Lecture_8_Image_segmentation

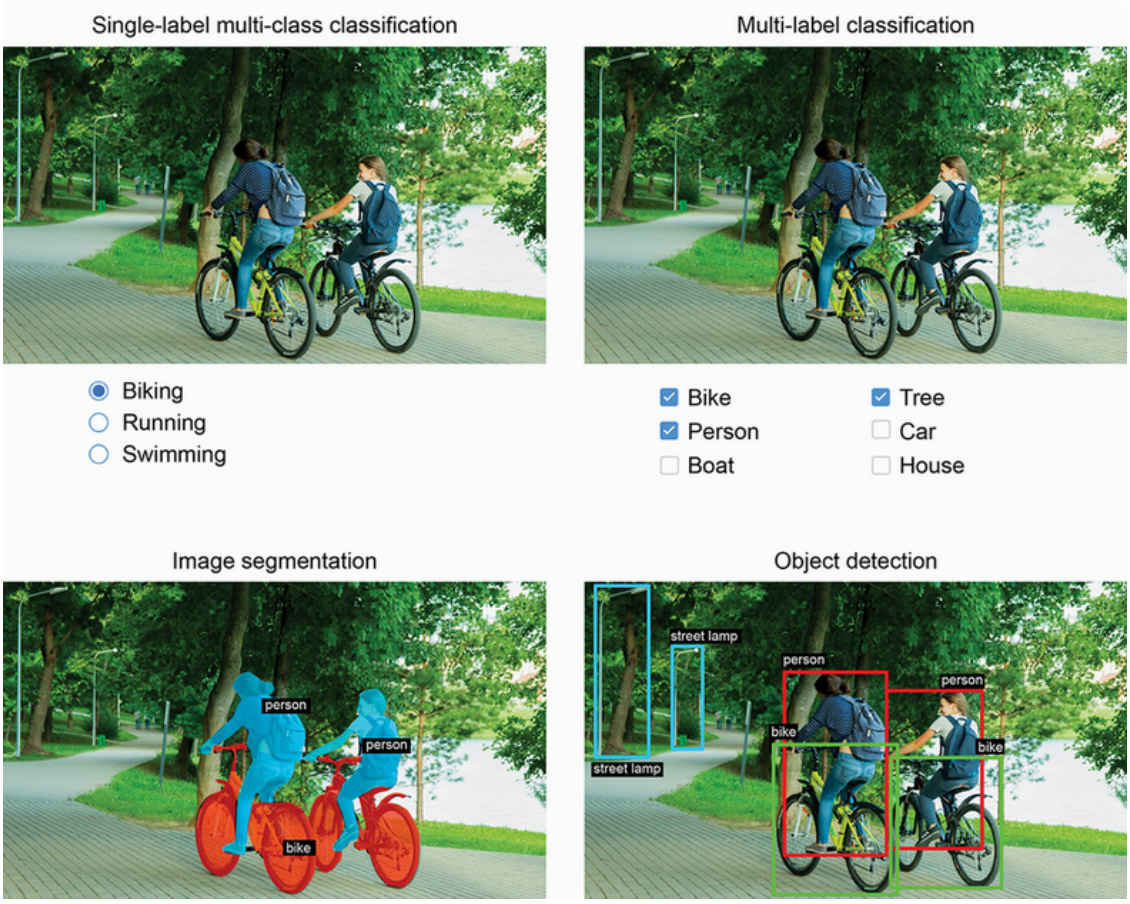
November 30, 2022

1 Advanced deep learning for computer vision. More on Functional API

1.1 Three essential computer vision tasks

So far, we've focused on image classification models: an image goes in, a label comes out. But image classification is only one of several possible applications of DL in computer vision. In general, there are three essential computer vision tasks we need to know about: - **Image classification** - The goal is to assign one or more labels to an image. It may be either *single-label classification* (an image can only be in one category, excluding the others), or *multi-label classification* (tagging all categories that an image belongs to). For example, when we search for a keyword on the Google Photos app, behind the scenes we're querying a very large multilabel classification model - one with over 20,000 different classes, trained on millions of images. - **Image segmentation** - The goal is to "segment" or "partition" an image into different areas, with each area usually representing a category. For example, when Zoom or Google Meet diplays a custom background behind you in a video call, it's using an image segmentation model to tell your face apart from what's behind it, at pixel precision. - **Object detection** - The goal is to draw rectangles (called *bounding boxes*) around objects of interest in an image, and associate each rectangle with a class. A self-driving car could use an object-detection model to monitor cars, pedestrians, and signs in view of its cameras, for instance.

