UNIVERSITÀ DELLA CALABRIA

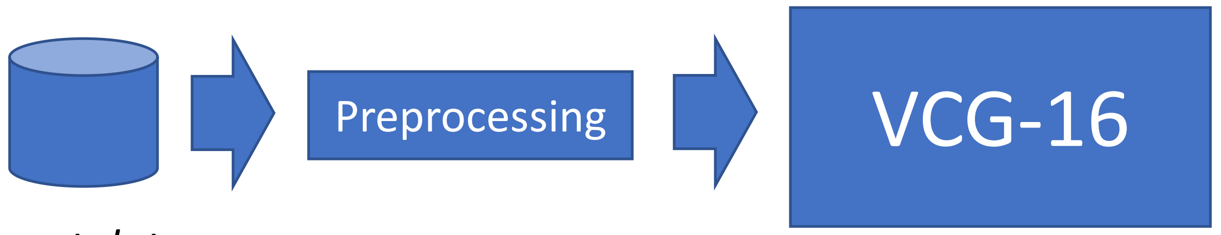
Pre-Trained ConvNets

VCG-16

- The VCG-16 network is a Deep ConvNet with 16 layers
  - The model has been trained on the ImageNet ILSVRC-2012 dataset, which includes images of 1,000 classes, and is split into three sets: training (1.3 million images), validation (50,000 images), and testing (100,000 images). Each image is (224×224) on 3 channels.
- The network (as well as others) is available in Keras
  - Building image-recognition application is rather easy

Tre-trained Networks

- VCG-16, as well as many other networks, is available in Keras
  - Re-using these networks requires some pre-processing on the input data



Input data

Input to VCG-16 = images 24x24 on 3 channels

VCG-16 Documentation

```
tf.keras.applications.VGG16(
    include_top=True,
    weights="imagenet",
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
)
```

Arguments

- **include_top**: whether to include the 3 fully-connected layers at the top of the network.
- **weights**: one of `None` (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded.
- **input_tensor**: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- **input_shape**: optional shape tuple, only to be specified if `include_top` is False (otherwise the input shape has to be `(224, 224, 3)` (with `channels_last` data format) or `(3, 224, 224)` (with `channels_first` data format). It should have exactly 3 input channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
- **pooling**: Optional pooling mode for feature extraction when `include_top` is `False`. - `None` means that the output of the model will be the 4D tensor output of the last convolutional block. - `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. - `max` means that global max pooling will be applied.
- **classes**: optional number of classes to classify images into, only to be specified if `include_top` is True, and if no `weights` argument is specified.
- **classifier_activation**: A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=True`. Set `classifier_activation=None` to return the logits of the "top" layer.

Example (1/4)

```
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
import matplotlib.pyplot as plt
import numpy as np
import cv2

# prebuild model with pre-trained weights on imagenet
model = VGG16(weights='imagenet', include_top=True)
model.compile(optimizer='sgd', loss='categorical_crossentropy')
```

Example (2/4)

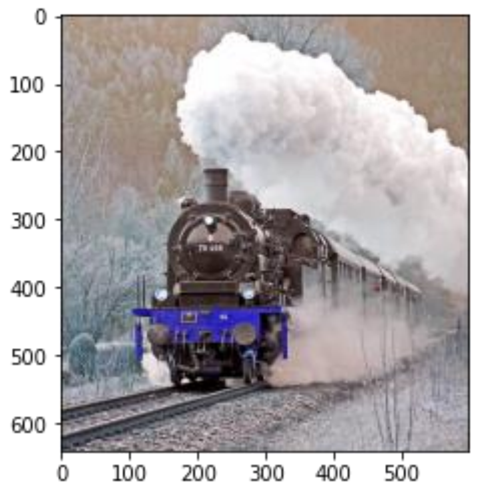
```
# summarize the mdodel
model.summary()
```

```
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_4 (InputLayer)        [(None, 224, 224, 3)]     0

_________________________________________________________________
block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

_________________________________________________________________
block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

_________________________________________________________________
block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

_________________________________________________________________
block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

_________________________________________________________________
block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

_________________________________________________________________
block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

_________________________________________________________________
block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

_________________________________________________________________
block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

_________________________________________________________________
block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

_________________________________________________________________
block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

_________________________________________________________________
block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

_________________________________________________________________
block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

_________________________________________________________________
block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

_________________________________________________________________
block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

_________________________________________________________________
block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

_________________________________________________________________
block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

_________________________________________________________________
block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

_________________________________________________________________
block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

_________________________________________________________________
flatten (Flatten)           (None, 25088)             0

_________________________________________________________________
fc1 (Dense)                 (None, 4096)              102764544

_________________________________________________________________
fc2 (Dense)                 (None, 4096)              16781312

_________________________________________________________________
predictions (Dense)         (None, 1000)              4097000
```

