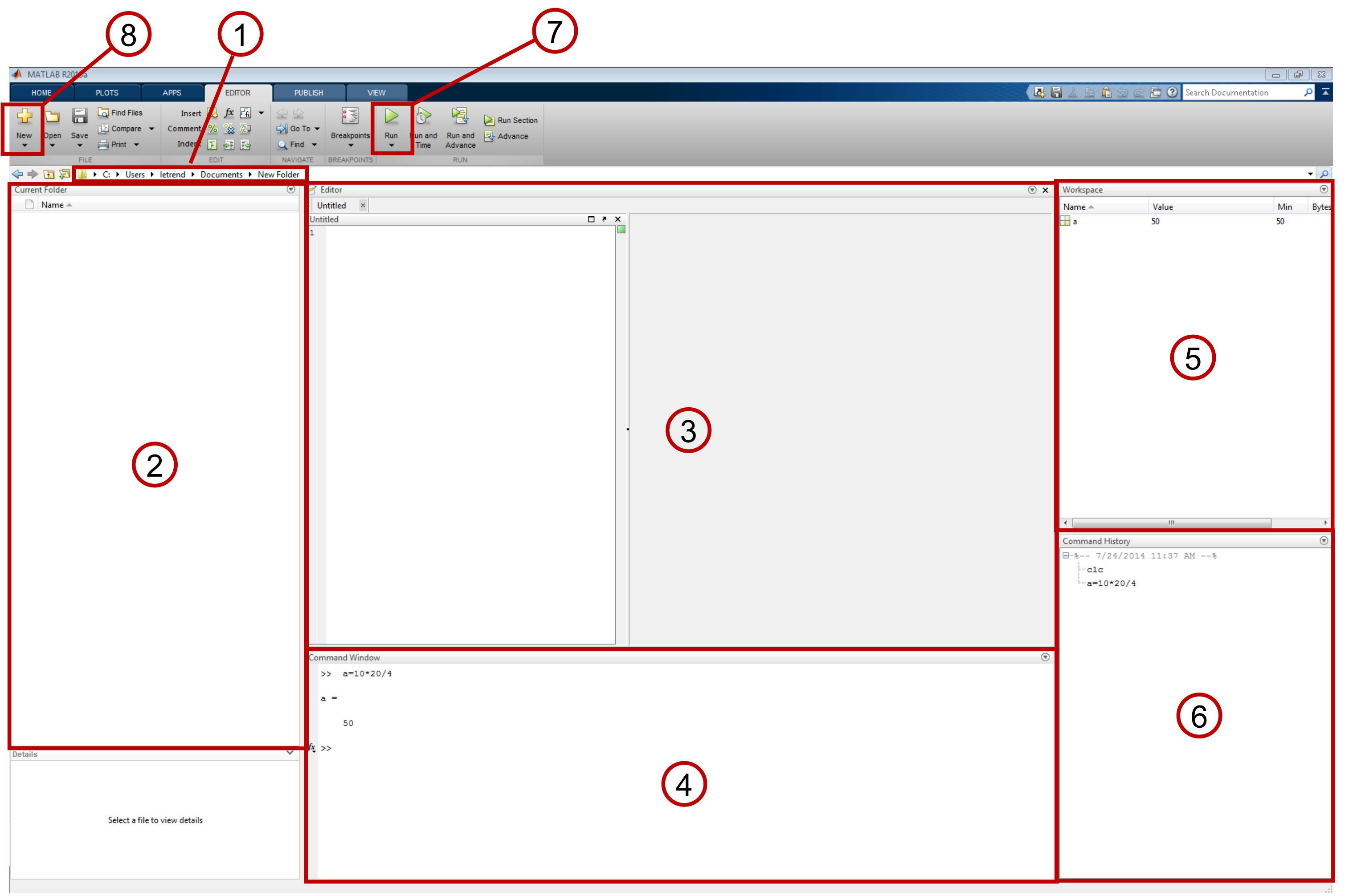
Matlab tutorial

Introduction

This tutorial is meant to kickstart you into matlab coding. Since learning by doing is the best way to learn a programming language you will find 9 short matlab tutorials. For each tutorial there will be some theoretic background you can find here, but also some tasks for you to complete. In the base folder you find the 9 tutorials. In each tutorial most of the code is given, except for spaces saying <YOUR CODE HERE>. In order to complete a tutorial you will have to insert your own code at the corresponding areas. If you're done with that, run the tutorial. In your command window you will see a message whether you did well or something is still wrong.

Tutorial 1

This is actually rather a look up script for basic operations you will need. Lets first explore the Matlab GUI.



```
>> a=10*20/4

a =

    50

>>
```

```
%-- 7/24/2014 11:37 AM --%
clc
a=10*20/4
```

