Applied Machine Learning - EDAN95
Assignment 3

Recognizing Flowers on Images
Managing an Image Dataset
Convolutional Networks on Images
Python Generators

Hicham Mohamad
hsmo@kth.se

November 22, 2018

1 Introduction

The objectives of this assignment are to:

- Write a program to recognize flowers on images
- Learn how to manage an image data set
- Apply **convolutional networks** to images
- Know what Python generators are

Programming

- Each group will have to write Python programs to recognize the sort of flower in an image.
- You will have to experiment different architectures and compare the results you obtained

2 Preliminaries: Convolutional Neural Networks (CNN)

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at other tasks, such as voice recognition or natural language processing (NLP); however, we will focus on visual applications for now.

As mentioned in chapter 9 in [5], Convolutional networks (LeCun, 1989), also known as **convolutional neural networks**, or CNNs, are a specialized kind of neural network for processing data that has a known *grid-like topology*. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications.

The name "convolutional neural network" indicates that the network employs a mathematical operation called *convolution*. Convolution is a specialized kind of linear operation. Convolutional networks are

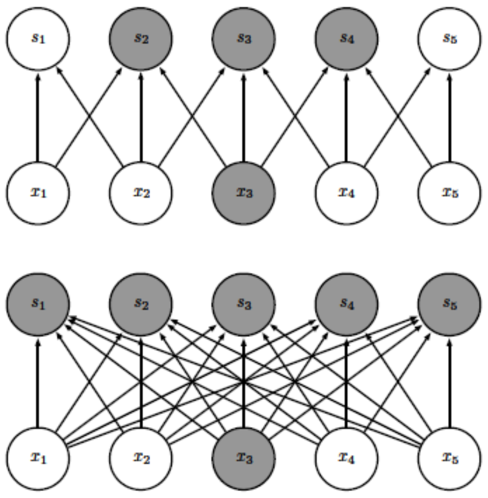

Figure 1: Sparse connectivity, viewed from below. We highlight one input unit, x_3, and highlight the output units in s that are affected by this unit. (Top) When s is formed by convolution with a kernel of width 3, only three outputs are affected by x. (Bottom) When s is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by x_3.

simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant representations**. Moreover, convolution provides a means for working with inputs of variable size.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. *Convolutional networks, however, typically have sparse interactions* (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input, as illustrated in Figure 1.

This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. *Parameter sharing* refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer.

Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

2.1 Performance

The factors determining how well a machine learning algorithm will perform are its ability to

1. Make the training error small.
2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: underfitting and overfitting. *Underfitting* occurs when the model is not able to obtain a sufficiently low error value on the training set. *Overfitting* occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its **capacity**. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its **hypothesis space**, the set of functions that the learning algorithm is allowed to select as being the solution.

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

The **no free lunch theorem** has made it clear that there is no best machine learning algorithm, and, in particular, no best form of regularization. Instead we must choose a form of regularization that is well suited to the particular task we want to solve. The philosophy of deep learning in general is that a wide range of tasks (such as all the intellectual tasks that people can do) may all be solved effectively using very general-purpose forms of regularization.

3 Collecting a Dataset

1. You will collect a dataset from Kaggle. Register, it is free, and you will have access to lots of datasets.
2. Download the *Flower corpus*. You can find a local copy in the /usr/local/cs/EDAN95/datasets folder.

   Alternatively, you can use another dataset, provided that it has more than two categories. Tell your teacher in advance in this case to check if it is acceptable. You can use the Google dataset search: https://toolbox.google.com/datasetsearch.
3. Split randomly your dataset into training, validation, and test sets: Use a 60/20/20 ratio. You will read all the file names and create a list of pairs, `(file_name, category)`. You will then shuffle your list and save your partition of the data. To speed up the lab, you can also start with the partition available in the `/usr/local/cs/EDAN95/datasets` folder. You can also use the code available from `https://github.com/pnugues/edan95`.

The Flowers dataset that we'll use isn't packaged with Keras. It was made available by Kaggle and downloaded from [13]. This dataset contains 4,326 images of flowers (769 Daisy class, 1,055 Dandelion class, 784 Rose class, 734 Sunflower class and 984 Tulip class) and is 225 MB (compressed). After downloading and uncompressing it, we'll create a new dataset containing three subsets:

- a training set with 2,595 samples of each class,
- a validation set with 865 samples of each class,
- and a test set with 866 samples of each class.

