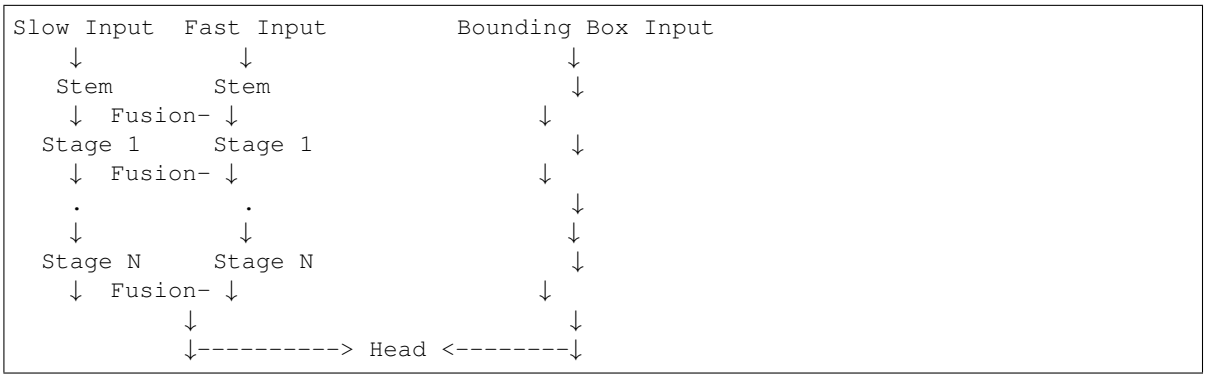
Returns *(nn.Module)* – SlowFast model.

Return type torch.nn.modules.module.Module

pytorchvideo.models.slowfast.**create_slowfast_with_roi_head**(*, *slowfast_channel_reduction_ratio=(8,)*, *slowfast_conv_channel_fusion_ratio=2*, *slowfast_fusion_conv_kernel_size=(7, 1, 1)*, *slowfast_fusion_conv_stride=(4, 1, 1)*, *fusion_builder=None*, *input_channels=(3, 3)*, *model_depth=50*, *model_num_class=80*, *dropout_rate=0.5*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_function=(<function create_res_basic_stem>, <function create_res_basic_stem>)*, *stem_dim_outs=(64, 8)*, *stem_conv_kernel_sizes=((1, 7, 7), (5, 7, 7))*, *stem_conv_strides=((1, 2, 2), (1, 2, 2))*, *stem_pool=(<class 'torch.nn.modules.pooling.MaxPool3d'>, <class 'torch.nn.modules.pooling.MaxPool3d'>)*, *stem_pool_kernel_sizes=((1, 3, 3), (1, 3, 3))*, *stem_pool_strides=((1, 2, 2), (1, 2, 2))*, *stage_conv_a_kernel_sizes=(((1, 1, 1), (1, 1, 1), (3, 1, 1), (3, 1, 1)), ((3, 1, 1), (3, 1, 1), (3, 1, 1), (3, 1, 1)))*, *stage_conv_b_kernel_sizes=(((1, 3, 3), (1, 3, 3), (1, 3, 3), (1, 3, 3)), ((1, 3, 3), (1, 3, 3), (1, 3, 3), (1, 3, 3)))*, *stage_conv_b_num_groups=((1, 1, 1, 1), (1, 1, 1, 1))*, *stage_conv_b_dilations=(((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 2, 2)), ((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 2, 2)))*, *stage_spatial_strides=((1, 2, 2, 1), (1, 2, 2, 1))*, *stage_temporal_strides=((1, 1, 1, 1), (1, 1, 1, 1))*, *bottleneck=((<function cre-*

Build SlowFast model for video detection, SlowFast model involves a Slow pathway, operating at low frame rate, to capture spatial semantics, and a Fast pathway, operating at high frame rate, to capture motion at fine temporal resolution. The Fast pathway can be made very lightweight by reducing its channel capacity, yet can learn useful temporal information for video recognition. Details can be found from the paper:

Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast networks for video recognition." https://arxiv.org/pdf/1812.03982.pdf



Parameters

- **slowfast_channel_reduction_ratio** (*int*) – Corresponds to the inverse of the channel reduction ratio, eta between the Slow and Fast pathways.
- **slowfast_conv_channel_fusion_ratio** (*int*) – Ratio of channel dimensions between the Slow and Fast pathways.
- **slowfast_fusion_conv_kernel_size** (*DEPRECATED*) – the convolutional kernel size used for fusion.
- **slowfast_fusion_conv_stride** (*DEPRECATED*) – the convolutional stride size used for fusion.
- **fusion_builder** (*Callable[[int, int], nn.Module]*) – Builder function for generating the fusion modules based on stage dimension and index
- **input_channels** (*tuple*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet.
- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_function** (*Tuple[Callable]*) – a callable that constructs stem layer. Examples include create_res_basic_stem. Indexed by pathway
- **stem_dim_outs** (*tuple*) – output channel size to stem.
- **stem_conv_kernel_sizes** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_strides** (*tuple*) – convolutional stride size(s) of stem.
- **stem_pool** (*Tuple[Callable]*) – a callable that constructs resnet head pooling layer. Indexed by pathway
- **stem_pool_kernel_sizes** (*tuple*) – pooling kernel size(s).

- **stem_pool_strides** (*tuple*) – pooling stride size(s).
- **stage_conv_a_kernel_sizes** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_sizes** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_num_groups** (*tuple*) – number of groups for groupwise convolution for conv_b. 1 for ResNet, and larger than 1 for ResNeXt.
- **stage_conv_b_dilations** (*tuple*) – dilation for 3D convolution for conv_b.
- **stage_spatial_strides** (*tuple*) – the spatial stride for each stage.
- **stage_temporal_strides** (*tuple*) – the temporal stride for each stage.
- **bottleneck** (*Tuple[Tuple[Callable]]*) – a callable that constructs bottleneck block layer. Examples include: create_bottleneck_block. Indexed by pathway and stage index
- **head** (*callable*) – a a callable that constructs the detection head which can take in the additional input of bounding boxes. Ex: create_res_roi_pooling_head
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_output_sizes** (*tuple*) – the size of output tensor for head.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.
- **head_spatial_resolution** (*tuple*) – h, w sizes of the RoI interpolation.
- **head_spatial_scale** (*float*) – scale the input boxes by this number.
- **head_sampling_ratio** (*int*) – number of inputs samples to take for each output sample interpolation. 0 to take samples densely.
- **head_pool_kernel_sizes** (*Tuple[Tuple[int]]*) –
- **head_output_size** (*Tuple[int]*) –

Returns *(nn.Module)* – SlowFast model.

Return type torch.nn.modules.module.Module

class pytorchvideo.models.slowfast.**PoolConcatPathway**(*retain_list=False*, *pool=None*, *dim=1*)

Given a list of tensors, perform optional spatio-temporal pool and concatenate the tensors along the channel dimension.

__init__(*retain_list=False*, *pool=None*, *dim=1*)

Parameters

- **retain_list** (*bool*) – if True, return the concatenated tensor in a list.
- **pool** (*nn.module_list*) – if not None, list of pooling models for different pathway before performing concatenation.
- **dim** (*int*) – dimension to performance concatenation.

Return type None

class pytorchvideo.models.slowfast.**FuseFastToSlow**(*conv_fast_to_slow*, *norm=None*, *activation=None*)

Given a list of two tensors from Slow pathway and Fast pathway, fusion information from the Fast pathway to the Slow on through a convolution followed by a concatenation, then return the fused list of tensors from Slow and Fast pathway in order.

__init__(*conv_fast_to_slow*, *norm=None*, *activation=None*)

Parameters

- **conv_fast_to_slow** (*nn.module*) – convolution to perform fusion.
- **norm** (*nn.module*) – normalization module.
- **activation** (*torch.nn.modules*) – activation module.

Return type None

3.8 pytorchvideo.models.r2plus1d

pytorchvideo.models.r2plus1d.**create_2plus1d_bottleneck_block**(*, *dim_in*, *dim_inner*, *dim_out*, *conv_a_kernel_size=(1, 1, 1)*, *conv_a_stride=(1, 1, 1)*, *conv_a_padding=(0, 0, 0)*, *conv_a=<class 'torch.nn.modules.conv.Conv3d'>*, *conv_b_kernel_size=(3, 3, 3)*, *conv_b_stride=(2, 2, 2)*, *conv_b_padding=(1, 1, 1)*, *conv_b_num_groups=1*, *conv_b_dilation=(1, 1, 1)*, *conv_b=<function create_conv_2plus1d>*, *conv_c=<class 'torch.nn.modules.conv.Conv3d'>*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

2plus1d bottleneck block: a sequence of spatiotemporal Convolution, Normalization, and Activations repeated in the following order:

```
   Conv3d (conv_a)
          ↓
Normalization (norm_a)
          ↓
  Activation (act_a)
          ↓
 Conv(2+1)d (conv_b)
          ↓
Normalization (norm_b)
          ↓
  Activation (act_b)
          ↓
   Conv3d (conv_c)
          ↓
Normalization (norm_c)
```

Normalization examples include: BatchNorm3d and None (no normalization). Activation examples include: ReLU, Softmax, Sigmoid, and None (no activation).

Parameters

- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_inner** (*int*) – intermediate channel size of the bottleneck.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **conv_a_stride** (*tuple*) – convolutional stride size(s) for conv_a.
- **conv_a_padding** (*tuple*) – convolutional padding(s) for conv_a.
- **conv_a** (*callable*) – a callable that constructs the conv_a conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_b_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **conv_b_padding** (*tuple*) – convolutional padding(s) for conv_b.
- **conv_b_num_groups** (*int*) – number of groups for groupwise convolution for conv_b.
- **conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **conv_b** (*callable*) – a callable that constructs the conv_b conv layer, examples include nn.Conv3d, OctaveConv, etc
- **conv_c** (*callable*) – a callable that constructs the conv_c conv layer, examples include nn.Conv3d, OctaveConv, etc
- **norm** (*callable*) – a callable that constructs normalization layer, examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – 2plus1d bottleneck block.

Return type torch.nn.modules.module.Module

pytorchvideo.models.r2plus1d.**create_r2plus1d**(*, *input_channel=3*, *model_depth=50*, *model_num_class=400*, *dropout_rate=0.0*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*, *stem_dim_out=64*, *stem_conv_kernel_size=(1, 7, 7)*, *stem_conv_stride=(1, 2, 2)*, *stage_conv_a_kernel_size=((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 1, 1))*, *stage_conv_b_kernel_size=((3, 3, 3), (3, 3, 3), (3, 3, 3), (3, 3, 3))*, *stage_conv_b_num_groups=(1, 1, 1, 1)*, *stage_conv_b_dilation=((1, 1, 1), (1, 1, 1), (1, 1, 1), (1, 1, 1))*, *stage_spatial_stride=(2, 2, 2, 2)*, *stage_temporal_stride=(1, 1, 2, 2)*, *stage_bottleneck=(<function create_2plus1d_bottleneck_block>, <function create_2plus1d_bottleneck_block>, <function create_2plus1d_bottleneck_block>, <function create_2plus1d_bottleneck_block>)*, *head_pool=<class 'torch.nn.modules.pooling.AvgPool3d'>*, *head_pool_kernel_size=(4, 7, 7)*, *head_output_size=(1, 1, 1)*, *head_activation=<class 'torch.nn.modules.activation.Softmax'>*, *head_output_with_global_average=True*)

Build the R(2+1)D network from:: A closer look at spatiotemporal convolutions for action recognition. Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, Manohar Paluri. CVPR 2018.

R(2+1)D follows the ResNet style architecture including three parts: Stem, Stages and Head. The three parts are assembled in the following order:

```
Input
  ↓
Stem
  ↓
Stage 1
  ↓
  .
  .
  .
  ↓
Stage N
  ↓
Head
```

Parameters

- **input_channel** (*int*) – number of channels for the input video clip.
- **model_depth** (*int*) – the depth of the resnet.

- **model_num_class** (*int*) – the number of classes for the video dataset.
- **dropout_rate** (*float*) – dropout rate.
- **norm** (*callable*) – a callable that constructs normalization layer.
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer.
- **stem_dim_out** (*int*) – output channel size for stem.
- **stem_conv_kernel_size** (*tuple*) – convolutional kernel size(s) of stem.
- **stem_conv_stride** (*tuple*) – convolutional stride size(s) of stem.
- **stage_conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **stage_conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **stage_conv_b_num_groups** (*tuple*) – number of groups for groupwise convolution for conv_b. 1 for ResNet, and larger than 1 for ResNeXt.
- **stage_conv_b_dilation** (*tuple*) – dilation for 3D convolution for conv_b.
- **stage_spatial_stride** (*tuple*) – the spatial stride for each stage.
- **stage_temporal_stride** (*tuple*) – the temporal stride for each stage.
- **stage_bottleneck** (*tuple*) – a callable that constructs bottleneck block layer for each stage. Examples include: create_bottleneck_block, create_2plus1d_bottleneck_block.
- **head_pool** (*callable*) – a callable that constructs resnet head pooling layer.
- **head_pool_kernel_size** (*tuple*) – the pooling kernel size.
- **head_output_size** (*tuple*) – the size of output tensor for head.
- **head_activation** (*callable*) – a callable that constructs activation layer.
- **head_output_with_global_average** (*bool*) – if True, perform global averaging on the head output.

Returns *(nn.Module)* – basic resnet.

Return type torch.nn.modules.module.Module

3.9 pytorchvideo.models.simclr

class pytorchvideo.models.simclr.**SimCLR**(*mlp*, *backbone=None*, *temperature=0.07*)

A Simple Framework for Contrastive Learning of Visual Representations Details can be found from: https://arxiv.org/abs/2002.05709

forward(*x1*, *x2*)

Parameters

- **x1** (*torch.tensor*) – a batch of image with augmentation. The input tensor shape should able to be feed into the backbone.
- **x2** (*torch.tensor*) – the size batch of image with different augmentation. The input tensor shape should able to be feed into the backbone.

Return type torch.Tensor

3.10 pytorchvideo.models.byol

class pytorchvideo.models.byol.**BYOL**(*backbone*, *projector=None*, *predictor=None*, *feature_dim=2048*, *predictor_inner=4096*, *mmt=0.99*, *norm=<class 'torch.nn.modules.batchnorm.SyncBatchNorm'>*)

Bootstrap Your Own Latent A New Approach to Self-Supervised Learning Details can be found in: https://arxiv.org/pdf/2006.07733.pdf

__init__(*backbone*, *projector=None*, *predictor=None*, *feature_dim=2048*, *predictor_inner=4096*, *mmt=0.99*, *norm=<class 'torch.nn.modules.batchnorm.SyncBatchNorm'>*)

Parameters

- **backbone** (*nn.Module*) – backbone for byol, input shape depends on the forward input size. Standard inputs include *B x C*, *B x C x H x W*, and *B x C x T x H x W*.
- **projector** (*nn.Module*) – stand projector is a mlp with 2 to 3 hidden layers, with (synchronized) BatchNorm and ReLU activation.
- **predictor** (*nn.Module*) – predictor MLP of BYOL of similar structure as the projector MLP.
- **feature_dim** (*int*) – output feature dimension.
- **predictor_inner** (*int*) – inner channel size for predictor.
- **mmt** (*float*) – momentum update ratio for the momentum backbone.
- **norm** (*callable*) – normalization to be used in projector, default is synchronized batchnorm.

