Experimental study of Manifold learning and tangent propagation

by

Temirlan Ashimov

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Master of Science in Applied Mathematics

at the

NAZARBAYEV UNIVERSITY

Feb 2021

Author .....
Department of Mathematics
Feb 25, 2021

Certified by .....
Rustem Takhanov
Assistant Professor
Thesis Supervisor

Accepted by .....
Daniel Pugh
Dean, School of Science and Humanities

Experimental study of Manifold learning and tangent propagation

by

Temirlan Ashimov

Submitted to the Department of Mathematics
on Feb 25, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Applied Mathematics

Abstract

In the Data Science routine, we often face the curse of dimensionality, dealing with high-dimensional data which, in turn, can be very difficult. The problems of this nature can be approached by methods of Dimensionality Reduction. These methods assume that data can be interpreted in a smaller dimension. The hypothesis proposed in this work is that data is located exactly or near along with a low dimension manifold and the tool for finding this manifold is auto-encoders. In particular, we calculate the basis of the tangent space of the low-dimensional manifold at each data point and up towards using it to the regularization of the regression task.

All calculations are implemented via Python 3 since this programming language includes a wide range of packages for dealing with Big Data.

Thesis Supervisor: Rustem Takhanov
Title: Assistant Professor

Acknowledgments

This thesis, in which I have put all of my commitment knowledge and skills, is the apotheosis of my master's program at Nazarbayev University. I would like to express gratitude to my supervisor assistant professor Rustem Takhanov, who assisted me during the whole process of my work. Especially, for his undoubted competence in Machine Learning, which provided me with a deep understanding of the topic.

I am also very thankful to the second reader professor Zhenisbek Assylbekov for his valuable advices and suggestions.

Contents

1 Overview **8**

1.1 Introduction 8

1.2 Background 9

1.3 Manifold Learning and Autoencoders 12

1.4 Tangent bundle by a CAE 16

1.5 Tangent Propagation 18

2 Main **19**

2.1 Manifold Tangent Classifier 19

2.2 Alternating Scheme 20

3 Experiments **23**

3.1 Dataset 23

3.2 Linear Regression 26

3.3 Manifold Tangent Classifier 28

3.3.1 Autoencoders 28

3.3.2 Tangent Propagation 36

3.4 Alternating Scheme 43

3.5 Results 44

4 Conclusion **45**

List of Figures

1-1 n-Dimensional Manifold. 10

1-2 Tangent space to the Manifold. 11

1-3 Mappings. 11

1-4 Data points on a lower-dimensional linear manifold. 12

1-5 An example of nonlinear manifold. 13

1-6 An illustration of auto-encoder's architecture. 15

1-7 Tangent bundle of a manifold which in circular form. 17

3-1 Distribution of Y vector. 26

3-2 Graphs of linear regression. 27

3-3 Cost function of auto-encoder with 1-hidden layer. 31

3-4 Cost function of auto-encoder with 2-hidden layers 35

3-5 Cost function with 1-hidden layer of auto-encoder. 39

3-6 Graphs of regression model with 1-hidden layer of auto-encoder. 40

3-7 Cost function with 2-hidden layers of auto-encoder. 41

3-8 Graphs of regression model with 2-hidden layers of auto-encoder. 42

List of Tables

3.1 Results. 44

3.2 Results. 44

Chapter 1

Overview

1.1 Introduction

Recently, the term Data Science (DS) became very popular. DS includes a lot of mathematically based methods for analyzing data. Several reasons cause the popularity of that term. One of them is that there might be some useful insights and relations within the data that can help people in numerous subject fields e.g. manufacturing, banking, finance, healthcare, and etc. These results might be obtained by statistics, time-series analysis, classification, regression, clustering, and etc. In fact, most DS methods exist for almost half of a century, and thanks to modern computer capabilities we are able to implement these methods fast enough while avoiding errors. Finally, due to the emergence of the "Big Data" paradigm, we are able to collect a huge amount of data and recognize patterns in it.

Contrastingly to this great phenomenon "Big Data", which implies not only a vast amount of data but also very high dimensionality of it, we face the "curse of dimensionality". The "curse of dimensionality" is a key obstacle for application of DS methods which

causes statistical and computational issues. [2, 5]

Fortunately, the existence of several hypotheses contributes to the solution to these problems. The key idea of them is that the dimensionality of the hidden space may be lower. It means that high-dimensional data can be transformed into lower-dimensional representations. The tasks of this kind in Machine Learning (ML) are called Dimensionality Reduction (DR) problems. In the case of that branch of ML, a lot of models are based on the geometrical approach.

The support of the data might be located on a lower-dimensional manifold. The application of Manifold Learning methods to the dataset can not only accelerate the processing of the dataset but also improve the accuracy of supervised tasks as well [1].

1.2 Background

Definition. Let a topological space D be given. If for any point x there exists such an open set O that contains this point such that a mapping $g : O \mapsto \mathbb{R}^n$ is homeomorphic, in such case D is called *n-dimensional* $Manifold$. A pair $(O, g : O \mapsto \mathbb{R}^n)$ is called a *chart*. [4]

Let $\mathbb{R}^m$ be an m-dimensional space and D be a "smooth-enough" k-dimensional data manifold such that $D \subseteq \mathbb{R}^m$ and $k \leq m$. Assume that a dimension of manifold D was calculated in advance and g is a diffeomorphic such that $g : B \mapsto D$ from $B \subseteq \mathbb{R}^k$ onto the manifold $D = g(B)$. By that way, the data manifold D is defined as $D = \{A = g(p) \in \mathbb{R}^m : p \in B \subset \mathbb{R}^k\}$, where B is an open set in $\mathbb{R}^k$. Thus, there exists an inverse mapping $f = g^{-1} : D \mapsto B$ because of homeomorphism of g, and that mapping defines low-dimensional structure on our data manifold D. [1]

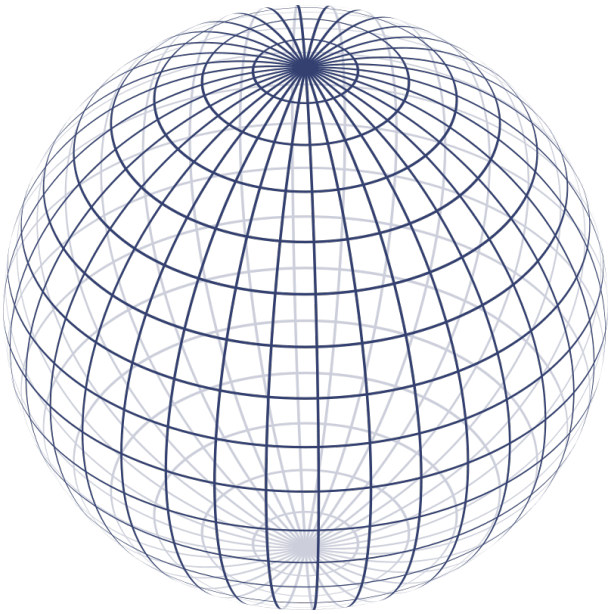

Figure 1-1: n-Dimensional Manifold.

Defnition. If $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a smooth function. Then the Jacobian matrix is defined as:

$$\mathbb{J}_f(x) = [\frac{\partial f}{\partial x_1}(x) \dots \frac{\partial f}{\partial x_n}(x)] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

Thus,

$$\text{span}(\mathbb{J}_f(x)) = \text{span}(\frac{\partial f}{\partial x_1}(x) \dots \frac{\partial f}{\partial x_n}(x)).$$

Definition. Suppose $f(A)$ and $g(p)$ are the smooth functions and Jacobian matrices of them have a form $\mathbb{J}_f(A) \subseteq \mathbb{R}^{k \times m}$ and $\mathbb{J}_g(p) \subseteq \mathbb{R}^{m \times k}$ respectively. Then *Tangent space* to the data manifold D at point $A \in D$ is defined as follows:

$$T(A) = Col(\mathbb{J}_g(f(A))),$$

where $T(A)$ is linear space in $\mathbb{R}^m$ with dimension k if $rank(\mathbb{J}_g(F(A))) = k$.

