Copy_of_Convolutional_Layer

September 7, 2022

1 Convolutional Neural Networks: Step by Step

In this tutorial, you will implement a convolutional (CONV) layer in numpy (just the forward propagation).

Notation: - Superscript $[l]$ denotes an object of the l^{th} layer. - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.

- Superscript (i) denotes an object from the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example input.
- Subscript i denotes the i^{th} entry of a vector.
  - Example: $a^{[l]}_i$ denotes the i^{th} entry of the activations in layer l, assuming this is a fully connected (FC) layer.
- n_H, n_W and n_C denote respectively the height, width and number of channels of a given layer. If you want to reference a specific layer l, you can also write $n_H^{[l]}$, $n_W^{[l]}$, $n_C^{[l]}$.
- $n_{H_{prev}}$, $n_{W_{prev}}$ and $n_{C_{prev}}$ denote respectively the height, width and number of channels of the previous layer. If referencing a specific layer l, this could also be denoted $n_H^{[l-1]}$, $n_W^{[l-1]}$, $n_C^{[l-1]}$.

We assume that you are already familiar with `numpy` and/or have completed the previous courses of the specialization. Let's get started!

[1]:
```python
import numpy as np
import h5py
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

1.1 2 - Outline of the Assignment

You will be implementing the building blocks of a convolutional neural network!

Convolution functions: - Zero Padding - Convolve window - Convolution forward

1.2 3 - Convolutional Neural Networks

A convolution layer transforms an input volume into an output volume of different size, as shown below.

In this part, you will build every step of the convolution layer. You will first implement two helper functions: one for zero padding and the other for computing the convolution function itself.

1.2.1 3.1 - Zero-Padding

Zero-padding adds zeros around the border of an image:

Exercise: Implement the following function, which pads all the images of a batch of examples X with zeros. Use np.pad. Note if you want to pad the array "a" of shape $(5, 5, 5, 5, 5)$ with `pad = 1` for the 2nd dimension, `pad = 3` for the 4th dimension and `pad = 0` for the rest, you would do:

```
a = np.pad(a, ((0,0), (1,1), (0,0), (3,3), (0,0)), mode='constant', constant_values = (0,0))
```

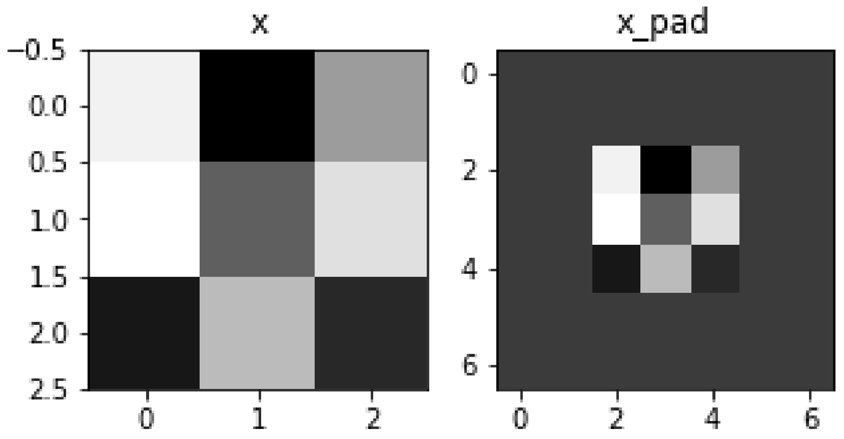
[26]:
```
# GRADED FUNCTION: zero_pad

def zero_pad(X, pad):
    """
    Argument:
    X -- python numpy array of shape (m, n_H, n_W, n_C) representing a batch of␣
 ↪m images
    pad -- integer, amount of padding around each image on vertical and␣
 ↪horizontal dimensions

    Returns:
    X_pad -- padded image of shape (m, n_H + 2*pad, n_W + 2*pad, n_C)
    """

    ### START CODE HERE ### ( 1 line)
    m, n_H, n_W, n_C = X.shape
    X_pad = np.pad(X ,((0,0), (pad,pad), (pad,pad), (0,0)), mode='constant',␣
 ↪constant_values = (0,0) )
    ### END CODE HERE ###

    return X_pad
```

[27]:
```
np.random.seed(1)
x = np.random.randn(4, 3, 3, 2)
x_pad = zero_pad(x, 2)
print ("x.shape =\n", x.shape)
print ("x_pad.shape =\n", x_pad.shape)
print ("x[1,1] =\n", x[1,1])
print ("x_pad[1,1] =\n", x_pad[1,1])
```