figure 1: Example of Matlab GUI

Tutorial 1

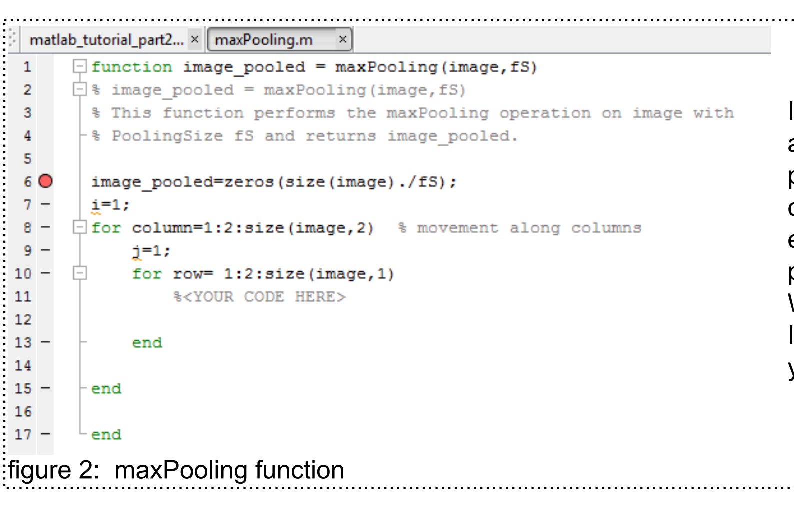
This is actually rather a look up script for basic operations you will need. Lets first explore the Matlab GUI.

① **current folder path**: This is your working folder path. Every script, function or data in here is visible to your current Matlab session.
② **current folder**: Here you can see the content of your current folder
③ **Editor**: This is the text editor for editing your scripts.
④ **Command Window**: You can type your commands in here.
⑤ **Workspace**: This is your Workspace, i.e. a stack where you can see the variables you have used.
⑥ **Command History**: Here you can see the commands you have writtten in the Command Window
⑦ **Run Button**: With this button you can run the sript you have written in the Editor (you can also press F5)
⑧ **New Button**: You can create new scripts with this button.

Now, let's open the first Tutorial in our Editor. Choose the tutorial folder as your current folder and double-click on matlab_tutorial_part1.m in your current folder. In your Editor you now see the first tutorial script. Read through it and follow the instructions.

Tutorial 2

In this tutorial you are going to implement a maxPooling Function, called 'maxPooling.m'. Open this function from your current folder in your Editor. A function in matlab can be implemented by writing 'function' followed by the variables the function returns, followed by '=functionName', followed by the variables the function receives as input. So in the case of our maxPooling function, the function will return 'image_pooled' and receives 'image' and 'fS' as input.



```
1   function image_pooled = maxPooling(image,fS)
2   % image_pooled = maxPooling(image,fS)
3   % This function performs the maxPooling operation on image with
4   % PoolingSize fS and returns image_pooled.
5
6       image_pooled=zeros(size(image)./fS);
7       i=1;
8       for column=1:2:size(image,2)   % movement along columns
9           j=1;
10          for row= 1:2:size(image,1)
11              %<YOUR CODE HERE>
12
13          end
14
15      end
16
17  end
```

figure 2: maxPooling function

of special interest

In figure 2 you see the maxPooling function, but you also see a red circle in line 6. This is a so called Breakpoint (Debug point). You can put Breakpoints anywhere in your code by clicking left of the line you want to put it. When your script is executed the program will pause at the lines with the Debug points, enabeling you to check on the variables in the Workspace or skipping line by line through your code (F10). If you want the program to continue without further annoying you, take out the Breakpoint and press Continue or F5.

Max Pooling consists of repeatedly taking the maximal value of a pooling area. If you evaluate the first cell 'Load Image' you will see a grayscale image in figure 1. If you check out the data for this image, i.e. 'alice' in your workspace, you will see that an image simply consists of intensity values for each pixel in the image. MaxPooling this image consists of simply shifting a Pooling Window of specified size (fS) over the image and saving the respective maximal value to the corresponding place in image_pooled (c.f. figure 3)