Example (3/4)

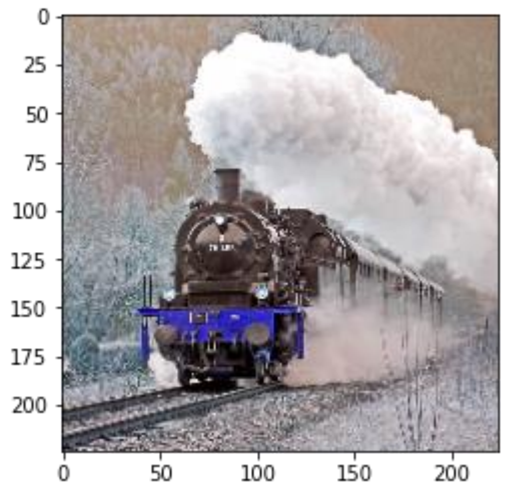
```
img = cv2.imread('steam-locomotive.jpg')
print(img.shape)
plt.imshow(img)
```

```
(640, 598, 3)
```



```
# resize into VGG16 trained images' format
# add a dummy iniziatl axis
im = cv2.resize(img, (224, 224))
plt.imshow(im)
im = np.expand_dims(im, axis=0)
im.astype(np.float32)
print(im.shape)
```

```
(1, 224, 224, 3)
```



Example (4/4)

```
# predict
out = model.predict(im)
index = np.argmax(out)
print(index)
```

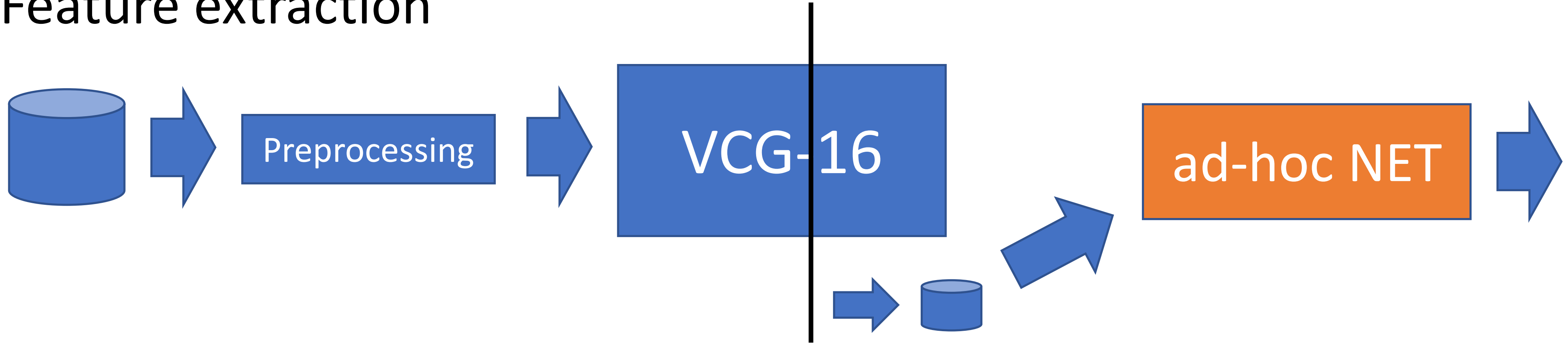
```
820
```

Transfer learning

- The knowledge inferred for solving a task can be reused in other tasks
- A VCG-16 network can be used to solve different classification problems
  - But we need to focus on the first layers, which extract the relevant features
  - Indeed, deeper layers are too specific for the application at hand

