Clustered Autoencoder Imputation

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Mathematics

in the

College of Graduate Studies

University of Idaho

by

Daniel J. Furman

May 2020

Authorization to Submit Thesis

This thesis of Daniel J. Furman, submitted for the degree of Master of Science with a Major in Mathematics and titled "Clustered Autoencoder Imputation," has been reviewed in final form. Permission, as indicated by the signatures and dates below is now granted to submit final copies for the College of Graduate Studies for approval.

Major Professor:	Fuchang Gao, Ph.D.	Date
Committee Members:	Stephen Krone, Ph.D.	Date
	Linh Nguyen, Ph.D.	Date
Department Chair:	Christopher Williams, Ph.D.	Date

Abstract

Many datasets have missing entries. Since downstream tasks often require full datasets with little noise, accurately imputing the missing data is quite valuable. Autoencoders have proven themselves as effective data imputers. However, while they exploit high order dependencies between the columns of a dataset, autoencoders typically treat each row independently. This produces two problems. First, imputation accuracy is suboptimal because not all of the data is used effectively. Second, downstream classification tasks suffer since rows belonging to different classes get treated the same. Presented in this thesis is CLAIM (CLustered Autoencoder IMputation), an algorithm that adapts existing autoencoder networks in a way that directly addresses these issues. CLAIM first separates rows into clusters based on similarity. Then, in the encoder, it applies different, loosely connected, learned linear transformations to each cluster. Results show that this method improves accuracy with typical autoencoder imputation strategies on large enough datasets. Also presented is a CLAIM-specific iterative clustering algorithm, which allows CLAIM to improve initial cluster assignments as needed.

Acknowledgements

I would like to thank my major professor Frank Gao for the research topic and his input throughout this project. I would also like to thank my committee members, Steve Krone, and Linh Nguyen for their helpful critiques.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Throughout this thesis, the word "dataset" will be used to mean a 2-d matrix of data. Each row corresponds to an entity (individual) and each column to a feature (attribute).

1.1 Motivation

In practice, data matrices with missing entries are prevalent [1]. Data can be missing for many reasons. Some common ones are: missing responses to survey questions, people forgetting or unable to take certain measurements, and machine error. An example of unavoidable machine error that results in missing values is data generated by single-cell RNA-sequencing (scRNA-seq). scRNA-seq is a state-of-the-art technology for measuring gene expression levels. While it can capture gene expression at the individual cell level [2] scRNA-seq suffers from a high rate of dropout [3]. Dropout is where a gene is read to have count zero due to a technical error with the procedure. While some of the zeros are a result of a true gene expression of zero, most are due to limitations in the sequencing technology itself and cannot be distinguished from biological zeros [4]. Because of this, zeros in the data should be treated as missing values.

Missing data is an obstacle for many downstream tasks. It is especially important to deal with missing data in a way that preserves its quality in the area of machine learning [5]. Because of this, many advanced imputation algorithms have been developed. One of the more recent methods called autoencoder imputation (detailed in 2) is the focus of this project. Autoencoders show strong performances in imputing a wide range of datasets, including electronic health records [6], scRNA-seq data [4][7][8][9], and traffic data [10]. Autoencoders exploit non-linear relationships between features to predict what the missing values should be. However, they do not inherently utilize relationships between the rows of the data. This thesis presents a novel method for allowing the autoencoder to exploit the similarity between rows.

1.2 Proposed Method

Consider a dataset where the rows represent different people, and the columns are various body dimensions. Suppose we know the hip circumference of the person, but waist circumference is missing. It is known that males have a larger waist to hip ratio than females on average [11] [12]. So if we knew the individual was male we would predict a larger waist size than if we knew they were female. Without knowing their sex, we would predict somewhere in-between those two values. In this example, we know that the same relationship between features does not necessarily hold when considering males as opposed to considering females. This intuition tells us that imbuing the autoencoder with knowledge of which group (cluster) each row belongs to should improve performance. It also tells us that one way to improve performance is to interpret the row differently based on the group to which it belongs. The proposed method is to train a different transformation matrix for each group, transforming the input data differently based on the cluster. On scRNA-seq data, results show that adding the cluster information improves the performance of the autoencoder architectures studied in this thesis, assuming the datasets are not too small.

Chapter 2: Background

2.1 Types of Missing Data

2.1 Missing Completely at Random

A common assumption about missing data, and the one made in this thesis, is that the data is missing completely at random (MCAR). Data is considered to be MCAR if every entry in the dataset is equally likely to be missing.

2.1 Missing at Random

In the case where data is missing at random, data is missing with a probability dependent on other, known features. For example, it may be that brain surgeons are less likely than bartenders to tell you how often they drink alcohol.

2.1 Missing Not at Random

Missing not at random is when the probability of a missing entry depends on factors not seen in the known data. It could even be dependent on the missing value itself. This might happen if people with lower incomes tend to not report their hourly wage.

2.2 Overview of Selected General Imputation Methods

2.2 Complete Case

The complete case (CC) method is possibly the most straightforward method for dealing with missing values. It simply removes all rows in the dataset that have missing values. This acts to produce a complete dataset, but removes possibly useful information. CC can only be considered if the fraction of rows with missing data is small.

2.2 Mean Imputation

Mean imputation replaces missing entries with the mean of the known values of the feature they represent. There are at least two problems with this method. First, it reduces the correlation between features. This is because any imputed values in one feature are guaranteed to be independent of known values of other features. This deteriorates the structure of the data, which makes extracting useful information more difficult. Second, decision-tree-based classification algorithms operate by splitting the data based on the order of the numerical values in each feature. In this case, mean imputation serves to keep errors from occurring when running the algorithm, but offers it no way to distinguish between the imputed values.

2.2 Multiple Imputation

Multiple imputation (MI) [13] improves imputation methods that produce imputations stochastically. MI is performed by producing m imputed datasets from a stochastic imputation method. The final imputation is the mean of the m imputed datasets. One advantage of MI is that the standard error of the imputation can also be calculated for each imputed value. Analysis on MCAR data has shown that MI with a Markov chain Monte Carlo (MCMC) imputation method that assumes multivariate normality [14] is more effective than either complete case or mean imputation [15]. However, it takes more time and is harder to implement.

2.2 K Nearest Neighbor

The k-nearest neighbor (KNN) imputation [16] algorithm takes each row of the dataset and finds the k most similar rows, based on known values. The common metric to determine similarity is the Euclidean distance. A weighted average is taken over non-missing entries in these k rows and is used to replace each missing value in the original row. The average is weighted based on the similarity score. There are two main drawbacks to KNN. The first is that on large datasets it can be slow. Secondly, it has been shown to distort the original structure of the data [17].

2.3 Artificial Neural Networks

2.3 Definition

Artificial neural networks (ANNs) are multivariate composite functions with trainable parameters. The vanilla ANN is the fully connected neural network (FCNN) (see figure 2.1 for a visual example of an FCNN). An FCNN can be seen as a sequence of layers. Each layer, except the input layer, is the result of a linear transformation acting on the previous layer, followed by a shift, and then run through a non-linear function a, called an activation function. The layers in-between the input layer and the output layer are called hidden layers. A typical fully connected neural network with $n-1$ hidden layers can be written recursively as a nested set of functions:

Let

$$n \in \{2, 3, \dots\},$$

$$m = (m_0, m_1, \dots, m_n) \in \mathbb{N}^{n+1},$$

$$W = [W_1, W_2, \dots, W_n],$$

and

$$b = [\vec{b_1}, \vec{b_2}, \dots, \vec{b_n}],$$

where

$$W_k = \begin{bmatrix} w_{k11} & w_{k12} & \cdots & w_{k1m_k} \\ w_{k21} & w_{k22} & \cdots & w_{k2m_k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{km_{k-1}1} & w_{km_{k-1}2} & \cdots & w_{km_{k-1}m_k} \end{bmatrix} \in M_{m_{k-1} \times m_k}(\mathbb{R}), \text{ and } \vec{b_k} = (b_{k1}, b_{k2}, \dots, b_{km_k}) \in \mathbb{R}^{m_k}$$

$$\text{for } 1 \leq k \leq n.$$

Let

$$\vec{x} = (x_1, x_2, \dots, x_{m_0}) \in \mathbb{R}^{m_0},$$

and define

$$\vec{O_0} = \vec{x}.$$

Then let

$$\vec{O_k} = a(\vec{O_{k-1}} W_k + \vec{b_k}) \text{ for } 1 \leq k \leq n-1,$$

and

$$\vec{O_n} = \vec{O_{n-1}} W_n + \vec{b_n}.$$

Then the fully connected neural network $f_{W,b} : \mathbb{R}^{m_0} \to \mathbb{R}^{m_n}$ can be defined by

$$f_{W,b}(\vec{x}) = \vec{O_n}, \tag{2.1}$$

where the weights W and biases b are trainable parameters. The standard method is to initialize b with all zeros, and W with either a normal [18] or uniform distribution [19] centered at zero.

