PyTorch Models and CNN Theory Part 1

INTRODUCTION

In workshop 2 of this series, we introduced PyTorch, and showed you how to create datasets, data loaders, and bind it all together into a training and validation loop. Now, we want to extend the power of our models by introducing CNNs to recognise patterns in visual data.

Using Pytorch, we can define our own models, which gives us much more flexibility and options. One of the types of models we can create is a Convolutional Neural Network (CNN). CNNs are perhaps the most famous type of neural network, as they form the basis of most image classification models.

The core part of a CNN is a convolutional layer, which extracts features from visual data. The idea is that these convolutional layers are stacked on top of each other, to extract increasingly complex information. Don't worry, we go into this in more detail later.

This workshop is split into two articles. First up we'll discuss the theory behind how convolutional neural networks work and afterwards we'll see how to implement them in PyTorch.

Source: https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/

CNNs are also perhaps one of the more controversial neural networks, as companies and researchers start to create facial recognition and other tools whose implications on free speech and protesting in countries with harsh or authoritarian regimes are still yet to be seen. However, when used for the right reasons, CNN's are an extremely useful tool, and a great entry point into deep learning.

CNN THEORY

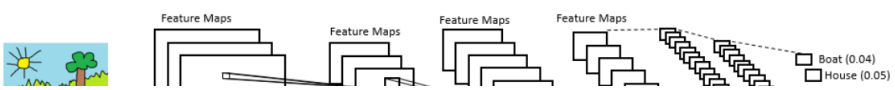
As mentioned above, CNN's are widely used for visual applications such as image recognition or image processing. They do this by helping detect "features" in an image. Features can be simple or complex. Examples of simple features include:

- Edges
- Colours
- Corners

More complex features could include:

- Faces
- Cars
- Birds

CNN's work by extracting low level features first, and using them to extract the intermediate and then high level features. Features are extracted using convolutional layers, which we come to later. After the features are extracted, the model will then classify the image or perform some other function.



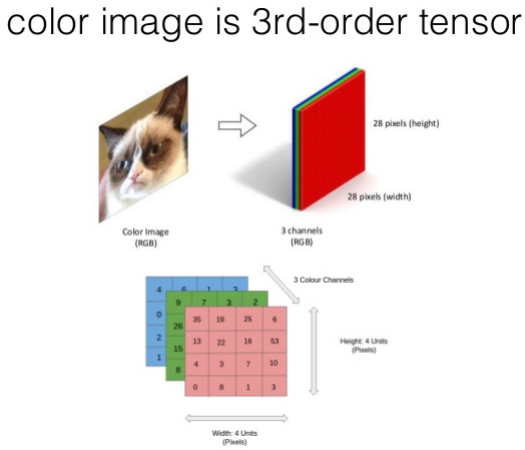
Full CNN overview

Source: https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

Representing Images as Tensors

Unfortunately for deep learning practitioners, neural networks can only take numbers as input, meaning that we need to convert different data types to numbers before being able to use them with a neural network. Before the discussion on the workings of a CNN, it can be useful to have a think about how images are represented as tensors before being passed into a neural network.

A tensor is a mathematical term describing multidimensional algebraic objects. To simplify that frankly terrifying phrase, a single number is a 0-dimensional tensor, a vector is a 1-dimensional tensor, and a matrix is a 2-dimensional tensor. Then, we can think of a 3-dimensional tensor as a "stack" of matrices, much like a pile of pancakes (if matrices were pancakes). A 4-dimensional tensor can be thought of as a line of 3-dimensional tensors, much like piles of pancakes stacked in a row. There is no limit to how high the dimensionality can go. Keeping up?



