TALLINN UNIVERSITY OF TECHNOLOGY School of Information Technologies

Serkan Ongan 156395IASM

A NOVEL ALGORITHM FOR TRAINING NEURAL NETWORKS ON INCOMPLETE INPUT DATA SETS

Master's thesis

Supervisor: Eduard Petlenkov

Associate Professor

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Serkan Ongan 156395IASM

UUS ALGORITM TEHISNÄRVIVÕRKUDE TREENIMISEKS PUUDULIKE ANDMETE PEAL

Magistritöö

Juhendaja: Eduard Petlenkov Dotsent

Tallinn 2017

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Serkan Ongan

11.05.2017

Abstract

With the emergence of Big Data, handling incomplete data sets has also become an interest area. This is especially true in the areas of human provided data. One way to deal with such data sets that includes incomplete input data is to create a smaller, complete subset of the input data, but this approach comes with its own disadvantages.

The aim of thesis is to provide a novel algorithm for training neural networks with incomplete input data sets.

The proposed method uses masking of weights in order to partially train the network, depending on the available input. In effect, this enables updating only the weights related to the available input.

The experimental results show that this method could be viable approach when dealing with less than 3-5% missing data.

This thesis is written in English and is 57 pages long, including 7 chapters, 12 figures and 11 tables.

Abstract Uus algoritm tehisnärvivõrkude treenimiseks puudulike andmete peal

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 57 leheküljel, 7 peatükki, 12 joonist, 11 tabelit.

List of abbrevations and terms

GA	Genetic algoritmhs
NaN	Not a number
NN	Neural networks
TLU	Threshold Logic Unit

Table of Contents

Intr	oduction 2
1.1	Problem definition
1.2	Goal
1.3	Methodology 3
Pvtł	ion 4
2.1	Language
2.2	Fundamental libraries
	2.2.1 Numpy
2.3	Pandas
2.4	Neural network libraries
	2.4.1 Tensorflow
	2.4.2 Keras
2.5	Why Python? 6
Neu	ral Networks 7
3.1	Biological model 7
3.2	Networks 8
5.2	3.2.1 Neurons and Threshold Logic Unit
	3.2.2 Activation functions
	3.2.3 Lavers
3.3	Types of neural networks 11
0.0	3.3.1 Networks by structure
	3.3.2 Networks by learning
3.4	Training
	3.4.1 Cost function
	3.4.2 Optimization
	3.4.3 Backpropagation
3.5	Applications
Mise	sing Data 17
<i>A</i> 1	Traditional methods
T . I	
7.1	4.1.1 Imputation
	Intro 1.1 1.2 1.3 Pyth 2.1 2.2 2.3 2.4 2.5 Neu 3.1 3.2 3.3 3.4 3.5 Miss 4.1

	4.2	Other methods	18
		4.2.1 Using neural networks to complete the dataset	18
		4.2.2 Using neural networks based on rough set theory	19
	4.3	Proposed method	20
		4.3.1 Theory	21
5	Proc	of of Concept	23
	5.1	Masking and NaN	23
		5.1.1 NaN	23
		5.1.2 Masking	24
	5.2	Feed forward processing	25
	5.3	Training	26
		5.3.1 Method: update_weights	27
		5.3.2 Method: backpropogation	28
6	Imp	lementations	31
	6.1	Skin Color Dataset	31
	6.2	MNIST Digit Dataset	32
	6.3	Summing Function Dataset	35
	6.4	Averaging Function Dataset	37
7	Con	clusion	38
Re	eferen	ices	41
Aŗ	opend	lices	42

List of Figures

3.1	Structure of biological neuron
3.2	Threshold logic unit
3.3	Neurol network layers
3.4	A recurrent network structure
3.5	Gradient descent on 2-dimensional function
4.1	Basic neural network structure
4.2	Missing data
6.1	Epoch 5 Match Rates32
6.2	Number visualization
6.3	Number visualization with missing data
6.4	Missing data histogram
6.5	Epoch 5 Match Rates

List of Tables

6.1	Skin color dataset NN model structure	31
6.2	Skin color dataset results	32
6.3	MNIST dataset NN model structure	32
6.4	Skin color dataset results	35
6.5	Summing formula dataset NN model structure	35
6.6	Summing function dataset training results with complete data	36
6.7	Summing function dataset training results with missing data	36
6.8	Comparison of Neural Network trained with missing data to the function	36
6.9	Averaging formula dataset NN model structure	37
6.10	Dataset variations	37
6.11	Training results	37

1 Introduction

1.1 Problem definition

Human provided data often comes with its unique data-related issues, since humans tend to provide imperfect data.

This kind of data can be complex and missing columns could be related to structure of the process that produced said data. For example, a dataset provided by a U.S.A based loan company [1] has information about each loan they provided to their customers. The loans are made jointly or by a single person. So for loans made by single person, data columns related to joint applications would be irrelevant, therefore missing.

To give another example, in surveys, respondents might refuse to answer certain questions. Or questions could be irrelevant to them [18] (questions about quality of marriage for unmarried people).

And in some cases, the reason for missing data depends on the dataset itself. (The mechanisms of missing data is discussed further in Chapter 4)

Cases of missing data are often encountered in areas of medical trials, social science studies and surveys, and financial industry.

With the emergence of Big Data, the amount of data we can access is increasing fast, but often the data can be imperfect. There are traditional ways to handle this issue (imputation and marginalisation for example) but they come with their own disadvantages and require extra steps of data manipulation [17].

It is, for this reason, important to be able to train neural networks with imperfect data or varying amount of data.

1.2 Goal

The aim of the thesis is to propose a novel algorithm for training neural networks with imperfect data. It should be noted that even though gross errors and noise are also categorized under **imperfect data** problem, the focus of the thesis is on the varying amount of input and missing data.

The proposed method/algorithm involves partially training neural networks depending on the completeness of the input vectors. This is achieved by training only the weights related to the existing input data with a method of masking the weight matrices.

However, due to the nature of the task, the thesis also includes theoretical and practical information in training and implementation of neural network in Python language, as well a brief look at libraries used or considered.

1.3 Methodology

The metholody I will follow involves laying the fundamentals of the problem with theoretical information. Following that, I will form a proof of concept for the proposed method, which includes implementations using relatively simple datasets.

Finally, I will conclude with the conclusion of those different implementations.

2 Python

This chapter includes a brief introduction to Python language and the libraries used for the implementations provided in the thesis, and why Python was chosen for the task.

2.1 Language

Python is a high-level scripting language and it supports several programming paradigms. This makes the language a very flexible tool for many different application areas. Another advantage of Python language is that it has a very clear, almost pseudocode-like syntax, making it easier to read and understand.

It offers an extensive list of third-party libraries (roughly 100.000) and it is a widely used tool in scientific programming. For my thesis, I have only included information about the libraries I used for the tasks described in the following chapters.

2.2 Fundamental libraries

Below can be found a list of fundamental libraries used in the thesis.

2.2.1 Numpy

Numpy is the fundamental scientific library in Python programming language [2]. It is used by many other scientific libraries. And it's mainly a tool for manipulating and using n-dimensional arrays. In some ways, it can be compared to Matlab, but there are some differences that might be important to notice for Matlab users.

- Basic type in Numpy is a multi-dimensional array and operations on these arrays are always element-wise.
- 0-based index is used.
- Being implemented in Python, it can be used for full-fledged applications easily.
- Arrays are pass-by-references, in contrast to Matlab's arrays working on pass-by-

value semantics.

Also, Numpy's masking features (which will be discussed in following chapters) was very important during the implementation of the proposed method. It allowed masking certain weights of the neural network.

2.3 Pandas

Pandas is an open-source library that offers data analysis tools for the Python programming language [3]. It aims to offer a single tool that bridges data preparations and analysis, mainly as an alternative to domain-specific language like R.

Some important features can be summarized below:

- Reading and writing data between many different formats.
- Reshaping and pivoting data sets.
- Intelligents tools to slice and subset datasets.
- High performance.

Below can be seen a few simple examples for library usage.

2.4 Neural network libraries

Below can be found a list of neural network related libraries that were considered for the thesis. However a more Numpy-based approach and implementation of neural network training were used in the proof of concept.

