Convolutional Neural Networks

Introduction

Convolutional Neural Networks (CNNs), build upon the fully connected neural networks you've seen to date. Since detailed images can have incredibly high dimensions based on the number of pixels, CNNs provide an alternative formulation for analyzing groups of pixels. Without the convolutional operation, fitting neural networks to medium to large images would be infeasible for all but the most powerful computers. For example, given a color image with 500 x 500 pixels, you would have 500 x 500 x 3 = 750,000 input features, $(x_1, \ldots, x_{750,000})$. From there, even having 2000 hidden units (3% of the input), in the first hidden layer, would result in roughly 1.5 billion parameters!

Objectives

You will be able to:

- Define what a convolution is, as it relates to CNNs
- Explain how convolutions work using RGB images
- Describe what a pooling layer is in a neural network
- Explain how padding works with convolution layers of a neural network

CNNs

CNNs have certain features that identify patterns in images because of "convolution operation" including:

- Dense layers learn global patterns in their input feature space
- Convolution layers learn local patterns, and this leads to the following interesting features:
    - Unlike with densely connected networks, when a convolutional neural network recognizes a pattern in one region, these insights can be shared and applied to other regions.
    - Deeper convolutional neural networks can learn spatial hierarchies. A first layer will learn small local patterns, a second layer will learn larger patterns using features of the first layer patterns, etc.

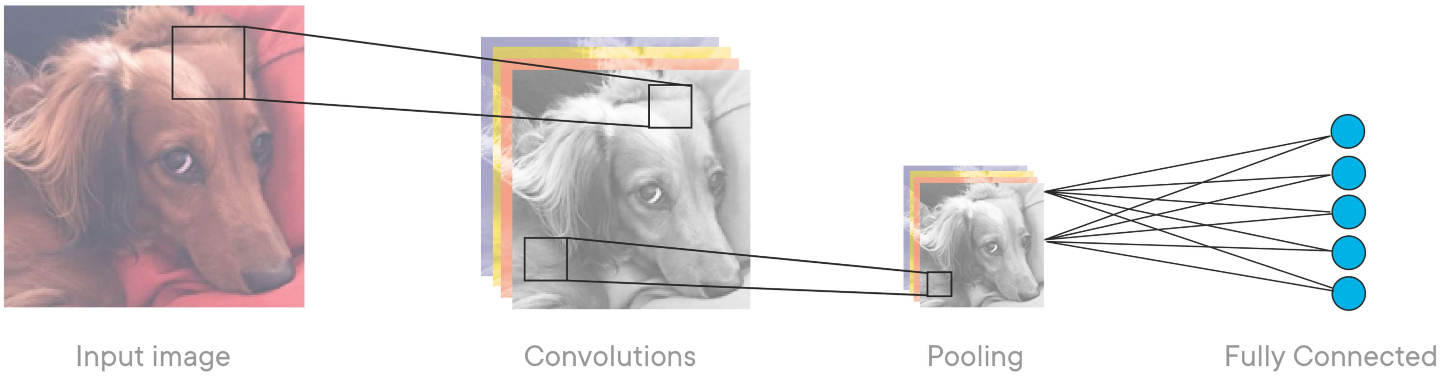
Because of these properties, CNNs are great for tasks like:

- Image classification
- Object detection in images

- Picture neural style transfer

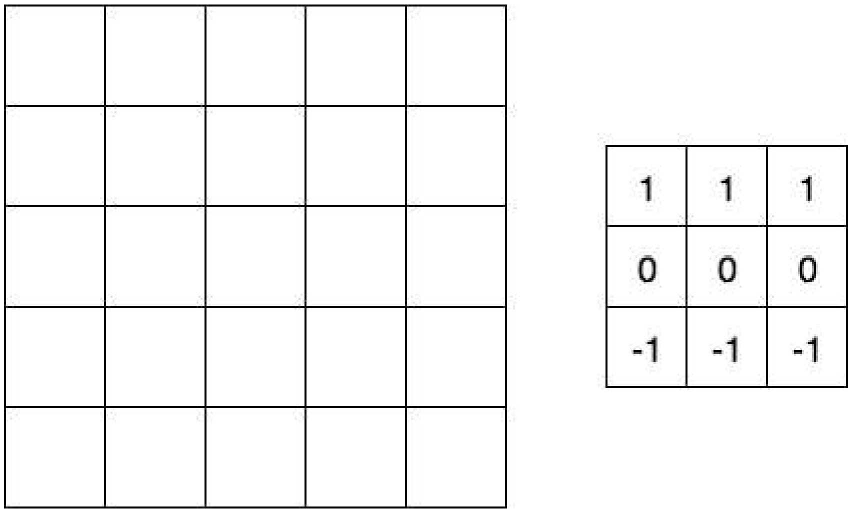
Building CNNs in Keras

Building a CNN in Keras is very similar to the previous neural networks that you've built to date. To start, you will initialize a sequential model as before and go on adding layers. However, rather then simply adding additional dense layers or dropouts between them, we will now start to investigate other potential layer architectures including convolutional layers.



