University of Szeged

Department of Informatics

Implementing a self-scaling convolutional neural network

Bachelor's thesis

Author:

Jenei Bendegúz

student

Supervisor:

Varga László Gábor

Senior lecturer

Supervisor:

Berend Gábor

Senior lecturer

Szeged

2018

Contents

Tartalmi összefoglaló
(content summary in Hungarian)

Ez a dolgozat a *gépi tanulás* és a *digitális képfeldolgozás* kategóriákhoz tartozik. Egy *teljesen konvolúciós hálózatot* alkalmaz, ami a gépi tanulásban haszlálatos technika, hogy *éldetektálást* végezzen, ami egy digitális képfeldolgozásbeli feladat. A hangsúly nem a tanításon vagy a feladaton van, hanem a hálózat paramétereinek optimalizálásán, konkrétabban a rétegek és a rétegbeli konvolúciós maszkok számának csökkentésén, miközben a hálózat pontossága változatlan marad. Már léteznek írások hasonló témával, lásd a [1], [2] és [3] hivatkozásokat.

A hálózat előnyös méretének meghatározását egy tantási folyamat végzi, ami képes elmenteni állapotát és később visszatölteni azt. Ezen kívül képes ideiglenesen módosítani a belső struktúráján. Ez különböző konfigurációk tesztelésére és összehasonlítására van használva. Ezek a technikák különböző mértékekkel együtt vannak alkalmazva. A tanítás akkor áll meg, amikor egy kilépési feltétel teljesül. Az eredmény egy olyan konfiguráció, amit az alkalmazott stratégia optimálisnak talált.

Naív és intelligens stratégiák is tervezve lettek annak érdekében, hogy kiderüljön melyik működik jobban. Egy egyszerű stratégia véletlenszerűen választ a maszkok közül, és ideiglenesen letiltja a kiválasztottakat. Ezután ellenőrzi, hogy folytatódhat-e a tanítás nélkülük az addigihoz hasonló pontossággal. Egy intelligens stratégia különböző mértékek alapján dönt. Mindkét stratégia bizonyulhat hasznosnak bizonyos esetekben.

A keretrendszer *Python*-ban lett kódolva a *tensorflow* könyvtár [4] használatával. A teljesítmény növelése érdekében *CUDA* magokat tartalmazó *NVIDIA* típusú videókártya segítségével hardveresen gyorsított tanítás volt alkalmazva. Ezt a tensorflow teszi lehetővé a *CUDA* és *cuDNN* könyvtárakkal.

Task proposal

The task of the candidate is to create a framework in which convolutional neural networks can be trained, and the size of these networks are determined automatically. The framework should be able to perform the following:

- Training of a fully convolutional neural network to solve a simple image processing task (e.g., edge detection, automatic thresholding, etc.).

- Determining the size of the ideal neural network. An ideal sized neural network solves the problem with a given layer count and structure with the maximum possible accuracy, while keeping the neuron count minimal among layers.

Content summary

This thesis falls into the topics of *machine learning* and *digital image processing*. It uses a machine learning technique of training a *fully convolutional network* to perform *edge detection*, which is an image processing task. The focus is not on the training or task itself, instead it is on optimizing the parameters of the network, specifically minimizing the number of layers and the number of masks in the layers, while not sacrificing the accuracy of the network. Similar topics have already been researched before, e.g. in [1], [2] and [3].

The beneficial size for the network is determined by a training process which is capable of saving its state and returning to it later. It is also capable of altering the inner structure of the network temporarily. This is used to test and compare different configurations. The previous techniques are combined with different metrics The training stops when an exiting criterion is met. This results in a network configuration that is regarded as optimal by the current strategy.

Naive and more intelligent strategies were designed, to see which one performs best. A simple strategy would randomly select masks in the network and disable them temporarily. Then it would check if the training can continue without them with around the same accuracy. An intelligent strategy would choose based on various metrics and indicators. There might be use cases for both approaches.

The framework was coded in the *Python* language, utilizing the *tensorflow* library [4]. To aid performance, the training was hardware-accelerated using *NVIDIA* graphics cards with CUDA cores, which are operated by tensorflow through the *CUDA* and *cuDNN* libraries.

Introduction

In this thesis, a special computer program will be written, which uses a convolutional neural network. It will be special because with the use of a neural network it will be able to dynamically refine how it works to produce a better result, i.e., learn. While classical algorithms are created using human intuition or mathematical formulation, machine learning algorithms are given examples of the right answers it should produce with the given data, allowing it to automatically develop ways of solving a specific problem based on feedback about its own performance.