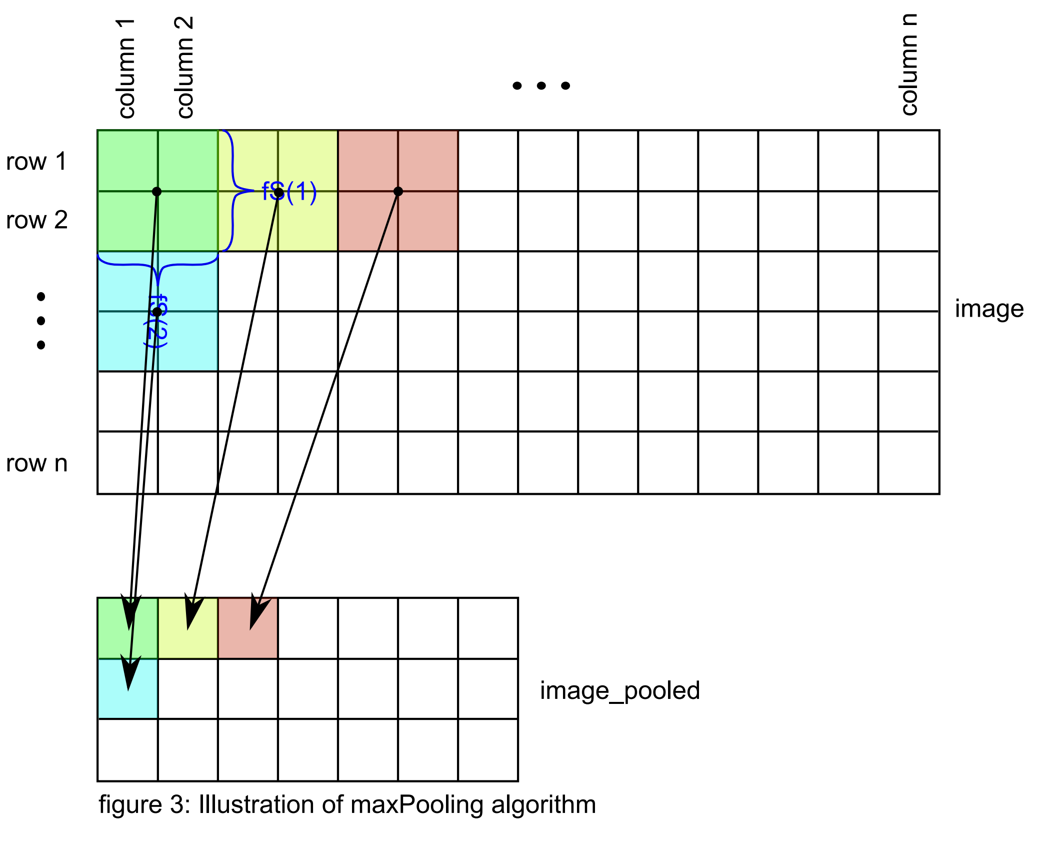


figure 3: Illustration of maxPooling algorithm

As you can see in figure 3 the resolution is being reduced through the maxPooling operation. What we did not care about was, which of the pixels in the pooling window is maximal. This will be dealt with in the following tutorial.

Tutorial 3

In this tutorial you are going to modify your max Pooling function to also save which of the pixels in the pooling window is maximal.

1	3
2	4

On the left you see the indices the way matlab adresses an array and we will adapt this scheme. So if pixel number 3 of the pooling window produced the maximal value, we will save a 3 to a separate array, called indices. Luckily matlabs 'max' function provides the indice as an output parameter (check it out with F1). All we have to do is to call 'max' with a vector containing the pixels of the pooling window ('(:)' should be helpful) and save the maximal value and the corresponding indice.

Checklist:
1. add an array the same size as 'image_pooled' called 'indices'
2. temporarily save the pixels of the pooling window
3. feed this 2x2 array as a vector to the 'max' function
4. make sure you add your 'indices' array as an output variable of your maxPooling function

Tutorial 4

In this tutorial you are going to write a function reversing the maxPooling operation, i.e. a function that places the maximal values at the correct indices. As you can see in figure 4, we want our function to place the maximal values at the indices we saved in our maxPooling function.

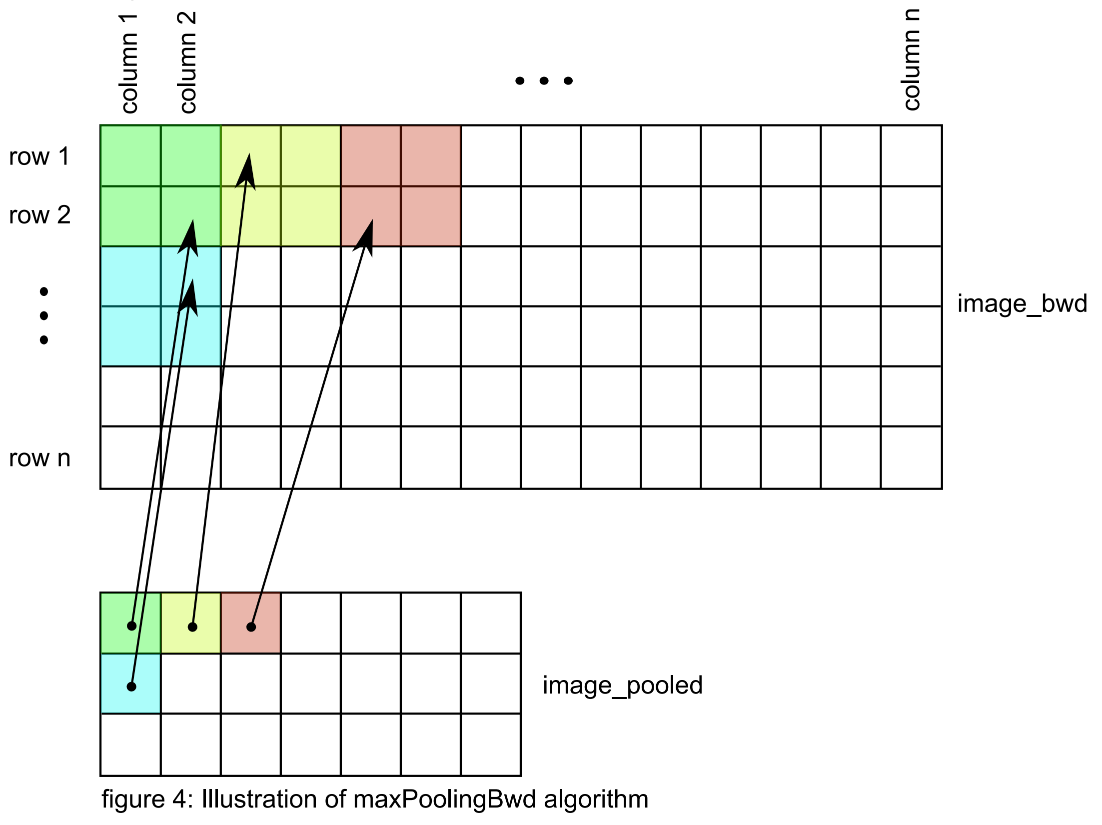
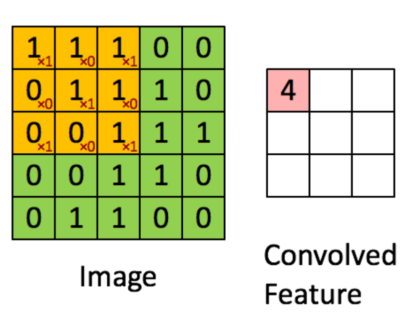


figure 4: Illustration of maxPoolingBwd algorithm

Checklist:
1. add an array the same size as 'image_pooled' called 'indices'
2. temporarily save the pixels of the pooling window
3. feed this 2x2 array as a vector to the 'max' function
4. make sure you add your 'indices' array as an output variable of your maxPooling function

