09.4 - Deep Learning - Convolutional Neural Networks

November 11, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 9.4 Deep Learning - Convolutional Neural Networks

1.1.1 Readings

- [WEIDMAN] Ch5
- [CHARU] Ch8
- A great guide to calculating padding, strides, etc. https://arxiv.org/pdf/1603.07285v1.pdf

[1]:
```
#import from last time work so we can extend further
from neuralnet.second_version import *
import numpy as np
from numpy import ndarray
```

So far we have focus on **Dense** layers (or also known as fully-connected layer) which are nice in understanding relationships. Adding **activation function** like Sigmoid or Tanh allows us to understand the non-linear relationship between features and output, with Tanh having a steeper gradient, allowing the network to learn faster. Adding **SoftMaxCrossEntropy** also enhance the gradient produced but remember that it only works with classification problems. Adding **Dropout** helps in overfitting; **glorot initialization** to make sure the weight is normally distributed, **learning decay** to make sure we eventually reach the minimum instead of hopping all over the places, and last, the **momentum** to make sure we do not stuck in local minimum. Such architecture is usally quite okay for **normal classification** problem.

However, when we talk about specific classification problem such as image or text or signal, they all have specific nature that would benefit from different architectures.

Today, we gonna work on image (this field is called computer vision) and discuss why Dense layer may not be the best, and propose CNN (Convolutional Neural Network) as a better way for dealing with image classification.

There are mainly three layers that can help dealing with images:

1. Convolutional layer
2. Max/Average pooling layer
3. Flatten layer

1.1.2 1. Convolutional Layer

Let's say given a image of 14 x 14 pixels = 196 features like this. Each data point is an array of numbers describing how dark each pixel is, where value range from 0 to 255. These values can be

normalized ranging from 0 to 1. For example, for the following digit (the digit 1), we could have:

It is first important to define the input shape of an image, which will be (input channels, image height, image width). If we have lots of images, the input shall be (batch size, input channels, image height, image width). For our case, if it is a grayscale image, the shape is (1, 14, 14). If it is a RGB image, it shall be (3, 14, 14). If it is a CMYK, it shall be (4, 14, 14). If I define batch size as 500 (out of many more images I have), my input is (500, 4, 14, 14). (Commonly, batch size is around few hundreds).

We might input these features into Dense layers and try to ask the Dense layers to understand the relationships. How do we input it? Well, we can actually try converting (500, 4, 14, 14) to (500, 784) and then simply feed to Dense Layer. What's wrong?

Obviously, this is not so optimal since we do not **actually understand the nature of image**. The key is that each single pixel actually holds very little information, right? However, pattern of image can be better recognized by patches of pixels, rather than single pixel. Imagine I give you a picture of cat, and I give you only a one-fourth of the picture, can you recognize that it's a cat? Probably yes. But what if I give you a single pixel…..you will have zero idea.

Why pattern of images are better recognized by patches?…because humans recognize some visual patterns like corners, edges, sharpness. Combining all these visual patterns form the image. This is how humans visualize, and in fact, we should also apply these principles to neural networks

So how do we generate each patch of feature?…actually, it is very easy. We simply perform a convolution operation like this:

Mathematically, it looks like this:

Let's say we have a 5 x 5 input image I of channel 0 of batch 0:

$$I = \begin{bmatrix} i_{11} & i_{12} & i_{13} & i_{14} & i_{15} \\ i_{21} & i_{22} & i_{23} & i_{24} & i_{25} \\ i_{31} & i_{32} & i_{33} & i_{34} & i_{35} \\ i_{41} & i_{42} & i_{43} & i_{44} & i_{45} \\ i_{51} & i_{52} & i_{53} & i_{54} & i_{55} \end{bmatrix}$$

Each of this pixel may represent the brightness ranging from 0 to 255. Or if normalized, shall be 0 to 1.

If we define a 3 x 3 patch which we commonly called **weights (W)** or in computer vision, we called **filters/kernels** like this (*we shall called filters in this lecture note for simplicity*) :

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

Let's say we are scanning the middle of the image, then the output feature would be (we'll denote this as o_{33}):

$$o_{33} = w_{11} * i_{22} + w_{12} * i_{23} + w_{13} * i_{24} + \ w_{21} * i_{32} + w_{22} * i_{32} + w_{23} * i_{34} + \ w_{32} * i_{43} + w_{33} * i_{44}$$

This will result in one output feature called **feature map**. Of course, we may add bias to it and then will be fed through an activation function.

When we do this operation across the whole image, that is, *sliding W over the input image*, taking the dot product of W with the pixels at each location of the image, and ending up with a new image O of almost identical size to the original image (Note that it may be slightly different, depending on how we convolve the edge of the image), this is called **convolution** which will result in the output features called **feature maps** or **output depth** or **output channels of the layer** (*we shall called output channels in this lecture for simplicity*). What the convolution does is basically detecting certain patterns defined by W at certain location of the input image.

Actual feature maps look like this. Each feature map is a output of a single training example and convolve each kernel over the sample. In simple words, if we have k filters, then we have k feature maps. They represent the activation part corresponding to the kernels.

In a CNN, there are 3 main hyperparameters to fine tune - (1) filter size, (2) padding, and (3) stride. To answer the principle in tuning them, it is more beneficial to first answer these questions (which will then naturally answer the former questions):

1. How the filters look like? What is the shape of filters? What is the width and height of filters? Why filter is typically odd-square size?
2. How should we convolve the edges?
3. How many step we should take slide our filter? Skip 2?
4. What would be the shape of the output matrix? Also in summary, what is the shape of the input, output, and filter?

A. Filters

1. **How the filters look like?**. It turns out that each filter actually detect the presence of certain visual pattern. For example, this filter below detects whether there is an edge at that location of the image. There are also other similar filters detecting corners, lines, etc. Check out https://setosa.io/ev/image-kernels/ and try changing the values

$$w = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Real filters can look like this. They may look somewhat random at first glance, but we can see that clear structure being learned in most kernels. For example, filters 3 and 4 seem to be learning diagonal edges in opposite directions, and other capture round edges or enclosed spaces:

However, **it is important to note that we DON'T need to decide the filters** to use. We can simply feed a random generated filter, and it is the job of CNN to learn these filters. These learned filters will learn what features are most efficient for the classification process.

What is the shape of filters?. For each image, we can apply multiple filters, depending on how many output channels we want. Let's say the input channel is 3, and we want the output channel to 64, then we apply a filter of size (3, 64, filter width, filter height), just like in Dense layer, where we input (input_neurons, output_neurons). What on earth is *output channel*, they are simply the number of patterns detected by your filters. How do we know how many output channel to use?

The answer is we don't know…we just try and see what works. **The rule of thumb** is to first try a few filter, and gradually increase.

We shall talk more about the size of output-width and output-height after convolution. However, it is important to note that it is NOT necessary that output-width = image-width, and same for height. That will be determined by how we convolve, which we shall discuss now.

What is the width and height of filters? If we use a filter width = filter height = 3, then our filter shape will be (input channel, output channel, 3, 3). The question is how to decide the filter height and width. The idea is that if we use a 3 x 3 filter, each pixel got 8 neighboring information. On the other hand, if we we use big filter like 9 x 9, then we got 80 neighboring information. To choose this depends on your image classification task, if you think separating cat and dog images requires fine details like the nose, use a smaller filter. But if you think it requires high-level details like the shape of the body, then maybe bigger filter size. **How to choose is NOT science, but is an art, so simply try out. Another way is to read papers and copy them. Third is following the typical filter size which is 3x3, 5x5, and 7x7**.

Why filter is typically odd-square size? The sole reason of being an odd-square size filter because it is easy to do paddings due to its symmetry. For example, when a 3x3 filter is used, there would be a padding of 1 in all sides (if you want the output to be of the same size as input). In case of 2x2 filters, there are four possible padding scenarios:

- Left = 1, Right = 0, Top = 0, Bottom = 1
- Left = 0, Right = 1, Top = 0, Bottom = 1
- Left = 1, Right = 0, Top = 1, Bottom = 0
- Left = 0, Right = 1, Top = 1, Bottom = 0

This is unnecessary computation without any benefit.