Listing 1: Read all the file names and create a list of pairs (filename and category) then shuffle the list and save the partition of the data to training validation and test directories.

```
"""
3.1-flowers_create_dataset.py
Categorizing the flower dataset
Creating the dataset
Author: Pierre Nugues
"""

import os
import random
import shutil
import numpy as np

vilde = False
np.random.seed(0)

if vilde:
    base = '/home/pierre/'
else:
    base = '/Users/pierre/Documents/'

```

```python
# Path to the directory where the original dataset was uncompressed
original_dataset_dir = os.path.join(base, 'Cours/EDAN95/datasets/flowers')

# Directory where we'll store our smaller dataset, base_dir
dataset = os.path.join(base, 'Cours/EDAN95/datasets/flowers_split')

# Directories for the training, validation and test splits
train_dir = os.path.join(dataset, 'train')
validation_dir = os.path.join(dataset, 'validation')
test_dir = os.path.join(dataset, 'test')

# Read all the file names
categories = os.listdir(original_dataset_dir)
categories = [category for category in categories if not category.
    startswith('.')]
print('Image types:', categories)
data_folders = [os.path.join(original_dataset_dir, category) for category
    in categories]

# Create a list of pairs (file_name, category)
pairs = []
for folder, category in zip(data_folders, categories):
    images = os.listdir(folder)
    images = [image for image in images if not image.startswith('.')]
    pairs.extend([(image, category) for image in images])

# shuffle() randomizes the items of a list in place
random.shuffle(pairs)
img_nbr = len(pairs)

# creates a new dataset containing 3 subsets 60/20/20 ratio
train_images = pairs[0:int(0.6 * img_nbr)]
val_images = pairs[int(0.6 * img_nbr):int(0.8 * img_nbr)]
test_images = pairs[int(0.8 * img_nbr):]

# print(train_images)
print(len(train_images))
print(len(val_images))
print(len(test_images))

# Copies the first 60 % flower images to train_dir
for image, label in train_images:
    src = os.path.join(original_dataset_dir, label, image)
    dst = os.path.join(train_dir, label, image)
    os.makedirs(os.path.dirname(dst), exist_ok=True)
    shutil.copyfile(src, dst)

# Copies the next 20 % flower images to validation_dir
for image, label in val_images:
    src = os.path.join(original_dataset_dir, label, image)
    dst = os.path.join(validation_dir, label, image)
    os.makedirs(os.path.dirname(dst), exist_ok=True)
    shutil.copyfile(src, dst)

# Copies the next 20 % flower images to test_dir
for image, label in test_images:
    src = os.path.join(original_dataset_dir, label, image)
    dst = os.path.join(test_dir, label, image)
    os.makedirs(os.path.dirname(dst), exist_ok=True)
    shutil.copyfile(src, dst)
```

NOTE: There are always a random component when splitting the data into `training / validation / test`. The network that had best performance on the validation set might not be the one with the best performance on new test data, even thought we try hard to make it be.

4 Building a Simple Convolutional Neural Network

1. Create a simple convolutional network and train a model with the train set. You can start from the architecture proposed by Chollet, `Listing 5.5`, and a small number of epochs. You will need to modify some parameters so that your network handles multiple classes. You will also adjust the number of steps so that your generator in the fitting procedure sees all the samples. You will report the training and validation losses and accuracies and comment on the possible overfit.
2. Apply your network to the test set and report the accuracy as well as the confusion matrix you obtained. As with fitting, you may need to adjust the number of steps so that your network tests all the samples.
3. Try to improve your model by modifying some parameters and evaluate your network again.

This task introduces *convolutional neural networks*, also known as *convnets*, a type of deep-learning model almost universally used in computer vision applications. we'll learn to apply convnets to image-classification problems.

We'll start by naively training a small convnet on the 2,595 training samples, without any regularization, to set a *baseline* for what can be achieved. This will get us to a *classification accuracy* of 71%. At that point, the main issue will be *overfitting*. Then we'll introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By using data augmentation, we'll improve the network to reach an accuracy of 82%.

In the next section, we'll review two more essential techniques for applying deep learning to small datasets: *feature extraction with a pretrained network* (which will get us to an accuracy of 90% to 96%) and *fine-tuning* a pretrained network (this will get us to a final accuracy of 97%). Together, these three strategies — training a small model from scratch, doing feature extraction using a pretrained model, and fine-tuning a pretrained model—will constitute our future toolbox for tackling the problem of performing image classification with small datasets.

It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundred can potentially suffice if the model is small and well regularized and the task is simple. Because convnets *learn local, translation-invariant features*, they're highly data efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering.

