



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

AUTOR PRÁCE

Bc. FILIP ŠPILA

SUPERVISOR

VEDOUCÍ PRÁCE

doc. Ing. JIŘÍ KREJSA, Ph.D.

BRNO 2020

Abstrakt

Tato bakalářská práce se zabývá návrhem, výrobou a realizací řízení nestabilního robota, balancujícího na sférické základně, známého také pod názvem ballbot. Předpokládá se kompletní návrh konstrukce, výběr pohonných jednotek, návrh, implementace a testování inteligentního řídicího algoritmu, který udrží robota v metastabilní rovnovážné poloze. Při vývoji budou využity softwarové nástroje MATLAB/Simulink. Práce také počítá s využitím mikrokontroleru dsPIC jako platformy pro finální řízení celého systému. Zadání projektu má interdisciplinární charakter a je realizováno jako týmová práce s jasně vymezenými úkoly pro jednotlivé členy.

Summary

This bachelor's thesis deals with the complete design, manufacture, and control of an unstable robot, balancing on a spherical base, also known as ballbot. The complete design of the construction, motor unit selection, design, implementation testing of an intelligent control algorithm to keep the robot in a meta-stable equilibrium is assumed. Multiple tools such as Matlab/Simulink are used for this approach. It also includes the final implementation of the code in the PIC microcontroller. The project has an interdisciplinary character and is meant to be done as teamwork whereby each team member has a strictly defined role.

Klíčová slova

Ballbot, konstrukce, inteligentní řízení, PID, MATLAB, Simulink

Keywords

Ballbot, construction, intelligent control, PID, MATLAB, Simulink

ŠPILA, F. *Semantic segmentation of images using convolutional neural networks*. Brno: Vysoké učení technické v Brně, Faculty of Mechanical Engineering, 2020. 18 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 příloženém 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.

V Brně 1. května 2017

Bc. Filip Špila

Děkuji Ing. Tomáši Spáčilovi, Matěji Rajchlovi a celému týmu z Mechlabu za podnětné připomínky a rady, které mi během práce poskytli. Dále děkuji své rodině za podporu jak během studií, tak během psaní této práce.

Bc. Filip Špila

Contents

1	Introduction	2
2	Problem statements	3
3	Research and theory	4
3.1	Architecture of artificial neural networks	4
3.1.1	Feed-forward networks	4
3.1.2	McCulloch-Pitts neurons	5
3.1.3	Activation functions	6
3.1.4	Multilayer perceptrons	8
3.2	Training of artificial neural networks	9
3.2.1	Loss function	9
3.2.2	Gradient optimization and backpropagation	10
3.2.3	Improving training performance	13
4	Bibliography	16
5	Seznam použitých zkratk a symbolů	17
6	Seznam příloh	18

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 in the most robust and reliable way possible. Speaking in terms of machine learning, each pixel of the input image is intended to belong to a specific class.

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

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

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



Figure 1.1: Segmentation of an urban road scene

2. Problem statements

The assignment of this thesis consists of several expected achievements. Firstly, a promising segmentation method using CNNs needs to be found and implemented. It is expected that the neural network will be as straightforward as possible while still being likely to be capable of giving satisfactory results for the chosen use case (segmentation of a path in outdoor environment for a robot navigation). The images will be provided by the supervisor of the thesis and used to train and validate the network performance. In addition, the author will pick an appropriate software tool for creating Ground Truths (FOOTNOTE manually created image labels that serve as a reference for the network to validate its current accuracy of prediction and to compute the needed adjustments of its parameters to get closer to the desired output) and use it to create the final training and validating datasets. Lastly, the network should be trained with various sets of hyperparameters (FOOTNOTE: hyperparameter definition) in order to get a closer idea of the network's training behaviour and to ensure the best possible results.

3. Research and theory

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

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

3.1. Architecture of artificial neural networks

Artificial neural network algorithms are inspired by the architecture and the dynamics of networks of neurons in human brain. They can learn to recognize structures in a given set of training data and generalize what they have learnt to other data sets (supervised learning). In supervised learning one uses a training data set of correct input/output pairs. One feeds an input from the training data into the input terminals of the network and compares the states of the output neurons to the target values. The network trainable parameters are changed as the training continues to minimize the differences between network outputs and targets for all input patterns in the training set. In this way the network learns to associate input patterns in the training set with the correct target values. A crucial question is whether the trained network can generalize: does it find the correct targets for input patterns that were not in the training set?