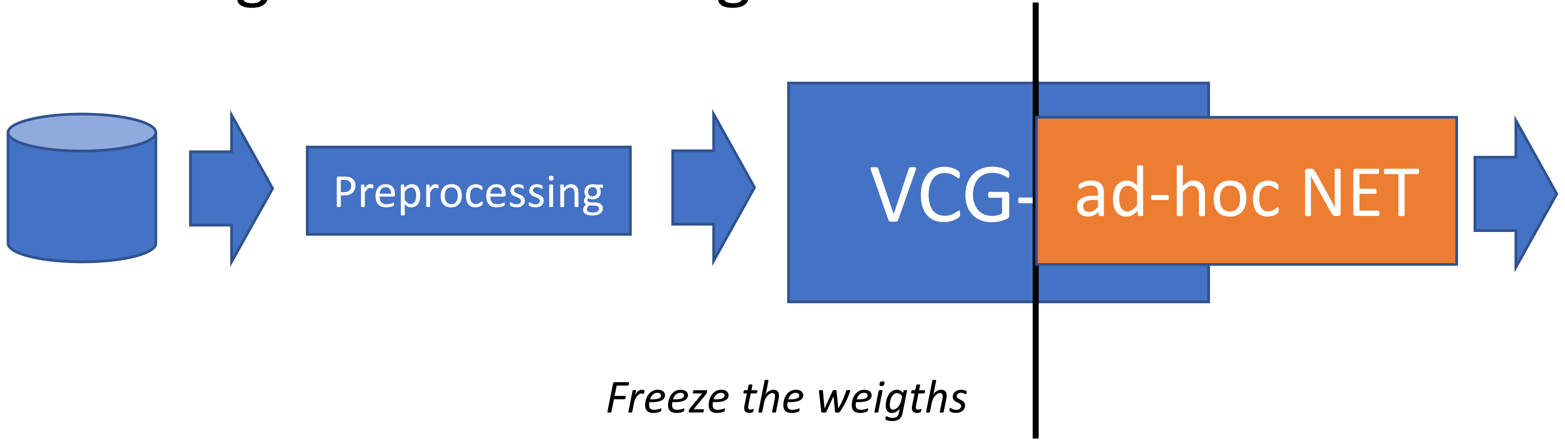
Usage of pre-trained nets

- Feature extraction



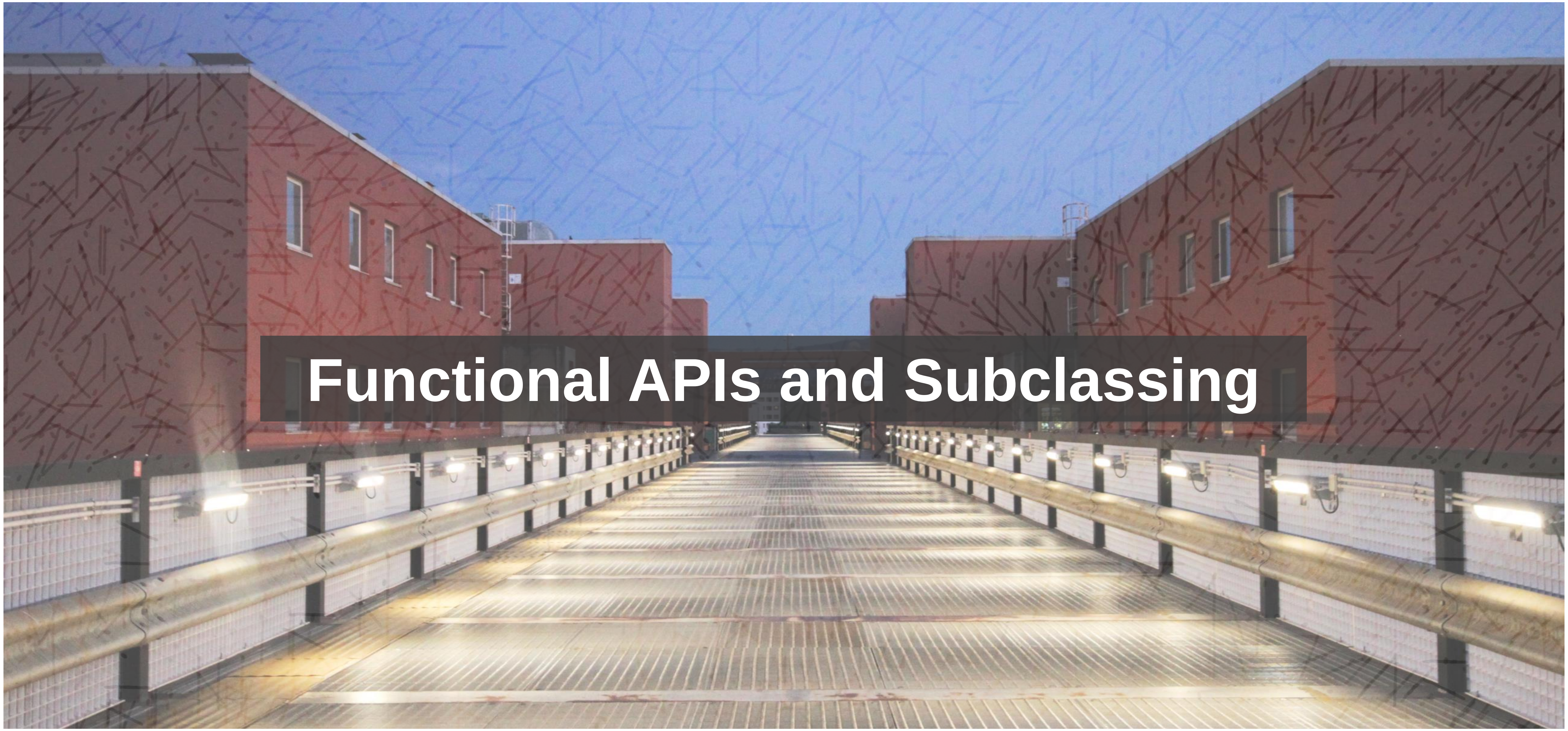
Features extracted from some given layer

