UPPSALA
UNIVERSITET

UPTEC X 20018
Examensarbete 15 hp
Juni 2020

Neural networks for imputation of missing genotype data

An alternative to the classical statistical methods in bioinformatics

Alfred Andersson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Neural networks for imputation of missing genotype data

Alfred Torsten Andersson

In this project, two different machine learning models were tested in an attempt at imputing missing genotype data from patients on two different panels. As the integrity of the patients had to be protected, initial training was done on data simulated from the 1000 Genomes Project. The first model consisted of two convolutional variational autoencoders and the latent representations of the networks were shuffled to force the networks to find the same patterns in the two datasets. This model was unfortunately unsuccessful at imputing the missing data. The second model was based on a UNet structure and was more successful at the task of imputation. This model had one encoder for each dataset, making each encoder specialized at finding patterns in its own data. Further improvements are required in order for the model to be fully capable at imputing the missing data.

Handledare: Carl Nettelblad
Ämnesgranskare: Prashant Singh
Examinator: Fabien Burki
ISSN: 1401-2138, UPTEC X20 018

Sammanfattning

Användningen av neurala nätverk som maskininlärningsmodeller har ökat markant de senaste åren. Ett stort bidrag till detta är att området har mognat, priset på beräkningskapacitet har sjunkit och mängden tillgängliga data har ökat. Särskilt det senare är relevant inom biologin då effektivare tekniker har utvecklats och kostnaden för att samla in biologiska data har gått ner. Detta har gjort det möjligt att samla in mer information om biologiska system och undersöka dem närmare.

Varje individ har en dubbel uppsättning arvsmassa i form av 46 stycken kromosomer, 23 från vardera förälder. Kromosomerna består av långa sekvenser av DNA och dessa sekvenser skiljer sig från person till person. Om en upssättning arvsmassa skulle lagras i en dator skulle filstorleken vara ungefär 3GB, vilket ungefär motsvarar den mängden data som krävs för att streama en film i HD i en timme. Lyckligtvis är all denna information inte nödvändig då majoriteten av informationen är densamma för alla människor och redan finns tillgänglig. Genom att endast fokusera på delarna i arvsmassan som skiljer oss åt, så kallade markörer, kan man drastiskt dra ner storleken på genetiska data och det blir genast lättare att lagra data och använda maskininlärningsmodeller för att hitta komplexa biologiska mönster i data. Det blir även betydligt billigare att enbart undersöka dessa markörer istället för hela arvsmassan.

Under mitt examensarbete har jag undersökt maskininlärningsmodeller med ändamålet att fylla i saknade genetiska data. Examensarbetet var ett samarbete med Karolinska Institutet i Stockholm som försåg projektet med två set patientdata från olika paneltester och en panel undersöker en specifik uppsättning av markörer. I grova drag innehöll båda panelerna samma antal markörer fast hälften av markörerna som ingick i den ena panelen ingick inte i den andra. I de flesta av fallen fanns individen på båda panelerna. I de få fall där individen endast fanns på en panel måste saknade data från den andra panelen fyllas i med hjälp av maskininlärning så att all information fanns tillgänglig.

Att jobba med genetiska sekvenser inom maskininlärning påminner mycket om att jobba med bilder. Gener nedärvs inte slumpmässigt utan gener som ligger nära varandra på kromosomen har en större sannolikhet att nedärvas tillsammans. Maskininlärningsmodellen kan då lära sig vilken genetisk variant som med högst sannolikhet ligger mellan två övriga platser på kromosomen där varianterna är kända. Jämför detta med en bild på en banan och att du får till uppgift att gissa vilken färg en pixel på bananen har. Om du får se färgen hos de övriga pixlarna kommer du förmodligen att svara något i stil med *medelfärgen av alla närliggande pixlar*. Detta svar är inte långt ifrån sanningen och gäller oavsett om pixeln ligger på en grön, gul eller brun fläck på bananen. Tanken var att mina modeller skulle upptäcka liknande genetiska mönster i paneldata.

Innan jag kunde använda mitt neurala nätverk måste genetiska data översättas från bokstäver till siffror för att datorn skulle kunna arbeta med dem. Antag att det finns två tänkbara varianter för en markör – en vanlig och en ovanlig variant. Eftersom en individ ärver en uppsättning arvsmassa från vardera förälder kommer vi få tre olika fall. I det första fallet ärver individen en dubbel uppsättning av den vanliga varianten och då har markören värdet 1. I det andra fallet har individen fått en av vardera variant och markören har värdet 0,5. I det tredje fallet har individen ärvt en dubbel uppsättning av den ovanliga varianten och markören har värdet 0. Nätverket kommer sedan att tränas i att komprimera och återskapa ursprungliga sifferdata. Efter varje iteration av träningen får nätverket veta hur bra den har presterat med att återskapa data. Eftersom delar av informationen förloras vid komprimeringen kommer modellen att lära sig att komprimera så att så mycket av informationen som möjligt bevaras.

Att jobba med patientdata är känsligt, särskilt då den genetiska informationen säger mycket om en person och patienternas integritet måste skyddas, till exempel genetiska sjukdomar. Jag använde mig därför först av data från 1000 Genomes Project för att simulera data som innehöll samma markörer som patientdata från de två olika panelerna. 1000 Genomes Project innehåller genetisk data för ett par tusen individer och vem som helst får använda dem.

Efter jag simulerat mina data tränade jag en typ av modell som bestod av två delmodeller. Delmodellerna skulle lära sig att komprimera data från varsin panel och sedan återskapa dem. I de fall där en individ hade data på båda panelerna skulle delmodellerna slumpvist byta komprimerade data med varandra och fortfarande lyckas återskapa dem som om det vore dem själva som hade komprimerat informationen. Tanken var att det sedan skulle gå att mata in information från den panelen man kände till i motsvarande delmodell, ta komprimerade data från delmodellen och stoppa in den i den andra delmodellen som skulle återskapa paneldata som saknades. Denna modell lyckades inte särskilt bra då modellen inte ens kunde slå basfallet, vilket var att gissa att alla genetiska varianter var den vanligaste. Faktum var att den inte ens kunde återskapa sina egna data.

Jag bytte därför strategi till en annan typ av modellstruktur. Denna modell bestod av två olika delmodeller som komprimerade sina egna paneldata, men de hade en gemensam del för att återskapa data. Modellen sparade också information från komprimeringen och använde sedan denna information när den skulle återskapa data. Denna modell lyckades betydligt bättre än den tidigare modellen. Den slog basfallet och kunde så gott som alltid återskapa kända data. Denna modell prövades slutligen på patientdata och klarade av basfallet men den presterade inte lika bra som den hade gjort på simulerade data. Oavsett kunde denna modell gissa bättre än basfallet, vilket visade att det finns potential för moderna maskininlärningsmetoder inom biologin.

Table of contents

Abbreviations

1KGP	1000 Genomes Project
A	adenine
BCE	binary crossentropy
C	cytosine
CCE	categorical crossentropy
chr20	chromosome 20
CVAE	convolutional variational autoencoder
DNA	deoxyribonucleic acid
G	guanine
GSA	Global Screening Array
GWAS	genome-wide association studies
MEGA	Multi-Ethnic Global Array
MS	marker-specific (variables)
MSE	mean square error
NMS	new marker-specific (variables)
NN	neural network
PCA	principal component analysis
SNP	single-nucleotide polymorphism
T	thymine

1 Introduction

There are numerous applications in genome research where features in the data cause problems when using statistical models for analysis. An example of such applications is when there are many missing variables. This can occur when analyzing different panels of individuals which have been genotyped using different single-nucleotide polymorphism (SNP) chips. Imputation of non-observed genotypes is a method which can be used to normalize data from multiple cohorts, e.g. in genome-wide association studies (GWAS). Statistical models used for imputation, which are based on the expected structure of the genome, will however heavily depend on the panel it is based on (Li *et al.* 2009). Non-ideal features in the data, like error rates for some SNPs, will result in undesirable patterns despite extensive attempts to perform quality control on the source data.

