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

Introduction Section

1. Problem Statement

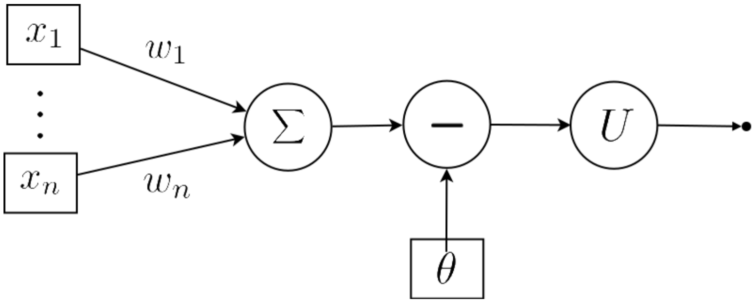

Figure 2.1: Basic Artificial Neuron

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 including recurrent neural network and auto-encoders. Second Section 2.2 compares deep learning libraries: DL4J, Theano, and Tensorflow. The last Section 2.3 introduces current trends in machine learning for driving data.

2.1 Deep Learning

This section covers basic concepts of deep learning. First Subsection 2.1.1 introduces deep learning, and next Section 2.1.2 deals with recurrent neural networks and long short term memory neural networks (LSTMs). The last Subsection 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 [1]. Initial artificial neurons are described as binary output logic gate from inputs. If sum of weighted inputs is greater than 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 activation function. In this case, the step function $U(z)$ is activation function.

Figure 2.1 also can be described by following equation:

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

Figure 2.2 shows an general artificial neuron. Threshold is replaced by bias and subtract node is replaced by add node. In general artificial neuron, activation function ϕ can be many different functions such as step, linear, or sigmoid function. An general artificial neuron also can be described by equation as follow:

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

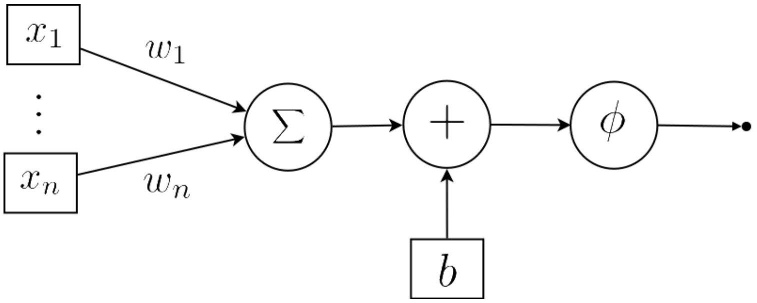

Figure 2.2: General Artificial Neuron

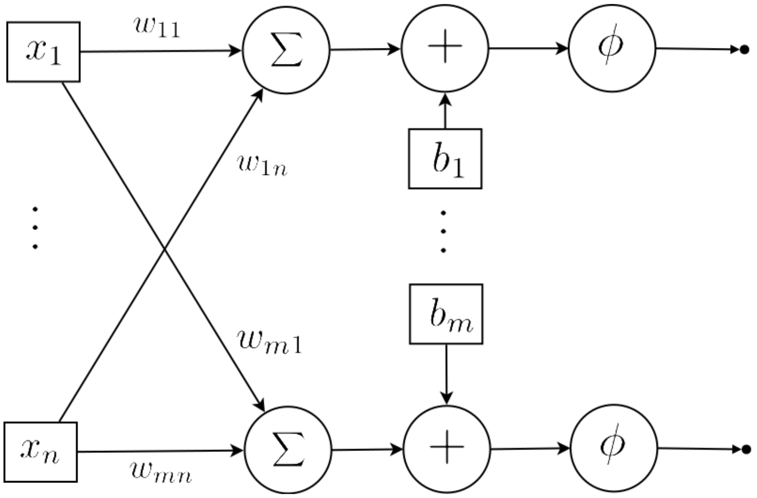

Figure 2.3: Single Layer in a Neural Network

Neural Network

Using single neuron has limitation because it can solve only few problems. For example, single neuron cannot solve xor problem. Instead of using single neuron, neural network has bundle of neurons. Figure 2.3 illustrates it with 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 a 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 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 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 add node anymore.