Tutorial 5

In this tutorial you are going to write a function performing convolution. Convolution is a mathematical operation:



$1_{\times1}$	$1_{\times0}$	$1_{\times1}$	0	0
$0_{\times0}$	$1_{\times1}$	$1_{\times0}$	1	0
$0_{\times1}$	$0_{\times0}$	$1_{\times1}$	1	1
0	0	1	1	0
0	1	1	0	0

On the left you see the image that is about to be convolved with a 3x3 kernel [1 0 1;0 1 0;1 0 1] (in yellow/red). The entries of the kernel are multiplied with the corresponding pixel values in image and then summed up. Then the kernel is shifted one pixel and the same steps are applied: sum(elementwise multiplication). This is done until the whole image has been convolved with the kernel.

Checklist:
1. implement the elementwise multiplication of the corresponding pixel values with the kernel
2. sum up the result and write it to 'image_convolved' at the correct place
3. if your convolution works, try out the second kernel and compare the results

Tutorial 6

In this tutorial you are going to implement an artificial Neural Network that is able to separate the data in figure 5.

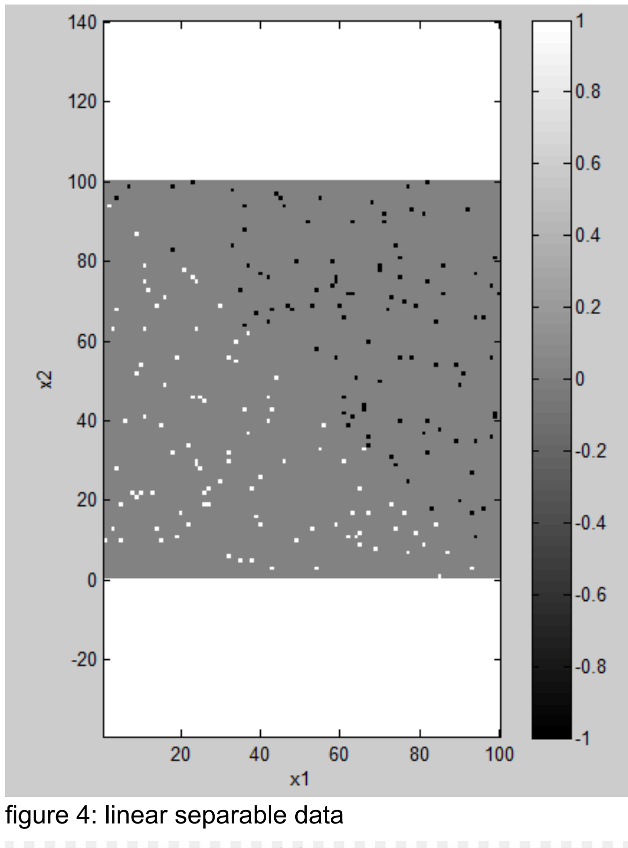


figure 4: linear separable data

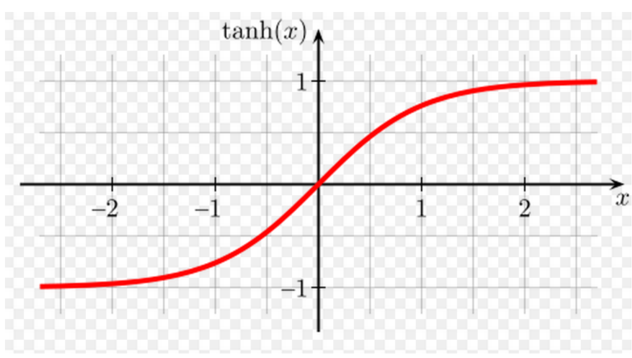


figure 6: activation function tangens hyperbolicus

chain rule of special interest
example f(x) = 1/2(x³+3)²
if we want to derive f(x) with respect to (w.r.t.) x we need to apply the chain rule:
f'(x) = 3x²
becaus f(x) actually composes of two functions:
u(v) = 1/2v²
v(x) = x³+1
f(x) = u(v(x))

Make sure you understand these terms!!!

The data consist of n=200 datapoints $\{(x_{11},x_{21}),\ldots,(x_{1n},x_{2n})\} \in [0,100]$ and their corresponding labels $\{y_1,\ldots,y_n\} \in$ -1/+1. We want to train our network to output +1 for datapoints below the decision boundary $x_2=100-x_1$ and -1 above this line.

