COSC401 – Machine Learning (2016S1): Assignment 2

Applying Convolutional Neural Networks to Pest Images

Value: 28% of final grade
Due date: 5pm Friday 27th May 2016 (15% penalty for submissions after the due time)
Drop dead date: one week after the due date

The assignment must be your own individual work.

Introduction

The Cacophony project aims to bring back the chorus of native bird song that Captain Cook might have experienced when he visited NZ in 1769, described by his botanist as "the most melodious wild music I have ever heard". One way to do this is to improve our ability to keep predators away. This project aims to help by automatically identifying animals from video footage.

The task

Your mission is to apply convolution neural networks (CNNs) to video footage that has been converted to a sequence of images. The videos have been taken in an uncontrolled environment, which poses significant challenges. You will research ways to maximise classification performance through techniques such as input data selection and downscaling, and optimisation of the CNN architecture used. You will also visualise the learned models to analyse what a CNN actually learns, and what effect the "convolutional" layers have on learning performance.

Convolution Neural Networks – LeNet

Convolutional neural networks differ from traditional feedforward networks in that they contain one or more "convolutional" layers that pre-process the image:

- A convolutional layer produces one or more "feature maps" – a transformation of the original image produced by passing a (learned) filter over it.
- Each feature map has an associated filter bank for each input feeding into it. For the initial layer (layer 0) there is just one filter bank per feature map to filter the input image; for subsequent layers for each feature map there is one filter bank per input, where each input is a feature map from the previous layer. A filter bank is a small filter that learns localised patterns; to "convolve" an image the filter bank is scanned over the input to produce each pixel of the output.
- A "max pooling" layer down-scales each feature map by outputting the strongest value in each 2x2 square, thus reducing the image to ¼ its original size and retaining the strongest features.

You have been given a simple implementation of CNN called LeNet (LeCun et al, 1998). Further details can be found below. See also http://deeplearning.net/tutorial/lenet.html for a tutorial on how it works. LeNet is implemented using the Theano python library; see Appendix 2 for a brief introduction to Theano and how we use it.

The input data

The raw input data from the cacophony project consists of 161 short videos, grouped into categories of "possum", "rat", "stoat" and "other". These have been converted into sequences of still images of

size 640x480. In theory these could simply be used as the training/test data, but there are some problems:

1. The number of inputs into the CNN (640x480 = 307,200) requires an enormous network with a massive number of parameters, which will be slow to train and highly prone to overfitting
2. There are a lot of images – roughly 500 per video or around 60000 in total
3. Image quality varies widely, both between and within videos
4. More than one camera has been used, raising the possibility that the CNN will learn extraneous features related to camera setup
5. The animals don't stay still and smile into the camera; your CNN will need to cope with this
6. For a given video, not all images contain the animal! In the worst case the animal is present for only a very small number of frames; in others there are two or more animals (thankfully of the same kind!) in the image.

Two versions of the images have been produced: the raw images and a set that has been down-scaled by a factor of 10 (i.e. 1/100th the original number of pixels). You will use this latter set to get started.

The sample code

The sample code consists of the following:

- cacophony.py – implements a LeNet CNN consisting of two convolutional layers, a fully connected hidden layer and a final layer that uses logistic regression with stochastic gradient descent to decide the final category. When run this program trains the network on a small set of sample images (in folders train, valid and test), outputs the results, and then displays the following:
    - the first image in the test data set, and the outputs (i.e. feature maps) from the first convolutional layer, as images;
    - the two convolutional layer filter banks as images.
- convolutional_mlp.py – implements the (convolutional + max pool) layer of the model
- mlp.py – implements a multi-layer perceptron. Used for the fully connected hidden layer
- logistic_sgd.py – implements the logistic regression layer
- data.py – loads the pest data into the appropriate data structure for the CNN. ***Note: this code is a sample only; you will need to modify it to pass the correct target vectors for your chosen training, test and validation sets***
- images.py – basic image processing. Not used by the project but might be a useful starting point if you want to adjust the images in any way.
- wolf.py – test program that builds a convolutional layer and runs it over a test image. Outputs the original image and the output from two feature maps (each using a random filter bank)
- possum.py – same as wolf.py but uses one of the small pest images. Used to verify that your setup can read the images correctly.

Required software

This project requires a large number of python modules to be installed. The departmental lab machines have all required software installed, but if you want to use your own machine you will need to install the environment yourself - see Appendix 1.

