CS 426/525
Fall 2023
Project 3
Due: 24/12/2023

In the first project, you implemented a parallel version of what a fully connected layer does in an artificial neural network using MPI. In this project, we ask you to implement some layers of Convolutional Neural Network (CNN) first as a serial code and then implement a parallel version of it using OpenMP. Before explaining the assignment, below is a brief introduction to CNNs.

Introduction

Neural networks, inspired by the human brain, have undergone significant evolution since their inception. The fully connected neural network, often referred to as a dense or multi-layer perceptron (MLP), served as the foundation for many early machine learning models. However, as the field progressed, researchers realized the limitations of fully connected architectures and embarked on a journey to invent diverse neural network structures.

Fully connected networks exhibit certain drawbacks that hinder their performance in handling complex tasks and large datasets. One of the main challenges is the sheer number of parameters in densely connected layers, leading to increased computational demands and potential overfitting. As the complexity of tasks grew, fully connected networks struggled to capture hierarchical and spatial dependencies within data, limiting their ability to generalize effectively.

To address the limitations of fully connected networks, CNNs were introduced. CNNs are particularly well-suited for image-related tasks, as they leverage convolutional layers to detect spatial patterns and hierarchical features. By sharing weights across the network, CNNs reduce the number of parameters, making them computationally efficient and better at handling image data. This innovation marked a significant step forward in improving the performance of neural networks, especially in computer vision applications.

Convolutional Neural Networks (CNNs)

Convolutional neural networks can be considered as multiple instances of three kinds of layers, namely, the convolution layer, the pooling layer, and the fully connected layer. Figure 1 shows an example CNN implementation used in an image classification problem.

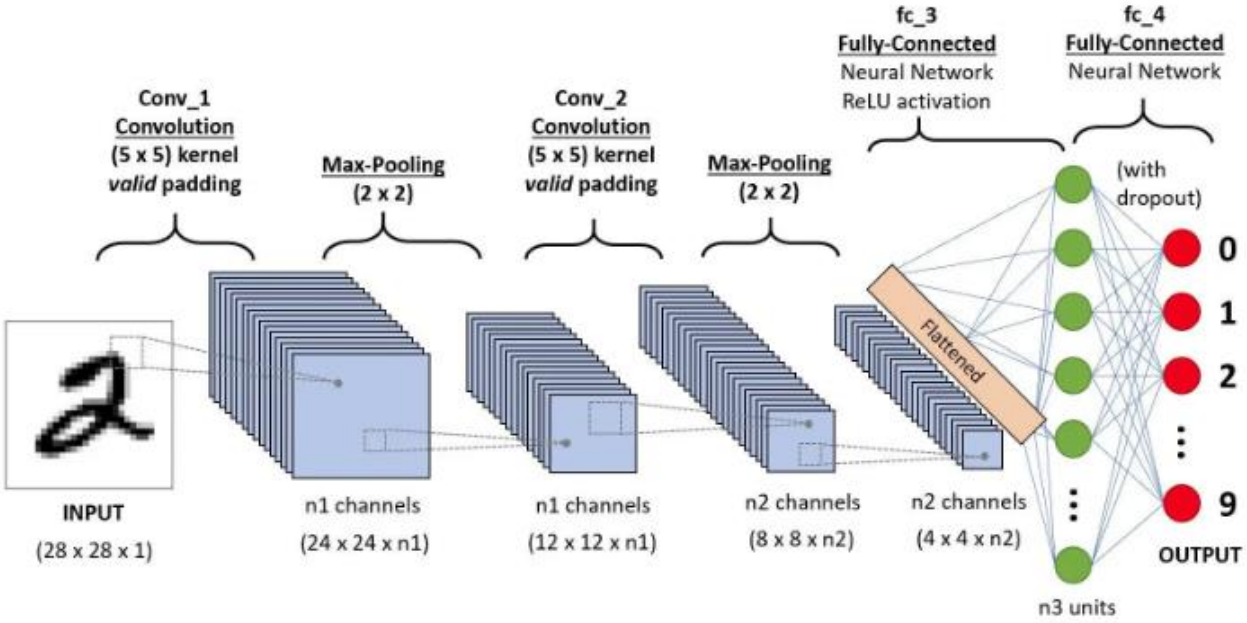

Figure 1 – Example CNN implementation with multiple layers.

