APUNTES CURSO COMPUTER VISION

- Use modern deep-learning networks to build an **image classifier** with Keras
- Design your own **custom convnet** with reusable blocks
- Learn the fundamental ideas behind visual **feature extraction**
- Master the art of **transfer learning** to boost your models
- Utilize **data augmentation** to extend your dataset

Our goal is to learn how a neural network can "understand" a natural image well-enough to solve the same kinds of problems the human visual system can solve.

The neural networks that are best at this task are called **convolutional neural networks**

We will apply these ideas to the problem of **image classification**: given a picture, can we train a computer to tell us what it's a picture of? That's an image classifier!

The Convolutional Classifier

Consists of two parts: a **convolutional base** and a **dense head**.

The base is used to **extract the features** from an image.

The head is used to **determine the class** of the image. It is formed primarily of dense layers, but might include other layers like dropout.

The goal of the network during training is to learn two things:

1. which features to extract from an image (base),
2. which class goes with what features (head).

These days, convnets are rarely trained from scratch. More often, we **reuse the base of a pretrained model**. Reusing a pretrained model is a technique known as **transfer learning**.

Step 1 - Load Data

Step 2 - Define Pretrained Base¶

The most commonly used dataset for pretraining is ImageNet, a large dataset of many kind of natural images. Keras includes a variety models pretrained on ImageNet in its applications module. The pretrained model we'll use is called **VGG16**.

```
pretrained_base = tf.keras.models.load_model(
    '../input/cv-course-models/cv-course-models/vgg16-pretrained-base',
)
pretrained_base.trainable = False
```

Step 3 - Attach Head¶

Next, we attach the classifier head. For this example, we'll use a layer of hidden units (the first Dense layer) followed by a layer to transform the outputs to a probability score for class 1. The Flatten layer transforms the two dimensional outputs of the base into the one dimensional inputs needed by the head.

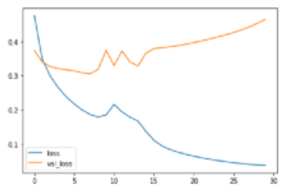
```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    pretrained_base,
    layers.Flatten(),
    layers.Dense(6, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])
```

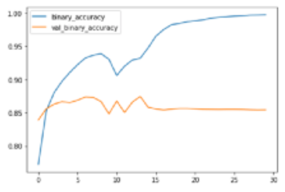
Step 4 - Train

```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)

history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=30,
    verbose=0,
)
```

When training a neural network, it's always a good idea to examine the loss and metric plots. The history object contains this information in a dictionary history.history. We can use Pandas to convert this dictionary to a dataframe and plot it with a built-in method.

```
import pandas as pd

history_frame = pd.DataFrame(history.history)
history_frame.loc[:, ['loss', 'val_loss']].plot()
history_frame.loc[:, ['binary_accuracy', ‘val_binary_accuracy’]].plot();
```





We saw that the VGG16 architecture was prone to overfitting this dataset

The first way you'll see is to use a base more appropriate to the dataset. The base this model comes from is called **InceptionV1**

The **InceptionV1** model pretrained on ImageNet is available in the TensorFlow Hub repository.

* Decide whether this base should be trainable or not.

pretrained_base.trainable = False

When doing transfer learning, it's generally not a good idea to retrain the entire base -- at least not without some care. The reason is that the random weights in the head will initially create large gradient updates, which propogate back into the base layers and destroy much of the pretraining.

The two most important types of layers that you'll usually find in the base of a convolutional image classifier. These are the **convolutional layer** with **ReLU activation**, and the **maximum pooling layer**.

