Convolutional Neural Networks for Image Classification

Francisco J. R. Ruiz

1 Introduction

Regular neural networks do not scale well to full images. Consider MNIST, which is composed of 28×28 images. A fully connected layer of a regular neural network would thus use 784 weights per neuron in the next layer. This is manageable, but consider a more realistic image of, say, 1000×1000 pixels and 3 color channels (red, green, and blue). In this case, a fully connected neural network would need $3,000,000$ weights per neuron. Clearly, this full connectivity becomes intractable and wasteful and the huge number of parameters would quickly lead to overfitting. As a solution, convolutional neural networks (ConvNets) allow to encode certain properties of images into the architecture of the network. This results in a more efficient implementation and a vast reduction on the number of parameters.

ConvNets are very similar to ordinary neural networks like the ones implemented on Day 4. They are too made up to neurons that have learnable weights and biases. Each neuron receives some inputs, performs products and sums, and optionally applies a non-linear function afterwards. However, ConvNets explicitly assume that the inputs are images, which allows to encode certain properties.

2 Architectures

A typical ConvNet consists of a sequence of layers. There are many types of layers, but we will focus on three of them: fully connected layers, convolutional layers, and pooling layers. ConvNets typically contain one or more of each type of layers. The layout of layers form what we call the architecture of the network. For instance, the architecture that we will use in this project is depicted in Figure 1. Under this architecture, the input images are passed (in this order) through a convolutional layer, a pooling later, a second

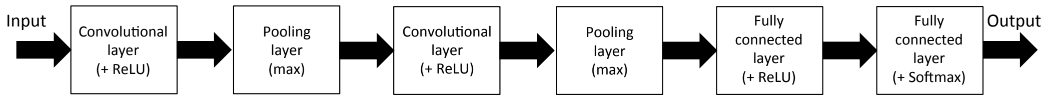
Input
Convolutional layer (+ ReLU)
Pooling layer (max)
Convolutional layer (+ ReLU)
Pooling layer (max)
Fully connected layer (+ ReLU)
Fully connected layer (+ Softmax)
Output

Figure 1: Architecture of the ConvNet considered in this project. The input is an image and the output are the predictions (probabilities) for each class. The network consists of two convolutional layers, each followed by a max-pooling layer, and two fully connected layers. The convolutional layers capture the spatial properties of the images, and the pooling layers reduce the dimensionality so that the final fully connected layers have less parameters.

convolutional layer, a second pooling layer, and two fully connected layers. The last fully connected layer performs a softmax operation, and thus the outputs are the probabilities for each class (for instance, the probability for each digit, in the case of MNIST).

Of course, this is not the only possible architecture. The number and type of layers varies depending on the application. In some applications, a cascade of several convolutional layers is used before the pooling layer. In other applications, no pooling layer is used.

In general, determining the best network architecture for a specific application is a hard problem that requires heuristics based on experience. However, in practice you should not have to worry about the specific architecture. Instead, we would recommend to use the architecture that currently works best on ImageNet.

We will next provide more details about these three main types of layers.

2.1 Fully Connected Layers

These are the layers that we considered on Day 4. In a fully connected layer, each output neuron is connected to *all* the input neurons. Figure 2 shows an illustration.

A fully connected layer has $K_1 \times K_2$ weights and K_2 biases, where K_1 is the number of neurons of the input layer and K_2 is the number of neurons of the output layer.

2.2 Convolutional Layers

Main ideas. Convolutional layers rely on two main ideas: local connectivity and parameter sharing.

Local connectivity means that neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons. See

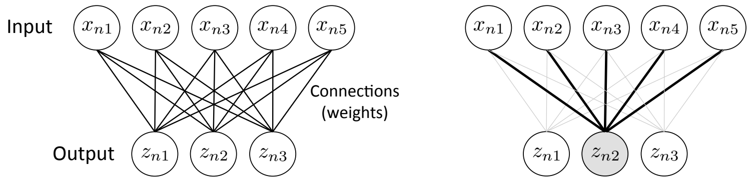

Figure 2: An illustration of a fully connected layer. (Left) It has 5 input neurons and 3 output neurons, and thus 5×3 weight parameters. (Right) Each output neuron (we show z_{n2} as an example) is connected to *all* input neurons.