In addition, why the filter is square is not of importance. It is also possible to use non-square size of filter (see Inception Network), but it is just more difficult to handle…that's it. Afterall, there is no real evidence that square-filters do not work, thus most of networks prefer square-sized filters.

B. Padding

2. **How should we convolve the edges?**. Should we do the entire image? Should we maintain the output features to be the same size as input features? Recall this image:

It has 4 x 4 pixels = 16 features. But after convolution, we only got 2 x 2 pixels = 4 features left. Is that good? There is no correct answers here but we are quite sure that we lose some information. In fact, it is always nice to **maintain the output features to be the same size as input features**, but how? There is no space to convolve since the filter is 2 x 2 and it can only shift right one time.

The answer is **padding**, where we can enlarge the input image by padding the surroundings with zeros. How much? Padding until we get the original size or larger size, for example, like this. The below put zero padding around which result the output features to be the same size as input features.

The below put even more padding which pad to make sure each single pixel is convoluted (full padding), which result the output features to be even large

Mathematically, it is easiest to understand padding from the 1D input like this:

$$input = [1, 2, 3, 4, 5]$$

to

$$input_{padded} = [0, 1, 2, 3, 4, 5, 0]$$

Normally, large size may benefit from more features, but also suffer from lengthy training time. It is probably best to only perform enough padding to get the same size as input features.

C. Strides

3. **How many step we should take slide our filter? Skip 2?** Should we shift 1 step per convolution, or 2 steps, or how many steps. **In fact, it really depends on how detail you want it to be. But defining bigger steps reduce the feature size and thus reduce the computation time.** Bigger step is like human scanning picture more roughly but can reduce the computation time….whether to use it is something to be experimented though.

In computer vision, we called this step as **stride**. Example is like this:

No padding with stride of 2

Padding with stride of 2

Actual image convolution can look like this (with stride 1 and no padding):

The convoluted image may look like this (nothing relate with the above matrix though):

The formula to be used to measure the padding value to get the spatial size of the input and output volume to be the same with stride 1 is

$$\frac{K-1}{2}$$

where K is the filter size.

This means that if our image is size $24 * 24$, and the filter size is $3x3$, then our K has size 3 so the padding should be $(3-1)/2 = 1$, then we need to add **a border of one pixel valued 0 around the outside of the image**, which would result in the input image of size $26 * 26$

D. Shape

4. **What would be the shape of the output matrix? Also in summary, what is the shape of the input, output, and filter?**. Recall that in Dense layer, the shape of weight matrix is defined as

(neuron_{in}, neuron_{out})

For example, given a image of 24 x 24 pixels = 576 features. Let's say we got around 1000 images, thus our input has a shape of (1000, 576). Thus the input layer should have 576 neurons. Let's say our next hidden layer has 10 neurons, what should be the shape of the weight matrix? The answer is easy, we need to simply find the ? here:

$$(1000, 576)@? = (1000, 10)$$

Obviously, the weight matrix would be

$$(576, 10)$$

where you can clearly see 576 is the number of input neurons and 10 is the number of output neurons.

Now our question is **how about convolutional layers**. In convolutional layer, the shape of input has shape of (batch size, input channels, image height, image width). For example, let's say after we have 1000 of batch size, 3 input channels, and image height and width to be 24, thus the shape of input is (1000, 3, 24, 24). Now let's say we would like output channel of 4? What should be the shape of the weight matrix? Also, what would be the shape of the output? The answer is a little tough, but we know one thing is that the number of samples will remain the same, thus we get:

$$(1000, 3, 24, 24) \circledast W = (1000, 4, O, O)$$

O actually depend on W, i.e., on the stride (denote as S), padding (denote as P), filter size (denote as F) as well as the input width and height (denote as I). O can be calculated with the formula as follows:

$$O = \frac{I - F + 2P}{S} + 1$$

Suppose we have an input image of size $3 * 24 * 24$, we apply filters of size $3 * 3$, with single stride and no zero padding.

Here I=24, F=3, P=0 and S=1.

The size of the output volume will be $([24 - 3 + 0]/1) + 1 = 22$. Thus

$$(1000, 3, 24, 24) \circledast (3, 4, 3, 3)_{p=0,s=1} = (1000, 4, 22, 22)$$

In conclusion,

- The input will have a 4D shape of (batch size, input channels, input height, input width)
- The output will have a 4D shape of (batch size, output channels, output height, output width)
- The convolutional filters will have 4D shape of (input channels, output channels, filter height, filter width)

Note: The order does not matter and it depends on the python library you use but these four dimensions always exist in CNN.

**The general rule of selecting padding, stride and filter size are of course of trial-and-error. But it's important to remember that they should result in output image size of integers not decimals"

Demo https://www.cs.ryerson.ca/~aharley/vis/conv/

1.1.3 2. Max/Average Pooling Layer

Talking about **reducing computation time**, a common way is to perform a **pooling layer** which simply downsample the image by average a set of pixels, or by taking the maximum value. If we define a pooling size of 2, this involves mapping each 2 x 2 pixels to one output, like this:

Nevertheless, pooling has a really big downsides, i.e., it basically lose a lot of information. Compared to strides, strides simply scan less but maintain the same resolution but pooling simply reduce the resolution of the images….As Geoffrey Hinton said on Reddit AMA in 2014 - **The pooling operation used in CNN is a big mistake and the fact that it works so well is a disaster**. In fact, in most recent CNN architectures like ResNets, it uses pooling very minimially or not at all. In this lecture, we are not going to implement pooling, but we just talk about it for the sake of completeness since very early architectures like AlexNet uses pooling.

1.1.4 3. Flatten Layer

It must be said that in CNN, probably there are many convolutional layers. However, in the last layer, typically, if we want to predict a certain class, it make sense to use Dense layer as the output layer. However, the question is how do we send input of shape $(\text{size}, \text{image height}, \text{image width}, \text{input channels})$ into Dense layer?

This is actually quite easy. What we can do is simply squash all these 4D vectors into 2D vectors. For example, given (1000, 2, 22, 22), through a *flatten* operation, the vector becomes (1000, 968), which we can then multiply with weight just like in Dense layer, make predictions, and calculate loss just like we did in previous class.

Why we can perform *flatten* operation? Does it not lost any information? This is because through flattening, it is just another representations, thus flattening does not result in any loss of information. It also allow the Dense layer to understand the relationships of visual patterns from prior convolutional layers to the output.

1.1.5 Let's start coding!!

1.1.6 1D input

First off, to make us easily understand CNN coding, let's start simple, working with 1D input. Also let's write some helpers to make our life easier, namely assert_same_shape, and assert_dim

[2]:
```
def assert_same_shape(A: ndarray, B: ndarray):
    assert A.shape == B.shape

def assert_dim(X: ndarray, dim: ndarray):
    assert len(X.shape) == dim
```

Padding Padding can be easily coded. Let's start simple with 1D input like this:

[3]:
```
input_1d = np.array([1,2,3,4,5])
param_1d = np.array([1,1,1])
```

[4]:
```
def _pad_1d(input_: ndarray,
            padding: int) -> ndarray:
    zero = np.array([0])
    zero = np.repeat(zero, padding)  #number of zeros * padding
    return np.concatenate([zero, input_, zero])
```

[5]:
```
_pad_1d(input_1d, 1)
```

[5]:
```
array([0, 1, 2, 3, 4, 5, 0])
```

Forward pass - convolution Convolution in 1D is simple.