From this point on the paper, vector in large case is with bias term and vector in small case

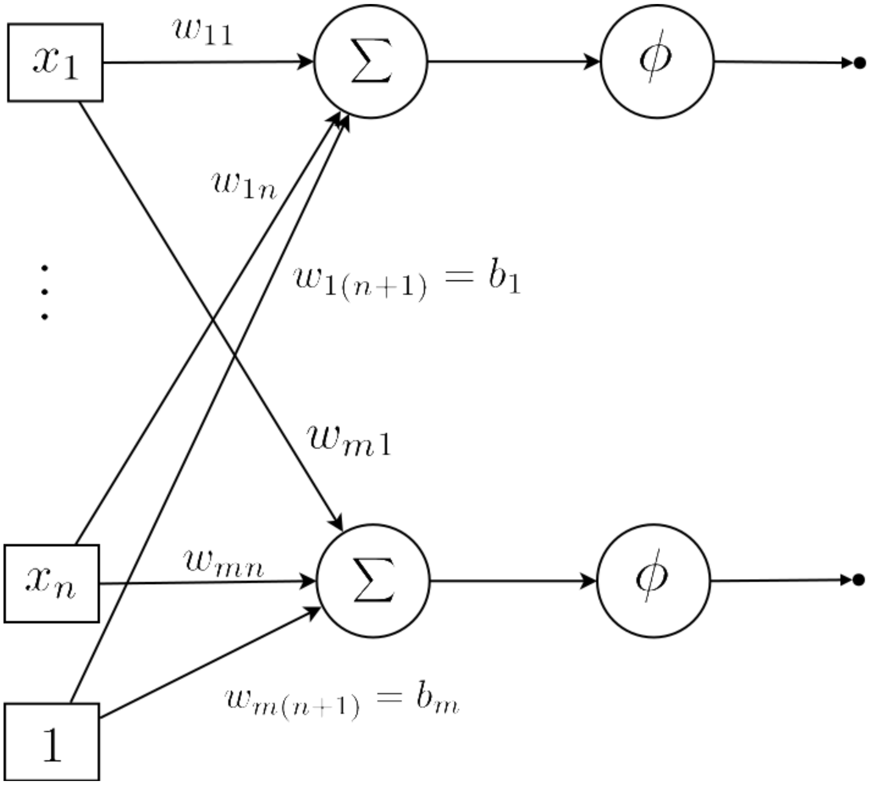

Figure 2.4: Simplified Neural Network

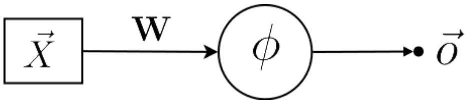

Figure 2.5: Simplified Neural Network

is without bias term. For example, $\vec{X}$ is with bias and $\vec{x}$ is without bias. Also all vectors is row vector. Transform matrix is bold character and if it includes bias, transform matrix uses large case, otherwise small case. For example, **W** is transform matrix with bias and **w** is transform matrix without bias. By applying this, the equation for neural network is

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

For all figures under this point, Σ node is skipped, square shape node represents input node, circle shape node represents compute node, edge with transform matrix represents weight matrix, edge without transform matrix represents identity matrix for weight matrix and arrow with small filled circle on end represents output vector. Figure 2.3, Figure 2.4, and 2.5 are equal.

Multilayer Perceptron

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

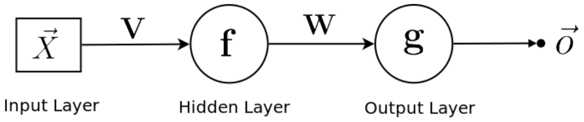

Figure 2.6: One Hidden Layer MLP

To compute output of MLP, forward-propagation is used. forward-propagation is to pass input $\vec{x}$ to output through hidden layers [2]. For example, the output from hidden layer on Figure 2.6 can be computed

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

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

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

Therefore, final equation for two layer neural network is

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

Back-propagation

Training neural network means finding weights and biases. Similar as forward-propagation, back-propagation is used when MLP is trained. As forward-propagation passes input $\vec{x}$ to output, back-propagation passes error from output layer to hidden layers [2]. Error function $E()$ (some times called a loss function) usually uses sum square error as follow:

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

where $\vec{y}$ is 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 input be fixed on the error function, then the output is depends on weights and biases and the train function $J()$ for one hidden MLP (Figure 2.6) can 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 weighs and biases, let $\vec{h'} = \vec{X}\mathbf{V}$ be input for hidden layer activation function $\mathbf{f}$ and $\vec{o'} = \vec{H}\mathbf{W}$ be input for 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 weight in hidden layer for input X_j and W_{ij} be ith neuron weight in output layer for input H_j.

