ConvNet 1

Part 1 Let's start with Convnet - Mnist dataset

We will use Convnet layers to classify MNIST digits, a task that you have analysed using a densely-connected network (test accuracy was 97.8%).

Dataset

Please reuse the code from the second exercise, which has already covered the MNIST database. Change the shape of the training testing images to tensor (60000, 28, 28, 1) and (10000, 28, 28, 1)).

Convnet architecture

A basic convnet architecture is a a stack of Conv2D and MaxPooling2D layers followed by the classification part.

A convnet takes as input tensors of shape (image_height, image_width, image_channels) (not including the batch dimension). In our case, we will configure our convnet to process inputs of size (28, 28, 1), which is the format of MNIST images.

- Pass the argument `input_shape=(28, 28, 1)` to our first layer
- Number of channels is controlled by the first argument passed to each Conv2D layer (here 32 or 64).
- The filters should have size 3×3.

Layers:

```
model.add(layers.Conv2D(nb of filters, (filter size, filter size),
activation='relu', input_shape=(image_height, image_width, image_channels)))
model.add(layers.MaxPooling2D((filter size, filter size)))
```

Please implement a ConvNet architecture consisting of 3 Conv2d layers with nb. of channels: 32→64→64. Between the Conv2d should be a MaxPolling2D layer with filter:size 2×2.

Display the architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
```

```
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (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
__________________________________
```

The next step would be to feed our last output tensor (of shape (3, 3, 64)) into a densely-connected **classifier network**: a stack of Dense layers. These classifiers process vectors, which are 1D, whereas our current output is a 3D tensor. So first, we will have to flatten our 3D outputs to 1D, and then add a few Dense layers on top.

```
model.add(layers.Flatten())
model.add(layers.Dense(nb. of channels -output of last Conv2d,
activation='Function'))
model.add(layers.Dense(nb. of classes, activation='Function'))
```

Only the last layer changes the activation function to `softmax`.

Model Summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (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
```

Training and evaluation

Since we are facing a multi-class classification problem the best loss function to use in this case is `categorical_crossentropy`. It measures the distance between two probability distributions: in our case, between the probability distribution output by our network, and the true distribution of the labels. By minimizing the distance between these two distributions, we train our network to output something as close as possible to the true labels.

Parameters:

- `rmsprop` optimizer
- `categorical_crossentropy` loss function
- `accuracy` metric

Fit the model for 5 epochs and batch size 64.

```
Epoch 5/5
60000/60000 [==============================] - 7s - loss: 0.0187 - acc:
0.9943
```

Verify your model on a test set. You should achieve test accuracy of about 0.99
Display the loss and accuracy curves on a plot.

Part 2 ConvNet for small dataset

Having to train an image classification model using very little data is a common situation, which you'll likely encounter in practice if you ever do computer vision in a professional context. A "few" samples can mean anywhere from a few hundred to a few tens of thousands of images. As a practical example, we'll focus on classifying images as dogs or cats, in a dataset containing 4,000 pictures of cats and dogs (2,000 cats, 2,000 dogs). We'll use 2,000 pictures for training – 1,000 for validation, and 1,000 for testing.

Dataset

Download the original dataset at https://www.kaggle.com/c/dogs-vs-cats/data (create a Kaggle account if you don't already have one).

If you use Google Colab and don't want to download the whole dataset on your computer, you can link a Kaggle account with Colab. For further information see link below:
https://www.kaggle.com/general/74235

This original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is ~800MB large (compressed). After downloading and uncompressing it, we will create a new dataset containing three subsets: a training set with 1000 samples of each class, a validation set with 500 samples of each class, and finally a test set with 500 samples of each class.

As we only need 4000 images, you do not need to download the whole dataset. You can download only train set (~500MB large) or copy the first 4000 images from someone else in the group.

After downloading the data, you have to arrange them correctly in the catalogues. Data should be split into separate catalogs for dogs and for cats. You can do it manually or in python (for example using `os` and `shutil` libraries.

```
main_directory/
...train/
......class_a/
.........a_image_1.jpg
.........a_image_2.jpg
......class_b/
.........b_image_1.jpg
.........b_image_2.jpg
...validation/
......class_a/
.........a_image_1.jpg
.........a_image_2.jpg
......class_b/
.........b_image_1.jpg
.........b_image_2.jpg
```

Please check the train/validation/test datasets. We now have 2000 training images, then 1000 validation images and 1000 test images (optional). In each split, there is the same number of samples from each class: this is a balanced binary classification problem, which means that classification accuracy will be an appropriate measure of success.

