Chair for Computer Architecture and Parallel Systems
School of Computation, Information and Technology
Technical University of Munich

Lab 1: Implementing CNN Layers

Practical Course: Accelerating CNNs using PL

26.04.2023

1 Introduction

This exercise sheet and the accompanying code is provided as an assistance to implement CNN inference in Software. You can use none, some or all of the code provided, as long as your finished Software supports the layers described in the exercises and the optimizations schemes (It is however recommended to use the code). Implementing a larger program at once can be challenging in the beginning and debugging it later is a time intensive task. To try to avoid the problem of having to find small errors later on, a set of testing utilities are provided to test the correctness of the implementations. Furthermore, we will start implementing sub sets of the final implementation. You will start by implementing the separate layers and testing them using input and output values, generated using the pytorch library. The ordering of input values will be according to the pytorch library, specifically the 2d convolution where in some languages (e.g. MATLAB) the weight matrix is flipped in comparison to pytorch.

2 Exercise

Complete the 5 functions in the `kernels.cpp` file. To test the correctness you can use the test cases generated by the `gentest.py` and the `lab1.cpp` file. The Makefile contains the recipes to build the `lab1.bin` target. Below is a recap of the different layers that have to be implemented. **If all Tensors are equal, when running `lab1.bin` you are done!**

3 Python

To run the `gentest.py` you need python 3.x as well as the packages `torch` and `numpy`. All two can be installed using pip that usually comes with your python installation. (Both Windows and linux)

```
> pip install numpy torch
```

To run the `gentest.py` file run:

```
> python gentest.py
```

4 gcc & GNU Make

The exercise folder comes with a makefile, which can be run using the GNU make program. GNU make comes pre-installed with most linux distros but if not can be installed using the package manager of choice for example using apt (Ubuntu,Debian):

```
> sudo apt install gcc
> sudo apt install make
```

The code has been tested using the gcc compiler, but other compilers probably will work as well. To test whether both GNU make and gcc are installed you can run:

```
> make -v
> gcc -v
```

Which should output the version of the respective program if it installed.

5 Getting Started

Download the code and extract the folder (`tar -xzf filename`). The folder contains a `src/`,`build/` and `utils/` directory. The `src/` directory contains the `kernels.cpp` file where you should implement the different layers. The `lab1.src` reads the test files from the `data` folder, these files are generated by running the `gentest.py` file.
To test whether everything is installed and working you can run the Makefile inside the folder using a terminal:

```
> python gentest.py
> make
```

Then run the test file:

```
> ./lab.bin
```

The Tensors should be unequal to the Reference values as none of the functions are implemented yet. If the data files have not been created `lab.bin` will throw a segmentation fault.

6 Installation on Windows

To get the lab running on windows follow the instruction in the video.

1. Install Python from (https://www.python.org/downloads/) adding python.exe to your path
2. Install a text editor for example VS code (https://code.visualstudio.com/)
3. Install msys2 from (https://www.msys2.org/)
4. Install the following packages using pacman inside the ucrt64 environment:

```
> pacman -S mingw-w64-ucrt-x86_64-gcc
> pacman -S mingw-w64-ucrt-x86_64-make
```

5. Add the ucrt64 bin folder to your windows path
6. Edit the TRG in the Makefile to `lab1.exe` from `lab1.bin`
7. Install numpy and torch using pip as explained in the python section

You can test whether everything is working by opening up a powershell navigating to the folder and running:

```
>python gentest.py
>mingw32-make.exe
>./lab1.exe
```

This should output all Tensors to be unequal.

7 Layers

7.1 Linear Layer (Fully Connected Layer)

The Liner layers connects all inputs to all outputs in the form of Perceptron:

$$\vec{z}(\vec{x}) = \mathbf{W}\vec{x} + b \tag{1}$$

Usually a Fully Connected Layer is followed by a nonlinear activation function, this can for example be a ReLU layer or a Softmax Layer if it is the final layer. If the previous layer is a multidimensional Tensor, the values have to be sequentially in one dimension. This is rather simple as iterating over the data array of the Tensor class gives us a flattened representation of the input Tensor.

