Deep Learning Lab Course 2018

Labs:
(Computer Vision) Thomas Brox,
(Robotics) Wolfram Burgard,
(Machine Learning) Frank Hutter,
(Neurorobotics) Joschka Boedecker

University of Freiburg

October 16, 2018
Technical Issues

vais

Location: Tuesday, 14:00 - 16:00, building 082, room 00 006 (Kinohoersaal)

Remark: We will be there for questions every week from 14:00 - 16:00.
  ▶ We expect you to work on your own. Your attendance is required during lectures/presentations
  ▶ We expect you have basic knowledge in ML (e.g. heard the Machine Learning lecture).

Contact information (tutors):
  Abhinav Valada valada@cs.uni-freiburg.de
  Maria Hügle hueglem@cs.uni-freiburg.de
  Aaron Klein kleinaa@cs.uni-freiburg.de
  Gabriel Leivas Oliveira oliveira@cs.uni-freiburg.de
  Tonmoy Saikia saikiat@cs.uni-freiburg.de
  Andreas Eitel eitel@cs.uni-freiburg.de

Homepage: http://dl-lab.informatik.uni-freiburg.de/
Schedule and outline

- **Phase 1**
  - **Today:** introduction deep learning (lecture).
  - **16.10 - 30.10** Assignment 1
  - **23.10:** Q/A session
  - **30.10:** introduction convolutional neural networks (lecture), hand in Assignment 1
  - **30.10 - 13.11:** Assignment 2
  - **06.11:** Q/A session
Schedule and outline

▶ Phase 1

▶ Today: introduction deep learning (lecture).
▶ 16.10 - 30.10 Assignment 1
▶ 23.10: Q/A session
▶ 30.10: introduction convolutional neural networks (lecture), hand in Assignment 1
▶ 30.10 - 13.11: Assignment 2
▶ 06.11: Q/A session

▶ Phase 2 (split into different tracks)

▶ 13.11 - 18.12: lectures and exercises of the different tracks
Schedule and outline

- **Phase 1**
  - **Today**: introduction deep learning (lecture).
  - **16.10 - 30.10**: Assignment 1
  - **23.10**: Q/A session
  - **30.10**: introduction convolutional neural networks (lecture), hand in Assignment 1
  - **30.10 - 13.11**: Assignment 2
  - **06.11**: Q/A session

- **Phase 2 (split into different tracks)**
  - **13.11 - 18.12**: lectures and exercises of the different tracks

- **Phase 3**
  - **08.01**: start of the final projects
  - **15.01**: Q/A session
  - **22.01**: Q/A session
  - **29.01**: Q/A session
  - **05.02**: poster session
Tracks (tentative topics)

▶ Track 1 Reinforcement Learning / Robotics
  ▶ Robot navigation
  ▶ Deep reinforcement learning
Tracks (tentative topics)

- **Track 1 Reinforcement Learning / Robotics**
  - Robot navigation
  - Deep reinforcement learning
- **Track 2 AutoML / Computer Vision**
  - Image segmentation
  - Autoencoders
  - Generative adversarial networks
  - Architecture search and hyperparameter optimization
Evaluation of Exercises

for each exercise:

▶ solve coding exercise alone
▶ hand-in **short** 1-2 page report explaining your results, typically accompanied by 1-2 figures (e.g. learning curves / table with comparisons)
▶ hand in your code
Final Project

- We will provide a list of different projects but feel free to propose own ideas.
- You will split up into small groups of 3 - 4 persons for the final project.
- At the end we will organize a poster session where you have to present your results.
- You need to register for the exams.
Today...

