A super-awesome jedi power title

Magnus Halvorsen

November 16, 2014

Abstract

Chapter 1

Introduction

Let me introduce you to a world of wonders. Pie.

Chapter 2

Related Work

In [1] a CNN was implemented on a Virtex-4 SX35 FPGA from Xilinx. In this implementation all the fundamental computations was hard-wired, and controlled by a 32bit soft processor using macro-instructions. Training was done offline, and a representation of the network was provided to the soft processor. With this implementation they were able to process a 512×384 grayscale image in $100ms$, i.e. 10 frames per second.

Chapter 3

Background

In this chapter we will go through the fundamental mathematics and concepts behind the *Convolutional Neural Network* (CNN) model, in order to be able to recognize which operations that can be hardware-accelerated. It gives a basic introduction to both general neural networks and *CNNs*.

3.1 Artificial Neural Networks

An *Artificial Neural Network* (ANN) is a computational model that is used for machine learning and pattern recognition. The name and basic concept is inspired by how the animal brain uses a network of neurons to recognize and classify objects.

An ANN can intuitively be viewed as a probabilistic classifier. Depending on the input data it will output the probability that the data belongs to a certain *class* (e.g. an object in an image or an investment decision). As the brain it can be trained to recognize different classes by being provided a set of labeled training data, e.g. a set of faces and a set of non-faces. It can then learn to decide whether a image contains a face or not. This is called supervised learning. The network can also be trained unsupervised, by providing it with a set of unlabeled images. It will then learn to recognize a set of classes, but will be unable to label them.

The topology of an ANN is a number of layers containing a set of so-called neurons. A neuron takes as in a set of inputs (e.g. image pixels), where each input is associated with a respective weight. The input and the weight are then multiplied and summed, and the result is used to calculate a non-linear activation function. More formally a neuron is defined as follows:

$$Input : \{x_1, x_2, \ldots, x_n\} = \mathbf{x}$$

$$Output : f(\mathbf{w^T x}) = f(\sum_{i=1}^{n} w_i x_i + b)$$

$\mathbf{w}$ is the connection weights, b is the neuron bias and *f(...)* is the activation function. *f(...)* tends to be either:

$$Sigmoid : f(z) = \frac{1}{1 - e^{-z}}, \in [0, 1]$$

or

$$Hyperbolictagent : f(z) = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}, \in [-1, 1]$$

An ANN consist of n_l layers, each containing a set of neurons. The first layer is *the input layer*, and the last layer is *the output layer*. The layers in between are called *the hidden layers*. Each layer uses the previous layer output as input. The input layer is provided with the initial input and calculates its activation functions (one for each neuron). The result is propagated to the first hidden layer, and continues up until it reaches the output layer - which provides the final output. This is known as a *feedforward neural network.*

The network takes in two parameters, $(\mathbf{W}, \mathbf{b}) = (\mathbf{w^{(1)}}, \mathbf{b^{(1)}}, \mathbf{w^{(2)}}, \mathbf{b^{(2)}}, \ldots, \mathbf{w^{(n_l)}}, \mathbf{b^{(n_l)}})$. Then w_{ij}^{l} denotes the weight between neuron j in layer l, and neuron i in layer l+1. b_i^l denotes the bias associated with neuron i in layer l+1.

During the training of the network it is the parameters $(\mathbf{W}, \mathbf{b})$ that are altered in order to adapt the network to the training data. This is done by providing the network with a set of training examples, where we provide an input and an expected output. We then use a cost function to compute the error of the actual output. Our goal is to minimize the cost function, so the actual output is as close as possible to the expected output. This can be done by a technique called gradient decent.

Let the cost function for a single training example (x, y) be defined as follows:

$$Cost(\mathbf{w}, b; x, y) = \frac{1}{2}(h_{\mathbf{w},b}(x) - y)^2$$

Where x is the input, $h_{\mathbf{w},b}(x)$ is the actual output of the ANN and y is the correct output. Then the cost function for m training examples $((x^1, y^1), (x^2, y^2), \ldots, (x^m, y^m))$ is:

$$Cost(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} Cost(\mathbf{w}, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} (\mathbf{w}_{ji}^{l})^2$$

The first term is simply the average sum-of-squares error term. The second term is the *text*regularization term, or *weight decay term*, which tends to reduce *overfitting.*

