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

In [1]:
```
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
```

```
--2025-11-27 11:43:06--  https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
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... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://thor.robots.ox.ac.uk/pets/images.tar.gz [following]
--2025-11-27 11:43:07--  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... connected.
HTTP request sent, awaiting response... 200 OK
Length: 791918971 (755M) [application/octet-stream]
Saving to: ‘images.tar.gz’

images.tar.gz       100%[===================>] 755.23M  27.2MB/s    in 30s

2025-11-27 11:43:37 (25.5 MB/s) - ‘images.tar.gz’ saved [791918971/791918971]

--2025-11-27 11:43:37--  https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
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... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://thor.robots.ox.ac.uk/pets/annotations.tar.gz [following]
--2025-11-27 11:43:38--  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... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19173078 (18M) [application/octet-stream]
Saving to: ‘annotations.tar.gz’

annotations.tar.gz  100%[===================>]  18.28M  12.7MB/s    in 1.4s

2025-11-27 11:43:40 (12.7 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.

In [2]:
```
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.

In [3]:

```
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 autocontr

  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 `load_img`
  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
```

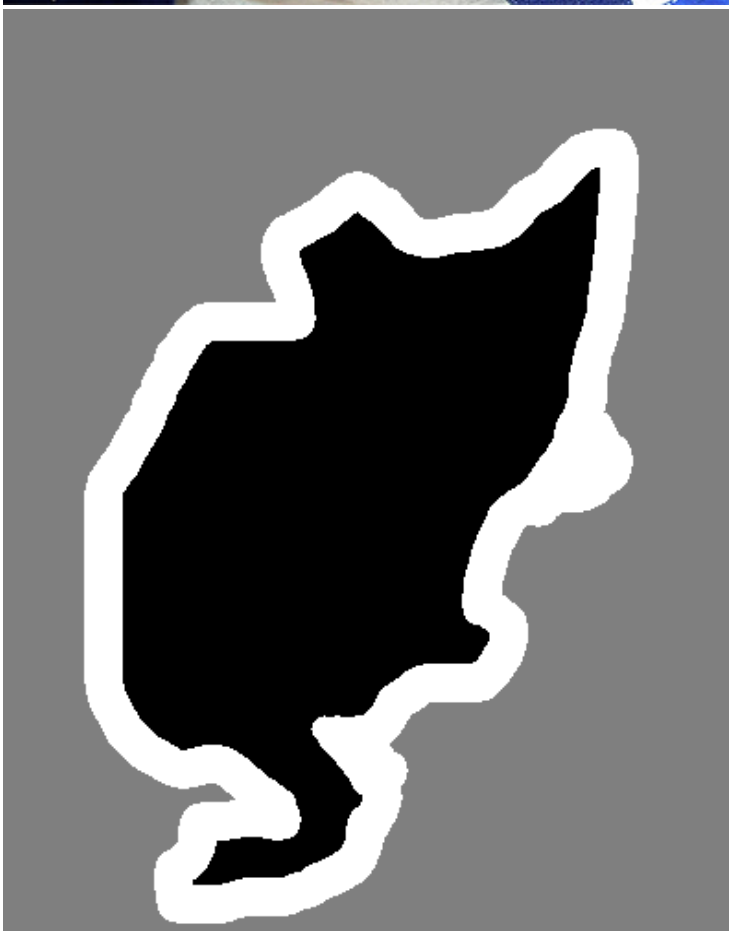
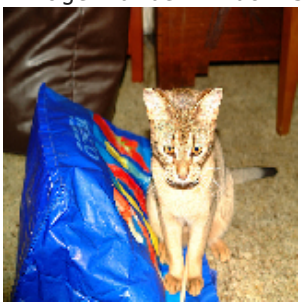


image number 2 downscaled.

image number 7

image number 7 downscaled.



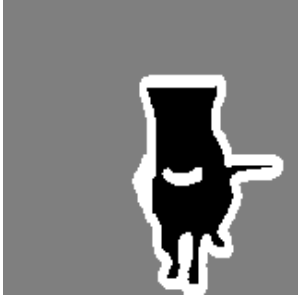


image number 3777

image number 3777 downscaled.

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.