The **feature extraction** performed by the base consists of **three basic operations**:

1. **Filter** an image for a particular feature (convolution)
2. **Detect** that feature within the filtered image (ReLU)
3. **Condense** the image to enhance the features (maximum pooling)

Filter with Convolution¶

A convolutional layer carries out the filtering step. You might define a convolutional layer in a Keras model something like this:

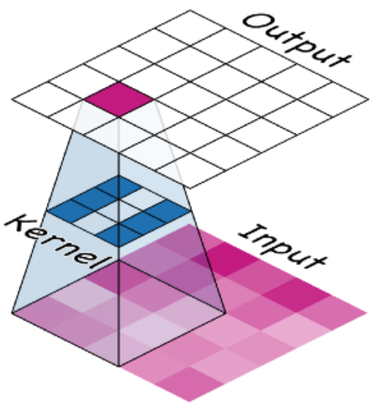
```
model = keras.Sequential([
    layers.Conv2D(filters=64, kernel_size=3), # activation is None
])
```

We can understand these parameters by looking at their relationship to the weights and activations of the layer

Weights¶

The **weights** a convnet learns during training are primarily contained in its convolutional layers. These weights we call **kernels**.

A kernel operates by scanning over an image and producing a weighted sum of pixel values.



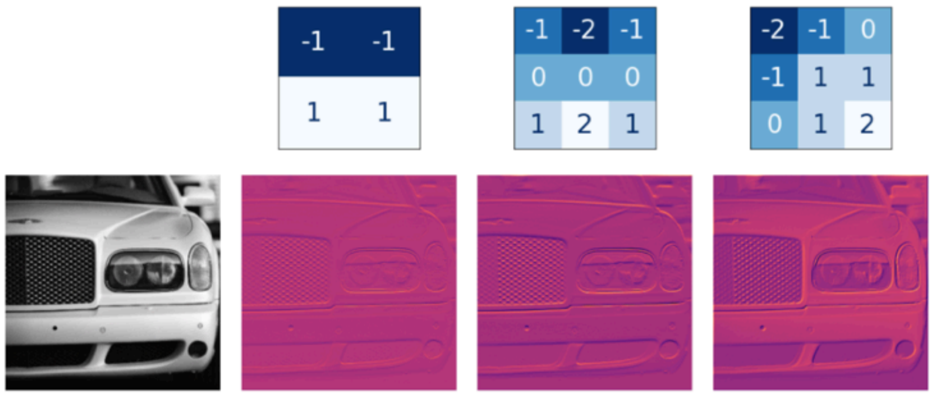
Kernels define how a convolutional layer is connected to the layer that follows. The kernel above will connect each neuron in the output to nine neurons in the input. By setting the dimensions of the kernels with kernel_size, you are telling the convnet how to form these connections. Most often, a kernel will have odd-numbered dimensions -- like kernel_size=(3, 3) or (5, 5)

The kernels in a convolutional layer determine what kinds of features it creates. During training, a convnet tries to learn what features it needs to solve the classification problem.

Activations¶

The **activations** in the network we call **feature maps**.

They are what result when we apply a filter to an image; they contain the visual features the kernel extracts.



Kernels and features.

From the pattern of numbers in the kernel, you can tell the kinds of feature maps it creates. Generally, what a convolution accentuates in its inputs will match the shape of the positive numbers in the kernel. The left and middle kernels above will both filter for horizontal shapes.

With the filters parameter, you tell the convolutional layer how many feature maps you want it to create as output.

Detect with ReLU¶

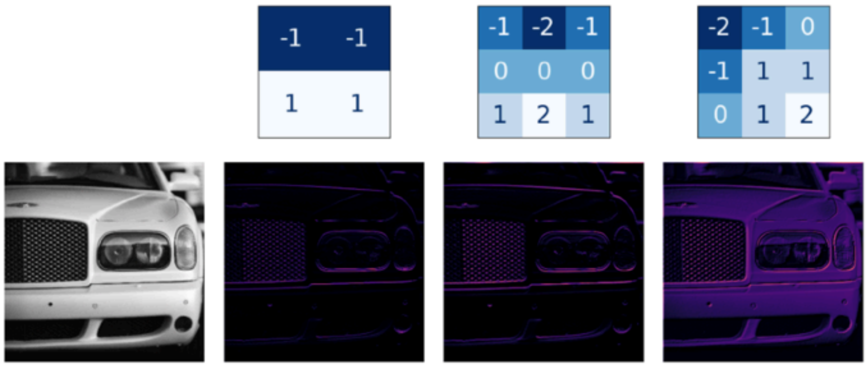
After filtering, the feature maps pass through the activation function.