Integrating the technique of convolution into these networks will make them especially efficient in working with images. This opens up new possibilities, like face recognition, object detection or image reconstruction. Convolutional neural networks are already used for these tasks in areas like security or industrial automation [5].

Because images contain large amounts of pixels, convolutional neural networks are computationally demanding, requiring expensive hardware and long training times. A larger neural network has more neurons, so it must perform more operations, making it more demanding than a smaller one. This is why it is important to use a network which has just enough neurons to perform the task accurately, but no more, to avoid unnecessary calculations and computing time.

This thesis attempts to come up with automatic ways of determining the right neuron count and network size for a given problem. This way, no repeated testing and trial-and-error is required, while manually trying different network sizes, because the network will try to find a fitting size for the given problem automatically without requiring human intervention. Existing automatic approaches include exhaustive grid search over a set of parameters. This means repeated training with each combination of the parameters. The method described in this thesis is an alternative way of finding the right parameters. The network is measured and changed during training, as opposed to running it multiple times as in grid search. This is in hope of saving computational resource and reducing training time.

The benefits of using properly sized neural networks include portability and resource

efficiency. With less demanding networks, cheaper hardware such as mobile devices will be capable of running convolutional neural networks, making them more widely available. The computational costs can also be reduced, producing results faster and with more modest hardware.

Chapter 1

Technical overview

1.1 Minimal fully convolutional networks

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 provide ways to make this network as small as possible without sacrificing a considerable amount of its accuracy.

First is a short description of *artificial neural networks*, or *NNs*, on which convolutional neural networks are based. Artificial neural networks are inspired by the biological neural networks in the brain. An artificial neuron is a simple model of a biological neuron. They can have any number of inputs, originating from other neurons, or directly from the input, sensory organs in case of real neurons. They have exactly one output, which can be connected to an input of another neuron, or channeled to the output. When a neuron gets its inputs, it produces a weighted sum of them, adds a bias, applies a function called activation function. Finally it puts the result on its output.

The *weights* and *bias* within a neuron are its unique parameters that define is characteristics. The activation function is used to introduce some nonlinearity. It is usually a simple linear or exponential function, often cutting negative values, or values falling outside of a range, substituting them with zeros. Neurons are usually grouped into layers, where they do not connect to each other, only to other immediate (also called hidden), input or output layers. When an immediate layer passes its outputs to another immediate layer, we are speaking of a *deep neural network (DNN)*.

To mimic the learning capability of intelligent species, artificial neural networks perform feedback based learning by altering the unique weights and biasing of the neurons. A loss function is defined for every neural network, which produces a value which is

proportional to how well the network performs. For example, in case of classification, it is proportional to the confidence of the network about an incorrect class, where bigger values mean poorer performances. Using this value, the weight and bias parameters of the neurons are altered based on their contribution to the error or score. This improves score value and thus the performance of the network. This process is called *error back-propagation* or *training*.

The size of a deep neural network is defined by two parameters: *number of layers*, and *number of neurons* in each layer, which can vary between 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 beneficial values for these two parameters, as opposed to defining them with trial & error, repeated manual testing, or based on experience.

Fully convolutional networks, or *FCNs* are derived from convolutional neural networks. Convolutional neural networks were first used for classification [5]. Their first layers were convolutional layers performing feature extraction. The last layers were fully connected layers of neurons, performing the classification or regression using the extracted features. FCNs let the images be images, making a network which gets images as input and throws back images as output. This makes them able to mark interesting parts of an image, or perform numerous other image processing tasks.

The network needs a task which gets an image as input and gives back another image. This leads us to a simple non-trivial two-dimensional image processing task that can be learned in feasible times. Meanwhile the quality of the edge maps are somewhat subjective, each method having its strong and weak areas. Below are the two chosen image processing tasks. Other, very different tasks could have been chosen, which is a possible direction for further experimentation.

1.2 Edge detection

An *edge* in image processing is a sudden change in pixel intensity within an image. This is common on the edges of objects, or at the intersection of different colors in a pattern. There are no criteria for the actual required value of the intensity-change per pixel, so there is no single right solution for edge detection. There are multiple techniques available for detecting edges, each having its 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 by thresholding, or using adaptive thresholding on parts of a continuous edge map.

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 could make sense to use a deep, multi-layered network to see if a layer could 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.

1.2.1 Conventional algorithmic edge detectors

The following two edge detectors are important, as they are widely used algorithms, their output will serve as examples of edge maps.

