Andrei Shikov

DEPTH ESTIMATION FROM A SINGLE IMAGE

Bachelor’s thesis
Information Technology

2017

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree	Time
Andrei Shikov	Bachelor of Information Technology	May 2017

Title

Depth estimation from a single image

50 pages

Commissioned by

Axmit

Supervisor

Matti Juutilainen

Abstract

The problem of depth estimation is an important component to understand the geometry of a scene and to navigate in space. More knowledge of the surroundings are bringing improvements in other areas, such as in recognition tasks as well. Many efficient algorithms already cover this problem using calibrated stereo cameras. When working with monocular images, a traditional algorithmic approach is experiencing many troubles, as one image can correspond to multiple geometries.

This study is considering neural networks as an algorithm of a depth estimation running on a web server in a cloud. The proposed solution defines concise, accurate and lagless results from a set of images, while executing at the server environment with the support of a GPU accelerator. The model achieves relatively high accuracy in identifying the depth, especially boundaries of closer objects, while keeping the network operating speeds on the satisfying level.

Keywords

depth estimation, robotics, machine learning, neural networks, TensorFlow, web server

CONTENTS

1 INTRODUCTION

According to Karpathy (2017), academic interest towards artificial intelligence and machine learning in general has sharply grown in the last two years. However, huge increase in research volume cannot be compared to the number of different neural network products used in real environments. Starting from the late 1990s, “artificially intelligent” models have been utilized in many different areas, mostly in recognition and classification fields.

A sharp growth in publications is efficiently supported with development tools. After the introduction of the Theano library by James Bergstra et al. in 2010, the representation of neural networks has become much easier and less tricky to deal with. The research in the field has overcome high entry barrier and nowadays is much closer to the people without robust math knowledge. Many scientists have received tools to back their interest and have forgotten about implementation bottlenecks.

These facts are bringing closer the future where machine learning and neural networks can be considered as desirable components of almost every piece of software written. People are training networks to write music, recognize authors by piece of text, drive a car or even to write other networks. Although the machine learning can bring an essential improvement to many areas, robotics have much higher benefit from this grow.

For a robot, the problem of depth estimation is an important component to understand the geometry of a scene and to navigate in space. Furthermore, more knowledge of the surroundings are bringing improvements in other areas, such as in recognition tasks. Many efficient algorithms already cover this problem using calibrated stereo cameras. When working with monocular images, an algorithmic approach is experiencing many troubles, as one image can correspond to multiple geometries. Most of them are not plausible in the physical world. Therefore, depth can be predicted from meaningful ones. Human brain is easily dealing with the problem using similar methods while producing reliable representation of a scene from a monocular picture.

Processing speed is vital to produce reliable data in nearly realtime. In robotics hardware limits computing capabilities, to improve battery life and portability as well as protection. To overcome these limitations, some companies design their own control blocks (NVIDIA Corp. 2016). However, another approach is to transfer some computations to the cloud clusters, where processing speed is only limited by connection speed which is constantly improving nowadays.

This study is considering neural networks as an algorithm of depth estimation running on a web server in a cloud. The solution should define concise, accurate and lagless results from a set of images, while executing at the server environment with the support of a GPU accelerator. Therefore, the goal is to achieve high accuracy in identifying the depth, especially boundaries of closer objects, while keeping the network operating speeds on the satisfying level. It could be achieved by choosing a network architecture wisely with an attention to the details of the problem (recognition/classification type) as well as to limitations of an executing environment.

In Chapter 2 a short overview of the neural networks structure and methods of their training is given. After that, in Chapter 3, recent works in the field of depth estimation are discussed. Chapter 4 discusses the networks in practice. Finally, Chapters 5 and 6 include a definition of the production network deployment.

2 NEURAL NETWORK CONCEPTS

Neural networks(NN) and especially deep networks were initially inspired by the structure of the human neural system. It consists of many simple parts(=neurons), which line inputs(=dendrites) and output(=axon). A single neuron is working as a simple function, using chemical reactions to activate on certain signal strength. However, the whole network is smart enough to make complicated decisions and to recognize tricky figures. (Caroline Clabaugh et al. 2000)

Figure 1 shows an artificial neuron featuring a structure similar to biological one. It takes several inputs and produces an output which, in turn, is connected to other neurons. The inputs are processed with inner weights by a predefined activation function, allowing neuron optimization during training.

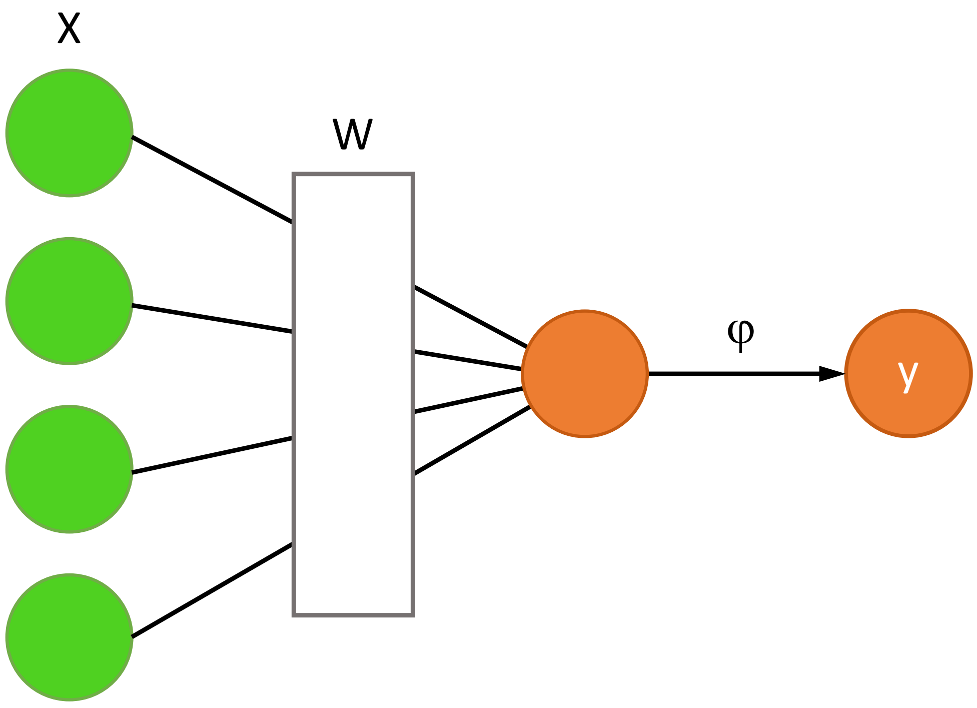

Figure 1. Simple artificial neuron with inputs x, weights w, output y and activation function φ.

Different combinations of neurons allow creating models for many applications. Commonly, the structure is built from predefined modules with already predefined area of usage. The module combinations produce accurate models when applied in a correct place.

2.1 Model structure

Figure 2 shows a sample model, where inputs *x* are used with weights *w* and biases *b*. The simplest linear output is formed by an expression:

$$y = x * w + b$$

Therefore, one receives the linear equation where he can tweak the desired outputs using weights and biases. Such a model is very efficient to compute and to train right values. However, this model is limited only to *(|w| + 1)* * |*b*| parameters and is not designed to reproduce complex behavior. (Coolen 2004)

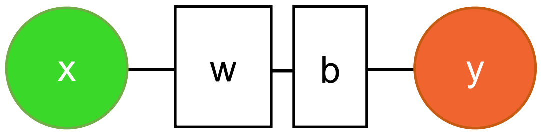

Figure 2. Simple linear neuron

In the previous example the identity function is used as the activation. However, when stacking such layers together, we will receive the network which is equivalent to one layered with a large number of parameters. To overcome such difficulties, non-linear activation functions were introduced. Inserting a non-linear activation between layers allows simulating much more complicated relations between input and output while preventing the network from growing in the number of parameters or in computational intensity.

Desirable properties of an activation function are as follows (Snyman 2005):

- It should introduce non-linearity to the input.
- It should be continuously differentiable to match the behavior of back propagation, which will be discussed below.
- It should be infinite to match as many of the weights values as possible.
- It should approximate identity near the origin to produce efficient results when they are initialized from a small-sized random set.

However, these characteristics are mostly optional and many successfully implemented networks are using functions which somehow are totally incompatible with some of the points in the list above. The activation functions

properties are varying in terms of the value space and the purpose. Below several most popular functions are listed with their definitions.
The simplest function utilized in the classification is a binary function. It is expressing natural way of a binary division, returning zero if the input value is less than zero and one otherwise, or in the equation form:

$$f(x) = \begin{cases} 1, & if\ x \geq 0 \\ 0, & otherwise \end{cases}$$

The sigmoid function is similar by its properties to the binary one, but has better approximation near zero (Han et al. 1995, pp. 195-201). It can be formed by the expression:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The rectifier linear unit (ReLU) and exponential linear unit (ELU) are more recent players in the field of neural networks. They have shown themselves as more efficient when building image recognition and classifier models (Glorot et al. 2010). ReLU is expressed with the following equation:

$$f(x) = \begin{cases} x, & if\ x \geq 0 \\ 0, & otherwise \end{cases}$$

ELU is more smooth and is computed as follows:

$$f(x) = \begin{cases} x, & x \geq 0 \\ a(e^x - 1), & x < 0, \end{cases}$$

where a is a hyper parameter which can be tuned.

2.2 Common network structures

As discussed above, modules of NNs have predefined targets which are noticeable by their structure. If we will discuss the structure inside, it is created by stacking simple operators, like multiplication or addition. These simple operations give a brief representation in math sense which will be described in Chapter 2.2. The section below gives a more abstract definition of modules based on article of Fjodor Van Veen (2016), moving away from detailed explanations in terms of computing.

Perceptron (Figure 3) is the simplest structure. It feeds information from input layers (yellow) to output layers (orange) while processing it with hidden layers (green). Theoretically, given that the network of this type has enough hidden neurons, it can model a relationship between given input and output. Practically, however, it is computationally ineffective, therefore the common use case is simple classification tasks or a combination with other modules.

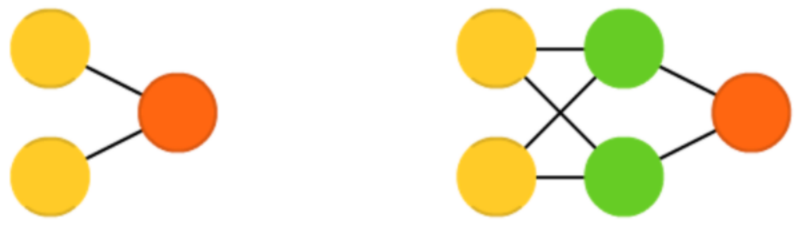

Figure 3. Fully connected (dense) NN structure. The left picture shows the least possible network.

