Classifying Human Driving Behavior via Deep Neural Networks

A Thesis
Submitted to the Faculty
of
Drexel University
by
Jae Hoon Kim
in partial fulfillment of the
requirements for the degree
of
Master in Computer Science
June 2017

Table of Contents

List of Tables

List of Figures

Abstract

Classifying Human Driving Behavior via Deep Neural Networks

Jae Hoon Kim

Santiago Ontañón, Ph.D.

The average person spends several hours a day behind the wheel of their vehicles, which are usually equipped with on-board computers capable of collecting real-time data concerning driving behavior. However, this data source has rarely been tapped for healthcare and behavioral research purposes. This MS thesis is done in the context of the *Diagnostic Driving* project, an NSF funded collaborative project between Drexel, Children Hospital of Philadelphia (CHOP) and the University of Central Florida that aims at studying the possibility of using driving behavior data to diagnose medical conditions. Specifically, this paper introduces focuses on the classification of driving behavior data collected in a driving simulator using deep neural networks. The target classification task is to differentiate novice versus expert drivers. The paper presents a comparative study on using different variants of LSTM (Long-Short Term Memory networks) and Auto-encoder networks to deal with the fact that we have a small amount of labels (16 examples of people driving in the simulator, each labeled with an "expert" or "inexpert" label), but each simulator drive is high dimensional and too densely sampled (each drive consists of 100 variables sampled at 60Hz). Our results show that using an intermediate number of neurons in the LSTM networks and using data filtering (only considering one out of each 10 samples) obtains better results, and that using Auto-encoders works worse than using manual feature selection.

Chapter 1: Introduction

Most people spend several hours per day for driving vehicle when they go to work, school or shopping. Modern car systems collect real-time data of the car status and driving behavior of the driver. Moreover, such recorded data is a very promising data source for healthcare and research, which has currently rarely been used. This paper is done in the context of the *Diagnostic Driving* project, an NSF funded collaborative project between Drexel, Children Hospital of Philadelphia, George Mason University and the University of Central Florida that aims at studying the possibility of using driving behavior data to diagnose medical conditions.

The paper focuses on the classification of driving simulation data from novice and expert drivers using several different neural network models. The task is thus, to predict, whether a new unseen before driver is an expert or a novice driving just by the data recorded from a driving simulator. Two datasets are used on the paper. The first dataset contains 16 traces from four drivers: two are inexpert drivers and two are expert drivers. Each drivers drove four different tracks. The second dataset is a modification of the first dataset but after having done feature selection manually to leave only the 23 features we considered most relevant. The three models used in this work are based on Long-Short Term Memory (LSTM) networks. The first model directly uses the second dataset to classify drivers using LSTMs. The second model uses a single layer auto-encoder to automatically reduce the dimensionality of first dataset and then classifies it via an LSTM. The last model uses a multiple layer auto-encoder to reduce the dimensionality of first dataset and then classifies it via an LSTM.

The remaining of this document is organized as follows. Chapter 2 introduces background knowledge for deep neural network including LSTM and Auto-encoder and compares three most popular deep learning libraries. Chapter 3 explains detail about dataset. Technical approach and three different neural networks used for experiments on the paper are described in Chapter 4. On Chapter 5, result from three different neural network models are compared and analyzed. The last chapter, Chapter 6, forms a conclusion. The chapter mentions limitation of auto-encoder and new way to train multiple layer auto-encoder.

Chapter 2: Background

This chapter presents the necessary background to understand the experiments presented later in the paper. This chapter is dived into three sections. First, Section 2.1 covers basic deep learning concepts including recurrent neural networks and auto-encoders. Second, Section 2.2 compares deep learning libraries: DL4J, Theano, and Tensorflow. Finally, Section 2.3 briefly discusses existing work in machine learning using for modeling driving data.

2.1 Deep Learning

This section covers basic concepts of deep learning. First, Section 2.1.1 introduces deep learning, and next Section 2.1.2 deals with recurrent neural networks and Long Short Term Memory networks (LSTMs). Finally, Section 2.1.3 covers auto-encoders.

2.1.1 Basic Concepts

Artificial Neuron

The concept of artificial neurons or perseptrons was introduced by Warren McCullock and Walter Pitts [**?**]. Initially artificial neurons were described as binary output logic gates from inputs. If the sum of weighted inputs is greater than a threshold, the output is 1, otherwise, the output is 0.

Figure 2.1 shows a basic artificial neuron. The input is $\vec{x} = \{x_1, ..., x_n\} \in \mathbb{R}^n$ and the weights for each input are $\vec{w} = \{w_1, ..., w_n\} \in \mathbb{R}^n$. Let θ be threshold and define step function as below:

$$U(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

The function deciding output o is called an activation function. In the case of the original McCullock and Pitts neurons, the step function $U(z)$ is used as the activation function.

Figure 2.1 also can be described by the following equation:

$$o = U(\vec{x} \cdot \vec{w} - \theta)$$

Figure 2.2 shows an general artificial neuron. The threshold is replaced by a *bias* term and the *subtract* node is replaced by an *add* node. In general artificial neurons, the activation function ϕ could be an arbitrary function such as step, linear, or sigmoid functions. An general artificial neuron also can be described by the following equation:

$$o = \phi(\vec{x} \cdot \vec{w} + b)$$

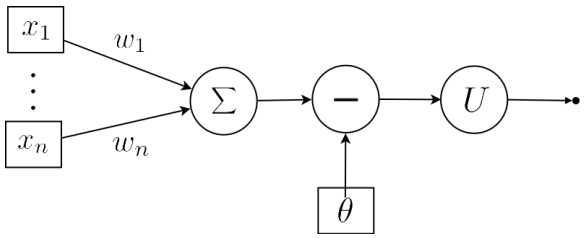


Figure 2.1: Basic Artificial Neuron

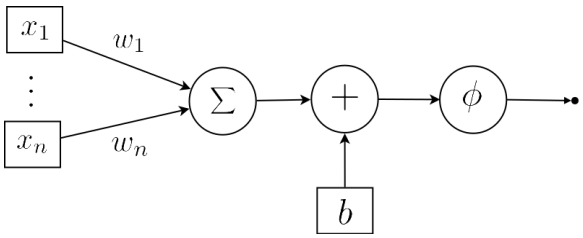


Figure 2.2: General Artificial Neuron

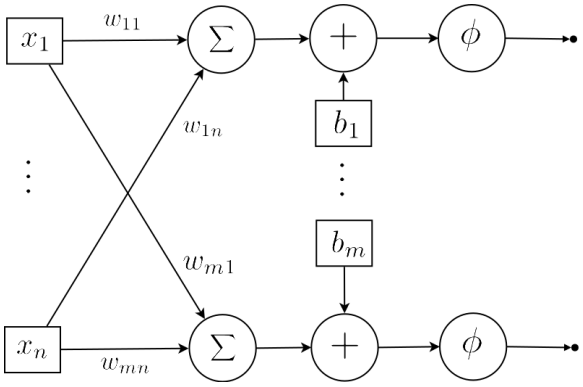


Figure 2.3: Single Layer in a Neural Network

Neural Network

Using a single neuron has many limitations, however, since a single neuron can solve only specific problems. For example, single neurons cannot solve the XOR problem. Instead of using a single neuron, standard neural networks contain a collection of neurons. Figure 2.3 illustrates this by showing n input and m neurons. The input is $\vec{x} = \{x_1, ..., x_n\} \in \mathbb{R}^n$, weights for ith neuron are $\vec{w_i} = \{w_{i1}, ..., w_{in}\} \in \mathbb{R}^n$, and the bias term for the ith neuron is b_i. The output $\vec{o} = \{o_1, ..., o_m\} \in \mathbb{R}^m$ of the network is calculated as follows:

$$\vec{o} = \boldsymbol{\phi}(\{(\vec{x} \cdot \vec{w_1} + b_1), (\vec{x} \cdot \vec{w_2} + b_2), ..., (\vec{x} \cdot \vec{w_m} + b_m)\})$$

Where the activation function is applied on each element of its input.

To simplify the equation, let a transform matrix $\mathbf{w} : \mathbb{R}^n \to \mathbb{R}^m$ be $\mathbf{w} = \begin{bmatrix} \vec{w_1} & \vec{w_2} & \cdots & \vec{w_m} \end{bmatrix}$ and bias vector $\vec{b} = \{b_1, ..., b_m\} \in \mathbb{R}^m$.

The equation is

$$\vec{o} = \boldsymbol{\phi}((\vec{x}^T \mathbf{w})^T + \vec{b})$$

The bias term can be skipped by extending the input vector and weight transform matrix. Let $\vec{X} \in \mathbb{R}^{n+1}$ be $\{x_1, x_2, ..., x_n, 1\}$ which is added one more dimension from $\vec{x}$ with value 1 and $\vec{W_i} \in \mathbb{R}^{n+1}$ be $\{w_{i1}, ..., w_{in}, b_i\}$ which is added by bias term into weights and $\mathbf{W} : \mathbb{R}^{n+1} \to \mathbb{R}^m$ be $\mathbf{W} = \begin{bmatrix} \vec{W_1} & \vec{W_2} & \cdots & \vec{W_m} \end{bmatrix}$

Figure 2.4 shows same neural network as Figure 2.3 with $\vec{X}$ and $\mathbf{W}$. Notice that the figure does not have an *add* node anymore.

In the remainder of this paper we will use the following convention: vector in upper case contain the bias term and vectors in lower case do not contain the bias term. For example, $\vec{X}$ is with bias and $\vec{x}$ is without bias. Also all vectors are row vectors. Transform matrix is bold character and if it includes bias, transform matrix uses upper case, otherwise it uses lower case. For example, $\mathbf{W}$ is a transform matrix with bias and $\mathbf{w}$ is a transform matrix without bias. By applying this, the

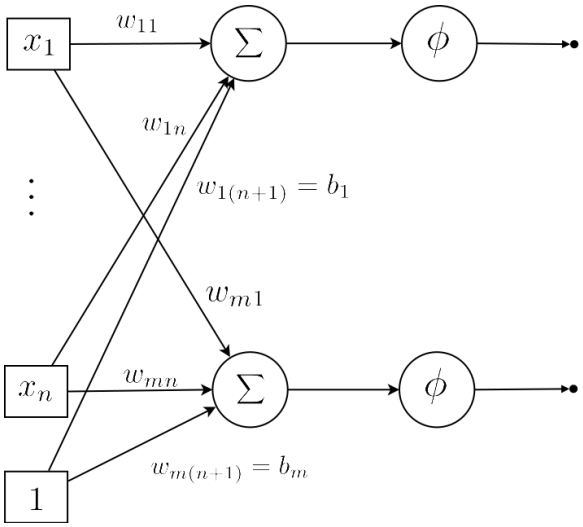


Figure 2.4: Simplified Neural Network

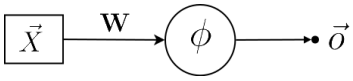


Figure 2.5: Simplified Neural Network

equation for a neural network is:

$$\vec{o} = \boldsymbol{\phi}(\vec{X}\mathbf{W})$$

For all figures in the remainder of this paper, the Σ node will be skipped, square shape nodes represent input nodes, circle shape nodes represent computation nodes, edges with transform matrix represent the weight matrix, edges without transform matrix represent the identity matrix as the weight matrix and arrow with small filled circle on the end represent the output vector. Figure 2.3, Figure 2.4, and 2.5 are thus equivalent.

Multilayer Perceptron

Single layer perceptron neural network has only an input layer and an output layer such as Figure 2.5 but multilayer perceptron (MLP) has several hidden layers between an input layer and an output layer. This section introduces the basic ideas of MLPs by describing a neural network with one hidden layer. Let $\vec{x} \in \mathbb{R}^n$ be inputs, $\vec{h} \in \mathbb{R}^m$ be outputs from hidden layers, $\vec{c} \in \mathbb{R}^m$ be bias for hidden layers, $\vec{o} \in \mathbb{R}^l$ be outputs from an output layer, $\vec{b} \in \mathbb{R}^l$ be a bias for an output layer, $\mathbf{v} : \mathbb{R}^n \to \mathbb{R}^m$ be a transform matrix for hidden layers and $\mathbf{w} : \mathbb{R}^m \to \mathbb{R}^l$ be a transform matrix for an output layer. The activation function $\mathbf{f}()$ is for hidden layers and the activation function $\mathbf{g}()$ is for an output layer. Figure 2.6 shows one hidden layer MLP.

To compute output of MLP, forward-propagation is used. Forward-propagation is to pass inputs $\vec{x}$ to outputs through hidden layers [**?**]. For example, the output from hidden layers on Figure 2.6 can be computed as:

$$\vec{h} = \mathbf{f}(\vec{x}\mathbf{v} + \vec{c}) = \mathbf{f}(\vec{X}\mathbf{V})$$

Then the output layer uses outputs from hidden layer as inputs. The outputs of the neural network can be computed as:

$$\vec{o} = \mathbf{g}(\vec{h}\mathbf{w} + \vec{b}) = \mathbf{g}(\vec{H}\mathbf{W})$$

Therefore, a final equation for two layer neural network is as follows:

$$\vec{o} = \mathbf{g}(\mathbf{h}(\vec{x}\mathbf{v} + \vec{c})\mathbf{w} + \vec{b}) = \mathbf{g}(\mathbf{h}(\vec{X}\mathbf{V})\mathbf{W})$$

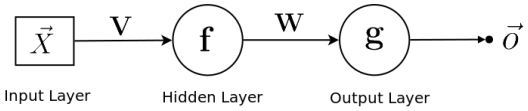


Figure 2.6: One Hidden Layer MLP

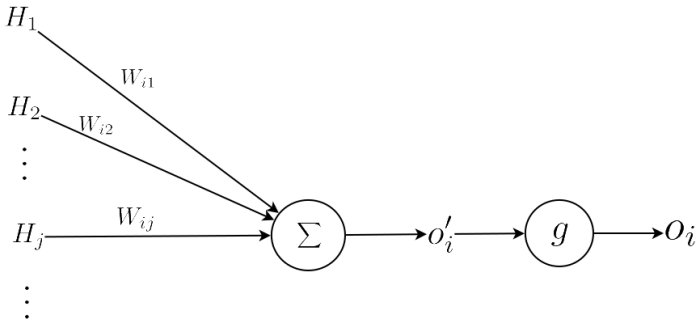


Figure 2.7: Back-propagation for W_{ij}

Back-propagation

Training a neural network means finding weights and biases. Similar as forward-propagation, back-propagation is used when MLP is trained. As forward-propagation passes inputs $\vec{x}$ to outputs, back-propagation passes an error from the output layer to hidden layers [**?**]. An error function $E()$ (some times called a loss function) usually uses a sum square error as follows:

$$E(\vec{y}, \vec{o}) = \frac{1}{2}\sum_i (y_i - o_i)^2$$

where $\vec{y}$ is an expected output from MLP.

