Towards Better Predictivity Between Two Random Variables

Major Qualifying Project

Advisor(s):

RANDY PAFFENROTH
JACOB WHITEHILL

Written By:

MINH PHAM

A Major Qualifying Project
WORCESTER POLYTECHNIC INSTITUTE

Submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science and Bachelor of Science in Mathematical Sciences.

AUGUST 2020 - MAY 2021

Abstract

This Major Qualifying Project dives into the classic problem in machine learning where we want to predict one random variable from another random variable and. Having taken inspiration from Canonical Correlation Analysis (linear methods) and autoencoders (non-linear methods), we propose various algorithms to improve the predictability between two random variables.

Our work provides interesting insights into how to train neural networks to reconstruct one variable from another. In particular, it underlines the importance of obtaining an information-rich latent representation of each variable. Moreover, we demonstrated the importance of the hidden representation of an autoencoder and the benefits of utilizing random data correctly towards our problem.

ACKNOWLEDGEMENTS

This Major Qualifying Project would hardly be possible without the guidance and support from both of my advisors, Professor Randy Paffenroth and Professor Jacob Whitehill.

TABLE OF CONTENTS

Page

List of Tables v

List of Figures vi

1 Introduction 1

1.1 Motivation 1

2 Background 4

2.1 Notation and Definitions 4

2.2 Canonical Correlation Analysis 4

2.3 Deep Neural Networks 5

2.4 Autoencoder 6

2.5 Denoising Autoencoders 7

2.6 Variational Autoencoders 8

2.7 Representation Learning 9

3 Methodology 11

3.1 Experimental Design 11

3.1.1 Data 11

3.1.2 Evaluation 12

4 Proposed Solutions 13

4.1 Baseline 13

4.1.1 Experiment Setup 13

4.1.2 Result 14

4.1.3 Discussion 15

4.2 Utilizing unpaired data 15

4.2.1 Experiment Setup 17

4.2.2 Results 18

4.2.3 Discussion 18

4.3 Autoencoder-Inspired Method 19

4.3.1 Experiment Setup 19
4.3.2 Results 20
4.3.3 Discussion 21
4.4 Mapping Enrichment Using Random Samples 22
4.4.1 Experiment Setup 22
4.4.2 Results 23
4.4.3 Discussion 24

5 Conclusion **25**
5.1 Future Work 25
5.2 Reflections 26

Bibliography **27**

List of Tables

Table **Page**

4.1 Accuracy Top 18

4.2 Accuracy Bottom 18

4.3 Testing MSE of Algorithm 4, corresponding to Figure 4.7. The first row is the testing MSE right after we finished training stage 1 for 300 epochs. The second row is the testing MSE right after we finished training stage 1 for 2 iterations. The third row is the testing MSE right after we finished training Algorithm 4 21

4.4 Reconstruction Error 21

4.5 Encoding error (Mean Sqaured Error) 22

4.6 Encoding error (Mean Absolute Percentage Error) 22

4.7 Testing MSE of Algorithm 1, corresponding to Figure 4.7 24

4.8 Reconstruction Error 24

4.9 Encoding error (Mean Sqaured Error) 24

4.10 Encoding error (Mean Absolute Percentage Error) 24

LIST OF FIGURES

FIGURE **Page**

1.1 Examples of paired data (left) and unpaired data (right). X is the front view, and Y is the side view. [1] 2

1.2 Examples of paired data. X is the transmitter positions, and Y is the radiation patterns. [2] 3

2.1 An example of a feed forward neural network 6

2.2 The general structure of an autoencoder. g maps an input x to the internal represen-taion c, while $g^{\dagger}$ maps c to the output x', i.e. the reconstruction 7

2.3 Example of an architecture of an autoencoder 8

2.4 For variational autoencoders, the encoder model is sometimes referred to as the recognition model whereas the decoder model is sometimes referred to as the generative model [3] 9

2.5 VAE as as generative model illustration [3]. 9

3.1 Examples of MNIST images from 0 to 9. 11

3.2 Examples of top halves of MNIST images from 0 to 9. 12

3.3 Examples of bottom halves of MNIST images from 0 to 9. 12

3.4 Three images with different MSE compared to the ground truth: Left (ground truth), Middle (MSE: 0.005), Right (MSE: 0.173) 12

4.1 f and $f^{\dagger}$ 14

4.2 Original testing images (top) versus reconstructed images using the baseline approach (bottom) 15

4.3 Neural Network Blocks Configuration 1. Training the top and bottom neural networks require Xs and Ys that are associated with each other. However, the C that arises from X, and the C that arises from Y are identical. 16

4.4 Neural Network Blocks Configuration 2. We can use Xs and Ys that are independent of each other to train the left and the right neural networks. However, the C that arises on the left and the C that arises on the right are not identical. 16

4.5 Networks Task Visualization 17

4.6 Testing MSE 18
4.7 Examples of reconstructed images using Algorithm 4 21

Chapter 1

Introduction

Since 2006, the availability of large datasets, the exponential growth in computer power, and advances in machine learning algorithms have lead to successes in various tasks such as regression, classification, clustering, or dimensionality reduction. Consequently, machine learning algorithms has powered parts of people's daily life. Applications of machine learning can be seen in fraud detection, credit scores, image and speech recognition, etc.

1.1 Motivation

A classic problem in machine learning is given two random variables X and Y, you want to predict a variable X from Y as well as Y from X. The application of this problem can be seen from various domains. In computer vision, an interesting scenario is predicting one view of an image from another view. For example, one random variable is the front view of an image while the other random variable is the side view of that image. Such a task can be beneficial to several problems such as self-driving cars or face frontalization [4] [5] [4]. In radio engineering, we might be interested in predicting the positions of transmitters X from radiation patterns Y.

Moreover, collecting paired data, i.e. X and Y that are associated with each other, might be a time-consuming task. On the other hand, generating only Xs and Ys individually, i.e. unpaired data, can be an easier task. For instance, in radio engineering, while collecting the radiation pattern associated with the positions of transmitters can be expensive, generating the patterns and positions separately might be easier.

In this Major Qualifying Project report we explore various algorithms to improve the predictability between two random variables. For example, a classic approach for mapping a set of random variables onto themselves by learning a linear transformation that projects the original data into another space where vectors of projections are defined by the variance of the data is called principal component analysis (PCA) [6]. Similarly, canonical correlation analysis (CCA) [7] [8], a related method to PCA, defines coordinate systems that optimally describe the cross-variance between two sets of random variables.

The above-mentioned examples are linear methods. However, it is much more interesting to consider nonlinear methods such as neural networks. A classic example, and a close cousin to PCA about would be an Autoencoder. Accordingly, here we wish to generalize an Autoencoder (which goes from X to itself) to a more general paradigm where X goes to Y and vice versa. We investigate various paradigms that apply Autoencoders to our task and study their performance.

The first approach one might think of would be having a function f that maps X to Y, and a pseudo-inverse function $f^\dagger$ that goes from Y to X. We can define f and $f^\dagger$ as two neural networks. This approach would be our baseline.

Moreover, we are interested in scenarios where we have an abundance of unpaired data, which can be easier to collect than paired data. Hence, in addition to investigating various algorithms to improve the predictability between two random variables, we explore how to utilize unpaired data for our problem.

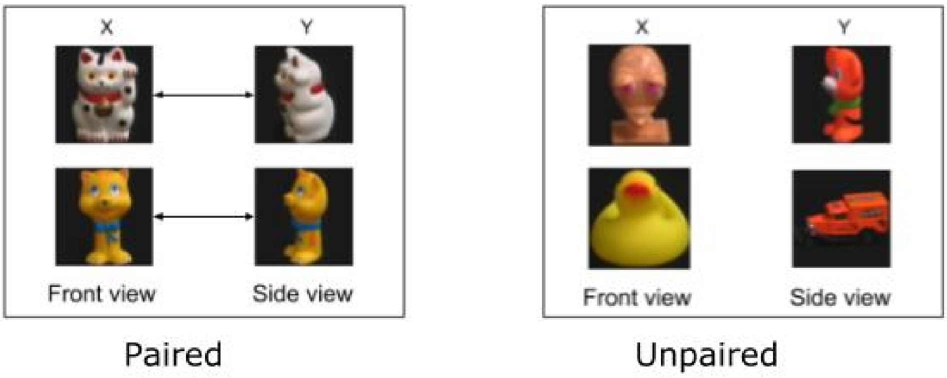

Figure 1.1: *Examples of paired data (left) and unpaired data (right). X is the front view, and Y is the side view. [1]*

In figure 1.1, notice on the left that we have the two views of the same object, which makes the data paired. We want to predict the right images from the left. On the right, we do not want to predict the truck from the duck. The goal is to predict the corresponding views of the duck and truck, i.e. predicting the truck from the truck or the duck from the duck. However, we are not provided with the corresponding views of the duck or the truck.

