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

Tato práce se zabývá rešerší a implementací vybraných architektur konvolučních neuronových sítí pro segmentaci obrazu. První části jsou shrnuty základní pojmy z teorie neronových sítí. Tato část také představuje silné stránky konvolučních sítí v oblasti rozpoznávání obrazových dat. Teoretická část je uzavřena rešerší zaměřenou na konkrétní architekturu používanou na segmentaci scén. Implementace této archiktury a jejích variant v Caffe je převzata a upravena pro konktrétní použití v praktické části práce. Nedílnou součástí tohoto procesu jsou kroky potřebné ke správnému nastavení softwarového a hardwarového prostředí. Příslusná kapitola proto poskytuje přesný návod, který ocení zejména noví uživatelé Linuxových operačních systémů. Pro trénování všech variant vybrané sítě je vytvořen vlastní dataset obsahující 2600 obrázků. Je také provedeno několik nastavení původní implementace, zvláště pro účely použití předtrénovaných parametrů sítí. Trénování zahrnuje výběr hyperparametrů, jakým je například typ optimalizačního algorithmu. Na závěr je provedeno vyhodnocení výkonu a výpočtové náročnosti všech natrénovaných sítí na testovacím datasetu.

Summary

This thesis deals with the research and implementation of selected architectures of Convolutional Neural Networks (CNNs) for image segmentation. In the first part, the fundamental terms from the theory of neural networks are summarized. It also presents the power of CNNs in the field of image data classification. The theoretical part finishes with the reseach focused on the particular network architecture and its variants used for scene segmentation. In the practical part, the Caffe implementation of the network introduced is retaken from the authors and tailored to a specific use case. The steps required to properly set up the software and hardware environments are the essential part of the process. Therefore, the corresponing chapter gives a step-by-step guide that is especially helpful to new Linux OS users. A custom dataset containing 2600 segmented images is created and used for training all variants of the selected network. Several adjustments of the original implementation are performed, especially for applying the method of using pre-trained parameters of the networks. The training phase includes the selection of hyperparameters, such as the type of the optimization algorithm. Finally, the performance and computational cost of the variants of the trained network is evaluated on a testing dataset.

Klíčová slova

sémantická segmentace, konvoluční neuronové sítě, SegNet, Caffe, Ubuntu

Keywords

semantic segmentation, convolutional neural networks, SegNet, Caffe, Ubuntu

ŠPILA, F.*Semantic segmentation of images using convolutional neural networks.* Brno: Vysoké učení technické v Brně, Faculty of Mechanical Engineering, 2020. 57 s. Vedoucí doc. Ing. Jiří Krejsa, Ph.D.

Prohlašuji, že jsem svou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, software atd.) citované v práci a uvedené v přiloženém seznamu a postup při zpracování práce je v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) v platném znění.

Bc. Filip Špila

Děkuji vedoucímu své práce panu doc. Ing. Jiřímu Krejsovi, Ph.D. za skvělou spolupráci a rychlost s jakou odpovídal na mé dotazy. Poděkování dále patří všem blízkým lidem z mého okolí, kteří mě podporovali během studií svou trpělivostí a pochopením. Bez některých z nich (...) bych dokonce nebyl schopen úspěšně dokončit svůj zahraniční pobyt v rámci programu Erasmus, jenž má přesah nejen do stránek této práce, ale i do ostatních aspektů v mém životě. Na závěr děkuji pánům Švandovi a Szmidtovi za jazykové korektury a kapele AC/DC za navození příjemné atmosféry pří psaní.

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 as robustly and reliably as possible.

We can see a real-world example in Figure 1.1. 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 image classification task where we classify the image scene as a whole. It is important 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 [12]. For instance, all people in Figure 1.1 (each instance of the 'person' class) would have a different colour.

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 developments show increasing demand for reliable object recognition in self-driving cars because the driving models must understand the context of the environment they are operating in. [1]

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 received 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 them as the first stage of the segmentation algorithm.

Figure 1.1: Segmentation of an urban road scene [2]

2. Problem statements

The goal of this thesis consists of several points. 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 being 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 them to create the final training and validating datasets. Lastly, the network should be trained with various sets of training parameters to get a better idea of the network's behaviour and to ensure the best possible results.

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

3. Research and theory

The first part of this chapter gives an introduction to artificial neural networks (ANNs). It begins with a definition of fundamental terms that explain the core principles of ANN. Because the research in this area is still 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 that will be relevant for the network architecture chosen at the end.

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

3.1. Architecture of artificial neural networks

The inspiration for neural networks comes from their resemblance to biological neurons and the way they are connected. Neural networks can recognize features in a given training set of data and apply this knowledge to previously unseen data after the training. This strategy is called supervised learning. In supervised learning, one periodically feeds the network with input/output pairs of training data. The network learns by comparing the correct and computed output values for the given input. The network's 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. [3]

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 [4]

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

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 [4]

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

where L is the 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. 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: [4]

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

These models are referred to as feed-forward because information flows from the deepest nested function $f^{(1)}$ which then takes 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, $f^{(2)}$ as the second layer and so on. The final layer of the network is called the output layer. [4]

Remember that in supervised learning one needs a set of training data, in this case a set of matching x, y[1] pairs. The training samples specify 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'. [4]

A neural network can be seen as something capable of modelling almost any function we can think of (general approximation theorem, see [5]). The power of this brings us to the definition of a classification task. In this task, the function which the network approximates has discrete states (true/false in the simplest case).

3.1.2. McCulloch-Pitts neurons

Layers of a feed-forward network further divide into distinct functions called neurons. This is where the resemblance to biological neurons comes into play: the neurons are mathematically modeled as linear threshold units (McCulloch-Pitts neurons). The output of a neuron is dependent on the output of the neurons in the previous layer. In the simplest form, the output of each neuron in the network has only two states: active or inactive. [3]

If the output exceeds a given threshold then the state of the neuron is said to be active, otherwise it is inactive. The model is illustrated in Figure 1.4. Neurons usually perform repeated computations in discrete time steps $t = 0, 1, 2, 3, ...$ The state of neuron number j at time step t is denoted by [3]

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

Given the signals $n_j(t+1)$, neuron number i computes [3]

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

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 $t + 1$.

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

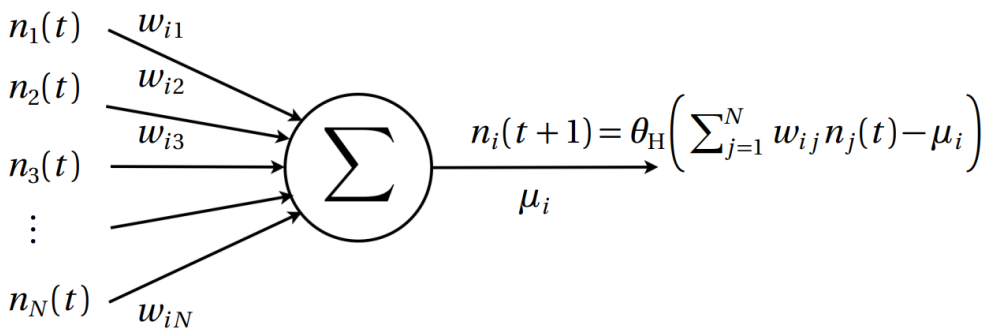


Figure 3.1: Schematic diagram of a McCulloch-Pitts neuron. The strength of the connection from neuron j to neuron i is denoted by w_{ij} [3]

Each incoming connection from other neurons has a different strength. This is determined by the parameters w_{ij} called weights . The first index i refers to the neuron whose output is being computed and j labels all neurons that connect to neuron i. The argument of θ_H of the neuron is often referred to as the local field [3]

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

where b_i is a weighted linear average of the inputs n_j and μ_i is an offset (threshold). Finally, the function θ_H is referred to as the activation function. [3]

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 [3]

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

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 [6]. Therefore, the term θ_H in Eq. (3.5) is replaced by a general continuous activation function $g(b)$. [3]

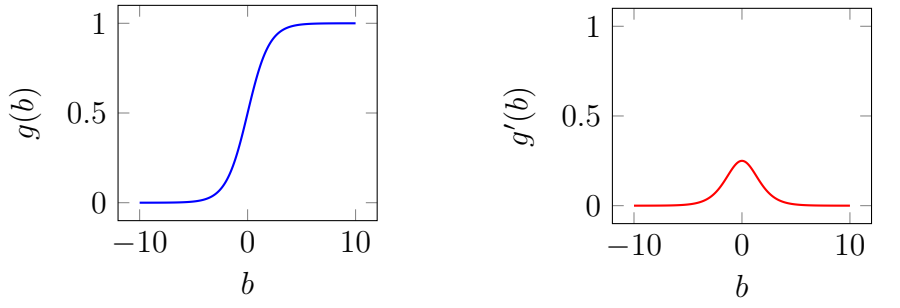
One can choose from several activation functions which all come with their pros and cons depending on the particular application of the network. In general, there are a few requirements these functions should meet: [6]

- **Nonlinearity**. As discussed above, non-linearity is a general ability of a neural network which allows it to model very complex functions.
- **Monotocity and nondecreasibility**. These allow certain optimization algorithms to perform with greater stability.
- **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, the commonly-used softmax activation function can be used. 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 are technically not. [7]

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: [6]

Sigmoid

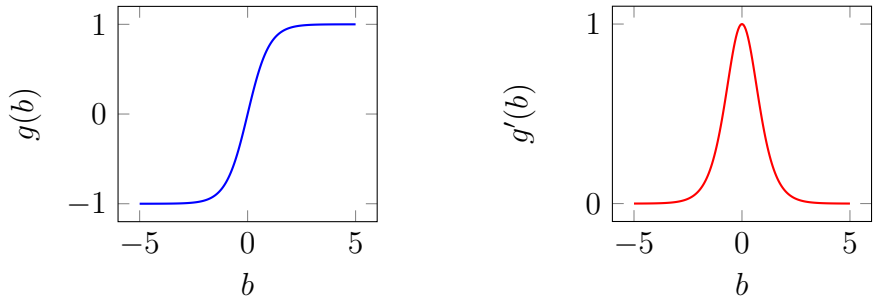


(a) $g(b) = \frac{1}{1+e^{-b}}$ **(b)** $g'(b) = g(b)(1 - g(b))$

Figure 3.2: Sigmoid function and its derivative. Notice that the derivative goes to zero very quickly.

This function has a clear interpretation of neuron states - active/inactive is represented by values 1/0. The sigmoid function is currently not favoured 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. [7] [6]

Hyperbolic tangent

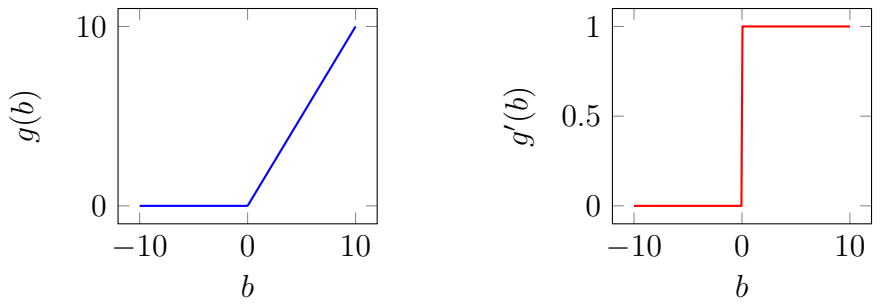


(a) $g(b) = \tanh(b)$ **(b)** $g'(b) = 1 - \tanh^2(b)$

Figure 3.3: Hyperbolic tangent and its derivative.