2.4.1 Tensorflow

Developed by Google's Machine Intelligence research organization, Tensorflow focuses on numerical computations using data flow graphs. Although built mainly for machine learning and deep neural network tasks, the library is suitable for other tasks as well.

It's a performant system that can work across many machines in a cluster and it can use

both CPU and GPU, as well as custom designed ASICs systems. It is used in many Google service and by over 15 teams in Google [5].

Main unit of data in Tensorflow is a tensor, which is a set of values shaped as multidimensional arrays.

2.4.2 Keras

Keras is a high-level API for deep learning. It is built around Theanos and Tensorflow backends. It's possible to choose either backend by configuration. It offers a fast tool to experiment on data or with different neural network structures [6].

Like Tensorflow, it can run CPU or GPU, supporting both convolutional recurrent networks.

For my purposes in the thesis, Keras was only used as a benchmark tool, to offer a starting point for match rates for the proposed solution.

2.5 Why Python?

Python offers a productive development environment with high-class libraries in scientific computing area. It's also open-source, meaning it can be used without licensing issues and the direction of the language and libraries are often decided by the actual user-needs.

However the main reason for choosing Python was the Numpy library and its masking feature. This single-handedly allowed to build the proof of concept with a simple implementation of masking of certain parts of the weights.

I believe this is done mostly thanks to the pass-by-reference nature of arrays in Python, which enables in place (in memory) changes of subsets of arrays without making new arrays. When compared to Matlab's pass-by-value approach to arrays, this was very useful.

Another great advantage is the flexibility the language offers in terms of programming styles and libraries.

3 Neural Networks

Neural networks are computational models used in approximation, classification, control and other similar tasks. They are assemblies of simpler processing elements known as neurons or nodes and the connections those elements make.

Neurons' functionalities are loosely based on how biological neurons works. And the weights representing the connections these neurons make store the network's ability to process information.

3.1 Biological model

As mentioned before, artificial neurons are loosely based on how biological neurons work. For this reason, taking a brief look at the biological model can help us understand their structure better. However, considering the complexity of biological structure of brain and neurons, only a brief look at the subject is included in the section.

An important distinction differing neurons from other cells is their ability to function as signal processing devices [12]. This signal processing happens through negatively and positively charged ions, and neuron membrane's properties.

In Figure 3.1, a general structure of biological neuron can be seen.



Neurons are connected to other neurons through their synapses, and form a network for data processing. Each neuron can connect to many thousands of other neurons through synapses, and they communicate via short-lived electrical signals, impulses, in the voltage of their membrane.

In general, inputs to a neuron are summed up as membrane potential and if this potential is over a certain threshold the neuron is excited. Neurons can be excited or inhibited. This simplified model basically gives us an idea how a Threshold Logic Unit (TLU) works.

A neural network is formed of such simplified versions of biological neurons, are capable of making approximation thanks to their ability to train using given data.

3.2 Networks

3.2.1 Neurons and Threshold Logic Unit

The basic building block of a neural network is a neuron. Taking a look at a TLU can give us a clear understanding of how these basic building blocks work.

A TLU is a neuron that excites when a certain threshold is reached after processing its inputs. This threshold is also known as bias. In general, below is a representation of how TLU works.



Figure 3.2: Threshold logic unit

Following this example, weighted sum of inputs is known as **activation** (eq:3.1). The output of the neuron depends on the **bias** (θ), weights, inputs and the activation function (a step function in this case).

$$a = \sum_{i=1}^{n} (W_i X_i)$$
(3.1)

$$y = F(W, \theta, X, a_f) \tag{3.2}$$

3.2.2 Activation functions

The primary reason for using activation functions is to introduce non-linearity to the network [13]. This enables the network learn non-linear functions. These functions also affect how informations flows through the layers of a network. This means they should be chosen according to the task. The most common functions used are hyperbolic tangent (eq. 3.3) and sigmoid functions (eq. 3.4).

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.3}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

In addition to these two functions, other functions can be used according to the task. For example, **softmax** function is a good choice for tasks similar to digit recognition, where the output size is bigger than 1. Softmax allows creating a vector of real values in the range of (0,1) that add up to 1. In other words, it creates an output of vector of probabilities that add up to 1.

For the proof of concept implementation, mainly linear, sigmoid and softmax functions were used.

3.2.3 Layers

Layers form the next level of basic structure after nodes. Each layer is a collection of nodes. The amount of layers in a network vary largely depending on the problem and network structure. However there are 3 specific and important layer types.

- Input layer: The size of this layer depends on the input data.
- **Output layer:** Likewise, the size of output layer depends on the the task and the output in the dataset. For example, for a digit recognition task, output layer would have 10 nodes, each representing a digit.
- **Hidden layer:** Number of layers and layer sizes in hidden layer depends on the problem, network structure and performance considerations.

Figure 3.3: Neurol network layers



3.3 Types of neural networks

Types of neural network can be varied depending on their structure, node connections and training methods. However a simple way to classify neural networks can be by structure and training.

3.3.1 Networks by structure

Feedforward networks:

The defining characteristic of feedforward networks is that the information moves in only one directions, from one layer to the next, starting at input layer and ending at output layer [14]. There are no connections among the nodes in the same layer of nodes, also there are no cycles or loops in the networkç

A node in a layer connects to all the other nodes in the next layer. This enables to the information to flow forward through the network without any feedback structure.

They are very commonly used for classification tasks, and for the proof of concept network written for this thesis also uses such a network structure.

Feedback networks:

In contrast to the feedforward networks, the flow of data can bi-directional [14] with the introduction of loops and cycles. These networks are also known as recurrent networks, and the feedback mechanism can be internal or external.





3.3.2 Networks by learning

It is also possible to classify neural networks by training methodologies used [14].

Supervised learning:

In supervised training, the network is trained by providing it with input and the matching output for that given input.

Reinforcement learning:

In this kind of learning, after processing of a sequence the network receives a value that defines if the results correct or not. Also, this value may include how correct or wrong the

result is. In a sense, the network receives *punishment* or *reward* [15].

Unsupervised learning:

In this type of training, the network is supposed to discover features of the input population, rather than being provided with set of categories. Only the inputs are given, and the output unit is trained to clusters of patterns in the input population.

3.4 Training

This subsection mainly involves training of feedforward networks with supervised learning, due to their use in the thesis. However a brief section on training in supervised learning method can be found at the end as well.

In **supervised learning**, a neural network gives us a model derived from our dataset, which includes both inputs and their respective outputs. During the training, weights of the neural network are updated according to the error in each iteration of the training.

During the training, the output of the network is compared to the actual output for the given input. This comparison gives us an error function. The objective of the training is to minimize this error function. So in a sense, training a network is a task of optimization of the global error of the neural network.

3.4.1 Cost function

Cost function or the error function is the representation of how well a network performs. And it depends on weights and biases of the network.

$$C = f(W, \theta) \tag{3.5}$$

Although the Mean Square Error is the most common functions used for calculating errors. There are several cost functions used in the neural networks [13].

Mean Square Error (MSE):

For i (ideal output) and a (actual output), MSE can be described as:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (i-a)^2$$
(3.6)

The square of the linear error negates the effect of some errors being negative while others positive. It's usually the most commonly used cost function in neural networks.

Sum of Squares Error:

In contrast to MSE, this function doesn't give the error in percentage. The result is a total of errors, meaning that SSE will get higher for more severe errors.

$$SSE = \frac{1}{2} \sum_{p} (i - a)^2$$
(3.7)

Algorithms like Levenberg Marquardt requires the use of SSE [13, pg 25].

Root Mean Square (RMS):

$$RMS = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (i-a)^2}$$
(3.8)

Although similar to MSE, RMS will always provide a higher error value. However this functions is rarely used in neural networks.

3.4.2 Optimization

During the training, we aim to minimize cost function. This is done through optimization methods. One popular way to do this task is using gradient descent optimization. Gradient descent methods are often used when we want minimize or maximize n-dimensional functions.





A gradient is a vector defined for any differentiable point on a function. It points from this point on function towards the steepest descent. So gradient descent is a means of taking steps in the directions that does most to decrease the given function. Derivatives are used to take small steps to make the cost function smaller.

