Deep Learning Lecture - Convolution NN

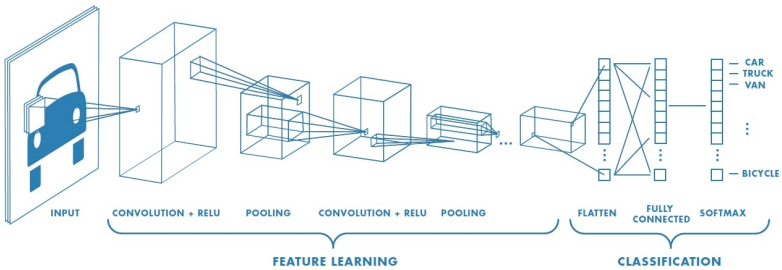
MA8701 General Statistical Methods

Thiago G. Martins, Department of Mathematical Sciences, NTNU

Spring 2019

Convolutional Neural Networks (Convnets or CNNs)

A typical CNN sketch:



Source: Mathworks page about CNN

Feature Learning part of the CNN defined in Keras:

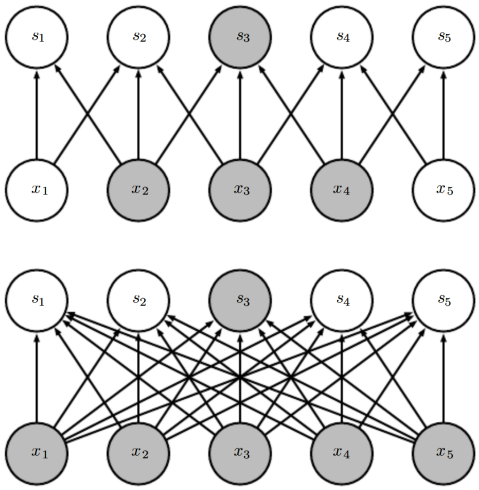
```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu")
```

Classification part of the CNN defined in Keras:

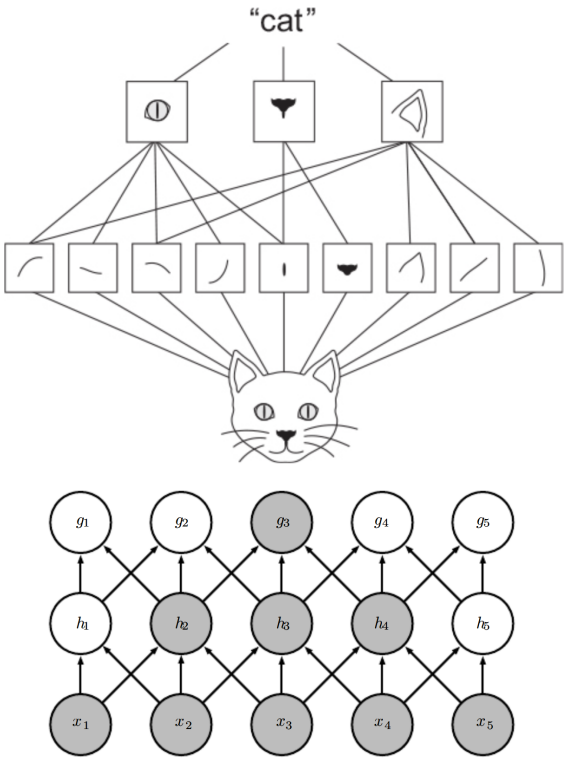
```
model <- model %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")
```

Convnet layers

- Dense layers learn global patterns in their input space
- Convolutional layers learn local patterns



- The patterns they learn are translation invariant
  - If they learn a pattern in the lower-right of an image, they would also recognise the pattern on the upper-left
  - Dense layers would have to learn the pattern anew in a different position
  - This makes convnets data efficient, needing less data to learn
- They can learn spatial hierarchies of patterns
  - The first layers learn small patterns such as edges
  - Later layers learn larger patterns based on the small patterns

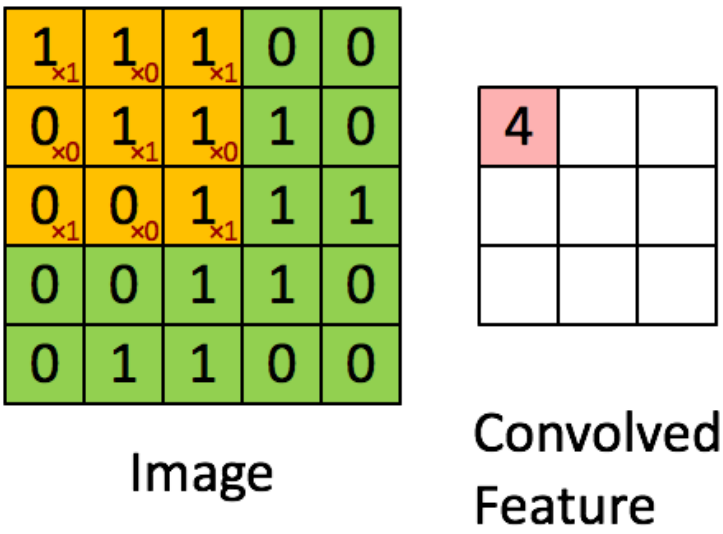


A CNN model alternate between convolutional and max-pooling layers.

2D convolutional layer

A 2D convolutional layer is defined by `layer_conv_2d` in Keras. The two main arguments for the layer are `filters` and `kernel_size`.