Unlike the sigmoid function, 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. [7]

Rectified linear unit (ReLu)

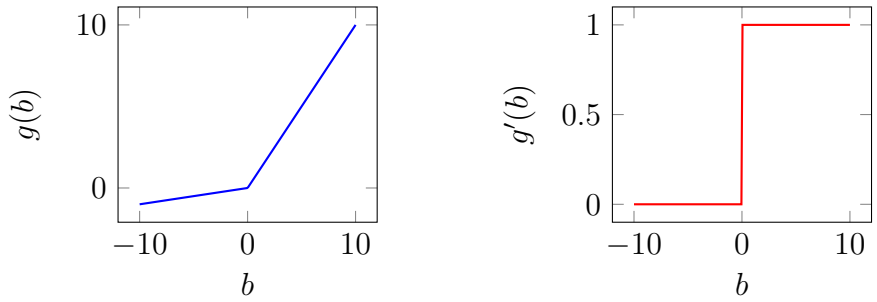


(a) $g(b) = \max(0, x)$ **(b)** $g'(b) = \theta_H(x)$

Figure 3.4: ReLu and its derivative. ReLu does not saturate!

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 the figure above. 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. [3]

Leaky ReLu



(a) $g(b) = \begin{cases} b & \text{for } b \geq 0, \\ 0.01b & \text{for } b < 0. \end{cases}$ **(b)** $g'(b) = \begin{cases} 1 & \text{for } b \geq 0, \\ 0.01 & \text{for } b < 0. \end{cases}$

Figure 3.5: Leaky ReLu and its derivative.

By modifying the previously introduced function one gets a version of ReLu intended to address its biggest drawback, which is the fact that some neurons may become dead (their output will be always zero) and thus they do not contribute to the network's output. Unfortunately, there's generally no guarantee that using Leaky ReLu instead of ReLu will always yield better results. [8]

3.1.4. Multilayer perceptrons

Perceptron is a feed-forward network. It is divided into layers consisting of McCulloch-Pitts neurons. The left-most layer of the network shown in Figure 3.6 is called input layer. The input layer takes the values of the input data and passes it to the next layer. The right-most layer is the output layer where the output of the network is read out. The other neuron layers are called hidden layers; their states are not read out directly. [3]

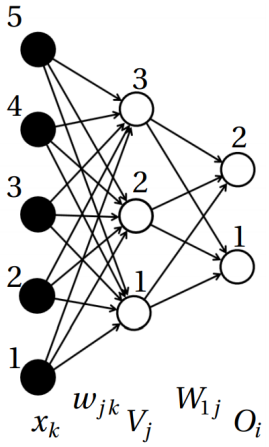


Figure 3.6: Perceptron with one hidden layer. [3]

"*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.*" [3] We denote the inputs coming to the input layer by [3]

$$x(\mu) = \begin{bmatrix} x_1^{(\mu)} \\ x_2^{(\mu)} \\ \vdots \\ x_N^{(\mu)} \end{bmatrix} \tag{3.8}$$

The index μ labels different input patterns in the training set. The perceptron in Figure 3.1 calculates the output as follows: [3]

$$V_j^{(\mu)} = g(b_j^{(\mu)}) \quad \text{where} \quad b_j^{(\mu)} = \sum_k w_{jk} x_k^{(\mu)} - \theta_j \tag{3.9}$$

$$O_i^{(\mu)} = g(B_i^{(\mu)}) \quad \text{where} \quad B_i^{(\mu)} = \sum_j W_{ij} V_j^{(\mu)} - \Theta_i \tag{3.10}$$

Here $V_j^{(\mu)}$ denotes 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. Corresponding computations are made for the output layer whose parameters are denoted by capital letters. [3] A multilayer perceptron generally has N hidden layers. If it has more than two hidden layers, it usually begins to be called a deep network.

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 [3]

$$O_i = \frac{e^{\alpha b_i^{(L)}}}{\sum_{k=1}^{M} e^{\alpha b_k^{(L)}}} \tag{3.11}$$

"*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* L*. 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* $\sum_{i=1}^{M} O_i = 1$*. 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.*" [3]

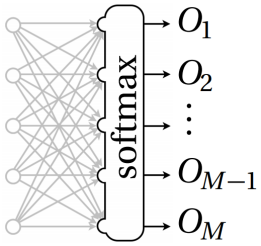


Figure 3.7: Softmax classifier: the neurons in this layer are not independent. [3]

Linear separability

The reason we use hidden layers is to tackle classification problems that are not linearly separable. Linear separability is shown in Figure 3.8, where the input to the network is two-dimensional. We classify the input data into two classes (marked as black and white points in the graph). [3]

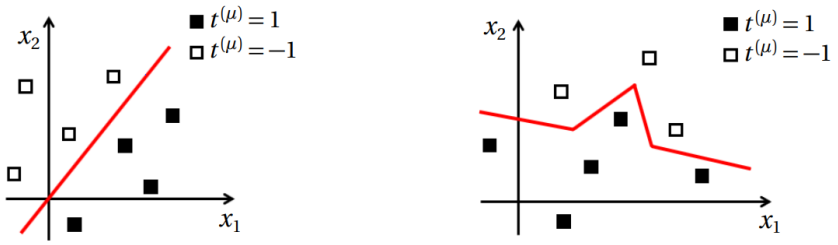


Figure 3.8: Linearly separable (left) and not linearly separable problems (right). The decision boundary needs to be piece-wise linear for the not linearly separable problem [3]

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. The curve that separates the space of inputs 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 3.8 (left), the line dividing the 2D space of inputs corresponds to the simplest possible case which is a single neuron in the network. In a not linearly separable task (Figure 3.8, right) we need to divide the input space into more than two regions to solve the classification. 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. [3]

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. To achieve this, trainable paremeters are changed in each step of optimization. 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. [39]

3.2.1. Loss function

Loss function is a metric of our satisfaction with the network's output. The choice depends on the nature of the task that 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 functions: [3]

Mean Squared Error (MSE)

$$L = \frac{1}{2} \sum_{\mu i} \left(t_i^{(\mu)} - O_i^{(\mu)} \right)^2 \tag{3.12}$$

MSE is used for regression tasks, often in combination with the sigmoid function in the output layer. [6]

Negative Log Likelihood

$$L = -\sum_{\mu i} t_i^{(\mu)} \ln(O_i^{(\mu)}) \tag{3.13}$$

The negative log likelihood is used for classification tasks in combination with the softmax classifier. [3]

Cross Entropy Loss

$$L = -\sum_{\mu i} t_i^{(\mu)} \ln(O_i^{(\mu)}) + (1 - t_i^{(\mu)}) \ln(1 - (O_i^{(\mu)})) \tag{3.14}$$

Very similar to the negative log likelihood loss. The difference is that it works with the sigmoid activation function. [3]

3.2.2. Gradient optimization and backpropagation

Backpropagation is a way in which information about the correctness of the output flows through the network so that the parameters in all layers can be adjusted. Everytime we feed the network with an input pattern μ we get the output values of the neurons in all layers. This is called a 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 backpropagation because the error propagates from the output layer to the layers on the left. [3]

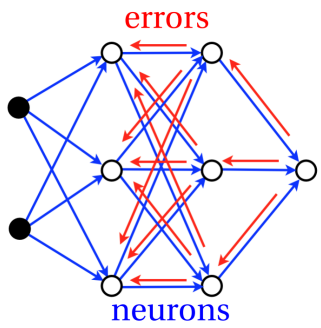


Figure 3.9: Backpropagation algorithm: the states of the neurons are updated forward (from left to right) while errors are updated backward (right to left) [3]

The optimization algorithm searches the most optimal value of the loss function whose value is dependent on the trainable parameters. For this, the algorithms needs to move in the direction of the steepest descent in the landscape of the loss function. In each step of the optimization, one needs to calculate partial derivatives of the loss function with respect to all trainable parameters. The derivative is found by applying the chain rule to the formula for calculating the loss function. [3]

Gradient descent

The general formula for the gradient descent algorithm goes as follows: [39]

$$\delta\phi = -\eta\frac{\partial L}{\partial \phi} \tag{3.15}$$

where ϕ is the parameter we care about (weights, thresholds, etc.) and L is the loss function. Parameter η is called the learning rate. This parameter determines the size of the step we take in the way of the steepest descent in the loss function's landscape (in the case of two parameters). [39]

Figure 3.10 shows that the choice of the learning rate value has a strong effect on the course of the optimization and the convergence of the algorithm. If the steps are too small, the training will be slow and the algorithm is prone to getting stuck in local minima. 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. [3]

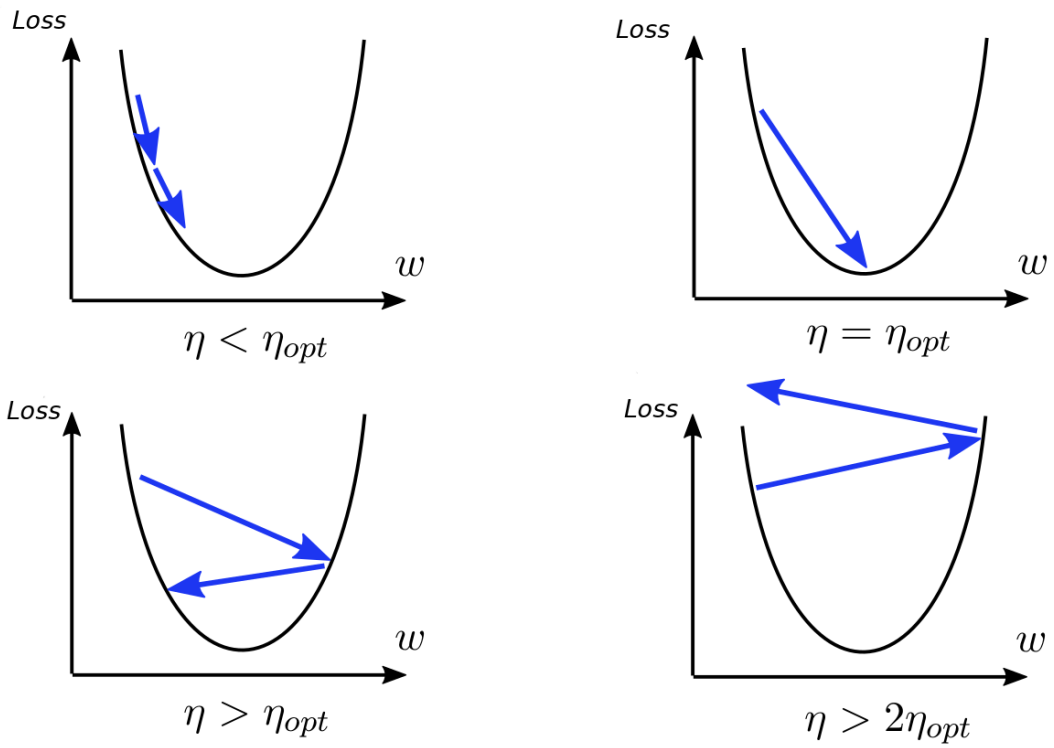


Figure 3.10: Effect of the learning rate on optimization: the value must be chosen carefully for the algorithm to converge. [13]

