firelight

Release 0.1.0

Roman Remme

Nov 11, 2019

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Firelight is a package for the visualization of pytorch tensors as images. It uses a flexible way of handling tensor shapes, which allows visualization of data of arbitrary dimensionality (See `firelight.utils.dim_utils` for details).

This documentation is work in progress, as is the package itself.

For now, have a look at the Examples, check out the currently available visualizers or read the docstrings.

CHAPTER

TWO

LIST OF VISUALIZERS

The following non-container visualizers are currently available. They all derive from *BaseVisualizer*.

CrackedEdgeVisualizer([width, connective_dims])	Visualize the boundaries of a segmentation.
DiagonalSplitVisualizer([offset])	Combine two input images, displaying one above and one below the diagonal.
IdentityVisualizer(**super_kwargs)	Visualizer that returns the tensor passed to it.
ImageVisualizer(**super_kwargs)	Same as *IdentityVisualizer*, but acting on 'image'.
InputVisualizer(**super_kwargs)	Same as *IdentityVisualizer*, but acting on 'input'.
MSEVisualizer(**super_kwargs)	Visualize the Mean Squared Error (MSE) between two tensors (e.g.
MaskVisualizer(mask_label, **super_kwargs)	Returns a mask that is 1 where the input image equals the mask label passed at initialization, and 0 elsewhere
MaskedPcaVisualizer([ignore_label, ...])	Version of PcaVisualizer that allows for an ignore mask.
NormVisualizer([order, dim])	Visualize the norm of a tensor, along a given direction (by default over the channels).
PcaVisualizer([n_components, joint_specs])	PCA Visualization of high dimensional embedding tensor.
PredictionVisualizer(**super_kwargs)	Same as *IdentityVisualizer*, but acting on 'prediction'.
RGBVisualizer(**super_kwargs)	Visualize the input tensor as RGB images.
SegmentationVisualizer(**super_kwargs)	Same as *IdentityVisualizer*, but acting on 'segmentation'.
TargetVisualizer(**super_kwargs)	Same as *IdentityVisualizer*, but acting on 'target'.
ThresholdVisualizer(threshold[, mode])	Returns a mask resulting from thresholding the input tensor.
TsneVisualizer([joint_dims, n_components])	tSNE Visualization of high dimensional embedding tensor.
UmapVisualizer([joint_dims, n_components, ...])	UMAP Visualization of high dimensional embedding tensor.
UpsamplingVisualizer(specs[, shape, factors])	Upsample a tensor along a list of axis (specified via specs) to a specified shape, by a list of specified factors or the shape of a reference tensor (given as an optional argument to visualize).

These are the available visualizers combining multiple visualizations. Their base class is the *ContainerVisualizer*.

ColumnVisualizer(*super_args, **super_kwargs)	Visualizer that arranges outputs of child visualizers in a grid of images, with different child visualizations stacked horizontally (side by side).
ImageGridVisualizer([row_specs, ...])	Visualizer that arranges outputs of child visualizers in a grid of images.
OverlayVisualizer(*super_args, **super_kwargs)	Visualizer that overlays the outputs of its child visualizers on top of each other, using transparency based on the alpha channel.
RiffleVisualizer([riffle_dim])	Riffles the outputs of its child visualizers along specified dimension.
RowVisualizer(*super_args, **super_kwargs)	Visualizer that arranges outputs of child visualizers in a grid of images, with different child visualizations stacked vertically.
StackVisualizer([stack_dim])	Stacks the outputs of its child visualizers along specified dimension.

CHAPTER

THREE

EXAMPLES

3.1 Using Firelight

Note: Click *here* to download the full example code

3.1.1 Realistic Example

A close-to-real-world example of how to use firelight.

First of all, let us get some mock data to visualize. We generate the following tensors:

- `input` of shape (B, D, H, W), some noisy raw data,
- `target` of shape (B, D, H, W), the ground truth foreground background segmentation,
- `prediction` of shape (B, D, H, W), the predicted foreground probability,
- `embedding` of shape (B, D, C, H, W), a tensor with an additional channel dimension, as for example intermediate activations of a neural network.

```
import numpy as np
import torch
from skimage.data import binary_blobs
from skimage.filters import gaussian


def get_example_states():
    # generate some toy foreground/background segmentation
    batchsize = 5  # we will only visualize 3 of the 5samples
    size = 64
    target = np.stack([binary_blobs(length=size, n_dim=3, blob_size_fraction=0.25,␣
↪volume_fraction=0.5, seed=i)
                       for i in range(batchsize)], axis=0).astype(np.float32)

    # generate toy raw data as noisy target
    sigma = 0.5
    input = target + np.random.normal(loc=0, scale=sigma, size=target.shape)

    # compute mock prediction as gaussian smoothing of input data
    prediction = np.stack([gaussian(sample, sigma=3, truncate=2.0) for sample in␣
↪input], axis=0)
    prediction = 10 * (prediction - 0.5)
```

(continues on next page)

(continued from previous page)

```
    # compute mock embedding (if you need an image with channels for testing)
    embedding = np.random.randn(prediction.shape[0], 16, *(prediction.shape[1:]))

    # put input, target, prediction in dictionary, convert to torch.Tensor, add␣
↪dimensionality labels ('specs')
    state_dict = {
        'input': (torch.Tensor(input).float(), 'BDHW'),  # Dimensions are B, D, H, W␣
↪= Batch, Depth, Height, Width
        'target': (torch.Tensor(target).float(), 'BDHW'),
        'prediction': (torch.Tensor(prediction).float(), 'BDHW'),
        'embedding': (torch.Tensor(embedding).float(), 'BCDHW'),
    }
    return state_dict


# Get the example state dictionary, containing the input, target, prediction.
states = get_example_states()

for name, (tensor, spec) in states.items():
    print(f'{name}: shape {tensor.shape}, spec {spec}')
```

Out:

```
input: shape torch.Size([5, 64, 64, 64]), spec BDHW
target: shape torch.Size([5, 64, 64, 64]), spec BDHW
prediction: shape torch.Size([5, 64, 64, 64]), spec BDHW
embedding: shape torch.Size([5, 16, 64, 64, 64]), spec BCDHW
```

The best way to construct a complex visualizer to show all the tensors in a structured manner is to use a configuration file.

We will use the following one:

```
RowVisualizer: # stack the outputs of child visualizers as rows of an image grid
  input_mapping:
    global: [B: ':3', D: '0:9:3'] # Show only 3 samples in each batch ('B'), and some␣
↪slices along depth ('D').
    prediction: [C: '0']  # Show only the first channel of the prediction

  pad_value: [0.2, 0.6, 1.0] # RGB color of separating lines
  pad_width: {B: 6, H: 0, W: 0, rest: 3} # Padding for batch ('B'), height ('H'),␣
↪width ('W') and other dimensions.

  visualizers:
    # First row: Ground truth
    - IdentityVisualizer:
        input: 'target' # show the target

    # Second row: Raw input
    - IdentityVisualizer:
        input: ['input', C: '0'] # Show the first channel ('C') of the input.
        cmap: viridis  # Name of a matplotlib colormap.

    # Third row: Prediction with segmentation boarders on top.
    - OverlayVisualizer:
        visualizers:
```

(continues on next page)