On the GIF below, we see one filter being produced by kernel size `(3, 3)` from an image with dimension `(5,5,1)`. This particular filter has dimension `(3,3)`.



The following layer will create 32 filters such as the one above by applying a convolution with kernel size `(3, 3)` into a image with dimension `(28, 28, 1)`. Each of the 32 filters would have dimension `(26, 26)`

```
layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input_s
```

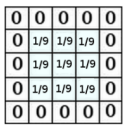
The dimension of the resulting filter is `(26, 26)` and not `(28, 28)` due to border effects.

Each filter will go through the following transformation:

```
filter = convolution(input)
output = activation(filter + bias)
```

The total number of parameters defined in `layer_conv_2d` is given by `kernel_height * kernel_width * input_channel * filters` (convolution operation) + `filters` (one bias per filter).

Averaging adjacent pixels blur the image:



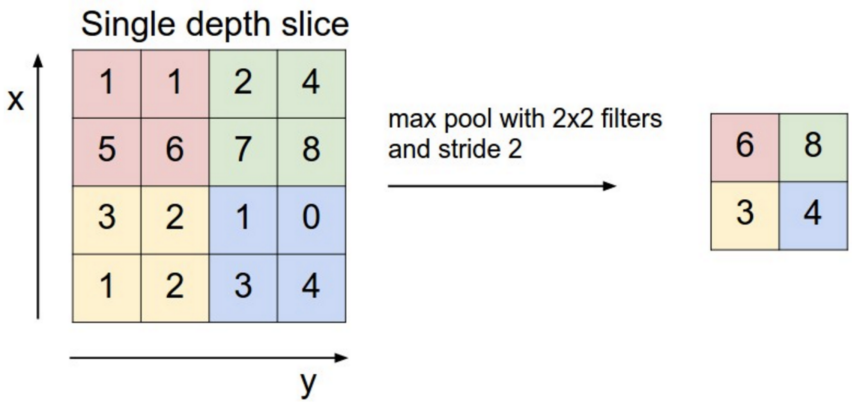
Adjacent pixels are very different in the direction perpendicular to the edge:

0	0	0	0	0
0	0	0	0	0
0	-1	1	0	0
0	0	0	0	0
0	0	0	0	0

2D max pooling layer

The max pooling layer is conceptually similar to the convolutional layer, with two main differences.

- Instead of transforming local patchs via a learned linear transformation, they are transformed via a hard-coded max operation.
- The window size is usually `(2,2)` and the stride is equal to 2 (instead of 1 for the convolutional layer), downsampling the filters.



The reasons to downsample are:

- to reduce the number of coefficients to process
- to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).

MNIST dataset

The objective here is to classify the digit contained in a image using a convnet model.

Convnet model

Convolutional Neural Network in Keras:

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
```

```
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu")
```

Adding a classifier on top of the convnet:

```
model <- model %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")
```

Inspect the model:

```
model
```

```
## Model
## ____________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ============================================================================
## conv2d_4 (Conv2D)                (None, 26, 26, 32)            320
## ____________________________________________________________________________
## max_pooling2d_3 (MaxPooling2D)   (None, 13, 13, 32)            0
## ____________________________________________________________________________
## conv2d_5 (Conv2D)                (None, 11, 11, 64)            18496
## ____________________________________________________________________________
## max_pooling2d_4 (MaxPooling2D)   (None, 5, 5, 64)              0
## ____________________________________________________________________________
## conv2d_6 (Conv2D)                (None, 3, 3, 64)              36928
## ____________________________________________________________________________
## flatten_2 (Flatten)              (None, 576)                   0
## ____________________________________________________________________________
## dense_3 (Dense)                  (None, 64)                    36928
## ____________________________________________________________________________
## dense_4 (Dense)                  (None, 10)                    650
## ============================================================================
## Total params: 93,322
## Trainable params: 93,322
## Non-trainable params: 0
## ____________________________________________________________________________
```

Training the convnet on MNIST images:

```
mnist <- dataset_mnist()
c(c(train_images, train_labels), c(test_images, test_labels)) %<-% mnist
train_images <- array_reshape(train_images, c(60000, 28, 28, 1))
train_images <- train_images / 255
test_images <- array_reshape(test_images, c(10000, 28, 28, 1))
test_images <- test_images / 255
train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)
```

```
model %>% compile(
  optimizer = "rmsprop",
```

```
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
model %>% fit(
  train_images, train_labels,
  epochs = 5, batch_size=64
)
```

Evaluate the model on the test data:

```
results <- model %>% evaluate(test_images, test_labels)
results
```

```
## $loss
## [1] 0.03167136
##
## $acc
## [1] 0.9902
```

Dealing with JPEG images

- Create image data generators

```
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory(
  train_dir,
  train_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)

validation_generator <- flow_images_from_directory(
  validation_dir,
  validation_datagen,
  target_size = c(150, 150),
  batch_size = 20,
  class_mode = "binary"
)
```

- Example of a sample from the generator

```
> batch <- generator_next(train_generator)
> str(batch)
List of 2
 $ : num [1:20, 1:150, 1:150, 1:3] 37 48 153 53 114 194 158 141 255 167 ...
 $ : num [1:20(1d)] 1 1 1 1 0 1 1 0 1 1 ...
```

- Training the model using a generator

```
history <- model %>% fit_generator(
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,
  validation_data = validation_generator,
  validation_steps = 50
)
```