Return type None

sim_loss(*q*, *k*)

Similarity loss for byol. :param q and k: inputs to calculate the similarity, expected to have

the same shape of *N x C*.

update_mmt(*mmt*)

Update the momentum. This function can be used to perform momentum annealing. :param mmt: update the momentum. :type mmt: float

Parameters **mmt** (*float*) –

get_mmt()

Get the momentum. This function can be used to perform momentum annealing.

Return type float

forward_backbone_mmt(*x*)

Forward momentum backbone. :param x: input to be forwarded. :type x: tensor

forward_backbone(*x*)

Forward backbone. :param x: input to be forwarded. :type x: tensor

forward(*x1*, *x2*)

Parameters

- **x1** (*torch.tensor*) – a batch of image with augmentation. The input tensor shape should able to be feed into the backbone.

- **x2** (*torch.tensor*) – the size batch of image with different augmentation. The input tensor shape should able to be feed into the backbone.

Return type torch.Tensor

3.11 pytorchvideo.models.memory_bank

class pytorchvideo.models.memory_bank.**MemoryBank**(*backbone*, *mlp=None*, *neg_size=4096*, *temperature=0.07*, *bank_size=1280000*, *dim=2048*, *mmt=0.999*)

Performs Non-Parametric Instance Discrimination for self supervised learning on video. A memory bank is built to keep and update the historical feature embedding and use them for contrastive learning.

The original paper is: Unsupervised Feature Learning via Non-Parametric Instance Discrimination https://arxiv.org/pdf/1805.01978.pdf

More details can be found from the memory bank part in the following paper: Momentum Contrast for Unsupervised Visual Representation Learning https://arxiv.org/pdf/1911.05722.pdf

__init__(*backbone*, *mlp=None*, *neg_size=4096*, *temperature=0.07*, *bank_size=1280000*, *dim=2048*, *mmt=0.999*)

Parameters

- **backbone** (*nn.Module*) – backbone used to forward the input.
- **mlp** (*nn.Module*) – multi-layer perception used in memory bank instance discrimination model.
- **neg_size** (*int*) – size of negative samples per instance.
- **temperature** (*float*) – temperature to use for contrastive learning.
- **bank_size** (*int*) – size of the memory bank, expected to be the same size as the training set.
- **dim** (*int*) – dimension of the channel.
- **mmt** (*float*) – momentum to use.

Return type None

forward(*x*, *x_ind*)

Perform contrastive learning with random sampled negative instance from the memory bank. During training, update the memory bank with latest feature embedding.

Parameters

- **x** (*torch.tensor*) – a batch of image with augmentation. The input tensor shape should able to be feed into the backbone.
- **x_ind** (*torch.tensor*) – the index of the image x from the dataset. Expected shape is B.

Return type torch.Tensor

3.12 pytorchvideo.models.masked_multistream

class pytorchvideo.models.masked_multistream.**MaskedTemporalPooling**(*method*)

Applies temporal pooling operations on masked inputs. For each pooling operation all masked values are ignored.

__init__(*method*)

method (str): the method of pooling to use. Options: 'max': reduces temporal dimension to each valid max value. 'avg': averages valid values in the temporal dimension. 'sum': sums valid values in the temporal dimension. Note if all batch row elements are invalid, the temporal dimension is pooled to 0 values.

Parameters **method** (*str*) –

forward(*x*, *mask=None*)

Parameters

- **x** (*torch.Tensor*) – tensor with shape (batch_size, seq_len, feature_dim)
- **mask** (*torch.Tensor*) – bool tensor with shape (batch_size, seq_len). Sequence elements that are False are invalid.

Returns Tensor with shape (batch_size, feature_dim)

Return type torch.Tensor

class pytorchvideo.models.masked_multistream.**TransposeMultiheadAttention**(*feature_dim*, *num_heads=1*)

Wrapper for nn.MultiheadAttention which first transposes the input tensor from (batch_size, seq_len, feature_dim) to (seq_length, batch_size, feature_dim), then applies the attention and transposes the attention outputs back to the input shape.

__init__(*feature_dim*, *num_heads=1*)

Parameters

- **feature_dim** (*int*) – attention embedding dimension
- **num_heads** (*int*) – number of attention heads

property attention_weights

Contains attention weights from last forward call.

forward(*x*, *mask=None*)

Parameters

- **x** (*torch.Tensor*) – tensor of shape (batch_size, seq_len, feature_dim)
- **mask** (*torch.Tensor*) – bool tensor with shape (batch_size, seq_len). Sequence elements that are False are invalid.

Returns Tensor with shape (batch_size, seq_len, feature_dim)

Return type torch.Tensor

class pytorchvideo.models.masked_multistream.**LearnMaskedDefault**(*feature_dim*, *init_method='gaussian'*, *freeze=False*)

Learns default values to fill invalid entries within input tensors. The invalid entries are represented by a mask which is passed into forward alongside the input tensor. Note the default value is only used if all entries in the batch row are invalid rather than just a portion of invalid entries within each batch row.

__init__ (*feature_dim*, *init_method='gaussian'*, *freeze=False*)

Parameters

- **feature_dim** (*int*) – the size of the default value parameter, this must match the input tensor size.
- **init_method** (*str*) – the initial default value parameter. Options: 'guassian' 'zeros'
- **freeze** (*bool*) – If True, the learned default parameter weights are frozen.

forward (*x*, *mask*)

Parameters

- **x** (*torch.Tensor*) – tensor of shape (batch_size, feature_dim).
- **mask** (*torch.Tensor*) – bool tensor of shape (batch_size, seq_len) If all elements in the batch dimension are False the learned default parameter is used for that batch element.

Returns Tensor with shape (batch_size, feature_dim)

Return type torch.Tensor

class pytorchvideo.models.masked_multistream.**LSTM** (*dim_in*, *hidden_dim*, *dropout=0.0*, *bidirectional=False*)

Wrapper for torch.nn.LSTM that handles masked inputs.

__init__ (*dim_in*, *hidden_dim*, *dropout=0.0*, *bidirectional=False*)

Parameters

- **dim_in** (*int*) – input feature dimension
- **hidden_dim** (*int*) – hidden dimesion of lstm layer
- **dropout** (*float*) – dropout rate - 0.0 if no dropout
- **bidirectional** (*bool*) – bidirectional or forward only

forward (*data*, *mask=None*)

Parameters

- **data** (*torch.Tensor*) – tensor with shape (batch_size, seq_len, feature_dim)
- **mask** (*torch.Tensor*) – bool tensor with shape (batch_size, seq_len). Sequence elements that are False are invalid.

Returns

Tensor with shape (batch_size, output_dim) - outoput_dim is determined by hidden_dim and whether bidirectional or not

Return type torch.Tensor

class pytorchvideo.models.masked_multistream.**TransposeTransformerEncoder** (*dim_in*, *num_heads=1*, *num_layers=1*)

Wrapper for torch.nn.TransformerEncoder that handles masked inputs.

__init__ (*dim_in*, *num_heads=1*, *num_layers=1*)

Parameters

- **dim_in** (*int*) – input feature dimension
- **num_heads** (*int*) – number of heads in the nn.MultiHeadAttention layers

- **num_layers** (*int*) – the number of sub-encoder-layers in the encoder

forward(*data*, *mask=None*)

Parameters

- **data** (*torch.Tensor*) – tensor with shape (batch_size, seq_len, feature_dim)
- **mask** (*torch.Tensor*) – bool tensor with shape (batch_size, seq_len). Sequence elements that are False are invalid.

Returns Tensor with shape (batch_size, feature_dim)

Return type torch.Tensor

class pytorchvideo.models.masked_multistream.**MaskedSequential**(**args*)

A sequential container that overrides forward to take a mask as well as the usual input tensor. This mask is only applied to modules in _MASK_MODULES (which take the mask argument).

class pytorchvideo.models.masked_multistream.**MaskedMultiPathWay**(**, multipathway_blocks, multipathway_fusion*)

Masked multi-pathway is composed of a list of stream nn.Modules followed by a fusion nn.Module that reduces these streams. Each stream module takes a mask and input tensor.

```
Pathway 1  ... Pathway N
    ↓              ↓
 Block 1        Block N
    ↓ --Fusion----↓
```

__init__(**, multipathway_blocks, multipathway_fusion*)

Parameters

- **multipathway_blocks** (*nn.module_list*) – list of models from all pathways.
- **multipathway_fusion** (*nn.module*) – fusion model.

Return type None

CHAPTER

FOUR

OVERVIEW

PyTorchVideo datasets are subclasses of either `torch.utils.data.Dataset` or `torch.utils.data.IterableDataset`. As such, they can all be used with a `torch.utils.data.DataLoader`, which can load multiple samples in parallel using `torch.multiprocessing` workers. For example:

```
dataset = pytorchvideo.data.Kinetics(
        data_path="path/to/kinetics_root/train.csv",
        clip_sampler=pytorchvideo.data.make_clip_sampler("random", duration=2),
)
data_loader = torch.utils.data.DataLoader(dataset, batch_size=8)
```

4.1 How do PyTorchVideo datasets work?

Although there isn't a strict interface governing how PyTorchVideo datasets work, they all share a common design as follows:

1. Each dataset starts by taking a list of video paths and labels in some form. For example, Kinetics can take a file with each row containing a video path and label, or a directory containing a `\<label\>/\<video_name\>.mp4` like file structure. Each respective dataset documents the exact structure it expected for the given data path.
2. At each iteration a video sampler is used to determine which video-label pair is going to be sampled from the list of videos from the previous point. For some datasets this is required to be a random sampler, others reuse the `torch.utils.data.Sampler` interface for more flexibility.
3. A clip sampler is then used to determine which frames to sample from the selected video. For example, your application may want to sample 2 second clips at random for the selected video at each iteration. Some datasets like Kinetics make the most of the `pytorchvideo.data.clip_sampling` interface to provide flexibility on how to define these clips. Other datasets simply require you to specify an enum for common clip sampling configurations.
4. Depending on if the underlying videos are stored as either encoded videos (e.g. mp4) or frame videos (i.e. a folder of images containing each decoded frame) - the video clip is then selectively read or decoded into the canonical video tensor with shape `(C, T, H, W)` and audio tensor with shape `(S)`. We provide two options for decoding: PyAv or TorchVision, which can be chosen in the interface of the datasets that supported encoded videos.
5. The next step of a PyTorchVideo dataset is creating a clip dictionary containing the video modalities, label and metadata ready to be returned. An example clip dictionary might look like this:

```
{
   'video': <video_tensor>,     # Shape: (C, T, H, W)
   'audio': <audio_tensor>,     # Shape: (S)
```

(continues on next page)

(continued from previous page)

```
    'label': <action_label>,      # Integer defining class annotation
    'video_name': <video_path>,   # Video file path stem
    'video_index': <video_id>,    # index of video used by sampler
    'clip_index': <clip_id>       # index of clip sampled within video
}
```

All datasets share the same canonical modality tensor shapes and dtypes, which aligns with tensor types of other domain specific libraries (e.g. TorchVision, TorchAudio).

6. The final step before returning a clip, involves feeding it into a transform callable that can be defined for of all PyTorchVideo datasets. This callable is used to allow custom data processing or augmentations to be applied before batch collation in the `torch.utils.data.DataLoader`. PyTorchVideo provides common `pytorchvideo.transforms` that are useful for this callable, but users can easily define their own too.

4.2 Available datasets:

- Charades
- Domsev
- EpicKitchen
- HMDB51
- Kinetics
- SSV2
- UCF101

CHAPTER

FIVE

DATA PREPARATION

5.1 Kinetics

For more information about Kinetics dataset, please refer the official website. You can take the following steps to prepare the dataset:

1. Download the videos via the official scripts.
2. Preprocess the downloaded videos by resizing to the short edge size of 256.
3. Prepare the csv files for training, validation, and testing set as `train.csv`, `val.csv`, `test.csv`. The format of the csv file is:

```
path_to_video_1 label_1
path_to_video_2 label_2
path_to_video_3 label_3
...
path_to_video_N label_N
```

All the Kinetics models in the Model Zoo are trained and tested with the same data as Non-local Network and PySlowFast. For dataset specific issues, please reach out to the dataset provider.

5.2 Charades

We follow PySlowFast to prepare the Charades dataset as follow:

1. Download the Charades RGB frames from official website.
2. Download the *frame list* from the following links: (train, val).

5.3 Something-Something V2

We follow PySlowFast to prepare the Something-Something V2 dataset as follow:

1. Download the dataset and annotations from official website.
2. Download the *frame list* from the following links: (train, val).
3. Extract the frames from downloaded videos at 30 FPS. We used ffmpeg-4.1.3 with command:

```
ffmpeg -i "${video}" -r 30 -q:v 1 "${out_name}"
```

4. The extracted frames should be organized to be consistent with the paths in frame lists.

5.4 AVA (Actions V2.2)

The AVA Dataset could be downloaded from the official site

We followed the same downloading and preprocessing procedure as the Long-Term Feature Banks for Detailed Video Understanding do.

You could follow these steps to download and preprocess the data:

1. Download videos

```
DATA_DIR="../../data/ava/videos"

if [[ ! -d "${DATA_DIR}" ]]; then
  echo "${DATA_DIR} doesn't exist. Creating it.";
  mkdir -p ${DATA_DIR}
fi

wget https://s3.amazonaws.com/ava-dataset/annotations/ava_file_names_trainval_v2.1.txt

for line in $(cat ava_file_names_trainval_v2.1.txt)
do
  wget https://s3.amazonaws.com/ava-dataset/trainval/$line -P ${DATA_DIR}
done
```

1. Cut each video from its 15th to 30th minute. AVA has valid annotations only in this range.

```
IN_DATA_DIR="../../data/ava/videos"
OUT_DATA_DIR="../../data/ava/videos_15min"

if [[ ! -d "${OUT_DATA_DIR}" ]]; then
  echo "${OUT_DATA_DIR} doesn't exist. Creating it.";
  mkdir -p ${OUT_DATA_DIR}
fi

for video in $(ls -A1 -U ${IN_DATA_DIR}/*)
do
  out_name="${OUT_DATA_DIR}/${video##*/}"
  if [ ! -f "${out_name}" ]; then
    ffmpeg -ss 900 -t 901 -i "${video}" "${out_name}"
  fi
done
```

1. Extract frames

```
IN_DATA_DIR="../../data/ava/videos_15min"
OUT_DATA_DIR="../../data/ava/frames"

if [[ ! -d "${OUT_DATA_DIR}" ]]; then
  echo "${OUT_DATA_DIR} doesn't exist. Creating it.";
  mkdir -p ${OUT_DATA_DIR}
fi

for video in $(ls -A1 -U ${IN_DATA_DIR}/*)
do
  video_name=${video##*/}

  if [[ $video_name = *".webm" ]]; then
```

(continues on next page)

(continued from previous page)

```
    video_name=${video_name::-5}
  else
    video_name=${video_name::-4}
  fi

  out_video_dir=${OUT_DATA_DIR}/${video_name}/
  mkdir -p "${out_video_dir}"

  out_name="${out_video_dir}/${video_name}_%06d.jpg"

  ffmpeg -i "${video}" -r 30 -q:v 1 "${out_name}"
done
```

1. Download annotations

```
DATA_DIR="../../data/ava/annotations"

if [[ ! -d "${DATA_DIR}" ]]; then
  echo "${DATA_DIR} doesn't exist. Creating it.";
  mkdir -p ${DATA_DIR}
fi

wget https://research.google.com/ava/download/ava_v2.2.zip -P ${DATA_DIR}
unzip -q ${DATA_DIR}/ava_v2.2.zip -d ${DATA_DIR}
```

1. Download "frame lists" (train, val) and put them in the `frame_lists` folder (see structure above).
2. Download person boxes that are generated using a person detector trained on AVA - (train, val, test) and put them in the `annotations` folder (see structure above). Copy files to the annotations directory mentioned in step 4. If you prefer to use your own person detector, please generate detection predictions files in the suggested format in step 6.