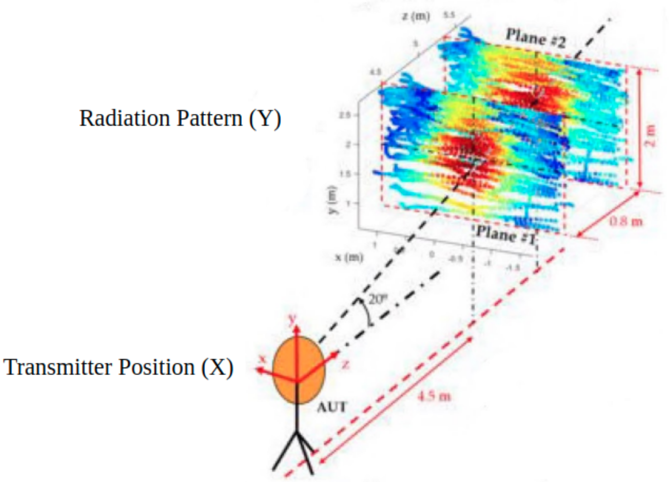

Figure 1.2: *Examples of paired data. X is the transmitter positions, and Y is the radiation patterns. [2]*

Chapter 2

Background

This section provides notation definitions as well as backgrounds on the techniques and algorithms that our work was built on or taken inspiration from.

2.1 Notation and Definitions

Notation	**Definition** *italic*
y, $\hat{y}$	Scalars are non-bold and *italic*
$\mathbf{z}$, $\mathbf{w}$	Vectors are lower-case and **bold**
$\mathbf{X}$	Matrices are UPPER-CASE and **bold**
$\mathbf{X}^T$	T indicates matrix transpose
x_{ij}	Matrix elements are written as scalars
$x_{i,:}$	Indicates rows of a matrix
$x_{:,j}$	Indicates columns of a matrix
n	Numbers of examples
m	Number of features

2.2 Canonical Correlation Analysis

Canonical correlation analysis (CCA) is a method for exploring the linear relationships between two multivariate sets of variables. For instance, let us consider variables measured on environmental health and environmental toxins. Several environmental health variables such as frequencies of sensitive species, species diversity, total biomass, the productivity of the environment, etc. may be measured and a second set of variables on environmental toxins are measured, such as the concentrations of heavy metals, pesticides, dioxin, etc.

CCA finds two bases, one for each variable, that are optimal with respect to correlations and, at the same time, it finds the corresponding correlations. In other words, it finds the two bases in which the correlation matrix between the variables is diagonal and the correlations on the diagonal are maximized. The dimensionality of these new bases is equal to or less than the smallest dimensionality of the two variables.

Let $(x_1, x_2) \in \mathbb{R}^{n_1} \times \mathbb{R}^{n_2}$ denote random vectors with covariances $(\Sigma_{11}, \Sigma_{22})$ and cross-covariance Σ_{12}. The goal of CCA is to find pairs of linear projections of the two views, $(w'_1 X_1, w'_2 X_2)$ that are maximally correlated:

$$(w_1^*, w_2^*) = \underset{w_1, w_2}{\operatorname{argmin}} \operatorname{corr}(w'_1 X_1, w'_2 X_2) = \underset{w_1, w_2}{\operatorname{argmin}} \frac{w'_1 \Sigma_{12} w_2}{\sqrt{w'_1 \Sigma_{11} w_1 w'_2 \Sigma_{22} w_2}}, \tag{2.1}$$

As the objective is invariant to scaling of w_1 and w_2, the projections are constrained to have unit variance:

$$(w_1^*, w_2^*) = \underset{w_1, w_2}{\operatorname{argmin}} \operatorname{corr}(w'_1 X_1, w'_2 X_2) \tag{2.2}$$

When finding multiple pairs of vectors (w_1^i, w_2^i), subsequent projections are also constrained to be uncorrelated with previous ones, that is $w_1^i \Sigma_1 1 w_1^j = w_2^i \Sigma_2 2 w_2^j$ for $i < j$. Assembling the top k projections veectors w_1^i into the columns of a matrix $A_1 \in \mathbb{R}^{n_1 \times k}$, and similarly placing w_2^i into $A_2 \in \mathbb{R}^{n_2 \times k}$, we obtain the following formulation to identify the top $k \leq min(n_1, n_2)$ projections:

$$\begin{aligned} \text{maximize:} \quad & tr(A'_1 \sum_{12} A_2) \\ \text{subject to:} \quad & A'_1 \sum_{11} A_1 = A'_2 \sum_{22} A_2 = I \end{aligned} \tag{2.3}$$

Canonical Correlation Analysis allows us to summarize the relationships into a lesser number of statistics while preserving the main facets of the relationships. In a way, the motivation for canonical correlation is very similar to principal component analysis. It is another dimension reduction technique.

2.3 Deep Neural Networks

Deep neural networks (DNNs) [9], also known as deep feedforward networks or multilayer perceptrons are systems that are composed of connected neurons that work in unison to solve specific tasks. They are a subset of machine learning and the key to deep learning algorithms. For instance, let f^* be a classifier function that maps input x to a class or category y, i.e. $y = f^*(x)$. A deep neural network becomes the best approximation of f by defining a mapping $y = f(x; \theta)$ and learning the value of parameters θ through a set of observations. These models are called "feedforward" because information goes through the function evaluated from x to the intermediate computations used to define f, and to the output y. They are also called "networks" as they are typically composed of many different functions. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain in order to form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. In this case, $f^{(1)}$ is called the first layer of the neural network. $f^{(2)}$ is the second layer and $f^{(3)}$ is the third or last layer. These chain structures are the most common structures used for neural networks. The number of functions that f consists of, i.e. the length of the chain, is the depth of the model. Training a neural network involves minimizing the error, i.e. minimizing the loss. During this process, we drive $f(x)$ to match $f^*(x)$, often through an algorithm called Gradient Descent.

Deep neural networks are called "neural" because they are inspired by neuroscience. In particular, they are inspired by the human brain, mimicking the way that biological neurons signal to one another. Since each hidden layer of a neural network is normally vector-valued. Every element of the vector can be interpreted as a neuron. Hence, instead of thinking of each layer as representing a single vector-to-vector function, we can think of the layer as consisting of many nodes act in parallel.

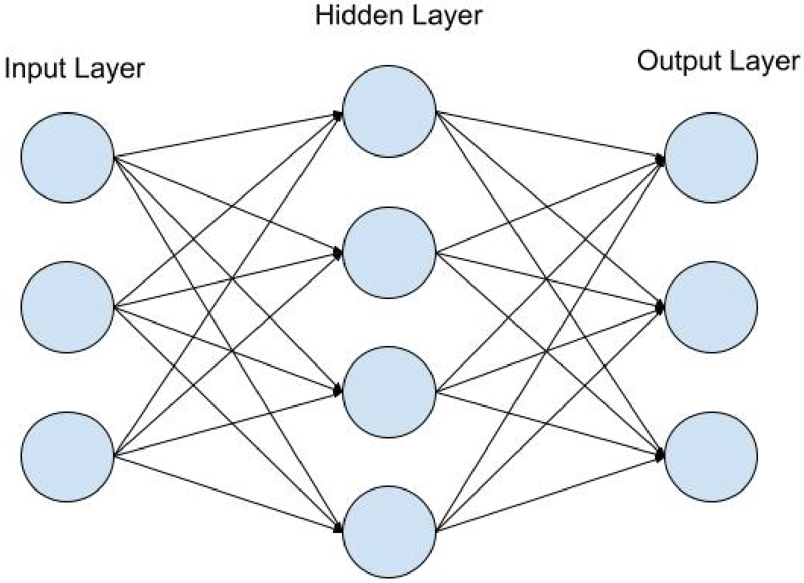

Figure 2.1: *An example of a feed forward neural network*

2.4 Autoencoder

The idea of autoencoders dates back to decades ago. They are first introduced in the 1980s by Hinton and the PDP group [10] to address the problem of "backpropagation without a teacher".