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

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

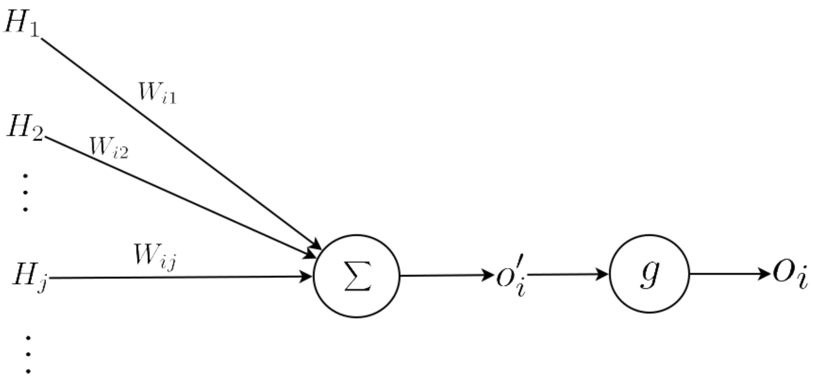

Figure 2.7: Back-propagation for W_{ij}

where η is learning rate. The equation tells to remove error, updated weight should be move opposite direction of gradient of error function. The 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 output layer is sigmoid function $sigm()$ then second part is

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

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 follow:

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

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 layer. Let's update V_{ij} and Figure 2.8 shows detail of neural network related with V_{ij}. The updated weight ${V_{ij}}^{next}$ can be calculated as follow:

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

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

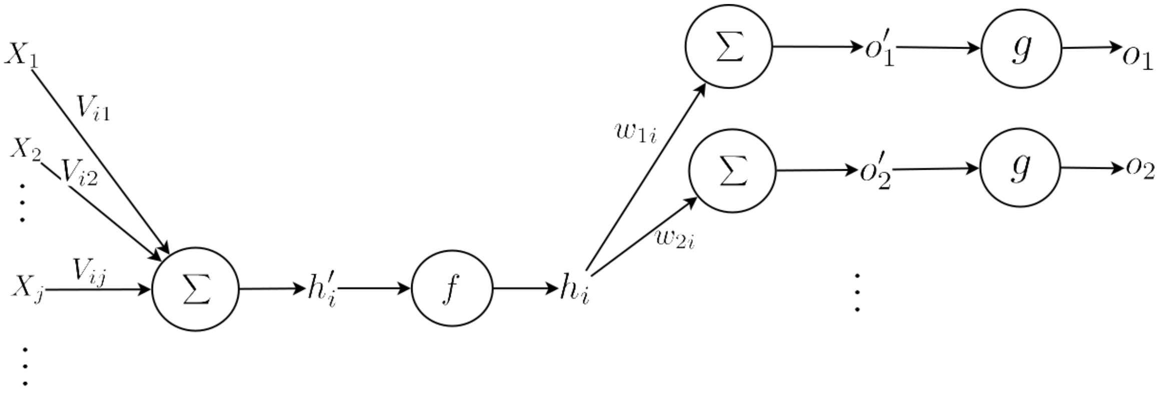

Figure 2.8: Back-propagation for V_{ij}

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 layer is also sigmoid function $sigm()$ then 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,

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

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

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 [3]. Figure 2.9 illustrates abstract structure of RNN. RNN has two input. One input is from data such as normal neural network input but another input is from previous output. The property gives benefit for sequential input data. This is because past outputs affect to current output. With sequential data, previous data can affect to current data and RNN considers previous outputs for current output. It means that even input from data are equal if previous outputs are different, current output are also different.

For example, human language sentences have series of words and meaning of words is different in different context. RNN can be used for that. Another example is driving data which is used later

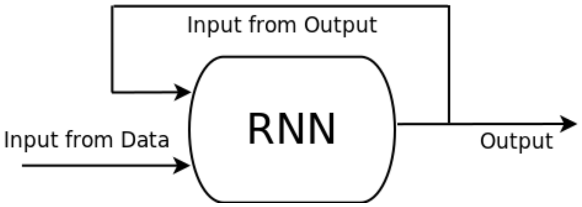

Figure 2.9: RNN

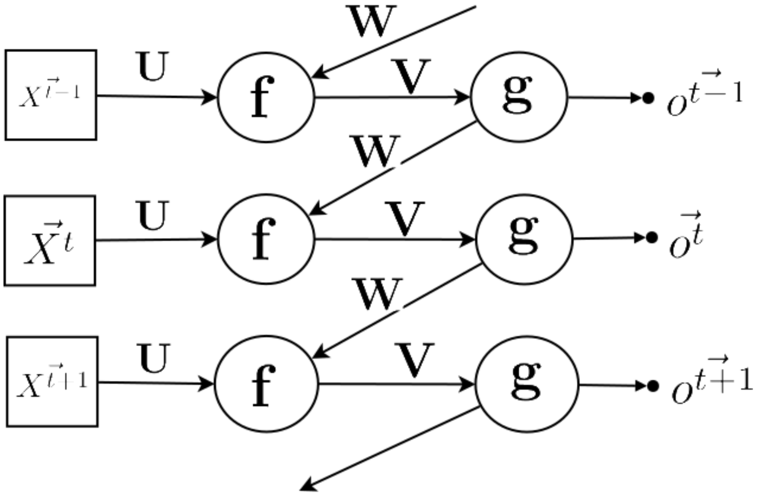

Figure 2.10: Unfold RNN

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 time domain.