Fine-tuning

You've installed the software and run it. Cacophony.py successfully trains a network from the images, but it performs relatively poorly. What next? Here are some things to try:

- Use (a lot) more images. Consider carefully which images to use. Since some of the images are in fact empty (i.e. there's no animals in them), you probably want to either exclude them, or relabel them as "other", or perhaps "empty". This will help your model to learn what a possum is, not what the background of the possum images is, for example. Ideally you want a good range of images that capture the animal in various positions, scales, poses and lighting.
- Use bigger images. The 64x48 images are very small and will have lost a lot of information, but the original large images may not be feasible. Consider using **images.py** to generate your own set of images at an appropriate size. Note that **cacophony.py** has image sizes hard-coded; you will either need to change these or parameterise the code and pass the image size in (or compute it).
- Make changes to the architecture of the network (in **cacophony.py** - *evaluate_lenet5*), for example:
    - Number of convolution layers
    - Number of feature maps per convolutional layer
    - Size of convolution filters (possibly different per layer)
    - Number of hidden units in the fully connected layer
    - Other parameters (learning rate, number of epoch, batch size)
- Pre-process the images, e.g. (extra for experts)
    - Background subtraction
    - Normalise brightness (e.g. lighten or darken to make the average constant)
    - Merge pairs of images to capture movement

Classification tasks

There are three classification tasks; you must attempt all three.

1. Classify each individual image according to whether it contains a possum, rat, stoat or other. You may optionally split "other" into "other" and "empty" if this helps you improve performance
2. Classify an entire movie clip. A movie is classified as "possum" if *any* of the images contain a possum, etc. Given that your image classifier is likely to produce a mixture of answers for each movie, you will need some way of selecting the final result based on the set of image classifications the model produced. This can be as simple (e.g. a vote) or as complex (e.g. logistic regression, stacking) as you wish.
3. Indicate which sections of a given movie contain an animal. Note that you will need to deal with noisy output from your classifier, similarly to 2.

For each task you need to train the model, and then test it on unseen examples. You may use whatever train, validation and test set sizes you like, but all images must be used (i.e. train + valid + test = the entire set of images) for task 1 and, similarly, all movies must be used for tasks 2 and 3. Note that, because multiple cameras have been used, the selection of train and test examples may have a significant influence on the outcome.

Understanding your model

Deep Learning is supposed to produce abstractions of the original data that provide more robust classification, but does this really happen? To answer this you need to inspect the learned model. There are two ways to do this:

- Visualised the filter weights
- Visualise the feature maps

The filter weights may form a recognisable shape, such as a horizontal line. **Cacophony.py** contains a routine, **display_conv_filters**, which attempts to do this. Does your model produce filters that make sense? How do they help it to perform the classification task? Alternatively, if the filter maps just look like noise, do they actually add any benefit? What happens if you simply remove (or disable) the convolutional layers altogether?

The feature maps are potentially more interesting: they show us what the filter's effect is on the image, so we can see what the model is "concentrating" on. Again, **cacophony.py** includes a sample function, **display_output**, which displays the feature map outputs of the first convolutional layer for the first image in the test set. You may wish to adapt this function to display the outputs from the other layers too. What do they tell you about the model that has been learned?

Marking scheme

Your final assignment mark will be based on a written report. This will contain:

- A description of what you have implemented, and the experiments you performed
- A presentation of your final results and an analysis of how the various tuning parameters or modifications affected the final accuracy and computational time
- Visualisation and analysis of the model learned, particularly by the convolutional layers
- A discussion of the quality of your final results and the performance of CNN.

You receive mark based on the content of the report (and not the code). However, your code will be inspected to verify your claims. You may lose mark if your code is difficult to read, or may not receive any mark if the code for what you claim in the report does not work.

Submission

You need to submit **two** files through Learn by the due date:

1. The pdf version of your report. Please have your name and ID written on the first page.
2. A zip file containing all your programs (and any nonstandard libraries that it depends on).

