

# Homework 2 Part 1

## An Introduction to Convolutional Neural Networks

11-785: Introduction to Deep Learning (Summer 2019)

OUT: May 31, 2019

DUE: NA

### 1 NumPy Based Convolutional Neural Networks

Your task for this homework is to implement the forward pass algorithm of a single 1D convolutional layer in the `layers.py` file. Your implementations will be compared with PyTorch. Python 3, NumPy $\geq$ 1.16 and PyTorch $\geq$ 1.0.0 are suggested for this homework.

Your implementation is tested using the local autograder `local_grader.py` file we provided in the handout. This file should be in the same directory as the `layers.py` file. Whenever you want to test your implementation, you execute the following command from the command line:

```
python local_grader.py
```

A detailed explanation of topics relevant to this homework can be found here:

<http://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2019/www/hwnotes/hw2/hw2.html>

### 2 Convolutional Neural Network (CNN) basics

In this section, some key concepts are discussed and definitions of some important terms are given. Links to additional information on some of the topics are also provided and we encourage you to use these resources to get more insight.

#### 2.1 What is a CNN?

Convolutional Neural Networks (CNNs) are a specialized kind of neural networks for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels [1].

Convolutional networks have been tremendously successful in practical applications. The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional

networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [1].

## 2.2 Kernels and Convolution

A filter (or kernel) is an integral component of a CNN.

Generally, it refers to an operator applied to the entirety of the image such that it transforms the information encoded in the pixels. In practice, however, the kernel is a smaller-sized matrix in comparison to the input dimensions of the image, that consists of real valued entries.

The kernels are then convolved with the input volume to obtain so-called ‘activation maps’ or feature maps. Activation maps indicate ‘activated’ regions, i.e. regions where features specific to the kernel have been detected in the input. The real values of the kernel matrix change with each learning iteration over the training set, indicating that the network is learning to identify which regions are of significance for extracting features from the data.

Links relevant to this topic:

- <http://cs231n.github.io/convolutional-networks/>
- <https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/>
- [http://machinelearningguru.com/computer\\_vision/basics/convolution/image\\_convolution\\_1.html](http://machinelearningguru.com/computer_vision/basics/convolution/image_convolution_1.html)
- [http://machinelearningguru.com/computer\\_vision/basics/convolution/convolution\\_layer.html](http://machinelearningguru.com/computer_vision/basics/convolution/convolution_layer.html)

## 2.3 Stride

A kernel is moved across the image left to right, top to bottom, with a one-pixel column change on the horizontal movements, then a one-pixel row change on the vertical movements.

The amount of movement between applications of the filter to the input image is referred to as the stride. I.e. stride is the number of pixels with which we slide our filter, horizontally or vertically. It is almost always symmetrical in height and width dimensions.

The default stride or strides in two dimensions is (1, 1) for the height and the width movement, performed when needed. And this default works well in most cases. The stride can be changed, which has an effect both on how the filter is applied to the image and, in turn, the size of the resulting feature map.

For example, the stride can be changed to (2, 2). This has the effect of moving the filter two pixels left for each horizontal movement of the filter and two pixels down for each vertical movement of the filter when creating the feature map.

Links relevant to this topic:

- <https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529>
- <https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/>

## 2.4 Channels

Color images have multiple channels, typically one for each color channel, such as red, green, and blue. From a data perspective, that means that a single image provided as input to the model is, in fact, three images.

A filter must always have the same number of channels as the input, often referred to as “depth“. If an input image has 3 channels (e.g. a depth of 3), then a filter applied to that image must also have 3 channels (e.g. a depth of 3). In this case, a 3×3 filter would in fact be 3×3×3 or [3, 3, 3] for rows, columns, and depth. Regardless of the depth of the input and depth of the filter, the filter is applied to the input using a dot product operation which results in a single value.

This means that if a convolutional layer has 32 filters, these 32 filters are not just two-dimensional for the two-dimensional image input, but are also three-dimensional, having specific filter weights for each of the three channels. Yet, each filter results in a single feature map. Which means that the depth of the output of applying the convolutional layer with 32 filters is 32 for the 32 feature maps created.

Links relevant to this topic:

- <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/>

## 3 Convolutional layer

Implement the forward function in the `Conv1D` class in `layers.py`. The class `Conv1D` has four arguments: `in_channel`, `out_channel`, `kernel_size` and `stride`. They are all positive integers. You can find the details of these arguments in the following PyTorch documentation:

<https://pytorch.org/docs/stable/nn.html#torch.nn.Conv1d>.