To give an example, for a cost function C of two variables, making small changes to those variables would result in small changes in C itself.

$$\Delta C(v_1, v_2) \equiv \frac{\delta C}{\delta v_1} \Delta v_1 + \frac{\delta C}{\delta v_2} \Delta v_2$$
(3.9)

$$\Delta v = \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \end{bmatrix}, \quad \nabla C = \begin{bmatrix} \frac{\delta C}{\delta v_1} \\ \frac{\delta C}{\delta v_2} \end{bmatrix}$$
(3.10)

And the amount of steps and direction we need to take to make C smaller can be calculated with the following formula, where η is the learning rate and a positive value.

$$\Delta v = -\eta \nabla C \tag{3.11}$$

However, initial starting point could put the gradient descent near a local minimum, and for this reason it's considered a good idea to train the network with different starting points. And limitations of gradient descent should be considered when gradient descent based methods are used for training.

Backpropagation is a training method that uses gradient descent. Since it requires desired outputs, it's more often a supervised learning tool. During the training neurons send their signals forward, as the information flows through the network, and then the errors are propogated.

The backpropagation can be done in two different approaches: online training and batch training.

In **online training**, gradients are calculated and applied to weights after each training set. In contrast, in the **batch training** approach batches of training data are taken. Gradients are summed for each training set in the batch, and only after that the summed gradient applied to the weights of the network.

Batch training is often the preferable method due performance consideration and difficulty of implementing online training in a multi-threaded manner [13].

3.5 Applications

Neural networks have been used in several areas in classification and regression tasks. Below can be found some areas where they have been used [14].

- Identifying tax fraud cases
- Financial predictions
- Economic forecasts
- Loan approvals and credit scoring
- Consumer spending classification
- Image recognition

4 Missing Data

This chapter mainly deals with several methods to handle missing data in neural network tasks, finally concluding with a section on the proposed method as part of the thesis.

There are three types of mechanisms that lead to missing data [9, pg 13].

- *MCAR Missing Completely at Random:* It should be noted that randomness doesn't mean that pattern of missing data is random, but that the missing data does not depend on data values. In other words, the probability of missing value for a variable doesn't depend on the variable itself.
- *MAR Missing at Random:* MAR is a case of probability of missing data on a certain variable X depending on other variables but not the variable X itself.
- *MNAR Missing not at Random*: In MNAR, missing data in X is related to the value of X.

Depending on the mechanism of missing data, different traditional methods are used to handle missing data cases.

4.1 Traditional methods

4.1.1 Imputation

Imputation methods replaces missing data in the dataset with approximated values calculated by using several statistical methods. These methods draw predictions from the distribution of missing data and rely on methods that can create those predictive models using the observed, available data.

These models can be explicit or implicit models. Some models are listed below [7]:

• Average and median: Average or median of observed values are used to replace missing data.

- Random: Missing values are filled with random values of the missing attribute in available data.
- k nearest neighbors method: This method uses k most similar samples to fill the missing data.
- Neural networks and fuzzy systems

The main disadvantage of this method is the likelihood of creating bias in the data.

4.1.2 Marginalisation

Marginalisation is a reduction method, where missing data is removed from the dataset. Some used methods are listed below:

- List-wise deletion: rows that contain missing data are removed from the dataset.
- Attribute deletion: columns (attributes) that contain missing data are removed.

Main disadvantage of marginalisation methods is the possible loss of crucial data from the dataset.

4.2 Other methods

This section mostly deals with using neural networks with missing data. For this reason, the subsection 4.2.1 is mentioned here, even though it can be classified as imputation.

4.2.1 Using neural networks to complete the dataset

A combination a genetic algorithms and neural networks can be used to complete missing data [10]. In such a model, the error (4.1) can be calculated with representation of the input with unknown and known parts.

$$e = \left(\left\{ \begin{array}{c} \overrightarrow{X_k} \\ \overrightarrow{X_u} \end{array} \right\} - f\left(\left\{ \begin{array}{c} \overrightarrow{X_k} \\ \overrightarrow{X_u} \end{array} \right\}, \overrightarrow{W} \right) \right)^2$$
(4.1)

And to approximate the missing values, the above equation is minimized using genetic algorithm, or other traditional optimization methods. But for the referred study [10], GA was preferred because it offered a better probability to find global optimums.

After experimenting with different amount of missing values in single records, the proposed method were found to be approximating with great accuracy [10, Conclusion].

4.2.2 Using neural networks based on rough set theory

Another neural network based approach could be using rough neural networks. These kind of networks work by implementing rough sets. The advantage this kind of networks bring is the ability to process imperfect data in the form of intervals or simply missing data [7]. And this enables the network to give imprecise answers when imperfect data exists.

Traditional methods of handling missing data via data preprocessing come with their own disadvantages, such as loss of crucial data or creating bias in the dataset. Machine learning systems that can directly work with missing data can be a good solution for such cases.

Rough sets:

Rough sets are fundamentally an approach to handling imperfect data and knowledge [8].

In traditional set theory, sets are crisp, in the sense that each element of such a set should be able to classified as belonging to the set or not. Rough sets theory take a step away from this approach to be able to address vagueness in data.

Fuzzy sets also deal with such vagueness, but they incorporate this vagueness by allowing each element to belong to a set to a certain degree. (0 < k < 1) In contrast, rough sets uses pair of sets for lower and upper approximation of the original set. Those lower and upper sets can be crisp sets or fuzzy sets.

Rough neural networks:

The proposed architecture in the rough neural network paper [7] has input and output values represented as intervals, while the weights are still scalar as in classic neural networks.

To be able to process such inputs, each layer in the network consists of two subsets of neuron related to the intervals in the input and output. Although these two subsets share their weights, their inputs are mapped to their respective interval.

Experiments using this method proved that rough sets are resistant to a certain degree of missing data, and when the amount of missing data is too high it will result in unresolved classification rather than misclassification [7, Conclusion].

4.3 Proposed method

The proposed method used in this thesis work in a similar way to using a dropout method in neural networks. In the dropout method, certain neurons are dropped out of the network depending on a given probability, to prevent over-training. This decreases the interdependency of neighboring neurons [19].

The proposed method is similar in the sense that it achieves a partial training for inputs with missing values. It is offered as a way of training neural networks with imperfect data.

4.3.1 Theory



Figure 4.1: Basic neural network structure

When the neural network processes a complete input, the weights are updated as according the training method. As it can be seen from the above image, the weights between the input and hidden layer can be represented in the following matrix.

_

$$\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$
(4.2)

In this matrix, each column represents an input element's connections to the next layer. So w_{31} is the weight connecting the 3rd input to the 1st neuron in the hidden layer.

In the proposed method, only the weights relevant to the existing input are used in the training process. This is done using a masking feature provided by the NumPy library of Python. Effectively, masked away weights are dropped out of the training process.

So for an imperfect input of [1, 2, nan]:





Weights related to the 3rd input will be masked away and a subset of the weights will be used by the training algorithm.

More details about the method and the algorithm is given in the next chapter (5), using one simple dataset and another more complex one. Finally in the Implementation chapter (??), the method is applied on a real-life application of credit scoring.

5 Proof of Concept

This chapters includes details about the proof of concept developed and tested on two seperate datasets. Although results for both datasets are represented, the explanations are done using an simpler NN model.

However before delving into explanations, a detailed look into the masking and NaN (not a number) features of the Numpy library is needed.

5.1 Masking and NaN

5.1.1 NaN

Numpy library offers a custom object call NaN number that can be implemented in the arrays. This NaN object is usually created when missing or non-number values are read from a file. The advantage it offers is that, this object acts in a special and expected way when fed into calculations.

As expected, calculations including a NaN object returns NaN results. This becomes even more evident in matrix calculations.

```
# Importing the library.
import numpy as np
3
4 3 * np.nan # Output: nan
5 3 + np.nan # Output: nan
```

Listing 5.1: NaN values

This becomes even more evident in matrix calculations. A and B matrices and their representations in Python could be seen below.