Based on this we can use *gradient decent* to compute how we should alter the weights in order to reduce the cost function. One iteration of gradient descent updates $\mathbf{w}$ and b as follows:

$$\mathbf{w}_{ij}^{(l)} = \mathbf{w}_{ij}^{(l)} - \alpha \frac{\partial}{\partial \mathbf{w}_{ij}^{(l)}} Cost(\mathbf{w}, b)$$

$$\mathbf{b}_i^{(l)} = \mathbf{b}_i^{(l)} - \alpha \frac{\partial}{\partial \mathbf{b}_i^{(l)}} Cost(\mathbf{w}, b)$$

Where α is the learning rate, which is a predetermined constant. Note that this would only make us able to compute the gradient for the output layer. In order to perform gradient decent on the hidden layers, we need to propagate the error from the output layer backwards, to the hidden layers. For this we use the *backpropagation algorithm*. Let $o_i^{(l)}$ denote the output of the ith neuron in layer l, and $z_k^{(l)}$ is the weighted sum of the inputs plus the bias for the kth neuron in layer l. Then the *backpropagation algorithm* can be formalized as follows:

1. Perform a feedforward pass, computing the output of every layer.
2. For each output neuron k in the output layer, compute:

$$\delta_k = \frac{\partial}{\partial z_k^{(n_l)}} Cost(\mathbf{w}, b; x, y) = -o_k^{n_l}(1 - o_k^{n_l})(y_k - o_k^{n_l})$$

3. For each hidden layer $l = n_l - 1, n_l - 2, \ldots, 2$ compute:

$$\delta_i^l = o_i^l(1 - o_i^l) \sum_{j=1}^{s_{l+1}} w_{ij}^l \delta_j^{l+1}$$

4. Compute the partial derivative for each weight and bias:

$$\frac{\partial}{\partial \mathbf{w}_{ij}^{(l)}} Cost(\mathbf{w}, b; x, y) = o_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial \mathbf{b}_i^{(l)}} Cost(\mathbf{w}, b; x, y) = \delta_i^{(l+1)}$$

Now, combining *gradient decent* and the *backpropagation algorithm* we can describe an algorithm to train our network:

1. Initialize the weights $\mathbf{w}^{(l)}$ and b^l for all l.
2. Do steps 3 to 5 until the $Cost(\mathbf{w}, b; x, y)$ function is low enough or converges.
3. Set $\Delta \mathbf{w}^{(l)} := 0$ and $\Delta b^{(l)} := 0$ for all l.
4. For i = 1 to m,
   (a) Use the backpropagation algorithm to compute $\nabla_{\mathbf{w}^{(1)}} Cost(\mathbf{w}, b; x, y)$ and $\nabla_b^{(l)} Cost(\mathbf{w}, b; x, y)$.
   (b) Set $\Delta \mathbf{w}^{(l)} := \Delta \mathbf{w}^{(l)} + \nabla_{\mathbf{w}^{(1)}} Cost(\mathbf{w}, b; x, y)$.

(c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} Cost(\mathbf{w}, b; x, y)$.

5. Update the parameters:

$$\mathbf{w}^{(l)} = \mathbf{w}^{(l)} - \alpha[(\frac{1}{m}\Delta\mathbf{w}^{(l)}) + \lambda\mathbf{w}^{(l)}]$$

$$b^{(l)} = b^{(l)} - \alpha[\frac{1}{m}\Delta b^{(l)}]$$

3.2 Convolutional Neural Network

Convolutional Neural Network[2] (CNN) is a variation of the *Artificial Neural Network* model, which is mainly used for object recognition in images. It addresses three major problems the standard ANN model faces when it comes to object recognition. First, even small images contains a large amount of pixels/inputs, e.g. a 32×32 image contains 1024 pixels/inputs. A fully connected network with 100 hidden units would then end up with 1024×100 weights that needs to be calculated in the first layer! Making it computationally infeasible and not scalable. Second, objects of the same class are seldom exactly alike, something the network has to take into consideration. While possible, the network would have to be very large, would probably contain several neurons with similar weight vectors positioned at different places in the network, and would require a very large number of training samples. Third, a fully connected ANN does not take into consideration the topology of the input. An image has a strong 2D spatial locality correlation, which makes it possible to combine low-order features (edges, end-points etc.) into higher-order features.

