ImageNet Classification with Deep Convolutional Neural Network

Advanced Algorithm and Parallel Programming

Lorenzo Federico Porro 859093
Matteo Pozzi 8410091

Table of contents

Introduction

Krizhevsky’ s and Sutskever’ s deep convolutional neural network is the state of the art for image recognition and classification problems.

The goal of the project was to implement a fully functional network and train it on a reduced size training set and classes: each input is categorized as belonging or not to the specific synset.

Tools

- C++11: to implement the network architecture and the training algorithms.
- Python: to download and preprocess image data (cropping and reading of the RGB pixel’ s values).
- OpenMP: to parallelize the execution.
- Git: version control and code sharing.

Pre-processing

Images are downloaded from ImageNet database along with bounding boxes (data on where the image subject really is), the images are already divided by their main category (synset).

Due to the variance in image size the images are cropped around their bounding box and scaled to the standard size of 224x224 pixels to feed to the network. Images too small are padded with RGB values of 0.

Pixel's values for each image are then read and saved into a .txt file.

- We used Python' s crop() and load() functions from PIL library.

Pre-processing - 2

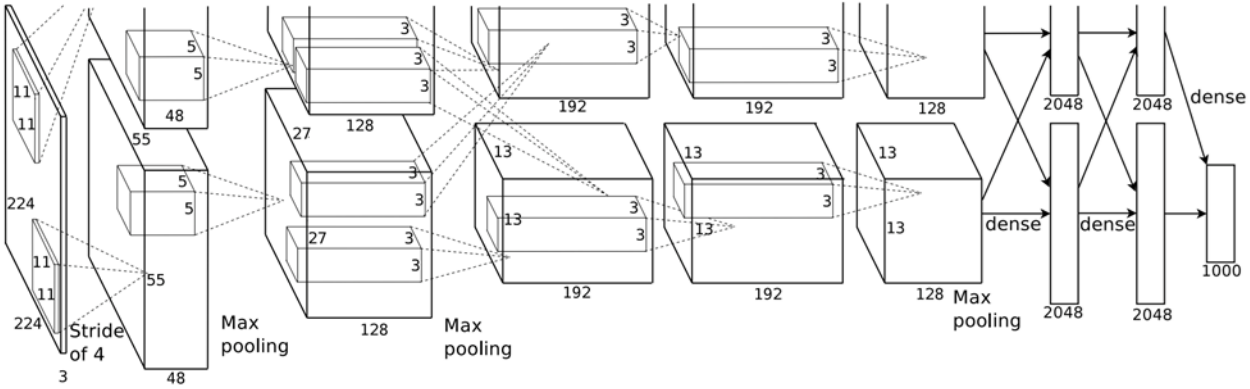
.txt files are read by input_reader.cpp class that extract for each image a 3-dimensional array of integer values (224x224x3).

We recovered around 400 images (the biggest category with bounding boxes available) of the synset ‘birds’.

Network architecture

The network is composed by 8 layers: the first 5 layers are convolutional and the rest is fully connected. Layers 1, 2 and 5 also apply a max pooling function.

The network is split on two separate threads: the goal of this division is not only to improve the speed of calculus but also to specialize the network to recognize different features due to the limited connectivity.



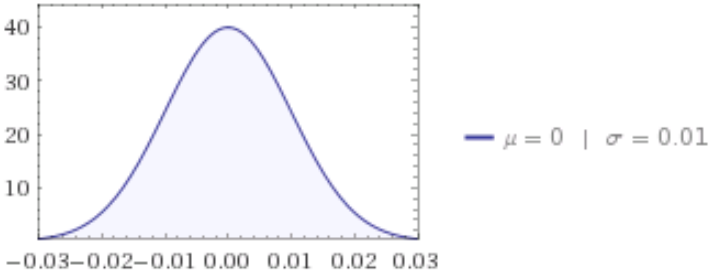
Architecture - 2

- Max-pooling layers are partially overlapped to reduce error rate and have dimension 3x3 with a stride of 2.
- The Non-saturating ReLU function $f(x) = \max(0,\ x)$ is used since is much faster in training with respect to saturating functions (like $\tanh(x)$).

Architecture - 3

Kernels for voncolutional layer are initialized with a normal distribution with 0 mean and a standard deviation of 0.01.

The small values helps in reducing overfitting for training and the randomization avoids any unwanted dependency amongst weights.