We do not consider other arguments such as padding.

Implement the `Conv1D` class so that it has similar usage and functionality to `torch.nn.Conv1d`.

**Note:** changing the shape/name of the provided attributes is not allowed. Like in HW1P1 we will check the value of these attributes.

### 3.1 Forward pass

Here, you are expected to implement the forward algorithm (shown below) of a single 1D CNN layer. The input  $\mathbf{x}$  is the input of the convolutional layer and the shape of  $\mathbf{x}$  is **(batch\_size, in\_channel, in\_width)**. The return value is the output of the convolutional layer and the shape is **(batch\_size, out\_channel, out\_width)**. Note that **in\_width** is an arbitrary positive integer and **out\_width** is determined by **in\_width**, **kernel\_size** and **stride**.

Below is a very high level description of the steps you can follow:

1. Compute the width of the output feature map:

$$output\ size = \lfloor \frac{(N - M)}{S} \rfloor + 1$$

Where:

N is the width of the input sequence.  
M is the size of the kernel.  
S is the stride.

Note the floor ( $\lfloor \ \rfloor$ ) function.

2. Loop through the width of the input sequence.
3. Take a slice of the input that has the same size as the kernel. Let the slice be denoted by the variable **segment**.
4. Compute the output. The output is simply the sum of the dot product between **segment** and the weight matrix (denoted in the layers.py file as **self.W**), and the bias (denoted as **self.b**). I.e.

$$Output = segment.dot(self.W.T) + self.b$$

The weight matrix and bias are already initialized for you with the proper shape.

5. Repeat steps 2-4 until it is impossible for you to slide the kernel anymore. Note: there is no padding.

## Note:

1. You have to know across which dimension to slice the input and which dimensions to index into.

2. This approach may require you to reshape variables `segment` and `self.W`.

You are NOT obliged to follow the above steps to arrive at the solution. In fact, there are other ways of solving the problem. For example, you can use NumPy's `tensor_dot` routine (instead of the regular `dot` routine) to perform element-wise matrix multiplication and followed by summing (which is basically convolution). Have a look at the following resources on how to use `tensor_dot` for convolution:

- [https://docs.scipy.org/doc/numpy/reference/generated/numpy.tensor\\_dot.html#numpy.tensor\\_dot](https://docs.scipy.org/doc/numpy/reference/generated/numpy.tensor_dot.html#numpy.tensor_dot)
- <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/>

**Test:** To test your implementation, we compare the results of your implementation with PyTorch class `torch.nn.Conv1d`. Details can be found in function `test_cnn_correctness_once` in the `local_grader.py` file. Below is a code snippet showing the logic that is used to test your solution.

```
import torch

import torch.nn as nn

import numpy as np

from torch.autograd import Variable

from layers import Conv1D

## initialize your layer and PyTorch layer

net1 = Conv1D(8, 12, 3, 2) # your layer

net2 = torch.nn.Conv1d(8, 12, 3, 2) # PyTorch layer

## initialize the inputs

x1 = np.random.rand(3, 8, 20)

x2 = Variable(torch.tensor(x1), requires_grad=True)

## Copy the parameters from the Conv1D class to PyTorch layer

net2.weight = nn.Parameter(torch.tensor(net1.W))
```

```
net2.bias = nn.Parameter(torch.tensor(net1.b))

## Your forward
y1 = net1(x1)

## PyTorch forward
y2 = net2(x2)

## Compare
def compare(x, y):
    y = y.detach().numpy()
    print(abs(x-y).max())
    return

compare(y1, y2)

1.7763568394002505e-15 ## example of good MAE
```

As you can see in the `local_grader.py` file, we check the maximum absolute error of the tensor elements. We run your codes 15 times to test your solution with different combinations of input channel, output channel, kernel size, stride, batch size, and input width. If the values your implementation returns can have a MAE smaller than  $1e-12$  (like the one shown in the above code snippet), then we think your implementation is right.

## 4 References

[1] I. Goodfellow, Y. Bengio and A. Courville, Deep learning. Cambridge (EE. UU.): MIT Press, 2016, p. 321.

## 5 Conclusion

That's all. As always, feel free to ask on Piazza if you have any questions.

Good luck and enjoy the challenge!