BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

SEMANTIC SEGMENTATION OF IMAGES USING CONVOLUTIONAL NEURAL NETWORKS

SÉMANTICKÁ SEGMENTACE OBRAZU POMOCÍ KONVOLUČNÍCH NEURONOVÝCH SÍTÍ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. FILIP ŠPILA**

AUTOR PRÁCE

SUPERVISOR — **doc. Ing. JIŘÍ KREJSA, Ph.D.**

VEDOUCÍ PRÁCE

BRNO 2020

Abstrakt

Summary

Klíčová slova

Keywords

ŠPILA, F.*Semantic segmentation of images using convolutional neural networks.* Brno: Vysoké učení technické v Brně, Faculty of Mechanical Engineering, 2020. 37 s. Vedoucí doc. Ing. Jiří Krejsa, Ph.D.

Prohlasuji

Bc. Filip Špila

Dekuji, FS

Bc. Filip Špila

Contents

1. Introduction

Image segmentation is one of the essential parts of computer vision and autonomous systems alongside with object detection and object recognition. The goal of semantic segmentation is to automatically assign a label to each object of interest (person, animal, car, etc.) in a given image while drawing the exact boundary of it and to do this most robustly and reliably possible.

We can see a real-world example in Figure 1. Each pixel of the image has been assigned to a specific label and represented by a different colour. Red for people, blue for cars, green for trees, etc. This is unlike the mere image classification task where we classify the image scene as a whole. It is appropriate to say that semantic segmentation is different from so-called instance segmentation, where one not only cares about drawing boundaries of objects of a certain class but also wants to distinguish between different instances of the given class. For instance, all people in the image (each instance of the 'person' class) would all have a different colour.

It turns out that semantic segmentation has many different applications in fields such as driving autonomous vehicles, human-computer interaction, robotics, and photo editing/creativity tools. The most recent development shows the increasing need for reliable object recognition in self-driving cars because it is crucial for the models to understand the context of the environment they're operating in.

The presented work focuses on research and implementation of one particular segmentation method that uses convolutional neural networks (CNNs). CNNs belong to the family of machine learning algorithms and got under attention mainly due to their groundbreaking success in image classification challenges (ImageNet). They subsequently found their use in segmentation tasks where researchers take the most well-performing CNN architectures and use it as the first stage of the segmentation algorithm.

Figure 1.1: Segmentation of an urban road scene

2. Problem statements

The assignment of this thesis consists of several expected achievements. Firstly, a promising segmentation method using CNNs needs to be found and implemented. It is expected that the neural network will be as straightforward as possible while still being likely to be capable of giving satisfactory results for the chosen use case (segmentation of a path in an outdoor environment for robot navigation). The images will be provided by the supervisor of the thesis and used to train and validate the network performance. Also, the author will pick an appropriate software tool for creating Ground Truths[1] and use it to create the final training and validating datasets. Lastly, the network should be trained with various sets of hyperparameters[2] to get a closer idea of the network's training behaviour and to ensure the best possible results.

[1] Manually created image labels that serve as a reference for the network to validate its current accuracy of prediction and to compute the needed adjustments of its parameters to get closer to the desired output

[2] See chapter XY for the hyperparameter definition

3. Research and theory

The first part of this chapter gives an introduction to artificial neural networks (ANN). It begins by definition of fundamental terms needed to understand the core principles of ANN. Due to the fact that the research in this area is still heavily ongoing, the more advanced techniques described here may soon be out of date or replaced by better-performing ones and therefore the theoretical background is limited only to the extent relevant for the finally chosen network architecture.

The second part presents some of the main approaches based on machine learning researches have recently used to tackle the semantic segmentation problem. However, not all of them use CNNs as the core algorithm. This part summarizes the main key points from the corresponding papers that contributed to this topic by presenting novel architectures and principles. It finishes by a more detailed description of a method that is eventually found the most promising and thus selected for the final implementation.

3.1. Architecture of artificial neural networks

Artificial neural network algorithms are inspired by the architecture and the dynamics of networks of neurons in human brain. They can learn to recognize structures in a given set of training data and generalize what they have learnt to other data sets (supervised learning). In supervised learning, one uses a training data set of correct input/output pairs. One feeds an input from the training data into the input terminals of the network and compares the states of the output neurons to the target values. The network trainable parameters are changed as the training continues to minimize the differences between network outputs and targets for all input patterns in the training set. In this way, the network learns to associate input patterns in the training set with the correct target values.

3.1.1. Feed-forward networks

The goal of a feed-forward neural network is to find a non-linear, generally n-dimensional function that maps the space of inputs x to the space of outputs y. In other words, to learn the function [zdroj SANTIAGO]

$$f^* : \mathbb{R}^m \to \mathbb{R}^n, f^*(x;\phi) \tag{3.1}$$

where ϕ are trainable parameters of the network. The goal is to learn the value of the parameters that result in the best function approximation, by solving the equation

$$\phi \leftarrow \text{arg min}\, L(y, f^*(x;\phi)) \tag{3.2}$$

where L is a loss function chosen for the particular task. One can understand the term 'loss function' simply as a metric of 'how happy we are about the output that the network gives us for a given input' and therefore $f^*(x;\phi)$ is driven to match the ideal function $f(x;\phi)$ during network training.

The structure of a feed-forward network is usually composed of many nested functions. For instance, there might be three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain to the form

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))) \tag{3.3}$$

These models are referred to as feed-forward because information flows from the deepest nested function $f^{(1)}$ taking x as its direct input to other functions in the chain and finally to the output y. One can name the functions starting by $f^{(1)}$ as the first layer (input layer) of the network, $f^{(2)}$ is called the second layer, and so on. The final layer of the network is called the output layer.

Recall that in supervised learning one needs a set of training data, in this case, a set of matching x, y[1] pairs. The training examples specify directly what the output layer must do at each point x; it must produce a value that is as close as possible to y. The behaviour of the other layers is not specified by the training data which is why we call these layers 'hidden layers'. Figure XY shows a feed-forward neural network with two hidden layers.

A neural network can be seen as something capable of modelling practically any function we can think of [see general approximation theorem]. The power of this brings us to the definition of a classification task. In this task, the function the network approximates has discrete states, true/false in the simplest case.

3.1.2. McCulloch-Pitts neurons

Layers in Figure XY can be further divided into distinct computational units (again, just another nested functions) called neurons. This is where the resemblance to biological neurons comes into play. The neurons are mathematically modelled as linear threshold units (McCulloch-Pitts neurons), they process all input signals coming to them from other neurons and compute the output. In its simplest form, the model for the artificial neuron has only two states, active or inactive. If the output exceeds a given threshold then the state of the neuron is said to be active, otherwise inactive. The model is illustrated in Figure 1.4. Neurons usually perform repeated computations, and one divides up time into discrete time steps $t = 0, 1, 2, 3, ...$ The state of neuron number j at time step t is denoted by

$$n_j(t) = \begin{cases} 0 & \text{inactive,} \\ 1 & \text{active.} \end{cases} \tag{3.4}$$

Given the signals $n_j(t+1)$, neuron number i computes

$$n_j(t+1) = \theta_H \left(\sum_j w_{ij} n_j(t) - \mu_i \right) \tag{3.5}$$

As written, this computation is performed for all i neurons in parallel, and the outputs n_i are the inputs to all neurons at the next time step, therefore the outputs have the time argument $t+1$.

[1] Outputs y are often called labels in classification tasks)

The weights w_{ij} are called synaptic weights. Here the first index i refers to the neuron that does the computation, and j labels all neurons that connect to neuron i. The connection strengths between different pairs of neurons are in general different, reflecting different strengths of the synaptic couplings.

