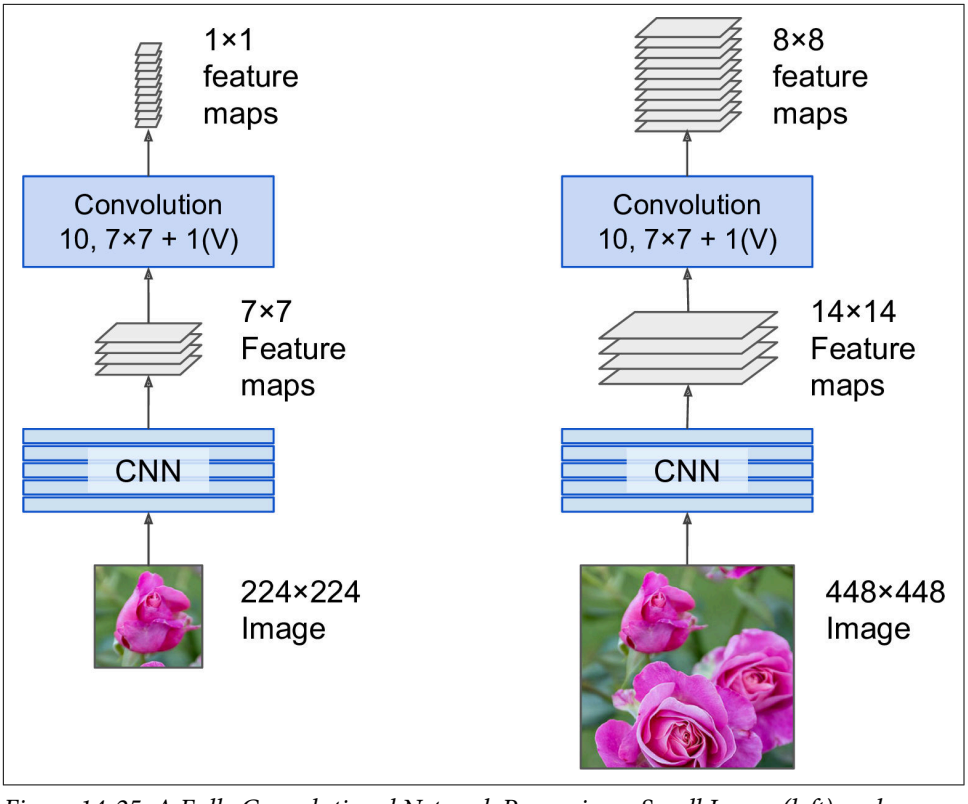

Figure 14-25. A Fully Convolutional Network Processing a Small Image (left) and a Large One (right)

You Only Look Once (YOLO)

YOLO is an extremely fast and accurate object detection architecture proposed by Joseph Redmon et al. in a 2015 paper[26], and subsequently improved in 2016[27] (YOLOv2) and in 2018[28] (YOLOv3). It is so fast that it can run in realtime on a video (check out this nice demo).

YOLOv3's architecture is quite similar to the one we just discussed, but with a few important differences:

26 "You Only Look Once: Unified, Real-Time Object Detection," J. Redmon, S. Divvala, R. Girshick, A. Farhadi (2015).

27 "YOLO9000: Better, Faster, Stronger," J. Redmon, A. Farhadi (2016).

28 "YOLOv3: An Incremental Improvement," J. Redmon, A. Farhadi (2018).