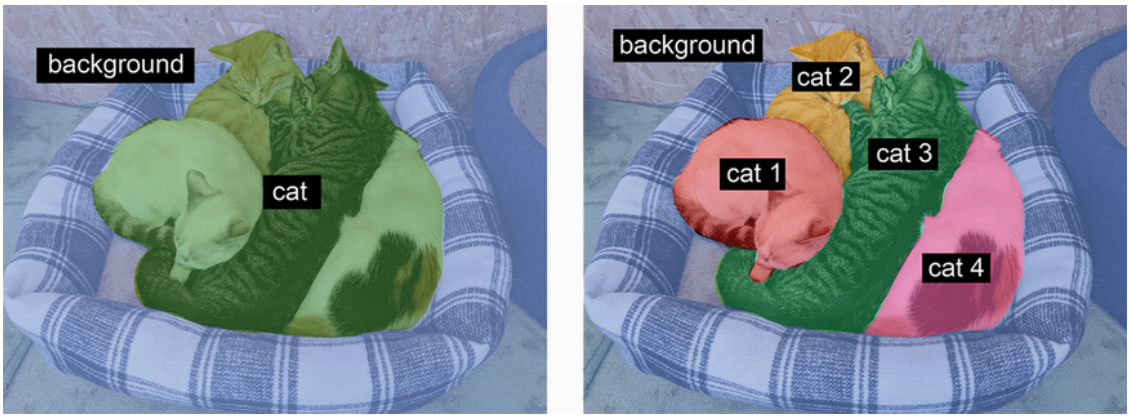
These tasks are illustrated in the figure below.



1.1.1 Example 1: Image segmentation

DL-based image segmentation is about using a model to assign a class to each pixel in an image, thus segmenting the image into different zones (such as "background" and "foreground," or "road," "car," and "sidewalk"). This general category of techniques can be used to power a considerable variety of valuable applications in image and video editing, autonomous driving, robotics, medical imaging, and so on.

There are two different flavors of image segmentation that we should know about: - **Semantic segmentation**, where each pixel is independently classified into a semantic category, like "cat." If there are two cats in the image, the corresponding pixels are all mapped to the same generic "cat" category (figure left below). - **Instance segmentation**, which seeks not only to classify image pixels by category, but also to parse out individual object instances. In an image with two cats in it, instance segmentation would treat "cat 1" and "cat 2" as two separate classes of pixels (figure right below).



In our example, we'll focus on the semantic segmentation: we'll be looking once again at images of cats and dogs, and this time we'll learn how to tell apart the main subject and its background.

We'll work with the Oxford-IIIT Pets dataset (www.robots.ox.ac.uk/~vgg/data/pets/), which contains 7,390 pictures of various breeds of cats and dogs, together with foreground-background segmentation masks for each picture. A segmentation mask is the image-segmentation equivalent of a label: it's an image the same size as the input image, with a single color channel where each integer value corresponds to the class of the corresponding pixel in the input image. In our case, the pixels of our segmentation masks can take one of three integer values: - 1 (foreground) - 2 (background) - 3 (contour)

[1]:
```
import os

input_dir = "Oxford_pets/images/"
target_dir = "Oxford_pets/annotations/trimaps/"

# Get sorted list of the names of input images
input_img_paths = sorted(
    [os.path.join(input_dir, name)
     for name in os.listdir(input_dir) if name.endswith(".jpg")])
# Get sorted list of the names of target images
target_paths = sorted(
    [os.path.join(target_dir, name)
     for name in os.listdir(target_dir)
     if name.endswith(".png") and not name.startswith(".")])
```

[2]:
```
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array
# load_img loads an image into PIL format
# img_to_array converts a PIL image instance to a Numpy array
# Python Imaging Library (PIL) is a free and open-source additional library for␣
 ↪Python
# that adds support for opening, manipulating, and saving many different image␣
 ↪file formats
```

```
plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

[2]: <matplotlib.image.AxesImage at 0x2ac16136130>

[3]:
```
def display_target(target_array):
    normalized_array = (target_array.astype("uint8") - 1) * 127
    plt.axis("off")
    plt.imshow(normalized_array[:, :, 0])

img = img_to_array(load_img(target_paths[9], color_mode="grayscale"))
display_target(img)
```

[4]:
```
import numpy as np
import random

img_size = (200, 200)
num_imgs = len(input_img_paths)

random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_paths)

def get_input_image_as_array(path):
    return img_to_array(load_img(path, target_size=img_size))

def get_target_image_as_array(path):
    img = img_to_array(
        load_img(path, target_size=img_size, color_mode="grayscale"))
    img = img.astype("uint8") - 1
    return img

input_imgs = np.zeros((num_imgs,) + img_size + (3,), dtype="float32")
targets = np.zeros((num_imgs,) + img_size + (1,), dtype="uint8")
# (num_imgs,) + img_size + (3,)  tuple concatenation (e.g., (2,3) + (4,) =␣
 ↪(2,3,4))
# Note: Tuple with one element is (4,), not (4). Comma is mandatory!
```

```
for i in range(num_imgs):
    input_imgs[i] = get_input_image_as_array(input_img_paths[i])
    targets[i] = get_target_image_as_array(target_paths[i])

num_val_samples = 1000
train_input_imgs = input_imgs[:-num_val_samples]
train_targets = targets[:-num_val_samples]
val_input_imgs = input_imgs[-num_val_samples:]
val_targets = targets[-num_val_samples:]
```

[5]:
```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Rescaling, Conv2D, Conv2DTranspose

def get_model(img_size, num_classes):
    inputs = Input(shape=img_size + (3,))
    x = Rescaling(1./255)(inputs)

    x = Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2D(64, 3, activation="relu", padding="same")(x)
    x = Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2D(128, 3, activation="relu", padding="same")(x)
    x = Conv2D(256, 3, strides=2, activation="relu", padding="same")(x)
    x = Conv2D(256, 3, activation="relu", padding="same")(x)

    x = Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = Conv2DTranspose(256, 3, activation="relu", padding="same", strides=2)(x)
    x = Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = Conv2DTranspose(128, 3, activation="relu", padding="same", strides=2)(x)
    x = Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = Conv2DTranspose(64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    return Model(inputs, outputs)