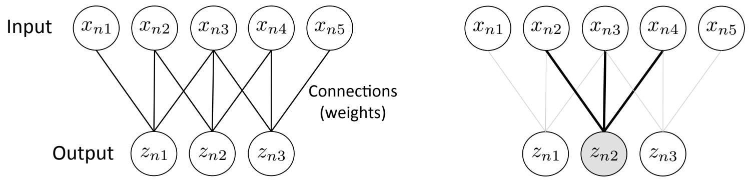

Figure 3: An illustration of local connectivity. (Left) This scheme also has 5 input neurons and 3 output neurons, but it has only 9 weight parameters due to local connectivity. If the parameters are shared, then this scheme has only 3 weights. (Right) Each output neuron is connected to a *small region* of the input space (3 neurons in this example).

Figure 3 for an illustration. The network in the figure has 9 connections and therefore 9 weight parameters, instead of 15 as the fully connected layer.

Parameter sharing means that the weights corresponding to different output neurons are set to the same value, which implies that we need to learn a smaller number of weights. In the example of Figure 3, this means that we would only need 3 weights, which are shared between z_{n1}, z_{n2}, and z_{n3}.

Convolutions. In Figure 3, each output neuron performs the operation $z_{nk} = \sum_{d=1}^{3} w_d x_{n(d+k-1)} + b$ (normally followed by a non-linear function). We have made explicit that we use parameter sharing and therefore we only have three weights (w_1, w_2, and w_3) and one bias (b). This operation is formally known as *convolution*. Note that it is just a collection of products and sums, arranged in a particular way.

In ConvNets, it is common to use two extra operations when performing convolutions: zero padding and striding. *Zero padding* means to add zeros on the "edges" of the input layer, as illustrated in Figure 4 (left). When dealing with images, zero padding allows to preserve the information on the boundaries of the image. It is typically applied to set the same number of output neurons equal to the number of input neurons. *Striding* means to discard some of the ouput neurons (evenly spaced) in order to reduce the dimensionality of the output layer. For instance, when the stride is 2, we

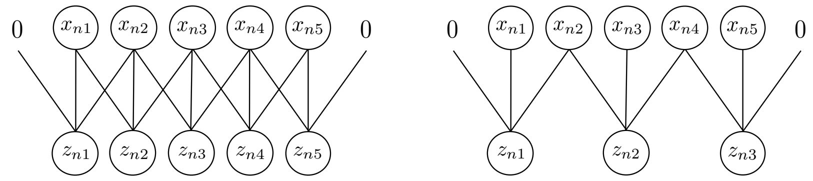
Figure 4: An illustration of zero padding to preserve the number of neurons. (Left) Without striding. (Right) When the stride is 2, we reduce the dimensionality of the output by half (except for rounding).

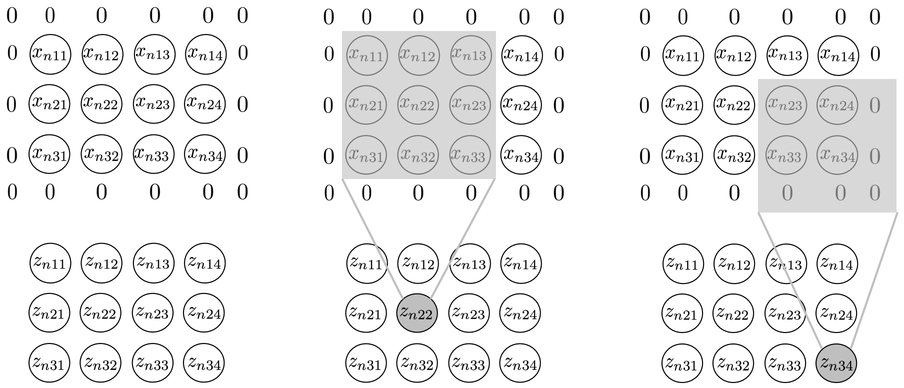
Figure 5: Illustration of a bidimensional convolution. The input space is a zero-padded 3×4 matrix of neurons, and the output space has also size 3×4. The weights are arranged in a 3×3 matrix. (Left) We do not show the connections for visual display, but each output neuron is connected to a small region of the input space (3×3 in this example). (Center) Neurons to which z_{n22} is connected. (Right) Neurons to which z_{n34} is connected.

only keep one out of every 2 output neurons. See Figure 4 (right) for an illustration of zero padding and striding.