The argument of θ_H is often referred to as the local field

$$b_i = \sum_j w_{ij} n_j(t) - \mu_i \tag{3.6}$$

Equation XY basically shows that the neuron performs a weighted linear average of the inputs n_j and applies an offset (threshold) which is denoted by μ_i. Finally, the function θ_H is referred to as the activation function.

3.1.3. Activation functions

The general motivation for using activation functions is to bring non-linearity to the model. In the simplest case that has been discussed so far, the neurons can only have the states 0/1, which in terms of the activation function corresponds to the Heaviside function

$$\theta_H(b) = \begin{cases} 1 & \text{for } b > 0, \\ 0 & \text{for } b < 0. \end{cases} \tag{3.7}$$

In practice, however, the simplest model must be generalized by allowing the neuron to respond continuously to its inputs. This is necessary for the optimization algorithms used in the training phase to operate smoothly. To this end, one replaces Eq. XY by a general continuous activation function $g(b)$. An example is shown in Figure XY.

One can choose from several activation functions which all come with their 'pros and cons' for a particular application the network is used for. In general, there are a few requirements these functions should meet:

- **Nonlinearity**. As discussed above, the non-linearity is a general ability of a neural network allowing it to model very complex functions.
- **Monotocity and nondecreasibility**. This allows certain optimization algorithms to perform more stable as we'll see further.
- **Differentiability (or at least piecewise differentiability)**. This is useful not only in terms of stability of the optimization algorithms but also for the analytical derivation of the update rule for the network parameters during optimization.

There are activation functions designed specifically for the output layer. The reason for that comes from the definition of a classification task, where we would like to interpret the outputs of the network as relative probabilities of the input belonging to a certain class. For this one can use the commonly-used softmax activation function. We say 'relative' because the network's decision is only based on the features of one particular pattern in comparison with other data we used during training. Hence, the probabilities computed by the softmax classifier are better thought of as confidences where the ordering of the scores is interpretable, but the absolute numbers (or their differences) technically are not. [https://cs231n.github.io/linear-classify/]

Another possibility for the output activation function is the sigmoid function, which is used for both input/hidden and output layers. Here are the most frequently used activation functions:

- **Sigmoid**

$$g(x) = \frac{1}{1 + e^{-x}} \tag{3.8}$$

  This function has a clear interpretation of neuron states - active/inactive is represented by values 0/1. The sigmoid function is currently out of favour for large networks. In short, it does not have optimal properties for the learning algorithm because it saturates very quickly. Also, the fact that its mean value is non-zero doesn't have a positive impact on the learning process either. [https://cs231n.github.io/neural-networks-1/] [Groman]

- **Hyperbolic tangent**

$$g(x) = \tanh(x) \tag{3.9}$$

  Unlike the sigmoid, the range of its output is in the interval <-1,1> and the output is therefore zero-centered. In practice, the tanh non-linearity is always preferred to the sigmoid non-linearity. [https://cs231n.github.io/neural-networks-1/]

- **Rectified Linear unit (ReLu)**

$$g(x) = \max(0, x) \tag{3.10}$$

  The authors of this function found the inspiration in real biological neurons: there is a threshold below which the response of the neuron is strictly zero, as shown in Figure XY. The derivative of the ReLU function is discontinuous at $x = 0$. A common convention is to set the derivative to zero at $x = 0$. It is now the standard function to use in large networks for image recognition. [mehlig]

- **Parametric (leaky) ReLu**

$$g(x) = \max(x, \alpha x) \tag{3.11}$$

  By modifying the previously introduced function one gets a version of ReLu intended to address the biggest drawback of ReLu, which is the fact that some neurons may become dead (their output will be always zero) and thus not contribute to the network's output. Unfortunately, there's generally no guarantee that using Leaky ReLu instead of ReLu will always mean better results. [stanford L4]

- **Output classifier - softmax**

  The softmax function is designed to be used in output layers. This so-called classifier differs from other activation functions by its dependency on other neurons in the layer

$$O_i = \frac{e^{\alpha b_i^{(L)}}}{\sum_{k=1}^{M} e^{\alpha b_k^{(L)}}} \tag{3.12}$$

  Here $b_i^{(L)} = \sum_j w_{ij}^{(L)} V_j^{(L-1)} - \theta_j^{(L)}$ are the local fields of the neurons in the output layer. The constant α is usually taken to be unity. Softmax has three important properties: first that $0 \geq O_i \geq 1$. Second, the values of the outputs sum to one $\sum_{i=1} O_i = 1$. This means that the outputs of Softmax units can be interpreted as probabilities. Third, the outputs are monotonous: when $b_i^{(L)}$ increases then O_i increases but the values O_k of the other output neurons $k \neq i$ decrease. [mehlig]

3.1.4. Multilayer perceptrons

Perceptron is a layered feed-forward network. An example of such a network is shown in Figure XY The left-most layer is the input layer. To the right follows a number of layers consisting of McCulloch-Pitts neurons. The right-most layer is the output layer where the output of the network is read out, usually as softmax probabilities. The other neuron layers are called hidden layers, their states are not read out directly. [mehlig]

In perceptrons, all connections (called weights) w_{ij} are one-way; every neuron (or input terminal) feeds only to neurons in the layer immediately to the right. There are no connections within layers, or back connections, or connections that jump over a layer. There are N input terminals. We denote the inputs coming to the input layer by

$$x(\mu) = \begin{bmatrix} x_1^{(\mu)} \\ x_2^{(\mu)} \\ \vdots \\ x_N^{(\mu)} \end{bmatrix} \tag{3.13}$$

The index μ labels different input patterns in the training set. The network shown in Figure XY would perform these computations

$$V_j^{(\mu)} = g(b_j^{(\mu)}) \quad \text{where} \quad b_j^{(\mu)} = \sum_k w_{jk} x_k^{(\mu)} - \theta_j \tag{3.14}$$

$$O_i^{(\mu)} = g(B_i^{(\mu)}) \quad \text{where} \quad B_i^{(\mu)} = \sum_j W_{ij} V_j^{(\mu)} - \Theta_i \tag{3.15}$$

Here $V_j^{(\mu)}$ denoted the output of hidden layer j based on the local field $b_j^{(\mu)}$ and activation function $g(b)$. The parameters w_{jk} and θ_j denote weights and thresholds of the layer j. The corresponding computations are made fop the output layer, whose parameters are denoted by capital letters.

Multilayer perceptron has generally N hidden layers. If it has more than two hidden layers, we usually start to talk about a deep network.

Linear separability

The reason we use hidden layers is to tackle linearly inseparable classification problems. Linear separability is shown in Figure XY, where the input to the network is two-dimensional and we classify the input data into two classes (marked as black and white points in the graph). A classification problem is linearly separable if one is able to draw a single line (a single plane in case of three inputs, etc.) to divide the input space into two distinct areas and hence solve the classification task. In general, the curve that separates the space into sub-areas each representing a class is called the decision boundary. The position of the decision boundary is determined by the values of weights and thresholds of the neurons. These parameters are found by training the network. In the case shown in Figure XY, the line dividing the 2D space of inputs corresponds to the simplest possible case: a single neuron in the network. [mehlig]

An example of a linearly inseparable task is shown in Figure XY. We need to divide the input space to more than two regions to solve the classification. The network corresponding to the case in Figure XY has one hidden layer with three neurons. By doing this we map the input space of size n = 2 to the hidden space of size m = 3 and use it as an input to other layers.