The *Sobel–Feldman operator*, or *Sobel filter* [6] 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.

The *Canny edge detector* [7] is a more complex one, consisting of five steps, including filtering, gradient computing, non-maximal edge suppression, double thresholding, and hysteresis to suppress weak edges. The first two steps can be carried out with convolutional masks, while the latter ones are algorithms. The Canny edge detector as opposed to the Sobel operator will result in a discrete edge map.

1.2.2 Using the detectors

Sobel edge detection and the first two steps of Canny edge detection can be carried out with convolutions, so they can theoretically be learned by the network. The latter steps of Canny edge detection are algorithmic, thus they cannot be directly learned. This will force the network to find its way around the problem, come up with alternative methods using convolutional masks in order to optimize the objective function.

The 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 train 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.

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.

1.3 Tools

There are a variety 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 not too high or too low-level. The physical tools, the hardware, must be chosen as well, based heavily on availability and possessions.

1.3.1 Software

Python is an open source interpreted programming language, which is widely used in machine learning, or for writing simple tools, thanks to its fluent syntax. Most major machine learning frameworks (tensorflow, Caffe, Caffe2, CNTK and so on) support Python, often as the main language. It is 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. The most demanding part of machine learning is not the network building part, which is the part Python is used for. It is the training itself, repeatedly evaluating the network, calculating the error and propagating it backwards. This part is usually made to be hardware accelerated by most frameworks.

Tensorflow, being developed by Google, is a framework for making and using computational graph, supporting the creation of deep neural networks. It is used for both 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, computational graphs can be executed in GPU accelerated mode, which is often faster than CPU mode by several magnitudes. The version used was 1.6, the latest stable release at the time of coding.

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, the *CUDA Deep Neural Network* library provides GPU-accelerated routines for common machine learning tasks. It is also available for free, but a registration is required, specifying the plan of usage and associated institutes as well. 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 in two different runs is 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 2.3.2.

1.3.2 Hardware

Initially a relatively low performance notebook was used to perform all the training. The main benefit of using this notebook was its 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, several tests were ran 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. This PC was an Intel Core i7-6700 with 16GB RAM, and it had an NVIDIA Tesla K40 GPU with 12GB VRAM and 2880 CUDA cores.

Chapter 2

Implementation

2.1 Code structure

The framework is spread across multiple source files dividing the code into groups based on roles.

$fcn\_run.py$	$: the\ interface\ of\ the\ framework$
$fcn\_train.py$	$: training\ session\ management$
$fcn\_model.py$	$: network\ definition$
$fcn\_input.py$	$: data\ reading\ \&\ preprocessing$
$metrics.py$	$: evaluation\ \&\ visualization\ metrics$
$shrink\_functions.py$	$: selection\ strategies$

A training run must be defined and initiated in *fcn_run.py*. It uses high level code to define the parameters of the run, usually in a few lines. It is an interface to *fcn_train.py*, where the main functionality is implemented. The model creation is implemented in *fcn_model.py*, code in this file takes care of building a model on the input prepared in *fcn_input.py*, where the input is read, and the images are preprocessed. Helper functions are kept in *metrics.py*, these are called during evaluation, and in *shrink_functions.py*, where the selection strategies are realized for minimization. The minimization process is described later in 2.4.

2.2 Preparing the input images

A large and openly available set of images were needed as input for the training. The Visual Object Classes Challenge 2012 (VOC2012) dataset [8] was chosen for this purpose.

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. Without the classes, the expected output for training had to be generated from the input images, this was done with matlab scripts and built-in image processing functions.

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 which can run trough the FCN 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 256×256, 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 will have three 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. Effectively using the color channels to improve edge detection precision is a different area of research, which is not discussed here. Finally, a number of images are batched together to enable bulk processing of inputs, leaving us with an input shape of $batch\ size \times image\ height \times image\ width$. With the batch size set to 3, the actual shape is $3 \times 256 \times 256$.

2.3 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 operations or placeholders while 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, images or groups of images for example. 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.

Our FCN, as seen on figure 2.1 (created with [9]), is made up by three layers, 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 a predefined distribution.

Figure 2.1

Each *convolutional layer* has its own set of five masks sized 5×5. 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 3×3 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 elementwise to the resulting pixels.

Truncated normal initialization [10] is used before the first training. The values are derived from a general normal distribution parametrized by 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. The mean and standard derivation were set to 0.0 and 5×10^{-2}.