To describe RNN in math 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 unfolded RNN with defined symbols. The result from hidden layer can be calculated by

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

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

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

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

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

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

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

Long Short Term Memory (LSTM) networks:

Basic RNN has the problems of long term dependency. When RNN passes previous output for current output, some information in the input might not need or might need for future. RNN does not have ability to filter unnecessary information or to store necessary information. Long Short Term Memory (LSTM) neural network can handle these problems.

LSTM is a type of RNN and introduced by Hochreiter and Schmidhuber [4]. LSTM solves long term dependency problems in RNN by memory cells. LSTM manages memory cells as 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, and output gate. Each gates have different purpose and Christopher Olah's blog [1] develops an intuition for concept of LSTM and explains purpose of each gates. The paper [5] describes LSTM in mathematical term. Let us first provide an intuitive description of how LSTM work 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 purpose of four different gates. Memory cell makes LSTM different from RNN. Memory cell stores important information and filters unimportant information for future. Three of four gates are involved in the memory cells to store and filter information.

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

2. Modeling LSTMs in mathematical terms:

   To describe LSTM in 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 result from input gate on time t, $\vec{m}^t = \{m_1^t, ..., m_m^t\} \in \mathbb{R}^m$ be result from input modulation gate on time t, $\vec{f^t} = \{f_1^t, ..., f_m^t\} \in \mathbb{R}^m$ be result from forget gate on time t, $\vec{o^t} = \{o_1^t, ..., o_m^t\} \in \mathbb{R}^m$ be result from 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 final result on time t. Each gates have transform matrices for inputs. Let $\mathbf{t_{n,m}} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a transform matrix from n dimension to m dimension. Let $\mathbf{t_{n,m}^{ii}}$ be a transform matrix for input from data in input gate, $\mathbf{t_{m,m}^{oi}}$ be a transform matrix for input from previous output in input gate, $\mathbf{t_{n,m}^{im}}$ be a transform matrix for input from data in input modulation gate, $\mathbf{t_{m,m}^{om}}$ be a transform matrix for input from previous output in input modulation gate, $\mathbf{t_{n,m}^{if}}$ be a transform matrix for input from data in forget gate, $\mathbf{t_{m,m}^{of}}$ be a transform matrix for input from previous output in forget gate, $\mathbf{t_{n,m}^{io}}$ be a transform matrix for input from data in output gate, $\mathbf{t_{m,m}^{oo}}$ be a transform matrix for input from previous output in output gate.

   The result of each gates are computed as following:

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

   Input gate uses sigmoid function **sigm**() for activation function

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

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

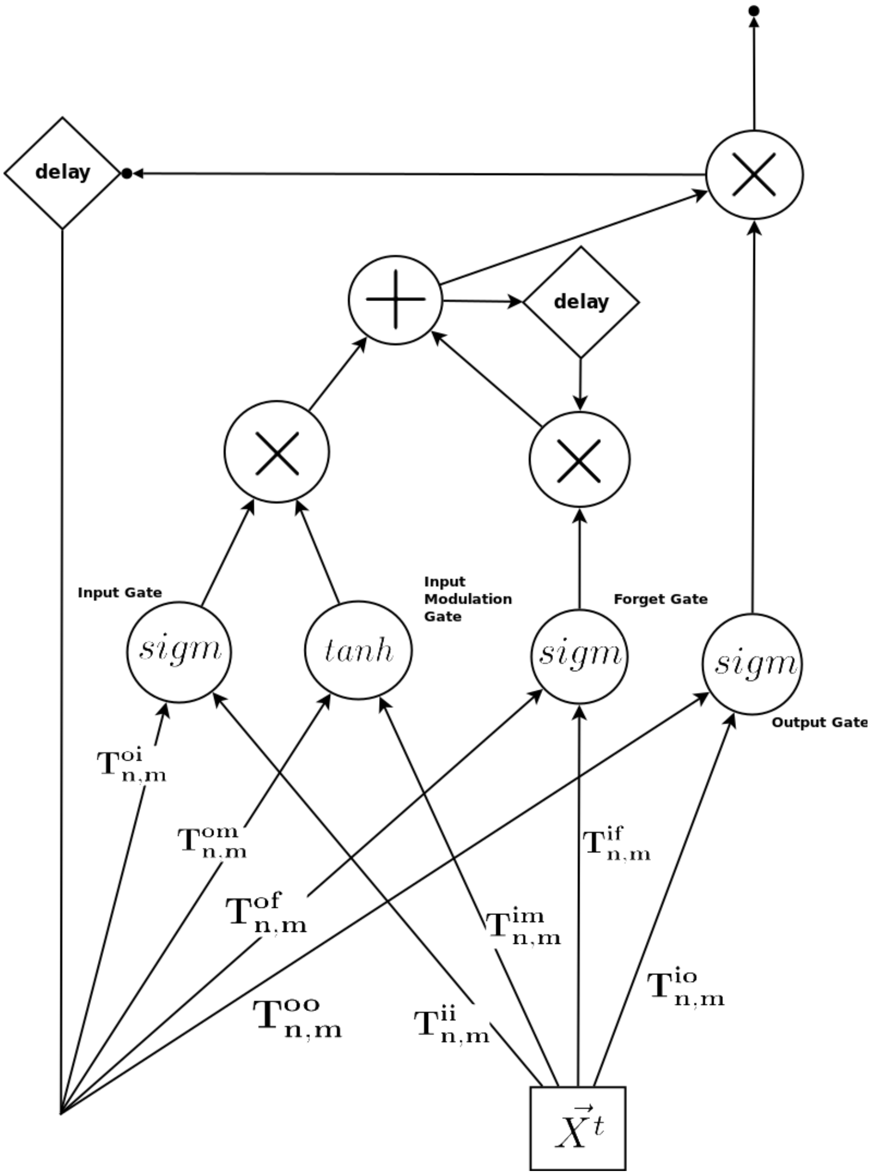

Figure 2.11: LSTM

Input modulation gate uses tanh function **tanh**() for activation function.

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

Forget gate uses sigmoid function **sigm**() for activation function.

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

Output gate uses sigmoid function **sigm**() for activation function.
Memory cells are computed as

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

And final result is computed as

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

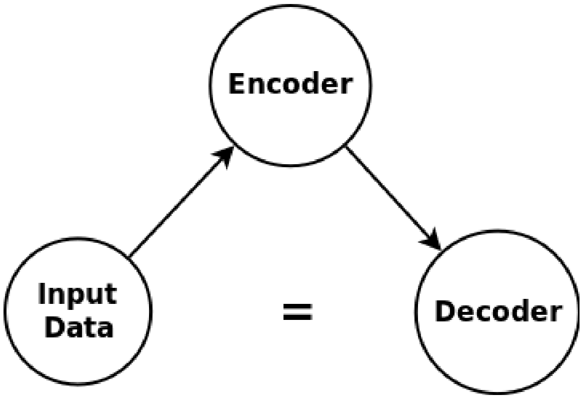

Figure 2.12: Abstract structure of Auto-encoder

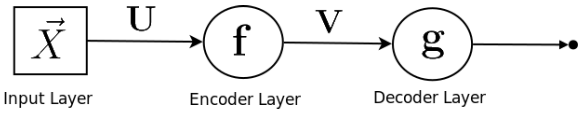

Figure 2.13: Basic Auto-encoder

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 kind 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 input to its output [3]. It means that AE takes input and sends it as output. However, AE does not directly send input to output. AE has two layers: encode layer and decode layer. Figure 2.12 illustrates abstract structure of AE. Purpose of AE is to make output from decode same as input.

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

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

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

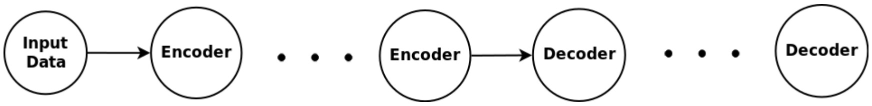

Figure 2.14: Multilayer Auto-encoder

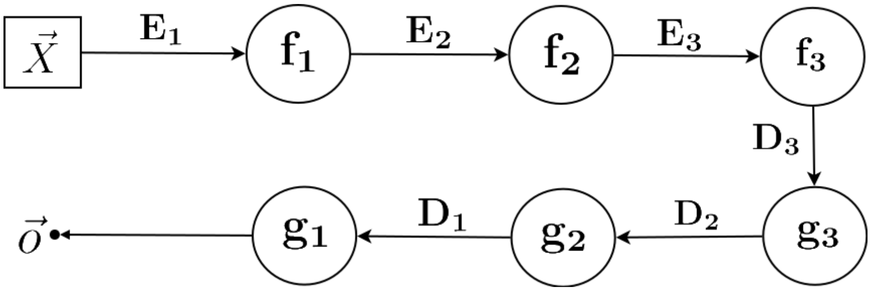

Figure 2.15: Multilayer Auto-encoder Example

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

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

On this paper and experiment, undercomplete AE is used for reducing input dimension. Undercomplete AE is often compared with principal components analysis (PCA) because both methods reduce dimension. The paper [6] compares undercomplete AE and PCA. The result on the paper is that undercomplete AE could keep more information than PCA. It means that reducing data to low dimension by undercomplete AE gives better performance. However, training AE takes long time than computing information gain for PCA.