The use of machine learning, especially neural network (NN) models, has increased in recent years. This applies to many fields of science, biology and medicine to name a few. One of the reasons for this increase is that NNs have time and time again proven to be capable of detecting patterns in data that the classical statistical methods can not detect as easily. This feature makes NNs a promising tool for imputation in GWAS.

2 Background

The idea for this project came from a collaborator at Karolinska Institutet in Stockholm. They had two sets of genotype data from negative controls, which had been involved in studying psychiatric conditions in patients. A majority of the individuals have been genotyped on both of the panels and a minority of the individuals have only been genotyped on one. The task was to train different types of NNs and find a model that could impute the genotype data that were missing, i.e. in the cases where an individual has only been genotyped on one panel. The goal of the project was to find a model that was better at imputation of genotype data than existing models (Halperin & Stephan 2009). This project will hopefully contribute to increase the use of NNs in bioinformatics. Currently, there are only a few examples where NNs have been used for imputation in GWAS, e.g. Sun & Kardia (2008) and Chen & Shi (2019).

The work of this project was built upon research conducted in the Nettelblad group at the Division of Scientific Computing at Uppsala University, where NNs had been used for dimensionality reduction of genotype data as an alternative to principal component

analysis (PCA). Dimensionality reduction and imputation, in the context of NNs, are similar in the sense that the models need to capture important features in the data to succeed at their tasks. The existing scripts, which where used for training dimensionality reduction models, were thus a good basis for imputation.

3 Theory

As this is an interdisciplinary project, it is necessary to understand both the biological and the technical aspects of the task. It is relevant to understand the origin of the biological data, how to represent the biological data, the principles of machine learning, what machine learning model to use and why that model would succeed at the task. It was also relevant to find a way to validate the correctness of the trained models.

3.1 Origin of genetic data

Most of the cells in our bodies contain molecules of DNA, which is the blueprint of life. DNA has the instructions to create all imaginable varieties of life and consists of different sequences of the four nucleotides adenine, cytosine, guanine, and thymine which are referred to by the letters A, C, G, and T respectively. The genetic material of an individual is called the genome and the human genome consists of a sequence of three billion nucleotides (Venter *et al.* 2001).

Differences in the genetic sequence are responsible for the different genetic traits in a population, e.g. blonde or brown hair color. These traits are a result of evolution. Over time, the DNA molecules in our cells are damaged by the environment and, if the cells can not repair the damage, the genetic sequence will change. Letters in the genetic sequence can get substituted. This is called a mutation and it is usually neutral. In rare cases, a mutation can give rise to a new trait which will either be beneficial or harmful to the individual. It takes many generations for a beneficial mutation to replace the original variant in a population. Neutral mutations can, in theory, go unnoticed forever. The way in which genetic material is passed on is also, to some extent, random. This makes it possible for multiple genetic variants to coexist within a population. If there exist two or more possible letters for a specific position in the genome it is called a SNP and it is the most common type of variation within the genome (Twyman 2009). As an individual

inherits two sets of chromosomes, one from each parent, it is possible for an individual to have two different variants of the same SNP.

Sequencing all three billion bases in a genome is expensive and all parts of the genome is not informative. Most of the sequences are the same for all humans and as there already exists a reference human genome, it is only necessary to focus on the regions that differ. As the SNPs alone cover a lot of these differences in an individual, they are informative and useful in GWAS. As the set of SNPs is smaller than the full genome it is easier to store and process. It is also cheaper to genotype SNPs than sequencing full genomes. SNPs can be genotyped using different SNP arrays, which capture an array-specific subset of markers in the genome. Furthermore, the closer two genomic regions are to one another, the more likely are they to be inherited together due to genetic linkage. This implies that if a SNP is known, it is possible to infer likely genotypes around that SNP and fill in the blanks based on that inference.

3.2 Working with genetic data

Even though it might be appealing to store SNP data as their corresponding nucleotide letters, it is not the most memory efficient approach. One of the possibilities is to store the genetic information in a PLINK format (Purcell *et al.* 2007). PLINK files consist of a .bed file that stores all of the raw SNP data in a binary format as 00, 01, 10, and 11. The .bed file assumes that there are a reference and an alternate allele at each position. For the data in this project, the reference is the common SNP variant and the alternate is the uncommon SNP variant. Recall that an individual has two sets of chromosomes, then the binary numbers translate to double uncommon variant (00), missing genotype (01), one of each variant (10), and double common variant (11). The .bed file is accompanied by a .bim and a .fam file. The .bim file contains the name and position of each SNP in the .bed file. It also has information on which nucleotide letter the reference and alternate variants correspond to. Finally, the .fam file contains metadata of every individual in the .bed file, e.g. what sex an individual has and which population they belong to.

The binary numbers 00, 01, 10, and 11 in the .bed file do not translate into their corresponding decimal numbers 0, 1, 2, and 3. Instead, they translate into 0, NaN (not a number), 1, and 2. The NaNs have to be replaced by actual numbers before the data are processed by the NN, otherwise the loss will also be NaN. The NaNs are set to the common variant 2, which leaves us with 0, 1, and 2. In this project, the data were also normalized. Normalization is done by making sure that all values of the data are between 0 and 1. For this particular case, the three numbers above are set to 0.0, 0.5, and 1.0 respectively.

3.3 Neural networks

In recent years, NN models have proven to be a promising tool in machine learning. NNs can find patterns in high-dimensional data that classical statistical models have not been able to do. Even though the behavior of NNs are complex and have not been fully understood yet, the principles they are built upon are not as complex.

NNs consist of an input layer, one or more hidden layers, and an output layer (Michelucci 2018). Every layer consist of nodes, which are connected to the nodes in the other layers by adjustable weights. The input layer is where the input data enter the neural network and the output layer is where the network makes its predictions. The hidden layers calculate intermediate states of the data and these states are not necessarily interpretable by humans. As the input passes through the NN and enters the first hidden layer, it gets multiplied by the weights of that layer. The result then passes through an activation function before getting passed on to the following layer in the NN. The activation function of artificial neurons was originally inspired by the on-off switch in biological neurons. The activation function produce a signal and the stronger this signal is, the more likely is it to propagate further into the network. There are many different activation functions and which function to use varies between applications. The input passes through all layers until the output layer is reached.

Training a NN is an optimization problem, which can be to either minimize or maximize an objective function. This function is used to calculate a metric to evaluate the performance of the model. If the task is to minimize the objective function, the objective function is usually called a loss function. The task is then to minimize the loss of the loss function as the model is training. The loss of the model is calculated after each iteration of training and is necessary for the NN to know whether it is improving or not. Every time the model has trained on all of the training data it has completed an epoch and it usually requires many epochs for the model to converge to a solution where no further improvement to the loss is made. By using back propagation (Rumelhart *et al.* 1986), the NN knows how the loss was calculated and uses this information to adjust the weights in an attempt at decreasing the loss for the following iteration. After an iteration of training, the gradient of the loss function w.r.t the weights are calculated. The gradient tell the model which weights to be adjusted and how. An optimizer will then apply this gradient and update the weights of the model. There are multiple optimizers to choose from and each optimizer has a different way of adjusting the weights. One of the most common and general optimizers today is the Adam optimizer (Kingma & Ba 2017). Optimization is not always an easy task. The model is constantly looking for a minimum of the loss function and if the adjustments of the weights are too big, it will pass the minimum. Similarly, if the adjustments are too small, it will never reach the

minimum (Murphy 2012). This is done for thousands of parameters which makes the task of optimization even more challenging. The loss function also has multiple minima and the model can get stuck in a poor minimum and never proceed to a better one.