Training neural network means finding weights and biases that make $\vec{o}$ close to $\vec{y}$. It means that training neural network needs a function of weights and biases. The train function $J()$ to train a neural network can be defined from error function $E()$. The error function $E()$ is a function of expected output and output from neural network with fixed weights and biases. Let weights and biases be changeable and the input be fixed on the error function, then the output is depends on weights and biases and the trained function $J()$ for one hidden MLP (Figure 2.6) can be define below:

$$J(\mathbf{V}, \mathbf{W}) = \frac{1}{2}\sum_i (y_i - o_i)^2 = \frac{1}{2}\sum_i (y_i - g(\vec{H}\mathbf{W_i})^2 = \frac{1}{2}\sum_i (y_i - g(\mathbf{h}(\vec{X}\mathbf{V})\mathbf{W_i}))^2$$

where $\mathbf{W_i}$ is ith column of $\mathbf{W}$

The training function $J()$ for one hidden layer MLP is a function of $\mathbf{V}$ and $\mathbf{W}$ with fixed input. Again capital letter of transform matrix contains bias. So $\mathbf{V}$ is weights and biases for hidden layer and $\mathbf{W}$ is weights and biases for output layer.

To train the neural network, weights and biases are regulated. To update weights and biases, let $\vec{h'} = \vec{X}\mathbf{V}$ be inputs for a hidden layer activation function $\mathbf{f}$ and $\vec{o'} = \vec{H}\mathbf{W}$ be inputs for an output layer activation function $\mathbf{g}$. So $\vec{h} = \mathbf{f}(\vec{h'}) = \mathbf{f}(\vec{X}\mathbf{V})$ and $\vec{o} = \mathbf{g}(\vec{o'}) = \mathbf{g}(\vec{H}\mathbf{W})$. Let V_{ij} be ith neuron weights in hidden layers for inputs X_j and W_{ij} be ith neuron weight in an output layer for inputs H_j.

Let's first update W_{ij} and Figure 2.7 shows the detail of an output layer related with W_{ij}. The updated weight ${W_{ij}}^{next}$ can be calculated as follows:

$${W_{ij}}^{next} = W_{ij} - \eta\frac{\partial J}{\partial W_{ij}}$$

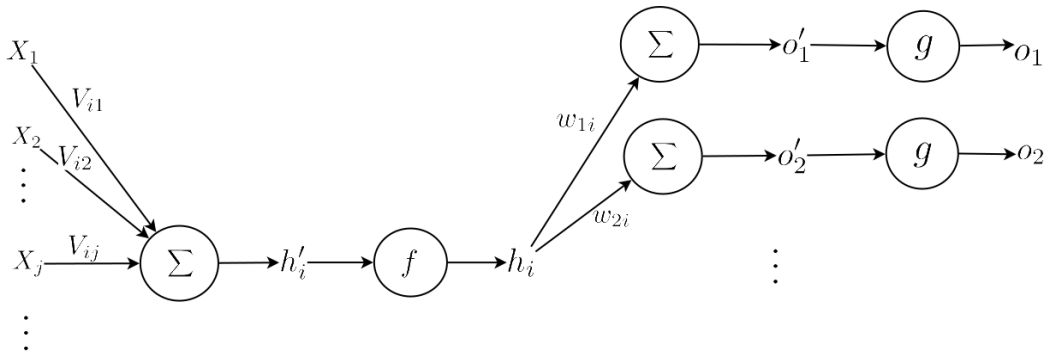


Figure 2.8: Back-propagation for V_{ij}

where η is a learning rate. The equation is intended to remove error, and thus the updated weight should move in the opposite direction of the gradient of the error function. Thus, this method is called 'gradient descent'.

Let's compute $\frac{\partial J}{\partial W_{ij}}$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial o_i}\frac{\partial o_i}{\partial W_{ij}} = \frac{\partial J}{\partial o_i}\frac{\partial o_i}{\partial o'_i}\frac{\partial o'_i}{\partial W_{ij}}$$

So the equation has three parts.

First part is

$$\frac{\partial o'_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}}\sum_k (H_k W_{ik}) = \frac{\partial}{\partial W_{ij}}(H_j W_{ij}) = H_j$$

Assume the activation function for an output layer is a sigmoid function $sigm()$ then the second part is

$$\frac{\partial o_i}{\partial o'_i} = \frac{\partial}{\partial o'_i} sigm(o'_i) = sigm(o'_i)\{1 - sigm(o'_i)\} = o_i(1 - o_i)$$

where

$$\frac{d}{dx} sigm(x) = \frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}(1 - \frac{1}{1+e^{-x}}) = sigm(x)\{1 - sigm(x)\}$$

Assume train function is sum square then the last part is

$$\frac{\partial J}{\partial o_i} = \frac{\partial}{\partial o_i}\frac{1}{2}\sum_k (y_k - o_k)^2 = \frac{\partial}{\partial o_i}\frac{1}{2}(y_i - o_i)^2 = -(y_i - o_i)$$

Therefore,

$$W_{ij}{}^{next} = W_{ij} - \eta\frac{\partial J}{\partial W_{ij}} = W_{ij} - \eta\frac{\partial J}{\partial o_i}\frac{\partial o_i}{\partial o'_i}\frac{\partial o'_i}{\partial W_{ij}} = W_{ij} + \eta(y_i - o_i)o_i(1 - o_i)H_j$$

Before moving to update weights for hidden layer, define δ_i as follows:

$$\delta_i = \frac{\partial J}{\partial o'_i} = \frac{\partial J}{\partial o_i}\frac{\partial o_i}{\partial o'_i}$$

The δ_i is useful because it is shared when back-propagation updates all weights related with ith neuron. Also the δ_i is used when back-propagation updates weights on hidden layers.

The next step is to update weights in hidden layers. Let's update V_{ij} and Figure 2.8 shows the

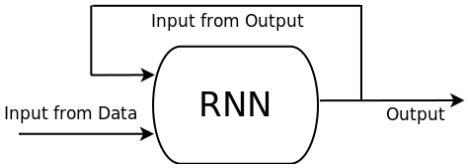


Figure 2.9: RNN

detail of neural network related with V_{ij}. The updated weight ${V_{ij}}^{next}$ can be calculated as follows:

$$ {V_{ij}}^{next} = V_{ij} - \eta \frac{\partial J}{\partial V_{ij}} $$

where η is a learning rate. The equation is similar as ${W_{ij}}^{next}$

Let's compute $\frac{\partial J}{\partial V_{ij}}$

$$ \frac{\partial J}{\partial V_{ij}} = \frac{\partial J}{\partial h'_i}\frac{\partial h'_i}{\partial V_{ij}} = \frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial h'_i}\frac{\partial h'_i}{\partial V_{ij}} $$

So the equation also has three parts.

First part is

$$ \frac{\partial h'_i}{\partial V_{ij}} = \frac{\partial}{\partial V_{ij}}\sum_k (X_k V_{ik}) = \frac{\partial}{\partial V_{ij}}(X_j V_{ij}) = X_j $$

Assume the activation function for hidden layers is also a sigmoid function $sigm()$ then the second part is

$$ \frac{\partial h_i}{\partial h'_i} = \frac{\partial}{\partial h'_i} sigm(h_i) = sigm(h'_i)\{1 - sigm(h'_i)\} = h_i(1 - h_i) $$

Then the last part is

$$ \frac{\partial J}{\partial h_i} = \sum_k (\frac{\partial J}{\partial o'_k}\frac{\partial o'_k}{\partial h_i}) = \sum_k (\delta_k w_{ki}) $$

Therefore,

$$ {V_{ij}}^{next} = V_{ij} - \eta\frac{\partial J}{\partial V_{ij}} = V_{ij} - \eta\frac{\partial J}{\partial h_i}\frac{\partial h_i}{\partial h'_i}\frac{\partial h'_i}{\partial V_{ij}} = V_{ij} - \eta\sum_k (\delta_k w_{ki}) h_i(1 - h_i) X_j $$

Notice that as forward-propagation sends output of a layer as input of the next layer, back-propagation sends the error of a the layer to the previous layer. For example, if the $\delta_i = \frac{\partial J}{\partial o'_i}$ is an error on the output layer, then it is sent to hidden layers.

2.1.2 Recurrent Neural Network

Basic concepts of recurrent neural networks:

A recurrent neural network (RNN) is a neural network that is specialized for processing a sequence of input values [**?**]. Figure 2.9 illustrates an abstract structure of RNN. RNN has two inputs. One input is from data such as a normal neural network input but the other input is from the previous output. This property makes recurrent neural networks be specially well suited to model sequential input data. This is because past outputs affect the current output. With sequential data, the previous data can affect the current data and RNN considers the previous outputs for the current output. This means that even if the inputs from the data are the same, if the previous outputs are different, the current output could also be different.

For example, human language sentences have series of words, and meanings of words are different depending on context. RNN can be used for that. Another example is driving data which is used

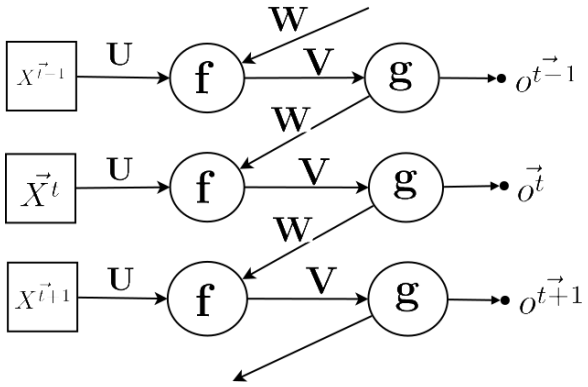


Figure 2.10: Unfolded RNN.

later on the paper. Driving data are multi-dimension data with time domain. RNN can be used for the data because it is sequential data with the time domain.

To describe RNN in a mathematical equation, let $\vec{x^t} = \{x_1^t, ..., x_n^t\} \in \mathbb{R}^n$ be a vector that represents the input data at time t, $\vec{h^t} = \{h_1^t, ..., h_m^t\} \in \mathbb{R}^m$ be result from hidden layer on time t, and $\vec{o^t} = \{o_1^t, ..., o_l^t\} \in \mathbb{R}^l$ be output on time t. For transform matrix, let $\mathbf{u} : \mathbb{R}^n \to \mathbb{R}^m$ be a transform matrix for input data, $\mathbf{v} : \mathbb{R}^m \to \mathbb{R}^l$ be a transform matrix for data from hidden layer, and $\mathbf{w} : \mathbb{R}^l \to \mathbb{R}^m$ be a transform matrix for previous output. Figure 2.10 illustrates an unfolded RNN with defined symbols. The result from hidden layer can be calculated by

$$\vec{h^t} = \mathbf{f}(\vec{X^t}\mathbf{U} + \vec{O^{t-1}}\mathbf{W})$$

where $\mathbf{f}$ is an activation function for the hidden layer. Then the output can be calculated by

$$\vec{o^t} = \mathbf{g}(\vec{H^t}\mathbf{V})$$

where $\mathbf{g}$ is an activation function for the output layer. Therefore, the RNN with bias is

$$\vec{o^t} = \mathbf{g}(\mathbf{f}(\vec{x^t}\mathbf{u} + \vec{b_x} + \vec{o^{t-1}}\mathbf{w} + \vec{b_o})\mathbf{v} + \vec{b_h})$$

where $\vec{b_x} = \{b_{x1}, ..., b_{xm}\} \in \mathbb{R}^m$ is the bias for the input data, $\vec{b_o} = \{b_{o1}, ..., b_{om}\} \in \mathbb{R}^m$ is the bias for the previous output data, and $\vec{b_h} = \{b_{h1}, ..., b_{hl}\} \in \mathbb{R}^l$ is the bias for the data from the hidden layers.

RNN is not always like Figure 2.10. Input from the previous output can be replaced by the result of the previous hidden layer. The main idea of RNN is that when neural network decides a current output, it considers a previous state.

Long Short Term Memory (LSTM) networks:

Basic RNNs are known to have problems modeling long term dependencies. When RNN passes the previous output for the current output, some information in the input might or might not be needed for future. RNN does not have an ability to filter unnecessary information or to store necessary information. Long Short Term Memory (LSTM) neural networks are precisely designed to handle these problems.

LSTM is a type of RNN introduced by Hochreiter and Schmidhuber [**?**]. LSTM solves long term dependency problems in RNN by memory cells. LSTM manages memory cells as a storage of knowledge. LSTM filters unnecessary information from memory cells and records necessary information on memory cells.

