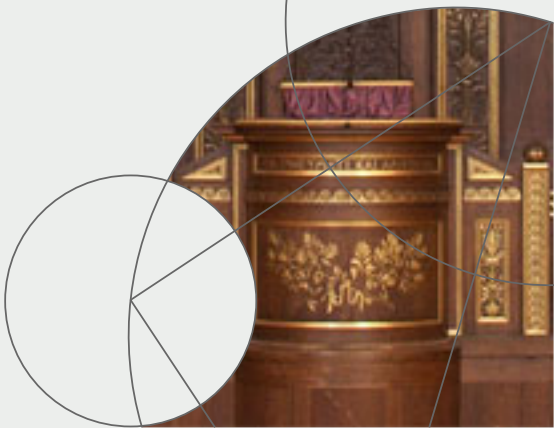
Deep Learning: Convolutional Neural Network

Rensheng Wang,
https://sit.instructure.com/courses/22680
August 31, 2019



Convolutional Neural Network

- Convolutional neural networks (CNNs), also known as *convnets*, a type of deep-learning model almost universally used in computer vision applications.
- *Convnets* a stack of Conv2D and MaxPooling2D layers .
- ☞ Example: Instantiating a small convnet

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Introduction to Convnets

- Importantly, a convnet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension).
- We configure the convnet to process inputs of size (28, 28, 1), which is the format of MNIST images. We pass the argument `input_shape=(28, 28, 1)` to the first layer.

```
>>> model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)  (None, 13, 13, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)  (None, 5, 5, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

Introduction to Convnets

- You can see that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as you go deeper in the network.
- The number of channels is controlled by the first argument passed to the Conv2D layers (32 or 64).
- The next step is to feed the last output tensor (of shape `(3, 3, 64)`) into a densely connected classifier network like those youre already familiar with: a stack of Dense layers. These classifiers process vectors, which are 1D, whereas the current output is a 3D tensor. First we have to flatten the 3D outputs to 1D, and then add a few Dense layers on top.

☞ Example: Adding a classifier on top of the convnet

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Introduction to Convnets

- We'll do 10-way classification, using a final layer with 10 outputs and a softmax activation.

```
>>> model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

- As you can see, the `(3, 3, 64)` outputs are flattened into vectors of shape `(576,)` before going through two Dense layers.

Introduction to Convnets

☞ Training the convnet on MNIST images

```
from keras.datasets import mnist
from keras.utils import to_categorical
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
model.compile(optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Introduction to Convnets

- Lets evaluate the model on the test data:

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
0.99080000000000001
```

- Why does this simple `convnet` work so well, compared to a densely connected model?

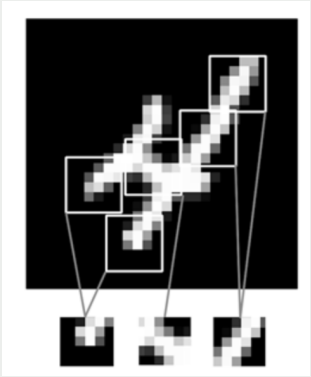
  To answer this, lets dive into what the `Conv2D` and `MaxPooling2D` layers do.

The Convolution Operation

- The fundamental difference between a densely connected layer and a convolution layer is this:

  Dense layers learn global patterns in their input feature space (for example, for a MNIST digit, patterns involving all pixels), whereas convolution layers learn local patterns: in the case of images, patterns found in small 2D windows of the inputs. In the previous example, these windows were all 3×3.

- Images can be broken into local patterns such as edges, textures, and so on:



The Convolution Operation

- This key characteristic gives convnets two interesting properties:
  - ☞ The patterns they learn are translation invariant.

    After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere: for example, in the upper-left corner.

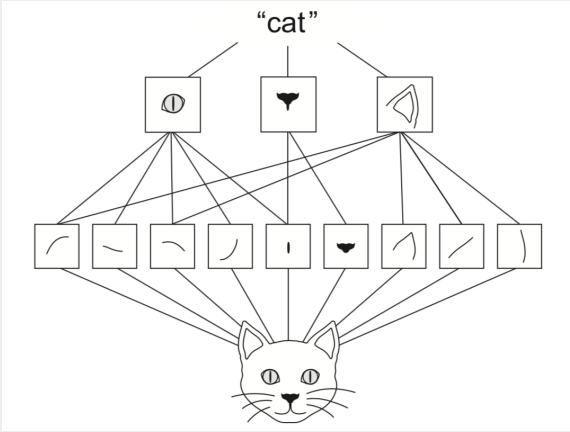
    A densely connected network would have to learn the pattern anew if it appeared at a new location.

