Deep Learning with Python Ch. 8 Computer Vision

Deep Learning for Image Processing

- First big success area for deep learning. Started winning competitions about 10 years ago.
- We can see the improvement on the MNIST problem.

Key is Convolutional Networks

- Convolutional networks (convnets) are what makes image processing work so well.
- Look at the MNIST example. . .
- Notice the mixing of conv2D layers with MaxPooling layers. At the end we see flatten, followed by one dense classifier layer of the form we are familiar with. Soon we will explain what these layers are doing.
- Notice at the end the accuracy on the test data is 99.1 percent, which is a significant improvement over what we saw before.

Convolution

- The big difference between a densely connected layer and a convolution layer is that the Dense layers learn global patterns in the input feature space, whereas the convolution layers learn local patterns in small 2D windows of the inputs. (See figure 8.1)
- Key characteristics:
  - The patterns they learn are translation-invariant. When a pattern is learned in the lower right corner, it can later be recognized in the upper left corner.
  - They can learn spatial hierarchies of patterns. First they can learn small local patterns, such as edges, then another layer can learn largers patterns made from features of the previous layer.
  - See cat example.

Feature maps

- Convolutions operate over rank-3 tensors called **feature maps** with two spatial axes (width and height) and a depth axis. For an RGB image the depth access can have dimension 3, representing the three color channels. For the black-and-white MNIST images the depth is 1 (gray scale).
- Figure 8.3 shows an example of what a filter might recognize.
- Convolutions are defined by the size of the patches extracted from the input (typically 3x3 or 5x5) and by the depth of the output feature map, which is the number of filters computed by the convolution. Our MNIST example started with a depth of 32 and ended with a depth of 64.

more Feature maps

- A convolution works by sliding these windows of size 3x3 (say) over the 3D input feature map, stopping at every possible location and extracting the 3D patch of surrounding features. Each such 3D patch is then transformed into a 1D vector of shape (output depth) which is done via a tensor product with a learned weight matrix (called the convolution kernel). This same kernel is used across every patch. All of these vectors are then reassembled into a 3D output map of shape (height, width, output depth). Positions on the map correspond to positions in the input.
- Figure 8.4 shows a picture.

Border effects

- If you think about sliding a 3x3 patch over, say, a 5x5 feature map, there are only 9 tiles around which you can center a 3x3 window. So your output feature map shrinks by exactly two tiles along each dimension. This happened in our MNIST example when the 28x28 input dropped to 26x26 after the first convolution layer.
- If you want to avoid this, you can use padding. The book has pictures if you are interested.

Strides

- Another factor that can influence output size are **strides**. So far our stride has just been 1. The centers of convolution windows have been assumed to be contiguous. But they don't have to be. Figure 8.7 shows what a stride of 2 looks like. Note that this has the effect of downsampling the data by a factor of 2 (apart from border effects).

Max-pooling operation

- The purpose of the max-pooling layers are to aggressively downsample features maps (much like strides do). Max-pooling consists of extracting windows from the input feature maps and outputting the max value of each channel. This is similar to convolution, except instead of transforming local patches using a learned linear transformation, they're transformed via a hardcoded max tensor operation. Max-pooling is usually done with a 2x2 window and a stride of 2, which downsamples the feature map by a factor of 2.

Why use max-pooling?

- Leaving out the max-pooling layers results in a network that isn't conducive to learning a spatial hierarchy of features. The deeper layers will still be working with the same small windows of the initial input. Imagine trying to recognize a digit with only a small window of it visible at any one time.
- The other big reason is that max-pooling reduces the size of the final feature map. Otherwise it would be huge, with many, many weights, which will almost definitely result in overfitting.

Summary so far

- At this point we have covered the basics of convnets: feature maps, convolution, and max pooling. We looked at how to build a small convnet to classify MNIST digits.

Training a convnet on a small dataset

- This is a great example, involving dogs and cats, just doing a binary classification, identifying each picture as containing a dog or a cat.
- Unfortunately, despite ridiculous amounts of effort, I was unable to get this data to load from Kaggle. As near as I can tell, at least part of the issue is that I'm using a Windows machine.
- So we will look at the results in the book but not be able to run them ourselves. I hope you have better luck if you try to use Kaggle data in your projects.

Building the model

- Look at cat and dog pictures. Aw. Cute.
- The model is quite similar to the one we looked at for MNIST, except bigger, because the images are bigger and they are in color.
- There is a bunch of data pre-processing necessary, including decoding the JPGs into RGB grids of pixels, converting the pixel data into floating-point (0 to 1) tensors, resizing them to be 180x180, and packing them into batches.
- The book takes you through the details.
- For this problem, the result is batches of 32 180x180x3 data.