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 200, 200, 3)]     0

 rescaling (Rescaling)       (None, 200, 200, 3)       0

 conv2d (Conv2D)             (None, 100, 100, 64)      1792
```

```
conv2d_1 (Conv2D)            (None, 100, 100, 64)      36928

conv2d_2 (Conv2D)            (None, 50, 50, 128)       73856

conv2d_3 (Conv2D)            (None, 50, 50, 128)       147584

conv2d_4 (Conv2D)            (None, 25, 25, 256)       295168

conv2d_5 (Conv2D)            (None, 25, 25, 256)       590080

conv2d_transpose (Conv2DTra  (None, 25, 25, 256)       590080
nspose)

conv2d_transpose_1 (Conv2DT  (None, 50, 50, 256)       590080
ranspose)

conv2d_transpose_2 (Conv2DT  (None, 50, 50, 128)       295040
ranspose)

conv2d_transpose_3 (Conv2DT  (None, 100, 100, 128)     147584
ranspose)

conv2d_transpose_4 (Conv2DT  (None, 100, 100, 64)      73792
ranspose)

conv2d_transpose_5 (Conv2DT  (None, 200, 200, 64)      36928
ranspose)

conv2d_6 (Conv2D)            (None, 200, 200, 3)       1731

=================================================================
Total params: 2,880,643
Trainable params: 2,880,643
Non-trainable params: 0
_________________________________________________________________
```

The first half of the model resembles the kind of convnet we used for image classification: a stack of `Conv2D` layers, with gradually increasing filter sizes. We downsample our images three times by a factor of two each, ending up with activations of size `(25, 25, 256)`. The purpose of this first half is to encode the images into smaller feature maps, where each spatial location (or pixel) contains information about a large spatial chunk of the original image. It can be understood as a kind of compression.

One important difference between the first half of the model and the classification models we've seen before is the way we do downsampling: in our prevoius classification convnets, we used `MaxPooling2D` layers to downsample feature maps. Here, we downsample by adding strides to every other convolution layer. We do this because, in the case of image segmentation, we **care a lot about the spatial location of information in the image**, since we need to produce

per-pixel target masks as output of the model. When we do 2×2 max pooling, we are completely destroying location information within each pooling window: one scalar value per window is returned, with zero knowledge of which of the four locations in the windows the value came from. So while max pooling layers perform well for classification tasks, they would hurt us quite a bit for a segmentation task. Meanwhile, strided convolutions do a better job at downsampling feature maps while retaining location information.

The second half of the model is a stack of `Conv2DTranspose` layers. What are those? The output of the first half of the model is a feature map of shape `(25, 25, 256)`, but we want our final output to have the same shape as the target masks, `(200, 200, 3)`. Therefore, we need to apply a kind of inverse of the transformations we've applied so far - something that will upsample the feature maps instead of downsampling them. That's the purpose of the `Conv2DTranspose` layer: we can think of it as a kind of convolution layer that learns to upsample.

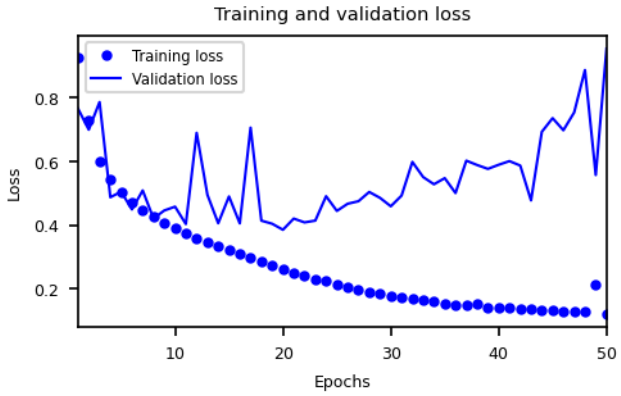
If we have an input of shape `(100, 100, 64)`, and we run it through `Conv2D(128, 3, strides=2, padding="same")`, we get an output of shape `(50, 50, 128)`. If we run this output through `Conv2DTranspose(64, 3, strides=2, padding="same")`, we get back an output of shape `(100, 100, 64)`, the same as the original! So after compressing our inputs into feature maps of shape `(25, 25, 256)` via a stack of `Conv2D` layers, we simply apply the corresponding sequence of `Conv2DTranspose` layers to get back to images of shape `(200, 200, 3)`.

We can now compile and fit our model:

[ ]:
```
from keras.callbacks import ModelCheckpoint

model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy")