The activation function that has received the most attention is the rectified linear unit (ReLU). It is defined as

$$ReLU(x) = max(x, 0).$$

The simplicity of the *ReLU* function makes it computationally efficient compared to other activation functions like *sigmoid* and *tanh*. Additionally, for positive path signals, *ReLU*'s linearity eliminates the vanishing gradient problem caused by the chain rule acting on many layers with small derivatives [20]. A variant of *ReLU* is *LeakyReLU* [21] which allows for small gradients to be passed through negative signals. It is defined by

$$LeakyReLU(x) = max(x, ax).$$

A typical value for a is 0.01.

2.3 Training

Suppose we wanted to predict how many points a player in the National Basketball Association (NBA) will score in a given season based on their height, weight, and age. To do this with a neural network we would first collect some data $\vec{x_i} = (h_i, w_i, a_i)$ where h_i, w_i, and a_i are respectively the height, weight, and age of the ith player we collected data on. We also record the number of points y_i scored by the ith player in the corresponding season.

To train a neural network to make these predictions we need a way to tell the network how to update its parameters based on the observed data. It is the loss function that does this. Here, a loss function will be defined as any function $L : \mathbb{R}^2 \to \mathbb{R}$ such that $L(y, \hat{y})$ is increasing with respect to $|y - \hat{y}|$, where y is the true value, and $\hat{y}$ is the predicted value. The loss function will serve to indicate to the ANN how far off its output is from the desired output, and in which direction. A typical loss function for regression tasks is the squared difference $L(y, \hat{y}) = (y - \hat{y})^2$. If we have a sample of size N (also called a batch size), we would use the mean squared error:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2.$$

The goal when training the network is to minimize the error with respect to all its parameters (all the elements of W and b in the case of a fully connected ANN) on the entire training set. This can be done by any optimization algorithm, but is typically done by using a variant of stochastic gradient descent (SGD) which requires taking the gradient with respect to the MSE. We will use the backpropagation algorithm to do this. The backpropagation algorithm is an efficient algorithm that employs the chain rule to find this gradient. The steps to train the network are as follows. First, obtain a random sample of data of size N, call it $x = (x_1, x_2, \ldots, x_N)$. Then $\hat{y} = (\hat{y_1}, \hat{y_2}, \ldots, \hat{y_N}) = f_{W,b}(x)$, where f is as defined as in eq. (2.1). So,

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - (f_{W,b}(x_i)))^2.$$

In general, the y_i's may be of arbitrary shape, and the MSE as defined above would take on that same shape. To calculate a gradient, the loss function has to output a single scalar. So, either the sum or mean over all the elements of the MSE is used as the loss (the mean is typically used). This result is differentiable with respect to all the parameters contained in W and b. Let $\lambda > 0$ (called the learning rate), then using vanilla SGD we can update the network parameters by doing

$$p \leftarrow p - \lambda \frac{\partial(MSE)}{\partial p}$$

for each parameter p of the network. The idea behind this parameter update is simple. The gradient is the direction of the fastest increase of the MSE function. To reduce the MSE, we should move all the parameters opposite the direction of the gradient (i.e. in the direction of the fastest decrease). We repeat the steps above using disjoint samples of the data, called batches, to calculate the error and update the parameters. Once all the data has been used once, we say that we have trained the network for one epoch. All the input/output pairs are shuffled in-between epochs. The number of epochs to train for is just one of many hyperparameters for a neural network. Other hyperparameters include the batch size, number of outputs in each layer, and the type of activation functions to use. They are usually set by the practitioner based on their intuition of the problem at hand, but can also be found using an optimization algorithm like grid search.

2.3 Validation

To make sure that what a network learns during training has predictive value, we need to perform validation. This consists of randomly splitting the collected data into two sets, the training set, and the validation set. A typical split ratio is to dedicate 30% of the data to validation and 70% to training. We train the network only using the training set. Then, every so often we send the validation data through the network, but do not perform an optimization step. Instead, we just calculate and record the error. If the error is decreasing on this validation set, we can be confident that the network is generalizing.

Sometimes half of the validation set is separated into what is known as a test set. This is useful when we want to tune hyperparameters (e.g. learning rate and layer sizes). We use the validation set to check how well the new hyperparameter configuration is performing, and change the hyperparameters to reduce the error. The error obtained from the validation set is now biased since the hyperparameters were selected to reduce it. However, the test set still provides an unbiased evaluation of the network.

2.3 k-fold Cross-Validation

A downside to the validation methods described above is that not all the data can be used for training. To circumvent this, we can use k-fold cross-validation. This type of validation is implemented by following these steps:

1. Randomly split the data into k equal-sized subsets.
2. Create k different models. From the k subsets, assign a different one to each model to be used as a validation set for that model.
3. Train each model on all the data not in its validation set.
4. Use the average of the validation errors over all models as the total validation error of the k models combined.
5. Use the average output of the k models for predictions.

A widely used, and empirically verified, value for k is 10 [22].

2.3 Early Stopping

Early stopping is a criterion for when to stop training. To perform early stopping we calculate the validation error every so often. If the error is lower than at all previous times we save the new model. Otherwise, we start counting the number of times it has failed to reach the previous lowest error. Once this count reaches a predefined number, called the stopping patience, training is stopped. The previous best model can then be loaded and used for predictions.

2.3 Reducing Overfitting with Dropout

If the network is more complex than the data (this is usually the case) the network will begin to simply memorize the input/output pairs given enough training time. This phenomenon, called overfitting, reduces the predictive power of the network (see figure 2.2). There are several methods to deal with overfitting, but arguably the most popular is dropout [23]. Dropout can be applied to any layer of the network except the last. At every training step dropout randomly zeros out a fraction q (a typical value is 0.5) of the features in that layer (a layer being O_i for some $0 \leq i \leq n-1$ in the above formulation). Without a correction, the signal from this layer would be diminished by approximately a factor of $1-q$. To account for this, we divide the values in the layer by $1-q$. Dropout causes the same input to look different each time it is passed through the network, preventing the network from simply memorizing input/output pairs.

2.3 L2 Regularization

Another method to attenuate overfitting in neural networks is to add L2 regularization. It penalizes the network for having large weights:

$$L_{\text{reg}} = L + \gamma \sum_{i=1}^{n} \sum_{w \in W_i} w^2$$

for some $\gamma > 0$, called the regularization coefficient. This addition to the loss function acts to penalize large weights unless they are necessary. This added constraint reduces the network complexity. If the network is more complex than the data it is handling, then overfitting can occur [24].

2.4 Denoising Autoencoders

An autoencoder is an FCNN where the input is also used as the target output during training. It is standard to make the network completely symmetric, but not required for many use cases. Two pieces make up an autoencoder, the encoder, and the decoder. The encoder section takes in the input data, and (usually) returns a lower-dimensional representation of the input. It is the job of the decoder to take the encoder's representation of the input to reconstruct the input. This encoder-decoder structure is shown in figure 2.3.

Denoising autoencoders [25] are autoencoders that were trained to remove noise from inputs. They are trained by corrupting the input before sending it through the network, and setting the target output to be the original, non-corrupted input.

2.5 Missing Data Imputation With Denoising Autoencoders

There are many forms of noise that could be added to the input when training a denoising autoencoder. The noise used to facilitate data imputation is dropout. By zeroing out some values on the input, we force the network to make predictions on these values by using the values it knows. As with regular dropout, we need to compensate for the reduced signal by dividing by one minus the fraction dropped. Compensating for dropout this way assumes that the autoencoder is one with a bottleneck much smaller than the original input. Only this case is treated here. See 6 for discussion on autoencoder imputation where this assumption does not hold.

