Master in Computer Vision *Barcelona*

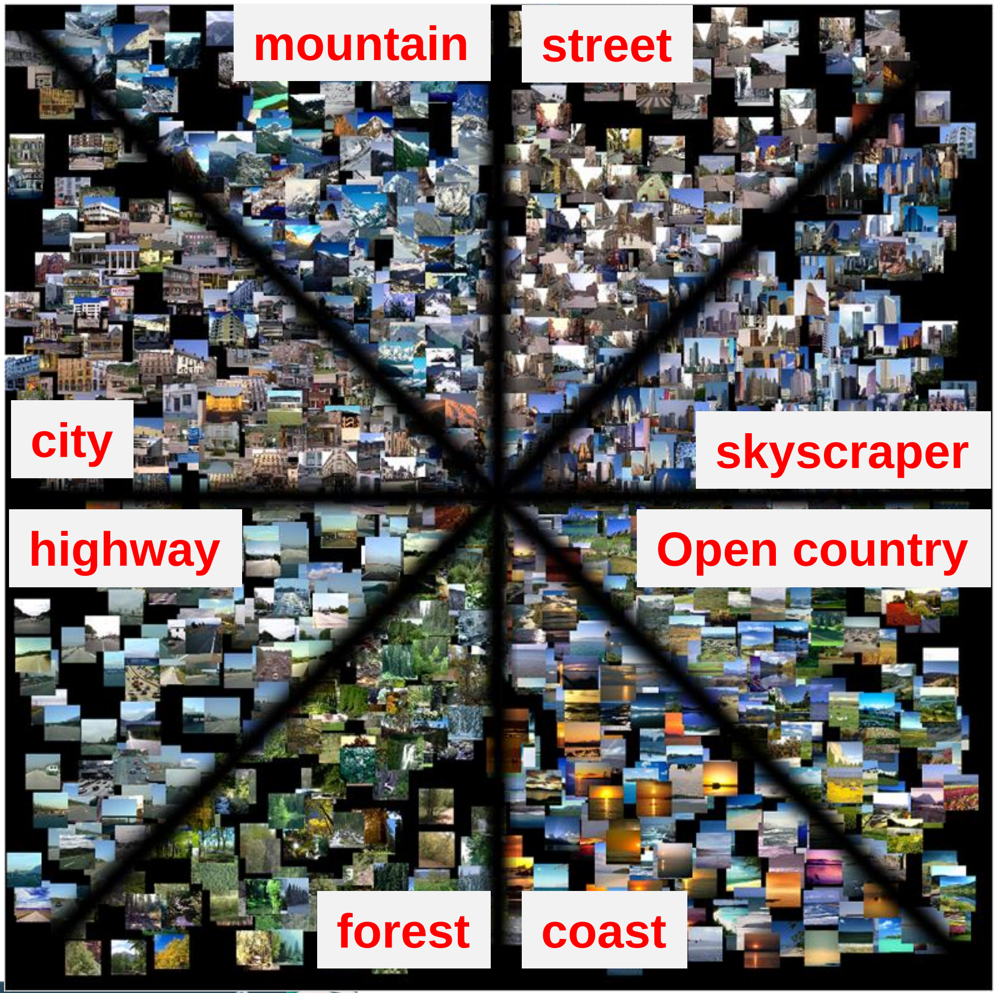
Module 3: Machine Learning for Computer Vision

Project: Deep learning classification

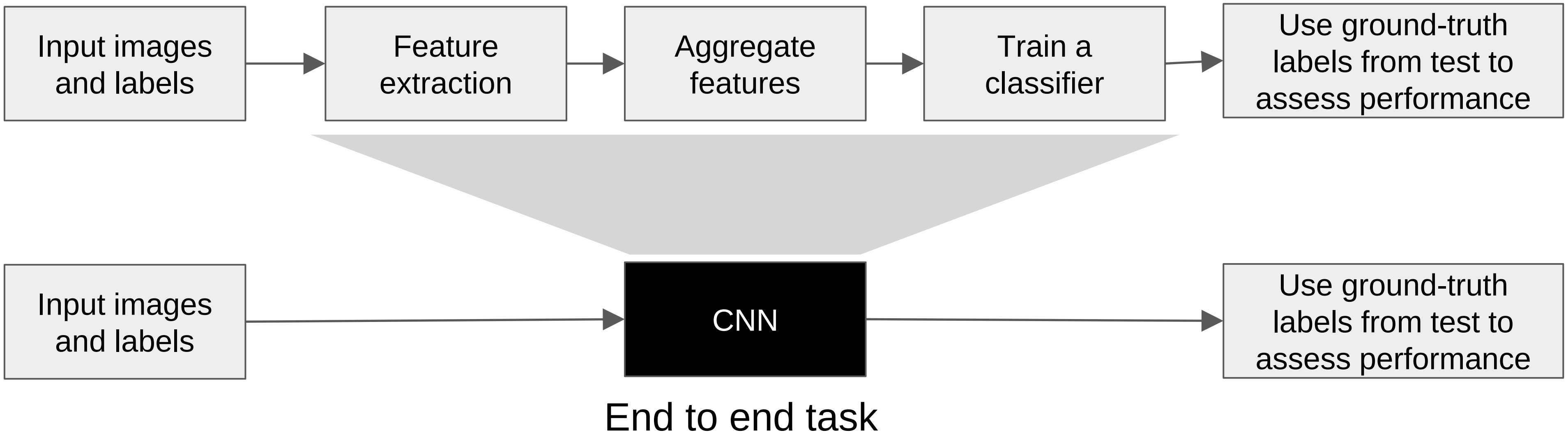
Lecturer: Ramon Baldrich

Module Goal

The aim of this module is to learn the techniques for category classification: handcrafted and learned.



