P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 7 - Image classification

Issued: November 9, 2018
Hand in (optional): November 16, 2018 23:59

In this exercise we will continue development of the deep-learning library. We will define the operation `conv2d` at the core of convolutional neural networks (CNN) and we will use CNN for their originally intended purpose: image classification. Specifically, we will learn to recognize the digits of the MNIST dataset (figure 1). Each element of MNIST is an image of 28 by 28 pixels in gray-scale (1 "color channel", as we shall see) which represents a handwritten digit between 0 and 9.

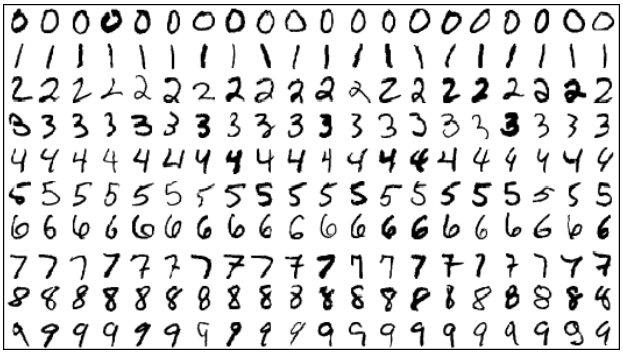

Figure 1: Examples from the MNIST dataset

Similarly to the non-linear network of the previous exercise, the convolutional layers of a CNN typically alternate the operation `conv2d` and non-linear element-wise operations. We will focus on `conv2d`, which treats its input I like an image of size InY by InX pixels and InC color channels. The convolution is performed by "applying" a number KnC of filters onto the input image. Each filter is a matrix of size $(KnY \times KnX \times InC)$ of parameters which can be learned to minimize a loss function. The sizes of the filters are usually smaller numbers than the image sizes.

`conv2d` is performed by moving each filter K across the input image. At regular intervals along the x and y axes of I, the convolution is performed by computing the inner product between K and the patch of $(KnY \times KnX \times InC)$ pixels of I. The output of each inner product defines the value of a color pixel in the output image. In the tutorial slides we go in more detail about convolutions. We introduce: padding (extending the input image with Px and Py 0-valued pixels along x and y respectively), striding (moving the point around which the convolution is performed by Sx pixels in x and Sy in y). These techniques define the size of the output image: $OpY = (InY - KnY + 2Py)/Sy + 1$ by $OpX = (InX - KnX + 2Px)/Sx + 1$ by KnC.

Each kernel acts like a feature detector. If a patch of the input matches the pattern encoded in the kernel's weights, the result from the inner product will be a larger number. This activation

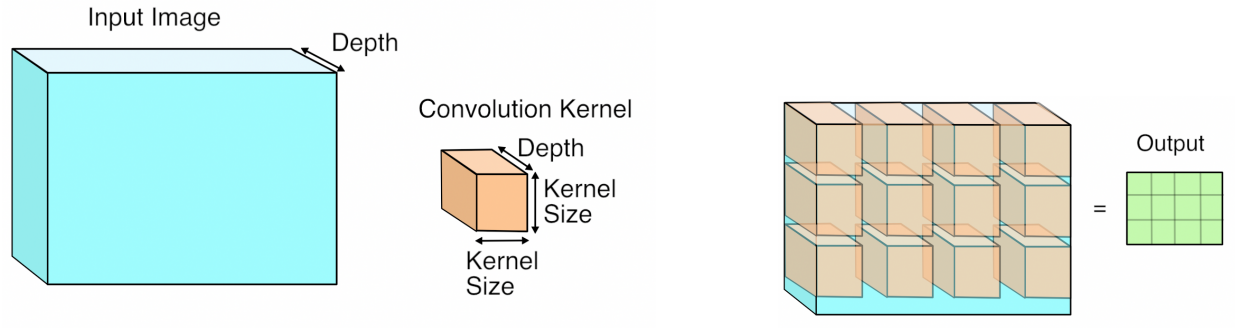

Figure 2: Visualization of the `conv2d` operation. The convolution kernel contains the parameters of `conv2d`. The kernel K is moved along the width and height of the input (I). Each position assumed by K corresponds to one pixel of the output O, which is computed through scalar product between K and the local patch of I.

will be stored in a pixel of the output image on the "color channel" corresponding to the filer, signifying that the pattern was found at a certain x and y coordinate.

After some convolutional layers, the output of the CNN will be a vector of 10 numbers encoding the predicted probability that the input is a digit in the dataset. Probabilities are bound between 0 and 1, and they should sum to 1. Since the outputs of the CNN are unbounded, we map the output vector to a probability space with the SoftMax function:

$$h_i^{(K)} = f(h_i^{(K-1)}) = \frac{\exp h_i^{(K-1)}}{\sum_{j=1}^{10} \exp h_j^{(K-1)}} \tag{1}$$

Here, $h_i^{(K)}$ is one output value of the last (K-th) layer associated with digit i. We aim to minimize the distance between the probabilities of an image being each possible digit predicted by the CNN and the true probabilities. Of course, because we know which digit is represented by each image, the true probabilities are 1 for the digit represented by the image, and 0 for the other digits. The loss function usually considered for classification problems is the cross-entropy, which measures the dissimilarity between two probability distributions:

$$H(\tilde{\mathbf{P}}, \mathbf{h}^{(K)}) = \mathbb{E}_{\mathbf{x}\sim D}\left[-\sum_{i=1}^{10} \tilde{P}_i \ \log h_i^{(K)}\right] \tag{2}$$

Here, $\tilde{P}_i = 0$ or 1 is the true probability of an image being digit i and $\log h_i^{(K)}$ is the predicted probability. The expectation denotes that we want to minimize the loss over the samples contained in the dataset D.