```
fig, axarr = plt.subplots(1, 2)
axarr[0].set_title('x')
axarr[0].imshow(x[0,:,:,0])
axarr[1].set_title('x_pad')
axarr[1].imshow(x_pad[0,:,:,0])
```

```
x.shape =
 (4, 3, 3, 2)
x_pad.shape =
 (4, 7, 7, 2)
x[1,1] =
 [[ 0.90085595 -0.68372786]
 [-0.12289023 -0.93576943]
 [-0.26788808  0.53035547]]
x_pad[1,1] =
 [[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
```

```
[27]: <matplotlib.image.AxesImage at 0x7f8aca959610>
```



Expected Output:

```
x.shape =
 (4, 3, 3, 2)
x_pad.shape =
 (4, 7, 7, 2)
```

```
x[1,1] =
 [[ 0.90085595 -0.68372786]
 [-0.12289023 -0.93576943]
 [-0.26788808  0.53035547]]
x_pad[1,1] =
 [[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

1.2.2 3.2 - Single step of convolution

In this part, implement a single step of convolution, in which you apply the filter to a single position of the input. This will be used to build a convolutional unit, which:

- Takes an input volume
- Applies a filter at every position of the input
- Outputs another volume (usually of different size)

Figure 2 : **Convolution operation** with a filter of 3x3 and a stride of 1 (stride = amount you move the window each time you slide)

Each value in the matrix on the left corresponds to a single pixel value, and we convolve a 3x3 filter with the image by multiplying its values element-wise with the original matrix, then summing them up and adding a bias. In this first step of the exercise, you will implement a single step of convolution, corresponding to applying a filter to just one of the positions to get a single real-valued output.

Later in this notebook, you'll apply this function to multiple positions of the input to implement the full convolutional operation.

Exercise: Implement conv_single_step(). Hint.

[43]:
```
# GRADED FUNCTION: conv_single_step

def conv_single_step(a_slice_prev, W, b):
    """
    Arguments:
    a_slice_prev -- slice of input data of shape (f, f, n_C_prev)
    W -- Weight parameters contained in a window - matrix of shape (f, f,␣
 ↪n_C_prev)
    b -- Bias parameters contained in a window - matrix of shape (1, 1, 1)

    Returns:
    Z -- a scalar value, the result of convolving the sliding window (W, b) on␣
 ↪a slice x of the input data
    """
    ### START CODE HERE ### ( 2 lines of code)
    # Element-wise product between a_slice_prev and W. Do not add the bias yet.
    s = np.multiply(a_slice_prev, W)
    # Sum over all entries of the volume s.
    Z = np.sum(s)
    # Add bias b to Z. Cast b to a float() so that Z results in a scalar value.
    Z = Z + b
    ### END CODE HERE ###

    return Z
```

[44]:
```
np.random.seed(1)
a_slice_prev = np.random.randn(4, 4, 3)
W = np.random.randn(4, 4, 3)
b = np.random.randn(1, 1, 1)

Z = conv_single_step(a_slice_prev, W, b)
print("Z =", Z)
```

```
Z = [[[-6.99908945]]]
```

Expected Output:

Z

-6.99908945068

1.2.3 3.3 - Convolutional Neural Networks - Forward pass

In the forward pass, you will take many filters and convolve them on the input. Each 'convolution' gives you a 2D matrix output. You will then stack these outputs to get a 3D volume

Exercise: Implement the function below to convolve the filters `W` on an input activation `A_prev`. This function takes the following inputs: * `A_prev`, the activations output by the previous layer (for a batch of m inputs); * Weights are denoted by `W`. The filter window size is `f` by `f`. * The bias vector is `b`, where each filter has its own (single) bias.

Finally you also have access to the hyperparameters dictionary which contains the stride and the padding.

Hint: 1. To select a 2x2 slice at the upper left corner of a matrix "a_prev" (shape (5,5,3)), you would do:

```
a_slice_prev = a_prev[0:2,0:2,:]
```

Notice how this gives a 3D slice that has height 2, width 2, and depth 3. Depth is the number of channels.
This will be useful when you will define `a_slice_prev` below, using the `start/end` indexes you will define. 2. To define a_slice you will need to first define its corners `vert_start`, `vert_end`, `horiz_start` and `horiz_end`. This figure may be helpful for you to find out how each of the corner can be defined using h, w, f and s in the code below.