Convolutional networks (Figure 4) feature different approach of processing inputs. Instead of processing each one separately and training weights separately, convolution networks are using filters with an approach similar to Gaussian blur, but are trained to “learn” how to transform pixels themselves. Applying these filters in a sliding window approach, the network is converting an image to a space of the filter which depends on the filter amount and the step of the window slide (stride) (Figure 5). Therefore, weights are common for all inputs, reducing memory overhead while training.

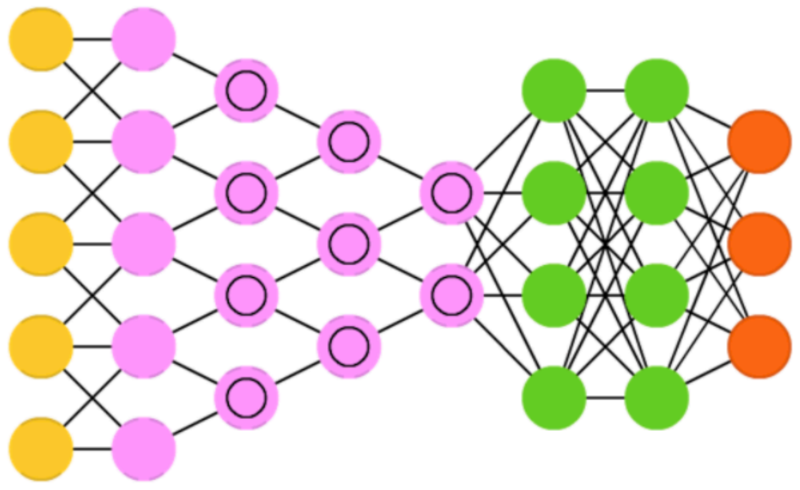

Figure 4. Deep convolutional NN (CNN or DCNN). Convolutional layers (purple) are connected to feed forward hiddel layers (green). Convolutions ensure feature learning while feed forward layers are classificators.

The main strength of this structure is feature extraction. These networks are heavily used in the image recognition field, although other fields of application, as audio analysis, are considered as well.

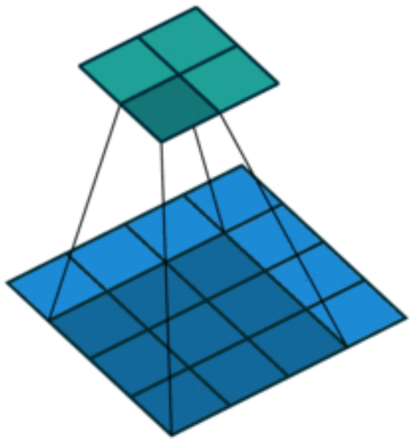

Figure 5. An example of a 3x3 convolution (Dumoulin 2017).

Convolutions are usually combined with pooling which is a method to decrease the amount of information in an image without losing any data. This is achieved by the same concept of sliding windows where the maximum or average in the window is taken to the next layer. In addition to a simple convolution, researchers are utilizing “deconvolution” or transpose convolution to restore features from filter fields. This reconstruction technique allows extracting semantic, stylistic and many other features from the image.

Autoencoders (Figure 6) are similar to fully connected networks: they utilize the same structure (input – hidden layer – output) and use the same training method. However, the idea behind these networks is totally different. They are trained to create data representation in smaller volume of resources, which one can recognize as compression.

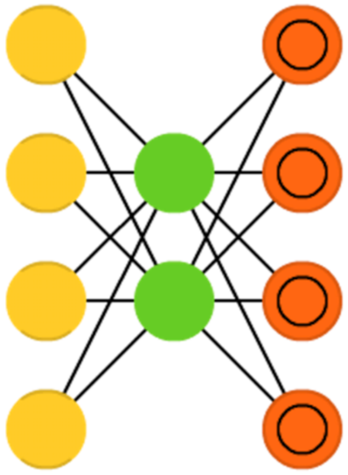

Figure 6. Auto encoder network (AE)

The network is creating an encoder to the middle layer and a decoder to the end, therefore the input should be the same as output. A symmetric structure is usual for this type of networks. Moreover, this structure has recently been taken into use to train convolutional AE networks for image segmentation, labeling and other per pixel computations.

Figure 7 illustrates a recurrent neural network which are inspired by flip-flops in electronics. It is not stateless, as other networks, introducing an ability to produce future output based on a current state. RNN and its currently most used subtype, long-short memory network, are particularly good in learning sequences including tasks from classification to reproducing features. They are widely used as the generators of text or music by indie developers.

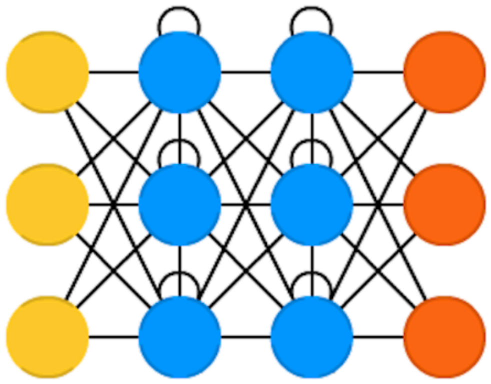

Figure 7. Recurrent neural network (RNN)

Despite the fact that SVMs are not always considered and implemented as neural networks, they are usually designed as feed forward ones with special algorithmic tweaks in the activation function and training approach. The main feature of SVM is an ability of fast classification of linearly separable data. Therefore, a common field of application is binary classification.

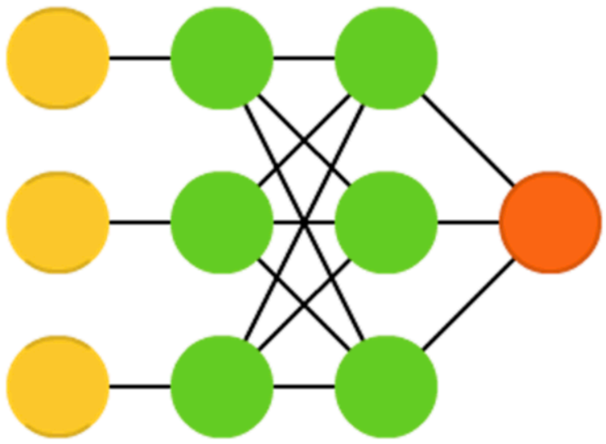

Figure 8. Support Vector Machine (SVM)

2.3 Training

Training (or learning) is usually divided into two types: supervised and unsupervised. The difference is that in unsupervised learning the model doesn’t receive the ground truth (desired outputs), whereas in supervised learning the final output is compared to the one from a dataset. In the modern artificial intelligence world most networks are trained with the supervised approach. This allows easing connection establishment between input and desired output.

Automatic encoders can be presented as a more practical example for unsupervised learning. The model is learning the internal representation itself while creating the ability to restore original data. One can notice that even though the target is internal state, training actually applies the same techniques as the supervised way.

The sections below describe the common training approach for supervised learning. Even though there are many ways of training the network, for deep learning network with a huge number of parameters we can distinguish only limited amount of practically efficient ones. For the purposes set in the beginning, we can focus only on the gradient descent algorithm and its optimizations.

2.3.1 Gradient descent

To optimize a neural network with supervised training, a simple technique is implemented. The dataset with some inputs and outputs is fed into the network which is producing some outputs. Afterwards, the error on the dataset is computed and then, based on this, update values for the weights are computed. After several steps of this algorithm, the weights are converging, minimizing the error (Figure 9).

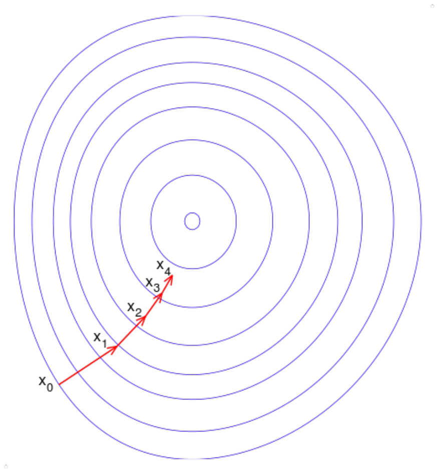

Figure 9. Gradient descent concept illustration. It is converging on error function after each iteration, aiming at the minimum (Wikipedia 2017).

A loss(error) function is usually defined in terms of probability model or energy model (LeCun et al. 2006). Firstly, the error was calculated simply by checking the probabilities of the reproduced results comparing to the ground truth. However, this estimation of error is considered as unreliable in many cases and is mostly replaced by loss computations which are measuring an error computing difference between the produced and desired outputs. As a loss function scientists usually use mean *squares* and *negative log likelihood (cross entropy),* although there are other examples, like *scale invariant error* in the research of David Eigen et al. in 2014.

To minimize an error, one needs to compute the gradient, or the change in the weights describing the direction in which the algorithm should move. Therefore, there is a need to compute the current “slope” of the function which is easily implemented by taking a derivative of the function (Snyman 2005).

Therefore, to compute the change of the weight in the linear example from Figure 8, an error should be computed. Recalling that with a given array of inputs *x*, outputs *y*, weights *w* and biases *b*, the output of the fully connected network can be considered as follows:

$$f(x) = x * w + b$$

Then the square error for a single element will be computed this way:

$$E^i = \left(y^i - f(x)^i\right)^2$$

The derivative of the defined function *f(x)* is equal to just *w*, and the weights can be updated using by this equation with a learning rate δ:

$$\Delta w = -\delta \frac{\partial E}{\partial w} = -2\delta \sum_{i=0}^{n-1} \left(y^i - f(x)^i\right) w^i$$

The learning rate adjusts a gradient step to create a better fit and to prevent the divergence of the steps. However, a small rate can result in slow convergence. Gradient descent is running for a limited number of iterations, considered as training epochs, usually found by a scientist with the method of trial and error.

2.3.2 Computing gradient in deep networks

A deep network is the one that consists of several “layers”, usually defined by separate functions with activations. To compute a derivative of this sequence, we can use a chain rule which is a key concept in modern deep learning. To update (or “backpropagate”) layers after feed forward computations, a gradient for each node need to be computed. The chain rule states as follows:

$$\left[g\left(f(x)\right)\right]' = g'\left(f(x)\right) * f'(x)$$

Therefore, if the result of previous function in the chain is known after feed forward and its derivative can be easily computed, the weights could be updated in the wave manner from back to front. This concept is known as *backpropagation*.

To train a network using supervised learning algorithms the dataset with desired inputs and outputs should be available. This, given the fact that the model is created consistently with its purpose, allows creating a connection from the input to the output. However, to achieve a good generalization, learning should prevent an overfitting of the model. Overfitting is a concept where the model is performing well during the tests (in the test data set), but in the real environment it shows poor results. There are many techniques to prevent overfitting including regularization, dataset split and dropout.