Convolution Operation

As can be seen from Figure 2, convolution operations are applied using a filter (kernel) that is slid around an input image. We multiply the corresponding elements of the filter with the pixels of the image, add them up to get a final value, and write it to the output image as we slide the filter throughout the image.

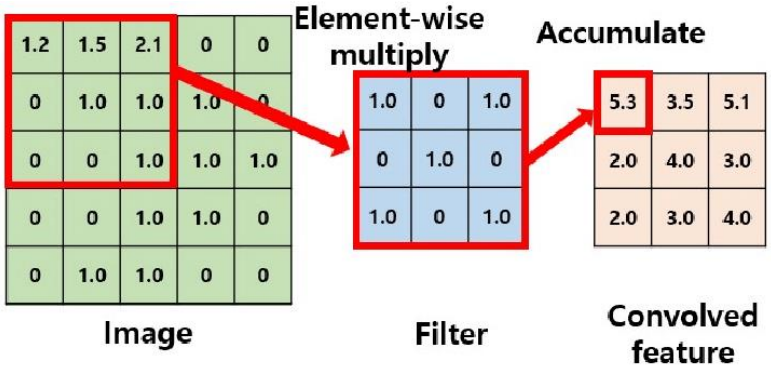

Figure 2 – Example convolution operation applied to a 5 x 5 input image.

Padding Operation

When an input image is convolved with a filter, the dimension of the input image reduces. If we want to preserve the dimensions, we can use padding. Figure 3 illustrates zero-padding where the padding size is 1.

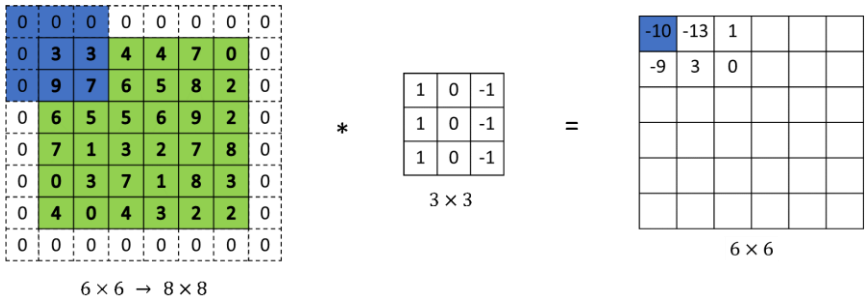

Figure 3 – Example padding applied to an original image.

Max Pooling

Figure 4 shows a max pooling example with kernel size 2 and stride 2. Stride refers to the step size used when sliding the filter. The stride determines how much the filter moves horizontally and vertically after each operation. Max pooling with kernel size 2 simply looks at 2 x 2 area and picks the maximum element and writes it to the output.

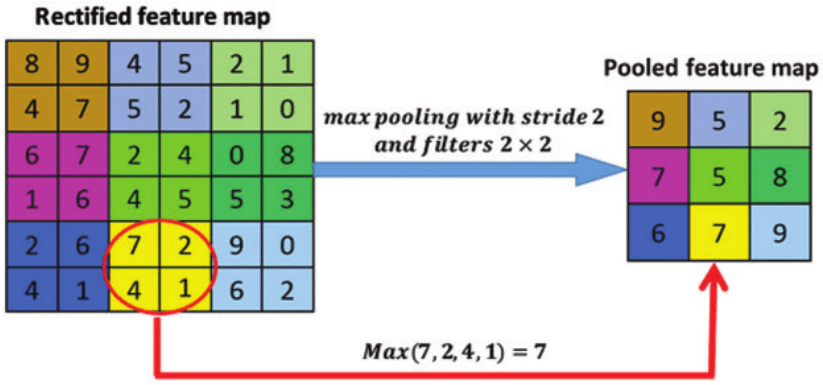

Figure 4 – Max pooling with kernel size 2 and stride 2.

