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.

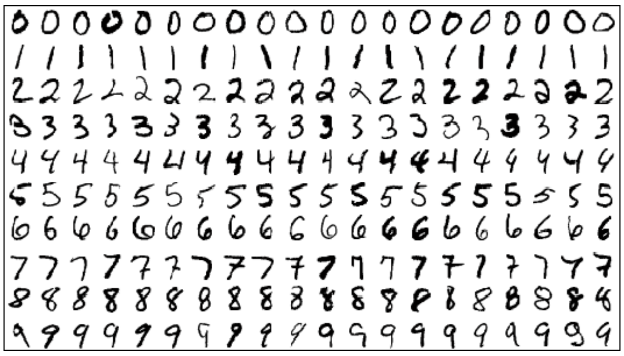

Figure 1: Examples from the MNIST dataset

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

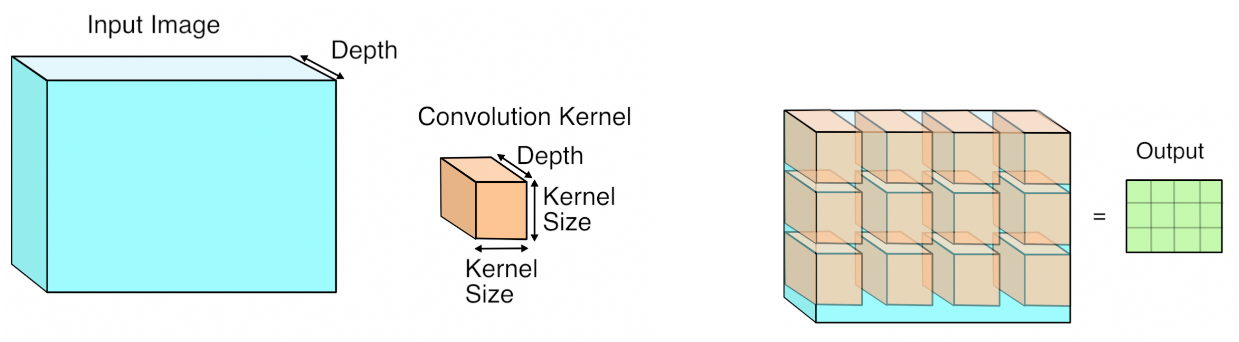

Figure 2: Visualization of the `conv2d` operation. The convolution kernel contains the parameters of `conv2d`. The kernel K is moved along the width and height of the input (I). Each position assumed by K corresponds to one pixel of the output O, which is computed through scalar product between K and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

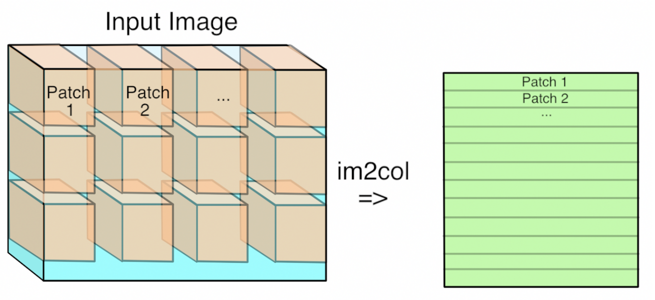

Figure 3: `im2col` operation: the input image is rearranged in memory in order to ease subsequent matrix-matrix multiplication.

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`.

