Problem 1 Linear Algebra Refresher

Rene Wilmes (2862149), David Grömling (1806724)

Problem 1.1 Matrix Properties

Not all properties of scalars are valid for matrices. To proove this, the following definitions are used:

$$A=\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, B=\begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix}, C=\begin{pmatrix} c_{11} & \cdots & c_{1l} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nl} \end{pmatrix}$$

Problem 1.1.1 Commutative

Addition

Show $A+B \stackrel{!}{=} B+A$

Using the commutative law of scalar addition yields:

$$A+B=\begin{pmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{b1}+b_{b1} & \cdots & a_{mn}+b_{mn} \end{pmatrix} \stackrel{scalar}{=} \begin{pmatrix} b_{11}+a_{11} & \cdots & b_{1n}+a_{1n} \\ \vdots & \ddots & \vdots \\ b_{a1}+a_{a1} & \cdots & b_{mn}+a_{mn} \end{pmatrix} = B+A$$

Matrix addition is commutative

Multiplication Show $A*C \stackrel{!}{=} C*A$

$A_{mxn}*C_{ixj}$ only defined for matrices that fulfill $n=i$. Therefore, $C_{ixj}*A_{mxn}$ only possible for $j=m$.

Even in that case, $A*C=C*A$ is not generally true, as the following equation shows.

$$A*C=\begin{pmatrix} \sum_{k=1}^{n} a_{1k}\cdot c_{k1} & \cdots & \sum_{k=1}^{n} a_{1k}\cdot c_{km} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{mk}\cdot c_{k1} & \cdots & \sum_{k=1}^{n} a_{mk}\cdot c_{km} \end{pmatrix} \stackrel{generally}{\neq} \begin{pmatrix} \sum_{k=1}^{n} c_{1k}\cdot a_{k1} & \cdots & \sum_{k=1}^{n} c_{1k}\cdot a_{km} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} c_{mk}\cdot a_{k1} & \cdots & \sum_{k=1}^{n} c_{mk}\cdot a_{km} \end{pmatrix} = C*A$$

Matrix multipllication is NOT commutative.

Problem 1.1.2 Distributive

Show $(A+B)*C \stackrel{!}{=} A*C+B*C$

Using the scalar version of the distributive law yields:

$(A+B)*C$

$$=\begin{pmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{b1}+b_{b1} & \cdots & a_{mn}+b_{mn} \end{pmatrix} * \begin{pmatrix} c_{11} & \cdots & c_{1l} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nl} \end{pmatrix}$$

$$=\begin{pmatrix} \sum_{k=1}^{n}(a+b)_{1k}\cdot c_{k1} & \cdots & \sum_{k=1}^{n}(a+b)_{1k}\cdot c_{kl} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n}(a+b)_{mk}\cdot c_{k1} & \cdots & \sum_{k=1}^{n}(a+b)_{mk}\cdot c_{kl} \end{pmatrix}$$

$$\stackrel{scalar}{=} \begin{pmatrix} \sum_{k=1}^{n} a_{1k}\cdot c_{k1} & \cdots & \sum_{k=1}^{n} a_{1k}\cdot c_{kl} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{mk}\cdot c_{k1} & \cdots & \sum_{k=1}^{n} a_{mk}\cdot c_{kl} \end{pmatrix} + \begin{pmatrix} \sum_{k=1}^{n} b_{1k}\cdot c_{k1} & \cdots & \sum_{k=1}^{n} b_{1k}\cdot c_{kl} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} b_{mk}\cdot c_{k1} & \cdots & \sum_{k=1}^{n} b_{mk}\cdot c_{kl} \end{pmatrix}$$

$$= A*C+B*C$$

Matrix operations are distributive.

Problem 1.1.3 Associative

Addition (In this section, $A,B,C \in \Re^{mxn}$)

Show $A+(B+C)=(A+B)+C$

Using the scalar version of the associative law yields:

$A+(B+C)$

$$=A+\begin{pmatrix} b_{11}+c_{11} & \cdots & b_{1n}+c_{1n} \\ \vdots & \ddots & \vdots \\ b_{c1}+c_{c1} & \cdots & b_{mn}+c_{mn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}+(b+c)_{11} & \cdots & a_{1n}+(b+c)_{1n} \\ \vdots & \ddots & \vdots \\ a_{(b+c)1}+(b+c)_{(b+c)1} & \cdots & a_{mn}+(b+c)_{mn} \end{pmatrix}$$

$$\overset{scalar}{=} \begin{pmatrix} (a+b)_{11}+c_{11} & \cdots & (a+b)_{1n}+c_{1n} \\ \vdots & \ddots & \vdots \\ (a+b)_{c1}+c_{c1} & \cdots & (a+b)_{mn}+c_{mn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{b1}+b_{b1} & \cdots & a_{mn}+b_{mn} \end{pmatrix} + C = (A+B)+C$$

Matrix addition is associative.