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 [3]

$$\delta W_{mn} = -\eta \frac{\partial L}{\partial W_{mn}} = \eta \sum_{\mu=1}^{p} (t_m^{(\mu)} - O_m^{(\mu)}) g'(B_m^{(\mu)}) V_n^{(\mu)} \tag{3.16}$$

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 [3]

$$\Delta_m^{(\mu)} = (t_m^{(\mu)} - O_m^{(\mu)}) g'(B_m^{(\mu)}) \tag{3.17}$$

The update rules for hidden layers are also obtained by using chain rule, which gives [3]

$$\delta w_{mn} = -\eta \frac{\partial L}{\partial w_{mn}} = \eta \sum_{\mu=1}^{p} \sum_{i=1}^{N} \Delta_i^{(\mu)} W_{im}\, g'(b_m^{(\mu)}) x_n^{(\mu)} \tag{3.18}$$

while putting [3]

$$\delta_m^{(\mu)} = \sum_{i=1}^{N} \Delta_i^{(\mu)} W_{im}\, g'(b_m^{(\mu)}) \tag{3.19}$$

Putting all the above together gives [3]

$$\delta w_{mn} = \eta \sum_{\mu=1}^{p} \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta W_{mn} = \eta \sum_{\mu=1}^{p} \Delta_m^{(\mu)} V_n^{(\mu)} \tag{3.20}$$

Similarly, we get the update rule for thresholds (see [3]). In summary, the steps of backpropagation + gradient descent are the following: [3]

Algorithm 1 Gradient descent [3]

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

Gradient methods are generally prone to getting stuck in local minima of the optimized function. The way to adress this is to add a little bit of noise to the process. In stochastic gradient descent, this is achieved by summing over smaller portions of the training data rather than over the entire dataset. These portions of the data are called mini-batches. [3]

In Equations 3.20 we see that in each iteration one needs to sum overall 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 is called a training epoch). In mini-batches, samples appear only once per epoch and the entire training set is usually shuffled after each epoch. [3] Applying the above, the Equations 3.20 slightly change to [3]

$$\delta w_{mn} = \eta \sum_{\mu=1}^{mb} \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta W_{mn} = \eta \sum_{\mu=1}^{mb} \Delta_m^{(\mu)} V_n^{(\mu)} \tag{3.21}$$

Vanishing and exploding gradient problems

When we compute the weight increments using MSE, the further from the output layer we go, the more the term $g'(b)$ accumulates (with each next layer). The point is that MSE is often used with the sigmoid activation functions whose derivative drops to a small number in its area of saturation resulting in very small weight increments. This phenomenon is known as the vanishing gradient problem [3]. 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. [14] One of the ways to address these problems is using activation functions that do not saturate (ReLu, Leaky ReLu, etc.). [3] [8]

Momentum

There are several ways to make the stochastic gradient descent algorithm perform better. The key is to prevent it from getting stuck in local minima. 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. This can, however, make the algorithm oscillate. [3] One way to tackle this is to implement the mechanism fittingly called momentum.

When using momentum, we can imagine that the SGD algorithm behaves like a ball that rolls downhill and develops speed over time [10]. 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 [3]

$$\delta w_{ij}^{(t)} = -\eta \frac{\partial L}{\partial w_{ij}^{(t)}} + \alpha \delta w_{ij}^{(t-1)} \tag{3.22}$$

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

There are other ways of implementing momentum, such as the commonly used Nesterov's accelerated gradient method (see [3] [7] 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 method performs better in practice [10].

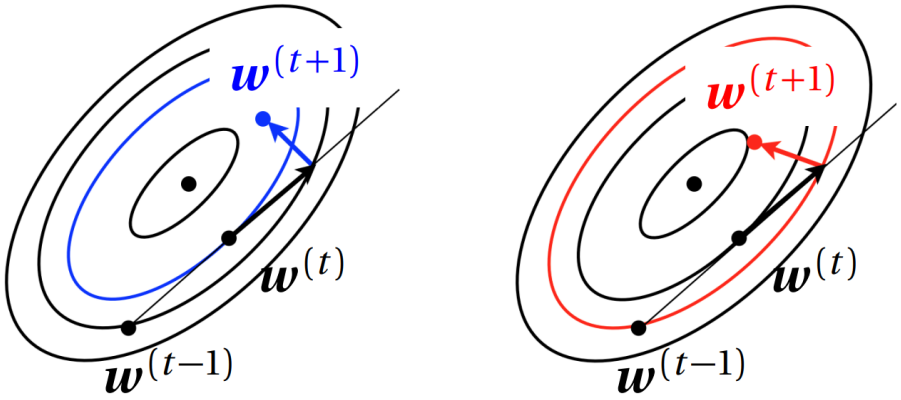


Figure 3.11: Momentum (left) and Nesterov's Momentum (right). [3]

Other optimization algorithms

The algorithms below extend the idea of stochastic gradient descent by introducing various strategies of learning rate adaptation during training. In most cases, using more advanced algorithms tends to speed up the training and usually helps finding more optimal parameters in terms of the loss value.

- **AdaGrad**

  AdaGrad is another gradient based algorithm. In the 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 [10]). 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. [10]

- **AdaDelta and RMSprop**

  These algorithms are an extension of AdaGrad and tackle its tendency to drop some of the learning rates to almost infinitely small values. They were published simultaneously but independently of one another. [6]

- **Adam**

  Adam can be seen as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using a moving average of the gradient instead of the gradient itself like SGD with momentum. [15] [6]

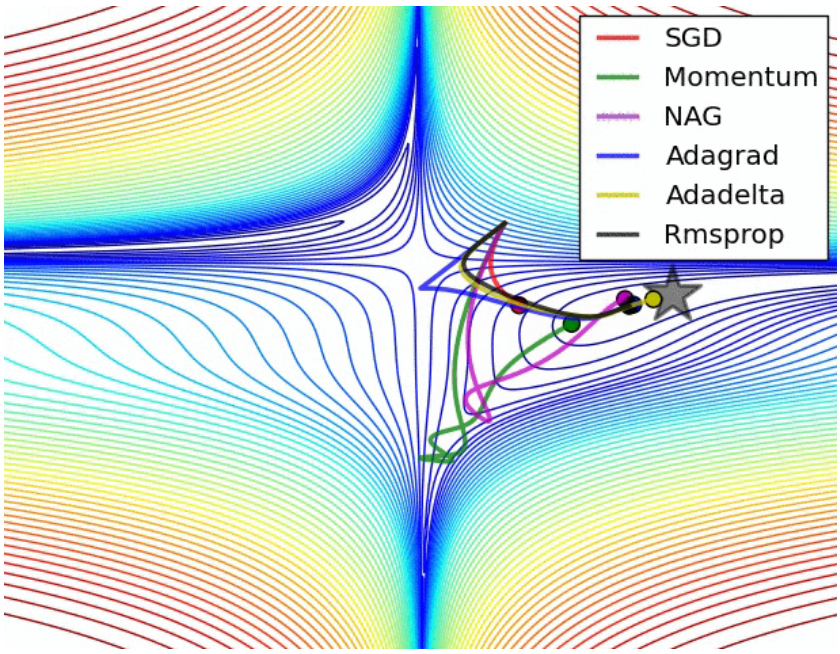


Figure 3.12: Comparison of different optimization algorithms. [6]

3.2.3. Improving training performance

Initialization of weights and thresholds

The standard approach is to initialise the weights to independent Gaussian random numbers with mean zero and unit variance and to set the thresholds to zero. But in networks that have large hidden layers with many neurons, this scheme may fail. 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. [3] Here are some of the more advanced initialization methods:

- **Xavier initialization**

  Xavier initialization sets the layer's weights to values from the 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. [9]

- **MSRA initialization**

  This method differs from Xavier only in its use of a different factor to scale the Gaussian distributed numbers. It turns out that this small change works much better when using ReLU activation function. [9]

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. 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. This problem is called overfitting: the network fits too fine details that have no general meaning.*" [3] The terms below are referred to as the L1 and L2 regularizations. Adding these terms to the loss function prevents the weight from growing. When the value of the weights gets very high, the local fileds of the neurons become very large too. In that case, some activation functions, like the sigmoid function or *tanh*, reach their maxima very quickly which causes the vanishing gradient problem. The formulas for L1 and L2 regularizations are: [3]

$$R_{L2}(w) = \frac{\gamma}{2}\sum_{ij} w_{ij}^2 \quad \text{or} \quad R_{L1}(w) = \frac{\gamma}{2}\sum_{ij} |w_{ij}| \tag{3.23}$$

"*These two regularization schemes tend to help against overfitting. (...) Weight decay dds a constraint to the problem of minimising the energy function. 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, it may happen that small changes in the input give significant differences in the training result that are difficult to generalise.*" [3]

Batch Normalisation

The idea of batch normalisation is to shift and normalise the input data for each hidden layer so that the distribution of its inputs becomes Gaussian. The values of mean and variance are computed during each forward pass (pass of a single mini-batch) and then applied to each neuron in the layer. The mean and variance are multiplied by trainable factors, usually called β, γ. [9] [3] When the training is done, the values of β, γ for each layer are re-computed using the mean and variance of the entire training dataset and no longer change. [16]

"*Batch normalisation helps to combat the vanishing-gradient problem because it prevents local fields of hidden neurons to grow. This makes it possible to use sigmoid functions in deep networks, because the distribution of inputs remains normalised. (...) It is an empirical fact that batch normalisation often speeds up the training.*" [3]

Dropout

Dropout is a very simple scheme that helps against overfitting. During training, a random portion of neurons in the network is 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 being dropped out during training (weight averaging). [3] [10]

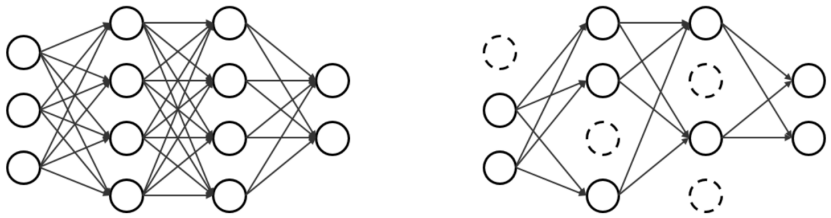


Figure 3.13: ANN without (left) and with dropout (right). [17]

Data augmentation

The general rule is that the bigger the training dataset, the better the network generalises. However, expanding a dataset manually can 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. [3]