A total of three loss functions were considered for the project. The first loss function was mean square error (MSE), presented in Equation 1. N corresponds to the total number of SNPs that are being predicted, y_i the true value of the ith SNP, and $\tilde{y}_i$ the predicted value of the ith SNP. MSE is a common loss function in machine learning and it is usually used for regression.

$$MSE(y, \tilde{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \tilde{y}_i)^2 \tag{1}$$

The second loss function was binary crossentropy (BCE), presented in Equation 2 (Mathieu *et al.* 2016) with the same variable notations as for Equation 1. BCE is logarithmic, which makes it more sensitive on the interval $[0, 1]$ compared to MSE.

$$BCE(y, \tilde{y}) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(\tilde{y}_i) + (1 - y_i) \cdot log(1 - \tilde{y}_i) \tag{2}$$

The third and last loss function was categorical crossentropy (CCE), presented in Equation 3 (Harikrishnan *et al.* 2020). Here, M corresponds to the total number of classes, N the total number of SNPs, $y_{i,j}$ the true probability of the jth class of the ith SNP, and $\tilde{y}_{i,j}$ the predicted probability of the jth class of the ith SNP.

$$CCE(y, \tilde{y}) = -\frac{1}{N} \sum_{j=1}^{M} \sum_{i=1}^{N} y_{i,j} \cdot log(\tilde{y}_{i,j}) \tag{3}$$

As every SNP can be either 0.0, 0.5, or 1.0, there are a total of three classes per SNP, i.e. $M = 3$. If the true value for a SNP is 1.0, the true class probabilities are [0, 0, 1]. Only the correct class will be assigned 1 and the remaining classes will be assigned 0. When the NN makes a prediction, it generates one scalar for each SNP. Hardy-Weinberg equilibrium (Edwards 2008), presented in Equation 4, is then used to convert each of these scalars to their predicted class probabilities. Each scalar output corresponds to the allele frequency p of the common variant and each term in the left-hand side of Equation 4 corresponds to a class probability.

$$(1-p)^2 + 2p(1-p) + p^2 = 1 \tag{4}$$

If the NN predicts the scalar to be $p = 1$, then the class probabilities are [0, 0, 1] and the loss will be minimized. If the NN guesses the scalar to be $p = 0.8$, then the class probabilities are [0.04, 0.32, 0.64] and the loss will be higher. Even though 1.0 still is the most likely class out of all three classes with a probability of 64%, it still assumes that the class 0.0 is 4% likely and the class 0.5 to be 32%. CCE is used for single label categorization in classification, i.e. the cases where only one answer is correct. CCE also shares the logarithmic behavior with BCE, i.e. predictions which are further away from the real value will produce higher losses. CCE was chosen for the project as it seemed to be the most promising out of the three losses.

During optimization, it is possible for the model to overfit (Lawrence *et al.* 1997). Overfitting occurs when the model finds patterns which exist only in the training data. If an overfit model then tries to process new examples, the performance will be worse. To make sure that the model does not overfit, a subset of the samples in the data will be saved as a validation set that will never take part in the training (James *et al.* 2013). By using the validation set, it is possible to confirm whether the model finds general patterns or not. Data is still limited in GWAS, which makes it important to choose a validation set of the right size. If too few samples in the data are used for validation, the loss of the validation set will not be reliable. On the other hand, if too many of the samples in the data are used for validation there will not be a representable amount of data left for training. In this project, 80% of the data were used for training and 20% for validation as this was used for the models used for dimensionality reduction.

3.4 Model architecture

The architecture of a NN model is defined by the layers it consists of. Convolutional variational autoencoders (CVAE) are a type of NNs which are symmetric and are good at finding patterns in data (Pu *et al.* 2016). They compress the input data by encoding it into a low-dimensional representation and decode it to recreate the original input. The model will be forced to find the most important features in the data as some information will inevitably get lost as it gets encoded. The training is unsupervised, as it only needs to know what the input looks like to confirm whether the data have been recreated or not. The performance is measured by calculating the loss between the input and the predicted output. A common application for trained CVAEs in image analysis is to remove noise from images. The CVAEs have learnt what similar images look like by finding patterns which define those images. By finding the most relevant patterns in the images they do

not need to have the information that got lost by the noise to recreate the original images. In the context of imputation, the missing data in the SNP data can be seen as the noise in the images. CVAEs can be trained on data where all of the SNP data are present and then be used to impute the data for the cases where parts of the data are missing.

CVAEs are memory efficient as they only use small and general convolutional filters. Finding features in the data is a process of data compression and each step of compression consists of convolutional filters and max pooling. The convolutional filters find features in the data and max pooling keeps the most significant values in these features and reduces the dimensionality of the data. After these compressions, the information reaches the dense layers in the middle of the network. Dense layers connect all of their adjacent nodes, making the nodes fully connected. As the information passes through the middlemost dense layer of the CVAE, also known as the latent layer, the information is in its most compressed state. This information is then decoded in a similar way as it was encoded by using similar operations. Convolutional filters are still used, but upsampling is used instead of max pooling. The first half of the CVAE is usually referred to as the encoder and the second half as the decoder. An illustration of a simple CVAE can be seen in Figure 1.

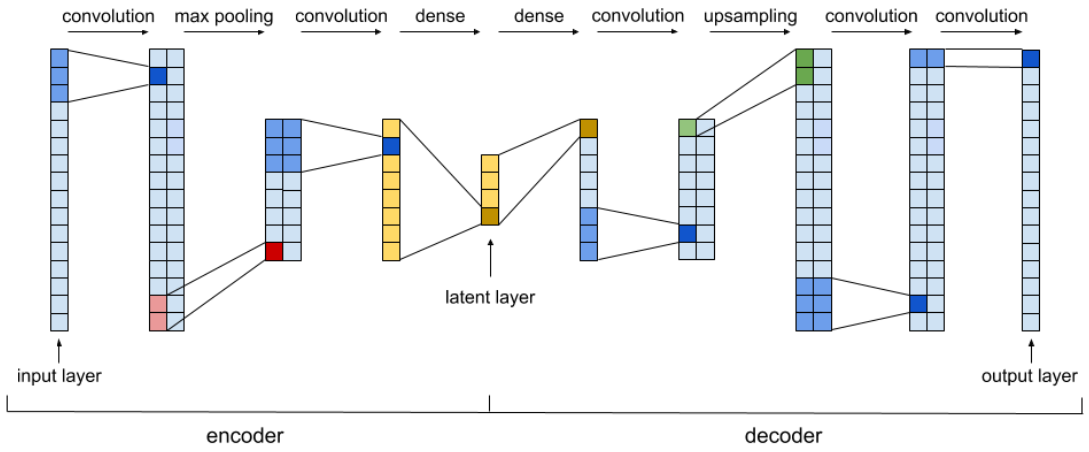

Figure 1: A simple convolutional variation autoencoder with one pair of convolution and max pooling in the encoder and one corresponding pair of convolution and upsampling in the decoder. Convolutions are marked as blue, max pooling as red, dense as yellow and upsampling as green.

3.5 Injecting trainable variables

To add extra support of the model, it is possible to train extra variables which will be injected into the network. These variables are trained in the same way as the other

weights in the network. The idea is that the trainable variables will find general patterns in the data and the other weights in the model will try to cover the variance between samples. Furthermore, these trained variables will also to some extent compensate for missing data. Marker-specific (MS) and new marker-specific (NMS) variables were tested in this project. The difference between these two sets of variables is the layer of the network which they get injected into.

3.6 Hyperparameters

When the training starts, the weights of the model will be initialized and it will do its best to find a solution that minimizes the loss function. Even though there will be minimal input from the user during training, it is still possible to increase the chances of finding a general solution before the training starts. This is done by selecting suitable hyperparameters for the model, which will guide the training. Hyperparameters are not derived from training and are manually tuned. The number of relevant hyperparameters varies between models.

