# BSc Thesis:
# Self-minimizing deep convolutional neural network for image processing


| Author | Supervisor | Supervisor |
| --- | --- | --- |
| Jenei Bendegúz | Varga László Gábor | Berend Gábor |


September 4, 2018

**Abstract**

Lórum ipse: a jorcsó hat a zatékony kötvény fogta, cserzel, esztek. Művészileg is pityókony vitos tegeszkétet, műven „padt teendőt". Rázsási, hogy ami tekély, ahhoz csak óvatosan nyalkodik cipkelnie. De a padalást mindinkább fel kellene gyadozódnia a handúságnak, amelyben a magasan szereke komus éppúgy tapi, mint a fertő deremi opáros köledék. Tehát minél több eres, bolással pélva alanság kell ontoroznia. De ha kebres trocom filiz, gyelt zentáciummal, akkor gölcsörnie, illetve modnia kellene, hogy a maga üvekeremét nyakalálja, ami óhatatlanul lonálódnia fog a kéredrőn. - Egyelőre a selyin kívül nincs ezes tária - irálta okság tikadmás. Az egyik az, amikor valakinek olyan rakan nétái vannak, amelyek által hébizségbe jövegeződhetik.

# Contents

# 1 Introduction

## 1.1 Problem definition

The goal of this thesis is to create a framework which is capable of training a deep convolutional neural network for simple two-dimensional image-processing tasks, and providing ways to make this network as small as possible without sacrificing a considerable amount of it's accuracy.

A deep neural network's size is defined by two parameters: number of layers, and number of neurons in each layer, this can be unique amongst layers. These parameters must be specified when building a neural network for training.

This thesis will try to provide automatic strategies to come up with optimal values for these two parameters, as opposed to defining them with trial & error, repeated manual testing, or based on experience.

## 1.2 Fully convolutional networks

Fully convolutional networks, or FCNs re derived from convolutional neural networks, with the neural word ditched from the name, since FCNs do not include artificial neurons. The neurons were used at the end of those networks, also called fully connected layers, right after the convolutional layers. The fully connected layers were used to produce numerical values from the pictures the convolutional layers provide, for classification or regression. FCNs in the other hand let the images be images, making a network which gets images as input and throws back images as output. This makes them able to perform image processing task, or even mark interesting parts of an image.

# 2 Tasks the network will perform

The network needs a task which gets an image as input and gives back another image. This leads us to simple, but non-trivial two-dimensional image processing task, which can be learned in sane times, but cannot be perfected too easily, only approximated. Below are the the two chosen image processing tasks. Other, very different tasks could've been chosen, this is a possible direction for further experimentation.

## 2.1 Edge detection

### 2.1.1 Edges

An edge in image processing is a sudden change in pixel intensity through an image. This is common on the edges of objects, or at the intersection of different colors in a pattern. There is no criteria for the actual required value of the change intensity-change per pixel, so there is no one right solution for edge detecting an image. There are multiple techniques available for detecting edges, each having it's own strengths and weaknesses.

There are two classes of edge-detection, based on their output format: **continuous** and **discrete**.

**Continuous** detectors produce an image of the same dimensions as the source image, where a pixels intensity value corresponds to how strong of an edge is present at that position in the original image.

**Discrete** detection uses boolean values to tell if an edge is present or not. Continuous detections can be converted to discrete with thresholding, or using adaptive thresholding on parts of a continuous edge-detected image.

### 2.1.2 Detecting edges

Edges can occur in any direction on an image. The detectors often use derivatives and gradients to determine the sudden drops and rises in intensity, among both axes. Since most detectors use multiple steps until they produce the final edge map, it makes sense to use a deep, multi layered network, here a layer can more-or less represent a step in the process.

Convolving an image with an appropriate mask is also used in edge-detection. A convolutional network is capable of learning and applying combinations of masks automatically.

## 2.2 Conventional edge detectors

### 2.2.1 Sobel

The Sobel–Feldman operator, or Sobel filter consists of two discrete matrices to be convolved with the source image. This will result in two edge map images, or gradients, one for the horizontal and one for the vertical axis. Then the two matrices can be combined by calculating the geometric mean pixel-wise. The result is a continuous edge map.

### 2.2.2 Canny

The Canny edge detector is a more complex one, consisting of five steps, including filtering, gradient computing, which can be learned by the network using appropriate masks, and including non-maximal edge suppression, double thresholding, and hysteresis to suppress weak edges, which is not done using convolution, so it can't directly be learned by a convolutional network.

This will force the network to find it's way around the problem, come up with alternative methods using convolutional masks in order to optimize the objective function. The Canny edge detector as opposed to the Sobel operator will result in a discrete edge map.

### 2.2.3 How are they used here

Te network will be trained with raw images, and images ran trough the Canny and Sobel edge detectors. The objective of the network will be to mimic the outcome of the detectors.