Architecture - 4

Convolution (1) and ReLu (2) computation code:

```
#pragma omp for collapse(3)
for(int f=0;f<feat;f++){    // for each feature

    for(int i=0;i<depth;i++){   // for each depth level

        // 2d-convolution
        for(int oy=0;oy<osize;oy++){

            iy=oy*stride;
            for(int ox=0;ox<osize;ox++){

                ix=ox*stride;
                for(int ky=0;ky<ksize;ky++){

                    for(int kx=0;kx<ksize;kx++){

                        // check if the kernel goes outside the input matrix
                        if(ky*oy<isize && kx*ox<isize && iy<isize && ix<isize){
                            output[f][oy][ox] += input[i][iy][ix] * kernel[f][i][ky][kx];
                        }
                        ix++;
                    }
                    ix=ox*stride;
                    iy++;
                }
                iy=oy*stride;
                output[f][oy][ox] += bias;
            }
        }
    }
}
```

```
#pragma omp for collapse(3)
for(int f=0;f<feat;f++){

    for(int i=0;i<size;i++){

        for(int j=0;j<size;j++){

            relu[f][i][j]=max((double)0, relu[f][i][j]);
        }
    }
}
```

Architecture - 5

Convolutional layer output computation code:

```
int stride2 = round((double)(layer1[0].size() - kernel21[0][0].size())/(conv2[0].size()-1));

//convolution between layer 1 and kernels of layer 2
convolution(layer1,kernel21,conv2,stride2,1);

//ReLU function
relu(conv2);

//overlapped max-pooling
maxpooling(conv2,layer2);


//saving output of the first CNN layer 2
out2[0]=layer2;
```

Architecture - 6

ReLu non linearity computation code:

```
#pragma omp for
for(int i=0;i<2048;i++){
    for(int j=0;j<2048;j++){

        layer7[i] = weight711[i][j]*layer61[j];
    }
    layer7[i]+=1;
    layer7[i] = max((double)0, layer7[i]);   //ReLU function
}
```

Architecture - parallelization

The network is split onto 2 separate threads.

Every thread is separate by a section and each contains the private layers execution. Sections terminates when synchronization between layers is needed (i.e. in maxpooling convolutional layers when the output of both sections must be feed to the next layer). Since Sections includes an implicit barrier no additional concurrency control is needed.

Architecture - parallelization - 2

Example of pseudocode for the first two layers:

```
#pragma omp sections
        #pragma omp section
                Layers 1 and 2 computation of the first CNN
        #pragma omp section
                Layers 1 and 2 computation of the second CNN
End sections (implicit barrier)
…successive layers…
```

Training

Training of the neural network is implementd with the standard backpropagation algorithm and batch gradient descent to optimize the weigths.

The batch size for training is 128 images.

Training - 2

Training of the neural network is implementd with the standard backpropagation algorithm and batch gradient descent to optimize the weigths and minimize the cost function.

Cost function $\frac{1}{2}\left\|y - h_{W,b}\right\|^2$

Weight update rule

$$v_{i+1} = 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \sigma_{i,j}$$

$$w_{i+1} = w_i + v_{i+1}$$

With $\sigma_{i,j}$ being the average over the j-th batch of the derivative of the objective with respect to w, evaluated at w_i.

Training - 3

With backpropagation σ is calculated for each weight as:

$$\sigma^{(l)} = \delta^{(l+1)}(a^{(l)})^T$$

$\delta^{(l)}$ is the error backpropagated from layer l:

$$\sigma^{(n_l)} = -(y - a^{(n_l)}) * f'(z^{(n_l)}) \qquad \text{for the output layer}$$

$$\delta^{(l)} = (\left(W^{(l)}\right)^T \delta^{(l+1)}) * f'(z^{(l)}) \qquad \text{for all other layers}$$

With $a^{(l)}$ activation function of layer l, $z^{(l)} = a^{(l)}\left(1 - a^{(l)}\right)$, $W^{(l)}$ the weight matrix of layer l, y the desired output and $*$ being the product-wise multiplication (Hadamard product).

Training - 4

Backpropagation errors calculation (1) and parameters update (2) code:

```
#pragma omp for collapse(2)
for (int i = 0; i < backwardValues.size(); i++){
    for (int j = 0; j < backwardValues[0].size(); j++){
        count++;
        deltaErrors[i][j] = params[j][i] * backwardValues[i][j] * max((double)0, forwardValues[i][j]) * (identity[i][j] - max((double)0, forwardValues[i][j]));
    }
}
```

```
#pragma omp for collapse(2)
for (int i = 0; i < backwardValues.size(); i++){
    for (int j = 0; j < backwardValues[0].size(); j++){
        layerWeights[i][j] = layerWeights[i][j] - (learningRate * (((1 / batchSize) * deltaW[i][j]) + weightDecay * layerWeights[i][j]));
        layerBiases[i][j] = layerBiases[i][j] - (learningRate * ((1 / batchSize) * deltaB[i][j]));
    }
}
```

Training - 5

Backvalues for last layer (1) and for a generic layer (2) calculation code:

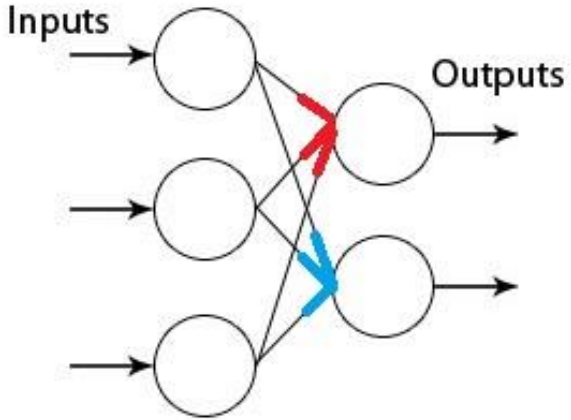
```
#pragma omp for collapse(2)
for (int i = 0; i < layerHeight; i++){
    for (int j = 0; j < layerWidth; j++){
        backValues[i][j] = -(desiredOutput - max((double)0, output)) * max((double)0, output) * (1 - max((double)0, output));
    }
}
```

```
#pragma omp for collapse(2)
for (int i = 0; i < weights.size(); i++){
    for (int j = 0; j < weights[0].size(); j++){
        backValues[i][j] = weights[i][j] * errors[i][j] * max((double)0, layerOutput[i][j]) * (1 - max((double)0, layerOutput[i][j]));
    }
}
```

Training - parallelization

For each layer of weights, $\delta^{(l+1)}$ (blue and red errors) to backpropagate are calculated in parallel and synchronized when the actual update occurs.

$a^{(l)}$ (the activation of layer l) comes from the random initialized network taht is already parallelized.



Training - parallelization - 2

Pseudocode:

For each layer do:

#pragma omp sections

    #pragma omp sections

        compute errors for layer l

    #pragma omp sections

        update weights

Problems

The integration between the actual network and the training algorithm has not been completed due to an issue in the saving of the trained weights: the sheer size of the files to write and read containing the values.

Results

Even though we couldn’ t run the network with proper weights and the training algorithm with a proper underlying network to calulated the precision of the classification, we got some experimental results that suggest that both are correctly implemented and working:

- The network initialized with random weights with 0 mean gives always an output probability of around 50% to belong or not belong to the synset.

Since without training layers cannot learn to recognize specific features this means simply that the network ‘has no clue’ on which class the input should belong and than assign an equal probability to each synset.

Results - 2

- Testing the training algorithm on a single layer with random initialized weights, weights values suggests that some neurons specialize in recognizing features.

Training a single layer with a batch of 128 different causes some weights to stabilize around values never greater than 100 circa and other to be between 0 and 1. Training instead the layer with a batch of 128 identical images some weights ‘explodes’ and reach values in the order of tens of thousands, while others remain in the same small range.

We interpreted this as a ‘proof’ that some neurons strongly learn to recognize certain features and increase their weight with respect to the others since they are learning the same features at each iteration.