- **Lecture:** Short recap on how MLPs (feed-forward neural networks) work and how to train them

- **First assignment:** implement a simple MLP in numpy and train it on MNIST dataset (more on this at the end)
What you need to do after today’s class

- decide whether you want to take the course and which track you want to join
- if you are enrolled in HISinONE for different tracks, unregister from all tracks except the one you want to take
- start working on exercise 1
(Deep) Machine Learning Overview

1. Learn Model $M$ from the data
2. Let the model $M$ infer unknown quantities from data
(Deep) Machine Learning Overview

Data $\rightarrow$ Model M $\rightarrow$ Inference

<table>
<thead>
<tr>
<th>Data</th>
<th>Sensory Information</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labeled Images</td>
<td>An image</td>
<td>Is a cat in the image?</td>
</tr>
<tr>
<td>Transcribed Speech</td>
<td>A speech segment</td>
<td>What is this person saying?</td>
</tr>
<tr>
<td>Paraphrases</td>
<td>A pair of sentences</td>
<td>Is this sentence a paraphrase?</td>
</tr>
<tr>
<td>Movie Ratings</td>
<td>Ratings of $Y$ and by $X$</td>
<td>Will a user $X$ like a movie $Y$?</td>
</tr>
<tr>
<td>Parallel Corpora</td>
<td>A Finnish sentence</td>
<td>What is “moi” in English?</td>
</tr>
</tbody>
</table>

(Examples by Kyunghyun Cho)
Machine Learning Overview

What is the difference between deep learning and a standard machine learning pipeline?
Standard Machine Learning Pipeline

(1) Engineer good features (not learned)
(2) Learn Model
(3) Inference e.g. classes of unseen data
Unsupervised Feature Learning Pipeline

(1a) Maybe engineer good features (not learned)
(1b) Learn (deep) representation unsupervisedly
(2) Learn Model
(3) Inference e.g. classes of unseen data
Supervised Deep Learning Pipeline

(1) Jointly **Learn** everything with a deep architecture
(2) **Inference** e.g. classes of unseen data
Training supervised feed-forward neural networks

▶ Let’s formalize!
▶ **We are given:**
  ▶ Dataset \( D = \{ (x^1, y^1), \ldots, (x^N, y^N) \} \)
  ▶ A neural network with parameters \( \theta \) which implements a function \( f_\theta(x) \)
▶ **We want to learn:**
  ▶ The parameters \( \theta \) such that \( \forall i \in [1, N] : f_\theta(x^i) = y^i \)
Training supervised feed-forward neural networks

- A neural network with parameters $\theta$ which implements a function $f_\theta(x)$
- $\theta$ is given by the network weights $w$ and bias terms $b$
Neural network forward-pass

- Computing $f_\theta(x)$ for a neural network is a forward-pass

- **unit $i$ activation:**
  
  $a_i = \sum_{j=i}^{N} w_{i,j} x_j + b_i$

- **unit $i$ output:**
  
  $h_i^{(1)}(x) = t(a_i)$ where $t(\cdot)$ is an activation or transfer function
Neural network forward-pass

- Computing $f_{\theta}(x)$ for a neural network is a forward-pass

Alternatively (and much faster) use vector notation:

- **Layer activation:**
  
  $a^{(1)} = W^{(1)}x + b^{(1)}$

- **Layer output:**
  
  $h^{(1)}(x) = t(a^{(1)})$

  where $t(\cdot)$ is applied element wise
Neural network forward-pass

- Computing $f_\theta(x)$ for a neural network is a forward-pass

Second layer
- layer 2 activation:
  $$a^{(2)} = W^{(2)} h^{(1)}(x) + b^{(2)}$$
- layer 2 output:
  $$h^{(1)}(x) = t(a^{(2)})$$
  where $t(\cdot)$ is applied element wise
Neural network forward-pass

- Computing $f_{\theta}(x)$ for a neural network is a forward-pass

Output layer

- output layer activation: $a^{(3)} = W^{(3)}h^{(2)}(x) + b^{(3)}$

- network output: $f(x) = o(a^{(3)})$
  where $o(\cdot)$ is the output nonlinearity

- for classification use softmax:
  $$o_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{\mid z \mid} e^{z_j}}$$
Training supervised feed-forward neural networks

- Neural network activation functions
- Typical nonlinearities for hidden layers are: $tanh(a_i)$, sigmoid $\sigma(a_i) = \frac{1}{1 + e^{-a_i}}$, ReLU $relu(a_i) = \max(a_i, 0)$
- tanh and sigmoid are both squashing non-linearities
- ReLU just thresholds at 0
→ Why not linear?
Training supervised feed-forward neural networks

- Train parameters $\theta$ such that $\forall i \in [1, N] : f_\theta(x^i) = y^i$
Training supervised feed-forward neural networks