Let X be a dataset with m rows and n columns, where each column represents a different feature. Suppose X has missing values. To account for this we need to first replace all missing values with zeros. Thus, actual missing values will behave the same as artificially dropped values. Second, the calculation for the MSE needs to only consider entries with known values. Take a sample of N rows from X, call it $x = [\vec{x_1}, \vec{x_2}, \ldots, \vec{x_N}]^T$. Let $\hat{x} = [\hat{\vec{x_1}}, \hat{\vec{x_2}}, \ldots, \hat{\vec{x_N}}]^T$ be the version of x with dropout applied. We take the MSE for the sample to be the MSE calculated only on the known values in the sample:

$$MSE_{\text{imputation}} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{n}(x_{ij} - f\left(\hat{\vec{x_i}}\right)_j)^2 * \mathbb{1}(x_{ij} \text{ is not missing})}{\sum_{i=1}^{N}\sum_{j=1}^{n}\mathbb{1}(x_{ij} \text{ is not missing})},$$

where

$$\mathbb{1}(\text{argument}) = \begin{cases} 1 & \text{argument holds} \\ 0 & \text{argument does not hold} \end{cases}.$$

The encoder reduces the dimensionality by finding relationships between the features. Whenever features are found together the encoder learns about the nature of their interdependence. Thus, a missing data point can be made up for in the encoded representation of the input by other, non-missing features, that it depends on. The decoder then uses these encoded features to reconstruct the input, with all previously-missing data imputed.

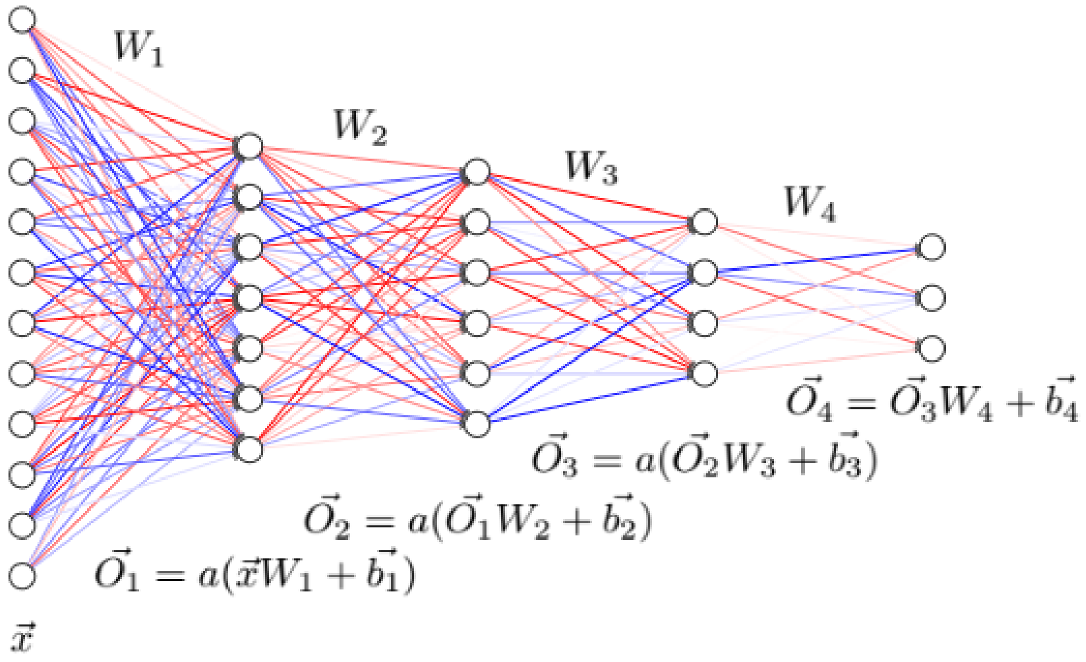
W_1
W_2
W_3
W_4
$\vec{O_4} = \vec{O_3}W_4 + \vec{b_4}$
$\vec{O_3} = a(\vec{O_2}W_3 + \vec{b_3})$
$\vec{O_2} = a(\vec{O_1}W_2 + \vec{b_2})$
$\vec{O_1} = a(\vec{x}W_1 + \vec{b_1})$
$\vec{x}$

Figure 2.1: A typical FCNN architecture [26].

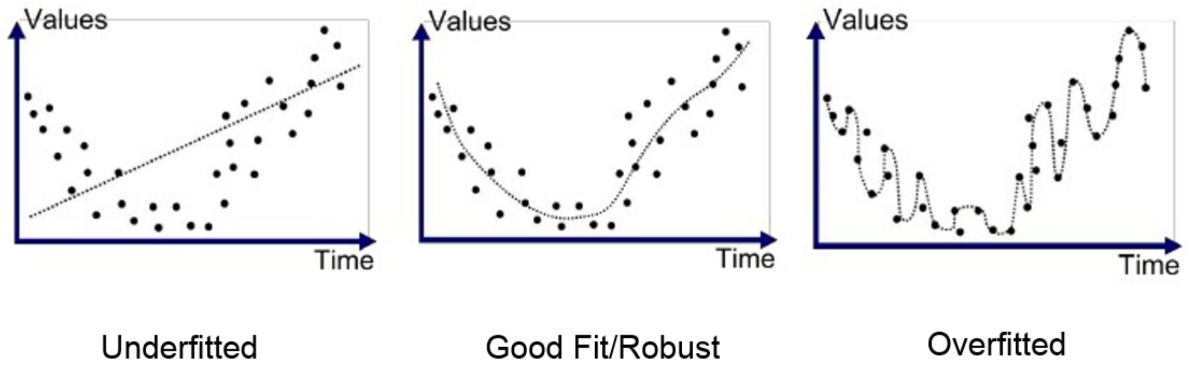

Figure 2.2: Example of overfitting [27].

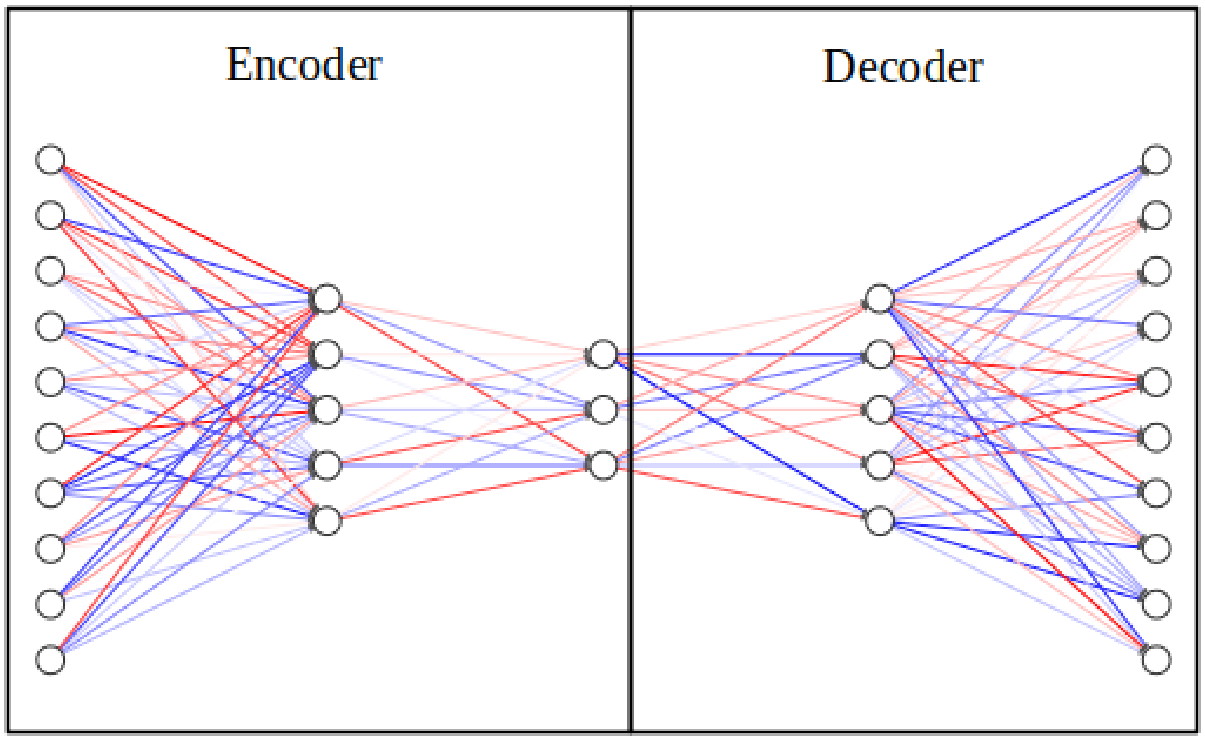

Figure 2.3: A typical autoencoder structure [26].