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & nan \end{bmatrix}$$
(5.1)

```
1 A = np.array([[1, 2, 1], [1, 0, 1], [0, 1, 1]])
2 B = np.array([1, 2, np.nan])
3 np.dot(A, B)  # Output: [nan, nan, nan]
```

Listing 5.2: NaN values in matrices

The reason this is a simple but important detail is that, in the datasets missing values are represented by NaN, and when an imperfect input fed into a NN model, the NaN value spread through the network and show up in the output.

This proved to be a simple way to check if missing values were properly masked away in the process.

5.1.2 Masking

Masking feature of Numpy works in the following way.

First step is to create an array of binary values that represents the structure of the matrix to be masked. In this new matrix 1 represents the elements to masked, and 0 for elements to be used.

For example, for matrix A, if we want to mask the first column and do an operation on the last two, we can do as follows:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, A_{mask} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$
(5.2)

```
1 A = np.array([[1, 1, 1], [1, 1, 1]])
2 # A[0] = [1, 1, 1], first row.
3 # A[0][0] = 1, first row, first element
4
5 mask_pattern = [[1, 0, 0], [1, 0, 0]]
6
7 masked_A = np.ma.array(A, mask=pattern)
8 # [[masked, 1, 1], [masked, 1, 1]]
9
10 masked_A = masked_A * 3
11 # [[masked, 3, 3], [masked, 3, 3]]
```

```
12
13 # Unmasking
14 masked_A = np.array(masked_A) # new array
15 # Output: [[1, 3, 3], [1, 3, 3]]
```

Listing 5.3: Masking example

Unmasking and compressing:

There are two different ways to removing a mask from an array, and each had its use different parts of the algoritm.

Unmasking (creating a new array with masks removed), as seen below, returns an array with original dimensions. This part was useful during the training part.

Differently, compressing returns an array with compressed dimensions. Compressing was used in feed forward processing of input values.

```
1 # for A = [[masked, 3, 3], [masked, 3, 3]]
2 compressed_A = np.ma.compress_cols(masked_A)
3
```

4 # Output: [[3, 3], [3, 3]]

Listing 5.4: Compressing example

5.2 Feed forward processing

Masking features of Numpy was used in the feed forward processing on input (predicting). The main problem here is to prevent NaN values spreading through the calculations.

In a normal case, activation of layers are calculated as follows:

$$a_k = f(w.a_{k-1} + b) \tag{5.3}$$

```
1 # In Python, it can represented as below
2 # where sigmoid is the activation function
3 # of the layer.
4 a = t.sigmoid(np.dot(w, a) + b)
```

In the case of missing value in an input, the weight matrix connecting input layer to the hidden layer and input itself are masked and compressed, as explained in a previous section 5.1.2.

```
1 # Activation of hidden layer.
2
3 # Masking for input and weights.
4 h_pattern = t.create_weight_mask(x)
5 h_mask = [h_pattern for i in range(len(self.weights[0]))]
6
7 # Masked and compressed inputs and weights.
8 masked_w = np.ma.array(self.weights[0], mask=h_mask)
9 masked_a = np.ma.array(x, mask=h_pattern)
10 c_w = np.ma.compress_cols(masked_w)
11 c_a = np.ma.compress_rows(masked_a)
12
13 a = t.sigmoid(np.dot(c_w, c_a) + self.biases[0])
```

In a NN model with hidden layer of 4 nodes and input layer of 3 nodes, a partial calculation would look similar to the below equation. However, if the last element in input is missing ([1, 1, nan] for example), the bold parts of the matrices would be ignore in this part of calculation due to masking and compressing.

$$a = f \left(\begin{bmatrix} w_{11} & w_{21} & \mathbf{w_{31}} \\ w_{12} & w_{22} & \mathbf{w_{32}} \\ w_{13} & w_{23} & \mathbf{w_{33}} \\ w_{14} & w_{24} & \mathbf{w_{34}} \end{bmatrix} x \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right)$$
(5.4)

5.3 Training

Training algoritm is applied by two separate functions. Training data is shuffled, to make sure each training epoch is unique, afterwards, for each input vector *update_weights* method is called.

```
    for data in training_data:
    # Weights and biases are updated according to the gradient descent calculated
```

```
# on each input and output pair.
self.__update_weights(data, eta)
# self refers to the network model.
```

In general, *update_weights* method calculates gradients using a *backpropogation* method and updates the weights accordingly.

$$\overrightarrow{w} = \overrightarrow{w} - \alpha \overrightarrow{\nabla C} \tag{5.5}$$

5.3.1 Method: update_weights

The aim of this method is to update weights with the calculated gradients received from the *backpropogation* method. With an input without missing values, weights are simply update as follows.

```
1 # Gradient calculation
2 nabla_b, nabla_w = self.__backpropogation(x, y)
3
4 # Weights and bias update.
5 self.weights = [w - (eta * nw) for w, nw in zip(self.weights, nabla_w)]
6 self.biases = [b - (eta * nb) for b, nb in zip(self.biases, nabla_b)]
```

But with an imperfect method, this simple calculation cannot be used because, the output of *backpropogation* method includes NaN values, due to way the backward propogation was implemented, which will be explained in the next sub-section. But for now, following matrix can represent the gradient calculation result for the weights connecting the input to the hidden layer.

```
# Gradient calculation result of hidden layer weights.
<sup>2</sup> # For an input with last element missing:
3 #
       [210, 120, nan]
4 [[ 0.0072174
                  0.0068817
                                       nan]
  [ 0.00523259 0.00498921
                                       nan]
  [-0.0063878]
                 -0.00609069
                                       nan]
  [-0.00178148 - 0.00169862]
                                       nan]
7
  [ 0.00932666 0.00889287
                                       nan]
  [-0.00095003 - 0.00090584]
                                       nan]
```

10	[0.00151193	0.00144161	nan]	
11	[-0.00074261	-0.00070807	nan]	
12	[0.00414442	0.00395166	nan]	
13	[0.00458916	0.00437571	nan]]	

This requires implementing masking during the weight update process. (It should be noted that, I discovered later that it's possible to replace nan values with 0 in this stage and simplify this process. Small details like this could be fixed in a later stage if further optimization is needed.)

At this point, the method creates empty arrays with the shape of weights and biases, and calculated gradients are add to these arrays accordingly. But before addition, small masking operation needs to be done, to update only weights relevant to the input values. As explained in a previous section (5.1), masking pattern is created using the input and the shape of weight matrix.

```
1 # Creating the mask
2 h_pattern = t.create_weight_mask(x)
3 h_mask = [h_pattern for i in range(len(nabla_w[0]))]
4
5 # Masked hidden layer weights.
6 masked_hw = np.ma.array(nabla_w[0], mask=h_mask)
7
8 # Updating and unmasking.
9 # nabla_w is an zeros array created in the shape
10 # weights of the network model. And delta_nabla_w
11 # is one of the outputs of backpropogation method.
12 masked_hw += delta_nabla_w[0]
13 nabla_w[0] = np.array(masked_hw)
```

5.3.2 Method: backpropogation

This method calculates the gradients which will be used in weight update part of the training, as explained in the above sub-section.

In normal operation, where the input is not imperfect, arrays with the shapes of weights and biases are created.

```
# Empty matrices with bias shapes / sizes.
```

```
2 nabla_b = [np.zeros(b.shape) for b in self.biases]
3 nabla_w = [np.zeros(w.shape) for w in self.weights]
```

In the forward phase, activations of layers and z-values are kept in seperate lists. It should be noted that the input is considered as the first activation. This, as it will be seen later, is the reason why the gradient calculated by the method includes NaN values.

```
1 activations = [x] # x: input
2 zs = []
3
4 # Empty matrices with bias shapes / sizes.
5 z = np.dot(w, activation) + b
6 zs.append(z)
7
8 # New activation.
9 activation = t.sigmoid(z)
10 activations.append(activation)
```

On the backwards pass, a delta value is calculated using output layer z-values and activations.

```
1 # activations[-1] : last activation (output)
2 # activations[-2] : hidden layer activation.
3
4 delta = self.cost_function_prime(activations[-1], y) * t.sigmoid_prime(
        zs[-1])
5
6 # Calculations for gradients for output layer.
7 nabla_b[-1] = delta
8 nabla_w[-1] = np.dot(delta, activations[-2].transpose())
```

Calculations for gradients for hidden layer uses update delta value, and the z-values and weights of the previous layer (output layer) and the first activation (input).

```
sp = t.sigmoid_prime(z)
delta = np.dot(self.weights[-1].transpose(), delta) * sp

# Gradients
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-3].transpose())
```

The last line that uses the first activation is the reason we still see NaN values even after masking. Further masking could be used here, but it was instead implemented in the function that calls the *backpropogation* method.