We are actually doing something like this: [0, 1, 2, 3, 4, 5, 0] [1, 1, 1] = 0*1 + 1*1 + 2*1 = 3 [1, 1, 1] = 1*1 + 2*1 + 3*1 = 6 [1, 1, 1] = 9 [1, 1, 1] = 12 [1, 1, 1] = 9

[6]:
```
def conv_1d(input_: ndarray,
            param: ndarray) -> ndarray:

    # assert 1D data
    assert_dim(input_, 1)
    assert_dim(param, 1)

    # 1. pad the input
    # (k - 1) / 2 can be implemented as k // 2 where // is floor division
    param_len = param.shape[0]  #3
    param_mid = param_len // 2  #3 // 2 = 1
    input_pad = _pad_1d(input_, param_mid) # [0, 1, 2, 3, 4, 5, 0]

    # initialize the output
    # we let output has the same shape of input
    output = np.zeros(input_.shape) # [0, 0, 0, 0, 0]

    # perform the 1d convolution
    # 2. Use the padded input and params to compute output
    for o in range(output.shape[0]): #0 to 4
        for w in range(param_len):  #0 to 2
            output[o] += param[w] * input_pad[o+w] #o move along with w thus o+w

    # ensure input has same shape as output
    # this is actually optional
    assert_same_shape(input_, output)

    return output
```

Next, we code the sum, which is basically sum everything return by the convolution.

[7]:
```
def conv_1d_sum(input_: ndarray,
                param: ndarray) -> ndarray:
    output = conv_1d(input_, param)
    return np.sum(output)
```

[8]:
```
conv_1d_sum(input_1d, param_1d)
```

[8]: 39.0

Gradients How to compute the gradients of convolution?

Let's first try some set of numbers and manually get the gradients:

[9]:
```
#randomly choose to increase 5th element by 1
#so we can know the gradient of 5th element in respect to the convolution sum
input_1d_2 = np.array([1,2,3,4,6])
param_1d = np.array([1,1,1])

conv_1d_sum(input_1d_2, param_1d)
```

[9]: 41.0

What does this mean? Since we change the 5th element by 1, which increase the convolution sum by 2, thus the gradient of the 5th element is 2.

Let's see how we actually get the 2.

Given

$$t = [0, 1, 2, 3, 4, 5, 0]$$

and

$$w = [1, 1, 1]$$

and let o be the output of convolution: $o_0 ... o_4$

First, let's look at the convolution equation like this:

$$o_0 = t_0 * w_0 + t_1 * w_1 + t_2 * w_2$$
$$o_1 = t_1 * w_0 + t_2 * w_1 + t_3 * w_2$$
$$o_2 = t_2 * w_0 + t_3 * w_1 + t_4 * w_2$$
$$o_3 = t_3 * w_0 + t_4 * w_1 + t_5 * w_2$$
$$o_4 = t_4 * w_0 + t_5 * w_1 + t_6 * w_2$$

Look at t_5 which is our 5th element, where t_5 changes O_3 based on w_2, O_4 based on w_1, and O_5 based on w_0 (which we don't have; the reason we wrote it so to detect the underlying pattern)

This gradient can be written as:

$$\frac{\partial L}{\partial t_5} = \frac{\partial L}{\partial o_3} * \frac{\partial o_3}{\partial t_5} + \frac{\partial L}{\partial o_4} * \frac{\partial o_4}{\partial t_5} + \frac{\partial L}{\partial o_5} * \frac{\partial o_5}{\partial t_5}$$

$$\frac{\partial L}{\partial t_5} = \frac{\partial L}{\partial o_3} * w_2 + \frac{\partial L}{\partial o_4} * w_1 + \frac{\partial L}{\partial o_5} * w_0$$

Of course, in this simple example, when the loss is just the sum, and since o_i is contributing to the sum of the convolution sum, i.e.,

$$L = o_0 + o_1 + o_2 + o_3 + o_4$$

its derivative is simply

$$\frac{\partial L}{\partial o_i} = 1$$

Thus,

$$\frac{\partial L}{\partial t_5} = \frac{\partial L}{\partial o_3} * w_2 + \frac{\partial L}{\partial o_4} * w_1 + \frac{\partial L}{\partial o_5} * w_0 = 1 * w_2 + 1 * w_1 + 0 * w_0 = 2$$

since o_5 does not exist

Since we need to code this, we need to see whether there is any general pattern. Let's look at other elements as well:

$$\frac{\partial L}{\partial t_5} = \frac{\partial L}{\partial o_3} * w_2 + \frac{\partial L}{\partial o_4} * w_1 + \frac{\partial L}{\partial o_5} * w_0$$
$$\frac{\partial L}{\partial t_4} = \frac{\partial L}{\partial o_2} * w_2 + \frac{\partial L}{\partial o_3} * w_1 + \frac{\partial L}{\partial o_4} * w_0$$
$$\frac{\partial L}{\partial t_3} = \frac{\partial L}{\partial o_1} * w_2 + \frac{\partial L}{\partial o_2} * w_1 + \frac{\partial L}{\partial o_3} * w_0$$
$$\frac{\partial L}{\partial t_2} = \frac{\partial L}{\partial o_0} * w_2 + \frac{\partial L}{\partial o_1} * w_1 + \frac{\partial L}{\partial o_2} * w_0$$
$$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial o_{-1}} * w_2 + \frac{\partial L}{\partial o_0} * w_1 + \frac{\partial L}{\partial o_1} * w_0$$

How should we code this? In terms of code, it is easy to represent

$$w_0, w_1, w_2$$

simply by iterating.

But we need to find a way to represent the output gradients of:

$$\frac{\partial L}{\partial o_{-1}} \text{ to } \frac{\partial L}{\partial o_5}$$

In fact, we know

$$\frac{\partial L}{\partial o_{-1}} = 0$$

as well as

$$\frac{\partial L}{\partial o_5} = 0$$

while other gradients are simply one. Thus, we can represent as a list of

$$grad = [0, 1, 1, 1, 1, 1, 0] = [\frac{\partial L}{\partial o_{-1}}, \frac{\partial L}{\partial o_0}, \frac{\partial L}{\partial o_2}, \frac{\partial L}{\partial o_3}, \frac{\partial L}{\partial o_4}, \frac{\partial L}{\partial o_5}]$$

Let's called this grad. It can be a bit confusing now that we are coding, since the indices start from 0. Let's rewrite the equation using these indices:

$$\frac{\partial L}{\partial t_5} = inputgrad_4 = grad_4 * w_2 + grad_5 * w_1 + grad_6 * w_0$$
$$\frac{\partial L}{\partial t_4} = inputgrad_3 = grad_3 * w_2 + grad_4 * w_1 + grad_5 * w_0$$
$$\frac{\partial L}{\partial t_3} = inputgrad_2 = grad_2 * w_2 + grad_3 * w_1 + grad_4 * w_0$$
$$\frac{\partial L}{\partial t_2} = inputgrad_1 = grad_1 * w_2 + grad_2 * w_1 + grad_3 * w_0$$
$$\frac{\partial L}{\partial t_1} = inputgrad_0 = grad_0 * w_2 + grad_1 * w_1 + grad_2 * w_0$$

Now we have to map the indices in coding. This is simple.

For each inputgrad, we need to repeatingly run, w_0 to w_2, we simply iterate using something like

for each inputgrad for each p in param #we call our w as param, just like pyTorch

Then for the first input $inputgrad_0$, we need to make sure to run the grad indicies to be 2, 1, 0 in this order. grad indices depend on two things: a) As w up by 1, grad index lower by 1; this can be easily coded simply by subtracting w, thus when w increases, the grad decreases; b) grad index starts at index of inputgrad + 2, where 2 is actually length of w - 1.