Download the ava dataset with the following structure:

```
ava
|_ frames
|  |_ [video name 0]
|  |  |_ [video name 0]_000001.jpg
|  |  |_ [video name 0]_000002.jpg
|  |  |_ ...
|  |_ [video name 1]
|     |_ [video name 1]_000001.jpg
|     |_ [video name 1]_000002.jpg
|     |_ ...
|_ frame_lists
|  |_ train.csv
|  |_ val.csv
|_ annotations
   |_ [official AVA annotation files]
   |_ ava_train_predicted_boxes.csv
   |_ ava_val_predicted_boxes.csv
```

CHAPTER

SIX

DATA API

6.1 pytorchvideo.data

pytorchvideo.data.**Ava**(*frame_paths_file*, *frame_labels_file*, *video_path_prefix=''*, *label_map_file=None*, *clip_sampler=<class 'pytorchvideo.data.clip_sampling.ClipSampler'>*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*)

Parameters

- **frame_paths_file** (*str*) – Path to a file containing relative paths to all the frames in the video. Each line in the file is of the form <original_vido_id video_id frame_id rel_path labels>
- **frame_labels_file** (*str*) – Path to the file containing containing labels per key frame. Acceptible file formats are, Type 1:

  <original_vido_id, frame_time_stamp, bbox_x_1, bbox_y_1, ... bbox_x_2, bbox_y_2, action_lable, detection_iou>

  Type 2: <original_vido_id, frame_time_stamp, bbox_x_1, bbox_y_1, ... bbox_x_2, bbox_y_2, action_lable, person_label>

- **video_path_prefix** (*str*) – Path to be augumented to the each relative frame path to get the global frame path.
- **label_map_file** (*str*) – Path to a .pbtxt containing class id's and class names. If not set, label_map is not loaded and bbox labels are not pruned based on allowable class_id's in label_map.
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Optional[Callable]*) – This callable is evaluated on the clip output and the corresponding bounding boxes before the clip and the bounding boxes are returned. It can be used for user defined preprocessing and augmentations to the clips. If transform is None, the clip and bounding boxes are returned as it is.

Return type None

class pytorchvideo.data.**Charades**(**args*, ***kwds*)

Bases: torch.utils.data.dataset.IterableDataset

Action recognition video dataset for Charades stored as image frames.

This dataset handles the parsing of frames, loading and clip sampling for the videos. All io is done through `iopath.common.file_io.PathManager`, enabling non-local storage uri's to be used.

NUM_CLASSES = 157

__init__(*data_path*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *video_path_prefix=''*, *frames_per_clip=None*)

Parameters

- **data_path** (*str*) – Path to the data file. This file must be a space separated csv with the format: (original_vido_id video_id frame_id path_labels)
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Optional[Callable]*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations on the clips. The clip output format is described in __next__().
- **video_path_prefix** (*str*) – prefix path to add to all paths from data_path.
- **frames_per_clip** (*Optional[int]*) – The number of frames per clip to sample.

Return type None

property video_sampler

__next__()

Retrieves the next clip based on the clip sampling strategy and video sampler.

Returns

A dictionary with the following format.

```
{
    'video': <video_tensor>,
    'label': <index_label>,
    'video_label': <index_label>
    'video_index': <video_index>,
    'clip_index': <clip_index>,
    'aug_index': <aug_index>,
}
```

Return type dict

class pytorchvideo.data.**ClipSampler**(*clip_duration*)

Bases: abc.ABC

Interface for clip samplers that take a video time, previous sampled clip time, and returns a named-tuple `ClipInfo`.

class pytorchvideo.data.**RandomClipSampler**(*clip_duration*)

Bases: pytorchvideo.data.clip_sampling.ClipSampler

Randomly samples clip of size clip_duration from the videos.

__call__(*last_clip_time*, *video_duration*, *annotation*)

Parameters

- **last_clip_time** (*float*) – Not used for RandomClipSampler.
- **video_duration** (*float*) – (float): the duration (in seconds) for the video that's being sampled
- **annotation** (*Dict*) – Not used by this sampler.

Returns *clip_info (ClipInfo)* – includes the clip information of (clip_start_time, clip_end_time, clip_index, aug_index, is_last_clip). The times are in seconds. clip_index, aux_index and is_last_clip are always 0, 0 and True, respectively.

Return type pytorchvideo.data.clip_sampling.ClipInfo

class pytorchvideo.data.**UniformClipSampler**(*clip_duration*, *stride=None*, *backpad_last=False*, *eps=1e-06*)

Bases: pytorchvideo.data.clip_sampling.ClipSampler

Evenly splits the video into clips of size clip_duration.

__init__(*clip_duration*, *stride=None*, *backpad_last=False*, *eps=1e-06*)

Parameters

- **clip_duration** (*float*) – The length of the clip to sample (in seconds)
- **stride** (*float, optional*) – The amount of seconds to offset the next clip by

  default value of None is equivalent to no stride => stride == clip_duration
- **eps** (*float*) – Epsilon for floating point comparisons. Used to check the last clip.
- **backpad_last** (*bool*) – Whether to include the last frame(s) by "back padding".

  For instance, if we have a video of 39 frames (30 fps = 1.3s) with a stride of 16 (0.533s) with a clip duration of 32 frames (1.0667s). The clips will be (in frame numbers):

  with backpad_last = False - [0, 32]

  with backpad_last = True - [0, 32] - [8, 40], this is "back-padded" from [16, 48] to fit the last window

__call__(*last_clip_time*, *video_duration*, *annotation*)

Parameters

- **last_clip_time** (*float*) – the last clip end time sampled from this video. This should be 0.0 if the video hasn't had clips sampled yet.
- **video_duration** (*float*) – (float): the duration of the video that's being sampled in seconds
- **annotation** (*Dict*) – Not used by this sampler.

Returns *clip_info* – (ClipInfo): includes the clip information (clip_start_time, clip_end_time, clip_index, aug_index, is_last_clip), where the times are in seconds and is_last_clip is False when there is still more of time in the video to be sampled.

Return type pytorchvideo.data.clip_sampling.ClipInfo

pytorchvideo.data.**make_clip_sampler**(*sampling_type*, **args*)

Constructs the clip samplers found in pytorchvideo.data.clip_sampling from the given arguments.

Parameters

- **sampling_type** (*str*) – choose clip sampler to return. It has three options:

- uniform: constructs and return `UniformClipSampler`
- random: construct and return `RandomClipSampler`
- constant_clips_per_video: construct and return `ConstantClipsPerVideoSampler`

- ***args** – the args to pass to the chosen clip sampler constructor.

Return type pytorchvideo.data.clip_sampling.ClipSampler

class pytorchvideo.data.**DomsevFrameDataset**(**args*, ***kwds*)

Bases: `torch.utils.data.dataset.Dataset`

Egocentric video classification frame-based dataset for DoMSEV

This dataset handles the loading, decoding, and configurable sampling for the image frames.

__init__(*video_data_manifest_file_path*, *video_info_file_path*, *labels_file_path*, *transform=None*, *multithreaded_io=False*)

Parameters

- **video_data_manifest_file_path** (*str*) – The path to a json file outlining the available video data for the associated videos. File must be a csv (w/header) with columns: `{[f.name for f in dataclass_fields(EncodedVideoInfo)]}`

  To generate this file from a directory of video frames, see helper functions in module: `pytorchvideo.data.domsev.utils`

- **video_info_file_path** (*str*) – Path or URI to manifest with basic metadata of each video. File must be a csv (w/header) with columns: `{[f.name for f in dataclass_fields(VideoInfo)]}`

- **labels_file_path** (*str*) – Path or URI to manifest with temporal annotations for each video. File must be a csv (w/header) with columns: `{[f.name for f in dataclass_fields(LabelData)]}`

- **dataset_type** (*VideoDatasetType*) – The data format in which dataset video data is stored (e.g. video frames, encoded video etc).

- **transform** (*Optional[Callable[[Dict[str, Any]], Any]]*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user-defined preprocessing and augmentations to the clips. The clip output format is described in __next__().

- **multithreaded_io** (*bool*) – Boolean to control whether io operations are performed across multiple threads.

Return type None

__getitem__(*index*)

Samples an image frame associated to the given index.

Parameters **index** (*int*) – index for the image frame

Returns

An image frame with the following format if transform is None.

```
{{
    'frame_id': <str>,
    'image': <image_tensor>,
    'label': <label_tensor>,
}}
```

Return type Dict[str, Any]

__len__()

Returns The number of frames in the dataset.

Return type int

class pytorchvideo.data.**DomsevVideoDataset**(**args*, ***kwds*)

Bases: torch.utils.data.dataset.Dataset

Egocentric classification video clip-based dataset for DoMSEV stored as an encoded video (with frame-level labels).

This dataset handles the loading, decoding, and configurable clip sampling for the videos.

__init__(*video_data_manifest_file_path*, *video_info_file_path*, *labels_file_path*, *clip_sampler*, *dataset_type=<VideoDatasetType.Frame: 1>*, *frames_per_second=1*, *transform=None*, *frame_filter=None*, *multithreaded_io=False*)

Parameters

- **video_data_manifest_file_path** (*str*) – The path to a json file outlining the available video data for the associated videos. File must be a csv (w/header) with columns: {[f.name for f in dataclass_fields(EncodedVideoInfo)]}

  To generate this file from a directory of video frames, see helper functions in module: pytorchvideo.data.domsev.utils

- **video_info_file_path** (*str*) – Path or URI to manifest with basic metadata of each video. File must be a csv (w/header) with columns: {[f.name for f in dataclass_fields(VideoInfo)]}

- **labels_file_path** (*str*) – Path or URI to manifest with annotations for each video. File must be a csv (w/header) with columns: {[f.name for f in dataclass_fields(LabelData)]}

- **(Callable[[Dict[str** (*clip_sampler*) – List[VideoClipInfo]]): Defines how clips should be sampled from each video. See the clip sampling documentation for more information.

- **Video]** – List[VideoClipInfo]]): Defines how clips should be sampled from each video. See the clip sampling documentation for more information.

- **Dict[str** – List[VideoClipInfo]]): Defines how clips should be sampled from each video. See the clip sampling documentation for more information.

- **List[LabelData]]]** – List[VideoClipInfo]]): Defines how clips should be sampled from each video. See the clip sampling documentation for more information.

- **clip_sampler** (*Callable[[Dict[str, pytorchvideo.data.video.Video], Dict[str, List[pytorchvideo.data.domsev.LabelData]]], List[pytorchvideo.data.dataset_manifest_utils.VideoClipInfo]]*) –

- **dataset_type** (*pytorchvideo.data.dataset_manifest_utils.VideoDatasetType*) –

- **frames_per_second** (*int*) –

- **transform** (*Optional[Callable[[Dict[str, Any]], Any]]*) –

- **frame_filter** (*Optional[Callable[[List[int]], List[int]]]*) –

- **multithreaded_io** (*bool*) –

Return type None

:param [List[VideoClipInfo]]):] Defines how clips should be sampled from each video. See the clip sampling documentation for more information.

Parameters

- **dataset_type** (*VideoDatasetType*) – The data format in which dataset video data is stored (e.g. video frames, encoded video etc).
- **frames_per_second** (*int*) – The FPS of the stored videos. (NOTE: this is variable and may be different than the original FPS reported on the DoMSEV dataset website – it depends on the preprocessed subsampling and frame extraction).
- **transform** (*Optional[Callable[[Dict[str, Any]], Any]]*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user-defined preprocessing and augmentations to the clips. The clip output format is described in __next__().
- **frame_filter** (*Optional[Callable[[List[int]], List[int]]]*) – This callable is evaluated on the set of available frame indices to be included in a sampled clip. This can be used to subselect frames within a clip to be loaded.
- **multithreaded_io** (*bool*) – Boolean to control whether io operations are performed across multiple threads.
- **video_data_manifest_file_path** (*str*) –
- **video_info_file_path** (*str*) –
- **labels_file_path** (*str*) –
- **clip_sampler** (*Callable[[Dict[str, pytorchvideo.data.video.Video], Dict[str, List[pytorchvideo.data.domsev.LabelData]]], List[pytorchvideo.data.dataset_manifest_utils.VideoClipInfo]]*) –

Return type None

__getitem__(*index*)

Samples a video clip associated to the given index.

Parameters **index** (*int*) – index for the video clip.

Returns

A video clip with the following format if transform is None.

```
{{
    'video_id': <str>,
    'video': <video_tensor>,
    'audio': <audio_tensor>,
    'labels': <labels_tensor>,
    'start_time': <float>,
    'stop_time': <float>
}}
```

Return type Dict[str, Any]

__len__()

Returns The number of video clips in the dataset.

Return type int

class pytorchvideo.data.**EpicKitchenForecasting**(**args*, ***kwds*)
Bases: pytorchvideo.data.epic_kitchen.epic_kitchen_dataset.EpicKitchenDataset

Action forecasting video data set for EpicKitchen-55 Dataset. <https://epic-kitchens.github.io/2019/>

This dataset handles the loading, decoding, and clip sampling for the videos.

class pytorchvideo.data.**EpicKitchenRecognition**(**args*, ***kwds*)
Bases: pytorchvideo.data.epic_kitchen.epic_kitchen_dataset.EpicKitchenDataset

Action recognition video data set for EpicKitchen-55 Dataset. <https://epic-kitchens.github.io/2019/>

This dataset handles the loading, decoding, and clip sampling for the videos.

pytorchvideo.data.**Hmdb51**(*data_path*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *video_path_prefix=''*, *split_id=1*, *split_type='train'*, *decode_audio=True*, *decoder='pyav'*)

A helper function to create LabeledVideoDataset object for HMDB51 dataset

Parameters

- **data_path** (*pathlib.Path*) – Path to the data. The path type defines how the data should be read:
  - For a file path, the file is read and each line is parsed into a video path and label.
  - For a directory, the directory structure defines the classes (i.e. each subdirectory is a class).
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Callable*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations to the clips. See the LabeledVideoDataset class for clip output format.
- **video_path_prefix** (*str*) – Path to root directory with the videos that are loaded in LabeledVideoDataset. All the video paths before loading are prefixed with this path.
- **split_id** (*int*) – Fold id to be loaded. Options are 1, 2 or 3
- **split_type** (*str*) – Split/Fold type to be loaded. Options are ("train", "test" or "unused")
- **decoder** (*str*) – Defines which backend should be used to decode videos.

Return type pytorchvideo.data.labeled_video_dataset.LabeledVideoDataset

pytorchvideo.data.**Kinetics**(*data_path*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *video_path_prefix=''*, *decode_audio=True*, *decoder='pyav'*)

A helper function to create LabeledVideoDataset object for the Kinetics dataset.

Parameters

- **data_path** (*str*) – Path to the data. The path type defines how the data should be read:

  - For a file path, the file is read and each line is parsed into a video path and label.
  - For a directory, the directory structure defines the classes (i.e. each subdirectory is a class).
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Callable*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations to the clips. See the LabeledVideoDataset class for clip output format.
- **video_path_prefix** (*str*) – Path to root directory with the videos that are loaded in LabeledVideoDataset. All the video paths before loading are prefixed with this path.
- **decode_audio** (*bool*) – If True, also decode audio from video.
- **decoder** (*str*) – Defines what type of decoder used to decode a video.