Regularization is a process of including additional information to a loss function. Usually it is achieved by an additional input signal scaled by parameter λ. In the form of equation the general regularization can be shown as follows:

$$loss = L(f) + \lambda * R(f)$$

Here, *L(f)* is the initial loss function, while *R(f)* is the regularization. Both are taking the original function *f* as an input.

Dropout is applied to the hidden layers in the model. It is erasing some fraction of the output (controlled by a parameter) by changing the their outputs to a zero. This simple and computationally cheap technique brings a better generalization, allowing the network to find several connections between the input and output. However, the main restriction of this method is its field of application. Dropout was invented for deep networks and performs best when is applied to them.

Dataset split is a common way to utilize the input data for the training. It is usually applied along with other techniques, but is essential by itself to achieve a good generalization. The method consists of a simple split of data to several parts: training, validation and test sets. The test set is often omitted, as it is not required during the training and is utilized when a model performance evaluation is required. The training set is used in the learning process and the validation set is applied to manage model performance during this phase. However, the final performance should be evaluated on the data that the network has never seen before. It shows model performance and overall generalization.

A deep neural network usually requires a lot of different data to create a solid relation between the input and output. Moreover, to prevent a bias towards training set, the data should differ from piece to piece, so that the model could be trained on different features and could generalize better. However, a large dataset raises another problem of the memory consumption. Nowadays datasets can take many hundreds of gigabytes in compressed mode and random access memory is not targeted to keep up with a data of this size. Furthermore, neural networks are usually trained on a GPU to accelerate mathematical operations, and therefore, data should be copied to the videocard which has limited capacities as well. To overcome this kind of difficulties, the dataset is divided into smaller batches which include several pairs of input - output. Moreover, for this problem an algorithm of gradient descent is modified to compute an error per batch and to optimize based on these computations. This approach is called Stochastic Gradient Descent (SGD). Moreover, this algorithm has received a lot of modifications recently, including Adaptive Gradient (AdaGrad) and Adaptive Moments (Adam) where an error is computed based on a moments of a gradient (Kingma & Lei Ba. 2017).

3 DEPTH ESTIMATION WITH NEURAL NETWORKS

To be able to estimate the depth from an image, a neural network should understand the geometric properties of a scene. This result is easily achievable by introduction of stereo camera and calibrating it. Two perspectives of one scene give a lot of information about how objects are located in a three dimensional space. Using the computer vision basics, the difference between locations of the objects on two images (disparity) can be used to calculate a third dimension value. From Figure 10 the following equation is received:

$$disparity = x - x' = \frac{Bf}{Z},$$

where *x* and *x′* are the distance between the points on the image space, *B* is the baseline or the distance between the cameras, *f* is the focal strength of the camera, which we calibrated (OpenCV 2017). Applying the received information, a depth map of an image is derived which means the calculation of objects' coordinates on the Z axis.

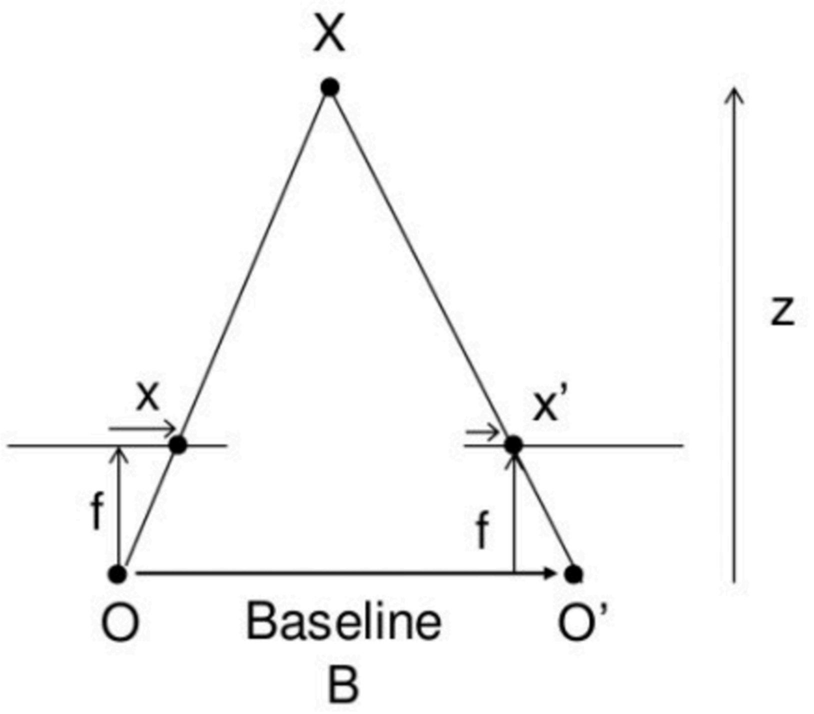

Figure 10. Scheme of the object and stereo cameras from bird-eye perspective (OpenCV 2017).

The code of implementing this in the OpenCV framework is fairly simple. Code 1 illustrates the given approach on Python.

```
import cv2

# With given left and right images
imgL, imgR
stereo = cv2.StereoBM_create(numDisparities=16, blockSize=15)
# Calculate disparity map
disparity = stereo.compute(imgL, imgR)
```

Code 1. Computing disparities from stereo images in OpenCV.

However, this approach is not suitable when only a single image is given. In that case there is no simple and reliable way to interpret geometry of a scene, as there is no known perspective change. Therefore, even given a sequence of frames, one cannot rely on changes between them, as the actual change in position is unknown or can be smaller or larger than supported boundaries.

Neural networks were widely applied in the field of depth estimation because of the limitations above. In more common sense, the models are trying to recognize the boundaries of the objects and their position in the scene, trying to achieve the closest possible estimation to reality. Especially, convolutional neural networks (CNN) and markov random fields (MRF) are heavily utilized by many researchers in image processing. Despite the fact that CNNs were mostly used in the classification, their utilization in more compex operations, like semantic labeling or depth estimation, has grown recently. The next sections discuss some examples of the models utilizing stereo and monocular images.

3.1 Depth estimation from stereo images

Alongside with the naïve computer vision approach illustrated above, neural networks are widely used to address more advanced topics using binocular images. In the sections below several ways of utilizing CNNs are discussed targeting topics adjacent to the topic of depth estimation.

3.1.1 FlowNet

Fischer et al. (2015) discussed the optical flow estimation from a pair of binocular images. Their main idea was to exploit the power of convolutional neural networks to learn features of the image in multiple levels of scale and abstraction and to train a flow prediction based on these features. They have created an architecture similar to Long et al. (2015) and Dosovitskiy et al. (2015), refining coarse levels with deconvolution (transposed convolution). The study introduced a piece of novelty utilizing not just results from the layers, but whole feature maps, transferring more information. The result of their study was named FlowNet.

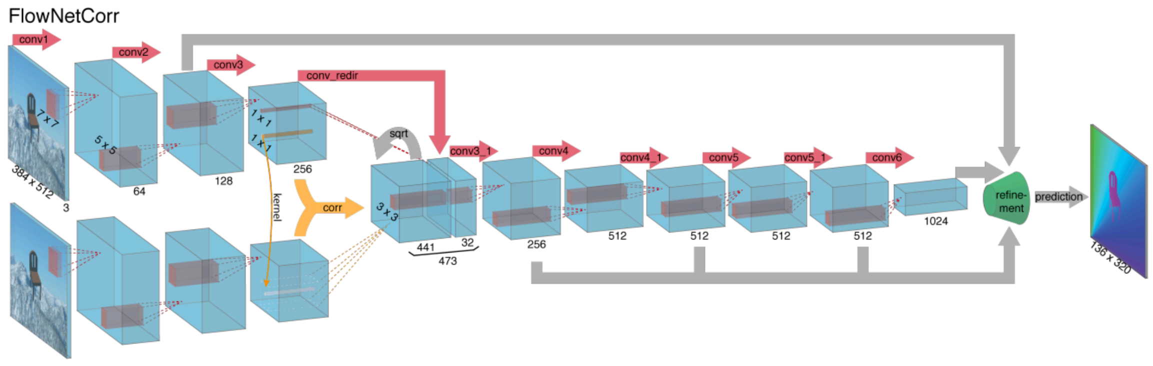

Figure 11. FlowNet arcitecture with correction using second image from binocular pair (Fischer et al. 2015).

Figure 11 shows the architecture uses multiple feed forward connections to the refinement layer to share information of earlier coarse layers. Moreover, the introduction of the correlation layer gives the network an ability to “compare” images and build a shared kernel among a binocular pair of images. Thus, the network is sharing all the information it computes and the suffering from gradient vanishing (information loss in deeper layers) is minimized.

Refinement consists of several deconvolution steps where each upsampling is associated with the connection from coarse network. Moreover, starting from the second refinement, lower resolution results are produced and combined with inputs for the following layers. With this simple trick the network receives the data from the previous layers as well as results of utilizing this data. The researchers have found that more than four consecutive connections are redundant and do not improve the output while being computationally intensive. Therefore, after four repeats the image was upscaled to the match the desired resolution.

Furthermore, the research has shown that the convolutional network of this size requires a large dataset when using supervised learning approach where the input and ground truth is given. However, existing datasets in the field of optical flow were too small or inconsistent. This problem was solved by generating their own images from flickr data on the background and random chair models on the foreground. The dataset was called Flying Chairs and using it in learning produced much more consistent results than real world data from the KITTI dataset (Geiger et al. 2013).

3.1.2 MC-CNN

Zbontar and LeCun (2016) have proposed a faster and simpler approach to estimate image disparity, implementing a simpler model and utilizing supervised learning based on stereo images and ground truth from the KITTI dataset (Geiger et al. 2013). They have introduced two architectures of siamese network: a fast one, which is less accurate and less computationally expensive, and a more complex one.