Thus, this can be summarized as the following python code:

for i in range(input_grad.shape[0]) #this is ok since inputgrad shape == input.shape for p in range(param.shape[0]) # this will represent index of w inputgrad_i += grad[i + (len(p) - 1) - p] * param[p]

This can be written as function like this:

[10]:
```
def _input_grad_1d(input_: ndarray,
                   param: ndarray,
                   grad: ndarray = None) -> ndarray:

    param_len = param.shape[0]
    param_mid = param_len // 2

    if grad is None:
        grad = np.ones_like(input_) #choose one so grad can be multiplied and␣
↪not become zero
    else:
        assert_same_shape(input_, grad)

    #1. pad the output gradients
    grad = _pad_1d(grad, param_mid)  #[0, 1, 1, 1, 1, 1, 0]

    #prepare input_grad which has grad of the five elements
    #thus the initial look can be [0, 0, 0, 0, 0]
    input_grad = np.zeros_like(input_)

    #2. Use the padded output gradients, along with param, to compute the input␣
↪gradient
    for i in range(input_grad.shape[0]):  #for each input grad which follows␣
↪the same shape as input
        for p in range(param.shape[0]):  #for each param
            input_grad[i] += grad[i +  param_len - 1 - p] * param[p]

    assert_same_shape(input_grad, input_)

    return input_grad
```

[11]:
```
_input_grad_1d(input_1d, param_1d)
```

[11]:
```
array([2, 3, 3, 3, 2])
```

So here, it means if I change the first input by 1, it shall increase the output by 2, if I change the second input by 1, it shall increase the output by 3, etc.

Now, we have learned how to find gradients of the input (i.e., input_grad). How about the gradients of the filters (i.e., param_grad?)

Let's try change element 1 of the param by 1

[12]:
```
input_1d = np.array([1,2,3,4,5])
param_1d_2 = np.array([2,1,1]) #increase first element by 1

print(conv_1d_sum(input_1d, param_1d_2) - conv_1d_sum(input_1d, param_1d))
```

```
10.0
```

So we find that

$$\frac{\partial L}{\partial w_0} = 10$$

Recall this:

$$t = [0, 1, 2, 3, 4, 5, 0]$$

$$w = [1, 1, 1]$$

$$o_0 = t_0 * w_0 + t_1 * w_1 + t_2 * w_2$$
$$o_1 = t_1 * w_0 + t_2 * w_1 + t_3 * w_2$$
$$o_2 = t_2 * w_0 + t_3 * w_1 + t_4 * w_2$$
$$o_3 = t_3 * w_0 + t_4 * w_1 + t_5 * w_2$$
$$o_4 = t_4 * w_0 + t_5 * w_1 + t_6 * w_2$$

We can clearly see that w_o is changing the convolution sum in respect of t_0 to t_4. Using the same logic as above which we can get

$$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial o_0} * \frac{\partial o_0}{\partial w_0} + \frac{\partial L}{\partial o_1} * \frac{\partial o_1}{\partial w_0} + \frac{\partial L}{\partial o_2} * \frac{\partial o_2}{\partial w_0} + \frac{\partial L}{\partial o_3} * \frac{\partial o_3}{\partial w_0} + \frac{\partial L}{\partial o_4} * \frac{\partial o_4}{\partial w_0}$$

$$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial o_0} * t_0 + \frac{\partial L}{\partial o_1} * t_1 + \frac{\partial L}{\partial o_2} * t_2 + \frac{\partial L}{\partial o_3} * t_3 + \frac{\partial L}{\partial o_4} * t_4$$

since t_0 is a 0, and $\frac{\partial L}{\partial o_i} = 1$, thus the gradient of the first element is indeed 10:

$$\frac{\partial L}{\partial w_0} = w_0^{grad} = t_1 + t_2 + t_3 + t_4 = 1 + 2 + 3 + 4 = 10$$

The general pattern is:

$$w_0^{grad} = t_0 + t_1 + t_2 + t_3 + t_4$$
$$w_1^{grad} = t_1 + t_2 + t_3 + t_4 + t_5$$
$$w_2^{grad} = t_2 + t_3 + t_4 + t_5 + t_6$$

How to code this?

Luckily, you can clearly see that the indices are moving the same direction, thus it is easy to code like this. We simply define grad to be

$$grad = [1, 1, 1, 1, 1] = [\frac{\partial L}{\partial o_0}, \frac{\partial L}{\partial o_1}, \frac{\partial L}{\partial o_2}, \frac{\partial L}{\partial o_3}, \frac{\partial L}{\partial o_4}]$$

and input_pad as follows:

$$input_{pad} = [0, 1, 2, 3, 4, 5, 0]$$

simply let it multiply with the input_pad of [0, 1, 2, 3, 4, 5, 0] like this:

[0, 1, 2, 3, 4, 5, 0] [1, 1, 1, 1, 1] = gradient of w0 [1, 1, 1, 1, 1] = gradient of w1 [1, 1, 1, 1, 1] = gradient of w2

[13]:
```
def _param_grad_1d(input_: ndarray,
                   param: ndarray,
                   grad: ndarray = None) -> ndarray:

    param_len = param.shape[0]
    param_mid = param_len // 2
    input_pad = _pad_1d(input_, param_mid)  #[0, 1, 2, 3, 4, 5, 0]

    #1. pepare the output gradients
    if grad is None:
        grad = np.ones_like(input_) #[1, 1, 1, 1, 1]
    else:
        assert_same_shape(input_, grad)

    #prepare param_grad which has grad of the three w
    #thus the initial look can be [0, 0, 0]
    param_grad = np.zeros_like(param) #[0, 0, 0]

    #2. Use the padded output gradients, along with padded input, to compute␣
↪the param gradient
    for i in range(input_.shape[0]):
        for p in range(param.shape[0]):
            #as w increase, shift input_pad right by w amount
            param_grad[p] += input_pad[i+p] * grad[i]

    assert_same_shape(param_grad, param)

    return param_grad
```

[14]:
```
_param_grad_1d(input_1d, param_1d)
```

[14]:
```
array([10, 15, 14])
```

1.1.7 1D input with batch (sample > 1)

How about if we have more samples of the 1D input like this?

[15]:
```
input_1d_batch = np.array([[0,1,2,3,4,5,6],
                           [1,2,3,4,5,6,7]])
```

Padding

In fact, this is simple, we simply run cov_1d on the first sample, and iterate and stack the results on top.

[16]:
```
def _pad_1d_batch(input_: ndarray,
                  padding: int) -> ndarray:
    outs = [_pad_1d(sample, padding) for sample in input_]
    return np.stack(outs)
```

[17]:
```
_pad_1d_batch(input_1d_batch, 1)
```

[17]:
```
array([[0, 0, 1, 2, 3, 4, 5, 6, 0],
       [0, 1, 2, 3, 4, 5, 6, 7, 0]])
```

Forward pass

Same concept. For forward pass, we simply iterate our previous method

[18]:
```
def conv_1d_batch(input_: ndarray,
                  param: ndarray) -> ndarray:

    outs = [conv_1d(sample, param) for sample in input_]
    return np.stack(outs)
```

[19]:
```
conv_1d_batch(input_1d_batch, param_1d)
```

[19]:
```
array([[ 1.,  3.,  6.,  9., 12., 15., 11.],
       [ 3.,  6.,  9., 12., 15., 18., 13.]])
```

Backward pass: gradients

For input_grad, it's the same concept. We simply do a for loop on the previous function we have already defined

[20]:
```
def input_grad_1d_batch(input_: ndarray,
                        param: ndarray) -> ndarray:

    #first perform a forward pass
    out = conv_1d_batch(input_, param)

    #generate grad for input to the _input_grad_1d function
    grad = np.ones_like(out)

    batch_size = grad.shape[0]

    grads = [_input_grad_1d(input_[i], param, grad[i]) for i in␣
↪range(batch_size)]

    return np.stack(grads)
```

[21]:
```
input_grad_1d_batch(input_1d_batch, param_1d)
```

[21]:
```
array([[2, 3, 3, 3, 3, 3, 2],
       [2, 3, 3, 3, 3, 3, 2]])
```

However, for param_grad, since param_grad is a filter that dependent across samples, thus we need to change our previous code. So, to compute the parameter gradient, we have to loop through all of the observations and increment the appropriate values of the parameter gradient as we do so. Still, this just involves adding an outer for loop to the code to compute the parameter gradient that we saw earlier.