The structure of LSTM consists of four gates: forget gate, input gate, input modulation gate,

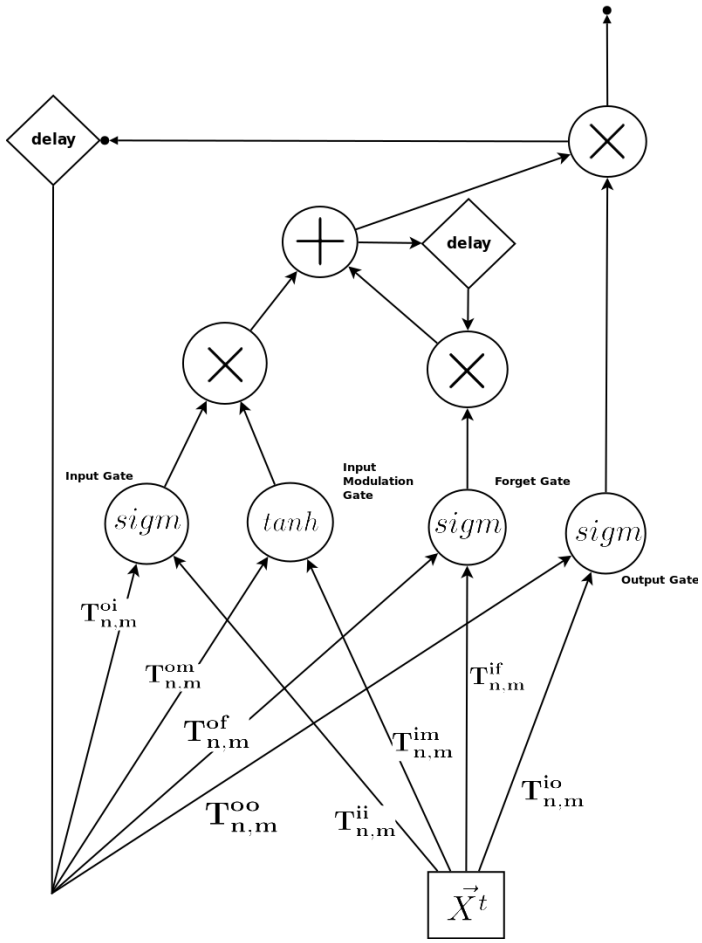


Figure 2.11: LSTM

and output gate. Each gates have different purposes and Christopher Olah's blog [1] develops an intuition for the concept of LSTM and explains the purpose of each gates. The paper [**?**] describes LSTM in a mathematical term. Let us first provide an intuitive description of how LSTM works by following Christopher Olah's blog, before providing a more in-depth formalization.

1. An intuitive explanation of LSTMs:

   The easiest way to understand LSTM is to understand memory cell and the purpose of four different gates. Memory cells are what make LSTM different from RNN. Memory cells store important information and filter unimportant information for future. Three of four gates are involved in the memory cells to store and filter information.

   The first gate affecting memory cells is a forget gate. The forget gate decides unnecessary data from input data and previous output then applies it to memory cells. Other two gates are an input gate and an input modulation gate and these also influence memory cells. The two gates decide what information to remember to apply to memory cells. The last gate is an output gate and it does not directly affect memory cells. The output gate also has two inputs from the input data and the previous output, and outputs from the output gate is multiplied by memory cells to make a final output. Figure 2.11 illustrates LSTM neural network with four gates and next part describes more detail of LSTM and the figure in a mathematical terms.

2. Modeling LSTMs in mathematical terms:

[1] `http://colah.github.io/posts/2015-08-Understanding-LSTMs`

To describe LSTM in a mathematical terms, let $\vec{x^t} = \{x_1^t, ..., x_n^t\} \in \mathbb{R}^n$ be a vector that represents the input data at time t, $\vec{i^t} = \{i_1^t, ..., i_m^t\} \in \mathbb{R}^m$ be a result from an input gate on time t, $\vec{m^t} = \{m_1^t, ..., m_m^t\} \in \mathbb{R}^m$ be a result from an input modulation gate on time t, $\vec{f^t} = \{f_1^t, ..., f_m^t\} \in \mathbb{R}^m$ be a result from a forget gate on time t, $\vec{o^t} = \{o_1^t, ..., o_m^t\} \in \mathbb{R}^m$ be a result from an output gate on time t, $\vec{c^t} = \{c_1^t, ..., c_m^t\} \in \mathbb{R}^m$ be memory cells on time t, and $\vec{h^t} = \{h_1^t, ..., h_m^t\} \in \mathbb{R}^m$ be a final result on time t. Each gate has transform matrices for inputs. Let $\mathbf{t_{n,m}} : \mathbb{R}^n \to \mathbb{R}^m$ be a transform matrix from n dimension to m dimension. Let $\mathbf{t^{ii}_{n,m}}$ be a transform matrix for the input from data in the input gate, $\mathbf{t^{oi}_{m,m}}$ be a transform matrix for the input from the previous output in the input gate, $\mathbf{t^{im}_{n,m}}$ be a transform matrix for the input from the data in the input modulation gate, $\mathbf{t^{om}_{m,m}}$ be a transform matrix for the input from the previous output in the input modulation gate, $\mathbf{t^{if}_{n,m}}$ be a transform matrix for the input from the data in the forget gate, $\mathbf{t^{of}_{m,m}}$ be a transform matrix for the input from the previous output in the forget gate, $\mathbf{t^{io}_{n,m}}$ be a transform matrix for the input from the data in the output gate, $\mathbf{t^{oo}_{m,m}}$ be a transform matrix for the input from the previous output in the output gate.

The results of each gate are computed as following:

$$\vec{i^t} = \mathbf{sigm}(\vec{X^t}\mathbf{T^{ii}_{n,m}} + \vec{H^{t-1}}\mathbf{T^{oi}_{m,m}})$$

The input gate uses a sigmoid function **sigm**() as an activation function

$$\vec{m^t} = \mathbf{tanh}(\vec{X^t}\mathbf{T^{im}_{n,m}} + \vec{H^{t-1}}\mathbf{T^{om}_{m,m}})$$

The input modulation gate uses a tanh function **tanh**() as an activation function.

$$\vec{f^t} = \mathbf{sigm}(\vec{X^t}\mathbf{T^{if}_{n,m}} + \vec{H^{t-1}}\mathbf{T^{of}_{m,m}})$$

The forget gate uses a sigmoid function **sigm**() as an activation function.

$$\vec{o^t} = \mathbf{sigm}(\vec{X^t}\mathbf{T^{io}_{n,m}} + \vec{H^{t-1}}\mathbf{T^{oo}_{m,m}})$$

The output gate uses a sigmoid function **sigm**() as an activation function.

Memory cells are computed as

$$\vec{c^t} = \vec{i^t} * \vec{m^t} + \vec{f^t} * \vec{c^{t-1}}$$

And final result is computed as

$$\vec{h^t} = \vec{c^t} * \vec{o^t}$$

Where $*$ is component-wise multiplication of two vectors.

LSTM neural network described above is one example of LSTMs. There are many other LSTMs but all LSTMs has memory cells to store information for long term memory and has four gates: input, input modulation, forget, and output gate.

2.1.3 Auto-encoder

This subsection introduces Auto-encoder and different kinds of Auto-encoder: undercomplete Auto-encoder, overcomplete Auto-encoder, and multilayer Auto-encoder.

Basic Auto-encoder

An Auto-encoder (AE) is defined as a neural network that is trained to attempt to copy its inputs to its outputs [**?**]. It means that AE takes inputs and sends it as outputs. However, AE does not directly send inputs to outputs. AE has two layers: a encode layer and a decode layer. Figure 2.12

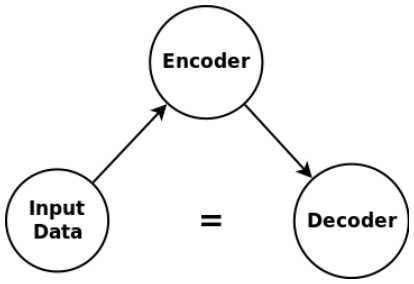


Figure 2.12: Abstract structure of Auto-encoder

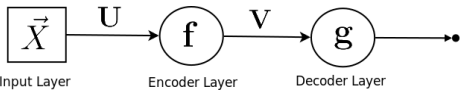


Figure 2.13: Basic Auto-encoder

illustrates an abstract structure of AE. The purpose of AE is to make outputs from a decode layer same as inputs.

To describe AE, let $\vec{x} = \{x_1, ..., x_n\} \in \mathbb{R}^n$ be an input, $\vec{e} = \{e_1, ..., e_m\} \in \mathbb{R}^m$ be an output from an encoder layer, and $\vec{d} = \{d_1, ..., d_n\} \in \mathbb{R}^n$ be an output from a decoder layer. For transform matrix, let $\mathbf{u} : \mathbb{R}^n \to \mathbb{R}^m$ be a transform matrix for an encode layer, $\mathbf{v} : \mathbb{R}^m \to \mathbb{R}^n$ be a transform matrix for a decode layer. Usually AE uses a sigmoid as an activation function. Figure 2.13 describes AE.

The output of encoder is

$$\vec{e} = \mathbf{f}(\vec{X}\mathbf{U})$$

where $\mathbf{f}$ is the activation function for encoder layer. It is usually a sigmoid function.

The output of decoder is

$$\vec{d} = \mathbf{g}(\vec{E}\mathbf{V})$$

where $\mathbf{g}$ is the activation function for decoder layer. It is usually a sigmoid or a linear function.

The purpose of AE copies the input as the output. Therefore, the error function is calculated by

$$E(\vec{x}, \vec{d}) = E(\vec{x}, \mathbf{g}(\vec{E}\mathbf{V})) = E(\vec{x}, \mathbf{g}(\mathbf{f}(\vec{x}\mathbf{u} + \vec{b_e})\mathbf{v} + \vec{b_d}))$$

Where $\vec{b_e} = \{b_{e1}, ..., b_{em}\} \in \mathbb{R}^m$ is the bias for the encoder layer and $\vec{b_d} = \{b_{d1}, ..., b_{dm}\} \in \mathbb{R}^n$ is the bias for the decoder layer.

AE seems not useful because it only tries to copy inputs. However, AE is usually used with other neural network that uses data from an encoder layer of AE, not from a decoder layer. Making values of an input and a decoder output similar guarantees that a result from an encode layer contains all information of an input. It means that AE can represent the same information of an input data in different dimension. When the dimension of an encoder is higher than the dimension of an input, it is called an overcomplete Auto-encoder. When the dimension of an encoder is lower than the dimension of an input, it is called an undercomplete Auto-encoder.

On this paper and experiment, an undercomplete AE is used for reducing an input dimension. An undercomplete AE is often compared with principal components analysis (PCA) because both methods reduce dimensions. The paper [**?**] compares undercomplete AE and PCA. The result on the paper is that an undercomplete AE could keep more information than PCA. It means that reducing data to low dimension by an undercomplete AE gives better performance. However, training AE takes long time than computing information gained from PCA.

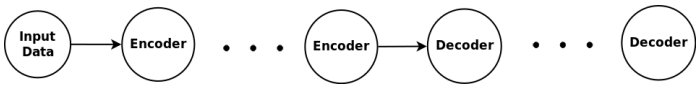


Figure 2.14: Multilayer Auto-encoder

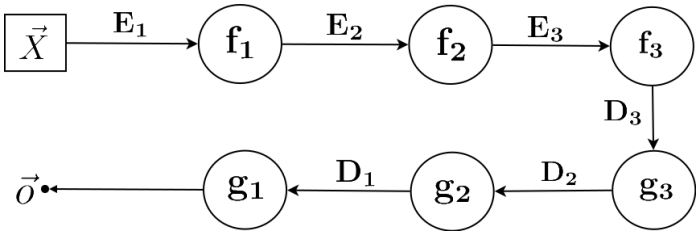


Figure 2.15: Multilayer Auto-encoder Example

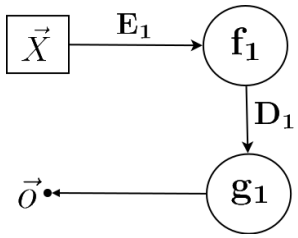


Figure 2.16: Pretraining First Step

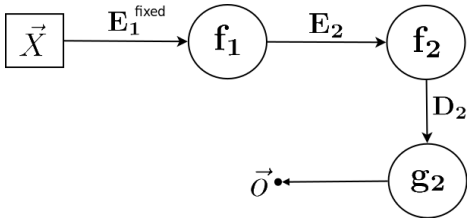


Figure 2.17: Pretraining Second Step

Multilayer Auto-encoder

Sometimes AE is designed with multiple encoder and decoder layers to reduce dimension. Figure 2.14 shows multilayer auto-encoder (MAE). However, it is difficult to find all the weights of encoders and decoders in MAE because all weights are initialized with random numbers and it makes difficult to find optimized weights [**?**]. So if the initial weights are close to optimized weights, training algorithm gradient descent can find optimized weights. The paper [**?**] shows "pretraining" method to initialize good weights.

Pretraining is to train each layer separately before training the whole layers. For example, Figure 2.15 has three encoder and decoder layers. Let $\vec{x} \in \mathbb{R}^n$ be an input vector, $\mathbf{e_1} : \mathbb{R}^n \to \mathbb{R}^m$ be a transform matrix for the first encoder layer, $\mathbf{e_2} : \mathbb{R}^m \to \mathbb{R}^l$ be a transform matrix for the second encoder layer, $\mathbf{e_3} : \mathbb{R}^l \to \mathbb{R}^k$ be a transform matrix for the third encoder layer, $\mathbf{d_1} : \mathbb{R}^m \to \mathbb{R}^n$ be a transform matrix for the first decoder layer, $\mathbf{d_2} : \mathbb{R}^l \to \mathbb{R}^m$ be a transform matrix for the second decoder layer, $\mathbf{d_3} : \mathbb{R}^k \to \mathbb{R}^l$ be a transform matrix for the third decoder layer, and $\vec{o} \in \mathbb{R}^n$ be an output vector.