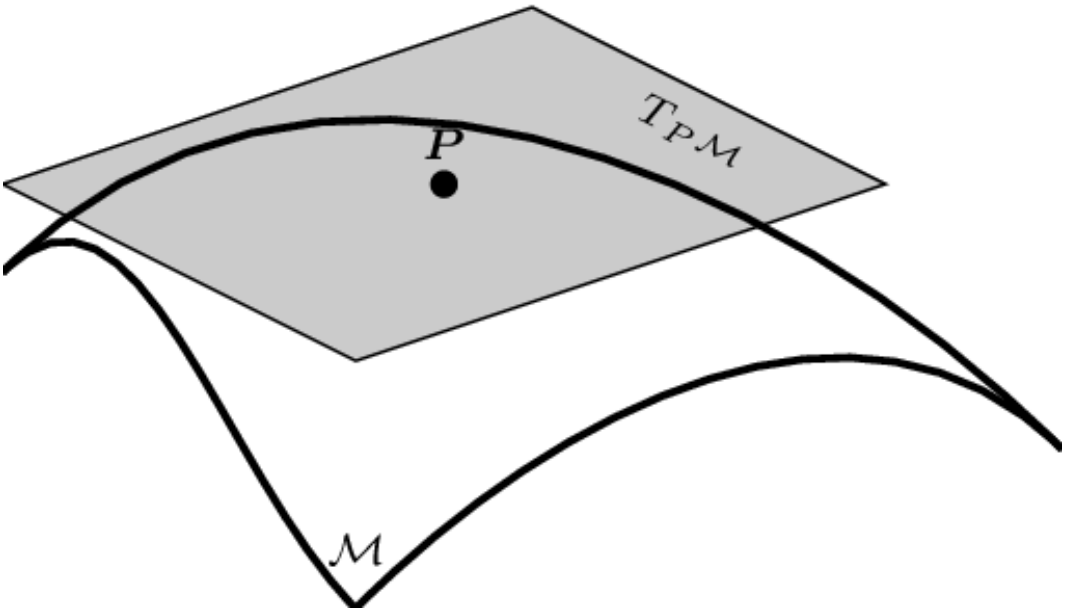

Figure 1-2: Tangent space to the Manifold.

We observe the following equivalences from the definitions above: $g(f(A)) \equiv A$ and $f(g(p)) \equiv p$. To be more explicit, the mappings $g : B \mapsto D$ and $f : D \mapsto B$ are called *recovery* and *embedding mappings* respectively.

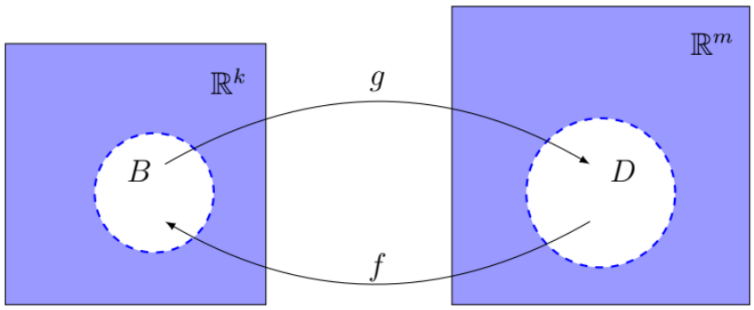

Figure 1-3: Mappings.

In the ML field, "auto-encoders" is a general concept that includes "recovery" and "embedding" mappings in itself. Auto-encoders are a tool for mapping from high-dimensional data into low dimensional space and then recovery into original space avoiding divergence from initial data points.

1.3 Manifold Learning and Autoencoders

We have discussed that the "curse of dimensionality" is an obstacle in analysis of data. In order to deal with this kind of problem several Dimensionality Reduction (DR) methods exist. As mentioned before, the key idea of these methods is that data points are sampled from a neighborhood of low-dimensional manifold, which is embedded in high-dimensional space [3]. For instance: given data points

$$\mathbf{X} = \{X_1, X_2, ..., X_n\} \subset \mathbb{R}^m.$$

The task is to find an embedding mapping f

$$f(\mathbf{X}) = \mathbf{Y} = \{Y_1, Y_2, ..., Y_n\} \subset \mathbb{R}^k$$

of m-dimensional $\mathbf{X}$ to k-dimensional $\mathbf{Y}$ where $m >> k$.

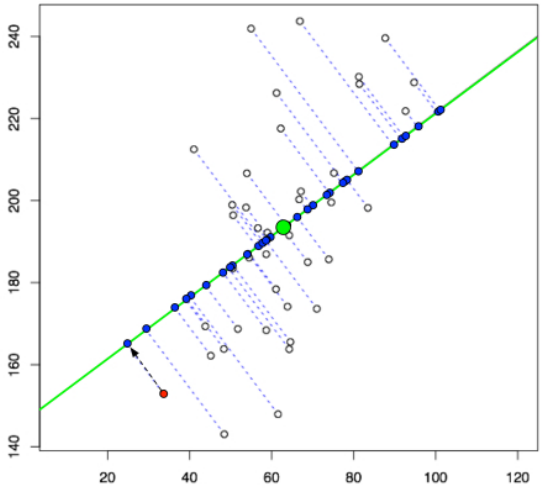

Figure 1-4: Data points on a lower-dimensional linear manifold.

There are two distinct approaches in the DR methods: *linear* and *nonlinear* (Manifold Learning - Nonlinear approach). The difference between them could be observed, such

that, in the linear model we are asked to find embedding mapping into the linear manifold, whereas the nonlinear model - the nonlinear manifold. Linear models have advantages and disadvantages against nonlinear models. For instance, linear models are faster in comparison with the nonlinear models, however, the quality of their representations of the data points leaves much to be desired. As can be seen from Figure 1-5 linear manifold causes loss of information. Obviously, there exists a nonlinear manifold that would capture data points more effectively and avoiding loss of information.

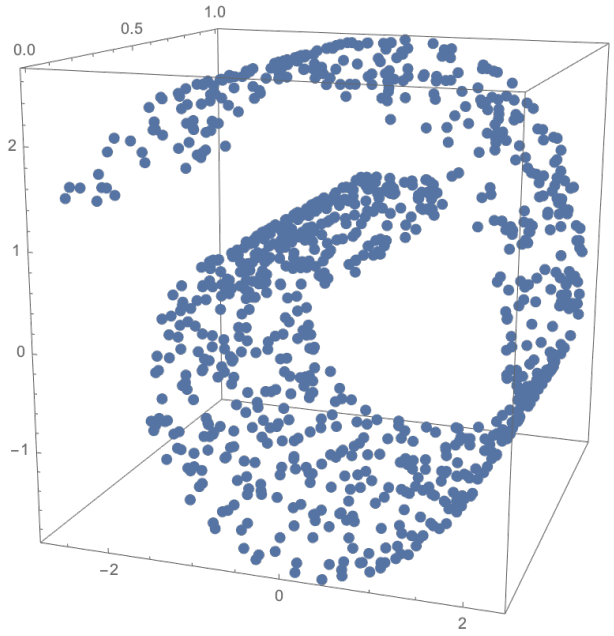

Figure 1-5: An example of nonlinear manifold.

Inside of DR, there are many techniques of nonlinear approach. One of them is a particular type of Artificial Neural Networks - **Auto-encoder** (AE). It learns an *encoder* function f which maps x from input m-dimensional space to a lower k-dimensional space, in combination with *decoder* function g that maps inversely to the original m-dimensional space. Any gradient descent technique can be chosen to calculate the optimal parameters of the auto-encoder by minimizing *loss function* $L(x, g(f(x)))$ for the points of the

training dataset.[6]

To clarify the explanation, let us consider the auto-encoder with a single hidden layer at *encoder* and *decoder* components. *Encoder* is a function of the form $f(x) = \sigma_1(Wx + b_e)$, where W is weight matrix, whose dimension $k \times m$, b_e is bias vector ($b_e \in \mathbb{R}^k$) and σ_1 is an element-wise nonlinear activation function (logistic sigmoid: $\sigma_1(z) = \frac{1}{1+e^{-z}}$). *Decoder* is a function of the form $r = g(f(x)) = \sigma_2(W^T f(x) + b_d)$, where parameter W^T is transpose of the weight matrix of the *encoder* function, b_d is a bias vector ($b_d \in \mathbb{R}^m$) and σ_2 is an element-wise activation function which can be either the same as σ_1 or identity function. It is worth mentioning if nonlinear activation functions at hidden layers of the auto-encoder were not used, an akin dimensionality reduction to PCA would be obtained. *Loss function* of the auto-encoder is either a standard squared error $\mathcal{L}(x, r) = \|x - r\|^2$ or Bernoulli cross-entropy $\mathcal{L}(x, r) = -\Sigma_{i=1}^{d} x_i \log(r_i) + (1 - x_i) \log(1 - r_i)$ if values of the input data points are between 0 and 1. As the result the *cost function* is:

$$\mathcal{J}_{AE}(W, b_e, b_d) = \sum\nolimits_{x \in D} \mathcal{L}(x, g(f(x))).$$