Chapter 3: Related Work

3.1 Overview of Imputation methods on scRNA-seq data

scRNA-seq data is very sparse, with as high as 90% of the data missing. Additionally, these datasets are larger than many imputation algorithms can handle. As a result, several methods have been developed specifically for imputing scRNA-seq data. Some of the most prominent are mentioned here. scImpute [28] and MAGIC [29] are similar in that they donate known values from cells to similar cells that have missing entries. They are variants for the KNN imputation algorithm outlined in 2.2. SAVER [30] instead uses the similarity between genes. DCA (deep count autoencoder network) [7] and scVI (single-cell variational inference) [8] are autoencoder methods that treat cells independently.

3.1 MAGIC

The scRNA-seq dataset used in this thesis was created via the MAGIC [29] algorithm. So the algorithm is described in greater detail in this subsection. MAGIC uses Euclidean distance to estimate the similarity between cells. If there are m cells, this produces a matrix of size $m \times m$. An affinity matrix A is computed by

$$A(i,j) = e^{-\left(\frac{Dist(i,j)}{\sigma}\right)^2},$$

with σ chosen based on the approximate number of cells wanted in each cluster. Then, a Markov transition matrix M is calculated by normalizing the rows of A to have unit sum:

$$M(i,j) = \frac{A(i,j)}{\sum_k A(i,k)}.$$

Then $M^t(i,j)$ represents the probability of cell i transitioning to cell j after t steps via a random walk. The imputed dataset is the weighted average (weighted by M^t) of the cells related to each cell. If X is the original dataset, then the imputed version of X is calculated by

$$X_{imputed} = Rescale(M^t X)$$

where the *Rescale* function accounts for the fact that multiplying by M^t spreads out the values along the gene dimension. This function brings the top gene expressions to similar levels as before imputation.

As shown in their paper, small values of t result in a sensitivity to noise, while large values result in a loss of information. They offer an algorithm that searches for the optimal value of t.

3.1 LATE

This project was inspired by the "LATE (Learning with AuToEncoder) combined" [4] method. The LATE paper claims superiority over all the imputation methods mentioned above on MSE, recovery of nonlinear gene-gene relationships, and the correlation between imputed and ground truth genes. LATE is a simple autoencoder with ReLU activation, and dropout for regularization.

While LATE is not new on its own, LATE combined appears to be the first autoencoder method which utilizes cell-cell relationships as well as gene-gene relationships. They do this by using the autoencoder to first train using genes as features, and cells as samples. This model is used to obtain imputed matrix $X^{(g)}$. The steps above are then repeated, but this time using cells as features, and genes as samples, to obtain a different imputed matrix $X^{(c)}$. For each matrix and each gene j calculate its MSE against the known values:

$$MSE_j^{(g)} = \frac{\sum_i (X_{ij} - X_{ij}^{(g)})^2 * \mathbb{1}(X_{ij} \text{ is not missing})}{\sum_i \mathbb{1}(X_{ij} \text{ is not missing})},$$

$$MSE_j^{(c)} = \frac{\sum_i (X_{ij} - X_{ij}^{(c)})^2 * \mathbb{1}(X_{ij} \text{ is not missing})}{\sum_i \mathbb{1}(X_{ij} \text{ is not missing})},$$

where X is the input matrix. Then construct $X_{imputed}$ by taking $X_j^{(g)}$ if $MSE_j^{(g)} \leq MSE_j^{(c)}$, and $X_j^{(c)}$ otherwise, for each gene j. While they offer no theoretical basis for why this should reduce the MSE on imputed values against the ground truth, they imply that it does. The two main failures of LATE combined are:

1. It lacks a theoretical basis.
2. It is computationally prohibitive on datasets with too many cells.

They also present an extension to LATE called TRANSLATE (TRANSfer learning with LATE) (as well as its counterpart - TRANSLATE combined). TRANSLATE runs LATE twice, and requires a reference dataset. It first runs LATE on the reference data, then the trained weights are transferred to a new LATE model as initialization to run on the dataset that requires imputing. However, it was deemed unfair to compare TRANSLATE to other methods that did not utilize a reference dataset. So only LATE and LATE combined were considered in their analysis.

3.2 Selected Autoencoder Imputation Variants

3.2 Cascaded Residual Autoencoder

The cascaded residual autoencoder (CRA) [31] is a stacked version of the traditional autoencoder. It takes multiple autoencoders and uses the output of one as the input to the next. At the output of each autoencoder the loss between the known data and imputed data is calculated (as in 2.5). Parameters are updated based on the sum of these losses. This acts to iteratively improve the imputation at each of the stacked autoencoders, while mitigating the vanishing gradient problem. The desired output can either be the output of the very last autoencoder, or an average of the outputs of all the autoencoders in the stack.

3.2 Overcomplete Denoising Autoencoder

An overcomplete autoencoder is an autoencoder which has higher-dimensional hidden layers than the input. Using an overcomplete autoencoder for imputation was introduced in the MIDA (Multiple Imputation Using Deep Denoising Autoencoders) paper [32]. The intuition is that more lateral connections may be required in order to recover data with high accuracy. Indeed, if the dimensionality of the raw data is brought lower than the dimensionality of the manifold it lies on, important information can be lost.

Chapter 4: Methods

4.1 Tools

All of the code for this project is written in Python 3.6. The neural networks are written in Pytorch, a flexible deep learning framework in Python. The Numpy package is used for its scientific computing operations and compatibility with Pytorch. Scikit-learn, a popular machine learning library, is used for its k-means clustering algorithm. Finally, the Pandas library was used for loading datasets into memory, and manipulating them.

Training is done on a laptop with a GeForce GTX 1050 Ti with 4 GB of dedicated memory.

4.2 Datasets

The MAGIC dataset was developed by the authors of the LATE paper. They started with a mouse bone marrow dataset of $16,114$ genes, and $2,576$ cells. The original count data were log-transformed by $\log_{10}(\text{count} + 1)$. This dataset was very sparse, so MAGIC (see 3.2) was used to impute the missing values. This imputed dataset is considered the ground truth. From the imputed dataset 90% of the values were randomly erased, and the result is used as the input data.

The OpenIntro [33] datasets were filtered from the full list of datasets on OpenIntro (found at https://www.openintro.org/data/). The steps for processing this data were, for each dataset:

1. Remove all categorical features.
2. Remove the dataset if it has fewer than five remaining features.
3. Remove the dataset if it has more than 90% missing values.
4. Normalize all remaining features to have values from zero to one by subtracting off the minimum then dividing by the range.

This left 26 datasets out of the original 195.

4.3 Autoencoder

4.3 Architecture

The LATE paper mentions several different architectures they experimented with. However, it does not state which one was used to produce their results. Based on the GitHub repository linked to in their paper, they used 3 hidden layers. The default structure found is 800, 400, and 800 nodes in the first, second, and last hidden layers respectively. They use a dropout rate of 0.8 in the input layer, and 0.5 in all hidden layers. The ReLU activation function is applied to all layers after the input layer, including the output layer.

Introduced in this thesis are some notable improvements on their architecture. To differentiate, the proposed architecture will be referred to as NEW. The differences between LATE and NEW are outlined below:

1. ReLU is not applied to the output layer in NEW. The rationale behind applying ReLU to the output seems to be that scRNA-seq data is non-negative. While ReLU enforces the non-negativity of the data, it also slows down training since no gradient is passed through negative outputs. In NEW, the non-negativity of the data is easily enforced by applying ReLU on the imputed data (post training).