Pipeline of the project W5 and W6



Machine learning for image classification:

- Data driven methods: Deep Convolutional Networks: 3 sessions
  - From hand-crafted to learnt features
  - Fine tuning of pre-trained CNNs
  - Training a CNN from scratch

Keras: first example

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))

inputs = Input(shape=None))
x = Dense(12, init='uniform', activation='relu', name='fc1')(x)
x = Dense(8, init='uniform', activation=‘sigmoid', name= 'predictions’)(x)
model = Model(inputs, x, name=‘example')
```

W3-5

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

W3-4

```
# predict with the model
features = model.predict(X)
```

W3-4

Understanding CNN topology: filtering

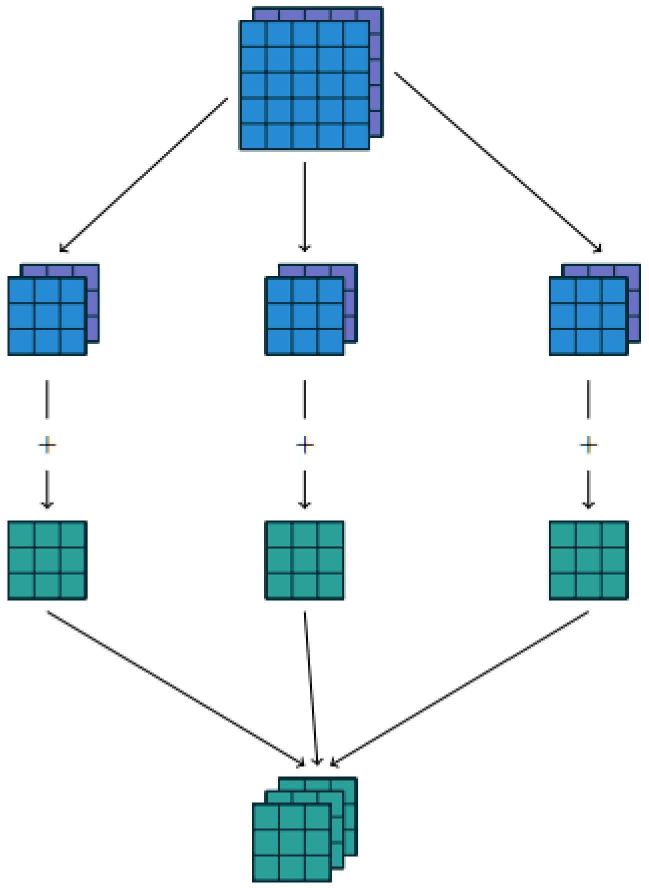
A guide to convolution arithmetic for deep learning

Vincent Dumoulin[1★] and Francesco Visin[2★†]

★MILA, Université de Montréal
†AIRLab, Politecnico di Milano

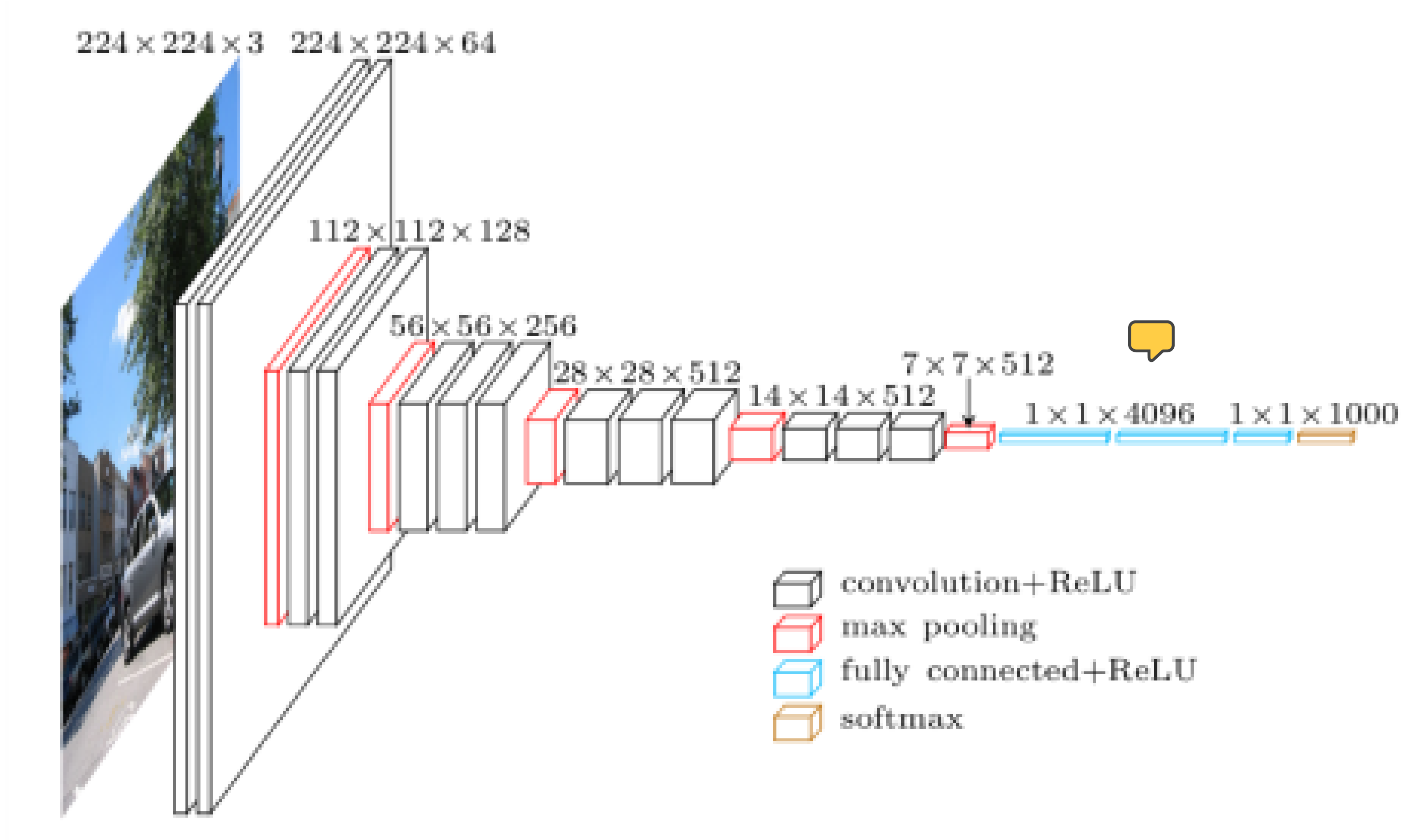
March 24, 2016

https://arxiv.org/pdf/1603.07285v1.pdf

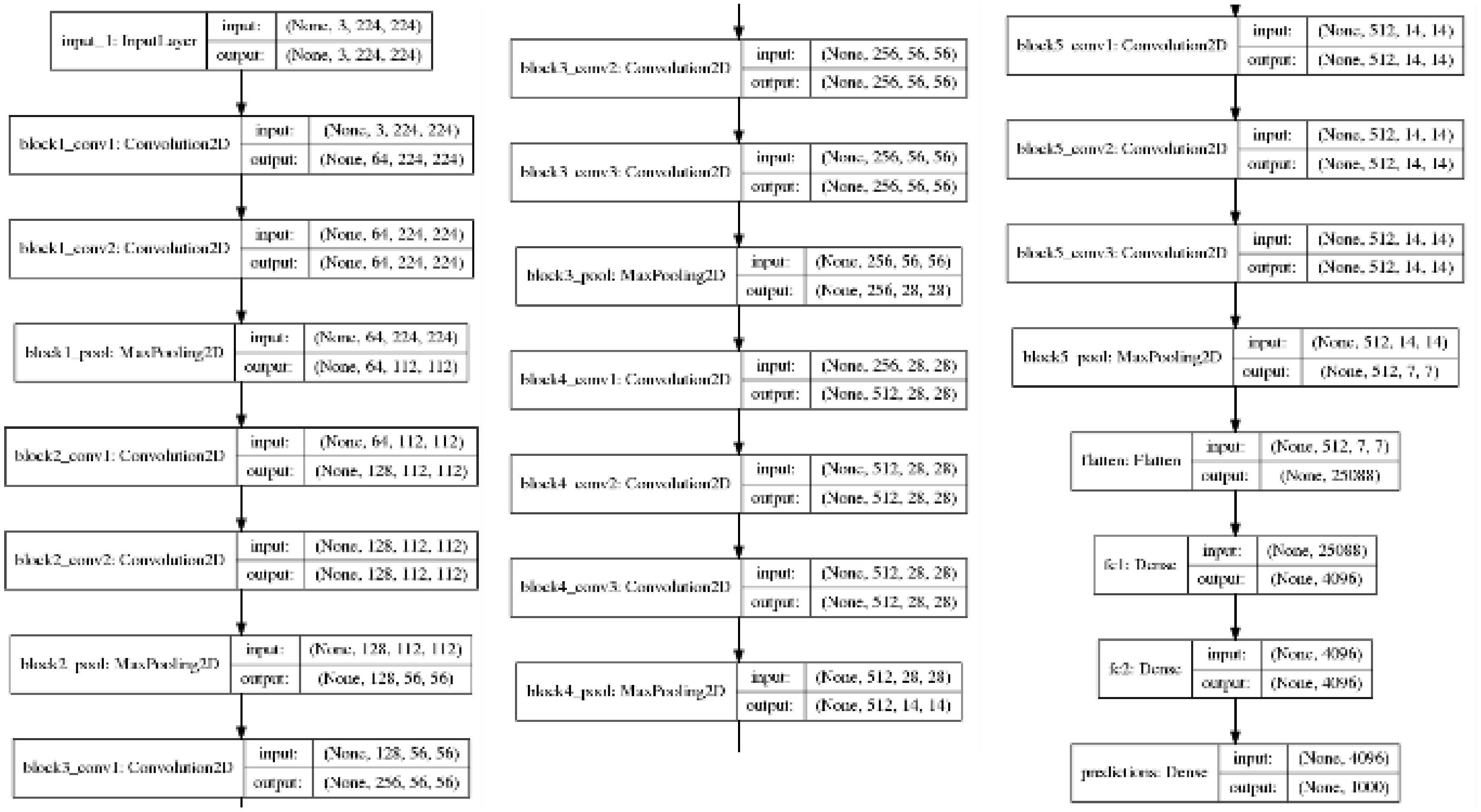


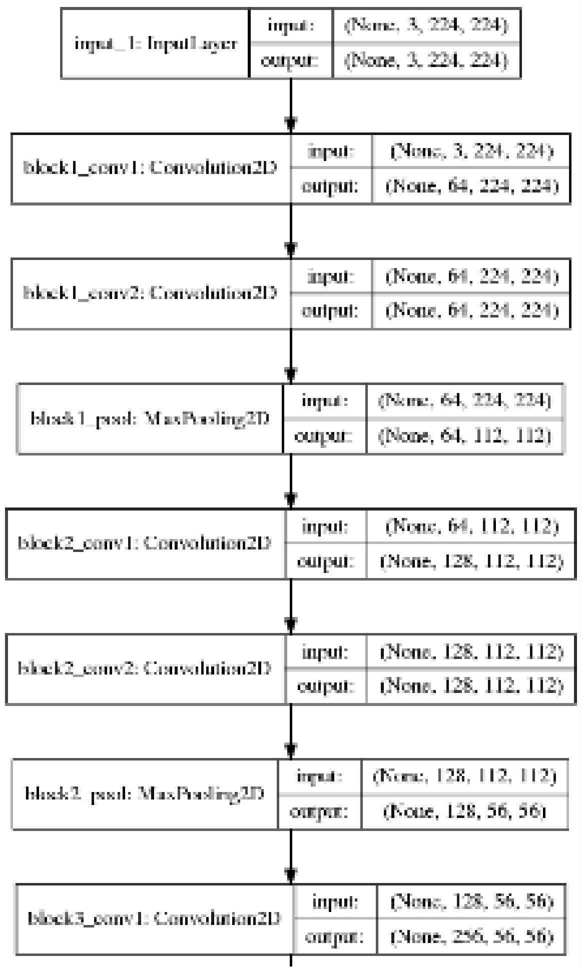
Input: 5x5x2
Filter: 3x3x3→3x3x2x3
Oputput: 3x3x3 or 5x5x3

Very deep convolutional networks for large-scale image recognition

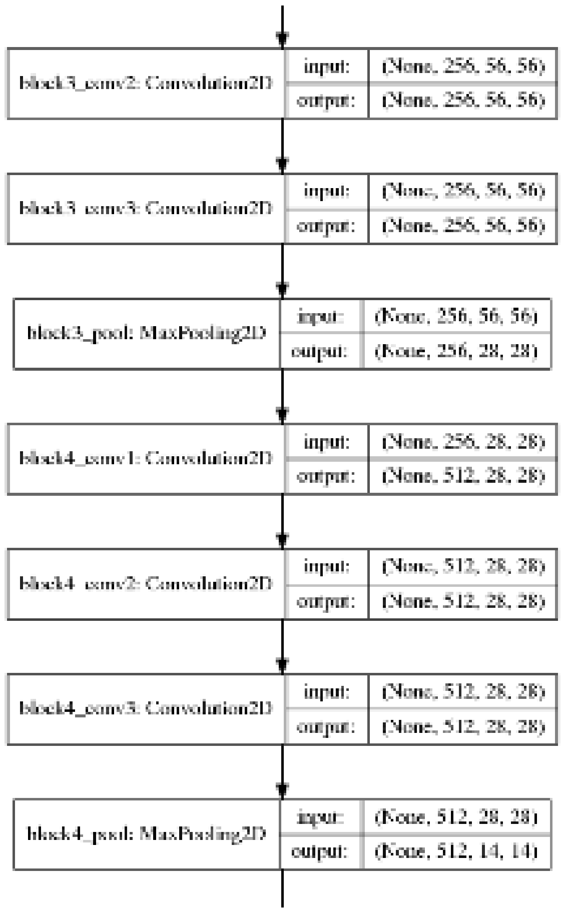


Credit Davi Frossard