[22]:
```
def param_grad_1d_batch(input_: ndarray,
                        param: ndarray) -> ndarray:

    grad = np.ones_like(input_)

    input_pad = _pad_1d_batch(input_, 1)

    param_grad = np.zeros_like(param)

    for s in range(input_.shape[0]):
        for i in range(input_.shape[1]):
            for w in range(param.shape[0]):
                param_grad[w] += input_pad[s][i+w] * grad[s][i]

    return param_grad
```

[23]:
```
param_grad_1d_batch(input_1d_batch, param_1d)
```

[23]:
```
array([36, 49, 48])
```

1.1.8 2D convolutions

The 2D convolution is a straightforward extension of the 1D case because, fundamentally, the way the input is connected to the output via the filters in each dimension of the 2D case is identical to the 1D case. Consider this data:

[24]:
```
imgs_2d_batch = np.random.randn(3, 5, 5)
param_2d = np.random.randn(3, 3)
```

Forward pass Recall that for 1D convolutions the code for computing the output given the input and the parameters on the forard pass looked as follows:

def conv_1d(input_: ndarray, param: ndarray) -> ndarray: param_len = param.shape[0]
param_mid = param_len // 2
input_pad = _pad_1d(input_, param_mid)
output = np.zeros(input_.shape) for o in range(output.shape[0]): for w in range(param_len): output[o] += param[w] * input_pad[o+w] return output

For 2D convolutions, instead of 1D output, we simply make it loop twice for height and width. In addition, instead of 1D param (filter), we break it into two loops, one for filter width, and another for filter height. The code will be simple as this:

[25]:
```
#for each sample
def _conv_sample_2d(sample: ndarray,
                            param: ndarray):

    param_mid = param.shape[0] // 2

    sample_pad = _pad_2d_sample(sample, param_mid)

    out = np.zeros_like(sample)

    for i_w in range(sample.shape[0]):  #loop through the image height
        for i_h in range(sample.shape[1]):  #loop through the image width
            for p_w in range(param.shape[0]): #loop through the filter width
                for p_h in range(param.shape[1]): #loop through the filter␣
 ↪height
                    out[i_w][i_h] += param[p_w][p_h] *␣
 ↪sample_pad[i_w+p_w][i_h+p_h]
    return out

#for many samples...simply a for loop
def _conv_2d(img_batch: ndarray,
                        param: ndarray):

    assert_dim(img_batch, 3)

    outs = [_conv_sample_2d(sample, param) for sample in img_batch]

    return np.stack(outs)
```

Padding Of course, this function would requires us to implement _pad_2d_samples which is actually very straightforward. Recall that the code is like this:

def *pad_1d(input*: ndarray, padding: int) -> ndarray: zero = np.array([0]) zero = np.repeat(zero, padding) #number of zeros * num return np.concatenate([zero, input_, zero])

[26]:
```
def _pad_2d_sample(sample: ndarray,
                   padding: int):

    input_pad = _pad_1d_batch(sample, padding)  #this will add left and right

    #these are zeros that will be added on top and bottom.  Padding*2 so␣
 ↪account for both left and right
    zero = np.zeros((padding, sample.shape[0] + padding * 2))
```

```
    return np.concatenate([zero, input_pad, zero])

def _pad_2d(img_batch: ndarray,
            padding: int):

    outs = [_pad_2d_sample(sample, padding) for sample in img_batch]

    return np.stack(outs)
```

Let's first check whether the padding works fine:

[27]:
```
print("Original shape: ", imgs_2d_batch.shape)
print("Padded shape (must plus 2): ", _pad_2d(imgs_2d_batch, 1).shape)
```

```
Original shape:  (3, 5, 5)
Padded shape (must plus 2):  (3, 7, 7)
```

Let's try our forward pass for 2D data with samples

[28]:
```
print("Original shape: ", imgs_2d_batch.shape)
print("Convoluted shape(must be same): ", _conv_2d(imgs_2d_batch, param_2d).
 ↪shape)
```

```
Original shape:  (3, 5, 5)
Convoluted shape(must be same):  (3, 5, 5)
```

Backward pass *Input gradients*

Recall that for input_grad, our code is like this:

def _input_grad_1d(input_: ndarray, param: ndarray, grad: ndarray = None) -> ndarray: param_len = param.shape[0] param_mid = param_len // 2 if grad is None: grad = np.ones_like(input_) else: assert_same_shape(input_, grad) grad = _pad_1d(grad, param_mid) #[0, 1, 1, 1, 1, 1, 0] input_grad = np.zeros_like(input_) for i in range(input_.shape[0]): for w in range(param.shape[0]): input_grad[i] += grad[i + param_len - 1 - w] * param[w] return input_grad

In the 2D case, we simply break our input to 2 loops since we have image width and height, and we also break the param into 2 loops since we have width and height of the filters. Yes, that's it!

[29]:
```
def _input_grad_sample_2d(sample: ndarray,
                          param: ndarray,
                          grad: ndarray) -> ndarray:

    param_size = param.shape[0]
    grad = _pad_2d_sample(grad, param_size // 2)  #sample refers to each sample
    input_grad = np.zeros_like(sample)

    for i_w in range(sample.shape[0]):  #img width
```

```
        for i_h in range(sample.shape[1]):  #img height
            for p_w in range(param_size):      #filter width
                for p_h in range(param_size):    #filter height
                    input_grad[i_w][i_h] += grad[i_w + param_size - 1 -␣
↪p_w][i_h + param_size - 1 - p_h] \
                    * param[p_w][p_h]

    return input_grad

def _input_grad_2d(samples: ndarray,
                   param: ndarray,
                   grad: ndarray) -> ndarray:
    grads = [_input_grad_sample_2d(samples[i], param, grad[i]) for i in␣
↪range(grad.shape[0])]
    return np.stack(grads)
```

[30]:
```
img_grads = _input_grad_2d(imgs_2d_batch,
                           param_2d,
                           np.ones_like(imgs_2d_batch))
img_grads.shape  ##img grad should equal to img size
```

[30]:
```
(3, 5, 5)
```

Param gradients

For the parameter gradient, we have to loop through all the images in the batch and add components from each one to the appropriate places in the parameter gradient. The reason is because the filter gradients overlap over multiple samples.

This is the code we have used earlier for 1D batch

def param_grad_1d_batch(input_: ndarray, param: ndarray) -> ndarray: grad = np.ones_like(input_) input_pad = _pad_1d_batch(input_, 1) param_grad = np.zeros_like(param)
for s in range(input_.shape[0]): for i in range(input_.shape[1]): for w in range(param.shape[0]): param_grad[w] += input_pad[s][i+w] * grad[s][i]
return param_grad

What we have to do is that we have to replace i with two loops representing img width and height, and replace w with two loops, representing filter width and height. Let's do it.

[31]:
```
def _param_grad_2d(input_: ndarray,
                   grad: ndarray,
                   param: ndarray) -> ndarray:

    param_size = param.shape[0]  #for filter width, height
    input_pad = _pad_2d(input_, param_size // 2)  #input_pad

    param_grad = np.zeros_like(param)
```

```
    img_shape = input_.shape[1:]  #get only 5, 5

    for s in range(input_.shape[0]):  #loop samples
        for i_w in range(img_shape[0]):  #loop img width
            for i_h in range(img_shape[1]):  #loop img height
                for p_w in range(param_size):   #loop param width
                    for p_h in range(param_size):   #loop param height
                        param_grad[p_w][p_h] += input_pad[s][i_w+p_w][i_h+p_h] \
                        * grad[s][i_w][i_h]
    return param_grad
```

[32]:
```
param_grad = _param_grad_2d(imgs_2d_batch,
                            np.ones_like(imgs_2d_batch),
                            param_2d)
print("Param shape: ", param_2d.shape)
print("Param_grad shape (must equal param2d): ", param_grad.shape)
```

```
Param shape:  (3, 3)
Param_grad shape (must equal param2d):  (3, 3)
```

Testing gradients Let's test whether our code really works. Let's make a code to perform convolution sum for 2d batch, and then we can manually subtract to find the gradients

[33]:
```
def conv_2d_sum(img_batch: ndarray,
                param: ndarray):

    out = _conv_2d(img_batch, param)

    return out.sum()
```

[34]:
```
#Testing input gradient

#let's randomly change one pixel of a sample by 1
imgs_2d_batch_2 = imgs_2d_batch.copy()
imgs_2d_batch_2[0][2][2] += 1

manual = conv_2d_sum(imgs_2d_batch_2, param_2d) - conv_2d_sum(imgs_2d_batch,␣
↪param_2d)
print("Manual: ", manual)
print("Our code: ", img_grads[0][2][2]) #this is already computed above
```

```
Manual:  -0.5138257424616004
Our code:  -0.5138257424615983
```

[35]:
```
#Testing param gradient

param_2d_2 = param_2d.copy()
```

```
param_2d_2[0][2] += 1

manual = conv_2d_sum(imgs_2d_batch, param_2d_2) - conv_2d_sum(imgs_2d_batch,␣
 ↪param_2d)
print("Manual: ", manual)
print("Our code: ", param_grad[0][2]) #this is already computed above
```

```
Manual:  -2.658933224682264
Our code:  -2.6589332246822615
```

1.1.9 Channels

Ok….so we have 2D input with batch size. But we still have one more dimension, and that is number of channels. Currently, we assume only one filter so far, what if we have more filters?