More recently, autoencoders have adopted deeper architectures [11] [12] [13] [14] [9].

An autoencoder is a type of neural network that is trained in order to copy its input to its output. It is a learning technique in which we leverage neural networks for the task of representation learning. Specifically, we will design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input. If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data, i.e. correlations between input features, this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck. Internally, it has a hidden layer h that can represent the input through a "code". The autoencoder can be viewed as consisting of two parts: an encoder and a decoder. The encoder is a function $c = g(x)$ and the decoder is a function that produces the reconstruction x’ $= g^{\dagger}(c)$. Normally, autoencoders are restricted in ways that only allow them to copy approximately, and to copy only input that resembles the training data. Since the model can prioritize which aspects of the input are copied, so it can learn useful information of the data. In addition to being used for dimensionality reduction or feature learning purposes, autoencoders have been used for generative modeling as well.

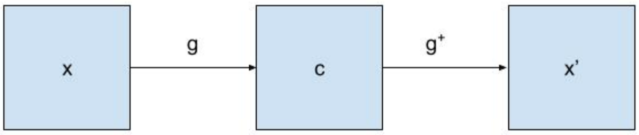

Figure 2.2: *The general structure of an autoencoder. g maps an input x to the internal represen-taion c, while $g^{\dagger}$ maps c to the output x’, i.e. the reconstruction*

2.5 Denoising Autoencoders

The denoising autoencoder (DAE) [9] [15] is autoencoder that receives a corrupted data point as input and is trained to predict the original, uncorrupted data point as its output. Traditionally, autoencoders are trained to minimize some objective function

$$L(x, g^{\dagger}(g(x))) \tag{2.4}$$

where L is a loss function penalizing $g^{\dagger}(g(x))$ for being dissimilar from x, i.e. the L^2 norm of their difference for example. This would encourage $g^{\dagger} \circ g$ to learn to become the identity function, given the capacity to do so.

For denoising autoencoders, we would minimize the following instead

$$L(x, g^{\dagger}(g(\tilde{x}))) \tag{2.5}$$

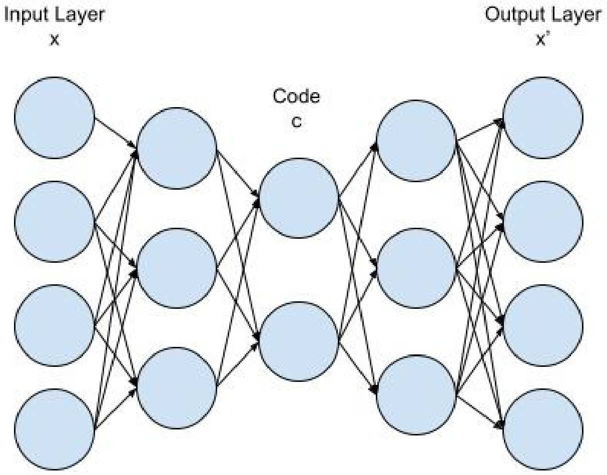

Figure 2.3: *Example of an architecture of an autoencoder*

where $\tilde{x}$ is a corrupted copy of x. This corruption is caused by some form or noise. Denoising autoencoders aim to remove this noise, i.e. undo this corruption, rather than simply copying their input.

2.6 Variational Autoencoders

A variational autoencoder (VAE) [16] provides a probabilistic manner for describing an observation in latent space. Thus, rather than building an encoder that outputs a single value to describe each latent state attribute, we will formulate our encoder to describe a probability distribution for each latent attribute.

By constructing our encoder model to output a range of possible values (a statistical distribution) from which we'll randomly sample to feed into our decoder model, we're essentially enforcing a continuous, smooth latent space representation. For any sampling of the latent distributions, we're expecting our decoder model to be able to accurately reconstruct the input. Thus, values that are nearby to one another in latent space should correspond with very similar reconstructions.

The loss of the variational autoencoder consist of two terms. One penalizes the reconstruction loss, while the other one encourages the learned distribution of the encoder to be similar to a true prior distribution $p(c)$, which we will assume to follow a unit Gaussian distribution, i.e. $\mathcal{N}(0,\mathbf{I})$.

$$\frac{1}{n}\sum_{i=1}^{n}\left\|\mathbf{x}^{(i)}-g^{\dagger}(g(\mathbf{x}^{(i)}))\right\|_2^2+D_{KL}(g(\mathbf{x}),\mathcal{N}(0,\mathbf{I})) \tag{2.6}$$

where D_{KL} is the Kullback-Leibler divergence.

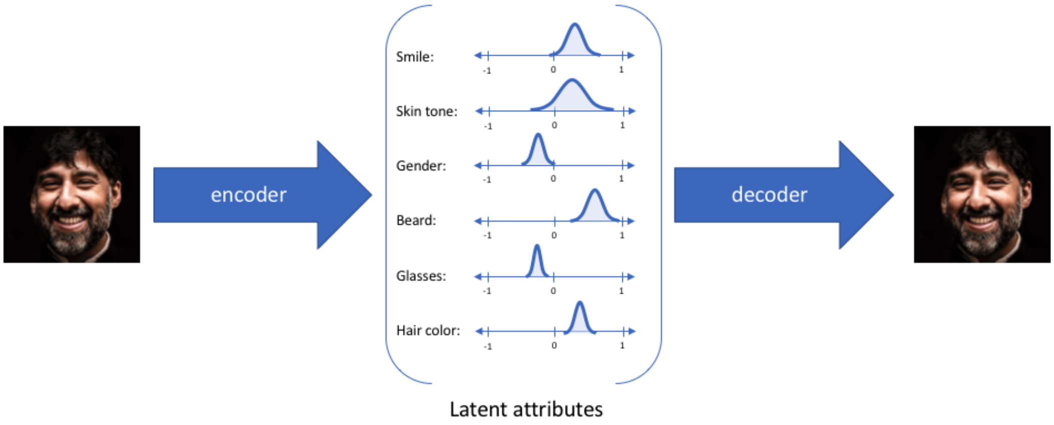

Figure 2.4: *For variational autoencoders, the encoder model is sometimes referred to as the recognition model whereas the decoder model is sometimes referred to as the generative model [3]*

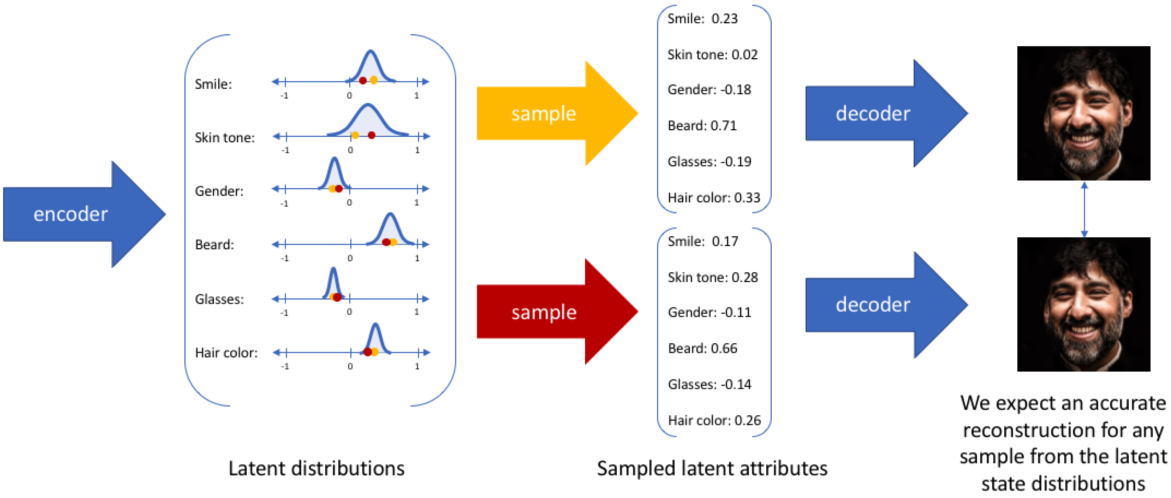

Figure 2.5: *VAE as as generative model illustration [3].*

2.7 Representation Learning

Representation learning [17] is learning representations of input data typically by transforming it or extracting features from it, which makes it easier to perform a task like classification or prediction. There are various ways of learning representations. For instance, in the case of probabilistic models, the goal is to learn a representation that captures the probability distribution of underlying explanatory features for the observed input. Such a learned representation can then be used for prediction. In deep learning, the representation is formed by the composition of multiple non-linear transformations of the input data to yield abstract and useful representations for tasks like classification, prediction, etc.

