02456 Deep Learning Exercise 1 Pen and Paper

Hoorcollege

NN's

- De hidden layers zijn hidden omdat ze niet direct te interpreteren zijn.
- De σ functie is nodig omdat het anders gewoon een heel complexe notatie zou zijn voor een lineair model! We gebruiken de ReLU ($\max(0, x)$)
- Softmax: als het non-negative is, sommeert de noemer naar 1.
- Conventie: talk about l layers als er l **hidden layers** zijn.
- Je kan eender welk neural network approximaten door meer artifical neurons toe te voegen aan 1 hidden layer.

Stochastic Gradient Descent

- Stel de weights in het begin random in
- Small step in steepest downhill direction and iterate

Contents and why we need this lab

The first lab is all about understanding the mathematical concepts behind neural networks. You will derive everything by hand this first week. The hand-in this week is your modifications to this notebook.

In week two you will program what you have derived this week in NumPy and in week three and onwards you will work in a framework dedicated to making it easy to write deep learning code. In this course we will use PyTorch because it is more Python-like than for example TensorFlow. However, TensorFlow has some advantages if you want to deploy deep learning models. For learning and research PyTorch is right now the preferred framework. But this is a quite dynamic field where a lot is happening, so it is hard to say what is the preferred framework a year from now.

Linear algebra, probability theory, statistics and optimization are all underpinning deep learning. The availability of coding frameworks for machine learning is to some degree hiding this and making it possible to make quite impressive applications without deep understanding of the underlying mathematical concepts. This is both a blessing and a curse. The ambition of this course is to educate first class deep learners that can develop deep learning solutions tailored to the problem at hand. Therefore, we need the fundamental understanding. We have experienced that neglecting the fundamental will always come back to hunt you later when we encounter real world problems. So sharpen your pen and get ready to learn new stuff or, more likely, refresh some things you forgot you knew years ago. :-)

List of contents

External sources of information

1. Book. The notation will to a high degree follow that of Chris Bishop, Pattern recognition and machine learning. This book is freely available online and is still a very valuable source of information on machine learning.
2. Jupyter notebook. You can find more information about Jupyter notebooks here. It will come as part of the Anaconda Python installation.
3. Markdown. Jupyter notebook's uses cells. A cell is executed by pressing the Run tab above. In the lab you will only use Markdown cells. You add cell by pressing + tab above of choosing it from the Insert menu also above. A good overview of Markdown formatting can be found here. For equations Markdown uses latex.
4. Mathematics for machine learning is a book that where the title says it all.

Neural networks – the feed forward model

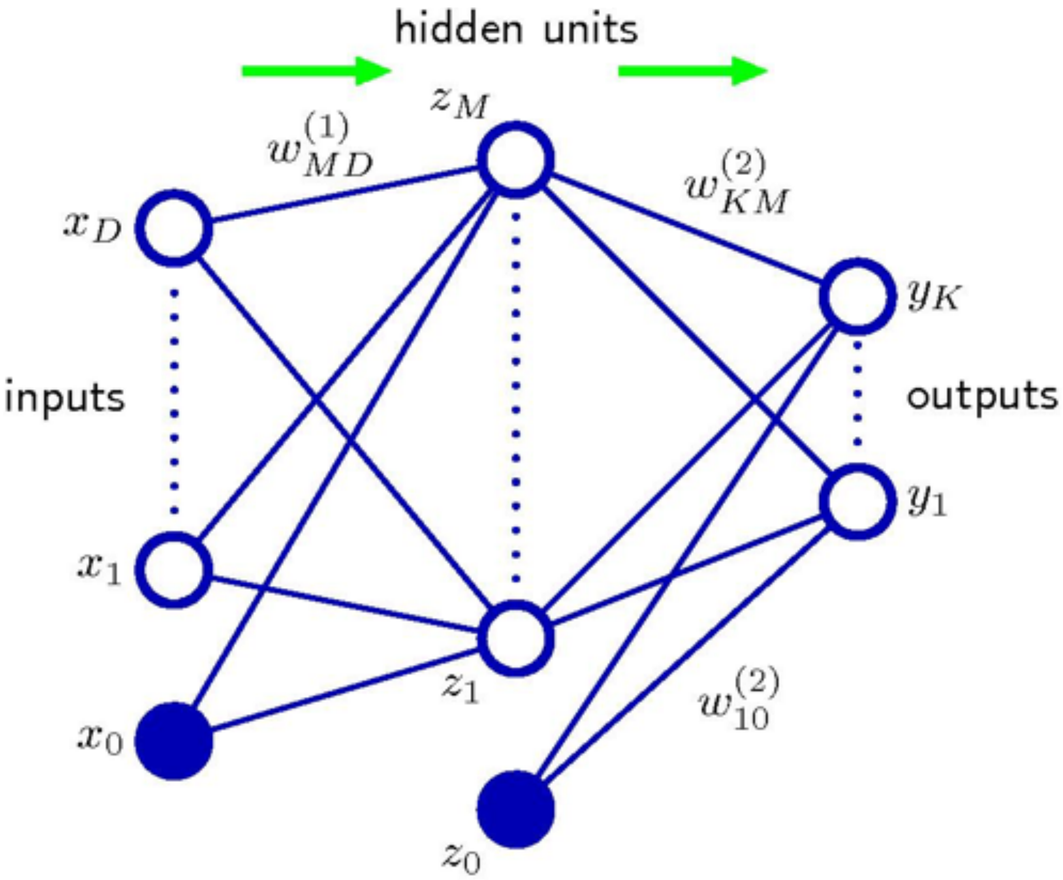
We will meet different variants of neural network models throughout the course. We need different architectures because they have different type of symmetries that reflect data we encounter with different spatial and temporal invariances.