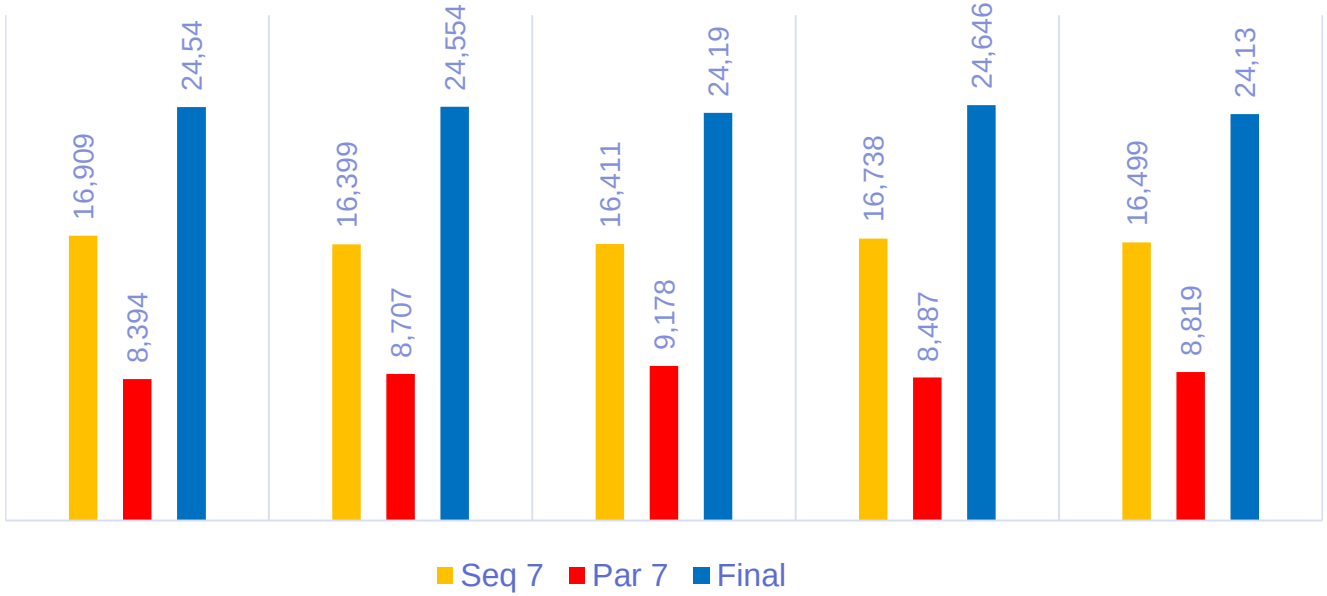
Speedup results

Average execution times for network and training algorithm (Intel 4 cores proc):

Training algorithm (for single layer):

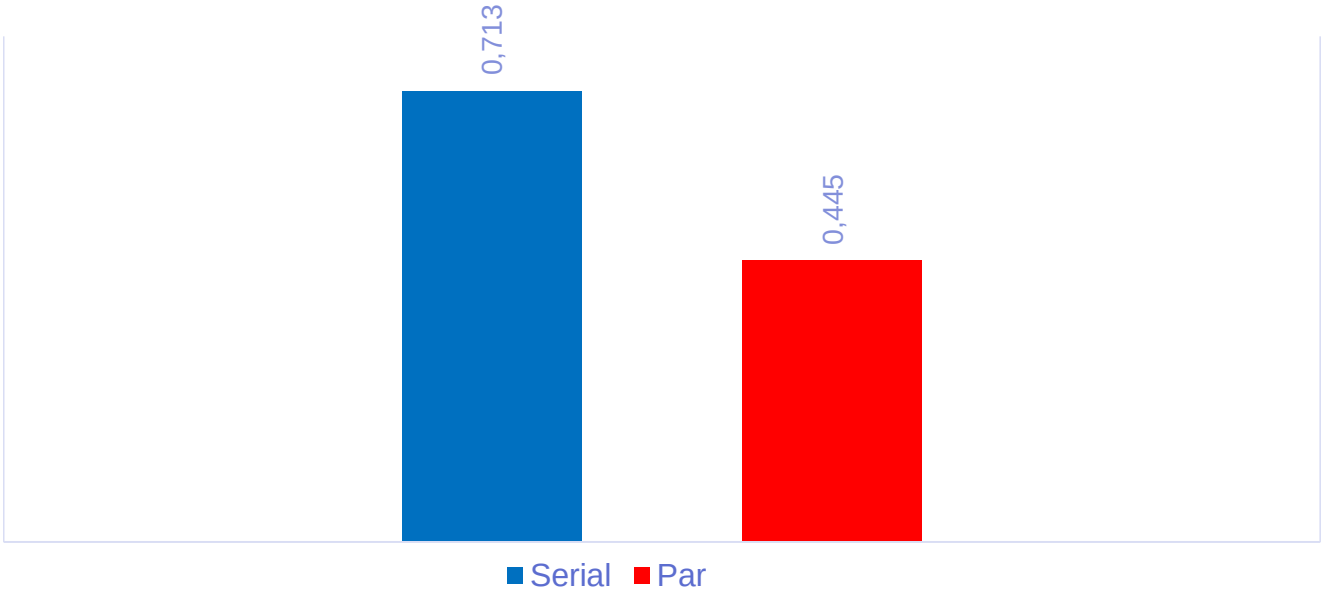
- Serial execution: 0.713 s
- Parallelized execution 0.445 s

Speedup results



- Seq 7 (single CNN until layer 7) : 16.5912 s (mean)
- Par 7 (single CNN with parallel loops until layer 7) : 8.717 s (mean)
- Final (two parallel CNN) : 24.412 s (mean)

Speedup results - 2



- Serial execution: 0.713 s (mean)
- Parallelized execution: 0.445 s (mean)

Speedup results - 3

- Speedup of around 62% for training
- Speedup of around 52% for the network

OpenMP

OpenMP is less powerful than CUDA but is much easier to program and portable on different machine without proprietary Nvidia hardware.

Since we wrote the code from scratch, without using any external library, the simplicity of the language was critical for our project due to the tight timeline and our choice of an ambitious project. Also be able to run the code on any machine made the sharing of the code a whole lot easier.

Future improvements

- Complete the integration between the network and the training algorithm
- Find a smart way to save the weights values in non volatile memory
- Rewrite part of the code to make it more modular and simplify the integration
- Test on different machine with different number of cores to check speedup scaling
- Extend the 2 synset classification solution to multiple synset
- Add top 3-class and top 5-class recognition capabilities to the network