7.2 Activation Function

The two activation functions used are `ReLU()` and `Softmax()` both are work element-wise. The Softmax function return a probability between 0 and 1 for each classification element, and is used as the activation function of the last fully connected layer. Both function should be very straightforward to implement:

$$z_i(x_i) = max(0, x_i) \tag{2}$$

$$z_i(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{N-1} e_j^x} \tag{3}$$

7.3 Max Pool

Pooling performs a reduction in the input, by performing an operation on a field of the input for example averaging a 2 × 2 area of the input feature map or performing a max operation only passing along the maximum of the input values. Pooling Layers are usually not followed by an activation layer (Usually preceded by one). The Max Pool function in our example performs max pool with a size and stride of 2.

7.4 Convolutional Layer

The convolutional performs the convolution operation on an multidimensional input (Tensor). The key parameters of a convolutional layer are:

Input size The dimension of the Input tensor in 3D: Input Channels X_C, Feature Map height X_M and Feature Map width X_N.

Kernel (Weight) size The dimension of the Weight (Kernel) Tensor in 3D: Weight Channels W_C, Kernel height W_M and Kernel width W_N. The kernels in this practical course are all square thus $W_M = W_N$ and we use a stride of 1 for every convolutional layers. The number of channels per weight tensor is equal to the number of input channels $W_C = X_C$. For every output channel Z_C another Weight Tensor is required!

Bias Every Weight Tensor has one bias value attached that is added to every element of the output feature map. This means that the bias is a 1D-Tensor with one element for every output channel $B_N = Z_C$

Output size The output tensor only has one flexible parameter and that is the number of output channels Z_C, the width and height of the output depend on the input and kernel width and height $Z_M = X_M - W_M + 1$ and $Z_N = X_N - W_N + 1$.

Calculation of a single output

$$z_{i,j,k} = \sum_{c}^{X_C} \sum_{p}^{W_M} \sum_{q}^{W_N} x_{c,j+p,k+q} w_{i,c,p,q} \tag{4}$$

i indicates the output channel and j, k the y-and x-position of the output. Following a convolution layer is usually a non-linear activation function. A Convolutional layer can be padded to maintain the input dimensions in the output, this however will be implemented outside the convolution function leaving it to be always not padded.

8 Testing

The `gentest.py` is an example to generate test cases to test the different layers. To test your implementation you can use the `lab1.cpp` file it loads a file for each layer apart from the ReLU one and performs tests. It reads all the test cases until no test cases are available anymore. The `compareTensor()` function is used to compare the tensors, N tensors are compared and the threshold can be used to avoid errors due to floating point rounding errors. The file `test_utils.py` includes some functions to generate test cases and append them to a file, the functions call the respective pytorch function and store inputs, outputs and weights in a file. You can modify the test cases by modifying the `gentest.py` file, it does for example make sense to start with simple examples when debugging.

Name:	Description
T.size[0]	Number of channels
T.size[1]	Tensor height
T.size[2]	Tensor width
T[i]	Returns 2d feature map
T[i][j]	Returns row of tensor
T[i][j][k]	Returns element of tensor
Tensor()	Creates empty Tensor with no memory allocation and size 0
Tensor(i,j,k)	Allocates Tensor of size: $i \times j \times k$
read(f)	Reads Tensor from file f (reallocates Tensor if required) returns status.
write(f)	Writes Tensor to file f
resize(i,j,k)	Resizes Tensor and reallocates data array
randomize(start,stop)	Fills Tensor with random values from start to stop

Table 1 Tensor function Reference

9 Tensor Class

The Tensor class is a 3d data structure that is used to store the Tensors between layers. A Tensor of size $Z \times Y \times X$ can be created using the constructor `Tensor(Z,Y,X)`. An element of a tensor can be accessed using brackets `e = T[z][y][x]` and all values a stored in one contiguous array so one could theoretically access the same element using `e = T[0][0][z*Y*X + y*X + x]`. To read a tensor from a file one can use the `read(FILE *f)` function, if the stored tensor is of different size than the Tensor calling the method the storage will be reallocated. To write a Tensor to a file the method `write(FILE * f)` can be used. Another useful function are the `randomize(FLOAT start, FLOAT stop)` that fills the Tensor with random values between `start` and `stop`.