To get started we work with the most basic variant: the so-called *feed-forward neural network* (FFNN). A FFNN with one hidden layer with M hidden unit and two layers of adaptable parameters (weights) denoted by $w = w^{(1)}, w^{(2)}$ is plotted below. In literature sometimes θ will be used instead of w.

In [ ]:
```
from IPython.display import Image
root = "https://raw.githubusercontent.com/DeepLearningDTU/02456-dee

# Figure 5.1 from Bishop
Image(root + "figures/Figure5.1.jpg", width=500)
```

Out[ ]:



We are considering supervised learning where we both have inputs (covariates) and associated outputs (labels, targets, response variables) available. The network has D inputs: $\mathbf{x} = x_1, \ldots, x_D$ and K outputs $\mathbf{y} = y_1, \ldots, y_K$. The algorithm is called a neural network because the architecture of the model resembles the biological network between neurons in our brains.

This computation is layered. Each layer first computes what is equivalent to the output a_j of a linear statistical model and then applies a non-linear function to the linear model output. For the first layer, which takes the network input x as input, these two steps look like this

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{1}$$

$$z_j^{(1)} = h_1(a_j^{(1)}) \, , \tag{2}$$

where h_1 is the non-linear function in the first layer. We can get rid of writing the so-called bias $w_{j0}^{(1)}$ explicitly by adding an extra input x_0 that is always set to one and extending the sum to go from zero:

$$a_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \ . \tag{3}$$

We will use this notation in the following for all layers and just write for example $\sum_i \ldots$ to be understood as $\sum_{i=0}^{D} \ldots$.

The second layer takes the output the of the first layer as input:

$$a_j^{(2)} = \sum_{i}^{M} w_{ji}^{(2)} z_i^{(1)} \ .$$

The second layer non-linear function is denoted by h_2 so the output of the network is

$$y_j = h_2(a_j^{(2)})$$

This gives an example of how the neural network model input to output mapping can be specified.

Let us do a few exercises, where you complete the equations by replacing $\ldots$ with the correct expressions.

Exercise a)

a) Write y_j directly as a function of x. That is, eliminate the a's and z's:

$$y_j = h_2 \left[\sum_{i_1=0}^{D} w_{ji_1}^{(2)} \cdot h_1 \left(\sum_{i_2=0}^{M} w_{i_1 i_2}^{(1)} x_{i2} \right) \right]$$

Exercise b)

b) Write the equation for a neural network with two hidden layers and three layers of weights $w = w^{(1)}, w^{(2)}, w^{(3)}$. Again, without using a's and z's.

$$y_j = h_3 \left[\sum_{i_1=0}^{M} w_{ji_1}^{(3)} \cdot h_2 \left(\sum_{i_2=0}^{M} w_{i_1 i_2}^{(2)} \cdot h_1 \left(\sum_{i_3=0}^{D} w_{i_3 i_2}^{(1)} x_{i3} \right) \right) \right]$$

Exercise c)

- L = aantal layers
- D = aantal inputs
- K = aantal outputs
- M = 'breedte' van elke layer

c) Write the equations for an FFNN with L layers as recursion. Use l as the index for the layer:

$$
\begin{aligned}
y_j &= z_j^{(L+1)} && && (4)\\
z_j^{(l)} &= h_l\left(a_j^{(l)}\right) && l = 1, \dots, L && (5)\\
a_j^{(l)} &= \sum_{i=0}^{M} w_{ji}^{(l)} a_i^{(l-1)} && l = 2, \dots, L && (6)\\
a_j^{(1)} &= \sum_{i=0}^{D} w_{ji}^{(1)} x_i && && (7)
\end{aligned}
$$

Exercise d) – optional

Optional exercises do not give extra credits but are included to give you an opportunity to dive a bit deeper. You can omit them for the first run through the lab and then return to them later on when the other exercises are completed.

d) Do we really need the non-linearities? Show that if we remove the non-linear functions h_l from the expressions above then the output becomes linear in x. This means that the model collapses back to the linear model and therefore cannot learn non-linear relations between x and y.