(continued from previous page)

```
          - CrackedEdgeVisualizer: # Show borders of target segmentation
              input_mapping:
                segmentation: 'target'
              width: 2
              opacity: 0.7 # Make output only partially opaque.
          - IdentityVisualizer: # prediction
              input_mapping:
                tensor: 'prediction'
              cmap: Spectral

    # Fourth row: Foreground probability, calculated by sigmoid on prediction
    - IdentityVisualizer:
        input_mapping: # the input to the visualizer can also be specified as a dict␣
↪under the key 'input mapping'.
          tensor: ['prediction', pre: 'sigmoid'] # Apply sigmoid function from torch.
↪nn.functional before visualize.
        value_range: [0, 1] # Scale such that 0 is white and 1 is black. If not␣
↪specified, whole range is used.

    # Fifth row: Visualize where norm of prediction is smaller than 2
    - ThresholdVisualizer:
        input_mapping:
          tensor:
            NormVisualizer: # Use the output of NormVisualizer as the input to␣
↪ThresholdVisualizer
              input: 'prediction'
              colorize: False
        threshold: 2
        mode: 'smaller'
```

Lets load the file and construct the visualizer using `get_visualizer`:

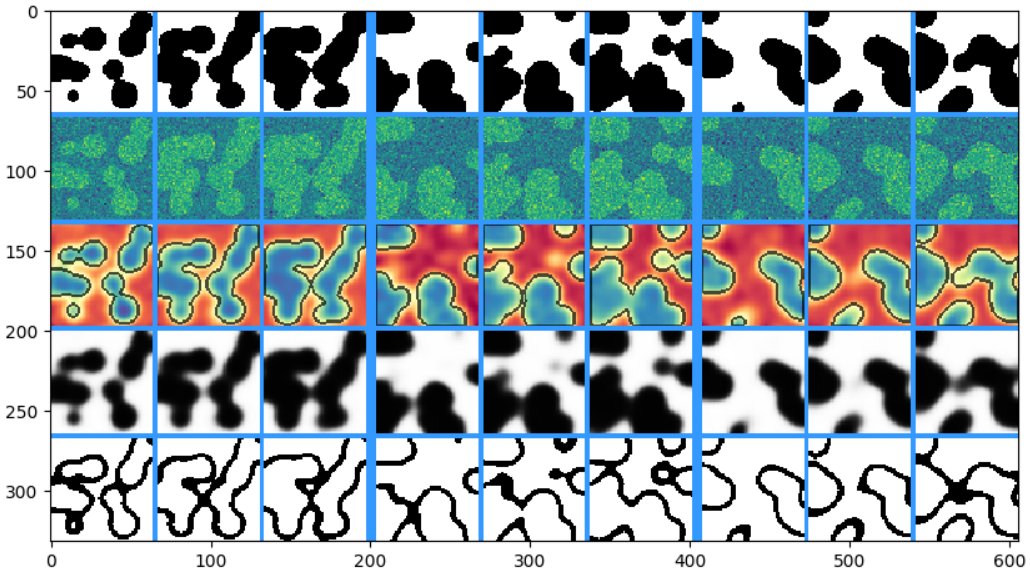
```
from firelight import get_visualizer
import matplotlib.pyplot as plt

# Load the visualizer, passing the path to the config file. This happens only once,␣
↪at the start of training.
visualizer = get_visualizer('example_config_0.yml')
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/firelight/checkouts/latest/firelight/
↪utils/io_utils.py:22: YAMLLoadWarning: calling yaml.load() without Loader=... is␣
↪deprecated, as the default Loader is unsafe. Please read https://msg.pyyaml.org/
↪load for full details.
  readict = yaml.load(f)
[+][2019-11-11 14:56:22,818][VISUALIZATION] Parsing RowVisualizer
[+][2019-11-11 14:56:22,818][VISUALIZATION] Parsing   IdentityVisualizer
[+][2019-11-11 14:56:22,818][VISUALIZATION] Parsing   IdentityVisualizer
[+][2019-11-11 14:56:22,818][VISUALIZATION] Parsing   OverlayVisualizer
[+][2019-11-11 14:56:22,818][VISUALIZATION] Parsing     CrackedEdgeVisualizer
[+][2019-11-11 14:56:22,825][VISUALIZATION] Parsing     IdentityVisualizer
[+][2019-11-11 14:56:22,826][VISUALIZATION] Parsing   IdentityVisualizer
[+][2019-11-11 14:56:22,826][VISUALIZATION] Parsing   ThresholdVisualizer
[+][2019-11-11 14:56:22,826][VISUALIZATION] Parsing     NormVisualizer
```

Now we can finally apply it on out mock tensors to get the visualization

```
# Call the visualizer.
image_grid = visualizer(**states)

# Log your image however you want.
plt.figure(figsize=(10, 6))
plt.imshow(image_grid.numpy())
```



Total running time of the script: (0 minutes 15.425 seconds)

3.2 Understanding Firelight

Note: Click *here* to download the full example code

3.2.1 SpecFunction Example

An example demonstrating the functionality of the `SpecFunction` class.

```
import torch
import matplotlib.pyplot as plt
from firelight.utils.dim_utils import SpecFunction
```

Let us define a function that takes in two arrays and masks one with the other:

```
class MaskArray(SpecFunction):
    def __init__(self, **super_kwargs):
```

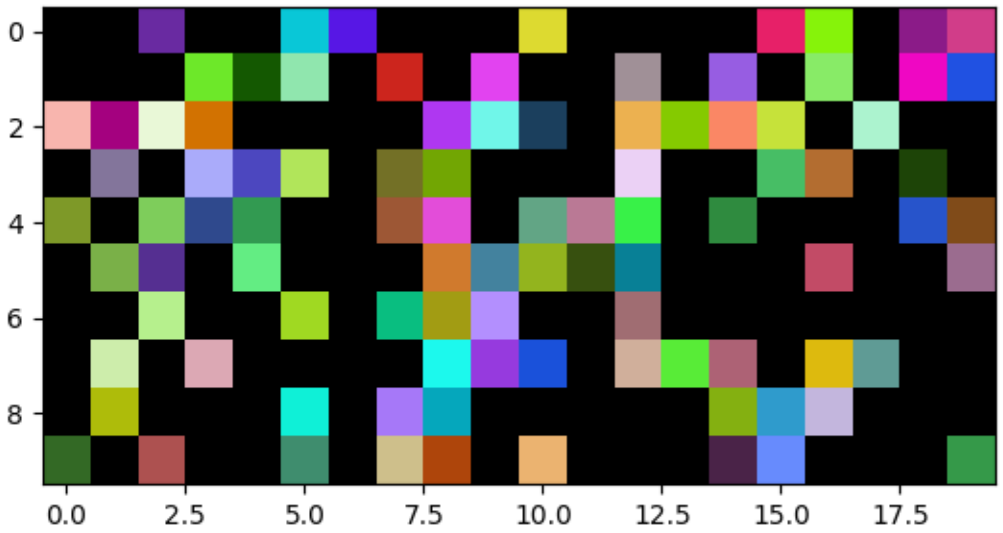
(continues on next page)

(continued from previous page)

```
        super(MaskArray, self).__init__(
            in_specs={'mask': 'B', 'array': 'BC'},
            out_spec='BC',
            **super_kwargs
        )

    def internal(self, mask, array, value=0.0):
        # The shapes are
        #   mask: (B)
        #   array: (B, C)
        # as specified in the init.

        result = array.clone()
        result[mask == 0] = value

        # the result has shape (B, C), as specified in the init.
        return result
```

We can now apply the function on inputs of arbitrary shape, such as images. The reshaping involved gets taken care of automatically:

```
W, H = 20, 10
inputs = {
    'array': (torch.rand(H, W, 3), 'HWC'),
    'mask': (torch.randn(H, W) > 0, 'HW'),
    'value': 0,
    'out_spec': 'HWC',
}

maskArrays = MaskArray()
result = maskArrays(**inputs)
print('output shape:', result.shape)

plt.imshow(result)
```



Out:

```
output shape: torch.Size([10, 20, 3])
```

Total running time of the script: (0 minutes 0.168 seconds)

CHAPTER

FOUR

FIRELIGHT PACKAGE

4.1 firelight.visualizers package

4.1.1 firelight.visualizers.base module

class firelight.visualizers.base.**BaseVisualizer**(*input=None*, *input_mapping=None*, *colorize=True*, *cmap=None*, *background_label=None*, *background_color=None*, *opacity=1.0*, *colorize_jointly=None*, *value_range=None*, *verbose=False*, *scaling_options=None*, ***super_kwargs*)

Bases: *firelight.utils.dim_utils.SpecFunction*

Base class for all visualizers. If you want to use outputs of other visualizers, derive from ContainerVisualizer instead.