It is expected that with respect to small changes of training data points, the auto-encoder should encode very similar values. In mathematical terms, we would say that for any small ϵ_1, ϵ_2 there is a small number δ such that $0 < |f(x + \epsilon_1) - f(x + \epsilon_2)| < \delta$. To fulfill this idea **Contractive Auto-encoder** (CAE) could be applied [6]. CAE penalizes sensitivity of $f(x)$ to the inputs by $L2$ regularization term which is calculated as the squared Frobenius norm of the Jacobian for the encoder function f ($J(x) = \frac{\partial f}{\partial x}$). The *cost function* of CAE is defined as:

$$\mathcal{J}_{CAE}(\theta) = \sum\nolimits_{x \in D} \mathcal{L}(x, g(f(x))) + \lambda \|J(x)\|^2,$$

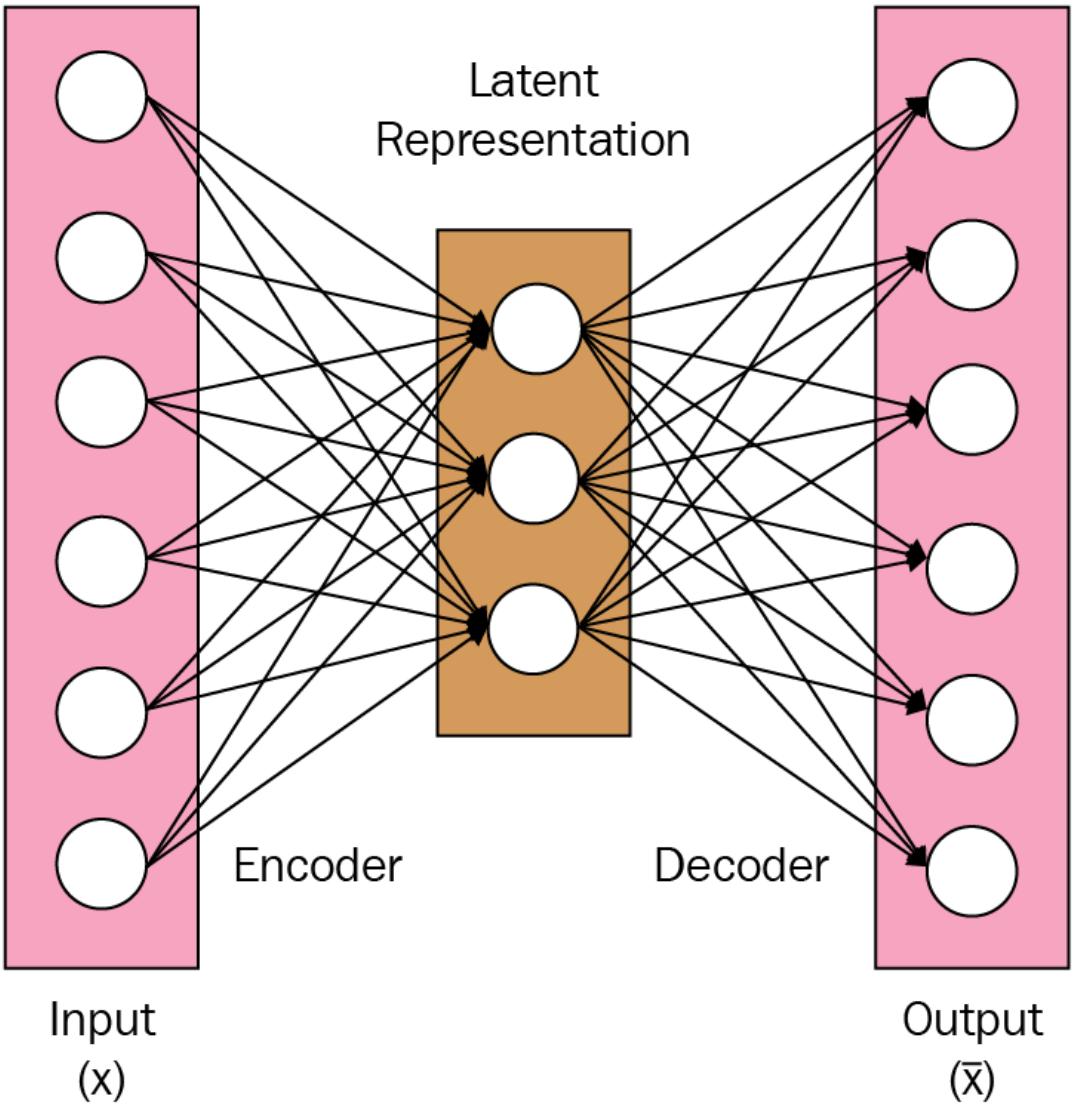

Figure 1-6: An illustration of auto-encoder's architecture.

where θ is a set of the auto-encoder's parameters and λ is a non-negative hyper-parameter that controls the effect of penalizing the Jaconbian's norm.

In addition, we are able to control the smoothness of the manifold by penalizing higher-order derivatives (Hessian). In fact, it is sufficient to apply an additional regularization term as a difference between the Jacobian at x and the Jacobian at points of neighborhood of x. The *cost function* of the CAE+H (CAE+Hessian) is defined as:

$$\mathcal{J}_{CAE+H}(\theta) = \sum_{x \in D} \mathcal{L}(x, g(f(x))) + \lambda \|J(x)\|^2 + \gamma \mathbb{E}_{\epsilon \sim \mathcal{N}(0, \sigma^2 \mathbb{I})}[\|J(x) - J(x+\epsilon)\|^2],$$

where γ is an extra non-negative hyper-parameter for controlling the effect of penalizing the Jacobian's neighbor variations.

1.4 Tangent bundle by a CAE

The *tangent bundle* of a differential manifold $\mathcal{M}$ is a collection of tangent planes at all points on the manifold $\mathcal{M}$. Every tangent plane has its own Euclidean coordinate system or *chart*, that defines an *atlas*. [6]

The chart on x is defined by the Singular Value Decomposition of the Jacobian for the encoder function f, where f is smooth ($J^T(x) = U(x)S(x)V^T(x)$, where $U(x), V(x)$ are orthogonal matrices and $S(x)$ is a diagonal matrix). The tangent plane $\mathcal{H}_x$ at x is obtained by the span of vectors $\mathcal{B}_x$ such that:

$$\mathcal{B}_x = \{U_k(x) | S_{kk}(x) > \epsilon\} \text{ and } \mathcal{H}_x = \{x + v | v \in \text{span}(\mathcal{B}_x)\},$$

where U_k is the k-column of the matrix $U(x)$.

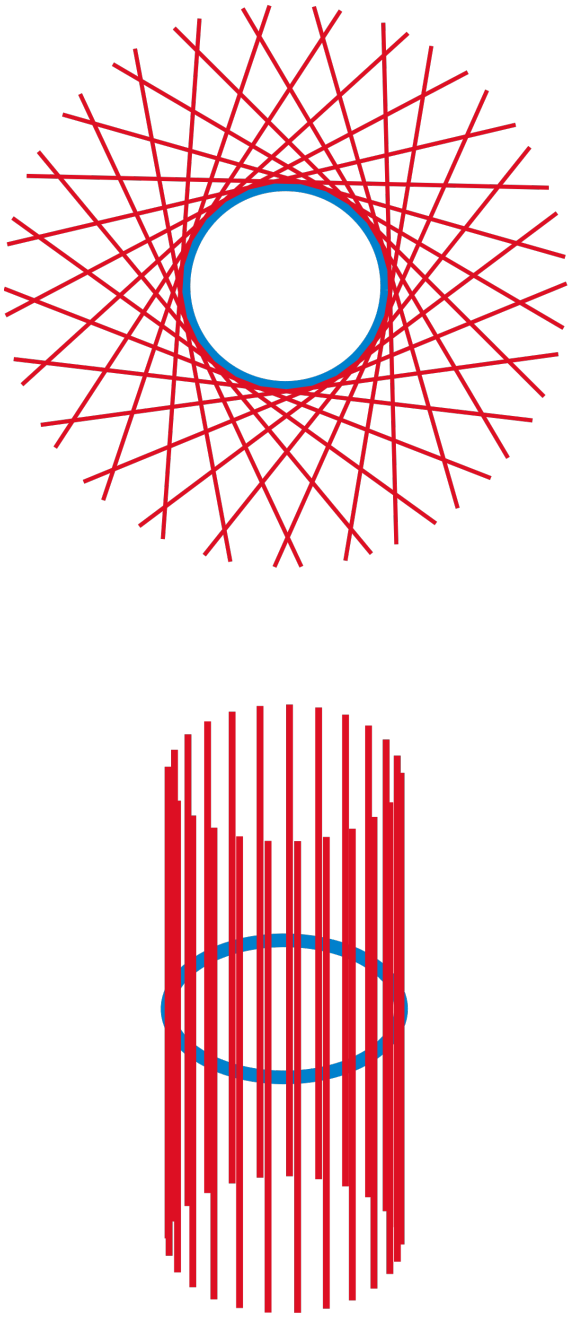

Figure 1-7: Tangent bundle of a manifold which in circular form.