One can ask how many hidden layers and neurons should we use for a particular task? In short, the answer depends on how complicated the distribution of input patterns is.

3.2. Training of artificial neural networks

Artificial neural networks are trained using iterative optimization algorithms. During training, one needs to choose the right loss function whose value goes to zero when the network produces the expected output. In each step of optimization, the trainable parameters are changed to achieve this. The effect each parameter has on the value of the loss function is determined by calculating the gradient of the loss function with respect to the particular parameter in the network. The way this information is used is then subject to the chosen algorithm.

3.2.1. Loss function

Loss function is a metric of our happiness with the network's output. The choice depends on the nature of the task the network is used for and on the activation function used in the output layer. During training, the loss function is the one whose value is being optimized. Here are the most commonly used ones:

- **Mean Squared Error (MSE)**

$$L = \frac{1}{2} \sum_{\mu i} \left(t_i^{(\mu)} - O_i^{(\mu)} \right)^2 \tag{3.16}$$

  MSE is used for regression tasks, often in the combination with the sigmoid function in the output layer. [groman]

- **Negative Log Likelihood**

$$L = -\sum_{\mu i} t_i^{(\mu)} \ln(O_i^{(\mu)}) \tag{3.17}$$

The negative log likelihood is used for classification tasks in the combination with the softmax classifier.

- **Cross Entropy Loss**

$$L = -\sum_{\mu i} t_i^{(\mu)} \ln(O_i^{(\mu)}) + (1 - t_i^{(\mu)}) \ln(1 - (O_i^{(\mu)})) \tag{3.18}$$

Very similar to the negative log likelihood loss. The difference is that it works with the sigmoid activation function. [mehlig]

3.2.2. Gradient optimization and backpropagation

Backpropagation is a way in which information about the correctness of the output flows through the network for the parameters in all layers to be adjusted. The scheme is shown in the network in Figure XY, where backpropagation is applied to a multilayer perceptron. Everytime we feed the network with an input pattern μ, we get the values of outputs of the neurons in all layers. This is called the forward pass (inference, left-to-right pass). Then we want to evaluate the correctness of the output and pass that information back to the network. The second phase is called the backpropagation because the error propagates from the output layer to the layers on the left. [mehlig]

The goal is to give the optimization algorithm values of gradients for all network parameters in each its iteration. One needs to find partial derivatives of the loss funtion with respect to these parameters. In deep networks, one achieves this by applying the chain rule to the formula for calculating the loss function. [mehlig]

Gradient descent

The general formula for the gradient descent algorithm goes as follows:

$$\delta\phi = -\eta \frac{\partial H}{\partial \phi} \tag{3.19}$$

where ϕ is the parameter we care about (weights, thresholds). In each iteration, we compute the derivative of the loss function with respect to all network parameters and thus get the increments $\delta\phi$. Parameter η is called the learning rate and is always a small number greater than zero. This parameter determines the size of the step we take in the way of the steepest descent in the landscape (in case of two parameters) of the loss function. This is shown in Figure XY. We see that the behaviour and convergence of the algorithm are strongly dependent on choosing the learning rate value: if the steps are too small, the learning will be slow and we are more likely to end up in a local minimum. [mehlig] On the other hand, if the value of it is too big, the algorithm may even start to 'climb up the hill' and cause the loss function to grow.

Given a multilayer perceptron with hidden layers and their parameters w_{mn}, θ_m, output layer with weights W_{mn}, Θ_m and the MSE loss function, the gradient descent algorithm gives the weight updates in the form

$$\delta W_{mn} = -\eta \frac{\partial L}{\partial W_{mn}} = \eta \sum_{\mu=1}^{p} (t_m^{(\mu)} - O_m^{(\mu)}) g'(B_m^{(\mu)}) V_n^{(\mu)} \tag{3.20}$$

where p is the total number of training samples, $V_n^{(\mu)}$ is the vector of outputs of neurons in the previous layer n for the sample μ. For clarity, one usually defines the 'weighted error' as

$$\Delta_m^{(\mu)} = (t_m^{(\mu)} - O_m^{(\mu)}) g'(B_m^{(\mu)}) \tag{3.21}$$

The update rules for hidden layers are also obtained by using chain rule, which gives

$$\delta w_{mn} = -\eta \frac{\partial L}{\partial w_{mn}} = \eta \sum_{\mu}^{p} \sum_{i}^{N} \Delta_i^{(\mu)} W_{im}\, g'(b_m^{(\mu)}) x_n^{(\mu)} \tag{3.22}$$

while putting

$$\delta_m^{(\mu)} = \sum_{i}^{N} \Delta_i^{(\mu)} W_{im}\, g'(b_m^{(\mu)}) \tag{3.23}$$

Putting all the above together gives

$$\delta w_{mn} = \eta \sum_{\mu}^{p} \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta W_{mn} = \eta \sum_{\mu=1}^{p} \Delta_m^{(\mu)} V_n^{(\mu)} \tag{3.24}$$

Similarly, we get the update rule for thresholds (see []). In summary, the steps of back-propagation + gradient descent are following:

Algorithm 1 Gradient descent

1: Pick input pattern μ from the training set and perform forward pass
2: Compute errors $\Delta_m^{(\mu)}$ for output layer
3: Compute errors $\delta_m^{(\mu)}$ for hidden layers
4: Perform updates $w_{mn} = w_{mn} + \delta w_{mn}$ and $\theta_{mn} = \theta_{mn} + \delta\theta_{mn}$, the same for the output layer

Stochastic gradient descent

One of the general issues one encounters when using gradient methods is the risk of getting stuck in a local minimum. To fight this, the idea is to add a little bit of noise to the optimization process. This can be achieved by introducing the concept of mini-batches (small groups of samples from the training data). In Equation XY we see that in each iteration one needs to sum over all training patterns in the set to obtain the value of the gradient. In stochastic gradient descent (SGD), one only sums over randomly chosen mb patterns from the training set and then immediately performs the weight update. The process is repeated until all training data have been used (this we call a training epoch). In mini-batches, samples appear only once per epoch and the entire training set is usually shuffled after each epoch.

Applying the above, the Equation XY slightly changes to

$$\delta w_{mn} = \eta \sum_{\mu=1}^{mb} \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta W_{mn} = \eta \sum_{\mu=1}^{mb} \Delta_m^{(\mu)} V_n^{(\mu)} \tag{3.25}$$

Vanishing and exploding gradient problem

When we compute the weight increments using MSE, it turns out that as we go further from the output layer, the term $g'(b)$ accumulates with each next layer. The point is that MSE is often used with the sigmoid activation functions, whose value of derivative drops to a small number in its area of saturation. In a result, we get very small weight increments as we go to the left in the network layers and the training of these layers slows down rapidly. This phenomenon is known as the vanishing gradient problem. One of the ways to address this issue is using activation functions that don't saturate, like ReLUs.

Similarly, one can run into trouble when the values of the derivative of activation function are larger than one. Then the value of the gradients may start growing exponentially: this is called the exploding gradient problem [https://medium.com/learn-love-ai/the-curious-case-of-the-vanishing-exploding-gradient-bf58ec6822eb]

Adaptation of the learning rate

There are several ways to make the stochastic gradient descent algorithm perform better. The key achievement is to prevent it from getting stuck in local minima. Secondly, gradient methods also tend to slow down in the areas of minima that are very shallow. The obvious solution to this is to take bigger steps by using a larger value of the learning rate, but this makes the algorithm oscillate in the minimum we'd consider to be optimal. [mehlig] One way to tackle this is to implement the mechanism called momentum, which is a good name because it tells a lot about the principle it introduces.