In [4]:
```
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)."""

    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 3 (the c
            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

In [5]:
```
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,))
num_classes = 3

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

# Entry block: start by adding a convolution layer.
x = layers.Conv2D(32, 3, padding="same")(inputs)
x = layers.BatchNormalization()(x) # using batch normalization after the convolution but befor
x = layers.Activation("relu")(x)

#TODO: Implement a UNet architecture here

# Downsampling 1
x1 = layers.SeparableConv2D(64, 3, padding="same", activation="relu")(x)
x1 = layers.SeparableConv2D(64, 3, padding="same", activation="relu")(x1)
p1 = layers.MaxPooling2D(2)(x1)

# Downsampling 2
x2 = layers.SeparableConv2D(128, 3, padding="same", activation="relu")(p1)
x2 = layers.SeparableConv2D(128, 3, padding="same", activation="relu")(x2)
p2 = layers.MaxPooling2D(2)(x2)

# Bottleneck
b = layers.SeparableConv2D(256, 3, padding="same", activation="relu")(p2)
b = layers.SeparableConv2D(256, 3, padding="same", activation="relu")(b)

# Upsampling 1
x = layers.UpSampling2D(2)(b)
x = layers.concatenate([x, x2])
x = layers.SeparableConv2D(128, 3, padding="same", activation="relu")(x)
x = layers.SeparableConv2D(128, 3, padding="same", activation="relu")(x)

# Upsampling 2
x = layers.UpSampling2D(2)(x)
x = layers.concatenate([x, x1])
x = layers.SeparableConv2D(64, 3, padding="same", activation="relu")(x)
x = layers.SeparableConv2D(64, 3, padding="same", activation="relu")(x)

# 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 to
input_layer (InputLayer)	(None, 160, 160, 3)	0	–
conv2d (Conv2D)	(None, 160, 160, 32)	896	input_layer[0][0]
batch_normalization (BatchNormalizatio…	(None, 160, 160, 32)	128	conv2d[0][0]
activation (Activation)	(None, 160, 160, 32)	0	batch_normalizat…
separable_conv2d (SeparableConv2D)	(None, 160, 160, 64)	2,400	activation[0][0]
separable_conv2d_1 (SeparableConv2D)	(None, 160, 160, 64)	4,736	separable_conv2d…
max_pooling2d (MaxPooling2D)	(None, 80, 80, 64)	0	separable_conv2d…
separable_conv2d_2 (SeparableConv2D)	(None, 80, 80, 128)	8,896	max_pooling2d[0]…
separable_conv2d_3 (SeparableConv2D)	(None, 80, 80, 128)	17,664	separable_conv2d…
max_pooling2d_1 (MaxPooling2D)	(None, 40, 40, 128)	0	separable_conv2d…
separable_conv2d_4 (SeparableConv2D)	(None, 40, 40, 256)	34,176	max_pooling2d_1[…
separable_conv2d_5 (SeparableConv2D)	(None, 40, 40, 256)	68,096	separable_conv2d…
up_sampling2d (UpSampling2D)	(None, 80, 80, 256)	0	separable_conv2d…
concatenate (Concatenate)	(None, 80, 80, 384)	0	up_sampling2d[0]… separable_conv2d…
separable_conv2d_6 (SeparableConv2D)	(None, 80, 80, 128)	52,736	concatenate[0][0]
separable_conv2d_7 (SeparableConv2D)	(None, 80, 80, 128)	17,664	separable_conv2d…
up_sampling2d_1 (UpSampling2D)	(None, 160, 160, 128)	0	separable_conv2d…
concatenate_1 (Concatenate)	(None, 160, 160, 192)	0	up_sampling2d_1[… separable_conv2d…
separable_conv2d_8 (SeparableConv2D)	(None, 160, 160, 64)	14,080	concatenate_1[0]…
separable_conv2d_9 (SeparableConv2D)	(None, 160, 160, 64)	4,736	separable_conv2d…
conv2d_1 (Conv2D)	(None, 160, 160, 3)	1,731	separable_conv2d…

Total params: 227,939 (890.39 KB)

Trainable params: 227,875 (890.14 KB)

Non-trainable params: 64 (256.00 B)

Set aside a validation split

In [6]:
```
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

In [7]:
```
# Configure the model for training.
# We use the "sparse" version of categorical_crossentropy
# because our target data is integers.
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(learning_rate=0.001), loss="sparse_categorical_crossentropy")