- First, it outputs 5 bounding boxes for each grid cell (instead of just 1), and each bounding box comes with an objectness score. It also outputs 20 class probabilities per grid cell, as it was trained on the PASCAL VOC dataset, which contains 20 classes. That's a total of 45 numbers per grid cell (5 * 4 bounding box coordinates, plus 5 objectness scores, plus 20 class probabilities).
- Second, instead of predicting the absolute coordinates of the bounding box centers, YOLOv3 predicts an offset relative to the coordinates of the grid cell, where (0, 0) means the top left of that cell, and (1, 1) means the bottom right. For each grid cell, YOLOv3 is trained to predict only bounding boxes whose center lies in that cell (but the bounding box itself generally extends well beyond the grid cell). YOLOv3 applies the logistic activation function to the bounding box coordinates to ensure they remain in the 0 to 1 range.
- Third, before training the neural net, YOLOv3 finds 5 representative bounding box dimensions, called *anchor boxes* (or *bounding box priors*): it does this by applying the K-Means algorithm (see ???) to the height and width of the training set bounding boxes. For example, if the training images contain many pedestrians, then one of the anchor boxes will likely have the dimensions of a typical pedestrian. Then when the neural net predicts 5 bounding boxes per grid cell, it actually predicts how much to rescale each of the anchor boxes. For example, suppose one anchor box is 100 pixels tall and 50 pixels wide, and the network predicts, say, a vertical rescaling factor of 1.5 and a horizontal rescaling of 0.9 (for one of the grid cells), this will result in a predicted bounding box of size 150×45 pixels. To be more precise, for each grid cell and each anchor box, the network predicts the log of the vertical and horizontal rescaling factors. Having these priors makes the network more likely to predict bounding boxes of the appropriate dimensions, and it also speeds up training since it will more quickly learn what reasonable bounding boxes look like.
- Fourth, the network is trained using images of different scales: every few batches during training, the network randomly chooses a new image dimension (from 330×330 to 608×608 pixels). This allows the network to learn to detect objects at different scales. Moreover, it makes it possible to use YOLOv3 at different scales: the smaller scale will be less accurate but faster than the larger scale, so you can choose the right tradeoff for your use case.

There are a few more innovations you might be interested in, such as the use of skip connections to recover some of the spatial resolution that is lost in the CNN (we will discuss this shortly when we look at semantic segmentation). Moreover, in the 2016 paper, the authors introduce the YOLO9000 model that uses hierarchical classification: the model predicts a probability for each node in a visual hierarchy called *WordTree*. This makes it possible for the network to predict with high confidence that an image represents, say, a dog, even though it is unsure what specific type of dog it is.

So I encourage you to go ahead and read all three papers: they are quite pleasant to read, and it is an excellent example of how Deep Learning systems can be incrementally improved.

Mean Average Precision (mAP)

A very common metric used in object detection tasks is the *mean Average Precision* (mAP). "Mean Average" sounds a bit redundant, doesn't it? To understand this metric, let's go back to two classification metrics we discussed in Chapter 3: precision and recall. Remember the tradeoff: the higher the recall, the lower the precision. You can visualize this in a Precision/Recall curve (see Figure 3-5). To summarize this curve into a single number, we could compute its Area Under the Curve (AUC). But note that the Precision/Recall curve may contain a few sections where precision actually goes up when recall increases, especially at low recall values (you can see this at the top left of Figure 3-5). This is one of the motivations for the mAP metric.

Suppose the classifier has a 90% precision at 10% recall, but a 96% precision at 20% recall: there's really no tradeoff here: it simply makes more sense to use the classifier at 20% recall rather than at 10% recall, as you will get both higher recall and higher precision. So instead of looking at the precision *at* 10% recall, we should really be looking at the *maximum* precision that the classifier can offer with *at least* 10% recall. It would be 96%, not 90%. So one way to get a fair idea of the model's performance is to compute the maximum precision you can get with at least 0% recall, then 10% recall, 20%, and so on up to 100%, and then calculate the mean of these maximum precisions. This is called the *Average Precision* (AP) metric. Now when there are more than 2 classes, we can compute the AP for each class, and then compute the mean AP (mAP). That's it!

However, in an object detection systems, there is an additional level of complexity: what if the system detected the correct class, but at the wrong location (i.e., the bounding box is completely off)? Surely we should not count this as a positive prediction. So one approach is to define an IOU threshold: for example, we may consider that a prediction is correct only if the IOU is greater than, say, 0.5, and the predicted class is correct. The corresponding mAP is generally noted mAP@0.5 (or mAP@50%, or sometimes just AP_{50}). In some competitions (such as the Pascal VOC challenge), this is what is done. In others (such as the COCO competition), the mAP is computed for different IOU thresholds (0.50, 0.55, 0.60, …, 0.95), and the final metric is the mean of all these mAPs (noted AP@[.50:.95] or AP@[.50:0.05:.95]). Yes, that's a mean mean average.