When using momentum, we can imagine the SGD algorithm behave like a ball that rolls downhill and develops speed over time. [stanford L7] The resulting move made by the algorithm in the landscape of the loss function is, therefore, a combination of the gradient vector and the velocity vector.
The update rule for weights gets modified to

$$\delta w_{ij}^{(t)} = -\eta \frac{\partial H}{\partial w_{ij}^{(t)}} + \alpha \delta w_{ij}^{(t-1)} \tag{3.26}$$

where $t = 0, 1, 2, .., n$ is the iteration number, while $\delta w_{ij}^{(0)} = \partial H / \partial w_{ij}^{(0)}$ is the increment in the zeroth time step. The parameter $\alpha > 0$ is the momentum constant.

There are other ways of implementing momentum, such as the nowadays most commonly used Nesterov's accelerated gradient method (see [mehlig] for details). This algorithm differs from the simple momentum by altering the steps the algorithm takes to do the final update: it first moves in the direction of the velocity, then evaluates the gradient at that point and corrects the previous step. It turns out that this scheme perform better in practice. [stanford L7]

Other optimization algorithms

- **AdaGrad**

  AdaGrad [] is another gradient based algorithm. In previously discussed gradient descent, the parameters were updated with the same learning rate in every step of the algorithm. AdaGrad adapts the learning rate based on the accumulated square of gradients. [see stanford L7]. The problem is that it might get stuck in the saddle points beacause the size of the steps it takes gets very small as the training goes on. [L7]

- **RMSprop**

  Another adaptive algorithm [original paper] that addresses the issue discussed in AdaGrad. It prevents the steps to get infinitely small by introducing a decay parameter to suppress the effect of the accumulated square of gradients.

- **Adam**

  Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of the gradient itself like SGD with momentum.

- **AdaDelta**

  This algorithm is an extension of AdaGrad and tackles its tendency to drop some of the learning rates to almost infinitely small values. []

- Second order optimization?

  ALGORITHM COMPARISON - FIGURE XY

3.2.3. Improving training performance

Initialization of weights and thresholds

The standard choice is to initialise the weights to independent Gaussian random numbers with mean zero and unit variance and the thresholds to zero. But in networks that have large hidden layers with many neurons, this scheme may fail. [mehlig] This is because the variance of weights is not taken care of, which leads to very large or small activation values, resulting in exploding or vanishing gradient problem during backpropagation. [https://medium.com/@shoray.goel/kaiming-he-initialization-a8d9ed0b5899] Here are some of the more advanced initialization methods:

- **Xavier initialization**

  Xavier initialization sets the layer's weights to values from Gaussian distribution. The mean and standard deviation are determined by the number of incoming and outcoming network connections to the layer. These random numbers are then divided by the square root of the number of incoming connections. This method works well with the tangent and sigmoid activation functions but fails when using ReLUs. see [stanford L6]

- **MSRA initialization**

  This method differs from Xavier only by using different factor to scale the Gaussian distributed numbers. It turns out that this small change works much better when using ReLU activation function. see [stanford L6]

Overfitting and regularization

A network with more neurons may classify the training data better because it accurately represents all specific features of the data. But those specific properties could look quite different in new data and the network may fit too fine details that have no general meaning. As a consequence, we must look for a compromise between the accurate classification of the training set and the ability of the network to generalise. The problem is illustrated in Figure XY and is called overfitting. [mehlig]

During training, one can run into trouble when the weights start to grow causing the local fields to become too large. When that happens, the activation function like the sigmoid reaches its plateau very soon which slows down the training (vanishing gradient). One way to solve this problem is to reduce weights by some factor every n iterations. [mehlig] This is done by adding another term to the loss function, like

$$R_{L2}(w) = \frac{\gamma}{2}\sum_{ij} w_{ij}^2 \quad \text{or} \quad R_{L1}(w) = \frac{\gamma}{2}\sum_{ij} |w_{ij}| \tag{3.27}$$

which are referred to as the L2 and L1 regularization. [mehlig]

These two regularization schemes tend to help against overfitting. They add a constraint to the problem of minimising the energy function. The result is a compromise between a small value of H and small weight values. The idea is that a network with smaller weights is more robust to the effect of noise. When the weights are small, then small changes in some of the patterns do not give a substantially different training result. When the network has large weights, by contrast, small changes in the input may give significant differences in the training result that are difficult to generalise (Figure 6.9).

Dropout

Dropout is a very simple scheme that helps against overfitting. During training, some random portion of the neurons in the network are ignored for each input pattern/mini-batch with the probability of p. This can be thought of as making the network adapt to the sparsity of the remaining neurons and making their effect on the output spread equally over the network. Another interpretation is that we are training different net architectures at the same time. When the training is done, the output of each neuron is multiplied by the probability p of a neuron to be dropped out during training (weight averaging [segnet bayesian source]). [mehlig][L7]

Pruning

Pruning is another regularization technique. The idea is to first train a deep network with many hidden layers and when the training is done, remove some portion of the hidden neurons completely. This turns out to be more efficient in terms of generalization properties than using small networks that were not trained with pruning. [mehlig]

Data augmentation

The general rule is that the bigger the training dataset the better the network generalises. However, expanding dataset manually may be very expensive. This leads to the idea of expanding it artificially. In image classification tasks, this can be done by randomly cropping, scaling, shifting and mirroring the data. [mehlig] [MEHHLIG 90]

Early stopping

One way to avoid overfitting is by using cross-validation and early stopping. The training data are split into two sets; the training set and the validation set. The network is trained on the training set. During training, one monitors not only the energy function for the training set but also the energy function evaluated on the validation data. As long as the network learns the general features of the input distribution, both training and validation energies decrease. But when the network starts to learn specific features of the training set, then the validation energy saturates or may start to increase. At this point, the training should be stopped. [mehlig]

FIGURE

Data pre-processing

For most cases, it is advisable to shift the data so it has zero mean before the training begins. When classifying images, for example, one can choose between two ways of doing this: first, by subtracting the mean image (image of size MxNx3) from the entire dataset or, to subtract the so-called per-channel mean (three numbers in total). The motivation behind this is illustrated in Figure XY. If we think of adjusting the weights in the network as moving the decision boundary, it is intuitive that the data not distributed around the origin will cause the classification success to get very sensitive to weight changes.

Sometimes it's also appropriate to scale the data so it has the same variance in all directions. See [mehlig chapter 6.3.1.] for more details and other techniques.

3.3. Convolutional Neural Networks

Convolutional Neural Networks (CNN) are very similar to Neural Networks from the previous chapter. They became widely used after Krizhevsky et al. [] won the ImageNet challenge with a CNN. One reason for the recent success of CNN is that they have fewer neurons. This has two advantages. Firstly, such networks are cheaper to train. Secondly, reducing the number of neurons regularises the network and reduces the risk of overfitting. CNN are trained with backpropagation as well as perceptrons.

3.3.1. CNN Layer Types

The fundamental blocks we developed for learning regular Neural Networks still apply here. CNN architectures make the explicit assumption that the inputs are images (usually of the size MxNx3 for RGB). Typical CNN architecture consists of layers that, in addition to the already presented principles, allow it to exploit the spatial and colour information encoded in the image.