Parameters

- **input** (*list or None*) – If the visualizer has one input only, this can be used to specify which state to pass (in the format of a value in input_mapping).
- **input_mapping** (*dict or list*) – Dictionary specifying slicing and renaming of states for visualization (see *apply_slice_mapping()*).
- **colorize** (*bool*) – If False, the addition/rescaling of a 'Color' dimension to RGBA in [0,1] is suppressed.
- **cmap** (*str or callable*) – If string, specifies the name of the matplotlib colormap to be used for colorization.

  If callable, must be a mapping from a [Batch x Pixels] to [Batch x Pixels x Color] numpy.ndarray used for colorization.
- **background_label** (*int or float*) – If specified, pixels with this value (after *visualize()*) will be colored with background_color.
- **background_color** (*float or list*) – Specifies the color for the background_label. Will be interpreted as grey-value if float, and RGB or RGBA if list of length 3 or 4 respectively.
- **opacity** (*float*) – Opacity of visualization, see colorization.py.

- **colorize_jointly** (*list of str*) – A list containing names of dimensions. Sets of data points separated only in these dimensions will be scaled equally at colorization (such that they lie in [0, 1]). Not used if 'value_range' is specified.

  Default: `['W', 'H', 'D']` (standing for Width, Height, Depth)

  Examples:

  - `color_jointly = ['W', 'H']` : Scale each image separately
  - `color_jointly = ['B', 'W', 'H']` : Scale images corresponding to different samples in the batch equally, such that their intensities are comparable

- **value_range** (*List*) – If specified, the automatic scaling for colorization is overridden. Has to have 2 elements. The interval [value_range[0], value_range[1]] will be mapped to [0, 1] by a linear transformation.

  Examples:

  - If your network has the sigmoid function as a final layer, the data does not need to be scaled further. Hence `value_range = [0, 1]` should be specified.
  - If your network produces outputs normalized between -1 and 1, you could set `value_range = [-1, 1]`.

- **verbose** (*bool*) – If true, information about the state dict will be printed during visualization.
- ****super_kwargs** – Arguments passed to the constructor of SpecFunction, above all the dimension names of inputs and output of visualize()

__call__(*return_spec=False*, ***states*)

Visualizes the data specified in the state dictionary, following these steps:

- Apply the input mapping (using `apply_input_mapping()`),
- Reshape the states needed for visualization as specified by in_specs at initialization. Extra dimensions are 'put into' the batch dimension, missing dimensions are added (This is handled in the base class, *firelight.utils.dim_utils.SpecFunction*)
- Apply *visualize()*,
- Reshape the result, with manipulations applied on the input in reverse,
- If not disabled by setting `colorize=False`, colorize the result, leading to RGBA output with values in $[0, 1]$.

Parameters

- **return_spec** (*bool*) – If true, a list containing the dimension names of the output is returned additionally
- **states** (*dict*) – Dictionary including the states to be visualized.

Returns result (*torch.Tensor or (torch.Tensor, list)*) – Either only the resulting visualization, or a tuple of the visualization and the corresponding spec (depending on the value of `return_spec`).

internal(**args*, ***kwargs*)

Function that is being wrapped.

visualize(***states*)

Main visualization function that all subclasses have to implement.

Parameters **states** (*dict*) – Dictionary containing states used for visualization. The states in in_specs (specified at initialization) will have dimensionality and order of dimensions as specified there.

Returns *torch.Tensor*

class firelight.visualizers.base.**ContainerVisualizer**(*visualizers*, *in_spec*, *out_spec*, *extra_in_specs=None*, *input_mapping=None*, *equalize_visualization_shapes=True*, *colorize=False*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Base Class for visualizers combining the outputs of other visualizers.

Parameters

- **visualizers** (*List of BaseVisualizer*) – Child visualizers whose outputs are to be combined.
- **in_spec** (*List of str*) – List of dimension names. The outputs of all the child visualizers will be brought in this shape to be combined (in combine()).
- **out_spec** (*List of str*) – List of dimension names of the output of combine().
- **extra_in_specs** (*dict*) – Dictionary containing lists of dimension names for inputs of combine that are directly taken from the state dictionary and are not the output of a child visualizer.
- **input_mapping** (*dict*) – Dictionary specifying slicing and renaming of states for visualization (see *apply_slice_mapping()*).
- **equalize_visualization_shapes** (*bool*) – If true (as per default), the shapes of the outputs of child visualizers will be equalized by repeating along dimensions with shape mismatches. Only works if the maximum size of each dimension is divisible by the sizes of all the child visualizations in that dimension.
- **colorize** (*bool*) – If False, the addition/rescaling of a 'Color' dimension to RGBA in [0,1] is suppressed.
- ****super_kwargs** – Dictionary specifying other arguments of BaseVisualizer.

__call__(*return_spec=False*, ***states*)

Like call in BaseVisualizer, but computes visualizations for all child visualizers first, which will be passed to combine() (equivalent of visualize for BaseVisualizer).

Parameters

- **return_spec** (*bool*) – If true, a list containing the dimension names of the output is returned additionally
- **states** (*dict*) – Dictionary including the states to be visualized.

Returns torch.Tensor or (torch.Tensor, list), depending on the value of `return_spec`.

combine(**visualizations*, ***extra_states*)

Main visualization function that all subclasses have to implement.

Parameters

- **visualizations** (`list` of `torch.Tensor`) – List containing the visualizations from the child visualizers. Their dimensionality and order of dimensions will be as specified in in_spec at initialization.

- **extra_states** (*dict*) – Dictionary containing extra states (not outputs of child visualizers) used for visualization. The states in `extra_in_specs` (specified at initialization) will have dimensionality and order of dimensions as specified there.

Returns *torch.Tensor*

internal(***states*)
Function that is being wrapped.

firelight.visualizers.base.**DEFAULT_SPECS = {3: ['B', 'H', 'W'], 4: ['B', 'C', 'H', 'W'],**
The default ways to label the dimensions depending on dimensionality.

- 3 Axes : (B, H, W)
- 4 Axes : (B, C, H, W)
- 5 Axes : (B, C, D, H, W)
- 6 Axes : (B, C, T, D, H, W)

Type dict

firelight.visualizers.base.**apply_slice_mapping**(*mapping*, *states*, *include_old_states=True*)
Add/Replace tensors in the dictionary 'states' as specified with the dictionary 'mapping'. Each key in mapping corresponds to a state in the resulting dictionary, and each value describes:

- from which tensors in *states* this state is grabbed (e.g. `['prediction']`)
- if a list of tensors is grabbed: which list index should be used (e.g `'[index': 0]`)
- what slice of the grabbed tensor should be used (e.g `['B': '0', 'C': '0:3']`). For details see *parse_named_slicing()*.
- what function in torch.nn.functional should be applied to the tensor after the slicing (e.g. `['pre': 'sigmoid']`). See *parse_pre_func()* for details.