2. The only dropout introduced is in the input layer at a rate of 0.5. Dropout on the input is important because it simulates instances of missing data, when, in fact, the true values are known and can be used for training. Applying dropout on the hidden layers did not show any increase in performance, and slowed training.

3. The bottleneck of the autoencoder is 1664 dimensions for the MAGIC dataset, which is about four times as large as in LATE. Each of the three hidden layers were given the same size as the bottleneck to control memory issues. This larger bottleneck was found to increase accuracy and training speed. In fact, research suggests that increasing hidden layer dimensions past that of the input continues to improve results [32]. This was confirmed in some preliminary tests for this thesis, but introduces other problems detailed in 6. Due to the added complexity that case is not treated here. Instead, on all but the MAGIC data, NEW uses a bottleneck size of one third the input size, rounded up.

4.3 Validation

For validation, let M be a matrix with the same shape as the dataset. Randomly set a validation fraction p of the data in M to 0, and set the rest to 1. Let X be the matrix that needs to be imputed, and let $X^{(M)} = X * M$ where $*$ is the Hadamard product. Take $X^{(o)}$ to be the output of the network given input $X^{(M)}$. The validation error is just calculated on the known values that were masked out of the input:

$$MSE_{val} = \frac{\sum_i \sum_j (X_{ij} - X_{ij}^{(o)})^2 * \mathbb{1}(X_{ij} \text{ is not missing}) * (1 - M_{ij})}{\sum_i \sum_j \mathbb{1}(X_{ij} \text{ is not missing}) * (1 - M_{ij})}.$$

The usual validation method is to withhold samples (rows) from the training for validation. Doing so resulted in low-quality imputation on the rows withheld and does not capture the real performance of the network, which is how well it does at imputing the values it cannot see. The validation method presented here captures that characteristic exactly.

4.3 Training

Suppose the dataset has m rows and n columns. For training, a batch size of $\lfloor m/10 \rfloor + 1$ was used. For the loss function, MSE as defined in 2 for autoencoder imputation was employed. The Adam [34] optimizer (a popular variant of SGD) was used to minimize this loss, with the learning rate defined by:

$$\text{learning rate} = 0.0003\sqrt{\frac{\text{batch size}}{258}}.$$

This learning rate is 0.0003 on the MAGIC dataset. The idea behind this dynamic learning rate is to keep the expected standard error of the parameter update constant when using SGD [35].

On the MAGIC dataset, training was done for 1000 epochs. The best model during training was chosen according to the validation metric above, with a validation fraction of $p = 0.01$. However, the ground truth dataset was used to calculate the final metrics displayed in 5.

On all other datasets, training ran for up to 5000 epochs. A patience of 15, checked every 10 epochs, was used for early stopping. Once again, the best model during training was used for imputation. Because these datasets contained some missing values for which no ground truth was known, the metrics were calculated based only on the known values. A validation fraction of $p = 0.5$ was used.

Also, while ReLU was used on the MAGIC data, it was found that with the smaller datasets, the encoded representation was often entirely zeros due to ReLU and large negative biases. So on these other datasets LeakyReLU with a slope on the negative side of 0.01 was used.

4.4 CLAIM

The CLAIM algorithm is a novel modification to existing autoencoder imputation methods. Suppose the encoder portion of the autoencoder has N fully connected layers with weights matrices $W_1, W_2, \ldots, W_N$, and respective bias vectors $\vec{b_1}, \vec{b_2}, \ldots, \vec{b_N}$. Then CLAIM is implemented as follows:

1. Use a clustering algorithm to cluster the data into k clusters.

2. Choose the weight matrix with fewest entries (to reduce memory consumption and fight overfitting), W_i, and produce k copies of it with the same initialization. Call these copies $W_i^{(1)}, W_i^{(2)}, \ldots, W_i^{(k)}$.

3. Whenever data belonging to cluster j is passed through the network, use weight matrix $W_i^{(j)}$ in place of W_i (see figure 4.1).

4. Suppose W_i is an $m \times n$ matrix. Let $c_1, c_2, \ldots, c_k$ be the fraction of data in clusters 1 through k respectively. During training, a regularization penalty is added to the loss function:

$$MSE_{CLAIM} = MSE_{imputation} + \epsilon \sum_{j=1}^{k} \sum_{a=1}^{m} \sum_{b=1}^{n} \left(\left(W_i^{(j)} \right)_{ab} - \sum_{l=1}^{k} c_l * \left(W_i^{(l)} \right)_{ab} \right)^2$$

   This added term penalizes the weights when they are different from the weighted mean of the weights of each cluster. This fights overfitting by keeping the network from memorizing the output from small clusters. It also allows clusters to "teach" each other things they have learned individually. The value $\epsilon = 10^{-5}$ was found to be a good default value.

Steps 2-4 can be done with the biases as well. In this research, the biases on the swapped layer were removed. Results for all architectures are compared with their CLAIM counterpart. For example, NEW will be compared to its CLAIM version, labeled as $\text{NEW}_{\text{CLAIM}}$.

4.5 Clustering

4.5 Single-Shot Clustering

For clustering, an autoencoder with the architecture outlined in the previous section, is used to deal with the missing values. For the MAGIC dataset, dropout was not used because tests indicated it was not needed. However, for other datasets a dropout rate of 0.5 was used only on the input layer, because it proved necessary to produce high-quality clusters. The rows are clustered via the following procedure:

1. Train the autoencoder using rows as samples. In this research, training was done for up to 2000 epochs with an early stopping patience of 15 checked every 10 epochs.

2. Use the encoder portion of the autoencoder to encode each row down to dimension d. This produces an $m \times d$ matrix.

3. (optional) Use principal component analysis (PCA) with k principal components on the $m \times d$ matrix to produce an $m \times k$ matrix ($k = 5$ was used in this project). This is a recommended step to improve the clustering performance of the k-means algorithm [36].

4. Cluster all the rows using the k-means algorithm.

Tests were done on the MAGIC dataset with and without PCA (step 3). Omitting PCA delivered better results on this dataset, so it was not used when clustering the other datasets. No biases were used for the autoencoder for clustering the OpenIntro data. This was done to fight a tendency for the autoencoder to simply return a zeroth degree regression for each feature. It would sometimes zero out all contributions from the inputs (possibly due to a bad initialization), and train the bias on the output. Due to the large number of features in the MAGIC data, this measure was not found to be necessary.

4.5 Iterative Clustering

The k-means algorithm uses Euclidean distance to determine clusters. However, clustering the encoded representation of the input by Euclidean distance may not result in clustering those inputs that should be treated with the same linear transformation. In fact, the results demonstrate that the above single-shot clustering does not consistently produce better results when using CLAIM than not clustering at all. To fix this, a CLAIM-specific iterative clustering algorithm is presented below.

1. Cluster the inputs into k clusters using any clustering algorithm. This initializes the cluster assignments.

2. Remove 50% of the known entries in the dataset and save those for cluster evaluation. Call this the cluster validation set.

3. Train the autoencoder using CLAIM for some time with the current cluster assignments (10 epochs is used in this thesis).

4. Run all the data through as if it was all in cluster one. Record the cluster validation loss. Then as if it all belonged to cluster two. Record the cluster validation loss. Do this all the way until cluster k.

5. Reassign each input to the cluster in which it produced the lowest loss on the cluster validation set.

6. Repeat steps 3-5 until a stopping criterion is reached. In this thesis, the number of times the cluster validation loss was not lower than the previous lowest was recorded. Also recorded was the number of times the fraction of inputs that received a different cluster assignment dropped below 0.005. If the sum of these two numbers was greater than five, the cluster search got terminated.

7. Add the cluster validation set to the training set, and train until convergence. Record the final imputed dataset.

In practice, steps 2-7 are repeated, but with the other 50% of known data for cluster validation. The cluster assignments found from the first clustering can be used to initialize the second one. Finally, take the element-wise average of the two imputed datasets to produce the final imputed dataset.

Only one 50% split was used to produce the results in 5. Otherwise, this method would have an unfair advantage compared to the others that were only run once. Though it may be time-consuming, one could perform multiple imputation with this method. Simply repeat the entire procedure n times using a different half of the data each time for validation. This should produce results with accuracy increasing with n. Additionally, one could calculate the standard error for each imputed entry.

