SCA Laboratory
Lab5: Convolutional Neural Networks

C. Álvarez and J.R. Herrero

Fall 2024

Contents

Note: The main purpose of this laboratory is to give you a deeper understanding of some the algorithms explained in the theoretical sessions concerning Convolutional Neural Networks (CNN).

You will have to deliver your report in a PDF file at the "Racó". There are some questions on the document that will guide you to elaborate it. Note that each of the **questions** includes several sub-questions. Please, answer them all. In addition to the pdf with your report, please, submit also at Racó the final file which defines your modified network in section 5.5.

Laboratory 5

Convolutional Neural Networks

The objective of this laboratory is to understand different ways of improving the performance of the execution of Convolutional Neural Networks (CNNs). As it has been explained in theory classes, CNNs are Deep Neural Networks whose main algorithmic computation is composed by Convolutional Layers whose main computation is the convolution of a (set of) filter(s) with the corresponding input activations. You should refer to the "DNN" set of slides for more information.

In this laboratory you will use the same development environment as in the first laboratory (lab1). For connection and execution details please refer to Laboratory 1 documentation.

All files necessary to do this laboratory assignment are available in `/scratch/nas/1/sca0/sessions`. Copy `lab5.tar.gz` from that location to your home directory in `boada.ac.upc.edu` and decompress it with this command line: `"tar -zxvf lab5.tar.gz"`.

5.1 Configuring and compiling Caffe

In this session we are going to work with the Caffe deep learning framework[1] Caffe is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. Yangqing Jia created the project during his PhD at UC Berkeley. Currently Caffe is released under the BSD 2-Clause license. Although there are more modern deep learning environments, Caffe has the advantage of being written in C++ and being reasonably limited in size. In fact, the files that you have copied are a subset of the whole available Caffe framework at github (some files have been deleted for storage reasons).

Important Note: Regrettably, Caffe uses old versions of some libraries which are currently unavailable in `boada`. Consequently, we've had to compile them ourselves and we will need to set up the environment variable `LD_LIBRARY_PATH` properly so that we can use such auxiliary library needed by `Caffe`. To this end you simply have to execute the command `"source env.bash` in directory `caffe-master`. Remember to do this each time you enter your account in `boada`. Also, if you prepare an shell script for launching an execution to the queues, remember to set the environment variable accordingly. However, contrary to previous assignments, for this one we should NOT source the `environment.bash` file with the command line `"source environment.bash"` in order to set all necessary environment variables. Thus, if you included this command in your .bashrc file, edit it to comment it out, exit the session and log in again. Otherwise a mixture of compilers and libraries will make the compilation and linking of libraries fail.

First of all you need to compile the Caffe framework in `boada`. Although this should be easily accomplished by simply doing `make`, some configuration is necessary. For instance, changes are needed due to the lack of HDF5 file format support in `boada`. Normally one would copy the config example file `Makefile.config.example` to the actual config file: `Makefile.config` and customize it. We have prepared this step for you so that the proper path for finding libraries and header files is prepared. In addition, we have changed the following lines in the new `Makefile.config` file:

[1] Available at `https://github.com/BVLC/caffe/tree/master`.

- Uncommented the `CPU_ONLY := 1` option. Although Caffe supports GPUs by default we are initially going to work with the CPU only version to have faster compilation times and less programmability issues. Later you will be able to compare the execution time with a precompiled version which uses the GPU.
- Uncommented the `USE_HDF5 := 0` option. As commented we do not have support for this file format in `boada`.
- Uncommented the `USE_OPENCV := 0` as there are some backwards compatibility issues with OpenCV and using it is not really necessary for our tests.

Regrettably, there is still an unsolved error in the HDF5 configuration that we also need to solve. We need to open the file `src/caffe/net.cpp` and in function `CopyTrainedLayersFrom` al line 772, change the code from:

```
  if (H5Fis_hdf5(trained_filename.c_str())) {
    CopyTrainedLayersFromHDF5(trained_filename);
  } else {
    CopyTrainedLayersFromBinaryProto(trained_filename);
  }
```

to:

```
#ifdef USE_HDF5
  if (H5Fis_hdf5(trained_filename.c_str())) {
    CopyTrainedLayersFromHDF5(trained_filename);
  } else //{
#endif  // USE_HDF5
    CopyTrainedLayersFromBinaryProto(trained_filename);
//  }
```

Again, we have already done this change so that you do not need to go through this. (You can see the original content of the file in `src/caffe/net.cpp_orig`.)