Multiplication (In this section: $A \in \Re^{mxn}, B \in \Re^{nxi}, C \in \Re ixj$)

Show $A*(B*C) = (A*B)*C$

At element (i,l) of the product:

$$\sum_{j=1}^{n} a_{ij}\left(\sum_{k=1}^{m} b_{jk} \cdot c_{kl}\right)$$
$$= \sum_{j=1}^{n}\sum_{k=1}^{m} a_{ij} b_{jk} c_{kl}$$
$$\overset{commutative}{=} \sum_{k=1}^{m}\sum_{j=1}^{n} a_{ij} b_{jk} c_{kl}$$
$$= \sum_{k=1}^{m} c_{kl}\left(\sum_{j=1}^{n} a_{ij} \cdot b_{jk}\right)$$

Matrix multiplication is associative.

Problem 1.2 Matrix Inversion

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 1 & 4 & 5 \end{pmatrix}$$

Inversion (Gauss)

$$\left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 1 & 2 & 4 & 0 & 1 & 0 \\ 1 & 4 & 5 & 0 & 0 & 1 \end{array}\right) \overset{(2)=(2)-(1),(3)=(3)-(1)}{->}$$

$$\left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 \\ 0 & 2 & 2 & -1 & 0 & 1 \end{array}\right) \overset{(1)=(1)-(3),(3)=[(3)-2(2)]/2}{->}$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 1 & 2 & 0 & -1 \\ 0 & 0 & 1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0.5 & -1 & 0.5 \end{array}\right) \overset{(1)=(1)-(2),(2)=(2)-(3)}{->}$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 3 & -1 & -1 \\ 0 & -1 & 1 & -1.5 & 2 & -0.5 \\ 0 & 1 & 0 & 0.5 & -1 & 0.5 \end{array}\right) \overset{(3)=(2)+(3)}{->}$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 3 & -1 & -1 \\ 0 & -1 & 1 & -1.5 & 2 & -0.5 \\ 0 & 0 & 1 & -1 & 1 & 0 \end{array}\right) \overset{(2)=-[(2)-(3)]}{->}$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 3 & -1 & -1 \\ 0 & 1 & 0 & 0.5 & -1 & 0.5 \\ 0 & 0 & 1 & -1 & 1 & 0 \end{array}\right)$$

$$A^{-1} = \begin{pmatrix} 3 & -1 & -1 \\ 0.5 & -1 & 0.5 \\ -1 & 1 & 0 \end{pmatrix}$$

$A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 1 & 2 & 5 \end{pmatrix}$ is not invertible because columns 1 and 2 are linearly dependent. In less obvious cases one could also check if the determinant $det(A)$ equals 0. In that case A is not invertable:

$$\begin{aligned} det(A) &= 1*2*4+1*2*5+1*2*3-1*2*4-1*2*3-1*2*5 \\ &= 8+10+6-8-6-10 \\ &= 0 \end{aligned}$$

Problem 1.3 Matrix Pseudoinverse

Definition of right and left pseudoinverse of a generic matrix A with dimension $n \times m$:

$$A_{left}^{-1} = (A^T A)^{-1} A^T, so\ that\ A_{left}^{-1} A = I_m$$

$$A_{right}^{-1} = A^T (AA^T)^{-1}, so\ that\ A_{right}^{-1} A = I_n$$

Note that I_k denotes the $k \times k$ identity matrix.