Yes. If we remove the non-linear functions h_l, the output becomes a linear combination of the inputs x. This is because each layer's output is simply a weighted sum of the previous layer's output, and without the non-linear activation functions, the model can only learn linear relationships. Therefore, the expressiveness of the model is significantly reduced, and it cannot capture complex, non-linear patterns in the data.

Loss functions and maximum likelihood

The FFNN model is quite flexible and can learn to approximate all sorts of functions from training data. Below are some examples of one-dimensional functions learned with a FFNN, i.e. we try to learn a function (the neural network) from a set of points (x, y) that can predict y from input x with as high accuracy as possible. The FFNN we use below consists of one hidden layer with three hidden units, $\tanh$ as the non-linear function in the hidden layer and linear output:

$L = 1, M = 3, D(\text{in}) = K(\text{out}) = 1$

$$y(x) = \sum_{i=0}^{3} w_i^{(2)} \tanh(w_i^{(1)} x)$$

The training data is visualized as the blue dots, the learned function $y(x)$ (only displayed in the interval where we have training data) is the full red line and the dashed lines are the output from the hidden units.

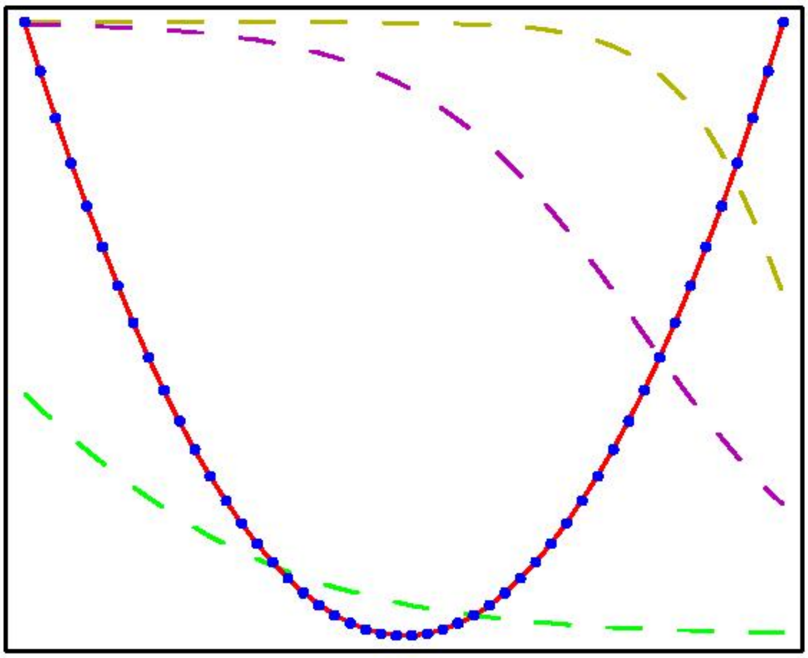
A very nice website to get intuition about how neural networks fit to data is the TensorFlow playground. The example shown is two-dimensional regression: the input x is two-dimensional and the output y is one-dimensional and continuous. Blue encodes high values and yellow low values. You can change many things including switching to classification tasks in the top menu to the right and to other more challenging problems to the left.

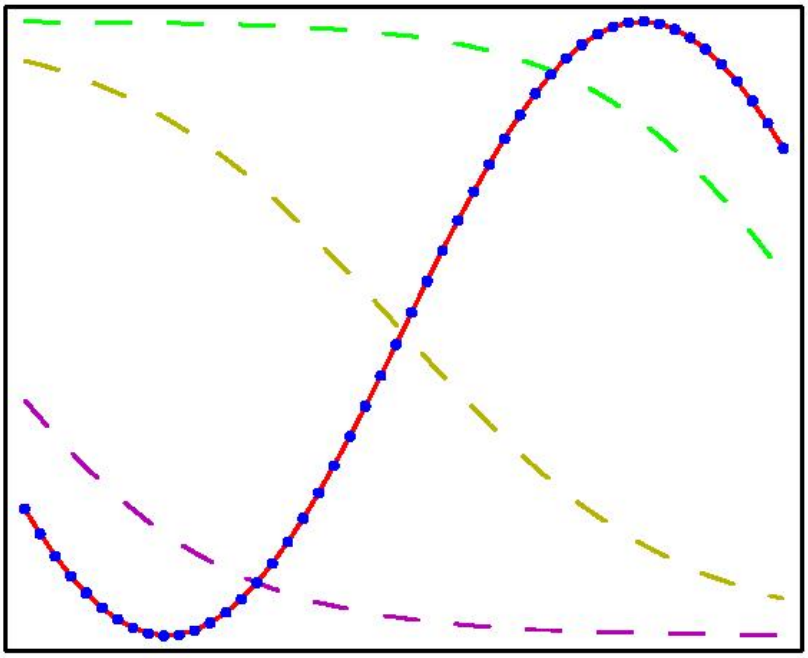
In [ ]:

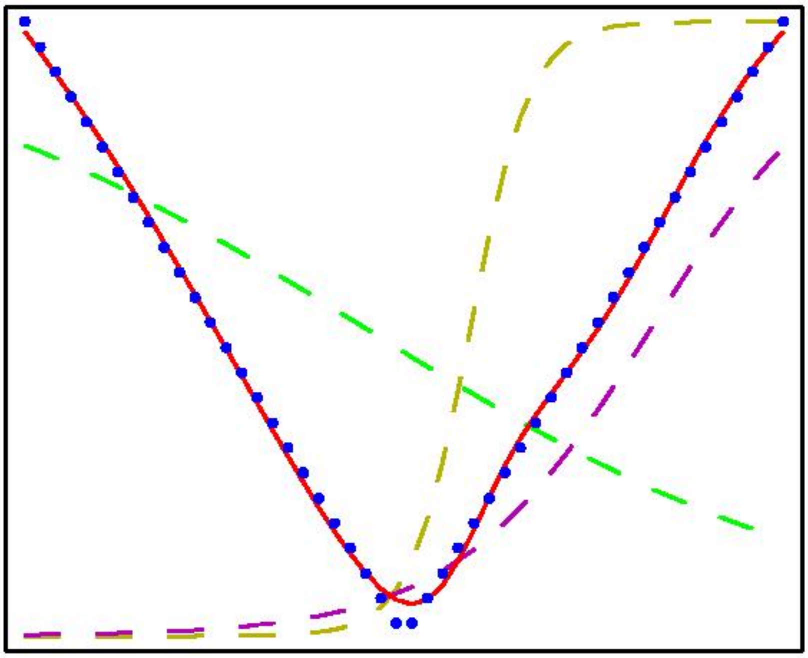
In [3]:

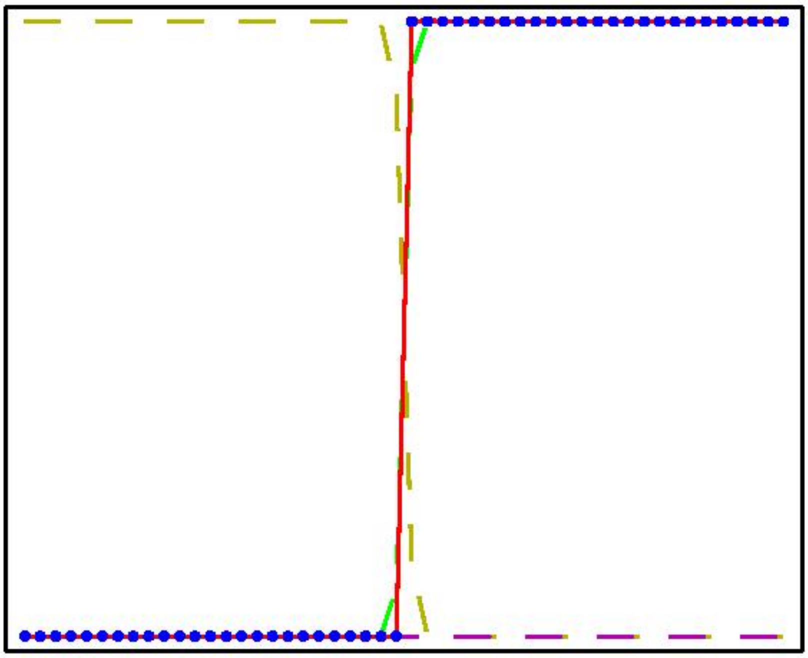
```
# Figure 5.3 from Bishop

from IPython.display import display
display(Image(root + "figures/Figure5.3a.jpg",width=400),\
        Image(root + "figures/Figure5.3b.jpg",width=400),\
        Image(root + "figures/Figure5.3c.jpg",width=400),\
        Image(root + "figures/Figure5.3d.jpg",width=400))
```









This gives some insight into how the neural network uses the hidden units to approximate the curve. But we need to formalize learning and fitting to data. Here the concept of a loss/error function and later on maximum likelihood will come in handy.

The training set $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{t}_n)|n = 1, \ldots, N\}$ consists of N input-output pairs. In the plots above these are all the blue dots.

We can define a loss or error function $E(w)$ as some measure of how far $y(x)$ is from the training data, that is the difference between the red line and the blue dots. As above, w denotes the set of weights of the FFNN, which are the parameters of the neural network we are trying to learn - i.e. we wish to determine a set of weights that results in a low value of the loss function. This is the training error as it is only computed for the training points. In literature sometimes L or J will be used instead of E.