Now that you're aware of the changes with respect to the files found in the distribution, compile it with `make`. Caffe should compile smoothly, though it will take quite a few minutes. If it fails to find the `glog` library read and process again the important note at the beginning of this section. There are up to 2 cores available at the interactive nodes in boada. Thus, you may compile in parallel with the option `-j` of `make`.

5.2 Loading the MNIST dataset

Although Caffe can be used to work with a large set of different CNNs, in this laboratory we are going to settle for the small LeNet NN. As you probably already know, LeNet is known because it works well on digit classification tasks. However, LeNet is, compared with other CNNs both small and fast while at the same time has all the main components present in nearly all other, more advanced NNs.

The next step in order to test a Neural Network is to get a good dataset to work with it. LeNet is mainly used with the `MNIST` dataset that has the advantage of being fast to download and use a small amount of storage due to its limited size. The process of downloading the dataset and converting it to the appropriate format is automatized in Caffe. Execute the following commands from the Caffe root location[2].

```
./data/mnist/get_mnist.sh
./examples/mnist/create_mnist.sh
```

You should get an output that indicates that you have 60000 train images and 10000 test images.

[2]Remember that if you get an error while loading shared library libglog.so.1 it means that you should read and follow the instructions in the important note at the beginning of section 5.1.

5.3 Configuring and executing the LeNet NN in Caffe

Now, we can look at the configuration of LeNet in Caffe. As commented before, LeNet contains the essence of CNNs that are still used in larger models such as the ones in ImageNet. In general, it consists of a convolutional layer followed by a pooling layer, another convolution layer followed by a pooling layer, and then two fully connected layers similar to the conventional multilayer perceptrons. The layers are defined in file: `examples/mnist/lenet_train_test.prototxt`.

Open the file and take a look at the layers definitions. You will see the input data layer specified first with its two variants: the TEST and TRAIN phases. Apart from the input file and the batch size they are essentially the same: both have the `type Data` that is how the first layer is called, and they scale the input values by 0.00390625 that is 1/256 to get a value from 0 to 1.

In order to understand the definition of the layers, you have to take into account that in Caffe, values flow from the bottom to the top, so the input layer does not have a bottom layer (because it essentially reads data from a file) and generates two outputs (the `top` values): `data` and `label` (in Caffe the set of values is called `blob`[3]). The `data` value is used by the next layer while the `label` value is going to be used for training.

After the `Data` layer, you can see the first `Convolution` layer. It reads data from the bottom `data` layer and generates the `conv1` layer. The layer has 20 5×5 filters that are applied with stride 1[4]. You can follow the description of the remaining layers[5]. Note that a Fully Connected Layer is referred to as of type `InnerProduct`[6].

Question 1: *How many layer types are there in the `LeNet` definition? Which ones?*

As you can see, what is coded is a slightly different version from the original LeNet implementation described in the course slides, replacing the sigmoid activations with Rectified Linear Unit (ReLU) activations for the neurons[7] . But, what is an activation function? An *Activation Function* is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data. It decides whether a neuron should be activated or not.

Question 2: *The purpose of an activation function is to add non-linearity to the neural network. Why is it needed? (Search for it and provide a brief answer.)*

Question 3: *Do a search and list some other types of activation functions used in machine learning.*

Weight initialization is important[8]. Is random initialization being used? You may have noticed that some layers show a `weight filler` of type `xavier`.

Question 4: *What is Xavier weight initialization? Is it the best possibility if we are using ReLU? (You don't need to change it)*

The other key component of the NN is the solver. The solver for `LeNet` is defined in the file `examples/mnist/lenet_solver.prototxt`. The meaning of the values is explained in the comments in the file. As you can guess, you have to change the `solver_mode` to CPU before executing (as we have compiled Caffe without GPU support). Also, for interactive execution you have to reduce `max_iter` to 2500. Now, we can test how the NN works. For this, we will execute the command `./examples/mnist/train_lenet.sh` possibly prepended with the `time` command to get a summary of the total execution time. Also, you might be interested in keeping the output in a file. In summary, you could run the execution as:

```
/usr/bin/time ./examples/mnist/train_lenet.sh |& tee train_lenet.out.txt
```

[3] You can get more information at `https://caffe.berkeleyvision.org/tutorial/net_layer_blob.html`.

[4] `https://caffe.berkeleyvision.org/tutorial/layers/convolution.html`

[5] `https://caffe.berkeleyvision.org/gathered/examples/mnist.html`

[6] `https://caffe.berkeleyvision.org/tutorial/layers/innerproduct.html`

[7] Since ReLU is an element-wise operation, we can do in-place operations to save some memory. This is achieved by simply giving the same name to the bottom and top blobs. Of course, do NOT use duplicated blob names for other layer types!