The skeleton code provided with this exercise contains many of the steps required to train the CNN model. These steps should be familiar to you from Ex.06. The data is read, the network initialized, and mini-batches are fed to the CNN. Network outputs are computed for the entire mini-batch. From the output we compute the error and the gradient of the error. Through back-propagation we compute the mini-batch estimate of the gradient of the loss w.r.t. the network weights. The gradient estimate is used to update the weights with some stochastic gradient descent algorithm. This time, optimizer and non-linearities are already implemented. We advise you to do Ex.06 before starting this one. However, you do not require any code written for Ex.06 to begin this one.

Question 1: Implement `conv2d` with GEMM

Convolutions require two things: some specialized algorithm to step across an input image, and many inner products. Most deep learning libraries rely on GEMM for the inner product. The intuition behind this technique is best explained with visual aid and therefore we refer to the tutorial slides.

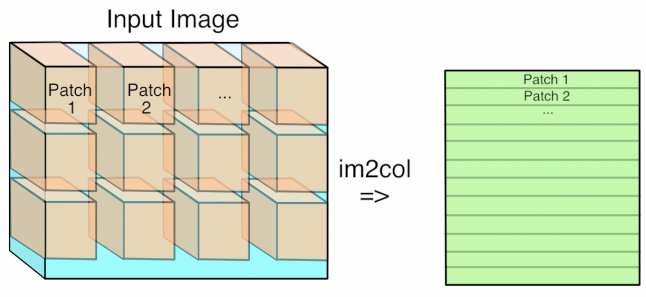

Figure 3: `im2col` operation: the input image is rearranged in memory in order to ease subsequent matrix-matrix multiplication.

a) Implement the `im2col` operation by modifying `Im2MatLayer::Im2Mat` in `Layer_Im2Mat.h` of the skeleton code. The input of `im2col` is B images of size $InY \times InX \times InC$, and the output is a matrix of $B \times OpY \times OpX$ rows and $KnY \times KnX \times InC$ columns. Here, B is the size of the mini-batch. Note that the number of columns of the output is exactly the size of a kernel, easing the `conv2d`.

b) Implement the backward operation of `im2col` by modifying `Im2MatLayer::Mat2Im` in `Layer_Im2Mat.h` of the skeleton code. This operation is often called `col2im` because it computes the gradient of the loss w.r.t. to the input of `im2col` layer. Therefore, it receives the gradient of the loss w.r.t. to the output of `im2col`, a deliberately shaped matrix, and transforms it into an image, with the size of the input of `im2col`.

c) Implement both the forward and the backward operations of the `conv2d` layer by modifying `Conv2DLayer::forward` and `Conv2DLayer::bckward` in `Layer_Conv2D.h` of the skeleton code. This layer should perform only GEMM and adding the bias. Therefore, these two functions should look very similar to what you implemented for Ex.06.

Again, you can test that your code is working correctly with `./exec_testGrad conv`, which checks that the gradient of a parameter computed through finite differences is equal to the same gradient computed by back-propagation.

Question 2: Parallelization

Like for the previous exercise, parallelize with OpenMP threads and motivate your implementation. For this exercise, only the parallelization of `conv2d`, `im2col`, and `main_classify.cpp` will be graded.

Question 3: Tweaking hyper-parameters

There is a commonly held belief that deep learning works by student gradient descent. This semi-serious statement alludes to the fact that deep parametric models are hard to train and

may require minor tweaks to work. For example, we are considering a CNN. Among many other strategies, we could increase the number of layers, change the shape and number of kernels, change the non-linearity, learning rate, batch-size, or linear layer after the convolutional layers.

Try to increase the classification accuracy of the CNN model. You can use any tool developed for this or the previous exercise. Failure to achieve better results than the baseline is an acceptable result, if it is motivated. If you are particularly proud, we may share your solution during the feedback for this exercise.

1 Notes

The MNIST dataset can be downloaded with the script `setup_mnist.sh`. These files should be in the same folder as the executables. Test your work on `euler.ethz.ch` and by loading the environment:

```
$ module load new gcc/6.3.0 openblas/0.2.13_seq
```

and requesting an interactive node:

```
$ bsub -Is -n 1 -W 04:00 bash
```

Request additional processors to test your parallelization with `openblas/0.2.13_par` and:

```
$ bsub -Is -R fullnode -n 24 -W 04:00 bash
```

Once you have acquired a node, test the scaling by varying both `OMP_NUM_THREADS` and `OPENBLAS_NUM_THREADS`.