    This makes convnets data efficient when processing images (because the visual world is fundamentally translation invariant): they need fewer training samples to learn representations that have generalization power.
  - ☞ They can learn spatial hierarchies of patterns

    A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on.

Spatial Hierarchies of Patterns

- It allows convnets to efficiently learn increasingly complex and abstract visual concepts (because the visual world is fundamentally spatially hierarchical)
- The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as cat.

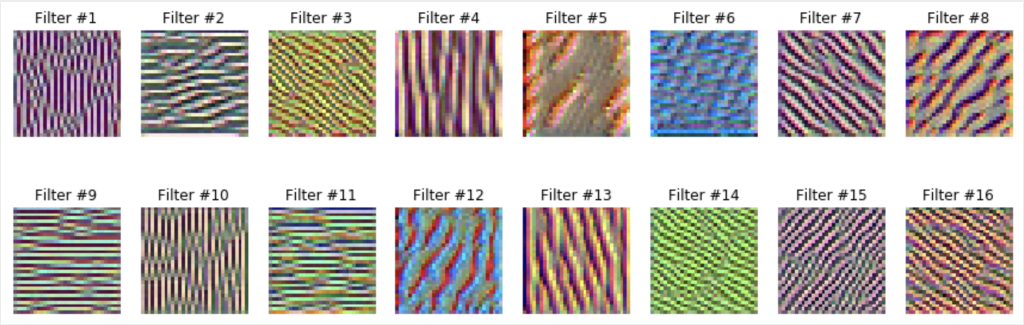


The Convolution Operation

- Convolutions operate over 3D tensors, called feature maps, with two spatial axes (height and width) as well as a depth axis (also called the channels axis).
- For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue.
- For a black-and-white picture, like the MNIST digits, the depth is 1 (levels of gray). The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an output feature map.
- This output feature map is still a 3D tensor: it has a width and a height. Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, they stand for *filters*.

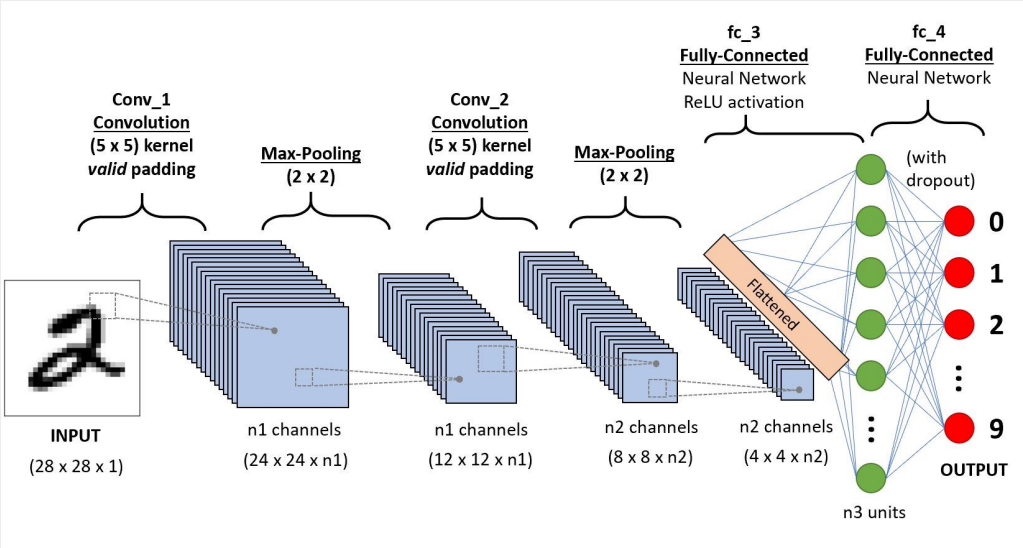
Visualize Learnt Filters

- Some filters learnt from the CNN training:



CNN Architecture Example

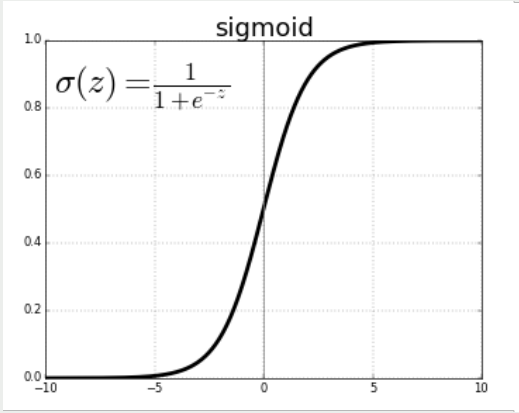
- A CNN sequence to classify handwritten digits