- Freezing and fine-tuning



Freeze the weigths

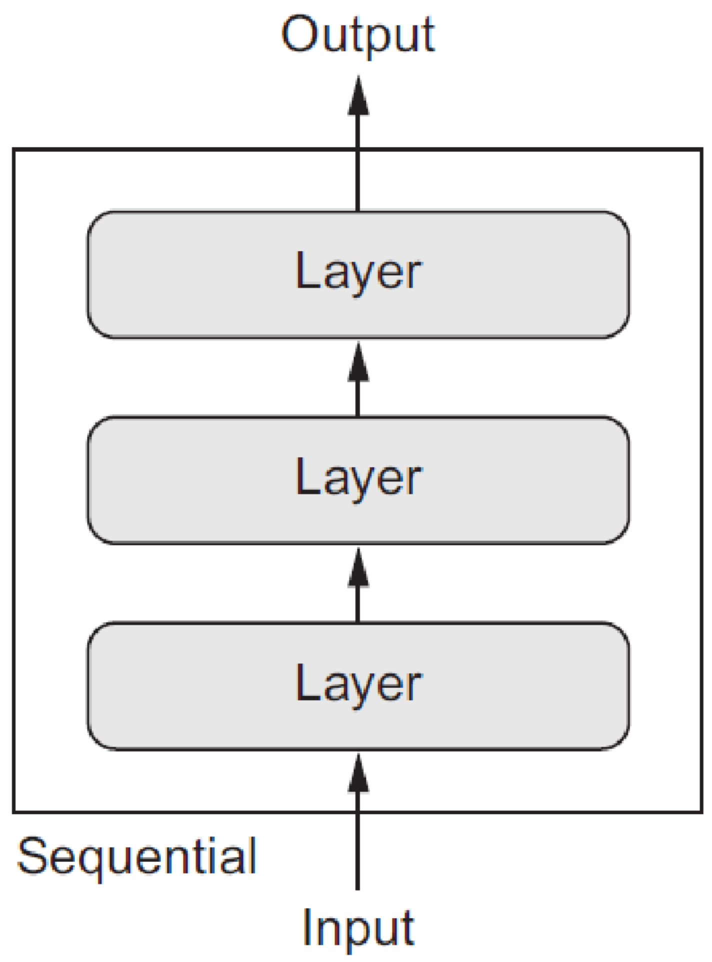
UNIVERSITÀ DELLA CALABRIA



Functional APIs and Subclassing

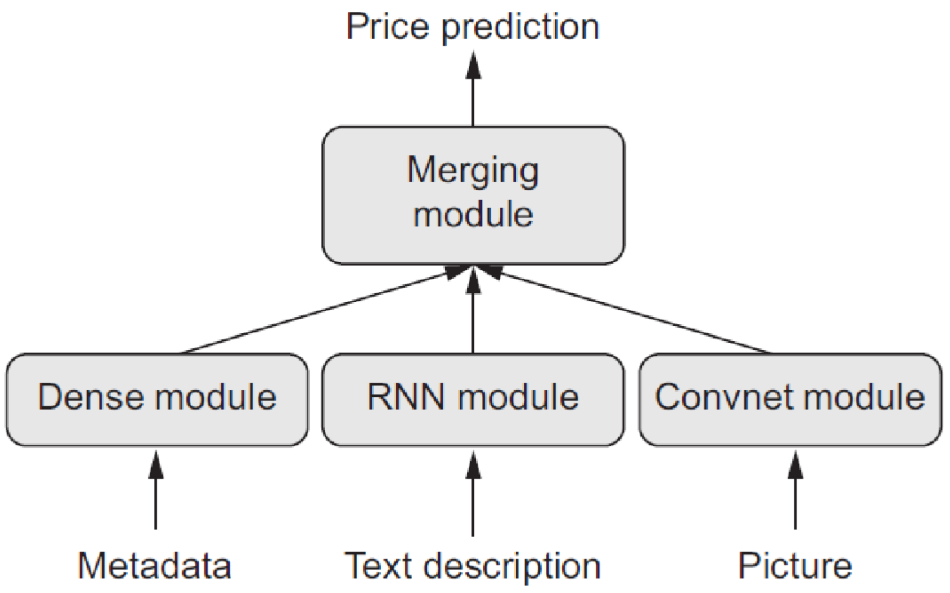
Limits of sequential models

- There are a number of applications where information does not just flow from an input to an output
- The sequential model is not flexible