2-D convolutions. The concepts above are useful in convolutional layers, but we have ignored that images are represented as matrices, i.e., represented in a 2-dimensional space.[1] The convolutions should then be performed in this bidimensional space. Fortunately, the ideas above about local connectivity and parameter sharing are still valid in bidimensional spaces.

As an example, consider Figure 5 (left). The input space consists in a 3×4 matrix of neurons, and the output space is also bidimensional. Each output neuron is connected to a small region of the input space; in this example, a 3×3 region. Thus, we use a 3×3 matrix of weights (also known as *filter*). We also use zero-padding to preserve the dimensionality of the output space.

[1]Actually, images are represented in a 3-dimensional space when we also consider the three color channels; we will consider that later.

Figure 5 (center) and Figure 5 (right) show the input neurons to which two specific output neurons are connected.

3-D volumes. In reality, images are arranged in 3-dimensional volumes (height $\times$ width $\times$ depth), where the term "depth" here does *not* refer to the depth of the network, but to the third dimension of the volume. For instance, a color image has a depth of 3 because it has 3 color channels (red, green, and blue).

In convolutional layers, the most common practice is as follows. The height and width dimensions are preserved using zero padding (as shown in Figure 5), but the depth is changed to a different value. For instance, an input image of size $100 \times 100 \times 3$ may be converted into an output of size $100 \times 100 \times 32$. We chose the number 32 as an example; in general, the depth of the hidden layers is a hyperparameter. Note that the input layer does not necessarily have depth of 3. For instance, the $100 \times 100 \times 32$ volume might be the input of another convolutional layer, in which case the input depth would be 32.

How are the outputs computed? Convolutions are carried out on bi-dimensional slices, as described earlier. The only difference is that the summations extend across the entire depth of the input layer. For instance, consider an input image of size $100 \times 100 \times 3$. Assume we use a convolution that looks at 5×5 (height $\times$ width) regions of the input space. In the bi-dimensional case, the weight matrix (filter) would be of size 5×5. In the three-dimensional case, the weights can be arranged in a $5 \times 5 \times 3$ tensor for each depth slice of the output. The $100 \times 100 \times 3$ input, convolved with a filter of size $5 \times 5 \times 3$ (and using zero padding) would result in a $100 \times 100 \times 1$ volume. This operation is repeated as many times as the output depth, with different weights. Thus, the weights would actually be a $5 \times 5 \times 3 \times 32$ tensor (height $\times$ width $\times$ input depth $\times$ output depth) if we wish to produce an output volume of $100 \times 100 \times 32$. Each $5 \times 5 \times 3 \times 1$ slice of the weight tensor would produce a $100 \times 100 \times 1$ slice of the output volume.

This might look complicated, but the most important thing to keep in mind are the sizes of the inputs, outputs, and weights. Convolutions are performed on bi-dimensional slices and across the entire depth of the input layer. The weight tensor has size (filter height $\times$ filter width $\times$ input depth $\times$ output depth). The biases are a vector of length equal to the output depth (32 bias terms in this example). The height and width of the input space are preserved if we use zero padding, and the only dimension that changes is the depth.

Non linearities. Similarly to fully connected layers, convolutional layers are also followed by a non-linear function. We will consider only the ReLU activation function here.

2.3 Pooling Layers

It is common to periodically insert a pooling layer in-between successive convolutional layers in a ConvNet architecture. Pooling layers are similar to convolutional layers in that they use local connectivity. Their function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The pooling layer operates independently on every depth slice of the input and resizes the height and width. Pooling layers use striding to resize (indeed, it is more common to resize the height and width with pooling layers than with convolutional layers).