The *activation function* is *ReLu* (rectified linear unit) [11] across the whole network. ReLu is a simple linear rectifier, returning zero for negative values and the original value otherwise. It is a convenient choice because due to its linearity it does not deform the images. It is also fast to compute and widely used for convolutional networks. Since ReLu has no upper limit, the final output must be scaled down and rounded to fit an image format standard, usually $[0; 255]$ or $[0; 1]$. First we shift the range if needed to get positive values, then scale it to $[0; 255]$. Finally we convert the image to *png* format using rounded and unsigned 8-bit integers.

2.3.1 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 this will be minimization, 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 gets the output and expected output as inputs, its result is used by the training operator, and the error can be written to the console for information.

Training 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:

$$MAE = \frac{1}{N}\frac{1}{H \times W}\sum_{y=1}^{H}\sum_{x=1}^{W}\left|O_{n,y,x} - \hat{O}_{n,y,x}\right| \quad ,$$

$$MSE = \frac{1}{N}\frac{1}{H \times W}\sum_{y=1}^{H}\sum_{x=1}^{W}\left(O_{n,y,x} - \hat{O}_{n,y,x}\right)^2 \quad ,$$

where N is the batch size (3), H and W are the image dimensions (128×128), O is the actual output of the network and $\hat{O}$ is the expected output. 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.

Training with the Canny edge map dataset, we have a 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 (*FP* and *FN*), and true positives and negatives (*TP* and *TN*), a modification of the confusion metric F1 score is also used. F1 score originally operates on discrete values, and here only the expected output is discrete, this will not be a true F1 score, but it will follow the same principles. The result must also be negated (subtracted from one) to get a loss, which is inversely proportional to the accuracy. The definition of F1 score is

$$\frac{2TP}{2TP + FP + FN} \quad ,$$

where we can replace the addition of both types of negatives with the error value to simplify the formula to

$$\frac{2TP}{2TP + E} \quad ,$$

where E is the error of the current example. The negated version to produce a loss instead of a score becomes

$$1 - \frac{2TP}{2TP + E} = \frac{2TP + E - 2TP}{2TP + E} = \frac{E}{2TP + E} \quad .$$

Defining F1 score and loss for the model in tensorflow:

fcn_model.py

```
...
def f1_loss(output_image, output_image_train):
    difference = tf.reduce_sum(tf.abs(output_image -
        output_image_train), [1,2,3])
    TP = tf.reduce_sum(output_image * output_image_train, [1,2,3])
    f1 = tf.reduce_mean(difference / (difference + 2 * TP))

    return f1

def f1_score(output_image, output_image_train):
    difference = tf.reduce_sum(tf.abs(output_image -
        output_image_train), [1,2,3])
    TP = tf.reduce_sum(output_image * output_image_train, [1,2,3])
    f1 = tf.reduce_mean(2 * TP / (2 * TP + difference))

    return f1
...
```

And outside of tensorflow with NumPy as a metric to use during validation:

metrics.py

```
...
def f1_score(output_image, output_image_train):
    if output_image.shape != output_image_train.shape:
        print('f1_score: image sizes not equal, exiting')
        import sys
        sys.exit()

    difference = np.sum(np.abs(output_image - output_image_train))
    TP = np.sum(output_image * output_image_train)
    f1 = np.mean(2 * TP / (2 * TP + difference), dtype=np.float64)

    return f1
...
```

2.3.2 Training and error back-propagation

The training operation is joined last after the layers, right after the operation that calculates the error from the output, but it reads and manipulates almost the whole graph. 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* [12] is used. It features momentum-based dynamic learning rate, which promises improved convergence time over SGD [13], on which it is based. SGD is short for *stochastic gradient descent*. It is a gradient descent

variant which only uses a random subset of the training examples to approximate the gradients in each training step. This makes the calculations cheaper and shortens the training steps. The approximation becomes more accurate as we increase the size of this subset. There is a variety of SGD modifications developed that add a dynamic and adaptive behavior, Adam is one of them. Adam has two built-in running variables, which change over time, resulting in smaller weight updates as time progresses during training. Because of cuDNN, as mentioned in section 1.3.1, Adam is slightly non-deterministic even when used with the same parameters and random-seed. The cause of the non-reproducible results was the implementation of some reductions using asynchronous CUDA atomic operations. This has been resolved in later tensorflow versions, but was still present in the used version (1.6). See [14] and [15].

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. Both steps iterate on their own separate set of data. These two sets are sometimes created by splitting one large dataset by a ratio favoring the training split. For both the Sobel and Canny tasks, we use 20000 images for training and roughly 14000 images for evaluation.

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 not 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.

