Exercise 7.5a Image segmentation with a U-Net architecture

In this exercise you train an image segmentation model from scratch on the Oxford Pets dataset. https://www.robots.ox.ac.uk/~vgg/data/pets/

Download the data

```
import os
if(not os.path.exists("images.tar.gz")):
  !wget https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
if(not os.path.exists("annotations.tar.gz")):
  !wget https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
!tar -xf images.tar.gz
!tar -xf annotations.tar.gz
```

```
--2024-11-15 19:20:14--  https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.ta
Resolving www.robots.ox.ac.uk (www.robots.ox.ac.uk)... 129.67.94.2
Connecting to www.robots.ox.ac.uk (www.robots.ox.ac.uk)|129.67.94.2|:443... connec
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://thor.robots.ox.ac.uk/pets/images.tar.gz [following]
--2024-11-15 19:20:14--  https://thor.robots.ox.ac.uk/pets/images.tar.gz
Resolving thor.robots.ox.ac.uk (thor.robots.ox.ac.uk)... 129.67.95.98
Connecting to thor.robots.ox.ac.uk (thor.robots.ox.ac.uk)|129.67.95.98|:443... con
HTTP request sent, awaiting response... 200 OK
Length: 791918971 (755M) [application/octet-stream]
Saving to: ‘images.tar.gz’

images.tar.gz       100%[===================>] 755.23M  67.2MB/s    in 9.0s

2024-11-15 19:20:24 (84.2 MB/s) - ‘images.tar.gz’ saved [791918971/791918971]

--2024-11-15 19:20:24--  https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotatio
Resolving www.robots.ox.ac.uk (www.robots.ox.ac.uk)... 129.67.94.2
Connecting to www.robots.ox.ac.uk (www.robots.ox.ac.uk)|129.67.94.2|:443... connec
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://thor.robots.ox.ac.uk/pets/annotations.tar.gz [following]
--2024-11-15 19:20:24--  https://thor.robots.ox.ac.uk/pets/annotations.tar.gz
Resolving thor.robots.ox.ac.uk (thor.robots.ox.ac.uk)... 129.67.95.98
Connecting to thor.robots.ox.ac.uk (thor.robots.ox.ac.uk)|129.67.95.98|:443... con
HTTP request sent, awaiting response... 200 OK
Length: 19173078 (18M) [application/octet-stream]
Saving to: ‘annotations.tar.gz’

annotations.tar.gz  100%[===================>]  18.28M   107MB/s    in 0.2s

2024-11-15 19:20:25 (107 MB/s) - ‘annotations.tar.gz’ saved [19173078/19173078]
```

Prepare paths of input images and target segmentation masks

you don't need to touch the code below. It creates lists of the filenames to the images and segmentation maps.

```
import os

input_dir = "images/"
target_dir = "annotations/trimaps/"
img_size = (160, 160) # all images get downscaled to this resolution
num_classes = 3
batch_size = 32

input_img_paths = sorted(
    [
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ]
)
target_img_paths = sorted(
    [
        os.path.join(target_dir, fname)
        for fname in os.listdir(target_dir)
        if fname.endswith(".png") and not fname.startswith(".")
    ]
)

print("Number of samples:", len(input_img_paths))

for input_path, target_path in zip(input_img_paths[:10], target_img_paths[:10]):
    print(input_path, "|", target_path)
```

```
Number of samples: 7390
images/Abyssinian_1.jpg | annotations/trimaps/Abyssinian_1.png
images/Abyssinian_10.jpg | annotations/trimaps/Abyssinian_10.png
images/Abyssinian_100.jpg | annotations/trimaps/Abyssinian_100.png
images/Abyssinian_101.jpg | annotations/trimaps/Abyssinian_101.png
images/Abyssinian_102.jpg | annotations/trimaps/Abyssinian_102.png
images/Abyssinian_103.jpg | annotations/trimaps/Abyssinian_103.png
images/Abyssinian_104.jpg | annotations/trimaps/Abyssinian_104.png
images/Abyssinian_105.jpg | annotations/trimaps/Abyssinian_105.png
images/Abyssinian_106.jpg | annotations/trimaps/Abyssinian_106.png
images/Abyssinian_107.jpg | annotations/trimaps/Abyssinian_107.png
```

What does one input image and corresponding segmentation mask look like?