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

history = model.fit(train_input_imgs, train_targets,
                    epochs=50,
                    callbacks=callbacks,
                    batch_size=16,
                    validation_data=(val_input_imgs, val_targets))
```

[8]:
```
from utility_functions import plot_helper

epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
val_loss = history.history["val_loss"]

plot_helper([epochs, epochs],
            [loss, val_loss],
            ["bo", "b"],
            ["Training loss", "Validation loss"],
            "Training and validation loss",
```

```
            "Epochs",
            "Loss")
```



[9]:
```
from keras.models import load_model
from tensorflow.keras.utils import array_to_img

model = load_model("models/oxford_segmentation.keras")

i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))

mask = model.predict(np.expand_dims(test_image, 0))[0]
# expand_dims inserts a new axis that will appear at the axis position in the␣
 ↪expanded array shape
# arr = np.array([[1,2],[3,4]]) - array of shape (2,2)
# expand_dims(arr, 0) returns an array of shape (1,2,2)

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

```
1/1 [==============================] - 1s 772ms/step
```

There are a couple of small artifacts in our predicted mask, caused by geometric shapes in the foreground and background. Nevertheless, our model appears to work nicely.

1.2 Back to the Keras functional API

Consider the following sequential API model:

[10]:
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([Conv2D(64, 3, activation='relu', input_shape=(28, 28, 3)),
                    MaxPooling2D(3),
                    Flatten(),
                    Dense(16, activation='relu'),
                    Dense(1, activation='sigmoid')])
```

The same convnet can be implemented using the following functional API:

[11]:
```
inputs = Input(shape=(28, 28, 3))
h = Conv2D(64, 3, activation='relu')(inputs)
h = MaxPooling2D(3)(h)
h = Flatten()(h)
h = Dense(16, activation='relu')(h)
outputs = Dense(1, activation='sigmoid')(h)
```

```
model = Model(inputs=inputs, outputs=outputs)
```

The **Sequential** model is easy to use, but its applicability is extremely limited: **it can only express models with a single input and a single output**, applying one layer after another in a sequential fashion. In practice, it's pretty common to encounter models with multiple inputs (say, an image and its metadata), multiple outputs (different things you want to predict about the data), or a nonlinear topology. In such cases, we'd build our model using the **Functional API**.

In the functional API, we instantiate the layers in the same way as we do for the sequential model, but we use those layer objects as functions that are called on an input tensor and that return an output tensor of the layer. So when does it make sense to use the functional API? For networks that can't easily be defined using the sequential API!

Let's take a look at an example.

[12]:
```
from tensorflow.keras.layers import Conv1D, AveragePooling1D, Concatenate

inputs = Input(shape=(32, 1))
h = Conv1D(16, 5, activation='relu')(inputs)
h = AveragePooling1D(3)(h)
h = Flatten()(h)
aux_inputs = Input(shape=(12,))     # Auxiliary input
h = Concatenate()([h, aux_inputs])
outputs = Dense(20, activation='sigmoid')(h)

model = Model(inputs=[inputs, aux_inputs], outputs=outputs)
```

This model has input layers, the main input layer and an auxiliary input layer. The auxiliary input is included in the model as an extra input to the final dense layer. It is one-dimensional, so it has the right shape to be fed into the dense layer.

Concatenation of the auxiliary input to the flattened tensor `h` is performed using the `Concatenate` layer, whose input is a list of two tensors. The tensors are concatenated along the last axis dimension, which creates the new tensor output `h`, which is further used as an input to the last `Dense` layer.

We have two inputs to our model, which is straightforward to implement using the functional API, we just pass a list of input tensors to the `inputs` keyword argument:

```
model = Model(inputs=[inputs, aux_inputs], outputs=outputs)
```

We can also do the same for outputs. Let's change our model to have an auxiliary output as well. For the auxiliary output, we'll take a single, real-value output from a dense layer called on the concatenated tensor from before. As with multiple inputs, we pass a list of output tensors to the `outputs` keyword argument.

[13]:
```
inputs = Input(shape=(32, 1))
h = Conv1D(16, 5, activation='relu')(inputs)
h = AveragePooling1D(3)(h)
h = Flatten()(h)
aux_inputs = Input(shape=(12,))
```

```
h = Concatenate()([h, aux_inputs])
outputs = Dense(20, activation='sigmoid')(h)
aux_outputs = Dense(1, activation='linear')(h)    # Auxiliary output

model = Model(inputs=[inputs, aux_inputs], outputs=[outputs, aux_outputs])
```

Now we need to think more carefully about how we compile and train our model. We've now got two outputs and these can be completely separate types of outputs like we have here. The main output is 20 units with a `sigmoid` activation. The auxiliary output is just a single real value. These two outputs most likely should have different loss functions associated with them.

[ ]:
```
model.compile(loss=['binary_crossentropy', 'mse'],
              loss_weights=[1, 0.4],
              metrics=['accuracy'])
history = model.fit([X_train, X_aux],
                    [y_train, y_aux],
                    validation_split=0.2,
                    epochs=20)
```

We just pass a list of loss functions to the `loss` keyword argument in the model dot compile method. These losses are in the same order as the order of outputs we defined when we created the model instance.

If we have two loss functions, we need to combine them somehow. We can only train our model using a gradient-based optimizer, if there is a single loss value that we're trying to optimize. That's what the new **`loss_weights`** keyword argument is doing here. These weights tell the model how to combine the loss functions. So here, the final loss is