Convolution Layers

In CNN, layer parameters consist of a set of learnable filters. Each filter is small spatially but extends through the full depth of the input volume. For example, a typical filter in the first layer of a CNN with RGB inputs has size 5x5x3. During the forward pass, we convolve each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. Intuitively, the network learns filters that activate when they see some type of visual features such as edges of certain orientation or a blotch of some colour in the first layer. Now, we have an entire set of filters in each CONV layer, and each of them will produce a separate 2-dimensional activation map (sometimes called feature map). Finally, we stack these activation maps along the depth dimension and produce the output volume that becomes an input for other layers. [https://cs231n.github.io/convolutional-networks/]

FIGURE AND MATHEMATICS

Pooling Layers

The function of pooling layers is to progressively reduce the spatial size of the layers in the network and thus reduce the number of parameters. [https://cs231n.github.io/convolutional-networks/] A neuron in a pooling layer takes the outputs of several neighbouring feature maps and summarises their outputs into a single number. Max-pooling units, for example, summarise the outputs of nearby feature maps (in a 2×2 square for instance) by taking the maximum over the feature-map outputs. There are no trainable parameters associated with the pooling layers, they compute the output from the inputs using a pre-defined prescription. [mehlig]

3.3.2. Examples of CNN Architectures

Most CNN architectures were developed for image classification. This is achieved by combining the properties of CNN and FCN (perceptrons). We see that in the deepest stage, the output of the network is followed by a standard multilayer perceptron with softmax output. The role of CNN here is only to encode the significant features of a particular image into a lower-level representation. The FCN then takes this output, literarly flattens the output tensor and learns to classify it.

There have been introduced various architectures, each of them having a different number of convolution layers, size of the filters, stride taken by the filters during convolution, etc. In practice, one rarely designs a CNN from scratch; instead, it is advisable to choose the currently best-performing network, usually one that performs best on the ImageNet challenge.

Here is a summary of the milestone architectures presented in recent years:

- **AlexNet**

  The first work that popularized Convolutional Networks in Computer Vision was the AlexNet []. The Network had very similar architecture to LeNet [], but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).

- **GoogLeNet**

  The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large number of parameters that do not seem to matter much.

- **VGGNet**

  The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end.

- **ResNet**

  Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special skip connections and heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network.

3.4. Semantic Segmentation

In semantic segmentation, one assigns a class to each pixel of an input image, unlike in the classification task, where one classifies the entire image. This section presents the most successful methods involving neural networks and supervised learning.

Segmentation has always been one of the most fundamental areas of computer vision. The classical approaches are mostly based on the standard signal processing theory and some of them can still be implemented and give satisfactory results. However, this applies only to a limited number of use cases, where the conditions are very close to idealistic/ideal and where the robustness of the algorithm is not crucial. To give an example of classical methods, one can refer to thresholding, region growing and mean-shift segmentation [coufal, vut]. More advanced methods using machine learning classification have also been introduced, such as TextonBoost [], TextonForest [] and Random Forest []. These algorithms have fallen out of favour due to the massive success of neural networks.

3.4.1. Encoder-Decoder Architecture

In the previous chapter, the CNN architectures designed for image classification were presented. The size of the output layer of these networks is determined by the number of categories of classification because the CNN transfers to a FCN in the end. In semantic segmentation, however, one needs to get an image of the same resolution as the input image containing the information about a class of every pixel. To do this, the common scheme is introduced: the first part of the network is left unchanged but now, instead of the transition to FCN, various methods are implemented to upsample the encoded image features from the deepest layer of the CNN. This scheme is referred to as the encoder-decoder architecture.

The purpose of the encoder is to downsample the input images while still representing the significant features. The decoder part of the algorithm then upsamples the output of the encoder to the original input image size. This is usually done by performing reverse operations to max-pooling and convolution. The last part of the decoder typically gives the final segmented image.

[DECONVOLUTION, stanford]

Shortly after the success of CNN in image classification challenges, there have been introduced several segmentation architectures using CNN as the encoder. Some of the state-of-the-art were, for instance, FCN [], DeconvNet [] and U-Net []. These networks share the idea of having CNN incorporated as the encoder but differ in the form of the decoder part. [] However, the problem of training such networks due to a large number of trainable parameters, the design of the decoder and hence the need of introducing the cumbersome multi-stage training made them very difficult to use in practice [segnet]. SegNet [] introduced in 2015 differs from these architectures as it has a significantly lower number of parameters and the design od the encoder-decoder network allows it to be trained via standard end-to-end method using backpropagation and SGD.

3.4.2. SegNet

SegNet is a deep encoder-decoder architecture for multi-class semantic segmentation researched and developed by members of the Computer Vision and Robotics Group at the University of Cambridge, UK. [https://mi.eng.cam.ac.uk/projects/segnet/]

The architecture consists of a sequence of encoders and a corresponding set of decoders followed by a pixel-wise Softmax classifier. Typically, each encoder consists of one or more convolutional layers with batch normalisation and a ReLU non-linearity, followed by max-pooling. SegNet uses max-pooling indices in the decoders to perform upsampling of low-resolution feature maps. The entire architecture can be trained end-to-end using stochastic gradient descent. [https://mi.eng.cam.ac.uk/projects/segnet/]

SegNet - Encoder

The architecture of the encoder network is topologically identical to the 13 convolutional layers in the VGG16 network [1]. Each encoder in the encoder network performs convolution with a filter bank to produce a set of feature maps. These are then batch normalized. Then an element-wise ReLU is applied. Following that, max-pooling with a 2×2 window and stride 2 is performed. Storing the max-pooling indices, i.e, the locations of the maximum feature value in each pooling window is memorized for each encoder feature map.

SegNet - Decoder

The appropriate decoder in the decoder network upsamples its input feature maps using the memorized max-pooling indices from the corresponding encoder feature maps. This step produces sparse feature maps, FIGURE XY. These feature maps are then convolved with a trainable decoder filter bank to produce dense feature maps. A batch normalization step is then applied to each of these maps. The high dimensional feature representation at the output of the final decoder is fed to a trainable soft-max classifier. The predicted segmentation corresponds to the class with maximum probability at each pixel.

3.4.3. Bayesian SegNet

Bayesian SegNet is a probabilistic variant of SegNet. It can predict pixel-wise class labels together with a measure of model uncertainty. This is achieved by Monte Carlo sampling with dropout at test time. The authors of the paper show that modelling uncertainty improves segmentation performance by 2-3 % compared to SegNet.

Monte Carlo Dropout

Monte Carlo dropout sampling allows to understand the model uncertainty of the result. As explained in Chapter XY, the standard weight averaging dropout proposes to remove dropout at test time and scale the weights proportionally to the dropout percentage. Monte Carlo dropout, on the other hand, samples the network with randomly dropped out units at test time. bayesian [13] [12]

It is important to highlight that the probability distribution from Monte Carlo sampling is significantly different from the 'probabilities' obtained from a softmax classifier. The softmax function approximates relative probabilities between the class labels, but not an overall measure of the model's uncertainty. [bayesian [13]].

4. Implementation and method

In this chapter, the original Caffe [] implementation of SegNet and Bayesian SegNet together with their simplified version SegNet Basic and Bayesian SegNet Basic will be tested on a custom dataset. Part of this will be evaluating the effect of various training hyperparameters, solvers, data augmentation techniques etc. This will also give the instructions on how to set up the software/hardware environment to run Caffe framework.

The Caffe code used in this section is available at [filip github].

4.1. CPU vs. GPU for Training ANN

CPU is the main processing unit of a computer. Current CPU's usually have 4 to 8 separate cores, which allows them to run several tasks in parallel. Graphics processing unit (GPU) was originally designed for performing only rendering computer graphics. Table XY gives and idea of how these two computational units differ in terms of the kind of task they're designed for. Note that CPU has much lower number of cores, but these run at high frequency and are very capable in terms of the instructions they perform. Therefore, CPU's are great for sequential tasks. On the other hand, GPU comprises of a large number of 'simple' cores, which makes it better for computing parallel tasks.