With each *training step*, the error is calculated and back-propagation is performed. 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 less rarely than training, so heavier computation on metrics are possible without slowing down the training process.

2.4 Minimization

Minimization refers to reduction in the size of the FCN. This section is about the methods and tricks which make it possible to change the network during training, step back to an earlier state and compare states. This way determining the minimal size for the FCN can be automatized.

After each change in the network, including all of the methods described below, additional steps are necessary. The Adam [12] optimizer uses parameters that change over time. It has accumulators, and also a variable storing the step number, which represents the passing of time. The greater these variables, the less likely that the trainable variables receive big updates. When they are reset, we are giving each network configuration equal chances. We reset these parameters every time the network is altered or reverted to give the modified network a fresh start in training. These variables are not easily accessible, there are no helper methods provided. We must manually reach into the graph and reinitialize them, as seen in the following code sample.

fcn_train.py

```
...
if zero is not None or reinit is not None or freeze is not None:
    optimizer_vars = [
        var for var in tf.global_variables() if 'train' in var.name
    ]
    optimizer_vars.append(restore_variable('global_step:0'))
    optimizer_vars.append(restore_variable('beta1_power:0'))
    optimizer_vars.append(restore_variable('beta2_power:0'))
    sess.run(tf.variables_initializer(optimizer_vars))
...
def restore_variable(name):
    variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES)
    for e in variables:
        if e.name == name:
            return e
    return None
```

Neglecting neurons

To simulate a graph with fewer masks, or „neurons”, the masks can be disabled any time during training. Masks are contained in multidimensional matrices, one mask for each edge or input that the vertex or operation receives. Since the structure of the cannot be modified, we include switches into it, which tell exactly which neurons are disabled. The disabled masks always produce an output of zeros, whatever their input might be. Disabling masks provides no performance advantage, since they are still in the graph, taking up memory, but if we find that the results are still accurate with a few masks disabled, it means that the problem is solvable with fewer masks.

Freezing neurons

Freezing a mask means that the frozen mask will not receive updates via back-propagation, but will still produce an output. This is implemented similarly to disabled masks, provid-

ing switches that toggle between giving the mask zero-gradient, or retaining its value.

Reinitializing neurons

Reinitializing the masks instead of giving them zero values is also considered. The reason is that before the first run, the masks were given initial values based on normal distribution, instead of zeros, which should provide accelerated learning speed.

Stepping back

After a couple of training steps, we evaluate the accuracy, and save a snapshot of the graph to disk. This allows for resumed training later. This way we can compare thecurrent metric with earlier ones, and revert to them if we wish, for example when the accuracy had decreased since. Tensorflow provides functions to save and resume the graph structure, called meta-graph, and a checkpoint, which will store the current values of all free variables.

This is not hassle free. Tensorflow stores the absolute paths of the meta-graph and checkpoints into a plain text file name 'checkpoint'. In order to be able to move a saved model around, this file is altered by the training script to reflect the actual folders when resuming the network.

Another problem is that by default, tensorflow automatically saves every variable it can find, and constant values are incorporated into the graph. The graph cannot be touched after it was reloaded, so everything what needs changing later must be a variable. In order to prevent the saver from incorporating the input file list into te graph, and bloating it with thousands of file paths on each resume, the input pipeline must be set up separately from the rest of the graph, which needs to be feedable with the filenames. This way the saved model can be resumed and initialized with a new set of inputs, without storing them in the graph.

2.4.1 Testing the worth of a neuron

There are a few ways to utilize freezing and disabling to determine how much a single mask contributes to the overall accuracy, and see how the network performs without it.

One disabled One selected mask is disabled. This is useful to test the accuracy without that mask, and also observe what the optimizer will train this mask the next time.

One disabled and frozen The same as the above, plus the mask is frozen as well. This

would provide insight to how the rest of the network will change when this mask is removed, whether it will learn to take its role, or remain unchanged.

One disabled, others frozen The rest of the net is frozen, this will prevent the net from compensating the absence of the mask, but lets the mask relearn.

2.4.2 The minimization process

This process encapsulates multiple training sessions, and uses the tools and methods described above between them in hope of determining a minimal configuration.