```
binary_crossentropy + 0.4 MSE
```

In the `model.fit` method, we pass in a list of inputs and a list of outputs, both in the same order as the order we defined when we created the model object. The same thing goes for the evaluate and predict methods as well.

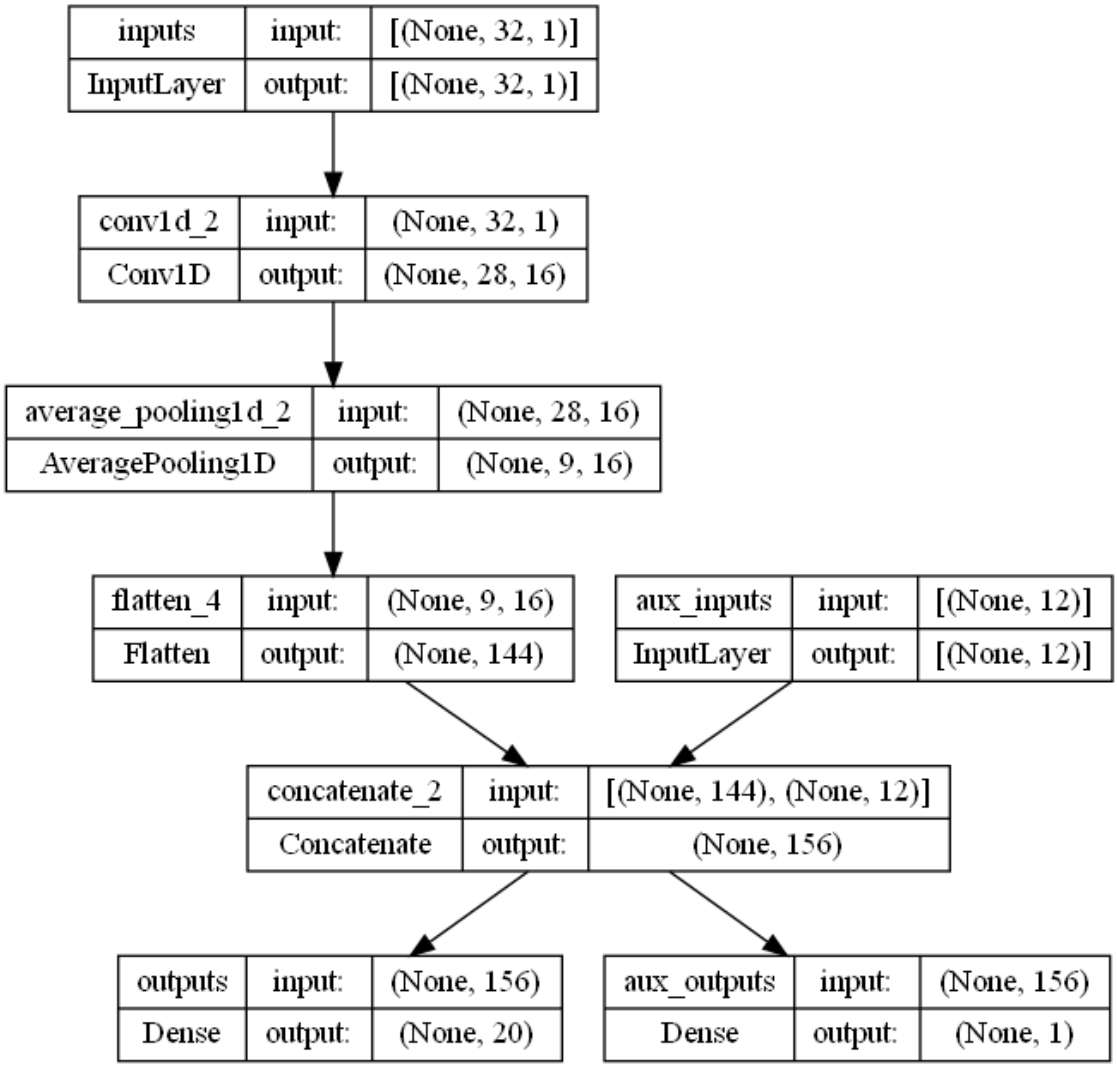
If we name the input and output layers, we could use an alternative way of compiling and fitting a model, given below:

[14]:
```
inputs = Input(shape=(32, 1), name='inputs')
h = Conv1D(16, 5, activation='relu')(inputs)
h = AveragePooling1D(3)(h)
h = Flatten()(h)
aux_inputs = Input(shape=(12,), name='aux_inputs')
h = Concatenate()([h, aux_inputs])
outputs = Dense(20, activation='sigmoid', name='outputs')(h)
aux_outputs = Dense(1, activation='linear', name='aux_outputs')(h)

model = Model(inputs=[inputs, aux_inputs], outputs=[outputs, aux_outputs])
```

[15]:
```
from tensorflow.keras.utils import plot_model
```

```
plot_model(model,
           'temp/multi_input_output_model.png',
           show_shapes=True)