In this case, pretraining has three steps because it has three encoder and decoder layers. Figure 2.16 shows train of first encoder and decoder layer. While the first encoder and decoder layer are

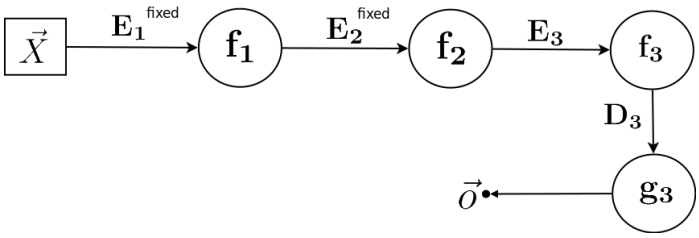


Figure 2.18: Pretraining Thrid Step

trained, the rest of encoders and decoders are ignored. So an error function is

$$E(\vec{x}, \mathbf{g_1}(\mathbf{f_1}(\vec{X}\mathbf{E_1})\mathbf{D_1}))$$

Figure 2.17 shows the training of the second encoder and decoder layer. While the second encoder and decoder layer are trained, the transform matrix $\mathbf{e_1}$ is fixed because the transform matrix $\mathbf{e_1}$ is to create input for second layer encoder. So an error function is

$$E(\mathbf{f_1}(\vec{X}\mathbf{E_1^{fixed}}), \mathbf{g_2}(\mathbf{f_2}(\mathbf{f_1}(\vec{X}\mathbf{E_1^{fixed}})\mathbf{E_2})\mathbf{D_2}))$$

The last step is Figure 2.18 and it trains the third encoder and decoder. During the training, the first and the second encoder weights are fixed. So an error function is

$$E(\mathbf{f_2}(\mathbf{f_1}(\vec{X}\mathbf{E_1^{fixed}})\mathbf{E_2^{fixed}}), \mathbf{g_3}(\mathbf{f_3}(\mathbf{f_2}(\mathbf{f_1}(\vec{X}\mathbf{E_1^{fixed}})\mathbf{E_2^{fixed}})\mathbf{E_3})\mathbf{D_3}))$$

After finishing pretraining step, all weights are close enough to optimal weights for training algorithm to find the optimized answer when it trains the whole neural network. So an error function training the whole network is

$$E(\vec{x}, \mathbf{g_1}(\mathbf{g_2}(\mathbf{g_3}(\mathbf{f_3}(\mathbf{f_2}(\mathbf{f_1}(\vec{X}\mathbf{E_1})\mathbf{E_2})\mathbf{E_3})\mathbf{D_3})\mathbf{D_2})\mathbf{D_1}))$$

2.2 Deep Learning Libraries

This section introduces three popular libraries for deep learning.

2.2.1 Tensorflow

Tensorflow [2] is an open source software library for numerical computation using data flow graphs. The biggest difference from other libraries is that Tensorflow treats all operators as nodes. For example, multiplication, addition, or sigmoid functions are treated as nodes. This helps developers to intuitively build neural networks. Another strength is the TensorBoard. TensorBoard is a tool for Tensorflow to visualize neural networks. Developers can also review how neural networks are trained from TensorBoard because TensorBoard visualizes logs to track all parameters while neural networks are trained.

The most famous example of the utilization of Tensorflow is AlphaGo [3]. A team of Google engineers built AlphaGo using Tensorflow and the neural network parts of AlphaGo run on Tensor Processing Units (TPU).

[2] `https://www.tensorflow.org/`

[3] `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`

2.2.2 Theano

Theano [4] is a Python library for machine learning. It integrates with Numpy and dynamically generates C code. Theano has many defined computations (called Op) and Theano allows to extend custom Ops written in C code.

Theano has more example code and is more stable than Tensorflow. This is because Tensorflow is newer than Theano and the APIs of Tensorlfow keep changing. For example, many APIs in Tensorflow 1.x have different interfaces compared to the APIs in Tensorflow 0.x.

2.2.3 DL4J

DL4J [5] is an open source, distributed deep-learning library for Java and Scala under the Apache 2.0 license. To compare with other deep learning libraries, DL4J can easily interact with Hadoop and Spark so neural networks built in DL4J can be used for big data.

2.3 Machine Learning with Driving Data

Due to its many practical applications and its importance to road safety, driving behavior modeling has been approached from many perspectives. A significant body of work, illustrated by the work of Macadam [**?**], exists on the manual creation of control models that exhibit specific aspects of human driving behavior (see Markkula et al. [**?**] for an in-depth review).

A number of approaches employ machine learning methods to automatically acquire models of human driving behavior [**?** **?**]. For example, case-based and instance-based learning have been deployed for driving tasks [**?**] as well as other LfD tasks [**?** **?** **?** **?**]. These approaches learn behavior by observing an expert's performance, but except for the work of Rubin and Watson, focus on small-scale datasets, much smaller than the datasets required to analyze driving behavior. Nechyba and Xu [**?**] developed a similarity measure based on Hidden Markov Models (HMM) [**?**]. The results demonstrated the ability of the models to accurately differentiate driving traces generated from one driver from those generated from other drivers however, the measures were based on relatively simple driving data (without other traffic). Moreover, the driving data used in this paper is significantly more complex than that used by Nechyba and Xu (for example, our data includes other traffic). In our previous work [**?**], we showed how choosing the right variables to model is key in accurately modeling driving behavior. For example, we showed that if the usual steering, throttle and brake variables are used as the target for prediction, then the learning problem becomes non-i.i.d.

Finally, previous work has utilized models to automatically predict driver states from driving data. Das et al. [**?**] showed that statistical measures over steering wheel data such as entropy and the Lyapunov exponent can determine whether a given driver is under the effect of alcohol or not.

[4] `http://deeplearning.net/software/theano/`
[5] `https://deeplearning4j.org/`

Chapter 3: Data Set

Data was collected in the high-fidelity simulator [**?**] of the Center for Injury Research Prevention Studies at the *Children's Hospital of Philadelphia* (CHOP). The driving simulator provides an environment similar to real driving to test users. It has a 160 degree front view, rear-view, left side, and right side mirror images. It also features a full car chassis with active pedals, steering wheel, and a full dashboard with even audio equipment.

The experiments on this paper use 16 traces from 4 drivers: 2 people were expert drivers and 2 people were inexpert drivers. Each person drove four different tracks and each track represents different traffic situations and interactions with other vehicles. Each track has multiple instances of three scenarios that have been found to result in a high likelihood of crashing for 16-18 year-old teen drivers driving alone or with a peer passenger according to the NMVCCS (National Motor Vehicle Crash Causation Survey): 1) turning into opposite directions (turning left), 2) right roadside departure, and 3) rear-end events [**?**]. Thus, this results on a dataset that contains 8 traces of expert drivers, and 8 traces of inexpert drivers.

The simulator records 100 features which include car status: velocity, steer, Brake, throttle and etc. and include environment status features such as the current speed limit, whether the driver is instructed to make the next left or right turn, etc. These data is collected at 60Hz. The traces vary in length from 26298 to 51295, with an average of 33224.6875 instances. The specific lengths of each trace are shown on Table 3.1

In this paper, we used two versions of the collected data set. A first version (*raw dataset*) contains 98 of the 100 features collected by the driving simulator (the two features that are removed are time stamps). A second version (*filtered dataset*) contains only 23 features. These 23 features were manually selected as are those that are most important for the classification task at hand. Reducing size of features helps save time to train neural network.

Table 3.1: Length of each of the traces used in our experiments.

Trace	*Driver*	*Track*	*Length*
Trace0	Expert0	Track0	50029
Trace1	Expert0	Track1	26375
Trace2	Expert0	Track2	29629
Trace3	Expert0	Track3	26298
Trace4	Expert1	Track0	51295
Trace5	Expert1	Track1	26674
Trace6	Expert1	Track2	29680
Trace7	Expert1	Track3	27075
Trace8	Inexpert0	Track0	49691
Trace9	Inexpert0	Track1	30058
Trace10	Inexpert0	Track2	26441
Trace11	Inexpert0	Track3	27373
Trace12	Inexpert1	Track0	47658
Trace13	Inexpert1	Track1	29380
Trace14	Inexpert1	Track2	26684
Trace15	Inexpert1	Track3	27255

Chapter 4: Technical Approach

This chapter covers the technical methods designed for the experiments reported later in this paper and also introduces three different neural network models used for these experiments.

4.1 Cross Validation

Cross validation (CV) is a common method used to validate models when data is not large enough [**?**]. CV divides a data set into K folds and then uses the ith fold as the test set and other folds as the training set to the target model. Experiments on this paper use 16 full traces but this has shown not to be enough to train the complex neural networks that we studied. Thus, CV was used to validate neural network models and to compare performance of these different models. The 16 traces dataset is divided to 4 folds. Each fold contain two traces from expert and two traces from inexpert. The neural networks are thus trained four times with different training sets and test sets. Figure 4.1 shows four folds cross validation. Red color is the fold for testing and other folds are for training. All the results presented in this paper are the result of repeating a 4-fold CV four times, and computing the average result.

4.2 LSTM

The LSTM neural network is used on three different neural network models to classify expert or inexpert drivers because the data has time domain and LSTM gives good performance for serial data. On the paper, LSTM neural networks are built with 16, 32, 64, 128, and 256 hidden neurons. The output from LSTM is sent to an output layer which has two additional neurons. By using softmax, the output from two neurons is classified. If it is [0, 1], it is classified as expert. Otherwise, it is classified as inexpert.

Figure 4.2 shows first neural network model (*first model*) on the paper. Its input is *filtered dataset* which has 23 chosen features. The *filtered dataset* is feed directly to LSTM then the result is passed to the output layer.

4.3 Auto-encoder

The *raw dataset* has 98 features and it takes too much time to train nerual network if all 98 features are used. AE can solve the problem by reducing dimensions.

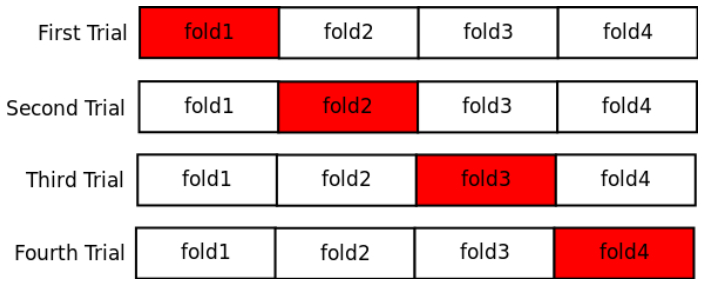


Figure 4.1: Cross Validation

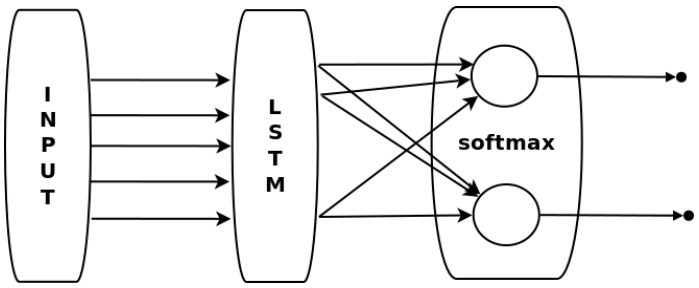


Figure 4.2: First experiment NN

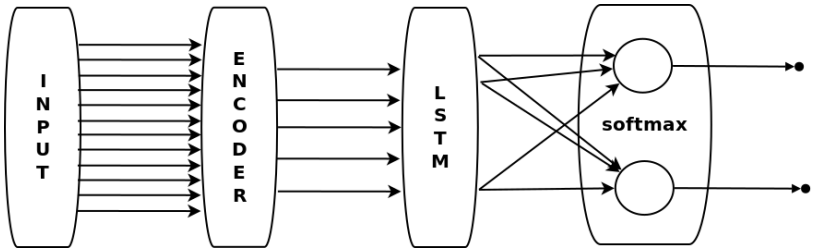


Figure 4.3: Second experiment NN

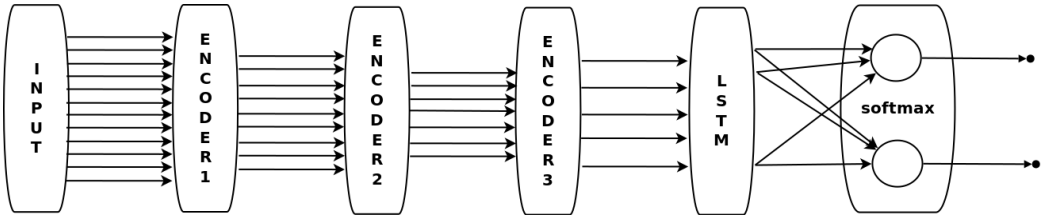


Figure 4.4: Third experiment NN

4.3.1 Single layer Auto-encoder

Figure 4.3 shows the second neural network model (*second model*) on the paper. The *second model* is added by one AE layer from *first model*. The purpose of the AE layer is to reduce 98 features to 25 features on *raw dataset*. Notice that the figure has an encoder only because after training AE, only an encoder is used to reduce dimensions. The output from the encoder is passed to LSTM.

4.3.2 Multiple layer Auto-encoder

Figure 4.4 shows third neural network model (*third model*) on the paper. To compare the performance between a single layer AE and multiple layer AE, the *third model* has three layers of AE. THe first AE reduces 98 dimensions to 75 dimensions, the second AE reduces 75 dimensions to 50 dimensions and the last AE reduces 50 dimensions to 25 dimensions. The figure describes it with three encoder layers.

4.4 Sampling

The simulator records 60 samples per second, which is too high a frequency. Thus, we re-sampled the dataset using three different periods: taking 1 out of each 10 samples, 1 out of each 20, and 1 out of each 50. For each period, we tested three different re-sampling methods. The first method *last* chooses the last sample of period, the second method *mean* computes mean of data in period, and the third method *gaussian* applies Gaussian filter. For the third method, window size is 11, 21,

Table 4.1: Example Data

Example Data
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

Table 4.2: Result of Re-Sampled Data

last	*mean*	*gaussian*
10	5.5	5.999996063
20	15.5	15.9999895

and 51 for each periods which are one more greater than periods. The example below explains how to re-sample data.