Listing 2: Instantiating a small convnet for Flowers classification

```

from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

5 Using Image Augmentation

The flower dataset is relatively small. A way to expand such datasets is to generate artificial images by applying small transformations to existing images. *Keras* provides a built-in class for this: `ImageDataGenerator`. You will reuse it and apply it to the flower data set.

1. Using the network from the previous exercise, apply some transformations to your images. You can start from Chollet, Listing 5.11.
2. Report the training and validation losses and accuracies and comment on the possible overfit.
3. Apply your network to the test set and report the accuracy as well as the confusion matrix you obtained.

Overfitting is caused by having too few samples to learn from, rendering you unable to train a model that can generalize to new data. Given infinite data, your model would be exposed to every possible aspect of the data distribution at hand: you would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by augmenting the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by the `ImageDataGenerator` instance.

Listing 3: Setting up a data augmentation via ImageDataGenerator

```
datagen = ImageDataGenerator(
rotation_range=40,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.2,
zoom_range=0.2,
horizontal_flip=True,
fill_mode='nearest')
```

6 Using a Pretrained Convolutional Base

Some research teams have trained convolutional neural networks on much larger datasets. We have seen during the lecture that the networks can model *conceptual patterns* as they go through the layers. This was identified by Le Cun in his first experiments `http://yann.lecun.com/exdb/lenet/`. In this last part, you will train classifiers on top of a pretrained convolutional base.

- Build a network that consists of the Inception V3 convolutional base and two dense layers. As in Chollet, Listing 5.17, you will program an `extract_features()` function.
- Train your network and report the training and validation losses and accuracies.
- Apply your network to the test set and report the accuracy as well as the confusion matrix you obtained.
- Modify your program to include an image transformer. Train a new model.
- Apply your network to the test set and report the accuracy as well as the confusion matrix you obtained. you need to reach an accuracy of 80 to pass.

A common and highly effective approach to deep learning on small image datasets is to use a *pretrained network*. A pretrained network is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task.

There are two ways to use a *pretrained network*: feature extraction and fine-tuning. Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.

Listing 4: Extracting features using the pretrained convolutional base

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20
def extract_features(directory, sample_count):
features = np.zeros(shape=(sample_count, 4, 4, 512))
labels = np.zeros(shape=(sample_count))
generator = datagen.flow_from_directory(
directory,
target_size=(150, 150),
batch_size=batch_size,
class_mode='binary')
i = 0
for inputs_batch, labels_batch in generator:
features_batch = conv_base.predict(inputs_batch)
features[i * batch_size : (i + 1) * batch_size] = features_batch
labels[i * batch_size : (i + 1) * batch_size] = labels_batch
i += 1
if i * batch_size >= sample_count:
break
return features, labels
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir,
    1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

7 Appendix

References

[1] VanderPlas, A Whirlwind Tour of Python. O'Reilly 2016. Online reading: `https://jakevdp.github.io/WhirlwindTourOfPython/`

[2] Pierre Nugues, EDAN95 github: `https://github.com/pnugues/edan95`

[3] Geron, Jupyter notebook on linear algebra from Machine Learning and Deep Learning in Python using Scikit-Learn, Keras and TensorFlow: `https://github.com/ageron/handson-ml2`

[4] Kevin P. Murphy: Machine Learning, "A Probabilistic Perspective". MIT Press, 2012, ISBN: 9780262018029.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville: "Deep Learning". MIT Press, 2016, ISBN: 9780262035613.

[6] Aurélien Géron: "Hands-On Machine Learning with Scikit-Learn and TensorFlow, Concepts, Tools, and Techniques to Build Intelligent Systems". O'Reilly Media, 2017, ISBN: 9781491962299.

[7] François Chollet: "Deep Learning with Python". Manning, 2018, ISBN: 9781617294433.

[8] Tom Mitchell: "Machine Learning". McGraw Hill, 1997, ISBN: 0070428077.

[9] David L. Poole, Alan K. Mackworth: "Artificial Intelligence - Foundations of Computational Agents (2e)". Cambridge University Press, 2017, ISBN:9781107195394.

[10] Stuart Russel, Peter Norvig: Artificial Intelligence - A Modern Approach (3e). Pearson, 2010, ISBN:10: 0132071487

[11] Richard Johansson, "Scientific Computing with Python". `https://github.com/jrjohansson/scientific-python-lectures`

[12] Yann LeCun, Marc'Aurelio Ranzato. Deep Learning Tutorial, ICML, Atlanta, 2013-06-16 [Online]. `http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf`

[13] Kaggle Kaggle Datasts, Flowers Recognition [Online]. `https://www.kaggle.com/alxmamaev/flowers-recognition`

[14] Andrea Vedaldi, Andrew Zisserman. VGG Convolutional Neural Networks Practical [Online]. `http://www.robots.ox.ac.uk/~vgg/practicals/cnn/`

[15] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-Based Learning Applied to Document Recognition.* PROC. OF THE IEEE, November 1998.