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 the all weights of encoder and decoder in MAE because all weights are initialized with random numbers and it makes difficult to find optimize weights [5]. So if the initial weights are close to optimize weights, training algorithm gradient descent can find optimize weights. The paper [5] shows ”pretraining” method to initialize good weights.

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

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

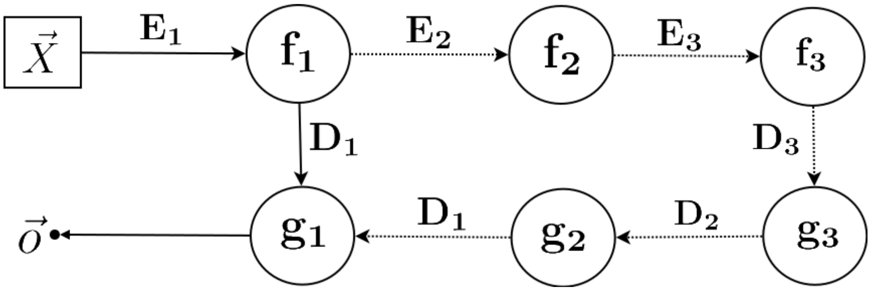

Figure 2.16: Pretraining First Step

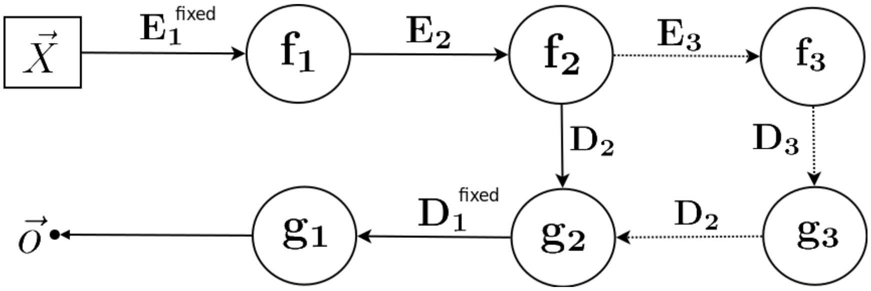

Figure 2.17: Pretraining Second Step

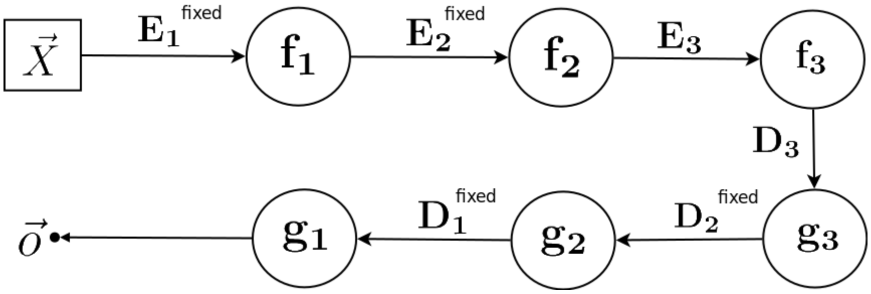

Figure 2.18: Pretraining Thrid Step

Figure 2.17 shows train of second encoder and decoder layer. While second encoder and decoder layer are trained, transform matrix $\mathbf{e_1}$ and $\mathbf{d_1}$ are fixed because two transform matrix are already trained previous step. So error function is

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

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

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

After finishing pretraining step, all weights are close enough to optimal weights for train algorithm to find optimize answer when it train whole neural network. So error function training 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

need to add tutorial

2.2.1 DL4J

DL4J [2] is open source, distributed deep-learning library for Java and Scala. It also integrates with Hadoop and Spark.

2.2.2 Theano

Theano [3] is a Python library for machine learning research. It integrates with Numpy and uses GPU power. The library is written in C code and optimizes user functions.

2.2.3 Tensorflow

Tensorflow [4] is an open source software library for numerical computation using data flow graphs. The biggest difference from other libraries is that Tensorflow treats all operator as node. Another strong point is that Tesorflow allows developer to deploy computation to one or more CPUs or GPUs.