Figure 3 : **Definition of a slice using vertical and horizontal start/end (with a 2x2 filter)** This figure shows only a single channel.

Reminder: The formulas relating the output shape of the convolution to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_C = \text{number of filters used in the convolution}$$

For this exercise, we won't worry about vectorization, and will just implement everything with for-loops.

[47]:
```
# GRADED FUNCTION: conv_forward

def conv_forward(A_prev, W, b, hparameters):
    """
    Arguments:
    A_prev -- output activations of the previous layer,
        numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    W -- Weights, numpy array of shape (f, f, n_C_prev, n_C)
    b -- Biases, numpy array of shape (1, 1, 1, n_C)
    hparameters -- python dictionary containing "stride" and "pad"

    Returns:
    Z -- conv output, numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache of values needed for the conv_backward() function
    """

    ### START CODE HERE ###
    # Retrieve dimensions from A_prev's shape (1 line)
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve dimensions from W's shape (1 line)
    (f, f, n_C_prev, n_C) = W.shape

    # Retrieve information from "hparameters" (2 lines)
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Compute the dimensions of the CONV output volume using the formula given␣
↪above.
    # Hint: use int() to apply the 'floor' operation. (2 lines)
    n_H = int((n_H_prev - f + (2 * pad)) / stride + 1)
    n_W = int((n_W_prev - f + (2 * pad)) / stride + 1)

    # Initialize the output volume Z with zeros. (1 line)
    Z = np.zeros([m, n_H, n_W, n_C])

    # Create A_prev_pad by padding A_prev
    A_prev_pad = zero_pad(A_prev, pad)

    for i in range(m):                    # loop over the batch of training␣
↪examples
        a_prev_pad = A_prev_pad[i]        # Select ith training example's padded␣
↪activation
        for h in range(n_H):              # loop over vertical axis of the output␣
↪volume
            # Find the vertical start and end of the current "slice" (2 lines)
            vert_start = stride * h
            vert_end = stride * h + f

            for w in range(n_W):          # loop over horizontal axis of the␣
↪output volume
                # Find the horizontal start and end of the current "slice" (2␣
↪lines)
                horiz_start =  stride * w
                horiz_end =  stride * w + f

                for c in range(n_C):      # loop over channels (= #filters) of the␣
↪output volume

                    # Use the corners to define the (3D) slice of a_prev_pad␣
↪(See Hint above the cell). (1 line)
                    a_slice_prev = A_prev_pad[i, vert_start:vert_end,␣
↪horiz_start:horiz_end, :]

                    # Convolve the (3D) slice with the correct filter W and␣
↪bias b, to get back one output neuron. (3 line)
                    weights = W[:, :, :, c]
                    biases = b[:, :, :, c]
                    Z[i, h, w, c] = conv_single_step(a_slice_prev, W[:, :, :,␣
↪c], b[:, :, :, c])

    ### END CODE HERE ###

    # Making sure your output shape is correct
    assert(Z.shape == (m, n_H, n_W, n_C))

    # Save information in "cache" for the backprop
    cache = (A_prev, W, b, hparameters)

    return Z, cache
```

[48]:
```python
np.random.seed(1)
A_prev = np.random.randn(10,5,7,4)
W = np.random.randn(3,3,4,8)
b = np.random.randn(1,1,1,8)
hparameters = {"pad" : 1,
               "stride": 2}

Z, cache_conv = conv_forward(A_prev, W, b, hparameters)
print("Z's mean =\n", np.mean(Z))
print("Z[3,2,1] =\n", Z[3,2,1])
print("cache_conv[0][1][2][3] =\n", cache_conv[0][1][2][3])
```

```
Z's mean =
 0.6923608807576933
Z[3,2,1] =
 [-1.28912231  2.27650251  6.61941931  0.95527176  8.25132576  2.31329639
 13.00689405  2.34576051]
cache_conv[0][1][2][3] =
 [-1.1191154   1.9560789  -0.3264995  -1.34267579]
```

Expected Output:

```
Z's mean =
 0.692360880758
Z[3,2,1] =
 [ -1.28912231   2.27650251   6.61941931   0.95527176   8.25132576
   2.31329639  13.00689405   2.34576051]
cache_conv[0][1][2][3] = [-1.1191154   1.9560789  -0.3264995  -1.34267579]
```

Finally, CONV layer should also contain an activation, in which case we would add the following line of code:

```python
# Convolve the window to get back one output neuron
Z[i, h, w, c] = ...
# Apply activation
A[i, h, w, c] = activation(Z[i, h, w, c])
```

You don't need to do it here.