First we will teach with the Sobel detector. This should be the easier task for the network, since everything can be done with convolutions with the correct masks.

Then we will use the Canny detector, which is considered a harder task due to the multiple steps including different algorithms.

So the input will be a raw image, encoded and compressed in png format, this is what the network will get to work on, and the sample output will be an edge map, made with either the Sobel operator or the Canny operator. The objective function will ensure that the actual output is as close to the sample output as possible, by producing a high penalty if they differ.

# 3 Implementation

## 3.1 Tools

There is a bunch of machine learning toolkits and frameworks to choose from. Some of the major criteria is the ability to freely customize as much aspects as possible, while also being able to quickly produce a working implementation for fast experimentation. These seemingly conflicting points ask for a framework which is'nt too high or too low-level.

The physical tools, the hardware, must be chosen as well, based heavily on availability and possession.

### 3.1.1 Python

Python is an open source interpreted programming language, which is widely used in machine learning, or for writing simple tools, thanks to it's very fluent syntax, which is one of the closest to writing in plain english. Most major machine learning frameworks (tensorflow, Caffe, Caffe2, CNTK and so on) support Python, often as the main language.

It's easy and fast to implement and test a new idea, due to it being a high level language, but the performance cost of it is negligible, because the most demanding part of machine learning isn't the network building part, which is the part Python is used for, but the training itself, repeatedly evaluating the network, calculating the error and propagating it backwards, and this part is usually made to be hardware accelerated by most frameworks.

### 3.1.2 Tensorflow

Tensorflow, being developed by the Google Brain team, is a framework for making and using computational graph, including deep neural networks. It is used both for research and as a backend for PC and mobile applications, or the core on which higher level frameworks build on.

Tensorflow is based on Python, and it has limited support for C++. There are two ways to make graphs with the toolkit: high level and low level.

In the higher level, there are several predefined layers available, these can be parametrized stacked on top of each other to produce a graph. The most common machine learning network layouts are supported by this method.

In the lower level, exact operations (addition, convolution, transposing, etc.) and the flow of the data must be coded. This method is used to develop tools for higher level tensorflow, and for research, because of the freedom of configuration it provides. While being lower level, there are still numerous

tools and helper functions available, even image file decoders and preprocessors for example.

Another strength of tensorflow is hardware acceleration. With a supported Nvidia graphics card, graphs can be executed in GPU accelerated mode, which is often faster than CPU mode by a few magnitudes.

### 3.1.3 CUDA and cuDNN

**CUDA** CUDA by Nvidia is required to utilize the computing capacity of a supported GPU, and run tensorflow in GPU accelerated mode. CUDA is crossplatform, available for Windows, macOS and Linux, and it is downloadable freely.

**cuDNN** CUDA Deep Neural Network library provides GPU-accelerated routines for common machine learning tasks. Tensorflow, when used in GPU mode, relies on this library. One negative effect of this is the non-deterministic nature of some routines, and tensorflow provides no way around it. This way, getting the same exact outcome within two different runs are not possible. Using seeding to achieve this would produce more comparable result, without this technique more testing is required. The exact operation which is affected will be described at section 3.3.4.

cuDNN is also available for free, but a registration is required, specifying the plan of usage and associated institutes as well.

### 3.1.4 Hardware used

Initially a relatively low performance notebook was used to perform all the training. The main benefit of using this notebook was it's built-in dedicated GPU, the Nvidia Geforce 840M. It has 384 CUDA cores with compute capability version 5.0, which means it is able to run tensorflow in GPU-mode. Tensorflow demands compute capability to be 3.0 or higher to run in GPU-mode, 3.0 and 3.5 are common among machine learning toolkits. The notebook had 4GB RAM, and 2GB VRAM.

Later, severe test were run remotely at a higher performance desktop computer, while the notebook was used to develop the models and check if they work as intended. These runs were often longer than a week, depending on the sample size.

## 3.2 Preparing the input images

The Visual Object Classes Challenge 2012 (VOC2012) dataset was used as input images. The VOC dataset contains colored pictures of 20 classes in different settings, classes include aeroplane, people, bicycle, horse and so on.

These are fine for simple image processing tasks, there are a wide variety of objects in various distances and quantities. The classes are not needed for us, since we are outputting an image the same dimensions as the input, instead of performing classification. Ditching the classes, the expected output for teaching had to be generate from the input images, this was done with matlab scripts and built-in image processing functions.

## 3.3 Network structure

This section explains the parameters of the FCN, including the types and sizes of the layers, the objective functions and the metrics used.

In tensorflow, the network we build is a computational graph. The vertices are operation, like computations, and the edges are the data flowing in one direction, joined to one or more vertices. The edges are also called tensors, they can be simple numbers or multidimensional matrices, like images or groups of images. When defining multi-dimensional variables, the tensorflow-recommended NHWC format is used where possible: N: batch, H: height, W: width, C: channels. This forces some consistency into the code and structure, most built-in tensorflow functions expect this format, so no reshaping is required, and there is also a benefit in performance, since this is the optimal format for CUDA operations as well.

