Some notes on CNN

1 Introduction

A collection of important things about convolutional neural networks (CNN). As was the case with the 'lectures' I gave, these will probably be extremely confusing. So remember: Google is your friend (at least the first page is. If you have to go to page 2, you probably made some bad life-choices).

2 CNN

- For the classification of images, CNN in general work better than DNN. The reason for this is that, if we want to feed an image of 64x64 pixels with 3 color channels to a DNN, we would have to transform the 64x64x3 tensor to a 12.228x1 Numpy array. Not that this wouldn't work, but it's better to keep the spatial structure of the image intact. The more detailed the data, the better the network can perform. Another advantage of CNN over DNN for image classification is that CNN are invariant under translations. This means that, if the CNN is successful in classifying cats, it doesn't matter where in the image the cat is.
- Probably the most important object of a CNN is the convolutional filter. A convolutional filter is just a matrix of which the entries constitute the weights of the network. This filter is then convoluted with the image. This is a 4-step process:
  a) Put the filter on the top left corner of the image
  b) Calculate the element-wise product between the elements of the filter and the corresponding elements of the image
  c) Sum the obtained products. This sum represents the output for the particular pixel we were considering
  d) Move the filter with some particular step-size (= stride) and repeat. In practice one usually uses a stride of 1.

  For example, imagine that the filter is

$$\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} \tag{1}$$

and the image was represented by (the numbers represent the intensity of the color channel, eg for grayscale lower values could mean more black while larger values could mean more gray)

$$\begin{pmatrix} 5 & 2 & 2 & 3 \\ 3 & 1 & 2 & 1 \\ 9 & 0 & 4 & 5 \end{pmatrix}. \tag{2}$$

Putting the filter in the top-left corner then yields as output 6. Moving the filter one place to the right gives as output 5 etc. It is possible that not all pixels are accessible to the filter near the edge. To solve this, we use padding; this is just adding extra columns to the image. Often one uses zero-padding (just adding zero-columns) or same-padding (adding the last column of the image).

- In a CNN, you of course don't just have one filter: you have a stack of them. Each filter then encodes a specific aspect of the input data. For example, if you want to classify cats, 1 filter could encode the presence of a tail; another the presence of paws etc.

- Knowing this, a convolutional layer is then just a stack of filters that turn the input image into a different output image. The number of weights in such a layer is the size of the filter times the number of filters. So, a layer using 4 filters of the same form as the one in (1) would have 16 weights. In practice, often 3x3, 5x5 and 7x7 filters are used.

- Neighboring pixels in an image usually contain similar information, which in turn means similar outputs after convolution. So, it will be redundant to keep all this information. The way to avoid this is called pooling. Two popular methods are max-pooling (if we have an AxB filter, we go over the image and replace each AxB patch by the highest value in that patch) and average-pooling (replace each patch by its average). It is this pooling that ensures the translational invariance of CNN. Usually, the size of the patch you consider for the pooling operation is the same as the size of the filter used, or smaller. It has been found that maxpooling usually performs better for image classification than average-pooling.

- Note that the pooling operation down-samples the input image. For example, maxpooling will turn a 64x64 image into a 32x32 image. Of course, in the backwards step, we would go from a 32x32 to a 64x64 image.

- Since pooling is best applied on the output of a convolutional layer, a typical block in a CNN should consist of a convolutional layer followed by the activation function followed by a pooling layer. This pooling layer has no weights associated to it, so can't be learned.

- In general it is a good idea to normalize the inputs given to a layer. This is called batch-normalization (BN). The point is to make sure that the

inputs are of the same order of magnitude, which in turn ensures that the weights all have an approximately equal contribution to the gradient of the loss function. Note that BN can also be useful for other types of NN beside CNN.

- In a general NN, the BN-operation is usually implemented as an extra layer before the activation (so eg for a CNN, you would have convolution $\rightarrow$ BN $\rightarrow$ activation $\rightarrow$ pooling), and corresponds to a standard-normal transformation. This means that the output X of some layer is transformed to
$$\hat{X} = \frac{X - \mu}{\sigma + \epsilon} \tag{3}$$
with μ the mean of the batch under consideration and σ the variance. We add a small positive constant ϵ to the variance to avoid division by zero. Usually one also introduces shift and scale parameters, using $\gamma\hat{X} + \beta$ instead of $\hat{X}$. These parameters are learnable. Note that for a CNN, you would have to take into account all values over all spatial locations when calculating the mean and variance. Also note that having a normalised input does not guarantee a normalised output.

- In a CNN, after going through the convolutional blocks, the final output tensor should be flattened and given to a densely connected subnetwork to make a final prediction. Here, you're no longer interested in the spatial structure of the data, and you just transform the tensor to a 1-D array (you can just use the Numpy method *flatten()* for this).

- If you want to test your CNN on a different dataset than the physics-related one, you can do this on the MNIST-dataset, which is basically the 'hello world' for CNN. You can find how to use this dataset here: `https://keras.io/datasets/#mnist-database-of-handwritten-digits`.

- Note that the backpropagation algorithm will get some extra contributions compared to the DNN. First backpropagation will hit is the densely connected subnetwork. For backpropagation on the maxpooling layer, think about how the loss would change if we slightly perturb a non-maximal pixel; and how it would change if we slightly perturb a maximal pixel. The Numpy method *amax* might be helpful here. For backpropagation on the convolutional layer, it might be helpful to first think about how the output of 1 pixel would change if we modify a filter. You can do this eg by first considering a filter completely filled with zeroes; and then replacing one of the zeroes by 1.

- **For the CNN you will implement**: I definitely want to see the max-pooling layer implemented. The batch-normalization layer however is optional. But, as said before, it might help increase the performance of your network.