The answer, as it was when we added batches earlier, is simple: we add two outer for loops to the code we've already seen—one loop for the input channels and another for the output channels. By looping through all combinations of the input channel and the output channel, we make each output feature map a combination of all of the input feature maps, as desired.

Forward pass Let's code the forward pass for each sample which have channels. This is actually easy, we simply copy our previous code _conv_sample_2d and add two more outer loops, make sure we loop through.

For your reference, this is the previous code we use:

def _conv_sample_2d(sample: ndarray, param: ndarray): param_mid = param.shape[0] // 2 sample_pad = _pad_2d_sample(sample, param_mid) out = np.zeros_like(sample) for o_w in range(out.shape[0]): #loop through the image height for o_h in range(out.shape[1]): #loop through the image width for p_w in range(param.shape[0]): #loop through the filter width for p_h in range(param.shape[1]): #loop through the filter height out[o_w][o_h] += param[p_w][p_h] * sample_pad[o_w+p_w][o_h+p_h] return out

[36]:
```
def _conv_sample_channels_2d(sample: ndarray,
                             param: ndarray):
    '''
    sample: [channels, img_width, img_height]
    param: [in_channels, out_channels, fil_width, fil_height]
    '''
    assert_dim(sample, 3)
    assert_dim(param, 4)

    param_size = param.shape[2]
    param_mid = param_size // 2
    sample_pad = _pad_2d_sample_channel(sample, param_mid)  #pad the input

    #define for loops
    in_channels = param.shape[0]
    out_channels = param.shape[1]
```

```
    img_size = sample.shape[1]

    out = np.zeros((out_channels,) + sample.shape[1:])

    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for i_w in range(img_size):
                for i_h in range(img_size):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            out[c_out][i_w][i_h] += \
                            param[c_in][c_out][p_w][p_h] *␣
 ↪sample_pad[c_in][i_w+p_w][i_h+p_h]
    return out

def _conv_channels_2d(input_: ndarray,
                      param: ndarray) -> ndarray:
    '''
    input_: [batch_size, channels, img_width, img_height]
    param: [in_channels, out_channels, fil_width, fil_height]
    '''
    outs = [_conv_sample_channels_2d(sample, param) for sample in input_]

    return np.stack(outs)
```

Before we can test this code, let's make sure we implement the padding for channels as well:

[37]:
```
def _pad_2d_sample_channel(input_: ndarray,
                           padding: int):
    '''
    input_ has dimension [num_channels, image_width, image_height]
    '''
    return np.stack([_pad_2d_sample(channel, padding) for channel in input_])

def _pad_2d_channel(input_: ndarray,
                    padding: int):
    '''
    input_ has dimension [batch_size, num_channels, image_width, image_height]
    '''
    return np.stack([_pad_2d_sample_channel(sample, padding) for sample in␣
 ↪input_])
```

Let's test our code whether it works. I will leave the gradient test to you.

[38]:
```
img = np.random.randn(10, 3, 32, 32)
param = np.random.randn(3, 16, 5, 5)
```

```
print(_conv_channels_2d(img, param).shape)  #must equal 10, 16, 32, 32
```

```
(10, 16, 32, 32)
```

Backward pass *Input gradients*

We simply add two outer loops for in and out channels.

Here is the previous code for input_grad for your comparison references

def _input_grad_sample_2d(sample: ndarray, param: ndarray, grad: ndarray) -> ndarray: param_size = param.shape[0] grad = _pad_2d_sample(grad, param_size // 2) #sample refers to each sample input_grad = np.zeros_like(sample) for i_w in range(sample.shape[0]): #img width for i_h in range(sample.shape[1]): #img height for p_w in range(param_size): #filter width for p_h in range(param_size): #filter height input_grad[i_w][i_h] += grad[i_w + param_size - 1 - p_w][i_h + param_size - 1 - p_h] * param[p_w][p_h] return input_grad def _input_grad_2d(samples: ndarray, param: ndarray, grad: ndarray) -> ndarray: grads = [_input_grad_sample_2d(samples[i], param, grad[i]) for i in range(grad.shape[0])]
return np.stack(grads)

[39]:
```
def _input_grad_sample_channel_2d(sample: ndarray,
                      grad: ndarray,
                      param: ndarray) -> ndarray:
    '''
    sample: [in_channels, img_width, img_height]
    grad: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''
    #for looping
    param_size = param.shape[2]
    img_size = sample.shape[1]
    in_channels = sample.shape[0]
    out_channels = param.shape[1]

    #for holding the result
    input_grad = np.zeros_like(sample)

    #for the output grad
    grad = _pad_2d_sample_channel(grad, param_size // 2)

    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for i_w in range(img_size):
                for i_h in range(img_size):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            input_grad[c_in][i_w][i_h] += \
                                ␣
↪grad[c_out][i_w+param_size-p_w-1][i_h+param_size-p_h-1] \
```

```
                                * param[c_in][c_out][p_w][p_h]
    return input_grad

def _input_grad_channel_2d(samples: ndarray,
                grad: ndarray,
                param: ndarray) -> ndarray:

    grads = [_input_grad_sample_channel_2d(samples[i], grad[i], param) for i in␣
 ↪range(grad.shape[0])]

    return np.stack(grads)
```

Param gradients

Recall that the previous code without channel looks like this:

def _param_grad_2d(input_: ndarray, grad: ndarray, param: ndarray) -> ndarray:= param_size = param.shape[0] #for filter width, height input_pad = _pad_2d(input_, param_size // 2) #input_pad param_grad = np.zeros_like(param) img_shape = input_.shape[1:] #get only 5, 5 for s in range(input_.shape[0]): #loop samples for i_w in range(img_shape[0]): #loop img width for i_h in range(img_shape[1]): #loop img height for p_w in range(param_size): #loop param width for p_h in range(param_size): #loop param height param_grad[p_w][p_h] += input_pad[s][i_w+p_w][i_h+p_h] * grad[s][i_w][i_h] return param_grad

[40]:
```
def _param_grad_2d_channel(input_: ndarray,
                grad: ndarray,
                param: ndarray) -> ndarray:
    '''
    input_: [in_channels, img_width, img_height]
    grad: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''

    #for looping
    sample_size = input_.shape[0]
    param_size = param.shape[2]
    img_size = input_.shape[2]
    in_channels = input_.shape[1]
    out_channels = grad.shape[1]
    img_shape = grad.shape[2:]

    #for holding the results
    param_grad = np.zeros_like(param)

    #use for calculating param grads
    input_pad = _pad_2d_channel(input_, param_size // 2)

    for i in range(sample_size):
```

```
        for c_in in range(in_channels):
            for c_out in range(out_channels):
                for i_w in range(img_shape[0]):
                    for i_h in range(img_shape[1]):
                        for p_w in range(param_size):
                            for p_h in range(param_size):
                                param_grad[c_in][c_out][p_w][p_h] += \
                                input_pad[i][c_in][i_w+p_w][i_h+p_h] \
                                * grad[i][c_out][i_w][i_h]
    return param_grad
```