1.3 4 - Pooling layer

The pooling (POOL) layer reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input. The two types of pooling layers are:

- Max-pooling layer: slides an (f, f) window over the input and stores the max value of the window in the output.
- Average-pooling layer: slides an (f, f) window over the input and stores the average value of the window in the output.

These pooling layers have no parameters for backpropagation to train. However, they have hyperparameters such as the window size f. This specifies the height and width of the $f \times f$ window you would compute a *max* or *average* over.

1.3.1 4.1 - Forward Pooling

Now, you are going to implement MAX-POOL and AVG-POOL, in the same function.

Exercise: Implement the forward pass of the pooling layer. Follow the hints in the comments below.

Reminder: As there's no padding, the formulas binding the output shape of the pooling to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f}{stride} \rfloor + 1$$

$$n_C = n_{C_{prev}}$$

[49]:
```
# GRADED FUNCTION: pool_forward

def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implements the forward pass of the pooling layer

    Arguments:
    A_prev -- Input data, numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    hparameters -- python dictionary containing "f" and "stride"
    mode -- the pooling mode you would like to use, defined as a string ("max"␣
 ↪or "average")

    Returns:
    A -- output of the pool layer, a numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache used in the backward pass of the pooling layer, contains the␣
 ↪input and hparameters
    """

    # Retrieve dimensions from the input shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve hyperparameters from "hparameters"
    f = hparameters["f"]
    stride = hparameters["stride"]
    # Define the dimensions of the output
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Initialize output matrix A
    A = np.zeros((m, n_H, n_W, n_C))

    ### START CODE HERE ###
    for i in range(m):                          # loop over the training examples
        for h in range(n_H):                    # loop on the vertical axis of␣
↪the output volume
            # Find the vertical start and end of the current "slice" (2 lines)
            vert_start = stride * h
            vert_end = stride * h +f

            for w in range(n_W):                # loop on the horizontal axis␣
↪of the output volume
                # Find the vertical start and end of the current "slice" (2␣
↪lines)
                horiz_start = stride * w
                horiz_end = stride * w + f

                for c in range (n_C):           # loop over the channels of␣
↪the output volume

                    # Use the corners to define the current slice on the ith␣
↪training example of A_prev, channel c. (1 line)
                    a_prev_slice = A_prev[i]

                    # Compute the pooling operation on the slice.
                    # Use an if statement to differentiate the modes.
                    # Use np.max and np.mean.
                    if mode == "max":
                        A[i, h, w, c] = np.max(a_prev_slice)
                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_prev_slice)

    ### END CODE HERE ###

    # Store the input and hparameters in "cache" for pool_backward()
    cache = (A_prev, hparameters)

    # Making sure your output shape is correct
    assert(A.shape == (m, n_H, n_W, n_C))

    return A, cache
```

[50]:
```python
# Case 1: stride of 1
np.random.seed(1)
A_prev = np.random.randn(2, 5, 5, 3)
hparameters = {"stride" : 1, "f": 3}

A, cache = pool_forward(A_prev, hparameters)
print("mode = max")
print("A.shape = " + str(A.shape))
print("A =\n", A)
print()
A, cache = pool_forward(A_prev, hparameters, mode = "average")
print("mode = average")
print("A.shape = " + str(A.shape))
print("A =\n", A)
```

```
mode = max
A.shape = (2, 3, 3, 3)
A =
 [[[[2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]]

  [[2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]]

  [[2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]]]


 [[[1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]]

  [[1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]]

  [[1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]]]]

mode = average
A.shape = (2, 3, 3, 3)
A =
 [[[[0.05958534 0.05958534 0.05958534]
```

```
   [0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]]

  [[0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]]

  [[0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]]]


 [[[0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]]

  [[0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]]

  [[0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]]]]
```