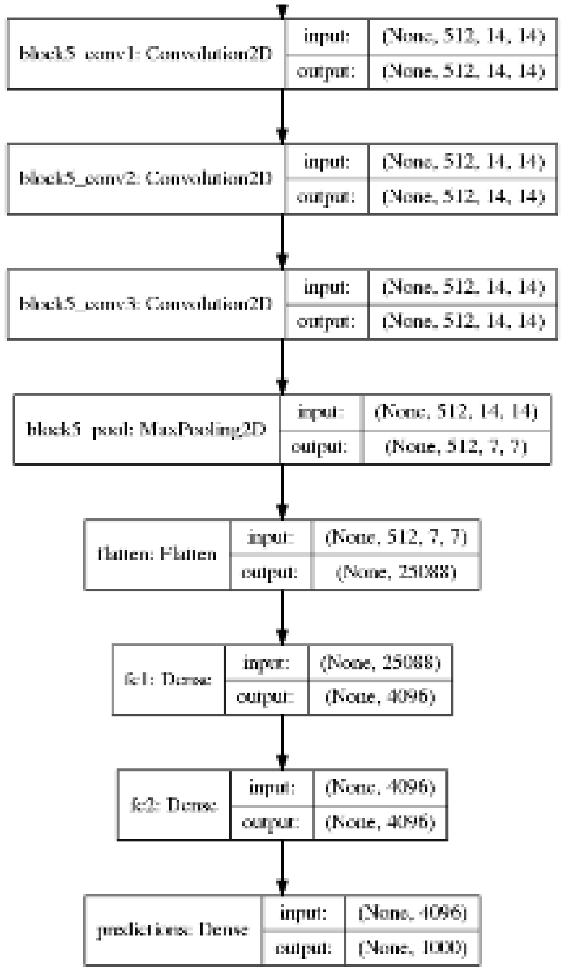

input_1: InputLayer | input: (None, 3, 224, 224) | output: (None, 3, 224, 224)
block1_conv1: Convolution2D | input: (None, 3, 224, 224) | output: (None, 64, 224, 224)
block1_conv2: Convolution2D | input: (None, 64, 224, 224) | output: (None, 64, 224, 224)
block1_pool: MaxPooling2D | input: (None, 64, 224, 224) | output: (None, 64, 112, 112)
block2_conv1: Convolution2D | input: (None, 64, 112, 112) | output: (None, 128, 112, 112)
block2_conv2: Convolution2D | input: (None, 128, 112, 112) | output: (None, 128, 112, 112)
block2_pool: MaxPooling2D | input: (None, 128, 112, 112) | output: (None, 128, 56, 56)
block3_conv1: Convolution2D | input: (None, 128, 56, 56) | output: (None, 256, 56, 56)
block3_conv2: Convolution2D | input: (None, 256, 56, 56) | output: (None, 256, 56, 56)
block3_conv3: Convolution2D | input: (None, 256, 56, 56) | output: (None, 256, 56, 56)
block3_pool: MaxPooling2D | input: (None, 256, 56, 56) | output: (None, 256, 28, 28)
block4_conv1: Convolution2D | input: (None, 256, 28, 28) | output: (None, 512, 28, 28)
block4_conv2: Convolution2D | input: (None, 512, 28, 28) | output: (None, 512, 28, 28)
block4_conv3: Convolution2D | input: (None, 512, 28, 28) | output: (None, 512, 28, 28)
block4_pool: MaxPooling2D | input: (None, 512, 28, 28) | output: (None, 512, 14, 14)