In order to determine A_{left}^{-1} and A_{right}^{-1} of a matrix $A \in \Re^{2\times 3}$ we can compute the following matrices in the intermediate steps:
For A_{left}^{-1}:

$$\begin{aligned} A^T &\in \Re^{3\times 2} \\ A^T A &\in \Re^{3\times 3} \\ (A^T A)^{-1} &\in \Re^{3\times 3} \\ (A^T A)^{-1} A^T &\in \Re^{3\times 2} \end{aligned}$$

For A_{right}^{-1}:

$$\begin{aligned} A^T &\in \Re^{3\times 2} \\ AA^T &\in \Re^{2\times 2} \\ (AA^T)^{-1} &\in \Re^{2\times 2} \\ A^T (AA^T)^{-1} &\in \Re^{3\times 2} \end{aligned}$$

Problem 1.4 Eigenvectors and Eigenvalues

Eigenvectors v_n of a Matrix M fullfill $\alpha_n v_n = M v_n$ with $\alpha_n \in \Re$. α_n is called Eigenvalue of M. In machine learning one has to cope with a lot of data, which can be seen as a high-dimensional vector space. Dealing with high dimensions can be very ressource intensive. Eigenvalues and eigenvectors indicate which dimensions have a high variance and therefore offer the *most* information about the data, allowing one to rank the dimensions and possibly reduce the vector space by filtering less useful dimensions.

Problem 2 Statistics Refresher

Problem 2.1 Expectation and Variance

1) $E(\Omega) = \sum_{\omega\in\Omega} p(\omega)\omega$
$Var(\Omega) = \sum_{\omega\in\Omega} p(\omega)(\omega - E(\Omega))^2$
The expectation is linear, $E(X+Y) = E(X) + E(Y)$. Because of the quadratic term in the variance, the variance is non-linear.

2)
Dice A:

$E(A) \approx \frac{1}{6}(3 \cdot 4 + 2 \cdot 1 + 1 \cdot 2) = \frac{1}{6} \cdot 16 = \frac{8}{3} \approx 2.67$

$Var(A) \approx \frac{1}{6-1}[3 \cdot (4 - \frac{8}{3})^2 + 2 \cdot (1 - \frac{8}{3})^2 + 1 \cdot (2 - \frac{8}{3})]$
$= \frac{1}{5}[3 \cdot (\frac{4}{3})^2 + 2 \cdot (\frac{5}{3})^2 + (\frac{2}{3})^2]$
$= \frac{1}{5}[\frac{48+50+4}{9}] = \frac{102}{45} = \frac{34}{15} \approx 2.27$

Dice B:

$E(B) \approx \frac{1}{6}(4 \cdot 3 + 4 + 6) = \frac{1}{6} \cdot 22 = \frac{11}{3} \approx 3.67$

$Var(B) = \frac{1}{5}(4 \cdot (3 - \frac{11}{3})^2 + (4 - \frac{11}{3})^2 + (6 - \frac{11}{3})^2)$
$= \frac{16+1+49}{45} = \frac{22}{5} = 4.4$

Dice C:

$E(C) \approx \frac{1}{6} \cdot (2 \cdot 5 + 2 + 3 \cdot 1) = \frac{15}{6} = \frac{5}{3} \approx 1.67$

$Var(C) \approx \frac{1}{5} \cdot (2 \cdot (5 - \frac{5}{3})^2 + (2 - \frac{5}{3})^2 + 3 \cdot (1 - \frac{5}{3})^2)$
$= \frac{1}{5} \cdot \frac{200+1+12}{9} = \frac{71}{15} \approx 4.73$

3)
The most rigged dice is dice C, since its mean differs most from the expected $\frac{1}{6} \cdot \sum_{i=1}^{6} i = 3.5$

Problem 2.2 It's a cold world

a)
has cold: C
has backpain: B

b)
Both random variables are binary: True and false

c)
$P(B|C) = 30\%$
$P(C) = 5\%$
$P(B|\bar{C}) = 10\%$

d)
Looking for $P(C|B) = \frac{P(C,B)}{P(B)}$
$P(B,C) = P(B|C) \cdot P(C) = 30\% \cdot 5\% = 1.5\%$
$P(\bar{C}) = 1 - P(C) = 0.95$
$P(B,\bar{C}) = P(B|\bar{C}) \cdot P(\bar{C}) = 10\% \cdot 95\% = 9.5\%$
$P(B) = P(B,C) + P(B,\bar{C}) = 1.5\% + 9.5\% = 11\%$ $P(C|B) = \frac{P(C,B)}{P(B)} = \frac{1.5\%}{11\%} = 13.6\%$
The chance that some with a backpain has a cold is 13.6 %.