These arguments can be specified in one dictionary or a list of length one dictionaries.

Parameters

- **mapping** (*dict*) – Dictionary describing the mapping of states
- **states** (*dict*) – Dictionary of states to be mapped. Values must be either tensors, or tuples of the form (tensor, spec).
- **include_old_states** (*bool*) – Whether or not to include states in the ouput dictionary, on which no operations were performed.

Returns *dict* – Dictionary of mapped states

firelight.visualizers.base.**get_single_key_value_pair**(*d*)
Get the key and value of a length one dictionary.

Parameters **d** (*dict*) – Single element dictionary to split into key and value.

Returns *tuple* – of length 2, containing the key and value

Examples

```
>>> d = dict(key='value')
>>> get_single_key_value_pair(d)
('key', 'value')
```

firelight.visualizers.base.**list_of_dicts_to_dict**(*list_of_dicts*)
Convert a list of one element dictionaries to one dictionary.

Parameters **list_of_dicts** (list of dict) – List of one element dictionaries to merge.

Returns *dict*

Examples

```
>>> list_of_dicts_to_dict([{'a': 1}, {'b': 2}])
{'a': 1, 'b': 2}
```

firelight.visualizers.base.**parse_named_slicing**(*slicing*, *spec*)
Parse a slicing as a list of slice objects.

Parameters

- **slicing** (*str or list or dict*) – Specifies the slicing that is to be applied. Depending on the type:
  - str: slice strings joined by ',', In this case, spec will be ignored. (e.g. '0, 1:4')
  - list: has to be list of one element dictionaries, that will be converted to one dict with *list_of_dicts_to_dict()*
  - dict: keys are dimension names, values corresponding slices (as strings) (e.g. {'B': '0', 'C': '1:4'})
- **spec** (*list*) – List of names of dimensions of the tensor that is to be sliced.

Returns *list* – List of slice objects

Examples

Three ways to encode the same slicing:

```
>>> parse_named_slicing(':5, :, 1', ['A', 'B', 'C'])
[slice(None, 5, None), slice(None, None, None), slice(1, 2, None)]
>>> parse_named_slicing({'A': ':5', 'C': '1'}, ['A', 'B', 'C'])
[slice(None, 5, None), slice(None, None, None), slice(1, 2, None)]
>>> parse_named_slicing([{'A': ':5'}, {'C': '1'}], ['A', 'B', 'C'])
[slice(None, 5, None), slice(None, None, None), slice(1, 2, None)]
```

firelight.visualizers.base.**parse_pre_func**(*pre_info*)
Parse the pre-processing function for an input to a visualizer (as given by the 'pre' key in the input_mapping).

Parameters **pre_info** (*list, dict or str*) – Depending on the type:

- str: Name of function in torch, torch.nn.functional, or dotted path to function.
- list: List of functions to be applied in succession. Each will be parsed by this function.
- dict: Has to have length one. The key is the name of a function (see str above), the value specifies additional arguments supplied to that function (apart from the tensor that will be transformed). Either positional arguments can be specified as a list, or keyword arguments as a dictionary.

Examples:

- pre_info = 'sigmoid'

- `pre_info = {'softmax': [1]}}`
- `pre_info = {'softmax': {dim: 0}}}`

Returns *Callable* – The parsed pre-processing function.

`firelight.visualizers.base.`**`parse_slice`**(*slice_string*)

Parse a slice given as a string.

Parameters **`slice_string`** (*str*) – String describing the slice. Format as in fancy indexing: 'start:stop:end'.

Returns *slice*

Examples

Everything supported in fancy indexing works here, too:

```
>>> parse_slice('5')
slice(5, 6, None)
>>> parse_slice(':5')
slice(None, 5, None)
>>> parse_slice('5:')
slice(5, None, None)
>>> parse_slice('2:5')
slice(2, 5, None)
>>> parse_slice('2:5:3')
slice(2, 5, 3)
>>> parse_slice('::3')
slice(None, None, 3)
```

4.1.2 firelight.visualizers.colorization module

`class` `firelight.visualizers.colorization.`**`Colorize`**(*background_label=None*, *background_color=None*, *opacity=1.0*, *value_range=None*, *cmap=None*, *colorize_jointly=None*, *scaling_options=None*)

Bases: *firelight.utils.dim_utils.SpecFunction*

Constructs a function used for the colorization / color normalization of tensors. The output tensor has a length 4 RGBA output dimension labeled 'Color'.

If the input tensor is continuous, a color dimension will be added if not present already. Then, it will be scaled to $[0, 1]$. How exactly the scaling is performed can be influenced by the parameters below.

If the tensor consists of only ones and zeros, the ones will become black and the zeros transparent white.

If the input tensor is discrete including values different to zero and one, it is assumed to be a segmentation and randomly colorized.

Parameters

- **`background_label`** (*int or tuple, optional*) – Value of input tensor that will be colored with background color.
- **`background_color`** (*int or tuple, optional*) – Color that will be assigned to regions of the input having the value background_label.

- **opacity** (*float, optional*) – Multiplier that will be applied to alpha channel. Useful to blend images with *OverlayVisualizer*.
- **value_range** (*tuple, optional*) – Range the input data will lie in (e.g. $[-1, 1]$ for l2-normalized vectors). This range will be mapped linearly to the unit interval $[0, 1]$. If not specified, the output data will be scaled to use the full range $[0, 1]$.
- **cmap** (*str or callable or None, optional*) – If str, has to be the name of a matplotlib colormap, to be used to color grayscale data.

  If callable, has to be function that adds a RGBA color dimension at the end, to an input `numpy.ndarray` with values between 0 and 1.

  If None, the output will be grayscale with the intensity in the opacity channel.
- **colorize_jointly** (*list, optional*) – List of the names of dimensions that should be colored jointly. Default: `['W', 'H', 'D']`.

  Data points separated only in these dimensions will be scaled equally. See *StackVisualizer* for an example usage.

add_alpha(*img*)

internal(*tensor*)
If not present, add a color channel to tensor. Scale the colors using *Colorize.normalize_colors()*.

normalize_colors(*tensor*)
Scale each color channel individually to use the whole extend of $[0, 1]$. Uses *ScaleTensor*.

class firelight.visualizers.colorization.**ScaleTensor**(*invert=False*, *value_range=None*, *scale_robust=False*, *quantiles=(0.05, 0.95)*, *keep_centered=False*)

Bases: *firelight.utils.dim_utils.SpecFunction*

Parameters

- **invert** (*bool*) – Whether the input should be multiplied with -1.
- **value_range** (*[float, float] or None, optional*) – If specified, tensor will be scaled by a linear map that maps `value_range[0]` will be mapped to 0, and `value_range[1]` will be to 1.
- **scale_robust** (*bool, optional*) – Whether outliers in the input should be ignored in the scaling.

  Has no effect if `value_range` is specified.
- **quantiles** (*(float, float), optional*) – Values under the first and above the second quantile are considered outliers for robust scaling.

  Ignored if `scale_robust` is False or `value_range` is specified.
- **keep_centered** (*bool, optional*) – Whether the scaling should be symmetric in the sense that (if the scaling function is f):

$$f(-x) = 0.5 - f(x)$$

  This can be useful in combination with diverging colormaps.

internal(*tensor*)
Scales the input tensor to the interval $[0, 1]$.

quantile_scale(*tensor*, *quantiles=None*, *return_params=False*)
Scale tensor linearly, such that the `quantiles[i]`-quantile ends up on `quantiles[i]`.

scale_tails(*tensor*)
Scale the tails (the elements below `self.quantiles[0]` and the ones above `self.quantiles[1]`) linearly to make all values lie in $[0, 1]$.

firelight.visualizers.colorization.**add_alpha**(*img*)
Adds a totally opaque alpha channel to a tensor, whose last axis corresponds to RGB color.