Several YOLO implementations built using TensorFlow are available on github, some with pretrained weights. At the time of writing, they are based on TensorFlow 1, but by the time you read this, TF 2 implementations will certainly be available. Moreover, other object detection models are available in the TensorFlow Models project, many

with pretrained weights, and some have even been ported to TF Hub, making them extremely easy to use, such as SSD[29] and Faster-RCNN.[30], which are both quite popular. SSD is also a "single shot" detection model, quite similar to YOLO, while Faster R-CNN is more complex: the image first goes through a CNN, and the output is passed to a Region Proposal Network (RPN) which proposes bounding boxes that are most likely to contain an object, and a classifier is run for each bounding box, based on the cropped output of the CNN.

The choice of detection system depends on many factors: speed, accuracy, available pretrained models, training time, complexity, etc. The papers contain tables of metrics, but there is quite a lot of variability in the testing environments, and the technologies evolve so fast that it is difficulty to make a fair comparison that will be useful for most people and remain valid for more than a few months.

Great! So we can locate objects by drawing bounding boxes around them. But perhaps you might want to be a bit more precise. Let's see how to go down to the pixel level.

Semantic Segmentation

In *semantic segmentation*, each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.), as shown in Figure 14-26. Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the right side of the segmented image end up as one big lump of pixels. The main difficulty in this task is that when images go through a regular CNN, they gradually lose their spatial resolution (due to the layers with strides greater than 1): so a regular CNN may end up knowing that there's a person in the image, somewhere in the bottom left of the image, but it will not be much more precise than that.

29 "SSD: Single Shot MultiBox Detector," Wei Liu et al. (2015).

30 "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," Shaoqing Ren et al. (2015).

Figure 14-26. Semantic segmentation

Just like for object detection, there are many different approaches to tackle this problem, some quite complex. However, a fairly simple solution was proposed in the 2015 paper by Jonathan Long et al. we discussed earlier. They start by taking a pretrained CNN and turning into an FCN, as discussed earlier. The CNN applies a stride of 32 to the input image overall (i.e., if you add up all the strides greater than 1), meaning the last layer outputs feature maps that are 32 times smaller than the input image. This is clearly too coarse, so they add a single *upsampling layer* that multiplies the resolution by 32. There are several solutions available for upsampling (increasing the size of an image), such as bilinear interpolation, but it only works reasonably well up to ×4 or ×8. Instead, they used a *transposed convolutional layer*:[31] it is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 14-27). Alternatively, some people prefer to think of it as a regular convolutional layer that uses fractional strides (e.g., 1/2 in Figure 14-27). The *transposed convolutional layer* can be initialized to perform something close to linear interpolation, but since it is a trainable layer, it will learn to do better during training.

31 This type of layer is sometimes referred to as a *deconvolution layer*, but it does *not* perform what mathematicians call a deconvolution, so this name should be avoided.