The ReLU activation can be defined in its own Activation layer, but most often you'll just include it as the activation function of Conv2D.

```
model = keras.Sequential([
    layers.Conv2D(filters=64, kernel_size=3, activation='relu')
])
```

The ReLU activation says that negative values are not important and so sets them to 0.

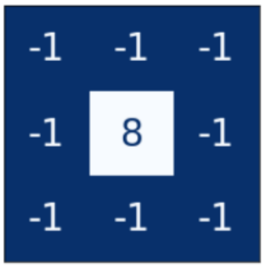
Notice how it succeeds at isolating the features.



* Like other activation functions, the ReLU function is **nonlinear**. The nonlinearity ensures features will combine in interesting ways as they move deeper into the network.

The kernel in this case is an "edge detection" kernel. You can define it with tf.constant. This creates a tensor of the sort TensorFlow uses.

```
import tensorflow as tf

kernel = tf.constant([
    [-1, -1, -1],
    [-1,  8, -1],
    [-1, -1, -1],
])

plt.figure(figsize=(3, 3))
show_kernel(kernel)
```



TensorFlow includes many common operations performed by neural networks in its tf.nn module. The two that we'll use are **conv2d and relu.**

```
image_filter = tf.nn.conv2d(
    input=image,
    filters=kernel,
    strides=1,
    padding='SAME',
)

plt.imshow(tf.squeeze(image_filter))
plt.show();
```

Next is the detection step with the ReLU function.

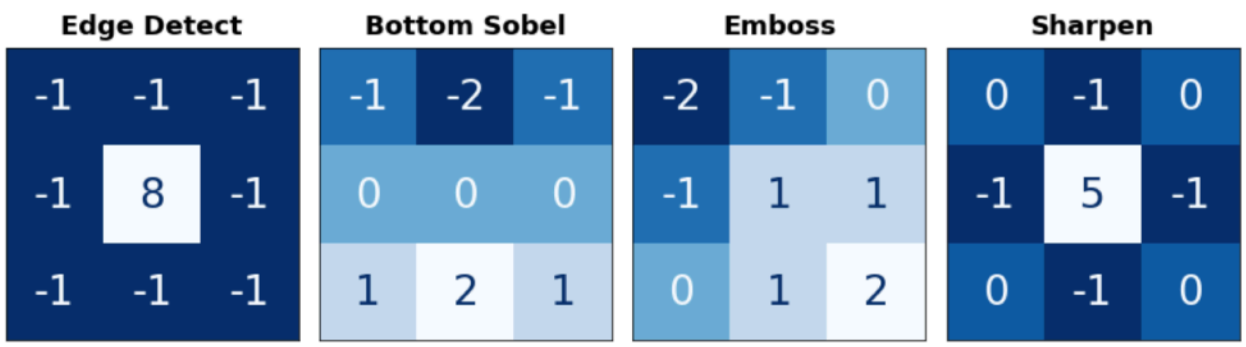
```
image_detect = tf.nn.relu(image_filter)

plt.imshow(tf.squeeze(image_detect))
plt.show();
```

And now we've created a feature map! Images like these are what the head uses to solve its classification problem.

The task of a convnet during training is to create kernels that can find those features.

We saw the first two steps a convnet uses to perform feature extraction: **filter** with Conv2D layers and **detect** with relu activation.

Standard kernels used in image processing.