55 points total

Each row of the `im2col` matrix has the size of a convolution kernel. Within the row, we keep the spatial/color structure of the i-th kernel `K_i[KnY][KnX][InC]`. Therefore, a compact way to define the `im2col` is by treating its output as a 5-dimensional (6 with mini-batches) nd-array of size $B \times OpY \times OpX \times KnY \times KnX \times InC$. The remaining challenge of this question is how to map between the starting index of a patch and the corresponding row of of the output matrix. This issue is best understood graphically and we refer to the tutorial slides.

```
using InputImages     = Real[][InY][InX][InC];
using OutputMatrices = Real[][OpY][OpX][KnY][KnX][InC];

const InputImages & __restrict__ INP = * (InputImages*) lin_inp;
OutputMatrices & __restrict__ OUT = * (OutputMatrices*) lin_out;
memset(lin_out, 0, BS * OpY * OpX * KnY * KnX * InC * sizeof(Real) );

// Solution starts here. Confront to slide 13 of the tutorial.
#pragma omp parallel for collapse(3) schedule(static)
for (int bc = 0; bc < BS;  bc++)
    for (int oy = 0; oy < OpY; oy++)
    for (int ox = 0; ox < OpX; ox++) {
        //starting position along input map for convolution with kernel
        const int ix0 = ox * Sx - Px, iy0 = oy * Sy - Py;
        for (int fy = 0; fy < KnY; fy++)
        for (int fx = 0; fx < KnX; fx++) {
            //index along input map of the convolution op:
            const int ix = ix0 + fx, iy = iy0 + fy;
            //padding: skip addition if outside input boundaries
            if (ix < 0 || ix >= InX || iy < 0 || iy >= InY) continue;
            for (int ic = 0; ic < InC; ic++) //loop over inp feature maps
                OUT[bc][oy][ox][fy][fx][ic] = INP[bc][iy][ix][ic];
        }
    }
```

Here we added the `__restrict__` qualifier which tells the compiler that `OUT` and `INP` do not overlap, allowing some compile-time optimizations. In this case, the compiler may copy all elements of the `InC` loop with a single instruction rather than copying one element at the time. Neither inclusion of this qualifier nor the `pragma` instruction are graded here.
An analogous (and correct) loop can be defined by looping over iy and ix and computing oy and ox from fx and fy. The correct mapping function can be found in the next solution snippet.

- **15 points if correct.**
- **10 points if only padding is handled incorrectly.**
- **5 points if only the mappings from `ix` to `ox` or from `iy` to `oy` (or vice-versa) contain errors.**
- **0 points otherwise.**

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`.

The operation here is best understood by observing that one pixel of the input image is copied to multiple locations of the output matrix. At the same time, an element of the output matrix carries the value of a single pixel of the input image. Therefore, the error carried by an element of the output matrix is caused by a single input pixel. For the back-propagation, we receive an error signal for each element of the output matrix. The gradient for a single pixel in the input image is obtained by summing together the error signals from each spot in the output matrix where that pixel was copied to.

```
using InputImages    = Real[][InY][InX][InC];
using OutputMatrices = Real[][OpY][OpX][KnY][KnX][InC];

InputImages __restrict__ & dLdINP = * (InputImages*) lin_out;
const OutputMatrices __restrict__ & dLdOUT = * (OutputMatrices*) lin_inp;
memset(lin_out, 0, BS * OpY * OpX * KnY * KnX * InC * sizeof(Real) );

// Solution starts here. Confront to slide 15 of the tutorial.
#pragma omp parallel for collapse(3) schedule(static)
for (int bc = 0; bc < BS;  bc++)
    for (int iy = 0; iy < InY; iy++)
    for (int ix = 0; ix < InX; ix++) {
        for (int fy = 0; fy < KnY; fy++)
        for (int fx = 0; fx < KnX; fx++) {
            const int oy = ( iy + Py - fy ) / Sy;
            const int ox = ( ix + Px - fx ) / Sx;
            //padding: skip addition if outside input boundaries
            if (oy < 0 || oy >= OpX || ox < 0 || ox >= OpY) continue;
            for (int ic = 0; ic < InC; ic++) //loop over inp feature maps
                dLdINP[bc][iy][ix][ic] += dLdOUT[bc][oy][ox][fy][fx][ic];
        }
    }
```

Also in this case, an analogous loop can be defined by looping over oy and ox. The `pragma` is not graded here.

- **15 points if correct.**
- **10 points if only padding is handled incorrectly.**

- 5 points if only the mappings from `ix` to `ox` or from `iy` to `oy` (or vice-versa) contain errors.
- 0 points otherwise.

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.