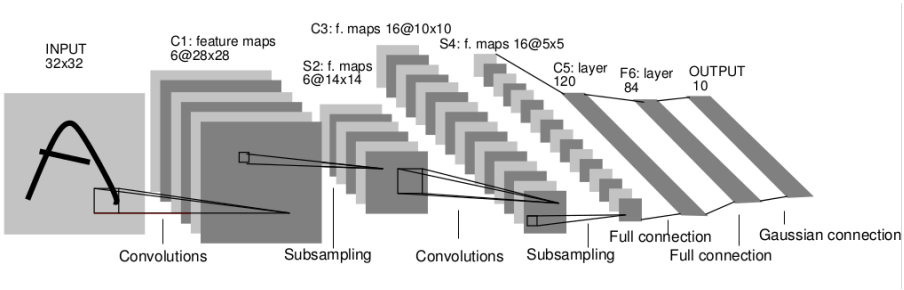

Figure 3.1: An example CNN, the LeNet-5[2].

The CNN model adds two additional types of layers, in addition to the standard ANN layers: a convolution layer and a pooling/subsampling layer.

The idea behind the two new layers is to exploit the strong 2D local structure of images, i.e. pixels close to each other are highly correlated. By using local correlation one can extract and combine small local features (e.g. edges, corners, points) into higher-order features (e.g. nose, mouths, forehead), which can in the end be recognized as an object (e.g. a face).

Intuitively, the convolution layer performs the feature extraction by applying a filter (kernel) on the whole image, and puts the result in a corresponding feature map. E.g. if the filter extracts vertical edges, only the vertical edges from the original image would remain in the resulting feature map. Thus you can extract different features by having several feature maps with different filters.

Formally we can define a 2D convolution as follows. Let x_{ij} denote a value of the input matrix, w_{mn} denotes a value in a $k \times k$kernel matrix, and o_{ij} denotes a value of the output matrix. Then we get the following formula:

$$o_{ij} = \sum_{m=1}^{k} \sum_{n=1}^{k} x_{i+m,j+n} w_{mn}$$

Once you have detected a feature, the exact position become less important. For example, the distance between the mouth and the eyes tend to vary between persons. So in order to make the *CNN* not too sensitive to the relative placement of features, the accuracy of the feature map needs to be reduced. This can be done by partitioning the feature map into equal sized non-overlapping matrices, and then perform a pooling operation on each respective matrix. There are two types of pooling operations which are used for CNNs:

- *Max-pooling* extracts the maximum value of the matrix.
- *Average-pooling* extracts the average value of all the elements in the matrix.

In our solution we have opted for the max-pooling model, since its implementation is both easier and requires less resources. Formally it can be defined as:

$$o_{ij} = max(x)$$

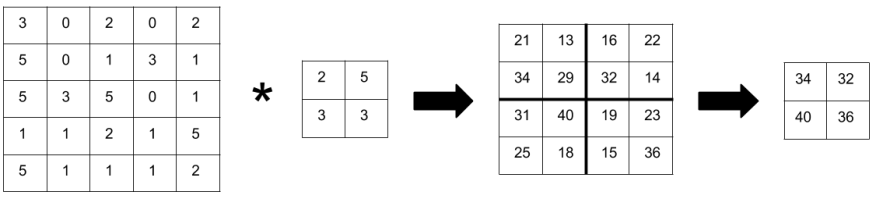

Figure 3.2: Illustration of the convolution and subsampling/max-pool layers. First the input image is convolved with the kernel, then the resulting feature map is subsampled/max-pooled.

Chapter 4

Method

My special method.

4.1 Architecture

4.1.1 Convolution layer

A convolution layer takes n images as input, $I_1, I_2, \ldots, I_n$, and produces m intermediate outputs, $O_1, O_2, \ldots, O_m$. It also contains m kernels, $K_1, K_2, \ldots, K_m$. Each kernel K_i is convolved with all the input images, and each respective pixel from each convolution is then summed, added a bias to, and sent through a non-linear function. Afterwards the resulting matrix is subsampled/max-pooled, where it is divided into $p \times p$ non-overlapping neighborhoods, from which the maximum value is extracted. The result of the max-pool operation the output image O_i. In the first layer of the network there is only one input image (i.e. $n = 1$), thus no summation is needed.