Table 4.1 shows example data and Table 4.2 shows result of re-sampled data by 1 over 10 from each methods. To explain three re-sample methods, let *first period* be $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ and *second period* be $(11, 12, 13, 14, 15, 16, 17, 18, 19, 20)$. The *last* method takes last sample from each periods. The result from *last* method is $(10, 20)$ because last sample from *first period* is 10 and last sample from *second period* is 20. The *mean* method computes mean of samples in a period. The result from *mean* is $(5.5, 15.5)$ because mean of *first period* is 5.5 and mean of *second period* is 15.5.

Before explaining *gaussian* method, Gaussian filter should be defined. The filter size is greater than sampling size by 1. In this case, Gaussian filter size is 11 because sampling size is 10. Below equation is Gaussian equation and used to make Gaussian filter.

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

where $\sigma = \frac{filter\_size-1}{6.0}$ because the sum of filter closes to 1 with the σ. With filter size 11, σ is 1.667.

Table 4.3 shows size 11 Gaussian filter with $\sigma = 1.667$. Re-sampling example data by the Gaussian filter is just multiplication between filter and data. Table 4.4 shows Gaussian filtered values for *first period*. The re-sampled value from *first period* is sum of Gaussian filtered values. That is 5.999996063. Table 4.5 shows Gaussian filtered values for *second period*. The re-sampled value is 15.9999895.

Table 4.3: Gaussian Filter (size=11, σ=1.667)

x	Gaussian Value
5	0.00266126292
4	0.01344760189
3	0.04740846466
2	0.1166060072
1	0.2000967085
0	0.2395592537
-1	0.2000967085
-2	0.1166060072
-3	0.04740846466
-4	0.01344760189
-5	0.00266126292

Table 4.4: First Period applied by Gaussian Filter

First Period	Gaussian Value	Filtered Value
1	0.00266126292	0.00266126292
2	0.01344760189	0.02689520378
3	0.04740846466	0.142225394
4	0.1166060072	0.4664240287
5	0.2000967085	1.000483542
6	0.2395592537	1.437355522
7	0.2000967085	1.400676959
8	0.1166060072	0.9328480573
9	0.04740846466	0.4266761819
10	0.01344760189	0.1344760189
11	0.00266126292	0.02927389212
Sum		5.999996063

Table 4.5: Second Period applied by Gaussian Filter

First Period	Gaussian Value	Filtered Value
11	0.00266126292	0.02927389212
12	0.01344760189	0.1613712227
13	0.04740846466	0.6163100406
14	0.1166060072	1.6324841
15	0.2000967085	3.001450627
16	0.2395592537	3.832948059
17	0.2000967085	3.401644044
18	0.1166060072	2.098908129
19	0.04740846466	0.9007608285
20	0.01344760189	0.2689520378
21	0.00266126292	0.05588652132
Sum		15.9999895

4.5 Normalization

After feeding re-sampled data to three neural network models, the data is normalized: mean is 0, standard deviation is 1. CV is used for all experiments. Therefore, normalization is done in a training set without a testing set in CV. This is because the testing set should be hidden until final neural network models are applied on that. The testing set refers mean and standard deviation from the training set to normalize.

Chapter 5: Experimental Result

This chapter summarizes results from all experiments. Experiments are divided to two parts: 1) experiments with *filtered dataset* 2) experiments with *raw dataset*. First part of experiments used *first model* which only has LSTM. Each experiments used 5 different number of neurons: 16, 32, 64, 128, and 256. Second part of experiments used *second model* which has SAE and *third model* which has MAE then compared result of three models.

5.1 Experiments with *filtered dataset*

The first part of experiments used *filtered dataset* which has chosen 23 features. The dataset was re-sampled by 1 over 10, 1 over 20, and 1 over 50 with three different methods: *last*, *mean*, and *gaussian* so it was 9 different re-sampled datasets.

5.1.1 Resample 1 over 50

This subsection compares results from experiments with 1 over 50 re-sampled *filtered dataset* by *last*, *mean*, and *gaussian* methods. Each dataset were tested four times with different number of neurons in LSTM hidden layer: 16, 32, 64, 128, and 256 neurons. Figure 5.1 shows average accuracy from four tests on different number of neurons. Table 5.1 shows average accuracy of four tests on different number of neurons, Table 5.2 shows standard deviation of four tests on different number of neurons, and Table 5.3 shows test time of four tests on different number of neurons.

The best performance was from *last* method with 16 number of neurons. The accuracy of this was 0.59375 (59.375%). The worst performance was from *gaussian* method with 64 number of neurons. It was 0.40625 (40.625%). Test with 64 neurons gave most stable result because standard deviation of this was the smallest. Many accuracy were below 0.5 (50%) and total average accuracy is 0.492708 (49.2708%). The experiments with 1 over 50 re-sampled data took short time than experiments with 1 over 20 and 1 over 10 re-sampled data on later of this chapter.

5.1.2 Resample 1 over 20

This subsection compares results from experiments with 1 over 20 re-sampled *filtered dataset* by *last*, *mean*, and *gaussian* methods. Each dataset were tested four times with different number of neurons in LSTM hidden layer: 16, 32, 64, 128, and 256 neurons. Figure 5.2 shows average accuracy from four tests on different number of neurons. Table 5.4 shows average accuracy of four tests on different number of neurons, Table 5.5 shows standard deviation of four tests on different number of neurons, and Table 5.6 shows test time of four tests on different number of neurons.

The best performance was from *mean* method with 32 and 256 number of neurons. The accuracy of this was 0.59375 (59.375%) as same as the best performance from experiments with 1 over 50 re-sampled data. The worst performance was also from *mean* method with 128 number of neurons.

Table 5.1: Average Accuracy of Result from 1 over 50 Re-sampled *filtered dataset*

1 over 50	The number of neurons				
	16	32	64	128	256
last	0.59375	0.484375	0.546875	0.5625	0.515625
mean	0.421875	0.5	0.484375	0.421875	0.484375
gaussian	0.578125	0.421875	0.40625	0.515625	0.453125
Average	0.53125	0.46875	0.4791666667	0.5	0.484375

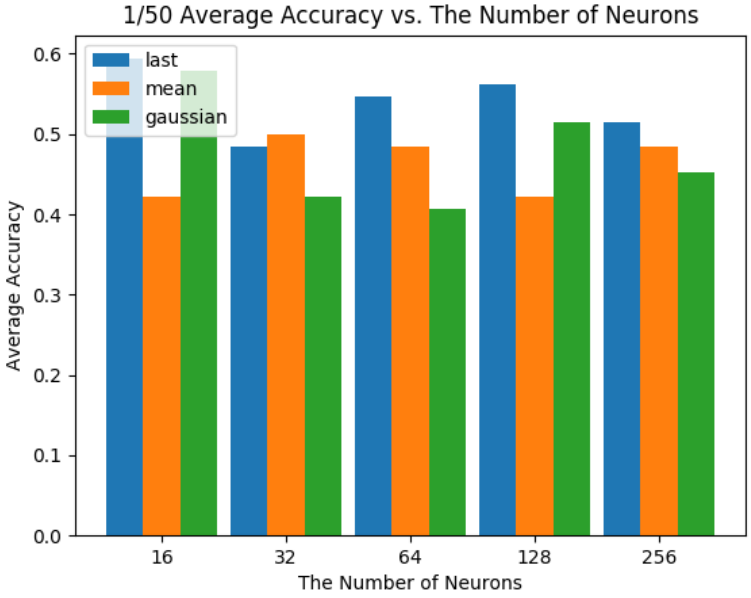


Figure 5.1: Result of 1 over 50 Re-sampled *filtered dataset*

Table 5.2: Standard Deviation of Result from 1 over 50 Re-sampled *filtered dataset*

1 over 50	The number of neurons				
	16	32	64	128	256
last	0.1983000672	0.2536196299	0.1547847968	0.249478623	0.2452677109
mean	0.218303115	0.1825741858	0.213478141	0.1983000672	0.2656556355
gaussian	0.1983000672	0.2536196299	0.1547847968	0.249478623	0.2452677109
Average	0.2049677498	0.2299378152	0.1743492449	0.2324191044	0.2520636858

It was 0.375 (37.5%), which was worse than the worst performance from experiments with 1 over 50 re-sampled data. Test with 16 neurons gave most stable result because standard deviation of this was the smallest. Test with 64 neurons was stable because its standard deviation was similar as standard deviation of test with 16 neurons. The total average was 0.515625 (51.5625%) which was slightly higher than the total average 0.492708 (49.2708%) from experiments with 1 over 50 re-sampled data because size of 1 over 20 re-sampled data was 2.5 times more than size of 1 over 50 re-sampled data. More amount of data also affected the test time. On the experiments, when size of the data was 2.5 times more, the test time took little bit more than 2.5 times than the test time from experiments with 1 over 50 re-sampled data.

Table 5.3: Test Time from 1 over 50 Re-sampled *filtered dataset*

1 over 50	The number of neurons				
	16	32	64	128	256
last	0:17:03	0:18:48	0:22:43	0:35:43	0:51:25
mean	0:17:06	0:18:17	0:23:10	0:35:23	0:50:42
gaussian	0:16:49	0:18:20	0:22:28	0:35:38	0:52:43
Average	0:16:59	0:18:28	0:22:47	0:35:35	0:51:37

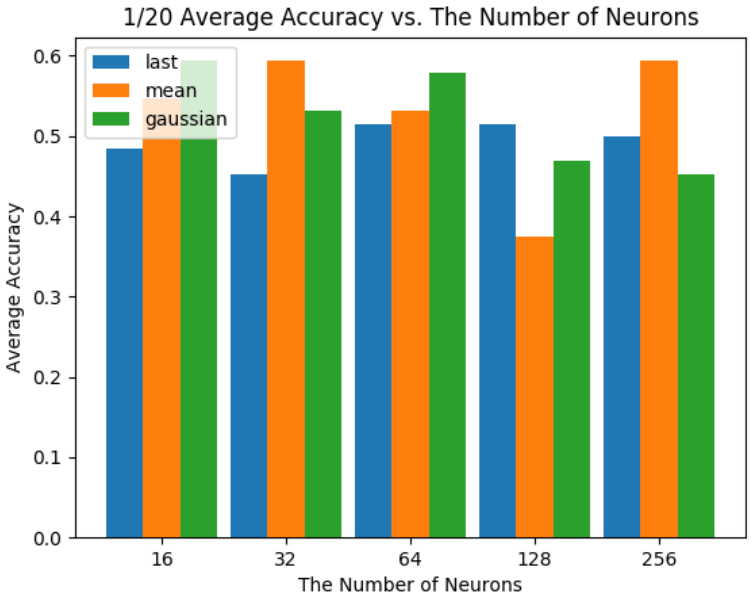


Figure 5.2: Result of 1 over 20 Re-sampled *filtered dataset*

Table 5.4: Average Accuracy of Result from 1 over 20 Re-sampled *filtered dataset*

1 over 20	The number of neurons				
	16	32	64	128	256
last	0.484375	0.453125	0.515625	0.515625	0.5
mean	0.546875	0.59375	0.53125	0.375	0.59375
gaussian	0.59375	0.53125	0.578125	0.46875	0.453125
Average	0.5416666667	0.5260416667	0.5416666667	0.453125	0.515625

Table 5.5: Standard Deviation of Result from 1 over 20 Re-sampled *filtered dataset*

1 over 20	The number of neurons				
	16	32	64	128	256
last	0.1929756029	0.2617051458	0.213478141	0.213478141	0.2415229458
mean	0.1875	0.2561737691	0.1547847968	0.2581988897	0.2393567769
gaussian	0.2393567769	0.3145764348	0.2536196299	0.2015564437	0.2617051458
Average	0.2066107933	0.2774851166	0.2072941892	0.2244111581	0.2475282895

Table 5.6: Test Time from 1 over 20 Re-sampled *filtered dataset*

1 over 20	The number of neurons				
	16	32	64	128	256
last	0:53:22	0:58:52	1:30:51	2:15:33	3:49:07
mean	0:48:29	0:59:33	1:27:01	2:14:11	3:46:40
gaussian	0:49:00	0:57:27	1:28:35	2:18:22	3:47:48
Average	0:50:17	0:58:37	1:28:49	2:16:02	3:47:52

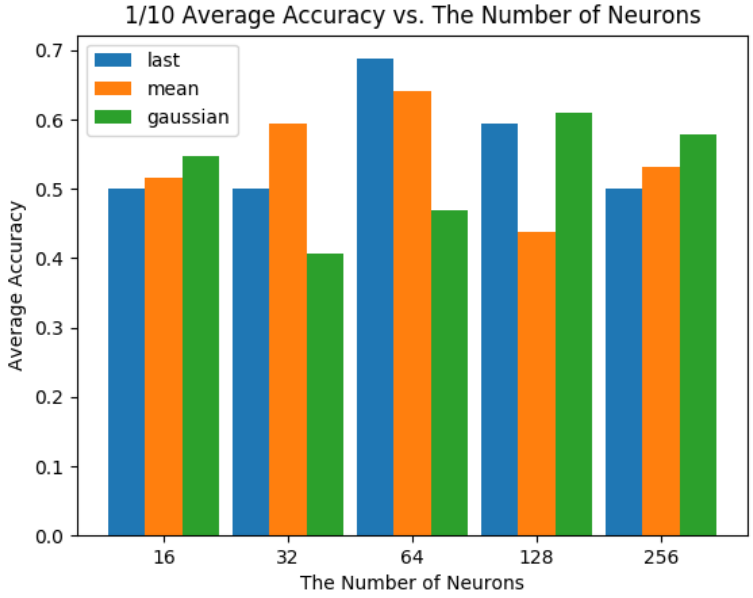


Figure 5.3: Result of 1 over 10 Re-sampled *filtered dataset*

Table 5.7: Average Accuracy of Result from 1 over 10 Re-sampled *filtered dataset*

1 over 10	The number of neurons				
	16	32	64	128	256
last	0.5	0.5	0.6875	0.59375	0.5
mean	0.515625	0.59375	0.640625	0.4375	0.53125
gaussian	0.546875	0.40625	0.46875	0.609375	0.578125
Average	0.5208333333	0.5	0.5989583333	0.546875	0.5364583333

5.1.3 Resample 1 over 10

This subsection compares results from experiments with 1 over 10 re-sampled *filtered dataset* by *last*, *mean*, and *gaussian* methods. Each dataset were tested four times with different number of neurons in LSTM hidden layer: 16, 32, 64, 128, and 256 neurons. Figure 5.3 shows average accuracy from four tests on different number of neurons. Table 5.7 shows average accuracy of four tests on different number of neurons, Table 5.8 shows standard deviation of four tests on different number of neurons, and Table 5.9 shows test time of four tests on different number of neurons.