Return type pytorchvideo.data.labeled_video_dataset.LabeledVideoDataset

class pytorchvideo.data.**LabeledVideoDataset**(**args*, ***kwds*)

Bases: torch.utils.data.dataset.IterableDataset

LabeledVideoDataset handles the storage, loading, decoding and clip sampling for a video dataset. It assumes each video is stored as either an encoded video (e.g. mp4, avi) or a frame video (e.g. a folder of jpg, or png)

__init__(*labeled_video_paths*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *decode_audio=True*, *decoder='pyav'*)

Parameters

- **labeled_video_paths** (*List[Tuple[str, Optional[dict]]]*) – List containing video file paths and associated labels. If video paths are a folder it's interpreted as a frame video, otherwise it must be an encoded video.
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Callable*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations on the clips. The clip output format is described in __next__().
- **decode_audio** (*bool*) – If True, also decode audio from video.
- **decoder** (*str*) – Defines what type of decoder used to decode a video. Not used for frame videos.

Return type None

property video_sampler

Returns: The video sampler that defines video sample order. Note that you'll need to use this property to set the epoch for a torch.utils.data.DistributedSampler.

property num_videos

Returns: Number of videos in dataset.

__next__()
Retrieves the next clip based on the clip sampling strategy and video sampler.

Returns

A dictionary with the following format.

```
{
    'video': <video_tensor>,
    'label': <index_label>,
    'video_label': <index_label>
    'video_index': <video_index>,
    'clip_index': <clip_index>,
    'aug_index': <aug_index>,
}
```

Return type dict

pytorchvideo.data.**labeled_video_dataset**(*data_path*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *video_path_prefix=''*, *decode_audio=True*, *decoder='pyav'*)
A helper function to create LabeledVideoDataset object for Ucf101 and Kinetics datasets.

Parameters

- **data_path** (*str*) – Path to the data. The path type defines how the data should be read:
  - For a file path, the file is read and each line is parsed into a video path and label.
  - For a directory, the directory structure defines the classes (i.e. each subdirectory is a class).
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Callable*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations to the clips. See the LabeledVideoDataset class for clip output format.
- **video_path_prefix** (*str*) – Path to root directory with the videos that are loaded in LabeledVideoDataset. All the video paths before loading are prefixed with this path.
- **decode_audio** (*bool*) – If True, also decode audio from video.
- **decoder** (*str*) – Defines what type of decoder used to decode a video.

Return type pytorchvideo.data.labeled_video_dataset.LabeledVideoDataset

class pytorchvideo.data.**SSv2**(**args*, ***kwds*)
Bases: torch.utils.data.dataset.IterableDataset

Action recognition video dataset for Something-something v2 (SSv2) stored as image frames.

This dataset handles the parsing of frames, loading and clip sampling for the videos. All io is done through iopath.common.file_io.PathManager, enabling non-local storage uri's to be used.

__init__(*label_name_file*, *video_label_file*, *video_path_label_file*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *video_path_prefix=''*, *frames_per_clip=None*, *rand_sample_frames=False*)

Parameters

- **label_name_file** (*str*) – SSV2 label file that contains the label names and indexes.
- **video_label_file** (*str*) – a file that contains video ids and the corresponding video label.
- **video_path_label_file** (*str*) – a file that contains frame paths for each video and the corresponding frame label. The file must be a space separated csv of the format: (original_vido_id video_id frame_id path labels).
- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Optional[Callable]*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations on the clips. The clip output format is described in __next__().
- **video_path_prefix** (*str*) – prefix path to add to all paths from data_path.
- **frames_per_clip** (*Optional[int]*) – The number of frames per clip to sample.
- **rand_sample_frames** (*bool*) – If True, randomly sampling frames for each clip.

Return type None

property video_sampler

__next__()

Retrieves the next clip based on the clip sampling strategy and video sampler.

Returns

A dictionary with the following format.

```
{
    'video': <video_tensor>,
    'label': <index_label>,
    'video_label': <index_label>
    'video_index': <video_index>,
    'clip_index': <clip_index>,
    'aug_index': <aug_index>,
}
```

Return type dict

pytorchvideo.data.**Ucf101**(*data_path*, *clip_sampler*, *video_sampler=<class 'torch.utils.data.sampler.RandomSampler'>*, *transform=None*, *video_path_prefix=''*, *decode_audio=True*, *decoder='pyav'*)

A helper function to create LabeledVideoDataset object for the Ucf101 dataset.

Parameters

- **data_path** (*str*) – Path to the data. The path type defines how the data should be read:
  - For a file path, the file is read and each line is parsed into a video path and label.
  - For a directory, the directory structure defines the classes (i.e. each subdirectory is a class).

- **clip_sampler** (ClipSampler) – Defines how clips should be sampled from each video. See the clip sampling documentation for more information.
- **video_sampler** (*Type[torch.utils.data.Sampler]*) – Sampler for the internal video container. This defines the order videos are decoded and, if necessary, the distributed split.
- **transform** (*Callable*) – This callable is evaluated on the clip output before the clip is returned. It can be used for user defined preprocessing and augmentations to the clips. See the LabeledVideoDataset class for clip output format.
- **video_path_prefix** (*str*) – Path to root directory with the videos that are loaded in LabeledVideoDataset. All the video paths before loading are prefixed with this path.
- **decode_audio** (*bool*) – If True, also decode audio from video.
- **decoder** (*str*) – Defines what type of decoder used to decode a video.

Return type pytorchvideo.data.labeled_video_dataset.LabeledVideoDataset

CHAPTER

SEVEN

OVERVIEW

The PyTorchVideo transforms package contains common video algorithms used for preprocessing and/or augmenting video data. The package also contains helper dictionary transforms that are useful for interoperability between PyTorchVideo dataset's clip outputs and domain specific transforms. For example, here is a standard transform pipeline for a video model, that could be used with a PyTorchVideo dataset:

```
transform = torchvision.transforms.Compose([
  pytorchvideo.transforms.ApplyTransformToKey(
    key="video",
    transform=torchvision.transforms.Compose([
      pytorchvideo.transforms.UniformTemporalSubsample(8),
      pytorchvideo.transforms.Normalize((0.45, 0.45, 0.45), (0.225, 0.225, 0.225)),
      pytorchvideo.transforms.RandomShortSideScale(min_size=256, max_size=320),
      torchvision.transforms.RandomCrop(244),
      torchvision.transforms.RandomHorizontalFlip(p=0.5),
    )]
  )
])
dataset = pytorchvideo.data.Kinetics(
  data_path="path/to/kinetics_root/train.csv",
  clip_sampler=pytorchvideo.data.make_clip_sampler("random", duration=2),
  transform=transform
)
```

Notice how the example also includes transforms from TorchVision? PyTorchVideo uses the same canonical tensor shape as TorchVision for video and TorchAudio for audio. This allows the frameworks to be used together freely.

7.1 Transform vs Functional interface

The example above demonstrated the `pytorchvideo.transforms` interface. These transforms are `torch.nn.module` callable classes that can be stringed together in a declarative way. PyTorchVideo also provides a `pytorchvideo.transforms.functional` interface, which are the functions that the transform API uses. These allow more fine-grained control over the transformations and may be more suitable for use outside the dataset preprocessing use case.

7.2 Scriptable transforms

All non-OpenCV transforms are TorchScriptable, as described in the TorchVision docs, in order to script the transforms together, please use `ltorch.nn.Sequential` instead of `torchvision.transform.Compose`.

CHAPTER

EIGHT

TRANSFORMS API

8.1 pytorchvideo.transforms

class pytorchvideo.transforms.**AugMix**(*magnitude=3*, *alpha=1.0*, *width=3*, *depth=- 1*, *transform_hparas=None*, *sampling_hparas=None*)

Bases: object

This implements AugMix for video. AugMix generates several chains of augmentations on the original video, which are then mixed together with each other and with the original video to create an augmented video. The input video tensor should have shape (T, C, H, W).

AugMix: A Simple Data Processing Method to Improve Robustness and Uncertainty (https://arxiv.org/pdf/1912.02781.pdf)

__init__(*magnitude=3*, *alpha=1.0*, *width=3*, *depth=- 1*, *transform_hparas=None*, *sampling_hparas=None*)

Parameters

- **magnitude** (*int*) – Magnitude used for transform function. Default is 3.
- **alpha** (*float*) – Parameter for choosing mixing weights from the beta and Dirichlet distributions. Default is 1.0.
- **width** (*int*) – The number of transformation chains. Default is 3.
- **depth** (*int*) – The number of transformations in each chain. If depth is -1, each chain will have a random length between 1 and 3 inclusive. Default is -1.
- **transform_hparas** (*Optional[Dict[Any]]*) – Transform hyper parameters. Needs to have key fill. By default, the fill value is (0.5, 0.5, 0.5).
- **sampling_hparas** (*Optional[Dict[Any]]*) – Hyper parameters for sampling. If gaussian sampling is used, it needs to have key sampling_std. By default, it uses SAMPLING_AUGMIX_DEFAULT_HPARAS.

Return type None

__call__(*video*)

Perform AugMix to the input video tensor.

Parameters **video** (*torch.Tensor*) – Input video tensor with shape (T, C, H, W).

Return type torch.Tensor

class pytorchvideo.transforms.**MixVideo**(*cutmix_prob=0.5*, *mixup_alpha=1.0*, *cutmix_alpha=1.0*, *label_smoothing=0.0*, *num_classes=400*)

Bases: torch.nn.modules.module.Module

Stochastically applies either MixUp or CutMix to the input video.

__init__(*cutmix_prob=0.5*, *mixup_alpha=1.0*, *cutmix_alpha=1.0*, *label_smoothing=0.0*, *num_classes=400*)

Parameters

- **cutmix_prob** (*float*) – Probability of using CutMix. MixUp will be used with probability 1 - cutmix_prob. If cutmix_prob is 0, then MixUp is always used. If cutmix_prob is 1, then CutMix is always used.
- **mixup_alpha** (*float*) – MixUp alpha value.
- **cutmix_alpha** (*float*) – CutMix alpha value.
- **label_smoothing** (*float*) – Label smoothing value.
- **num_classes** (*int*) – Number of total classes.

forward(*x*, *labels*)

The input is a batch of samples and their corresponding labels.

Parameters

- **x** (*torch.Tensor*) – Input tensor. The input should be a batch of videos with shape (B, C, T, H, W).
- **labels** (*torch.Tensor*) – Labels for input with shape (B).

training: **bool**

class pytorchvideo.transforms.**CutMix**(*alpha=1.0*, *label_smoothing=0.0*, *num_classes=400*)

Bases: torch.nn.modules.module.Module

CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features (https://arxiv.org/abs/1905.04899)

__init__(*alpha=1.0*, *label_smoothing=0.0*, *num_classes=400*)

This implements CutMix for videos.

Parameters

- **alpha** (*float*) – CutMix alpha value.
- **label_smoothing** (*float*) – Label smoothing value.
- **num_classes** (*int*) – Number of total classes.

Return type None

forward(*x*, *labels*)

The input is a batch of samples and their corresponding labels.

Parameters

- **x** (*torch.Tensor*) – Input tensor. The input should be a batch of videos with shape (B, C, T, H, W).
- **labels** (*torch.Tensor*) – Labels for input with shape (B).

Return type Tuple[torch.Tensor, torch.Tensor]

training: **bool**

class pytorchvideo.transforms.**MixUp**(*alpha=1.0*, *label_smoothing=0.0*, *num_classes=400*)

Bases: torch.nn.modules.module.Module

Mixup: Beyond Empirical Risk Minimization (https://arxiv.org/abs/1710.09412)

__init__(*alpha=1.0*, *label_smoothing=0.0*, *num_classes=400*)
This implements MixUp for videos.

Parameters

- **alpha** (*float*) – Mixup alpha value.
- **label_smoothing** (*float*) – Label smoothing value.
- **num_classes** (*int*) – Number of total classes.

Return type None

forward(*x*, *labels*)
The input is a batch of samples and their corresponding labels.

Parameters

- **x** (*torch.Tensor*) – Input tensor. The input should be a batch of videos with shape (B, C, T, H, W).
- **labels** (*torch.Tensor*) – Labels for input with shape (B).

Return type Tuple[torch.Tensor, torch.Tensor]

training: bool

class pytorchvideo.transforms.**RandAugment**(*magnitude=9*, *num_layers=2*, *prob=0.5*, *transform_hparas=None*, *sampling_type='gaussian'*, *sampling_hparas=None*)

Bases: object

This implements RandAugment for video. Assume the input video tensor with shape (T, C, H, W).

RandAugment: Practical automated data augmentation with a reduced search space (https://arxiv.org/abs/1909.13719)

__init__(*magnitude=9*, *num_layers=2*, *prob=0.5*, *transform_hparas=None*, *sampling_type='gaussian'*, *sampling_hparas=None*)
This implements RandAugment for video.

Parameters

- **magnitude** (*int*) – Magnitude used for transform function.
- **num_layers** (*int*) – How many transform functions to apply for each augmentation.
- **prob** (*float*) – The probablity of applying each transform function.
- **transform_hparas** (*Optional[Dict[Any]]*) – Transform hyper parameters. Needs to have key fill. By default, it uses transform_default_hparas.
- **sampling_type** (*str*) – Sampling method for magnitude of transform. It should be either gaussian or uniform.
- **sampling_hparas** (*Optional[Dict[Any]]*) – Hyper parameters for sampling. If gaussian sampling is used, it needs to have key sampling_std. By default, it uses SAMPLING_RANDAUG_DEFAULT_HPARAS.

Return type None

__call__(*video*)
Perform RandAugment to the input video tensor.

Parameters **video** (*torch.Tensor*) – Input video tensor with shape (T, C, H, W).

Return type torch.Tensor

pytorchvideo.transforms.**create_video_transform**(*mode*, *video_key=None*, *remove_key=None*, *num_samples=8*, *convert_to_float=True*, *video_mean=0.45, 0.45, 0.45*, *video_std=0.225, 0.225, 0.225*, *min_size=256*, *max_size=320*, *crop_size=224*, *horizontal_flip_prob=0.5*, *aug_type='default'*, *aug_paras=None*, *random_resized_crop_paras=None*)

Function that returns a factory default callable video transform, with default parameters that can be modified. The transform that is returned depends on the `mode` parameter: when in "train" mode, we use randomized transformations, and when in "val" mode, we use the corresponding deterministic transformations. Depending on whether `video_key` is set, the input to the transform can either be a video tensor or a dict containing `video_key` that maps to a video tensor. The video tensor should be of shape (C, T, H, W).

"train" mode "val" mode

(UniformTemporalSubsample) (UniformTemporalSubsample)

↓

(RandAugment/AugMix) ↓ ↓

(ConvertUint8ToFloat) (ConvertUint8ToFloat)

↓ ↓

Normalize Normalize ↓ ↓

RandomResizedCrop/RandomShortSideScale+RandomCrop ShortSideScale+CenterCrop

↓

RandomHorizontalFlip

(transform) = transform can be included or excluded in the returned composition of transformations

Parameters

- **mode** (*str*) – 'train' or 'val'. We use randomized transformations in 'train' mode, and we use the corresponding deterministic transformation in 'val' mode.
- **video_key** (*str, optional*) – Optional key for video value in dictionary input. When video_key is None, the input is assumed to be a torch.Tensor. Default is None.
- **remove_key** (*List[str], optional*) – Optional key to remove from a dictionary input. Default is None.
- **num_samples** (*int, optional*) – The number of equispaced samples to be selected in UniformTemporalSubsample. If None, then UniformTemporalSubsample will not be used. Default is 8.
- **convert_to_float** (*bool*) – If True, converts images from uint8 to float. Otherwise, leaves the image as is. Default is True.
- **video_mean** (*Tuple[float, float, float]*) – Sequence of means for each channel to normalize to zero mean and unit variance. Default is (0.45, 0.45, 0.45).
- **video_std** (*Tuple[float, float, float]*) – Sequence of standard deviations for each channel to normalize to zero mean and unit variance. Default is (0.225, 0.225, 0.225).