1.5 Tangent Propagation

One crucial issue in Machine Learning (ML) is avoiding the over-fitting. Often the regularization might be a tool to overcome this problem. Data scientists define several regularization techniques such that: L^2, L^1, dropout for neural networks and etc.

In the tangent propagation algorithms, we encourage the model to be robust with respect to small changes of input data points. In order to achieve the robustness, $\frac{\partial o}{\partial x}$ must be orthogonal to the tangent vectors u of the manifold at x, or equivalently the dot product be as small as possible which is added to the cost function as:

$$\Omega(x) = \sum_{u \in \mathcal{B}_x} \| \frac{\partial o}{\partial x} u \|^2,$$

where o is the output of neural networks. Alongside all ML models, we can control effect of regularization by hyper-parameter of the penalty term.

Chapter 2

Main

2.1 Manifold Tangent Classifier

In previous sections, auto-encoder was introduced as a manifold learning technique. In fact, the auto-encoder is a smooth "correcting" function ϕ (composition of encoder and decoder mappings) from m-dimensional space to k-dimensional space (*code space*) where the difference between original data points (x_i) and "corrected" data points ($\phi(x_i)$) is being minimized:

$$\phi^* \leftarrow \arg\min_{\phi:\mathbb{R}^m \to \mathcal{M}^*} \frac{1}{N} \sum_{i=1}^{N} \|x_i - \phi(x_i)\|^2.$$

Manifold Tangent Classifier (MTC) algorithm includes second desirable property of "correcting" function ϕ - smoothness of the hidden manifold $\mathcal{M}$. The smoothness is implemented by additional penalty term [6]:

$$\gamma \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2\mathbb{I})} \| \frac{\partial f}{\partial x}(x_i) - \frac{\partial f}{\partial x}(x_i + \epsilon) \|^2.$$

Thus, the *cost* function of MTC's auto-encoder (contractiveness is added as well) is defined as:

$$\phi^* \leftarrow \arg\min_{\phi:\mathbb{R}^m \to \mathcal{M}^*} \frac{1}{N}\sum_{i=1}^{N} \|x_i - \phi(x_i)\|^2 + \lambda \|\frac{\partial f}{\partial x}(x_i)\|^2 + \gamma \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2\mathbb{I})} \|\frac{\partial f}{\partial x}(x_i) - \frac{\partial f}{\partial x}(x_i + \epsilon)\|^2.$$

2.2 Alternating Scheme

The goal of Alternating scheme algorithm is reduction of dimension of data points from m-dimensional space into d-dimensional manifold, where $d < k$ of k-dimensional *code space*.

Hence, the problem of finding optimal parameters of "correcting" mapping in the minimization objective should be revised as:

$$\phi^* \leftarrow \arg\min_{\phi:\mathbb{R}^m \to \mathbb{R}^m, \forall x: rank(\frac{\partial \phi}{\partial x}(x)) \leq k} \frac{1}{N}\sum_{i=1}^{N} \|x_i - \phi(x_i)\|^2,$$

where $\phi^* : \mathbb{R}^m \to \mathbb{R}^m$ is a smooth function whose Jacobian's rank is, ideally less than k, or equal to k at all data points.

In fact, the auto-encoder is "correcting" function $\phi(x) = g(f(x))$ (e.g. MTC's auto-encoder), where $f : \mathbb{R}^m \to \mathbb{R}^k$ and $g : \mathbb{R}^k \to \mathbb{R}^m$. In order to achieve a reduction of the dimension of the manifold we implement an additional penalty term to the *cost* function above (smoothness penalty regularization term is added as well). As a result, the *cost* function is defined as:

$$F(\theta, \langle B_j \rangle)_{j=1}^{M} = \sum_{i=1}^{N} \|x_i - \phi_\theta(x_i)\|^2 +$$
$$\gamma \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2 I_n)} \|\frac{\partial \phi_\theta}{\partial x}(x_i + \epsilon) - \frac{\partial \phi_\theta}{\partial x}(x_i)\|^2 +$$
$$\lambda \sum_{j=1}^{M} \|\frac{\partial \phi}{\partial x}(x_{i_j}) - B_{x_{i_j}}\|_F^2,$$

where the *cost* function should be minimized at the same time over θ and matrices $B_{x_{i_j}}$ such that $rank(B_{x_{i_j}}) \leq k$, $j = \overline{1, M}$. For fixed matrices $B_{x_{i_j}}$, minimization over θ can be achieved by any gradient descent technique. For fixed parameter θ, minimization over $\{B_{x_{i_j}}\}_{j=1}^{M}$ is equivalent to setting

$$B_{x_{i_j}} \leftarrow U^j_{1:n,1:k} \Sigma^j_{1:k,1:k} (V^j_{1:n,1:k})^T$$

where $\frac{\partial \phi_\theta}{\partial x}(x_{i_j}) = U^j \Sigma^j (V^j)^T$ is a singular value decomposition of $\frac{\partial \phi_\theta}{\partial x}(x_{i_j})$.

Our algorithm is following:

Algorithm 1: The alternating algorithm. Hyper-parameters: $m, \lambda, \gamma, \sigma, \alpha$

for $j = 1, ..., M$ **do**

$B_{x_{i_j}} \leftarrow 0$

end

for $t = 1, ..., T$ **do**

while θ *has not converged* **do**

$Sample\ \{y_i\}_{i=1}^m \sim \mathrm{P}_{data(x_1,...,x_N)}$

$Sample\ \{\epsilon_i\}_{i=1}^m \sim \mathcal{N}(0, \sigma^2)$

$Sample\ \{z_i\}_{i=1}^m \sim \mathrm{P}_{data(x_{i_1},...,x_{i_M})}$

$L \leftarrow \frac{1}{m}\sum_{i=1}^m (y_i - \phi_\theta(y_i))^2 + \frac{\gamma}{m}\sum_{i=1}^m \| \frac{\partial \phi_\theta(y_i+\epsilon_i)}{\partial x} - \frac{\partial \phi_\theta(y_i)}{\partial x} \|^2 + \frac{\lambda}{m}\sum \| \frac{\partial \phi_\theta(z_i)}{\partial x} - B_{z_i} \|^2$

$\theta \leftarrow Optimizer(\nabla_\theta L, \theta, \alpha)$

end

$\theta_t \leftarrow \theta$

for $j = 1, ..., M$ **do**

$U^j \Sigma^j (V^j)^T \leftarrow SVD(\frac{\partial g_{\theta_t}}{\partial x}(x_{i_j}))$

$B_{x_{i_j}} \leftarrow U^j_{1:n,1:k} \Sigma^j_{1:k,1:k} (V^j_{1:n,1:k})^T$

end

end

Output: $f \leftarrow \phi_{\theta_t}$

Chapter 3

Experiments

In this work, we implemented Standard Neural Network without regularization, Manifold Tangent Classifier (MTC), and Alternatig Scheme (AS) for an autoencoder (our algorithm). Further task is to compare the accuracy of the algorithm to other models. All experiments were executed on a FIFA 20 dataset using Python 3 programming language.

3.1 Dataset

The FIFA dataset was collected from the website "$fifaindex.com$" by using *requests* and $BeautifulSoup4$ packages. Dataset contains characteristics of all football players (2940 rows) from TOP-5 league clubs of Europe of season 19/20. 58 columns of the dataset are the football players' name, country, current rating, potential rating, height, weight, and etc. Worth mentioning that some columns of the dataset have *string* type data.

