# ConvNet 2

## Part 1 Data Augmentation

Data augmentation takes the approach of generating more training data from existing training samples, by “augmenting” the samples via a number of random transformations that yield believable-looking images. The common-sense approach would be the manual creation of more images and applying small changes. However, it can be done easier and more efficiently in Keras, using ImageDataGenerator.

The goal is that at training time, our model would never see the exact same picture twice. It will change each image at the loading stage. This helps the model get exposed to more aspects of the data and generalize better. Below you can see an example. Where before we use only the rescale parameter, there are now several other random transformations:

```
datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

- `rotation_range` is a value in degrees (0-180), a range within which to randomly rotate pictures.
- `width_shift` and height_shift are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half of the images horizontally – relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

If you load images using `image_dataset_from_directory` you can't do the augmentation while loading. Instead, you have to do Random layers as the first layers of your model. See documentation below.

RandomRotation RandomWidth

```
#Create data augmentation as a separate model

data_augmentation = models.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(#TODO),
        layers.RandomZoom(#TODO),
        layers.RandomTranslation(#TODO),

    ]
)

# Nest data augmentation model as a layer

model = models.Sequential()
model.add(data_augmentation)
model.add(layers.Conv2D(...))
model.add(layers.MaxPooling2D(...))
model.add(models.layers.Dense(...))
```

In the next steps of the instruction you wont be able to easily visualise augmented data like in the code below. It would require seting parameter `data_augmentation.training = True`, then calling `data_augmentation.predict(…)` on a single image/batch.

Let's take a look at our augmented images:

```
# This is module with image preprocessing utilities
from keras.preprocessing import image
from matplotlib import pyplot as plt

fnames = [os.path.join(train_cats_dir, fname) for fname in
os.listdir(train_cats_dir)]

# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed
images.
# It will loop indefinitely, so we need to `break` the loop at some point!
```

```
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```

If we train a new network using this data augmentation configuration, our network will never see twice the same input. However, the inputs that it sees are still heavily intercorrelated, since they come from a small number of original images, so the results won't be that much better.

Train the network using data augmentation:

```
train_datagen = ImageDataGenerator(
        #TODO)

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # Set target directory:
        train_dir,
        # Resize images to 150x150 pixels:
        target_size=(150, 150),
        # Experiment with different batches:
        batch_size=20,
        # Labels needed for binary_crossentropy loss:
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

history = model.fit(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```

Note, that this time we train the network longer (for about 100 epochs) and it still increases. If training takes too much time on your machine, you can try lower numbers. After 100 epochs you can expect validation accuracy of about 80-82%, after 30 epochs about 77%. Plot the results of 30 epochs and compare them with those without augmentation.

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

# Part 2 Adding Dropout Layer

A dropout layer is a very simple, yet quite counter-intuitive method. What it does is basically cutting off half of the connection between the layers. You may ask yourself a question - *Why do I train the network so hard, only to randomly throw half of the information away?* Surprisingly, this works. In many situations network is overfitting to the data so much, that randomly cutting 20%, 30%, or even 50% of connection during training phase will be beneficial.

Therefore, to further combat overfitting, we will also add a Dropout layer to our Dogs and Cats example. Please configure the model with the same four convolutional layers and four max-pooling layers like last week, but this time add the Dropout layer right before the densely-connected classifier (with parameter 0.5).

```
#TODO: Implement the model
#...
model.add(layers.Dropout(0.5))
#...

#TODO: Compile the model
```

Train the model again, using ImageDataGenerator.

```
train_datagen = ImageDataGenerator(#TODO)
test_datagen = ImageDataGenerator(#TODO)

train_generator = train_datagen.flow_from_directory(#TODO)
validation_generator = test_datagen.flow_from_directory(#TODO)

history = model.fit(#TODO)
```

Wy should expect higher accuracy, about 80-82% compared to 77-80% in the previous lab. Plot the results and compare them.

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

If you want, you can also try training the model with a dropout layer, but without image augmentation, to see which one is better. Both methods should give better results than training without any of them, but worse than having them combined. You may, however, notice, that dropout is faster.

If you want, you can change the dropout parameter. Maybe we cut off too much? You can also try to experiment with putting the dropout layer in different places (maybe between convolutions), or even adding more dropout layers. There is hardly ever one universal way to implement a model. Make some modifications and comment on the results.