The best performance was from *last* method with 64 number of neurons. The accuracy of this was 0.6875 (68.75%) and it was also the best accuracy from all experiments. The worst performance was also from *gaussian* method with 32 number of neurons. It was 0.40625 (40.625%). The total average was 0.540625 (54.0625%). Test with 32 neurons gave the smallest standard deviation.

Overall performance from experiments with 1 over 10 gave better result than from experiments with 1 over 20 and 1 over 50 re-sampled data. This is because 1 over 10 re-sampled data had largest amount of data and it caused better performance. However, test time increased as size of data increased. Especially, experiments with 256 neurons in LSTM hidden layer took average 6 hours 48 miniatures and total time from 12 experiments with 256 neurons took more than 3 days.

Table 5.8: Standard Deviation of Result from 1 over 10 Re-sampled *filtered dataset*

1 over 10	The number of neurons				
	16	32	64	128	256
last	0.2581988897	0.2041241452	0.2140872096	0.2015564437	0.2236067977
mean	0.1929756029	0.2015564437	0.2230237282	0.2140872096	0.1547847968
gaussian	0.2085415626	0.1796988221	0.2393567769	0.2576941016	0.2366211811
Average	0.2199053517	0.1951264703	0.2254892382	0.2244459183	0.2050042585

Table 5.9: Test Time from 1 over 10 Re-sampled *filtered dataset*

1 over 10	The number of neurons				
	16	32	64	128	256
last	2:11:58	2:21:11	3:04:34	4:28:24	6:35:52
mean	1:59:20	2:24:28	3:05:15	4:26:12	6:57:59
gaussian	2:00:20	2:22:21	3:05:58	4:29:00	6:52:17
Average	2:03:53	2:22:40	3:05:16	4:27:52	6:48:43

Table 5.10: Accuracy of three models with *last* method

Average	The number of neurons					
Model	16	32	64	128	256	Average
first	0.5	0.5	0.6875	0.59375	0.5	0.55625
second	0.421875	0.359375	0.34375	0.28125	0.28125	0.3375
third	0.359375	0.46875	0.453125	0.40625	0.453125	0.428125

5.2 Experiments with *raw dataset*

The second part of experiments used *raw dataset* which has 98 features. The dataset was re-sampled only by 1 over 10 with *last* method. This is because the experiment time with *second model* with SAE and *third model* with MAE took much more than the experiment time with *first model*, and from the result of the first part of experiments, the best overall performance was from 1 over 10 re-sampled data and the best performance was from *last* method. The *second model* and *third model* were also tested four times with different number of neurons in LSTM hidden layer: 16, 32, 64, 128, and 256 neurons. The purpose of the second part of experiments is performance comparison between three different ways to reduce dimensions: human(*first model*), SAE(*second model*), and MAE(*third model*).

5.2.1 Comparison of results from three models

Figure 5.4 compares accuracy of three models with data reduced dimensions by human, SAE, and MAE respectively. Table 5.10 shows accuracy of three models, Table 5.11 shows detail standard deviation of three models, and Table 5.12 shows test time of three models.

The overall average accuracy from SAE is 0.3375 (33.75%) and the overall average accuracy from MAE is 0.428125 (42.8125%). Both performance are much lower than 0.55625 (55.625%) performance from the experiments with data filtered by human. Next subsection explains the reasons why SAE and MAE gave worse performance.

5.2.2 Analysis Training LSTM in *second model* and *third model*

The difference between *first model* and other models is that other models have AE. Therefore, AE layer from *second model* and *third model* caused worse performance. The evidence of this can be found during training LSTM in both models. Following six figures help to visualize it. Figure 5.5

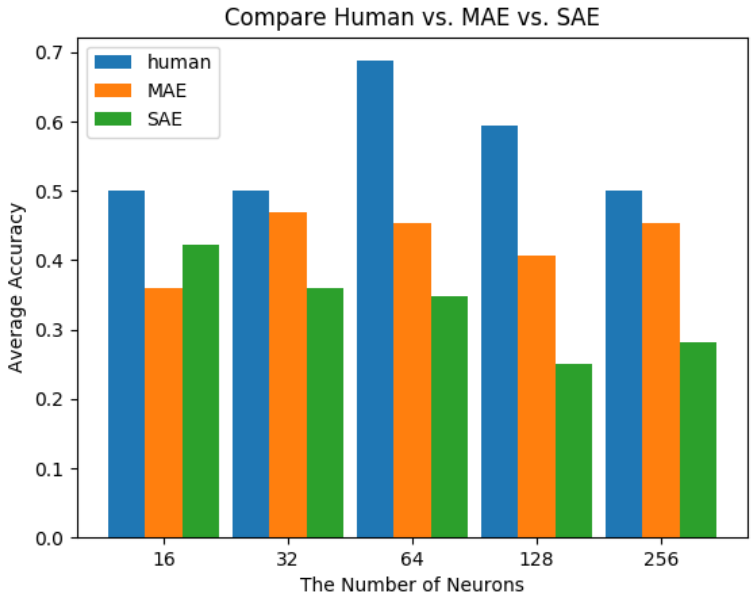


Figure 5.4: Result of three models with *last* method

Table 5.11: Standard Deviation of three models with *last* method

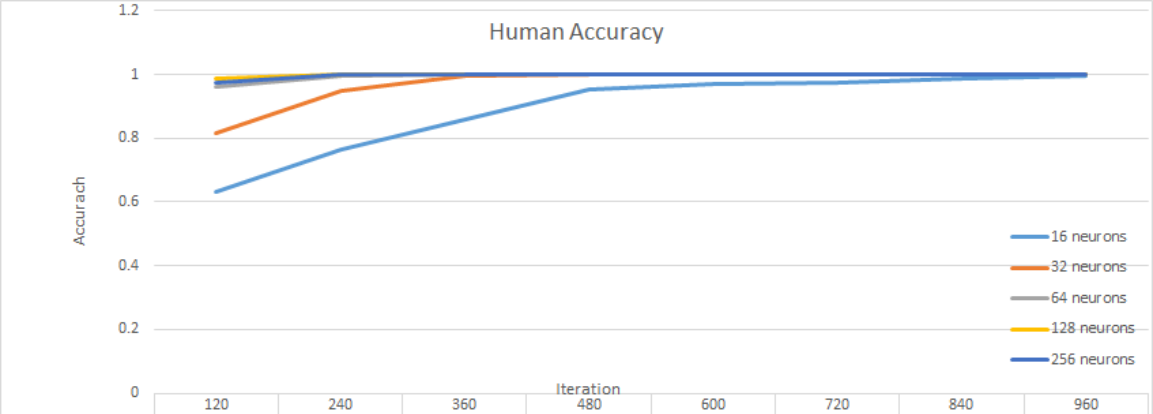
Std.	The number of neurons					
Model	16	32	64	128	256	Average
first	0.2581988897	0.2041241452	0.2140872096	0.2015564437	0.2236067977	0.2203146972
second	0.1983000672	0.2576941016	0.2212653008	0.2212653008	0.2719528145	0.234095517
third	0.2733854117	0.2561737691	0.2085415626	0.2015564437	0.1359764073	0.2151267189

Table 5.12: Test Time of three models with *last* method

Std.	The number of neurons					
Model	16	32	64	128	256	Average
first	2:11:58	2:21:11	3:04:34	4:28:24	6:35:52	3:44:24
second	3:55:11	4:26:36	5:45:11	6:21:02	8:33:53	5:48:23
third	6:41:47	6:23:12	7:11:05	8:17:21	10:27:01	7:48:05

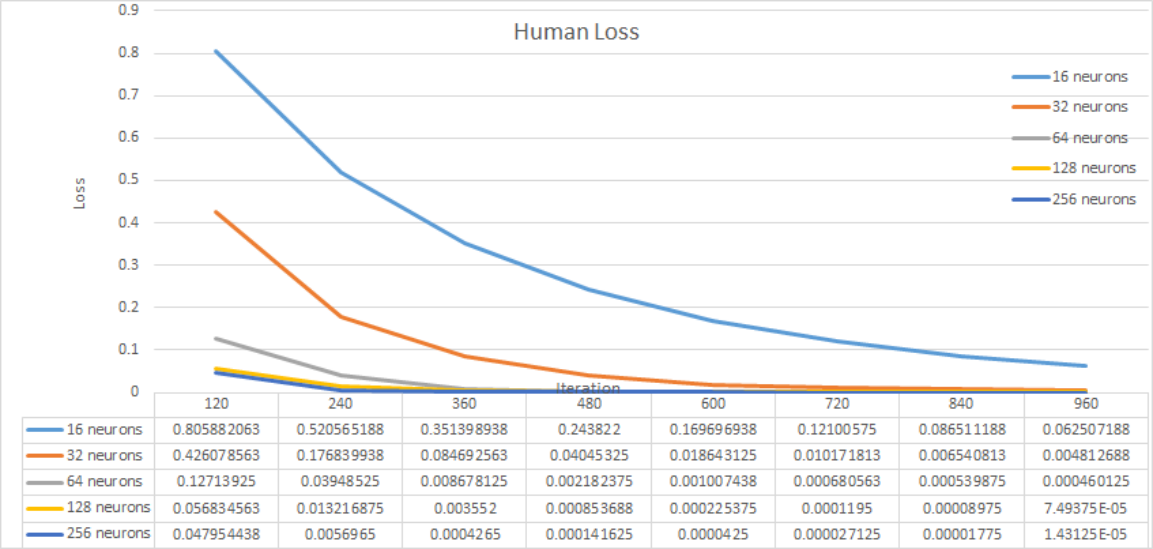
and Figure 5.6 respectively show the accuracy and loss from LSTM of *first model* on each iterations during training. Figure 5.7 and Figure 5.8 respectively show the accuracy and loss from LSTM of *second model* on each iterations during training. Figure 5.9 and Figure 5.10 respectively show the accuracy and loss from LSTM of *third model* on each iterations during training.

Training LSTM in *second model* and *third model* were slow and most cases did not reach accuracy 1 (100%) while the training LSTM in *first model* easily reached accuracy 1 (100%). Also loss of *second model* and *third model* did not close to 0. The *third model* gave increasing loss when it was trained more. It implies that the data fed to LSTM had noise or problems. It also means that AE in *second model* and *third model* did not work correctly. The next subsection covers more detailed explanation about the error from AE in *second model* and *third model*.



	120	240	360	480	600	720	840	960
16 neurons	0.630208125	0.765624375	0.859375	0.95312625	0.96875125	0.973959375	0.98958375	0.994791875
32 neurons	0.817708125	0.947918125	0.994791875	1	1	1	1	1
64 neurons	0.96354125	0.994791875	1	1	1	1	1	1
128 neurons	0.98958375	1	1	1	1	1	1	1
256 neurons	0.973959375	1	1	1	1	1	1	1

Figure 5.5: Human Training Accuracy



	120	240	360	480	600	720	840	960
16 neurons	0.805882063	0.520565188	0.351398938	0.243822	0.169696938	0.12100575	0.086511188	0.062507188
32 neurons	0.426078563	0.176839938	0.084692563	0.04045325	0.018643125	0.010171813	0.006540813	0.004812688
64 neurons	0.12713925	0.03948525	0.008678125	0.002182375	0.001007438	0.000680563	0.000539875	0.000460125
128 neurons	0.056834563	0.013216875	0.003552	0.000853688	0.000225375	0.0001195	0.00008975	7.49375E-05
256 neurons	0.047954438	0.0056965	0.0004265	0.000141625	0.0000425	0.000027125	0.00001775	1.43125E-05

Figure 5.6: Human Training Loss

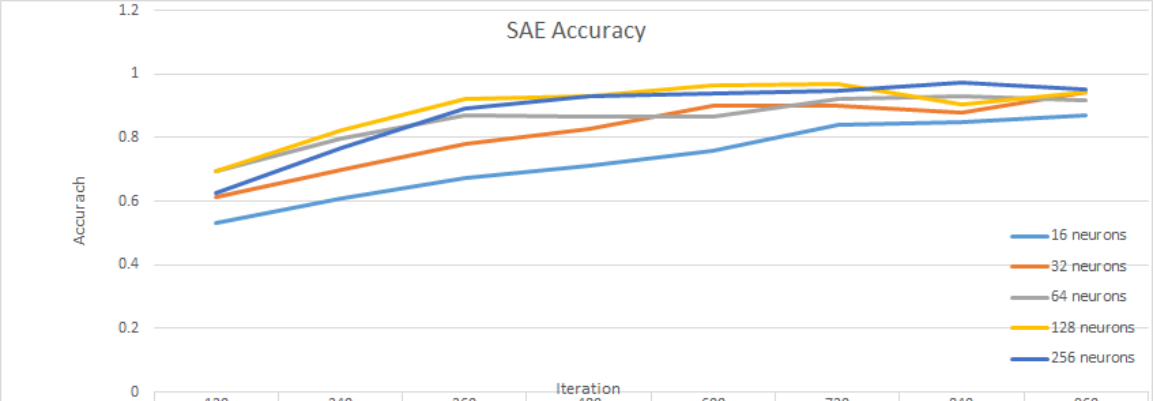
5.2.3 Error from AE in *second model* and *third model*