Focusing specifically on deep learning, representation learning is the consequence of the function a model learns where the learning is captured in the parameters of the model, as the function transforms input to output, during training. Representation learning here is referring to the nature/characteristics of the transformed input - not the model parameters/ function that is causal to it. The casual role is played both by the architecture of the model, and the learned parameters (e.g. does a parameter play a role in representing part or all of the input, etc.) in mapping input to output.

CHAPTER 3

METHODOLOGY

This section discusses the dataset we used for our experiments and the evaluation metric we utilized to measure the performances of our neural networks.

3.1 Experimental Design

3.1.1 Data

Throughout different stages of this project, we used the MNIST (Modified National Insitute of Standards and Technology) dataset [18] of handwritten digits that is commonly used for training image processing systems. It includes 70,000 small square 28x28 pixel grayscale images of handwritten single digits from 0 to 9. The dataset is split into a training set, which consists of 60,000 images, and a testing set, which consists of 10,000 images.

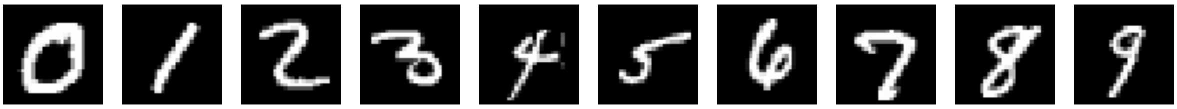

Figure 3.1: *Examples of MNIST images from 0 to 9.*

As a simple example of our desire to transform X to Y, we we slice the images horizontally and wish to train a neural network to predict the top halves from the bottom halves and vice versa. We also scaled the pixel values to range from -1 to 1. We chose to use MNIST since it is a free dataset with tens of thousands of images. Secondly, since it is an image dataset, the evaluation can be easily defined, i.e. we can use mean squared error (MSE), root mean squared error (RMSE), binary cross-entropy, etc. Lastly, as the dimension of an MNIST image is 28×28, it is faster to train our neural networks.

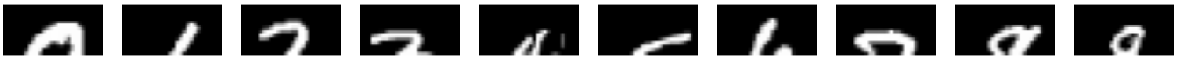

Figure 3.2: *Examples of top halves of MNIST images from 0 to 9.*

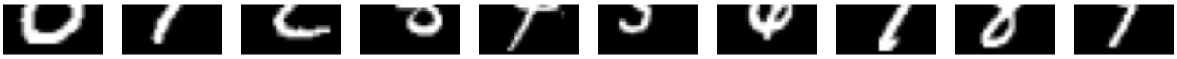

Figure 3.3: *Examples of bottom halves of MNIST images from 0 to 9.*

3.1.2 Evaluation

Once we finished training our neural networks, we would need to compare the images that our models produced with the original images, i.e, the ground truths, and assess their visual quality. The two metrics we used were Mean Square Error (MSE).

In order to compute the MSE, we took the average of the square of the difference in pixel values between the actual and the predicted images. In some cases, the MSE alone does not tell us anything about the visual quality of the output images. However, for our problem, we did observe a negative correlation between the MSE and the quality of the predicted images, i.e. the lower the MSE, the better the image quality. Hence, we decided to use MSE as one of our metrics.

$$MSE = \frac{1}{n} \sum_{n=1}^{n} (y_i - \widetilde{y_i})^2 \tag{3.1}$$

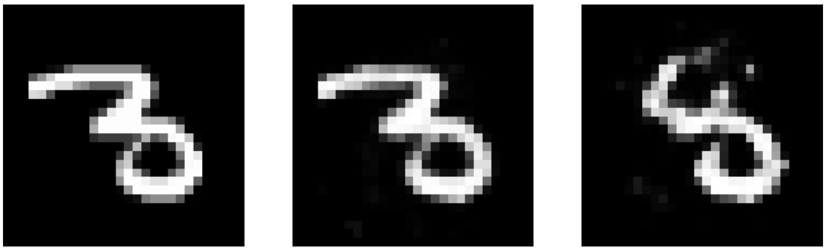

Figure 3.4: *Three images with different MSE compared to the ground truth: Left (ground truth), Middle (MSE: 0.005), Right (MSE: 0.173)*

CHAPTER 4

PROPOSED SOLUTIONS

In this chapter, we propose several solutions to our problem. These algorithms took inspiration from both canonical correlation analysis and autoencoders. Additionally, some are based on the performances of others. Besides describing each approach in detail, we discuss how we came up with the original idea as well as analyzed its performances.

4.1 Baseline

One simple baseline solution to this problem is training directly two neural networks to predict the variable Y from X as well as X from Y. We defined two neural networks $f : \mathbb{R}^m \to \mathbb{R}^m$ and $f^\dagger : \mathbb{R}^m \to \mathbb{R}^m$. We want f to be able to take in X and predict Y, while $f^\dagger$ performs the opposite task. For the baseline approach, we have access to 10,000 paired MNIST halves, i.e. 10,000 top halves and the corresponding 10,000 bottom halves.

4.1.1 Experiment Setup

For our experiment, we sliced MNIST images into top halves and bottom halves. The dimension of an MNIST image is 28×28, so the dimension of the top and bottom half is 14×28. Afterward, we reshaped the top and bottom halves to have a dimension of 392×1. We let the variable X and Y be the top and bottom halves respectively. Hence, $m = 392$ and in our implementations, f and $f^\dagger$ with architectures shown below:

$f : FCN(392) \to LeakyRelu \to FCN(256) \to LeakyRelu \to FCN(100) \to Sigmoid \to FCN(100) \to LeakyRelu \to FCN(256) \to LeakyRelu \to FCN(392) \to Tanh$

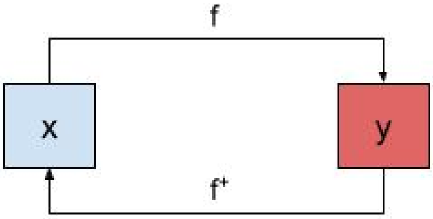

Figure 4.1: *f and $f^{\dagger}$*

$f^{\dagger}: FCN(392) \rightarrow LeakyRelu \rightarrow FCN(256) \rightarrow LeakyRelu \rightarrow FCN(100) \rightarrow Sigmoid \rightarrow FCN(100) \rightarrow LeakyRelu \rightarrow FCN(256) \rightarrow LeakyRelu \rightarrow FCN(392) \rightarrow Tanh$

LeakyRelu is defined as $max(0.1\mathbf{x}, \mathbf{x})$

We want f to be able to map the top halves of MNIST images to the bottom halves, and $f^{\dagger}$ to perform the opposite task. Hence, we train them to minimize equation 4.1. This equation computes the average squared pixel difference between the predicted image and the ground truth one.

$$L_{baseline}(\theta^{f}, \theta^{f^{\dagger}}) = \frac{1}{n} \sum_{i=1}^{n} \left\| f(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)} \right\|_2^2 + \left\| f^{\dagger}(\mathbf{y}^{(i)}) - \mathbf{x}^{(i)} \right\|_2^2 \tag{4.1}$$

We trained both networks for 1,000 epochs on 10,000 MNIST images using opt optimizer and a learning rate of lr. Finally, we computed the testing MSE at every epoch and reported the best one. We varied $lr \in \{0.001, 0.0001\}$, and $opt \in \{SGD, Adam, AdaGrad\}$.

Algorithm 1: Baseline

```
for number of training iterations do
    • Sample minibatch of n images, and sliced them horizontally into top halves
      $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n)}\}$ and bottom halves $\{\mathbf{y}^{(1)}, ..., \mathbf{y}^{(n)}\}$.
    $\theta^{f} \leftarrow \theta^{f} - \alpha \times \nabla_{\theta^{f}\, baseline}(\theta^{f}, \theta^{f^{\dagger}})$
    $\theta^{f^{\dagger}} \leftarrow \theta^{f^{\dagger}} - \alpha \times \nabla_{\theta^{f^{\dagger}}} L_{baseline}(\theta^{f}, \theta^{f^{\dagger}})$
end
```

4.1.2 Result

The best recorded testing MSE is 0.1357.