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 the inputs x to the space of the outputs y . In other words, to learn the function [zdroj SANTIAGO]

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

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

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

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

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

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

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

Recall that in supervised learning one needs a set of training data, in this case a set of matching x , y (footnote: y are often called LABELS) pairs. The training examples specify directly what the output layer must do at each point x ; it must produce a value that is as close as possible to y . The behaviour of the other layers is not specified by the training data which is why we call these layers 'hidden layers'. In Figure XY, an image of a four-layer feed-forward neural network with two hidden layers can be seen.

It is now very important to highlight that the neural network must be seen as something that is capable of modeling practically any function we can think of [general approximation theorem]. The power of this brings us to the definition of a classification task. In this task, the function the network approximates has discrete states, true/false. To give the most typical example, let's consider we have a large set of black and white images containing hand-written digits from 0 to 9. Now we want the network to basically associate the information encoded in the pixel values of the input image with the information of what digit that particular image shows. In this case, the vector space of inputs (flattened image array in the simplest case of size $m = \text{width} \times \text{height}$) is projected to the space of size $n = 10$ categories (digits 0 to 9).

3.1.2. McCulloch-Pitts neurons

Layers in Figure XY can be further divided into distinct computational units (again, just another nested functions) called neurons. This is where the resemblance with/to biological neurons comes into play. The neurons are mathematically modelled as linear threshold units (McCulloch-Pitts neurons), they process all input signals coming to them from other neurons and compute an output. In its simplest form, the model for the artificial neuron has only two states, active or inactive. Let's stick with this simple model for a while. If the output exceeds a given threshold then the state of the neuron is said to be active, otherwise inactive. The model is illustrated in Figure 1.4. Neurons usually perform repeated computations, and one divides up time into discrete time steps $t = 0, 1, 2, 3, \dots$. The state of neuron number j at time step t is denoted by

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

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

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

3.1. ARCHITECTURE OF ARTIFICIAL NEURAL NETWORKS

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

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

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

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

Equation (1.2) basically shows that the neuron performs a weighted linear average of the inputs n_j and applies an offset (threshold) which is denoted by μ_i . Finally, the function θ_H is referred to as the activation function.

3.1.3. Activation functions

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

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

In practice however, the simplest model must be generalized by allowing the neuron to respond continuously to its inputs, which is necessary for the optimization algorithms used in the training phase to perform well. To this end one replaces Eq. (1.2) by a general continuous activation function $g(b)$. An example is shown in Figure 1.6. This dictates that the states assume continuous values too, not just the discrete values 0 and 1 as given in Equation (1.1).

There are several choices for the activation function which all come with their 'pros and cons' for a particular application the network is used for. However, there's a few requirements all of these functions should meet:

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

One also needs to distinguish between activation functions used in neurons in the input/hidden layers and neurons in 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 'probabilities' of the input belonging to a certain class. For this one can use commonly used Softmax classifier (function). In the example with hand-written digits, when we feed the network with an image showing digit 7, we want the network to spit out the 100 percent probability for the image belonging to the class 'digit 7' and 0 percent probability of it belonging to the classes 'digit 0', 'digit 1'... In such case, the output of the last layer of the network not only needs to be within 0 and 1, but in case of using Softmax the sum of all outputs must give unity as they are interpreted as (relative!) probabilities. We say 'relative' because the network's decision is only based on the features of this particular image differing in comparison with other data we fed in during training and does not reflect 'outer' probability at all.

Another possibility for the output activation function is the sigmoid function.

- **Sigmoid**

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

This function used to be broadly used mainly thank to the clear interpretation of the state of the neuron - active/inactive is represented by values 0/1. Sigmoid is currently obsolete for large networks. In short, it does not have optimal properties for the learning algorithm called 'backpropagation', which is to be discussed in next chapters. Also, the fact that its mean value in non-zero doesn't have a positive impact on the learning process either. [Groman]

- **Rectified Linear unit (ReLU)**

$$g(x) = \max(0, x)$$

The authors of this concept found the inspiration in real biological neurons. The idea comes from a model for the relation between the electrical current through the cell membrane into the neuron cell, and the membrane potential. The main message is that there is a threshold below which the response of the neuron is strictly zero, as shown in Figure XY. The derivative of the ReLU function is discontinuous at $x = 0$. A common convention is to set the derivative to zero at $x = 0$.