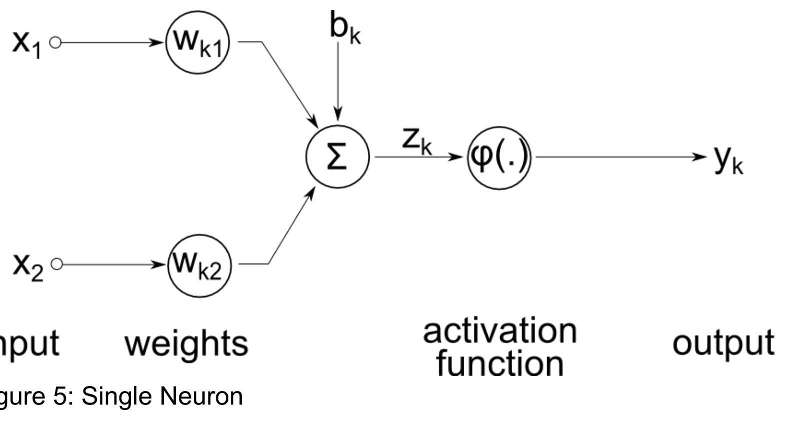


figure 5: Single Neuron

In figure 5 you see a single Neuron which we are going to use to separate the data in figure 4. As you can see the neuron receives as input the x_1 and x_2 coordinates of a datapoint. Each value is then multiplied by a weight (w_{k1}, w_{k2}: at the beginning these weights are simply small random values), the results are summed up, a small value b_k called bias is added and this is then fed into a activation function. The output of this Neuron is y_k. The activation function commonly used is a sigmoid function (e.g. tanh in figure 6). This has the effect of limiting the output to be in the range (-1,1). The output for this neuron can therefor be written as: $y_k=\tanh(wx+b_k)$. Where w is a row vector containing the weights and x is a column vector containing the data coordinates ($w=[w_{k1}\ w_{k2}]$, $x=[x_1;x_2]$). Because at the beginning we have initialized the weights and the bias with small random values the output of the network will be a random number between (-1,1). We can now calculate the difference between the output of the network and the value we want it to output for the given datapoint. We call the value we want it to output *target* and the difference *error*. So the error for a datapoint is: $error=y_k-target$. In order to adjust the weights so that the network outputs what we want, we somehow need to find out which weight is responsible for the error we calculated at the end of the network. This is done through Error Back Propagation, which is basically just the repetitive application of the chain rule (see box of special interest). Lets define a **Error function** $E=1/2error^2$ which is called squared error out of obvious reasons. If we want to find out which weight is responsible for the error we can simply derive this Error function w.r.t. the weights: δE/δw. Becaus the value of E is a composition of multiple functions, we need to apply the chain rule. The functions involved are the **scalar product wx**, the **activation function tanh** and the **square** in the Error function. Let's write out these functions for our network. If you look at figure 5 again, the result from the scalar product is saved as $z_k = wx + b_k$, the output is $y_k = \varphi(z_k)$ and the Error function is $E = 1/2\ (y_k - target)^2$. Obviously the E is a composition of these functions, so in order to derive E w.r.t. to w we can apply the chainrule as follows:

$$\frac{\partial E}{\partial w_{km}} = \frac{\partial E}{\partial y_k}\ \frac{\partial y_k}{\partial z_k}\ \frac{\partial z_k}{\partial w_{km}}$$

$$= \boxed{(y_k - target)}\boxed{\varphi'(z_k)}\boxed{x_m}$$

What we have calculated through this is actually the gradient of our Error function w.r.t. our weights. This gradient points locally from the point we are into a direction that if we followed it would increase the error. This is illustrated in figure 7. Here you see a surface plot of the Error function w.r.t. our weights for the data point x=(1,1), y=1. We have initialized the weights randomly at first, so let's say we start at Point A in figure 7. The gradient δE/δw points into a certain direction increasing the error. Therefor we go in the opposite direction to decrease the error: -δE/δw. Ok so far so good, now all we need to do is to adjust our weights for each data point with the corresponding gradient (for easier notation we will from now on call this gradient Δw). This is referred to as **stocastic gradient descent**:

for iterations
  1. feed forward (randomly selected data point)
  2. back propagate (with corresponding label)
  3. apply gradient
    w = w - η Δw
end

In the application of the gradient you see a new variable η which is called learning rate and chosen to be 1>η>0.

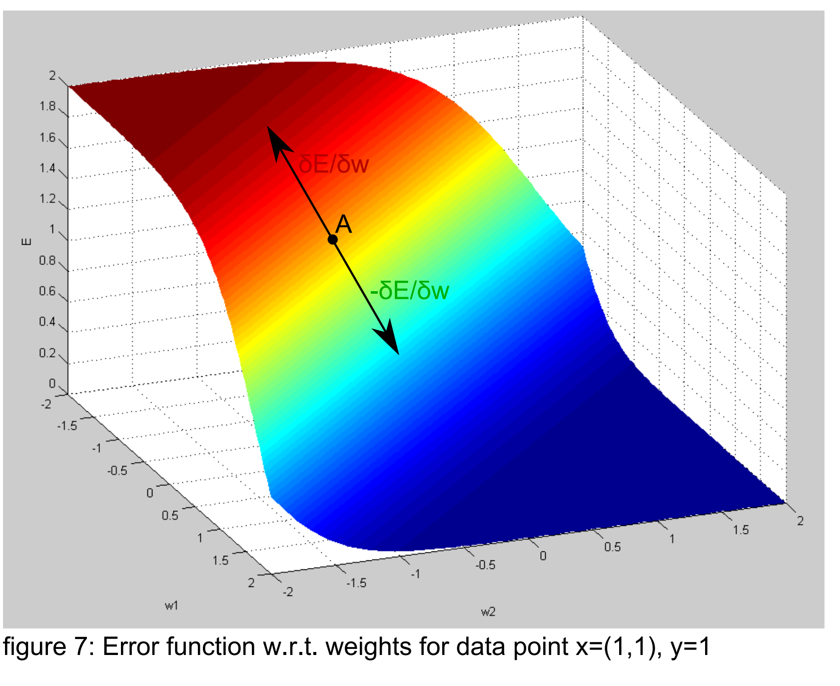


figure 7: Error function w.r.t. weights for data point x=(1,1), y=1

Checklist:
1. edit the feedForward function
2. edit the backProp function
3. apply the gradients inside the learning for loop

hints:
You need to push the average error below 0.001 to suceed in this tutorial. There are three ways of achieving this (adjusting the learning rate η, increasing the training iterations, adjusting the activation function). While the first two are pretty straight-forward, adjusting the activation function is rather involved. Think about the limits of the tanh function we are using, is it even possible with this function to get an error of 0?