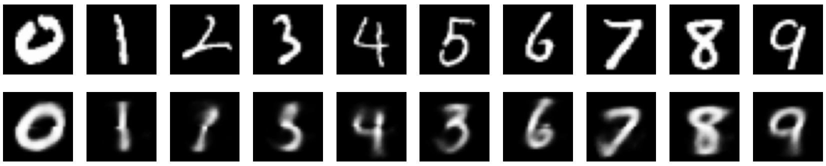

Figure 4.2: *Original testing images (top) versus reconstructed images using the baseline approach (bottom)*

4.1.3 Discussion

One interesting question is whether the bad reconstruction loss described above is because (a) for the baseline method, there exists no such mapping (i.e., no instantiation of the weights) from X to Y or from Y to X that delivers good accuracy; or because (b) such a mapping does exist but it is unlikely to be found by a search procedure such as SGD [19], Adam [20], or Adagrad [21]. One example of where (a) could be true is if Y is statistically independent of X; in that case, it is obvious that no good mapping can exist. However, given that in the MNIST problem there is obviously a strong dependency between the top and bottom halves of the image, we suspect this is not the case.

4.2 Utilizing unpaired data

Our goal is to do better than just training directly two neural networks. We are inspired by canonical analysis to think about our problem by cutting the neural networks f and $f^\dagger$. The "canonical" in CCA means we have a special space, **c**, where we can get by taking f or $f^\dagger$ and indentifying one of its layers as the canonical space. This would split f and $f^\dagger$ each into two neural networks. f is split into $h^\dagger$ and g (i.e. $f = h^\dagger \circ g$), while $f^\dagger$ is split into h and $g^\dagger$ (i.e. $f^\dagger = h \circ g^\dagger$). We defined neural networks $g : \mathbb{R}^m \to \mathbb{R}^d$, $g^\dagger : \mathbb{R}^d \to \mathbb{R}^m$, $h : \mathbb{R}^m \to \mathbb{R}^d$, and $h^\dagger : \mathbb{R}^d \to \mathbb{R}^m$ ($m > d$). We want f to be able to take in X and predict Y, while $f^\dagger$ performs the opposite task.

Given f and $f^\dagger$ have the same architectures, we choose the layer to slice f similar to that we choose to slice $f^\dagger$. Hence, from figure 4.3, we can see that all Cs are the same. This gives us the intuition that these four networks $g^\dagger$, h, g, and $h^\dagger$ are like LEGO blocks that can be switched and connected in various ways. Figure 4.3 is the baseline configuration that shows that f consists of g, which maps X to C, and $h^\dagger$, which maps C to Y; $f^\dagger$ consists of h, which maps Y to C, and $g^\dagger$, which maps C to X.

We can also rearrange the neural networks as figure 4.4. Now the new configuration shows that g maps X to C and $g^\dagger$ maps C to X, while h maps Y to C and h† maps C to Y. We can easily notice that this configuration in fact consists of 2 autoencoders trained on X and Y respectively. Training a forward mapping from X to Y and a backward mapping from Y to X "magically" gives us autoencoders. This raises the question: Can we utilize the abundance of unpaired data to perform pre-training on g and $g^\dagger$ as well as h and $h^\dagger$ as two autoencoders?

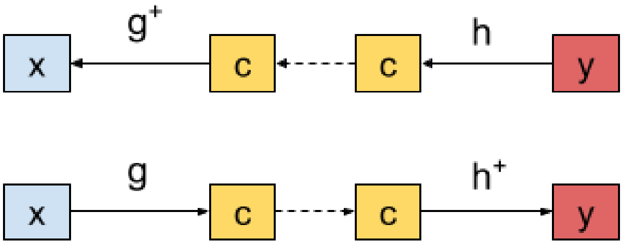

Figure 4.3: *Neural Network Blocks Configuration 1. Training the top and bottom neural networks require Xs and Ys that are associated with each other. However, the C that arises from X, and the C that arises from Y are identical.*

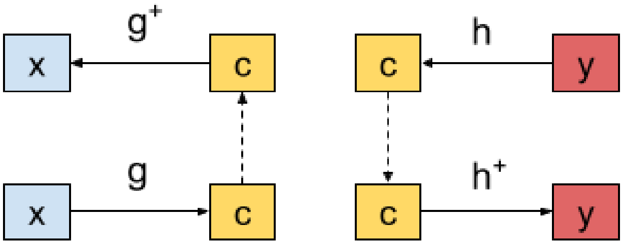

Figure 4.4: *Neural Network Blocks Configuration 2. We can use Xs and Ys that are independent of each other to train the left and the right neural networks. However, the C that arises on the left and the C that arises on the right are not identical.*

Before training f and $f^\dagger$ to map the variable X to Y and Y to X respectively, we propose to train g and $g^\dagger$ as well as h and $h^\dagger$ as autoencoders. g and $g^\dagger$ are trained using the variable X, while h and $h^\dagger$ are trained on the variable Y. Notice that during the pre-training phase the first autoencoder is only trained on the variable X, and the second autoencoder is trained on the variable Y. Hence, we do not need to sample X and Y that are dependent on each other, i.e. we can use X and Y that are collected independently for unsupervised pre-training, which can be easier to do than collecting paired X - Y.

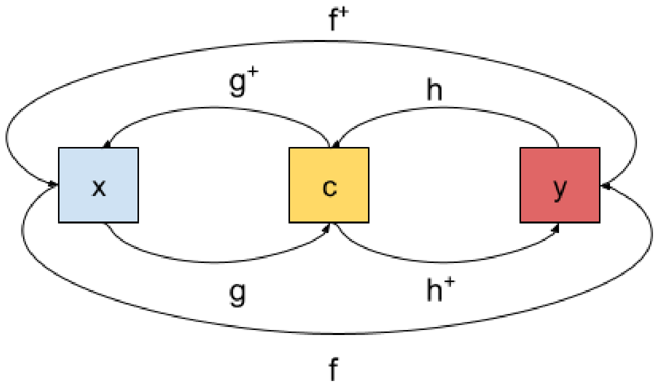

Figure 4.5: *Networks Task Visualization*

4.2.1 Experiment Setup

Let us denote the unpaired data as $\mathbf{x}_{unpaired}$ and $\mathbf{y}_{unpaired}$. We began by training g, $g^{\dagger}$, h, and $h^{\dagger}$ to optimize equation 4.2. We trained those networks on 60,000 images (60,000 top halves and 60,000 bottom halves) using Adam with a learning rate of 0.0001 for *pre_training* epochs, where *pre_training* $\in \{25, 50, 100, 200\}$

$$L_1(\theta^g, \theta^{g^{\dagger}}, \theta^h, \theta^{h^{\dagger}}) = \frac{1}{n}\sum_{i=1}^{n} \left\| g^{\dagger}(g(\mathbf{x}^{(i)}_{unpaired})) - \mathbf{x}^{(i)}_{unpaired} \right\|_2^2 + \left\| h^{\dagger}(h(\mathbf{y}^{(i)}_{unpaired})) - \mathbf{y}^{(i)}_{unpaired} \right\|_2^2 \tag{4.2}$$

Afterward, we trained f and $f^{\dagger}$ similar to the baseline method in the previous section for 150 epochs with a learning rate of 0.0001 using Adam.

Algorithm 2: Unsupervised Pre-training

```
for number of training iterations for unsupervised pre-training do
    • Sample minibatch of n top halves $\{\mathbf{x}^{(1)}_{unpaired}, ..., \mathbf{x}^{(n)}_{unpaired}\}$ and bottom halves
      $\{\mathbf{y}^{(1)}_{unpaired}, ..., \mathbf{y}^{(n)}_{unpaired}\}$.
    $\theta^z \leftarrow \theta^z - \alpha \times \nabla_{\theta^z} L_1(\theta^z, \theta^{z^{\dagger}}))$
    $\theta^z \leftarrow \theta^z - \alpha \times \nabla_{\theta^{z^{\dagger}}} L_1(\theta^{z^{\dagger}}, \theta^{z^{\dagger}}))$
end
for number of training iterations do
    • Sample minibatch of n images, and sliced them vertically into top halves
      $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(n)}\}$ and bottom halves $\{\mathbf{y}^{(1)}, ..., \mathbf{y}^{(n)}\}$.
    $\theta^f \leftarrow \theta^f - \alpha \times \nabla_{\theta^f baseline}(\theta^f, \theta^{f^{\dagger}})$
    $\theta^{f^{\dagger}} \leftarrow \theta^{f^{\dagger}} - \alpha \times \nabla_{\theta^{f^{\dagger}}} L_{baseline}(\theta^f, \theta^{f^{\dagger}})$
end
```

4.2.2 Results

Figure 4.6 shows that all models converge to a testing MSE of approximately 0.28.