block5_conv1: Convolution2D | input: (None, 512, 14, 14) | output: (None, 512, 14, 14)
block5_conv2: Convolution2D | input: (None, 512, 14, 14) | output: (None, 512, 14, 14)
block5_conv3: Convolution2D | input: (None, 512, 14, 14) | output: (None, 512, 14, 14)
block5_pool: MaxPooling2D | input: (None, 512, 14, 14) | output: (None, 512, 7, 7)
flatten: Flatten | input: (None, 512, 7, 7) | output: (None, 25088)
fc1: Dense | input: (None, 25088) | output: (None, 4096)
fc2: Dense | input: (None, 4096) | output: (None, 4096)
predictions: Dense | input: (None, 4096) | output: (None, 1000)


input_1: InputLayer | input: (None, 3, 224, 224) | output: (None, 3, 224, 224)
block1_conv1: Convolution2D | input: (None, 3, 224, 224) | output: (None, 64, 224, 224)
block1_conv2: Convolution2D | input: (None, 64, 224, 224) | output: (None, 64, 224, 224)
block1_pool: MaxPooling2D | input: (None, 64, 224, 224) | output: (None, 64, 112, 112)
block2_conv1: Convolution2D | input: (None, 64, 112, 112) | output: (None, 128, 112, 112)
block2_conv2: Convolution2D | input: (None, 128, 112, 112) | output: (None, 128, 112, 112)
block2_pool: MaxPooling2D | input: (None, 128, 112, 112) | output: (None, 128, 56, 56)
block3_conv1: Convolution2D | input: (None, 128, 56, 56) | output: (None, 256, 56, 56)

