LIGHT ON MATH MACHINE LEARNING

Intuitive Guide to Convolution Neural Networks

Thushan Ganegedara May 31, 2018 · 10 min read

Image by Free-Photos from Pixabay

This is the second article on my series introducing machine learning concepts with while stepping very lightly on mathematics. If you missed previous article you can find in here (on KL divergence). *Fun fact*, I'm going to make this an interesting adventure by introducing some machine learning concept for every letter in the alphabet (This would be for the letter **C**).

A B C D* E F G* H I J K L* M N O P Q R S T U V W X Y Z

* denotes articles behind the Medium paywall

Introduction

Convolution neural networks (CNNs) are a family of deep networks that can exploit the spatial structure of data (e.g. images) to learn about the data, so that the algorithm can output something useful. Think of a problem where we want to identify if there is a person in a given image. For example, if I give the the CNN an image of a person, this deep neural network first needs to learn some local features (e.g. eyes, nose, mouth, etc.). These local features are learnt in ***convolution layers***.

Then the CNN will look at what local features are present in a given image and then produce specific activation patterns (or an activation vector) which globally represents the existence of those local features maps. These activation patterns are produced by ***fully connected*** layers in the CNN. For example, if the image is a non-person, the activation pattern will be different from what it gives for an image of a person.

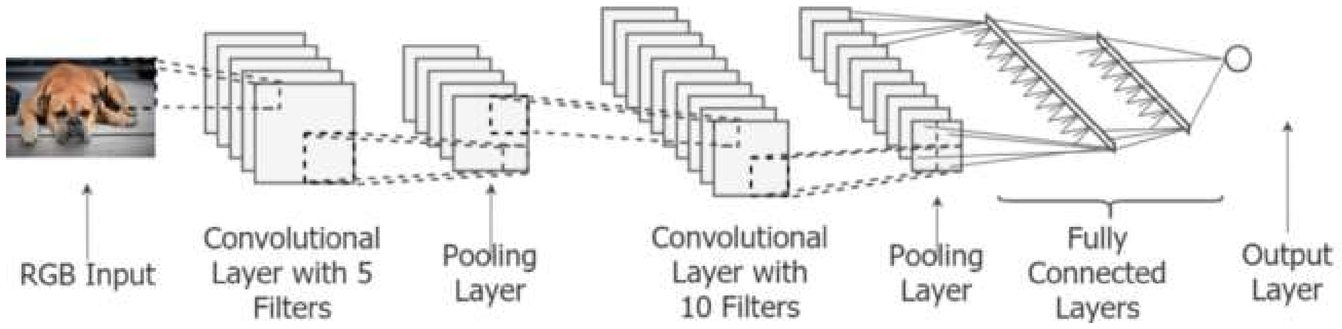
CNN at a Modular Level

Now let's look at what sort of sub modules are present in a CNN. There are three different components in a typical CNN. They are, convolution layers, pooling layers and fully-connected layers. We already have a general idea about what a convolution layer and a fully connected layer is. One thing we did not discuss is the ***pooling layer***, which we will discuss soon.

First we discuss what a convolution layer does in depth. A convolution layer consists of many ***kernels***. These kernels (sometimes called ***convolution filters***) present in the convolution layer, learn local features present in an image (e.g. how the eye of a person looks like). Such a local feature that a convolution layer learns is called a ***feature map***. Then these features are convolved over the image. This convolution operation will result in a matrix (that is sometimes called an ***activation map***). The activation map produces a high value at a given location, if the feature represented in the convolution filter is present at that location of the input.

The pooling layer make these features learnt by the CNN translation invariant (e.g. no matter the person's eye is at [*x=10, y=10*] or [*x=12,y=11*] positions, the output of the pooling layer will be same). Note that we talk about slight translation variations per layer. However aggregating several such layers, allows us to have higher translation invariance.

Finally we have the fully connected layer. Fully connected layers are responsible for producing different activation patterns based on the set of activated feature maps and the locations in the image, the feature maps are activated for. This is what CNN looks like visually.