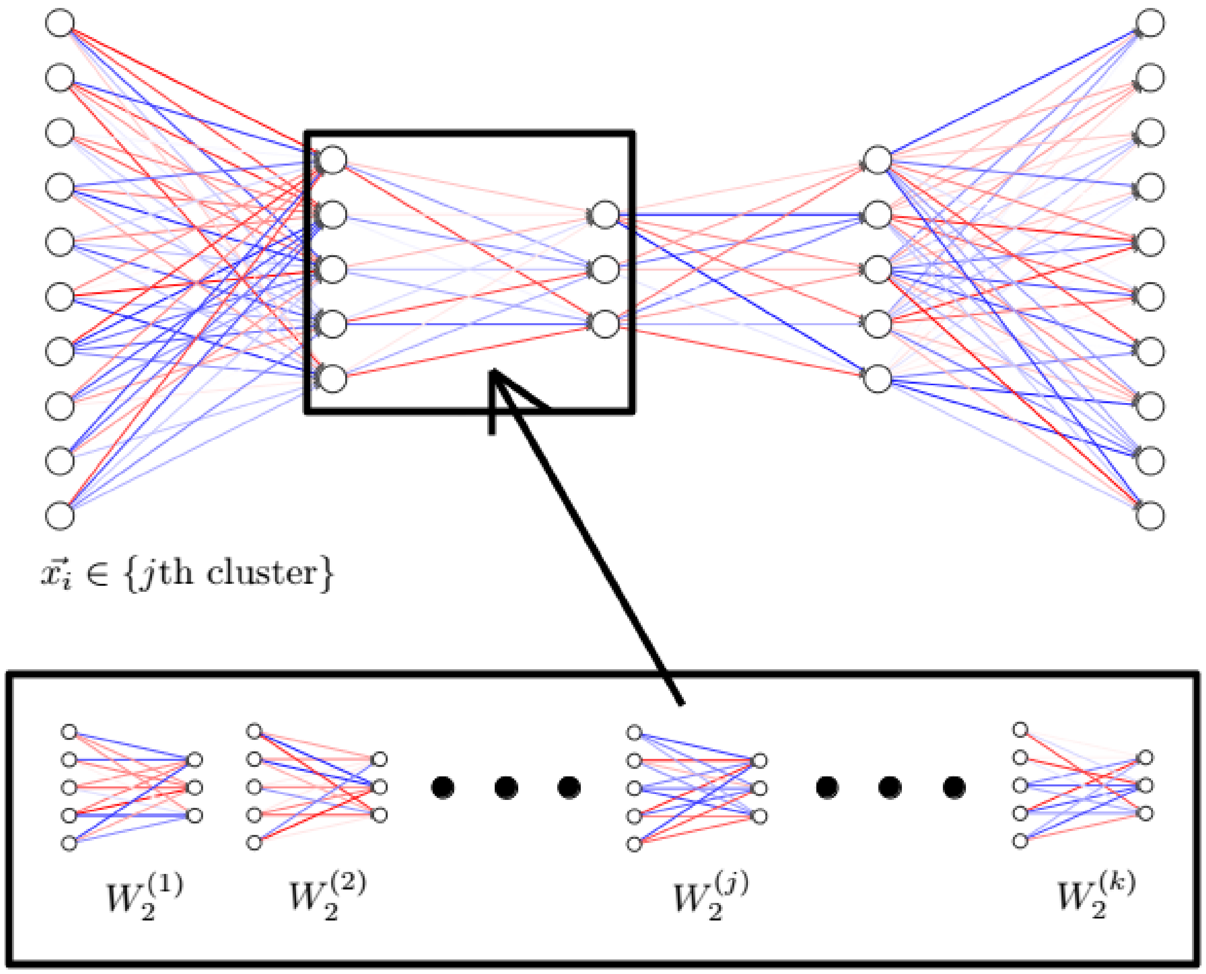

Figure 4.1: CLAIM inserts a different fully connected layer into the encoder for each cluster [26].

Chapter 5: Results

5.1 Single-Shot Clustering

There are three metrics used in this thesis to determine the performance of each method. The first is the MSE between the imputed and ground truth values. Second and third are the median and mean of the correlations between each column of the imputed dataset, and the corresponding column in the ground truth dataset. Because autoencoder imputation produces predictions for each entry (including non-missing entries), previously known values will be altered. These altered values could be reset to the known values, but are not. This is because many downstream tasks rely on the order of the data, and not on their exact values.

On the MAGIC data, each method was run nine times, and their metrics recorded. The means and standard errors of these runs are shown in table 5.1. Three different versions of CLAIM are compared to the original method without CLAIM. For this data, 10 clusters were used. The versions are:

1. using PCA before clustering
2. using PCA before clustering, and waiting 100 epochs to apply the regularization
3. not using PCA before clustering, and waiting 100 epochs to apply the regularization

We see that waiting to apply the regularization improves performance by a significant margin. It also appears that PCA removes some important distinctions between clusters on this dataset, resulting in a worse performance. All three metrics tended to agree with each other about which method performed the best, though not in every case. The method that performed the best in both the LATE and NEW architectures was CLAIM, without PCA, and with delayed regularization. This was true across all three metrics.

For the OpenIntro data, all clustering was done without PCA, and regularization was only applied after 100 epochs. Additionally, the data were grouped into just two clusters. Each of the 26 datasets were imputed 10 times. Each time, a random 50% of the entries were removed before the data were sent through the imputation process and used for validation. The metrics for the original version were subtracted from the CLAIM version for each run. The mean of these differences was recorded. Figure 5.1 shows the histogram produced by the mean difference for each of the 26 datasets. Based on this data, there is no evidence that CLAIM can be used to improve imputation on a wide variety of datasets. Many of the imputations were harmed by adding CLAIM, likely due to overfitting caused by separating the clusters. The results for OpenIntro used a regularization coefficient of 0.001. Other coefficients were tested but did not give any noticeable improvement. It was found that CLAIM improved imputation on the larger datasets more than smaller ones.

It was of interest to further inspect a dataset containing data of people's body dimensions. Also contained in the dataset, but not used as input, was the sex of the person. Because men and women on average have different body dimensions, it would make sense to cluster the data based on sex. In figure 5.2, two different versions of CLAIM are compared to not using CLAIM. CLAIM with single-shot clustering significantly lowered the MSE, but was not able to increase either the mean or median correlations. However, when clusters were separated manually by if the person was male or female, the results were superior to the other two methods. This suggests that given the right clustering and dataset, the CLAIM algorithm does in fact improve autoencoder imputation. The problem then lies in developing the right clustering algorithm. This is why the iterative clustering algorithm for CLAIM was developed.

5.2 Iterative Clustering

Also shown in figure 5.2 are the results from iterative clustering on the body dimensions data. We can see that iterative clustering significantly improves the performance on this data. Using the other half of the data on a second pass of iterative clustering increases the accuracy further. This result shows that the k-means algorithm does not accurately cluster data that have similar relationships between variables. It is likely that the clustering is based more on the magnitude of the signal, which is less important for imputation. We see that CLAIM improves performance as long as it receives high-quality cluster assignments.

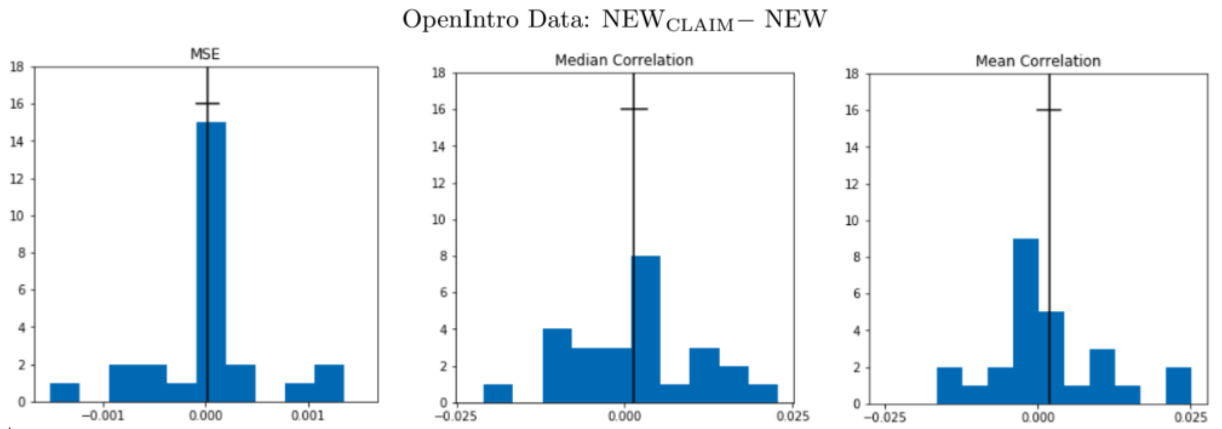

Figure 5.1: Difference between metrics using $\text{NEW}_{\text{CLAIM}}$ and NEW. Vertical lines designate means. Horizontal lines represent the standard error. Lower MSE is better, and higher correlations are better.