```
img_input = Input(shape=(3,224,224))

x = Convolution2D(64, 3, 3, activation='relu',
border_mode='same', name='block1_conv1')(img_input)

x = Convolution2D(64, 3, 3, activation='relu',
border_mode='same', name='block1_conv2')(x)

x = MaxPooling2D((2, 2), strides=(2, 2),
name='block1_pool')(x)

x = Convolution2D(128, 3, 3, activation='relu',
border_mode='same', name='block2_conv1')(x)

x = Convolution2D(128, 3, 3, activation='relu',
border_mode='same', name='block2_conv2')(x)

x = MaxPooling2D((2, 2), strides=(2, 2),
name='block2_pool')(x)

x = Convolution2D(256, 3, 3, activation='relu',
border_mode='same', name='block3_conv1')(x)
```


block3_conv2: Convolution2D
input: (None, 256, 56, 56)
output: (None, 256, 56, 56)
block3_conv3: Convolution2D
input: (None, 256, 56, 56)
output: (None, 256, 56, 56)
block3_pool: MaxPooling2D
input: (None, 256, 56, 56)
output: (None, 256, 28, 28)
block4_conv1: Convolution2D
input: (None, 256, 28, 28)
output: (None, 512, 28, 28)
block4_conv2: Convolution2D
input: (None, 512, 28, 28)
output: (None, 512, 28, 28)
block4_conv3: Convolution2D
input: (None, 512, 28, 28)
output: (None, 512, 28, 28)
block4_pool: MaxPooling2D
input: (None, 512, 28, 28)
output: (None, 512, 14, 14)

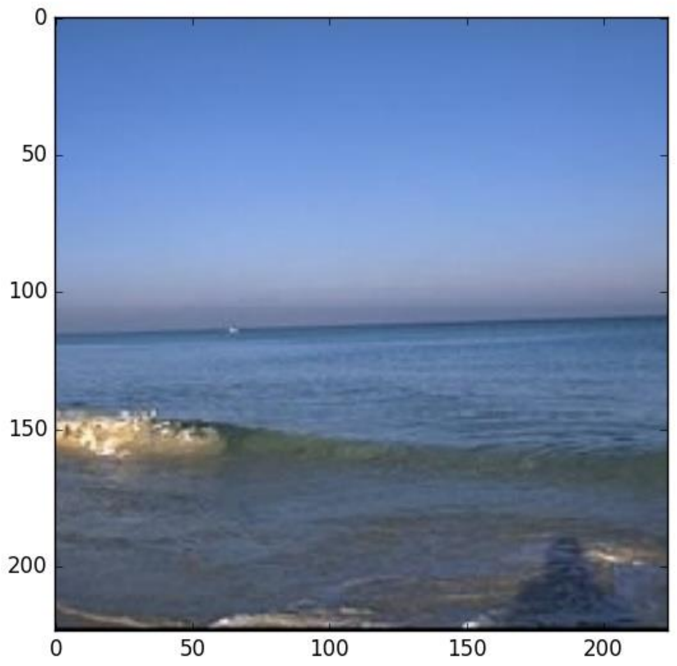
```
x = Convolution2D(256, 3, 3, activation='relu',
border_mode='same', name='block3_conv2')(x)

x = Convolution2D(256, 3, 3, activation='relu',
border_mode='same', name='block3_conv3')(x)

x = MaxPooling2D((2, 2), strides=(2, 2),
name='block3_pool')(x)

x = Convolution2D(512, 3, 3, activation='relu',
border_mode='same', name='block4_conv1')(x)

x = Convolution2D(512, 3, 3, activation='relu',
border_mode='same', name='block4_conv2')(x)

x = Convolution2D(512, 3, 3, activation='relu',
border_mode='same', name='block4_conv3')(x)

x = MaxPooling2D((2, 2), strides=(2, 2),
name='block4_pool')(x)
```

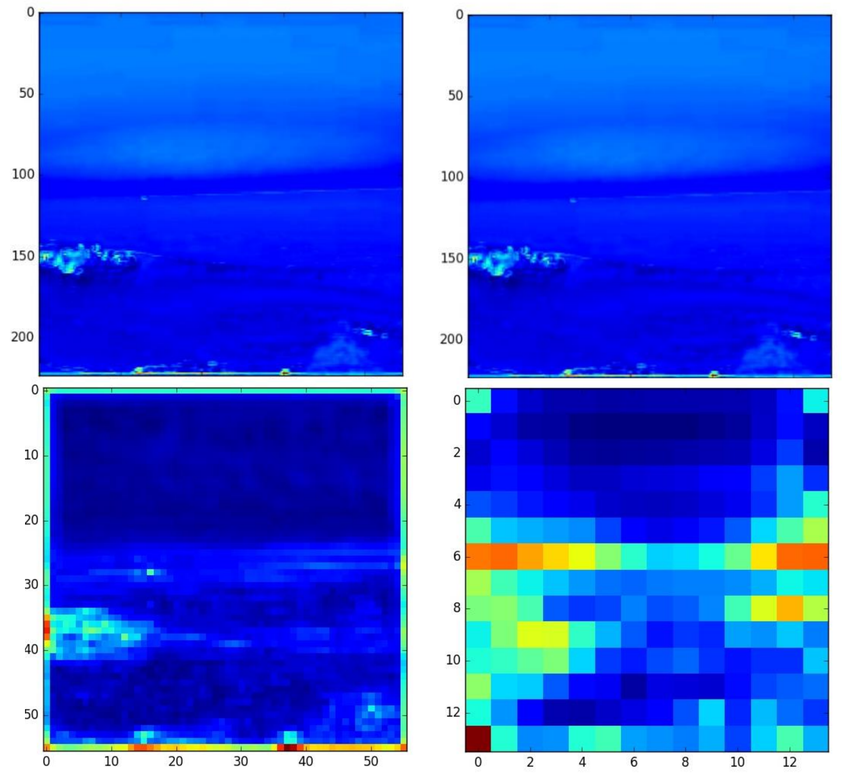

block5_conv1: Convolution2D
input: (None, 512, 14, 14)
output: (None, 512, 14, 14)
block5_conv2: Convolution2D
input: (None, 512, 14, 14)
output: (None, 512, 14, 14)
block5_conv3: Convolution2D
input: (None, 512, 14, 14)
output: (None, 512, 14, 14)
block5_pool: MaxPooling2D
input: (None, 512, 14, 14)
output: (None, 512, 7, 7)
flatten: Flatten
input: (None, 512, 7, 7)
output: (None, 25088)
fc1: Dense
input: (None, 25088)
output: (None, 4096)
fc2: Dense
input: (None, 4096)
output: (None, 4096)
predictions: Dense
input: (None, 4096)
output: (None, 1000)