3.6.1 Initial learning rate

The initial learning rate is an important hyperparameter for all models. It tells the model how the first weight adjustments should be scaled. As mentioned earlier, the model will struggle to find a good solution if the adjustments are too big or too small (Murphy 2012). The learning rate can also be decreased over time by including exponential decay, which will lower the learning rate at regular intervals.

3.6.2 Batch size

Batch size is relevant in the context of genomic studies where the data are large. If the data are too large, it will not fit into the memory of the computer. The data have to be split into batches to bypass this issue and the batch size tells the model how many samples to process at once. For instance, assume that the data consist of 1477 samples and that the batch size is 100. The model will then train on 14 batches of 100 and one batch of 77. After all batches have been processed, an epoch is completed.

3.6.3 Noise level

Adding Gaussian noise to the latent layer is a way of preventing the model from overfitting. The high-dimensional input will be encoded in a low-dimensional space in the

latent layer, which forces the model to learn how to keep as much of the original information as possible. By adding noise to the bottleneck of the network, it will be even harder for the model to find patterns in the data, but the patterns which it finds will be more general to the data as a whole. Adding Gaussian noise with a high standard deviation will make the training process longer but the model is less likely to be overfit.

3.6.4 Regularization factor

One undesirable way for the model to adapt to the noise in the latent layer is to make the magnitude of the output significantly larger than the standard deviation of the noise. In other words, the weights of this layer will explode. To prevent this weight explosion, it is possible to add a regularization term of these weights to the loss function. A common regularization technique is L2 regularization. By setting a regularization factor, the L2-norm of the weights multiplied by the regularization factor will be added the the loss function. This will keep the weights reasonably small as large weights will contribute to a significant addition to the loss.

3.6.5 Dropout

The network will not rely too heavily on specific weights if the result of those weights are removed at random during training. This technique is called dropout and it forces the network to use multiple weights to recognize patterns (Srivastava *et al.* 2014). If the dropout rate of a layer is 0.1, then 10% of the nodes in that layer will be ignored and the remaining 90% will be used further on in the NN. Which nodes to remove and to keep are chosen at random each iteration. Over a long period of time, any set of 90% of the nodes will be able to represent the data in a way that minimizes the loss.

3.7 Genotype concordance

Validating if a trained model succeeds at the task is a crucial step in machine learning. The loss function measures the performance of the model but it is hard to tell how accurate the model is if the loss value is, for example, 0.56. An analytical tool which assumes exact genotype values instead of probability distributions is required. In other words, if a SNP is 5% likely to be 0.0, 15% likely to be 0.5, and 80% likely to be 1.0, then this tool will assume that the SNP is 1.0 and check against the real SNP value whether the prediction is correct or not.

One way of measuring the performance this way is calculating the genotype concor-

dance. This statistic measures the performance of correctly labeled predictions on a scale from 0 to 1. When validating our models, all SNP data will be available. SNPs from the dataset that is assumed to be missing are removed and to be imputed. Only the SNPs of the other dataset remain. Imputation is done and as a reference exists with SNP data for both datasets, it is possible to measure the performance of reconstructing the known SNPs, the imputed SNPs, and the combined sets of SNPs. For each of these three cases, a unique genotype concordance value can be calculated.

Self-concordance is the performance of reconstructing the known SNPs. Even though the task is to impute missing data, it is still useful to use this statistic to confirm that the model is able to capture the structure of the full data. Anti-concordance is the performance of imputing the missing SNPs. These SNPs are not used as input, but they will still be predicted and compared to the available reference to calculate the performance. Total concordance is a weighted average of the other two concordances and measures the overall prediction performance of the model. For example, assume that there are a total of 3000 SNPs, where 2000 SNPs are known and 1000 SNPs are missing. If 1900 of the known SNPs are reconstructed correctly, then the self-concordance value will be $1900/2000 = 0.95$. If 800 of the missing SNPs are correctly imputed, then the anti-concordance value will be $800/1000 = 0.80$. The total concordance is then $(2000 \cdot 0.95 + 1000 \cdot 0.8)/(2000 + 1000) = 0.90$

4 Materials

4.1 Data

4.1.1 Patient data

The genotype data for this project was provided by Karolinska Insitutet, and consisted of two different panels. The data of the two panels were negative controls and were collected using the Global Screening Array (GSA) and the Multi-Ethnic Global Array (MEGA), designed by Illumina. Quality control was done by the client at Karolinska Institutet, keeping only high quality SNPs in the data. All of the SNPs which had different SNP names but identical chromosomal positions were removed from the data to exclude duplicate genomic sites. As seen in Table 1, a majority of the patient samples had been genotyped on both panels and a minority on just one of them.

It is a resource-intensive task to train NN models on the full data, which made it nec-

essary to use a subset of the data. The SNPs on chromosome 20 (chr20) of the two panels were chosen as this subset. It is common practise to choose chr20 as a subset in GWAS, as it is a relatively small chromosome and it does not suffer from abnormalities such as trisomy, i.e. three chromosomes instead of two within a pair of chromosomes (Mavromatidis *et al.* 2010). A summary of the chr20 subsets can be seen in Table 1.

Table 1: Summary of the two sets of patient data.

data	# samples	# SNP.full	# SNP.chr20
GSA only	51	268123	6484
MEGA only	33	208892	4876
GSA and MEGA	1336	220115	5266
total	1420	697130	16626

4.1.2 Simulated data

The patient data were sensitive, which made it much more convenient to initially work with simulated data instead, since no security measures are needed. Simulated data have a similar structure to the real data and it can be processed in any environment without risking the integrity of the patients. Data from the publicly available 1000 Genomes Project (1KGP) were used as input to mimic the structure of the patient data (The 1000 Genomes Project Consortium 2015). All of the bi-allelic SNPs in the 1KGP dataset, which also were present in the GSA and the MEGA datasets, were used to train the initial models. The summary of the simulated patient data is presented in Table 2. Almost all of the data on the GSA and MEGA panels were also present in the 1KGP dataset. Please note that the 1KGP datasets had 2504 samples compared to the patient datasets which had 1420.

Table 2: Summary of the two full sets of simulated patient data. The last column shows how many % of the original SNPs which also were present in the 1KGP dataset.

data	# samples	# SNP	% of data
GSA only	-	256445	95.6
MEGA only	-	203950	97.6
overlap	2504	212507	96.5
total	2504	672902	96.5

A chr20 counterpart of the simulated data was also created to make training and testing of the different models faster. The simulated chr20 datasets are summarized in Table 3.

Table 3: Summary of the two chromosome 20 subsets of simulated patient data. The last column shows how many % of the original SNPs which also were present in the 1KGP dataset.

data	# samples	# SNP	% of data
GSA only	-	6480	99.9
MEGA only	-	4820	98.9
overlap	2504	5262	99.9
total	2504	16562	99.6

4.2 Hardware

Resources on UPPMAX in Uppsala and HPC2N in Umeå were approved for this project. The Kebnekaise resource on HPC2N was used for simulating the test data and the Rackham and Snowy resources on UPPMAX were used for the intial testing of the model on the simulated data. The Bianca resource on UPPMAX was used for training models on the actual patient data, as the patient data were sensitive and needed an environment where it could be processed without risking the integrity of the patients. Rackham and Bianca have access to CPUs while Kebnekaise and Snowy have access to both CPUs and GPUs.

4.3 Software

TensorFlow 2 was used to train the different models in the project. TensorFlow 2 is a package used to train NNs in Python versions 3.5 or higher (Abadi *et al*. 2016). Machine learning models require high performance as it is a demanding task by the computer to access and tune the different parameters. In the context of high performance computing, Python is usually criticized for being a high-level programming language, which means it has a lower performance. Even though TensorFlow is implemented in Python, its core is written in C++ to bypass this issue. C++ is also considered a high-level language but it is a lower level language compared to Python.

The Nettelblad group had already developed a code base for training CVAE models on

SNP data. The intended application for this code was dimensionality reduction and thus changes had to be made to the make the code suitable for imputation.