Early stopping

"*One way of avoiding overfitting is to use cross validation and early stopping. One splits the training data into two sets: a* ***training set*** *and a* ***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 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." [3]

When the training is done, the performance is measured using a set of 'unseen' data: the **test set**. [3]

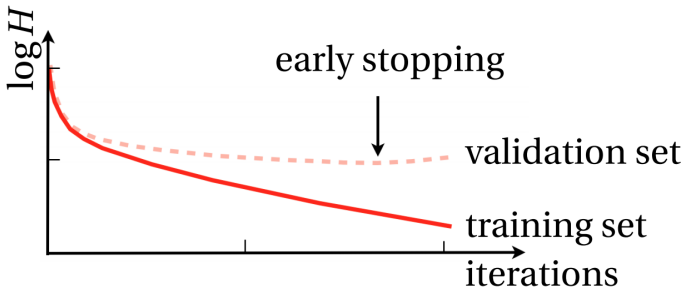


Figure 3.14: Progress of training and validation losses. The plot is schematic, and the data is smoothed. The training is stopped when the validation energy begins to increase. [3]

Transfer Learning

To create a well-generalising neural network, one need to have access to a dataset of a sufficient size. Therefore in practice, it is unusual to train an entire CNN from scratch (with random initialization). Instead, it is common to pretrain a CNN on a very large dataset (e.g. ImageNet) and then use the CNN either as an initialization or a fixed feature extractor for the task of interest. [10]

One strategy here is to fine-tune the weights of the pretrained network by continuing backpropagation. It is possible to keep some of the earlier layers fixed and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a CNN contain more generic features (e.g. edge detectors) and the later layers become progressively focused on the details. It is common to use a smaller learning rate for CNN weights that are being fine-tuned, in comparison to the randomly-initialized weights. This is because we expect that the CNN weights already perform well and hence distorting them is not desirable. [10] [7]

Data pre-processing

For most cases, it is advisable to shift the data so it has a zero mean before the training begins. When classifying images, for example, there are two ways of doing this: first, by subtracting the mean image (image of size MxNx3 for RGB) from the entire dataset or, to subtract the so-called per-channel mean (three numbers in total). The motivation behind this is the following: if we think of adjusting the weights in the network as moving the decision boundary (Figure 3.8 (left)), it is intuitive that the data which is not distributed around the origin will cause the classification success to get very sensitive to weight changes.[2] [8] Sometimes it is also appropriate to scale the data so it has the same variance in all directions. See [3] for more details and other techniques.

[2]Weights in Figure 3.8 are the parameters that determine the slope of the decision boundary

3.3. Convolutional neural networks

Convolutional Neural Networks (CNNs) are specifically designed to classify images. The biggest advantage they have in comparison to perceptrons is that they have fewer parameters. The number of inputs to the network for RGB images requires very large number of weights between the inputs and other layers. Reducing the number of neurons also regularises the network and reduces the risk of overfitting [3]. CNNs are trained with backpropagation as well as perceptrons.

3.3.1. CNN layer types

The fundamental blocks for learning regular ANN still apply here. CNNs are composed of McCulloch-Pitts neurons with activation functions. 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. [7] In CNNs, it's common to divide up the operations the neuron performs into separate layers (for instance, applying activation function is implemented as an 'activation layer').

Convolution layers

The weights in CNNs can be interpreted as learnable filters. Each of these filters is learnt to extract different features from the input image. Inputs of the convolutional layers in CNN are three-dimensional tensors. The result of the convolution operation (which is extended to the full depth of the input tensor) for a specific filter is an activation map: a two-dimensional representation of the locations of the specific feature in the image. In the very first layers of the network, the filters extract simple features such as corners, curves, edges of certain orientation, etc. When the input image is RGB, the filters in the first layer have the dimensions of MxMx3, where M is a small number (typically 3, 5, 7, etc.). As we go deeper into the network's layers, the filters are looking for more complex features. The number of filters per layer, the stride of the convolution operation, and the size of the filters are subject to different network architectures. When all filters are applied to the input tensor of the convolutional layer, their activation maps are stacked onto each other and become the input tensor for other layers. [7]

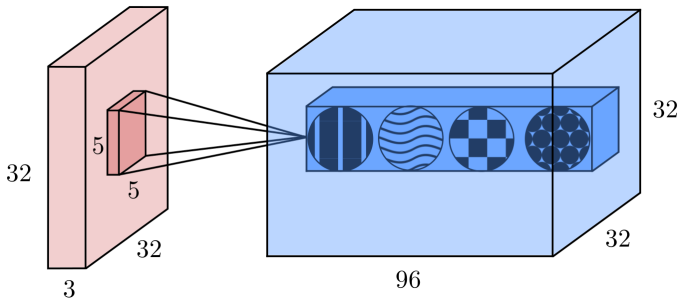


Figure 3.15: The full-depth convolution operation in a convolutional layer. The input size corresponds to a small RGB image. The result of the series of convolutions is a tensor of stacked activation maps for the filters used in the layer. [13]

Pooling layers

The function of pooling layers is to reduce the size of the layers in the network. Pooling operation performs downsampling of the data encoded in the layers while retaining the spatial information about the locations of the detected features. Pooling can be interpreted as summarizing a small area of pixels to a single pixel based on a certain criterion. The most commonly used criterion is replacing a small pixel group by one with a maximum value. This is referred to as max-pooling. Similarly to *conv* layers, the size of the pooled sub-regions and the stride of the pooling operation are subject to the network architecture. [3]

Max-pooling layers have no trainable parameters. Sometimes it is necessary to keep track of the original locations of the maximum elements. [25]

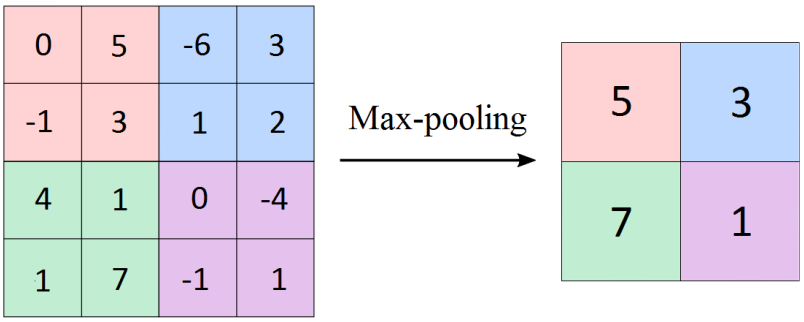


Figure 3.16: Max-pooling of size 2x2 and stride 2. [13]

Fully connected layers (FCN)

CNNs were originally designed for image classification, where one classifies the entire image. The structure of these networks consists of a series of *conv* layers followed by *pool* layers. When the input is downsampled (pooled) to a certain level, the output tensor is flattened and becomes an input to a multilayer perceptron (FCN - fully connected network). The role of the convolutional part here is to create a downsampled representation of the features in the image. The perceptron then learns to classify this feature vector into the desired number of classes. [7] [39]

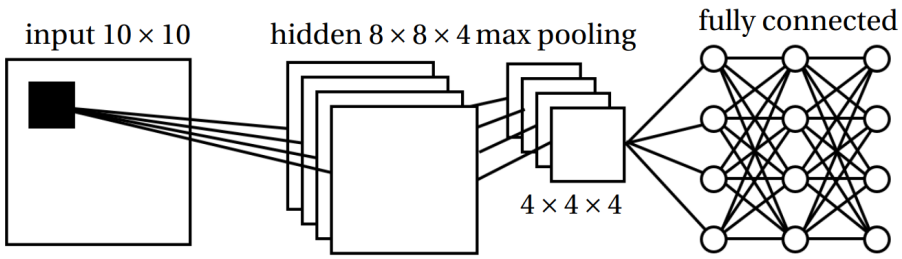


Figure 3.17: Schematic of the standard CNN topology for image classification. [3]

3.3.2. Examples of CNN architectures

Various architectures have been introduced, each having a different number of convolution layers, size of the filters, strides 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. [7] Here is a summary of milestone architectures presented in recent years:

- **AlexNet**

  The first work that popularized CNNs in Computer Vision was AlexNet [18]. The Network had very similar architecture to LeNet [19], 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 which was always immediately followed by a *pool* layer). [7]

- **GoogLeNet**

  The ILSVRC 2014 winner was a Convolutional Network from Google [20]. Its main contribution was dramatically reducing the number of parameters in the network compared to AlexNet. [7]

- **VGGNet**

  The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet [21]. 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. [7]

- **ResNet**

  Residual Network developed by Kaiming He et al. [22] was the winner of ILSVRC 2015. It features special skip connections and a heavy use of batch normalization. [7]

3.4. Semantic segmentation

This section presents the most successful methods involving neural networks and supervised learning. 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.

Segmentation has always been one of the most fundamental areas of computer vision. The classic 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 ideal and where the robustness of the algorithm is not crucial. To give an example of classic methods, one can refer to thresholding, region growing and mean-shift segmentation [23]. More advanced methods which use machine learning classification have also been introduced, such as TextonBoost, TextonForest and Random Forest [25] [24]. These algorithms have fallen out of favour due to the massive success of ANN.

3.4.1. Encoder-decoder architecture

In the previous chapter, 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 an 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. [12]

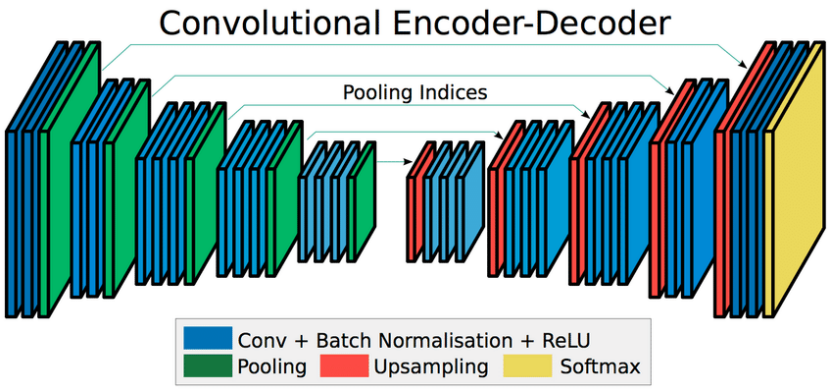


Figure 3.18: SegNet - an example of encoder-decoder CNN architecture. [25]

The purpose of the encoder is to downsample the input images while still representing their 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.

Shortly after the success of CNN in image classification challenges, there have been several segmentation architectures introduced which use CNN as the encoder. Some of the state-of-the-art architectures 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 [25], introduced in 2015, differs from these architectures as it has a significantly lower number of parameters and the design of the encoder-decoder network allows it to be trained via standard method using backpropagation and SGD. [25]

Input upsampling

The upsampling in the decoding part of the network is done via two mechanisms: learnable transposed convolution and unpooling.