As to the how the this calculation is done with an imperfect value, the masking is used with the forwards pass where z-values are calculated. So that NaN wouldn't propogate through the network.

```
1 compressed_w = np.ma.compress_cols(masked_w)
2 compressed_a = np.ma.compress_rows(masked_a)
3 z = np.dot(compressed_w, compressed_a) + b
```

The rest of the calculations are done in a similar way to the scenario where the input is not imperfect.

6 Implementations

This proof of concept has been used on four separate datasets. Skin color (6.1) and number recognition datasets (6.2) are typical classification problems while the last two datasets (6.3 and 6.4) are simple regression problems.

For the datasets, no action was taken to remove the randomness in the process. This means that during each epoch training dataset was shuffled.

For classification datasets, missing values were created randomly over the whole dataset. This resulted in some inputs having only a single NaN while others having more than one.

However, for the summing function dataset I have only randomly removed the last variable due to small amount of variables used in the dataset, and to be able to compare the results more efficiently. The last dataset includes input with a range [13-15] numbers.

For this reason each data test, 3 different training was done and the average was taken.

The individual results of training with variable amount of missing value can be seen in the following sections.

6.1 Skin Color Dataset

Input Size	3
Hidden Layer Size	10
Output Layer Size	2
Training Rate	0.5

Table 6.1: Skin color dataset NN model structure

This is a very simple classification dataset, with 3 inputs and an output with 2 classes. Numbers represent red, green and blue values of pixes and output represents whether this pixel belongs to skin or not.

75% percent of the data (50000 records) was used on training and the rest on validation.

This dataset was choosen for its simplicity and small amount inputs, as it made reasoning about the partial training of weights slightly easier during the initial development of the proof of concept.

The training results and comparison can be seen below.

Missing (%)	Initial Match (%)	E1 (%)	E2 (%)	E3 (%)	E4 (%)	E5 (%)
0	56.99	97.70	98.84	98.79	99.19	99.03
5	57.11	97.82	98.66	99.20	99.02	99.04
10	51.46	98.26	98.84	98.77	98.18	98.63
20	57.11	91.13	96.06	97.13	96.21	96.34
30	42.89	94.91	95.91	96.26	96.45	95.49

 Table 6.2: Skin color dataset results



6.2 MNIST Digit Dataset

Table 6.3: MNIST dataset NN model structure

Input Size	784
Hidden Layer Size	12
Output Layer Size	10
Training Rate	0.5

MNIST database is a handwritten digits database that is commonly used with machine learning tasks for benchmarking purposes.

Being a modified version of the original dataset from National Institute of Standards and Technology located in United States, the digits are normalized and centered at a fixed 28x28 size. Each pixel represent its darkness with a fraction in the [0, 1] range.

Inputs are vectors with 784 elements, and outputs are represented as vectors with a size of 10, each element representing a different digit. The training data size is 50 thousand, and 10 thousand records are used for validation and testing.

Such a vectorized number looks as following when visualized by Python's plotting library.



Figure 6.2: Number visualization

With 5% and 10% percent missing data, this number would look as follows:



Figure 6.3: Number visualization with missing data

This dataset was chosen for its somewhat more complex structure and benchmarking opportunities. It provided a good tool to test the proof of concept.

It should be noted that following results are for random missing data created over **the whole data**. This means that individual missing data points for each row of data varies. Although with higher percentages, it results in a dataset with each row having missing data.

Below can be seen the histogram of rows with missing data for randomly created 5% missing data.





As it can be seen below, this situation highly affects the end results of the training.

Missing (%)	Initial Match (%)	E1 (%)	E2 (%)	E3 (%)	E4 (%)	E5 (%)
0	08.81	88.31	90.03	90.51	90.65	90.89
5	08.63	43.36	41.14	42.95	41.34	43.93
10	12.74	42.77	38.57	36.47	38.23	37.06
15	11.10	32.04	31.21	27.45	25.49	26.14

Table 6.4: Skin color dataset results

Figure 6.5: Epoch 5 Match Rates



6.3 Summing Function Dataset

Input Size	4
Hidden Layer Size	10
Output Layer Size	1
Training Rate	0.05

Table 6.5: Summing formula dataset NN model structure

This data set is created for floating numbers between 0 and 10 over the following summing function:

$$f(a,b,c,d) = 2a - 3b + 4c - 5d \tag{6.1}$$

Over 50000 rows are created, out of these rows random 10000 were chosen for training and 5000 rows were chosen for validation. Due to the limited number of variables and being a regression problem, missing variable was created only in the last variable (d).

Since this is mostly a floating-point arithmetics, a precision amount (epsilon) was used to calculate the correctness of the results.

Initial results after training with complete data can be seen below.

Table 6.6: Summing function dataset training results with complete data.

Epsilon	Average Match
0.01	98.58

And the results for training with missing data can be seen below.

Table 6.7: Summing function dataset training results with missing data.

Missing (%)	Match (%), eps = 0.01	Match (%), eps = 0.1
5	27.77	88.92
10	20.83	71.00

A neural network trained with 5% missing data was also put to a simple test using random number choices that wasn't included in the validation dataset. The following neural network gives 49.66% match rate at epsilon = 0.01 and 92.68% match rate at epsilon = 0.1.

Table 6.8: Comparison of Neural Network trained with missing data to the function

a	b	c	d	Function	NN	Deviation
13	17	21	10	9	8.9909	0.0090
1.25	13.6	-4.5	-9.4	-9.3	-9.4359	0.1359
4	5	6	7	-18	-18.001	0.0012
13	-50	12.41	99.9	-273.86	-273.94	0.0857

6.4 Averaging Function Dataset

Input Size	15
Hidden Layer Size	25
Output Layer Size	1
Training Rate	0.001

Table 6.9: Averaging formula dataset NN model structure

This dataset was used to see if this method could be used for creating a NN model that can take average of varying amount on input numbers. So the missing variables in these inputs didn't represent actual missing information.

Different combinations of training and validation datasets were used.

Table 6.10: Dataset variations					
Training Dataset					
Туре	15 numbers	14 numbers	13 numbers	Total Size	
Full	12000	0	0	12000	
Missing	10000	1000	1000	12000	

Validation Dataset

Туре	15 numbers	14 numbers	13 numbers	Total Size	
Full	12000	0	0	12000	
Missing	10000	1000	1000	12000	

The goal was to check the NN output results with different combinations of data. Missing datasets contain a combination of inputs that have 15, 14, or 13 numbers. In general, this came to a **1.70%** missing data over all the datapoints.

The training results for different combinations can be seen below.

		Match rates in different precisions				
Training Data	Validation Data	0.1	0.2	0.3	0.4	0.5
Full	Full	81.70%	96.54%	99.20%	99.75%	99.86%
Full	Missing	68.82%	83.07%	88.69%	92.47%	94.76%
Missing	Full	52.95%	90.54%	96.75%	98.63%	99.29%
Missing	Missing	44.79%	79.09%	88.52%	91.95%	94.68%

Table 6.11: Training results

7 Conclusion

Method

The results of training with different sets of data shows that the acceptability of this method depends highly on the amount and **distribution** of missing data, precision required and the task at hand.

Classification tasks are slightly more fitting for this method, considering how they might have differing amount of redundant data. However, it was shown that above 5% missing data, match rates drop fast. Although it was possible to get somewhat satisfying match rates below 1% percent (75.27% match rate at 0.5% missing data).

One reason for the low match rates above 5% can be seen at the histogram of the dataset provided above (fig. 6.4). This is the result of the method used to create the missing datapoints. Missing points are created over the whole data set. This means that in higher percentages, each row has missing values.

In a more organic dataset, missing values would be limited to only certain rows of data, hence increasing the usability of this method for such datasets.