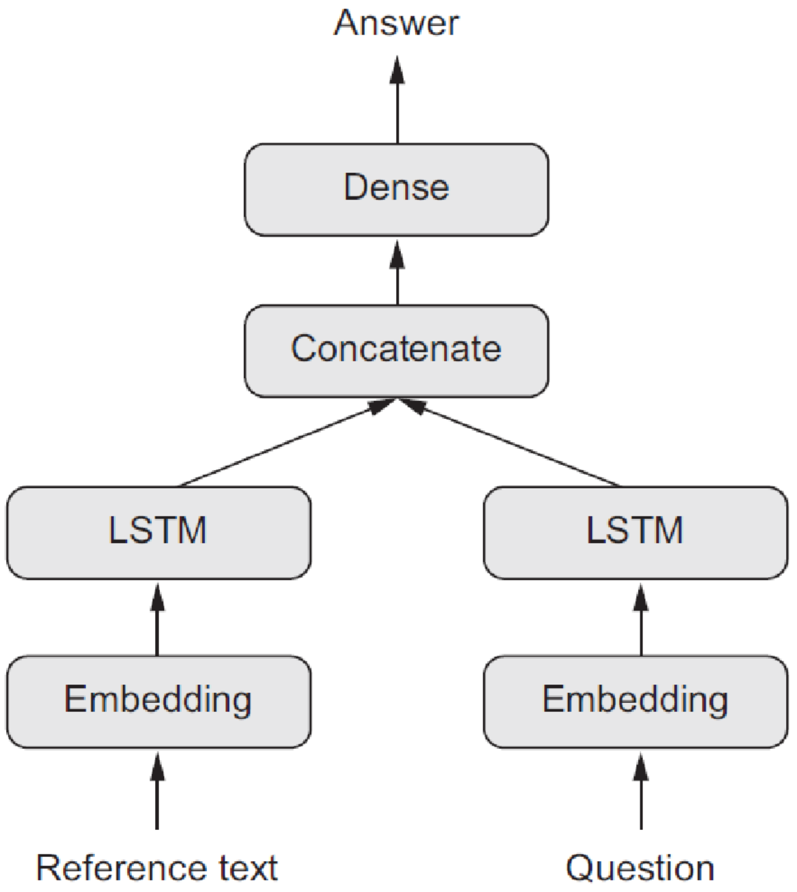


Multi-input

- I The various inputs need different kinds of specialized architectures to be dealt with