References

1. LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton: *Deep learning*, Nature 521.7553 (2015): 436-444. doi:10.1038/nature14539
2. Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document *Recognition*, Proceedings of the IEEE, 86(11):2278-2324, November 1998. http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

Appendix 1: software installation

This project requires the installation of Theano, which has a lot of library dependencies, plus a few other libraries. The simplest way to do this is to install Anaconda, an open source analytics environment for python that comes bundled with all of the other libraries - https://www.continuum.io/

For windows users, the process is as follows - based on https://blogs.msdn.microsoft.com/lukassteindl/2015/11/23/deep-learning-simple-installation-guide-for-theano-on-windows/

1. Download Anaconda 64 from https://www.continuum.io/downloads and install it. Note that by default it makes Anaconda your default python environment. I didn't want this so on my PC I unchecked "make default", hence I provide the path to python.exe when running it
2. Install mingw and libpython (mingw is minimalist GNU for Windows):
   ```
   > conda install mingw libpython
   ```
3. install theano into anaconda
   - download Theano from https://github.com/Theano/Theano/releases/
   - Unzip, then run the setup:
     ```
     > (your Anaconda bin path here)\python setup.py install
     ```
4. Confirm Theano has installed correctly:
   ```
   > (your Anaconda bin path here)\python
   > import theano
   ```
5. Run the program possum.py to confirm all dependencies are present
   ```
   > (your Anaconda bin path here)\python possum.py
   ```
6. Run the program cacophony.py to verify you're ready to go:
   ```
   (your Anaconda bin path here)\python cacophony.py
   ```

Non-windows users: see http://deeplearning.net/software/theano/install.html but note that it's not for the faint-hearted. You *may* find it easiest to use Anaconda.

Appendix 2: a quick guide to Theano

"*Theano is a Python library that allows you to define, optimize, and efficiently evaluate mathematical expressions involving multi-dimensional arrays. It is built on top of NumPy*"
- https://pypi.python.org/pypi/Theano

The LeNet CNN we are using is written using Theano. In very general terms, Theano is used as follows:

- All data is represented as Theano tensors (n-dimensional arrays, where n is arbitrary). For convolutional neural networks on images, it is possible to perform updates (e.g. gradient descent) over an entire convolutional layer, for all inputs, by representing the layer as a 4-dimensional (4D) tensor, where the dimensions are (number of input images, number of feature maps to be computed, image height, image width). Thus during convolution each image computes a data cube whose height and width is (roughly) that of the image, and the "depth" is the number of feature maps in the layer. Similarly, the weights into a convolutional layer form a 4D tensor of dimensions (number of feature maps to be computed; number of inputs from the previous layer – 1 for the input layer, number of feature maps in the previous layer for subsequent layers; filter bank height; filter bank width).

- To perform an operation, we define an expression over the input that indicates what we want to compute, and then declare a function that maps the inputs to the outputs. For example (from possum.py):

```
input = T.tensor4(name='input')
conv_out = conv2d(input, W)
output = T.nnet.sigmoid(conv_out + b.dimshuffle('x', 0, 'x', 'x'))
f = theano.function([input], output)    # The input is a list of sources)
```

- A slightly more complex function definition is used in **cacophony.py::evaluate_lenet5** to define the train, test and evaluation functions. These functions accept as input the batch number, and use this to call the underlying function by passing "givens", which substitute an unbound Theano variable (x and y in this case) for the input values required. For example, the function **test_model** substitutes x for the set of input images in the required batch. If you look carefully at the code for **evaluate_lenet5** you will see that layer 0 gets its input from the variable x, so **test_model** has used the "givens" to tell Theano what x is bound to during evaluation. Further, **evaluate_lenet5** declares an expression "updates", which tells Theano how it wants the model to be modified during training; the function **train_model** includes an "updates" parameter that binds to this expression, which causes the required updates to occur during evaluation of the function.
- The Theano library also implements various statistical and machine learning functions. For example, we make use of **theano.tensor.nnet.conv2d** to perform the convolution operation over an image, **theano.tensor.signal.downsample.max_pool_2d** to perform the max pooling, and **theano.tanh** to perform the "squashing" of the computed values.
- Data is typically passed into Theano by declaring a "shared variable". This wraps the pure python data (e.g. an array) in a Theano shared constant of the appropriate type. For example, in possum.py the following statement creates a shared Tensor variable containing image data in an array (the **asarray** function first converts the list of images into a numpy array):
  imagesTensor = theano.shared(numpy.asarray(images))
- Theano can be configured to run on GPUs to accelerate its performance; if you're keen you might like to experiment with this ☺

For more information see http://deeplearning.net/software/theano/tutorial/