Edge Detect

-1	-1	-1
-1	8	-1
-1	-1	-1

Bottom Sobel

-1	-2	-1
0	0	0
1	2	1

Emboss

-2	-1	0
-1	1	1
0	1	2

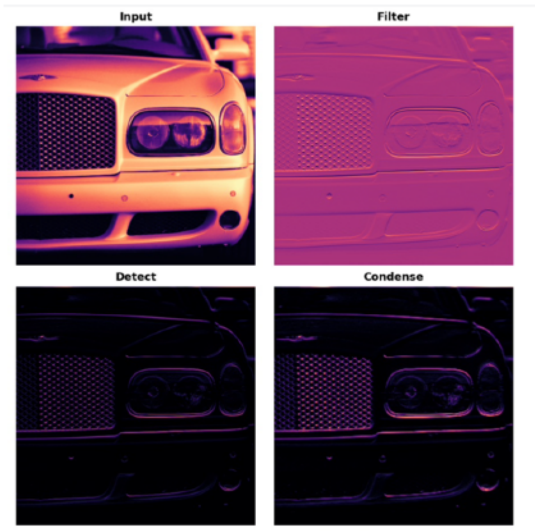
Sharpen

0	-1	0
-1	5	-1
0	-1	0

Third (and final) operation in this sequence: **condense** with **maximum pooling**, which in Keras is done by a **MaxPool2D** layer.

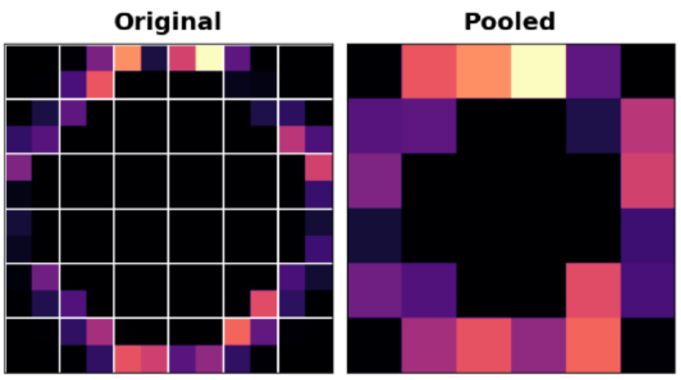
```
model = keras.Sequential([
    layers.Conv2D(filters=64, kernel_size=3),
    layers.MaxPool2D(pool_size=2),
])
```

A MaxPool2D layer is much like a Conv2D layer, except that it uses a simple maximum function instead of a kernel, with the **pool_size** parameter analogous to kernel_size. A MaxPool2D layer doesn't have any trainable weights like a convolutional layer does in its kernel, however.



Notice that after applying the ReLU function (**Detect**) the feature map ends up with a lot of "dead space," that is, large areas containing only 0's (the black areas in the image). Having to carry these 0 activations through the entire network would increase the size of the model without adding much useful information. Instead, we would like to condense the feature map to retain only the most useful part.

This in fact is what **maximum pooling** does. Max pooling takes a patch of activations in the original feature map and replaces them with the maximum activation in that patch.



- After the ReLU activation, it has the effect of "intensifying" features.

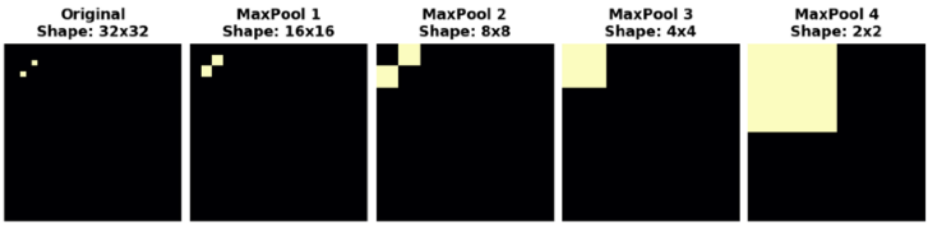
We'll use another one of the functions in tf.nn to apply the pooling step, tf.nn.pool. This is a Python function that does the same thing as the MaxPool2D layer.

```
image_condense = tf.nn.pool(
    input=image_detect,
    window_shape=(2, 2),
    pooling_type='MAX',
    strides=(2, 2),
    padding='SAME',
)
```

Translation Invariance¶

We called the zero-pixels "unimportant". Does this mean they carry no information at all? In fact, the zero-pixels carry positional information.

When MaxPool2D removes some of these pixels, it removes some of the positional information in the feature map. This gives a convnet a property called **translation invariance**. This means that a convnet with maximum pooling will tend not to distinguish features by their location in the image.