W colabie:

```
! gdown --id 18PsGch7egeczC4zLjYw3dXGjHyL-Kebw
! unzip /content/dataset_cats_dogs.zip
```

Bezpośrednio z Google Drive: https://drive.google.com/file/d/18PsGch7egeczC4zLjYw3dXGjHyL-Kebw/view?usp=sharing

Network architecture

You've already built a small convnet for MNIST in the previous exercise. You will reuse the same general structure: our convnet will be a stack of alternated Conv2D (with relu activation) and MaxPooling2D layers.

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have one more Conv2D + MaxPooling2D stage. This serves both to augment the capacity of the network and to further reduce the size of the feature maps so that they aren't overly large when we reach the Flatten layer. Here, since we start from inputs of size 150×150 (a somewhat arbitrary choice), we end up with feature maps of size 7×7 right before the Flatten layer. Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148×148 to 7×7). This is a pattern that you will see in almost all convnets. Since we are attacking a binary classification problem, we are ending the network with a single unit (a Dense layer of size 1) and a sigmoid activation. This unit will encode the probability that the network is looking at one class or the other.

Please create your network architecture based on the information above (4x Conv2d, 4x Maxpooling layers and then Flatten, 2 x Dense layer).

```
#TODO: Configure the model to look at the following summary
```

Model summary should look like this:

```
Layer (type)                 Output Shape              Param #
=================================================================
rescaling_1 (Rescaling)     (None, 150, 150, 3)       0

_________________________________________________________________
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896

_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0

_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496

_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0

_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856

_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0

_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584

_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0

_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0

_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776

_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
__________________________
```

Compilation step:

- RMSprop optimizer - with learning rate parameter(lr=1e-4)
- Binary crossentropy loss funtion
- Accuracy metric

```
#TODO: Compile the model
```

Data preprocessing

Data preprocessing steps include the following:

- Read the picture files.
- Decode the JPEG content to RBG grids of pixels.
- Convert these into floating-point tensors.
- Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

It may seem a bit daunting, but thankfully Keras has utilities to take care of these steps automatically. It has a module with image processing helper tools, located at `tensorflow.keras.preprocessing`. In particular, it contains the class ~~`ImageDataGenerator`~~ (**In older version of tensorflow only. It is now replaced by** `image_dataset_from_directory`) which allows to quickly set up Python generators that can automatically turn image files on disk into batches of pre-processed tensors. This is what we will use here. One of the useful properties of ImageDataGenerator is the ability to automatically label data based on which folder they are in. That is why, at the beginning of the exercise, we put dogs and cats into the appropriate folders.

Be careful with polish characters when specifying paths to directories.

```
from tensorflow.keras.preprocessing import  image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    train_dir,  # This is the target directory
    labels="inferred",
    label_mode="binary", # Since we use binary_crossentropy loss, we need
binary labels
    class_names=None,
    color_mode="rgb",
    batch_size=20,
    image_size=(150,150), # All images have different sizes and will be
resized to 150x150
    shuffle=True,
)

val_dataset = #TODO same as above
```

Let's fit our model to the dataset. We may pass a validation_data argument.

```
history = model.fit(train_dataset, epochs=30, validation_data=val_dataset)
```

Outcome analysis

Plot the loss and accuracy of the model over the training and validation data during training.

```
#TODO: plot the results
```

In these plots, you should see that the model is overfitting. Our training accuracy increases linearly over time, until it reaches nearly 100%, while our validation accuracy stalls at 70-72%. Our validation loss reaches its minimum after only five epochs then stalls, while the training loss keeps decreasing linearly until it reaches nearly 0.

Because we only have relatively few training samples (2000), overfitting is going to be our number one concern. There are many techniques that can help mitigate overfitting, such as reducing the number of epochs, adding dropout layers dropout, or weight decay (L2 regularization). We will look at them next weeks.