The codeblock below shows how you can display the images and the target segmentation mask. To reduce the computing load, we will downscale all images to a size of 160x160 pixels as defined above. The goal of this task is to predict the segmentation mask as precisely as possible.

This example uses a few libraries to display and modify images (e.g. to rescale them to 160x160 pixels). The code below shows how these libraries can be used.

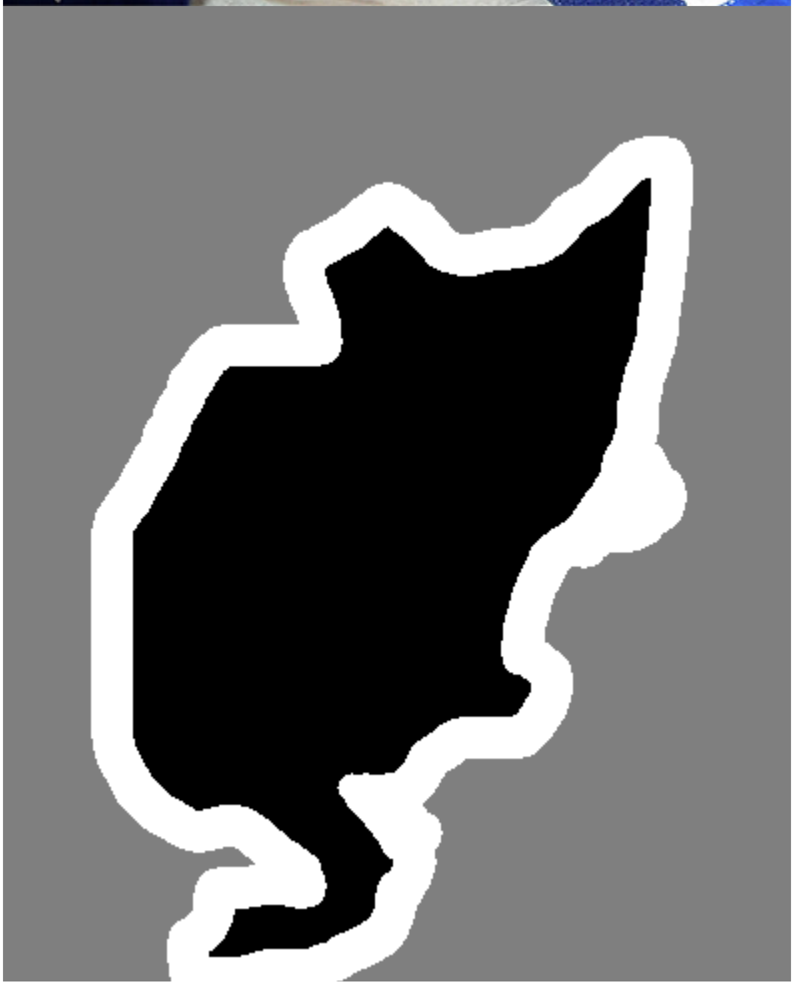
```
from IPython.display import Image, display
from tensorflow.keras.preprocessing.image import load_img
import PIL
from PIL import ImageOps
import numpy as np

# Display input image #2 and #7
for i_sample in [2,7, 3777]:
  print(f"image number {i_sample}")
  display(Image(filename=input_img_paths[i_sample]))

  # Display auto-contrast version of corresponding target (per-pixel categories)
  # all pixels have either the value 1, 2 or 3:
  # 1: Foreground 2:Background 3: Not classified
  img =load_img(target_img_paths[i_sample])
  display(PIL.ImageOps.autocontrast(img)) # to properly display the image, we set an a

  print(f"image number {i_sample} downscaled.")
  # the task is done on a downscaled version of the image
  # the downscaling can be achieved by just passing a `targer_size` argument to the `l
  display(load_img(input_img_paths[i_sample], target_size=img_size))
  img = load_img(target_img_paths[i_sample], target_size=img_size)
  display(PIL.ImageOps.autocontrast(img))
```

```
image number 2
```