The two dots in the original image became indistinguishable after repeated pooling. In other words, pooling destroyed some of their positional information.

* Suppose you had made a small shift in a different direction -- what effect would you expect that have on the resulting image?

In the tutorial, we talked about how maximum pooling creates **translation invariance** over small distances. This means that we would expect small shifts to disappear after repeated maximum pooling. If you run the cell multiple times, you can see the resulting image is always the same; the pooling operation destroys those small translations.

1. filter with a **convolution** layer
2. detect with **ReLU** activation
3. condense with a **maximum pooling** layer

The convolution and pooling operations share a common feature: they are both performed over a **sliding window**. With convolution, this "window" is given by the dimensions of the kernel, the parameter kernel_size. With pooling, it is the pooling window, given by pool_size.

There are two additional parameters affecting both convolution and pooling layers -- these are the strides of the window and whether to use padding at the image edges. The strides parameter says how far the window should move at each step, and the padding parameter describes how we handle the pixels at the edges of the input.

```
model = keras.Sequential([
    layers.Conv2D(filters=64,
                  kernel_size=3,
                  strides=1,
                  padding='same',
                  activation='relu'),
     layers.MaxPool2D(pool_size=2,
                     strides=1,
                     padding='same')
])
```

Stride¶

The distance the window moves at each step is called the **stride**. We need to specify the stride in both dimensions of the image: one for moving left to right and one for moving top to bottom.

What effect does the stride have? Whenever the stride in either direction is greater than 1, the sliding window will skip over some of the pixels in the input at each step.

Because we want high-quality features to use for classification, convolutional layers will most often have strides=(1, 1). Increasing the stride means that we miss out on potentially valuble information in our summary. Maximum pooling layers, however, will almost always have stride values greater than 1, like (2, 2) or (3, 3), but not larger than the window itself.

Padding¶

When performing the sliding window computation, there is a question as to what to do at the boundaries of the input. Staying entirely inside the input image means the window will never sit squarely over these boundary pixels like it does for every other pixel in the input. Since we aren't treating all the pixels exactly the same, could there be a problem?

What the convolution does with these boundary values is determined by its padding parameter. In TensorFlow, you have two choices: either padding='same' or padding='valid'. There are trade-offs with each.

When we set padding='valid', the convolution window will stay entirely inside the input. The drawback is that the output shrinks (loses pixels), and shrinks more for larger kernels. This will limit the number of layers the network can contain, especially when inputs are small in size.

The alternative is to use padding='same'. The trick here is to **pad** the input with 0's around its borders, using just enough 0's to make the size of the output the same as the size of the input. This can have the effect however of diluting the influence of pixels at the borders.

* We've seen how convolutional networks can learn to extract features from (two-dimensional) images. It turns out that convnets can also learn to extract features from things like time-series (one-dimensional) and video (three-dimensional).

detrend = tf.constant([-1, 1], dtype=tf.float32)

average = tf.constant([0.2, 0.2, 0.2, 0.2, 0.2], dtype=tf.float32)

spencer = tf.constant([-3, -6, -5, 3, 21, 46, 67, 74, 67, 46, 32, 3, -5, -6, -3], dtype=tf.float32) / 320

The difference is just that a sliding window on a sequence only has one direction to travel -- left to right -- instead of the two directions on an image.

Custom Convnets

How to build a custom convnet composed of many **convolutional blocks** and capable of complex feature engineering.

We saw how convolutional networks perform **feature extraction** through three operations: **filter**, **detect**, and **condense**. A single round of feature extraction can only extract relatively simple features from an image, things like simple lines or contrasts. These are too simple to solve most classification problems. Instead, convnets will repeat this extraction over and over, so that the features become more complex and refined as they travel deeper into the network.

It does this by passing them through long chains of **convolutional blocks** which perform this extraction.