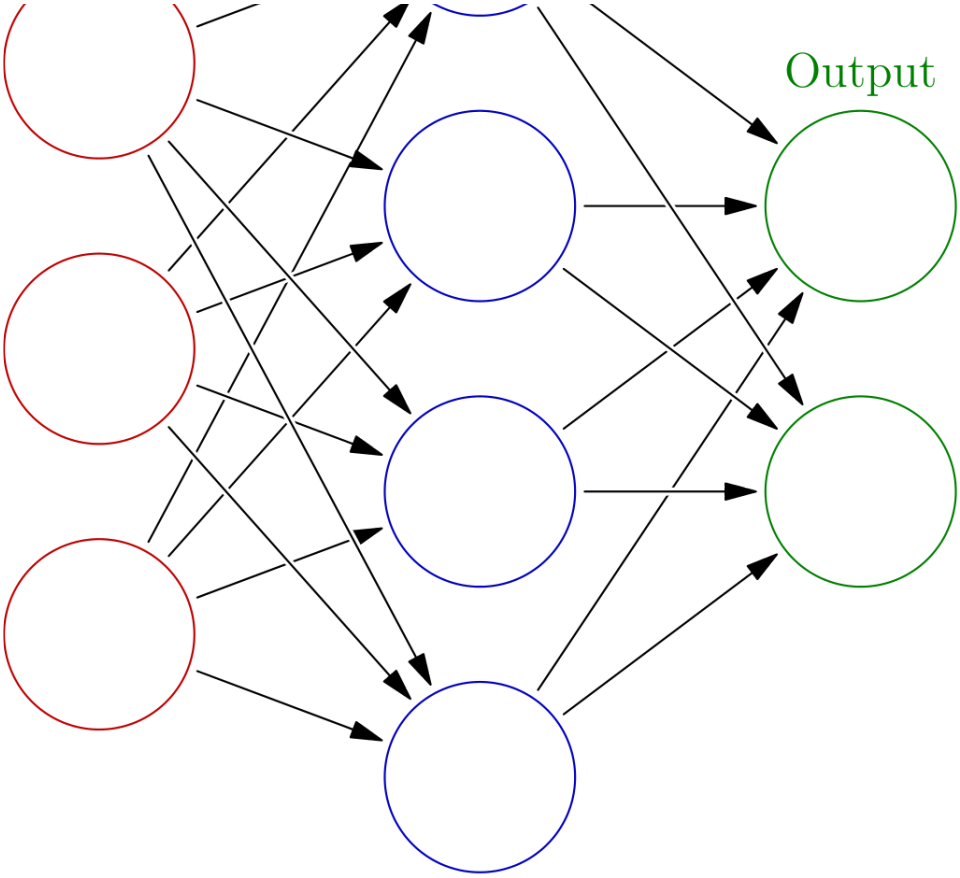
Source: https://lisaong.github.io/mldds-courseware/01_GettingStarted/numpy-tensor-slicing.slides.html

Images stored in a computer are stored as pixels, where each pixel has some information stored with it describing the colour of that pixel. Typically, RGB values are used, meaning the strength of red, green and blue in the pixel are stored. To convert an image to a tensor, we make a matrix of each pixel's "redness", another matrix for all of the "blueness" values, and another matrix for all of the "greenness" values. We refer to each of these matrices as channels. We then stack those three matrices on top of each other to make a 3-dimensional tensor. This 3-dimensional tensor is what we use to describe the image in a way that can be passed into a neural network.

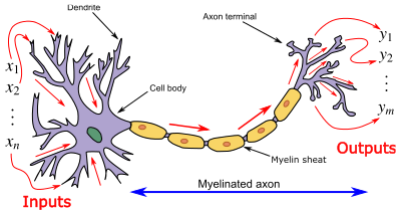
LAYERS

Fully Connected Layers

Also called Dense or Linear layers, the Fully Connected Layer (FCL) is the most basic type of layer, and is used in almost every type of neural network. Each node in the layer is connected to every node in the previous layer, and each connection has a weight and a bias. The fully connected layer is based loosely upon a set of neurons - each node/neuron receives input signals from the previous layer, weights the signals according to importance, and then uses that input to produce an output.