batch training of special interest
instead of randomly choosing a training example at each iteration, we could propagate all our data and find the "true" gradient for the whole data. Even though this seems to be a good idea, in most cases the training data is prohibitively large and stocastic gradient descent has proven to lead to faster convergence.

Tutorial 7

Ok, that's pretty awesome. We have used an artificial neuron to classify our data. Before we use more neurons we will check whether the gradients we have calculated are really correct. This is going to proof very helpful later on, when we are using bigger neural networks or even more complex architectures. The method to check on your gradients is called **Numerical Gradient Check** and as the name already suggests it involves calculating the gradients numerically instead of analytically. Take for instance the function $f(x)=x^2$. As you know the analytical gradient is obviously: $f'(x)=2x$. Calculating the gradient numerically can only be done for one point x_0 of the function at a time. What we do is, we calculate the function value for x0 , then we add a small ε on x0 and calculate the function value for this as well. Then we substract the two values from each other and divide by ε. This is called one-sided gradient:

$$f'(x) = \frac{f(x_0)-f(x_0+\varepsilon)}{\varepsilon}$$

one-sided numerical gradient

We are going to use the two-sided numerical gradient because it delivers more precise results. For functions depending on multiple variables, the partial derivatives can be found by holding all variables fixed and adding and substracting the ε only on one variable. ε should be chosen to be quite small to make the numerical gradient as precise as possible. Due to roundoff errors when choosing ε to small, you shouldn't make it too small, 1e-4<ε<1e-7 is a good range.

$$f'(x) = \frac{f(x_0+\varepsilon)-f(x_0-\varepsilon)}{2\varepsilon}$$

two-sided numerical gradient

Checklist:
1. implement two-sided numericalGradientCheck function
2. vary ε and check how this affects the resulting numerical gradient precision
3. check the gradients from tutorial 6 numerically

Tutorial 8

Wow your doing great. So now let's get to some interesting data and bigger networks. In this tutorial you are going to implement a neural network with one so-called hidden layer.

In figure 8 you can see a so-called multi layer perceptron (MLP). As you can see this network still receives the two inputs x1 and x2 but now in the first layer we are using three neurons, each neuron processing the input separately with their own weights. Also each neuron in such a network has its own bias b. As you can see the output from the three neurons in the first layer is then processed by a single neuron in the second layer which produces the final output of the network. Because the input singals are often referred to as the input layer, the first layer in figure 8 is called hidden layer, while the second layer is also called output layer. So what you see in figure 8 is a single-hidden-layer neural network with two inputs, three hidden neurons and one output. The general architecture of MLPs can be seen in figure 9. These MLPs are universal approximators, meaning they can learning any smooth function provided we are using enough neurons (think about that for a second...awesome).

In this tutorial you will teach a MLP a sinus function. But before that we need to check on the back propagation algorithm again, because for more layers it needs to be adapted. We will go backwards through the MLP in figure 8. At each number a caculation is performed.

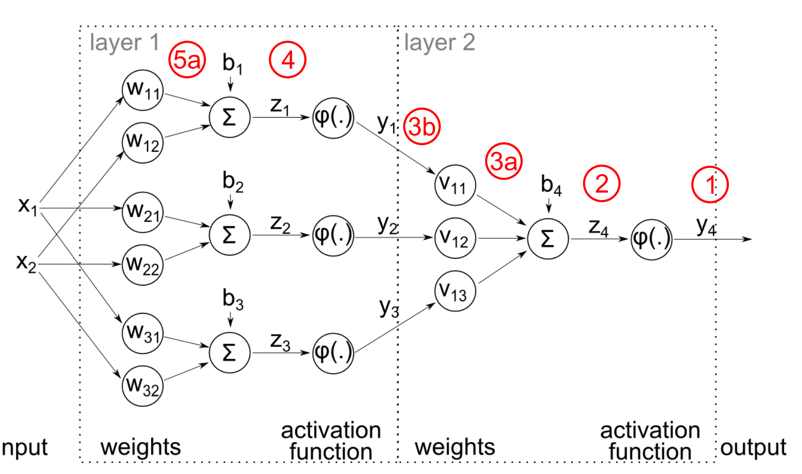


figure 8: multi layer percetron (MLP)