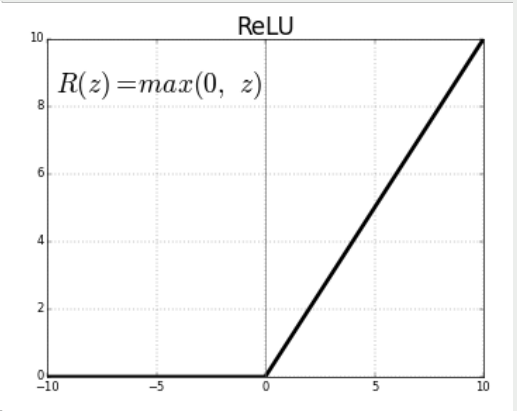


Activation Functions Comparison

- `sigmoid` function is differentiable everywhere but for a relatively large $|z|$, differentiation of `sigmoid` is also zero, i.e., no change for network gradients learning

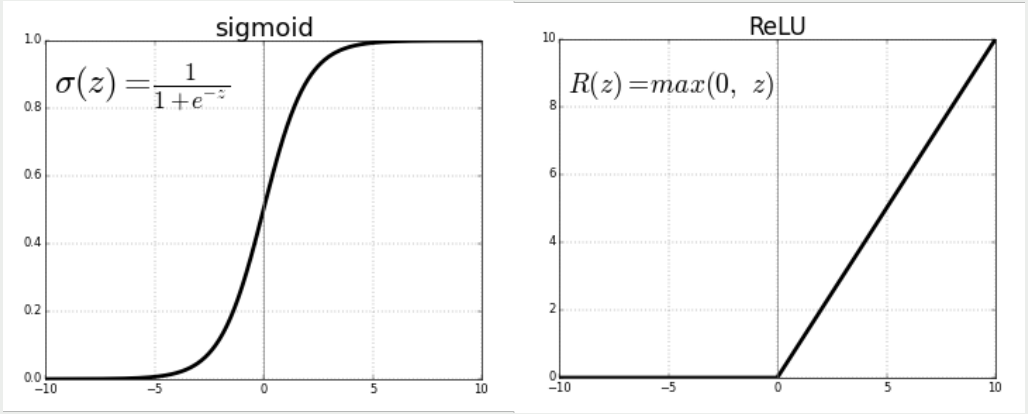
☞ Check for the range $|z| > 5$, $\partial\sigma/\partial z \approx 0$.





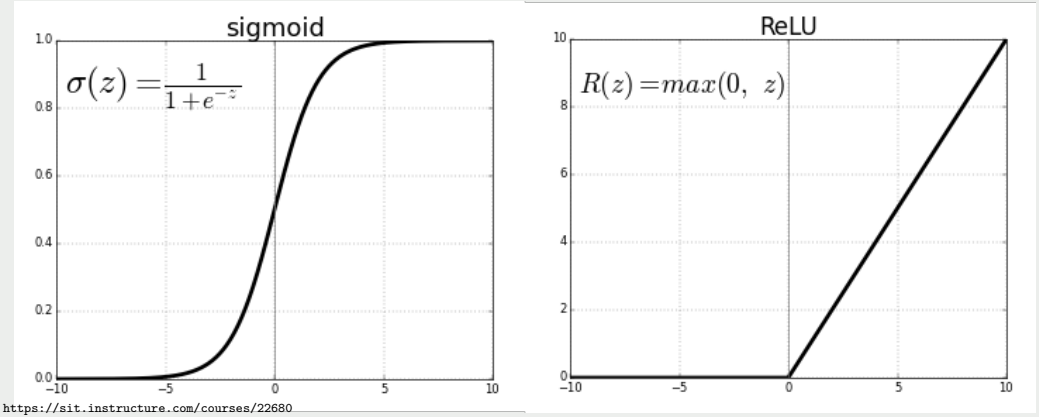
Activation Functions Comparison

- `sigmoid` function is differentiable everywhere but for a relatively large $|z|$, differentiation of `sigmoid` is also zero, i.e., no change for network gradients learning
- ☞ Check for the range $|z| > 5$, $\partial\sigma/\partial z \approx 0$.
- The differentiation of the exponential function is expensive for computations.



Activation Functions Comparison

- ReLU function is not differentiable everywhere but the differentiation of ReLU is a constant.
- The differentiation of the ReLU is very efficient.
- ☞ For $z < 0$, if the neural network neurons hit lucky for ZERO values, then they might keep zeros for the following layers' neurons since the gradients learning over ZERO is still "0", no improvements.



ReLU vs. Leaky ReLU

- Leaky ReLU is an attempt to solve the dying ReLU problem for ZERO values.
- `ReLU` $= \max(0, y)$ vs. `Leaky ReLU`$=\max(ay, y)$, for instance, choose $a = 0.1$, then `Leaky ReLU`$=\max(0.1y, y)$