Initial results

- Figure 8.9 shows the training and validation metrics for the initial configuration.
- They show significant overfitting pretty quickly and about 70 percent accuracy at best. The test data gets an accuracy of 69.5 percent when stopped. Note that, for a binary problem like this, the base expectation is 50 percent. So this is better than nothing.

Data augmentation

- Data augmentation is the approach of generating more training data from existing training samples by augmenting the samples via a number of random transformations that yield believable-looking images. So your model will see more plausible samples without just being exposed to the same sample repeatedly.
- In Keras this can be done by adding data augmentation layers at the start of the model as in Listing 8.14. These include:
  - Random flip (horizontal)
  - Random rotation
  - Random zoom
- Figure 8.10 shows examples

One more thing

- One last thing about random image augmentation is that, just like Dropout, they are inactive during actual inference (meaning prediction or testing on the actual test data).

New model

- The next model includes image augmentation and dropout. We expect overfitting to start later, so we will go for more epochs.
- Figure 8.11 shows the results. Now the validation accuracy is up to the 80-85 percent range. The test accuracy is 83.5 percent, quite an improvement.
- Additional improvements might be possible by adjusting the number of filters per convolution layer or the number of layers in the model, but it would be difficult to get much above 90 percent.

Leveraging a pretrained model

- One really interesting idea that has become a big deal in the deep learning world is using a pretrained model. A pretrained model is a model that was previously trained on a large dataset, typically a large-scale image classification task.
- If this original dataset is large enough and general enough, the spatial hierarchy of features learned by the pretrained model can effectively act as a generic model of the visual world.
- So the same feature detectors can be useful for many different computer vision problems, even those involving completely different classes than the original task.

ImageNet

- One huge dataset that some of these pretrained models are pretrained on is ImageNet, which is a dataset containing animals and everyday objects. Models pretrained on this data can be useful for lots of new problems.
- This portability of learned features across different problems is a key advantage of deep learning compared to many older, shallow learning approaches.
- It is especially helpful for small-data problems like the one we are looking at.

Pretrained models

- The book example uses a model called VGG16, but it mentions a bunch of other models. VGG16 dates from 2014, so it is old and out of date, but apparently useful for illustrative purposes.
- There are two ways (at least) to use pretrained models: feature extraction and fine-tuning.

Feature extraction

- Feature extraction consists of using the representations learned by a previously trained model to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.
- So we take the convolutional base (the conv and pooling layers) from the pretrained model, run the data through it (without updating the weights), then train a new classifier with the results.
- Could we reuse the classifier? No, not very well. Generally what is learned by the convolutional base is generic, but the classifier learns things specific to its training dataset.

more details

- Note that early layers in the convolutional base are likely to be learning truly generic features, such as visual edges, colors and textures, while those coming later are more likely to learn more abstract concepts such as “cat ear” or “dog eye”. So if your new dataset differs a lot from the original, then you may only want the first few layers of the convolutional base.
- For the particular example we are looking at, ImageNet contains many cat and dog pictures (along with lots of other things) so VGG16 will probably be helpful, using all of its layers.

even more details

- The book goes through two different ways to do this experiment. One involves running the VGG16 pretrained system once on each image, saving the data in Numpy arrays, then using those to input to the Dense classifier level in our model. But this precludes doing data augmentation.
- The alternative is to hook everything together, which is more expensive, but more flexible and allows data augmentation.
- Figure 8.13 shows the results for the first approach. It achieves about 97 percent accuracy, an improvement.
- The second approach involves **freezing** the weights in the pretrained model, so that they don't get changed as the Dense models (on both ends, for augmentation and for classification) are learning. Figure 8.14 shows the results for this approach, which achieve over 98 percent accuracy.

Fine-tuning

- Fine-tuning involves retraining some of the pretrained layers to let them adapt to the potentially different abstractions in the new dataset. It involves unfreezing a few of the frozen layers from the pretrained model after everything else is trained. It uses a very low learning rate to very slightly modify these weights. For the author, this results in 98.5 percent accuracy, which would have been near the top of the original competition.
- But the pretrained systems weren't available to the original competitors, so that isn't quite a fair comparison. But this system learned with 2000 training examples, instead of the 20,000 available to the original competitors. The big idea here is that pretrained models allow us to build new systems with relatively small amounts of data.