Testing gradients To test, let's simply create a convolution sum based on channels

[41]:
```
def conv_2d_channel_sum(imgs: ndarray,
                        param: ndarray):
    return _conv_channels_2d(imgs, param).sum()
```

[42]:
```
#let's create a random image
imgs = np.random.randn(10, 3, 32, 32)  #(samples, channels, width, height)
param = np.random.randn(3, 16, 5, 5)   #(in_channels, out_channels, width,␣
 ↪height)

imgs_2 = imgs.copy()
imgs_2[3][1][2][19] += 1

#let's test our input_grad function

print("Manual: ", conv_2d_channel_sum(imgs_2, param) -␣
 ↪conv_2d_channel_sum(imgs, param))

input_grad = _input_grad_channel_2d(imgs,
                        np.ones((10, 16, 32, 32)),
                        param)

print("Our code: ", input_grad[3][1][2][19])
```

```
Manual:  -5.345935650419051
Our code:  -5.345935650418792
```

[43]:
```
#let's test our param_grad function
param_2 = param.copy()
param_2[0][8][0][2] += 1

print("Manual: ", conv_2d_channel_sum(imgs, param_2) -␣
 ↪conv_2d_channel_sum(imgs, param))

param_grad = _param_grad_2d_channel(imgs,
```

```
                          np.ones((10, 16, 32, 32)),
                          param)

print("Our code: ", param_grad[0][8][0][2])
```

```
Manual:  55.34918873134302
Our code:  55.34918873134217
```

1.1.10 Putting everything together!!

We need to implement a few more pieces before we can have a working CNN model:

1. We have to implement the Flatten operation which is just simple reshape method
2. We have to incorporate this Operation as well as the Conv2DOpOperation into a Conv2D Layer
3. Finally, for it to be runnable, we have to write an optimized version. I have included this code in the

Flattening Flatten is fairly easy:

[44]:
```
class Flatten(Operation):
    def __init__(self):
        super().__init__()

    def _output(self, inference: bool = False) -> ndarray:
        #squeeze everything to the second dimension
        #(10, 3, 32, 32) --> (10, 3072)
        return self.input_.reshape(self.input_.shape[0], -1)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        #simply transform back
        return output_grad.reshape(self.input_.shape)
```

Layer Setting up the Layer is also fairly easy. We just have to make sure it is possible to set the flag flatten depending on whether we want the output of this layer to be passed forward into another convolutional layer or passed into another fully connected layer for predictions. Flatten should be set True when the next layer is a Dense layer where it expects a flatten input of shape (num_samples, num_features)

[45]:
```
class Conv2D(Layer):
    def __init__(self,
                 out_channels: int,
                 param_size: int,
                 dropout: int = 1.0,
                 weight_init: str = "glorot",
                 activation: Operation = Linear(),
                 flatten: bool = False) -> None:
```

```
        super().__init__(out_channels)
        self.param_size = param_size
        self.activation = activation
        self.flatten = flatten
        self.dropout = dropout
        self.weight_init = weight_init
        self.out_channels = out_channels

    def _setup_layer(self, input_: ndarray) -> ndarray:

        self.params = []
        in_channels = input_.shape[1]

        if self.weight_init == "glorot":
            scale = 2/(in_channels + self.out_channels)
        else:
            scale = 1.0

        conv_param = np.random.normal(loc=0,
                                      scale=scale,
                                      size=(input_.shape[1],  # input channels
                                     self.out_channels,
                                     self.param_size,
                                     self.param_size))

        self.params.append(conv_param)

        self.operations = []
        self.operations.append(Conv2D_Op(conv_param))
        self.operations.append(self.activation)

        if self.flatten:
            self.operations.append(Flatten())

        if self.dropout < 1.0:
            self.operations.append(Dropout(self.dropout))

        return None
```

Operation and code optimization The Operation will need to be revised, to be a more optimized version. As those of you who are familiar with computational complexity will realize, this code is catastrophically slow: to calculate the parameter gradient, we needed to write seven nested for loops! There's nothing wrong with doing this, since the purpose of writing the convolution operation from scratch was to solidify our understanding of how CNNs work.

We'll show how to express the batch, multichannel convoution operation in terms of a batch matrix multiplication to implement it efficiently in NumPy. To understand how the convolution works,

consider what happens in the forward pass of a fully connected neural network:

We receive an input of size [batch_size, in_features]. We multiply it by a parameter of size [in_features, out_features]. We get a resulting output of size [batch_size, out_features].

In a convolutional layer, by contrast:

We receive an input of size [batch_size, in_channels, img_height, img_width]. We convolve it with a parameter of size [in_channels, out_channels, param_height, param_width]. We get a resulting output of size [batch_size, out_channels, img_height, img_width].

So the question is how to make the multiplication possible, since we need to prepare it to be something in the form of (i, j) @ (j, k) = (i, k)

The key to making the convolution operation look more like a regular feed-forward operation is to first extract img_height × img_width "image patches" from each channel of the input image. Once these patches are extracted, the input can be reshaped so that the convolution operation can be expressed as a batch matrix multiplication using NumPy's np.matmul function."

The steps are the followings:

1. Get image patches of size [img_height x img_width, batch_size, in_channels, filter_size, filter_size]
2. Reshape this to be [batch_size, img_height × img_width, in_channels × filter_size× filter_size]
3. Reshape parameter to be [in_channels × filter_size × filter_size, out_channels]
4. After we do a batch matrix multiplication, the result will be [batch_size, img_height × img_width, out_channels]
5. Reshape this to be [batch_size, out_channels, img_height, img_width]

That's it. That's the forward pass!

For backward pass, it's the same concept. We are just finding the right squeeze-reorder-reshape method so that things can be multiplied into desired shape. Fall in love with vectors? Fantastic right?