Transposed convolution, just like the standard convolution used in CNNs, uses learnable filters. The difference is that it takes a single input point instead of a region, uses it to multiply each element of the filter and creates its imprint in the output layer. This scheme is illustrated in Figure 3.19 (left). [12]

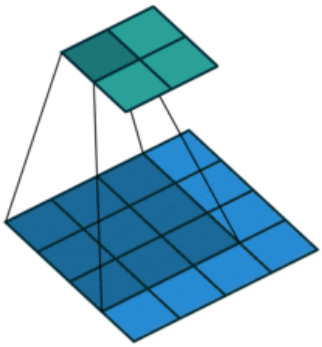


Figure 3.19: Transposed convolution. [37]

There are several ways to impelement unpooling. In an encoder-decoder architecture, the corresponding layers in the encoder and decoder can, for example, share the original locations of the elements that were pooled in the encoding part. The decoder then uses these indices for upsampling, as shown in Figure 3.19 (right). This reconstructs the original positions of the features in the original image. Unpooling operation does not have any learnable parameters. [25] [12]

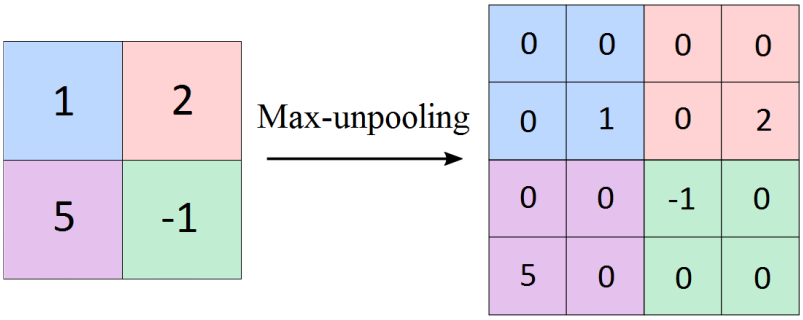


Figure 3.20: Max-unpooling. The locations of the maximum elements were saved during max-pooling. The remaining elements are set to zero.

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. [26]

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 activation maps (Figure 3.18). The entire architecture can be trained using stochastic gradient descent. [26]

SegNet - encoder

The architecture of the encoder network is topologically identical to the 13 convolutional layers in the VGG16 network. Each encoder in the encoder network performs convolution with a filter bank to produce a set of activation 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. [25]

SegNet - decoder

The decoders in the network upsample their input feature maps using the memorized max-pooling indices from the corresponding encoder feature maps. These feature maps are then convolved (using transposed convolution) 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. [25] The schematic of the SegNet architecture can be found in Attachment XY.

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. The schematic of the Bayesian SegNet architecture can be found in Attachment XY. [24]

Monte Carlo Dropout

Monte Carlo Dropout (MCDO) sampling helps us understand the model uncertainty of the result. As explained in Chapter 3.2.3, the standard weight averaging dropout proposes to remove dropout at test time and scale the weights proportionally to the dropout percentage. MCDO, on the other hand, samples the network with randomly dropped out units at test time. [24]

It is important to highlight that the probability distribution from MCDO 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. [24]

3.4.4. 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 the figure below. This metric ranges from 0–1 (0–100%) with 0 signifying no overlap and 1 signifying perfectly overlapping segmentation. [29]

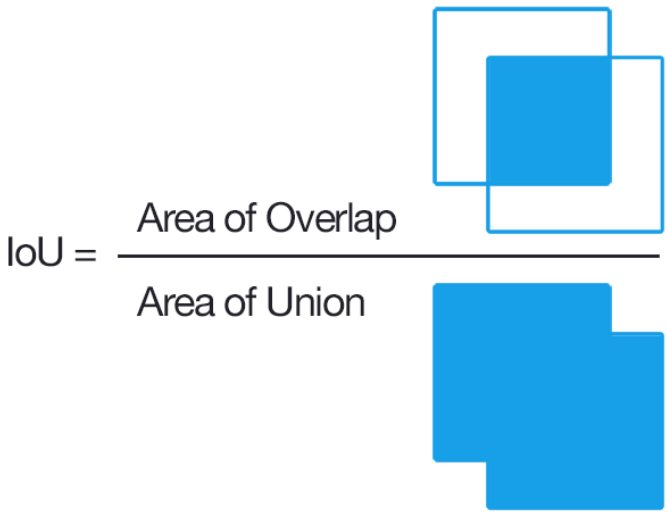


Figure 3.21: Intersection over union. [29]

4. Implementation and method

In this chapter, the original Caffe implementation of SegNet and Bayesian SegNet with their simplified versions SegNet Basic and Bayesian SegNet Basic will be tested on a custom dataset. Part of this will be evaluating the effect of various hyperparameters on training. This chapter will also give instructions on how to set up the software and hardware environments for running Caffe library for ANN. The entire network architecture and other code used in this section are available at [30] and [34].

4.1. CPU vs. GPU for training ANN

Central Processing Unit (CPU) is the main computational unit of a computer and is designed to perform a wide variety of complex instructions. Current CPUs usually have 4 to 8 separate cores, which allow them to run several tasks in parallel. Graphics processing unit (GPU), on the other hand, was originally designed for rendering computer graphics only. CPU has a much lower number of cores, but these run at a high frequency and are very capable in terms of the instructions they perform. Therefore, CPUs are great for sequential tasks. GPU comprises of a large number of 'simple' cores which makes it more suitable for computing parallel tasks. [11]

The main part of the computations in ANN is matrix multiplication where GPU has the power of performing these operations by parts in parallel and speeds up the training significantly. [11]

There are libraries such as CUDA and OpenCL that allow programmers to write their code in a usual manner and run it directly on a GPU. For the purposes of ANN, NVIDIA has also developed a library of the most commonly used CUDA primitives named cuDNN. [11]

A CPU does not have its own memory resources (apart from very small memory sections called caches) and only has access to the system's RAM. External GPUs always come with their own block of RAM on the chip. The size of the RAM for the top-end GPUs ranges from 8 to 12 GB. When using GPUs to train ANN, the size of the RAM is crucial because the model with all its parameters resides in this memory.

Tensor cores

Tensor Core is a special GPU feature offered by NVIDIA cards. It enables mixed-precision computing by 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). [31]

4.2. Libraries for ANN

As the architecture and training of ANN are getting more complex, it is very helpful to use programming tools with higher abstraction for their design. There are libraries such as Caffe, TensorFlow, and PyTorch for this. The common idea of these libraries is to make an abstraction of the network's architecture called computational graph. Therefore, the user can think of designing and training the network separately by applying an optimizer to the computational graph that represents the network's layers. [11]

4.2.1. Caffe

Caffe is a deep learning library made with expression, speed, and modularity in mind. It has been developed by Berkeley AI Research (BAIR) and by community contributors. [33] The main difference between this and other libraries is that the user often does not need to write any code at all. The architecture of the network (the computational graph) is described in a *.prototxt* file where one creates the layers of the network in the desired order. Also, rather than having an optimizer object (in Tensorflow for example), 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 these files are created, the user runs Caffe computation from the command line. The library is written in C++ and the pre-built binaries are executed when the computation is executed. [11]

Caffe comes with bindings for Python (CPW - Caffe Python Wrapper, or pycaffe) and Matlab, which is very useful for the inference phase. The biggest downside of Caffe and CPW is that they are very poorly documented.

Figure 4.1: Examples of the best deep learning frameworks. [32]

4.3. Setting up environment for Caffe

4.3.1. Hardware configuration

The GPU used for the computations has been selected according to the most up-to-date benchmarks and recommendations found online. When choosing a GPU in general, one needs to decide between AMD and NVIDIA chips. For ANN however, NVIDIA is the default choice because it's way more 'ANN-friendly' as it offers more features specifically designed for ANN computations.

It's advisable to use an SSD in the training PC, because the data flow begins with reading the training data (images) from a storage, in this case from the computer's hard drive. Another possibility that some libraries offer is moving the training data into RAM before the training is initiated. Figure 4.2 shows the GPU used for training SegNet.

Figure 4.2: GIGABYTE GeForce RTX 2060 SUPER AORUS 8G. [38]

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 was Ubuntu 18.04 LTS 64 bit. It 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 the installation. Before taking other steps, it is **vital** to disable the Nouveau driver and use the NVIDIA driver instead. This is done by navigating to *Application menu -> Software & Updates -> Additional drivers -> Using NVIDIA driver metapackage from nvidia-driver-XYZ (proprietary, tested) -> Apply changes.* The driver version used was **nvidia-driver-440**.

CUDA installation

CUDA version is determined by the version of cuDNN compatible with the used Caffe version, which is cuDNN 5.1 in our case. The corresponding CUDA version is CUDA 8.0. On Ubuntu 18.04, the procedure is as follows: [32]

- **Download CUDA 8.0 runfile.** Go to CUDA Legacy Releases and look for *CUDA Toolkit 8.0 GA2 (Feb 2017)*. The standard *.deb* installer supports only Ubuntu 16.04 LTS. 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 and run: [32]

```
$ 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 in 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 usual way is to set the "PATH" variables in the current session of the Ubuntu Terminal. However, it is useful to add these permanently to system's *~/.bashrc* file:

```
$ 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 on, all newly opened Terminal sessions should have the paths set correctly.

Installation of cuDNN

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. [32]

- **Download cuDNN 5.1 for CUDA 8.0.** To get the corresponding 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: [32]

```
$ 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 is advisable to use Pycharm Community Edition as an editor, because it offers a very convenient combination of GUI and the standard command line environment.

It is good practice to use Python Virtual Environment to easily maintain the required packages and to make the project transferable to another Linux PC. In Pycharm, one can do this in an active Pycharm project by navigating 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 that includes all package files we install.

When the *virtualenv* is configured properly, it will automatically activate when we enter the Ubuntu Terminal session by clicking on the *Terminal* button located at the bottom bar of Pycharm window. From this Terminal, we will be launching all SegNet scripts and use it to install the required packages by calling:

```
(venv) user@user:/current/path$ pip3 install <package-name>
```

4.3.3. Building Caffe for SegNet

Caffe is an open-source library. The authors of the SegNet created a slightly modified version of Caffe called *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 by newer GPUs, there's another version of *caffe-segnet* from [34] that supports cuDNN 5.1. The author claims that it decreases the inference time by 25 % to 35 %. Therefore, this version was 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 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: [33]

```
$ 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 [34] 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 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++
```

- **Initiate the build.** Once the *Makefile.config* is located in the *caffe-segnet-cudnn5-master* directory, everything should be ready for the final step. Execute 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
```

4.4. Image annotation

In supervised learning, one needs to manually create the training data consisting of inputs and corresponding targets (called Ground Truths in segmentation). There's a variety of annotation tools available on the internet, both under commercial and free licenses. The final train, validation and test datasets used contain 2600 + 90 + 179 images from an outdoor environment.

Labelbox

Labelbox [36] is a paid online annotation tool. The best feature of Labelbox is that it allows sharing the datasets with other users and therefore speeding up the labeling significantly. Labelbox offers free access to students to the full version. When the labeling is finished, one exports the image/label pairs to a *.JSON* file. This file contains links to the annotated images that are stored online and it is necessary to download them separately (Labelbox is still in development, this is valid at the time of publishing). To automate this process, one can call the *download() function* from the *utilities.py* script (Attachment XY).