First, a training session is invoked to produce a working network that is close to convergence and is capable of inference. This is longer than the later sessions, 10000 steps in most runs. We are ready to stop the training when error stagnates. The network is then evaluated and saved to disk. Then one or more masks are *selected* and altered. Altering includes freezing and disabling masks, and there are multiple strategies for *selection*, they are described below. After this, another training session and evaluation is performed with the altered network. This is shorter because the network is already semi-converged, 2000 steps exactly. The results of the new evaluation and the old one are compared, and if the new one is weaker, the old network is reloaded from disk. The strength of the new network compared to the old one is determined by a threshold over the ratio of their evaluation performance. Evaluation performance is measured by computing the mean F1 score over the evaluation dataset. Multiple thresholds were tested: 85%, 90%, 95%, 98% and 100%. Then comes another selection, which makes sure to select different masks if the network was restored. The sequence of training, evaluation, comparison and *selection* is repeated until the stopping criteria are met. We are left with the final network, and a one-hot matrix which

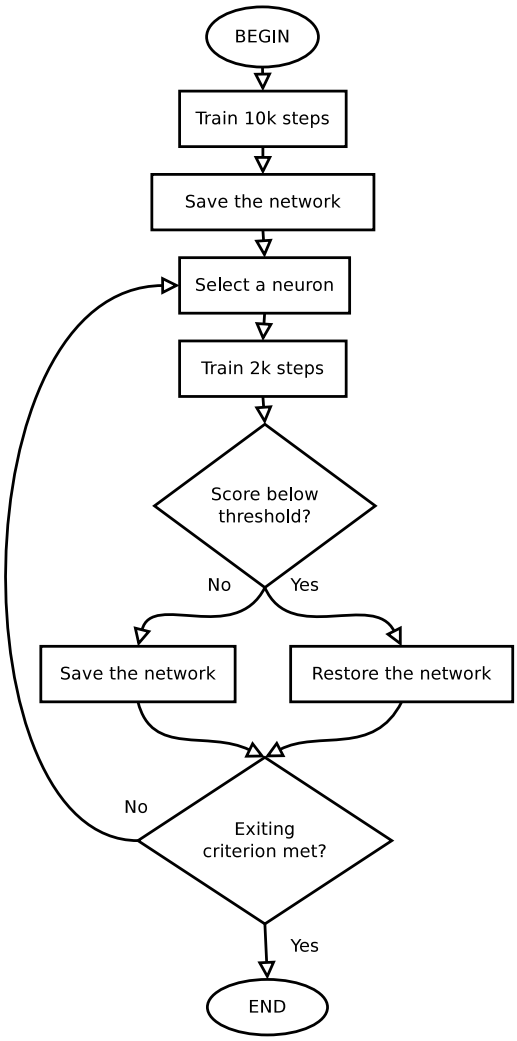


Figure 2.2

describes which masks ca be removed to get the minimal structure. This process is illustrated by the flowchart at 2.2.

2.4.3 Selection

Masks have to be selected, for freezing or disabling, the choice should reflect the contribution of the selected mask to the overall performance, or in other words, the least important mask should be the one that gets selected, so the network could possibly perform with one less mask.

Cosine-similarity

Cosine similarity tries to represent how similar two n-dimensional vectors are. A mask is a set of two-dimensional matrices plus scalar biases, so they are reshaped to a vector format, with the bias appended to the end of the vector. Then all the vectors are compared to each other using cosine similarity, and the end result is stored in a matrix, where the element at indices i and j represents the cosine similarity between the the mask in place i and the mask in place j. These values have a range of $[-1; 1]$, where 1 represent the greatest similarity (vectors facing the same direction), while negative numbers represent opposites (vectors facing the opposite direction). This is done for each layer, producing a matrix for each layer.

Using the matrices, selection could be done by selecting the mask which is most similar to the others, so it has the highest average similarity, or the one which has the maximal single similarity. These strategies hypothesize that a mask that is really similar to other masks is redundant, while unique masks play a key role in the inference. Absolute value could be used on the similarity values, since negative similarity could still mean similarity in the FCN.

An extension to this method is to use the second, or n-th maximal mask after the previous one failed and produced a significant decrease in performance.

Naive selection strategies

Sequential This method simply goes through every mask in a sequential order. This is mostly useful for comparison for smarter strategies.

Random Randomly picks a mask. Also useful for comparison, but it also could be a viable strategy.

Chapter 3

Metrics

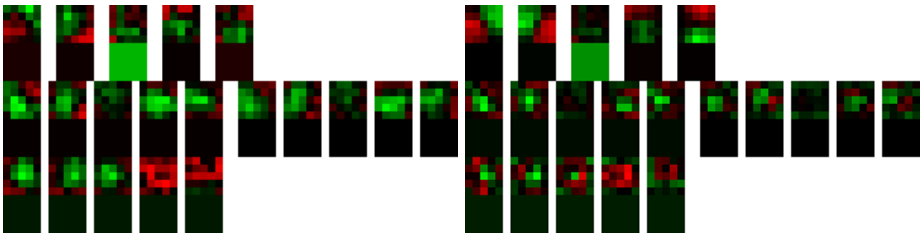
In addition to the objective function, or error, which is a metric itself the following metrics were used to judge the performance. These were used in the validation steps, or separately from training to benchmark the network.