In machine learning we are very much concerned with what the model predicts for inputs which are not in the training set. We often talk about the generalization error which measures the loss function on data points sampled randomly from the data distribution. In general, we do not know the data distribution, so instead we set aside a subset of the data we have - the test set - to compute an estimate of the generalization error. An important skill to have in machine learning is to perform the optimization of the model in such a way that it generalizes well. Much more about that in the coming weeks...

An example of a loss function is the sum of squares:

$$E(w) = \sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n) - \mathbf{t}_n||_2^2 \, ,$$

where $||\ldots||_2^2$ is shorthand for the sum of squared components.

This sum of squares error function is *appropriate for regression* (= targets are continuous) as in the example above. It is *not useful for classification* where targets are discrete class labels. So in order to come up with a proper loss **function for all cases we will appeal to the maximum likelihood**. The likelihood is defined as the probability of the data - in supervised learning the **target** $\mathbf{t}_1, \ldots, \mathbf{t}_N$ - given the model parameters and the inputs:

$$p(\mathbf{t}_1, \ldots, \mathbf{t}_N | \mathbf{x}_1, \ldots, \mathbf{x}_N, w) \ .$$

Maximum likelihood simply says that we should use the set of parameters w that assigns the highest possible probability to the **observed targets** $\mathbf{t}_1, \ldots, \mathbf{t}_N$.

In order to go further from here we need to make some assumptions:

1. Unless we are working with series data - such as time series - we will assume that each data point is independently sampled and each data point is sampled from the same distribution (also known as iid = Independent and identically distributed):

$$p(\mathbf{t}_1, \ldots, \mathbf{t}_N | \mathbf{x}_1, \ldots, \mathbf{x}_N, w) = \prod_{n=1}^{N} p(\mathbf{t}_n | \mathbf{x}_n, w) \ .$$

2. To connect with the sum of squares loss we will assume that the target can be written as the model output plus some random error ϵ and that ϵ has a Gaussian distribution with zero mean and covariance $\sigma^2 \mathbf{I}$. This means that targets themselves are Gaussian distributed with mean $\mathbf{y}(\mathbf{x})$ and covariance $\sigma^2 \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Denoting the Gaussian distribution by $\mathcal{N}$ we can write this as:

$$p(\mathbf{t}_n | \mathbf{x}_n, w) = \mathcal{N}(\mathbf{t}_n | \mathbf{y}(\mathbf{x}_n), \sigma^2 \mathbf{I}) \ .$$

Exercise e)

e) In this exercise you will show that with the two above assumptions, we can derive a loss function that contains $E(w)$ as a term. Two hints

1. With the used covariance we can write the Gaussian distribution as

$$\mathcal{N}(\mathbf{t}_n | \mathbf{y}(\mathbf{x}_n), \sigma^2 \mathbf{I}) = \frac{1}{\sqrt{2\pi\sigma^2}^D} \exp(-||\mathbf{y}(\mathbf{x}_n) - \mathbf{t}_n||_2^2 / 2\sigma^2)$$

2. In order to turn maximum likelihood into a loss/error function apply the (natural) logarithm to the likelihood objective and multiply by minus one.

Show that the loss we get is

$$\text{LOSS} = \frac{ND}{2}\log 2\pi\sigma^2 + \frac{1}{2\sigma^2}E(w)\ .$$

Further, argue why applying the log and multiplying by minus one is the right thing to do in order to get a loss function. *Hint:* Will the optimum of the likelihood function change if we apply the logarithm?

- No, the optimum won't change when applying the logarithm, because the logarithm is a monotonic function. This means that if we find the maximum of the likelihood function, the maximum of the log-likelihood will occur at the same point.

- Wanneer we het product van de kansen nemen, wordt de kans steeds kleiner en kleiner want die zitten in $[0, 1]$. Dit moeten we dus maximaliseren. Wanneer we logaritmes nemen, worden het sommen en wordt het steeds groter en groter. Dit moeten we dus *minimaliseren!*

Exercise f) – optional

f)

- Show that the optimum (= minimum of the loss) with respect to w is not affected by the value of σ^2.
- Find the optimum for σ^2 as a function of w.

Answers

$$\frac{\partial E(w)}{\partial w} \cdot \frac{1}{2\sigma^2} = 0, \text{dan valt de sigma weg}$$

$$\sigma^2 = \frac{E(w)}{ND}$$

This means that for the problem of finding w, sum of squares and maximum likelihood for the Gaussian likelihood with $\sigma^2\mathbf{I}$ covariance are equivalent.

Classification, one-hot and softmax

We will now turn to classification. In classification, the K-dimensional target vector $\mathbf{t}$ encodes exactly one out of K possible classes. It is convenient to use the so-called one-hot encoding for this, which means that the $\mathbf{t}$ vector will have $K-1$ zeros and a single one at position k if the target for the data point is class k. For example, if we have $K = 4$ and the correct label is class three then $\mathbf{t} = (0, 0, 1, 0)$.

