**Notes of Study on Mar 13 (Mon), 2023**

Based on lecture notes of Stanford CS231n

Module 2: Convolutional Neural Networks

[Convolutional Neural Networks: Architectures, Convolution / Pooling Layers](https://cs231n.github.io/convolutional-networks/)

layers, spatial arrangement, layer patterns, layer sizing patterns, AlexNet/ZFNet/VGGNet case studies, computational considerations

This section talks about the architecture of CNN and how to implement it with numpy and PyTorch respectively.

Suppose we have an input **X** of size

**(N, C, W, H)**

i.e.

N images each of size (image width, image height, # channels)

A simplified convolutional neural network looks like:

ダイアグラム

自動的に生成された説明

Where a **convolutional layer**:

With filters Filter of a size (filter weight, filter height) and bias B.

For convenience, let

Computes the output of neurons, each computing a dot product between their weights and a small region they are connected to in the input volume, where

And a **pooling layer**:

Applies perform a downsampling operation along the spatial dimensions (width, height).

Reduce the amount of parameters and computation in the network, and hence to also control overfitting.

**Convolutional layer computation**

An example of elementwise Conv. computation:

テーブル

自動的に生成された説明

1. Input X of size:

(N, C, H, W)

1. With filters/kernels of size:

(# output channels, # input channels, filter width, filter height)

Where

To simplify, filters W of size:

(K, C, F, F)

1. And with bias b of size:

(K, )

1. With stride S, where

In practice, we let S = 1.

Because we don’t have to increase the stride to improve computational efficiency anymore, as at which lower-level details are obscured.

1. And zero-padding P, defined as

To pad evenly each side and output a matrix of the same size as input,

Thus,

F must be odd

1. And a padded X of size:

(N, C, H+2P, W+2P)

X\_pad = np.pad(X, ((0,0), (0,0), (P, P), (P, P)))

1. Output of size:

(N, K, H, W)

Note

When

recalling we do zero-padding to exactly let the input and output width and height be the same.

Similarly when

For the output of n-th image at the k-th channel, at the 2D position of (h, w):

At S = 1, we have:

If we do the computation elementwise, there would be 5 loops!

()

**Convolutional layer backpropagation**

To get the gradient of Filter (filters) for a smaller loss, and the gradient of input to the Conv. layer.

Let’s have a look at the conclusions.

**For the gradient of filters Filter:**

Thus, when implemented with numpy:

N, K, H, W = dOut.shape

N, C, H\_pad, W\_pad = X\_pad.shape

*# H\_pad = H + 2\*P*

F = H\_pad - H + 1 if S == 1

*# F = H\_pad - H + S + W - S\*W*

dF= Conv(X\_pad.reshape(C, N, H\_pad, W\_pad) , dOut.reshape(K, N, H, W))

dF = dF.reshape(K, C, F, F)

**For the gradient of input to Conv. layer X:**

where

which implemented with numpy at S = 1,

N, K, H, W = dOut.shape

N, C, H\_pad, W\_pad = X\_pad.shape

P = H\_pad - H

F\_fullyPad = np.pad(Filter, ((0, 0), (0, 0), (H-1, H-1), (W-1, W-1)))

F\_fullyPad= F\_fullyPad .reshape(C, K, 2\*H+2\*P-1, 2\*W+2\*P-1)

dOut\_180 = np.rot90(dOut, 2)

dX\_pad = Conv(F\_fullyPad , dOut\_180)

dX\_pad = dX\_pad.reshape(N, C, H+2\*P, W+2\*P)

dX = dX\_pad[:][:][P:-P][P:-P]

Refer to [Convolutions and Backpropagation](https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c) for derivation based on a simple non-padded example.

And by chain rule:

then we have

**Pooling layer computation**

1. Input X of size:

(N, C, W, H)

1. With a filter/kernel of size:

(F, F)

Make sure that

and

1. And stride S.

In practice let S = F.

1. Output of size:

(N, C, W/F, H/F)

P.S.

Two kinds of pooling: max pooling and average pooling.

A simple illustration of MaxPool for 2D images of size (4, 4) with F = S= 2:

テーブル

低い精度で自動的に生成された説明

**Max pooling backpropagation**

Unlike conv. backpropagation, we do not compute the gradient of the filter/kernel here because that stays unchanged!

Instead, **we are interested in the gradient of max pooling input**, which is essential for the backward computation of the former convolutional layer.