3.1 Universal Image Quality Index

This metric is used to rank the similarity of two images. It is produced from three components, measuring the correlation, luminance difference and contrast difference between the two input images. It works by splitting the images into sections with a chosen size, and calculating the metric on those. We calculate an average of these to get a single value. When used alongside the simpler MAE and F1 score, this could provide an alternative metric and view on accuracy. A matlab implementation was provided by the authors, which was then converted to python so it can be integrated into the training process.

3.2 Visualization

Visual representation of the masks and snapshots of the inputs at each stage as they travel through the network are printed as image files to disk. This way the inner workings of the net are somewhat observable. Masks and images apart from the final output can contain negative values. To illustrate the whole range, negative values are visualized red, positive values are visualized green and color intensity represents absolute value. Examples of these visualizations are shown below. Note that some of the top masks resemble edge detector operators (see [6]), and the output of the very first mask resembles a diagonal edge map.



(a) Masks after a few steps... (b) ...and after convergence

Figure 3.1: Mask visualizations, rows on the picture correspond to layers in the network. The first row represents the masks in the first layer. Three groups of five masks from the second layer are omitted.

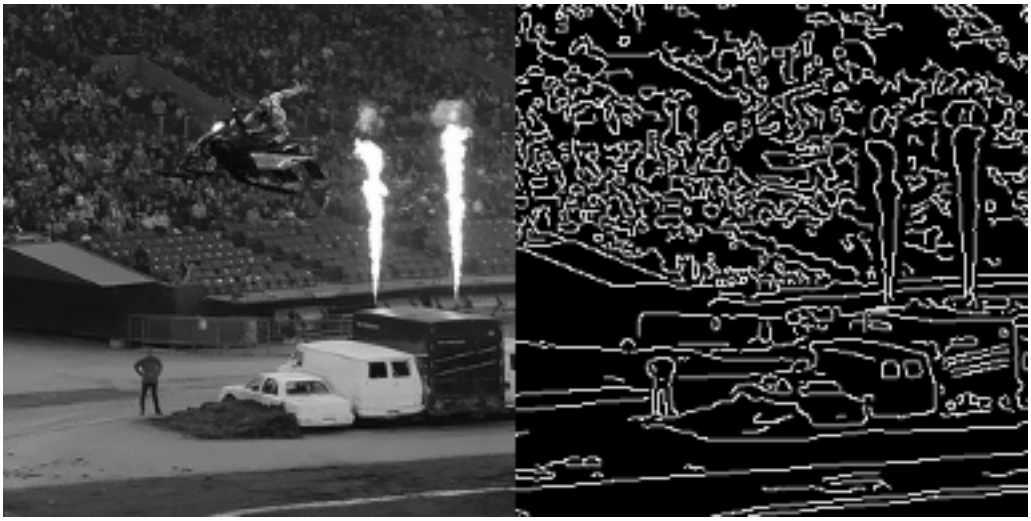


Figure 3.2: An input and ground truth image

Figure 3.3: Output after a few iterations and after convergence

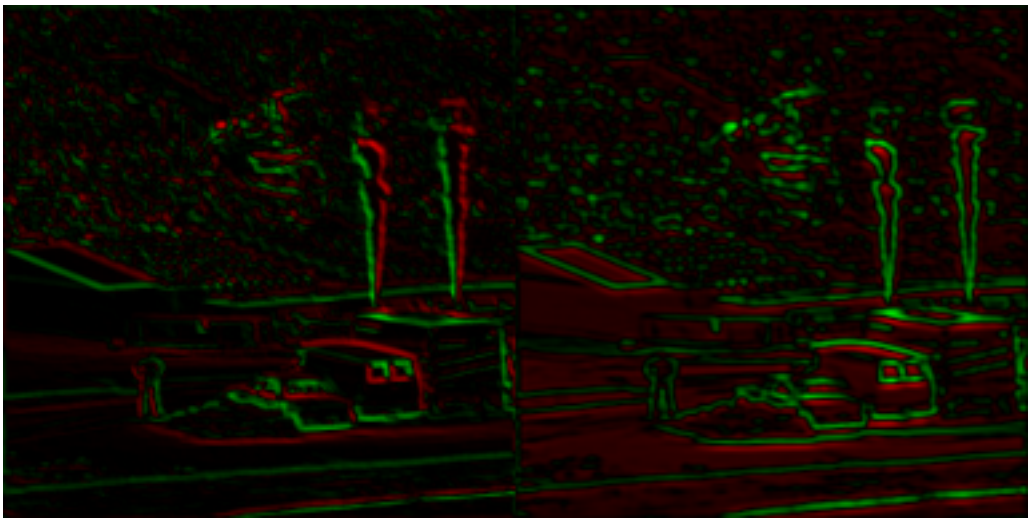


Figure 3.4: The output of the first mask in the first layer and the final output before activation, both after a few steps...

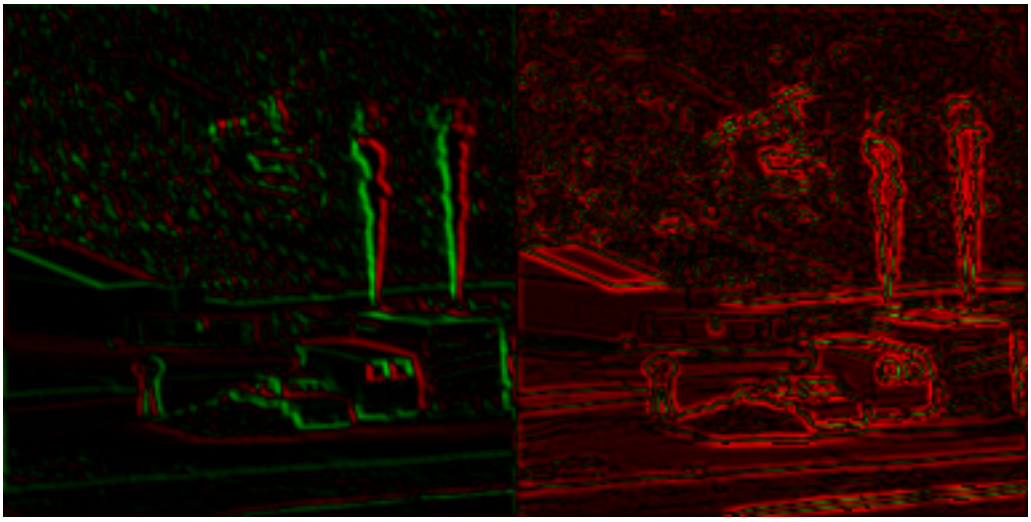


Figure 3.5: ...and the same after convergence