The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 along both width and height. In contrast to convolutional layers, pooling layers *do not have parameters*, i.e., they do not have weights or biases. Instead of sums and products, pooling layers perform other operations with the input regions. The most common operation is $\max(\cdot)$, and thus they output the maximum of each 2×2 region. Thus, the width and height are decreased by half, and the depth dimension remains unchanged.

2.4 Dropout Layers

Another technique to avoid overfitting in neural networks is dropout. The key idea is to randomly drop neurons (along with their connections) at each iteration during training. That is, at each iteration we randomly ignore each neuron with a certain probability p, and keep only the remaining neurons. In the next iteration, the set of "ignored" neurons will vary. This signicantly reduces overfitting and gives major improvements over other regularization methods.

Dropout is typically applied before the readout layer (i.e., between the two fully connected layers in Figure 1). However, on this project, the application of dropout is optional.

3 Implementation on TensorFlow

Each of the operations in Section 2 is implemented in TensorFlow.
Convolution. The convolution operation with input `x` and filter `W` can be performed with

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
                        padding='SAME')
```

Here, we have specified not to use striding (or, equivalently, use striding of 1 across all dimensions), and use zero padding to preserve the input size (`SAME`). Recall that the input `x` must have size (input height $\times$ input width $\times$ input depth), the filter `W` must be (filter height $\times$ filter width $\times$ input depth $\times$ output depth), and the output of the convolution will be (input height $\times$ input width $\times$ output depth).

Pooling. The max-pooling operation can be performed with this auxiliary function:

```
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')
```

This will reduce the height and width of the input by a factor of 2, while preserving the depth.

Dropout. If you wish to add a dropout layer, we recommend using the TensorFlow function `tf.nn.dropout`, which takes care of rescaling of the gradients.

```
def dropout(x, keep_prob):
    return tf.nn.dropout(x, keep_prob)
```

The parameter `keep_prob` controls the probability of keeping a neuron. For training, a common value is 0.5. Importantly, this parameter should be set to 1.0 for testing.

Reshaping. We strongly recommend to be aware of the size of the input and output of each layer. You may need to introduce *reshape* operations throughout the code to adapt the sizes of the tensors. For handling the dimensionality of tensors, TensorFlow provides the functions `tf.reshape()` and `tf.transpose()`.

4 Datasets

We will use these two datasets:

- The MNIST dataset, a collection of 60,000 images of handwritten digits, of 28×28 pixels each. This has been used in previous lab sessions. Note that the input tensor in this case would be $28 \times 28 \times 1$ in size, since there is no color channel.

- The CIFAR-10 dataset,[2] which contains 50,000 color images of 32×32 pixels that are classified into one of the following categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

It can take long (more than 30 minutes) to train the model on this data. If this is the case, we suggest considering only a randomly chosen subset of the training set.

5 Project Tasks

Implement a ConvNet for multiclass image classification. You can use the architecture of Figure 1, optionally adding a dropout layer. As a starting point, you may use a depth of 32 for the output of the first convolutional layer, depth of 64 for the output of the second convolutional layer, and 1024 output neurons for the first fully connected layer. Use stochastic gradient descent with RMSProp (`tf.train.RMSPropOptimizer`) or Adam (`tf.train.AdamOptimizer`), and set the learning rate to a small value (say, 10^{-5} or 10^{-4}).

Fit the model on the two datasets detailed above, reporting the predictive performance for each case. For each dataset, plot some examples of correctly classified images, and some examples of images that were not correctly classified. For the latter, visualize the output probabilities of the predicted classes.

If you have enough time, you can also: (i) explore how the number of hidden units of the convolutional layers and/or the fully connected layer affects performance; or (ii) explore variations of the architecture depicted in Figure 1 (for instance, you may remove the first pooling layer or introduce a third convolutional layer).

[2] It is available at `https://www.cs.toronto.edu/~kriz/cifar.html`. Download the Python version *before* coming to class (it may take some time).