Prepare `Sequence` class to load & vectorize batches of data

You do not need to touch the codeblock below. It is a helper class that returns batches of images and their target masks in the downscaled version. This is an alternative way to provide the training and validation data to the KERAS fit function. A large library of images are typically too big too keep them all in memory. Instead, a so-called "generator" function returns a new batch of images everytime it is called. This is implemented below. When the `__get_item(idx)` method is called, it loads all images from batch `idx` into memory and returns it as a NumPy array. The `__len__` method returns how many batches are in one epoch.

```
from tensorflow import keras
import numpy as np
from tensorflow.keras.preprocessing.image import load_img


class OxfordPets(keras.utils.Sequence):
    """Helper to iterate over the data (as Numpy arrays)."""
```

```
    Helper to iterate over the data (as Numpy arrays).

    def __init__(self, batch_size, img_size, input_img_paths, target_img_paths):
        self.batch_size = batch_size
        self.img_size = img_size
        self.input_img_paths = input_img_paths
        self.target_img_paths = target_img_paths

    def __len__(self):
        return len(self.target_img_paths) // self.batch_size

    def __getitem__(self, idx):
        """Returns tuple (input, target) correspond to batch #idx."""
        i = idx * self.batch_size
        batch_input_img_paths = self.input_img_paths[i : i + self.batch_size]
        batch_target_img_paths = self.target_img_paths[i : i + self.batch_size]
        x = np.zeros((self.batch_size,) + self.img_size + (3,), dtype="float32")
        for j, path in enumerate(batch_input_img_paths):
            img = load_img(path, target_size=self.img_size)
            x[j] = img
        y = np.zeros((self.batch_size,) + self.img_size + (1,), dtype="uint8")
        for j, path in enumerate(batch_target_img_paths):
            img = load_img(path, target_size=self.img_size, color_mode="grayscale")
            y[j] = np.expand_dims(img, 2) # one hot encoding:
            # Ground truth labels are 1, 2, 3. Subtract one to make them 0, 1, 2:
            # i.e. background is 0, foreground (the animal) is 1, and unclassified is
            y[j] -= 1
        return x, y
```

Prepare U-Net model

Hints:

- The final layer should have three feature maps with a softmax activation. This is because we want to predict the segmentation mask wich has three possible values: 0, 1, 2. The softmax activation works on each pixel, i.e., per pixel, the values of the feature maps add up to 1. Per pixel, the feature map with the highest propability indicates if we have "background", "foreground" or "unclassified". By using the `numpy.argmax` function, we can get back the integer for plotting the mask later (see `display_mask` function defined curther below).
- instead of using standard convolutions you can use `SeparableConv2D` to reduce the number of trainable parameters
- layers `x` and `x2` can be concacenated via `x = layers.concatenate([x, x2])`
- upsampling can be done with `x = layers.UpSampling2D(2)(x)`
- Always use `padding="same"` to kee the spatial dimension constant

```
print(img_size + (3,))
```

```
(160, 160, 3)
```

```
from tensorflow.keras import layers
# Free up RAM in case the model definition cells were run multiple times
keras.backend.clear_session()


# TODO: define a network with the UNet architecture below.
inputs = keras.Input(shape=img_size + (3,))

### [First half of the network: downsampling inputs] ###

# Entry block: start by adding a convolution layer.
x = layers.SeparableConv2D(32, 3, padding="same")(inputs)
x = layers.BatchNormalization()(x) # using batch normalization after the convolution b
x = layers.Activation("relu")(x)

#TODO: Implement a UNet architecture here
#Smaller segment (/4)
x2 = layers.SeparableConv2D(64, 3, padding="same", strides=(2,2))(x)
x2 = layers.BatchNormalization()(x2) # using batch normalization after the convolution
x2 = layers.Activation("relu")(x2)
x2=layers.MaxPooling2D((2,2), strides=(2,2))(x2)

#Even smaller (/2) segment
x3 = layers.SeparableConv2D(128, 2, padding="same")(x2)
x3 = layers.BatchNormalization()(x3) # using batch normalization after the convolution
x3 = layers.Activation("relu")(x3)
x3=layers.MaxPooling2D((2,2), strides=(2,2))(x3)

#perform another convolution in the same segment, this also reduces width (/2)
x3 = layers.SeparableConv2D(128, 2, padding="same", strides=(2,2))(x3)
x3 = layers.BatchNormalization()(x3) # using batch normalization after the convolution
x3 = layers.Activation("relu")(x3)

#Now upsample (*4)
x3=layers.UpSampling2D(size=(4,4))(x3)
#and merge with x2
x2=layers.concatenate([x2, x3])

#And upsample this (*4)
x2=layers.UpSampling2D(size=(4,4))(x2)
#And merge with original segment
x=layers.concatenate([x, x2])

# Add a per-pixel classification layer
outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

# Define the model
model = keras.Model(inputs, outputs)

model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #	Connected
input_layer (InputLayer)	(None, 160, 160, 3)	0	-
separable_conv2d (SeparableConv2D)	(None, 160, 160, 32)	155	input_laye
batch_normalization (BatchNormalization)	(None, 160, 160, 32)	128	separable_
activation (Activation)	(None, 160, 160, 32)	0	batch_norm
separable_conv2d_1 (SeparableConv2D)	(None, 80, 80, 64)	2,400	activation
batch_normalization_1 (BatchNormalization)	(None, 80, 80, 64)	256	separable_
activation_1 (Activation)	(None, 80, 80, 64)	0	batch_norm
max_pooling2d (MaxPooling2D)	(None, 40, 40, 64)	0	activation
separable_conv2d_2 (SeparableConv2D)	(None, 40, 40, 128)	8,576	max_poolin
batch_normalization_2 (BatchNormalization)	(None, 40, 40, 128)	512	separable_
activation_2 (Activation)	(None, 40, 40, 128)	0	batch_norm
max_pooling2d_1 (MaxPooling2D)	(None, 20, 20, 128)	0	activation
separable_conv2d_3 (SeparableConv2D)	(None, 10, 10, 128)	17,024	max_poolin
batch_normalization_3 (BatchNormalization)	(None, 10, 10, 128)	512	separable_
activation_3 (Activation)	(None, 10, 10, 128)	0	batch_norm
up_sampling2d (UpSampling2D)	(None, 40, 40, 128)	0	activation
concatenate (Concatenate)	(None, 40, 40, 192)	0	max_poolin up_samplin
up_sampling2d_1 (UpSampling2D)	(None, 160, 160, 192)	0	concatenat
concatenate_1 (Concatenate)	(None, 160, 160, 224)	0	activation up_samplin
conv2d (Conv2D)	(None, 160, 160, 3)	6,051	concatenat

Total params: 35,614 (139.12 KB)
Trainable params: 34,910 (136.37 KB)
Non-trainable params: 704 (2.75 KB)

Set aside a validation split

Set aside a validation split

```
import random

# Split our img paths into a training and a validation set
val_samples = 1000
random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_img_paths)
train_input_img_paths = input_img_paths[:-val_samples]
train_target_img_paths = target_img_paths[:-val_samples]
val_input_img_paths = input_img_paths[-val_samples:]
val_target_img_paths = target_img_paths[-val_samples:]

# Instantiate data Sequences for each split
train_gen = OxfordPets(
    batch_size, img_size, train_input_img_paths, train_target_img_paths
)
val_gen = OxfordPets(batch_size, img_size, val_input_img_paths, val_target_img_paths)
```

Train the model

```
# Configure the model for training.
# We use the "sparse" version of categorical_crossentropy
# because our target data is integers.
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer="adam",
    metrics=["accuracy"])

callbacks = [
    keras.callbacks.ModelCheckpoint("oxford_segmentation.keras", save_best_only=True)
]