Chapter 4

Results

4.1 Run 1

freezing

4.2 Run 2

correlations

4.3 Run 3

final

Chapter 5

Conclusion

Chapter 6

Appendix

Statement

Alulírott ..... szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet ..... Tanszékén készítettem, ... diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, November 11, 2018 .....

aláírás

Acknowledgments

Ezúton szeretnék köszönetet mondani *X. Y-nak* ezért és ezért …

Bibliography

[1] Jacob Gildenblat. Pruning deep neural networks to make them fast and small. https://jacobgil.github.io/deeplearning/pruning-deep-learning, 2016.

[2] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient iference. https://arxiv.org/pdf/1611.06440.pdf, Nov 2016.

[3] Ari Morcos and David Barrett. Understanding deep learning through neuron deletion. https://deepmind.com/blog/understanding-deep-learning-through-neuron-deletion/, Mar 2018.

[4] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. http://tensorflow.org/, 2015. Software available from tensorflow.org.

[5] Convolutional neural network#Applications. https://en.wikipedia.org/wiki/Convolutional_neural_network#Applications.

[6] Sobel operator. https://en.wikipedia.org/wiki/Sobel_operator.

[7] Canny edge detector. https://en.wikipedia.org/wiki/Canny_edge_detector.

[8] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012). http://host.robots.ox.ac.uk/pascal/VOC/voc2012/.

[9] Netscope. http://ethereon.github.io/netscope/quickstart.html.

[10] Truncated normal distribution. https://en.wikipedia.org/wiki/Truncated_normal_distribution.

[11] Rectifier (neural networks). https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. https://arxiv.org/pdf/1412.6980.pdf, Dec 2014.

[13] Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[14] Backward pass of broadcasting on gpu is non-deterministic. https://github.com/tensorflow/tensorflow/issues/2652.

[15] Mention that gpu reductions are nondeterministic in docs. https://github.com/tensorflow/tensorflow/issues/2732.