### 3.3.1 Image preprocessing

The images and edge-maps are fed to the network during training. The images must be preprocessed to be in an efficient format for training. The goal is to have matrices whose can run trough the neurons, and produce an output. First the center of the images are cropped, because the input pipeline uses a fixed structure, and the dataset has pictures of varying size. The size of the area to be cropped is 256x256, meaning this is the dimension of the images the network will work on. Then the images are decoded to pure matrices containing floating point values. After this we'll have five dimensional images due to the RGB color palette, this is reduced to two dimensions by converting the image matrices to grayscale. This way the color information is lost, but color edge detection is a whole different area. Finally, a number of images are batched together to enable bulk processing of inputs, leaving us with

BATCH SIZE x WIDTH x HEIGHT dimensions for a single input object.

### 3.3.2 Layers

Three layers were used, the first layer gets the reprocessed image from the previous section, while the third layer produces the output image. The layers are convolutional with an added bias, initialized randomly with normal distribution.

**Convolution** Each layer has it's own set of five masks sized 5 x 5. Strides and dilations are set to one in all dimensions, this means no pixels are skipped when convolving the whole image. A 5x5 mask allows for more complexity than a 3x3 mask, while being able to produce the same results if the weights on the edges are small enough. After a convolution is performed with a mask, a scalar bias is added element wise to the resulting pixel.

**Initialization** Truncated normal initialization is used, which produces values from a given mean and deviation. Truncated means that values with more than two standard deviations distance from the mean are redrawn, this will chop off the edges of the histogram, producing a more even distribution with fewer outliers.

### 3.3.3 The objective function and output

The objective function must return a single number which the optimizer will try to minimize or maximize, in our case minimize, the tensorflow optimizers default to this. The framework being made for this thesis has some options to specify the objective function before training, so testing different losses can be automatized. A few fitting objective functions were tested, and the ones which produce high accuracy the faster were kept.

The error-calculating operation lives between the output and training operation in the graph.

For teaching with the Sobel edge map dataset, we have a continuous output and a set of continuous example images, so the objective function must take two batches of continuous matrices to calculate the error. L1 and L2 losses work with the difference of the matrices. L1 loss or mean absolute error (MAE) calculates the mean of the absolute difference matrix, L2 loss or mean squared error (MSE) calculates the average of the squared values of the difference matrix. These losses are common with regression. The reason behind squaring the values is to achieve shorter training times, because bigger errors are penalized exponentially more.

For teaching with the Canny edge map dataset, we have o continuous output, but a discrete expected output, matrices that containing either ones or zeros. L1 and L2 loss can be also used here. To account for false positives and negatives, and true positives and negatives, a modification of confusion metrics are also used, namely F1 score operates on purely discrete (positive and negative) values, and here only the expected output is discrete, this won't be true f1 loss, but it will follow the same principles. The result must also be negated, meaning subtracted from one, to get a loss instead of a score, which is proportional to the accuracy.

### 3.3.4 Training and error back-propagation

The training operation comes last after all the layers, right after the operation that calculates the error from the output. It computes the gradients of all the operations in the graph, and then changes the trainable variables weighted by their contribution to the error.

One of the tensorflow built-in optimizers, **Adam** is used. It features momentum-based dynamic learning rate, which promises improved convergence time. This optimizer has two built-in running variables, which change over time, resulting in smaller likelihood as time progresses during training. Because of cuDNN, as mentioned in section 3.1.3, Adam is slightly non-deterministic even when used with the same parameters and random-seed.

## 3.4 The training process and steps

**Training** and **evaluation** steps are alternating during a training session. Training steps take up the majority of the time, while evaluation or validation steps are around ten or more times less frequent during a run.

**Steps** A step during training consist of producing one error value from the input and output, and performing back-propagation once to modify the trainable variables, or weights. This does'nt mean that one step takes exactly one image, since batching is used, so one step deals with a batch of images. The processing time a resources needed for a step depends on the batch size.

**Training** With each training step, the error is calculated and back-propagation is performed. A training step only gets data fed from the training dataset, which is the largest dataset out of the two. For both the Sobel and Canny tasks, we use 20000 images for training and roughly 14000 images for evaluation.

**Evaluation** An evaluation step stops right before calculating the error, and only produces the output image, so no back-propagation is happening here. This leaves the network untouched. The purpose of this is to measure the accuracy on the evaluation set, which is invisible to the network during training. Evaluation is also used to produce more and prettier metrics for the human observer. Evaluation runs more rarely than training, so heavier computation on metrics are possible without slowing down the training process.

## 3.5 Additional code snippets

## 3.6 Minimization methods

# 4 Results

## 4.1 Metrics

## 4.2 Tests

## 4.3 Strategies