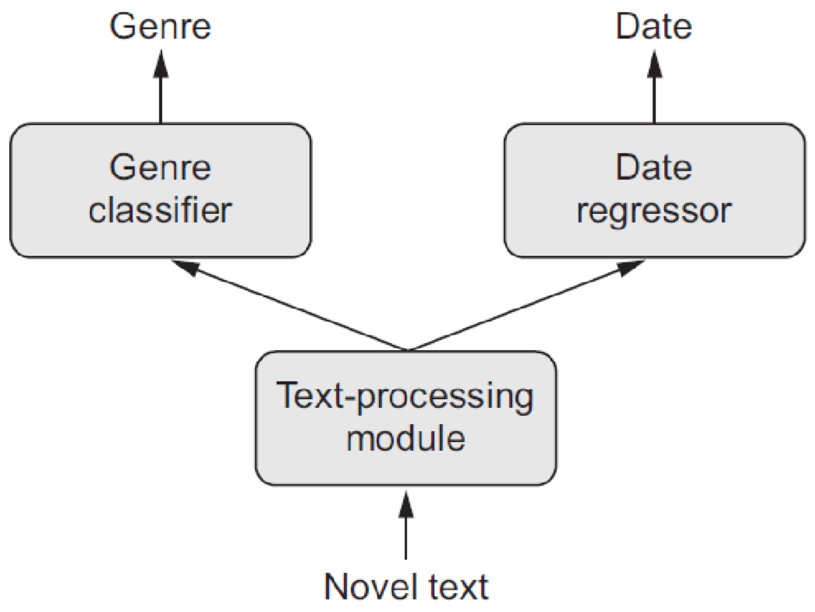
Price prediction



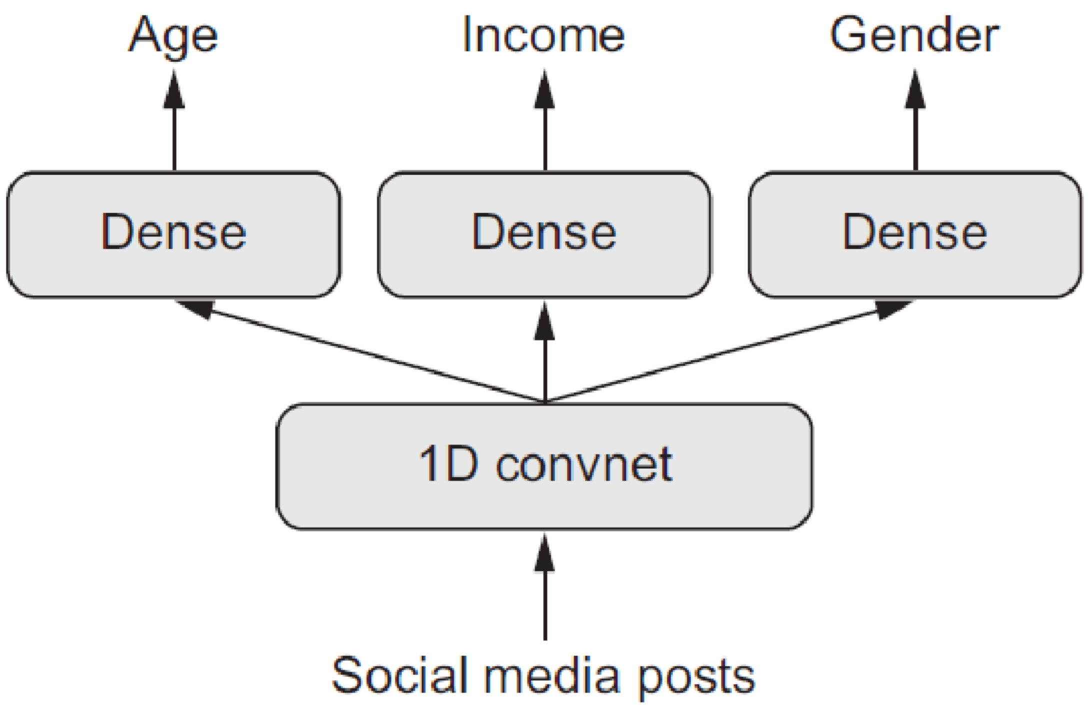
Question answering

Multi-output

- There are applications in which we predict different kinds of heterogenous information