We need to modify the network output $\mathbf{y}(\mathbf{x})$ in order to get a likelihood function. In the example above, the likelihood should be the probability that the model assigns to class three. This means that the output should be probabilities. We can achieve this with the softmax function

$$y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \ ,$$

where a_j is shorthand for the linear output from the last layer: $a_j^{(L)}$. The a vector is what in statistics is called the multinomial link function and they are also called the logits. Combining the one-hot encoding of the target with the probabilistic model outputs we can write:

$$p(\mathbf{t}_n|\mathbf{x}_n, w) = \prod_{k=1}^{K} \left[y_k(\mathbf{x}_n)\right]^{t_{nk}} \ .$$

- Het bovenstaande is één van de producten uit het grote product $\prod_{n=1}^{N} p(\mathbf{t}_n|\mathbf{x}_n, w)$.
- Als ons model bevoorbeeld voorspelt dat $y_3 = 0.64$, dan denkt het model met 64 procent zekerheid dat de input correspondeert met $k = 3 \ (\text{met k} \in \{1, \ldots, K\})$
- Als nu effectief de input moet overeenkomen met de derde klasse, telt die 64 procent mee in het grote product (de likelihood). Alle andere voorspellingen van ons model (de overige 36) voor deze specifieke x-waarden tellen niet mee, want voor die waarden is (door de one-hot encoding) de target gelijk aan 0. In het 'kleine' product worden ze tot de 0-de macht verheven en wordt er dus gewoon maal 1 gedaan.

We can now see that the one-hot encoding selects exactly the right output term and the other terms with $t_{nj} = 0$ with contribute ones because $[y_{nj}]^0 = 1$.

- Inderdaad! :)

Exercise g)

g) Show using the same procedure we used for regression that the loss function for classification is

$$E(w) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_k(\mathbf{x}_n) \ .$$

This is also known as the cross entropy loss.

Stochastic gradient descent

To paraphrase Stan Lee: With great flexibility comes complicated fitting to data! We therefore have to resort to general purpose optimization. Optimization refers to the process of searching for a set of weights w that achieves some good results, such as a low loss on the training set. The loss function is differentiable, so we can use gradient information to guide our search. In deep learning some variant of stochastic gradient descent is almost always used for optimization when the gradients can be computed. **Stochastic here means that the gradient is computed on a subset of the training set.**

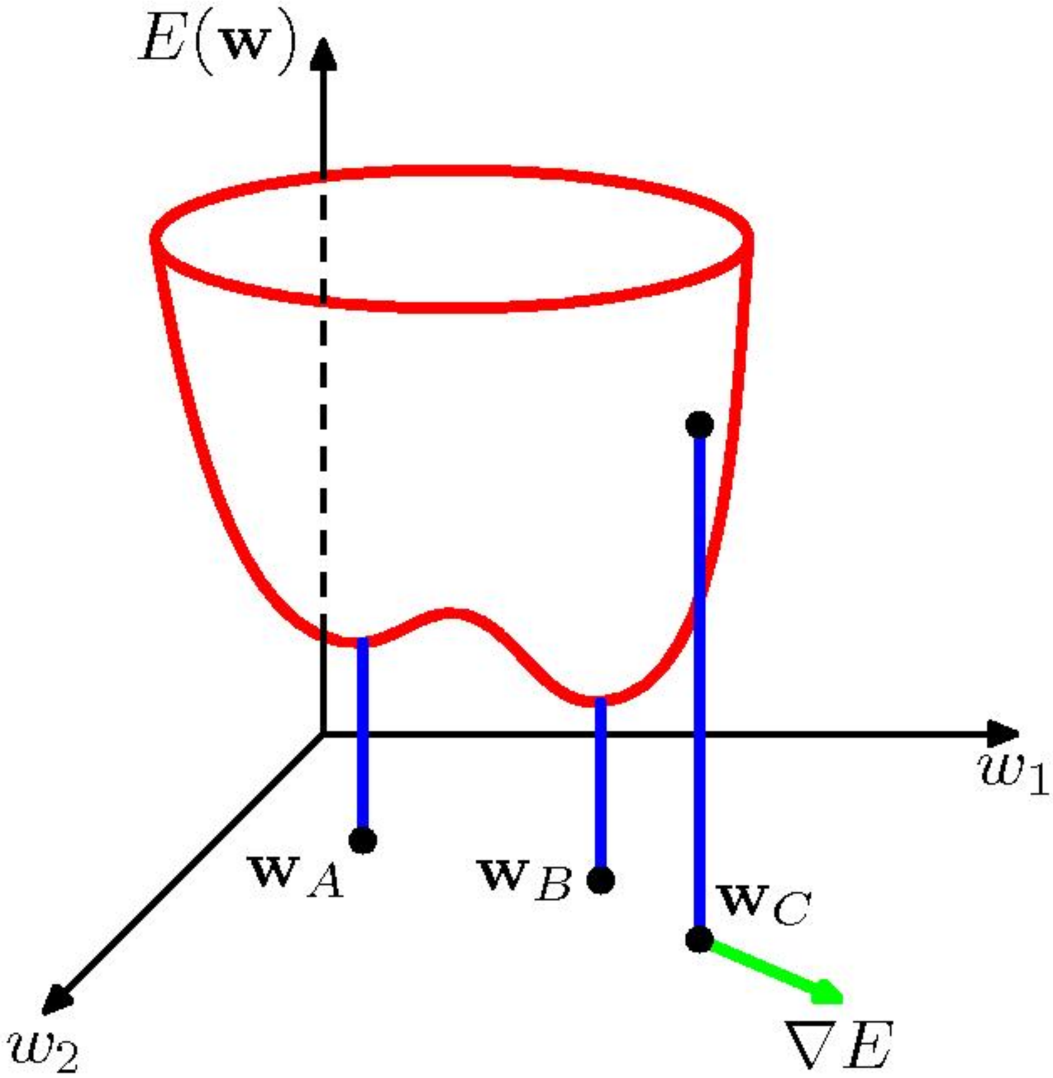
Basic gradient descent means that we take a step with **step-size η opposite the parameter gradient direction**