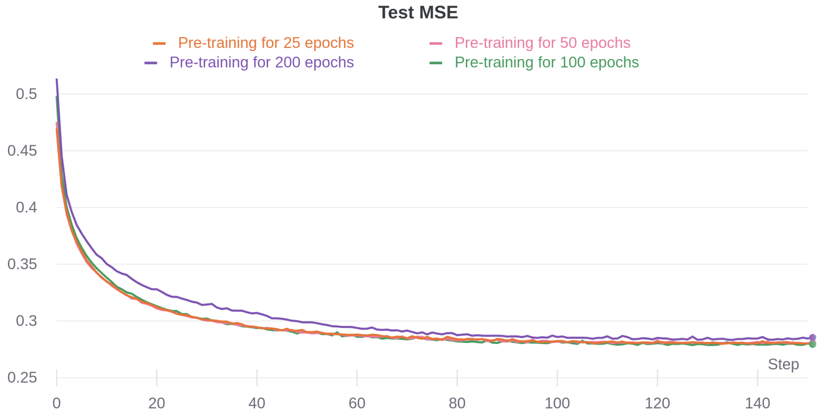

Figure 4.6: *Testing MSE*

4.2.3 Discussion

Unsupervised pre-training hurts the performance of our models, yielding a 79% increase in terms of testing MSE. We conducted a sub experiment to see why there is such a surge in MSE. We trained a separate neural network to predict the digit from the encoded representation $h(\mathbf{y})$ or $g(\mathbf{x})$ at different stages of the algorithm.

	Accuracy
Baseline for 400 epochs	0.9268
Pretrain 200 + Baseline 400	0.9172
Pretrain 1000 + Baseline 400	0.9131

Table 4.1: *Accuracy Top*

	Accuracy
Baseline for 400 epochs	0.8945
Pretrain 200 + Baseline 400	0.8901
Pretrain 1000 + Baseline 400	0.8852

Table 4.2: *Accuracy Bottom*

The first row of table 4.2 and 4.1 shows the accuracy when training a separate neural network to predict $h(\mathbf{y})$ and $g(\mathbf{x})$ after training only stage 2 of Algorithm 2 for 400 epochs. Row 2 and row 3 add unsupervised pre-training as a prior step, both of which yield a decrease in accuracy. The reason for this is that by adding an unsupervised pre-training step, the encoded representations might lose information and therefore result in reduced accuracy.

4.3 Autoencoder-Inspired Method

In figure 4.3, to train the top and bottom neural networks, we will need the Xs and the Ys that are associated with each other. However, in figure 4.4, we can use Xs and Ys that are independent of each other. Canonical correlation analysis teaches us that C is important. In figure 4.3, we compute the C from X and Y. The C that arises from X and the C that arises from Y are identical, but we would need paired data for it. While in figure 4.4, if we take the autoencoders perspective, we can compute C by just X or Y. The C that C that arises on the left, and the C that arises on the right are not connected anymore. Nonetheless, we can take advantage of the abundance of unpaired data for this approach. This raises a question: "How we can reconnect the C in a way?"

From an experiment that shows that the top and bottom of an image can be well reconstructed from an information-rich representation, i.e. a representation that contains information of both the top and bottom of an image. We investigate if an autoencoder might be used as a way of "bootstrapping" the training process of the map from X to Y, and vice versa. We defined an encoder $z:\mathbb{R}^{2m}\to\mathbb{R}^d$, $z^\dagger:\mathbb{R}^d\to\mathbb{R}^{2m}$, $g:\mathbb{R}^m\to\mathbb{R}^d$, $g^\dagger:\mathbb{R}^d\to\mathbb{R}^m$, $h:\mathbb{R}^m\to\mathbb{R}^d$, and $h^\dagger:\mathbb{R}^d\to\mathbb{R}^m$. We want to train z and $z^\dagger$ as an autoencoder so that z can take in both the variable X as well as the variable Y to encode a hidden representation that is information-rich. We want g and h to map X and Y respectively to the same rich-information representation produced by z.

4.3.1 Experiment Setup

The architectures of h, $h^\dagger$, g, and $g^\dagger$ are keps the same as the previous sections. The architectures of z and $z^\dagger$ are:

$$z: FCN(784)\to LeakyRelu\to FCN(256)\to LeakyRelu\to FCN(100)\to Sigmoid$$
$$z^\dagger: FCN(100)\to LeakyRelu\to FCN(256)\to LeakyRelu\to FCN(784)\to Tanh$$

We first trained z and $z^\dagger$ as an autoencoder using the objective function 4.3. This objective function lets z learn an "information-rich" C.

$$L_1(\theta^z,\theta^{z^\dagger}) = \frac{1}{n}\sum_{i=1}^{n}\left\|z^\dagger(z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})) - (\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 \tag{4.3}$$

We then fixed parameters of z and trained encoders g and h following 4.4. We want to train g and h to encode the top (X) and bottom of an image (Y) respectively to a latent variable similar to when we encode the whole image (X,Y) with z. This second objective helps h and g approximate z.

$$L_2(\theta^g,\theta^h) = \frac{1}{n}\sum_{i=1}^{n}\left\|g(\mathbf{x}^{(i)}) - z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 + \frac{1}{n}\sum_{i=1}^{n}\left\|h(\mathbf{y}^{(i)}) - z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 \tag{4.4}$$

Finally, after freezing the parameters of g and h, we trained $g^\dagger$ and $h^\dagger$ with the objective function 4.5. The last objective function makes $g^\dagger$ and $h^\dagger$ learn to reconstruct the approximations of C encoded by g and h respectively.

$$L_3(\theta^{g^\dagger}, \theta^{h^\dagger}) = \frac{1}{n}\sum_{i=1}^{n} \left\| \mathbf{y}^{(i)} - h^\dagger(g(\mathbf{x}^{(i)})) \right\|_2^2 + \frac{1}{n}\sum_{i=1}^{n} \left\| \mathbf{x}^{(i)} - g^\dagger(h(\mathbf{y}^{(i)})) \right\|_2^2 \tag{4.5}$$

Algorithm 3: Autoencoder-Inpsired Training Algorithm

```
for number of training iterations for stage 1 do
    • Sample minibatch of n images, and sliced them horizontally into top halves
      {$\mathbf{x}^{(1)},...,\mathbf{x}^{(n)}$} and bottom halves {$\mathbf{y}^{(1)},...,\mathbf{y}^{(n)}$}.
    $\theta^z \leftarrow \theta^z - \alpha_1 \times \nabla_{\theta^z} L_1(\theta^z, \theta^{z^\dagger}))$
    $\theta^z \leftarrow \theta^z - \alpha_1 \times \nabla_{\theta^{z^\dagger}} L_1(\theta^{z^\dagger}, \theta^{z^\dagger}))$
end
Freeze parameters of $z$.
for number of training iterations for stage 2 do
    • Sample minibatch of n images, and sliced them vertically into top halves
      {$\mathbf{x}^{(1)},...,\mathbf{x}^{(n)}$} and bottom halves {$\mathbf{y}^{(1)},...,\mathbf{y}^{(n)}$}.
    $\theta^g \leftarrow \theta^g - \alpha_2 \times \nabla_{\theta^g} L_2(\theta^g, \theta^h))$
    $\theta^h \leftarrow \theta^h - \alpha_2 \times \nabla_{\theta^h} L_2(\theta^g, \theta^h))$
end
Freeze parameters of $g$ and $h$.
for number of training iterations for stage 3 do
    • Sample minibatch of n images, and sliced them vertically into top halves
      {$\mathbf{x}^{(1)},...,\mathbf{x}^{(n)}$} and bottom halves {$\mathbf{y}^{(1)},...,\mathbf{y}^{(n)}$}.
    $\theta^{g^\dagger} \leftarrow \theta^{g^\dagger} - \alpha_3 \times \nabla_{\theta^{g^\dagger}} L_3(\theta^{g^\dagger}, \theta^{h^\dagger}))$
    $\theta^{h^\dagger} \leftarrow \theta^{h^\dagger} - \alpha_3 \times \nabla_{\theta^{h^\dagger}} L_3(\theta^{g^\dagger}, \theta^{h^\dagger}))$
end
```

We trained stage 1 for 300 epochs, stage 2 for 300 epochs, and stage 3 for 2,000 epochs on 10,000 MNIST images. For all three stages, we trained all neural networks using Adam with a learning rate of 0.0001.