Source: https://en.wikipedia.org/wiki/Artificial_neural_network

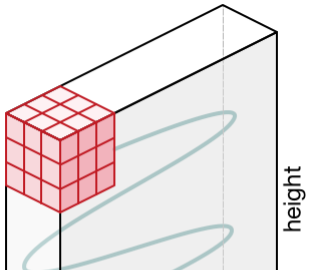


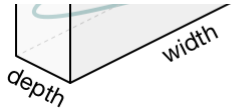
Source: https://en.wikipedia.org/wiki/Artificial_neural_network

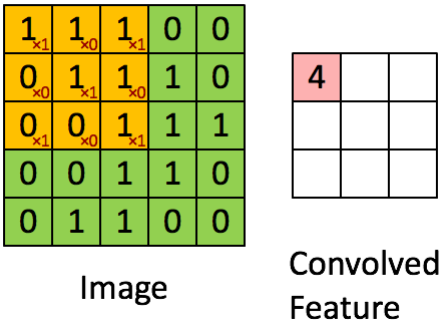
Convolutional Layers

The core layer in a CNN is the convolutional layer. The convolutional layer is the layer which extracts features from an image. They can work in 2D (e.g. for an image), 3D (for a point cloud or other similar data representation), or any other dimensionality. For simplicity, in this article, we'll focus on 2D images.

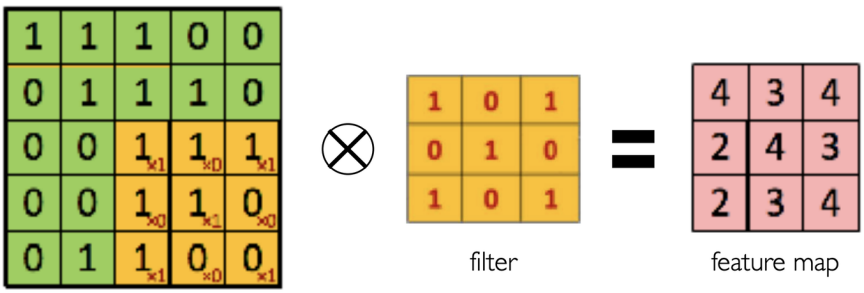
Convolutional layers work by producing feature maps. A single convolutional layer can produce multiple feature maps. For example, a single convolutional layer might produce one feature map describing the edges in an image, another feature map describing the colours, and another describing any dots. Of course, the magic of a convolutional layer is that we don't need to define what each feature map represents, but it can be useful to think of them as beginning with low level features and then moving up to higher level ones. These feature maps are then fed into the next convolutional layer in the CNN, which will use the lower level features to build up a representation of some higher level features.







Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53



Source: © MIT 6.S191: Introduction to Deep Learning

Convolutional layers use kernels (sometimes called filters) to produce feature maps. A filter is a small matrix (much smaller than the size of the image). The filter "convolves" across the input image, and at each location takes the *dot product* of the input matrix it is currently "hovering" over. For example, in the image above, the kernel is hovering over the bottom right corner of the input. The dot product of the highlighted region of the input and the kernel is 4 - hence the 4 in the bottom right of the feature map.

This is actually a bit of a simplification. As a convolutional layer will take multiple channels or feature maps as input, each feature map that the layer produces has a separate kernel for each input channel/feature map. This isn't super important to getting the idea of how a convolutional layer works though.

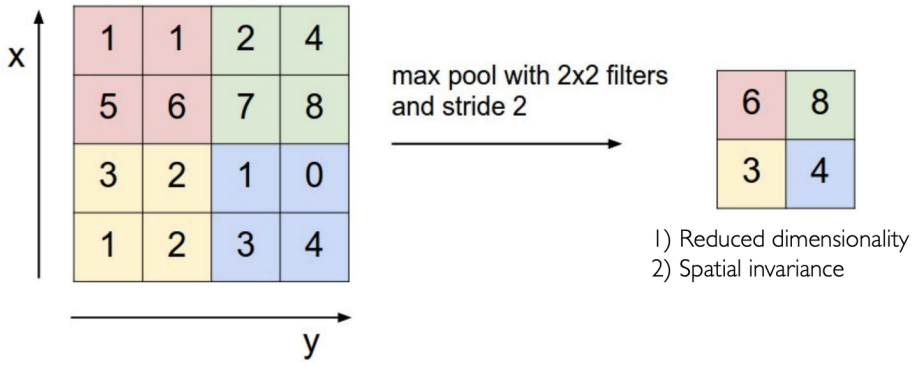
To help visualise this, use this fantastic tool. It shows how a kernel convolves across an input to produce an output. The tool also allows you to adjust some settings that convolutional layers use:

- Kernel Size: the size of the kernel can be adjusted. A larger kernel can be more computationally expensive but might capture features better. Like so many things in deep learning, there's a trade-off of compute efficiency versus model effectiveness
- Stride: it's not necessary to take the dot product at every point on the input. We can increase the stride to skip some of the inputs
- Padding: As can be seen in the image above, generally the outputted feature map is smaller than the input. To preserve the size of the output with respect to the input, we can pad the input with some zeros around its edge
- Dilation (uncommon): It's also possible to have a kernel which has some blank spaces (see the linked tool for a good visualisation). Generally dilation is used for 3D inputs to lower compute time.

Source: © MIT 6.S191: Introduction to Deep Learning

Max Pooling

A Max Pooling layer is used to downsample the data but still keep important information.



These layers are typically used in CNNs when we want to reduce the amount of data that needs to be processed, without losing too much of the important parts of the data. You may be wondering why max pooling is used and not something like an average pooling? Average pooling has its applications, but for simple downsampling, it has been found that max pooling is good enough at retaining important information, and is very cheap computationally.

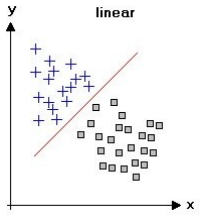
Batch Normalisation

A batch normalisation layer normalises data (duh) - means we standardise the means and variances of the inputs to a layer. This increases the stability and speed of training and stops exploding gradient problems.

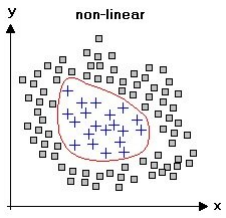
There are numerous theories for exactly *why* batch normalisation is effective, generally agreed to be either a smoothing of the objective landscape or mitigating internal covariate shift.

Activation Functions

Most real world data can't be modelled by a linear function, so we need to introduce non-linearities.



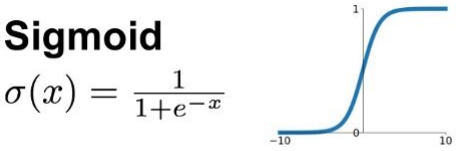
Source:
http://www.statistics4u.com/fundstat_eng/cc_linvsnonlin.html



Source:
http://www.statistics4u.com/fundstat_eng/cc_linvsnonlin.html

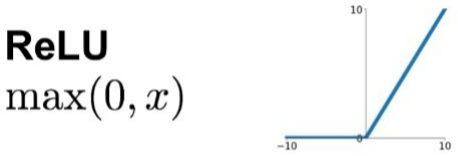
A non-linearity function must be applied after each layer. This is because all the layers we commonly work with are

The oldest and most well known non-linearity is the Sigmoid function.



Source: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

However, the much more common layer used today is the ReLU function:



Source: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

$$ReLU(x) = \begin{cases} 0, & x < 0 \\ x, & otherwise \end{cases}$$

There are a few reasons why ReLU is often preferred over Sigmoid. ReLU is much cheaper than Sigmoid (one if statement compared to multiple maths operations). If ignoring computation speed - sigmoid stops forward pass exploding, relu stops backwards pass disappearing. This means that sigmoid can be more stable, but in general ReLU is better because of some nice properties when back-propagating loss, meaning training is faster.

THANKS FOR READING

You've made it to the end of the third Introduction to deep learning article. So far we've introduced the background theory behind CNN's by talking about the layers involved in their use. At this point it may seem quite abstract and mathsy, however next week we'll dig into how we actually put all this information to practice! If you still feel a little shaky don't worry it's natural (you've gone through a lot of information by now, congratulations for making it half way through!).

Written by William Maclean (Deep Learning Lead)

Edited by Kamrom Bhavnagri (Deep Learning Training Manager)

William Maclean (left) and Kamrom Bhavnagri (right)

FACEBOOK TWITTER REDDIT 0 LIKES

Comments (0)

Newest First

Preview POST COMMENT...

PREVIOUS
PyTorch Models and CNN Theory Part 2

NEXT
Introduction to Pytorch

ATTRIBUTIONS

Main banner gif: http://maximschoemaker.com/