[8] `https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks`

If you later want to execute larger experiments you will need to send large jobs, specified in a shell script, to the queue using `sbatch`.

Question 5: *How much time has taken the network to train? Which has been the accuracy obtained? And the loss?*

After that you have your network trained. Look at the new files generated at the `examples/mnist/` directory. You should find the network model file (`<name>.caffemodel`). We don't need this now, but if we had executed with a number of iterations larger than 5000 we would also find a *snapshot* file (`<name>.solverstate`). The snapshots can be used to resume training after a certain point. For instance:

```
caffe train -solver examples/mnist/lenet_solver.prototxt -snapshot examples/mnist/lenet_iter_5000.solverstate
```

The model is used to test the network without having to train again. Test the network now:

```
time ./build/tools/caffe test -model examples/mnist/lenet_train_test.prototxt \
   -weights examples/mnist/lenet_iter_2500.caffemodel -iterations 100 |& tee test.out
```

Question 6: *Which is the size of the `LeNet` model that we are working with (size of the file with name ending in `.caffemodel`)? How much time does it take to test the model?*

Another interesting characteristic of Caffe is that it has some integrated profiling capabilities that give you an idea of your network performance. Try the following command:

```
./build/tools/caffe time -model examples/mnist/lenet_train_test.prototxt -iterations 10
```

You will see the average time it takes to compute a Forward or Backward step for each layer.

Question 7: *What percentage of time does it take to train the convolutional layers in `LeNet`?*

It is also interesting to see how the network computes a layer. In fact, if you want to make the network perform faster, you can change the actual code that does the computation. You can search the code to find how the layers are computed, but a good point to start looking is file `src/caffe/layers/relu_layer.cpp` (in fact, each of the files in the directory `layers` contain the code of the corresponding layer). As you can see there are two functions in each file, the `Forward_cpu` and the `Backward_cpu`. The former is used for testing while the latter is used in order to train the network. As you can see the `Forward_cpu` only computes the result of the layer while the `Backward_cpu` version computes the difference introduced by the layer in order to adjust the weights in the training phase. With this organization it is pretty straightforward to introduce new layers in the framework.

Question 8: *Is de code for the `ReLU` layer parallelized?*

Question 9: ***(Optional)*** *What is the optimization level introduced by the compiler when compiling `Caffe`? Can you improve the `Makefile` to get better automatic optimization? How? Do it and profile the Network training again. Has the percentage of time taken by the convolutional layers changed?*

5.4 Computing the convolutions

As you have already seen in this laboratory, the convolutions are the main computation part of LeNet and this effect increases dramatically in more modern CNNs. When Caffe was designed the author tried to make it fast with a "quick and dirty" approach. You can look at their thoughts in `https://github.com/Yangqing/caffe/wiki/Convolution-in-Caffe:-a-memo` (We think that it is worth spending 5 minutes reading it). You can look at the code of the convolutional layer in file `src/caffe/layers/conv_layer.cpp` and follow the call to function `forward_cpu_gemm` until you get to file `src/caffe/util/math_functions.cpp` and reach the following code:

```
template<>
void caffe_cpu_gemm<float>(const CBLAS_TRANSPOSE TransA,
    const CBLAS_TRANSPOSE TransB, const int M, const int N, const int K,
    const float alpha, const float* A, const float* B, const float beta,
    float* C) {
  int lda = (TransA == CblasNoTrans) ? K : M;
  int ldb = (TransB == CblasNoTrans) ? N : K;
  cblas_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha, A, lda, B,
      ldb, beta, C, N);
}

template<>
void caffe_cpu_gemm<double>(const CBLAS_TRANSPOSE TransA,
    const CBLAS_TRANSPOSE TransB, const int M, const int N, const int K,
    const double alpha, const double* A, const double* B, const double beta,
    double* C) {
  int lda = (TransA == CblasNoTrans) ? K : M;
  int ldb = (TransB == CblasNoTrans) ? N : K;
  cblas_dgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha, A, lda, B,
      ldb, beta, C, N);
}
```

So, eventually, we reach the same problem as (almost) always, how to do matrix multiplications faster. You can introduce some counters in the code in order to get a proper statistic of the problem (and print them using the `LOG(INFO) << <data>;` command). **Hint:** You can print all three values of M, N and K in the same line. Next, using `Unix` standard commands you can filter the information to quickly be able to fill in the table. To this end, consider using commands `sort` and `uniq -c`.