Table 5.13 and Table 5.14 show error from AE. We repeated each experiment for times (Test), and we also show the error in each of the folds of the cross validation run (CV). The error function $E()$ is defined as following:

$$E(\vec{x}, \vec{d}) = \sum_{k} (x_i - d_i)^2$$

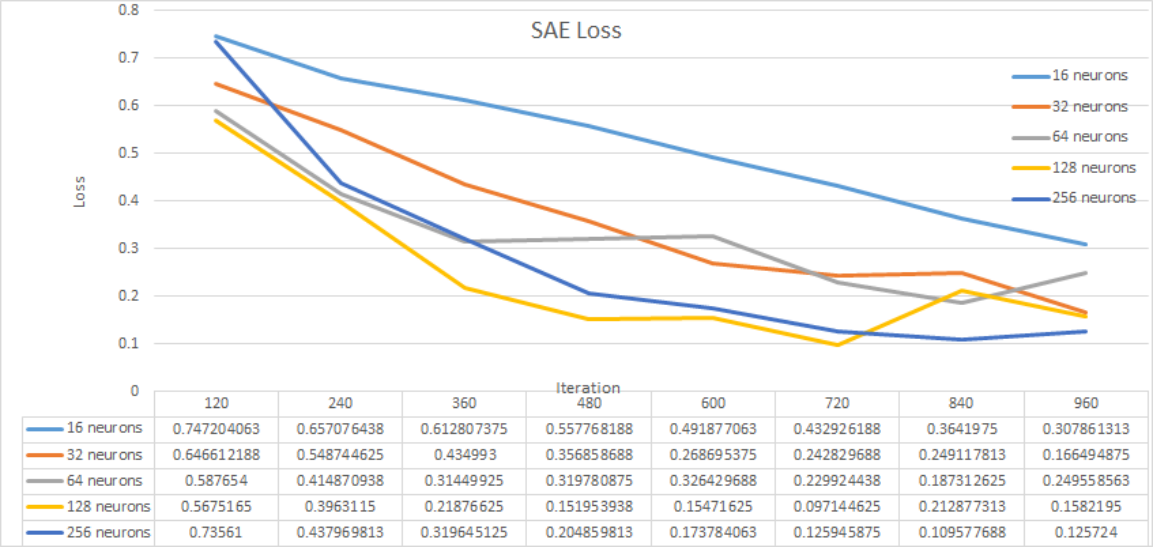
where $\vec{x}$ is input signal, $\vec{d}$ is output signal from last decoder. The total average error from AE in *second model* is 0.1587907 and the total average error from AE in *third model* is 0.1720756. Both error are big enough to affect performance because most range of normalized data for experiments is between 3 and -3. Figure 5.11, Figure 5.12, Figure 5.13, and Figure 5.14 help visualize error.

Such big error between original input and output from decode layer implies that data from final encode layer also has big error. The error in encoded data can interrupt train followed neural



	120	240	360	480	600	720	840	960
16 neurons	0.531249375	0.609375625	0.671875625	0.713540625	0.760415625	0.838541875	0.84895875	0.8697925
32 neurons	0.614583125	0.69791625	0.78125125	0.828125625	0.901041875	0.90104125	0.880208125	0.942708125
64 neurons	0.692708125	0.796875625	0.869793125	0.86458375	0.86458375	0.921875625	0.9322925	0.916666875
128 neurons	0.69270875	0.822915625	0.921874375	0.9322925	0.9635425	0.968750625	0.90625	0.942708125
256 neurons	0.625000625	0.765625	0.890625	0.932291875	0.93750125	0.947916875	0.973959375	0.95312625

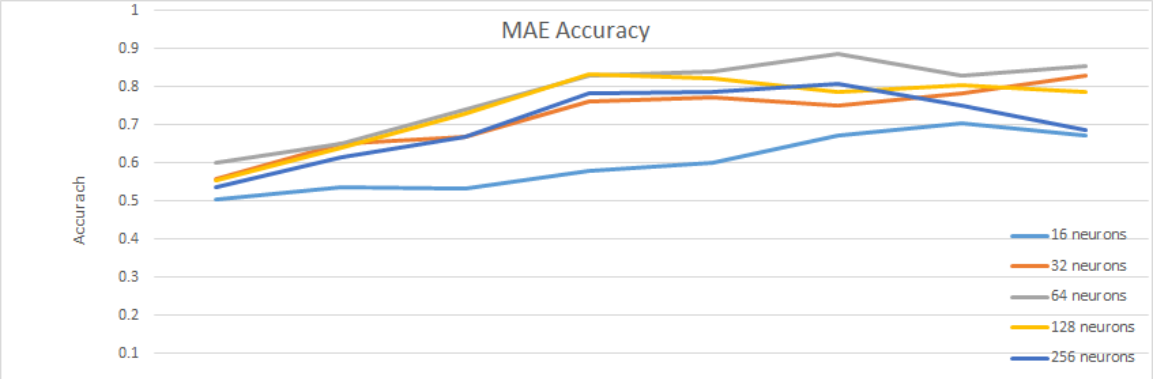
Figure 5.7: SAE Training Accuracy



	120	240	360	480	600	720	840	960
16 neurons	0.747204063	0.657076438	0.612807375	0.557768188	0.491877063	0.432926188	0.3641975	0.307861313
32 neurons	0.646612188	0.548744625	0.434993	0.356858688	0.268695375	0.242829688	0.249117813	0.166494875
64 neurons	0.587654	0.414870938	0.31449925	0.319780875	0.326429688	0.229924438	0.187312625	0.249558563
128 neurons	0.5675165	0.3963115	0.21876625	0.151953938	0.15471625	0.097144625	0.212877313	0.1582195
256 neurons	0.73561	0.437969813	0.319645125	0.204859813	0.173784063	0.125945875	0.109577688	0.125724

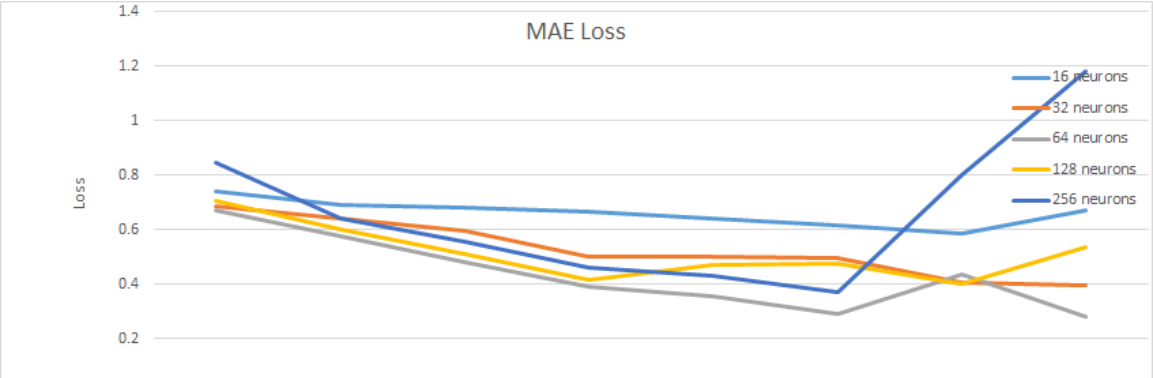
Figure 5.8: SAE Training Loss

networks. In this case, the error interrupts LSTM layer.



	120	240	360	480	600	720	840	960
16 neurons	0.50520875	0.5364575	0.531249375	0.578125	0.598959375	0.67187375	0.703125625	0.671875625
32 neurons	0.55729125	0.65104125	0.666666875	0.7604175	0.770833125	0.75	0.78124875	0.828125
64 neurons	0.59895875	0.651041875	0.739583125	0.828125625	0.838540625	0.885416875	0.828124375	0.854166875
128 neurons	0.552083125	0.640625	0.72916625	0.8333325	0.822916875	0.78645875	0.80208375	0.786458125
256 neurons	0.536458125	0.614583125	0.666665625	0.78125	0.78645875	0.8072925	0.75	0.6875

Figure 5.9: MAE Training Accuracy



	120	240	360	480	600	720	840	960
16 neurons	0.740459438	0.691438125	0.681954625	0.666309188	0.64164775	0.614527625	0.583129938	0.66829875
32 neurons	0.686523063	0.63892425	0.596218375	0.500950688	0.49947975	0.494506125	0.403456313	0.394951063
64 neurons	0.669182938	0.574360188	0.482696375	0.391021625	0.357045938	0.2914615	0.43715225	0.281894313
128 neurons	0.703381625	0.59838225	0.510078188	0.417459438	0.472507063	0.47559575	0.401782438	0.533872875
256 neurons	0.845653688	0.641895	0.556838125	0.462811063	0.430349188	0.368360938	0.80201525	1.181336688

Figure 5.10: MAE Training Loss

Table 5.13: Error from AE in *second model*

Test	CV	The number of neurans				
		16	32	64	128	256
1	1	0.178460395	0.1239250116	0.1748844814	0.1502201539	0.1886790823
	2	0.1658136155	0.1384476442	0.102677634	0.2454849277	0.1165074687
	3	0.1409358419	0.1701477412	0.1447305065	0.1517742854	0.1233751141
	4	0.13444083	0.1528848018	0.1856264398	0.1555761881	0.1862711851
2	1	0.1573844403	0.1424119417	0.1411998086	0.1525888257	0.1580447592
	2	0.141391445	0.1517100111	0.1507830191	0.1466212031	0.1407112777
	3	0.1822773032	0.1744369417	0.1704759225	0.2495105062	0.1746258009
	4	0.1417274438	0.1453869026	0.1316085961	0.1373841651	0.1503036991
3	1	0.1496374961	0.1435624324	0.1280109156	0.1312412135	0.1869615819
	2	0.1594685651	0.1389588062	0.1716334522	0.114069676	0.166966049
	3	0.2582480293	0.158985598	0.1400841698	0.1527932584	0.1612278037
	4	0.157985406	0.156085087	0.1775525697	0.2080688961	0.1272955388
4	1	0.1478289105	0.3086702116	0.1200938132	0.140119426	0.179127017
	2	0.1227029786	0.1786029078	0.1612949632	0.1493861414	0.1175681986
	3	0.2519034538	0.1407385003	0.2101189978	0.1865597609	0.1705561783
	4	0.1378952377	0.1178480741	0.1207066998	0.1200009286	0.1612533517
Mean		0.164256337	0.1589251633	0.1519676243	0.1619624723	0.1568421316
Std		0.03874978321	0.04322578706	0.02855709465	0.04024832136	0.0250276511

Table 5.14: Error from AE in *third model*

Test	CV	The number of neurans				
		16	32	64	128	256
1	1	0.1731109992	0.1436949167	0.1456636973	0.187595576	0.151621541
	2	0.2256366871	0.2343511172	0.225972496	0.1990794409	0.1417004224
	3	0.1696365457	0.1366730388	0.1627431139	0.2070423551	0.1462982614
	4	0.1305910405	0.1537955459	0.1439991277	0.1212673876	0.2361955196
2	1	0.1933368258	0.2014074977	0.1557413656	0.1772724874	0.1504026726
	2	0.194492355	0.1620466933	0.186969487	0.1627819967	0.1599432845
	3	0.1982949749	0.1404892672	0.1382220406	0.1454426982	0.2511638664
	4	0.1465102062	0.161547171	0.2015030943	0.2068295442	0.1813242622
3	1	0.1649940163	0.1923567355	0.1391313598	0.1653740592	0.1548881214
	2	0.181799693	0.1197193041	0.2793735377	0.2084757499	0.1446572952
	3	0.1511649378	0.1994536668	0.1662912238	0.1363047659	0.2827695534
	4	0.1715208553	0.1815536954	0.1574255787	0.1797324885	0.1525612958
4	1	0.139421409	0.179470161	0.1902030185	0.1344845295	0.154265251
	2	0.157693075	0.1295783296	0.1185395364	0.1784827951	0.2548566163
	3	0.2060995549	0.1818246227	0.1764678694	0.2163151987	0.1157963872
	4	0.1625140198	0.1539499387	0.1419902053	0.1478702892	0.1442933325
Mean		0.1729260747	0.1669944813	0.170639797	0.1733969601	0.1764211052
Std		0.0258008777	0.03077204756	0.03992570438	0.02998005823	0.05004155784