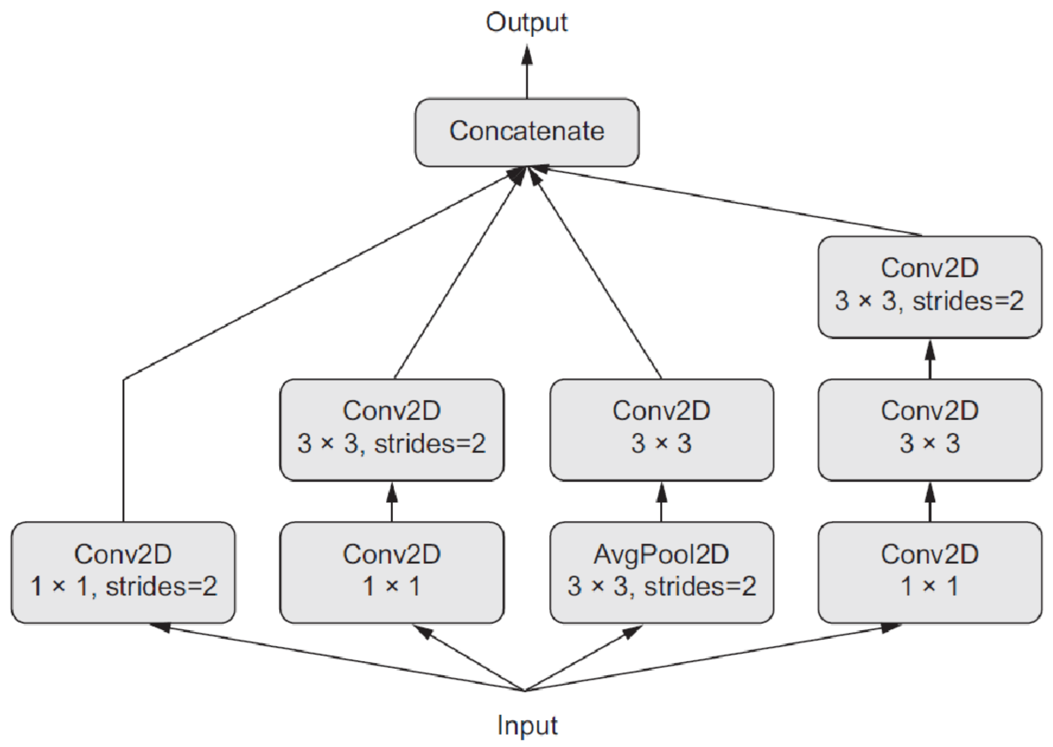


Advanced novel text classification

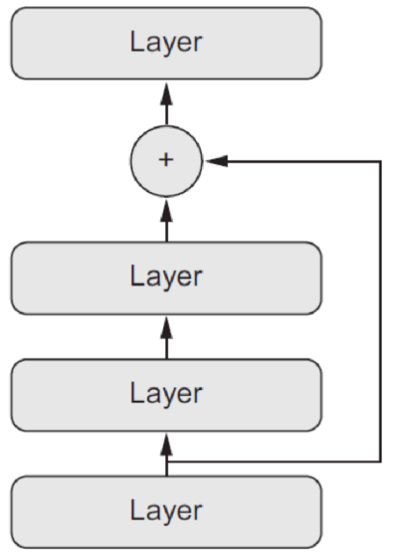


Social media analysis

Advanced Architectures



Inception



Residual

Advanced Keras - Subclassing

- Custom Loss
- Custom Metrics
- Custom Layers
- Custom Models

Object-Oriented Style

Custom Loss

```
def custom_mean_squared_error(y_true, y_pred):
    return tf.math.reduce_mean(tf.square(y_true - y_pred))

model.compile(optimizer=keras.optimizers.Adam(), loss=custom_mean_squared_error)
```

```
class CustomMSE(keras.losses.Loss):
    def __init__(self, regularization_factor=0.1, name="custom_mse"):
        super().__init__(name=name)
        self.regularization_factor = regularization_factor

    def call(self, y_true, y_pred):
        mse = tf.math.reduce_mean(tf.square(y_true - y_pred))
        reg = tf.math.reduce_mean(tf.square(0.5 - y_pred))
        return mse + reg * self.regularization_factor

model.compile(optimizer=keras.optimizers.Adam(), loss=CustomMSE())
```

Custom Metrics

```
class MyMetric(keras.metrics.Metric):
    def __init__(self, **kwargs):
        super(MyMetric, self).__init__(**kwargs)
        self.value = self.add_weight(name="value", initializer="zeros")
        self.num = self.add_weight(name="num", initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.value.assign_add(tf.constant(100.0))
        self.num.assign_add(tf.constant(1.0))

    def result(self):
        return self.value

    def reset_states(self):
        # The state of the metric will be reset at the start of each epoch.
        self.value.assign(0.0)