Architecture

When the **conv_1** operation in Figure 1 is considered, it can be observed that a 5 x 5 kernel is used with no padding. With this, the dimension of the input reduces from 28x28 to 24x24. Here n1 refers to the number of filters or kernels applied. In conv_1, the architecture simply applies n1 number of different filters each having 5x5 dimensions to the input image separately and each filter returns a 24x24 image. To get the output, we stack the output of each individual filter and obtain a 24x24xn1 dimensional image.

Project Details

In this project, you will implement a layer of a CNN network. Be careful, the elements of an image and kernel are double-float values (not integer). You will have four files named input.txt, kernel1.txt, kernel2.txt, kernel3.txt. Assume the kernel and input are square matrices. Also assume input has even dimension whereas kernel has odd dimension. kernel1, kernel2 and kernel3 have the same dimensions. Example file contents are given below:

```
$ cat input.txt
4
0.6 2.3 -3.6 7.1
1.2 -1.54 3.8 2.1
-0.21 7.89 5.75 -1.5
-0.31 3.49 5.21 -0.5


$ cat kernel1.txt
3
5.6 -2.4 3.5
2.1 -0.6 2.89
3.7 -3.2 4.7
```

Below are the steps you should follow:

1 – Read input.txt, kernel1.txt, kernel2.txt, kernel3.txt and store them in arrays.

2 – Make proper **zero padding** to the input so that when the convolution operation is applied, the output should have the same dimensions as input (look at the kernel size when determining how many zeros to pad in each dimension).

3 – Apply convolution operations using kernel1, kernel2, kernel3 with stride = 1 and stack their result. If the original input has dimensions NxN, output should have NxNx3 where 3 corresponds to the number of filters.

4 – Apply sigmoid activation function elementwise.

5 – Apply max-pooling with kernel size 2 and stride 2. In input has NxNx3 dimensions in this step, you should get (N/2)x(N/2)x3 dimensional output. When applying max pooling consider each channel separately (here we have 3 channels).

6 – Write the result into output.txt.

This constitutes the serial part of the project. After that, you will profile your code using **gprof** and try to find the functions that are needed to be parallelized. For **gprof** details, there are many online tutorials.

- Second, you will implement parallel version of the same problem.
    - You will insert OpenMP pragmas to parallelize your code.
- You will profile your code again using gprof.

Grading

- Sequential: 20 points
- Parallel: 40 points
- Gprof profiling results: 10 points
- Report: 30 points

Submission

Put everything under the same directory, do not structure your project under directories like parallel, sequential etc. Put all of them in the same directory, one called **yourname_lastname_p3**. You will zip this directory and submit. When it is unzipped, it should provide the directory and files inside.

1. Your code:

- serial.c, parallel.c, make file and any other file that you have implemented.

2. Profiling results, where you will submit gprof profiling results.

- prof_sequential.txt file for sequential implementation's profiling results
- prof_parallel.txt file for parallel implementation's profiling results
  - Include profiling results for several runs with different number of threads, with inputs having different dimensions and with kernels having different dimensions.

3. Your report:

- **Reports should be in .pdf format.**
- Detailed description of gprof's profiling outputs.
- Detailed description of your implementation details
  - Explain pragmas that you have used
    - Why did you insert this pragma to this specific region?
    - What does this pragma do?
    - What are the possible options that can be used with this pragma?
    - What are the options that you have used?
- Plot the execution times with different threads.
- Discussion of your results
  - Don't forget to use gprof's profiling outputs in your discussion.

- **Zip File name to upload:** yourname_lastname_p3.zip
- **No Late Submission Allowed!**

Disclaimer: Figures are from several resources on the web, below are the references.

References

- https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/
- https://datahacker.rs/what-is-padding-cnn/
- https://www.researchgate.net/figure/Max-pooling-processing-with-filters-2-2-and-stride-2_fig5_348909522
- https://www.researchgate.net/figure/Convolution-operation_fig2_355656417