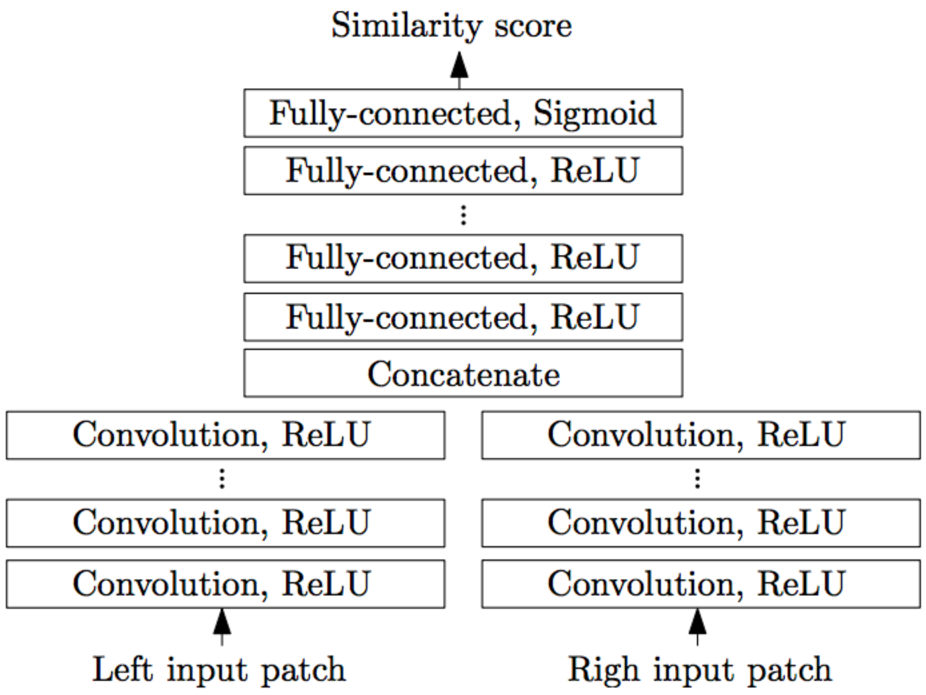

Figure 12. MC-CNN (Zbontar & Lecun, 2016). Proposed architecture. The faster approach can be derived by replacing fully connected layers with a dot product and appending normalization to convolutions.

Figure 12 illustrates an example of an network architecture. The result is a similarity score which is used to initialize a matching cost functiion. As the figure shows, the image is split into patches and then def to the network, so it does not consume the whole image at once. Using a similarity score *s*, the matching cost is derived as:

$$C(p,d) = -s(< patch^L(p), patch^R(p-d) >)$$

where $s(< patch^L(p), patch^R(p-d) >)$ is a similarity score when a patch *p* is fed to the left input and a patch *p* – *d* is fed to the right input.

The authors also mentioned the importance of speed and computation resources management, as their approach allows producing
inputs for the convolutional part in just a single feed forward pass. Moreover, replacing fully connected layers with 1x1 convolution patches allows computing them in a single forward pass as well.

The disparity is predicted by taking the value *d* which minimizes matching cost *C(p, d).* When the disparities for the left and right images are found, they are interpolated to resolve possible conflicts and refined using a bilateral filter.

3.2 Depth estimation from monocular images

The operation of extracting scene information from a single image is a much more complicated issue compared to the stereo environments where we can easily estimate the scene geometry. The methods used for such an estimation are much more complex and much less reliable. Results are very dependent on the environment and training data, and therefore the model should be flexible enough to generalize on many conditions.

However, there are several works worth mentioning that are covering this area and have implemented successful solutions. Therefore, two key architectures are discussed below in more descriptive manner.

3.2.1 Multi-Scale Deep Network

In 2014, Eigen et al. proposed a multi-scaled convolutional neural network to produce a depth map of the image. The main feature of that network was an ability to find common details and then fine-tune contours. This optimization allows utilizing global prediction from coarse scale networks to predict finer details of the image (Figure 13).

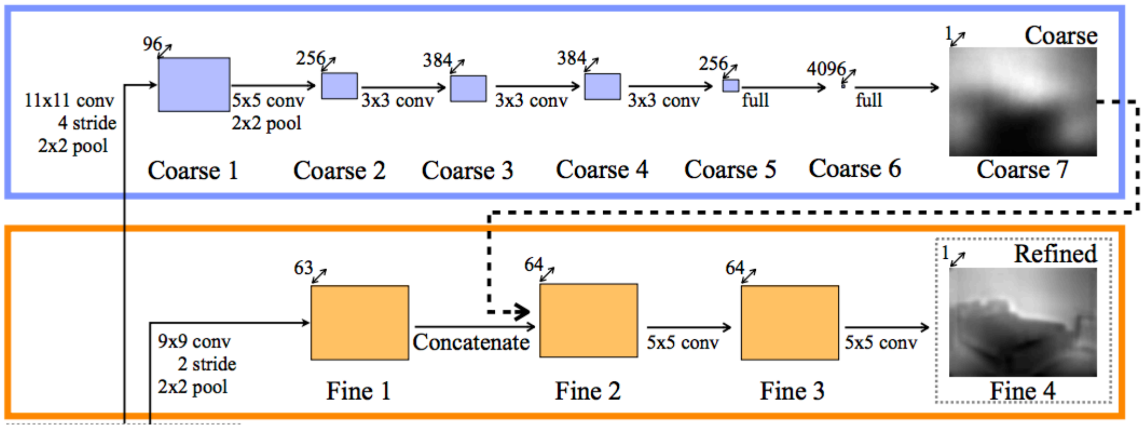

Figure 13. Multi-Scale Deep Network architecture. (Eigen et al., 2014). Blue border contains coarse layers of the network, orange ones described fine scale ones.

The coarse network consists of several convolution layers finishing with a fully-connected layer. This architecture brings feature recognition finished by a classifier and network self-learned upscaler. After the coarse network is pretrained, the refining part, which consists of several convolutions, is applied to the same input. Prediction from coarse layers concatenated with the inputs from of the second fine-scale layers and the model is relying on this while producing a result.

As can be observed, two network parts are mostly independent of each other. As a consequence, the best measurements are produced when the coarse layers are first trained given the input and ground truth and then the whole network is retrained while freezing the coefficients on pretrained layers.

In addition to the network split, Eigen et al. (2014) were using non-standard loss function. As described above, for convolutional networks one usually utilizes negative log-likelihood (cross-entropy) or mean square error. Nevertheless, these functions suit well for classification tasks, while depth prediction is very different and requires a definition of its own loss which can better classify the degree error of the network error. In the study, *scale invariant mean square error* (in log space) was used as a loss function. It can be defined as follows:

$$E = \frac{1}{2n}\sum_{i=0}^{n-1}\left(\log y^i - \log f(x)^i \ + \ \alpha(y, f(x))\right)^2$$

where y is a ground truth, $f(x)$ is a result of the NN and $\alpha(y, f(x))$ equals

$$\frac{1}{n}\sum(\log f(x)^i - \log y^i).$$

According to the authors, this loss function allows the model to converge better, resulting in a relative improvement up to 20%.

3.2.2 Deep Convolutional Neural Fields

Liu et al. in their study in 2015 made an assumption that an image consists of small regions (superpixels) of homogeneous pixels, and were considering a model of nodes defined on top of superpixels. The name of the result is combined from deep convolutional neural networks and conditional random fields, two techniques which were used in the model. It is worth mentioning, that their idea can work with usual pixels, although it usually is computationally ineffective. Their idea was similar to the one described above, dividing the network into coarse and fine tune parts. However, while using very similar architecture for rough approximation, their refinement had completely different idea.

First, an image is split into clusters which represent superpixels. Then, for each superpixel the network computes: unary (coarse) part where a patch of an image centered around a centroid of the superpixel goes through the convolution and depth computation; pairwise potential computation of neighbouring pixels enforcing smoothness on the final depth image.

Therefore, the unary part is a regression which computes the depth map for the center of each centroid. However, to apply a fine-tune matrix R which keeps the values for a pair of superpixels p and q, we need to apply the following equation.

$$f(x) = A^{-1}z$$

where z is an output of the coarse network and A is computed this way:

$$A = I + D - R$$

where I is an identity matrix and D is a diagonal matrix where $D_{pp} = \sum_p R_{pq}$.

3.3 Chosen approach

The chosen approach is based on the work of Godard et al. (2017). Their idea is completely different from the models described above, as it does not require ground truth during training, which is rather compex to acquire, as depth cameras introduce many limitations in real environments and require heavy postprocessing. As the right image can be derived from the left in a stereo pair, we can train the network to produce disparities by feeding a pair of calibrated binocular images. After receiving a disparity map, we can produce the depth given formulas from the beginning of Chapter 3.

The network implements autoencoder architecture with convolutions and upconvolutions (upsampling with convolution) or an encoder and decoder, where the information from the first layers is shared with the later ones to prevent information loss and being higher resolution features. (Figure 14). This architecture was inspired by DispNet, successor of FlowNet described in Section 3.1.1. The outputs are defined for four different scales differ by a multiplier of 2. The training of a network is based on the right and left pictures, trying to produce left from right and vice versa and to be consistent in both predictions.

The loss is defined for each of the output disparities, summing them to the final loss *E.* The loss for each scale is a combination of three main terms: similarity of the produced image to the training input, smoothness of disparity and consistency of left and right disparities. The functions below are provided for the L (left) and R (right), where an image could be the original of its stereo pair reconstruction depending on the output the error computed for. For appearance matching SSIM algorithm is implemented, calculating structural dissimilarity for *L* and *R* as follows:

$$E_{ssim} = 1 - \frac{SSIM(L, R)}{2}$$

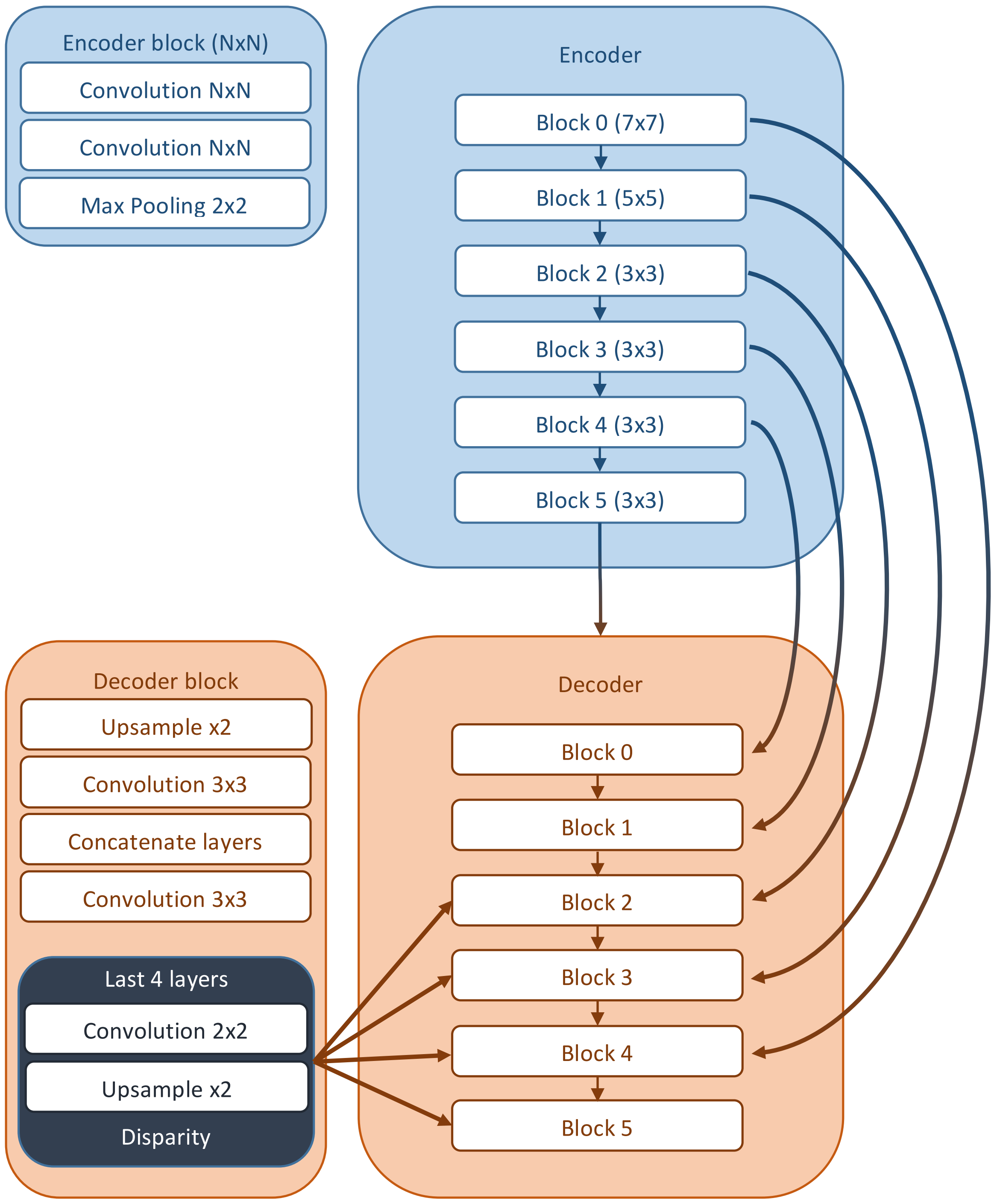

Figure 14. Architecture of a network suggested by Godard et al (2017). Encoder and decoder are built from similar blocks which are shown on the left. Layers from encoder are fed to the concatenation of layers in decoder. Disparities from the last four layers of decoder are fed to the inputs of the following layers similar to FlowNet (Section 3.1.1)

To simplify the algorithm during computations, fast Fourier transform convolution with Gaussian in SSIM was replaced by 3x3 average pooling. Furthermore, *l1* loss from the left and right image is added and both are weighted, resulting in the following equation:

$$E_{appearance} = \frac{1}{n}\sum_{i,j} \alpha(\frac{1 - SSIM\ (L, R)}{2}) + (1 - \alpha)|L - R|$$

The left-right consistency in disparities is checked through projecting the right map and ensuring that the left output is equal to the right. Disparity smoothness is calculated by translating the result to one pixel in the x and y direction and checking the values of the gradient with weighting with an edge-aware term.

During training, left and right outputs are produced, although in the test workflow only left disparity is used. The outputs are fixed from 0 to the maximum distance equal to $0.3 * d_{max}$, where d_{max} is a width of the image. The effect is achieved using a *sigmoid* activation for the output. After computing disparities, using known baseline and focal distance, we can evaluate a depth map. However, this study targets mostly relative distance, so the network outputs contain enough information.

4 TRAINING

The project implementation consists of two parts: training a network and linking a model to a web server. Both of them require the setup of a server with a GPU support to produce the results fast enough. The implementation of a network is based on TensorFlow framework, and therefore, this chapter discusses the definition of a network and a training approach, whereas chapters 5 and 6 introduces topics connected to the web server setup.

4.1 What is TensorFlow

According to TensorFlow official website (Google 2017), it is an open source library based on the design Google Brain Team. Its purpose is mathematical computations using a graph model. In this graph, nodes are operations and edges are multidimentional arrays (tensors). Therefore, a model is usually described as input tensors and some operations on it. This flexible structure allows a definition of complex operations and their optimization on the fly. Furthermore, the framework is able to scale to multiple devices (CPUs, GPUs or even several machines) without changing the code. A vizualization is supported as well with introduction of supporting tools, such as TensorBoard. The framework is relatively new, existing for only one year, but it is intensively used by scientists to apply complex computations during research.

Code 2 shows definition of the simplest fully connected model discussed in Chapter 2.1 and illustrated in Figure 3 (left image).

```
# Here and after the framework
# is considered as imported by an alias
# The values from this snippet are utilized in other examples as well
import tensorflow as tf

# Define the input
x = tf.placeholder(tf.float32, (None, 2))
# Define the weights
w = tf.Variable(tf.truncated_normal((2, 1)))
# Define the bias
b = tf.Variable(tf.zeros(1))
# Define the output
y = tf.nn.sigmoid(tf.add(tf.matmul(x, w), b), , name="output_node")
```

Code 2. Definition of a simple model in the framework.

An input *x* is defined as a 32-bit integer with the shape of `(None, 2)`. `None` means that any size can be set for the dimension. Therefore, for the input a two-dimensional array of any length and size of the second dimension of 2 is accepted. Trainable tensor values (weights and biases) should be wrapped with the `tf.Variable` instruction to allow an optimizer to change those values.

To define the weights, a random initializer is used. The truncated normal function samples values from a Gaussian distribution, and repicks the values when they are highly deviating. This allows creating a sample where the results are random, but deviation among them is low, which is good for the setup of the neural network weights. The shape of the variable is `(2, 1)` and is defining weights for the connection of two inputs to a single output. The bias is a vector of zeros with the length of 1, finishing the variable setup.

The operation `y` is defined following the formula from Chapter 2.1. Afterwards, it is activated using a *sigmoid function* to introduce a non-linearity. After executing the code, the graph of computations is defined and hold in the memory. To execute a graph, a session is required. A session allocates resources and holds the actual values of the computations. Code 3 illustrates how the operations can be executed.

```
# Execute session and close it automatically
with tf.Session() as s:
    # Initialize global variables
    s.run(tf.global_variables_initializer())
    # Calculate the result of operation
    output = s.run(y, feed_dict={x: data})
```

Code 3. Computing the tensor value.

The session requires initialization of the variables defined above to perform computations. Afterwards, the operation can be executed using the `run` method. Since a placeholder was defined for the input, the data should be fed into the graph using the `feed_dict` parameter. After an execution of the method, the desired output is received and written to the `output` variable available for a further processing.

4.2 Definition of more complex constructions

In practice, models and operations used in the network are much more complex than simple additions and multiplications. This includes RNN cells, convolutions, pooling and many others. Fortunately, the framework defines functions to describe all these constructions and optimizing them. This allows scientists to do the research in the field without concentrating on the code and boilerplate creation. Simple 3x3 convolution with the stride of two and the output depth of six can be defined as in Code 4.

```
# Define input tensor (for example as a grayscale image with 1 channel)
image
# Define weights variable for convolution filter with desired shape
w = tf.Variable(tf.truncated_normal(shape=(3, 3, 1, 6)))
# Define bias variable
b = tf.Variable(tf.zeros(6))
# Define convolution
conv = tf.nn.conv2d(image, w, strides=[1, 2, 2, 1]) + b
# Activation.
conv = tf.nn.relu(conv)
```

Code 4. Definition of 2D convolution in the framework.

The convolution is defined here with separate weights' and biases' variables. The size of a convolutional filter is controlled by its weights. Strides are defined in the format of four 1D vectors, describing the sliding window step shift in any of the directions. However, usually only vertical and horizontal shifts are defined, resulting in the construction `[1, v_stride, h_stride, 1]`. Even though the functions from the snippet before are removing much of the boilerplate and are optimizing the code, still many of them can be shortened to simpler ones. This was achieved by the `tensorflow.contrib.layers` package. The example above can be simplified to the Code 5.

```
from tensorflow.contrib.layers import convolution2d

convolution2d(
    image,
    output_depth=6,
    kernel_size=(3, 3),
    stride=2,
    activation_fn=tf.nn.relu)
```

Code 5. Simplified convolution definition.

The neural network blocks can be easily combined to more general functions if a built model can be considered as a combination of layers. Let's consider as an example the encoder block from Figure 14, where the implemented network is shown (Code 6).

```
from tensorflow.contrib.layers import *

def encoder_block(tensor, output_depth, kernel_size=(3, 3)):
    # Define the first convolution
    conv1 = convolution2d(tensor, output_depth, kernel_size)
    # Define the second convolution
    conv2 = convolution2d(conv1,  output_depth, kernel_size)
    # Pooling with stride of 2 and kernel size of 2
    return max_pool2d(conv2, 2)
```

Code 6. Definition of the encoder block from the implemented network.

Here the instructions define two simple convolutions with customized parameters and a single pooling with a stride of two (default parameter) and a kernel size of two. This brings more simplicity in the definition of the network, allowing using the encoder block without many complex manipulations.

4.3 Training in TensorFlow

When the network is defined, supervised training is simple to setup using TensorFlow. First, the desired results should be represented in the way which could be fed into the network. Then, a loss function is defined to calculate how wrong the network is in its predictions with the current parameters. Afterwards, the optimizer should be initialized. It will be training the network, computing the gradient value from the loss function and update values using the gradient. The last step is to evaluate the network on the validation set. Code 7 shows an example of this workflow.

```
# Given training data
train_data, train_output
# And validation data
valid_data, valid_output
# Define a tensor for desired output
output = tf.placeholder(tf.float32, None)
# Define a loss function (tf.l2_loss does the same)
loss = tf.reduce_mean(tf.square(output - y))
# Define a gradient descent optimizer
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
# Define a train operation
train = optimizer.minimize(loss)
```

Code 7. Training a network.

MSE (mean square error) is used here for the loss function is used and the variables are optimized using the simplest gradient descent algorithm with a learning rate of 0.001. Afterwards, the session is created and the operations are executed (Code 8).

```
with tf.Session() as s:
    s.run(tf.global_variables_initializer())
    # Run train operation with training set
    s.run(train, feed_dict={x: train_data, output: train_output})
    # Run validation operation with validation set
    loss_value = s.run(loss, feed_dict={x: valid_data, output: valid_output})
```

Code 8. Training and then validating results.

The loss value can be then used as a metric to measure success during training. The last two operations should be repeated several times to achieve a better convergence of the gradient. To save a trained model a checkpoint can be created. It gives an possibility to persist a model during training and utilize it later.

4.4 Exporting a model

Model export allows persisting results during or after training as well as utilizing them in other environments. There are two simple ways to export a model. First, a checkpoint is created during the training and then it is used in the same codebase. However, this approach is not suitable for the production deployment, as it is bound to the training code. The second approach is to write the whole graph and its variables' values into binary file. This allows loading the definition of the model and its parameters into an optimized graph definition.