4.3.2 Results

After training, we achieved a MSE of 0.0143 (Figure 4.7 row 2), which showed that we can get a visually pleasing reconstruction of the top and bottom of MNIST images from $\mathbf{c}$ - the encoded representation of the whole MNIST images, i.e. $z(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) = \mathbf{c}$. In order to investigate the relationship between the encoded representation c and the prediction performance, we trained z and $z^\dagger$ from stage 1 only for 2 iterations while keeping the number of epochs of the other stages the same. This experiment resulted in a testing MSE of 0.0683 (Figure 4.7 row 3). This shows that the encoded representation c is crucial for our problem. An information-rich C that stores sufficient information of the top and bottom halves of the MNIST images can facilitate the prediction task. In contrast, an information-poor **c** will lead to a worse MSE. Our proposed algorithm yields a testing MSE of 0.1456, which is still higher than the baseline (MSE of 0.1369).

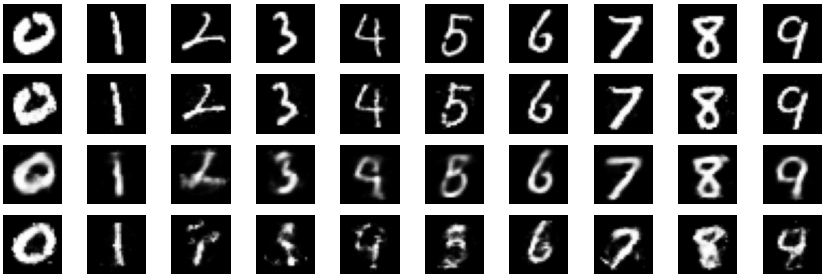

Figure 4.7: *Examples of reconstructed images using Algorithm 4*

	MSE
Reconstructing Images From Rich-Information C	0.0143
Reconstructing Images From Rich-Information C (training stage 1 for 2 iterations)	0.0683
Autoencoder-inspired	0.1456

Table 4.3: *Testing MSE of Algorithm 4, corresponding to Figure 4.7. The first row is the testing MSE right after we finished training stage 1 for 300 epochs. The second row is the testing MSE right after we finished training stage 1 for 2 iterations. The third row is the testing MSE right after we finished training Algorithm 4*

4.3.3 Discussion

In this section, we provide an analysis of the representations encoded by f,g and z. Let us denote the approximations of C using h and g as $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_1$ respectively, i.e. $h(y) = \tilde{\mathbf{c}}_1$ and $g(x) = \tilde{\mathbf{c}}_2$.

	Column 1: MSE($(g^{\dagger}(\tilde{\mathbf{c}}_1), h^{\dagger}(\tilde{\mathbf{c}}_2))$,$(\mathbf{x},\mathbf{y})$)	Column 2: MSE($(g^{\dagger}(\mathbf{c}), h^{\dagger}(\mathbf{c}))$,$(\mathbf{x},\mathbf{y})$)
Training	0.14604566	0.06882905
Testing	0.14566344	0.06830042

Table 4.4: *Reconstruction Error*

Column 1 in 4.4 reports mean squared error between the original and the reconstructed images from C in the training and testing data, while column 2 reports the same measurements but using $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_2$. We can see that using $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_2$, rather than **c**, to reconstruct the top and bottom of the images will lead to an increase of 117% and 131% in mean squared error on training and testing data respectively. Such an increase in MSE might result from the fact that $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_2$ are bad approximations of **c**.

	MSE($\mathbf{c}$,$\tilde{\mathbf{c}}_2$)	MSE($\mathbf{c}$,$\tilde{\mathbf{c}}_1$)
Training	0.017277006	0.015227813
Testing	0.017503608	0.015561547

Table 4.5: *Encoding error (Mean Sqaured Error)*

	MAPE($\mathbf{c}$,$\tilde{\mathbf{c}}_2$)	MAPE($\mathbf{c}$,$\tilde{\mathbf{c}}_1$)
Training	23.4239%	21.3456%
Testing	23.5342%	22.6087%

Table 4.6: *Encoding error (Mean Absolute Percentage Error)*

Having observed such a significant in MSE, one might think that $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_2$ are bad approximations of C. Table 3 reports the Mean Absolute Percentage Error between each pair of the encoded representation. Indeed, $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_2$ are bad approximations of C, yielding a MAPE of around 22% - 24%.

4.4 Mapping Enrichment Using Random Samples

In the previous section, we managed to show that our approximations of $\mathbf{c}$ using either the top or bottom half of an image, i.e. $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_2$ yielded a MAPE of 22% - 24% and a MSE of 0.145. Consequently, one idea comes up: we do not just want g and h to match z on just training data but on arbitrary data as well, which allows us to evaluate them anywhere. Hence, we proposed to make some modifications to the algorithm in the previous section by adding random data to stage 2 and stage 3. The only reason we are allowed to use this trick is due to the objective function of the second stage of the algorithm 4.

In this section, we investigate an approach to reduce the error both in terms of MAPE and MSE to see how it would affect the reconstruction error.

4.4.1 Experiment Setup

We propose to use Algorithm 4 with some small modifications to the loss functions 4.4 and 4.5. In particular, we replaced equation 4.4 with the following objective function:

$$L_4(\theta^g,\theta^h) = \frac{1}{n}\sum_{i=1}^{n} \left\|g(\mathbf{x}^{(i)}) - z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 + \left\|h(\mathbf{y}^{(i)}) - z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 + \left\|h(\mathbf{r}^{(i)}) - z(\mathbf{p}^{(i)},\mathbf{q}^{(i)})\right\|_2^2 \tag{4.6}$$

$$\nabla_{\theta^g,\theta^h}\frac{1}{n}\sum_{i=1}^{n} \left\|g(\mathbf{x}^{(i)}) - z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 + \left\|h(\mathbf{y}^{(i)}) - z(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\|_2^2 + \left\|h(\mathbf{r}^{(i)}) - z(\mathbf{p}^{(i)},\mathbf{q}^{(i)})\right\|_2^2 \tag{4.7}$$

Additionally, we replaced function 4.5 with the the loss function 4.10:

(4.8) $$L_{5\_1}(\theta^{g^\dagger},\theta^{h^\dagger}) = \frac{1}{n}\sum_{i=1}^{n}\left\|\mathbf{y}^{(i)} - h^\dagger(g(\mathbf{x}^{(i)}))\right\|_2^2 + \left\|\mathbf{x}^{(i)} - g^\dagger(h(\mathbf{y}^{(i)}))\right\|_2^2$$

(4.9) $$L_{5\_2}(\theta^{g^\dagger},\theta^{h^\dagger}) = \frac{1}{n}\sum_{i=1}^{n}\left\|\mathbf{r}^{(i)} - h^\dagger(g(\mathbf{p}^{(i)}))\right\|_2^2 + \left\|\mathbf{p}^{(i)} - g^\dagger(h(\mathbf{r}^{(i)}))\right\|_2^2$$

(4.10) $$L_5(\theta^{g^\dagger},\theta^{h^\dagger}) = L_{5\_1}(\theta^{g^\dagger},\theta^{h^\dagger}) + L_{5\_2}(\theta^{g^\dagger},\theta^{h^\dagger})$$

where **r**, **p** and **q** are random data drawn from a known data prior.