4.5. Setting up SegNet

Caffe implementation of ANN 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 of the network. 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. [33] The *inference* file is used for running the trained network. The files used in this section are available at [30]

4.5.1. Solver settings

The *solver* file contains the optimization parameters. The description of the parameters can be found in the original Caffe documentation [33]. An example 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"
```

Listing 4.1: Contents of *solver.prototxt* [30]

4.5.2. Training

Input layer and input pre-processing

The *train* file begins with the *DenseImageData* layer. This layer specifies the size of the mini-batch. The value is limited by the amount of memory that the GPU offers. When a larger size of the mini batch is needed, Caffe can 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. [33]

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 adding more stochasticity to 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 is necessary to perform them separately and feed the *DenseImageData* layer with already processed images. [33]

```
// The first layer in the network
name: "segnet_train"
layer {
name: "data"
type: "DenseImageData"
top: "data"
top: "label"
dense_image_data_param {
   source: "/path/to/train_image_paths.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
   }
}
```

Listing 4.2: Input layer in *train.prototxt* [30]

Images and labels are loaded as *.jpg* and *.png* files directly from the hard drive (there are more methods that Caffe offers). The path to the *image_paths.txt* file that contains the image/label paths in the following format

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

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

The method used for the mean subtraction was the per-channel mean. The *per_channel_mean* function in *utilities.py* calculates the mean values for R, G and B components of the images in 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, the segmentation mask for the class number 1 has a pixel value of 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 dimensions of the last *conv* 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
  }
}
```

Listing 4.3: Setting number of outputs in *train.prototxt* [30]

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 CamVid dataset) then there is a need to weight the loss differently based on the true class. This is termed class balancing. We use median frequency balancing [13] 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. We also experimented with training the different variants without class balancing or equivalently using natural frequency balancing.*" [25]

```
// 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"
}
```

Listing 4.4: Output layers of *train.prototxt* [30]

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 of the *SoftmaxWithLoss* layer. Since the dataset used has only two classes whose occurrences can be considered balanced, this option is set to *false*.

Training initialization

Training the network from scratch is initiated by entering these commands:

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

```
$ ./caffe train -solver /path/to/segnet_solver.prototxt -snapshot
    /path/to/snapshot_iter_XY.solverstate
```

The encoder and decoder weights are initialized using the MSRA method by default. Another scenario is when we want to use transfer learning (Caffe library has a Model Zoo where people share their network weights). In this case, Caffe needs a path to the *.caffemodel* 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
```

There are multiple ways of tuning the pre-trained model when using transfer learning. For instance, one can experiment with the learning rate of the pre-trained weights: they can either stay unchanged (zero learning rate) or the learning rate applied to them is lower than the global value used in other layers. [7] In encoder-decoder architecture, one usually applies transfer learning only to the encoder network as it has no other purpose than extracting general features from the image. The corresponding setting in the *train* file is the set of *lr_mult* parameters by which the learning rate for the layer is multiplied. An example of setting a Caffe layer where that layer stays unchanged can be found in the snippet below.

```
layer {
  bottom: "data"
  top: "conv1_1"
  name: "conv1_1"
  type: "Convolution"
  # Learning rate factor - weights
  param {
    lr_mult: 0     # Remains unchanged during training
    decay_mult: 0  # Remains unchanged during training
  }
  # Learning rate factor - thresholds
  param {
    lr_mult: 0     # Remains unchanged during training
    decay_mult: 0  # Remains unchanged during training
  }
  .
  .
  .
}
```

Listing 4.5: Setting up *train.prototxt* for transfer learning [30]

4.5.3. Inference

The network is ready to be deployed in this phase. At this point, it is very convenient to use pycaffe for running the model by feeding it with input data and calculating the segmentation accuracy. To run the segmentation, several preparation steps must be taken first.

Calculating statistics for batch normalisation

The batch normalisation layers in SegNet shift the input feature maps according to their mean and variance statistics for each mini- batch during training [3]. At inference time, we must use the statistics for the entire dataset and obtain the final *.caffemodel* for the inference phase. [28] We do this by calling *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 network architecture for the inference is now in the *inference* file and the same is in the *train* file apart from the input and output layers and the settings of the batch normalisation layers. The snippet below shows the changes of the output: the loss function is no longer computed and the only output we care about is the set of softmax probabilities. The *DenseImageData* layer is also skipped, because the data will be provided via pycaffe. Part of this is switching all batch normalisation layers to the INFERENCE mode. [16]

The script takes the desired *.caffemodel* file specified in *snap_iter_XY.caffemodel*, calculates new γ, β parameters for the batch normalisation layers and saves everything to *final_weights.caffemodel*. The new *.caffemodel* file is now stored in the specified *inference_folder*. [16]

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

Listing 4.6: Replacing input layer type in *inference.prototxt* [30]

Running the Segmentation

The script *segnet_inference.py* is used for running the segmentation. One must provide the network with images either by specifying a path to a video file or by specifying a sequence of image names to look for in the image folder (this is a standard OpenCV convention). In each step of the algorithm, we must subtract the per-channel mean from the input image that is being processed. This is part of the script and one only needs to provide the BGR values used at train time.

Once an appropriate test set of images is ready, the segmentation is started 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
```

4.5.4. Testing

The *test* file is used only for calculating the loss of the validation dataset. It is very similar to the *train* file: it has a *DenseImageData* layer with a path to the validation dataset, *mirror* and *shuffle* parameters set to false, *batch_size* to 1 and the *SoftmaxWithLoss* followed by *Accuracy* layers as the output. The subtraction of the per-channel mean is still present and the values computed from the training dataset are the same as in the training phase.

For testing, it is necessary to use the *.caffemodel* file generated by *compute_bn_statistics.py* to ensure the proper function of the batch normalisation layers, which must be in the INFERENCE mode and must differ from the settings of the *train* file.

```
name: "segnet_test"
layer {
  name: "data"
  type: "DenseImageData"
  top: "data"
  top: "label"
  dense_image_data_param {
    source: "/media/phil/SegNet/data/custom/val_linux.txt"
    batch_size: 1   # Always 1 for SegNet
  }
  # BGR order
  transform_param {
    mean_value: 129
    mean_value: 126
    mean_value: 126
  }
```

Listing 4.7: Setting up the input layer of *test.prototxt* [30]

Testing is executed similarly as training using the command line:

```
# Navigate to the caffe-segnet folder
$ cd /path/to/caffe-segnet/build/tools/
# Initiate testing
$ ./caffe train -model /path/to/segnet_test.prototxt -weights
    /path/to/final_weights.caffemodel
```

4.5.5. Bayesian SegNet

Since Bayesian SegNet differs from SegNet only in terms of added dropout layers and a different method of performing the inference the above-mentioned procedures for setting the solver and training are also applicable. Therefore, one can start the training by using commands from the previous section. One must also not forget to replace the paths of the *train* and *solver* files.

The input layer in the *inference* file has one major difference: unlike in SegNet, the first *input_dim* parameter at the top of the *inference* file represents the number of MCDO samples and can be adjusted. At inference time, the script passes the same image *input_dim* times and simply averages the output of the network. For this reason, the dropout layers that are inactive by default when Caffe is performing inference (TEST, in Caffe terminology) must be set to active in this case. The corresponding parameter in the dropout layer is *sample_weights_test: true*.

The batch normalisation layers are set to INFERENCE mode. The final *.caffemodel* is obtained the same way as in SegNet by calling *compute_bn_statistics.py*. Here, unlike during inference time, the network's output is computed using the weight averaging technique instead of MCDO.

```
layer {
  bottom: "conv1_1"
  top: "conv1_1"
  name: "conv1_1_bn"
  type: "BN"
  bn_param {
    bn_mode: INFERENCE     # Inference mode of BN
    .
    .
    .
  }
}
.
.
.
layer {
  name: "encdrop5"
  type: "Dropout"
  bottom: "pool5"
  top: "pool5"
  dropout_param {
    dropout_ratio: 0.5
    sample_weights_test: true # For Monte Carlo Dropout
  }
}
```

Listing 4.8: Setting MCDO in *inference.prototxt* [30]

The setting of the *test* file remains the same as in SegNet: input is provided by the *DenseImageData* layer, *batch_size* is set to 1 and the batch normalisation layers are in INFERENCE mode. The dropout layers can also be set to active here. This *test* file still only serves for checking the validation loss.