- **min_size** (*int*) – Minimum size that the shorter side is scaled to for RandomShortSideScale. If in "val" mode, this is the exact size the the shorter side is scaled to for ShortSideScale. Default is 256.
- **max_size** (*int*) – Maximum size that the shorter side is scaled to for RandomShortSideScale. Default is 340.
- **crop_size** (*int or Tuple[int, int]*) – Desired output size of the crop for RandomCrop in "train" mode and CenterCrop in "val" mode. If size is an int instead of sequence like (h, w), a square crop (size, size) is made. Default is 224.
- **horizontal_flip_prob** (*float*) – Probability of the video being flipped in RandomHorizontalFlip. Default value is 0.5.
- **aug_type** (*str*) – Currently supports 'default', 'randaug', or 'augmix'. No augmentations other than RandomShortSideScale and RandomCrop area performed when aug_type is 'default'. RandAugment is used when aug_type is 'randaug' and AugMix is used when aug_type is 'augmix'. Default is 'default'.
- **aug_paras** (*Dict[str, Any], optional*) – A dictionary that contains the necessary parameters for the augmentation set in aug_type. If any parameters are missing or if None, default parameters will be used. Default is None.
- **random_resized_crop_paras** (*Dict[str, Any], optional*) – A dictionary that contains the necessary parameters for Inception-style cropping. This crops the given videos to random size and aspect ratio. A crop of random size relative to the original size and a random aspect ratio is made. This crop is finally resized to given size. This is popularly used to train the Inception networks. If any parameters are missing or if None, default parameters in _RANDOM_RESIZED_CROP_DEFAULT_PARAS will be used. If None, RandomShortSideScale and RandomCrop will be used as a fallback. Default is None.

Returns A factory-default callable composition of transforms.

Return type Union[Callable[[torch.Tensor], torch.Tensor], Callable[[Dict[str, torch.Tensor]], Dict[str, torch.Tensor]]]

class pytorchvideo.transforms.**ApplyTransformToKey**(*key*, *transform*)

Bases: object

Applies transform to key of dictionary input.

Parameters

- **key** (*str*) – the dictionary key the transform is applied to
- **transform** (*callable*) – the transform that is applied

Example

```
>>>   transforms.ApplyTransformToKey(
>>>       key='video',
>>>       transform=UniformTemporalSubsample(num_video_samples),
>>>   )
```

pytorchvideo.transforms.**Callable**

Callable type; Callable[[int], str] is a function of (int) -> str.

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types or ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments, such function types are rarely used as callback types.

alias of Callable

class pytorchvideo.transforms.**ConvertUint8ToFloat**
Bases: torch.nn.modules.module.Module

Converts a video from dtype uint8 to dtype float32.

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor with shape (C, T, H, W).

Return type torch.Tensor

training

pytorchvideo.transforms.**Dict**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Dict

class pytorchvideo.transforms.**Div255**
Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.div_255.

forward(*x*)
Scale clip frames from [0, 255] to [0, 1]. :param x: A tensor of the clip's RGB frames with shape:

(C, T, H, W).

Returns *x (Tensor)* – Scaled tensor by dividing 255.

Parameters **x** (*Tensor*) –

Return type torch.Tensor

training

pytorchvideo.transforms.**List**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of List

class pytorchvideo.transforms.**Normalize**(*mean*, *std*, *inplace=False*)
Bases: torchvision.transforms.transforms.Normalize

Normalize the (CTHW) video clip by mean subtraction and division by standard deviation

Parameters

- **mean** (*3-tuple*) – pixel RGB mean
- **std** (*3-tuple*) – pixel RGB standard deviation

- **inplace** (*boolean*) – whether do in-place normalization

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor with shape (C, T, H, W).

Return type torch.Tensor

training

class pytorchvideo.transforms.**OpSampler**(*transforms_list*, *transforms_prob=None*, *num_sample_op=1*, *randomly_sample_depth=False*, *replacement=False*)

Bases: torch.nn.modules.module.Module

Given a list of transforms with weights, OpSampler applies weighted sampling to select n transforms, which are then applied sequentially to the input.

__init__(*transforms_list*, *transforms_prob=None*, *num_sample_op=1*, *randomly_sample_depth=False*, *replacement=False*)

Parameters

- **transforms_list** (*List[Callable]*) – A list of tuples of all available transforms to sample from.
- **transforms_prob** (*Optional[List[float]]*) – The probabilities associated with each transform in transforms_list. If not provided, the sampler assumes a uniform distribution over all transforms. They do not need to sum up to one but weights need to be positive.
- **num_sample_op** (*int*) – Number of transforms to sample and apply to input.
- **randomly_sample_depth** (*bool*) – If randomly_sample_depth is True, then uniformly sample the number of transforms to apply, between 1 and num_sample_op.
- **replacement** (*bool*) – If replacement is True, transforms are drawn with replacement.

forward(*x*)

Parameters **x** (*torch.Tensor*) – Input tensor.

Return type torch.Tensor

training

class pytorchvideo.transforms.**Permute**(*dims*)

Bases: torch.nn.modules.module.Module

Permutes the dimensions of a video.

__init__(*dims*)

Parameters **dims** (*Tuple[int]*) – The desired ordering of dimensions.

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor whose dimensions are to be permuted.

Return type torch.Tensor

training

class pytorchvideo.transforms.**RandomResizedCrop**(*target_height*, *target_width*, *scale*, *aspect_ratio*, *shift=False*, *log_uniform_ratio=True*, *interpolation='bilinear'*, *num_tries=10*)

Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.random_resized_crop.

__call__(*x*)

Parameters **x** (*torch.Tensor*) – Input video tensor with shape (C, T, H, W).

Return type torch.Tensor

training

class pytorchvideo.transforms.**RandomShortSideScale**(*min_size*, *max_size*)

Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.short_side_scale. The size parameter is chosen randomly in [min_size, max_size].

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor with shape (C, T, H, W).

Return type torch.Tensor

training

class pytorchvideo.transforms.**RemoveKey**(*key*)

Bases: torch.nn.modules.module.Module

Removes the given key from the input dict. Useful for removing modalities from a video clip that aren't needed.

__call__(*x*)

Parameters **x** (*Dict[str, torch.Tensor]*) – video clip dict.

Return type Dict[str, torch.Tensor]

training

class pytorchvideo.transforms.**ShortSideScale**(*size*)

Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.short_side_scale.

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor with shape (C, T, H, W).

Return type torch.Tensor

training

pytorchvideo.transforms.**Tuple**

Tuple type; Tuple[X, Y] is the cross-product type of X and Y.

Example: Tuple[T1, T2] is a tuple of two elements corresponding to type variables T1 and T2. Tuple[int, float, str] is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use Tuple[T, ...].

alias of Tuple

class pytorchvideo.transforms.**UniformCropVideo**(*size*, *video_key='video'*, *aug_index_key='aug_index'*)

Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.uniform_crop.

__call__(*x*)

Parameters **x** (*Dict[str, torch.Tensor]*) – video clip dict.

Return type Dict[str, torch.Tensor]

training

class pytorchvideo.transforms.**UniformTemporalSubsample**(*num_samples*)

Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.uniform_temporal_subsample.

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor with shape (C, T, H, W).

Return type torch.Tensor

training

class pytorchvideo.transforms.**UniformTemporalSubsampleRepeated**(*frame_ratios*)

Bases: torch.nn.modules.module.Module

nn.Module wrapper for pytorchvideo.transforms.functional.uniform_temporal_subsample_repeated.

forward(*x*)

Parameters **x** (*torch.Tensor*) – video tensor with shape (C, T, H, W).

training

8.2 pytorchvideo.transforms.functional

pytorchvideo.transforms.functional.**Tuple**

Tuple type; Tuple[X, Y] is the cross-product type of X and Y.

Example: Tuple[T1, T2] is a tuple of two elements corresponding to type variables T1 and T2. Tuple[int, float, str] is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use Tuple[T, ...].

alias of Tuple

pytorchvideo.transforms.functional.**uniform_temporal_subsample**(*x*, *num_samples*, *temporal_dim=-3*)

Uniformly subsamples num_samples indices from the temporal dimension of the video. When num_samples is larger than the size of temporal dimension of the video, it will sample frames based on nearest neighbor interpolation.

Parameters

- **x** (*torch.Tensor*) – A video tensor with dimension larger than one with torch tensor type includes int, long, float, complex, etc.

- **num_samples** (*int*) – The number of equispaced samples to be selected
- **temporal_dim** (*int*) – dimension of temporal to perform temporal subsample.

Returns An x-like Tensor with subsampled temporal dimension.

Return type torch.Tensor

pytorchvideo.transforms.functional.**short_side_scale**(*x*, *size*, *interpolation='bilinear'*, *backend='pytorch'*)

Determines the shorter spatial dim of the video (i.e. width or height) and scales it to the given size. To maintain aspect ratio, the longer side is then scaled accordingly. :param x: A video tensor of shape (C, T, H, W) and type torch.float32. :type x: torch.Tensor :param size: The size the shorter side is scaled to. :type size: int :param interpolation: Algorithm used for upsampling,

options: nearest' | 'linear' | 'bilinear' | 'bicubic' | 'trilinear' | 'area'

Parameters

- **backend** (*str*) – backend used to perform interpolation. Options includes *pytorch* as default, and *opencv*. Note that opencv and pytorch behave differently on linear interpolation on some versions. https://discuss.pytorch.org/t/pytorch-linear-interpolation-is-different-from-pil-opencv/71181
- **x** (*torch.Tensor*) –
- **size** (*int*) –
- **interpolation** (*str*) –

Returns An x-like Tensor with scaled spatial dims.

Return type torch.Tensor

pytorchvideo.transforms.functional.**uniform_temporal_subsample_repeated**(*frames*, *frame_ratios*, *temporal_dim=-3*)

Prepare output as a list of tensors subsampled from the input frames. Each tensor maintain a unique copy of subsampled frames, which corresponds to a unique pathway.

Parameters

- **frames** (*tensor*) – frames of images sampled from the video. Expected to have torch tensor (including int, long, float, complex, etc) with dimension larger than one.
- **frame_ratios** (*tuple*) – ratio to perform temporal down-sampling for each pathways.
- **temporal_dim** (*int*) – dimension of temporal.

Returns *frame_list (tuple)* – list of tensors as output.

Return type Tuple[torch.Tensor]

pytorchvideo.transforms.functional.**convert_to_one_hot**(*targets*, *num_class*, *label_smooth=0.0*)

This function converts target class indices to one-hot vectors, given the number of classes.

Parameters

- **targets** (*torch.Tensor*) – Index labels to be converted.
- **num_class** (*int*) – Total number of classes.

- **label_smooth** (*float*) – Label smooth value for non-target classes. Label smooth is disabled by default (0).

Return type torch.Tensor

pytorchvideo.transforms.functional.**short_side_scale_with_boxes**(*images*, *boxes*, *size*, *interpolation='bilinear'*, *backend='pytorch'*)

Perform a spatial short scale jittering on the given images and corresponding boxes. :param images: images to perform scale jitter. Dimension is

channel x *num frames* x *height* x *width*.

Parameters

- **boxes** (*tensor*) – Corresponding boxes to images. Dimension is *num boxes* x 4.
- **size** (*int*) – The size the shorter side is scaled to.
- **interpolation** (*str*) – Algorithm used for upsampling, options: nearest' | 'linear' | 'bilinear' | 'bicubic' | 'trilinear' | 'area'
- **backend** (*str*) – backend used to perform interpolation. Options includes *pytorch* as default, and *opencv*. Note that opencv and pytorch behave differently on linear interpolation on some versions. https://discuss.pytorch.org/t/pytorch-linear-interpolation-is-different-from-pil-opencv/71181
- **images** (*tensor*) –

Returns

(tensor) –

the scaled images with dimension of *channel* x *num frames* x *height* x *width*.

(tensor): the scaled boxes with dimension of *num boxes* x 4.

Return type Tuple[torch.Tensor, numpy.ndarray]

pytorchvideo.transforms.functional.**random_short_side_scale_with_boxes**(*images*, *boxes*, *min_size*, *max_size*, *interpolation='bilinear'*, *backend='pytorch'*)

Perform a spatial short scale jittering on the given images and corresponding boxes. :param images: images to perform scale jitter. Dimension is

channel x *num frames* x *height* x *width*.

Parameters

- **boxes** (*tensor*) – Corresponding boxes to images. Dimension is *num boxes* x 4.
- **min_size** (*int*) – the minimal size to scale the frames.
- **max_size** (*int*) – the maximal size to scale the frames.

- **interpolation** (*str*) – Algorithm used for upsampling, options: nearest' | 'linear' | 'bilinear' | 'bicubic' | 'trilinear' | 'area'
- **backend** (*str*) – backend used to perform interpolation. Options includes *pytorch* as default, and *opencv*. Note that opencv and pytorch behave differently on linear interpolation on some versions. https://discuss.pytorch.org/t/pytorch-linear-interpolation-is-different-from-pil-opencv/71181
- **images** (*tensor*) –

Returns

(tensor) –

the scaled images with dimension of *channel* x *num frames* x *height* x *width*.

(tensor): the scaled boxes with dimension of *num boxes* x 4.

Return type Tuple[torch.Tensor, torch.Tensor]

pytorchvideo.transforms.functional.**random_crop_with_boxes**(*images*, *size*, *boxes*)

Perform random spatial crop on the given images and corresponding boxes. :param images: images to perform random crop. The dimension is

channel x *num frames* x *height* x *width*.

Parameters

- **size** (*int*) – the size of height and width to crop on the image.
- **boxes** (*tensor*) – Corresponding boxes to images. Dimension is *num boxes* x 4.
- **images** (*tensor*) –

Returns

cropped (tensor) –

cropped images with dimension of *channel* x *num frames* x *height* x *width*.

cropped_boxes (tensor): the cropped boxes with dimension of *num boxes* x 4.

Return type Tuple[torch.Tensor, torch.Tensor]

pytorchvideo.transforms.functional.**uniform_crop**(*images*, *size*, *spatial_idx*)

Perform uniform spatial sampling on the images and corresponding boxes. :param images: images to perform uniform crop. The dimension is

channel x *num frames* x *height* x *width*.

Parameters

- **size** (*int*) – size of height and weight to crop the images.
- **spatial_idx** (*int*) – 0, 1, or 2 for left, center, and right crop if width is larger than height. Or 0, 1, or 2 for top, center, and bottom crop if height is larger than width.
- **images** (*tensor*) –

Returns

cropped (tensor) –

images with dimension of *channel* x *num frames* x *height* x *width*.

Return type torch.Tensor

pytorchvideo.transforms.functional.**uniform_crop_with_boxes**(*images*, *size*, *spatial_idx*, *boxes*)

Perform uniform spatial sampling on the images and corresponding boxes. :param images: images to perform uniform crop. The dimension is

channel x *num frames* x *height* x *width*.