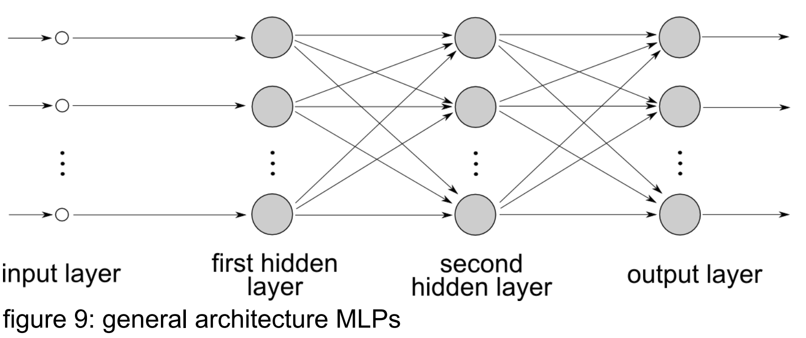


figure 9: general architecture MLPs

① at this point we derive the **cost function** E w.r.t. y_4 and get the $error_2 = y_4$-target

② here we derive the **activation function** φ w.r.t. z_4 and get $\varphi'(z_4)$. We define $\delta_2=error_2.*\varphi'(z_4)$. From this we get the gradient for the bias b_4: $db_4=sum(\delta_2)$. (In case of stocastic gradient descent (SGD) where we present one sample to the network at a time $db_4=\delta_2$)

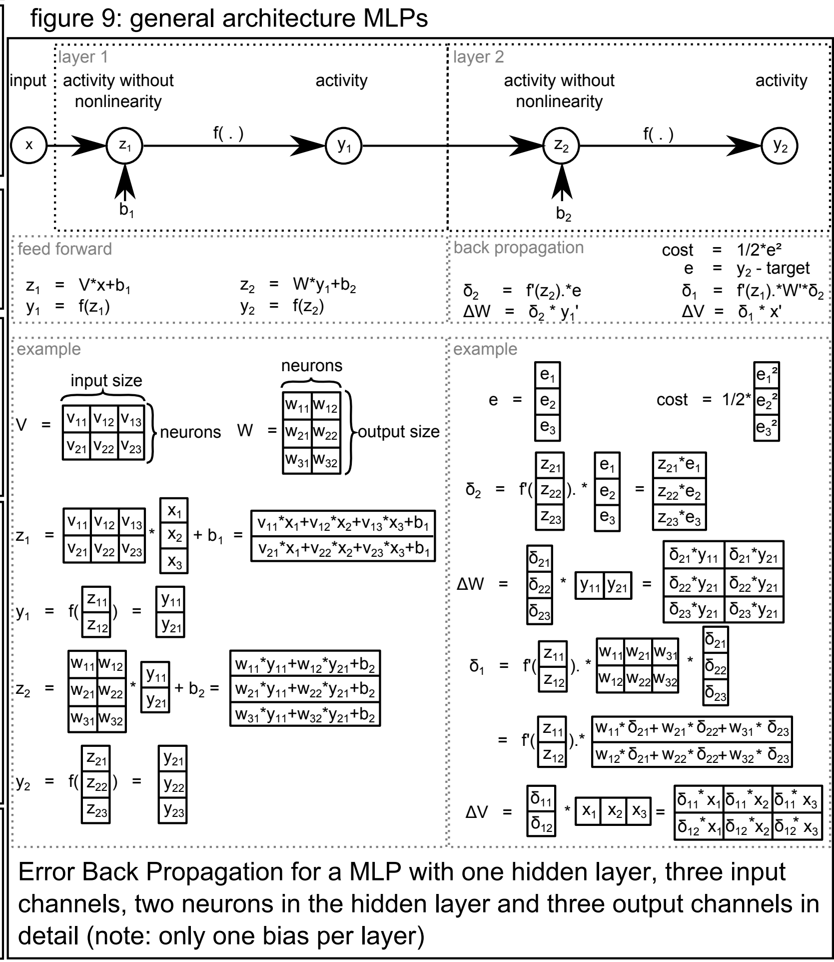
(3a) at this point we get the gradients for the weights in the second layer:
  $dv=\delta_2*y'$ $([dv_{11}\ dv_{12}\ dv_{13}]=\delta_2*[y_1\ y_2\ y_3])$

(3b) Now comes the difference to a single neuron. Because the error at each neuron in the first layer is split depending on the weights v in the second layer, we need to multiply δ_2 with the conjugated weightmatrix v:
$error_1=v'*\delta_2$ $(error_1=[v_{11};v_{12};v_{13}]*\delta_2)$

④ the following steps are again equivalent to the steps 2 and 3a. We derive the **activation function** φ w.r.t. z_1,z_2 and z_3 and define the delta for layer 1:
$\delta_1=error_1.*\varphi'(z_1,z_2,z_3)$ $(\delta_1=[v_{11}*\delta_2*\varphi'(z_1);$
$v_{12}*\delta_2*\varphi'(z_2);$
$v_{13}*\delta_2*\varphi'(z_1)])$
From δ_1 we get the three bias values:
$[db_1;db_2;db_3]=sum(\delta_1)$
(in case of SGD this is again just δ_1)

(5a) and finally we get the gradients for the weights in the first layer:
  $dw=\delta_1*x'$
$([dw_{11};dw_{12};dw_{21};dw_{22};dw_{31};dw_{32}]=\delta_1*[x_1\ x_2])$

-
- this could go on for further layers
-



Error Back Propagation for a MLP with one hidden layer, three input channels, two neurons in the hidden layer and three output channels in detail (note: only one bias per layer)

Checklist:
1. implement feed forward for the given architecture
2. implement back propagation
3. apply the gradients inside the learning for loop
4. adjust the training parameters to push the average error < 0.015

Tutorial 9

Ok, teaching a neural network a sine function is still not too exciting, so let's get to some image classification task. Imagine for example we wanted to train our neural network to detect Waldo in an image like in figure 10.

