patchify

May 22, 2021

1 Function for extraction of patches

Import libraries

[1]:
```
import numpy as np
import math
import torch
```

Patchify function

[2]:
```
def patchify_quick_and_dirty(img, patch_shape, overlap=0.5):
    r"""This function calculates patches of an image with specifed overlap␣
↪using the specified patch_size.
        Args:
            img: torch tensor with dimensions (depth (nr_slices), channels,␣
↪height (y), width (x))
            patch_size: tuple consisting of (channels, height (y), width (x),␣
↪depth (nr_slices))
                        --> not the same as img dims
            overlap: percentage of the overlap between patches --> if overlap␣
↪is 1 or negative the function
                     will raise an error, since 100% overlap is not possible␣
↪--> endless loop.
                     Default: 0.5 for 50% overlap.

        Returns (list): List with all patches whereas each patch is a torch␣
↪tensor and
                        has the dimension of patch_size
    """
    # Change dimensions so we can work with them
    img = img.permute(1, 0, 2, 3)[0]
    patch_shape = (patch_shape[3], patch_shape[1], patch_shape[2])

    # Define empty list for patches
    patches = list()

    # Extract image dimensions
    img_size = img.shape
```

```
    # Define overlap number
    assert overlap < 1 and overlap >= 0, "Overlap can not be 100% or smaller than 0%."
    overlap = 100 * overlap
    if overlap == 0:
        overlap = 1
    else:
        overlap = 100 / overlap


    # Calculate the steps to skip between slides (--> here 50% overlap)
    x_step = patch_shape[0]//overlap
    y_step = patch_shape[1]//overlap
    z_step = patch_shape[2]//overlap
    assert x_step != 0 and y_step != 0. and z_step != 0,\
    "Overlap is too small and will cause a division by 0 --> specify higher overlap percentage."

    # Initialize patch idxs lists
    x_patches_idx = list()
    y_patches_idx = list()
    z_patches_idx = list()

    # Initialize variables necessary if number of patches per dimension is not even
    x_help = None
    y_help = None
    z_help = None

    # Define number od idx we need --> nurmber of patches per dimension
    nr_x_idxs = (img_size[0] / x_step)-1  # How often fits x_step in first dimension
    nr_y_idxs = (img_size[1] / y_step)-1  # How often fits y_step in second dimension
    nr_z_idxs = (img_size[2] / z_step)-1  # How often fits z_step in third dimension

    # If the number of patches is not even, the last patch will have >50% overlap
    if not (nr_x_idxs).is_integer(): # Does not fit whole in first dimension
        x_help = (img_size[0]-patch_shape[0], img_size[0])  # Last patch idxs
    if not (nr_y_idxs).is_integer(): # Does not fit whole in second dimension
        y_help = (img_size[1]-patch_shape[1], img_size[1])  # Last patch idxs
    if not (nr_z_idxs).is_integer(): # Does not fit whole in third dimension
        z_help = (img_size[2]-patch_shape[2], img_size[2])  # Last patch idxs
```

```
    # Transform float value to integer
    nr_x_idxs = math.floor(nr_x_idxs)
    nr_y_idxs = math.floor(nr_y_idxs)
    nr_z_idxs = math.floor(nr_z_idxs)

    # Loop through  number of patches and calculate the start and end idx for␣
↪the patches
    run_step = 0
    for x in range(0, nr_x_idxs, 2):
        run_step += max(0, (x-1))*x_step    # Start idx for patch
        # (Start, End) idx for patch
        x_patches_idx.append((int(run_step), int(min(run_step+patch_shape[0],␣
↪img_size[0]))))
        if run_step+(x+1)*patch_shape[0] >= img_size[0]:  # If next patch not␣
↪in range break from loop
            break
    if x_help is not None:  # Add the last patch if it exists, ie. number of␣
↪patches was not even
        x_patches_idx.append(x_help)

    # Loop through  number of patches and calculate the start and end idx for␣
↪the patches
    run_step = 0
    for y in range(0, nr_y_idxs, 2):
        run_step += max(0, (y-1))*y_step    # Start idx for patch
        # (Start, End) idx for patch
        y_patches_idx.append((int(run_step), int(min(run_step+patch_shape[1],␣
↪img_size[1]))))
        if run_step+(y+1)*patch_shape[1] >= img_size[1]:  # If next patch not␣
↪in range break from loop
            break
    if y_help is not None:
        y_patches_idx.append(y_help)

    # Loop through  number of patches and calculate the start and end idx for␣
↪the patches
    run_step = 0
    for z in range(0, nr_z_idxs, 2):
        run_step += max(0, (z-1))*z_step    # Start idx for patch
        # (Start, End) idx for patch
        z_patches_idx.append((int(run_step), int(min(run_step+patch_shape[2],␣
↪img_size[2]))))
        if run_step+(z+1)*patch_shape[2] >= img_size[2]:  # If next patch not␣
↪in range break from loop
            break
```

```
    if z_help is not None:
        z_patches_idx.append(z_help)


    # Extract the patches based on patch idxs
    for x_idxs in x_patches_idx:
        for y_idxs in y_patches_idx:
            for z_idxs in z_patches_idx:
                patch = img[x_idxs[0]:x_idxs[1], y_idxs[0]:y_idxs[1], z_idxs[0]:
↪z_idxs[1]]
                patch = patch.unsqueeze(0).permute(0, 2, 3, 1)
                patches.append(patch)

    # Return the patches
    return patches
```