- Train parameters $\theta$ such that $\forall i \in [1, N] : f_\theta(x^i) = y^i$

- We can do this via minimizing the empirical risk on our dataset $D$

\[
\min_\theta L(f_\theta, D) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} l(f_\theta(x^i), y^i),
\]

(1)

where $l(\cdot, \cdot)$ is a per example loss
Training supervised feed-forward neural networks

- Train parameters $\theta$ such that $\forall i \in [1, N] : f_\theta(x^i) = y^i$
- We can do this via minimizing the empirical risk on our dataset $D$

$$\min_{\theta} L(f_\theta, D) = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} l(f_\theta(x^i), y^i), \quad (1)$$

where $l(\cdot, \cdot)$ is a per example loss

- For regression often use the squared loss:

$$l(f_\theta(x), y) = \frac{1}{2} \sum_{j=1}^{M} (f_{j,\theta}(x) - y_j)^2$$

- For $M$-class classification use the negative log likelihood:

$$l(f_\theta(x), y) = \sum_{j} -\log(f_{j,\theta}(x)) \cdot y_j$$
The simplest approach to minimizing $\min_{\theta} L(f_\theta, D)$ is gradient descent.

**Gradient descent:**

$\theta^0 \leftarrow$ init randomly  

**do**  

$\theta^{t+1} = \theta^t - \gamma \frac{\partial L(f_\theta, D)}{\partial \theta}$  

while $(L(f_{\theta^{t+1}}, V) - L(f_{\theta^t}, V))^2 > \epsilon$
Gradient descent

- The simplest approach to minimizing $\min_\theta L(f_\theta, D)$ is gradient descent

**Gradient descent:**

\[ \theta^0 \leftarrow \text{init randomly} \]

\[ \text{do} \]

\[ \theta^{t+1} = \theta^t - \gamma \frac{\partial L(f_\theta, D)}{\partial \theta} \]

\[ \text{while } (L(f_{\theta^{t+1}}, V) - L(f_{\theta^t}, V))^2 > \epsilon \]

- Where $V$ is a validation dataset (why not use $D$?)

- Remember in our case: $L(f_\theta, D) = \frac{1}{N} \sum_{i=1}^{N} l(f_\theta(x^i), y^i)$

- We will get to computing the derivatives shortly
Gradient descent

- Gradient descent example: \( D = \{(x^1, y^1), \ldots, (x^{100}, y^{100})\} \) with
  \[
  x \sim U[0, 1] \\
  y = 3 \cdot x + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1)
  \]
Gradient descent

- **Gradient descent example:** \( D = \{(x^1, y^1), \ldots, (x^{100}, y^{100})\} \) with
  \[
  x \sim U[0, 1] \\
  y = 3 \cdot x + \epsilon \text{ where } \epsilon \sim N(0, 0.1)
  \]

- **Learn** parameters \( \theta \) of function \( f_\theta(x) = \theta x \) using loss
  \[
  l(f_\theta(x), y) = \frac{1}{2} \| f_\theta(x) - y \|_2^2 = \frac{1}{2} (f_\theta(x) - y)^2 \\
  \frac{\partial L(f_\theta, D)}{\partial \theta} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(f_\theta, D)}{\partial \theta} = \frac{1}{N} \sum_{i=1}^{N} (\theta x - y)x
  \]
Gradient descent

- Gradient descent example: \( D = \{(x^1, y^1), \ldots, (x^{100}, y^{100})\} \) with
  \[
  x \sim U[0, 1] \\
  y = 3 \cdot x + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1)
  \]

- Learn parameters \( \theta \) of function \( f_\theta(x) = \theta x \) using loss
  \[
  l(f_\theta(x), y) = \frac{1}{2} \| f_\theta(x) - y \|_2^2 = \frac{1}{2} (f_\theta(x) - y)^2
  \]
  \[
  \frac{\partial L(f_\theta, D)}{\partial \theta} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(f_\theta, D)}{\partial \theta} = \frac{1}{N} \sum_{i=1}^{N} (\theta x - y) x
  \]
Gradient descent

- Gradient descent example $\gamma = 2$. 

gradient descent
Stochastic Gradient descent (SGD)

- There are two problems with gradient descent:
  1. We have to find a good $\gamma$
  2. Computing the gradient is expensive if the training dataset is large!

- Problem 2 can be attacked with online optimization (we will have a look at this)

- Problem 1 remains but can be tackled via second order methods or other advanced optimization algorithms (rprop/rmsprop, adagrad)
Gradient descent

1. We have to find a good $\gamma$ ($\gamma = 2., \gamma = 5.$)
Stochastic Gradient descent (SGD)

2 Computing the gradient is expensive if the training dataset is large!
▶ What if we would only evaluate $f$ on parts of the data?

**Stochastic Gradient Descent:**

$\theta^0 \leftarrow \text{init randomly}$

do

▶ $(x', y') \sim D$
  sample example from dataset $D$

▶ $\theta^{t+1} = \theta^t - \gamma^t \frac{\partial l(f_{\theta}(x'), y')}{\partial \theta}$

while $(\infty \sum_{t=1}^{\infty} (f_{\theta^t}(V) - f_{\theta^{t+1}}(V))^2 > \epsilon$

where $\sum_{t=1}^{\infty} \gamma^t \rightarrow \infty$ and $\sum_{t=1}^{\infty} (\gamma^t)^2 < \infty$

($\gamma$ should go to zero but not too fast)

→ SGD can speed up optimization for large datasets
→ but can yield very noisy updates
→ in practice mini-batches are used (compute $l(\cdot, \cdot)$ for several samples and average)
→ we still have to find a learning rate schedule $\gamma^t$
Stochastic Gradient descent (SGD)

Computing the gradient is expensive if the training dataset is large!
- What if we would only evaluate \( f \) on parts of the data?

\[
\text{Stochastic Gradient Descent:} \\
\theta^0 \leftarrow \text{init randomly} \\
\text{do} \\
\quad (x', y') \sim D \quad \text{sample example from dataset } D \\
\quad \theta^{t+1} = \theta^t - \gamma^t \frac{\partial l(f_\theta(x'), y')}{\partial \theta} \\
\text{while } (L(f_{\theta^{t+1}}, V) - L(f_{\theta^t}, V))^2 > \epsilon \\
\text{where } \sum_{t=1}^{\infty} \gamma^t \to \infty \text{ and } \sum_{t=1}^{\infty} (\gamma^t)^2 < \infty \\
(\gamma \text{ should go to zero but not too fast})
\]

- SGD can speed up optimization for large datasets
- but can yield very noisy updates
- in practice mini-batches are used
  (compute \( l(\cdot, \cdot) \) for several samples and average)
- we still have to find a learning rate schedule \( \gamma^t \)
Stochastic Gradient descent (SGD)

→ Same data, assuming that gradient evaluation on all data takes 4 times as much time as evaluating a single datapoint

\((\text{gradient descent } (\gamma = 2), \text{ stochastic gradient descent } (\gamma^t = 0.01 \frac{1}{t}))\)
Neural Network backward pass

→ Now how do we compute the gradient for a network?

▶ Use the chain rule:
\[
\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}
\]

▶ first compute loss on output layer

▶ then backpropagate to get
\[
\frac{\partial l(f(x), y)}{\partial W^{(3)}} \text{ and } \frac{\partial l(f(x), y)}{\partial a^{(3)}}
\]
Neural Network backward pass

→ Now how do we compute the gradient for a network?

▷ gradient wrt. layer 3 weights:
\[
\frac{\partial l(f(x), y)}{\partial W^{(3)}} = \frac{\partial l(f(x), y)}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial W^{(3)}}
\]

▷ assuming \( l \) is NLL and softmax outputs, gradient wrt. layer 3 activation is:
\[
\frac{\partial l(f(x), y)}{\partial a^{(3)}} = -(y - f(x))
\]
y is one-hot encoded

▷ gradient of \( a \) wrt. \( W^{(3)} \):
\[
\frac{\partial a^{(3)}}{\partial W^{(3)}} = h^{(2)}(x)^T
\]

→ recall
\[
a^{(3)} = W^{(3)} h^{(2)}(x) + b^{(3)}
\]
Neural Network backward pass

→ Now how do we compute the gradient for a network?

- gradient wrt. layer 3 weights:
  \[
  \frac{\partial l(f(x), y)}{\partial W^{(3)}} = \frac{\partial l(f(x), y)}{\partial a^{(3)}} \frac{\partial a^{(3)}}{W^{(3)}}
  \]

- assuming \(l\) is NLL and softmax outputs, gradient wrt. layer 3 activation is:
  \[
  \frac{\partial l(f(x), y)}{\partial a^{(3)}} = -(y - f(x))
  \]

- gradient of \(a\) wrt. \(W^{(3)}\):
  \[
  \frac{\partial a^{(3)}}{\partial W^{(3)}} = h^{(2)}(x)^T
  \]

- combined:
  \[
  \frac{\partial l(f(x), y)}{\partial W^{(3)}} = -(y - f(x))(h^{(2)}(x))^T
  \]

→ recall
\[
a^{(3)} = W^{(3)} h^{(2)}(x) + b^{(3)}
\]
Neural Network backward pass

→ Now how do we compute the gradient for a network?

▶ gradient wrt. layer 3 weights:
\[
\frac{\partial l(f(x), y)}{\partial W^{(3)}} = \frac{\partial l(f(x), y)}{\partial a^{(3)}} \frac{\partial a^{(3)}}{W^{(3)}}
\]

▶ assuming \( l \) is NLL and softmax outputs, gradient wrt. layer 3 activation is:
\[
\frac{\partial l(f(x), y)}{\partial a^{(3)}} = -(y - f(x))
\]

▶ gradient of \( a \) wrt. \( W^{(3)} \):
\[
\frac{\partial a^{(3)}}{\partial W^{(3)}} = (h^{(2)}(x))^T
\]

▶ gradient wrt. previous layer:
\[
\frac{\partial l(f(x), y)}{\partial h^{(2)}(x)} = \frac{\partial l(f(x), y)}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial h^{(2)}(x)} = (W^{(3)})^T \frac{\partial l(f(x), y)}{\partial a^{(3)}}
\]

→ recall
\[
a^{(3)} = W^{(3)} h^{(2)}(x) + b^{(3)}
\]
Neural Network backward pass

→ Now how do we compute the gradient for a network?

▶ gradient wrt. layer 2 weights:
\[
\frac{\partial l(f(x), y)}{\partial W^{(2)}} = \frac{\partial l(f(x), y)}{\partial h^{(2)}(x)} \frac{\partial h^{(2)}(x)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{W^{(2)}
\]

▶ same schema as before just have to consider computing derivative of activation function \( \frac{\partial h^{(2)}(x)}{\partial a^{(2)}} \), e.g. for sigmoid \( \sigma(\cdot) \)
\[
\frac{\partial h^{(2)}(x)}{\partial a^{(2)}} = \sigma(a_i)(1 - a_i)
\]

▶ and backprop even further

→ recall
\[
a^{(3)} = W^{(3)} h^{(2)}(x) + b^{(3)}
\]
Gradient Checking

→ Backward-pass is just repeated application of the **chain rule**
→ However, there is a huge potential for bugs ...
→ Gradient checking to the rescue (simply check code via finite-differences):

**Gradient Checking:**

\[
\theta = (W^{(1)}, b^{(1)}, \ldots) \text{ init randomly}
\]
\[
x \leftarrow \text{init randomly} \ ; \ y \leftarrow \text{init randomly}
\]
\[
g_{\text{analytic}} \leftarrow \text{compute gradient } \frac{\partial l(f_\theta(x), y)}{\partial \theta} \text{ via backprop}
\]

for \( i \) in \( \#\theta \)

- \( \hat{\theta} = \theta \)
- \( \hat{\theta}_i = \theta_i + \epsilon \)
- \( g_{\text{numeric}} = \frac{l(f_{\hat{\theta}}(x), y) - l(f_\theta(x), y)}{\epsilon} \)
- \( \text{assert}(\|g_{\text{numeric}} - g_{\text{analytic}}\| < \epsilon) \)

- can also be used to test partial implementations (i.e. layers, activation functions)
  → simply remove loss computation and backprop **ones**
Overfitting

- If you train the parameters of a large network $\theta = (W^{(1)}, b^{(1)}, \ldots)$ you will see overfitting!
  $\rightarrow L(f_{\theta}(x), D) \ll L(f_{\theta}(x), V)$
- This can be at least partly conquered with regularization

$$L(f_{\theta}(x), D) = \frac{1}{N} \sum_{i=1}^{N} l(f_{\theta}(x_i), y_i) + \frac{1}{\#\theta} \sum_{i} \|\theta_i\|_2$$

$\rightarrow$ enforces small weights (Occam's razor)

- dropout: kill $\approx 50\%$ of the activations in each hidden layer during training forward pass. Multiply hidden activations by $\frac{1}{2}$ during testing
  $\rightarrow$ prevents co-adaptation / enforces robustness to noise
- Many, many more!
Overfitting

- If you train the parameters of a large network \( \theta = (W^{(1)}, b^{(1)}, \ldots) \) you will see overfitting!
  \[ L(f_\theta(x), D) \ll L(f_\theta(x), V) \]
- This can be at least partly conquered with regularization
  - **weight decay**: change cost (and gradient)
    \[
    L(f_\theta, D) = \frac{1}{N} \min_\theta \sum_{i=1}^{N} l(f_\theta(x^i), y^i) + \frac{1}{\#\theta} \sum_i \|\theta_i\|^2
    \]
  - \( \rightarrow \) enforces small weights (occams razor)
Overfitting

- If you train the parameters of a large network $\theta = (W^{(1)}, b^{(1)}, \ldots)$ you will see overfitting!
  $\rightarrow L(f_\theta(x), D) \ll L(f_\theta(x), V)$

- This can be at least partly conquered with regularization
  - **weight decay**: change cost (and gradient)
    
    $$L(f_\theta, D) = \frac{1}{N} \min_\theta \sum_{i=1}^{N} l(f_\theta(x^i), y^i) + \frac{1}{\#\theta} \sum_{i} \|\theta_i\|^2$$

  $\rightarrow$ enforces small weights (occams razor)

- **dropout**: kill $\approx 50\%$ of the activations in each hidden layer during training forward pass. Multiply hidden activations by $\frac{1}{2}$ during testing

  $\rightarrow$ prevents co-adaptation / enforces robustness to noise
Overfitting

- If you train the parameters of a large network $\theta = (W^{(1)}, b^{(1)}, \ldots)$ you will see overfitting!
  $\rightarrow L(f_\theta(x), D) \ll L(f_\theta(x), V)$

- This can be at least partly conquered with regularization
  - **weight decay**: change cost (and gradient)
    \[
    L(f_\theta, D) = \frac{1}{N} \min_\theta \sum_{i=1}^N l(f_\theta(x^i), y^i) + \frac{1}{\#\theta} \sum_i \|\theta_i\|^2
    \]
  $\rightarrow$ enforces small weights (occams razor)
  - **dropout**: kill $\approx 50\%$ of the activations in each hidden layer during training forward pass. Multiply hidden activations by $\frac{1}{2}$ during testing
  $\rightarrow$ prevents co-adaptation / enforces robustness to noise
  - Many, many more!
Assignment

- **Implementation**: Implement a simple feed-forward neural network by completing the provided *stub* this includes:
  - possibility to use 2-4 layers
  - sigmoid/tanh and ReLU for the hidden layer
  - softmax output layer
  - optimization via gradient descent (gd)
  - optimization via stochastic gradient descent (sgd)
  - weight initialization with random noise (!!!) (use normal distribution with changing std. deviation for now)

- Bonus points for testing some advanced ideas:
  - implement dropout, weight decay
  - implement a different optimizer (rprop, rmsprop, adagrad)


- **Evaluation**:
  - Find good parameters (learning rate, number of iterations etc.) using a validation set (usually take the last 10k examples from the training set)
  - After optimizing parameters run on the full dataset and test once on the test-set (you should be able to reach $\approx 1.6 - 1.8\%$ error)

- **Submission**: Clone our github repo and send us the link to your github/bitbucket repo including your solution code and the report as a pdf-file. Email to kleinaa@cs.uni-freiburg.de with subject: *dl-lab-course 18*
Thanks to Tobias Springenberg who generated most of these slides for the DL Lab Course WS 2016/2017.