RGB Input
Convolutional Layer with 5 Filters
Pooling Layer
Convolutional Layer with 10 Filters
Pooling Layer
Fully Connected Layers
Output Layer

With a good understanding about what the overall structure of a CNN looks like, let us move on to understanding each of these sub components, that make up a CNN.

Convolution Layer

What does the convolution operation exactly do? Convolution operation outputs a high value for a given position if the convolution feature is present in that location, else outputs a low value. More concretely, at a given position of the convolution kernel, we take the element-wise multiplication of each kernel cell value and the corresponding image pixel value that overlaps the kernel cell, and then take the sum of that. The exact value is decided according to the following formula (*m* — kernel width and height, *h* — convolution output, *x* — input, *w* — convolution kernel).

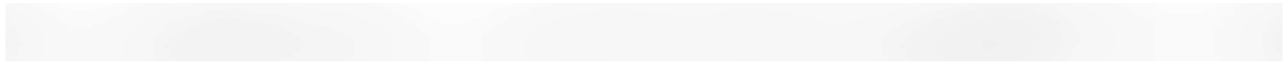
The convolution process on an image, can be visualised as follows.

It is not enough to know what the convolution operation does, we also need to understand what the convolution output represents. Just imagine colours for the values in the convolution output (0 — black, 100 — white). If you visualise this image, it will represent a binary image that lights up at the location the eyes are at.

The convolution operation also can be thought as performing some transformation on a given image. This transformation can result in various effects (e.g. extracting edges, blurring, etc.). Let us more concretely understand what the convolution operation does to an image. Consider the following image and the convolution kernel. You can find more about this in this Wikipedia article.

Pooling Layer

Let us now learn what the pooling operation does. Pooling (or sometimes called subsampling) layer make the CNN a little bit translation invariant in terms of the convolution output. There are two different pooling mechanisms used in practice (max-pooling and average-pooling). We will refer to max-pooling as pooling as, max-pooling is widely used compared to average pooling. More precisely, the pooling operation, at a given position, outputs the maximum value of the input, that falls within the kernel. So mathematically,



Let us understand how pooling works, by applying the pooling operation on the convolution output we saw earlier.

As you can see, we use two variants of the same image; one original image and another image translated slightly on the x-axis. However, the pooling operation outputs the exact same feature map for both images (black — 0, white — 100). Therefore we say the pooling operation make the knowledge in the CNN translation invariant. One thing to note is that we are not moving 1 pixel at a time, but 2 pixels at a time. This is known as the ***strided-pooling***, meaning that we are performing pooling with a stride of 2.

Fully Connected Layers

Fully connected layers will combine features learnt by different convolution kernels so that the network can build a global representation about the holistic image. We can understand the fully connected layer as below.

The neurons in the fully connected layer will get activated based on whether various entities represented by convolution features is actually present in the inputs. As the fully connected neurons get activated for this, it will produced different activation patterns based on what features are present in the input images. This provides a compact representation of what exists in the image, to the output layer, that the output layer can easily use to correctly classify the image.

Weaving Them Together

Now all we have to do is put all these together, to form an end-to-end model, from raw images, to decisions. And once connected the CNN will look like this. To summarise, the convolution layers will learn various local features in the data (e.g. what an eye looks like), then the pooling layer will make the CNN invariant to translations of these features (e.g. *if the eye appear slightly translated in two images, the CNN will still recognise it as an eye*). Finally we have fully connected layers, that says, “we found two eyes, a nose and a mouth, so this must be a person, and activate the correct output.

What does adding more and more layers does?

Adding more layers, obviously boosts up the performance of deep neural networks. In fact, the most of notable ground breaking research in deep learning had to do with solving the problem of ***how do we add more layers?***, while not disrupting the training of the model. Because deeper the model is, the more difficult it is to train.

But having more layers helps the CNN to learn features in a hierarchical manner. For example the first layer learns various edge orientations in the image, the second layer learns basic shapes (circles, triangles, etc.) and the third layer learns more advance shapes (e.g shape of an eye, shape of a nose), and so on. This delivers better performance, compared to what you would learn with a CNN that has to learn all these with a single layer.