This could already be seen at the summing function implementation, where the missing data is only in a 10% of the rows. The method can provide passable results in such cases, depending on the precision required.

For the last implementation, good results were acquired for 0.3 precision when using missing training data. However, in terms of creating a NN structure that mimics a generic averaging function, it wasn't as successful.

In general, I believe that masking method would deserve further research in the area of handling imperfect missing data. The results show that it could have promise for handling such cases, however it doesn't quite offer a complete solution to the problem yet.

Python

Python and its libraries proved to be very capable tools in both handling datasets and implementing neural networks. They offer a lot of tools and methods, and more importantly, flexibility, to experiment with different structures and data.

References

- LendingClub Open Dataset, https://www.lendingclub.com/info/downloaddata.action [Accessed February 1, 2017]
- [2] Numpy, Python Library, http://www.numpy.org/ [Accessed March 10, 2017]
- [3] Pandas, Python Data Analysis Library, http://pandas.pydata.org/ [Accessed March 10, 2017]
- [4] Tensorflow library, https://www.tensorflow.org/ [Accessed March 10, 2017]
- [5] Abadi, Martin, Tensorflow: a system for large scale machine learning. 2016, OSDI 2016
- [6] Keras, https://keras.io/ [Accessed March 10, 2017]
- [7] Nowicki, R.K., Scherer, R., Rutkowski, L., 2016. Novel rough neural network for classification with missing data, in: 2016 21st International Conference on Methods and Models in Automation and Robotics, MMAR 2016. doi:10.1109/MMAR.2016.7575243
- [8] Pawlak, Z., 1982. *Rough sets*. International Journal of Computer & Information ...
 1–51. doi:10.1007/978-94-011-3534-4
- [9] Little, R.J. a, Rubin, D.B., 2002. Statistical Analysis with Missing Data, 2nd editio. ed, Statistical analysis with missing data. John Wiley & Sons, Hoboken. doi:10.2307/1533221
- [10] Abdella, M., Marwala, T., 2005. Treatment of missing data using neural networks and genetic algorithms. Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. 1, 598–603. doi:10.1109/IJCNN.2005.1555899
- [11] Mnist, digit recognition dataset, http://yann.lecun.com/exdb/mnist/ [Accessed February 10, 2017]

- [12] Kevin Gurney. 1997. An Introduction to Neural Networks. Taylor & Francis, Inc., Bristol, PA, USA.
- [13] Jeff Heaton. 2012. Introduction to Math of Neural Networks. Heaton Research, USA.
- [14] Dr. Singh Yashpal, S.C.A., 2009. Neural Networks In Data Mining. Journal of Theoretical and Applied Information Technology 5, 1–154.
- [15] David Kriesel, 2007, A Brief Introduction to Neural Networks, available at http://www.dkriesel.com.
- [16] Michael Nielsen, http://neuralnetworksanddeeplearning.com [Accessed January 10, 2017]
- [17] S. Thirukumaran and A. Sumathi, "*Missing value imputation techniques depth survey and an imputation Algorithm to improve the efficiency of imputation*," 2012
 Fourth International Conference on Advanced Computing (ICoAC), Chennai, 2012, pp. 1-5.
- [18] Paul D. Allison. 2002. Missing Data. Sage Publications, USA.
- [19] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15, 1929–1958.

Appendices

Appendix A - tools.py

This is a small script of helper functions used in other code parts.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
4 class DF(object):
      """ Class to handle dataframe related tasks. """
      @staticmethod
7
      def col_values(df, col, display=False):
8
          """ Return a count of values of the given column. """
9
          res = df[col].value_counts()
10
          return res
12
      @staticmethod
      def col_null_count(df, col, display=False):
14
          """ Return a count of null values in the given column. """
15
          count = df[col].isnull().sum()
16
          return count
17
18
  def divide_data(data, t_percentage):
19
      """ Divide the given data by fraction. """
20
      portion = len(data) * t_percentage / 100
21
      dt = data [: portion]
23
      dv = data [portion :]
24
      return (dt, dv)
25
26
  def visualize_number(a_vector):
27
      """ Takes a Numpy array vector and visualizes it. """
28
29
      a_vector = np.nan_to_num(a_vector) # reshaping
30
      a_vector = np.array(a_vector, dtype=np.float64)
31
```

```
32
      pixels = a_vector.reshape(28, 28)
33
      plt.title("Number")
34
      plt.imshow(pixels, cmap="gray_r")
35
      plt.show()
36
37
  def nan_input(inp, percentage):
38
      """ Make random missing in each input. """
30
      inp = np.array(inp)
40
      amount = int(len(inp) * percentage / 100.0)
41
      inp.ravel()[np.random.choice(inp.size, amount, replace=False)] = np
42
      . nan
      return inp
43
44
  # ACTIVATION FUNCTIONS
45
  def sigmoid(x):
46
      """ A sigmoid activation function."""
47
      return 1.0 / (1.0 + np.exp(-x))
48
49
  def softmax(x):
50
      return np.exp(x) / np.sum(np.exp(x), axis=0)
51
52
  def sigmoid_prime(x):
53
      """ Derivation sigmoid(x) """
54
      return sigmoid(x) * (1 - sigmoid(x))
55
56
 def linear(x): return x
57
58
  def linear_prime(x): return np.ones(x.shape)
59
60
  def quadratic_cost_prime(z, a, y):
61
      """ Return the error delta from the output layer."""
62
      return (a-y) * sigmoid_prime(z)
63
64
  def ReLU(x): return np.maximum(0, x)
65
66
  def ReLU_prime(x): return (x > 0). astype(float)
67
68
69 def leakyReLU(x, leakage=0.01):
```

```
70
      output = np.copy(x)
      output[output < 0] *= leakage</pre>
71
      return output
73
 def leakyReLU_prime(x, leakage=0.01):
74
      return np. clip (x > 0, leakage, 1.0)
75
76
77 # MASKING RELATED FUNCTIONS
 def create_weight_mask(arr):
78
      """ Create a mask for a given input. """
79
    return [1 if np.isnan(i) else 0 for i in arr]
80
```

Appendix B - mnist.py

This script contains functions and classes that help with parsing MNIST data for the neural network.

```
import pickle, gzip
2 import numpy as np
4 class MnistData():
      dim = 784
5
      def __init__(self, path):
          """ Load the raw data from a pickle file. """
          file = gzip.open(path, "rb")
8
          self.training_data , self.validation_data , self.test_data =
9
     pickle.load(file)
          file.close()
10
      def make_vector(self, num):
12
          """ Create 10-d vector out of given num. """
13
          zeros = np. zeros((10, 1))
14
          zeros[num] = 1.0
15
          return zeros
16
17
      def get_training_data(self):
18
          """ Return a zipped training data. """
19
          inputs = [np.reshape(x, (784, 1)) for x in self.training_data
20
     [0]]
```

```
21
          results = [self.make_vector(y) for y in self.training_data[1]]
          return zip(inputs, results)
22
23
      def get_missing_training_data(self, percentage):
24
          # Random missing values.
          inputs = [np.reshape(x, (784, 1)) for x in self.training_data
26
     [0]]
          results = [self.make_vector(y) for y in self.training_data[1]]
27
28
          # Amount of missing data points.
29
          amount = int(len(inputs) * percentage / 100.0)
30
31
          # Converting to float64 to prevent overflow.
32
          inputs = np. array (inputs, dtype=np. float128)
33
          inputs.ravel()[np.random.choice(inputs.size, amount, replace=
34
     False ] = np.nan
          return zip(inputs, results)
35
36
      def get_all_missing_training_data(self, percentage):
37
          """ Create missing data all training inputs. """
38
          # Random missing values.
39
          inputs = [np.reshape(x, (784, 1)) for x in self.training_data
40
     [0]]
          results = [self.make_vector(y) for y in self.training_data[1]]
41
42
          # Amount of missing data points.
43
          amount = int(len(inputs[0]) * percentage / 100.0)
44
          # Converting to float64 to prevent overflow.
45
          inputs = np. array (inputs, dtype=np. float128)
46
47
          for inp in inputs:
48
             inp.ravel()[np.random.choice(inp.size, amount, replace=False)
49
     ] = np.nan
          return zip(inputs, results)
50
51
      def get_new_missing_training_data(self, percentage):
52
          """ Create missing data over all training inputs. """
53
54
```