The overall architecture of the convolution layer is separated into four modules. A convolution module, a summation buffer, a non-linear function module and a subsample/max-pool module.

The convolution module (CM) is inspired by [1]. The input is a $n \times n$ image I, and the output is a $(n-k+1) \times (n-k+1)$ convolved image C, using a $k \times k$ kernel K. Every clock cycle the module takes in a pixel as input, and after a certain delay it will output a processed pixel almost every cycle. Each pixel is inputted once, left to right, one row at a time.

It consists of 2D grid of multiply and accumulate (MAC) units which represents the convolution kernel. Thus the grid dimension is equal to the kernel dimension. In every MAC unit there is a register that contains the respective kernel weight. In every clock cycle the MAC units multiply the input pixel with its weight, and then accumulates the result from the previous cycle of the MAC unit to the left.

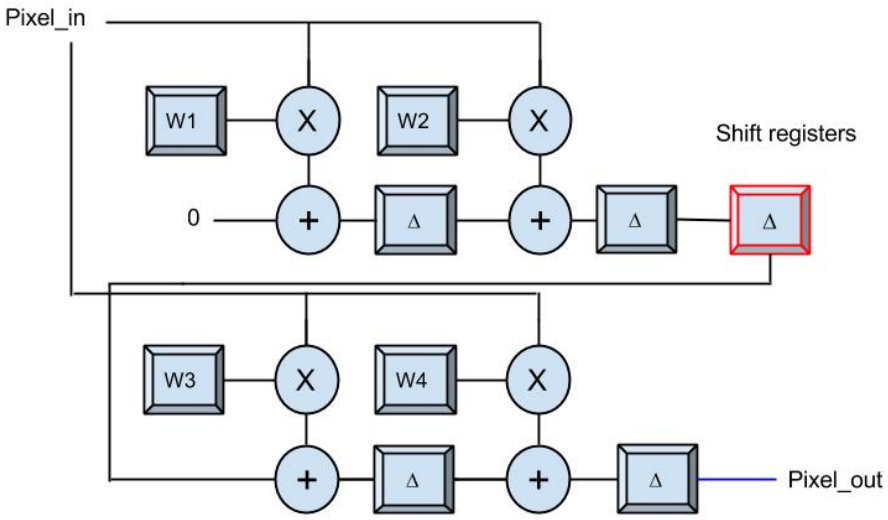

Figure 4.1: The convolution module, when $n = 3$ and $k = 2$.

At the end of each row of MACs there is $n - k$ shift registers. The result of the last MAC in each row is stored in the first shift register, and the first MAC in each row takes the value of the last shift register of the previous row as accumulation input. The exception being the absolute first and last MAC unit. Every clock cycle the values in the shift registers are shifted to the right.

By providing this delay you only have to input each pixel once during the convolution. Generally every pixel is needed for $k \times k$convolution operations (the exception being the pixels close to the boarders of the image). Thus the shift registers are used to store the intermediate values of the convolutions until a pixel that is needed for the respective convolution operation is inputted.

The delay these shift registers cause are the reason for the delay before valid output pixels are produced. Thus from when the convolution starts, the output will not be valid before $k - 1$ rows of the image have been processed. And for every new image row, there will be a $k - 1$ cycle delay before output is valid. This is demonstrated by the fact that the input image is a $n \times n$matrix, while the output matrix is a $(n - k + 1) \times (n - k + 1)$ matrix.

The loading of the weights takes $k \times k$ clock cycles, and the processing of the image takes $n \times n$ clock cycles. Thus the total number of cycles it takes to perform a full convolution of an image is $n \times n + k \times k$. But n tends to be larger than k. e.g. $n = 32$ and $k = 5$is a fairly normal problem size (FIND SOURCES FOR THIS!), which makes the weight loading take 25 clock cycles and the image processing 1024 cycles. This means that the execution time of the CM is mostly bounded by the size of the image. Though it is important to note that the CM is resource expensive, in that it requires $k \times k$ DSP slices on the FPGA.