Parameters

- **size** (*int*) – size of height and weight to crop the images.
- **spatial_idx** (*int*) – 0, 1, or 2 for left, center, and right crop if width is larger than height. Or 0, 1, or 2 for top, center, and bottom crop if height is larger than width.
- **boxes** (*tensor*) – Corresponding boxes to images. Dimension is *num boxes* x 4.
- **images** (*tensor*) –

Returns

cropped (tensor) –

images with dimension of *channel* x *num frames* x *height* x *width*.

cropped_boxes (tensor): the cropped boxes with dimension of *num boxes* x 4.

Return type Tuple[torch.Tensor, numpy.ndarray]

pytorchvideo.transforms.functional.**horizontal_flip_with_boxes**(*prob*, *images*, *boxes*)

Perform horizontal flip on the given images and corresponding boxes. :param prob: probility to flip the images. :type prob: float :param images: images to perform horizontal flip, the dimension is

channel x *num frames* x *height* x *width*.

Parameters

- **boxes** (*tensor*) – Corresponding boxes to images. Dimension is *num boxes* x 4.
- **prob** (*float*) –
- **images** (*tensor*) –

Returns

images (tensor) –

images with dimension of *channel* x *num frames* x *height* x *width*.

flipped_boxes (tensor): the flipped boxes with dimension of *num boxes* x 4.

Return type Tuple[torch.Tensor, torch.Tensor]

pytorchvideo.transforms.functional.**clip_boxes_to_image**(*boxes*, *height*, *width*)

Clip an array of boxes to an image with the given height and width. :param boxes: bounding boxes to perform clipping.

Dimension is *num boxes* x 4.

Parameters

- **height** (*int*) – given image height.
- **width** (*int*) – given image width.
- **boxes** (*tensor*) –

Returns

clipped_boxes (tensor) –

the clipped boxes with dimension of *num boxes* x 4.

Return type torch.Tensor

pytorchvideo.transforms.functional.**crop_boxes**(*boxes*, *x_offset*, *y_offset*)

Peform crop on the bounding boxes given the offsets. :param boxes: bounding boxes to peform crop. The dimension

is *num boxes* x 4.

Parameters

- **x_offset** (*int*) – cropping offset in the x axis.
- **y_offset** (*int*) – cropping offset in the y axis.
- **boxes** (*torch.Tensor*) –

Returns

cropped_boxes (torch.Tensor) –

the cropped boxes with dimension of *num boxes* x 4.

Return type torch.Tensor

pytorchvideo.transforms.functional.**random_resized_crop**(*frames*, *target_height*, *target_width*, *scale*, *aspect_ratio*, *shift=False*, *log_uniform_ratio=True*, *interpolation='bilinear'*, *num_tries=10*)

Crop the given images to random size and aspect ratio. A crop of random size relative to the original size and a random aspect ratio is made. This crop is finally resized to given size. This is popularly used to train the Inception networks.

Parameters

- **frames** (*torch.Tensor*) – Video tensor to be resized with shape (C, T, H, W).
- **target_height** (*int*) – Desired height after cropping.
- **target_width** (*int*) – Desired width after cropping.
- **scale** (*Tuple[float, float]*) – Scale range of Inception-style area based random resizing. Should be between 0.0 and 1.0.
- **aspect_ratio** (*Tuple[float, float]*) – Aspect ratio range of Inception-style area based random resizing. Should be between 0.0 and +infinity.
- **shift** (*bool*) – Bool that determines whether or not to sample two different boxes (for cropping) for the first and last frame. If True, it then linearly interpolates the two boxes for other frames. If False, the same box is cropped for every frame. Default is False.
- **log_uniform_ratio** (*bool*) – Whether to use a log-uniform distribution to sample the aspect ratio. Default is True.
- **interpolation** (*str*) – Algorithm used for upsampling. Currently supports 'nearest', 'bilinear', 'bicubic', 'area'. Default is 'bilinear'.

- **num_tries** (*int*) – The number of times to attempt a randomly resized crop. Falls back to a central crop after all attempts are exhausted. Default is 10.

Returns *cropped (tensor)* – A cropped video tensor of shape (C, T, target_height, target_width).

Return type torch.Tensor

pytorchvideo.transforms.functional.**div_255**(*x*)

Divide the given tensor x by 255.

Parameters **x** (*torch.Tensor*) – The input tensor.

Returns *y (torch.Tensor)* – Scaled tensor by dividing 255.

Return type torch.Tensor

CHAPTER

NINE

OVERVIEW

PyTorchVideo is an open source video understanding library that provides up to date builders for state of the art video understanding backbones, layers, heads, and losses addressing different tasks, including acoustic event detection, action recognition (video classification), action detection (video detection), multimodal understanding (acoustic visual classification), self-supervised learning.

The layers subpackage contains definitions for the following layers and activations:

- Layer
  - BatchNorm
  - 2+1 Conv
  - ConCat
  - MLP
  - Nonlocal Net
  - Positional Encoding
  - Squeeze and Excitation
  - Swish

9.1 Build standard models

PyTorchVideo provide default builders to construct state-of-the-art video understanding layers and activations.

9.1.1 Layers

You can construct a layer with random weights by calling its constructor:

```
import pytorchvideo.layers as layers

nonlocal = layers.create_nonlocal(dim_in=256, dim_inner=128)
swish = layers.Swish()
conv_2plus1d = layers.create_conv_2plus1d(in_channels=256, out_channels=512)
```

You can verify whether you have built the model successfully by:

```
import pytorchvideo.layers as layers

nonlocal = layers.create_nonlocal(dim_in=256, dim_inner=128)
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = nonlocal(input_tensor)

swish = layers.Swish()
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = swish(input_tensor)

conv_2plus1d = layers.create_conv_2plus1d(in_channels=256, out_channels=512)
B, C, T, H, W = 2, 256, 4, 14, 14
input_tensor = torch.zeros(B, C, T, H, W)
output = conv_2plus1d(input_tensor)
```

CHAPTER

TEN

LAYERS API

10.1 pytorchvideo.layers.batch_norm

class pytorchvideo.layers.batch_norm.**NaiveSyncBatchNorm1d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*)

An implementation of 1D naive sync batch normalization. See details in NaiveSyncBatchNorm2d below.

class pytorchvideo.layers.batch_norm.**NaiveSyncBatchNorm2d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*)

An implementation of 2D naive sync batch normalization. In PyTorch<=1.5, `nn.SyncBatchNorm` has incorrect gradient when the batch size on each worker is different. (e.g., when scale augmentation is used, or when it is applied to mask head).

This is a slower but correct alternative to *nn.SyncBatchNorm*.

Note: This module computes overall statistics by using statistics of each worker with equal weight. The result is true statistics of all samples (as if they are all on one worker) only when all workers have the same (N, H, W). This mode does not support inputs with zero batch size.

---

class pytorchvideo.layers.batch_norm.**NaiveSyncBatchNorm3d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*)

An implementation of 3D naive sync batch normalization. See details in NaiveSyncBatchNorm2d above.

10.2 pytorchvideo.layers.convolutions

class pytorchvideo.layers.convolutions.**ConvReduce3D**(*, *in_channels*, *out_channels*, *kernel_size*, *stride=None*, *padding=None*, *padding_mode=None*, *dilation=None*, *groups=None*, *bias=None*, *reduction_method='sum'*)

Builds a list of convolutional operators and performs summation on the outputs.

```
Conv3d, Conv3d, ...,  Conv3d
                ↓
               Sum
```

__init__(*, *in_channels*, *out_channels*, *kernel_size*, *stride=None*, *padding=None*, *padding_mode=None*, *dilation=None*, *groups=None*, *bias=None*, *reduction_method='sum'*)

Parameters

- **int** (*out_channels*) – number of input channels.
- **int** – number of output channels produced by the convolution(s).
- **tuple** (*bias*) – Tuple of sizes of the convolutionaling kernels.
- **tuple** – Tuple of strides of the convolutions.
- **tuple** – Tuple of paddings added to all three sides of the input.
- **tuple** – Tuple of padding modes for each convs. Options include *zeros*, *reflect*, *replicate* or *circular*.
- **tuple** – Tuple of spacings between kernel elements.
- **tuple** – Tuple of numbers of blocked connections from input channels to output channels.
- **tuple** – If *True*, adds a learnable bias to the output.
- **str** (*reduction_method*) – Options include *sum* and *cat*.
- **in_channels** (*int*) –
- **out_channels** (*int*) –
- **kernel_size** (*Tuple[Union[int, Tuple[int, int, int]]]*) –
- **stride** (*Optional[Tuple[Union[int, Tuple[int, int, int]]]]*) –
- **padding** (*Optional[Tuple[Union[int, Tuple[int, int, int]]]]*) –
- **padding_mode** (*Optional[Tuple[str]]*) –
- **dilation** (*Optional[Tuple[Union[int, Tuple[int, int, int]]]]*) –
- **groups** (*Optional[Tuple[int]]*) –
- **bias** (*Optional[Tuple[bool]]*) –
- **reduction_method** (*str*) –

Return type None

pytorchvideo.layers.convolutions.**create_conv_2plus1d**(*, *in_channels*, *out_channels*, *inner_channels=None*, *conv_xy_first=False*, *kernel_size=(3, 3, 3)*, *stride=(2, 2, 2)*, *padding=(1, 1, 1)*, *bias=False*, *dilation=(1, 1, 1)*, *groups=1*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*, *activation=<class 'torch.nn.modules.activation.ReLU'>*)

Create a 2plus1d conv layer. It performs spatiotemporal Convolution, BN, and Relu following by a spatiotemporal pooling.

```
Conv_t (or Conv_xy if conv_xy_first = True)
                  ↓
             Normalization
                  ↓
               Activation
                  ↓
Conv_xy (or Conv_t if conv_xy_first = True)
```

Normalization options include: BatchNorm3d and None (no normalization). Activation options include: ReLU, Softmax, Sigmoid, and None (no activation).

Parameters

- **in_channels** (*int*) – input channel size of the convolution.
- **out_channels** (*int*) – output channel size of the convolution.
- **kernel_size** (*tuple*) – convolutional kernel size(s).
- **stride** (*tuple*) – convolutional stride size(s).
- **padding** (*tuple*) – convolutional padding size(s).
- **bias** (*bool*) – convolutional bias. If true, adds a learnable bias to the output.
- **groups** (*int*) – Number of groups in convolution layers. value >1 is unsupported.
- **dilation** (*tuple*) – dilation value in convolution layers. value >1 is unsupported.
- **conv_xy_first** (*bool*) – If True, spatial convolution comes before temporal conv
- **norm** (*callable*) – a callable that constructs normalization layer, options include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer, options include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).
- **inner_channels** (*int*) –

Returns *(nn.Module)* – 2plus1d conv layer.

Return type torch.nn.modules.module.Module

class pytorchvideo.layers.convolutions.**Conv2plus1d**(*, *conv_t=None*, *norm=None*, *activation=None*, *conv_xy=None*, *conv_xy_first=False*)

Implementation of 2+1d Convolution by factorizing 3D Convolution into an 1D temporal Convolution and a 2D spatial Convolution with Normalization and Activation module in between:

```
Conv_t (or Conv_xy if conv_xy_first = True)
                  ↓
             Normalization
                  ↓
               Activation
                  ↓
Conv_xy (or Conv_t if conv_xy_first = True)
```

The 2+1d Convolution is used to build the R(2+1)D network.

__init__(*, *conv_t=None*, *norm=None*, *activation=None*, *conv_xy=None*, *conv_xy_first=False*)

Parameters

- **conv_t** (*torch.nn.modules*) – temporal convolution module.
- **norm** (*torch.nn.modules*) – normalization module.
- **activation** (*torch.nn.modules*) – activation module.
- **conv_xy** (*torch.nn.modules*) – spatial convolution module.
- **conv_xy_first** (*bool*) – If True, spatial convolution comes before temporal conv

Return type None

10.3 pytorchvideo.layers.fusion

pytorchvideo.layers.fusion.**make_fusion_layer**(*method*, *feature_dims*)

Parameters

- **method** (*str*) – the fusion method to be constructed. Options: - 'concat' - 'temporal_concat' - 'max' - 'sum' - 'prod'
- **feature_dims** (*List[int]*) – the first argument of all fusion layers. It holds a list of required feature_dims for each tensor input (where the tensor inputs are of shape (batch_size, seq_len, feature_dim)). The list order must corresponds to the tensor order passed to forward(...).

class pytorchvideo.layers.fusion.**ConcatFusion**(*feature_dims*)

Concatenates all inputs by their last dimension. The resulting tensor last dim will be the sum of the last dimension of all input tensors.

property output_dim

Last dimension size of forward(..) tensor output.

forward(*input_list*)

Parameters **input_list** (*List[torch.Tensor]*) – a list of tensors of shape (batch_size, seq_len, feature_dim).

Returns

Tensor of shape (batch_size, seq_len, sum(feature_dims)) where sum(feature_dims) is the sum of all input feature_dims.

Return type torch.Tensor

class pytorchvideo.layers.fusion.**TemporalConcatFusion**(*feature_dims*)
Concatenates all inputs by their temporal dimension which is assumed to be dim=1.

property output_dim
Last dimension size of forward(..) tensor output.

forward(*input_list*)

Parameters **input_list** (*List[torch.Tensor]*) – a list of tensors of shape (batch_size, seq_len, feature_dim)

Returns

Tensor of shape (batch_size, sum(seq_len), feature_dim) where sum(seq_len) is the sum of all input tensors.

Return type torch.Tensor

class pytorchvideo.layers.fusion.**ReduceFusion**(*feature_dims*, *reduce_fn*)
Generic fusion method which takes a callable which takes the list of input tensors and expects a single tensor to be used. This class can be used to implement fusion methods like "sum", "max" and "prod".

property output_dim
Last dimension size of forward(..) tensor output.

forward(*input_list*)

Parameters **input_list** (*List[torch.Tensor]*) – a list of tensors of shape (batch_size, seq_len, feature_dim).

Returns Tensor of shape (batch_size, seq_len, feature_dim).

Return type torch.Tensor

10.4 pytorchvideo.layers.mlp

pytorchvideo.layers.mlp.**make_multilayer_perceptron**(*fully_connected_dims*, *norm=None*, *mid_activation=<class 'torch.nn.modules.activation.ReLU'>*, *final_activation=<class 'torch.nn.modules.activation.ReLU'>*, *dropout_rate=0.0*)
Factory function for Multi-Layer Perceptron. These are constructed as repeated blocks of the following format where each fc represents the blocks output/input dimension.

```
       Linear (in=fc[i-1], out=fc[i])
                         ↓
           Normalization (norm)
                         ↓
         Activation (mid_activation)
                         ↓
      After the repeated Perceptron blocks,
a final dropout and activation layer is applied:
                         ↓
         Dropout (p=dropout_rate)
                         ↓
         Activation (final_activation)
```

Parameters

- **fully_connected_dims** (*List[int]*) –
- **norm** (*Optional[Callable]*) –
- **mid_activation** (*Callable*) –
- **final_activation** (*Optional[Callable]*) –
- **dropout_rate** (*float*) –

Return type Tuple[torch.nn.modules.module.Module, int]

10.5 pytorchvideo.layers.nonlocal_net

class pytorchvideo.layers.nonlocal_net.**NonLocal**(*, *conv_theta*, *conv_phi*, *conv_g*, *conv_out*, *pool=None*, *norm=None*, *instantiation='dot_product'*)

Builds Non-local Neural Networks as a generic family of building blocks for capturing long-range dependencies. Non-local Network computes the response at a position as a weighted sum of the features at all positions. This building block can be plugged into many computer vision architectures. More details in the paper: Wang, Xiaolong, Ross Girshick, Abhinav Gupta, and Kaiming He. "Non-local neural networks." In Proceedings of the IEEE conference on CVPR, 2018.

pytorchvideo.layers.nonlocal_net.**create_nonlocal**(*, *dim_in*, *dim_inner*, *pool_size=(1, 1, 1)*, *instantiation='softmax'*, *norm=<class 'torch.nn.modules.batchnorm.BatchNorm3d'>*, *norm_eps=1e-05*, *norm_momentum=0.1*)

Builds Non-local Neural Networks as a generic family of building blocks for capturing long-range dependencies. Non-local Network computes the response at a position as a weighted sum of the features at all positions. This building block can be plugged into many computer vision architectures. More details in the paper: https://arxiv.org/pdf/1711.07971 :param dim_in: number of dimension for the input. :type dim_in: int :param dim_inner: number of dimension inside of the Non-local block. :type dim_inner: int :param pool_size: the kernel size of spatial temporal pooling,

temporal pool kernel size, spatial pool kernel size, spatial pool kernel size in order. By default pool_size is None, then there would be no pooling used.