Parameters **img** (*torch.Tensor*) – The RGB image.

Returns *torch.Tensor* – The resulting RGBA image.

firelight.visualizers.colorization.**colorize_segmentation**(*seg*, *ignore_label=None*, *ignore_color=(0, 0, 0)*)
Randomly colorize a segmentation with a set of distinct colors.

Parameters

- **seg** (*numpy.ndarray*) – Segmentation to be colorized. Can have any shape, but data type must be discrete.
- **ignore_label** (*int*) – Label of segment to be colored with ignore_color.
- **ignore_color** (*tuple*) – RGB color of segment labeled with ignore_label.

Returns *numpy.ndarray* – The randompy colored segmentation. The RGB channels are in the last axis.

firelight.visualizers.colorization.**from_matplotlib_cmap**(*cmap*)
Converts the name of a matplotlib colormap to a colormap function that can be applied to a `numpy.ndarray`.

Parameters **cmap** (*str*) – Name of the matplotlib colormap

Returns *callable* – A function that maps greyscale arrays to RGBA.

firelight.visualizers.colorization.**get_distinct_colors**(*n*, *min_sat=0.5*, *min_val=0.5*)
Generates a list of distinct colors, evenly separated in HSV space.

Parameters

- **n** (*int*) – Number of colors to generate.
- **min_sat** (*float*) – Minimum saturation.
- **min_val** (*float*) – Minimum brightness.

Returns *numpy.ndarray* – Array of shape (n, 3) containing the generated colors.

firelight.visualizers.colorization.**hsv_to_rgb**(*h*, *s*, *v*)
Converts a color from HSV to RGB

Parameters

- **h** (*float*) –
- **s** (*float*) –
- **v** (*float*) –

Returns *numpy.ndarray* – The converted color in RGB space.

4.1.3 firelight.visualizers.container_visualizers module

class firelight.visualizers.container_visualizers.**ColumnVisualizer**(**super_args*, ***super_kwargs*)

Bases: *firelight.visualizers.container_visualizers.ImageGridVisualizer*

Visualizer that arranges outputs of child visualizers in a grid of images, with different child visualizations stacked horizontally (side by side). For more options, see ImageGridVisualizer

Parameters

- ***super_args** –
- ****super_kwargs** –

class firelight.visualizers.container_visualizers.**ImageGridVisualizer**(*row_specs=('H', 'C', 'V')*, *column_specs=('W', 'D', 'T', 'B')*, *pad_width=1*, *pad_value=0.5*, *upsampling_factor=1*, **super_args*, ***super_kwargs*)

Bases: *firelight.visualizers.base.ContainerVisualizer*

Visualizer that arranges outputs of child visualizers in a grid of images.

Parameters

- **row_specs** (*list*) – List of dimension names. These dimensions of the outputs of child visualizers will be put into the height dimension of the resulting image, according to the order in the list.

  In other words, data points only separated in dimensions at the beginning of this list will be right next to each other, while data points separated in dimensions towards the back will be further away from each other in the output image.

  A special dimension name is 'V' (for visualizers). It stands for the dimension differentiating between the child visualizers.

  Example: Given the tensor `[[1, 2 , 3 ], [10, 20, 30]]` with shape (2, 3) and dimension names `['A', 'B']`, this is the order of the rows, depending on the specified row_specs (suppose `column_specs = []`):

  – If `row_specs = ['B', 'A']`, the output will be `[1, 2, 3, 10, 20, 30]`

  – If `row_specs = ['A', 'B']`, the output will be `[1, 10, 2, 20, 3, 30]`

- **column_specs** (*list*) – As row_specs but for columns of resulting image. Each dimension of child visualizations has to either occur in row_specs or column_specs. The intersection of row_specs and column specs has to be empty.

- **pad_width** (*int or dict*) – Determines the width of padding when concatenating images. Depending on type:
  - **int: Padding will have this width for concatenations along all dimensions, apart from H and W (no** padding between adjacent pixels in image)
  - **dict: Keys are dimension names, values the padding width when concatenating along them. Special key** 'rest' determines default value if given (otherwise no padding is used as default).
- **pad_value** (*int or dict*) – Determines the color of padding when concatenating images. Colors can be given as floats (gray values) or list of RGB / RGBA values. If dict, interpreted as pad_width
- **upsampling_factor** (*int*) – The whole resulting image grid will be upsampled by this factor. Useful when visualizing small images in tensorboard, but can lead to unnecessarily big file sizes.
- ***super_args** (*list*) –
- ****super_kwargs** (*dict*) –

get_pad_kwargs(*spec*)

internal(**args*, *return_spec=False*, ***states*)
Function that is being wrapped.

visualization_to_image(*visualization*, *spec*)

class firelight.visualizers.container_visualizers.**OverlayVisualizer**(**super_args*, ***super_kwargs*)

Bases: *firelight.visualizers.base.ContainerVisualizer*

Visualizer that overlays the outputs of its child visualizers on top of each other, using transparency based on the alpha channel. The output of the first child visualizer will be on the top, the last on the bottom.

Parameters

- ***super_args** –
- ****super_kwargs** –

combine(**visualizations*, ***_*)
Main visualization function that all subclasses have to implement.

Parameters

- **visualizations** (list of torch.Tensor) – List containing the visualizations from the child visualizers. Their dimensionality and order of dimensions will be as specified in in_spec at initialization.
- **extra_states** (*dict*) – Dictionary containing extra states (not outputs of child visualizers) used for visualization. The states in `extra_in_specs` (specified at initialization) will have dimensionality and order of dimensions as specified there.

Returns *torch.Tensor*

class firelight.visualizers.container_visualizers.**RiffleVisualizer**(*riffle_dim='C'*, **super_args*, ***super_kwargs*)

Bases: *firelight.visualizers.base.ContainerVisualizer*

Riffles the outputs of its child visualizers along specified dimension.

For a way to also scale target and prediction equally, have a look at StackVisualizer (if the range of values is known, you can also just use value_range: [a, b] for the child visualizers

Parameters

- **riffle_dim** (*str*) – Name of dimension which is to be riffled
- ***super_args** –
- ****super_kwargs** –

Examples

Riffle the channels of a multidimensional target and prediction, such that corresponding images are closer spatially. A possible configuration file would look like this:

```
RiffleVisualizer:
    riffle_dim: 'C'
    visualizers:
        - ImageVisualizer:
            input_mapping:
                image: 'target'
        - ImageVisualizer:
            input_mapping:
                image: 'prediction'
```

combine(**visualizations*, ***_*)

Main visualization function that all subclasses have to implement.

Parameters

- **visualizations** (list of torch.Tensor) – List containing the visualizations from the child visualizers. Their dimensionality and order of dimensions will be as specified in in_spec at initialization.
- **extra_states** (*dict*) – Dictionary containing extra states (not outputs of child visualizers) used for visualization. The states in extra_in_specs (specified at initialization) will have dimensionality and order of dimensions as specified there.

Returns *torch.Tensor*

class firelight.visualizers.container_visualizers.**RowVisualizer**(**super_args*, ***super_kwargs*)

Bases: *firelight.visualizers.container_visualizers.ImageGridVisualizer*

Visualizer that arranges outputs of child visualizers in a grid of images, with different child visualizations stacked vertically. For more options, see ImageGridVisualizer

Parameters

- ***super_args** –
- ****super_kwargs** –

class firelight.visualizers.container_visualizers.**StackVisualizer**(*stack_dim='S'*, **super_args*, ***super_kwargs*)

Bases: *firelight.visualizers.base.ContainerVisualizer*

Stacks the outputs of its child visualizers along specified dimension.