$$w^{\text{new}} = w - \eta \nabla_w E(w) \ ,$$

where ∇_w is the parameter gradient operator. Applying ∇ to a scalar function like $E(w)$ will produce a vector as output where the j'th component of the vector is the derivative of $E(w)$ with respect to the j'th parameter. A conceptional sketch of the optimization problem is given in the figure below.

In [4]:
```
# Bishop figure 5.5
Image(root + "figures/Figure5.5.jpg",width=500)
```

Out[4]:



Exercise h) – optional

Prove that the gradient calculated on a random subset of the training set on average is proportional to the true gradient.

In [7]:
```
Image("figures/expected.jpg",width=500)
```

Out[7]:

Write the total loss as a sum (or average) over per-example losses:

$$E(w) = \frac{1}{N}\sum_{n=1}^{N} E_n(w),$$

so the full gradient is

$$\nabla_w E(w) = \frac{1}{N}\sum_{n=1}^{N} \nabla_w E_n(w).$$

Let a mini-batch S be a set of m indices sampled uniformly at random from $\{1, \ldots, N\}$.
Define the mini-batch loss as the average over the batch:

$$E_S(w) = \frac{1}{m}\sum_{n\in S} E_n(w),$$

and the mini-batch gradient

$$g_S(w) \equiv \nabla_w E_S(w) = \frac{1}{m}\sum_{n\in S} \nabla_w E_n(w).$$

We consider sampling either with replacement or without replacement; both give the same expectation result.

Expectation of the mini-batch gradient

Take the expectation over the random choice of S. For sampling with replacement it is easiest: each draw picks index n with probability $1/N$. The expected gradient of a single sampled example is

$$\mathbb{E}[\nabla_w E_i(w)] = \sum_{n=1}^{N} \frac{1}{N} \nabla_w E_n(w) = \frac{1}{N}\sum_{n=1}^{N} \nabla_w E_n(w).$$

A mini-batch average of m independent draws has expectation

$$\mathbb{E}[g_S(w)] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} \nabla_w E_i(w)\right] = \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}[\nabla_w E_i(w)] = \frac{1}{m} \cdot m \cdot \frac{1}{N}\sum_{n=1}^{N} \nabla_w E_n(w) = \frac{1}{N}\sum_{n=1}^{N} \nabla_w E_n(w).$$

But $\frac{1}{N}\sum_{n=1}^{N} \nabla_w E_n(w) = \nabla_w E(w)$. Hence

$\mathbb{E}[g_S(w)] = \nabla_w E(w)$,

i.e. the mini-batch gradient is an **unbiased estimator** of the true gradient.

(If you sample *without* replacement uniformly, the expectation is the same — linearity of expectation and symmetry of sampling give the same result.)

Error backpropagation

So-called error backpropagation is simply a recipe for calculating gradients of layered models such as neural networks. Efficient computation is based upon

1. the chain-rule of differentiation:

$$\frac{\partial f(g(w))}{\partial w} = \left.\frac{\partial f(g)}{\partial g}\right|_{g=g(w)} \frac{\partial g(w)}{\partial w}$$

2. storing intermediate computations

In the following, we will omit writing $g = g(w)$ and instead let it be understood from the context.

Gradient for layer L

We return to the FFNN with L layers. We start by computing the gradient with respect to a weight in the last layer:

$$\frac{\partial E(w)}{\partial w_{ji}^{(L)}} \ .$$

To make life a bit simpler for ourselves, we will assume that our training set consists of one example, so we can drop the training point index n. As an **optional** exercise below, you can put the summation over the training examples back in.