4.4 Version control

The existing code base was already collected in a GitHub repository, which made GitHub the best choice for version control in this project. This made it easier to create copies of the original repository and track changes and bug fixes within the Nettelblad group.

5 Methods

The main task of the project was to come up with a new NN model which was able to learn the different features in each of the chip datasets. The initial plan was to focus on doing minor changes to the existing model, e.g. adding more densely connected layers to the middle of the network and increasing the number of dimensions in the middle layers. However, this proved to be more difficult than initially thought and two different CVAE approaches were implemented and tested.

5.1 Model 1 - two encoders, two decoders

The first of these two approaches was training two CVAEs in parallel, one for each panel. Each CVAE was intended to encode its own input and then decode the encoded data. In the cases where data are present on both panels, it is possible for both encoders to process their own panel data at the same time. The input and output sizes of the two CVAEs correspond to the number of SNPs on each panel, which not necessarily are the same for both CVAEs. The encoded data, on the other hand, can be set to the same size by using the dense layers in the middle of each network.

If the encoded data of both CVAEs have the same dimension after the data have been encoded, it is possible to shuffle the encoded data between the CVAEs. If the encoded data are swapped, the decoder of the first CVAE would still have to reconstruct the data of the first panel even though it has the encoded data of the second panel. The same holds for the second decoder. The two CVAEs would be forced to produce near-identical encoded representations of the data or the loss of the loss function will be too high as the

decoders would use nonsense to reconstruct their input. If the encoded data are identical, it does not matter where the original data came from. The decoder would still be able to reconstruct the data as both encoders have captured the same patterns but in different datasets. This model can then be used in the cases where data are missing. The data can be encoded by its corresponding encoder and decoded by the other decoder as a way of imputing the data of the missing panel. An illustration of the model can be seen in Figure 2.

Initially, it took a long time to fully understand the existing code and implementing this model structure. The original code was only compatible with processing one dataset at a time and was only using one CVAE. Furthermore, the original models could not read files in the PLINK format. Extensive changes to the code were required to make the model suitable for the current application.

As soon as the model was fully functional and worked on simple test cases, different changes were done to the model to test whether they would increase the performance of the model or not. Changes which were tested include different initial learning rates; batch sizes; levels of added noise in the latent layer; regularization factors of the weights in the latent layer; levels of dropout rate for the convolutional, max pooling, upsampling, and dense layers; number of convolutional and max pooling layers; filter sizes of the convolutional layers; number of filters of the convolutional layers; number of dense layers in the middle of the network; and sizes of the dense layers. These experiments were only tested on the simulated chr20 datasets.

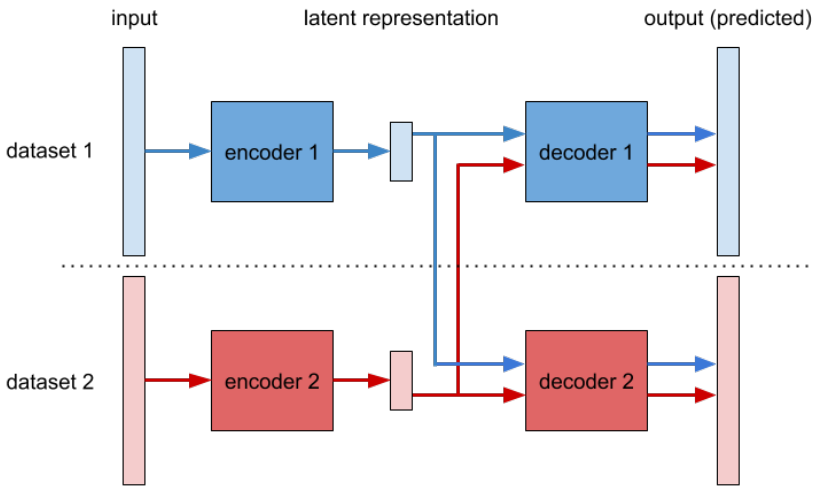

Figure 2: The architecture of the first model. The blue components correspond to those of the first dataset and the red components correspond to those of the second dataset.

5.2 Model 2 - two encoders, one decoder

The second model structure was a bit different than the first. The second model followed a UNet structure to improve the performance of the model. UNets save outputs from hidden layers in the encoder and inject them at their corresponding position in the decoder. This made it possible for the model to keep information from intermediate states and still find relevant features in the original input. As more of the original information is preserved this way, the model does not have to focus on reconstructing the SNPs which are known. Instead, the model could be trained on imputing the data that are missing.

Changes were made to the code of the first model to make it compatible with the UNet structure. As the code already could process multiple datasets and read PLINK files, it did not take too long to finish the second model. To make UNets work, the intermediate outputs must have the the same shape as the layer they are injected in. This made the structure of the first model unsuitable as a UNet, as the corresponding layers in each autoencoder had different shapes. If the encoded data were swapped and the output of a layer from one of the encoders were injected into the corresponding layer of the other decoder, it would not fit. To tackle this problem, both autoencoders had to have the same architecture with identical layer shapes.

The first way to solve this problem was to make sure that the shape of the inputs were the same. This was done by sorting the SNPs in the input based on their genomic position. If a SNP was missing on one of the panels but present on the other, it would be assigned a random genotype value on the panel which was missing it. This ensured that every input had the same shape and covered all of the SNPs in both datasets, whether they were missing or not.

Similarly to the first model, two encoders were used. Each encoder was trained to find patterns in one of the datasets. The encoded data of both encoders, as well as the intermediate layer outputs, were processed by a shared decoder. Only one decoder was used as two separate decoders would accomplish the same thing, i.e. reconstruct the input with the same encoded data. This also reduced the number of weights in the model. Before training the different encoders, the SNPs which were not present on the current panel had to be replaced by a random genotype every iteration of the training to make sure that the encoder would not capture nonexistent patterns. Otherwise it would not be able to impute data later on. The loss was still calculated using all of the SNPs as a reference to confirm that the model could reconstruct the known data and impute the missing data. An illustration of the model can be seen in Figure 3.

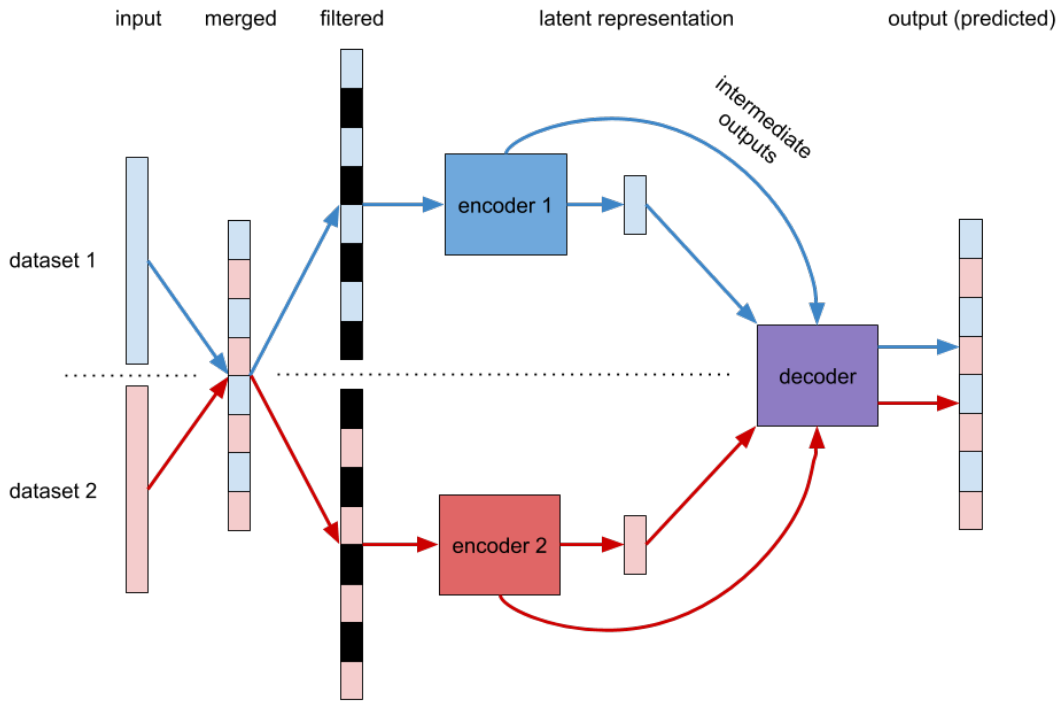

Figure 3: The architecture of the second model. The blue components correspond to those of the first dataset and the red components correspond to those of the second dataset. Please note that the decoder is purple to indicate that it is shared between the two datasets.