The Convolution Operation

The idea behind the convolutional operation is to detect complex building blocks, or features, that can aid in the larger task such as image recognition. For example, we'll detect vertical or horizontal edges present in the image. Let's look at what horizontal edge detection would look like:



This is a simplified 5 x 5 pixel image (grayscale!). You use a so-called "filter" (denoted on the right) to perform a convolution operation. This particular filter operation will detect horizontal edges. The matrix in the left should have number in it (from 1-255, or let's assume we rescaled it to number 1-10). The output is a 3 x 3 matrix. (*This example is for computational clarity, no clear edges*)

In Keras, function for the convolution step is `Conv2D` .

The convolutional operation applies this filter (typically 3x3 or 5x5) to each possible 3x3 or 5x5 region of the original image. The graphic below demonstrates this process.

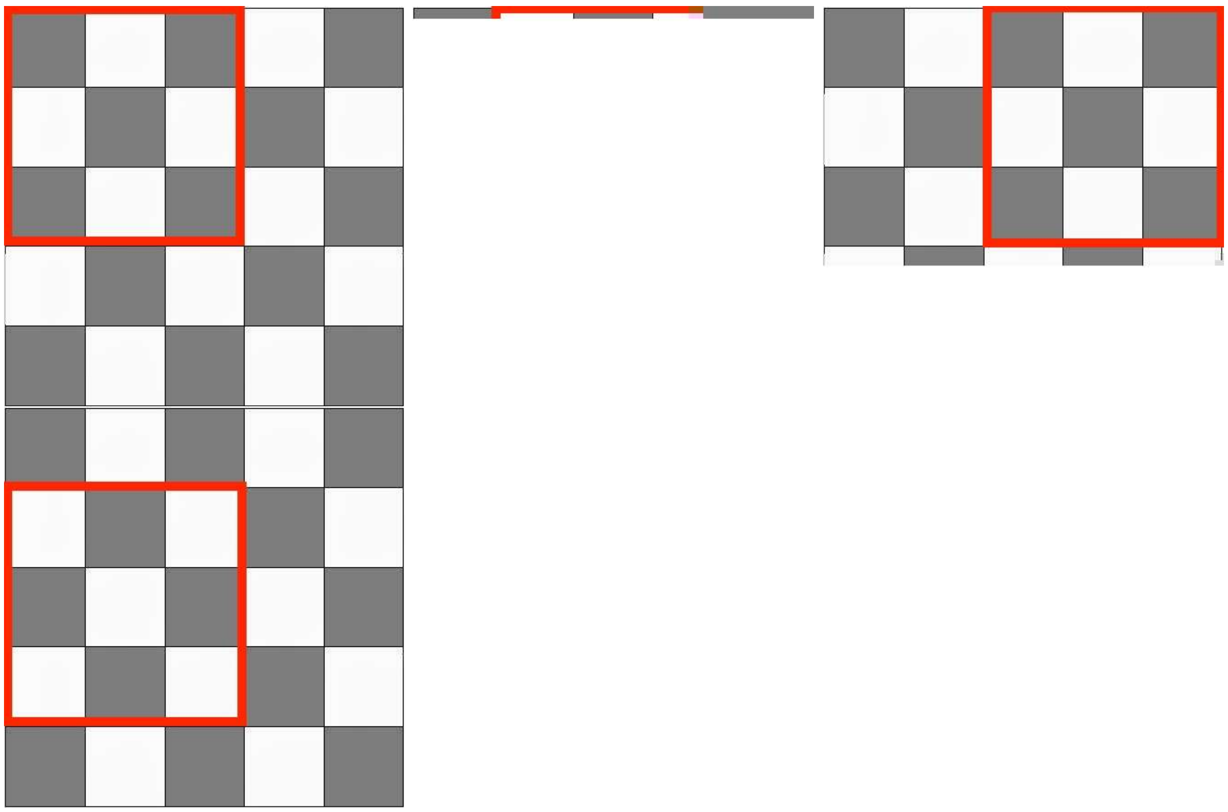
[gif courtesy of Stanford University (https://stanford.edu/%7Eshervine/teaching/cs-230/cheatsheet-convolutional-neural-networks)](https://stanford.edu/%7Eshervine/teaching/cs-230/cheatsheet-convolutional-neural-networks)