Parameters

- **instantiation** (*string*) – supports two different instantiation method: "dot_product": normalizing correlation matrix with L2. "softmax": normalizing correlation matrix with Softmax.
- **norm** (*nn.Module*) – nn.Module for the normalization layer. The default is nn.BatchNorm3d.
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **dim_in** (*int*) –
- **dim_inner** (*int*) –
- **pool_size** (*tuple[int]*) –

10.6 pytorchvideo.layers.positional_encoding

class pytorchvideo.layers.positional_encoding.**PositionalEncoding**(*embed_dim*, *seq_len=1024*)

Applies a positional encoding to a tensor with shape (batch_size x seq_len x embed_dim).

The positional encoding is computed as follows: PE(pos,2i) = sin(pos/10000^(2i/dmodel)) PE(pos,2i+1) = cos(pos/10000^(2i/dmodel))

where pos = position, pos in [0, seq_len) dmodel = data embedding dimension = embed_dim i = dimension index, i in [0, embed_dim)

Reference: "Attention Is All You Need" https://arxiv.org/abs/1706.03762 Implementation Reference: https://pytorch.org/tutorials/beginner/transformer_tutorial.html

class pytorchvideo.layers.positional_encoding.**SpatioTemporalClsPositionalEncoding**(*embed_dim*, *patch_embed*, *sep_pos_em*, *has_cls=Tru*

Add a cls token and apply a spatiotemporal encoding to a tensor.

__init__(*embed_dim*, *patch_embed_shape*, *sep_pos_embed=False*, *has_cls=True*)

Parameters

- **embed_dim** (*int*) – Embedding dimension for input sequence.
- **patch_embed_shape** (*Tuple*) – The number of patches in each dimension (T, H, W) after patch embedding.
- **sep_pos_embed** (*bool*) – If set to true, one positional encoding is used for spatial patches and another positional encoding is used for temporal sequence. Otherwise, only one positional encoding is used for all the patches.
- **has_cls** (*bool*) – If set to true, a cls token is added in the beginning of each input sequence.

Return type None

forward(*x*)

Parameters **x** (*torch.Tensor*) – Input tensor.

Return type torch.Tensor

10.7 pytorchvideo.layers.swish

class pytorchvideo.layers.swish.**Swish**

Wrapper for the Swish activation function.

class pytorchvideo.layers.swish.**SwishFunction**(**args*, ***kwargs*)

Implementation of the Swish activation function: x * sigmoid(x).

Searching for activation functions. Ramachandran, Prajit and Zoph, Barret and Le, Quoc V. 2017

10.8 pytorchvideo.layers.squeeze_excitation

class pytorchvideo.layers.squeeze_excitation.**SqueezeAndExcitationLayer2D**(*in_planes*, *reduction_ratio=16*, *reduced_planes=None*)

2D Squeeze and excitation layer, as per https://arxiv.org/pdf/1709.01507.pdf

__init__(*in_planes*, *reduction_ratio=16*, *reduced_planes=None*)

Parameters

- **in_planes** (*int*) – input channel dimension.
- **reduction_ratio** (*int*) – factor by which in_planes should be reduced to get the output channel dimension.
- **reduced_planes** (*int*) – Output channel dimension. Only one of reduction_ratio or reduced_planes should be defined.

forward(*x*)

Parameters **x** (*tensor*) – 2D image of format C * H * W

Return type torch.Tensor

pytorchvideo.layers.squeeze_excitation.**create_audio_2d_squeeze_excitation_block**(*dim_in*, *dim_out*, *use_se=False*, *se_reduction_ra*, *branch_fusion=<lambda>>*, *conv_a_kernel_*, *conv_a_stride=*, *conv_a_paddin*, *conv_b_kernel_*, *conv_b_stride=*, *conv_b_paddin*, *norm=<class 'torch.nn.modul*, *norm_eps=1e-05*, *norm_momentu*, *activation=<class 'torch.nn.modul*)

2-D Residual block with squeeze excitation (SE2D) for 2d. Performs a summation between an identity shortcut in branch1 and a main block in branch2. When the input and output dimensions are different, a convolution followed by a normalization will be performed.

```
Input
  |-------+
  ↓       |
conv2d    |
```

(continues on next page)

(continued from previous page)

```
    ↓       |
   Norm     |
    ↓       |
activation   |
    ↓       |
  conv2d     |
    ↓       |
   Norm     |
    ↓       |
   SE2D     |
    ↓       }
Summation ←-+
    ↓
Activation
```

Normalization examples include: BatchNorm3d and None (no normalization). Activation examples include: ReLU, Softmax, Sigmoid, and None (no activation). Transform examples include: BottleneckBlock.

Parameters

- **dim_in** (*int*) – input channel size to the bottleneck block.
- **dim_out** (*int*) – output channel size of the bottleneck.
- **use_se** (*bool*) – if true, use squeeze excitation layer in the bottleneck.
- **se_reduction_ratio** (*int*) – factor by which input channels should be reduced to get the output channel dimension in SE layer.
- **branch_fusion** (*callable*) – a callable that constructs summation layer. Examples include: lambda x, y: x + y, OctaveSum.
- **conv_a_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_a.
- **conv_a_stride** (*tuple*) – convolutional stride size(s) for conv_a.
- **conv_a_padding** (*tuple*) – convolutional padding(s) for conv_a.
- **conv_b_kernel_size** (*tuple*) – convolutional kernel size(s) for conv_b.
- **conv_b_stride** (*tuple*) – convolutional stride size(s) for conv_b.
- **conv_b_padding** (*tuple*) – convolutional padding(s) for conv_b.
- **norm** (*callable*) – a callable that constructs normalization layer. Examples include nn.BatchNorm3d, None (not performing normalization).
- **norm_eps** (*float*) – normalization epsilon.
- **norm_momentum** (*float*) – normalization momentum.
- **activation** (*callable*) – a callable that constructs activation layer in bottleneck and block. Examples include: nn.ReLU, nn.Softmax, nn.Sigmoid, and None (not performing activation).

Returns *(nn.Module)* – resnet basic block layer.

Return type torch.nn.modules.module.Module

CHAPTER

ELEVEN

OVERVIEW

Our vision for PyTorchVideo/Accelerator is to enable video understanding models to run efficiently on all tiers of hardware devices, from mobile phone to GPU. PyTorchVideo/Accelerator (Accelerator) is aimed to accelerate the speed of video understanding model running on various hardware devices, as well as the whole process of design and deploy hardware-aware efficient video understanding models. Specifically, Accelerator provides a complete environment which allows users to:

- Design efficient models for target hardware with carefully tuned efficient blocks;
- Fine tune efficient model from Model Zoo;
- Optimize model kernel and graph for target device;
- Deploy efficient model to target device.

We benchmarked the latency of SOTA models (X3D-XS and X3D-S) on a mainstream mobile device (Samsung S9 International, released in 2018). With Accelerator, we not only observed 4-6X latency reduction on fp32, but also enabled int8 operation which has not been supported in vanilla Pytorch. A table summarizing latency comparison is shown below.

11.1 Components in PyTorchVideo/Accelerator

11.1.1 Efficient block library

Efficient block library contains common building blocks (residual block, squeeze-excite, etc.) that can be mapped to high-performance kernel operator implementation library of target device platform. The rationale behind having an efficient block library is that high-performance kernel operator library generally only supports a small set of kernel operators. In other words, a randomly picked kernel might not be supported by high-performance kernel operator library. By having an efficient block library and building model using efficient blocks in that library can guarantee the model is deployable with high efficiency on target device.

Efficient block library lives under `pytorchvideo/layers/accelerator/<target_device>` (for simple layers) and `pytorchvideo/models/accelerator/<target_device>` (for complex modules such as residual block). Please also check Build your model with PyTorchVideo/Accelerator tutorial for detailed examples.

11.1.2 Deployment

Deployment flow includes kernel optimization as well as model export for target backend. Kernel optimization utilities can be an extremely important part that decides performance of on-device model operation. Accelerator provides a bunch of useful utilities for deployment under `pytorchvideo/accelerator/deployment`. Please also check related tutorials (Build your model with PyTorchVideo/Accelerator, Accelerate your model with model transmuter in PyTorchVideo/Accelerator) for detailed examples.

11.1.3 Model zoo

Accelerator provides efficient model zoo for target devices, which include model builder (under `pytorchvideo/models/accelerator/<target_device>`) as well as pretrained checkpoint. Please also refer to Use PyTorchVideo/Accelerator Model Zoo for how to use model zoo.

11.2 Supported devices

Currently mobile cpu (ARM-based cpu on mobile phones) is supported. We will update this page once more target devices are supported.

11.3 Demo

Checkout our on-device video classification demos running on mobile phone!

Android demo

iOS demo

11.4 Jumpstart

Refer to following tutorial pages to get started!

Build your model with PyTorchVideo/Accelerator

Use PyTorchVideo/Accelerator Model Zoo

Accelerate your model with model transmuter in PyTorchVideo/Accelerator

PYTHON MODULE INDEX

p

pytorchvideo.data, 65
pytorchvideo.layers.batch_norm, 97
pytorchvideo.layers.convolutions, 98
pytorchvideo.layers.fusion, 100
pytorchvideo.layers.mlp, 101
pytorchvideo.layers.nonlocal_net, 102
pytorchvideo.layers.positional_encoding, 103
pytorchvideo.layers.squeeze_excitation, 104
pytorchvideo.layers.swish, 103
pytorchvideo.models.byol, 54
pytorchvideo.models.csn, 34
pytorchvideo.models.head, 25
pytorchvideo.models.masked_multistream, 56
pytorchvideo.models.memory_bank, 55
pytorchvideo.models.net, 24
pytorchvideo.models.r2plus1d, 50
pytorchvideo.models.resnet, 7
pytorchvideo.models.simclr, 53
pytorchvideo.models.slowfast, 44
pytorchvideo.models.stem, 31
pytorchvideo.models.x3d, 36
pytorchvideo.transforms, 79
pytorchvideo.transforms.functional, 87

INDEX

Symbols

__call__() (*pytorchvideo.data.RandomClipSampler method*), 66
__call__() (*pytorchvideo.data.UniformClipSampler method*), 67
__call__() (*pytorchvideo.transforms.AugMix method*), 79
__call__() (*pytorchvideo.transforms.RandAugment method*), 81
__call__() (*pytorchvideo.transforms.RandomResizedCrop method*), 86
__call__() (*pytorchvideo.transforms.RemoveKey method*), 86
__call__() (*pytorchvideo.transforms.UniformCropVideo method*), 87
__getitem__() (*pytorchvideo.data.DomsevFrameDataset method*), 68
__getitem__() (*pytorchvideo.data.DomsevVideoDataset method*), 70
__init__() (*pytorchvideo.data.Charades method*), 66
__init__() (*pytorchvideo.data.DomsevFrameDataset method*), 68
__init__() (*pytorchvideo.data.DomsevVideoDataset method*), 69
__init__() (*pytorchvideo.data.LabeledVideoDataset method*), 72
__init__() (*pytorchvideo.data.SSv2 method*), 73
__init__() (*pytorchvideo.data.UniformClipSampler method*), 67
__init__() (*pytorchvideo.layers.convolutions.Conv2plus1d method*), 100
__init__() (*pytorchvideo.layers.convolutions.ConvReduce3D method*), 98
__init__() (*pytorchvideo.layers.positional_encoding.SpatioTemporalClsPositionalEncoding method*), 103
__init__() (*pytorchvideo.layers.squeeze_excitation.SqueezeAndExcitationLayer2D method*), 104
__init__() (*pytorchvideo.models.byol.BYOL method*), 54
__init__() (*pytorchvideo.models.head.ResNetBasicHead method*), 29
__init__() (*pytorchvideo.models.head.ResNetRoIHead method*), 30
__init__() (*pytorchvideo.models.head.SequencePool method*), 25
__init__() (*pytorchvideo.models.head.VisionTransformerBasicHead method*), 30
__init__() (*pytorchvideo.models.masked_multistream.LSTM method*), 57
__init__() (*pytorchvideo.models.masked_multistream.LearnMaskedDe method*), 56
__init__() (*pytorchvideo.models.masked_multistream.MaskedMultiPat method*), 58
__init__() (*pytorchvideo.models.masked_multistream.MaskedTempora method*), 56
__init__() (*pytorchvideo.models.masked_multistream.TransposeMultih method*), 56
__init__() (*pytorchvideo.models.masked_multistream.TransposeTransf method*), 57
__init__() (*pytorchvideo.models.memory_bank.MemoryBank method*), 55
__init__() (*pytorchvideo.models.net.DetectionBBoxNetwork method*), 25
__init__() (*pytorchvideo.models.net.MultiPathWayWithFuse method*), 25
__init__() (*pytorchvideo.models.net.Net method*), 24
__init__() (*pytorchvideo.models.resnet.BottleneckBlock method*), 23
__init__() (*pytorchvideo.models.resnet.ResBlock method*), 22
__init__() (*pytorchvideo.models.resnet.ResStage method*), 24
__init__() (*pytorchvideo.models.resnet.SeparableBottleneckBlock method*), 23
__init__() (*pytorchvideo.models.slowfast.FuseFastToSlow method*), 50
__init__() (*pytorchvideo.models.slowfast.PoolConcatPathway method*), 49
__init__() (*pytorchvideo.models.stem.ResNetBasicStem method*), 33
__init__() (*pytorchvideo.models.x3d.ProjectedPool method*), 42

__init__() (*pytorchvideo.transforms.AugMix method*), 79
__init__() (*pytorchvideo.transforms.CutMix method*), 80
__init__() (*pytorchvideo.transforms.MixUp method*), 80
__init__() (*pytorchvideo.transforms.MixVideo method*), 80
__init__() (*pytorchvideo.transforms.OpSampler method*), 85
__init__() (*pytorchvideo.transforms.Permute method*), 85
__init__() (*pytorchvideo.transforms.RandAugment method*), 81
__len__() (*pytorchvideo.data.DomsevFrameDataset method*), 69
__len__() (*pytorchvideo.data.DomsevVideoDataset method*), 70
__next__() (*pytorchvideo.data.Charades method*), 66
__next__() (*pytorchvideo.data.LabeledVideoDataset method*), 72
__next__() (*pytorchvideo.data.SSv2 method*), 74

A

ApplyTransformToKey (*class in pytorchvideo.transforms*), 83
attention_weights() (*pytorchvideo.models.masked_multistream.TransposeMultiheadAttention property*), 56
AugMix (*class in pytorchvideo.transforms*), 79
Ava() (*in module pytorchvideo.data*), 65

B

BottleneckBlock (*class in pytorchvideo.models.resnet*), 23
BYOL (*class in pytorchvideo.models.byol*), 54