```
55
          inputs = [np.reshape(x, (784, 1)) for x in self.training_data
     [0]]
          results = [self.make_vector(y) for y in self.training_data[1]]
56
57
          # Amount of missing data points. 50000 rows * 784 columns.
58
          datasize = len(inputs) * len(inputs[0])
59
          amount = int (datasize * percentage / 100.0)
60
61
          # Converting to float64 to prevent overflow.
62
          inputs = np. array (inputs, dtype=np. float128)
63
          inputs.ravel()[np.random.choice(inputs.size, amount, replace=
64
     False)] = np.nan
          return zip(inputs, results)
65
66
      def get_validation_data(self):
67
          """ Return a zipped validation data. """
68
          inputs = [np.reshape(x, (784, 1)) for x in self.validation_data
69
     [0]]
          results = [self.make_vector(y) for y in self.validation_data
70
      [1]]
          return zip(inputs, results)
71
72
      def get_test_data(self):
73
          """ Return a zipped validation data. """
74
          inputs = [np.reshape(x, (784, 1)) for x in self.test_data[0]]
75
          results = [self.make_vector(y) for y in self.test_data[1]]
76
          return zip(inputs, results)
77
78
      def get_training_images(self): return self.training_data[0]
79
80
      def get_validation_images(self): return self.validation_data[0]
81
82
      def get_test_images(self): return self.test_data[0]
83
84
      def get_training_labels(self, vector=False):
85
          labels = self.training_data[1]
86
          if vector:
87
               labels_v = list()
88
               for x in labels:
89
```

```
90
                     labels_v.append(self.make_vector(x))
                return labels_v
91
           else:
92
                return labels
93
94
       def get_validation_labels(self, vector=False):
95
           labels = self.validation_data[1]
96
           if vector:
97
                labels_v = list()
98
                for x in labels:
99
                     labels_v.append(self.make_vector(x))
100
                return labels_v
101
           else:
102
                return labels
103
104
       def get_test_labels(self, vector=False):
105
           labels = self.test_data[1]
106
           if vector:
107
                labels_v = list()
108
                for x in labels:
109
                     labels_v.append(self.make_vector(x))
110
                return labels_v
           else :
112
                return labels
113
```

Appendix C - neural.py

Neural network class implemented in Numpy and Python. Following structure was used with linear neural network implementations.

```
import pandas as p
import numpy as np
import tools as t
from keras.utils.np_utils import to_categorical
from math import fabs
class LinearNeuralNetwork(object):
    """ A neural network model with linear activation functions. """
```

```
10
      mini_batch_size = 10
11
      # File names for the saved states.
      bias_filename = "states/bn_biases.npy"
13
      weights_filename = "states/bn_weights.npy"
14
15
      epsilon = 0.1 \# default precision
16
      def __init__(self, layers, debug=False):
18
           """ Initialization. """
19
20
           self.number_of_layers = len(layers) # Total number of layers.
21
          self.layers = layers
                                                  # Layers and their size.
22
          self.weights = list()
24
          self.biases = list()
25
          self.debug = debug
26
27
          if debug:
28
               # Load pre-saved states.
29
               # This will provide a similar starting point.
30
               print "> Loading states from file."
31
               self.biases, self.weights = self.__load_states()
32
33
          else :
34
                          Randomly initializing states."
               print ">
35
               self.biases = self.__initialize_biases()
36
               self.weights = self.__initialize_weights()
37
38
          # Training and validation data.
39
          self.dt = None
40
           self.dv = None
41
           print "> Network initialized randomly."
42
43
      # PRIVATE METHODS
44
      def __save_state(self):
45
           """ Save the current state of the network. """
46
47
          np.save(self.bias_filename, self.biases)
48
```

```
49
          np.save(self.weights_filename, self.weights)
           print ">
                      Network state is saved."
50
51
      def __load_states(self):
52
           """ Loads state from save *npy file. """
53
54
          b = np.load(self.bias_filename)
55
          w = np.load(self.weights_filename)
56
57
          return b, w
58
      def __initialize_weights(self):
59
           """ Randomly initialize weights. """
60
          # We create a transposed weights lists.
61
          # weights [0] \rightarrow weights connecting layers [0] \rightarrow layers [1]
62
63
          # Creating a list of tuples that represent w connections.
64
          # Example: layers = [A B C] where A, B, C are integers.
65
          #
                        zipped = [(A, B), (B, C)]
66
          zipped = zip(self.layers[:-1], self.layers[1:])
67
68
          return [np.random.randn(y, x) for x, y in zipped]
          # Example:
70
          \# layers = [2, 3, 1]
71
          \# weights = [array ([[ 0.75054289, -0.18324735],
                                 [0.32141292, -0.54226539],
          #
                                 [-0.53575605, 0.25761202]]),
          #
74
          #
                         array ([[ 0.22304971, 1.29879581, -0.49437018]])]
75
76
      def __initialize_biases(self):
77
           """ Randomly initialize biases. """
78
          # We basically create a vector via transpose ...
79
          # column vectors are needed.
80
          # If a layer has 3 neurons, randn(3,1) creates a 3x1 matrix.
81
          # Which represented in Python in the following way:
82
          # [[0.1],
83
          # [0.2],
84
          # [0.1]]
85
          # biases [0] -> biases in layers [1]
86
          return [np.random.randn(x, 1) for x in self.layers[1:]]
87
```

```
# METHODS.
89
       def process(self, a, label=False):
90
           """ Input to the system is processed through the layers
91
                of the network, and output is produced. """
92
           if np.isnan(a).sum() > 0:
93
                return self.process_nan(a, label)
94
           else:
95
               # With each step, activation of layer is updated
96
               # Hidden layer activation
97
               z = np.dot(self.weights[0], a) + self.biases[0]
98
               a = t.leakyReLU(z)
99
               # Output layer activation
100
               z = np.dot(self.weights[1], a) + self.biases[1]
101
               a = t.leakyReLU(z)
102
103
                if label: return np.argmax(a)
104
                return a
105
106
       def process_nan(self, x, label=False):
107
           """ Calculation the activation with missing values. """
108
109
           # HIDDEN LAYER
           # Mask is used to calculated the output.
111
           h_pattern = t.create_weight_mask(x)
           h_mask = [h_pattern for i in range(len(self.weights[0]))]
113
114
           masked_w = np.ma.array(self.weights[0], mask=h_mask)
115
           masked_a = np.ma.array(x, mask=h_pattern)
116
           a = t.leakyReLU(np.dot(np.ma.compress_cols(masked_w),
               np.ma.compress_rows(masked_a)) + self.biases[0])
118
119
           # OUTPUT LAYER
120
           a = t.leakyReLU(np.dot(self.weights[1], a) + self.biases[1])
121
           if label: return np.argmax(a)
123
124
           return a
125
       def train (self, epochs, eta):
126
```

88

```
.....
               Training the neural network using training_data. """
127
128
           training_data = self.dt
129
           test_data = self.dv
130
                       Starting the training."
           print ">
                       Epochs \t: ", epochs
           print ">
           print ">
                       Eta \t: ", eta, "\n"
           print ">
                       Initial match rate: {0} %".format(self.evaluate(
134
      test_data))
135
           for e in xrange(epochs):
136
               # Shuffling the data to increase the randomness.
               if not self.debug:
138
                    np.random.shuffle(training_data)
139
                    print ">
                                Shuffling data."
140
                          Epoch {0} : Updating.".format(e+1)
               print ">
141
               for data in training data:
142
                    # Weights and biases are updated according to
143
                    # the gradient descent calculated.
144
                    self.__update_weights(data, eta)
145
146
               print ">
                           Epoch {0}: Completed.".format(e+1)
147
                print ">
                           Epoch \{0\}: Match rate: \{1\}\%".format(e+1, self.
148
      evaluate(test_data))
149
       def __update_weights(self, data, eta):
150
           """ Update the weights and biases.
                                                 .....
           x = data[0]
           y = data[1]
153
154
           # Check for missing values.
155
           if np.isnan(x).sum() > 0:
156
               # Creating empty matrices with bias and weight dims.
157
               nabla_b = [np.zeros(b.shape) for b in self.biases]
158
               nabla_w = [np.zeros(w.shape) for w in self.weights]
159
160
               # Calculate a subset of Delta Nablas.
161
               delta_nabla_b, delta_nabla_w = self.__backpropogation(x, y)
162
               # Before we add calculated gradients, we need to
163
```