First we observe the w dependence in $E(w)$ is through $a_1^{(L)}, \ldots, a_K^{(L)}$, so we can now apply the chain-rule

$$\frac{\partial E(w)}{\partial w_{ji}^{(L)}} = \sum_{k=1}^{K} \frac{\partial E(w)}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial w_{ji}^{(L)}} \ .$$

We can use that $a_k^{(L)} = \sum_i w_{ki}^{(L)} z_i^{(L-1)}$ to conclude that $\frac{\partial a_k^{(L)}}{\partial w_{ji}^{(L)}} = z_i^{(L-1)}$ when $j = k$ and zero otherwise. So we can write

$$\frac{\partial E(w)}{\partial w_{ji}^{(L)}} = \delta_j^{(L)} z_i^{(L-1)} \ ,$$

where we, with foresight, have defined

$$\delta_j^{(L)} = \frac{\partial E(w)}{\partial a_j^{(L)}} \ .$$

These δ's will become very practical for bookkeeping purposes.

Exercise i)

Calculate

$$\delta_j^{(L)} = \frac{\partial E(w)}{\partial a_j^{(L)}}$$

for classification.

Hint: It is much easier to find the derivative, if we write the loss function directly in terms of the logits $a_j^{(L)}$. So show first using the definition of the loss and the softmax that

$$E(w) = -\sum_{k=1}^{K} t_k a_k^{(L)} + \log \sum_{k=1}^{K} \exp(a_k^{(L)}) \ .$$

Finally show that

$$\frac{\partial}{\partial a_j^{(L)}} \log \sum_{k=1}^{K} \exp(a_k^{(L)}) = y_j$$

to get the final result.

Gradient for layer $L-1$

Now we proceed to the second last layer

$$\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}} \ .$$

This will allow us to identify a pattern that will enable us to generalize to any layer.

We now use that the model is layered so the chain-rule we need is:

$$\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}} = \sum_{k=1}^{K} \frac{\partial E(w)}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial w_{ji}^{(L-1)}} \ .$$

We can now again use $a_k^{(L)} = \sum_i w_{ki}^{(L)} z_i^{(L-1)} = \sum_i w_{ki}^{(L)} h_{L-1}(a_i^{(L-1)})$ and the definition of $\delta_j^{(L)}$:

$$\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}} = \sum_{k=1}^{K} \delta_k^{(L)} w_{kj}^{(L)} h'_{L-1}(a_j^{(L-1)}) z_i^{(L-2)} \ .$$

We can now define

$$\delta_j^{(L-1)} = \sum_{k=1}^{K} \delta_k^{(L)} w_{kj}^{(L)} h'_{L-1}(a_j^{(L-1)}) \ .$$

To write the gradient in the same way as for layer L

$$\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}} = \delta_j^{(L-1)} \ z_i^{(L-2)} \ .$$

If we look a bit closer at the definition of $\delta_j^{(L-1)}$, we see it is actually nothing but:

$$\delta_j^{(L-1)} = \frac{\partial E(w)}{\partial a_j^{(L-1)}} \ .$$

You will use this in the next exercise to derive the backpropagation rule for a general layer $l < L$.

Exercise j) - the backpropagation rule for layer $l < L$

j) Use the above results to argue why the general backpropagation rule for $l < L$ is written as:

$$\frac{\partial E(w)}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)} \tag{8}$$

$$\delta_j^{(l)} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) \tag{9}$$

Exercise k) – optional

k) Derive the backpropagation rule for regression, that is, perform the calculation in exercise i) for the regression loss. Everything else stays the same. Actually it turns out that if you use $h_L(a) = a$ then you even get the same result as in i).

Exercise l) – optional

l) Modify the backpropagation rules to be able to take a dataset of size greater than one.

Hint: This only requires introducing a sum over n and n indices in a few places.

Final comments on backpropagation and what is next

So optimizing a feed-forward neural network with gradient descent is pretty straightforward!

1. You forward propagate with the rules you derived in exercise c), storing the results for the a's and z's.
2. Then you run the backward propagation recursion for the δ's, taking $\delta_j^{(L)}$ from exercise i) and using the recursion from exercise j).
3. You get the gradients from the expression in exercise j).

4. Finally, you update the weights of the network according to the gradients, and the network should now have better performance on the training set.

If you wonder why the forward recursion for a and z is non-linear and the backward recursion for δ is linear, then remember that the derivative is a linear operator).

So what comes next? For the next two weeks, we will stay with the FFNN model. You will implement the forward and backward pass in NumPy yourself next week, and in two weeks, we will see how we can do this in PyTorch. One of the features of modern algebra frameworks like PyTorch and TensorFlow is that we only need to bother about the forward pass. We have finally "taught" computers to differentiate. ;-)

In three weeks+ we will look at other generalizations of the FFNN. They all build upon the same principles as for the FFNN, so the same insights apply.