Table 5.1: Compares performance on three metrics between the original version, compared to CLAIM used three ways: (1) Using PCA before clustering. (2) Using PCA before clustering, and delaying regularization for 100 epochs. (3) Not using PCA before clustering, while delaying regularization for 100 epochs.

Dataset - MAGIC			
	MSE	Median Gene Correlation	Mean Gene Correlation
LATE	$(7.6 \pm 0.1) * 10^{-4}$	0.98829 ± 0.00006	0.9674 ± 0.0003
LATE$_{\text{CLAIM}}$ (PCA)	$(7.1 \pm 0.1) * 10^{-4}$	0.98816 ± 0.00004	0.9675 ± 0.0005
LATE$_{\text{CLAIM}}$ (PCA & delayed reg)	$(6.96 \pm 0.08) * 10^{-4}$	0.98833 ± 0.00003	0.9668 ± 0.0003
LATE$_{\text{CLAIM}}$ (no PCA & delayed reg)	$\mathbf{(6.8 \pm 0.1) * 10^{-4}}$	$\mathbf{0.98857 \pm 0.00003}$	$\mathbf{0.9679 \pm 0.0005}$
NEW	$(1.655 \pm 0.009) * 10^{-4}$	0.99279 ± 0.00003	0.98818 ± 0.00006
NEW$_{\text{CLAIM}}$ (PCA)	$(1.64 \pm 0.01) * 10^{-4}$	0.99287 ± 0.00005	0.98849 ± 0.00009
NEW$_{\text{CLAIM}}$ (PCA & delayed reg)	$(1.577 \pm 0.007) * 10^{-4}$	0.99325 ± 0.00002	0.98890 ± 0.00003
NEW$_{\text{CLAIM}}$ (no PCA & delayed reg)	$\mathbf{(1.548 \pm 0.009) * 10^{-4}}$	$\mathbf{0.99340 \pm 0.00002}$	$\mathbf{0.98911 \pm 0.00002}$

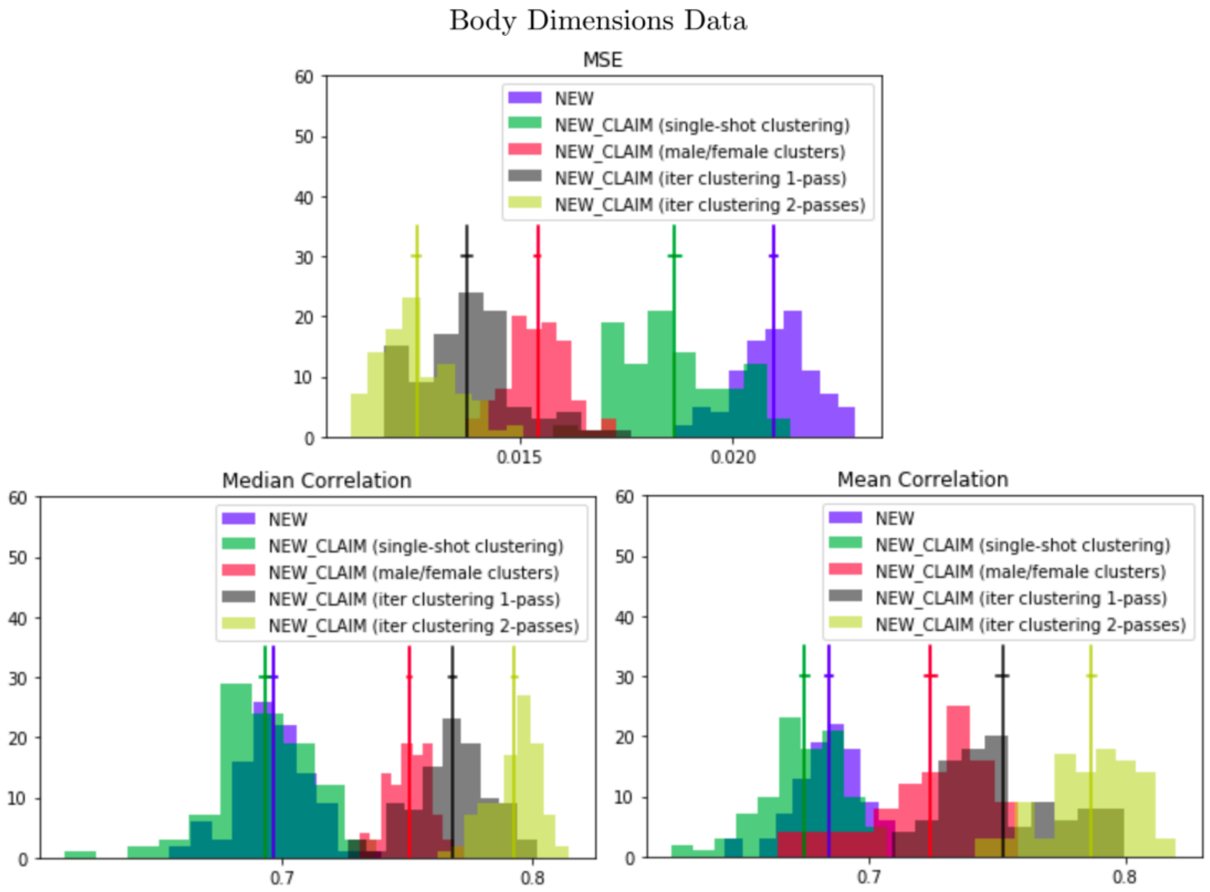

Figure 5.2: Distributions of the metrics from three methods: without CLAIM, with CLAIM and automatic clustering, with CLAIM and manual clusters, with CLAIM and one pass of iterative clustering, with CLAIM and two passes of iterative clustering. 100 runs were executed for each method to obtain these histograms. Vertical lines represent means, horizontal lines represent standard error.

Chapter 6: Future Work

The regularization scheme that was found to be most effective was not intuitive. It is not clear why waiting for 100 epochs before applying regularization was helpful. One explanation is that it just needed more freedom, but reducing the regularization coefficient produced a worse performance. In contrast to the observation here, some evidence has been found that applying regularization early, and then removing it later, is better for classification tasks [37]. It could be that the 1000 epochs used in this thesis are too few, and different results could surface given enough training. Additionally, regularization in CLAIM is different from standard regularization, where the weights decay towards zero. Future research into this phenomenon is warranted.

Better regularization strategies may exist. One idea would be to swap some cluster assignments randomly during training. Also, the regularization technique used to tie clusters may be improved by allowing for parameters to be off by a certain factor. This could be done by instead encouraging cosine similarity between the parameters of different clusters.

In this thesis, CLAIM was used to replace a single layer based on the cluster assignment of the input data. No testing was done to see if replacing multiple layers would produce better results. If memory constraints can be avoided, a group of loosely tied autoencoders could be employed. In this case, an entire autoencoder would be dedicated to each cluster. Also, CLAIM can be adapted to any neural network where the inputs can be clustered. This means that it may find use in many other architectures and fields, not just autoencoders for imputation.

There has been some evidence that using an autoencoder which has larger hidden layers than the input is better for imputation [32]. This is called an overcomplete autoencoder. These findings were confirmed during the research that went into this thesis. However, it was also discovered that, in this case, dropout cannot be applied as normal. Typically when a fraction p of the nodes are dropped, the rest of the inputs are divided by $1 - p$ to compensate for the reduced signal strength in the next layer. With overcomplete autoencoders, dividing by $1 - p$ caused the validation accuracy to diverge. This seems to indicate that the signal from each input becomes mostly separated from other inputs. There should still be some interaction between signals to get optimal accuracy, so there needs to be compensation for the lost signal caused by dropout. One solution may be to randomly choose the dropout fraction p at each training iteration. This way the network would be forced to learn how to deal with different signal strengths on its own. It may also be possible to 'tell' the network the dropout rate in some useful way, allowing it to dynamically adapt.