Problem 2.3 Deadly Virus

$$s_{t+1} = \begin{pmatrix} 0.35 & 0.015 \\ 1-0.35 & 1-0.015 \end{pmatrix} \cdot s_t = \begin{pmatrix} 0.35 & 0.015 \\ 0.65 & 0.985 \end{pmatrix} \cdot s_t$$

Problem 3 Optimization and Information Theory

Problem 3.1 Entropy

Definition of Entropy: $H = -\sum_n p_n log_2(p_n)$
With sample data: $H = -0.01 \cdot log_2(0.01) - 0.65 \cdot log_2(0.65 - 0.25 \cdot log_2(0.25)) - 0.09 \cdot log_2(0.09) bit$
$= 1.2831 bit$

Since only full integer numbers of bits can be transmitted, the minimum number of required bits to transmit one symbol is ceil(1.2831)bit = 2bit

Problem 3.2 Constrained Optimization

a) In order to gain a useful solution we have to add one additional constrained, which makes sure that however we choose the probabilities we maintain a valid probability distribution. That means all probabilities have to sum up to 1:

$$\sum_{i=1}^{4} p_i = 1$$

Formulating it as a constrained optimization problem requires us to transform the constraints and set them to zero. Let f be the function we want to maximize and g_1, g_2 our constraints as follows:

$$\begin{aligned} f(p_1,p_2,p_3,p_4) &= -\sum_{i=1}^{4} p_i log_2(p_i) \\ &= -(p_1 log_2(p_1) + p_2 log_2(p_2) + p_3 log_2(p_3) + p_4 log_2(p_4)) \\ g_1(p_1,p_2,p_3,p_4) &= -4 + \sum_{i=1}^{4} 2p_i i \\ &= p_1 + 2p_2 + 3p_3 + 4p_4 - 2 \\ g_2(p_1,p_2,p_3,p_4) &= -1 + \sum_{i=1}^{4} p_i \\ &= p_1 + p_2 + p_3 + p_4 - 1 \end{aligned}$$

We then get our constrained optimization problem:

$$\begin{aligned} f(p_1,p_2,p_3,p_4) &\rightarrow MAX \\ g_1(p_1,p_2,p_3,p_4) &= 0 \\ g_2(p_1,p_2,p_3,p_4) &= 0 \end{aligned}$$

b) We get our lagrangian from the general form:

$$L(x_1,...,x_n,\lambda_1,...,\lambda_k) = f(x_1,...,x_n) + \lambda_1 \cdot g_1(x_1,...,x_n) + \cdots + \lambda_k \cdot g_k(x_1,...,x_n)$$

Therefore:

$$\begin{aligned} L(p_1,p_2,p_3,p_4,\lambda_1,\lambda_2) &= f(p_1,p_2,p_3,p_4) + \lambda_1 \cdot g_1(p_1,p_2,p_3,p_4) + \lambda_2 \cdot g_2(p_1,p_2,p_3,p_4) \\ &= -p_1 log(p_1) - p_2 log(p_2) - p_3 log(p_3) - p_4 log(p_4) \\ &\quad + \lambda_1 p_1 + 2\lambda_1 p_2 + 3\lambda_1 p_3 + 4\lambda_1 p_4 - 2\lambda_1 \\ &\quad + \lambda_2 p_1 + \lambda_2 p_2 + \lambda_2 p_3 + \lambda_2 p_4 - \lambda_2 \end{aligned}$$

c) We have to compute 6 partial derivatives, one for each variable $p_1,...,p_4$ and one for each lagrangian multiplier λ_1 and λ_2. The derivatives look as follows:

$$\begin{aligned} \frac{\delta L}{\delta p_1} &= -log_2(p_1) - \frac{1}{ln(2)} + \lambda_1 + \lambda_2 \\ \frac{\delta L}{\delta p_2} &= -log_2(p_2) - \frac{1}{ln(2)} + 2\lambda_1 + \lambda_2 \\ \frac{\delta L}{\delta p_3} &= -log_2(p_3) - \frac{1}{ln(2)} + 3\lambda_1 + \lambda_2 \\ \frac{\delta L}{\delta p_4} &= -log_2(p_4) - \frac{1}{ln(2)} + 4\lambda_1 + \lambda_2 \\ \frac{\delta L}{\delta \lambda_1} &= p_1 + 2p_2 + 3p_3 + 4p_4 - 2 \\ \frac{\delta L}{\delta \lambda_2} &= p_1 + p_2 + p_3 + p_4 - 1 \end{aligned}$$