C

Callable (*in module pytorchvideo.transforms*), 83
Charades (*class in pytorchvideo.data*), 65
clip_boxes_to_image() (*in module pytorchvideo.transforms.functional*), 91
ClipSampler (*class in pytorchvideo.data*), 66
ConcatFusion (*class in pytorchvideo.layers.fusion*), 100
Conv2plus1d (*class in pytorchvideo.layers.convolutions*), 99
convert_to_one_hot() (*in module pytorchvideo.transforms.functional*), 88
ConvertUint8ToFloat (*class in pytorchvideo.transforms*), 84
ConvReduce3D (*class in pytorchvideo.layers.convolutions*), 98
create_2plus1d_bottleneck_block() (*in module pytorchvideo.models.r2plus1d*), 50
create_acoustic_bottleneck_block() (*in module pytorchvideo.models.resnet*), 8
create_acoustic_res_basic_stem() (*in module pytorchvideo.models.stem*), 32
create_acoustic_resnet() (*in module pytorchvideo.models.resnet*), 19
create_audio_2d_squeeze_excitation_block() (*in module pytorchvideo.layers.squeeze_excitation*), 104
create_bottleneck_block() (*in module pytorchvideo.models.resnet*), 7
create_conv_2plus1d() (*in module pytorchvideo.layers.convolutions*), 98
create_conv_patch_embed() (*in module pytorchvideo.models.stem*), 34
create_csn() (*in module pytorchvideo.models.csn*), 34
create_nonlocal() (*in module pytorchvideo.layers.nonlocal_net*), 102
create_r2plus1d() (*in module pytorchvideo.models.r2plus1d*), 51
create_res_basic_head() (*in module pytorchvideo.models.head*), 25
create_res_basic_stem() (*in module pytorchvideo.models.stem*), 31
create_res_block() (*in module pytorchvideo.models.resnet*), 10
create_res_roi_pooling_head() (*in module pytorchvideo.models.head*), 27
create_res_stage() (*in module pytorchvideo.models.resnet*), 12
create_resnet() (*in module pytorchvideo.models.resnet*), 14
create_resnet_with_roi_head() (*in module pytorchvideo.models.resnet*), 16
create_slowfast() (*in module pytorchvideo.models.slowfast*), 44
create_slowfast_with_roi_head() (*in module pytorchvideo.models.slowfast*), 46
create_video_transform() (*in module pytorchvideo.transforms*), 81
create_vit_basic_head() (*in module pytorchvideo.models.head*), 26
create_x3d() (*in module pytorchvideo.models.x3d*), 40
create_x3d_bottleneck_block() (*in module pytorchvideo.models.x3d*), 36
create_x3d_head() (*in module pytorchvideo.models.x3d*), 39
create_x3d_res_block() (*in module pytorchvideo.models.x3d*), 38
create_x3d_res_stage() (*in module py-*

torchvideo.models.x3d), 38
create_x3d_stem() (*in module pytorchvideo.models.x3d*), 36
crop_boxes() (*in module pytorchvideo.transforms.functional*), 92
CutMix (*class in pytorchvideo.transforms*), 80

D

DetectionBBoxNetwork (*class in pytorchvideo.models.net*), 25
Dict (*in module pytorchvideo.transforms*), 84
Div255 (*class in pytorchvideo.transforms*), 84
div_255() (*in module pytorchvideo.transforms.functional*), 93
DomsevFrameDataset (*class in pytorchvideo.data*), 68
DomsevVideoDataset (*class in pytorchvideo.data*), 69

E

EpicKitchenForecasting (*class in pytorchvideo.data*), 71
EpicKitchenRecognition (*class in pytorchvideo.data*), 71

F

forward() (*pytorchvideo.layers.fusion.ConcatFusion method*), 100
forward() (*pytorchvideo.layers.fusion.ReduceFusion method*), 101
forward() (*pytorchvideo.layers.fusion.TemporalConcatFusion method*), 101
forward() (*pytorchvideo.layers.positional_encoding.SpatioTemporalClsPositionalEncoding method*), 103
forward() (*pytorchvideo.layers.squeeze_excitation.SqueezeAndExcitationLayer2D method*), 104
forward() (*pytorchvideo.models.byol.BYOL method*), 54
forward() (*pytorchvideo.models.head.ResNetRoIHead method*), 30
forward() (*pytorchvideo.models.masked_multistream.LearnMaskedDefault method*), 57
forward() (*pytorchvideo.models.masked_multistream.LSTM method*), 57
forward() (*pytorchvideo.models.masked_multistream.MaskedTemporalPooling method*), 56
forward() (*pytorchvideo.models.masked_multistream.TransposeMultiheadAttention method*), 56
forward() (*pytorchvideo.models.masked_multistream.TransposeTransformerEncoder method*), 58
forward() (*pytorchvideo.models.memory_bank.MemoryBank method*), 55
forward() (*pytorchvideo.models.net.DetectionBBoxNetwork method*), 25
forward() (*pytorchvideo.models.simclr.SimCLR method*), 53
forward() (*pytorchvideo.transforms.ConvertUint8ToFloat method*), 84
forward() (*pytorchvideo.transforms.CutMix method*), 80
forward() (*pytorchvideo.transforms.Div255 method*), 84
forward() (*pytorchvideo.transforms.MixUp method*), 81
forward() (*pytorchvideo.transforms.MixVideo method*), 80
forward() (*pytorchvideo.transforms.Normalize method*), 85
forward() (*pytorchvideo.transforms.OpSampler method*), 85
forward() (*pytorchvideo.transforms.Permute method*), 85
forward() (*pytorchvideo.transforms.RandomShortSideScale method*), 86
forward() (*pytorchvideo.transforms.ShortSideScale method*), 86
forward() (*pytorchvideo.transforms.UniformTemporalSubsample method*), 87
forward() (*pytorchvideo.transforms.UniformTemporalSubsampleRepeat method*), 87
forward_backbone() (*pytorchvideo.models.byol.BYOL method*), 54
forward_backbone_mmt() (*pytorchvideo.models.byol.BYOL method*), 54
FuseFastToSlow (*class in pytorchvideo.models.slowfast*), 49

G

get_mmt() (*pytorchvideo.models.byol.BYOL method*), 54

H

Hmdb51() (*in module pytorchvideo.data*), 71
horizontal_flip_with_boxes() (*in module pytorchvideo.transforms.functional*), 91

K

Kinetics() (*in module pytorchvideo.data*), 71

L

LabeledVideoDataset() (*in module pytorchvideo.data*), 73
LabeledVideoPaths (*class in pytorchvideo.data*), 72
LearnMaskedDefault (*class in pytorchvideo.models.masked_multistream*), 56
List (*in module pytorchvideo.transforms*), 84

LSTM (*class in pytorchvideo.models.masked_multistream*), 57

M

make_clip_sampler() (*in module pytorchvideo.data*), 67
make_fusion_layer() (*in module pytorchvideo.layers.fusion*), 100
make_multilayer_perceptron() (*in module pytorchvideo.layers.mlp*), 101
MaskedMultiPathWay (*class in pytorchvideo.models.masked_multistream*), 58
MaskedSequential (*class in pytorchvideo.models.masked_multistream*), 58
MaskedTemporalPooling (*class in pytorchvideo.models.masked_multistream*), 56
MemoryBank (*class in pytorchvideo.models.memory_bank*), 55
MixUp (*class in pytorchvideo.transforms*), 80
MixVideo (*class in pytorchvideo.transforms*), 79
module
    pytorchvideo.data, 65
    pytorchvideo.layers.batch_norm, 97
    pytorchvideo.layers.convolutions, 98
    pytorchvideo.layers.fusion, 100
    pytorchvideo.layers.mlp, 101
    pytorchvideo.layers.nonlocal_net, 102
    pytorchvideo.layers.positional_encoding, 103
    pytorchvideo.layers.squeeze_excitation, 104
    pytorchvideo.layers.swish, 103
    pytorchvideo.models.byol, 54
    pytorchvideo.models.csn, 34
    pytorchvideo.models.head, 25
    pytorchvideo.models.masked_multistream, 56
    pytorchvideo.models.memory_bank, 55
    pytorchvideo.models.net, 24
    pytorchvideo.models.r2plus1d, 50
    pytorchvideo.models.resnet, 7
    pytorchvideo.models.simclr, 53
    pytorchvideo.models.slowfast, 44
    pytorchvideo.models.stem, 31
    pytorchvideo.models.x3d, 36
    pytorchvideo.transforms, 79
    pytorchvideo.transforms.functional, 87
MultiPathWayWithFuse (*class in pytorchvideo.models.net*), 25

N

NaiveSyncBatchNorm1d (*class in pytorchvideo.layers.batch_norm*), 97
NaiveSyncBatchNorm2d (*class in pytorchvideo.layers.batch_norm*), 97
NaiveSyncBatchNorm3d (*class in pytorchvideo.layers.batch_norm*), 97
Net (*class in pytorchvideo.models.net*), 24
NonLocal (*class in pytorchvideo.layers.nonlocal_net*), 102
Normalize (*class in pytorchvideo.transforms*), 84
NUM_CLASSES (*pytorchvideo.data.Charades attribute*), 66
num_videos() (*pytorchvideo.data.LabeledVideoDataset property*), 72

O

OpSampler (*class in pytorchvideo.transforms*), 85
output_dim() (*pytorchvideo.layers.fusion.ConcatFusion property*), 100
output_dim() (*pytorchvideo.layers.fusion.ReduceFusion property*), 101
output_dim() (*pytorchvideo.layers.fusion.TemporalConcatFusion property*), 101

P

PatchEmbed (*class in pytorchvideo.models.stem*), 33
Permute (*class in pytorchvideo.transforms*), 85
PoolConcatPathway (*class in pytorchvideo.models.slowfast*), 49
PositionalEncoding (*class in pytorchvideo.layers.positional_encoding*), 103
ProjectedPool (*class in pytorchvideo.models.x3d*), 42
pytorchvideo.data
    module, 65
pytorchvideo.layers.batch_norm
    module, 97
pytorchvideo.layers.convolutions
    module, 98
pytorchvideo.layers.fusion
    module, 100
pytorchvideo.layers.mlp
    module, 101
pytorchvideo.layers.nonlocal_net
    module, 102
pytorchvideo.layers.positional_encoding
    module, 103
pytorchvideo.layers.squeeze_excitation
    module, 104
pytorchvideo.layers.swish
    module, 103
pytorchvideo.models.byol

module, 54
pytorchvideo.models.csn
module, 34
pytorchvideo.models.head
module, 25
pytorchvideo.models.masked_multistream
module, 56
pytorchvideo.models.memory_bank
module, 55
pytorchvideo.models.net
module, 24
pytorchvideo.models.r2plus1d
module, 50
pytorchvideo.models.resnet
module, 7
pytorchvideo.models.simclr
module, 53
pytorchvideo.models.slowfast
module, 44
pytorchvideo.models.stem
module, 31
pytorchvideo.models.x3d
module, 36
pytorchvideo.transforms
module, 79
pytorchvideo.transforms.functional
module, 87

R

RandAugment (*class in pytorchvideo.transforms*), 81
random_crop_with_boxes() (*in module pytorchvideo.transforms.functional*), 90
random_resized_crop() (*in module pytorchvideo.transforms.functional*), 92
random_short_side_scale_with_boxes() (*in module pytorchvideo.transforms.functional*), 89
RandomClipSampler (*class in pytorchvideo.data*), 66
RandomResizedCrop (*class in pytorchvideo.transforms*), 85
RandomShortSideScale (*class in pytorchvideo.transforms*), 86
ReduceFusion (*class in pytorchvideo.layers.fusion*), 101
RemoveKey (*class in pytorchvideo.transforms*), 86
ResBlock (*class in pytorchvideo.models.resnet*), 22
ResNetBasicHead (*class in pytorchvideo.models.head*), 29
ResNetBasicStem (*class in pytorchvideo.models.stem*), 33
ResNetRoIHead (*class in pytorchvideo.models.head*), 29
ResStage (*class in pytorchvideo.models.resnet*), 24

S

SeparableBottleneckBlock (*class in pytorchvideo.models.resnet*), 22
SequencePool (*class in pytorchvideo.models.head*), 25
short_side_scale() (*in module pytorchvideo.transforms.functional*), 88
short_side_scale_with_boxes() (*in module pytorchvideo.transforms.functional*), 89
ShortSideScale (*class in pytorchvideo.transforms*), 86
sim_loss() (*pytorchvideo.models.byol.BYOL method*), 54
SimCLR (*class in pytorchvideo.models.simclr*), 53
SpatioTemporalClsPositionalEncoding (*class in pytorchvideo.layers.positional_encoding*), 103
SqueezeAndExcitationLayer2D (*class in pytorchvideo.layers.squeeze_excitation*), 104
SSv2 (*class in pytorchvideo.data*), 73
Swish (*class in pytorchvideo.layers.swish*), 103
SwishFunction (*class in pytorchvideo.layers.swish*), 103

T

TemporalConcatFusion (*class in pytorchvideo.layers.fusion*), 101
training (*pytorchvideo.transforms.ConvertUint8ToFloat attribute*), 84
training (*pytorchvideo.transforms.CutMix attribute*), 80
training (*pytorchvideo.transforms.Div255 attribute*), 84
training (*pytorchvideo.transforms.MixUp attribute*), 81
training (*pytorchvideo.transforms.MixVideo attribute*), 80
training (*pytorchvideo.transforms.Normalize attribute*), 85
training (*pytorchvideo.transforms.OpSampler attribute*), 85
training (*pytorchvideo.transforms.Permute attribute*), 85
training (*pytorchvideo.transforms.RandomResizedCrop attribute*), 86
training (*pytorchvideo.transforms.RandomShortSideScale attribute*), 86
training (*pytorchvideo.transforms.RemoveKey attribute*), 86
training (*pytorchvideo.transforms.ShortSideScale attribute*), 86
training (*pytorchvideo.transforms.UniformCropVideo attribute*), 87

training (*pytorchvideo.transforms.UniformTemporalSubsample attribute*), 87
training (*pytorchvideo.transforms.UniformTemporalSubsampleRepeated attribute*), 87
TransposeMultiheadAttention (*class in pytorchvideo.models.masked_multistream*), 56
TransposeTransformerEncoder (*class in pytorchvideo.models.masked_multistream*), 57
Tuple (*in module pytorchvideo.transforms*), 86
Tuple (*in module pytorchvideo.transforms.functional*), 87

U

Ucf101() (*in module pytorchvideo.data*), 74
uniform_crop() (*in module pytorchvideo.transforms.functional*), 90
uniform_crop_with_boxes() (*in module pytorchvideo.transforms.functional*), 91
uniform_temporal_subsample() (*in module pytorchvideo.transforms.functional*), 87
uniform_temporal_subsample_repeated() (*in module pytorchvideo.transforms.functional*), 88
UniformClipSampler (*class in pytorchvideo.data*), 67
UniformCropVideo (*class in pytorchvideo.transforms*), 86
UniformTemporalSubsample (*class in pytorchvideo.transforms*), 87
UniformTemporalSubsampleRepeated (*class in pytorchvideo.transforms*), 87
update_mmt() (*pytorchvideo.models.byol.BYOL method*), 54

V

video_sampler() (*pytorchvideo.data.Charades property*), 66
video_sampler() (*pytorchvideo.data.LabeledVideoDataset property*), 72
video_sampler() (*pytorchvideo.data.SSv2 property*), 74
VisionTransformerBasicHead (*class in pytorchvideo.models.head*), 30