```
x = Convolution2D(512, 3, 3, activation='relu',
border_mode='same', name='block5_conv1')(x)

x = Convolution2D(512, 3, 3, activation='relu',
border_mode='same', name='block5_conv2')(x)

x = Convolution2D(512, 3, 3, activation='relu',
border_mode='same', name='block5_conv3')(x)

x = MaxPooling2D((2, 2), strides=(2, 2),
name='block5_pool')(x)

x = Flatten(name='flatten')(x)

x = Dense(4096, activation='relu', name='fc1')(x)

x = Dense(4096, activation='relu', name='fc2')(x)

x = Dense(1000, activation='softmax',name='predictions')(x)
```

Extract features maps

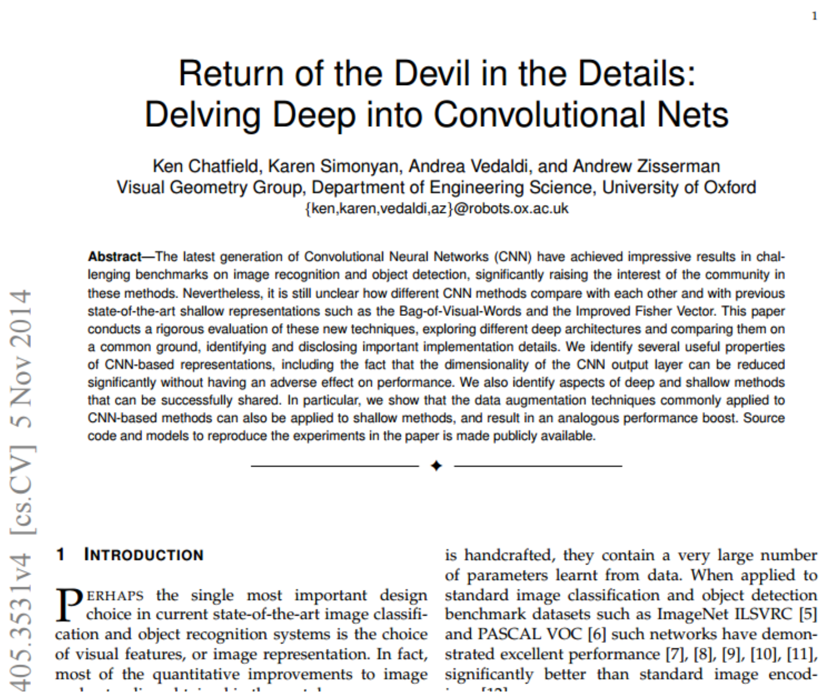
```
img_path = '/data/MIT/test/coast/art1130.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