# If you get message
# You must install pydot (`pip install pydot`) and install graphviz (...) for␣
 ↪plot_model to work
# run:
# pip install pydot
# conda install graphviz
```

[15]:



Compile the model and run training

[ ]:
```
model.compile(loss={'outputs': 'binary_crossentropy', 'aux_outputs': 'mse'},
              loss_weights={'outputs': 1, 'aux_outputs': 0.4},
              metrics=['accuracy'])
history = model.fit({'inputs': X_train, 'aux_inputs': X_aux},
                    {'outputs': y_train, 'aux_outputs': y_aux},
                    validation_split=0.2,
                    epochs=20)
```

When we compile the model, we associate loss functions with the right outputs by passing in a dictionary instead of a list. The dictionary maps the output layer name to the loss function. Similarly, the `loss_weights` keyword argument now also has a dictionary where the output layer name is mapped to the corresponding loss weight.

The same goes also for `model.fit`. The training inputs and outputs are passed in with a dictionary using the input and output layer names as keys.

1.2.1 Example 2: Diagnosis of two diseases of the urinary system

Load the acute inflammations dataset The `acute inflammations` was created by a medical expert as a data set to test the expert system, which will perform the presumptive diagnosis of two diseases of the urinary system: acute inflammations of urinary bladder and acute nephritises. You can find out more about the dataset here.

Attribute information:

Inputs: - Temperature of patient: 35C-42C - Occurrence of nausea: yes/no - Lumbar pain: yes/no - Urine pushing (continuous need for urination): yes/no - Micturition pains: yes/no - Burning of urethra, itch, swelling of urethra outlet: yes/no

Outputs: - decision 1: Inflammation of urinary bladder : yes/no - decision 2: Nephritis of renal pelvis origin : yes/no

Import the data The dataset required for this tutorial can be downloaded from the following link:

https://drive.google.com/open?id=1CDPQSqpI7OjNIgOERWaI-BlQMI6vjzb9

Load the data

[16]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Load the dataset
pd_dat = pd.read_csv('acute_inflammations/diagnosis.csv')
dataset = pd_dat.values

# Build train and test data splits
X_train, X_test, Y_train, Y_test = train_test_split(dataset[:,:6], dataset[:,6:
 ↪], test_size=0.33)
```

```
# Assign training and testing inputs/outputs
temp_train, nocc_train, lumbp_train, up_train, mict_train, bis_train = np.
 ↪transpose(X_train)
temp_test, nocc_test, lumbp_test, up_test, mict_test, bis_test = np.
 ↪transpose(X_test)

inflam_train, nephr_train = Y_train[:, 0], Y_train[:, 1]
inflam_test, nephr_test = Y_test[:, 0], Y_test[:, 1]
```

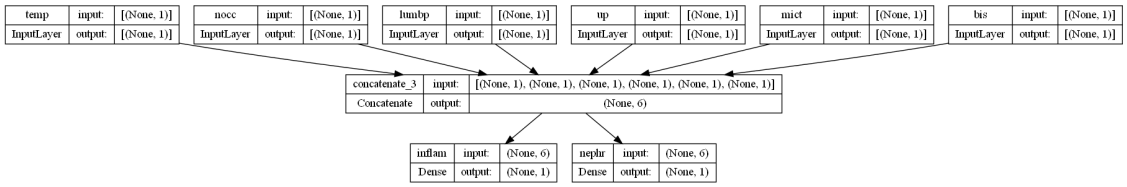
Build the model

[17]:
```
# Build the input layers
from tensorflow.keras.layers import Input, Concatenate, Dense
from tensorflow.keras.models import Model

inputs_shape = (1,)
temp = Input(shape=inputs_shape, name='temp')
nausea_occ = Input(shape=inputs_shape, name='nocc')
lumb_pain = Input(shape=inputs_shape, name='lumbp')
urine_push = Input(shape=inputs_shape, name='up')
mict_pain = Input(shape=inputs_shape, name='mict')
burn_itch_swell = Input(shape=inputs_shape, name='bis')

# Create a list of all the inputs
inputs_list = [temp,
               nausea_occ,
               lumb_pain,
               urine_push,
               mict_pain,
               burn_itch_swell]

# Merge all input features into a single large vector
x = Concatenate()(inputs_list)

# Use a logistic regression classifier for disease prediction
inflammation_pred = Dense(1, activation='sigmoid', name='inflam')(x)
neprhitis_pred = Dense(1, activation='sigmoid', name='nephr')(x)

# Create a list of all the outputs
outputs_list = [inflammation_pred, neprhitis_pred]

# Create the model object
model_diagnosis = Model(inputs=inputs_list, outputs=outputs_list)
```

Plot the model

[18]:
```
# Display the multiple input/output model
from tensorflow.keras.utils import plot_model

plot_model(model_diagnosis,
           'temp/urin_disease_diagnosis_model.png',
           show_shapes=True)
```

[18]:



Compile the model

[19]:
```
from tensorflow.keras.optimizers import RMSprop

# Compile the model
model_diagnosis.compile(optimizer=RMSprop(2e-3),
                        loss={'inflam': 'binary_crossentropy',
                              'nephr': 'binary_crossentropy'},    # or␣
 ↪loss=['binary_crossentropy', 'binary_crossentropy']
                        metrics=['acc'],
                        loss_weights = [1., 0.2])
```

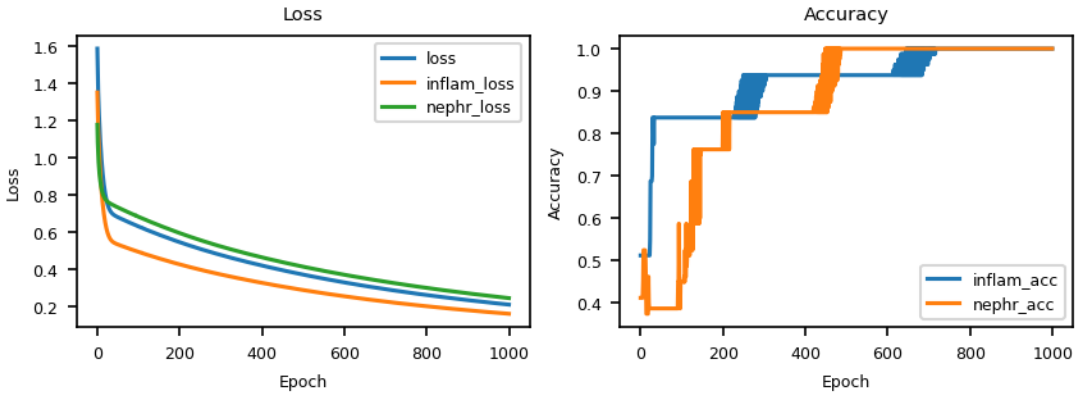
Fit the model

[20]:
```
# Define training inputs and outputs
inputs_train = {'temp': temp_train, 'nocc': nocc_train, 'lumbp': lumbp_train,
                'up': up_train, 'mict': mict_train, 'bis': bis_train}
outputs_train = {'inflam': inflam_train, 'nephr': nephr_train}

# Alternatively
# inputs_train = [temp_train, nocc_train, lumbp_train, up_train, mict_train,␣
 ↪bis_train]
# outputs_train = [inflam_train, nephr_train]
```

[21]:
```
# Train the model
history = model_diagnosis.fit(inputs_train,
                              outputs_train,
                              epochs=1000,
                              batch_size=128,
                              verbose=False)
```

Plot the learning curves

[22]:
```
# Plot the training accuracy
acc_keys = ['inflam_acc', 'nephr_acc']
loss_keys = ['loss', 'inflam_loss', 'nephr_loss']

plt.rc('axes', titlesize=7)  # fontsize of the axes title
plt.rc('axes', labelsize=6)  # fontsize of the x and y labels
plt.rc('xtick', labelsize=6)  # fontsize of the tick labels
plt.rc('ytick', labelsize=6)  # fontsize of the tick labels

plt.figure(figsize=(17 / 2.54, 5 / 2.54), dpi=150)

for k, v in history.history.items():
    if k in loss_keys:
        plt.subplot(1,2,1)
        plt.plot(v)
    else:
        plt.subplot(1,2,2)
        plt.plot(v)

plt.subplot(1,2,1)
plt.title('Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loss_keys, loc='best', fontsize=6)

plt.subplot(1,2,2)
plt.title('Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(acc_keys, loc='best', fontsize=6)

plt.show()
```



Evaluate the model

[23]:
```
model_diagnosis.evaluate([temp_test, nocc_test, lumbp_test, up_test, mict_test,␣
↪bis_test],
                         [inflam_test, nephr_test])
```

```
2/2 [==============================] - 0s 4ms/step - loss: 0.2389 - inflam_loss:
0.1939 - nephr_loss: 0.2251 - inflam_acc: 1.0000 - nephr_acc: 1.0000
```

[23]:
```
[0.2388899326324463, 0.19386401772499084, 0.2251296043395996, 1.0, 1.0]
```

1.2.2 Example 3: Do two MNIST images represent the same digit or not?

[24]:
```
from tensorflow.keras.datasets import mnist
import numpy as np

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype("float32") / 255
```

Preparing the dataset

Since our model will have two inputs, both MNIST images, we'll create two (shuffled) copies of both training and testing datasets.