Training the CNNs (aka Backpropagation)

Now, one thing to keep in mind is that, these convolution features (eyes, nose, mouth) don’t magically appear when you implement a CNN. The objective is to learn these features given data. To do this, we define a cost function, that rewards the correctly identified data and penalise misclassified data. Example cost function would be the root mean squared error or the binary cross entropy loss.

After we define the loss, we can optimise the weights of the features (that is each cell value of the features) to reflect useful features that lead the CNN to correctly identify a person. More concretely, we optimise each convolution kernel and fully-connected neurons, by taking a small step in the opposite direction shown by the gradient of each parameter with respect to the loss. However, to implement a CNN you don’t need to know the exact details of how to implement gradient propagation. This is because, most of deep learning libraries (e.g. TensorFlow, PyTorch) implement these differentiation operations internally, when you define the forward computations, automatically.

Implementing and Running a CNN with Keras

Here we will briefly discuss how to implement a CNN. Knowing the basics is not enough, we also should understand how to implement a model using a standard deep learning library like Keras. Keras is a wonderful tool, especially to quickly prototype models, to see them in action! The exercise is available **here**.

First we define the Keras API we want to use. We will go with the sequential API. You can read more about it here:

```
# Define a sequential model
model = Sequential()
```

Then we define a convolution layer as follows:

```
# Added a convolution layer
model.add(Conv2D(32, (3,3), activation='relu', input_shape=[28, 28,
  1]))
```

Here, `32` is the number of kernels in the layer, `(3,3)` is the kernel size (height and width) of the convolution layer. We use the non-linear activation `Relu` and the input shape `[28, 28, 1]` which is `[image height, image width, color channels]`. Note that the input shape should be the shape of the output produced by the previous layer. So for the first convolution layer we have the actual data input. For the rest of layer, it will be the output produced by previous layer. Next we discuss how to implement a max pooling layer:

```
# Add a max pool lyer
model.add(MaxPool2D())
```

Here we don’t provide any parameters as we’re going to use default values provided in Keras. If you do not specify the argument, Keras will use a kernel size of (2,2) and a stride of (2,2). Next we define the fully-connected layers. However before that we need to flatten our output, as the fully connected layers process 1D data:

```
model.add(Flatten())

model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Here we define two fully-connected or dense layers. The first fully connected layer has `256` neurons and uses `Relu` activation. Finally we define a dense layer with 10 output nodes with `softmax` activation. This acts as the output layer, that will activate a particular neuron for images having the same object. Finally we compile our model with,

```
model.compile(
 optimizer='adam', loss='categorical_crossentropy', metrics=
['accuracy']
)
```

Here we say use the `Adam` optimiser (to train the model), with the cross entropy loss and use the `accuracy` of the model to evaluate the model. Finally we can train and test our model using data. We are going to use MNIST dataset, which we will download and read into the memory using `maybe_download` and `read_mnist` functions defined in the exercise. The

MNIST dataset contains images of hand written digits (0–9) and the objective is to classify the images correctly by assigning the digit, that the image represents.

Next we train our model by calling the following function:

```
model.fit(x_train, y_train, batch_size = batch_size)
```

And we can test our model with some test data as below:

```
test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)
```

We will run this for several epochs, and this will allow you to increase your models performance.

Conclusion

We wrap our discussion about the convolution neural network here. We first discussed what takes place within a CNN from a higher vantage point and kept in closing in section by section. Then we discussed the major components within a typical CNN such as, convolution layers, pooling layers and fully connected layers. Finally we walked through each of these components in much more detail. Then we discussed how the training happens in a CNN very briefly. Finally we discussed you we can implement a standard CNN with Keras: a high-level TensorFlow library. You can find the exercise for this tutorial **here**.

Cheers!

Want to get better at deep networks and TensorFlow?

Checkout my work on the subject.

[1] (Book) TensorFlow 2 in Action — Manning

[2] (Video Course) Machine Translation in Python — DataCamp

[3] (Book) Natural Language processing in TensorFlow 1 — Packt

Keras Computer Vision Convolutional Network Artificial Intelligence Light On Math