- **Parametric (leaky) ReLu**

$$g(x) = \max(x, \alpha x)$$

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

- **Output activation function - Softmax**

The Softmax function is only used in output layer. This classifier differs from other activation functions by its dependency on other neurons in the layer

3.1. ARCHITECTURE OF ARTIFICIAL NEURAL NETWORKS

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

Here $b_i^{(L)} = \sum_j w_{ij}^{(L)} V_j^{(L-1)} - \theta_j^{(L)}$ are the local fields of the neurons in the output layer. The constant α is usually taken to be unity. Let's repeat three important properties of softmax outputs: first that $0 \leq O_i \leq 1$. 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.

3.1.4. Multilayer perceptrons

A perceptron is a layered feed-forward network. The left-most is the input layer. To the right follows a number of layers consisting of McCulloch-Pitts neurons. The right-most layer of neurons is the output layer where the output of the network is read out, usually as softmax probabilities. The other neuron layers are called hidden layers, their states are not read out directly.

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

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

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

$$V_j^{(\mu)} = g(b_j^{(\mu)}) \quad \text{where} \quad b_j^{(\mu)} = \sum_k w_{jk} x_k^{(\mu)} - \theta_j \quad (3.1)$$

$$O_i^{(\mu)} = g(B_i^{(\mu)}) \quad \text{where} \quad B_i^{(\mu)} = \sum_j W_{ij} V_j^{(\mu)} - \Theta_i \quad (3.2)$$

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

The reason we use hidden layers is to tackle linearly inseparable classification problems. Linear separability is shown in figure XY, where the input to the network is two-dimensional and we classify the input data into two classes (marked as black and white points in the graph). Classification problem is linearly separable if one is able to draw a single line (single plane in case of three inputs, etc.) to divide the input space into two distinct areas and hence solve the classification task. In general, the curve that separates the space into sub areas each representing a class is called the decision boundary. The

position of the decision boundary is determined by the values of weights and thresholds in the neurons (W and THR). These parameters are found by training the network. In the case shown in Figure XY, the line dividing the 2D space of inputs corresponds to the simplest possible case: a single neuron in the network.

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

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

3.2. Training of artificial neural networks

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

The principle explained above is in the neural networks theory known as backpropagation.

3.2.1. Loss function

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

- **Mean Squared Error (MSE)**

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

MSE is used for regression tasks, when the output layer of the network uses Sigmoid functions. When used, the training suffers from the vanishing gradient problem, which will be discussed later.

- **Negative Log Likelihood**

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

Commonly used for classification tasks in the combination with Softmax classifier. The vanishing gradient problem is less of an issue here (see []).

3.2. TRAINING OF ARTIFICIAL NEURAL NETWORKS

- **Cross Entropy Loss**

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

Very similar to the negative log likelihood loss. The difference is that it works with Sigmoid activation function.

3.2.2. Gradient optimization and backpropagation

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

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

Gradient descent

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

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

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

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

$$\delta W_{mn} = -\eta \frac{\partial L}{\partial W_{mn}} = \eta \sum_{\mu=1}^p (t_m^{(\mu)} - O_m^{(\mu)}) g'(B_m^{(\mu)}) V_n^{(\mu)} \quad (3.3)$$

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

$$\Delta_m^{(\mu)} = (t_m^{(\mu)} - O_m^{(\mu)})g'(B_m^{(\mu)}) \quad (3.4)$$

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

$$\delta w_{mn} = -\eta \frac{\partial L}{\partial w_{mn}} = \eta \sum_{\mu}^p \sum_i^N \Delta_i^{(\mu)} W_{im} g'(b_m^{(\mu)}) x_n^{(\mu)} \quad (3.5)$$

while putting

$$\delta_m^{(\mu)} = \sum_i^N \Delta_i^{(\mu)} W_{im} g'(b_m^{(\mu)}) \quad (3.6)$$

Putting all the above together gives

$$\delta w_{mn} = \eta \sum_{\mu}^p \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta W_{mn} = \eta \sum_{\mu=1}^p \Delta_m^{(\mu)} V_n^{(\mu)} \quad (3.7)$$

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

Algorithm 1 Stochastic gradient descent

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

Vanishing and exploding gradient problem

The reason for using the Negative log likelihood is the fact that during the derivation of the update rule for weights and thresholds, the resulting formula does not contain the term $g'(b)$. This is the biggest drawback of using MSE loss function. When we compute the weight increments using MSE, it turns out that as we go further from the output layer, the term $g'(b)$ accumulates with each next layer. The point is that MSE is often used with Sigmoid activation functions, whose value of derivative drops to a small number in its area of saturation. In a result, we get very small weight increments as we go to the left in the network layers and the training of these layers slows down rapidly. This phenomenon is known as the vanishing gradient problem. One of the ways to address this issue is using activation functions that don't saturate, like ReLUs.