** Expected Output**

```
mode = max
A.shape = (2, 3, 3, 3)
A =
 [[[[ 1.74481176  0.90159072  1.65980218]
   [ 1.74481176  1.46210794  1.65980218]
   [ 1.74481176  1.6924546   1.65980218]]

  [[ 1.14472371  0.90159072  2.10025514]
   [ 1.14472371  0.90159072  1.65980218]
   [ 1.14472371  1.6924546   1.65980218]]

  [[ 1.13162939  1.51981682  2.18557541]
   [ 1.13162939  1.51981682  2.18557541]
   [ 1.13162939  1.6924546   2.18557541]]]


 [[[ 1.19891788  0.84616065  0.82797464]
   [ 0.69803203  0.84616065  1.2245077 ]
   [ 0.69803203  1.12141771  1.2245077 ]]

  [[ 1.96710175  0.84616065  1.27375593]
   [ 1.96710175  0.84616065  1.23616403]
   [ 1.62765075  1.12141771  1.2245077 ]]
  [[ 1.96710175  0.86888616  1.27375593]
   [ 1.96710175  0.86888616  1.23616403]
   [ 1.62765075  1.12141771  0.79280687]]]]

mode = average
A.shape = (2, 3, 3, 3)
A =
 [[[[ -3.01046719e-02  -3.24021315e-03  -3.36298859e-01]
   [  1.43310483e-01   1.93146751e-01  -4.44905196e-01]
   [  1.28934436e-01   2.22428468e-01   1.25067597e-01]]

  [[ -3.81801899e-01   1.59993515e-02   1.70562706e-01]
   [  4.73707165e-02   2.59244658e-02   9.20338402e-02]
   [  3.97048605e-02   1.57189094e-01   3.45302489e-01]]

  [[ -3.82680519e-01   2.32579951e-01   6.25997903e-01]
   [ -2.47157416e-01  -3.48524998e-04   3.50539717e-01]
   [ -9.52551510e-02   2.68511000e-01   4.66056368e-01]]]


 [[[ -1.73134159e-01   3.23771981e-01  -3.43175716e-01]
   [  3.80634669e-02   7.26706274e-02  -2.30268958e-01]
   [  2.03009393e-02   1.41414785e-01  -1.23158476e-02]]

  [[  4.44976963e-01  -2.61694592e-03  -3.10403073e-01]
   [  5.08114737e-01  -2.34937338e-01  -2.39611830e-01]
   [  1.18726772e-01   1.72552294e-01  -2.21121966e-01]]

  [[  4.29449255e-01   8.44699612e-02  -2.72909051e-01]
   [  6.76351685e-01  -1.20138225e-01  -2.44076712e-01]
   [  1.50774518e-01   2.89111751e-01   1.23238536e-03]]]]
```

[51]:
```python
# Case 2: stride of 2
np.random.seed(1)
A_prev = np.random.randn(2, 5, 5, 3)
hparameters = {"stride" : 2, "f": 3}

A, cache = pool_forward(A_prev, hparameters)
print("mode = max")
print("A.shape = " + str(A.shape))
print("A =\n", A)
print()

A, cache = pool_forward(A_prev, hparameters, mode = "average")
print("mode = average")
print("A.shape = " + str(A.shape))
print("A =\n", A)
```

```
mode = max
A.shape = (2, 2, 2, 3)
A =
 [[[[2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]]

  [[2.18557541 2.18557541 2.18557541]
   [2.18557541 2.18557541 2.18557541]]]


 [[[1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]]

  [[1.96710175 1.96710175 1.96710175]
   [1.96710175 1.96710175 1.96710175]]]]

mode = average
A.shape = (2, 2, 2, 3)
A =
 [[[[0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]]

  [[0.05958534 0.05958534 0.05958534]
   [0.05958534 0.05958534 0.05958534]]]


 [[[0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]]

  [[0.07763108 0.07763108 0.07763108]
   [0.07763108 0.07763108 0.07763108]]]]
```

Expected Output:

```
mode = max
A.shape = (2, 2, 2, 3)
A =
 [[[[ 1.74481176  0.90159072  1.65980218]
   [ 1.74481176  1.6924546   1.65980218]]

  [[ 1.13162939  1.51981682  2.18557541]
   [ 1.13162939  1.6924546   2.18557541]]]


 [[[ 1.19891788  0.84616065  0.82797464]
   [ 0.69803203  1.12141771  1.2245077 ]]

  [[ 1.96710175  0.86888616  1.27375593]
   [ 1.62765075  1.12141771  0.79280687]]]]
```

```
mode = average
A.shape = (2, 2, 2, 3)
A =
 [[[[-0.03010467 -0.00324021 -0.33629886]
   [ 0.12893444  0.22242847  0.1250676 ]]

  [[-0.38268052  0.23257995  0.6259979 ]
   [-0.09525515  0.268511    0.46605637]]]


 [[[-0.17313416  0.32377198 -0.34317572]
   [ 0.02030094  0.14141479 -0.01231585]]

  [[ 0.42944926  0.08446996 -0.27290905]
   [ 0.15077452  0.28911175  0.00123239]]]]
```