[46]:
```
class Conv2D_Op(ParamOperation):

    def __init__(self, W: ndarray):
        super().__init__(W)
        self.param_size = W.shape[2]
        self.param_pad = self.param_size // 2

    def _pad_1d(self, inp: ndarray) -> ndarray:
        z = np.array([0])
        z = np.repeat(z, self.param_pad)
        return np.concatenate([z, inp, z])

    def _pad_1d_batch(self,
                      inp: ndarray) -> ndarray:
```

```
        outs = [self._pad_1d(obs) for obs in inp]
        return np.stack(outs)

    def _pad_2d_obs(self,   #obs stands for observation
                    inp: ndarray):
        inp_pad = self._pad_1d_batch(inp)

        other = np.zeros((self.param_pad, inp.shape[0] + self.param_pad * 2))

        return np.concatenate([other, inp_pad, other])

    def _pad_2d_channel(self,
                        inp: ndarray):
        '''
        inp has dimension [num_channels, image_width, image_height]
        '''
        return np.stack([self._pad_2d_obs(channel) for channel in inp])

    def _get_image_patches(self,
                           input_: ndarray):
        '''
        imgs_batch: [batch_size, channels, img_width, img_height]
        '''

        #pad the images
        imgs_batch_pad = np.stack([self._pad_2d_channel(obs) for obs in input_])
        patches = []
        img_height = imgs_batch_pad.shape[2]

        #for each location in the images, cut the filter width x filter height␣
↪image
        #and stack them
        for h in range(img_height-self.param_size+1):
            for w in range(img_height-self.param_size+1):
                patch = imgs_batch_pad[:, :, h:h+self.param_size, w:w+self.
↪param_size]
                patches.append(patch)
        #[img_height * img_width, batch_size, in_channels, param_width,␣
↪param_height]
        return np.stack(patches)

    def _output(self,
                inference: bool = False):
        '''
        conv_in: [batch_size, channels, img_width, img_height]
        param: [in_channels, out_channels, fil_width, fil_height]
        '''
```

```
    #     assert_dim(obs, 4)
    #     assert_dim(param, 4)
        batch_size = self.input_.shape[0]
        img_height = self.input_.shape[2]
        img_size = self.input_.shape[2] * self.input_.shape[3]
        patch_size = self.param.shape[0] * self.param.shape[2] * self.param.
↪shape[3]

        #shape: [img_height * img_width, batch_size, in_channels, param_width,␣
↪param_height]
        patches = self._get_image_patches(self.input_)

        #reshape to: [batch_size, img_height * img_width, in_channels,␣
↪param_width, param_height]
        #then squeeze into [batch_size, img_height * img_width, in_channels *␣
↪param_width * param_height]
        patches_reshaped = (patches
                            .transpose(1, 0, 2, 3, 4)
                            .reshape(batch_size, img_size, -1))

        #shape of param: [in_channels, out_channels, param_width, param_height]
        #make it into: [in_channels, param_width, param_height, out_channels]
        #then squeeze into: [in_channel * param_width * param_height,␣
↪out_channels]
        param_reshaped = (self.param
                          .transpose(0, 2, 3, 1)
                          .reshape(patch_size, -1))

        #now patch @ param =
        #[batch_size, img_height * img_width, in_channels * param_width *␣
↪param_height] @
        #[in_channel * param_width * param_height, out_channels] =
        #[batch_size, img_height * img_width, out_channels]
        #then reshape into [batch_size, img_height, img_width, out_channels]
        #then reorder into [batch_size, out_channels, img_height, img_width]
        #yay!
        output_reshaped = (
            np.matmul(patches_reshaped, param_reshaped)
            .reshape(batch_size, img_height, img_height, -1)
            .transpose(0, 3, 1, 2))

        return output_reshaped


    def _input_grad(self, output_grad: np.ndarray) -> np.ndarray:
```

```
        #simple parameter
        batch_size = self.input_.shape[0]
        img_size = self.input_.shape[2] * self.input_.shape[3]
        img_height = self.input_.shape[2]

        #first the output_grad has shape of [batch_sizes, out_channel,␣
↪param_width, param_height]
        #then, _get_image_patches will give out a shape of
        #[img_height * img_width, batch_size, out_channels, param_width,␣
↪param_height]
        #then we reorder to [batch_size, img_height * img_width, out_channels,␣
↪param_width, param_height]
        #then squeeze into [batch_size * img_height * img_width, out_channels *␣
↪param_width * param_height]
        output_patches = (self._get_image_patches(output_grad)
                          .transpose(1, 0, 2, 3, 4)
                          .reshape(batch_size * img_size, -1))
        #param shape is
        #[in_channels, out_channels, param_width, param_height]
        #reshape into [in_channels, out_channels * param_width * param_height]
        #transpose to get [out_channels * param_width * param_height,␣
↪in_channels]
        param_reshaped = (self.param
                          .reshape(self.param.shape[0], -1)
                          .transpose(1, 0))

        #gradient of input is simply gradients @ W.T
        #output_patches have shape of
        #[batch_size * img_height * img_width, out_channels * param_width *␣
↪param_height]
        #param shape of [out_channels * param_width * param_height, in_channels]
        #the dot product is [batch_size * img_height * img_width, in_channels]
        #then we reshape to [batch_size, img_height, img_width, in_channels]
        #then we reorder to [batch_size, in_channels, img_height, img_width]
        #input_grad should be the same shape as input, which is correct
        return (
            np.matmul(output_patches, param_reshaped)
            .reshape(batch_size, img_height, img_height, self.param.shape[0])
            .transpose(0, 3, 1, 2)
        )


    def _param_grad(self, output_grad: ndarray) -> ndarray:

        batch_size = self.input_.shape[0]
        img_size = self.input_.shape[2] * self.input_.shape[3]
```

```
        in_channels = self.param.shape[0]
        out_channels = self.param.shape[1]

        #input has a shape of
        #[batch_size, in_channels, img_width, img_height]
        #_get_image_patches will get [img_width * img_height, batch_size,␣
↪in_channels, param_height, param_width]
        #reshape to [batch_size * img_width * img_height, in_channels *␣
↪param_height * param_width]
        #then reorder to [in_channels * param_height * param_width, batch_size␣
↪* img_width * img_height]
        in_patches_reshape = (
            self._get_image_patches(self.input_)
            .reshape(batch_size * img_size, -1)
            .transpose(1, 0)
            )

        #output grad has a shape of
        #[batch_sizes, out_channel, img_width, img_height]
        #reorder to [batch_sizes, img_width, img_height, out_channel]
        #reshape to [batch_sizes * img_width * img_height, out_channel]
        out_grad_reshape = (output_grad
                            .transpose(0, 2, 3, 1)
                            .reshape(batch_size * img_size, -1))

        #gradient of param is simply X.T @ gradients
        #in_patch @ out_grad
        #[in_channels * param_height * param_width, batch_size * img_width *␣
↪img_height]  @
        #[batch_sizes * img_width * img_height, out_channel] =
        #[in_channels * param_height * param_width, out_channel]
        #then we reshape to [in_channels, param_width, param_height,␣
↪out_channel]
        #reorder to [in_channels, out_channels, param_width, param_height]
        return (np.matmul(in_patches_reshape,
                          out_grad_reshape)
                .reshape(in_channels, self.param_size, self.param_size,␣
↪out_channels)
                .transpose(0, 3, 1, 2))
```

Now we can actually try out code here

[47]:
```
from sklearn.datasets import fetch_openml

#1. Load data
mnist = fetch_openml('mnist_784', version=1, cache=True)
mnist.target = mnist.target.astype(int)
```

```
#2. Test train split
X_train = mnist['data'][:60000]
y_train = mnist['target'][:60000]

X_test = mnist['data'][60000:]
y_test = mnist['target'][60000:]

#3. Standardize data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#4. Reshape X so that it has channel data
print("X train shape: ", X_train.shape)
X_train_conv, X_test_conv = X_train.reshape(-1, 1, 28, 28), X_test.reshape(-1,␣
 ↪1, 28, 28)
print("X conv shape: ", X_train_conv.shape)

#4. One hot encoding
from sklearn import preprocessing
onehot = preprocessing.OneHotEncoder()

#sklearn expects a 2D array thus we have to reshape to (-1, 1)
y_train_encode = onehot.fit_transform(y_train.reshape(-1, 1)).toarray()
y_test_encode = onehot.fit_transform(y_test.reshape(-1, 1)).toarray()

print(y_train_encode.shape, y_test_encode.shape)
```

```
X train shape:  (60000, 784)
X conv shape:  (60000, 1, 28, 28)
(60000, 10) (10000, 10)
```

Let's try a CNN model now! We gonna use only one CNN layer...since training CNN can take a lot of time.

[48]:
```
model = NeuralNetwork(
    layers=[Conv2D(out_channels=32,
                   param_size=5,
                   dropout=0.8,
                   weight_init="glorot",
                   flatten=True,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Linear())],
            loss = SoftmaxCrossEntropy(),
seed=20200720)
```

```
trainer = Trainer(model, SGDMomentum(lr = 0.01, momentum=0.9))
trainer.fit(X_train_conv, y_train_encode, X_test_conv, y_test_encode,
            epochs = 1,
            eval_every = 1,
            seed=20200720,
            batch_size=60)
```

```
Validation loss after 1 epochs is 0.508
```

[49]:
```
#5. define a simple accuracy function
from sklearn.metrics import accuracy_score

def calc_accuracy(model, X_test, y_test):
    #getting the accuracy score with testing data
    preds = model.forward(X_test, inference=True)
    preds = np.argmax(preds, axis=1)
    print("Accuracy: ", accuracy_score(y_test, preds))

calc_accuracy(model, X_test_conv, y_test)
```

```
Accuracy:  0.9153
```

Yay, with only one layer of CNN, and one iteration, we can already achieved over 90% accuracy!

Phew! We are finally done. It's a bit tough but it's satisfying, right?

Next class, we shall explore other type of data, namely, sequential data (text, signal), in which the previous data affects how we interpret the current data. In this case, we have a specialized network called RNN (Recurrent Neural Netork).

[ ]: