UNIVERSITY NAME

DOCTORAL THESIS

Thesis Title

Author:

Jan Alexander

FREUDENTHAL

Supervisor:

Prof. Arthur KORTE

A thesis submitted in fulfillment of the requirements

for the degree of Ph.D.

in the

Research Group Name

Department or School Name

October 18, 2019

Declaration of Authorship

I, Jan Alexander FREUDENTHAL, declare that this thesis titled, "Thesis Title" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITY NAME

Abstract

Faculty Name

Department or School Name

Ph.D.

Thesis Title

by Jan Alexander FREUDENTHAL

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

List of Abbreviations

ANN	**A**rtificial **N**eural **N**etwork
BLUB	**B**est **L**inear **U**nbiased **P**redictor
BLUE	**B**est **L**inear **U**nbiased **E**stimator
CPU	**C**ore **P**rocessing **U**nit
EMMA	**E**fficient **M**ixed **M**odel **A**ssociations
FCL	**F**ully **C**onnected **L**ayer
GP	**G**enomic **P**redicton
GPU	**G**raphical **P**rocessing **U**nit
GS	**G**enomic **S**election
GWAIS	**G**enome **W**ide **I**nteraction **A**ssociation **S**tudies
GWAS	**G**enome **W**ide **A**ssociation **S**tudies
HDF	**H**ierarchical **D**ata **F**ormat
LCL	**L**ocally **C**onnected **L**ayer
LD	**L**inkage **D**isequilibrium
LMM	**L**inear **M**ixed **M**odel
MLP	**M**ulti **L**ayer **P**erceptron
ML	**M**achine **L**earning
QTL	**Q**uantitative **T**rait **L**ocus
RKHS	**R**erproducing **K**ernel **H**ilbert **S**paces
RSS	**R**esidual **S**um of **S**quares
SNP	**S**ingle **N**ucleotide **P**olymorphism
TRN	**TR**ai**N**ing subset
TST	**TeST**ing subset

WGS	**W**hole **G**enome **S**equencing
LSC	**L**arge **S**ingle **C**opy
SSC	**S**mall **S**ingle **C**opy
IR	**I**nverted **R**epeat
DNA	**D**eoxyribo**N**ucleic **A**cid
DNA	**R**ibo**N**ucleic **A**cid
GUI	**G**raphical **U**ser **I**nterface

For/Dedicated to/To my…

Chapter 1

Benchmarking of Chloroplast Genome Assembly tools

1.1 Introduction

Here I will but the introduction to from the paper

1.2 Material and Methods

1.3 Results

$$score = \frac{1}{4} \cdot \left(cov_{ref} + cov_{qry} + min \left\{ \frac{cov_{qry}}{cov_{ref}}, \frac{cov_{ref}}{cov_{qry}} \right\} + \frac{1}{n_{contigs}} \right) \cdot 100 \quad (1.1)$$

FIGURE 1.1: An electron (artist's impression).

TABLE 1.1: The effects of treatments X and Y on the four groups studied.

Groups	Treatment X	Treatment Y
1	0.2	0.8
2	0.17	0.7
3	0.24	0.75
4	0.68	0.3

1.3.1 Performance metrics

1.3.2 Qualitative

1.3.3 Simulated data

1.3.4 Real data sets

1.3.5 Consistency

1.4 Disucssion

Chapter 2

Understanding the hapoltype structure of Arabidopisis thaliana

2.1 Introduction

2.2 Haplotyping of A. thaliana

2.3 Results

2.4 Disucssion

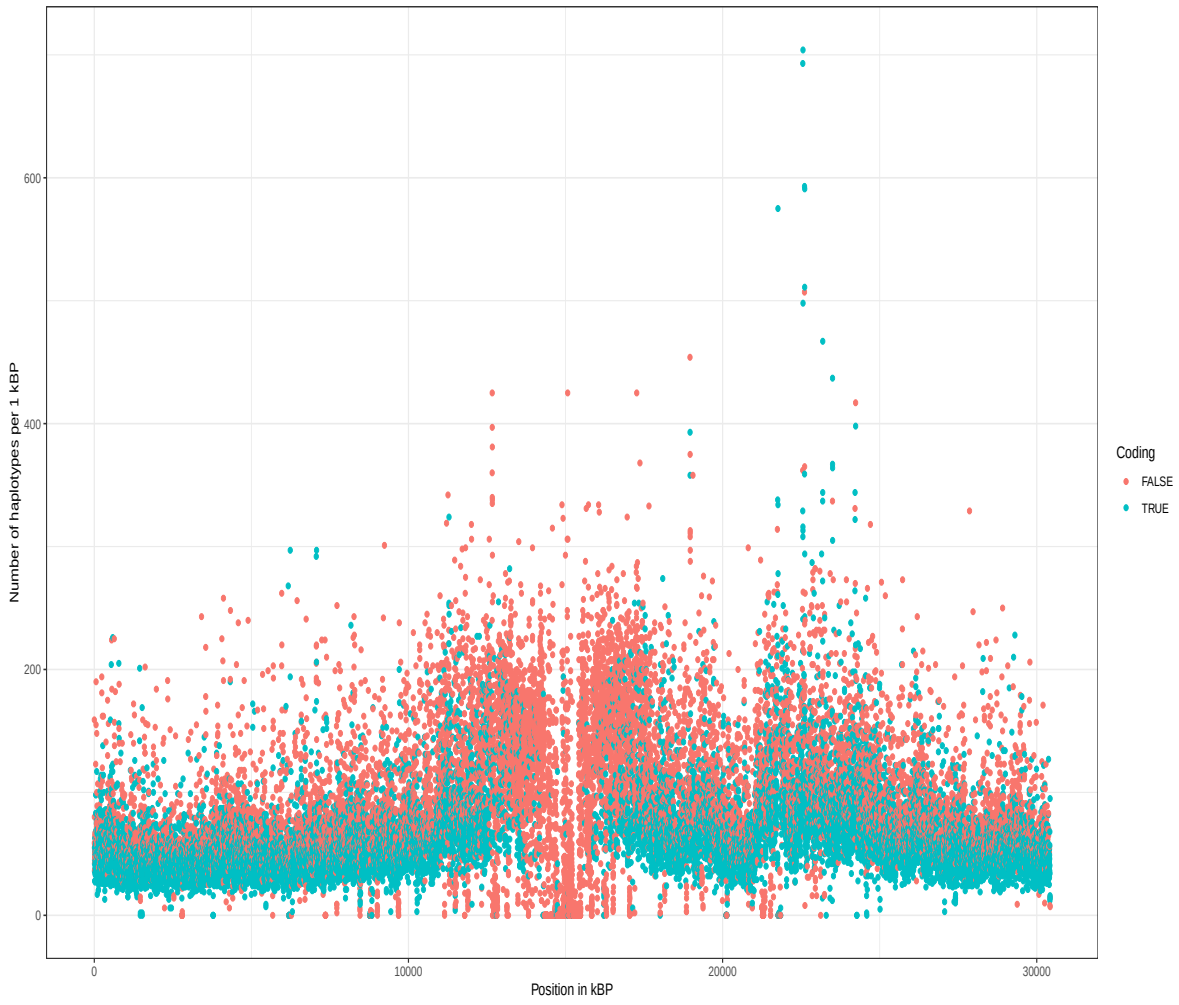

FIGURE 2.1: The number of segregating haplotypes with a polymorphism in at least one position over a stretch of 1 kBP.

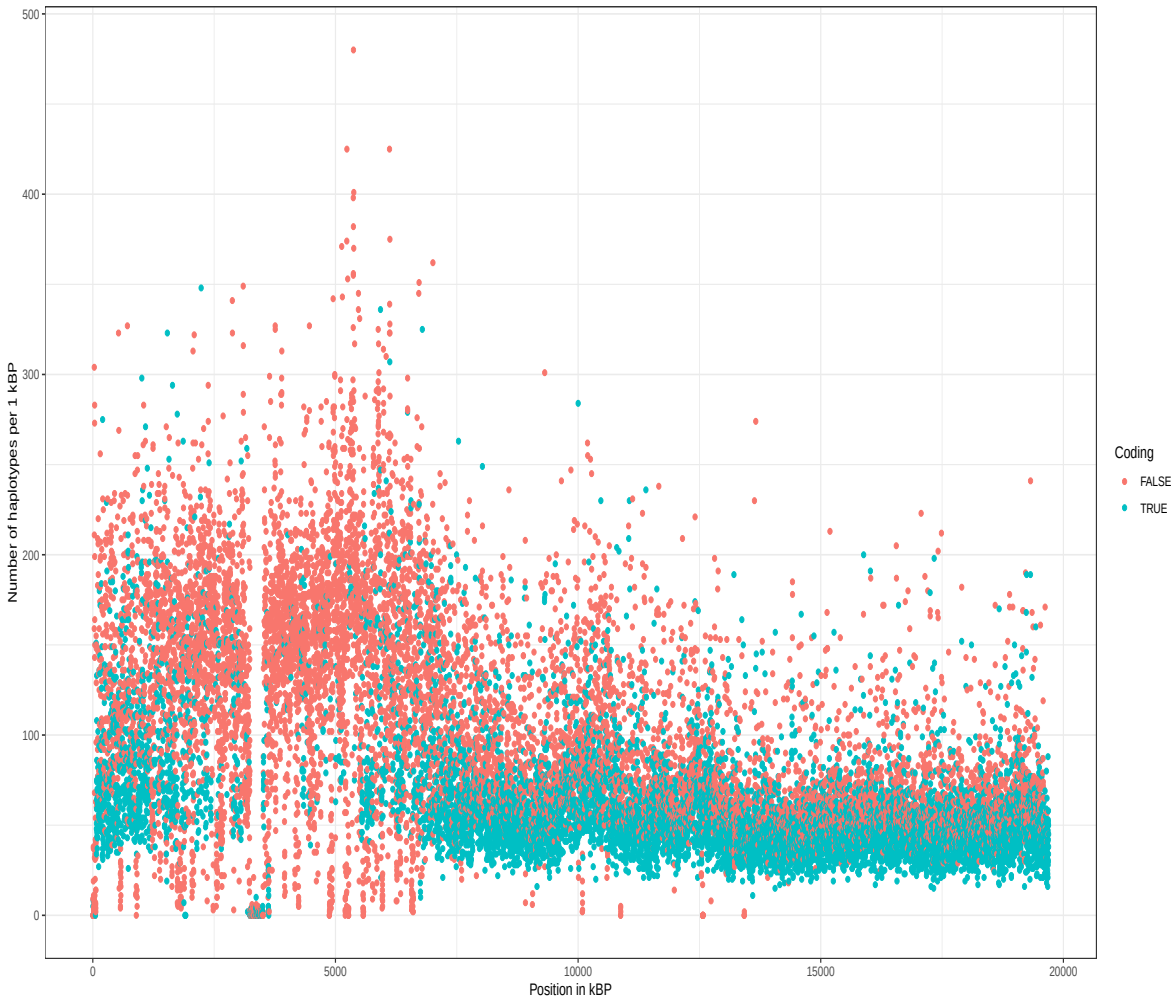

FIGURE 2.2: Number of segregating haplotypes with a polymorphism in at least one position over a stretch of 1 kBP.

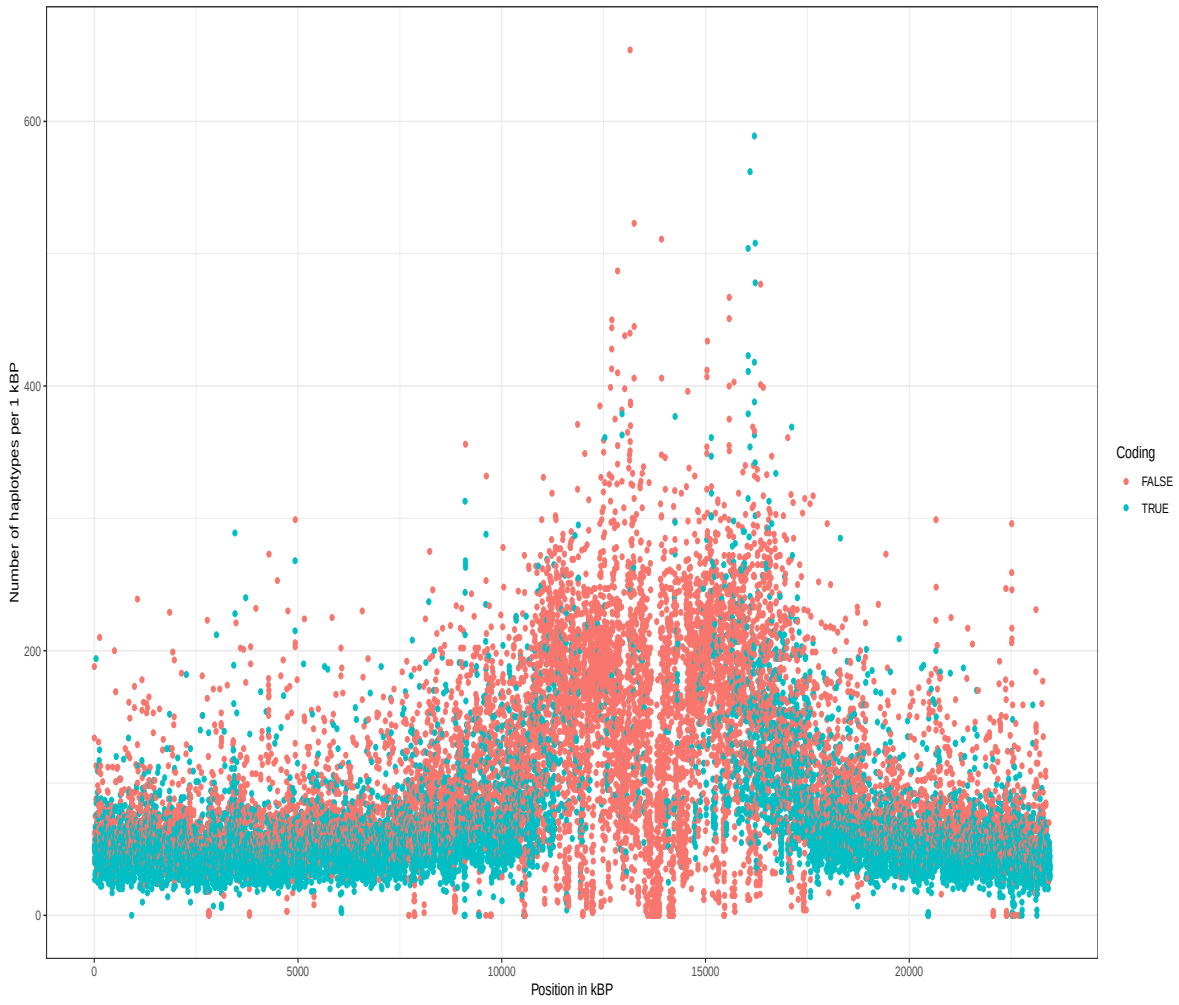

FIGURE 2.3: Number of segregating haplotypes with a polymorphism in at least one position over a stretch of 1 kBP.

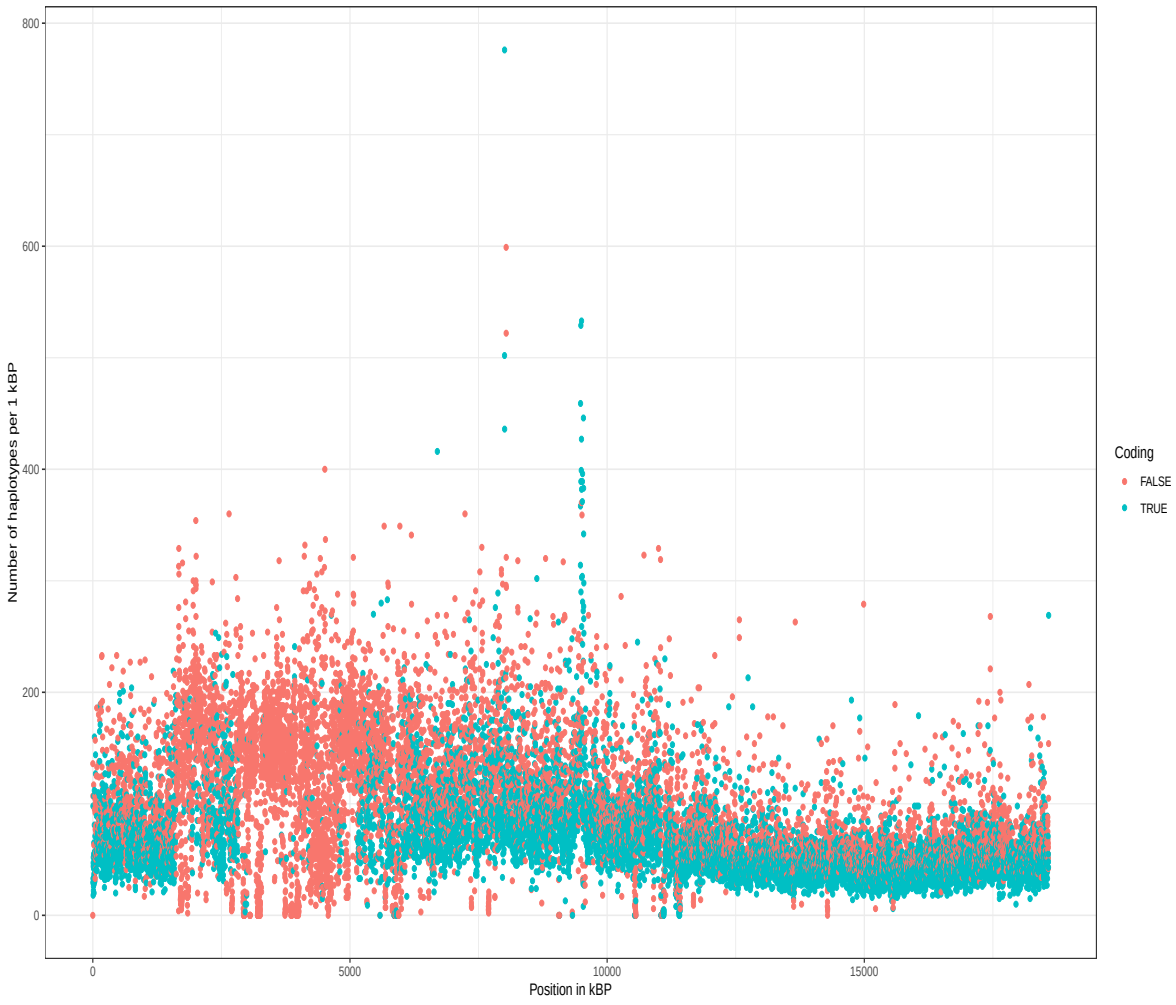

FIGURE 2.4: Number of segregating haplotypes with a polymorphism in at least one position over a stretch of 1 kBP.

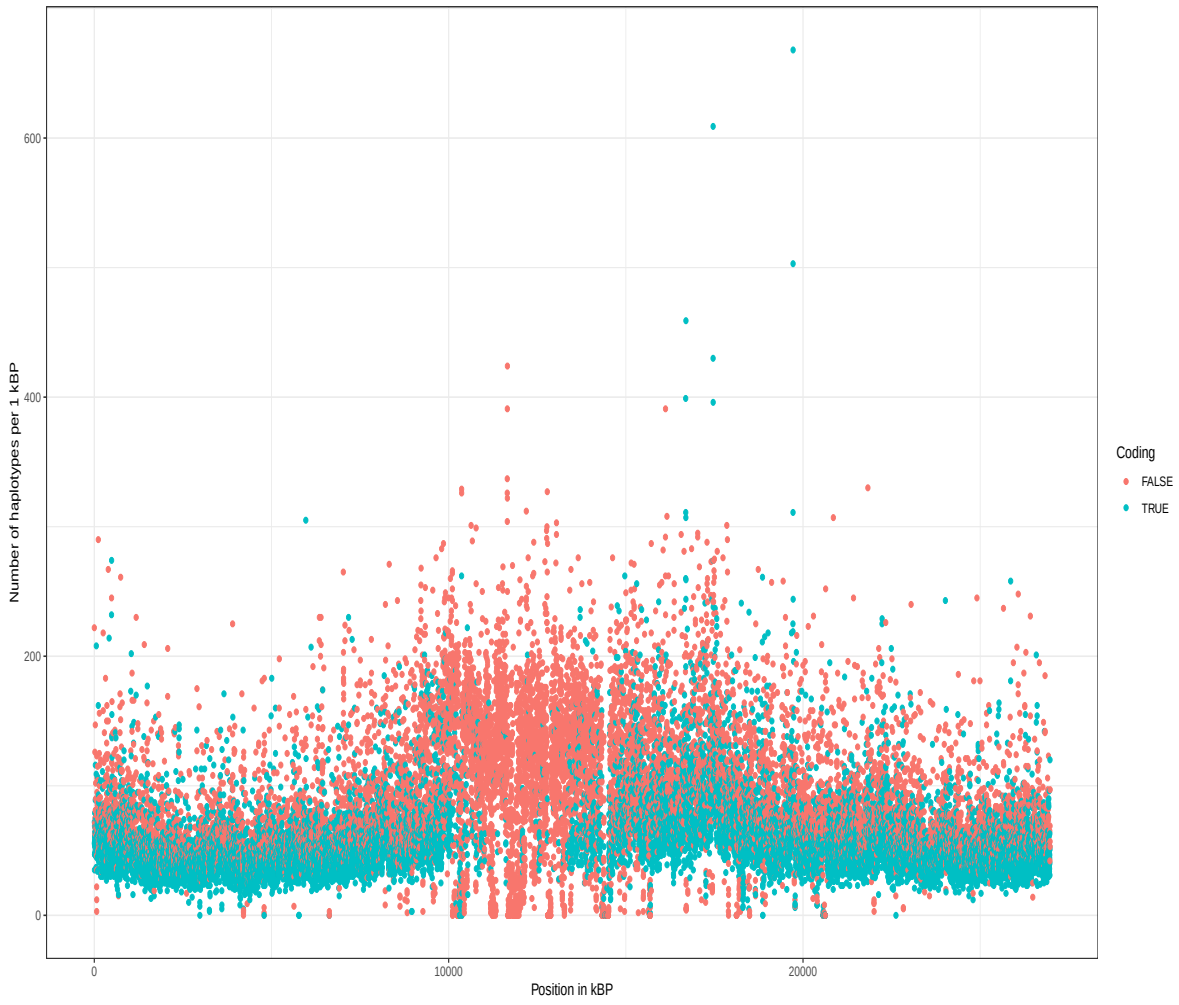

FIGURE 2.5: Number of segregating haplotypes with a polymorphism in at least one position over a stretch of 1 kBP.

Chapter 3

GWAS Flow a gpu-accelerated software for large-scale genome-wide association studies

3.1 Introduction

Genome-wide association studies, pioneered in human genetics **Hirschhorn2005** in the last decade, have become the predominant method to detect associations between phenotypes and the genetic variations present in a population. Understanding the genetic architecture of traits and mapping the underlying genomic polymorphisms is of paramount importance for successful breeding both in plants and animals, as well as for studying the genetic risk factors of diseases. Over the last decades, the cost for genotyping have been reduced dramatically. Early GWAS consisted of a few hundred individuals which have been phenotyped and genotyped on a couple of hundreds to thousands of genomic markers. Nowadays, marker density for many species easily exceed millions of genomic polymorphisms. Albeit commonly SNPs are used for association studies, standard GWAS models are flexible to handle different genomic features as input. The *Arabidopsis* 1001 genomes project features

for example 1135 sequenced *Arabidopsis thaliana* accessions with over 10 million genomic markers that segregate in the population **1001genome**. Other genome projects also yielded large amounts of genomic data for a substantial amount of individuals, as exemplified in the 1000 genomes project for humans **1000genome**, the 2000 yeast genomes project or the 3000 rice genomes project **3000genome**. Thus, there is an increasing demand for GWAS models that can analyze these data in a reasonable time frame. One critical step of GWAS is to determine the threshold at which an association is termed significant. Classically the conservative Bonferroni threshold is used, which accounts for the number of statistical tests that are performed, while many recent studies try to use significance thresholds that are based on the false-discovery rate (FDR) **Storey9440**. An alternative approach are permutation-based thresholds **che2014adaptive**. Permutation-based thresholds estimate the significance by shuffling phenotypes and genotypes before each GWAS run, thus any signal left in the data should not have a genetic cause, but might represent model mis-specifications or uneven phenotypic distributions. Typically this process is repeated hundreds to thousands of times and will lead to a distinct threshold for each phenotype analyzed **togninalli2017aragwas**. The computational demand of permutation-based thresholds is immense, as per analysis not one, but at least hundreds of GWAS need to be performed. Here the main limitation is the pure computational demand. Thus, faster GWAS models could easily make the estimation of permutation-based thresholds the default choice.

3.2 Methods

GWAS Model

The GWAS model used for `GWAS-Flow` is based on a fast approximation of the linear-mixed-model described in **kang2010variance**; **Zhang2010**, which estimates the variance components σ_g and σ_e only once in a null model that includes the genetic relationship matrix, but no distinct genetic markers. These components are thereafter used for the tests of each specific marker. Here, the underlying assumption is, that the ratio of these components stays constant, even if distinct genetic markers are included into the GWAS model. This holds true for nearly all markers and only markers which posses a big effect will alter this ratio slightly, where now σ_g would become smaller compared to the null model. Thus, the p-values calculated by the approximation might be a little higher (less significant) for strongly associated markers.

The GWAS-Flow Software

The `GWAS-Flow` software was designed to provide a fast and robust GWAS implementation that can easily handle large data and allows to perform permutations in a reasonable time frame. Traditional GWAS implementations that are implemented using Python **van1995python** or R **R** cannot always meet these demands. We tried to overcome those limitations by using TensorFlow **tensorflow2015-whitepaper**, a multi-language machine learning framework published and developed by Google. GWAS calculations are composed of a series of matrix computations that can be highly parallelized, and easily integrated into the architecture provided by TensorFlow. Our implementation allows both, the classical parallelization of code on multiple processors (CPUs) and the use of graphical processing units (GPUs). `GWAS-Flow` is written using the Python TensorFlow API. Data import is done with *pandas*

mckinney-proc-scipy-2010 and/or *HDF5* for Python **hdf5_2014**. Preprocessing of the data (e.g filtering by minor Allele count (MAC)) is performed with *numpy* **oliphant2006guide**. Variance components for residual and genomic effects are estimated with a slightly altered function based on the Python package *limix* **Lippert003905**. The GWAS model is based on the following linear mixed model that takes into account the effect of every marker with respect to the kinship:

$$Y = \beta_0 + X_i\beta_i + u + \epsilon, u \sim N(0, \sigma_g K), \epsilon \sim N(0, \sigma_e I) \tag{3.1}$$

From this LMM the residual sum of squares for marker i are calcucated as descirebed in 3.2

$$RSS_i = \sum Y - (X_i\beta_0 + I_i\beta_1) \tag{3.2}$$

The residuals are used to calculate a p-value for each marker according to an overall F-test that compares the model including a distinct genetic effect to a model without this genetic effect:

$$F = \frac{RSS_{env} - R1_{full}}{\frac{R1_{full}}{n-3}} \tag{3.3}$$

Apart from the p-values that derive from the F-distribution, `GWAS-Flow` also report summary statistics, such as the estimated effect size (β_i) and its standard error for each marker.

Calculation of permutation-based thresholds for GWAS

To calculate a permuation-based threshold, we essentially perform n repetitions ($n > 100$) of the GWAS on the same data with the sole difference that before each GWAS we randomize the phenotypic values. Thus any correlation between the phenotype and the genotype will be broken and indeed for

over 90% of these analyses the estimated pseudo-heritability is close to zero. On the other hand, the phenotypic distribution will stay unaltered by this randomization. Hence, any remaining signal in the GWAS has to be of a non-genetic origin and could be caused by e.g. model mis-specifications. Now we take the lowest p-value (after filtering for the desired minor allele count) for each permutation and take the 5% lowest value as the permutation-based threshold for the GWAS.

Benchmarking

For benchmarking of `GWAS-Flow` we used data from the *Arabidopsis* 1001 Genomes Project **1001genome**. The genomic data we used were subsets between 10,000 and 100,000 markers. We chose not to include subsets that exceed 100,000 markers, because there is a linear relationship between the number of markers and the computational time demanded, as all markers are tested independently. We used phenotypic data for flowering time at ten degrees (FT10) for *A. thaliana*, published and downloaded from the AraPheno database **seren2016arapheno**. We down- and up-sampled sets to generate phenotypes for sets between 100 and 5000 accessions. For each set of phenotypes and markers we ran 10 permutations to assess the computational time needed. All analyses have been performed with a custom R script that has been used previously **togninalli2017aragwas**, `GWAS-Flow` using either a CPU or a GPU architecture and *GEMMA* **Zhou2012**. *GEMMA* is a fast and efficient implementation of the mixed model that is broadly used to perform GWAS. All calculations were run on the same machine using 16 i9 virtual CPUs. The GPU version ran on an NVIDIA Tesla P100 graphic card. Additionally to the analyses of the simulated data, we compared the times required by *GEMMA* and both `GWAS-Flow` implementations for $>$ 200 different real datasets from *A. thaliana* that have been downloaded from the AraPheno **seren2016arapheno** database and have been analyzed with the available fully imputed genomic dataset of ca. 10 million

markers, filtered for a minor allele count greater five.

3.3 Results

The two main factors influencing the computational time for GWAS are the number of markers incorporated in such an analysis and the number of different accessions, while the latter has an approximate quadratic effect in classical GWAS implementations **Zhou2012**. Figure 1A shows the time demand as a function of the number of accessions used in the analysis with 10,000 markers. The quadratic increase in time demand is clearly visible for the custom R implementation, as well as for the CPU-based `GWAS-Flow` implementation and *GEMMA*. The `GWAS-Flow` implementation and *GEMMA* clearly outperforms the R implementation in general, while for a small number of accessions `GWAS-Flow` is slightly faster then *GEMMA*. For the GPU-based implementation the increase in run-time with larger sample sizes is much less pronounced. While for small ($<$ 1,000 individuals) data, there is no benefit compared to running `GWAS-Flow` on CPUs or running *GEMMA*, the GPU-version clearly outperforms the other implementations if the number of accessions increases. Figure 1B shows the computational time in relation to the number of markers and a fixed amount of 2000 accessions for the two different `GWAS-Flow` implementations. Here, a linear relationship is visible in both cases. To show the performance of `GWAS-Flow` not only for simulated data, we also run both implementations on more than 200 different real datasets downloaded from the AraPheno database. Figure 1C shows the computational time demands for all analyses comparing both `GWAS-Flow` implementation to *GEMMA*. Here, the CPU-based `GWAS-Flow` performs comparable to *GEMMA*, while the GPU-based implementation outperforms both, if the number of accessions is above 500. Importantly all obtained GWAS results (p-values, beta estimates and standard errors of the beta estimates) are

nearly (apart from some mathematical inaccuracies) identical between the three different implementations.

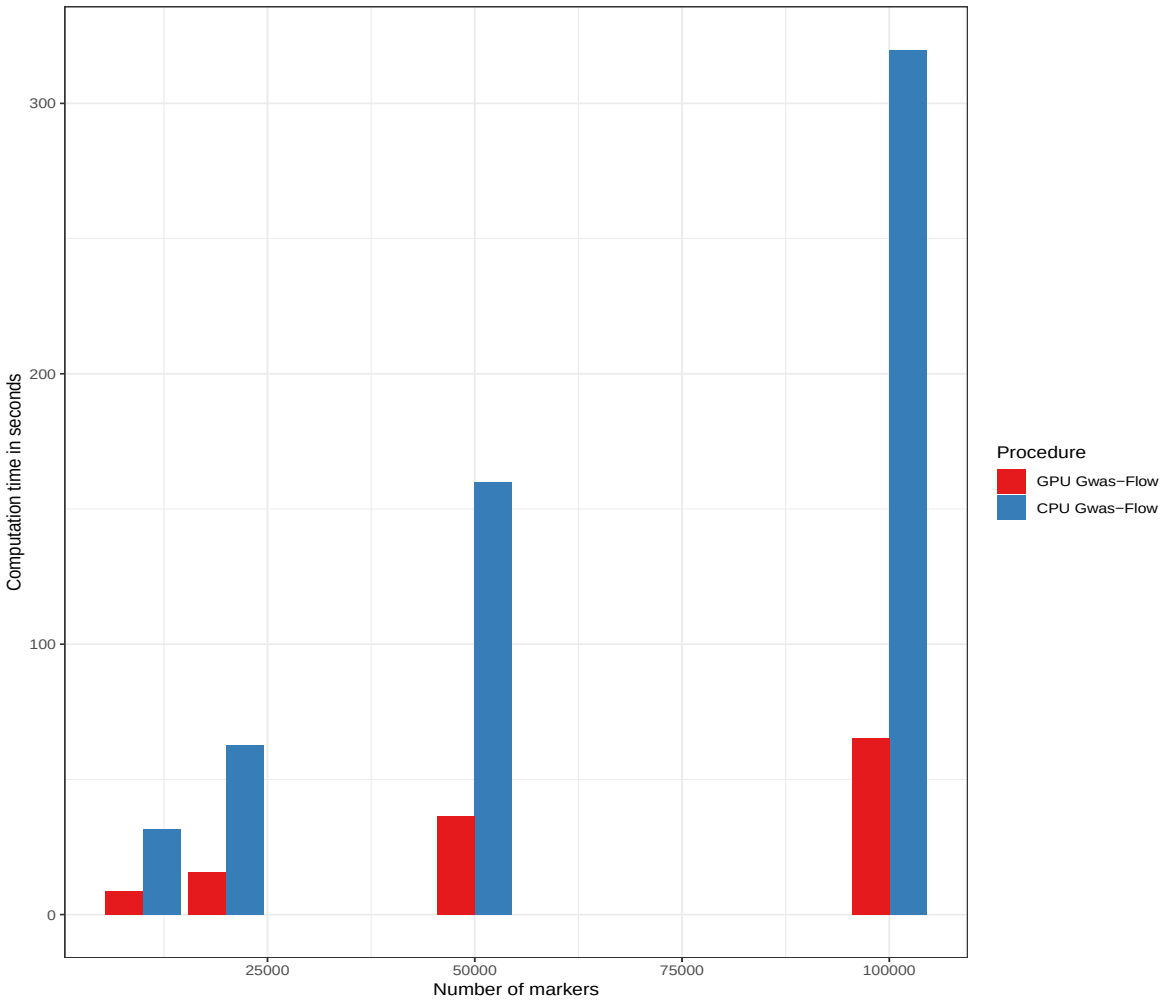

FIGURE 3.1: Computational time as a function of the number of genetic markers with constantly 2000 accessions for both `GWAS-Flow` versions

3.4 Disucssion

We made use of recent developments of computational architecture and software to cope with the increasing computational demand in analyzing large GWAS datasets. With `GWAS-Flow` we implemented both, a CPU- and a GPU-based version of the classical linear mixed model commonly used for GWAS. Both implementations outperform custom R scripts on simulated and real

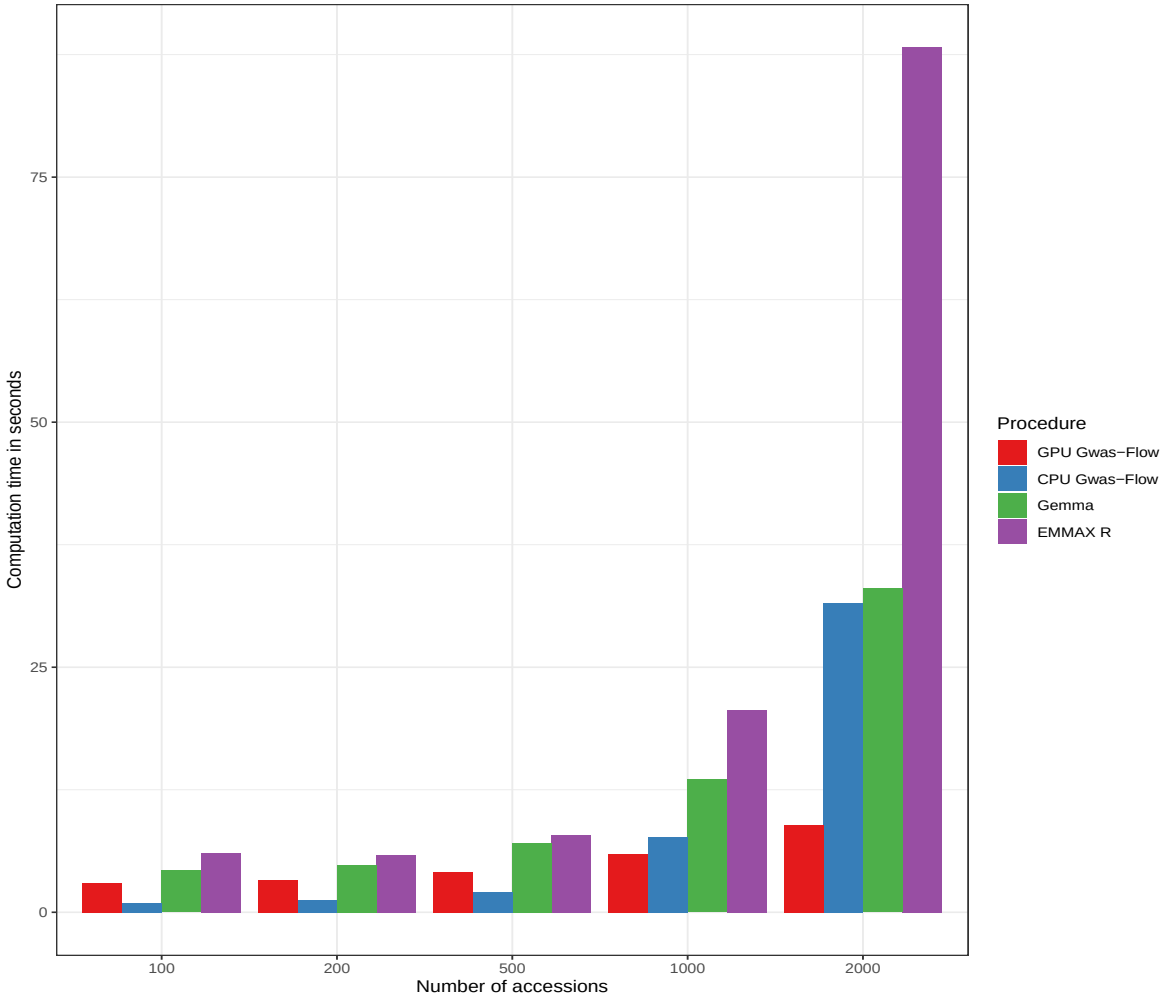

FIGURE 3.2: Computational time as a function of the number of accessions with 10000 markers each.

data. While the CPU-based version performs nearly identical compared to *GEMMA*, a commonly used GWAS implementation, the GPU-based implementation outperforms both, if the number of individuals, which have been phenotyped, increases. For analyzing big data, here the main limitation would be the RAM of the GPU, but as the individual test for each marker are independent, this can be easily overcome programmatically. The presented `GWAS-Flow` implementations are markedly faster compared to custom GWAS scripts and even outperform efficient fast implementations like *GEMMA* in

terms of speed. This readily enables the use of permutation-based thresholds, as with `GWAS-Flow` hundred permutations can be performed in a reasonable time even for big data. Thus, it is possible for each analyzed phenotype to create a specific, permutation-based threshold that might present a more realistic scenario. Importantly the permutation-based threshold can be easily adjusted to different minor allele counts, generating different significance thresholds depending on the allele count. This could help to distinguish false and true associations even for rare alleles. `GWAS-Flow` is a versatile and fast software package. Currently `GWAS-Flow` is and will remain under active development to make the software more versatile. This will e.g. include the compatibility with TensorFlow v2.0.0 and enable data input formats, such as PLINK **purcell2007plink**. The whole framework is flexible, so it is easy to include predefined co-factors e.g to enable multi-locus models **segura2012efficient** or account for multi-variate models like the multi-trait mixed model **korte2012mixed**. Standard GWAS are good in detecting additive effects with comparably large effect sizes, but lack the ability to detect epistatic interactions and their influence on complex traits **mckinney2012six**; **korte2013advantages**. To catch the effects of these gene-by-gene or SNP-by-SNP interactions, a variety of genome-wide association interaction studies (GWAIS) have been developed, thoroughly reviewed in **ritchie2018GWAIS**. Here, `GWAS-Flow` might provide a tool that enables to test the full pairwise interaction matrix of all SNPs. Although this might be a statistic nightmare, it now would be computationally feasible.

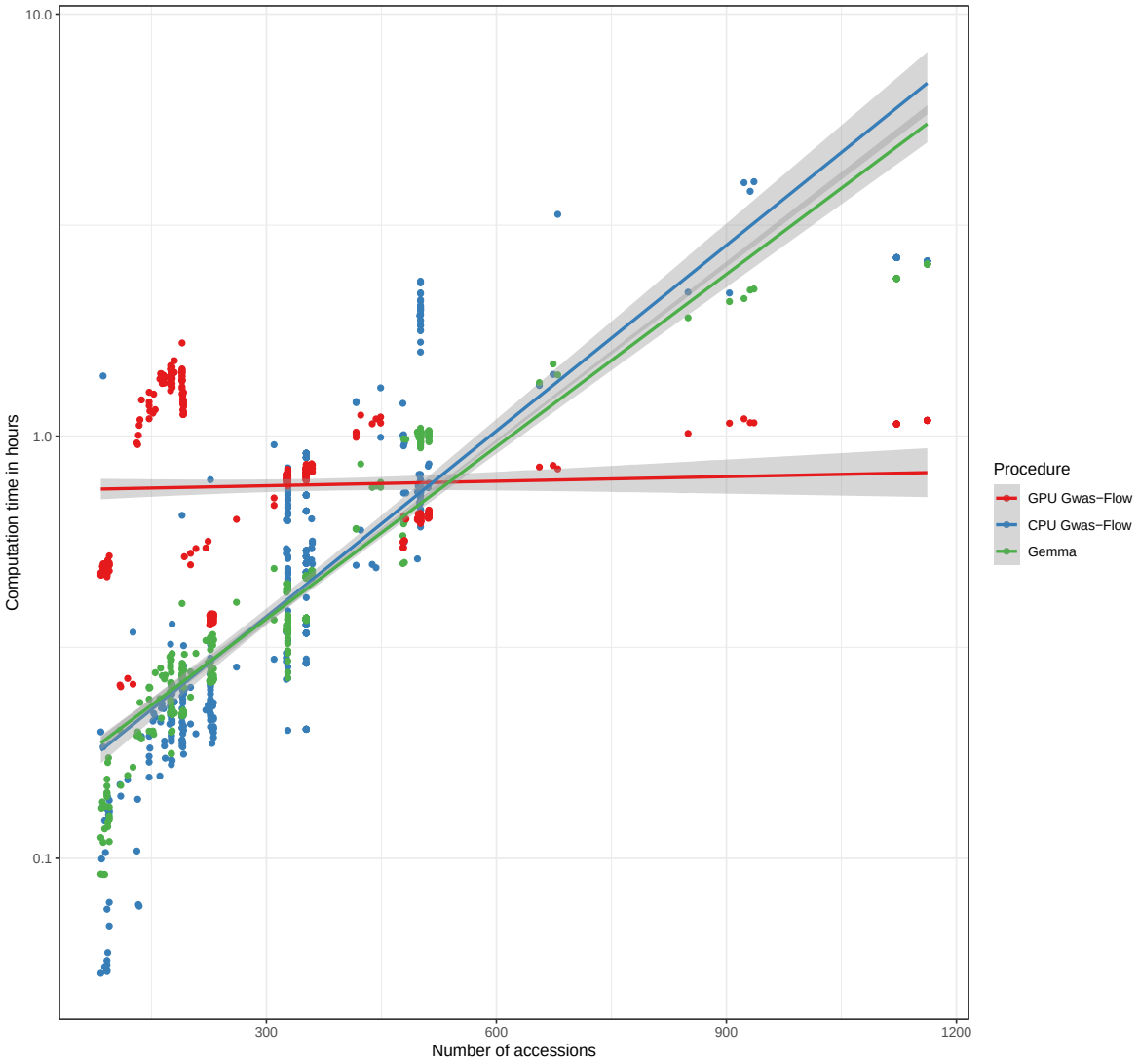

FIGURE 3.3: Comparison of the computational time for the analyses of $>$ 200 phenotypes from *Arabidopsis thaliana* as a function of the number of accessions for *GEMMA* and the CPU- and GPU-based version of `GWAS-Flow`. GWAS was performed with a fully imputed genotype matrix containing 10.7 M markers and a minor allele filter of MAC $>$ 5

Chapter 4

Genomic prediction of phenotypic values of quantitative traits using Artificial nerual networks

4.1 Introduction

Genomic selection (GS) has been successfully applied in animal **gianola2015one**, **hayes2010genome** and plant breeding **crossa2010**, **desta2014genomic**, **heffner2010plant**, **crossa2017genomic** as well as in medical applications, since it was first reported **hayes2001**. Since then the repertoire of methods for predicting phenotypic values has increased rapidly e.g.**dlc2009**, **habier2011**, **gianola2013** , **crossa2017**. The most commonly applied methods include GBLUP and a set of related algorithms known as the bayesian alphabet **gianola2009**. Genomic prediction in general has repeatedly been shown to outperform pedigree-based methods **crossa2010**, **albrecht2011** and is nowadays used in many plant and animal breeding shemes. It has also been shown that using whole-genome information is superior to using only feature-selected markers with known QTLs for a given trait **bernardo2007**, **heffner2011** in some cases. A

more recent study **azodi2019** compared 11 different genomic prediction algorithms with a variety of data sets and found contradicting results, indicating that feature selection can be use ull in some cases the when the whole genome regression is performed by neural nets While every new method is a valuable addition to the tool-kits for genomic selection, some fundamental problems remain unsolved, of which the n»p problematic stands out. Usually in genomic selection settings the size of the training population (TRN) with n phenotypes is substantially smaller than the number of markers (p) **fan2014challenges**. Making the number of features immensely large, even when SNP-SNP interactions are not considered. Furthermore each marker is treated as an independent observation neglecting collinearity and linkage disequilibrium (LD). Further difficulties arise through non-additive, epistatic and dominance marker effects. The main problem with epistasis issue quantitative genetics is the almost infinite amount of different marker combinations, that cannot be represented within the size of TRN in the thousands, the same problems arises for example in GWA studies **korte2013**. With already large p the number of possible additive SNP-SNP interactions potentiates to $p^{(p-1)}$. Methods that attempt to overcome those issues are EG-BLUP, using an enhanced epistatic kinship matrix and reproducing kernel Hilbert space regression (RKHS) **jiang2015**, **martini2017genomic**.

In the past 10 years, due to increasing availability of high performance computational hardware with decreasing costs and parallel development of free easy-to-use software, most prominent being googles library TensorFlow **TF2016** and Keras **keras2015**, machine learning (ML) has experienced a renaissance. ML is a set of methods and algorithms used widely for regression and classification problems. popular among those are e.g. support vector machines, multi-layer perceptrons (MLP) and convolutional neural networks. The machine learning mimics the architecture of neural networks and are therefore commonly referred to as artificial neural networks (ANN).

Those algorithms have widely been applied in many biological fields **min2017deep , lan2018survey, mamoshina2016applications, angermueller2016 , webb2018deep, rampasek2016tensorflow**.

A variety of studies assessed the usability of ML in genomic prediction **gonzalez2018applications, gonza2016, ogutu2011comparison, montesinos2019benchmarking, grinberg2018evaluation , cuevas2019deep, montesinos2019new, ma2017deepgs, qiu2016application, gonza2012 li2018genomic**. Through all those studies the common denominator is that there is no such thing as a gold standard for genomic prediction. No single algorithm was able to outperform all the others tested in a single of those studies, let alone in all. While the generally aptitude of ML for genomic selection has been repeatedly shown, how no evidence exists that neural networks can outperform or in many cases perform on that same level as mixed-model approaches as GBLUP **hayes2001prediction**. While in other fields like image classification neural networks have up to 100s of hidden layers **he2016deep** the commonly used fully-connected networks in genomic prediction of 1 - 3 hidden layers. With 1 layer networks often being the most successful among those. Contradicting to the idea behind machine learning in genomic selection 1 hidden layer networks will be inapt to capture interactions between loci and thus only account for additive effects. As shown in **azodi2019** convolutional networks perform worse than fully-connected networks in genomic selection, which again is contradicting to other fields where convolutional layers are applied successfully, e.g natural language processing **dos2014deep** or medical image analysis **litjens2017survey**. Instead of using convolutional layers and fully-connected layers only, as show in Pook et al 2019, we also propose to use locally-connected layer in combination with fully-connected layers. While CL and LCL are closely related they have a significant difference. While in CL weights are shared between neurons in LCLs each neuron as its own weight. This leads to a reduced number of parameters to be trained in the following

FCLs, and should therefore theoretically lead to a decrease in overfitting a common problem in machine learning. To evaluate the results of Pook et al. 2019 accomplished with simulated data we used the data sets generated in the scope of the 1001 genome project of *Arabidopsis thaliana* **1001genome**

4.2 Material

4.3 Methods

4.4 Results

4.5 Disucssion

Appendix A

Source code GWAS-Flow

A.1 gwas.py

```
import os
import sys
import time
import numpy as np
import pandas as pd
import main
import h5py

# set defaults
mac_min = 1
batch_size =  500000
out_file = "results.csv"
m = 'phenotype_value'
perm = 1
mac_min= 6

X_file = 'gwas_sample_data/AT_geno.hdf5'
Y_file = 'gwas_sample_data/phenotype.csv'
K_file = 'gwas_sample_data/kinship_ibs_binary_mac5.h5py'



for i in range (1,len(sys.argv),2):
    if sys.argv[i] == "-x" or sys.argv[i] == "--genotype":
        X_file = sys.argv[i+1]
```

```
26      elif sys.argv[i] == "-y" or sys.argv[i] == "--phenotype":
27          Y_file = sys.argv[i+1]
28      elif sys.argv[i] == "-k" or sys.argv[i] == "--kinship":
29          K_file = sys.argv[i+1]
30      elif sys.argv[i] == "-m":
31          m = sys.argv[i+1]
32      elif sys.argv[i] == "-a" or sys.argv[i] == "--mac_min":
33          mac_min = int(sys.argv[i+1])
34      elif sys.argv[i] == "-bs" or sys.argv[i] == "--batch-size":
35          batch_size = int(sys.argv[i+1])
36      elif sys.argv[i] == "-p" or sys.argv[i] == "--perm":
37          perm  = int(sys.argv[i+1])
38      elif sys.argv[i] == "-o" or sys.argv[i] == "--out":
39          out_file = sys.argv[i+1]
40      elif sys.argv[i] == "-h" or sys.argv[i] == "--help":
41          print("-x , --genotype :file containing marker
    information in csv or hdf5 format of size")
42          print("-y , --phenotype: file container phenotype
    information in csv format"  )
43          print("-k , --kinship : file containing kinship matrix
    of size k X k in csv or hdf5 format")
44          print("-m : name of columnn containing the phenotype :
    default m = phenotype_value")
45          print("-a , --mac_min : integer specifying the minimum
    minor allele count necessary for a marker to be included.
    Default a = 1" )
46          print("-bs, --batch-size : integer specifying the number
     of markers processed at once. Default -bs 500000" )
47          print("-p , --perm : single integer specifying the
    number of permutations. Default 1 == no perm ")
48          print("-o , --out : name of output file. Default -o
    results.csv  ")
49          print("-h , --help : prints help and command line
    options")
50          quit()
51      else:
52          print('unknown option ' + str(sys.argv[i]))
53          quit()
```

```
54
55
56
57 print("parsed commandline args")
58
59 start = time.time()
60
61 X,K,Y_,markers = main.load_and_prepare_data(X_file,Y_file,K_file
       ,m)
62
63
64 ## MAF filterin
65 markers_used , X , macs = main.mac_filter(mac_min,X,markers)
66
67 ## prepare
68 print("Begin performing GWAS on ", Y_file)
69
70 if perm == 1:
71     output = main.gwas(X,K,Y_,batch_size)
72     if( X_file.split(".")[-1] == 'csv'):
73         chr_pos = np.array(list(map(lambda x : x.split("- "),
       markers_used)))
74     else:
75         chr_reg = h5py.File(X_file,'r')['positions'].attrs['
       chr_regions']
76         mk_index= np.array(range(len(markers)),dtype=int)[macs
       >= mac_min]
77         chr_pos = np.array([list(map(lambda x: sum(x > chr_reg
       [:,1]) + 1, mk_index)), markers_used]).T
78         my_time = np.repeat((time.time()-start),len(chr_pos))
79     pd.DataFrame({
80         'chr' : chr_pos[:,0] ,
81         'pos' : chr_pos[:,1] ,
82         'pval': output[:,0] ,
83         'mac' : np.array(macs[macs >= mac_min],dtype=np.int) ,
84         'eff_size': output[:,1] ,
85         'SE' : output[:,2]}).to_csv(out_file,index=False)
86 elif perm > 1:
```

```
87      min_pval = []
88      perm_seeds = []
89      my_time = []
90      for i in range(perm):
91          start_perm = time.time()
92          print("Running permutation ", i+1, " of ",perm)
93          my_seed  = np.asscalar(np.random.randint(9999,size=1))
94          perm_seeds.append(my_seed)
95          np.random.seed(my_seed)
96          Y_perm = np.random.permutation(Y_)
97          output = main.gwas(X,K,Y_perm,batch_size)
98          min_pval.append(np.min(output[:,0]))
99          print("Elapsed time for permuatation",i+1 ," with p_min"
    , min_pval[i]," is",": ", round(time.time() - start_perm,2))
100         my_time.append(time.time()-start_perm)
101     pd.DataFrame({
102         'time': my_time ,
103         'seed': perm_seeds ,
104         'min_p': min_pval }).to_csv(out_file,index=False)
105
106 print("done")
107
108 end = time.time()
109 eltime = np.round(end -start,2)
110
111 if eltime <= 59:
112     print("Total time elapsed",  eltime, "seconds")
113 elif eltime > 59 and eltime <= 3600:
114     print("Total time elapsed",  np.round(eltime / 60,2) , "
    minutes")
115 elif eltime > 3600 :
116     print("Total time elapsed",  np.round(eltime / 60 / 60,2), "
    hours")
117
118
```

A.2 main.py

```
    import pandas as pd
import numpy as np
from scipy.stats import f
import tensorflow as tf
import limix
import herit
import h5py
import limix
import multiprocessing as mlt

def load_and_prepare_data(X_file,Y_file,K_file,m):
    type_K = K_file.split(".")[-1]
    type_X = X_file.split(".")[-1]

    ## load and preprocess genotype matrix
    Y = pd.read_csv(Y_file,engine='python').sort_values(['
   accession_id']).groupby('accession_id').mean()
    Y = pd.DataFrame({'accession_id' :  Y.index, '
   phenotype_value' : Y[m]})
    if type_X == 'hdf5' or type_X == 'h5py'  :
        SNP = h5py.File(X_file,'r')
        markers= np.asarray(SNP['positions'])
        acc_X =  np.asarray(SNP['accessions'][:],dtype=np.int)
    elif type_X == 'csv' :
        X = pd.read_csv(X_file,index_col=0)
        markers = X.columns.values
        acc_X = X.index
        X = np.asarray(X,dtype=np.float32)/2
    else :
        sys.exit("Only hdf5, h5py and csv files are supported")

    if type_K == 'hdf5' or type_K == 'h5py':
        k = h5py.File(K_file,'r')
        acc_K = np.asarray(k['accessions'][:],dtype=np.int)
    elif type_K == 'csv':
        k = pd.read_csv(K_file,index_col=0)
        acc_K = k.index
        k = np.array(k, dtype=np.float32)
```

```
37
38      acc_Y =  np.asarray(Y[['accession_id']]).flatten()
39      acc_isec = [isec for isec in acc_X if isec in acc_Y]
40
41      idx_acc = list(map(lambda x: x in acc_isec, acc_X))
42      idy_acc = list(map(lambda x: x in acc_isec, acc_Y))
43      idk_acc = list(map(lambda x: x in acc_isec, acc_K))
44
45      Y_ = np.asarray(Y.drop('accession_id',1),dtype=np.float32)[
    idy_acc,:]
46
47      if type_X == 'hdf5' or type_X == 'h5py' :
48          X = np.asarray(SNP['snps'][0:(len(SNP['snps'])+1),],
    dtype=np.float32)[:,idx_acc].T
49          X = X[np.argsort(acc_X[idx_acc]),:]
50          k1 = np.asarray(k['kinship'][:])[idk_acc,:]
51          K  = k1[:,idk_acc]
52          K = K[np.argsort(acc_X[idx_acc]),:]
53          K = K[:,np.argsort(acc_X[idx_acc])]
54      else:
55          X  = X[idx_acc,:]
56          k1 = k[idk_acc,:]
57          K  = k1[:,idk_acc]
58
59
60      print("data has been imported")
61      return X,K,Y_,markers
62
63
64  def mac_filter(mac_min, X, markers):
65      ac1 = np.sum(X,axis=0)
66      ac0 = X.shape[0] - ac1
67      macs = np.minimum(ac1,ac0)
68      markers_used  = markers[macs >= mac_min]
69      X = X[:,macs >= mac_min]
70      return markers_used, X, macs
71
72  def gwas(X,K,Y,batch_size):
```

```
73      n_marker = X.shape[1]
74      n = len(Y)
75      ## REML
76      K_stand = (n-1)/np.sum((np.identity(n) - np.ones((n,n))/n) *
        K) * K
77      vg, delta, ve  = herit.estimate(Y,"normal",K_stand,verbose =
        False)
78      print(" Pseudo-heritability is " , vg / (ve + vg + delta))
79      print(" Performing GWAS on ", n , " phenotypes and ",
    n_marker ,"markers")
80      ## Transform kinship-matrix, phenotypes and estimate
    intercpt
81      Xo = np.ones(K.shape[0]).flatten()
82      M = np.transpose(np.linalg.inv(np.linalg.cholesky(vg *
    K_stand + ve  * np.identity(n)))).astype(np.float32)
83      Y_t = np.sum(np.multiply(np.transpose(M),Y),axis=1).astype(
    np.float32)
84      int_t = np.sum(np.multiply(np.transpose(M),np.ones(n)),axis
    =1).astype(np.float32)
85      ## EMMAX Scan
86      RSS_env = (np.linalg.lstsq(np.reshape(int_t,(n,-1)) , np.
    reshape(Y_t,(n,-1)))[1]).astype(np.float32)
87      ## calculate betas and se of betas
88      def stderr(a,M,Y_t2d,int_t):
89            x = tf.stack((int_t,tf.squeeze(tf.matmul(M.T,tf.reshape
    (a,(n,-1))))),axis=1)
90            coeff = tf.matmul(tf.matmul(tf.linalg.inv(tf.matmul(tf.
    transpose(x),x)),tf.transpose(x)),Y_t2d)
91            SSE = tf.reduce_sum(tf.math.square(tf.math.subtract(Y_t
    ,tf.math.add(tf.math.multiply(x[:,1],coeff[0,0]),tf.math.
    multiply(x[:,1],coeff[1,0])))))
92            SE = tf.math.sqrt(SSE/(471-(1+2)))
93            StdERR = tf.sqrt(tf.linalg.diag_part(tf.math.multiply(
    SE , tf.linalg.inv(tf.matmul(tf.transpose(x),x)))))[1]
94            return tf.stack((coeff[1,0],StdERR))
95      ## calculate residual sum squares
96      def rss(a,M,y,int_t):
97            x_t = tf.reduce_sum(tf.math.multiply(M.T,a),axis=1)
```

```
98          lm_res = tf.linalg.lstsq(tf.transpose(tf.stack((int_t,
    x_t),axis=0)),Y_t2d)
99          lm_x = tf.concat((tf.squeeze(lm_res),x_t),axis=0)
100         return tf.reduce_sum(tf.math.square(tf.math.subtract(tf
    .squeeze(Y_t2d),tf.math.add(tf.math.multiply(lm_x[1],lm_x
    [2:]), tf.multiply(lm_x[0],int_t)))))
101     ## loop over the batches
102     for i in range(int(np.ceil(n_marker/batch_size))):
103         tf.reset_default_graph()
104         if n_marker < batch_size:
105             X_sub = X
106         else:
107             lower_limit = batch_size * i
108             upper_limit = batch_size * i + batch_size
109             if upper_limit <= n_marker :
110                 X_sub = X[:,lower_limit:upper_limit]
111                 print("Working on markers ", lower_limit , " to
    ", upper_limit, " of ", n_marker )
112             else:
113                 X_sub = X[:,lower_limit:]
114                 print("Working on markers ", lower_limit , " to
    ", n_marker, " of ", n_marker )
115         config = tf.ConfigProto()
116         n_cores = mlt.cpu_count()
117         config.intra_op_parallelism_threads = n_cores
118         config.inter_op_parallelism_threads = n_cores
119         sess = tf.Session(config=config)
120         Y_t2d = tf.cast(tf.reshape(Y_t,(n,-1)),dtype=tf.float32)
121         y_tensor =  tf.convert_to_tensor(Y_t,dtype = tf.float32)
122         StdERR = tf.map_fn(lambda a : stderr(a,M,Y_t2d,int_t),
    X_sub.T)
123         R1_full = tf.map_fn(lambda a: rss(a,M,Y_t2d,int_t),
    X_sub.T)
124         F_1 = tf.divide(tf.subtract(RSS_env, R1_full),tf.divide(
    R1_full,(n-3)))
125         if i == 0 :
126             output = sess.run(tf.concat([tf.reshape(F_1,(X_sub.
    shape[1],-1)),StdERR],axis=1))
```

```
127         else :
128             tmp = sess.run(tf.concat([tf.reshape(F_1,(X_sub.
    shape[1],-1)),StdERR],axis=1))
129             output = np.append(output,tmp,axis=0)
130         sess.close()
131         F_dist = output[:,0]
132      pval  = 1 - f.cdf(F_dist,1,n-3)
133      output[:,0] = pval
134      return output
135
136
137
```

A.3 herit.py

```
1
2 def estimate(y, lik, K, M=None, verbose=True):
3     from numpy_sugar.linalg import economic_qs
4     from numpy import pi, var, diag
5     from glimix_core.glmm import GLMMExpFam
6     from glimix_core.lmm import LMM
7     from limix._data._assert import assert_likelihood
8     from limix._data import normalize_likelihood,
    conform_dataset
9     from limix.qtl._assert import assert_finite
10    from limix._display import session_block, session_line
11    lik = normalize_likelihood(lik)
12    lik_name = lik[0]
13    with session_block("Heritability analysis", disable=not
    verbose):
14        with session_line("Normalising input...", disable=not
    verbose):
15            data = conform_dataset(y, M=M, K=K)
16        y = data["y"]
17        M = data["M"]
18        K = data["K"]
19        assert_finite(y, M, K)
20        if K is not None:
```

```
21            # K = K / diag(K).mean()
22             QS = economic_qs(K)
23        else:
24             QS = None
25        if lik_name == "normal":
26             method = LMM(y.values, M.values, QS, restricted=True
    )
27             method.fit(verbose=verbose)
28        else:
29             method = GLMMExpFam(y, lik, M.values, QS, n_int=500)
30             method.fit(verbose=verbose, factr=1e6, pgtol=1e-3)
31        g = method.scale * (1 - method.delta)
32        e = method.scale * method.delta
33        if lik_name == "bernoulli":
34             e += pi * pi / 3
35        v = var(method.mean())
36        return g , v , e
```