Contents

1 CNNS

IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996

CNNs emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s.

Due to the increase in computational power, the amount of available training data, CNNs have managed to achieve superhuman performance on some complex visual tasks.

They power image search services, self-driving cars, automatic video classification systems, and voice recognition and natural language processing.

2 Convolutional Layers

The most important building block of a CNN is the convolutional layer.

Neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields.

Each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer.

This architecture allows the network to concentrate on small low-level features in the first hidden layer.

Then assemble them into larger higher-level features in the next hidden layer, and so on.

In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in

Position of a neuron	Connections in the previous layer
(i,j)	$(i{:}i+f_h-1, j{:}j+f_w-1)$

where f_h and f_w are the height and width of the receptive field.

2.1 Padding

In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs. This is called zero padding.

2.2 Strides

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 14-4. This dramatically reduces the model's computational complexity. The shift from one receptive field to the next is called the stride.

Position of a neuron	Connections in the previous layer
(i,j)	$(i*s_h{:}i*s_h+f_h-1, j{:}j*s_w+f_w-1)$

where s_h and s_w are the vertical and horizontal strides.

2.3 Filters (Convolutional Kernel)

A neuron's weights can be represented as a small image the size of the receptive field, called **filters** or **convolution kernels**. Thus, a layer full of neurons using the same filter outputs a **feature map**, which highlights the areas in an image that activate the filter the most.

You do not have to define the filters manually. During training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

In reality a convolutional layer has multiple filters and outputs **one feature map per filter**.

It has one neuron per pixel in each feature map,

All neurons within a given feature map share the same parameters.

A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps.

A convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model.

Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location.

layer l, feature map k	layer l-1
(i,j)	(ixs$_h$:ixs$_h$+f$_h$-1, jxs$_w$:jxs$_w$+f$_w$-1)

3 Memory Requirements

The reverse pass of backpropagation requires all the intermediate values computed during the forward pass. If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices.

4 Pooling Layers

Each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. define its size, the stride, and the padding type, no weights;

A max pooling layer also introduces some level of invariance to small translations.

By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale.

Max pooling offers a small amount of rotational invariance and a slight scale invariance.

Such invariance can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

5 TensorFlow Implementation

Strides default to the kernel size,

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

the depthwise max pooling layer average pooling layer, AvgPool2D max pooling preserves only the strongest features,

max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Keras does not include a depthwise max pooling layer, but TensorFlow's low-level Deep Learning API does: #+begin$_{\mathrm{src}}$ python output = tf.nn.max$_{\mathrm{pool}}$(images, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3), padding="valid") #+end$_{\mathrm{src}}$ python

You can include this as a layer in your Keras models

#+begin$_{\mathrm{src}}$ python depth$_{\mathrm{pool}}$ = keras.layers.Lambda(lambda X: tf.nn.max$_{\mathrm{pool}}$(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3), padding="valid")) #+end$_{\mathrm{src}}$ python

Global average pooling layer.

#+begin$_{\mathrm{src}}$ python global$_{\mathrm{avgpool}}$ = keras.layers.GlobalAvgPool2D() #+end$_{\mathrm{src}}$ python It's equivalent to this simple Lambda layer, which computes the mean over the spatial dimensions (height and width):

#+begin$_{\mathrm{src}}$ python global$_{\mathrm{avgpool}}$ = keras.layers.Lambda(lambda X: tf.reduce$_{\mathrm{mean}}$(X, axis= [1, 2])) #+end$_{\mathrm{src}}$ python

6 Tensor Flow Implementation

Input image: [height, width, channels]. A mini-batch: [mini-batch size, height, width, channels]. Weights of convolutional layer: [f$_{\mathrm{h}}$, f$_{\mathrm{w}}$, f$_{\mathrm{n}}$, f$_{\mathrm{n}}$]. The bias terms of a convolutional layer: 1D tensor of shape [f$_{\mathrm{n}}$].

```
from sklearn.datasets import load_sample_image
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line
outputs = tf.nn.conv2d(images, filters, strides=1, padding="same")
plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd
feature map
plt.show()
```

Strides is equal to 1, Strides could could also be a 1D array with four elements, where the two central elements are the vertical and horizontal strides (s_h and s_w). The first and last elements must currently be equal to 1.

If set to "valid", the convolutional layer does not use zero padding and may ignore some rows and columns at the bottom and right of the input image, depending on the stride. This means that every neuron's receptive field lies strictly within valid positions inside the input, hence the name valid.

In a real CNN you would normally define filters as trainable variables so the neural net can learn which filters work best.

```
conv = keras.layers.Conv2D(filters=32,
                           kernel_size=3,
                           strides=1,
                           padding="same",
                           activation="relu")
```

7 CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on.

The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper.