In pre-processing stage, to initiate X matrix following columns were used: "Height (cm)", "Weight (kg)", "Age", "Weak Foot", "Skill Moves", "Contract Length", "Ball Control", "Dribbling", "Marking", "Slide Tackle", "Stand Tackle", "Aggression", "Reactions", "Att. Position", "Interceptions", "Vision", "Composure", "Crossing", "Short Pass", "Long Pass", "Acceleration", "Stamina", "Strength", "Balance", "Sprint Speed", "Agility", "Jumping", "Heading", "Shot Power", "Finishing", "Long Shots", "Curve", "FK Acc.", "Penalties", "Volleys", "GK Positioning", "GK Diving", "GK Handling", "GK Kicking", "GK Reflexes". To initiate Y vector we used column "Value (€)". Worth mentioning that "Current Rating", "Potential Rating", and "Wage (€)" columns are being excluded from X matrix because of a strong correlation with "Value (€)". Afterwards, in order to make calculations of models faster and more robust feature standardization for the X matrix was applied by subtracting column's entries by column's mean, then dividing them by column's standard deviation ($\frac{x_i - \mu_i}{\sigma_i}$) and normalization for the Y vector dividing all entries by 1000000. At the end of the pre-processing, we utilized $train\_test\_split$ from $scikit - learn$ for the data where proportions of a training set is equal to 0.8; evaluation set is equal to 0.1; and test set is equal to 0.1. Code as follows:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


players = pd.read_excel('players (extension).xlsx', index_col = 0)
```

```
# creating X-matrix and Y-vector
x = players.loc[:, ['Height (cm)', 'Weight (kg)', 'Age', 'Weak Foot',
'Skill Moves', 'Contract Length', 'Ball Control', 'Dribbling',
'Marking', 'Slide Tackle', 'Stand Tackle', 'Aggression',
'Reactions', 'Att. Position', 'Interceptions', 'Vision',
'Composure', 'Crossing', 'Short Pass', 'Long Pass',
'Acceleration', 'Stamina', 'Strength', 'Balance',
'Sprint Speed', 'Agility', 'Jumping', 'Heading',
'Shot Power', 'Finishing', 'Long Shots', 'Curve',
'FK Acc.', 'Penalties', 'Volleys', 'GK Positioning',
'GK Diving', 'GK Handling', 'GK Kicking', 'GK Reflexes']].values.copy()
y = players.loc[:, 'Value (    )'].values.copy()

# normalization of the matrix X
scaler = StandardScaler().fit(x)
x_scaled = scaler.transform(x)

# normalization of the vector Y
y = y / 1000000

# train, validation and test splits
x_train, x_val_test, y_train, y_val_test = \
train_test_split(x_scaled, y, test_size = 0.2, random_state = 42)
x_val, x_test, y_val, y_test = \
train_test_split(x_val_test, y_val_test, test_size = 0.5,
```

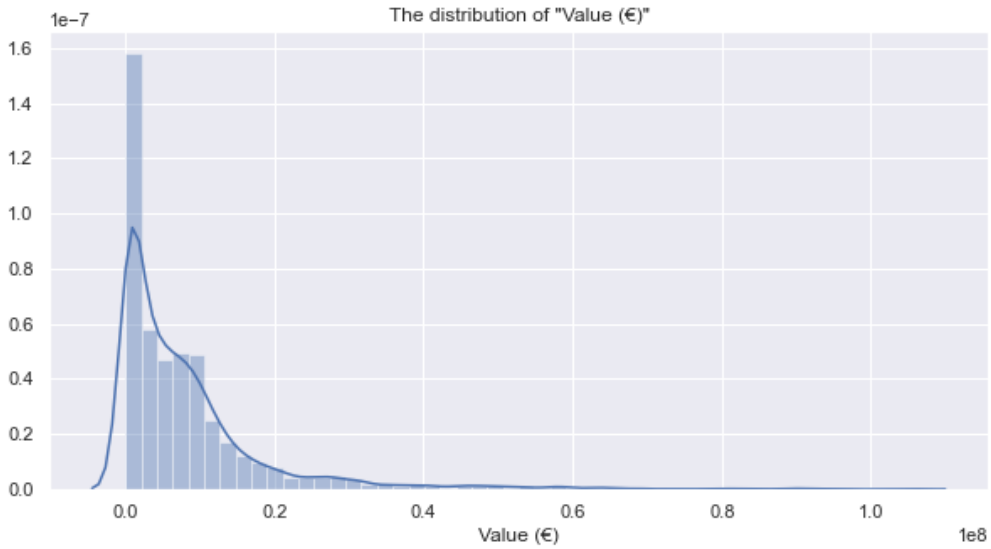

Figure 3-1: Distribution of Y vector.

```
random_state = 42)
```

3.2 Linear Regression

Initially, linear regression (baseline) has been implemented. Code is below:

```
from sklearn import *


model = linear_model.LinearRegression()
model.fit(x_train, y_train)
```

Graphs of *regplots* and *distplots* are below (train, validation, and test sets, respectively).

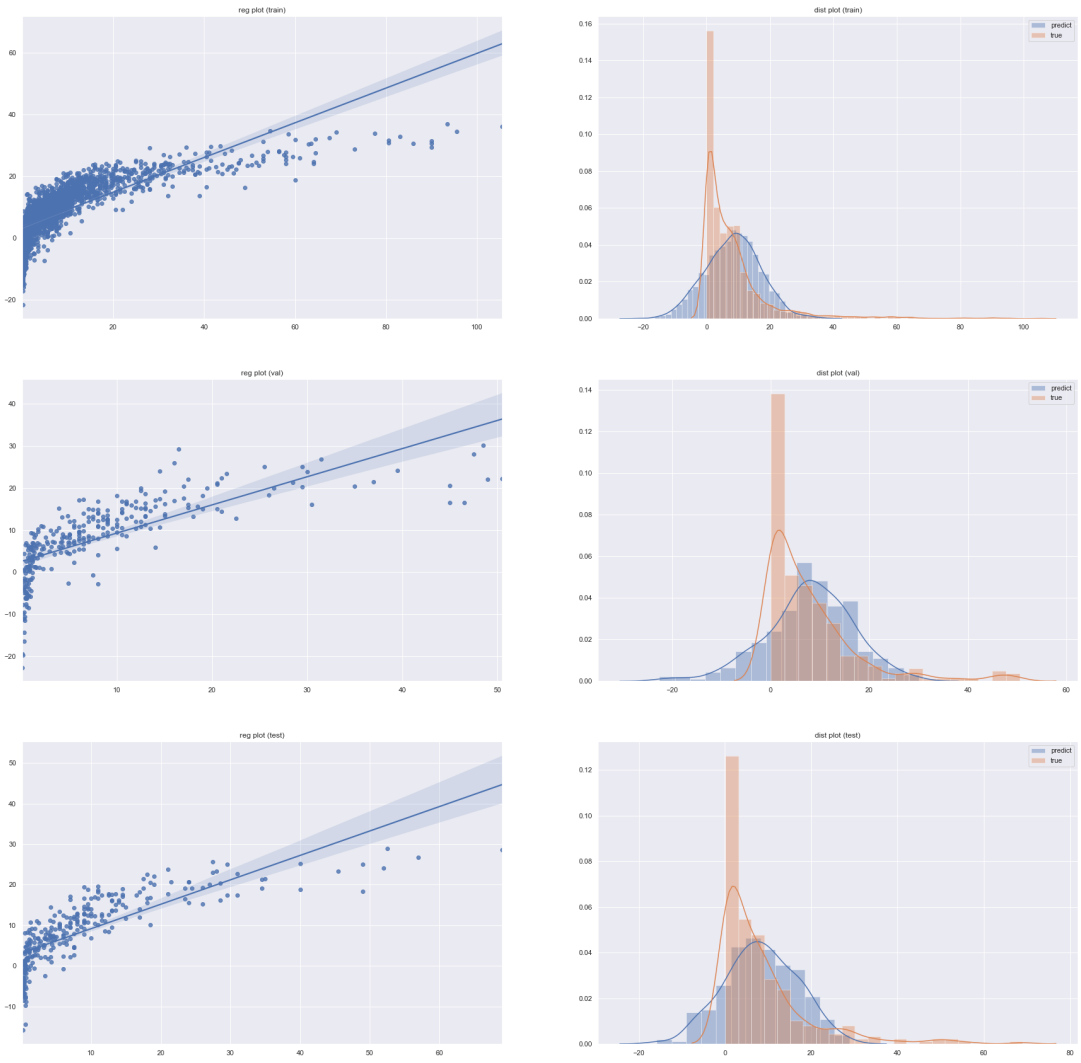

Figure 3-2: Graphs of linear regression.

3.3 Manifold Tangent Classifier

In this section, there is code of MTC with 1 and 2-hidden layers of auto-encoder.

3.3.1 Autoencoders

Code of auto-encoder with 1-hidden layer is below:

```
import tensorflow as tf


dimensionality = tensor_x_train.shape[1] # dimensionality = 40
code_size = 20
k = 7 # dimension of manifold
epsilon = 0.1
gamma = 1.0
epochs = 1500
learning_rate = 0.1


# Autoencoder
## define parameters of autoencoder
W = tf.Variable(initial_value = tf.random.truncated_normal(shape = \
(dimensionality, code_size), mean = 0, stddev = 0.1))
b = tf.Variable(initial_value = tf.constant(0.1, shape = \
(code_size)))
b_r = tf.Variable(initial_value = tf.constant(0.1, shape = \
(dimensionality)))
```

```
## define autoencoder
def autoencoder(X):
    embedding_mapping = tf.math.sigmoid(tf.linalg.matmul(X, W) + b)
    recovery_mapping = tf.linalg.matmul(
    embedding_mapping, W, transpose_b = True) + b_r
    return recovery_mapping

## define Jacobian
def jacobian(X):
    embedding_mapping = tf.math.sigmoid(tf.linalg.matmul(X, W) + b)
    sigma_prime = tf.math.multiply(
    embedding_mapping, 1 - embedding_mapping)
    diag_sigma_prime = tf.linalg.diag(sigma_prime)
    jacobian_matrix = tf.linalg.matmul(
    diag_sigma_prime, W, transpose_b = True)
    return jacobian_matrix

## smoothness penalty term of autoencoder's loss function
rand_x_train = tensor_x_train + tf.random.truncated_normal(
shape = (tensor_x_train.shape), mean = 0, stddev = epsilon)
jacobian_x_train = jacobian(tensor_x_train)
jacobian_rand_x_train = jacobian(rand_x_train)

## define autoencoder's loss function
def autoencoder_loss_function(X, Xhat):
```

```
    return tf.math.reduce_mean(tf.math.square(X - Xhat)) + \
           gamma * tf.math.reduce_mean(
           tf.square(jacobian_x_train - jacobian_rand_x_train))


## train autoencoder
loss_values = list()
for epoch in range(epochs):
    with tf.GradientTape() as tape:
        xhat = autoencoder(tensor_x_train)
        loss_value = autoencoder_loss_function(tensor_x_train, xhat)
        loss_values.append(loss_value)
        ### get gradients
        gradients = tape.gradient(loss_value, [W, b, b_r])
        ### compute and adjust weights
        W.assign_sub(gradients[0] * learning_rate)
        b.assign_sub(gradients[1] * learning_rate)
        b_r.assign_sub(gradients[2] * learning_rate)


## bases of the tangent spaces
bases_tangent_spaces = list()
for x_train_i in jacobian(tensor_x_train):
    _, _, v = tf.linalg.svd(x_train_i)
    bases_tangent_spaces.append(v[:, :k])
tensor_bases_tangent_spaces = \
tf.convert_to_tensor(bases_tangent_spaces)
```

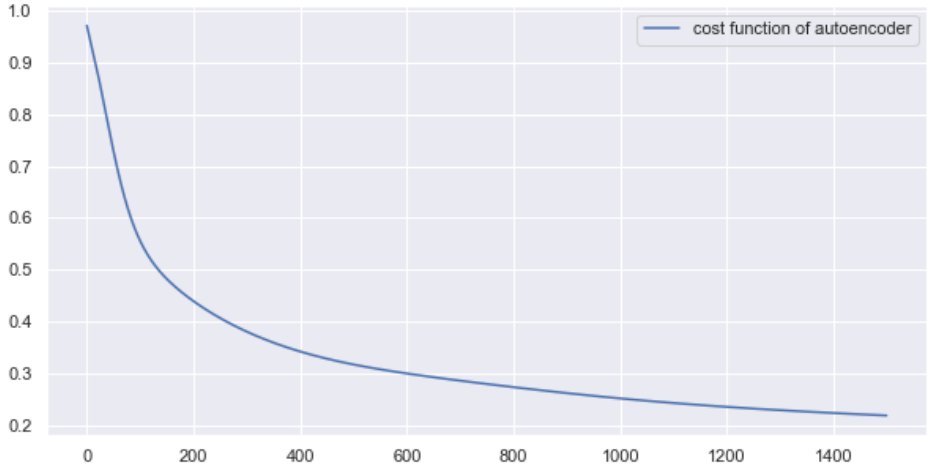

Figure 3-3: Cost function of auto-encoder with 1-hidden layer.

Code of auto-encoder with 2-hidden layers is below:

```
import tensorflow as tf


dimensionality = tensor_x_train.shape[1] # dimensionality = 40

code_size1 = 20

code_size2 = 10

k = 7 # dimension of manifold

epsilon = 0.1

gamma = 1.0

epochs = 1500

learning_rate = 0.1


# Autoencoder

## define parameters of the autoencoder

W1 = tf.Variable(initial_value = tf.random.truncated_normal(shape = \
```

```
(dimensionality, code_size1), mean = 0, stddev = 0.1))
b1 = tf.Variable(initial_value = tf.constant(0.1, shape = \
(code_size1)))
W2 = tf.Variable(initial_value = tf.random.truncated_normal(shape = \
(code_size1, code_size2), mean = 0, stddev = 0.1))
b2 = tf.Variable(initial_value = tf.constant(0.1, shape = \
(code_size2)))
W3 = tf.Variable(initial_value = tf.random.truncated_normal(shape = \
(code_size2, code_size1), mean = 0, stddev = 0.1))
b3 = tf.Variable(initial_value = tf.constant(0.1, shape = \
(code_size1)))
W4 = tf.Variable(initial_value = tf.random.truncated_normal(shape = \
(code_size1, dimensionality), mean = 0, stddev = 0.1))
b4 = tf.Variable(initial_value = tf.constant(0.1, shape = \
(dimensionality)))

## define autoencoder
def autoencoder(X):
    embedding_mapping1 = tf.math.sigmoid(
    tf.linalg.matmul(X, W1) + b1)
    embedding_mapping2 = tf.math.sigmoid(
    tf.linalg.matmul(embedding_mapping1, W2) + b2)
    recovery_mapping1 = tf.math.sigmoid(
    tf.linalg.matmul(embedding_mapping2, W3) + b3)
    recovery_mapping2 = tf.linalg.matmul(recovery_mapping1, W4) + b4
```

```
    return recovery_mapping2

## define Jacobian
def jacobian(X):
    embedding_mapping1 = tf.math.sigmoid(
    tf.linalg.matmul(X, W1) + b1)
    embedding_mapping2 = tf.math.sigmoid(
    tf.linalg.matmul(embedding_mapping1, W2) + b2)

    sigma_prime1 = tf.math.multiply(
    embedding_mapping1, 1 - embedding_mapping1)
    diag_sigma_prime1 = tf.linalg.diag(sigma_prime1)
    grad1 = tf.linalg.matmul(
    diag_sigma_prime1, W1, transpose_b = True)

    sigma_prime2 = tf.math.multiply(
    embedding_mapping2, 1 - embedding_mapping2)
    diag_sigma_prime2 = tf.linalg.diag(sigma_prime2)
    grad2 = tf.linalg.matmul(
    diag_sigma_prime2, W2, transpose_b = True)

    jacobian_matrix = tf.linalg.matmul(grad2, grad1)
    return jacobian_matrix

## smoothness penalty term of autoencoder's loss function
```

```
rand_x_train = tensor_x_train + tf.random.truncated_normal(
shape = (tensor_x_train.shape), mean = 0, stddev = epsilon)
jacobian_x_train = jacobian(tensor_x_train)
jacobian_rand_x_train = jacobian(rand_x_train)

## define autoencoder's loss function
def autoencoder_loss_function(X, Xhat):
    return tf.math.reduce_mean(tf.math.square(X - Xhat)) + \
           gamma * tf.math.reduce_mean(
           tf.square(jacobian_x_train - jacobian_rand_x_train))

## train autoencoder
loss_values = list()
for epoch in range(epochs):
    with tf.GradientTape() as tape:
        xhat = autoencoder(tensor_x_train)
        loss_value = autoencoder_loss_function(tensor_x_train, xhat)
        loss_values.append(loss_value)
        ### get gradients
        gradients = tape.gradient(loss_value,
        [W1, b1, W2, b2, W3, b3, W4, b4])
        ### compute and adjust weights
        W1.assign_sub(gradients[0] * learning_rate)
        b1.assign_sub(gradients[1] * learning_rate)
        W2.assign_sub(gradients[2] * learning_rate)
```