```
164
                # use masking because, for inputs with nan, there
                # are also nan's related to these values from the
165
                # backpropogation output.
166
                h_{pattern} = t.create_{weight_{mask}(x)}
167
                h_mask = [h_pattern for i in range(len(nabla_w[0]))]
168
169
                # Masked hidden layer weights.
170
                masked_hw = np.ma.array(nabla_w[0], mask=h_mask)
                # Updating masked weights.
                masked_hw += delta_nabla_w[0]
                nabla_w[0] = np. array (masked_hw)
174
                # Updating output layer.
175
                nabla_w[1] += delta_nabla_w[1]
176
177
                # Updating biases as usual.
178
                nabla_b = [nb + dnb for nb, dnb in zip(nabla_b,
179
      delta nabla b)]
           else: # Full set of data comes in.
180
                nabla_b , nabla_w = self.__backpropogation(x, y)
181
182
           self.weights = [w - (eta * nw) \text{ for } w, nw \text{ in } zip(self.weights,
183
      nabla_w)]
           self.biases = [b - (eta * nb) for b, nb in zip(self.biases,
184
      nabla_b)]
185
       def __backpropogation(self, x, y):
186
           """ Backpropogation method used with supplied input and output.
187
               REF: http://neuralnetworksanddeeplearning.com/chap2.html
188
           .....
189
           # Empty matrices with bias shapes / sizes.
190
           nabla_b = [np.zeros(b.shape) for b in self.biases]
191
           nabla_w = [np.zeros(w.shape) for w in self.weights]
192
193
           if np.isnan(x).sum() > 0:
194
                h_pattern = t.create_weight_mask(x)
195
                h_mask = [h_pattern for i in range(len(nabla_w[0]))]
196
197
                                   # The initial activation.
                activation = x
198
                activations = [x] # List of activations
199
```

```
200
                zs = []
                                   # List of outputs
201
                # HIDDEN LAYER.
202
                # We need to mask weights and activation for this
203
                # layer because they included nan values.
204
                masked_w = np.ma.array(nabla_w[0], mask=h_mask)
205
                masked_a = np.ma.array(activation, mask=h_pattern)
206
                b = self.biases[0]
207
208
                # Masking out the nan related values.
209
                z = np.dot(np.ma.compress_cols(masked_w), np.ma.
210
      compress_rows(masked_a)) + b
                zs.append(z)
211
                # New activation.
213
                activation = t.leakyReLU(z)
214
                activations.append(activation)
215
216
               # OUTPUT LAYER.
               w = self.weights[1]
218
               b = self.biases[1]
219
                z = np.dot(w, activation) + b
220
                zs.append(z)
221
                # New activation.
                activation = t.leakyReLU(z)
224
                activations.append(activation)
225
226
                # BACKPROPOGATION PART.
227
                delta = self.cost_function_prime(activations[-1], y) * t.
228
      leakyReLU_prime(zs[-1])
                nabla_b[-1] = delta
229
                nabla_w[-1] = np.dot(delta, activations[-2].transpose())
230
231
                # LAYERS
                # Handles only one hidden layer currently.
233
                \# 1 = 2 because with [inp, hidden, outp] structure...
234
                # the last layer. index is 2. Now we go "back"
235
                1 = 2
236
```

```
z = zs[-1]
237
238
                sp = t.leakyReLU_prime(z)
239
                 delta = np.dot(self.weights[-1 + 1].transpose(), delta) *
240
      sp
                nabla_b[-1] = delta
241
                nabla_w[-1] = np.dot(delta, activations[-1 - 1].transpose()
242
      )
243
                return nabla_b , nabla_w
244
            else :
245
                 activation = x
246
                 activations = [x]
247
                zs = []
248
                # FORWARD PASS
249
                # HIDDEN LAYER
250
                w = self.weights[0]
251
                b = self.biases[0]
252
253
                z = np.dot(w, activation) + b
254
                zs.append(z)
255
256
                # New activation.
257
                 activation = t.leakyReLU(z)
258
                 activations.append(activation)
259
260
                # OUTPUT LAYER
261
                w = self.weights[1]
262
                b = self.biases[1]
263
264
                z = np.dot(w, activation) + b
265
                zs.append(z)
266
267
                # New activation.
268
                 activation = t.leakyReLU(z)
269
                 activations.append(activation)
270
271
                # BACKPROPAGATION
272
```

```
273
                delta = self.cost_function_prime(activations[-1], y) * t.
      leakyReLU_prime(zs[-1])
                nabla_b[-1] = delta
274
                nabla_w[-1] = np.dot(delta, activations[-2].transpose())
275
276
                for 1 in xrange(2, self.number_of_layers):
277
                    z = zs[-1]
278
                    sp = t.leakyReLU_prime(z)
279
                    delta = np.dot(self.weights[-1 + 1].transpose(), delta)
280
       * sp
                    nabla_b[-1] = delta
281
                    nabla_w[-1] = np.dot(delta, activations[-1 - 1]).
282
      transpose())
283
                return nabla_b, nabla_w
284
285
       def cost function prime(self, output activation, y):
286
           """ Derivation of the used cost function. """
287
           return output_activation - y
288
289
       def evaluate(self, test_data):
290
           """ Evaluate the neural network success rate.
291
                Returns match rate as percentage. """
292
           correct = 0
293
294
           for (x, y) in test_data:
295
               # y
                      -> [[ <val> ]]
296
               # res -> [[ <val> ]]
297
                res = self.process(x)
298
299
                if abs(res - y) <= self.epsilon:
300
                    correct += 1
301
           return 100.0 * correct / (len(test_data) * 1.0)
302
```

Appendix D - train.py

A sample training script to used with above neural network structure.

```
import numpy as np
```

```
2 import pandas as pd
3 import tools as t
4 from neural import LinearNeuralNetwork
6 def get_data(training=True, missing=False):
      """ Get training data. """
8
      if training:
0
           if missing:
10
               filename_xs = "data/avg/training_missing_xs.csv"
               filename_y = "data/avg/training_missing_y.csv"
12
          else :
13
               filename_xs = "data/avg/training_full_xs.csv"
14
               filename_y = "data/avg/training_full_y.csv"
15
      else:
16
          if missing:
17
               filename_xs = "data/avg/validation_missing_xs.csv"
18
               filename_y = "data/avg/validation_missing_y.csv"
19
          else :
20
               filename_xs = "data/avg/validation_full_xs.csv"
               filename_y = "data/avg/validation_full_y.csv"
23
      # Preparing X's.
24
      xs = pd.read_csv(filename_xs).as_matrix()
25
      temp_x s = list()
26
      for i in xs:
27
          temp_xs.append(i.reshape(15, 1))
28
      xs = np.array(temp_xs)
29
      xs = xs.astype(np.float128)
30
31
      # Preparing Y's.
32
      ys = pd.read_csv(filename_y).as_matrix()
33
      temp_y s = list()
34
      for i in ys:
35
          temp_ys.append(i.reshape(1,1))
36
      ys = np.array(temp_ys)
37
      ys = ys.astype(np.float128)
38
39
      return zip(xs, ys)
40
```

```
<sup>41</sup>
<sup>42</sup> # Training data, full and mising versions
<sup>43</sup> tfd = get_data(training=True, missing=False)
<sup>44</sup> tmd = get_data(training=True, missing=True)
<sup>45</sup> # Validation data, full and missing versions
<sup>46</sup> vfd = get_data(training=False, missing=False)
<sup>47</sup> vmd = get_data(training=False, missing=True)
<sup>48</sup>
<sup>49</sup> n = LinearNeuralNetwork([15, 75, 1])
<sup>50</sup> n. dt = tmd
<sup>51</sup> n. dv = vfd
<sup>52</sup> n. train(10, 0.001)
```