To create a checkpoint the following code 9 can be applied.

```
with tf.Session() as s:
    s.run(tf.global_variables_initializer())
    # Create a saver
    saver = tf.train.Saver()
    # Save a model
    saver.save(s, "model/saved_checkpoint")
```

Code 9. Saving a model to a checkpoint.

Here an instance of a saver is created. It is persisting a session state to the specified path. The main use case for this approach is saving while training a network, so only parameters are saved and graph definition is restored from its initial definition. To restore the network from a given checkpoint, one can use the following code.

```
with tf.Session() as s:
    s.run(tf.global_variables_initializer())
    # Create a loader (which is the same object of a saver)
    loader = tf.train.Saver()
    # Restore session
    loader.restore(s, "model/saved_checkpoint")
```

Code 10. Restoring the model value from a checkpoint.

When the network should be utilized in an external environment, the graph should be exported as a whole. Moreover, the variables inside should be converted to constants and compressed to optimize computation speeds and to allow easier export. The procedure is based on checkpoints and allows bringing a network to the higher spectrum of platforms. However, the solution here is more complex in terms of implementation and is more fixed to the predefined input and output nodes. To prepare the network for the export, first the graph definition should be exported without details in a text or binary format. The following example is writing a graph to a text file.

```
with tf.Session() as s:
    s.run(tf.global_variables_initializer())
    # Write graph
    tf.train.write_graph(s.graph_def, "models", "graph_def.pbtxt")
```

Code 11. Writing a graph structure.

Then the node is defined in the human readable format and it could be easily redefined even without the framework. As an example the multiplication node is shown below as Code 12:

```
node {
  name: "MatMul"
  input: "x"
  input: "w"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "transpose_a"
    value {
      b: false
    }
  }
}
```

Code 12. An example of the graph structure saved in text file.

Combined with the checkpoint saved above, the model can be “frozen” and written to the persistent storage using the example from the official Tensorflow tutorial (Google 2017). They are presenting a `freeze_graph` script for this purpose. After writing the graph definition and checkpoint data it could be exported with the following method (Code 13).

```
with tf.Session() as s:
    # Freeze graph
    freeze_graph(
        input_graph="models/graph_def.pbtxt",
        input_saver="",
        input_binary=False,
        initializer_nodes="",
        input_checkpoint="models/saved_checkpoint",
        output_node_names="output_node",
        restore_op_name="save/restore_all",
        filename_tensor_name="save/Const:0",
        output_graph="models/result.pb",
        clear_devices=False)
```

Code 13. Saving a graph with its values.

The parameters are self-explanatory and are optimized for the command line, so most of them are strings of booleans. Providing the graph definition, the saved checkpoint, and the name of the output node (defined as the function parameters), the graph is written to the output file. This definition can be then used in multiple environments to export this graph. Executing a script with the graph defined in this chapter will result in the following result.

```
Converted 2 variables to const ops.
8 ops in the final graph.
```

Code 14. Output after running Code 13.

5 WEB SERVER

The network can be connected to the client(robot) either through program interfaces or through the internet, basing on a web server. A local deployment implies faster speeds of reaction and removal of the network lag. However, it also limits the network to resources of the platform of a client whereas the model in the project is heavily relying on many computations. Therefore, it was decided to deploy the network in the cloud and to be accessed through a web interface.

To be able to train the network and produce the results efficiently, the web server should be GPU accelerated. This allows moving the most intensive part of the calculations to a faster and more effective parallel environment. However, that kind of virtual machines are usually unavailable on many cloud providers or are extremely expensive. Thus, Amazon, Microsoft and Google have recently started to advertise their services in this field catching a growing interest in the field of machine learning. For the purposes of the current project the Azure services were used, as credits for the platform were given by the commissioner.

5.1 Choice of tools

The network is trained using TensorFlow and its Python bindings. The best advantage of the framework is its ability to compile and optimize models for reusing in different environments. The choice of languages for the model embedding is based on C++ and an open source community is working on developing bindings for the most popular languages.

The choice of the web server framework is usually determined by a programming language. However, for the purpose of the neural networks the language is restricted by the neural network framework choice. Based on the fact that a neural network is trained using the TensorFlow framework to utilize the simplicity of Python for training, the choice is restricted to Go, Python, JVM family and C++. Based on this list, personal familiarity with the frameworks and restrictions from the commissioner, the final decision was to use Scala as a basis.

In the world of Scala one of the best choices for the microservices in terms of stability and performance is Akka Http library (TechEmpower 2016). This lightweight HTTP framework allows defining a service using concise idioms and leveraging the power of actors brought by the Akka library.

The actor model (Figure 15) is really powerful when one tries to define an independent worker for the system or separate parts of business logic and to move some of them to the background. With Scala the Akka library brings smart thread scheduling, reliable and concise state control and messaging management.

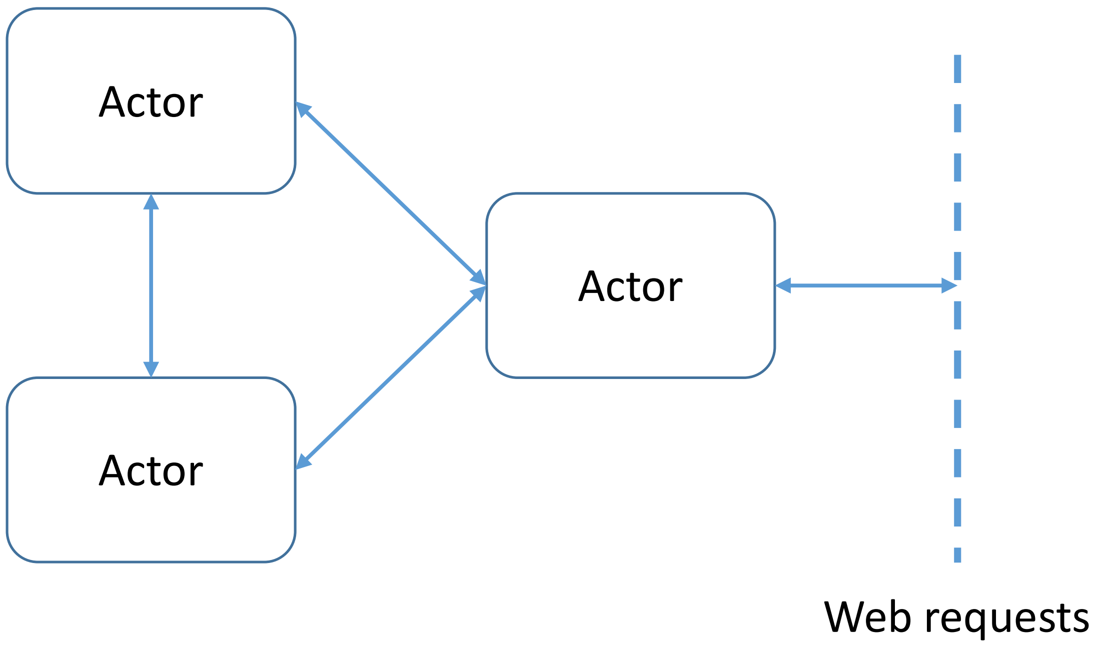

Figure 15. Actor model with three independent actors sharing their state and the web server based in the main thread

Based on the chosen tools, the architecture can be defined. The client is capturing images from the camera and sending them for the estimation through the HTTP protocol. The web interface on the server is receiving the images and is sending them further to the actor to compute depth estimation. The actor is connected through the Java Native Interface (JNI) to a C++ shared library which contains the neural network constant operations, exported after training and optimized for better performance. Afterwards, a computed depth is propagated backwards through the chain to a client (Figure 16).

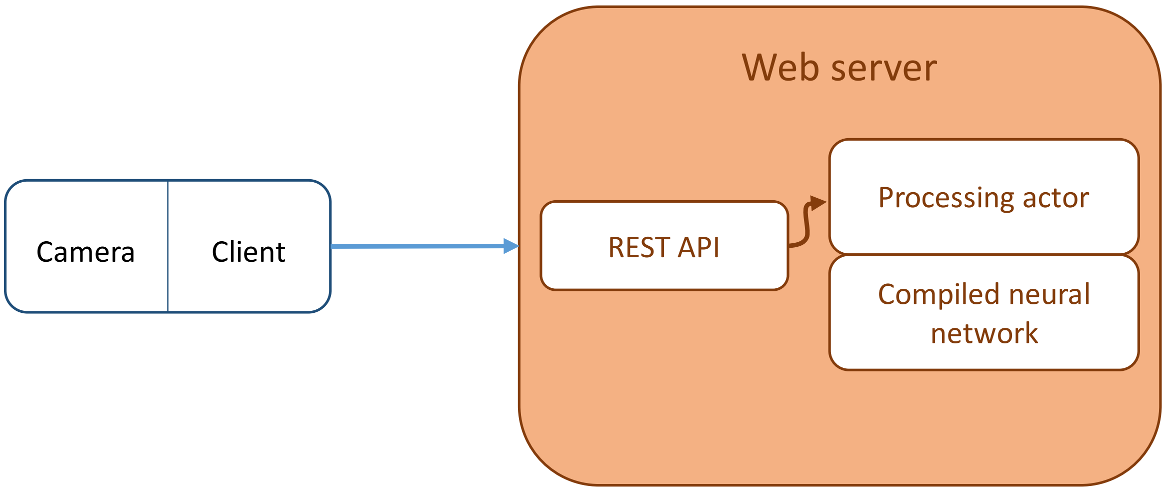

Figure 16. Final server architecture. The client is sending an image through the HTTP protocol, afterwards the web interface (REST API on the figure) is rerouting the image to the actor through a built-in actor messaging interface.

6 CONNECTING TO SERVER

After exporting the model, the server wrapper should be established to proceed with the idea of running the network in the GPU environment. As was described above, the Azure cloud should be used for the purpose of tests. Therefore, the first step is to setup a machine. Currently, Azure Portal offers GPU accelerated Linux virtual machines (VMs) in two regions: South Central US and East US (Microsoft 2017). While exploring the types of the available instances with desirable properties, one can find two categories: NV and NC.

The difference between them is in the GPU type used inside. NV machine series are using Tesla M60 graphics providing high quality desktop virtualization, while NC series are utilizing Tesla K80 power, which is perfectly suitable for the data mining and high-end computation (NVIDIA 2017). For the purposes of the project, a NC6 instance was used with one K80 GPU and Azure specific Ubuntu 16.04. After the VM was set up inside Azure, the SSH connection to it is available and an instance is ready to further setup. In the sections below, the TensorFlow library set up is described. After that, the process of building the web server is explained. Last, the final benchmark results are presented.

6.1 TensorFlow installation

Officially, there are two ways to install TensorFlow. The first one is a download through the Python package manager (`pip`), which includes two different packages: pure CPU support or an acceleration of a GPU. The drawback is a generalization of these builds, so many system specific acceleration features are not supported. The second one is far more complicated than a single command and implies a compilation from the sources. Despite its complexity for beginners, a possible improvement can reach five times less processing time in some environments.