Optimize the quick and dirty patchify function by making it more generic

[3]:
```
def patchify(img, patch_shape, overlap=0.5):
    r"""This function calculates patches of an image with specifed overlap␣
↪using the specified patch_size.
        Args:
            img: torch tensor with dimensions (depth (nr_slices), channels,␣
↪height (y), width (x))
            patch_size: tuple consisting of (channels, height (y), width (x),␣
↪depth (nr_slices))
                        --> not the same as img dims
            overlap: percentage of the overlap between patches --> if overlap␣
↪is 1 or negative the function
                     will raise an error, since 100% overlap is not possible␣
↪--> endless loop.
                     Default: 0.5 for 50% overlap.

        Returns (list): List with all patches whereas each patch is a torch␣
↪tensor and
                        has the dimension of patch_size
    """
    # Change dimensions so we can work with them
    img = img.permute(1, 0, 2, 3)[0]
    patch_shape = (patch_shape[3], patch_shape[1], patch_shape[2])

    # Define empty list for patches
    patches = list()

    # Extract image dimensions
    img_size = img.shape
```

```
    # Define overlap number
    assert overlap < 1 and overlap >= 0, "Overlap can not be 100% or smaller than 0%."
    overlap = 100 * overlap
    if overlap == 0:
        overlap = 1
    else:
        overlap = 100 / overlap

    # Define steps, patches_idx and helper dictionaries
    step = dict()
    nr_patches = dict()
    patches_idx = dict()
    helper = dict()
    dimensions = ['x', 'y', 'z'] # --> to loop through

    # Calculate the steps to skip between slides
    for i, dim in enumerate(dimensions):
        # Calculate step size
        step[dim] = patch_shape[i]//overlap
        assert step[dim] != 0,\
        "Overlap is too small and will cause a division by 0 --> specify higher overlap percentage."

        # Define number of necessary idx --> nurmber of patches per dimension
        nr_patches[dim] = (img_size[i] / step[dim])-1  # How often fits step in img dimension dimension
        # Extract last patch if number of patches in not even
        if not (nr_patches[dim]).is_integer(): # Does not fit whole in dimension
            helper[dim] = (img_size[i] - patch_shape[i], img_size[i])  # Last patch idxs
        # Transform float value to integer
        nr_patches[dim] = math.floor(nr_patches[dim])

        # Calculate the start and end idx for the patches
        run_step = 0
        patches_idx[dim] = list()
        for idx in range(0, nr_patches[dim], 2):
            run_step += max(0, (idx-1))*step[dim]    # Start idx for patch
            # (Start, End) idx for patch
            patches_idx[dim].append((int(run_step), int(min(run_step+patch_shape[i], img_size[i]))))
            if run_step+(idx+1)*patch_shape[i] >= img_size[i]:  # If next patch not in range break from loop
                break
        if helper.get(dim, None) is not None:
```

```
            # Add the last patch if it exists, ie. number of patches was not␣
↪even
            patches_idx[dim].append(helper[dim])

    # Extract the patches based on patch idxs and store in patches list
    for x_idxs in patches_idx['x']:
        for y_idxs in patches_idx['y']:
            for z_idxs in patches_idx['z']:
                patch = img[x_idxs[0]:x_idxs[1], y_idxs[0]:y_idxs[1], z_idxs[0]:
↪z_idxs[1]]
                patch = patch.unsqueeze(0).permute(0, 2, 3, 1)
                patches.append(patch)

    # Return the patches
    return patches
```

Test the functionality

```
[4]: def test_patchify():
    a = np.array([[[1, 4, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]],
                  [[1, 1, 1, 1, 2, 2],
                   [1, 6, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]],
                  [[1, 1, 1, 1, 2, 2],
                   [1, 2, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]],
                  [[1, 1, 1, 1, 2, 2],
                   [1, 2, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]],
                  [[1, 1, 1, 1, 2, 2],
                   [1, 2, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]],
                  [[1, 1, 1, 1, 2, 2],
                   [1, 2, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]],
                  [[1, 1, 1, 1, 2, 2],
                   [1, 2, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2],
                   [1, 1, 1, 1, 2, 2]]])
```

```
    a = torch.from_numpy(a).unsqueeze(0).permute(1, 0, 2, 3)

    b = patchify(a, (1, 4, 4, 5), 0.5)
    c = patchify(a, (1, 3, 5, 3), 0.5)

    # Test 1
    assert len(b) == 2 and b[0].shape == (1, 4, 4, 5) and b[1].shape == (1, 4,␣
↪4, 5),\
                        "Number of patches or patch dimensions are not as␣
↪expected."

    # Test 2
    assert len(c) == 4 and c[0].shape == (1, 3, 5, 3) and c[1].shape == (1, 3,␣
↪5, 3)\
                        and c[2].shape == (1, 3, 5, 3) and c[3].shape == (1, 3,␣
↪5, 3),\
                        "Number of patches or patch dimensions are not as␣
↪expected."

    # Test 3
    try:
        patchify(a, (1, 3, 5, 3), 0.2)
        assert False, "Expected an AssertionError due to possible 0 division."
    except Exception as ex:
        if type(ex).__name__ != "AssertionError":
            assert False, "Expected an AssertionError due to possible 0␣
↪division."

    # Test 4
    try:
        patchify(a, (1, 3, 5, 3), 1)
        assert False, "Expected an AssertionError due to a desired overlap of␣
↪100%."
    except Exception as ex:
        if type(ex).__name__ != "AssertionError":
            assert False, "Expected an AssertionError because of 100% overlap."
```

[5]:
```
test_patchify()
```