base_model = VGG16(weights=‘imagenet’)
model = Model(inputs=base_model.input, outputs=base_model.get_layer('block1_conv1').output)
features = model.predict(x)
```





For visualizing purposes:
How to get rid of 3rd dimensión?

Week 5: Fine tune end to end classification



1

Return of the Devil in the Details: Delving Deep into Convolutional Nets

Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman
Visual Geometry Group, Department of Engineering Science, University of Oxford
{ken,karen,vedaldi,az}@robots.ox.ac.uk

Abstract—The latest generation of Convolutional Neural Networks (CNN) have achieved impressive results in challenging benchmarks on image recognition and object detection, significantly raising the interest of the community in these methods. Nevertheless, it is still unclear how different CNN methods compare with each other and with previous state-of-the-art shallow representations such as the Bag-of-Visual-Words and the Improved Fisher Vector. This paper conducts a rigorous evaluation of these new techniques, exploring different deep architectures and comparing them on a common ground, identifying and disclosing important implementation details. We identify several useful properties of CNN-based representations, including the fact that the dimensionality of the CNN output layer can be reduced significantly without having an adverse effect on performance. We also identify aspects of deep and shallow methods that can be successfully shared. In particular, we show that the data augmentation techniques commonly applied to CNN-based methods can also be applied to shallow methods, and result in an analogous performance boost. Source code and models to reproduce the experiments in the paper is made publicly available.

arXiv:1405.3531v4 [cs.CV] 5 Nov 2014

1 INTRODUCTION

PERHAPS the single most important design choice in current state-of-the-art image classification and object recognition systems is the choice of visual features, or image representation. In fact, most of the quantitative improvements to image

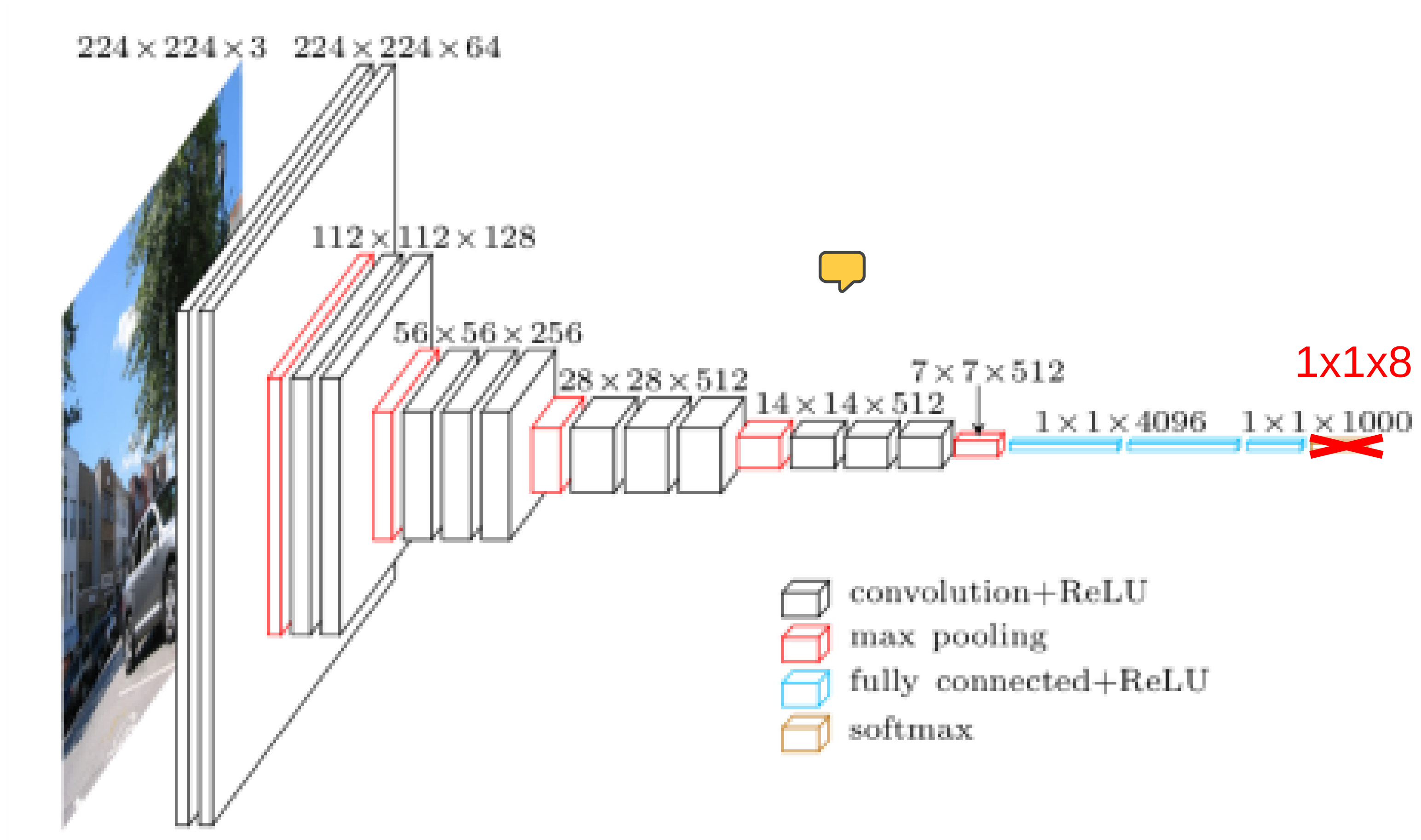
is handcrafted, they contain a very large number of parameters learnt from data. When applied to standard image classification and object detection benchmark datasets such as ImageNet ILSVRC [5] and PASCAL VOC [6] such networks have demonstrated excellent performance [7], [8], [9], [10], [11], significantly better than standard image encod-

Goals:

- Understand layer manipulation
- Deal with dataset loading
- Hyperparameter optimization

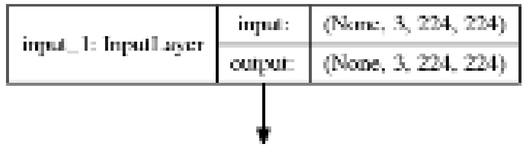
Chatfield, Ken, et al. "Return of the devil in the details: Delving deep into convolutional nets." *arXiv preprint arXiv:1405.3531* (2014).

Very deep convolutional networks for large-scale image recognition

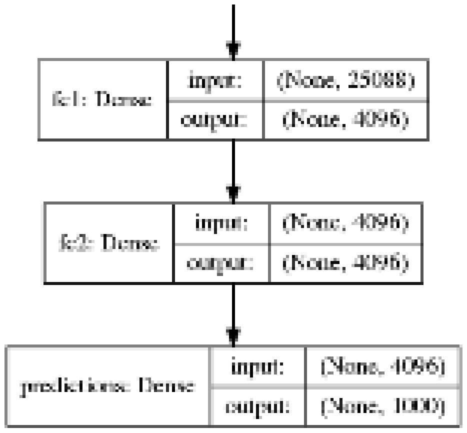


Credit Davi Frossard

Understand layer manipulation



img_input = Input(shape=(3,224,224))



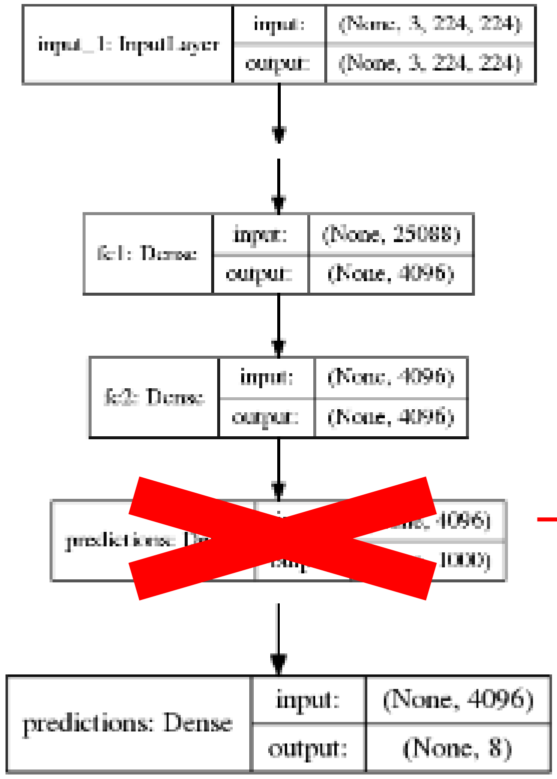
x = Dense(4096, activation='relu', name='fc1')(x)

x = Dense(4096, activation='relu', name='fc2')(x)

x = Dense(1000, activation='softmax',name='predictions')(x)

base_model = Model(img_input, x, name='vgg16')

Understand layer manipulation



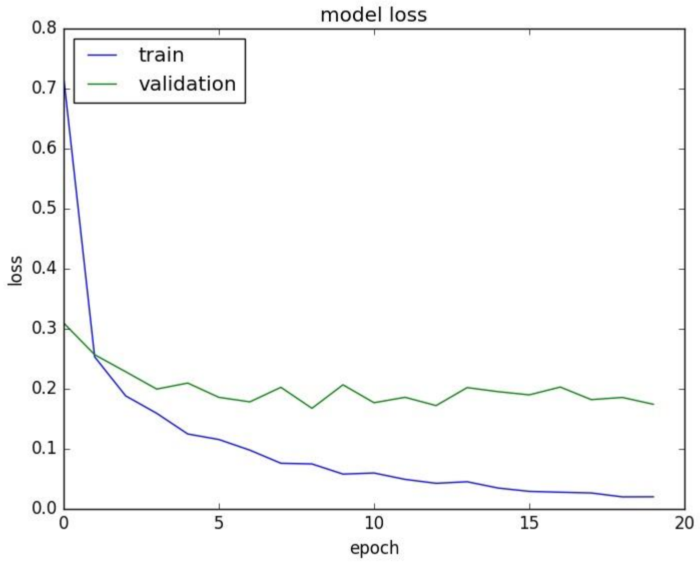
```
img_input = Input(shape=(3,224,224))