# Train the model, doing validation at the end of each epoch.
epochs = 15
model.fit(train_gen, epochs=epochs, validation_data=val_gen, callbacks=callbacks)
```

```
Epoch 1/15
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_datase
  self._warn_if_super_not_called()
199/199 ━━━━━━━━━━━━━━━━━━━━ 56s 183ms/step - accuracy: 0.6341 - loss: 0.8317 - va
Epoch 2/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 36s 176ms/step - accuracy: 0.7049 - loss: 0.6912 - va
Epoch 3/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 168ms/step - accuracy: 0.7161 - loss: 0.6694 - va
Epoch 4/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 41s 167ms/step - accuracy: 0.7295 - loss: 0.6475 - va
Epoch 5/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 36s 176ms/step - accuracy: 0.7370 - loss: 0.6318 - va
Epoch 6/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 168ms/step - accuracy: 0.7424 - loss: 0.6222 - va
Epoch 7/15
```

```
199/199 ━━━━━━━━━━━━━━━━━━━━ 41s 167ms/step - accuracy: 0.7502 - loss: 0.6055 - va
Epoch 8/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 36s 175ms/step - accuracy: 0.7477 - loss: 0.6101 - va
Epoch 9/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 36s 174ms/step - accuracy: 0.7497 - loss: 0.6064 - va
Epoch 10/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 36s 176ms/step - accuracy: 0.7497 - loss: 0.6060 - va
Epoch 11/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 168ms/step - accuracy: 0.7574 - loss: 0.5903 - va
Epoch 12/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 35s 170ms/step - accuracy: 0.7608 - loss: 0.5841 - va
Epoch 13/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 34s 168ms/step - accuracy: 0.7604 - loss: 0.5843 - va
Epoch 14/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 41s 167ms/step - accuracy: 0.7608 - loss: 0.5846 - va
Epoch 15/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 42s 174ms/step - accuracy: 0.7621 - loss: 0.5832 - va
<keras.src.callbacks.history.History at 0x7a3a593a6650>
```

Visualize predictions

```
# Generate predictions for all images in the validation set
model.load_weights("oxford_segmentation.keras") # the last iteration might not be the
val_gen = OxfordPets(batch_size, img_size, val_input_img_paths, val_target_img_paths)
val_preds = model.predict(val_gen)


def display_mask(i):
    """Quick utility to display a model's prediction."""
    mask = np.argmax(val_preds[i], axis=-1) # find which feature map has the highest v
    mask = np.expand_dims(mask, axis=-1) # The image plotting library requires that th
    img = PIL.ImageOps.autocontrast(keras.preprocessing.image.array_to_img(mask))
    display(img)
```

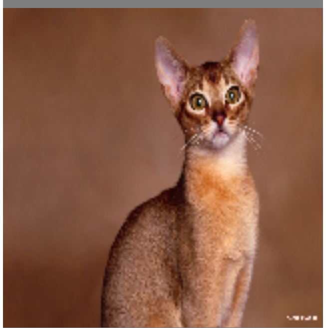
```
31/31 ━━━━━━━━━━━━━━━━━━━━ 7s 169ms/step
```

```
# Display results for validation image #10
for i in range(10):

  # Display input image
  display(load_img(val_input_img_paths[i], target_size=img_size))

  # Display ground-truth target mask
  img = PIL.ImageOps.autocontrast(load_img(val_target_img_paths[i], target_size=img_si
  display(img)

  # Display mask predicted by our model
  display_mask(i)  # Note that the model only sees inputs at 160x160.
```



Notes

Decent result, it has problem with other edges in the image. Missclasifications are often box-shaped with boxes 1/16th of the width and height of the image. This corresponds to the upsampling of the smallest segment. This has many more filters than the other segments and maybe I should compensate this by for example assigning weights to the filter, reducing the amount of filters before merging with the other segments or adding a strong regularization to the smallest segment.