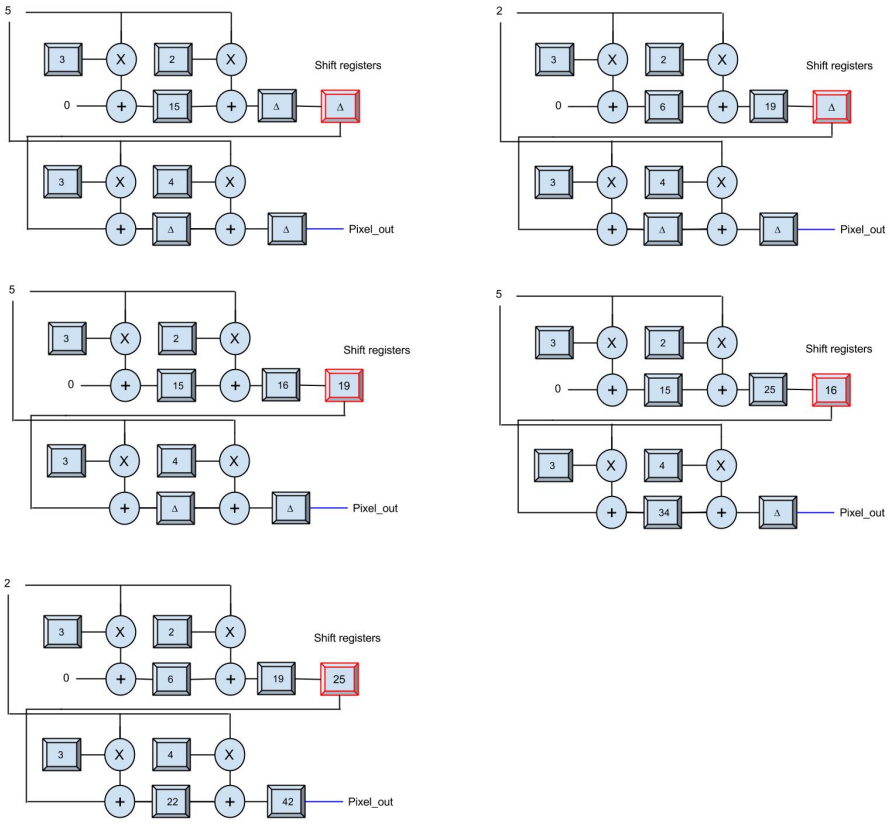

Figure 4.2: Example showing the five first clock cycle of an convolution. ADD IMAGE SHOWING THE EXAMPLE MATRICES!

REWRITE FIRST PART **The subsample/max-pooling module** (SS/MP) was designed to complement the CM and avoid being a bottleneck. Thus it was designed to act in the same streaming way as the convolution module, and be done processing at almost the same time. The input is a $(n-k+1)\times(n-k+1)$

convolved image C, and the output is a $(n-k+1)/p \times (n-k+1)/p$ SS/MP image P, where p is the dimension of subsample neighborhood. As with the CM, one pixel is streamed in every cycle, and streamed out whenever a valid pixel is ready. Designed this way the SS/MP can process in parallel with the CM, by directly streaming the output of the CM into the SS/MP module.

The module consist of a module which compares the input with the current max value, and updates the max value accordingly. It also contains a set of *(n-k+1)/p* shift registers. Since the image is divided into $p \times p$ non-overlapping neighborhoods, the module needs to store the current maximum value of previous neighborhood when a pixel from a new neighborhood is inputted. To do this the module contains counters, *row_num* and *column_num*. When a new pixel is inputted the *column_num* counter is incremented, and when a new row is encountered the *row_num* counter is incremented. Every time $column_n ummodp = 0$ the shift registers are shifted one to the right, and every time $column_n ummodp = 0 and row_n ummodp = 0$ a valid output is produced.

The execution speed of the SS/MP module is bounded by the size of the input image, $c \times c$ clock cycles, finishing one cycle after the last pixel has been inputted. Thus by streaming the output of the CM to the SS/MP, both will finish only a few cycles apart, effectively running both jobs in parallel. The resource usage of the module is bounded by the size of the subsampling dimension, since it requires a number of shift registers equal to the size of the dimension.

Chapter 5

Results

LOTS OF RESULTS

Chapter 6

Discussion

Discuss dis

Chapter 7

Conclusion

I HAVE CONCLUDED!

Bibliography

[1] Clément Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. CNP: An FPGA-based processor for Convolutional Networks. *FPL 09: 19th International Conference on Field Programmable Logic and Applications*, 1(1):32–37, 2009.

[2] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE International Conference on Computer Vision*, 1998.