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

3.2. TRAINING OF ARTIFICIAL NEURAL NETWORKS

Stochastic gradient descent - SGD

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

Applying the above, the equation XY slightly changes to

$$\delta w_{mn} = \eta \sum_{\mu=1}^{mb} \delta_m^{(\mu)} x_n^{(\mu)} \quad \text{and} \quad \delta W_{mn} = \eta \sum_{\mu=1}^{mb} \Delta_m^{(\mu)} V_n^{(\mu)} \quad (3.8)$$

Adaptation of the learning rate

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

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

The update rule for weights gets modified to

$$\delta w_{ij}^{(t)} = -\eta \frac{\partial H}{\partial w_{ij}^{(t)}} + \alpha \delta w_{ij}^{(t-1)} \quad (3.9)$$

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

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

Other optimization algorithms

- AdaGrad

AdaGrad [1] is another gradient based algorithm. In previously discussed gradient descent, the parameters were updated with the same learning rate in every step of the algorithm. AdaGrad adapts the learning rate based on the accumulated square of gradients. [see stanford L7]. The problem is that it might get stuck in the saddle points because the size of the steps it takes gets very small as the training goes on. [L7]

- **RMSprop**

Another adaptive algorithm introduced in [2], which addresses the issue discussed in AdaGrad. It prevents the steps to get infinitely small by introducing a decay parameter to suppress the effect of the accumulated square of gradients.

- **Adam**

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

- **AdaDelta**

This algorithm is an extension of AdaGrad and tackles its tendency to drop some of the learning rates to almost infinitely small values. [3]

- Second order optimization?

OVERLOOK

3.2.3. Improving training performance

Initialization of weights and thresholds

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

- Xavier initialization

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

- MSRA initialization

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

3.2. TRAINING OF ARTIFICIAL NEURAL NETWORKS

Overfitting and regularization

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

The problem is illustrated in Figure XY and is called overfitting.

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

$$R_{L2}(w) = \frac{\gamma}{2} \sum_{ij} w_{ij}^2 \quad \text{or} \quad R_{L1}(w) = \frac{\gamma}{2} \sum_{ij} |w_{ij}| \quad (3.10)$$

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

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

Dropout

Dropout is a very simple scheme that helps against overfitting. During training, some random portion of the neurons in the network are ignored for each input pattern/mini-batch with the probability of p . This can be thought of as making the network adapt to the sparsity of the remaining neurons and making their effect on the output spread equally over the network. Another way to interpret this is that we are basically 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. [mehlig][L7]

Pruning

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

Data augmentation

The general rule is that the bigger the training dataset is the better the network generalises. However, expanding dataset manually may be very time expensive which leads

to the idea of expanding it artificially. In image classification tasks, this can be done by randomly cropping, scaling, shifting and mirroring the data. [mehlig] [MEHHLIG 90]

Early stopping

One way of avoiding overfitting is to use cross validation and early stopping. The training data are split into two sets: a training set and a validation set. Although training and validation data are drawn from the same distribution and share the same global features, they differ in details that are not of interest. 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. [citace mehlig]

The training is stopped when the validation energy begins to increase. In Section a precise criterion for this early stopping is introduced, one that works for fluctuating data.

Data preprocessing

For most cases, it is advisable to shift the data so it has zero mean before the training begins. When classifying images for example, one can choose between two ways of doing this: first, by subtracting the mean image (for RGB, we have the image of size $M \times N \times 3$) from the entire dataset or, to subtract so called per-channel mean from every image (it means we have three numbers in total). The reason for doing this is illustrated in Figure XY. Thinking of adjusting the weights in the network as of moving along the decision boundary, it is intuitive that data which are not distributed around the origin will cause the classification success to be very sensitive to weight changes.

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

HYPERPARAMETR CHOICE

transfer learning

4. Bibliography

- [1] ZOUHAR, František. *Návrh konstrukce, řízení a elektroniky pro nestabilní balancující vozidlo* [online]. Brno: Vysoké učení technické v Brně. Fakulta strojního inženýrství, 2011 [cit. 2017-05-21]. Dostupné z: <http://hdl.handle.net/11012/21119>. Diplomová práce. Vysoké učení technické v Brně. Fakulta strojního inženýrství. Ústav mechaniky těles, mechatroniky a biomechaniky. Vedoucí práce Robert Grepl.

5. Seznam použitých zkratek a symbolů

CMU

Carnegie Mellon University

6. Seznam příloh

- Nastavení režimu External mode: *external.txt*