Each kernel applies its own bias: **5 points.**

```
const Real* const __restrict__ B = param[ID]->biases; // size is KnC
#pragma omp parallel for schedule(static)
for (int i=0; i<batchSize * OpY * OpX * KnC; i++) OUT[i] = B[i % KnC];
```

The forward operation is performed by the following GEMM call: **5 points.**

```
const int mm_outRow = batchSize * OpY * OpX;
const int mm_nInner = KnY * KnX * InC;
const int mm_outCol = KnC;
gemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, mm_outRow, mm_outCol, mm_nInner,
          (Real) 1.0, act[ID-1]->output,          mm_nInner,
                      param[ID]->weights,          mm_outCol,
          (Real) 1.0, act[ID]->output,             mm_outCol);
```

Like in the linear layer, the gradient of the bias is a sum over the gradients of all outputs to which the bias was added: **5 points.**

```
std::fill(B, B+KnC, 0);
#pragma omp parallel for schedule(static) reduction(+ : B[:KnC])
for (int i=0; i<batchSize * OpY * OpX * KnC; i++) B[i % KnC] += dEdO[i];
```

The following operations are analogous to the behavior of a linear layer: **10 points.**

```
const int mm_outRow = KnY * KnX * InC;
const int mm_nInner = batchSize * OpY * OpX;
const int mm_outCol = KnC;
gemm(CblasRowMajor, CblasTrans, CblasNoTrans, mm_outRow, mm_outCol, mm_nInner,
          (Real) 1.0, act[ID-1]->output,        mm_outRow,
                      act[ID]->dError_dOutput,   mm_outCol,
          (Real) 0.0, grad[ID]->weights,         mm_outCol);
```

```
const int mm_outRow = batchSize * OpY * OpX;
const int mm_nInner = KnC;
const int mm_outCol = KnY * KnX * InC;
gemm(CblasRowMajor, CblasNoTrans, CblasTrans, mm_outRow, mm_outCol, mm_nInner,
          (Real) 1.0, act[ID]->output,            mm_nInner,
                      param[ID]->weights,          mm_nInner,
          (Real) 0.0, act[ID-1]->dError_dOutput,  mm_outCol);
```

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

10 points total

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.

- `Conv2DLayer::forward` Refer to the solution of question (c) and to the previous exercise. **1 point.**
- `Conv2DLayer::bckward` Refer to the solution of question (c) and to the previous exercise. Any approach that does not lead to race-conditions in the bias' gradient is accepted. **2 points.**
- `Im2Mat` We accept most parallelization strategies here. Eg. simply a parallel for over the batch-size would be correct. **1 point.**
- `Mat2Im` Here we have a possible race condition if the parallel loops span OpY and OpX instead of InY and InX. Other approaches are accepted. In other words, for this point, the following would be **wrong**:

  ```
      #pragma omp parallel for
      for (int oy = 0; oy < OpY; oy++)
          for (int ox = 0; ox < OpX; ox++) // even with the loops swapped!
  ```

  2 points.
- There are for loops that can be parallelized with threads in `main_classify.cpp`, refer to the solution code. The submission will be considered wrong if the `reduction` clause is missing wherever needed. **4 points.**

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.

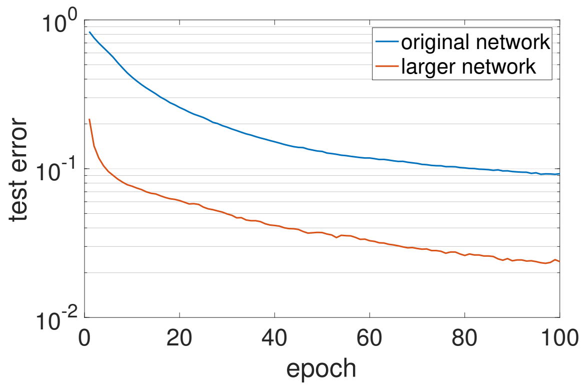

Figure 4: Test error (fraction of wrong predictions) for two net sizes in the solution code: original net, and larger net with more kernels per convolution and an added linear layer.

Many hyper-parameters can be changed to improve the accuracy of the model. The easiest way to achieve improvements is just to make the model bigger. The model included in the solution code, which was not extensively "tweaked" can easily beat all non-convolutional models described by [1]. **15 points total**

Maximum number of points for this exercise: 80.
Number of points required for grade 6.0: 70

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`.
To use the MKL library, load:

```
$ module purge && module load new gcc/6.3.0 mkl/2018.1
```

Compile the code with `make blas=mkl` and set `MKL_NUM_THREADS`.

References

[1] LeCun, Yann and Bottou, Léon and Bengio, Yoshua and Haffner, Patrick, Gradient-based learning applied to document recognition, Proceedings of the IEEE, 1998.