2.3 Machine Learning with Driving Data

[2] https://deeplearning4j.org/

[3] http://deeplearning.net/software/theano/

[4] https://www.tensorflow.org/

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 3: Data Set

Data was collected in the high-fidelity simulator [7] 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 are expert drivers and 2 people are inexpert drivers. Each person drove four different tracks and each track represent 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 [8]. 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.

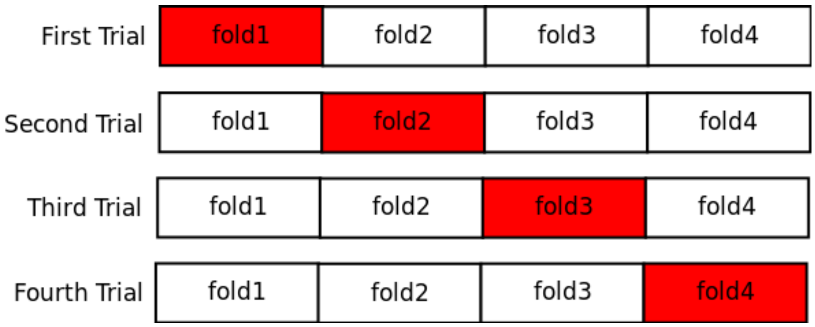

Figure 4.1: Cross Validation

Chapter 4: Technical Approach

This chapter covers technical methods for experiments on the paper also introduces to three different neural network used for the experiments.

4.1 Cross Validation

Cross validation (CV) is used to validate model when data is not enough [3]. CV divides data set to K folds then applies ith fold as test set and other folds as train set to target model. Experiments on the paper use 16 traces but it is not enough to train neural network. CV can be used to validate neural network model and compare performance of three different models. The 16 traces dataset is divided to 4 folds. Each folds contain two traces from expert and two traces from inexpert. The neural networks are trained four times with different train (3 folds) and test (1 fold) set. Figure 4.1 shows four folds cross validation. Red color is the fold for testing and other folds are for training.