This leads us to the following linear system of equations:

$$0 = -log_2(p_1) - \frac{1}{ln(2)} + \lambda_1 + \lambda_2 \tag{1}$$
$$0 = -log_2(p_2) - \frac{1}{ln(2)} + 2\lambda_1 + \lambda_2 \tag{2}$$
$$0 = -log_2(p_3) - \frac{1}{ln(2)} + 3\lambda_1 + \lambda_2 \tag{3}$$
$$0 = -log_2(p_4) - \frac{1}{ln(2)} + 4\lambda_1 + \lambda_2 \tag{4}$$
$$0 = p_1 + 2p_2 + 3p_3 + 4p_4 - 2 \tag{5}$$
$$0 = p_1 + p_2 + p_3 + p_4 - 1 \tag{6}$$

The usual approach would be to transform the first four equations (1) - (4) to get values for p_i and insert them in (5) and (6) to eventually get values for λ_1 and λ_2. However an analytical solution is hard to obtain because transforming (1) - (4) will lead to exponential functions which can not be eliminated. Therefore it is a non-linear equation system.

d) Given a our lagrangian function L from above the lagrange dual function is defined as follows:

$$\theta(v, \lambda) = \inf_{x \in D} L(x, v, \lambda) = \inf_{x \in D} \left(f_0(x) + \sum_{i=1}^{p} v_i f_i(x) + \sum_{i=1}^{m} \lambda_i g_i(x) \right)$$

Where D denotes the domain which the values of x are taken from, and v_i are lagrangian multipliers from inequality constraints (which are not present in our case). We deduce the following dual function θ^* for our case:

$$\begin{aligned}
\theta^*(\lambda) &= \inf_{x \in D} L(x, \lambda) \\
&= \inf_{x \in D} \left(f_0(x) + \sum_{i=1}^{m} \lambda_i g_i(x) \right) \\
&= \inf_{x \in D} (f(x) + \lambda_1 \cdot g_1(x) + \lambda_2 \cdot g_2(x)) \\
&= \inf_{p_1,p_2,p_3,p_4 \in D} \big(f(p_1,p_2,p_3,p_4) + \lambda_1 \cdot g_1(p_1,p_2,p_3,p_4) + \lambda_2 \cdot g_2(p_1,p_2,p_3,p_4)\big) \\
&= \inf_{p_1,p_2,p_3,p_4 \in D} (-p_1 log(p_1) - p_2 log(p_2) - p_3 log(p_3) - p_4 log(p_4) \\
&\qquad + \lambda_1 p_1 + 2\lambda_1 p_2 + 3\lambda_1 p_3 + 4\lambda_1 p_4 - 2\lambda_1 \\
&\qquad + \lambda_2 p_1 + \lambda_2 p_2 + \lambda_2 p_3 + \lambda_2 p_4 - \lambda_2)
\end{aligned}$$

The dual function θ is a concave function by its composition from affine functions. Therefore it can also be viewed as a convex function (if θ is concave then $-\theta$ is convex) and leads to a convex optimization problem. In general there are no analytical solutions to this problem, however one may apply certain numerical techniques.

e) In order to numerically solve this problem one could use the steepest descent algorithm. That algorithm starts at an (arbitrary or specifically chosen) initial point and computes the gradient (which is a vector pointing towards the maximum ascend). Then one moves to another point in the opposite direction direction of the gradient with a given (or computed) stepsize. This is done repeatedly until one reaches a local minimum (or is satisfactory close to it).

Problem 3.3 Numerical Optimization

The learning rate has a huge impact on the algorithms performance. We made different tests with a constant and dynamic learning rates, see figure 1. While the best performing constant rate a_0 learns way faster in the first 60-70 iterations, the dynamic rate a_1 catches up and ultimately converges faster towards 0. Figures 2 and 3 show our used python code.