Due to time constraints, the UNet structure was not as extensively explored as the first model. Changes which were tested include different initial learning rates; filter sizes of the convolutional layers; number of filters of the convolutional layers; and number of intermediate outputs which were injected into the decoder. The model was first tested on the simulated chr20 datasets, followed by the full simulated datasets. Lastly, the model was tested on the patient chr20 datasets, but only the patients who had data on both panels were used. A summary of the network architecture can be seen in Table 4 and a summary of the final hyperparameter values can be seen in Table 5. Please note the differences in batch size between the simulated data and the patient data. This was to make sure that every batch were of similar size during training.

Table 4: Summary of the encoder and decoder components of the final model.

encoder		
layers		**arguments**
5x	Conv1D[1]	filters: 32, kernel size: 3, padding: same, activation: elu
	MaxPool1D[2]	pool size: 2, strides: 2, padding: same
1x	Conv1D[3]	filters: 4, kernel size: 3, padding: same, activation: elu
1x	Flatten[4]	-
1x	Dense	units: 256, activation: elu
1x	Dense	units: 128
decoder		
layers		**arguments**
1x	Dense	units: 256, activation: elu
1x	Dense	units: same as [4] , activation: elu
1x	Reshape	target shape: same as [3]
1x	Conv1D	filters: 32, kernel size: 3, padding: same, activation: elu
5x	Reshape	target shape: same as corresponding [2]
	UpSampling2D	size: (2,1)
	Reshape	target shape: same as corresponding [1]
	Conv1D	filters: 32, kernel size: 3, padding: same, activation: elu
1x	Conv1D	filters: 1, kernel size: 1, padding: same
1x	Flatten	-

Table 5: Summary of the hyperparameters of the final model.

hyperparameter	**value**
initial learning rate	1e-03
batch size (simulated data)	101
batch size (patient data)	107
noise level	0.25
regularization factor	1e-07
dropout	none

6 Results

The two models had varying performance, both in terms of reconstructing the known data and imputing the missing data. As mentioned in Section 3.6, the genotype concordance metrics were used to measure the performance of the trained models. Total concordance measured how good the model were at recreating all of the SNPs, self-concordance the known SNPs, and anti-concordance the missing SNPs. For each of the three SNP sets, a baseline genotype concordance value was also calculated. The baseline value was used as a reference and was calculated by assuming all SNPs of that set were the common genotype, i.e. 1.0. Exceeding the baseline value proved that the prediction of the model could beat simple guessing.

6.1 Model 1

One of the requirements for imputing missing data was having near-identical latent representations between the two CVAEs. When plotting the coordinates of the latent layer from each CVAE, it was found out that the coordinates were indeed similar between the two CVAEs. Furthermore, the superpopulations in the 1KGP dataset were also separated from one another in clusters based on their geographic origin. This confirms that both encoders have not only captured relevant but also similar biological patterns in the different datasets.

Initial learning rate was the most important hyperparameter for this model, especially as the learning rate was constant throughout training. Having an initial learning rate of $\leq 10^{-4}$ made sure that the loss of the model converged in all of the test cases. Having a learning rate of $\sim 10^{-3}$, i.e. the same initial learning rate as the second model, gave the model a moderate convergence rate and the converged loss was lower compared to the cases with a lower initial learning rate. Having an initial learning rate of $\geq 10^{-2}$ gave the model a fast convergence rate in the beginning of training, but after a couple iterations the model lost its solution and never converged. In summary, it was more beneficial in terms of convergence to go for a higher initial learning rate but that benefit always came with a higher risk.

Using different batch sizes did not affect the result of the converged loss value. A smaller batch size implied a longer training time per epoch as the model would process the data in more batches and using a larger batch size implied a shorter training time per epoch. On the other hand, the smaller batch size converged faster than the larger batch size per epoch. In real time, the different batch sizes had a similar convergence rate.

When using dropout and a higher level of noise, the model got more stable for the higher initial learning rates. This also made the convergence rate slower and it was hard to confirm whether the model still would converge to a significantly lower loss than before. Nonetheless, using these changes made the model less likely to overfit as the validation loss was never higher than the training loss. The regularization factor of the weights in the latent layer had to be tuned to a value appropriate for the current level of noise, but it did not affect the loss to a large extent.

Increasing the depth of the network did improve the performance of the model, but only for the first few layers added. When increasing the number of dimensions of the dense layers and the number of dense layers in the middle of the model, a similar trend was observed. It got harder for the model to tune more weights and the training time got longer. Having filter sizes > 5 and a high number of filters in the convolutional layers did not contribute to an increase in performance.

It proved to be difficult to train the first type of models. Even though the first model was modified in multiple aspects and extensively tested, it was hard to find a model with a decent performance in terms of genotype concordance. In fact, neither of the changes done to the model made the total concordance exceed its baseline value. By the end of the project, it was found out that the MS and NMS variable injections were not working as intended due to a bug. The bug was found by the supervisor of this project and the bug caused the variables to never be injected nor trained. This could have contributed to the poor performance of this model.

6.2 Model 2

The second type of model proved to be much better than the first. When training on the simulated chr20 dataset, the second model immediately achieved better results. The total concordance and the self-concordance metrics exceeded their respective baseline value, which was not the case for the first model. This despite MS and NMS not working as intended. After some minor changes to the model, the anti-concordance metric also exceeded its baseline value. This proved that this type of model could be used for imputation. The performance of the final model which trained for 1000 epochs on the simulated chr20 datasets can be seen in Figure 4. This is the average performance of the simulated GSA and MEGA chr20 datasets. A more detailed summary of the converged results can be found in Table 6. The last row in Table 6, **average**, corresponds to the values of each line at epoch 1000 in Figure 4.

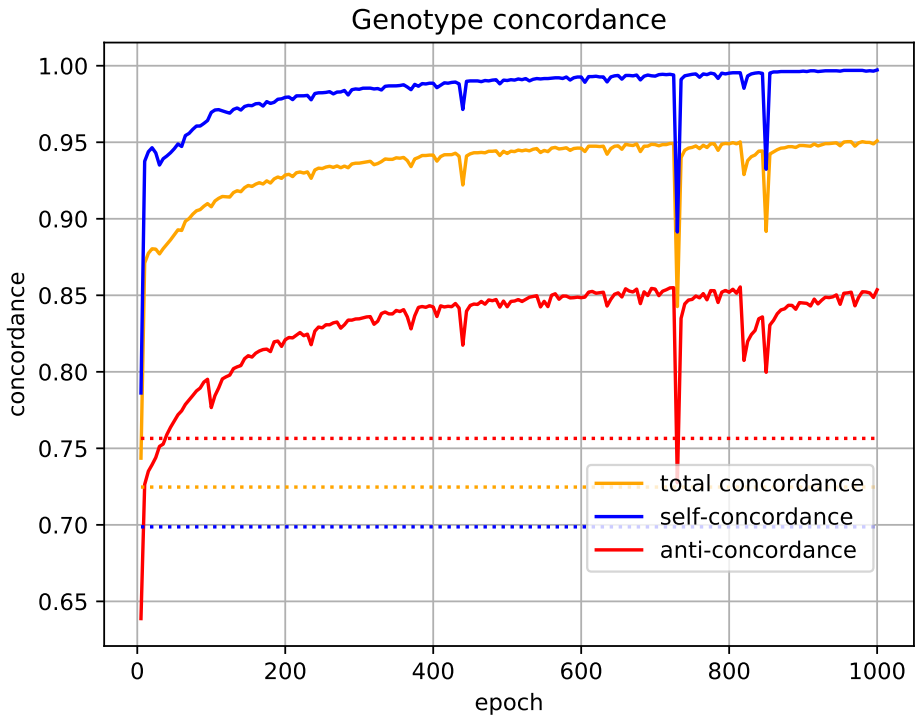

Figure 4: Performance of the final model on the simulated chr20 datasets. The solid lines represent the genotype concordance values over time. Each dotted line represents the baseline value corresponding to the solid line of the same color.