```
            b2.assign_sub(gradients[3] * learning_rate)
            W3.assign_sub(gradients[4] * learning_rate)
            b3.assign_sub(gradients[5] * learning_rate)
            W4.assign_sub(gradients[6] * learning_rate)
            b4.assign_sub(gradients[7] * learning_rate)

## bases of the tangent spaces
bases_tangent_spaces = list()
for x_train_i in jacobian(tensor_x_train):
    _, _, v = tf.linalg.svd(x_train_i)
    bases_tangent_spaces.append(v[:, :k])
tensor_bases_tangent_spaces = \
tf.convert_to_tensor(bases_tangent_spaces)
```

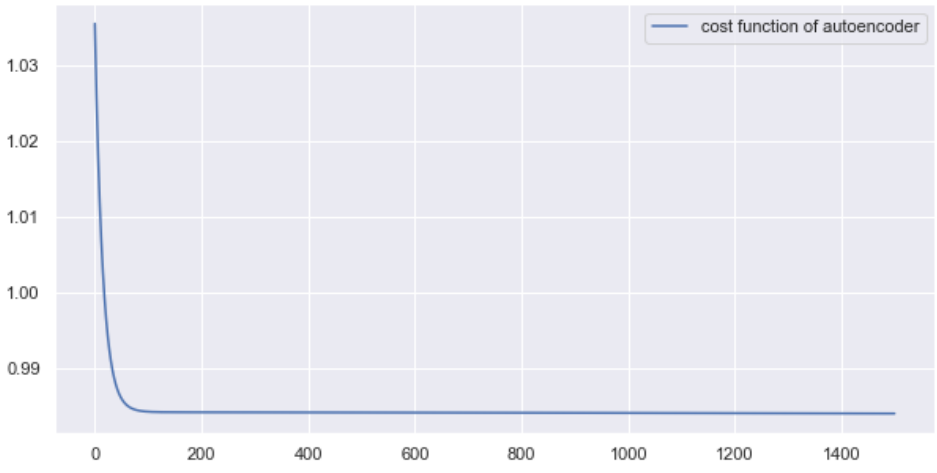

Figure 3-4: Cost function of auto-encoder with 2-hidden layers

3.3.2 Tangent Propagation

Afterward, tangent spaces at every training data points are implemented in regularization term of regression model. Code is below:

```
from tensorflow.keras import *

def model(X_train, Y_train, X_val, Y_val, activation, neurons,
learning_rate, epochs, lmbda,
          tensor_bases_tangent_spaces = tensor_bases_tangent_spaces):
    results = {'cost_train': [], 'cost_val': []}

    # Regression model
    m = Sequential()
    ## layers
    m.add(layers.Input(shape = (X_train.shape[1], )))
    for neuron in neurons:
        if activation == 'relu':
            m.add(layers.Dense(neuron, activation = 'relu',
             kernel_initializer = tf.keras.initializers.HeNormal()))
        elif activation == 'tanh':
            m.add(layers.Dense(neuron, activation = 'tanh',
             kernel_initializer = tf.keras.initializers.GlorotNormal()))
        else:
            raise Exception('activation must be either relu or tanh')
    m.add(tf.keras.layers.Dense(1, activation = 'relu'))
```

```
## cost funtion (val)
def cost(y, yhat):
    return tf.math.reduce_mean(
    tf.math.square(tf.math.subtract(y, yhat)))

## cost function (train)
def cost_with_reg(y, yhat, m_gradients, tensor_bases_tangent_spaces):
    regularizer = 0
    for gradient, basis_tg_space in \
    zip(m_gradients, tensor_bases_tangent_spaces):
        regularizer += tf.math.reduce_mean(
        tf.math.square(tf.linalg.matmul(tf.reshape(gradient, \
        shape = [1, gradient.shape[0]]), basis_tg_space)))
    return tf.math.reduce_mean(
    tf.math.square(tf.math.subtract(y, yhat))) + \
    lambda * (1/y.shape[0]) * regularizer

## train model
for epoch in range(epochs):
    with tf.GradientTape() as tape1:
        ### prepare components of regularizer
        var_X_train = tf.Variable(X_train)
        m_gradients = tape1.gradient(m(var_X_train), var_X_train)
    with tf.GradientTape() as tape2:
```

```
            ### compute model's loss function
            yhat = m(X_train)
            loss_train = cost_with_reg(
            Y_train, yhat, m_gradients, tensor_bases_tangent_spaces)
            results['cost_train'].append(loss_train)
            #### get gradients
            loss_gradients = tape2.gradient(loss_train, m.weights)
            #### compute and adjust weights
            for gradient, weight in zip(loss_gradients, m.weights):
                weight.assign_sub(gradient * learning_rate)
            loss_val = cost(Y_val, m(X_val))
            results['cost_val'].append(loss_val)

    ## rsquare
    results['r2_train'] = r2_score(Y_train, m(X_train))
    results['r2_val'] = r2_score(Y_val, m(X_val))

    return results, m
```

In order to obtain the best accuracy of regression model grid search has been applied. Set of hyper-parameters is below:

```
param_grid = {
    'activation': ['relu'],
    'neurons': [[5], [20], [5, 5], [10, 10], [20, 20]],
    'learning_rate': [0.0005],
```