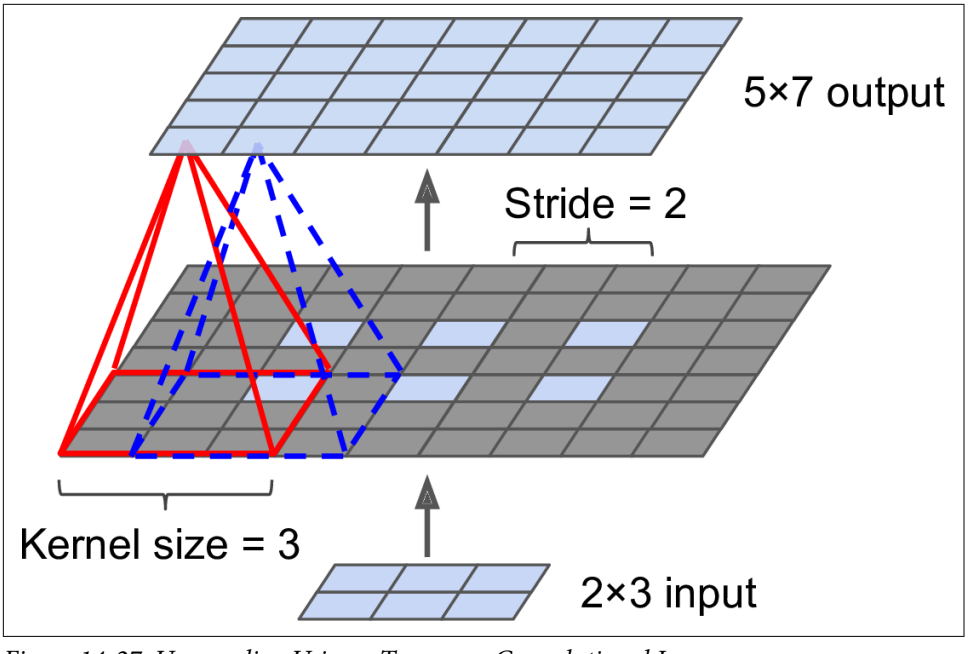

Figure 14-27. Upsampling Using a Transpose Convolutional Layer

In a transposed convolution layer, the stride defines how much the input will be stretched, not the size of the filter steps, so the larger the stride, the larger the output (unlike for convolutional layers or pooling layers).

TensorFlow Convolution Operations

TensorFlow also offers a few other kinds of convolutional layers:

- `keras.layers.Conv1D` creates a convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as we will see in ???.
- `keras.layers.Conv3D` creates a convolutional layer for 3D inputs, such as 3D PET scan.
- Setting the `dilation_rate` hyperparameter of any convolutional layer to a value of 2 or more creates an *à-trous convolutional layer* ("à trous" is French for "with holes"). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a 1 × 3 filter equal to `[[1,2,3]]` may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`. This allows the convolutional layer to

have a larger receptive field at no computational price and using no extra parameters.

- `tf.nn.depthwise_conv2d()` can be used to create a *depthwise convolutional layer* (but you need to create the variables yourself). It applies every filter to every individual input channel independently. Thus, if there are f_n filters and $f_{n'}$ input channels, then this will output $f_n \times f_{n'}$ feature maps.

This solution is okay, but still too imprecise. To do better, the authors added skip connections from lower layers: for example, they upsampled the output image by a factor of 2 (instead of 32), and they added the output of a lower layer that had this double resolution. Then they upsampled the result by a factor of 16, leading to a total upsampling factor of 32 (see Figure 14-28). This recovered some of the spatial resolution that was lost in earlier pooling layers. In their best architecture, they used a second similar skip connection to recover even finer details from an even lower layer: in short, the output of the original CNN goes through the following extra steps: upscale ×2, add the output of a lower layer (of the appropriate scale), upscale ×2, add the output of an even lower layer, and finally upscale ×8. It is even possible to scale up beyond the size of the original image: this can be used to increase the resolution of an image, which is a technique called *super-resolution*.