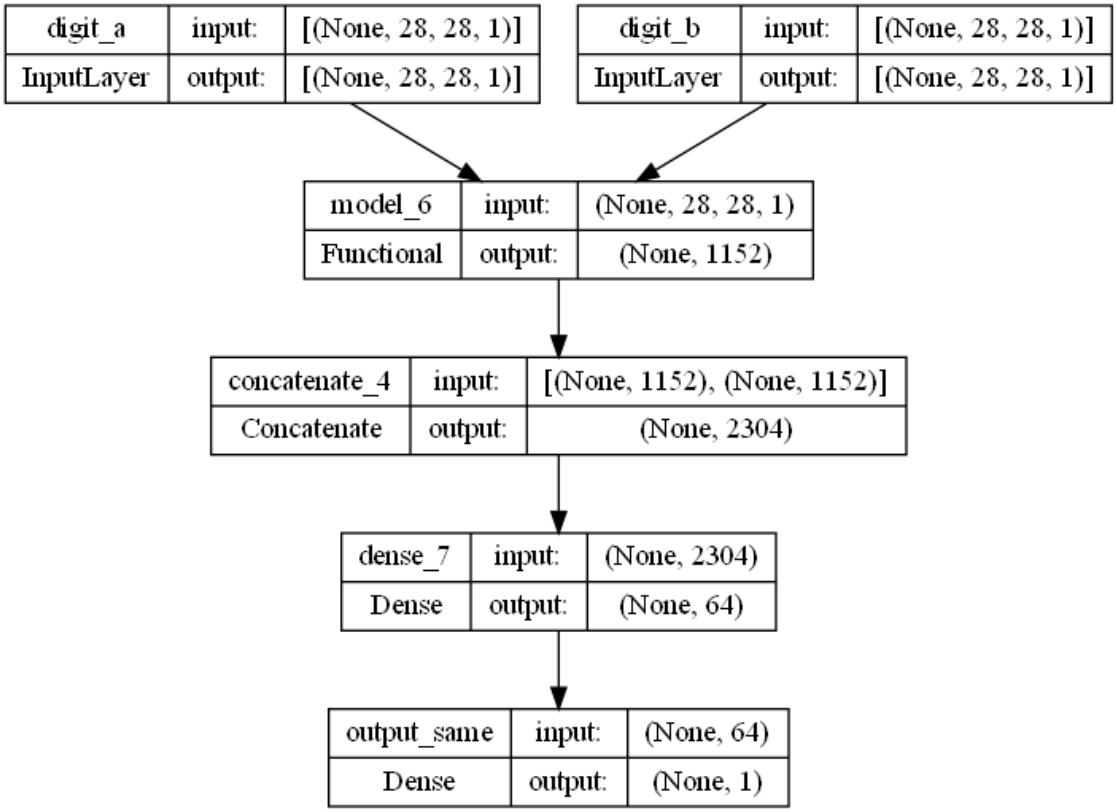
Output labels will be **0** (two MNIST images DO NOT represent the same digit) or **1** (two MNIST images represent the same digit).

[25]:
```
random_perm_a = np.random.permutation(60000)
train_images_a = train_images.copy()[random_perm_a]
train_labels_a = train_labels.copy()[random_perm_a]

random_perm_b = np.random.permutation(60000)
train_images_b = train_images.copy()[random_perm_b]
train_labels_b = train_labels.copy()[random_perm_b]

train_output = (train_labels_a == train_labels_b).astype('int')

random_perm_a = np.random.permutation(10000)
test_images_a = test_images.copy()[random_perm_a]
test_labels_a = test_labels.copy()[random_perm_a]

random_perm_b = np.random.permutation(10000)
test_images_b = test_images.copy()[random_perm_b]
test_labels_b = test_labels.copy()[random_perm_b]

test_output = (test_labels_a == test_labels_b).astype('int')
```

[26]:
```
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten,␣
 ↪Concatenate
from keras.models import Model

digit_input = Input(shape=(28, 28, 1))
x = Conv2D(32, 3, activation="relu")(digit_input)
x = MaxPooling2D(2)(x)
x = Conv2D(64, 3, activation="relu")(x)
x = MaxPooling2D(2)(x)
x = Conv2D(128, 3, activation="relu")(x)
out = Flatten()(x)

conv_model = Model(digit_input, out)

digit_a = Input(shape=(28, 28, 1), name='digit_a')
digit_b = Input(shape=(28, 28, 1), name='digit_b')

# The convolution part of the model will be shared
out_a = conv_model(digit_a)
out_b = conv_model(digit_b)

concatenated = Concatenate()([out_a, out_b])
last_hidden = Dense(64, activation='relu')(concatenated)
out = Dense(1, activation='sigmoid', name='output_same')(last_hidden)

model_digits = Model([digit_a, digit_b], out)
```

[27]:
```
from tensorflow.keras.utils import plot_model

plot_model(model_digits,
           'temp/MNIST_digits_same_model.png',
           show_shapes=True)
```

[27]:



[30]:
```
from keras.callbacks import ModelCheckpoint

callbacks = [
    ModelCheckpoint(
        filepath="models/mnist_digits_same.keras",
        save_best_only=True,
        monitor="val_loss")
]

model_digits.compile(optimizer="rmsprop",
                     loss='binary_crossentropy',
                     metrics=['accuracy'])
history = model_digits.fit({'digit_a': train_images_a, 'digit_b':␣
 ↪train_images_b},
                           train_output,
                           validation_split=0.2,
                           batch_size=128,
                           epochs=30,
                           callbacks=callbacks,
                           verbose=False)
```

[31]:
```
from keras.models import load_model

best_model_digits = load_model("models/mnist_digits_same.keras")
test_loss, test_acc = best_model_digits.evaluate({'digit_a': test_images_a,␣
 ↪'digit_b': test_images_b},
                                                  test_output)
print(f"Test accuracy: {test_acc:.3f}")
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0680 -
accuracy: 0.9905
Test accuracy: 0.990
```