The algorithm can now be summarized as:

Algorithm 4: Autoencoder-Inpsired Training Algorithm

```
for number of training iterations for stage 1 do
    • Sample minibatch of n images, and sliced them vertically into top halves
      $\{\mathbf{x}^{(1)},...,\mathbf{x}^{(n)}\}$ and bottom halves $\{\mathbf{y}^{(1)},...,\mathbf{y}^{(n)}\}$.
    $\theta^z \leftarrow \theta^z - \alpha_1 \times \nabla_{\theta^z} L_1(\theta^z,\theta^{z^\dagger}))$
    $\theta^z \leftarrow \theta^z - \alpha_1 \times \nabla_{\theta^{z^\dagger}} L_1(\theta^{z^\dagger},\theta^{z^\dagger}))$
end
Freeze parameters of $z$.
for number of training iterations for stage 2 do
    • Sample minibatch of n images, and sliced them vertically into top halves
      $\{\mathbf{x}^{(1)},...,\mathbf{x}^{(n)}\}$ and bottom halves $\{\mathbf{y}^{(1)},...,\mathbf{y}^{(n)}\}$.
    • Sample minibatch of $3 \times n$ random vectors from a known data prior.
    $\theta^g \leftarrow \theta^g - \alpha_2 \times \nabla_{\theta^g} L_4(\theta^g,\theta^h))$
    $\theta^h \leftarrow \theta^h - \alpha_2 \times \nabla_{\theta^h} L_4(\theta^g,\theta^h))$
end
Freeze parameters of $g$ and $h$.
for number of training iterations for stage 3 do
    • Sample minibatch of n images, and sliced them vertically into top halves
      $\{\mathbf{x}^{(1)},...,\mathbf{x}^{(n)}\}$ and bottom halves $\{\mathbf{y}^{(1)},...,\mathbf{y}^{(n)}\}$.
    • Sample minibatch of $3 \times n$ random vectors from a known data prior.
    $\theta^{g^\dagger} \leftarrow \theta^{g^\dagger} - \alpha_3 \times \nabla_{\theta^{g^\dagger}} L_5(\theta^{g^\dagger},\theta^{h^\dagger}))$
    $\theta^{h^\dagger} \leftarrow \theta^{h^\dagger} - \alpha_3 \times \nabla_{\theta^{h^\dagger}} L_5(\theta^{g^\dagger},\theta^{h^\dagger}))$
end
```

4.4.2 Results

By making some modifications, i.e. adding random data, to stage 2 and stage 3 of algorithm 4, we can see that the testing MSE reduced from 0.1456 to 0.138, which is much closer to the baseline

	MSE
Baseline	0.1357
Utilizing Unpaired Data	0.2723
Autoencoder-Inspired Method	0.1456
Mapping Enrichment Using Random Samples	0.138

Table 4.7: *Testing MSE of Algorithm 1, corresponding to Figure 4.7*

approach. It very surprising that random data helps. But it not very surprising that we want g, h, and z to agree everywhere not just agreeing on the training data.

4.4.3 Discussion

In this section, we again provide an analysis of the representations encoded by f,g and z. Let us denote the approximations of C using h and g as $\tilde{\mathbf{c}}_1$ and $\tilde{\mathbf{c}}_1$ respectively, i.e. $h(y) = \tilde{\mathbf{c}}_1$ and $g(x) = \tilde{\mathbf{c}}_2$.

	Column 1: MSE(($g^\dagger(\tilde{\mathbf{c}}_1), h^\dagger(\tilde{\mathbf{c}}_2)$),($\mathbf{x},\mathbf{y}$))	Column 2: MSE(($g^\dagger(\mathbf{c}), h^\dagger(\mathbf{c})$),($\mathbf{x},\mathbf{y}$))
Training	0.13401326	0.05452401
Testing	0.13564417	0.05635012

Table 4.8: *Reconstruction Error*

	MSE($\mathbf{c}$,$\tilde{\mathbf{c}}_2$)	MSE($\mathbf{c}$,$\tilde{\mathbf{c}}_1$)
Training	0.017277006	0.015227813
Testing	0.017503608	0.015561547

Table 4.9: *Encoding error (Mean Sqaured Error)*

	MAPE($\mathbf{c}$,$\tilde{\mathbf{c}}_2$)	MAPE($\mathbf{c}$,$\tilde{\mathbf{c}}_1$)
Training	23.4239%	21.3456%
Testing	23.5342%	22.6087%

Table 4.10: *Encoding error (Mean Absolute Percentage Error)*

By enforcing g and h to match z on arbitrary data, table **??** and table 4.10 shows that we achieve a reduction in both MSE and MAPE for C and its approximations.

Chapter 5

Conclusion

This Major Qualifying Project presented an investigation into the classic machine learning problem of predicting one random variable from another. Particularly, we propose 4 approaches to tackle this problem and provide an analysis of each of them. The first obvious baseline approach is to train directly two neural networks f and $f^{\dagger}$ to map X to Y, and vice versa. We reported a testing MSE of 0.1357 using this approach. For the second approach, we took an autoencoder perspective and performed unsupervised pre-training first before training two neural networks similar to the baseline method. This approach yielded a significantly higher testing MSE than the previous one. We show that this is because the unsupervised pre-training step is making the encoded representations C lose information, i.e. less rich-information, which leads to a surge in testing MSE. In the Autoencoder-inspired method, we demonstrated the importance of a rich-information representation C to our problem. Moreover, we provided an analysis of why our testing MSE is close but still higher than the baseline method. The reason was that the approximations of C using g and h were not sufficient. Hence, from the same analysis, we proposed the final approach that utilizes random data to enhance the approximations of C. In particular, we want g, h, and z to match anywhere not just on training data. A second analysis showed that we achieve better approximations of C using this approach and a testing MSE closer to the baseline than without using random data.

5.1 Future Work

To begin with, during our project, we only used simple feedforward neural networks to train our neural networks. Since our experiments use the MNIST dataset, i.e. an image dataset, we might want to try using convolutional neural networks instead. Secondly, in section 4.4, we show that a better approximation of the rich-information C can lead to a reduction in the testing MSE. Hence, better approaches to improve this approximation are worth investigating.

5.2 Reflections

The process of completing this Major Qualifying Project was a valuable and rewarding experience. To begin with, I learn that in addition to having a good research plan that needs to be followed diligently, the ability to adapt and shift the project's focus is equally important. Secondly, I realize the importance of being attentive to details as well as having a well-documented research journal about your experiments and ideas. Thirdly, I attained a lot of knowledge regarding Generative Adversarial Networks [22], especially how to train them stably, and Autoencoders. Last but not least, my ability to convey scientific ideas concisely through writing has been sharpened throughout the MQP.

For future students who would like to continue this project, I would suggest spending time conducting a literature review in the early stages of the project and finding a way to keep track of the experiments as well as ideas.

Bibliography

[1] S. Nayar, "Columbia object image library (coil100)," 1996.

[2] M. García Fernández, Y. Álvarez López, and F. Las-Heras, "Dual-probe near-field phaseless antenna measurement system on board a uav," *Sensors*, vol. 19, no. 21, 2019.

[3] J. Jordan, "Variational autoencoders.," Jul 2018.

[4] T. Hassner, S. Harel, E. Paz, and R. Enbar, "Effective face frontalization in unconstrained images," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4295–4304, 2015.

[5] X. Yin, X. Yu, K. Sohn, X. Liu, and M. Chandraker, "Towards large-pose face frontalization in the wild," in *Proceedings of the IEEE international conference on computer vision*, pp. 3990–3999, 2017.

[6] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, 2016.

[7] H. Hotelling, "Relations between two sets of variates," in *Breakthroughs in statistics*, pp. 162–190, Springer, 1992.

[8] T. W. Anderson, T. W. Anderson, T. W. Anderson, and T. W. Anderson, *An introduction to multivariate statistical analysis*, vol. 2.
Wiley New York, 1958.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*.
MIT Press, 2016.
http://www.deeplearningbook.org.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[11] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[12] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[13] Y. Bengio, Y. LeCun, *et al.*, “Scaling learning algorithms towards ai,” *Large-scale kernel machines*, vol. 34, no. 5, pp. 1–41, 2007.

[14] D. Erhan, A. Courville, Y. Bengio, and P. Vincent, “Why does unsupervised pre-training help deep learning?,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 201–208, JMLR Workshop and Conference Proceedings, 2010.

[15] G. Alain and Y. Bengio, “What regularized auto-encoders learn from the data-generating distribution,” *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593, 2014.

[16] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” *arXiv preprint arXiv:1312.6114*, 2013.

[17] Y. Bengio, A. Courville, and P. Vincent, “Representation learning: A review and new perspectives,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[18] Y. LeCun, C. Cortes, and C. Burges, “Mnist handwritten digit database,” *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[19] H. Robbins and S. Monro, “A stochastic approximation method,” *The annals of mathematical statistics*, pp. 400–407, 1951.

[20] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[21] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization.,” *Journal of machine learning research*, vol. 12, no. 7, 2011.

[22] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” *arXiv preprint arXiv:1406.2661*, 2014.