Before TensorFlow compilation, official drivers and CUDA toolkits from NVIDIA should be installed to a system. Following the instructions from Google (2017), the CUDA toolkit should be downloaded to a system and then installed using the following commands:

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb
sudo apt-get update
sudo apt-get install cuda
```

Code 15. Installing CUDA drivers to the Ubuntu.

After the toolkit is installed to a system, the cuDNN library should be set up as well. From the downloads page Runtime and Developer packages are required to proceed to a successful compilation. After installing the packages using the `sudo dpkg -i` command, the configuration of environment is required. First of all, `bazel` building and dependency management system should be installed. Secondly, the `pip` package manager which comes as bundled with Python and several TensorFlow dependency packages: `numpy, dev, wheel` are required. Then, the code of the TensorFlow should be cloned from GitHub, using `git clone https://github.com/tensorflow/tensorflow`. Finally, the configuration of the framework package is available. The official tutorial includes exhaustive exploration of this process with an explanation of the possible tweaks. After the framework is configured, the package can be compiled using the following commands:

```
bazel build --config=opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

Code 16. Building TensorFlow package from sources.

This will result in the built pip package which will reside in the `/tmp/tensorflow_pkg` folder. It can be installed issuing `pip install` command. To install an additional Java wrapper, which is required for the server project, an additional JNI bridge is required. Described in the same set of the instructions from TensorFlow official website (Google 2017), the installation consists of copying jar to the project and additional native libraries specific to the device.

6.2 Server definition

As was discussed in Chapter 4, the web server is based on the Scala programming language and, particularly, the platform of the Akka Http library. This library implies concise and well elaborated interfaces for developers. Futhermore, a JVM platform has many strategies for the deployment preventing unknown issues and bottlenecks for the server environment. The Akka Http is based on the Akka Actor framework which includes a capability of flexible and fast management of threads and can be tuned to suit many use cases. First, for the server definition, a configuration of the endpoints and their purpose is required to prevent a mess when developing application interfaces. Three endpoints used in the project are defined in Table 1.

Table 1. Definition of the server endpoints.

HTTP method	Relative path	Input data	Output data	Purpose
`GET`	`/status`	`None`	`String: “UP”`	`Check if the server is available`
`POST`	`/image/depth`	`Multipart binary data (JPG image)`	`JPG image`	`Computation of the depth map of the applied image`

After the endpoints are configured, the routes can be defined inside the code. A route is a combination of the HTTP method, the path and the reaction of a server to the call with an input data. As an example of the route definition in the Akka Http, the code of the routes from Table 1 is shown in Code 17.

```
import akka.http.scaladsl.model.StatusCodes._
import akka.http.scaladsl.model._
import akka.http.scaladsl.server.Directives._

def route = path("status") {
  get {
    complete("UP")
  }
} ~ path("image" / "depth") {
  post {
    entity(as[Multipart.FormData]) { input =>
      /*
       * Do something with the input
       * Return the result
       */
      val result = Array[Byte]()
      complete(
        HttpResponse(entity = HttpEntity(
          ContentType(MediaTypes.`image/jpeg`),
          result)))
    }
  }
}
```

Code 17. Definition of the routes.

The method to define these routes is concise and simple. Each path represents a section in the object which can be easily restructured on demand. First, under the path “`status`” the static answer for a GET method is defined. Next, the definition of this path is concatenated with the next one with the tilde symbol. Inside the “`image/depth`” part an answer for POST request with a multipart data is defined. Based on this definition it is returning an empty JPG image to the client.

After the endpoints are defined, the actor could be set up. In the world of Akka an actor is located inside an actor system where it can be retrieved from. The actor can be defined as a class extension of `akka.actor.Actor`. The main method for an actor class is `receive`. It accepts messages sent to the actor and defines its behavior based on the received messages. A basic example of the actor is shown in Code 18.

```
import akka.actor.Actor


class TestActor extends Actor {

  override def receive = {
    case source: Array[Byte] => sender() ! depthMap(source)
    case _ => sender() ! None
  }
}
```

Code 18. Definition of the actor.

An example actor, named `TestActor`, has two replies: when the string “`say`” is given, it replies “`Hello!`” to a sender or `None` otherwise. To call the actor, it should be instantiated in the system and then called. The code below is illustrating the workflow in Code 19.

```
val actor = system.actorOf(Props[TestActor])
val future: Future[Array[Byte]] = actor ? image
```

Code 19. Calling an actor from a business logic.

The actor returns a future from the call. The future is similar to a lightweight thread and is utilized as a placeholder of the object that does not exist yet. This object can be easily composed and managed to create faster and consise asynchronous environment (Haller et al. 2017).

6.3 Connecting the network to the server

After the network was exported and the server basic environment is defined, the graph definition can be imported to the actor. Along with the receive method, the actor lifecycle is supported with methods that allow allocating and deallocationg resources. They can be used to initialize and release the network definition and to introduce better memory management. In the example the actor methods `preStart` and `postStop` are utilized for the graph import and initialization. The code is shown in Code 20.

```
private var graph: Graph = null

override def preStart() {
  val bytes = Files.readAllBytes(new File("/tmp/graph.pb").toPath)
  val g = new Graph()
  g.importGraphDef(bytes)
  if (graph != null) {
    graph.close()
  }
  graph = g
}

override def postStop() {
  if (graph != null) graph.close()
}
```

Code 20. Definition of the network graph in the actor.

Before starting the network is read to the RAM and a graph is defined from the values. After a stop method is called, the graph is closed to prevent memory leaks. The important issue is image decoding from compressed formats. The easiest way to execute that kind of operations and feed results to the network is to add corresponding operations to the graph. TensorFlow has operations to encode and decode the images, possessing more optimized and code efficient way to solve the problem. Therefore, after applying those operations, the network is expecting an array of bytes from a JPG image and the result is the depth map in the same format. An example code for execution is shown below:

```
private def depthMap(bytes: Array[Byte]) = {
  val s = new Session(graph)
  val t = Tensor.create(bytes)
  try {
    val result = s.runner.feed("input", t).fetch("output").run.get(0)
    val bytes = result.bytesValue
    result.close()
    bytes
  } finally {
    s.close()
    t.close()
  }
}
```

Code 21. Calling the network from the actor.

In the JVM environment it is very important to release resources from tensors and session to prevent memory leaks. The issue is addressed when calling the `close()` method of the objects.

7 RESULTS

To test the performance, two topics were considered. First, network was trained with different parameters on large dataset and the outcomes of that part were observed. Later, the server part was deployed to a remote virtual machine with a GPU acceleration. There, latency and other performance criteria were observed.

7.1 Training outcomes

The model for the project is trained on the KITTI dataset (Geiger et al. 2013) to produce a proof of the model concept. The images are taken from the raw dataset and are split into two parts: training set and validation set with the fraction of 80 to 20. Therefore, from 30 159 pairs of images, 6 032 are kept for a validation and 24 127 for a learning phase. The network is trained for 50 epochs, with a batch of eight pictures for each gradient computation because of the limitations of the memory on the GPU. The images are augmented on the fly to produce more deviation in the dataset and generalize a model better. A lightness and a color are scaled and a random shadow is added to the images.

As described in Section 3.3, the training consists of finding the disparity maps for four scales based on the pair of left and right images. During the learning process, the loss of the model has shown itself consistent and more performant than the *scale invariant mean square error* from the work of Eigen et al. (2014) and *structural dissimilarity* based on SSIM alone. Without the additional components in loss functions, the model tends to produce empty disparities or coarse ones without detailed structures. The training has shown that rectified linear units (ReLU) activations are not suitable for this type of network, as they tend to fix the predicted disparities at some point, so the activations were replaced with the exponential linear units (ELU). The approach of progressive schedules, where images at lower scales are optimized first was also implemented, but it has shown itself less consistent (Mayer et al. 2016).
The model was trained for 30 hours on a GPU accelerated Azure cloud virtual machine. The results were saved as checkpoints and later introduced for the evaluation. Using a test split, based on the KITTI dataset (Geiger et al. 2013) with

precomputed disparities, the model was measured for the mean square error on the closer half of the preliminary scaled produced disparities and ground truth ones. The second metric to measure was the accuracy measured by the values with a difference less than 1.25. This resulted in the mean error of 6.124 and relative accuracy around 0.89. The examples from the run on the test images are shown in Figure 17.

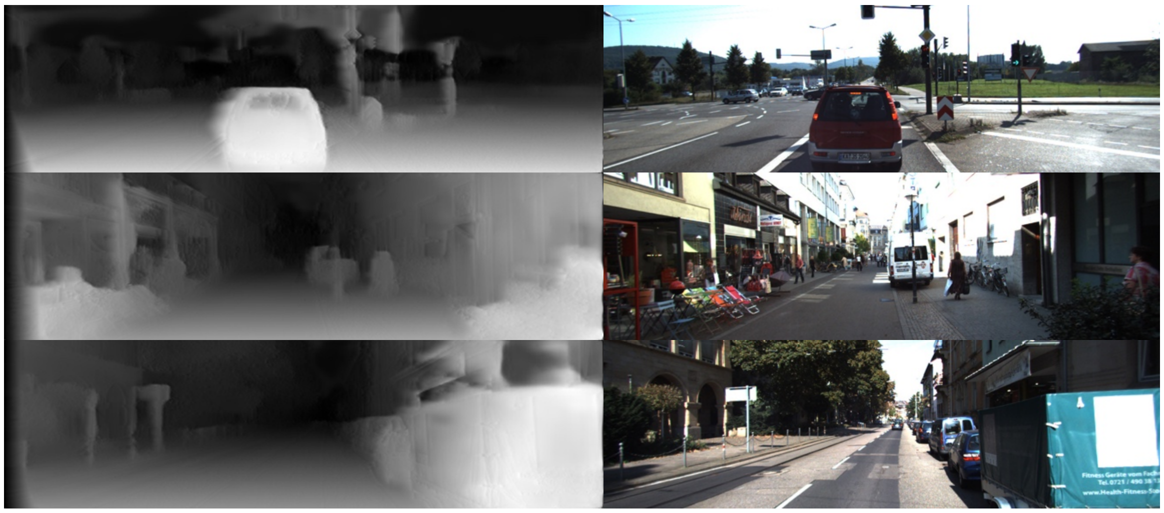

Figure 17. Produced disparities (left) and original images (right). Note that only the largest disparity used as an output here.

The figure shows that the model is able to distinguish details and select closer objects especially when they are located in the center of an image. However, many reflected details and the objects which are hardly distinguishable on the background are disappearing from the depth map. It possesses a serious problem and could possibly be solved by the combination with the approach of using binocular cameras to fix the details (Saxena et al. 2007).