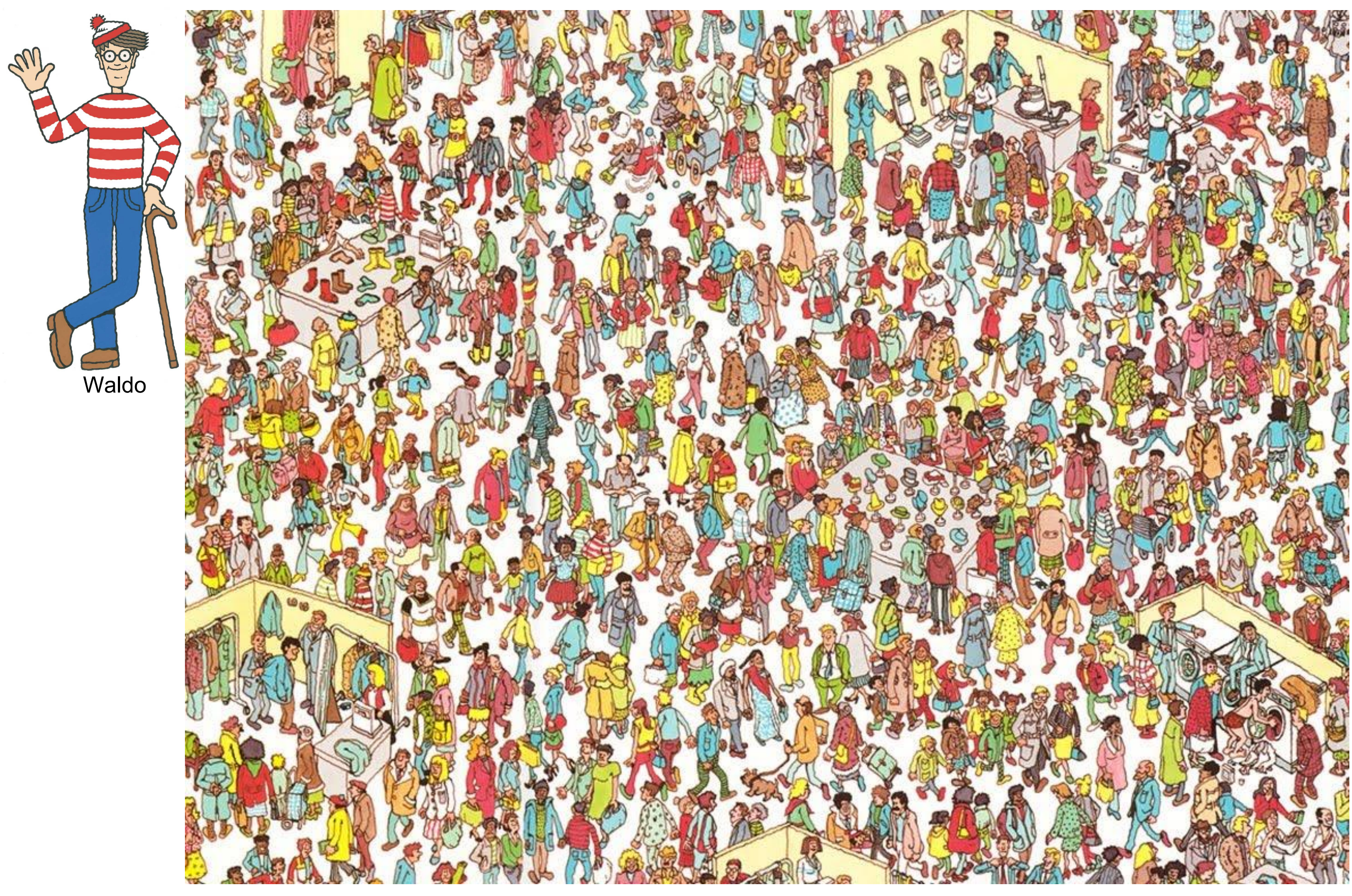


figure 10: find Waldo

Because such an image consists as you already know of pixel values, we could shift a search window over this image, and feed the pixel values from each of these patches into a MLP which we train to output 0 if there is no Waldo in the current position of the search window or 1 if Waldo is there. And this is exactly what you are going to implement in this tutorial, though on easier input data. Your task is to train a MLP on the 20x20 pixel patches you see in figure 11. You may wonder how to feed 2D data into a MLP, well simply make it a 1D vector. In case of the 20x20 pixel patches this vector would be of size 400x1. Before you start training take a look at the pixel values of the different patches (think about the activity function and how you could possibly adjust the pixel values for successful training, check out normalizeImage.m). After you have trained your network edit the scanImage.m function. Figure out how you could visualize the detection of the targets in the image.

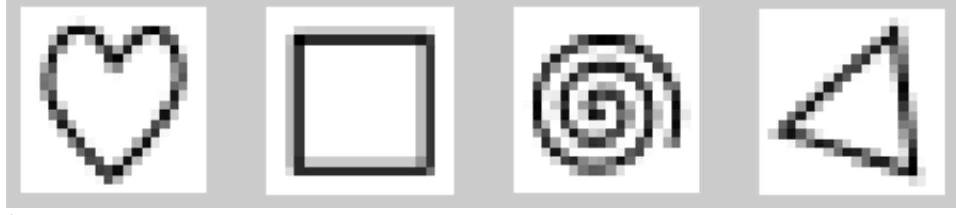


figure 11: training data