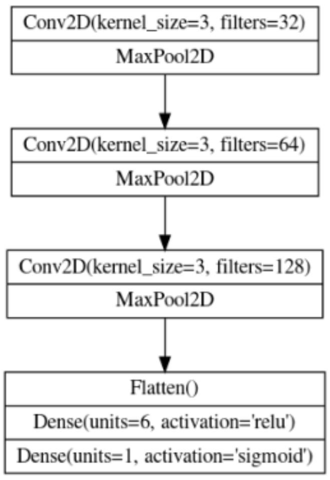
These convolutional blocks are stacks of Conv2D and MaxPool2D layers.

Each block represents a round of extraction, and by composing these blocks the convnet can combine and recombine the features produced, growing them and shaping them to better fit the problem at hand.

Step 1 - Load Data

Step 2 - Define Model

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([

        layers.Conv2D(filters=32, kernel_size=5, activation="relu",
padding='same',
              input_shape=[128, 128, 3]),
    layers.MaxPool2D(),

        layers.Conv2D(filters=64, kernel_size=3, activation="relu",
padding='same'),
    layers.MaxPool2D(),

       layers.Conv2D(filters=128, kernel_size=3, activation="relu",
padding='same'),
    layers.MaxPool2D(),

    layers.Flatten(),
    layers.Dense(units=6, activation="relu"),
    layers.Dense(units=1, activation="sigmoid"),
])
```


Conv2D(kernel_size=3, filters=32)
MaxPool2D
Conv2D(kernel_size=3, filters=64)
MaxPool2D
Conv2D(kernel_size=3, filters=128)
MaxPool2D
Flatten()
Dense(units=6, activation='relu')
Dense(units=1, activation='sigmoid')

Step 3 - Train

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(epsilon=0.01),
    loss='binary_crossentropy',
    metrics=['binary_accuracy']
)

history = model.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=40,
    verbose=0,
)
```

It might be that our simple network lacks the ability to extract sufficiently complex features. We could try improving the model either by adding more blocks or by adding convolutions to the blocks we have. We'll keep the three block structure, but i**ncrease the number of Conv2D layer in the second block to two, and in the third block to three.**

The learning curves for the model from the tutorial diverged fairly rapidly. This would indicate that it was prone to overfitting and in need of some regularization. The additional layer in our new model would make it even more prone to overfitting. However, adding some regularization with the Dropout layer helped prevent this. **These changes improved the validation accuracy of the model by several points.**

Data Augmentation

The best way to improve the performance of a machine learning model is to train it on more data. The more examples the model has to learn from, the better it will be able to recognize which differences in images matter and which do not. More data helps the model to generalize better.

One easy way of getting more data is to use the data you already have. If we can transform the images in our dataset in ways that preserve the class, we can teach our classifier to ignore those kinds of transformations. For instance, whether a car is facing left or right in a photo doesn't change the fact that it is a Car and not a Truck. So, if we **augment** our training data with flipped images, our classifier will learn that "left or right" is a difference it should ignore.

And that's the whole idea behind data augmentation: add in some extra fake data that looks reasonably like the real data and your classifier will improve.

Keras lets you augment your data in two ways. The first way is to include it in the data pipeline with a function like ImageDataGenerator.

The second way is to include it in the model definition by using Keras's **preprocessing layers**.

```
model = keras.Sequential([
    # Preprocessing
    preprocessing.RandomFlip('horizontal'), # flip left-to-right
    preprocessing.RandomContrast(0.5), # contrast change by up to 50%
    # Base
    pretrained_base,
    # Head
    layers.Flatten(),
    layers.Dense(6, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])
```

And now we'll start the training!

The learning curves for this model were able to stay closer together, and we achieved some modest improvement in validation loss and accuracy. This suggests that the dataset did indeed benefit from the augmentation.

DIFFERENT TYPES OF LAYERS FOR PREPROCESSING

```
# preprocessing.RandomContrast(factor=0.5),

# preprocessing.RandomFlip(mode='horizontal'), # meaning, left-to-right

# preprocessing.RandomFlip(mode='vertical'), # meaning, top-to-bottom

# preprocessing.RandomWidth(factor=0.15), # horizontal stretch

# preprocessing.RandomRotation(factor=0.20),

# preprocessing.RandomTranslation(height_factor=0.1, width_factor=0.1),
```