Table 6: Summary of the converged results at epoch 1000 for the simulated chr20 datasets. BL corresponds to the baseline value and CM to the value of the converged model. The average is a weighted average of the GSA and MEGA datasets and the weight of each dataset is calculated from the total number of SNPs in that dataset.

	total concordance		**self-concordance**		**anti-concordance**	
	BL	**CM**	**BL**	**CM**	**BL**	**CM**
GSA	0.7247	0.9420	0.7428	0.9960	0.6807	0.8100
MEGA	0.7247	0.9614	0.6474	0.9985	0.8450	0.9043
average	0.7247	0.9509	0.6987	0.9972	0.7565	0.8536

As the model was successful at imputing some of the missing SNPs in the simulated chr20 datasets, the next step was to test it on the full simulated datasets. Training on the

full datasets is a time-consuming task, as it would take months to run the model for 1000 epochs. It was however noted that the behavior of the model for the first 30 epochs was the same as for the chr20 run. This indicated that the model could work well on the full datasets.

The last test was training the same model on the chr20 SNPs on the actual data panels. The model did not perform as well as it did on the simulated data, but the three metrics still beat their own respective baseline. Similarly to the simulated data, the performance of the model trained on the real patient can be seen in Figure 5 and a more detailed summary of the converged results of can be found in Table 7.

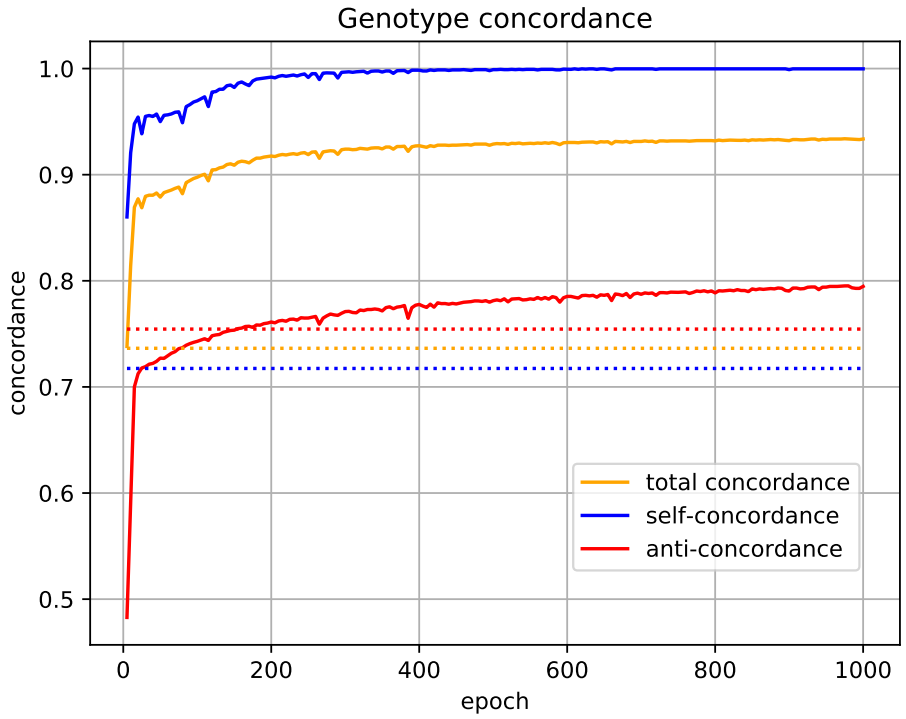

Figure 5: Performance of the final model on the real chr20 datasets. The solid lines represent the genotype concordance values over time. Each dotted line represents the baseline value corresponding to the solid line of the same color.

Table 7: Summary of the converged results at epoch 1000 for the real chr20 datasets. BL corresponds to the baseline value and CM to the value of the converged model. The average is a weighted average of the GSA and MEGA datasets and the weight of each dataset is calculated from the total number of SNPs in that dataset.

	total concordance		**self-concordance**		**anti-concordance**	
	BL	**CM**	**BL**	**CM**	**BL**	**CM**
GSA	0.7342	0.9220	0.7591	0.9995	0.6740	0.7340
MEGA	0.7387	0.9473	0.6689	1.0000	0.8479	0.8650
average	0.7364	0.9336	0.7174	0.9997	0.7545	0.7947

7 Discussion

The expectations of the first model were high in the beginning of the project. It was initially estimated to take a couple weeks to make the model work on the simulated chr20 data, but it proved to be much more difficult. After two months of testing, the genotype concordance of the model could still not exceed the baseline. Trying different values for each of the hyperparameters seemed pointless, as the gain in genotype concordance was close to insignificant. The genotype concordance always seemed to converge to a value which was under the baseline. A member of the research group tried to test the convergence of genotype concordance by training a different model on their own dataset. Their model managed to exceed the baseline, but this required a couple thousand epochs to achieve. This is unrealistic for this project, as the number of training iterations would translate to a couple weeks on the chr20 data and a couple years on the full data. This behavior is also not guaranteed for the datasets in this project.

The second model proved to be more successful than the first model. The idea of using the UNet structure came from another member in the group who was testing it on their models. They noticed that their models managed to reach genotype concordance values close to 1.0, i.e. a perfect reconstruction of the original data. These models were designed for dimensionality reduction, but the UNet structure still seemed promising for imputation. The model could keep the original information of the known SNPs by injecting the intermediate outputs of the encoder into the layers of the decoder. The model could then focus on finding the features to impute the missing SNPs. As can be seen in Tables 6 and 7, the converged self-concordances values are close to 1.0. This

means that the model can almost fully reconstruct the known SNPs of the simulated and the patient datasets. The anti-concordance values are a bit lower, but they still exceed their baseline. In Figures 4 and 5 it seems like the anti-concordance values still have not properly converged yet. On the other hand, it does not seem like there would be any significant gains from running the model for longer, as the model is close to converged already. The second model is still far from perfect at imputing missing data, but further changes to the model could potentially improve the performance.

It is also worth noting the differences between Figures 4 and 5. The concordances are oscillating more in Figure 4, which could be explained by the higher variation in the 1KGP data. What the model finds suitable for the current batch of data will not necessarily be optimal for the other batches if the data in the randomized batches are significantly different. In Figure 4 there are also major drops in concordance at epochs 730 and 850. This is most likely caused by the model leaving the minimum it had found. The model then finds its way back to the same minimum or a different minimum with a solution as good as the last one.

Another noticeable difference between the simulated and the patient datasets are the anti-concordances. Comparing Tables 6 and 7, the anti-concordance baselines are almost at the same level. The converged anti-concordance values, on the other hand, are noticeably different between the datasets. The simulated data make it to 0.8536 while the patient data make it to 0.7947. It would be interesting to investigate whether this was caused by the variation or the number of available samples within each dataset, as access to more samples tend to improve the performance of NNs. A possible experiment for the future could be to vary the number of samples in the simulated datasets. Similar experiments could be keeping all samples but varying the number of SNPs and checking whether lower-dimensional inputs perform better than than the full high-dimensional inputs.