7.2 Server performance

The latency is the most vital issue of a server which produces information for a robot. The robot usually is expecting the information in near realtime to be able to react appropriately. Therefore, a desired margin for the response set by commissioner was around 200 ms. The reaction time has a short lag, but it can be improved with some approximations to prevent inappropriate actions from the robot side.

The server was tested on three machines (Figure 18). First, the GPU acceleration was turned off and latency times were tested to produce a reference point. Afterwards, the performance was tested on the machine with a support with the GPU. Then, the measurements were acquired from the virtual machine with two GPUs attached. Each case was tested with 20 requests where the same image file with the resolution of 640x480 was sent. The requests were produced with a delay of 100 ms to emulate a real environment.

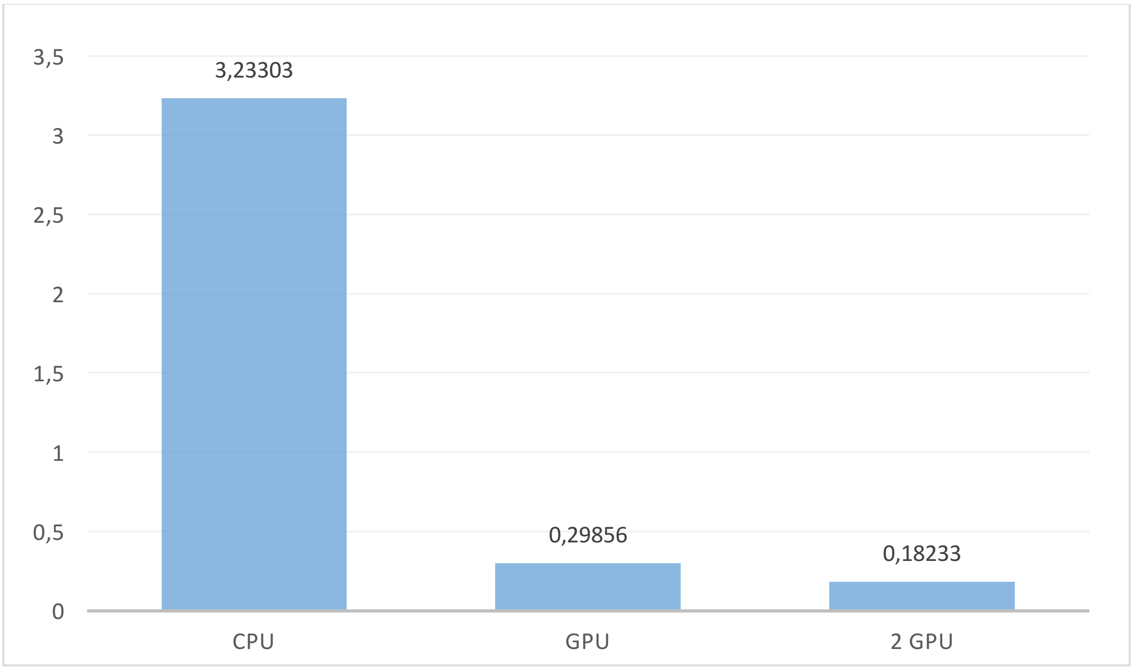

Figure 18. Average of 20 requests to different machines in seconds.

The outcomes have shown that a single GPU machine is able to produce the results near the margin. The response delay was almost ten times faster than with a CPU computations, although not enough to satisfy the requirements. However, with an addition of the second accelerator, most of the delay was coming from a network overhead, as the machine was replying almost in realtime.

8 CONCLUSION

The aim of the study was to create an accurate algorithm of the depth estimation based on a single image. Several restrictions were applied to the algorithm: deployment to the real environment, precision for the boundaries of closer objects and reasonable reaction times. As was shown above, the chosen algorithm was satisfying the requirements staying accurate while the implementation was keeping response times on a good level.

However, the final neural network introduces some challenges. Despite the fact that the network was performing well in the dataset tests and some photos taken from the real car, the depth estimation in an office or other indoor environments was not that accurate. Moreover, the network error rate on the reflecting and transparent objects was even higher than indoor. These problems can be solved by capturing an additional data of problematic scenes and fine-tuning the parameters. Nevertheless, the ability to generalize well even with the new data is questionable.

The server is performing well to satisfy expected delay rates. The actor and the chosen framework are introducing high performant and concurrent environment where overlapping requests are not dropped. Finally, after introduction of a machine with two GPUs, the main overhead of the workflow is the transmission of image files. This overhead cannot be ignored and could be possibly optimized during futher development.

The main point for the project performance is an ability to produce the results that suit for the actual robot. Despite the accuracy and the response times, tests with the real robot can show the vector for an additional development. Therefore, the work could be continued from an achieved point whenever the robot will be available for tests.

REFERENCES

Bergstra J., Breuleux O., Bastien F., Lamblin P., Pascanu R., Desjardins G., Turian J., Warde-Farley D. & Bengio Y. 2010. Theano: A CPU and GPU Math Compiler in Python. PDF Document. Available at: http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf [Accessed 17 April 2017]

Clabaugh C., Myszewski D. & Pang J. 2000. Neural Networks. WWW document. Available at: http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/index.html [Accessed 17 April 2017]

Coolen A.C.C. 2004. A Beginner's Guide to theMathematics of Neural Networks. PDF document. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.3556&rep=rep1&type=pdf [Accessed 15 May 2017]

Dosovitskiy A., Springenberg J. T., Tatarchenko M. & Brox T. 2015. Learning to Generate Chairs, Tables and Cars with Convolutional Networks. PDF document. Available at: https://arxiv.org/pdf/1411.5928.pdf [Accessed 3 May 2017]

Dumoulin V. 2017. Convolution arithmetic. WWW document. Available at: https://github.com/vdumoulin/conv_arithmetic [Accessed 11 May 2017]

Eigen D., Puhrsch C. & Fergus R. 2014. Depth Map Prediction from a Single Image using a Multi-Scale Deep Network. PDF document. Available at: https://papers.nips.cc/paper/5539-depth-map-prediction-from-a-single-image-using-a-multi-scale-deep-network.pdf [Accessed 17 April 2017]

Fischer P., Dosovitskiy A. & Ilg E. 2015. FlowNet: Learning Optical Flow with Convolutional Networks. PDF document. Available at: https://arxiv.org/pdf/1504.06852.pdf [Accessed 3 May 2017]

Geiger A., Lenz P. & Urtasun R. 2012. Vision meets Robotics: The KITTI Dataset. PDF document. Available at: http://www.cvlibs.net/publications/Geiger2013IJRR.pdf [Accessed 3 May 2017]

Glorot X., Bengio Y. 2010. Understanding the difficulty of training deep feedforward neural networks. PDF document. Available at: http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf [Accessed 13 May 2017]

Godard C., Aodha O. M & Brostow G. J. 2017. Unsupervised Monocular Depth Estimation with Left-Right Consistency. PDF document. Available at: https://arxiv.org/pdf/1609.03677.pdf [Accessed 5 May 2017]

Google. 2017. A Tool Developer’s Guide to TensorFlow Model Files. WWW document. Available at: https://www.tensorflow.org/extend/tool_developers/ [Accessed 13 May 2017]

Google. 2017. TensorFlow installation from sources. WWW document. Available at: https://www.tensorflow.org/install/install_sources [Accessed 14 May 2017]

Google. 2017. TensorFlow. WWW document. Available at: https://www.tensorflow.org/ [Accessed 11 May 2017]

Haller P., Prokopec A, Miller H., Klang V., Kuhn R & Jovanovic V. 2016. Futures and promises. WWW document. Available at: http://docs.scala-lang.org/overviews/core/futures.html [Accessed 15 May 2017]

Han J., Morag C. 1995. The influence of the sigmoid function parameters on the speed of backpropagation learning. PDF document. Available at: http://dx.doi.org/10.1007/3-540-59497-3_175 [Accessed 13 May 2017]

Karpathy A. 2017. A Peek at Trends in Machine Learning. WWW document. Available at: https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106 [Accessed 16 April 2017]

Kingma D. P., Lei Ba J. 2017. Adam: A Method for Stochastic Optimization. PDF document. Available at: https://arxiv.org/pdf/1412.6980.pdf [Accessed 15 May 2017]

LeCun Y., Chopra S., Hadsell R., Ranzato M. & Jie Huang F. 2006. A Tutorial on Energy-Based Learning. PDF document. Available at: http://yann.lecun.com/exdb/publis/pdf/lecun-06.pdf [Accessed 15 May 2017]

Liu F., Shen C. & Lin G. 2015. Deep Convolutional Neural Fields for Depth Estimation from a Single Image. PDF document. Available at: http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Liu_Deep_Convolutional_Neural_2015_CVPR_paper.pdf [Accessed 5 May 2017]

Long J., Shelhamer E. 2015. Fully Convolutional Networks for Semantic Segmentation. PDF document. Available at: https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf [Accessed 3 May 2017]

Microsoft. 2017. Microsoft Azure: Linux Virtual Machine Pricing. WWW document. Available at: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/ [Accessed 14 May 2017]

NVIDIA Corporation. 2016. End to end learning for self-driving cars. PDF document. Available at: http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf [Accessed 17 April 2017]

NVIDIA. 2017. Tesla server solutions. WWW document. Available at: http://www.nvidia.com/object/tesla-servers.html [Accessed 14 May 2017]

OpenCV. 2017. Depth map from stereo images. WWW document. Available at: http://docs.opencv.org/trunk/dd/d53/tutorial_py_depthmap.html [Accessed 2 May 2017]

Saxena A., Chung S.H. & Ng A.Y. 2007. 3-D Depth Reconstruction from a Single Still Image. PDF document. Available at: https://link.springer.com/content/pdf/10.1007%2Fs11263-007-0071-y.pdf [Accessed 4 May 2017]

Snyman J. 2005. Practical mathematical optimization: an introduction to basic optimization theory and classical and new gradient-based algorithms. Springer Science & Business Media.

TechEmpower. 2016. Web Network Benchmarks: Single query. WWW document. Available at: https://www.techempower.com/benchmarks/#section=data-r13&hw=ph&test=db [Accessed 6 May 2017]

Van Veen F. 2016. The neural network zoo. WWW document. Available at: http://www.asimovinstitute.org/neural-network-zoo/ [Accessed 17 April 2017]

Wikipedia. 2017. Gradient descent. WWW document. Available at: https://en.wikipedia.org/wiki/Gradient_descent [Accessed 17 April 2017]

Zbontar Z. & LeCun Y. 2016. Stereo Matching by Training a Convolutional Neural Network to Compare Image Patches. PDF document. Available at: https://arxiv.org/pdf/1510.05970.pdf [Accessed 4 May 2017]