At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs),

and the final layer outputs the prediction.

A common mistake is to use convolution kernels that are too large. Smaller kernels use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer

```
model = keras.models.Sequential([
keras.layers.Conv2D(64, 7, activation="relu", padding="same", input_shape=[28, 28, 1])
keras.layers.MaxPooling2D(2),
keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
keras.layers.MaxPooling2D(2),
keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
keras.layers.MaxPooling2D(2),
keras.layers.Flatten(),
keras.layers.Dense(128, activation="relu"),
keras.layers.Dropout(0.5),
keras.layers.Dense(64, activation="relu"),
keras.layers.Dropout(0.5),
keras.layers.Dense(10, activation="softmax")])
```

This CNN reaches over 92% accuracy on the test set.

8 LeNet-5

It was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST).

9 AlexNet

The AlexNet CNN architecture won the 2012 ImageNet ILSVRC challenge it achieved a top-five error rate of 17%, Alex Krizhevsky et al. it was the first to stack convolutional layers directly on top of one another,

To reduce over-fitting, the authors used two regularization techniques.

First, they applied dropout with a 50% AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called **local response normalization** (LRN): the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps. This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization.

10 GoogLeNet

Developed by Christian Szegedy et al. from Google Research. Won the ILSVRC 2014; the top-five error rate below 7%. much deeper than previous CNNs Sub-networks called inception modules which allow GoogLeNet to use parameters much more efficiently Inception module: "$3 \times 3 + 1$(S)" means that the layer uses a 3×3 kernel, stride 1, and "same" padding.

The input signal is first copied and fed to four different layers.

All convolutional layers use the ReLU activation function.

Concatenate all the outputs along the depth dimension in the final depth concatenation layer.

- In fact, the layers serve three purposes: Although they cannot capture spatial patterns, they can capture patterns along the depth dimension.
- They are configured to output fewer feature maps than their inputs, so they serve as bottleneck layers, meaning they reduce dimensionality.
- Each pair of convolutional layers acts like a single powerful convolutional layer, capable of capturing more complex patterns. simple linear classifier across the image, this pair of convolutional layers sweeps a two-layer neural network across the image.

The number of convolutional kernels for each convolutional layer is a hyperparameter. Includes nine inception modules. The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module

The first two layers divide the image's height and width by 4 (so its area is divided by 16) Then the local response normalization layer ensures that the previous layers learn a wide variety of features.

Two convolutional layers follow, where the first acts like a bottleneck layer.

Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN this considerably reduces the number of parameters in the network and limits the risk of overfitting.

11 VGGNet

It had a very simple and classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on, plus a final dense network with 2 hidden layers and the output layer. It used only 3×3 filters, but many filters.

12 ResNet

- Models are getting deeper and deeper
- fewer and fewer parameters
- use skip connections
- The goal is to make it model a target function h(x)
- If you add the input x to the output of the network,
- Then the network will be forced to model f(x) = h(x) - x
- If the target function is fairly close to the identity function,
- This will speed up training considerably.
- If you add many skip connections, the network can start making Progress even if several layers have not started learning yet.
- The signal can easily make its way across the whole network.
- The deep residual network can be seen as a stack of residual units,
- Where each residual unit is a small neural network with a skip connection.

Each residual unit is composed of two convolutional layers

number of feature maps is doubled every few residual units their height and width are halved. When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape.

We solve this problem, the inputs are passed through a 1×1 convolutional layer with stride 2 and the right number of output feature maps.

13 Xception

it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a depthwise separable convolution layer.

These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture.

A separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately.

Thus, it is composed of two parts:

- A single spatial filter for each input feature map,
- Then the second part looks exclusively for cross-channel patterns

Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions, plus a few max pooling layers and the usual final layers.

Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and in general they even perform better, so you should consider using them by default.

14 SENet

The extended versions of inception networks and ResNets are called SE-Inception and SE-ResNet, respectively.

The boost comes from the fact that a SENet adds a small neural network, called an SE block, to every unit in the original architecture,

An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps, as shown in Figure 14-21. For example, an SE block may learn that mouths, noses, and eyes usually appear together in pictures: if you see a mouth and a nose, you should expect to see eyes as well. So if the block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map (more accurately, it will reduce irrelevant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see Figure 14-

1.

As earlier, the global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256 numbers representing the overall level of response for each filter. The next layer is where the "squeeze" happens: this layer has significantly fewer than 256 neurons—typically 16 times fewer than the number of feature maps (e.g., 16 neurons)—so the 256 numbers get compressed into a small vector (e.g., 16 dimensions). This is a low- dimensional vector representation (i.e., an embedding) of the distribution of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations (we will see this principle in action again when we discuss autoencoders in Chapter 17). Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.