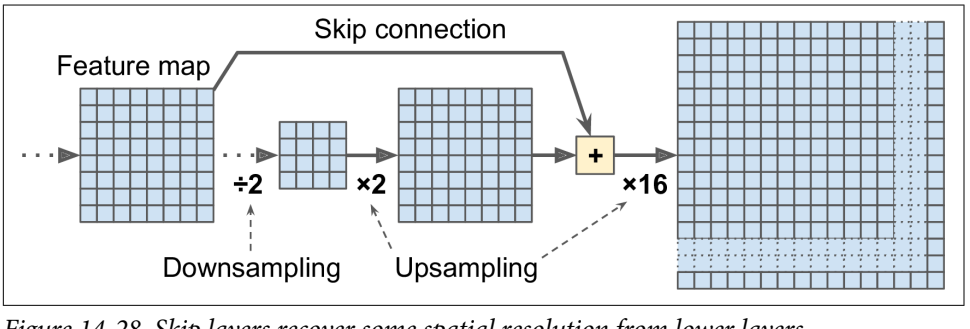

Figure 14-28. Skip layers recover some spatial resolution from lower layers

Once again, many github repositories provide TensorFlow implementations of semantic segmentation (TensorFlow 1 for now), and you will even find a pretrained *instance segmentation* model in the TensorFlow Models project. Instance segmentation is similar to semantic segmentation, but instead of merging all objects of the same class into one big lump, each object is distinguished from the others (e.g., it identifies each individual bicycle). At the present, they provide multiple implementations of the *Mask R-CNN* architecture, which was proposed in a 2017 paper: it extends the Faster R-CNN model by additionally producing a pixel-mask for each bounding box. So not only do you get a bounding box around each object, with a set of estimated class probabilities, you also get a pixel mask that locates pixels in the bounding box that belong to the object.

As you can see, the field of Deep Computer Vision is vast and moving fast, with all sorts of architectures popping out every year, all based on Convolutional Neural Networks. The progress made in just a few years has been astounding, and researchers are now focusing on harder and harder problems, such as *adversarial learning* (which attempts to make the network more resistant to images designed to fool it), explainability (understanding why the network makes a specific classification), realistic *image generation* (which we will come back to in ???), *single-shot learning* (a system that can recognize an object after it has seen it just once), and much more. Some even explore completely novel architectures, such as Geoffrey Hinton's *capsule networks*[32] (I presented them in a couple videos, with the corresponding code in a notebook). Now on to the next chapter, where we will look at how to process sequential data such as time series using Recurrent Neural Networks and Convolutional Neural Networks.

Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with 3×3 kernels, a stride of 2, and SAME padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200×300 pixels. What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? What about when training on a mini-batch of 50 images?
3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. When would you want to add a *local response normalization* layer?
6. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet and Xception?
7. What is a Fully Convolutional Network? How can you convert a dense layer into a convolutional layer?
8. What is the main technical difficulty of semantic segmentation?
9. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.

32 "Matrix Capsules with EM Routing," G. Hinton, S. Sabour, N. Frosst (2018).

10. Use transfer learning for large image classification.
    a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can just use an existing dataset (e.g., from TensorFlow Datasets).
    b. Split it into a training set, a validation set and a test set.
    c. Build the input pipeline, including the appropriate preprocessing operations, and optionally add data augmentation.
    d. Fine-tune a pretrained model on this dataset.
11. Go through TensorFlow's DeepDream tutorial. It is a fun way to familiarize yourself with various ways of visualizing the patterns learned by a CNN, and to generate art using Deep Learning.

Solutions to these exercises are available in ???.

About the Author

Aurélien Géron is a Machine Learning consultant. A former Googler, he led the YouTube video classification team from 2013 to 2016. He was also a founder and CTO of Wifirst from 2002 to 2012, a leading Wireless ISP in France; and a founder and CTO of Polyconseil in 2001, the firm that now manages the electric car sharing service Autolib'.

Before this he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He published a few technical books (on C++, WiFi, and internet architectures), and was a Computer Science lecturer in a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn and TensorFlow* is the fire salamander (*Salamandra salamandra*), an amphibian found across most of Europe. Its black, glossy skin features large yellow spots on the head and back, signaling the presence of alkaloid toxins. This is a possible source of this amphibian's common name: contact with these toxins (which they can also spray short distances) causes convulsions and hyperventilation. Either the painful poisons or the moistness of the salamander's skin (or both) led to a misguided belief that these creatures not only could survive being placed in fire but could extinguish it as well.

Fire salamanders live in shaded forests, hiding in moist crevices and under logs near the pools or other freshwater bodies that facilitate their breeding. Though they spend most of their life on land, they give birth to their young in water. They subsist mostly on a diet of insects, spiders, slugs, and worms. Fire salamanders can grow up to a foot in length, and in captivity, may live as long as 50 years.

The fire salamander's numbers have been reduced by destruction of their forest habitat and capture for the pet trade, but the greatest threat is the susceptibility of their moisture-permeable skin to pollutants and microbes. Since 2014, they have become extinct in parts of the Netherlands and Belgium due to an introduced fungus.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.