x = Dense(4096, activation='relu', name='fc1')(x)

x = Dense(4096, activation='relu', name='fc2')(x)
```

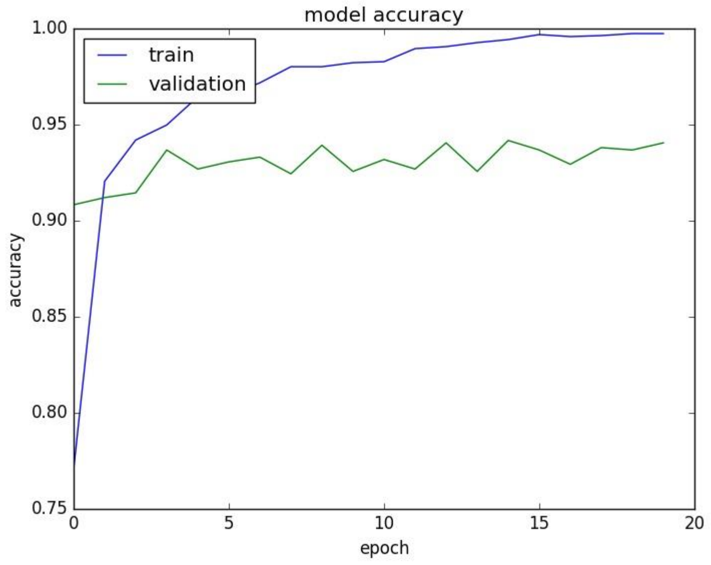
~~x = Dense(1000, activation='softmax',name='predictions')(x)~~

```
x = base_model.layers[-2].output
x = Dense(8, activation='softmax',name='predictions')(x)

model = Model(inputs=base_model.input, outputs=x)
```

Minimum results

Full training dataset
No hyperparameter optimization





Let's do things more interesting:

- cut the architecture in a lower layer
- use less training data (no more than 400)

Preparing the model

- Set goal function and process model

```
model.compile(loss='categorical_crossentropy',
    optimizer='adadelta', metrics=['accuracy'])
```

- Do not train on full network at starting point

```
for layer in base_model.layers:
    layer.trainable = False
```

Deal with dataset loading

```
from keras.applications.inception_v3 import preprocess_input

datagen = ImageDataGenerator(featurewise_center=False,
          samplewise_center=False,
          featurewise_std_normalization=False,
          samplewise_std_normalization=False,
          preprocessing_function=preprocess_input,   IMPORTANT
          rotation_range=0.,
          width_shift_range=0.,
          height_shift_range=0.,
          shear_range=0.,
          zoom_range=0.,
          fill_mode='nearest',
          horizontal_flip=False,
          vertical_flip=False,
          rescale=None)
```

Deal with dataset loading

```
train_generator = datagen.flow_from_directory(train_data_dir,
                      target_size=(img_width, img_height),
                      batch_size=batch_size,
                      class_mode='categorical')

test_generator = datagen.flow_from_directory(test_data_dir,
                      target_size=(img_width, img_height),
                      batch_size=batch_size,
                      class_mode='categorical')

validation_generator =datagen.flow_from_directory(val_data_dir,
                      target_size=(img_width, img_height),
                      batch_size=batch_size,
                      class_mode='categorical')
```

Deal with dataset loading

```
history=model.fit_generator(train_generator,
                    samples_per_epoch=400,
                    nb_epoch=number_of_epoch,
                    validation_data=validation_generator,
                    nb_val_samples=800)

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])


result = model.evaluate_generator(test_generator)
```

Afterwards, retrain in full model

Hyperparamter optimization

Per model
  batch_size = [10, 20, 40, 60, 80, 100]
  epochs = [10, 50, 100]
  optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
  learn_rate = [0.0001 0.001, 0.01, 0.1, 0.2, 0.3]
  momentum = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]
  data augmentation: flip, zoom, rescale, …

Per layer:
  activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']

  init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_uniform',
  'he_normal', 'he_uniform'] (Not useful in our case)

Topology:
  drop-out layers: p % of inactive weights
  batchnormalization
  regularizers

Tasks

Understanding layer manipulation

0. Fine tune an existing architeture

    https://keras.io/applications/

1. Set a new model from an existing architecutre.
2. Apply the model to a small set of data (no more than 400)

    /ghome/mcv/m3/datasets/MIT_small_train_X

Deal with dataset loading

3. Introduce and evaluate the usage of data augmentation

Hyperparameter optimization

4. Introduce and evaluate the usage of any suitable methodology to improve learning curve (dropout layer, batch norm, …)
5. Apply random search / optuna on per model hyperparameters

	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88	79.0%	94.5%	22.9M	81
VGG16	528	71.3%	90.1%	138.4M	16
ResNet50	98	74.9%	92.1%	25.6M	107
InceptionV3	92	77.9%	93.7%	23.9M	189
InceptionResNetV2	215	80.3%	95.3%	55.9M	449
MobileNet	16	70.4%	89.5%	4.3M	55
DenseNet121	33	75.0%	92.3%	8.1M	242
NASNetMobile	23	74.4%	91.9%	5.3M	389
EfficientNetB0	29	77.1%	93.3%	5.3M	132
ConvNeXtTiny	109.42	81.3%	-	28.6M	-

Grades, deliverables and deadline

- Deliver source code and a **short** slide presentation of the work done
  - For each task, all the tests with their associated results
  - 1 slide summarizing the best yielded result and configuration for each task
- Delivered by Monday 30th at 10:30AM

Control pipeline: Callbacks

- ModelCheckpoint
- EarlyStopping
- ReduceLROnPlateau
- CSVLogger
- LambdaCallback
- …

Usage:

callbacks = [ModelCheckpoint(….), EarlyStopping(…),…]

model.fit(…, callbacks)

Control pipeline: Callbacks

Example:

```
plot_loss_callback = LambdaCallback(on_epoch_end=lambda epoch, logs: plot_loss(epoch,logs))
save_callback=ModelCheckpoint(filepath=
                              'weights.{epoch:02d}-{val_loss:.2f}.hdf5',
                              monitor='val_acc', verbose=1, save_best_only=True,
                              mode='max')

history=model.fit(train_generator,
        samples_per_epoch=1900,
        nb_epoch=number_of_epoch,
        validation_data=validation_generator,
        nb_val_samples=800,
        callbacks=[plot_loss_callback,save_callback])
```