        self.num.assign(0.0)

model.compile(optimizer=keras.optimizers.Adam(), metric=MyMetric())
```

Custom Layers (1/2)

```
class ActivityRegularizationLayer(layers.Layer):
    def call(self, inputs):
        self.add_loss(tf.reduce_sum(inputs) * 0.1)
        return inputs  # Pass-through layer.


inputs = keras.Input(shape=(784,), name="digits")
x = layers.Dense(64, activation="relu", name="dense_1")(inputs)

# Insert activity regularization as a layer
x = ActivityRegularizationLayer()(x)

x = layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = layers.Dense(10, name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-3),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
)
```

Custom Layers (2/2)

```
class LogisticEndpoint(keras.layers.Layer):
    def __init__(self, name=None):
        super(LogisticEndpoint, self).__init__(name=name)
        self.loss_fn = keras.losses.BinaryCrossentropy(from_logits=True)
        self.accuracy_fn = keras.metrics.BinaryAccuracy()

    def call(self, targets, logits, sample_weights=None):
        # Compute the training-time loss value and add it
        # to the layer using `self.add_loss()`.
        loss = self.loss_fn(targets, logits, sample_weights)
        self.add_loss(loss)

        # Log accuracy as a metric and add it
        # to the layer using `self.add_metric()`.
        acc = self.accuracy_fn(targets, logits, sample_weights)
        self.add_metric(acc, name="accuracy")

        # Return the inference-time prediction tensor (for `.predict()`).
        return tf.nn.softmax(logits)
```