Parameters

- **stack_dim** (*str*) – Name of new dimension along which the child visualizations will be stacked. None of the child visualizations should have this dimension.
- ***super_args** –
- ****super_kwargs** –

Example

Stack a multidimensional target and prediction along an extra dimension, e.g. 'TP'. In order to make target and prediction images comparable, disable colorization in the child visualizers and colorize only in the StackVisualizer, jointly coloring along 'TP', thus scaling target and prediction images by the same factors. The config would look like this:

```
StackVisualizer:
    stack_dim: 'TP'
    colorize: True
    color_jointly: ['H', 'W', 'TP']  # plus other dimensions you want to scale␣
↪equally, e.g. D = depth
    visualizers:
        - ImageVisualizer:
            input_mapping:
                image: 'target'
            colorize = False
        - ImageVisualizer:
            input_mapping:
                image: 'target'
            colorize = True
```

combine(**visualizations*, ***_*)

Main visualization function that all subclasses have to implement.

Parameters

- **visualizations** (list of torch.Tensor) – List containing the visualizations from the child visualizers. Their dimensionality and order of dimensions will be as specified in in_spec at initialization.
- **extra_states** (*dict*) – Dictionary containing extra states (not outputs of child visualizers) used for visualization. The states in `extra_in_specs` (specified at initialization) will have dimensionality and order of dimensions as specified there.

Returns *torch.Tensor*

4.1.4 firelight.visualizers.visualizers module

class firelight.visualizers.visualizers.**CrackedEdgeVisualizer**(*width=1*, *connective_dims=('H', 'W')*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Visualize the boundaries of a segmentation.

Parameters

- **width** (*int, optional*) – width of the boundary in every direction
- **connective_dims** (*tuple, optional*) – Tuple of axis names. Edges in those axes will be shown.

  E.g. use `('D', 'H', 'W')` to visualize edges in 3D.
- ****super_kwargs** –

make_pad_slice_tuples()

visualize(*segmentation*, **_)

class firelight.visualizers.visualizers.**DiagonalSplitVisualizer**(*offset=0*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Combine two input images, displaying one above and one below the diagonal.

Parameters

- **offset** (*int, optional*) – The diagonal along which the image will be split is shifted by offset.
- ****super_kwargs** –

visualize(*upper_right_image*, *lower_left_image*, **_)

class firelight.visualizers.visualizers.**IdentityVisualizer**(***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Visualizer that returns the tensor passed to it. Useful to visualize each channel of a tensor as a separate image.

visualize(*tensor*, **_)

class firelight.visualizers.visualizers.**ImageVisualizer**(***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Same as *IdentityVisualizer*, but acting on 'image'.

visualize(*image*, **_)

class firelight.visualizers.visualizers.**InputVisualizer**(***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Same as *IdentityVisualizer*, but acting on 'input'.

visualize(*input*, **_)

class firelight.visualizers.visualizers.**MSEVisualizer**(***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Visualize the Mean Squared Error (MSE) between two tensors (e.g. prediction and target).

visualize(*prediction*, *target*, **_)

class firelight.visualizers.visualizers.**MaskVisualizer**(*mask_label*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Returns a mask that is 1 where the input image equals the mask label passed at initialization, and 0 elsewhere

Parameters

- **mask_label** (*float*) – Label to be used for the construction of the mask
- ****super_kwargs** –

visualize(*tensor*, ***states*)

class firelight.visualizers.visualizers.**MaskedPcaVisualizer**(*ignore_label=None*, *n_components=3*, *background_label=0*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Version of PcaVisualizer that allows for an ignore mask. Data points which are labeled with `ignore_label` in the segmentation are ignored in the PCA analysis.

Parameters

- **ignore_label** (*int or float, optional*) – Data points with this label in the segmentation are ignored.
- **n_components** (*int, optional*) – Number of components for PCA. Has to be divisible by 3, such that a whole number of RGB images can be returned.
- **background_label** (*float, optional*) – As in BaseVisualizer, here used by default to color the ignored region.
- ****super_kwargs** –

visualize(*embedding*, *segmentation*, ***_*)

class firelight.visualizers.visualizers.**NormVisualizer**(*order=2*, *dim='C'*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Visualize the norm of a tensor, along a given direction (by default over the channels).

Parameters

- **order** (*int, optional*) – Order of the norm (Default is 2, euclidean norm).
- **dim** (*str, optional*) – Name of the dimension in which the norm is computed.
- ****super_kwargs** –

visualize(*tensor*, ***_*)

class firelight.visualizers.visualizers.**PcaVisualizer**(*n_components=3*, *joint_specs=('D', 'H', 'W')*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

PCA Visualization of high dimensional embedding tensor. An arbitrary number of channels is reduced to a multiple of 3 which are interpreted as sets RGB images.

Parameters

- **n_components** (*int, optional*) – Number of components to use. Must be divisible by 3.
- **joint_specs** (tuple of str, optional) – Entries only separated along these axis are treated jointly.

  Defaults to spatial dimensions.

  Use e.g. `('B', 'H', 'W')` to run PCA jointly on all images of the batch. #TODO: make this example work. Right now, all dimensions except 'B' work.
- ****super_kwargs** –

visualize(*embedding*, ***_*)

class firelight.visualizers.visualizers.**PredictionVisualizer**(***super_kwargs*)
Bases: *firelight.visualizers.base.BaseVisualizer*

Same as *IdentityVisualizer*, but acting on 'prediction'.

visualize(*prediction*, **_)

class firelight.visualizers.visualizers.**RGBVisualizer**(***super_kwargs*)
Bases: *firelight.visualizers.base.BaseVisualizer*

Visualize the input tensor as RGB images. If the input has n * 3 channels, n color images will be returned.

visualize(*tensor*, **_)

class firelight.visualizers.visualizers.**SegmentationVisualizer**(***super_kwargs*)
Bases: *firelight.visualizers.base.BaseVisualizer*

Same as *IdentityVisualizer*, but acting on 'segmentation'.

visualize(*segmentation*, **_)

class firelight.visualizers.visualizers.**TargetVisualizer**(***super_kwargs*)
Bases: *firelight.visualizers.base.BaseVisualizer*

Same as *IdentityVisualizer*, but acting on 'target'.

visualize(*target*, **_)

class firelight.visualizers.visualizers.**ThresholdVisualizer**(*threshold*, *mode='greater_equal'*, ***super_kwargs*)
Bases: *firelight.visualizers.base.BaseVisualizer*

Returns a mask resulting from thresholding the input tensor.

Parameters

- **threshold** (*int or float*) –
- **mode** (*str, optional*) – one of the *ThresholdVisualizer.MODES*, specifying how to threshold.
- **super_kwargs** –

MODES = ['greater', 'smaller', 'greater_equal', 'smaller_equal']

visualize(*tensor*, **_)

class firelight.visualizers.visualizers.**TsneVisualizer**(*joint_dims=None*, *n_components=3*, ***super_kwargs*)
Bases: *firelight.visualizers.base.BaseVisualizer*

tSNE Visualization of high dimensional embedding tensor. An arbitrary number of channels is reduced to a multiple of 3 which are interpreted as sets RGB images.

Parameters

- **n_components** (*int, optional*) – Number of components to use. Must be divisible by 3.
- **joint_dims** (tuple of str, optional) – Entries only separated along these axis are treated jointly.