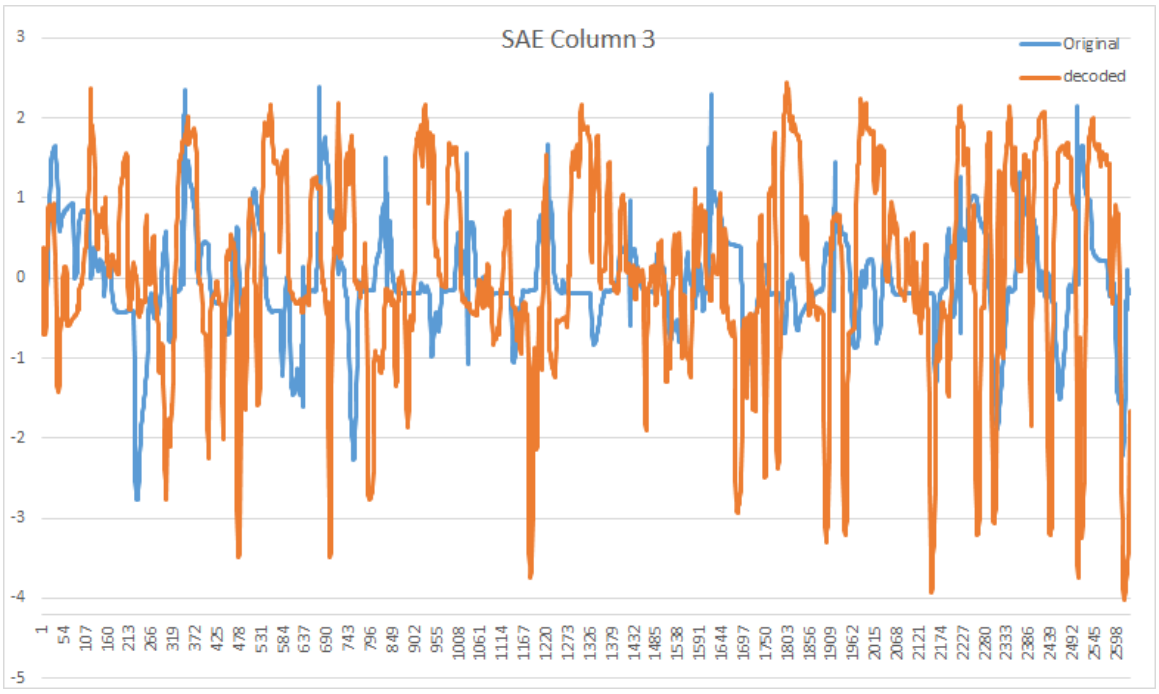


Figure 5.11: SAE, AE Error for Column 3

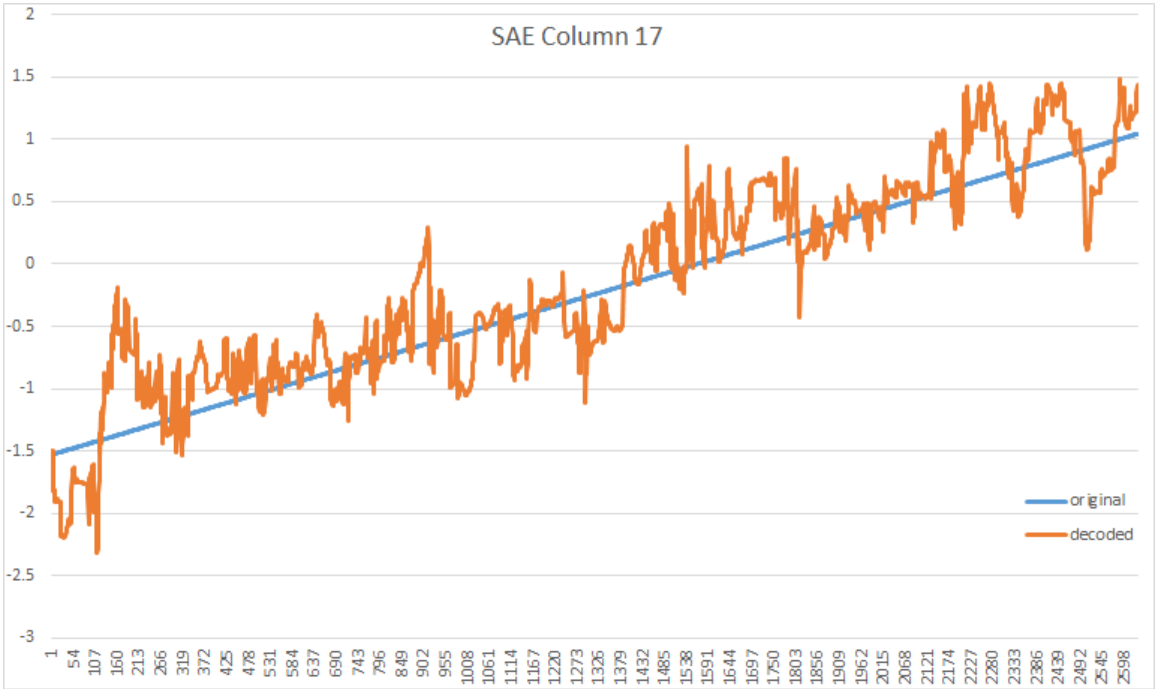


Figure 5.12: SAE, AE Error for Column 17

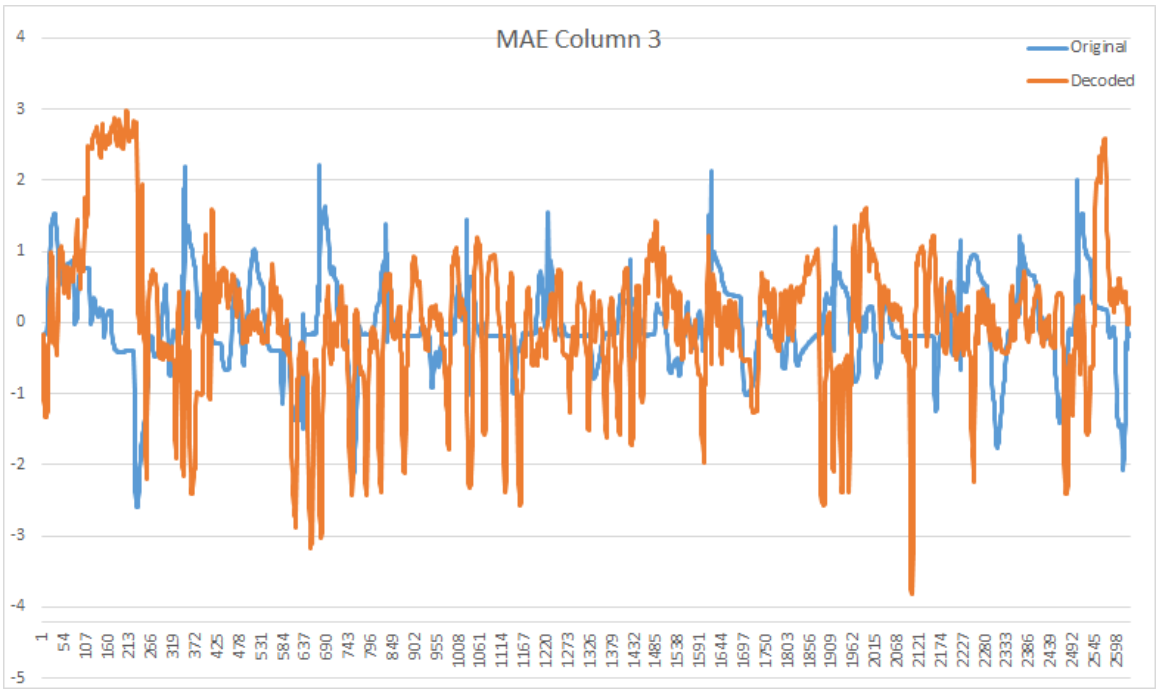


Figure 5.13: MAE, AE Error for Column 3

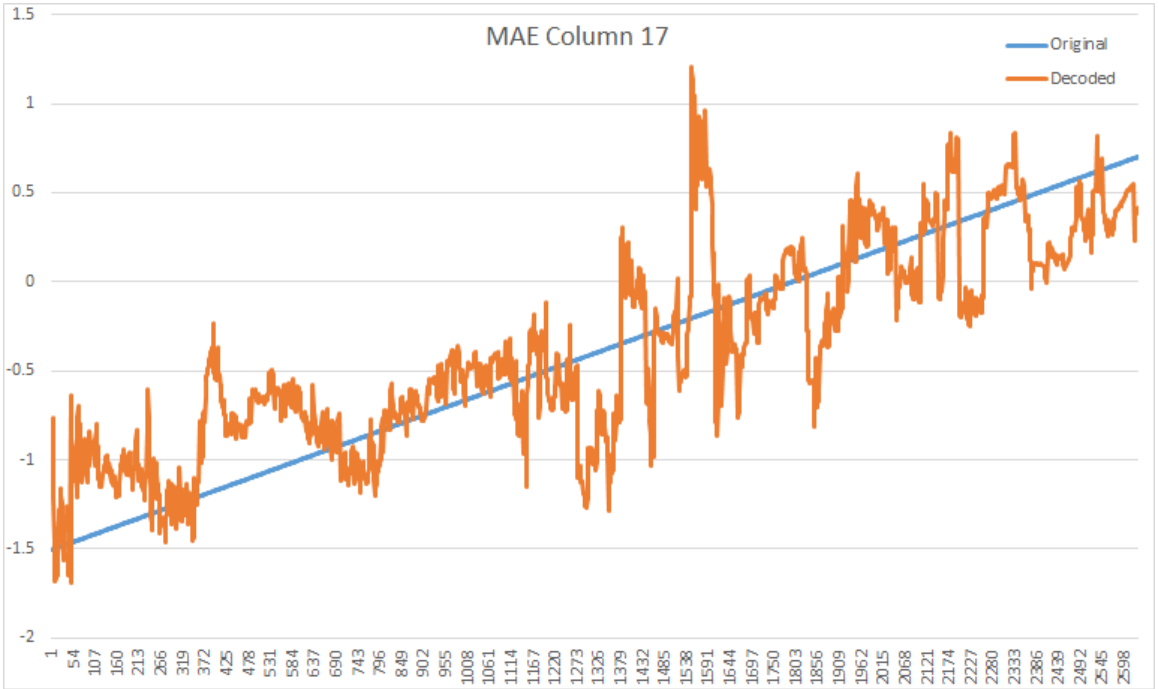


Figure 5.14: MAE, AE Error for Column 17

Chapter 6: Conclusion and Future work

6.1 Data Size

Two different data sizes should be considered on this paper. The first data size is the number of samples in each traces. The data with more samples in each traces gave better performance from the first part of experiments with *first model*. The average accuracy with 1 over 10 sampled data show a higher accuracy than the average accuracy with 1 over 20 sampled data and 1 over 50 sampled data. From this fact, experiments with more samples (in the extreme, using the full data without doing any re-sampling or data from longer driving time) is expected to have better performance. Verifying this hypothesis is part of our future work.

Another data size is the number of traces. All experiments on the paper used 16 traces. 12 traces were used for training and the 4 traces were used for testing in each of the folds of the cross validation runs. Using just 12 traces did not seem to be enough to train the proposed neural networks: *first model*, *second model*, and *third model*. Especially, *second model* and *third model* need more data than *first model* because two models are more complex than *first model*. To train complex neural network needs more data. Next experiments in the future should use the data with enough sample in each traces and enough number of traces.

6.2 Limitation of Auto-encoders

The result from experiments with with SAE in *second model* and MAE in *third model* show bad performance. The principle of AE is to find two matrices: the first matrix is used for compression and the second matrix is used for decompression. If these two matrices do not exist, it is impossible to reduce original dimensions to the target dimensions, and either SAE or MAE do not work. For example, during training the first layer of MAE in *third model* which reduced 98 features to 75 features, the error from this was less than 0.009. This is much better performance than reducing 98 dimensions to 25 dimensions. Therefore, the bad performance of AE implies that it is not possible to reduce 98 features of original data to 25 features by AE.

The second limitation of AE is that auto-encoder always tries to keep all information on encoded data. It means that auto-encoder cannot remove unimportant features other feature selection methods do. For future work, it might be necessary to study auto-encoder methods that accept some form of supervision and can reduce dimensions with the ability to ignore unimportant features and keep information from the important features for the classification task at hand.

6.3 Training Multiple Layer Auto-encoders

Section 2.1.3 explains how to train MAE based on the work of Hinton and Salakhutdinov [**?**]. The paper introduces a pretraining process, and shows it can help for initializing good weights. However, processing of pretraining takes a really long time. For example, average testing time for *third model* with 256 neurons took 10 hours and 27 minutes but the testing time for *second model* with 256 neurons took 8 hours and 33 minutes. The process of pretraining took about 2 hours.

Unlike training each layer separately in pretraining, add more AE until it gives a big error or it reaches goal dimensions. Figure 6.1, Figure 6.2, and Figure 6.3 show how to train MAE.

Figure 6.1 shows SAE and its error function is

$$E(\vec{x}, \mathbf{g_1}(\mathbf{f_1}(\vec{X}\mathbf{E_1})\mathbf{D_1}))$$

Now if it gives small enough error, add one more layer as Figure 6.2. In this case, the error function

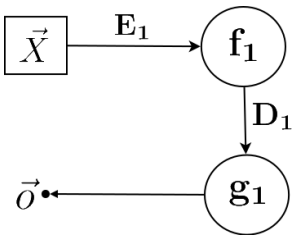


Figure 6.1: First Step of New Way to train MAE

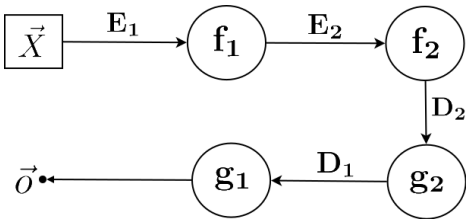


Figure 6.2: Second Step of New Way to train MAE

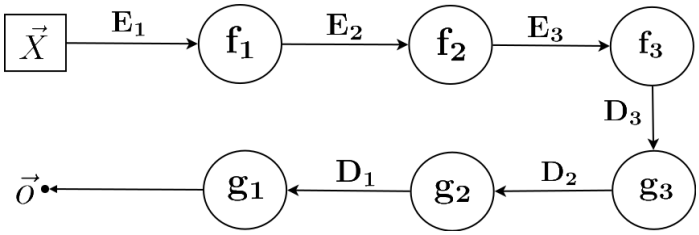


Figure 6.3: Thrid Step of New Way to train MAE

is

$$E(\vec{x}, \mathbf{g_1}(\mathbf{g_2}(\mathbf{f_2}(\mathbf{f_1}(\vec{X}\mathbf{E_1})\mathbf{E_2})\mathbf{D_2})\mathbf{D_1}))$$

If it gives a small enough error, add one more layer as Figure 6.3. In this case, the error function is

$$E(\vec{x}, \mathbf{g_1}(\mathbf{g_2}(\mathbf{g_3}(\mathbf{f_3}(\mathbf{f_2}(\mathbf{f_1}(\vec{X}\mathbf{E_1})\mathbf{E_2})\mathbf{E_3})\mathbf{D_3})\mathbf{D_2})\mathbf{D_1}))$$

If it does not give a small error, stop adding more layers.

This process can give two benefits. The first benefit is to save training time. The first methods is that compared with the method in [**?**], it does not train the whole network. Another benefit is being able to find how much can the AE reduce dimensionality. As part of our future work, it is worth comparing two different training methods for MAE and finding minimized dimensions reduced by AE.