Padding

There are some issues with using filters on images including:

- The image shrinks with each convolution layer: you're throwing away information in each layer! For example:
    - Starting from a 5 x 5 matrix, and using a 3 x 3 matrix, you end up with a 3 x 3 image
    - Starting from a 10 x 10 matrix, and using a 3 x 3 matrix, you end up with a 8 x 8 image, etc.
- The pixels around the edges are used much less in the outputs due to the filter

For example, if you apply 3x3 filters to a 5x5 image, the original 5x5 image contains 25 pixels, but tiling the 3x3 filter only has 9 possible locations. Here's the 4 of the 9 possible locations for the 3x3 filter on a 5x5 image:



Fortunately, padding solves both of these problems! Just one layer of pixels around the edges preserves the image size when having a 3 x 3 filter. We can also use bigger filters, but generally the dimensions are odd!

Some further terminology regarding padding that you should be aware of includes:

- "Valid" - no padding
- "Same" - padding such that output is same as the input size

By adding padding to our 5x5 image, (now a 6x6 image by adding a border of pixels) we can add padding so that each pixel of our original 5x5 image can be the center of a 3x3 convolution window filter.

Strided convolutions

Another method to change the output of your convolutions is to change the stride. The stride is how the convolution filter is moved over the original image. In our above example, we moved the filter one pixel to the right starting from the upper left hand corner, and then began to do this again after moving the filter one pixel down. Alternatively, by changing the stride, we could move our filter by 2 pixels each time, resulting in a smaller number of possible locations for the filter.

Strided convolutions are rarely used in practice but a good feature to be aware of for some models.

Convolutions on RGB images

Instead of 5 x 5 grayscale, imagine a 7 x 7 RGB image, which boils down to having a 7 x 7 x 3 tensor. (The image itself is compromised by a 7 by 7 matrix of pixels, each with 3 numerical values for the RGB values.) From there, you will need to use a filter that has the third dimension equal to 3 as well, let's say, 3 x 3 x 3 (a 3D "cube").

This allows you to detect horizontal edges in the blue channel.

Then, in each layer, you can convolve with several 3D filters. Afterwards, you stack every output of the result together, giving you a matrix of shape 5 x 5 x `number_of_filters` .

If you think of it, the filter plays the same role as the w^{[1]} in our densely connected networks.

The advantage is, while your image may be huge, the amount of parameters you have still only depends on how many filters you're using!

Imagine 20 (3 x 3 x 3) --> 20 * 27 + a bias for each filter (1* 20) = 560 parameters.

Notation:

- $f^{[l]}$ = size of the filter
- $p^{[l]}$ = padding
- $s^{[l]}$ = amount of stride
- $n_c^{[l]}$ = number of filters
- filter: $f^{[l]}$ x $f^{[l]}$ x $n_c^{[l-1]}$

- Input = $n_h^{[l-1]} * n_w^{[l-1]} * n_c^{[l-1]}$
- Output = $n_h^{[l]} * n_w^{[l]} * n_c^{[l]}$

Height and width are given by:

$$n_h^{[l]} = \left\lfloor \frac{n_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

$$n_w^{[l]} = \left\lfloor \frac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

Activations: $a^{[l]}$ is of dimension $n_h^{[l]} * n_w^{[l]} * n_c^{[l]}$

Pooling layer

The last element in a CNN architecture (before fully connected layers as we have previously discussed in other neural networks) is the pooling layer. This layer is meant to substantially downsample the previous convolutional layers. The idea behind this is that the previous convolutional layers will find patterns such as edges or other basic shapes present in the pictures. From there, pooling layers such as Max pooling (the most common) will take a summary of the convolutions from a larger section. In practice, Max pooling (taking the max of all convolutions from a larger area of the original image) works better than average pooling as we are typically looking to detect whether a feature is present in that region. Downsampling is essential in order to produce viable execution times in the model training.

Max pooling has some important hyperparameters: - f (filter size) - S (stride)

Common hyperparameters include: `f=2` , `s=2` and `f=3` , `s=2` , this shrinks the size of the representations.

If a feature is detected anywhere in the quadrants, a high number will appear, so max pooling preserves this feature.

Fully Connected Layers in CNN

Once you have added a number of convolutional layers and pooling layers, you then will add fully connected (dense) layers as we did before in previous neural network models. This now allows the network to learn a final decision function based on these transformed informative inputs generating from the convolutional and pooling layers.

Summary

In this lesson, you learned about the basic concepts behind CNNs including their use cases and general architecture. In the upcoming lab, you'll begin to look at how you can build these models in Python using Keras.