Question 10: *Which data type (`single` or `double`) is `Caffe` using?*

Question 11: *How many times is the `GEMM` call executed? Which is the size, or dimensions M, N, K, of the matrices multiplied? Show a table with the number of times each combination of M, N and K appears.*

Question 12: *Would it make sense that these matrix multiplications were performed in `half` floating-point precision (16 bits)? Justify your answer.*

5.5 Stacking filters

Although "Stacking Filters" is a technique that was developed after `LeNet` and is not really adequate to such a (from the current perspective) small network, it is very easy to apply it to `LeNet`. It can be done by decomposing a 5×5 filter into two consecutive 3×3 filters or even into a 5×1 and a 1×5 ones (the latter option can be accomplished by using `kernel_h` and `kernel_w` parameters instead `kernel_size` as Convolution layer parameters). Change the code to replace the first of the two convolutional layers by a set of two stacked filters.

Question 13: *Copy the code of the new layers introduced*

Now profile the training of the network again.

Please, in addition to the pdf with your report, submit also at Racó, as a separate file, the final version of the file which defines your network.

Question 14: *How has the number of calls to matrix multiplication changed? And the time to compute one training step? And the size of the model?*

Question 15: *How has the time to train the Network changed? And the accuracy obtained?*

5.6 Training with a GPU

We have compiled `caffe` so that it can use the GPU available at `boada-10`. We have prepared a script which will use that binary instead of the one in your directory. However, you must execute the command from the root of the `caffe-master` subdirectory, i.e. the very same directory where you were executing the previous commands in order to read the configuration files which define the network and input data that you want to use for training and testing.

Recall that the solver for `LeNet` is defined in the file `examples/mnist/lenet_solver.prototxt`. The meaning of the values is explained in the comments in the file. As you can guess, you have to change the `solver_mode` to GPU (`solver_mode: GPU`) before executing (as we have a version of Caffe compiled with GPU support). Also, for interactive execution we had reduced `max_iter` to 2500. Now, it's time to set that value back to the original 10000 (`max_iter: 10000`). Once you've made these two changes to the configuration file for `LeNet` solver we can test how the NN works on the GPU. For this, you will execute the command `sbatch ~sca0/sessions/ML/bin/job_caffegpu.sh`. In file `train_lenet_gpu.jobid` you will get a summary of the execution time for each phase. In file `out-test_caffe_GPU.ejobid` you will have the detailed output.

Question 16: *Using 10000 iterations, which are the execution time, loss and accuracy when training in the GPU?*

Question 17: *Using 100 iterations, which are the execution time, loss and accuracy when testing?*

Question 18: *Is the GPU used for testing? Would it be necessary in this case?*

Are you're curious about the time taken using the previous problem size? In any case it's worth comparing the CPU and the GPU time. You can set `max_iter: 2500` again and run the command `sbatch ~sca0/sessions/ML/bin/job_caffegpu2500.sh`.

Question 19: *Which are the time, loss and accuracy obtained for 2500 iterations trained on the GPU? Compare them against the ones provided by the initial execution in the CPU.*

5.7 Making changes to the network

One of the advantages of having a framework is that it is pretty easy to make small changes in the network without having to program the change yourselves. We are going to test the effect of using the original `sigmoid` activation function in the network. To ease your work we have prepared a network which uses the `sigmoid` activation function instead of ReLU. This has been defined in two new files, namely `lenet_train_test_sigmoid.prototxt` and `lenet_solver_sigmoid.prototxt`, both within subdirectory `examples/mnist`. We invite you to inspect them. Testing this new definition of the network on the GPU with 10000 iterations can be done by executing the following command from the root of the `caffe-master` subdirectory: `sbatch ~sca0/sessions/ML/bin/job_caffegpu_sigmoid.sh`.

Question 20: *Changing the `ReLU` layer by a `Sigmoid` layer, how has the training time changed? And the accuracy and loss?*

Question 21: ***(Optional)*** *Inspecting the source code and the caffe tutorial, do you feel like testing other changes to the network? Or try other examples? If positive, please report your experiments and findings.*

5.8 Conclusions

Question 22: ***Conclusions:*** *once the assignment is complete and you have a global view, please draw some conclusions.*

Note: conclusions are not a summary of what has been done. Instead, you should highlight your insights.

5.9 Further possibilities

If you have more interest, here are some ideas involving support for DNN from vendors or researchers which you could explore:

- Get the Intel variant of Caffe (`git clone https://github.com/intel/caffe.git`) and test if `mkldnn` improves performance.
- The current version of `cuDNN` seems incompatible with `caffe`. You could try other more recent frameworks which can exploit it.
- Study efficient convolution algorithms and report what you found out.

Enjoy!