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)
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:
121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constr
uctor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass t
hese arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
Epoch 1/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 113s 320ms/step - loss: 0.9619 - val_loss: 0.9555
Epoch 2/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 294ms/step - loss: 0.7787 - val_loss: 0.7042
Epoch 3/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 296ms/step - loss: 0.6758 - val_loss: 0.6720
Epoch 4/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 294ms/step - loss: 0.6078 - val_loss: 0.5998
Epoch 5/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 295ms/step - loss: 0.5685 - val_loss: 0.5436
Epoch 6/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 297ms/step - loss: 0.5332 - val_loss: 0.5569
Epoch 7/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 298ms/step - loss: 0.5046 - val_loss: 0.6015
Epoch 8/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 295ms/step - loss: 0.4825 - val_loss: 0.4789
Epoch 9/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 295ms/step - loss: 0.4776 - val_loss: 0.5111
Epoch 10/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 295ms/step - loss: 0.4538 - val_loss: 0.4883
Epoch 11/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 298ms/step - loss: 0.4387 - val_loss: 0.4393
Epoch 12/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 295ms/step - loss: 0.4343 - val_loss: 0.4241
Epoch 13/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 295ms/step - loss: 0.4180 - val_loss: 0.5357
Epoch 14/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 296ms/step - loss: 0.4192 - val_loss: 0.5386
Epoch 15/15
199/199 ━━━━━━━━━━━━━━━━━━━━ 59s 297ms/step - loss: 0.4176 - val_loss: 0.4421
```

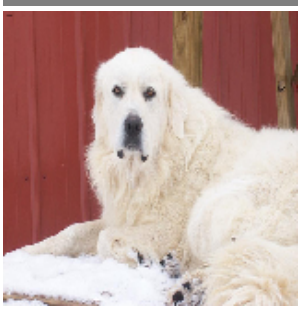
Out[7]: <keras.src.callbacks.history.History at 0x7d3c3a3580b0>

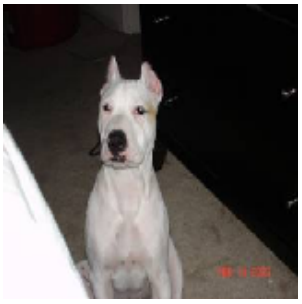
Visualize predictions

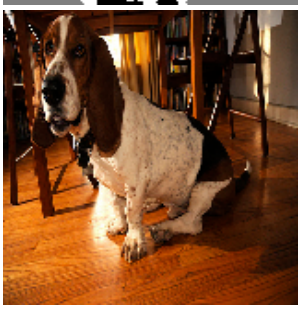
In [8]:
```
# Generate predictions for all images in the validation set
model.load_weights("oxford_segmentation.keras") # the last iteration might not be the best. So
val_gen = OxfordPets(batch_size, img_size, val_input_img_paths, val_target_img_paths)
val_preds = model.predict(val_gen)


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 value per
    mask = np.expand_dims(mask, axis=-1) # The image plotting library requires that the color
    img = PIL.ImageOps.autocontrast(keras.preprocessing.image.array_to_img(mask))
    display(img)
```

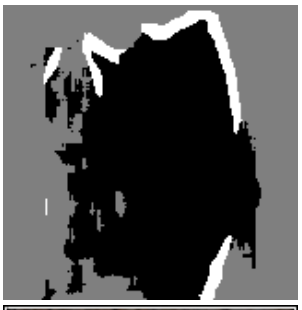
```
31/31 ━━━━━━━━━━━━━━━━━━━━ 8s 118ms/step
```

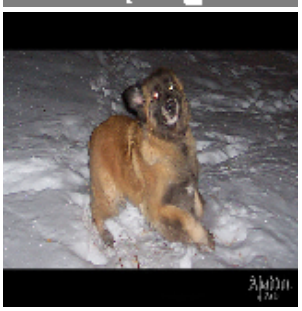
In [9]:
```
# Display results for validation image #10
for i in range(10):

  # 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_size))
  display(img)

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















Final comments

- The implemented U-Net uses an initial Conv2D block followed by SeparableConv2D layers for downsampling and upsampling, with skip connections to preserve spatial information. MaxPooling2D reduces dimensions in the encoder, UpSampling2D restores them in the decoder. The final softmax layer outputs per-pixel probabilities for the three classes, making the model efficient and suitable for segmentation.