4.2 LSTM

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

4.3.1 Single layer Auto-encoder

Figure 4.3 shows 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

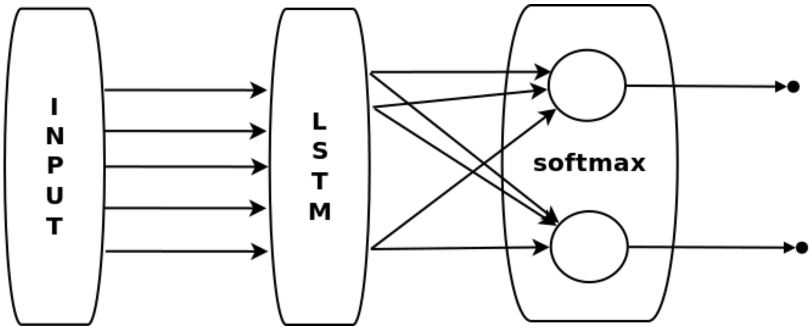

Figure 4.2: First experiment NN

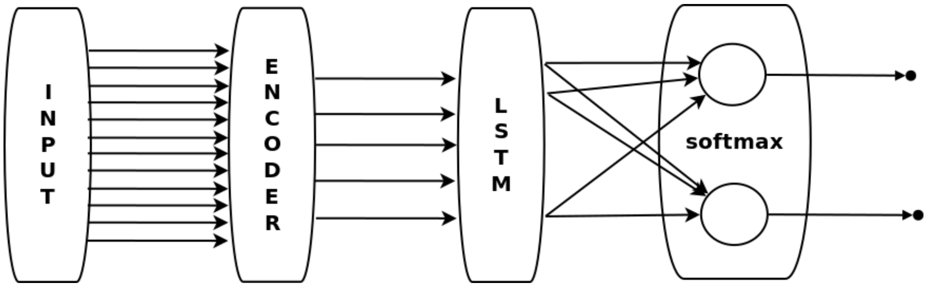

Figure 4.3: Second experiment NN

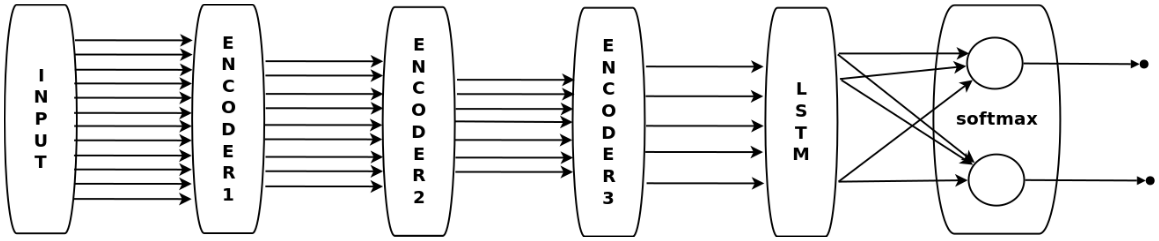

Figure 4.4: Third experiment NN

25 features on *raw dataset*. Notice that the figure only has encoder because after training AE, only encoder is used to reduce dimensions. The output from 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 performance between single layer AE and multiple layer AE, the *third model* has three layers of AE. First AE reduces 98 dimensions to 75 dimensions, 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 and Standardization

The simulator records 60 samples per second. It is too often recorded. The dataset is re-sampled by three different periods: 1 over 10, 1 over 20, and 1 over 50. When it is re-sampled, three methods are used. First method chooses last sample of period, second method computes mean of data in period, and third method applies Gaussian filter. For third method, window size is 11, 21, and 51 for each period.

Chapter 5: Experiment Result

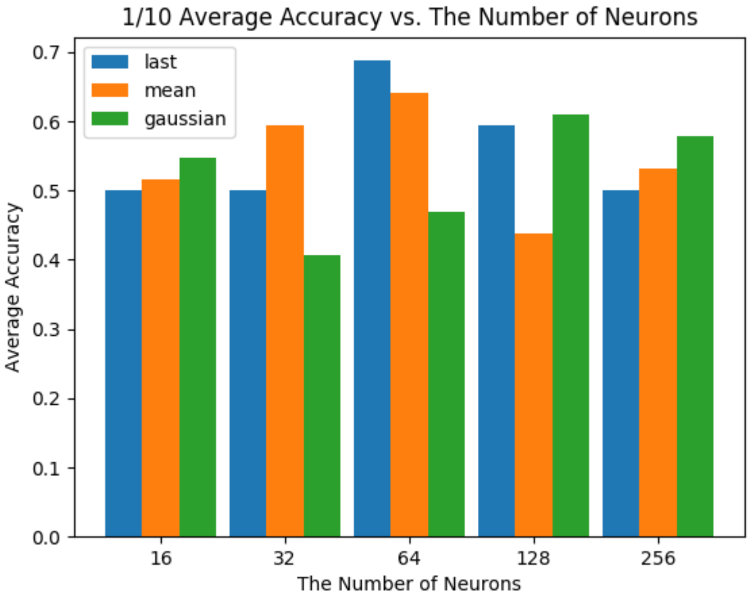

Figure 5.1: Accuracy

Table 5.1: Raw 1/10 Result Summary

		The number of neurans				
Test	n	16	32	64	128	256
1	1	0.5	0.25	1.0	0.5	0.5
	2	0.75	0.75	0.5	0.5	0.5
	3	1.0	0.5	0.5	0.75	0.5
	4	0.5	0.5	0.75	0.5	0.5
2	1	0.5	0.75	0.75	0.75	0.75
	2	0.25	0.25	0.75	0.5	0.25
	3	0.5	0.5	0.25	0.75	0.25
	4	0.25	0.25	0.5	0.25	0.75
3	1	0.75	0.25	1.0	0.75	0.25
	2	0.25	0.5	0.75	0.5	1.0
	3	0.5	0.75	0.75	0.5	0.5
	4	0.5	0.25	0.75	0.75	0.5
4	1	0.75	0.5	0.5	0.5	0.5
	2	0.25	0.5	0.5	1.0	0.25
	3	0.0	0.75	1.0	0.75	0.25
	4	0.75	0.75	0.75	0.25	0.75
Average		0.5	0.5	0.6875	0.59375	0.5

Chapter 6: Conclusion and Future work

Conclusion and Future work

Bibliography

[1] Sebastian Raschka. *Python machine learning.* Packt Publishing Ltd, 2015.

[2] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[5] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[6] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[7] Santiago Ontañón, Yi-Ching Lee, Sam Snodgrass, Flaura K Winston, and Avelino J Gonzalez. Learning to predict driver behavior from observation. In *AAAI 2017 Spring Symposium on Learning from Observation of Humans*, 2017.

[8] Catherine McDonald, Jason Tanenbaum, Yi-Ching Lee, Donald Fisher, Daniel Mayhew, and Flaura Winston. Using crash data to develop simulator scenarios for assessing novice driver performance. *Transportation Research Record: Journal of the Transportation Research Board*, (2321):73–78, 2012.