Categorical data was not used in this thesis. In future work it should be utilized. Preliminary tests show that one-hotting each categorical variable and using the resulting full matrix for imputation hurts the performance on the ordinal data. It seems that this very sparse input acts to obscure the existing structure in the data. It may be better to embed each categorical variable into just a few dimensions (maybe just one) for the input, with the target being the one-hotted version. If the accuracy is still worse on the ordinal data, one solution is to make the target be only the ordinal data. A separate model could be used on the categorical data. Preliminary tests indicated this last option does improve performance on ordinals, compared to not using categorical data at all.

Finally, it remains to be seen how well CLAIM performs on data missing at random and not at random, as only data missing not at random was treated here.

Chapter 7: Discussion and Conclusion

Because datasets that contain missing data are prevalent, highly accurate imputation algorithms are very valuable. The literature shows that autoencoder imputation is a particularly powerful algorithm. However, autoencoders do not innately exploit the dependencies between rows, and the dependencies between columns simultaneously. Though the authors of the LATE paper attempt to do this, their method lacks a theoretical basis.

Presented in this thesis was CLAIM, a novel method for utilizing row similarities in addition to column dependencies with autoencoders. This was accomplished through clustering the rows, and applying a different fully-connected layer in the encoder for each cluster. Without some regularization the network could overfit to small clusters. Additionally, clusters can benefit from information contained in other clusters. Thus it is natural to loosely tie the cluster transformation matrices together through regularization.

Another contribution was the iterative clustering algorithm that allows CLAIM to dynamically determine the best cluster assignments. This algorithm also reduces the problem of overfitting small clusters since the iterative clustering can choose how many clusters to keep based on performance.

Through the CLAIM algorithm, we see that autoencoders can indeed utilize relationships between the rows of a dataset and not just the columns. Furthermore, CLAIM can be used to improve any autoencoder imputation design and is backed both theoretically and empirically. It only requires that the dataset not be too small, so that the clustering does not introduce too much bias.

References

[1] Andrew Gelman and Jennifer Hill. *Missing-data imputation*, page 529544. Analytical Methods for Social Research. Cambridge University Press, 2006.

[2] Ashraful Haque, Jessica Engel, Sarah A. Teichmann, and Tapio Lnnberg. A practical guide to single-cell RNA-sequencing for biomedical research and clinical applications. *Genome Med*, 9(75), 2017.

[3] Peter V. Kharchenko, Lev Silberstein, and David T. Scadden. Bayesian approach to single-cell differential expression analysis. *Nat Methods*, 11:740–742, 2014.

[4] Md. Bahadur Badsha, Rui Li, Boxiang Liu, Yang I. Li, Min Xian, Nicholas E. Banovich, and Audrey Qiuyan Fu. Imputation of single-cell gene expression with an autoencoder neural network. *Nat Methods*, 8:78–94, 2020.

[5] Bhavisha Suthar, Hemant Patel, Ankur Goswami, and M. tech. Scholar. A survey: Classification of imputation methods in data mining. 2012.

[6] BRETT BEAULIEU-JONES and Jason Moore. Missing data imputation in the electronic health record using deeply learned autoencoders. volume 22, pages 207–218, 02 2017.

[7] Gkcen Eraslan, Lukas M. Simon, Maria Mircea, Nikola S. Mueller, and Fabian J. Theis. Single-cell RNA-seq denoising using a deep count autoencoder. *Nat Commun*, 10:390, 2019.

[8] Romain Lopez, Jeffrey Regie, Michael B. Cole, Michael I. Jordan, and Nir Yosef. Deep generative modeling for single-cell transcriptomics. *Nat Methods*, 15:1053–1058, 2018.

[9] Divyanshu Talwar, Aanchal Mongia, Debarka Sengupta, and Angshul Majumdar. Autoimpute: Autoencoder based imputation of single-cell rna-seq data. *Scientific Reports*, 8, 12 2018.

[10] Yanjie Duan, Yisheng Lv, Yu-Liang Liu, and Fei-Yue Wang. An efficient realization of deep learning for traffic data imputation. *Transportation Research Part C: Emerging Technologies*, 72:168 – 181, 2016.

[11] Eric B. Rimm, Meir J. Stampfer, Graham A. Colditz, Christopher G. Chute, Lisa B. Litin, and Walter C. Willett. Validity of self-reported waist and hip circumferences in men and women. *Epidemiology*, 1(6):466–473, 1990.

[12] Susan M. Hughes and Gordon G. Gallup. Sex differences in morphological predictors of sexual behavior: Shoulder to hip and waist to hip ratios. *Evolution and Human Behavior*, 24(3):173–178, 2003.

[13] Donald B. Rubin. *Multiple Imputation for Nonresponse in Surveys.* Wiley, 1987.

[14] SAS Institute Inc. Sas/stat users guide, version 8. *Cary*, 1999.

[15] Michikazu Nakai, Ding-Geng Chen, Kunihiro Nishimura, and Yoshihiro Miyamoto. Comparative study of four methods in missing value imputations under missing completely at random mechanism. *Open Journal of Statistics*, 4(1), 2014.

[16] Maria-Carolina Monard. A study of k-nearest neighbour as an imputation method. 10 2002.

[17] Lorenzo Beretta and Alessandro Santaniello. Nearest neighbor imputation algorithms: A critical evaluation. *BMC Medical Informatics and Decision Making*, 16, 07 2016.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[19] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[21] A.L. Maas, A.Y. Hannun, and A.Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the International Conference on Machine Learning*, Atlanta, Georgia, 2013.

[22] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. 14, 03 2001.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[24] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 950–957. Morgan-Kaufmann, 1992.

[25] Pascal Vincent, Hugo Larochelle, Y. Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. pages 1096–1103, 01 2008.

[26] Alexander LeNail. NN-SVG: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747, 2019.

[27] Anup Bhande. What is underfitting and overfitting in machine learning and how to deal with it. 2018.

[28] Wei V. Li and Jingyi J. Li. An accurate and robust imputation method scimpute for single-cell RNA-seq data. *Nat Commun*, 9:997, 2018.

[29] David van Dijk, Roshan Sharma, Juozas Nainys, Kristina Yim, Pooja Kathail, Ambrose J. Carr, Cassandra Burdziak, Kevin R. Moon, Christine L. Chaffer, Diwakar Pattabiraman, Brian Bierie, Linas Mazutis, Guy Wolf, Smita Krishnaswamy, and Dana Pe'er. Recovering gene interactions from single-cell data using data diffusion. *Cell*, 174:716–729.e27, 2018.

[30] Mo Huang, Jingshu Wang, Eduardo Torre, Hannah Dueck, Sydney Shaffer, Roberto Bonasio, John I. Murray, Arjun Raj, Mingyao Li, and Nancy R. Zhang. SAVER: gene expression recovery for single-cell RNA sequencing. *Nat Methods*, 15:539–542, 2018.

[31] Luan Tran, Xiaoming Liu, Jiayu Zhou, and Rong Jin. Missing modalities imputation via cascaded residual autoencoder. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[32] Lovedeep Gondara and Ke Wang. MIDA: Multiple imputation using denoising autoencoders. In *Advances in Knowledge Discovery and Data Mining*, pages 260–272. Springer International Publishing, 2018.

[33] Openintro datasets. Retrieved from https://www.openintro.org/data/.

[34] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[35] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[36] Petros Drineas, Alan Frieze, Ravindran Kannan, S. Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition: Theoretical advances in data clustering (guest editors: Nina mishra and rajeev motwani). *Machine Learning*, 56, 01 2004.

[37] Aditya Sharad Golatkar, Alessandro Achille, and Stefano Soatto. Time matters in regularizing deep networks: Weight decay and data augmentation affect early learning dynamics, matter little near convergence. In H. Wallach, H. Larochelle, A. Beygelzimer, F. 'Alch'e-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 10678–10688. Curran Associates, Inc., 2019.