  Defaults to spatial dimensions.
- ****super_kwargs** –

visualize(*embedding*, **_)

class firelight.visualizers.visualizers.**UmapVisualizer**(*joint_dims=None*, *n_components=3*, *n_neighbors=15*, *min_dist=0.1*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

UMAP Visualization of high dimensional embedding tensor. An arbitrary number of channels is reduced to 3 which are interpreted as RGB.

For a detailed discussion of parameters, see https://umap-learn.readthedocs.io/en/latest/parameters.html.

Parameters

- **joint_dims** (tuple of str, optional) – Entries only separated along these axis are treated jointly.

  Defaults to spatial dimensions.

- **n_components** (*int, optional*) – Number of components to use. Must be divisible by 3.
- **n_neighbors** (*int, optional*) – controls how many neighbors are considered for distance estimation on the manifold. Low number focuses on local distance, large numbers more on global structure, default 15.
- **min_dist** (*float, optional*) – minimum distance of points after dimension reduction, default 0.1.
- ****super_kwargs** –

visualize(*embedding*, **_)

class firelight.visualizers.visualizers.**UpsamplingVisualizer**(*specs*, *shape=None*, *factors=None*, ***super_kwargs*)

Bases: *firelight.visualizers.base.BaseVisualizer*

Upsample a tensor along a list of axis (specified via specs) to a specified shape, by a list of specified factors or the shape of a reference tensor (given as an optional argument to visualize).

Parameters

- **specs** (*list of str*) – Specs of the axes to upsample along.
- **shape** (*None or int or list, optional*) – Shape after upsampling.
- **factors** (*None or int or list, optional*) – Factors to upsample by.
- ****super_kwargs** –

visualize(*tensor*, *reference=None*, **_)

firelight.visualizers.visualizers.**pca**(*embedding*, *output_dimensions=3*, *reference=None*, *center_data=False*)

Principal component analysis wrapping sklearn.decomposition.PCA. Dimension 1 of the input embedding is reduced

Parameters

- **embedding** (*torch.Tensor*) – Embedding whose dimensions will be reduced.
- **output_dimensions** (*int, optional*) – Number of dimension to reduce to.

- **reference** (*torch.Tensor, optional*) – Optional tensor that will be used to train PCA on.
- **center_data** (*bool, optional*) – Whether to subtract the mean before PCA.

Returns *torch.Tensor*

4.2 firelight.utils package

4.2.1 firelight.utils.dim_utils module

class firelight.utils.dim_utils.**SpecFunction**(*in_specs=None*, *out_spec=None*, *collapse_into=None*, *suppress_spec_adjustment=True*)

Bases: object

Class that wraps a function, specified in the method *internal()*, to be applicable to tensors with of almost arbitrary dimensionality. This is achieved by applying the following steps when the function is called:

- The inputs are reshaped and their dimensions are permuted to match their respective order of dimensions specified in in_specs. Dimensions present in inputs but not requested by in_specs are collapsed in the batch dimension, labeled 'B' (per default, see collapse_into). Dimensions not present in the inputs but requested by in_specs are added (with length 1).
- If the batch dimension 'B' is present in the in_specs, 'internal' is applied on the inputs, returning a tensor with dimensions as specified in out_spec. If 'B' is not present in the in_specs, this dimension is iterated over and each slice is individually passed through 'internal'. The individual outputs are then stacked, recovering the 'B' dimension.
- Finally, the output is reshaped. The dimensions previously collapsed into 'B' are uncollapsed, and dimensions added in the first step are removed.

Parameters

- **in_specs** (*dict, optional*) – Dictionary specifying how the dimensionality and order of dimensions of input arguments of *internal()* should be adjusted.
  - Keys: Names of input arguments (as in signature of *internal()*)
  - Values: List of dimension names. The tensor supplied to internal under the name of the corresponding key will have this order of dimensions.
- **out_spec** (*list, optional*) – List of dimension names of the output of *internal()*
- **collapse_into** (*list, optional*) – If given, the default behaviour of collapsing any extra given dimensions of states into the batch dimension 'B' is overridden. Each entry of collapse_into must be a two element tuple, with the first element being the dimension to collapse, the second one being the dimension to collapse it into (prior to passing the tensor to *internal()*).
- **suppress_spec_adjustment** (*bool, optional*) – Argument to completely suppress the adjustment of dimensionalities in call(), for example if it is taken care of in call() of derived class (see firelight.visualizers.base.ConatainerVisualizer)

__call__(**args*, *out_spec=None*, *return_spec=False*, ***kwargs*)

Apply the wrapped function to a set of input arguments. Tensors will be reshaped as specified at initialization.

Parameters

- **args** (*list*) – List of positional input arguments to the wrapped function. They will be passed to *internal()* without any processing.
- **out_spec** (*list, optional*) – List of dimension names of the output.
- **return_spec** (*bool, optional*) – Weather the output should consist of a tuple containing the output tensor and the resulting spec, or only the former.
- ****kwargs** – Keyword arguments that will be passed to *internal()*. The ones with names present in *SpecFunction.in_specs* will be reshaped as required.

Returns *torch.Tensor or tuple*

internal(**args*, ***kwargs*)
Function that is being wrapped.

firelight.utils.dim_utils.**add_dim**(*tensor*, *length=1*, *new_dim=None*, *spec=None*, *return_spec=False*)
Adds a single dimension of specified length (achieved by repeating the tensor) to the input tensor.

Parameters

- **tensor** (*torch.Tensor*) –
- **length** (*int*) – Length of the new dimension.
- **new_dim** (*str, optional*) – Name of the new dimension
- **spec** (*list, optional*) – Names of dimensions of the input tensor
- **return_spec** (*bool, optional*) – If true, a dictionary containing arguments to reverse the conversion (with this function) are added to the output tuple.

Returns *torch.Tensor or tuple*

firelight.utils.dim_utils.**collapse_dim**(*tensor*, *to_collapse*, *collapse_into=None*, *spec=None*, *return_spec=False*)
Reshapes the input tensor, collapsing one dimension into another. This is achieved by

- first permuting the tensors dimensions such that the dimension to collapse is next to the one to collapse it into,
- reshaping the tensor, making one dimension out of the to affected.

Parameters

- **tensor** (*torch.Tensor*) –
- **to_collapse** (*int or str*) – Dimension to be collapsed.
- **collapse_into** (*int or str, optional*) – Dimension into which the other will be collapsed.
- **spec** (*list, optional*) – Name of dimensions of input tensor. If not specified, will be taken to be range(len(tensor.shape())).
- **return_spec** (*bool, optional*) – Weather the output should consist of a tuple containing the output tensor and the resulting spec, or only the former.

Returns *torch.Tensor or tuple*

Examples

```
>>> tensor = torch.Tensor([[1, 2, 3], [10, 20, 30]]).long()
>>> collapse_dim(tensor, to_collapse=1, collapse_into=0)
tensor([ 1,  2,  3, 10, 20, 30])
>>> collapse_dim(tensor, to_collapse=0, collapse_into=1)
tensor([ 1, 10,  2, 20,  3, 30])
```

firelight.utils.dim_utils.**convert_dim**(*tensor*, *in_spec*, *out_spec=None*, *collapsing_rules=None*, *uncollapsing_rules=None*, *return_spec=False*, *return_inverse_kwargs=False*)

Convert the dimensionality of tensor from in_spec to out_spec.