The inference is initiated by calling:

```
(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 MCDO sampling: the variance of the output segmentation computed from all MCDO samples.

4.5.6. SegNet Basic and Bayesian SegNet Basic

SegNet Basic and Bayesian SegNet Basic are networks provided by the SegNet authors and are similar to their full versions but have fewer layers (see Attachment XY). These shallow versions are used in the same way as their parent architectures. Therefore, the same training and inference procedures apply to SegNet+SegNet Basic and Bayesian SegNet+Bayesian SegNet Basic.

4.6. Optimization of Hyperparameters

Hyperparameters are parameters that are set before the training begins and do not change during the training. The choice of hyperparameters is a task in its own right and requires a sufficient amount of trial and error. There are some general approaches (mostly empirical) one can follow to find the right parameters. The goal is to ensure that the network reaches an optimal value of the loss function. [7]

Optimizer

Every training of a neural network starts with the choice of an optimizer. As the most recent research suggests, Adam is the default choice for training CNNs. If the CNN is built from scratch, it is advisable to start from the simplest SGD optimizer and observe the values of the loss function to detect potential problems in the architecture or the code. [10]

Learning Rate

The parameter that has the biggest effect on training is the learning rate: it is the first parameter one should set. It is recommended to start a coarse search first while observing the loss for both training and validation datasets for a few initial epochs. Then, after the training is done, choose a thinner interval of optimal learning rates and perform a finer search. [9]

As the learning rate has a multiplicative effect on the gradient accumulation during mini-batch training, it is logical to pick the values from the logarithmic space. [9]

Cross-validation Strategy

This strategy is also referred to as early stopping. The idea is that one observes both training and validation loss during training. When these losses go apart, the network tends to overfit to the training data. This is a crucial step when finding optimal hyperparameters and it must always be checked. [7]

Regularisation

When building a network from scratch, one starts with a simple SGD algorithm with no regularisation involved to ensure that the loss values are reasonable. After we check for errors in the code and after the network trains with SGD, regularisation is turned on. It is usually set to a very small value, typically of the order 10^{-4} [9].

5. Results

The segmentation networks introduced in the previous chapter, SegNet, Bayesian SegNet and their simplified versions (Basic) were trained using the described techniques and various hyperparameters. The default optimizer choice for training CNNs is usually Adam, but its implementation in Caffe takes much more memory than other algorithms. That is why the algorithm chosen for training was AdaDelta, which is used by many users of the SegNet architecture [35].

As AdaDelta adapts the learning rate over the course of training, there is no longer a need to manually tune the learning rate decay scheme (which would become another hyperparameter). Therefore, the first hyperparameter tuned was the base learning rate for the AdaDelta algorithm. The search was initiated within a coarse interval of values: $< 10^{-3}, 10^{0} >$. Since the Caffe implementation of SegNet comes with custom scripts for calculating batch normalisation statistics for the inference phase, checking the validation loss periodically becomes extremely memory demanding and time inefficient. Therefore, the validation loss was checked only once at the end of the last training epoch to ensure that the values of losses had not diverged.

All variants of SegNet were trained using transfer learning where the encoder weights are pre-trained and either stay unchanged or their learning rate is decreased. In case of Bayesian SegNet and SegNet, the encoder was initiated using VGG16 weights. For SegNet Basic and Bayesian SegNet Basic, the encoder was initiated from a model trained on the CamVid dataset which is available at SegNet Model Zoo.

After a reasonable learning rate value was found, the random search was limited to the close interval around it. Then, the training was executed until no further change in the loss function was observed. In the original paper [25], the authors use L2 regularization. The value of the corresponding *weight_decay* hyperparameter was left unchanged and remained as the SegNet authors suggest.

The difference observed across the network variants was the time it took to achieve low loss values. This is influenced by the size of the network (Basic versions train faster) and the dropout settings (dropout slows down the training).

Figures 5.1 and 5.2 are examples of tuning of the learning rate (Bayesian SegNet). The network was trained using transfer learning. The figures below show two training schemes applied to the pre-trained encoder: in Figure 5.1, the encoder weights stay unchanged during the training. This apparently makes it harder for the decoder to adapt. Also, training with learning rates that initially seem to work well makes the loss diverge from the optimal value in the last few epochs. In Figure 5.2 on the other hand, the encoder weights are allowed to change but only with a decreased learning rate. This scheme tends to give more stable training results and speeds up the training. Therefore, this second scheme was applied to all variants of SegNet.

It turns out that larger values of learning rate tend to work better with AdaDelta and lead to better values of the loss function.

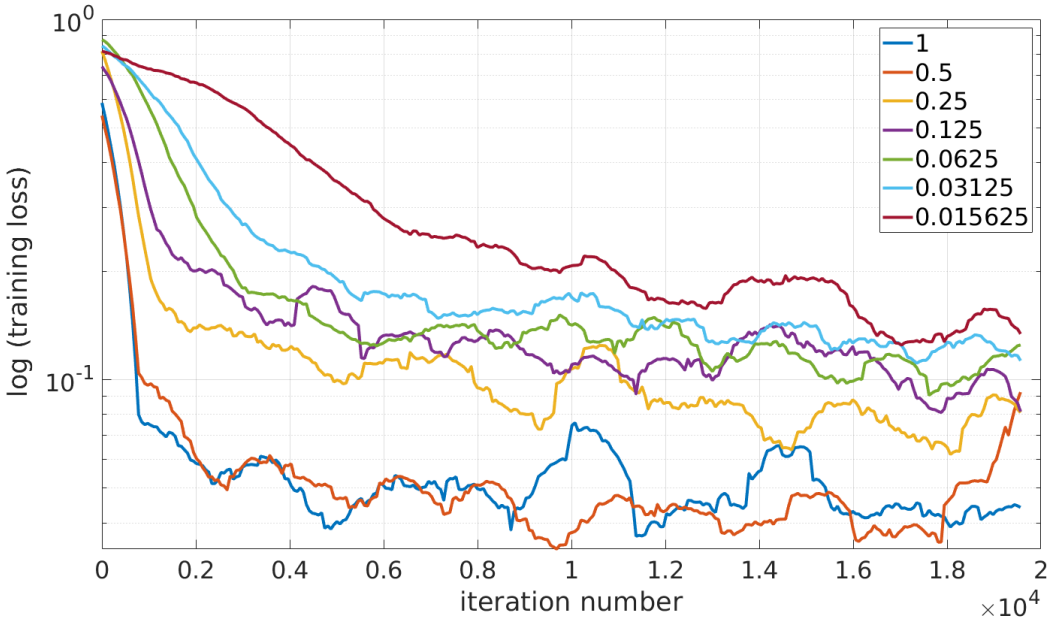


Figure 5.1: Coarse search of the *learning_rate* parameter, Bayesian SegNet. The training loss is observed for 30 epochs and the data is smoothed. The encoder was initialized using pre-trained VGG16 model and the corresponding layers stayed **unchanged** during the training.

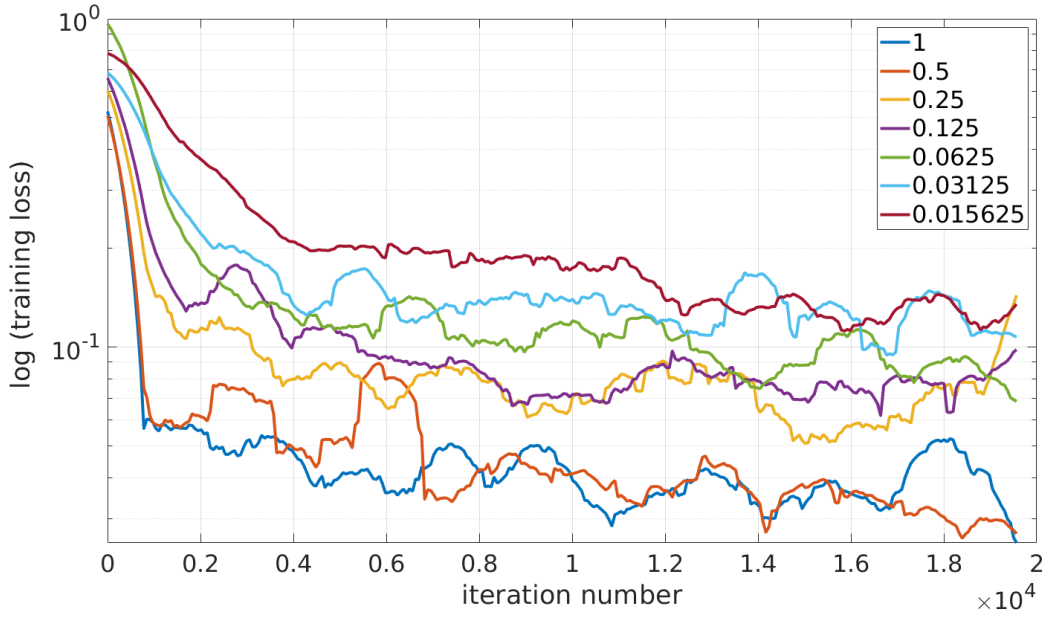


Figure 5.2: Coarse search of the *learning_rate* parameter, Bayesian SegNet. The training loss is observed for 30 epochs and the data is smoothed. The encoder was initialized using pre-trained VGG16 model and the learning rate of the corresponding layers is **decreased** by the factor of 10 during the training.

Table 5.1 summarizes the best training results obtained for all SegNet architectures. The metrics used for evaluation is IOU for each class: class 0 (*background*), class 1 (*path*). It also contains other useful information, such as the inference and training times. In terms of computational cost, it is evident that the best-performing architecture is SegNet Basic. Bayesian versions of SegNet repeat the inference based on the number of MCDO samples and hence take longer to evaluate. The inference runs on GPU as well as the training.

Architecture	**base_lr**	**weight decay**	**batch size**	**MCDO samples**
SegNet	0.95	0.0005	4	-
SegNet Basic	0.75	0.0005	4	-
Bayesian SegNet	0.5	0.0005	4	8
Bayesian SegNet Basic	0.85	0.0005	4	8
	IOU class 0	**IOU class 1**	**Inference time [ms]**	**Training epoch time [s]**
SegNet	0.965	0.971	42	368
SegNet Basic	0.966	0.972	23	312
Bayesian SegNet	0.974	0.979	305	432
Bayesian SegNet Basic	0.967	0.972	177	313

Table 5.1: Statistics for all SegNet variants on the *test* dataset. The inference was ran on GPU.

Figure 5.3 shows the final segmentation results for several image scenes from the *test* dataset. For Bayesian versions of SegNet, the segmentation comes with the uncertainty plot where light regions mean larger variance of MCDO samples during inference. The uncertainty is averaged over all segmentation classes. We see that the network is more uncertain in the regions that are close to the object boundaries. Also, the full versions of the architectures, SegNet and Bayesian SegNet tend to give more precise results on the boundaries. They are primarily designed for more complex scenes with multiple classes and the encoder is more capable of extracting finer features as the model capacity is higher. In addition, the pre-trained encoder for the full versions was trained on more images compared to the one used for the initialization of the Basic versions. On the other hand, the Basic versions might offer much better performance for practical applications where the number of classes is small.

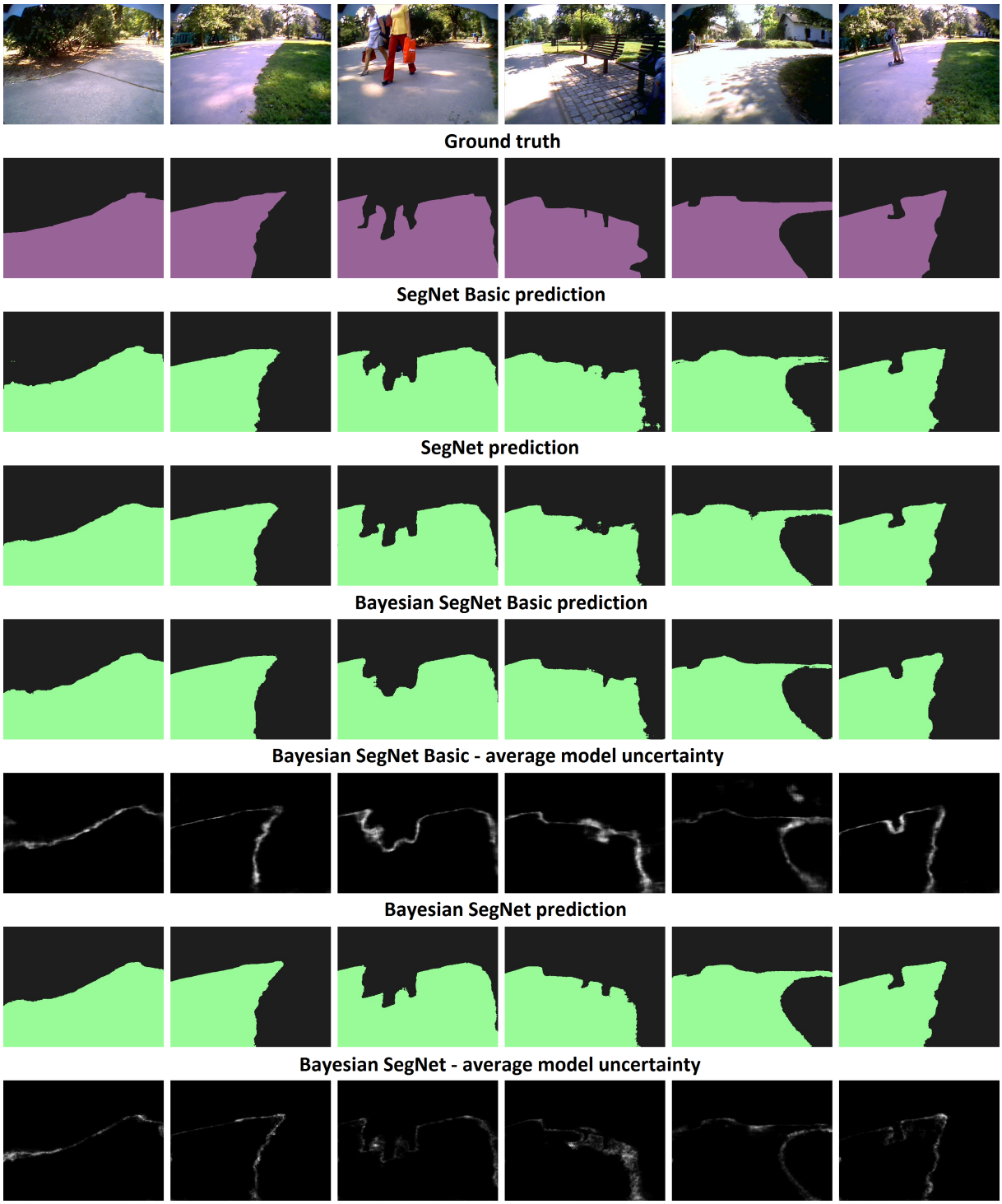


Figure 5.3: Comparison of the segmentation performance of all SegNet variants. The Bayesian versions of the architecture give the estimate of the model uncertainty, where the lighter regions mean higher variance across the MCDO samples taken during inference.

6. Conclusion and future work