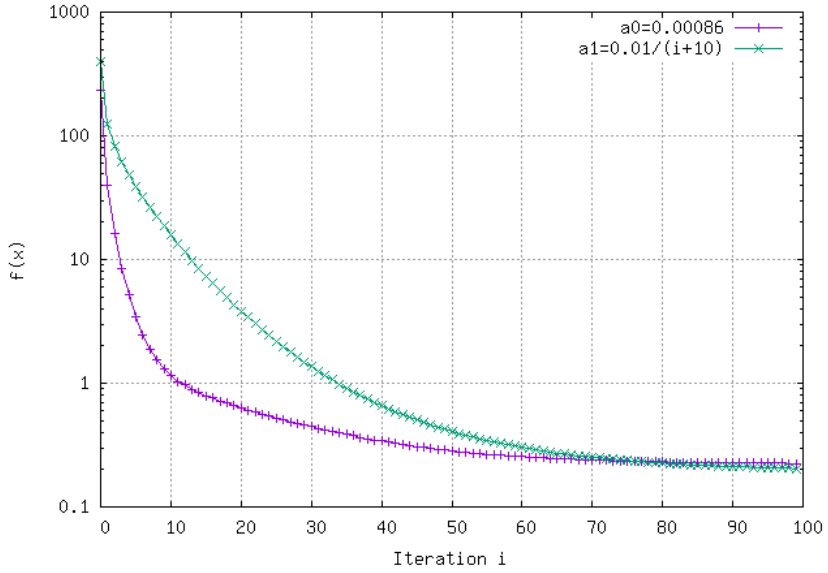

Figure 1: Learning curves from initial point $(2, 2, ..., 2)$.

Problem 3.4 Gradient Descent Variants

There are three main variants of gradient descent.

- Batch gradient descent: This is the simplest version of gradient descent. At each iteration, it computes the gradient of the loss function globally. This comes at the cost of very high memory usage, since the whole dataset needs to be kept in memory at every iteration. Therefore, this algorithm is hardly used without the following two optimizations
- Stochastic gradient descent (SGD): The first step of SGD is to shuffle the dataset. Then, instead of computing the global gradient of the loss function, SGD calculates the loss function with respect to the current sample and then moves on to the next sample until all samples were passed. This process is repeated until the end-criteria is met. The advantage of this approach is that it requires less computation resources than batch gradient descent, since only one sample has to be kept in memory. However, the optimization for one sample can lead to jumps in the global loss function, which is often not desired.
- Mini-batch gradient descent: This optimization takes the best of both worlds. The main difference to SDG is, that is minimizes the loss-function for a number of samples at a time and therefore prevents big errors introduced by some samples. However, since the number of chosen samples is typically small, compared to the whole sample set, it is still computationally efficient.

Momentums are a fraction of the last update vector that are added to the current update vector. This is especially useful in areas with a steep curvature in one dimension and a low one in the other. In such a situation, momentum prevents oscillation and accelerates convergence. However, in situations of a broad "valley" with steep turns towards the minimum, the momentum has a negative effect.

```
#!/usr/local/bin/python2.7

import sys
import numpy as np

# main function
def main(argv):
        # params
        steps = argv[0]
        learnRate = float(argv[1])

        n = 20

        # init data array
        x = []
        for i in range(0, int(n)):
                x.append(2)

        for i in range(0, int(steps)):
                # adjust learning rate here
                #rate = learnRate/(i+10)
                rate = learnRate

                # compute gradient
                grad = getGradient(x)

                # update x
                x = updateX(x, rate, grad)

                # print f(x)
                print(str(i) + "\t" + str(f(x)))

        # exit
        sys.exit()

# updates the date given the learn-rate and gradient
def updateX(x, learnRate, grad):
        for j in range(0, len(x)):
                x[j] = x[j] - learnRate * grad[j]

        return x
```

Figure 2: Gradient descent python code snippet 1 of 2.

```
# computes the rosenbrocks function
def f(x):
        sum = 0
        for i in range(0, len(x)-1):
                sum += 100*np.power((x[i+1]-x[i]*x[i]), 2)
                sum += np.power((x[i] - 1), 2)
        return sum

# computes the gradient
def getGradient(x):
        length = len(x)
        list = []
        for i in range(0, length):
                list.append(getFirstDerivative(x, i))

        # return list
        grad = np.array(list)
        grad = np.transpose(grad)
        return grad

# returns the first derivative of the rosenbrocks function
def getFirstDerivative(x, i):
        length = len(x)

        if (i == 0):
                # i==0
                x0 = x[i]
                return -400*x0*(x[i+1]-x0*x0)+2*(x0-1)
        elif (i == length-1):
                # i==N
                return 200*(x[i]-np.power(x[i-1], 2))
        else:
                # 0 < i < N
                x0 = x[i]
                y = x[i+1]
                z = x[i-1]

                return -400*x0*(y-x0*x0)+200*(x0-z*z)+2*(x0-1)

if __name__ == "__main__":
        main(sys.argv[1:])
```

Figure 3: Gradient descent python code snippet 2 of 2.