```
    'epochs': [10000],
    'lmbda': [1.0, 0.1, 0.0]
}
```

Main disadvantage of grid-search approach is slow speed of code running.

Following that, graph of optimal regression model's cost function with 1-hidden layer of auto-encoder is below:

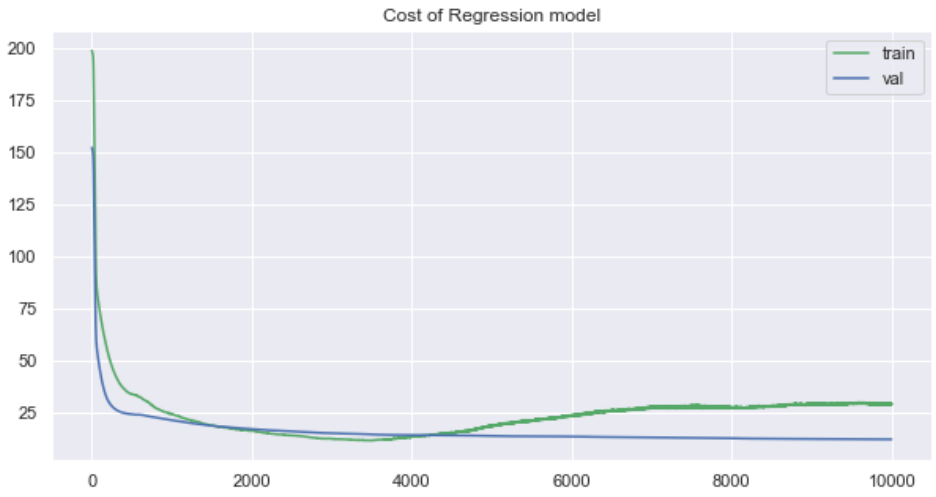

Figure 3-5: Cost function with 1-hidden layer of auto-encoder.

Graphs of *regplot* and *displot* are below (train, validation, and test sets, respectively):

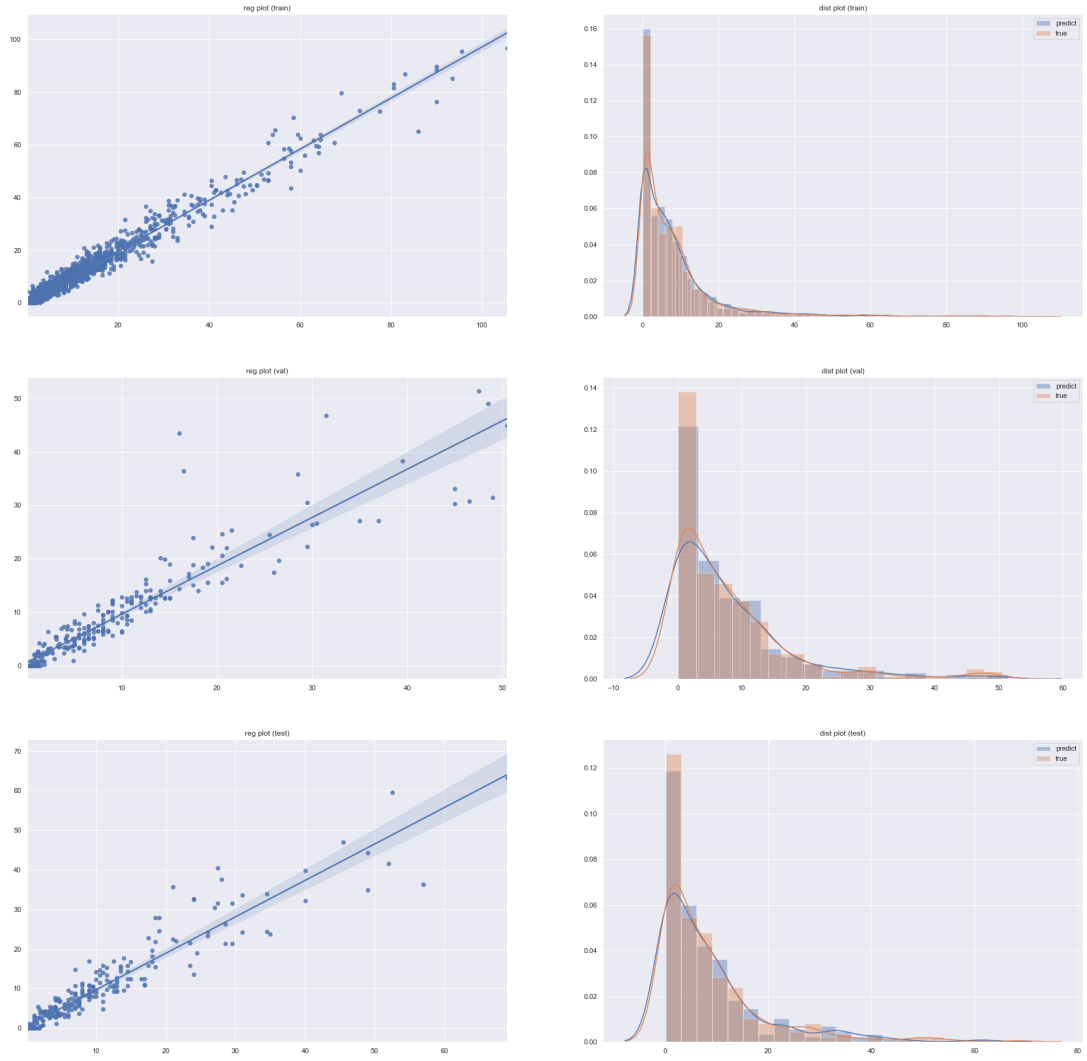

Figure 3-6: Graphs of regression model with 1-hidden layer of auto-encoder.

Finally, graph of optimal regression model's cost function with 2-hidden layers of auto-encoder is below:

Figure 3-7: Cost function with 2-hidden layers of auto-encoder.

Graphs of *regplot* and *displot* are below (train, validation, and test sets, respectively):

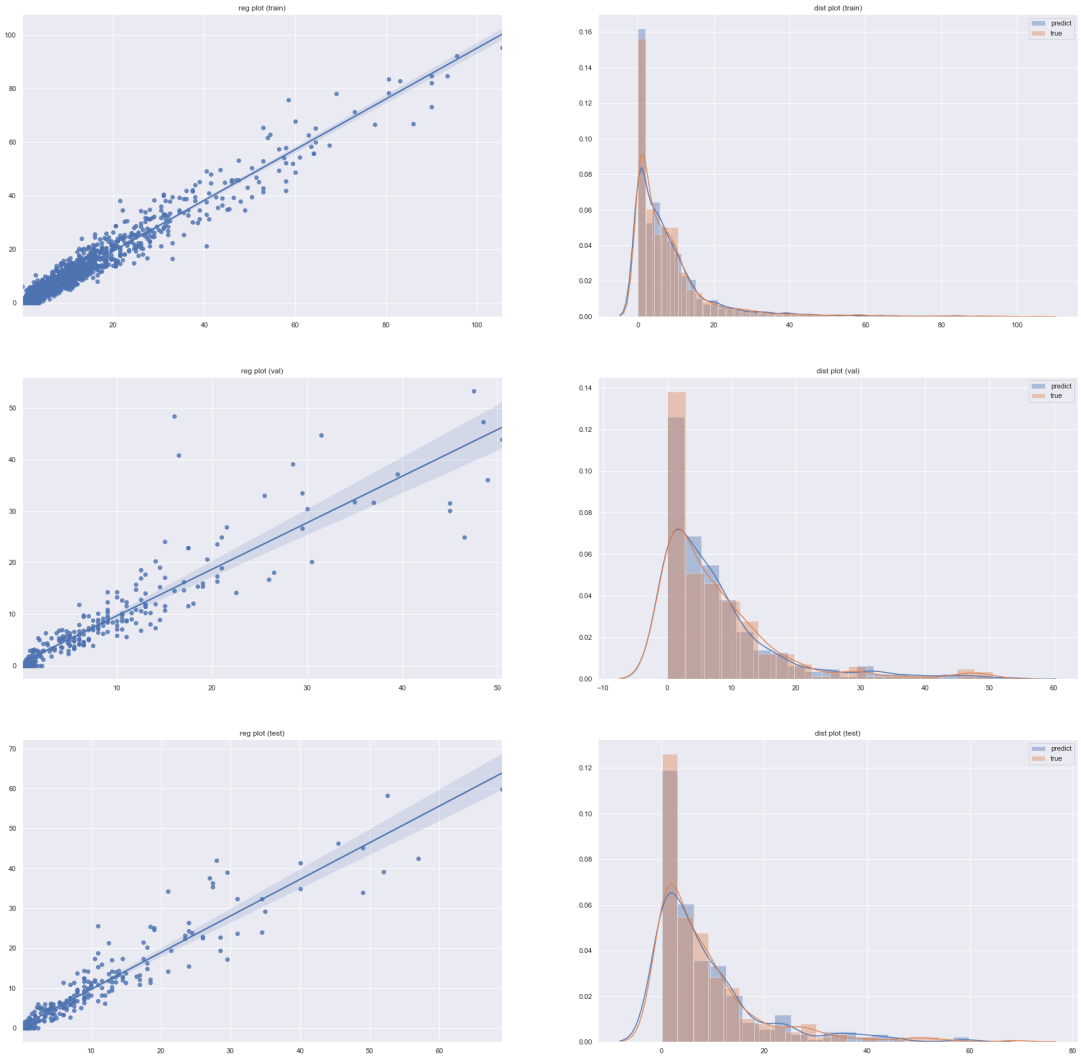

Figure 3-8: Graphs of regression model with 2-hidden layers of auto-encoder.

3.4 Alternating Scheme

3.5 Results

R^2 is selected as a metric that determines the accuracy of regression model where R^2 = 1 is the highest possible value which implies perfect predictions.

Results			
Metrics	Linear Regression	Standard Neural Network	1-Layer MTC
R^2	0.34	0.89	0.91

Table 3.1: Results.

Results			
Metrics	2-Layer MTC	1-Layer AS	2-Layer AS
R^2	0.91	???	???

Table 3.2: Results.

Chapter 4

Conclusion

We have considered auto-encoder as a manifold learning technique in Machine Learning with its possible difficulties in minimization of *cost* function. We implemented an additional penalty term (based on Alternating Scheme algorithm) to the Manifold Tangent Classifier's *cost* function in order to compress the data into a lower-dimensional hidden manifold. Hypothetically, this approach gives a better performance than MTC does.

In previous section, we have obtained that MTC shows better performance than Standard Neural Network (without regularization) does.

Bibliography

[1] Yu. A. Yanovich A. P. Kuleshov, A. V. Bernstein. Manifold learning based on kernel density estimation. *Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki*, 2018, Volume 160, Book 2, 327–338.

[2] A. V. Bernstein. Manifold learning in statistical tasks. *Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki*, 2018, Volume 160, Book 2, 229–242.

[3] Cayton L. Algorithms for manifold learning //univ. of california at san diego tech. Rep. – 2005. – . 12. – №. 1-17. – . 1.

[4] W. Tu. Loring. "An introduction to manifolds.". (2011), Springer.

[5] Verleysen M. Learning high-dimensional data. in: Ablameyko s. et al. (eds.). *Limitations and Future Trends in Neural Computation*, IOS Press, 2003. pp. 141-162.

[6] Salah Rifai, Yann N Dauphin, Pascal Vincent, Yoshua Bengio, and Xavier Muller. The manifold tangent classifier. *Advances in neural information processing systems*, 24:2294–2302, 2011.