This thesis presented some of the most recent ANN architectures used for image segmentation together with their Caffe implementations. An extensive step-by-step procedure for setting up the software and hardware environments was described and tested on a fresh installation of the operating system. Part of this was showing the benefits of using Debian based distributions of Linux for working with libraries for ANN: the procedure described by shell commands is very clear and can be easily repeated on a different machine.

The Caffe implementation and auxiliary Python scripts for the presented networks were tuned for the purpose of the thesis. The goal was to perform segmentation on a custom dataset with two object classes. The dataset consisitng of more than 2600 images was created using the best currently available online annotation tool (Labelbox). In the training phase, the networks were adapted for various transfer learning strategies and showed the power of using pre-trained encoders when the dataset is small. The training hyperparameters were tuned according to the common strategies. As the result, all SegNet variant were successfully trained using AdaDelta optimization and achieve very good values of segmentation accuracy: over 90 % IOU on the *test* dataset. There is always room for tuning of hyperparameters and achieve even better values of the loss function.

During the inference phase, the performance of the various architectures was observed and compared. This can give an idea for the computational power needed for further implementations. The probabilistic variants of SegNet can estimate the overall model uncertainty and hence support the decision making when the network is used in practical applications, such as self-driving robots.

7. Bibliography

[1] MWITI, Derrick. 2019. A 2019 Guide to Semantic Segmentation. In: *Heartbeat* [online]. Fritz AI. Available at: https://heartbeat.fritz.ai/a-2019-guide-to-semantic-segmentation-ca8242f5a7fc

[2] KARAGIANNAKOS, Sergios. 2019. Semantic Segmentation in the era of Neural Networks. In: *AI SUMMER* [online]. Available at: https://theaisummer.com/Semantic_Segmentation/

[3] MEHLIG, Bernhard. 2019. Artificial Neural Networks. ArXiv.org [online]. Available at: https://arxiv.org/abs/1901.05639

[4] PUENTE, Santiago. 2018. *Single and Multi-Label Environmental Sound Classification Using Convolutional Neural Networks* [online]. Gothenburg. Available at: https://odr.chalmers.se/handle/20.500.12380/255604. Master's thesis. Chalmers University of Technology.

[5] GOODFELLOW, Ian, Yoshua BENGIO a Aaron COURVILLE. *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016. ISBN 978-026-2035-613.

[6] GROMAN, Martin. Tvorba umělé neuronové sítě pro výpočet termodynamických veličin [online]. Brno, 2019 [cit. 2020-06-07]. Available at: http://hdl.handle.net/11012/175381. Master's thesis. Vysoké učení technické v Brně. Fakulta strojního inženýrství. Ústav matematiky. Supervisor Tomáš Mauder.

[7] CS231n: Convolutional Neural Networks for Visual Recognition: Lecture Notes. *CS231n: Convolutional Neural Networks for Visual Recognition* [online]. Stanford: Stanford University. Available at: https://cs231n.github.io/

[8] Lecture 4 | Introduction to Neural Networks *YouTube* [online]. 11. August 2018. Available at: https://www.youtube.com/watch?v=d14TUNcbn1k&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=4

[9] Lecture 6 | Training Neural Networks I *YouTube* [online]. 11. August 2018. Available at: https://www.youtube.com/watch?v=wEoyxE0GP2M&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=6

[10] Lecture 7 | Training Neural Networks II *YouTube* [online]. 11. August 2018. Available at: https://www.youtube.com/watch?v=_JB0AO7QxSA&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=7

[11] Lecture 8 | Deep Learning Software *YouTube* [online]. 11. August 2018. Available at: https://www.youtube.com/watch?v=6SlgtELqOWc&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=8

[12] Lecture 11 | Detection and Segmentation *YouTube* [online]. 11. August 2018. Available at: https://www.youtube.com/watch?v=nDPWywWRIRo

[13] COORS, Benjamin. 2016. *Navigation of Mobile Robots in Human Environments with Deep Reinforcement Learning* [online]. Stockholm. Available at: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A967644&dswid=9005.Degreeproject.KTHRoyalInstituteofTechnology.

[14] ALESE, Eniola. 2018. The curious case of the vanishing and exploding gradient. In: *Medium* [online]. Available at: https://medium.com/learn-love-ai/the-curious-case-of-the-vanishing-exploding-gradient-bf58ec6822eb

[15] BUSHAEV, Vitaly. 2018. Adam — latest trends in deep learning optimization. In: Towards Data Science [online]. Towards Data Science. Available at: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

[16] Batch Normalization Issue in SegNet. 2017. In: *GitHub* [online]. GitHub. Available at: https://github.com/alexgkendall/caffe-segnet/issues/109

[17] JONNARTH, Arvi. 2018. *Camera-Based Friction Estimation with Deep Convolutional Neural Networks* [online]. Uppsala. Available at: https://pdfs.semanticscholar.org/4c35/becacb2aab803468eb38f19d8418d79c7c08.pdf. Master's thesis. Uppsala Universitet.

[18] KRIZHEVSKY, Alex, Ilya SUTSKEVER and Geoffrey HINTON. 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM* [online]. ACM. Available at: http://web.b.ebscohost.com.ezproxy.lib.vutbr.cz/ehost/detail/detail?vid=0&sid=dcac7028-11f2-41e4-ba7d-d517a5a51a6f%40sessionmgr101&bdata=Jmxhbmc9Y3Mmc2l0ZT1laG9zdC1saXZl#AN=123446102&db=bth

[19] LECUN, Y, L BOTTOU, Y BENGIO and P HAFFNER. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* [online]. IEEE, 86(11), 2278-2324. Available at: https://ieeexplore-ieee-org.ezproxy.lib.vutbr.cz/document/726791

[20] SZEGEDY, Christian, Wei LIU, Yangqing JIA, Pierre SERMANET, Scott REED, Dragomir ANGUELOV, Vincent VANHOUCKE and Andrew RABINOVICH. 2014. Going Deeper with Convolutions. *ArXiv.org* [online]. Ithaca: Cornell University Library, arXiv.org. Available at: http://search.proquest.com/docview/2084489417/

[21] SIMONYAN, Karen and Andrew ZISSERMAN. 2014. Very Deep Convolutional Networks for Large-Scale Visual Recognition. *Visual Geometry Group* [online]. Oxford: University of Oxford. Available at: http://www.robots.ox.ac.uk/~vgg/research/very_deep/

[22] HE, Kaiming, Xiangyu ZHANG, Shaoqing REN and Jian SUN. 2015. Deep Residual Learning for Image Recognition. *ArXiv.org* [online]. Ithaca: Cornell University Library, arXiv.org. Available at: http://search.proquest.com/docview/2083823373

[23] COUFAL, J. Detekce cesty pro mobilní robot analýzou obrazu. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2010. 49 s. Master's thesis. Supervisor: Ing. Jiří Krejsa, Ph.D

[24] KENDALL, Alex, Vijay BADRINARAYANAN and Roberto CIPOLLA. 2016. Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding. *ArXiv.org* [online]. Ithaca: Cornell University Library, arXiv.org. Available at: https://arxiv.org/abs/1511.02680

[25] BADRINARAYANAN, Vijay, Alex KENDALL and Roberto CIPOLLA. 2016. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *ArXiv.org* [online]. Ithaca: Cornell University Library, arXiv.org. Available at: https://arxiv.org/abs/1511.00561

[26] KENDALL, Alex, Vijay BADRINARAYANAN and Roberto CIPOLLA. 2015. SegNet. *Machine Intelligence Laboratory* [online]. Cambridge: University of Cambridge. Available at: https://mi.eng.cam.ac.uk/projects/segnet/

[27] ZELTNER, Felix. 2016. Autonomous Terrain Classification Through Unsupervised Learning [online]. Luleå. Available at: http://ltu.diva-portal.org/smash/record.jsf?pid=diva2%3A1051763&dswid=-6301.Degreeproject.LuleåUniversityofTechnology.

[28] KENDALL, Alex. 2015. Getting Started with SegNet. *Machine Intelligence Laboratory* [online]. Cambridge: University of Cambridge. Available at: http://mi.eng.cam.ac.uk/projects/segnet/tutorial.html

[29] ROSEBROCK, Adrian. 2016. Intersection over Union (IoU) for object detection. In: *Pyimagesearch* [online]. pyimagesearch. Available at: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

[30] SegNet-Tutorial. 2020. *GitHub* [online]. GitHub. Available at: https://github.com/filipovyfusky/SegNet-Tutorial

[31] *NVIDIA* [online]. 2020. USA: NVIDIA. Available at: https://www.nvidia.com/

[32] *NVIDIA Developer* [online]. 2020. USA: NVIDIA. Available at: https://developer.nvidia.com/

[33] *Caffe* [online]. Berkeley: Berkeley AI Reseach. Available at: https://caffe.berkeleyvision.org/

[34] caffe-segnet-cudnn5. 2020. *GitHub* [online]. GitHub. Available at: https://github.com/filipovyfusky/caffe-segnet-cudnn5

[35] SegNet implementation in Tensorflow. 2020. *GitHub* [online]. GitHub. Available at: https://github.com/aizawan/segnet

[36] *Labelbox* [online]. 2020. Available at: https://labelbox.com

[37] Convolution arithmetic tutorial. 2018. *Deep Learning* [online]. LISA lab. Available at: http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html

[38] *GIGABYTE* [online]. 2020. GIGABYTE. Available at: https://www.gigabyte.com/

[39] MEHLIG, Bernhard. 2019. FFR135 - *Artificiella neurala nätverk: Lecture Notes*. Gothenburg.

8. Seznam použitých zkratek a symbolů

CMU	Carnegie Mellon University

9. Seznam příloh

- Nastavení režimu External mode: *external.txt*