Parameters

- **tensor** (*torch.Tensor*) –
- **in_spec** (*list*) – Name of dimensions of the input tensor.
- **out_spec** (*list, optional*) – Name of dimensions that the output tensor will have.
- **collapsing_rules** (list of tuple, optional) – List of two element tuples. The first dimension in a tuple will be collapsed into the second (dimensions given by name).
- **uncollapsing_rules** (list of tuple, optional) – List of three element tuples. The first element of each specifies the dimension to 'uncollapse' (=split into two). The second element specifies the size of the added dimension, and the third its name.
- **return_spec** (*bool, optional*) – Weather the output should consist of a tuple containing the output tensor and the resulting spec, or only the former.
- **return_inverse_kwargs** (*bool, optional*) – If true, a dictionary containing arguments to reverse the conversion (with this function) are added to the output tuple.

Returns *torch.Tensor or tuple*

Examples

```
>>> tensor = torch.Tensor([[1, 2, 3], [10, 20, 30]]).long()
>>> convert_dim(tensor, ['A', 'B'], ['B', 'A'])
tensor([[ 1, 10],
        [ 2, 20],
        [ 3, 30]])
>>> convert_dim(tensor, ['A', 'B'], collapsing_rules=[('A', 'B')])
tensor([ 1, 10,  2, 20,  3, 30])
>>> convert_dim(tensor, ['A', 'B'], collapsing_rules=[('B', 'A')])
tensor([ 1,  2,  3, 10, 20, 30])
>>> convert_dim(tensor.flatten(), ['A'], ['A', 'B'], uncollapsing_rules=[('A', 3,
↪'B')])
tensor([[ 1,  2,  3],
        [10, 20, 30]])
```

firelight.utils.dim_utils.**equalize_shapes**(*tensor_spec_pairs*)

Manipulates a list of tensors such that their shapes end up equal.

Axes that are not present in all tensors will be added as a trivial dimension to all tensors that do not have them.

If shapes do not match along a certain axis, the tensors with the smaller shape will be repeated along that axis. Hence, the maximum length along each axis present in the list of tensors must be divisible by the lengths of all other input tensors along that axis.

Parameters **tensor_spec_pairs** (list of tuple) – List of two element tuples, each consisting of a tensor and a spec (=list of names of dimensions).

Returns *torch.Tensor*

firelight.utils.dim_utils.**equalize_specs**(*tensor_spec_pairs*)

Manipulates a list of tensors such that their dimension names (including order of dimensions) match up.

Parameters **tensor_spec_pairs** (list of tuple) – List of two element tuples, each consisting of a tensor and a spec (=list of names of dimensions).

Returns *torch.Tensor*

firelight.utils.dim_utils.**extend_dim**(*tensor*, *in_spec*, *out_spec*, *return_spec=False*)

Adds extra (length 1) dimensions to the input tensor such that it has all the dimensions present in out_spec.

Parameters

- **tensor** (*torch.Tensor*) –
- **in_spec** (*list*) – spec of the input tensor
- **out_spec** (*list*) – spec of the output tensor
- **return_spec** (*bool, optional*) – Weather the output should consist of a tuple containing the output tensor and the resulting spec, or only the former.

Returns *torch.Tensor or tuple*

Examples

```
>>> tensor, out_spec = extend_dim(
...     torch.empty(2, 3),
...     ['A', 'B'], ['A', 'B', 'C', 'D'],
...     return_spec=True
... )
>>> print(tensor.shape)
torch.Size([2, 3, 1, 1])
>>> print(out_spec)
['A', 'B', 'C', 'D']
```

firelight.utils.dim_utils.**join_specs**(**specs*)

Returns a list of dimension names which includes each dimension in any of the supplied specs exactly once, ordered by their occurrence in specs.

Parameters **specs** (*list*) – List of lists of dimension names to be joined

Returns *list*

Examples

```
>>> join_specs(['B', 'C'], ['B', 'H', 'W'])
['B', 'C', 'H', 'W']
>>> join_specs(['B', 'C'], ['H', 'B', 'W'])
['B', 'C', 'H', 'W']
```

firelight.utils.dim_utils.**moving_permutation**(*length*, *origin*, *goal*)

Returns a permutation moving the element at position origin to the position goal (in the format requested by torch.Tensor.permute)

Parameters

- **length** (*int*) – length of the sequence to be permuted
- **origin** (*int*) – position of the element to be moved
- **goal** (*int*) – position the element should end up after the permutation

Returns list of int

Examples

```
>>> moving_permutation(length=5, origin=1, goal=3)
[0, 2, 3, 1, 4]
>>> moving_permutation(length=5, origin=3, goal=1)
[0, 3, 1, 2, 4]
```

firelight.utils.dim_utils.**uncollapse_dim**(*tensor*, *to_uncollapse*, *uncollapsed_length*, *uncollapse_into=None*, *spec=None*, *return_spec=False*)

Splits a dimension in the input tensor into two, adding a dimension of specified length.

Parameters

- **tensor** (*torch.Tensor*) –
- **to_uncollapse** (*str or int*) – Dimension to be split.
- **uncollapsed_length** (*int*) – Length of the new dimension.
- **uncollapse_into** (*str or int, optional*) – Name of the new dimension.
- **spec** (*list, optional*) – Names or the dimensions of the input tensor
- **return_spec** (*bool, optional*) – Weather the output should consist of a tuple containing the output tensor and the resulting spec, or only the former.

Returns *torch.Tensor or tuple*

Examples

```
>>> tensor = torch.Tensor([1, 2, 3, 10, 20, 30]).long()
>>> uncollapse_dim(tensor, 0, 3, 1)
tensor([[ 1,  2,  3],
        [10, 20, 30]])
```

4.2.2 firelight.utils.io_utils module

firelight.utils.io_utils.**yaml2dict**(*path*)

Read a yaml file.

Parameters **path** (*str or dict*) – Path to the file. If dict, will be returned as is.

Returns *dict*

4.3 firelight.config_parsing module

firelight.config_parsing.**get_single_key_value_pair**(*d*)
Returns the key and value of a one element dictionary, checking that it actually has only one element

Parameters **d** (*dict*) –

Returns *tuple*

firelight.config_parsing.**get_visualizer**(*config*, *indentation=0*)
Parses a yaml configuration file to construct a visualizer.

Parameters

- **config** (*str or dict or BaseVisualizer*) – Either path to yaml configuration file or dictionary (as constructed by loading such a file). If already visualizer, it is just returned.
- **indentation** (*int, optional*) – How far logging messages arising here should be indented.

Returns *BaseVisualizer*

firelight.config_parsing.**get_visualizer_class**(*name*)
Parses the class of a visualizer from a String. If the name is not found in globals(), tries to import it.

Parameters **name** (*str*) – Name of a visualization class imported above, or dotted path to one (e.g. your custom visualizer in a different library).

Returns *type or None*

4.4 firelight.inferno_callback module

firelight.inferno_callback.**get_visualization_callback**(*config*)
Gets an inferno callback for logging of firelight visualizations.

Uses the inferno.trainers.basic.Trainer state dictionary as input for the visualizers.

The logging frequency is taken from the trainer's inferno.trainers.callbacks.logging.tensorboard.TensorboardLogger.

Parameters **config** (*str or dict*) – If str, will be converted to dict using pyyaml.

If dict, the keys are the tags under which the visualizations will be saved in Tensorboard, while the values are the configuration dictionaries to get the visualizers producing these visualizations, using *firelight.config_parsing.get_visualizer()*.

Returns inferno.trainers.callbacks.base.Callback

Examples

The structure of a configuration file could look like this:

```
# visualize model predictions
predictions:
    RowVisualizer:
        ...
```

(continues on next page)

(continued from previous page)

```
# visualize something else
fancy_visualization:
    RowVisualizer:
        ...
```

This configuration would produce images that are saved under the tags `predictions` and `fancy_visualization` in Tensorboard.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

Y