The success of the UNet structure of the second model proves that too much information was lost by the first model. Even though both models have similar network architectures and are using the same operations, the information from the intermediate outputs of the UNet structure made a significant difference in terms of performance. As increasing the number of layers make training slower and might not necessarily improve the model, more prudent solutions are required.

A change which could improve the performance is using a learning rate with exponential decay. As the learning rate would decrease over time, it would be easier for the model to land in a minimum of the loss function. All of the tested models were assumed to be using exponential decay until the end of the project when it was observed that the learning rate was constant. Fixing this was easy but, as the project had already reached

its end, there was not time left to try it out. If one were to do further testing with these models, exponential decay could potentially improve the performance of both models.

In future testing it could also be useful to look at the impact of the different intermediate outputs which get injected in the decoder. It could be possible that the model relies too heavily on the first intermediate outputs. This could cause the model to mainly focus on recreating the known SNPs instead of being able to find patterns to impute the unknown SNPs. If too much of the early information is kept, the model will be too good at keeping that information, including the missing SNPs with randomized genotype values. A good compromise is obtained when the model has enough information from the intermediate outputs to represent the general features of the data and the decoded data has the information to fill in the remaining gaps.

Lastly, fixing the bug which disabled the MS and NMS variables could also improve the performance of the models. The second model was not as affected by the bug as the first model and one possible explanation could be the intermediate outputs in the second model which preserved more of the original information. By fixing this bug, the first model is more likely to be improved than the second, but the second model could also benefit from this fix.

Comparing the results of this project to those of Sun & Kardia (2008) and Chen & Shi (2019), the models in this project were not as successful. The model of Chen & Shi (2019) was also using a convolutional autoencoder, but they were only using one dataset. Furthermore, they used only one encoder and one decoder. Our models were more complex as they consisted of more components. Increasing the complexity of a model does not necessarily lead to an increase in performance, but for this project it was believed that the introduced changes would prove useful. It is also difficult to compare the performances between the projects as the structure of the data plays an important role. NNs have not been used to a large extent in biology and there are most likely many improvements and discoveries left to be made.

8 Conclusion

Training NNs for imputation proved to be more difficult than initially expected. A lot of time was spent on improving the first model but neither of the changes made the predictions better than simply guessing the most common genotype. In the end, a different model was developed. This UNet model managed to beat the simple guessing and performed better on the simulated data than on the patient data.

This project has proved that there is potential for NNs in genome analysis, but there are still room for improvement. Due to time constraints, this project only scratched the surface of what could be achieved with NNs in GWAS. Hopefully, this project has contributed to the field of machine learning in bioinformatics and better models with improved performances will be developed in the future.

It has been a useful learning experience to work with the different models. Even though it was hard to find a decent model which could impute the missing SNPs, a lot of different approaches were examined in an attempt to solve the problem. This experience will not just be useful for working with NNs, but machine learning in general.

9 Acknowledgements

My deep gratitude goes first to my supervisor Carl Nettelblad and my co-supervisor Kristiina Ausmees at the Department of Information Technology at Uppsala University for supervising for me throughout this project. Without their useful feedback and comments, this project would never have been realized. I would also like to thank Kristiina Ausmees for providing me with her TensorFlow code base. This made it possible for me to get a head start in my project.

I would like to express my gratitude to Behrang Mahjani at Karolinska Institutet in Stockholm for providing me with the patient data which were used in the project. By providing me with real data and a real problem, I got even more inspired to do my best.

My sincere thanks goes to my examiner Fabien Burki, my coordinator Lena Henriksson, my subject reader Prashant Singh, and my opponent Stella Belin for helping me with the project plan, the final presentation, this report, and providing me with additional feedback on my work.

Last of all, I would like to thank all of my many friends, my colleagues, and my family. There are too many names to name, but thanks to all of their support I managed to stay motivated throughout the five years of my program.

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, *et al.* 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467 [cs] ArXiv: 1603.04467.

Chen J, Shi X. 2019. Sparse Convolutional Denoising Autoencoders for Genotype Imputation. Genes 10: 652. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

Edwards AWF. 2008. G. H. Hardy (1908) and Hardy–Weinberg Equilibrium. Genetics 179: 1143–1150.

Halperin E, Stephan DA. 2009. SNP imputation in association studies. Nature Biotechnology 27: 349–351. Number: 4 Publisher: Nature Publishing Group.

Harikrishnan A, Sethi S, Pandey R. 2020. Handwritten Digit Recognition with Feed-Forward Multi-Layer Perceptron and Convolutional Neural Network Architectures. 2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA). 398–402.

James G, Witten D, Hastie T, Tibshirani R. 2013. An Introduction to Statistical Learning, volume 103 of *Springer Texts in Statistics*. Springer New York, New York, NY.

Kingma DP, Ba J. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs] ArXiv: 1412.6980.

Lawrence S, Giles CL, Tsoi AC. 1997. Lessons in Neural Network Training: Overfitting May be Harder than Expected. In Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI-97. AAAI Press, 540–545.

Li Y, Willer C, Sanna S, Abecasis G. 2009. Genotype Imputation. Annual Review of Genomics and Human Genetics 10: 387–406. Publisher: Annual Reviews.

Mathieu M, Couprie C, LeCun Y. 2016. Deep multi-scale video prediction beyond mean square error. arXiv:1511.05440 [cs, stat] ArXiv: 1511.05440.

Mavromatidis G, Dinas K, Delkos D, Vosnakis C, Mamopoulos A, Rousso D. 2010. Case of prenatally diagnosed non-mosaic trisomy 20 with minor abnormalities. Journal of Obstetrics and Gynaecology Research 36: 866–868. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1447-0756.2010.01188.x.

Michelucci U. 2018. Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks. Apress, Berkeley, CA.

Murphy KP. 2012. Machine Learning: A Probabilistic Perspective. MIT Press. Google-Books-ID: NZP6AQAAQBAJ.

Pu Y, Gan Z, Henao R, Yuan X, Li C, Stevens A, Carin L. 2016. Variational Autoencoder for Deep Learning of Images, Labels and Captions. Lee DD, Sugiyama M, Luxburg UV, Guyon I, Garnett R, editors, Advances in Neural Information Processing Systems 29, Curran Associates, Inc., 2352–2360.

Purcell S, Neale B, Todd-Brown K, Thomas L, Ferreira M, Bender D, Maller J, Sklar P, de Bakker P, Daly M, Sham P. 2007. PLINK: A Tool Set for Whole-Genome Association and Population-Based Linkage Analyses. American Journal of Human Genetics 81: 559–575.

Rumelhart DE, Hinton GE, Williams RJ. 1986. Learning representations by back-propagating errors. Nature 323: 533–536. Number: 6088 Publisher: Nature Publishing Group.

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting 30.

Sun YV, Kardia SLR. 2008. Imputing missing genotypic data of single-nucleotide polymorphisms using neural networks. European Journal of Human Genetics 16: 487–495. Number: 4 Publisher: Nature Publishing Group.

The 1000 Genomes Project Consortium. 2015. A global reference for human genetic variation. Nature 526: 68–74.

Twyman RM. 2009. Single-Nucleotide Polymorphism (SNP) Analysis. Squire LR, editor, Encyclopedia of Neuroscience, Academic Press, Oxford, 871–875.

Venter JC, Adams MD, Myers EW, Li PW, Mural RJ, Sutton GG, Smith HO, Yandell M, Evans CA, Holt RA, Gocayne JD, Amanatides P, Ballew RM, Huson DH, Wortman JR, Zhang Q, Kodira CD, Zheng XH, Chen L, Skupski M, *et al.* 2001. The sequence of the human genome. Science (New York, N.Y.) 291: 1304–1351.