Since max pooling is defined as

Which can be reorganized into

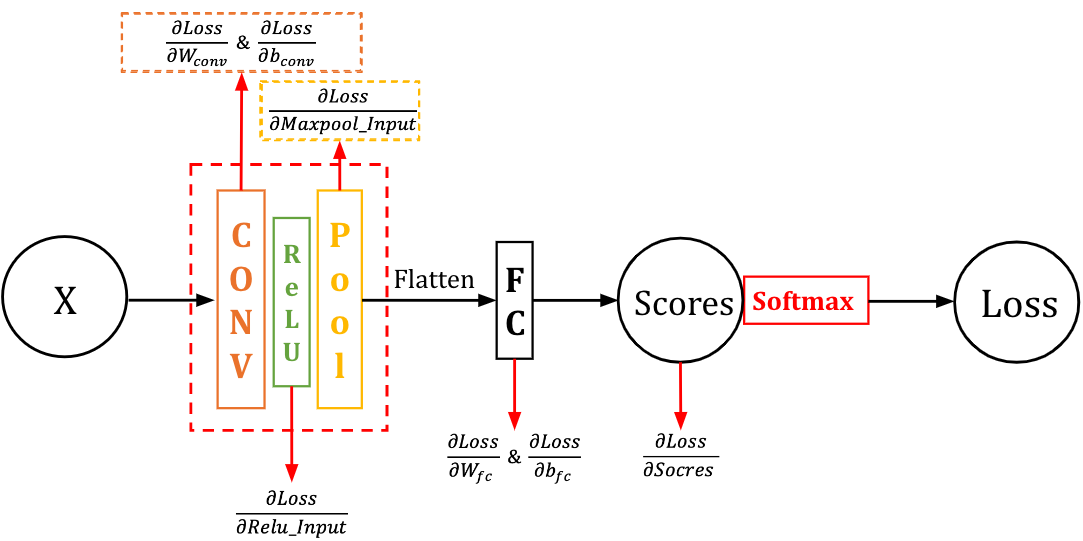
Where

Thus, we have the gradient of xi that

And by chain rule:

Which will be used future backward to compute the gradient of convolutional filters W (or denoted by F):

In conclusion:



Where

And

*P.S.*

*For the ReLU layer in this case,*

*And*

*Thus,*

*Where*

P.S.

To use PyTorch, import the functions.

import torch

import torch.nn as nn

import torch.optim as optiM

import torch.nn.functional as F # useful stateless functions

import numpy as np

USE\_GPU = True

if USE\_GPU and torch.cuda.is\_available():

device = torch.device('cuda')

else:

device = torch.device('cpu')

Note: we need a flatten function below which reshapes image data for use in a fully-connected neural network.

It’s okay when we use barebones of PyTorch or call its Module API.

def flatten(x):

*# read in N, C, H, W*

N = x.shape[0]

*# "flatten" the C \* H \* W values into a single vector per image*

return x.view(N, -1)

But when we’re calling the Pytorch Sequential API, we need to wrap the `flatten` function in a module in order to stack it in nn.Sequential.

class Flatten(nn.Module):

def forward(self, x):

return flatten(x)

def flatten(x):

*# read in N, C, H, W*

N = x.shape[0]

*# "flatten" the C \* H \* W values into a single vector per image*

return x.view(N, -1)

An example of PyTorch codes to implement the CNN architecture above using PyTorch Module API.

In the example, we set the CNN filter (kernel) size to be (5, 5) with stride at 1, and the MaxPool filter size (2, 2) with stride at 2.

Original images are of size (N, 3, 32, 32).

class simple\_CNN\_Module(nn.Module):

def \_\_init\_\_(self, in\_channel, out\_channel, num\_classes):

*# don't forget to call the super().\_\_init\_\_() first!*

super().\_\_init\_\_()

*# Set up the layers needed for a CNN network with the architecture defined above.*

*# CNN*

self.cnn = nn.Conv2d(in\_channel, out\_channel, kernel\_size = (5, 5), stride=1, padding=2)

*# nn.init package contains convenient initialization methods*

nn.init.kaiming\_normal\_(self.conv1.weight)

*# MaxPool*

self.maxpool = nn.MaxPool2d(kernel\_size = (2, 2), stride=2)

*# FC*

self.fc = nn.Linear(out\_channel // 2 \* 32 \* 32, num\_classes)

nn.init.kaiming\_normal\_(self.fc.weight)

def forward(self, x):

x = self.conv1(x)

x = F.relu(x)

x = self.maxpool(x)

*# flatten x to be the shape of (N, Channel\_final\*H\*W), before inputting it into FC*

x = flatten(x)

scores = self.fc(x)

return scores

def flatten(x):

*# read in N, C, H, W*

N = x.shape[0]

*# "flatten" the C \* H \* W values into a single vector per image*

return x.view(N, -1)

An example of codes when we use the Pytorch Sequential API.

class Flatten(nn.Module):

def forward(self, x):

return flatten(x)

def flatten(x):

N = x.shape[0]

return x.view(N, -1)

def simple\_CNN\_Sequential(in\_channel, out\_channel, num\_classes):

model = nn.Sequential(

*# Conv., (N, 3, 32, 32) -> (N, out\_channel, 32, 32)*

nn.Conv2d(in\_channel, out\_channel, kernel\_size = (5,5), stride=1, padding=2),

*# Relu*

nn.ReLU(),

*# MaxPool, (N, out\_channel, 32, 32) -> (out\_channel, 32, 16, 16)*

nn.MaxPool2d(kernel\_size = 2, stride=2),

*# call Flatten, (N, out\_channel, 8, 8) -> (N, out\_channel \* 8 \* 8)*

Flatten(),

*# FC, (N, out\_channel \* 8 \* 8) -> (N, num\_classes)*

nn.Linear(out\_channel \* 8 \* 8, num\_classes),

)

return model

**More Reading**

In many cases, we need to customize a loss function for our needs. Refer to [Implementing Custom Loss Functions in PyTorch](https://towardsdatascience.com/implementing-custom-loss-functions-in-pytorch-50739f9e0ee1).