In terms of the available memory, CPU doesn't have its own resources (apart from the very small memory sections called caches) and has access to the system's RAM, whose size is very often between 8 and 32 GB for powerful PC's. GPU's, on the other hand, have their own block of RAM on the chip because the access top the main system's RAM has usually many bottlenecks. The size of the RAM for the top-end GPU's ranges from 8 to 12 GB.

The main part of the computations in Neural Networks in general is matrix multiplication. For this, GPU has the power of performing these operations by parts in parallel which speeds up the training significantly.

There have been created abstraction frameworks such as CUDA and OpenCL, that allow programmers do write their code in an usual manner and run it directly on GPU. For the purposes of Neral Networks, NVIDIA has developed a library of the most commonly used CUDA primitives named cuDNN.

Tensor Cores

Tensor Core is a special GPU feature offered by NVIDIA cards. It enables mixed-precision computing, dynamically adapting calculations to accelerate throughput while preserving accuracy. The latest generation expands these speedups to a full range of workloads. From 10x speedups in AI training with Tensor Float 32 data type, to 2.5x boosts for high-performance computing with floating point 64 (double precision). [nvidia site]

4.2. ANN frameworks

As the architecture and training of Neural Networks are getting more complicated, there is a room for programmers to make ANN frameworks such as Caffe, TensorFlow, and PyTorch as user friendly as possible. The idea of these software tools is to make a higher abstraction of the architecture of the network called computational graph. The user can therefore think of designing and training the network separately by applying an optimizer to the computational graph that represents the layers of the network.

Caffe is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. [berkeley caffe] The main difference from the other mentioned frameworks is that the user often doesn't need to write any code at all. The architecture of the network (the computational graph) is created in a .prototxt file, which is a standard text file in which one fills in the subsequent layers of the network in the desired order. Also, rather than having a optimizer object, one creates another .prototxt file that contains parameters such as the optimizer type (SGD, Adam, etc.), learning rate, momentum constant and others. After both of this files are created, the user runs Caffe computation from the command line. The core of the framework is written in C++. Pre-built binaries are called When the computation is started.

Caffe also has bindings for Python (CPW - Caffe Python Wrapper) and Matlab, which if very useful for evaluating the training statistics.

4.3. Setting up Environment for Caffe

4.3.1. Hardware configuration

The GPU used for the computations has been picked according to the most up-to-date benchmarks and recommendations found online [source]. When choosing GPUs in general, one needs to decide between ATI/AMD and NVIDIA chips. For this case however, NVIDIA is the choice because it's way more 'ANN-friendly' as it's offering more features specifically designed for ANN computations.

It is also advisable to use SSD in the PC configuration, because the data flow begins from reading the training data (images) from a storage, in this case from the computer's hard drive. Another way is moving the training data into RAM before the training is initiated [source, dalasi info].

Table XY shows the complete PC specifications used for training SegNet for the purposes of this thesis.

4.3.2. Software configuration

Operating System

The standard platform for running Caffe is Ubuntu, which is a Linux distribution from Cannonical based on Debian. The environment used is Ubuntu 18.04 LTS 64 bit. Is is important to let the Ubuntu installer download the latest updates, or, after the installation, invoke the update command to ensure that the most up-to-date packages will be installed. For this, one can call

```
$ sudo apt update
$ sudo apt upgrade
```

Enabling NVIDIA driver

Ubuntu 18.04 enables the default Nouveau graphics driver after installation. Before taking other steps, it is vital to disable the Nouveau driver and use NVIDIA instead in *Application menu -> Software & Updates -> Additional drivers -> Using NVIDIA driver metapackage from nvidia-driver-XYZ (proprietary, tested) -> Apply changes.* The driver version used is nvidia-driver-440.

[https://www.linuxbabe.com/ubuntu/install-nvidia-driver-ubuntu-18-04]

CUDA installation

CUDA version is determined by the version of cuDNN compatible with the used Caffe version, which is cuDNN 5.1 in this case. The corresponding CUDA version is CUDA 8.0. On Ubuntu 18.04, the procedure is the following:

- **Download CUDA 8.0 runfile.** Go to CUDA Legacy Releases and look for 'CUDA Toolkit 8.0 GA2 (Feb 2017)'. The standard .deb installer support only Ubuntu 16.04 LTS and therefore the installation must be performed via the runfile method. Navigate to Linux -> x86_64 -> Ubuntu -> 16.04 -> runfile (local) -> Base installer. Also, download the Patch file.

- **Perform the runfile installation of CUDA.** Open the Ubuntu Terminal (Ctrl+Alt+T) and run

```
$ cd /path/to/cuda_8.0.61_375.26_linux.run # Navigates to folder with
   CUDA
$ sudo chmod a+x cuda*  # Makes the cuda*.run executable
$ ./cuda*.run --tar mxvf  # Unpacks the .runfile content
$ sudo cp InstallUtils.pm /usr/lib/x86_64-linux-gnu/perl-base # Copy
   one of the extracted files to perl-base
$ sudo sh cuda_8.0.61_375.26_linux.run --override # Start the
   installation
  # The licence agreement
  $ accept
  # You are attempting to install on an unsupported configuration. Do
     you wish to continue?
  $ yes
  # Install NVIDIA Accelerated Graphics Driver for Linux-x86_64
     375.26?
  $ no
  # Install the CUDA 8.0 Toolkit?
  $ yes
  $ <press enter> (leave deafult location)
  # Do you want to install a symbolic link at /usr/local/cuda?
  $ yes
  # Install the CUDA 8.0 Samples?
  $ no
```

  After the installation is done, ignore the '***WARNING: Incomplete installation!' statement, because the NVIDIA driver is already installed.

Now run the CUDA 8.0 Patch 2 installation is a similar fashion:

```
$ sudo sh cuda_8.0.61.2_linux.run
```

- **Perform the post-installation actions.** The system needs to know the location of CUDA executables. The common way is to set these "PATH" variables in the current session of the Terminal. However, it's useful to add these permanently to '~/.bashrc' :

```
$ sudo gedit ~/.bashrc # Opens the .bashrc file in text editor
```

In the text editor, append the following two statements to the end of the file:

```
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64\
     ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

From this point, all newly opened Terminal sessions should have the paths set correctly.

cuDNN installation

The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. [https://developer.nvidia.com/cudnn]

- **Download cuDNN 5.1 for CUDA 8.0.** To get the appropriate cuDNN version for Caffe and CUDA 8.0, go to cuDNN Archive (requires login) and look for *Download cuDNN v5.1 (Jan 20, 2017), for CUDA 8.0 -> cuDNN v5.1 Library for Linux*. Extract the archive, navigate to the extracted folder and copy the files to the CUDA 8.0 installation folder:

```
$ tar -xf cudnn-8.0-linux-x64-v5.1.tgz
$ cd cuda
$ sudo cp -a include/cudnn.h /usr/local/cuda/include/
$ sudo cp -a lib64/libcudnn* /usr/local/cuda/lib64/
```

Setting up Python Editor

The scripts for evaluating SegNet performace are written in Python. It's advisable to use Pycharm Community Edition for an editor, because it offers a very convenient combination of GUI and the standard command line environment.

A good practice is to use Python virtual environment to easily maintain the required packages and to make the project transferable to another Linux PC. In Pycharm, we can do this in an active Pycharm project by going to *File -> Settings -> Project -> Project Interpreter -> <wheel icon on the right> -> Add.* The standard choice is the Virtualenv Environment. The Base interpreter location on a fresh Ubuntu installation is

'/usr/bin/python3.6'. When we click OK, Pycharm creates a 'venv' folder at the specified location including all package files we install.

When the virtualenv is configured properly, is will automatically be activated when we enter the Linux Terminal session by clicking on 'Terminal' located at the bottom bar of Pycharm. From this Terminal, we'll be launching all SegNet scripts and use it to install the required packages by calling 'pip3 install <package-name>'.

4.3.3. Building Caffe for SegNet

The Caffe code is an open-source software. The authors of the SegNet created a slightly modified version of Caffe (caffe-segnet) that supports special SegNet layer types (upsample, bn, dense_image_data and softmax_with_loss (with class weighting)).

In addition, since the original caffe-segnet supports just cuDNN v2, which is not supported for new pascal based GPUs, there's another version of caffe-segnet from [TimmoSaemannGithub] that supports cuDNN 5.1 and decreases the inference time by 25 % to 35 %. This version has therefore been selected for running SegNet. From this point on, the term 'Caffe' will be equivalent to 'caffe-segnet' in the text.

- **Install Caffe dependencies.** Caffe is available as a source code and therefore needs to be compiled on the target platform. For this, several steps need to be taken to ensure that all libraries are available during the build.

```
$ sudo apt install python3-opencv    # OpenCV, version 3
$ sudo apt-get install libatlas-base-dev  # Atlas BLAS library
$ sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev
   libopencv-dev libhdf5-serial-dev protobuf-compiler
$ sudo apt-get install libboost-all-dev  # Boost
$ sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev
$ sudo apt-get install python3-pip
$ sudo pip3 install protobuf
$ sudo apt-get install the python3-dev
```

- **Download Caffe (caffe-segnet-cudnn5) source code.** Go to Timmoe Saemann's Github repository and clone/download it.

- **Set the build configuration file.** The build is done via the 'make' command, which needs the 'Makefile.config' file to be present in the parent directory ('caffe-segnet-cudnn5-master'). This file contains the build options and needs to be configured properly. Fortunately, the correct form of 'Makefile.config' is part of this thesis and can be found in the Attachment XY.

- **Install gcc/g++ compliers.** The CUDA/cuDNN libraries used during the build are compatible only with gcc/g++ compilers of version 5. To install these, run:

```
$ sudo apt install gcc-5 g++-5
# Create symbolic links so CUDA can see the proper compiler binaries
$ sudo ln -s /usr/bin/gcc-5 /usr/local/cuda/bin/gcc
$ sudo ln -s /usr/bin/g++-5 /usr/local/cuda/bin/g++
```

- **Start the build.** Once the 'Makefile.config' file is located in the 'caffe-segnet-cudnn5-master', everything should be ready for the final step. Type these commands to initiate and test the Caffe build (don't forget to build pycaffe (Caffe Python Wrapper)):

```
make all -j4 # start build
make test -j4 # test build
make runtest # run Caffe and test it
make pycaffe # build pycaffe
```

[https://mc.ai/installing-caffe-on-ubuntu-18-04-with-cuda-and-cudnn/]

4.4. Image Annotation

In supervised learning, one needs to manually create the training data consisting of inputs and corresponding targets (in segmentation, ground truths). There's a variety of annotation tools available on the internet, both under commercial and free licenses.

Labelbox

Labelbox in a paid online annotation tool. The best feature of Labelbox is that it allows sharing the datasets with other users and therefore speed up the labeling significantly. Labelbox offers free access to the full version to students. When the labeling is finished, one needs to export the image/label pairs to a .JSON file. This file contains links to the annotated images that are stored online and it's necessary to download them separately (Labelbox is still in development, this is valid by the time of publishing the thesis). To automate this process, one can call the function *download()* from the *utilities.py* file (Attachment XY).

4.5. Setting up SegNet

Caffe implementation of a neural network typically consists of four *.prototxt* files: *train.prototxt*, *solver.prototxt*, *test.prototxt* and *inference.prototxt*. The train, test and inference files are almost identical except for a few differences in the very first/last layers. The train file is used together with the solver file to train the network: the network architecture is determined by the train file and the parameters for optimization reside in the solver file. The test file is used by Caffe when one needs to test the network periodically during training on a validation dataset.

The network files used in this section are available at [filip github].

4.5.1. Solver Settings

The solver file contains the optimization parameters. The detailed description of the parameters can be found in the original Caffe documentation on GitHub [odkaz na dokumentaci]. The description of the parameters used can be found in the snippet below.

```
// Training file
net: "/path/to/train.prototxt"
// Caffe GPU version
solver_mode: GPU
// Solver type
type: "AdaDelta"
// Initial learning rate, changes according to lr_policy
base_lr: 0.061
// Determines how the learning rate changes during training
lr_policy: "fixed"
// Show loss and accuracy every 'display' iterations
display: 130
// Max number of iteration. One iteration = a pass of one mini-batch
max_iter: 3000
// Regularization technique called Weight decay
weight_decay: 0.0005
// Saves the weights after 'snapshot' iterations
snapshot: 1000000
snapshot_prefix: "/path/to/snap"

// Testing parameters
test_initialization: false
test_iter: 1
test_interval: 100000000
```

4.5.2. Training

Input Layer and Input pre-processing

The train file begins with the DenseImageData. Images and labels are loaded as *.jpg* and *.png* files directly from the hard drive (there are more methods Caffe offers). The path to the *image_paths.txt* file containg the image/label paths is entered as the *source* parameter of the *DenseImageData*. This layer also specifies the size of the mini batch. The value is limited by the amount of memory GPU offers. When a larger size of the mini batch is needed, one solution Caffe offers is to specify the *iter_size* parameter in the Solver file. The total mini batch size in Caffe is always a result of $iter\_size \cdot batch\_size$. By default, the value of *iter_size* is set to 1.

The *shuffle* parameter in the DenseImageData layer determines whether the training dataset is shuffled after each epoch. This is usually desirable as it helps the optimization algorithm by bringing another stochasticity into the computation. The *mirror* parameter applies random mirrors to the input data and hence augments the dataset. If one needs to apply more complex data augmentation techniques, it's necessary to perform them separately and feed the DenseImageData layer with already processed images.

```
// The first layer in the network
name: "bayesian_segnet_train"
layer {
name: "data"
```

```
type: "DenseImageData"
top: "data"
top: "label"
dense_image_data_param {
  source: "/SegNet_navganti/data/custom/train_linux.txt"
  batch_size: 4
  shuffle: true
  mirror: true
  }
# Per-channel mean
transform_param {
  mean_value: 129   #B component
  mean_value: 126   #G
  mean_value: 126   #R
  }
}
```

The training images and their ground truths are stored as *.jpg* and *.png* files. The corresponding pairs are denoted as rows in the *image_paths.txt* file in the format

/path/to/image.jpg /path/to/label.png

This file is generated using the function *make_txt()* from *utilities.py*. The script will also make separate directories for training, testing and validation datasets by calling *make_dirs()*.

The method used for mean subtraction was the per-channel mean method. The function per_channel_mean in utilities.py calculates the mean values for R, G and B components from the training set. These three numbers are then placed into the DenseImageData layer in BGR order (see Snippet XY).

Output Dimensions

In the original version, SegNet segments 11 classes. This corresponds to the pixel values in the .png label files starting from zero: for instance, segmentation mask for class number 1 has a pixel value 0 in the label file, etc. However, the goal of this thesis is to set the network to segment only two classes - *path, background*. To change the size of the output classifier, it is necessary to change the output dimension of the last layer:

```
// The last conv layer in the network
layer {
  bottom: "conv1_2_D"
  top: "conv1_1_D"
  name: "conv1_1_D"
  type: "Convolution"
  .
  .
  .
  convolution_param {
    .
```

```
        .
        .
        num_output: 2  // Set this to the number of classes
        pad: 1
        kernel_size: 3
    }
}
```

Softmax Classifier

When there is large variation in the number of pixels in each class in the training set (e.g road, sky and building pixels dominate the dataset) then there is a need to weight the loss differently based on the true class. This is called class balancing. The authors of SegNet use median frequency balancing [odkaz] where the weight assigned to a class in the loss function is the ratio of the median of class frequencies computed on the entire training set divided by the class frequency. This implies that larger classes in the training set have a weight smaller than 1 and the weights of the smallest classes are the highest. When no re-weighting is applied, we talk about natural frequency balancing. [doslovna citace??? segnet paper]

```
// The Softmax classifier with cross-entropy loss
layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "conv1_1_D"
    bottom: "label"
    top: "loss"
    softmax_param {engine: CAFFE}
    loss_param: {
        weight_by_label_freqs: false
    }
}
// The last layer of the network
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "conv1_1_D"
    bottom: "label"
    top: "accuracy"
    top: "per_class_accuracy"
}
```

SegNet uses the cross-entropy loss as the loss function for training the network. In Caffe, median frequency balancing is available via the 'weight_by_label_freqs' parameter in the *SoftmaxWithLoss* layer. Since the dataset used has only two classes whose occurence can be considered as balanced, this option is set to *false*.

Training Initialization

The tarining is initiated by entering these commands:

```
# Navigate to the caffe-segnet folder
$ cd /SegNet/caffe-segnet/build/tools/
# Initiate training from scratch
$ ./caffe train -solver /SegNet/Models/segnet_solver.prototxt
# Resume training from a solver checkpoint (snapshot)
$ ./caffe train -solver /path/to/solver.prototxt -snapshot
    /path/to/snapshot_iter_XY.solverstate
```

The encoder and decoder weights are all initialized using MSRA method by default. Another scenario is when it's desired to use transfer learning. In this case, Caffe needs a path to the weight file of the pre-trained network. The corresponding command would be

```
$ ./caffe train -solver /path/to/solver.prototxt -weights
    /path/to/pre_trained_weights.caffemodel
```

[https://arxiv.org/pdf/1411.4734.pdf]

4.5.3. Testing

4.5.4. Inference

In this phase, the network is ready to be deployed. From this point, it's very convenient to use the CPW for running the network, feeding it with input data and calculating the segmentation accuracy. To run the final segmentation, some preparation steps must be taken first.

Calculating Batch Statistics

The Batch Normalisation layers [3] in SegNet shift the input feature maps according to their mean and variance statistics for each mini batch during training. At inference time we must use the statistics for the entire dataset and obtain the final weights for the inference phase. We do this by calling the *compute_bn_statistics.py* which is meant to be run from the command line and needs to get command-line parameters. In PyCharm, we need to switch to Virtual Environment (venv) by opening Terminal and call:

```
(venv) user@user:/path/to/Scripts$ python3 original_compute_bn_statistics.py
    /path/to/train.prototxt /path/to/snap_iter_XY.caffemodel
    /path/to/inference_folder
```

The script automatically edits the TRAIN file and turns it into a new INFERENCE file by removing the layers that are no longer needed. It also takes the weights specified in *snap_iter_XY.caffemodel*, calculates the final weights and saves them to *final_weights.caffemodel*. Both the INFERENCE and weigh files are stored in the specified *inference_folder*.

The network architecture is now in the INFERENCE file and is the same as in the TRAIN file, apart from the input and output layers. The snippet below shows how the output changes: the loss function is no longer computed, the only output we care about are the Softmax probabilities. The DenseImageData layer is also skipped, because the data will be provided via CPW.

```
// Inference, input layer
name: "segnet_inference"
input: "data"
input_dim: 1 # Always 1 for SegNet
input_dim: 3
input_dim: 360
input_dim: 480
```

Running the Segmentation

For running the segmentation, we use the script *segnet_inference.py*. We must provide the network with images either by specifying a path to a video file, or by specifying a sequence of image names to follow in a folder (this is a standard OpenCV [odkaz na to jakej to ma mit format] convention). Once we have an appropriate test set of images, we run the segmentation by calling:

```
(venv) user@user:/path/to/Scripts$ python3 segnet_inference.py
    /path/to/inference.prototxt /path/to/final_weights.caffemodel
    /path/to/videofile.avi
```

Evaluating Segmentation Performance

The performance of semantic segmentation is often described by so called IoU (intercestion over union) metrics. IoU is the area of overlap between the predicted segmentation and the ground truth divided by the area of union between the predicted segmentation and the ground truth, as shown in Figure XY. This metric ranges from 0–1 (0–100%) with 0 signifying no overlap and 1 signifying perfectly overlapping segmentation.

[https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation-model-6bcb99639aa2]

[https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/]

http://mi.eng.cam.ac.uk/projects/segnet/tutorial.html

https://github.com/alexgkendall/caffe-segnet/issues/21

4.6. Setting up Bayesian SegNet

Since Bayesian SegNet differs from SegNet only in terms of added dropout layers and slightly different method of performing network inference, the above-mentioned procedures for setting the solver and training are applicable in the same way. One can therefore start the training by using the commands from the previous section and only replace the file paths of the TRAIN and SOLVER files. The TEST and INFERENCE files have only one major difference in the input layers.

Unlike in SegNet, the *batch_size* parameter in the DenseImageData layer of the TEST file now represents the number of Monte Carlo Dropout samples used for output averaging, as described in Section XY. The same corresponds to the first dimension of the input layer in the INFERENCE file.

After calling the script *compute_bn_statistics.py* on a trained Bayesian SegNet, we can start the inference by executing:

```
(venv) user@user:/path/to/Scripts$ python3 bayesian_segnet_inference.py
    /path/to/inference.prototxt /path/to/final_weights.caffemodel
    /path/to/videofile.avi
```

Here the scripts also visualizes the statistics of Monte Carlo sampling: the uncertainty and variance of the output segmentation.

4.7. SegNet Basic and Bayesian SegNet Basic

These shallow versions of SegNet and BayesianSegNet are used in the same way as their full versions above. The same procedures apply to SegNet/SegNet Basic and Bayesian SegNet/Bayesian SegNet Basic.

5. Results

ODECISTP PER CHANNEL MEAN PRO INFERENCI
bayesian takes longer to train
TRAINING STRATEGIES
TRANSFER LEARNING + BATCH NORMALIZATION
————TRAINING CAFFE NOTES————

We perform local contrast normalization [54] to the RGB input. We train the model with dropout and sample the posterior distribution over the weights at test time using dropout to obtain the posterior distribution of softmax class probabilities. We take the mean of these samples for our segmentation prediction and use the variance to output model uncertainty for each class. We take the mean of the per class variance measurements as an overall measure of model uncertainty. We also explored using the variation ratio as a measure of uncertainty (i.e. the percentage of samples which agree with the class prediction) however we found this to qualitatively produce a more binary measure of model uncertainty. Fig. 2 shows a schematic of the segmentation prediction and model uncertainty estimate process.

6. Discussion and Future Work

7. Bibliography

8. Seznam použitých zkratek a symbolů

CMU	Carnegie Mellon University

9. Seznam příloh

- Nastavení režimu External mode: *external.txt*