USING NEURAL NETWORKS TO MAXIMIZE A SUBMODULAR FUNCTION

A Thesis

by

WENXIU HU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Tie Liu
Committee Members,	Krishna R. Narayanan
	Xiaoning Qian
	Xianyang Zhang
Head of Department,	Miroslav M. Begovic

May 2020

Major Subject: Electrical Engineering

ABSTRACT

It is accurate to say that optimization plays a huge role in the field of machine learning. Majority of the machine learning problems can be reduced to optimization problems and having the ability to solve such group of optimization problems becomes the goal of all the people who are interested in diving into the deep machine learning ocean.

Speaking of doing optimization in continuous space, both convex and concave functions can be efficiently and effectively optimized due to its convexity and concavity. It seems that optimization in discrete functions is worth exploring. The most appealing point we see in a submodular set function has to be its natural diminishing returns property. This property makes the group of submodular functions fit in some of the real world machine learning optimization problems, where the problem objective functions are sharing the same characteristics. And the widely extended application of submodular maximization also goes to typical data mining problems that are highly related to submodularity, for example, maximizing the spread of influence in social networks.

In this thesis, we will be introducing a neural network model that has been designed specifically to maximize a submodular set function. This synthetic submodular set function represents a group of functions with certain properties, which will be talked about later in the thesis. This model has the fundamental structure that can be altered or used as a portion of a new model for any other submodular or not necessary submodular set function maximization problem. And empirical results from testing will support the liability of this designated model.

DEDICATION

This thesis is dedicated to all the people who have showed their support along the way.

CONTRIBUTORS AND FUNDING SOURCES

I would like to appreciate the contributions of my advisor, Professor Tie Liu, and Professor Yang Shen of the Department of Electrical and Computer Engineering, for their valuable inputs and endless support along the research journey. And I would also like to appreciate all my committee members, Professor Krishna R. Narayanan, Professor Xiaoning Qian and Professor Xianyang Zhang, for contributing their time to this thesis.

Special thanks to Ph.D. candidate Qiang Zhang for helping me constantly on my thesis, I could not have done all of theses works without Qiang's guidance.

Even though this work has been conducted without external financial support but I am still grateful for the great opportunity and research environment provided by my advisor Dr. Tie Liu and the Department of Electrical and Computer Engineering at Texas A&M University.

NOMENCLATURE

NMT	Neural Machine Translation
RNNs	Recurrent Neural Networks
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
MSE	Mean Square Error

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Most of the traditional machine learning problems can be taken as making predictions from some model. The goal of this type of machine learning problems will be predicting the behavior of the model based on the data generated from the model, which is highly related to that observed [1].

In machine learning, there are two main approaches of designing models for solving problems. One of the two models is called discriminative model and the other one is called generative model. The major difference between these two models can be concluded as the way they learn from data. Discriminative model will be learning the differences among all the possible classes, decision boundaries between classes and moreover, the conditional probability distribution, $p(y|x)$, of the label y, and the iputs x. On the other hand, generative model will be learning the distribution of each individual class. This is equivalent as learning joint probability distribution, $p(x, y)$, and estimating parameters of $p(y|x)$ using Bayes Theorem in (1.1).

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{1.1}$$

Training both generative classifiers and discriminative classifiers involve estimating the mapping function f from inputs x to label y. The discriminative classifier model approaches this by assuming some functional form of the conditional probability distribution, $p(y|x)$, and making predictions of $p(y|x)$ directly from data. The generative model assumes some functional form of both conditional probability distribution $p(x|y)$ and marginal probability distribution $p(y)$. Then estimating these two distribution functions from data and finally using equation (1.1) to compute $p(y|x)$. Examples of discriminative

models are logistic regression, scalar vector machine and nearest neighbor. For generative models, there are example classifiers such as naïve bayes, hidden markov models and bayes networks. One of the reasons why discriminative models is more compelling has been articulated by Vapnik [2], is that "one should solve the [classification] problem directly and never solve a more general problem as and intermediate step [such as modeling $p(x|y)$]."

Rather than making predictions on model's behavior, it is more practical to say that the real aim is to find the optimum from all given observations of the model. The influence maximization problem we mentioned in introduction can be used as one of the examples. In this influence maximization problem, a given social network is modeled as an undirected graph $G = (V, E)$, with V being the vertices and E being the edges. Vertices model the individuals and edges model the relationship between individuals in the given social network. There are many different models designated for solving this problem but the goal is the same, which is to find the optimal subset of individual influencers that can mostly spread information to bring out a large cascade. In order to maintain theoretical guarantees on decisions, the generative model will be used since its structure is amenable to optimization. A natural structure for the generative model is submodularity in the case of discrete decision variables [1].

Submodularity is a property of set functions. Here, the set function refers to a group of functions $f : 2^V \rightarrow \mathbb{R}$. V is a finite set called ground set and each subset of this ground set, $S \subseteq V$, has a corresponding value $f(S)$. There are two equivalent definitions of submodularity from [3] are included below.

Definition 1 (Discrete derivative)**.** For a set function $f : 2^V \rightarrow \mathbb{R}$, $S \subseteq V$, and $e \in V$, let $\Delta_f(e|S) := f(S \cup e) - f(S)$ be the discrete derivative of f at S with respect to e.

Where the function f is clear from the context, we drop the subscript and simply write

$\Delta(e|S)$.

Definition 2 (Submodularity)**.** A function $f : 2^V \rightarrow \mathbb{R}$ is submodular if for every $A \subseteq B \subseteq V$ and $e \in V$ B it holds that

$$\Delta(e|A) \geq \Delta(e|B) \tag{1.2}$$

Equivalently, a function $f : 2^V \rightarrow \mathbb{R}$ is submodular if for every $A, B \subseteq V$,

$$f(A \cap B) + f(A \cup B) \leq f(A) + f(B) \tag{1.3}$$

Where the function f is clear from the context, we drop the subscript and simply write $\Delta(e|S)$.

Discrete derivative from the first definition is also known as the marginal distribution. This is also helpful in submodular maximization because after performing a set A of actions, the marginal gain of any action does not increase over the actions region in $B \setminus A$ [3]. This leads to a conclusion that all the submodular set functions exhibit the diminishing returns property. We will be using the common sensor placement example to illustrate how the diminishing returns property implies. In the sensor placement example, the ground set indicated by V will be all the available locations that a new sensor can be placed and $f(S)$ refers to the detection performance measurement when location set S has been occupied by sensors. There exists a fixed location set S', two given location sets, $E1$ and $E2$ where $E1 = s_1, s_2$ and $E2 = s_1, s_2, s_3$. When the cardinality of the set $E1$ and $E2$ increases by adding S' to each of them, the additional coverage actually does not increase. There are more overlapping occurred if the intial cardinality is larger. Using notations from the first definition, we will have $\Delta(s'|\{s_1, s_2\}) \leq \Delta(s'|\{s_1, s_2, s_3\})$.

In general, submodular set function optimization problems are divided into two cate-

gories, submodular maximization problems and submodular minimization problems. It is not very clear whether submodular set functions are "convex" or "concave", we can see both of them from the definitions. The intuition of solving submodular set function optimization is to map the set function to a continuous space and then optimize the continuous function with existing algorithms according to its convexity or concavity. We know for every function $f : \{0,1\}^N \to \mathbb{R}$ has two canonical extensions, $f^+, f^- : [0,1]N \to \mathbb{R}$. These two functions are defined as the concave closure and convex closure of the submodular function f. It has been proved in Dughmi's work that both definition 3 and definition 4, which are shown below, are equivalent for convex closure [4].

Definition 3. For a set function $f : 2^X \to \mathbb{R}$, the convex closure $f^- : [0,1]^X \to \mathbb{R}$ is the point-wise highest convex function from $[0,1]^X$ to $\mathbb{R}$ that always lowerbounds f.

Definition 4. Fix a set functino $f : 2^X \to \mathbb{R}$. For every $x \in [0,1]^X$, let $D_f^-(x)$ denote a distribution over 2^X, with marginals x, minimizing $E_{S\sim D_f^-(x)}[f(S)]$ (breaking ties arbitraily). The Convex Closure f^- can be defined as follows: $f^-(x)$ is expected value of $f(S)$ over draws S from $D_f^-(x)$.

This convex closure concept is identical to the well-known "Lovász extension" for submodular set function. The definition of Lovász extension f^L is shown below.

Definition 5. For a function $f : \{0,1\}^N \to \mathbb{R}$, $f^L : [0,1]^N \to \mathbb{R}$ is defined by

$$f^L(x) = \sum_{i=0}^{n} \lambda_i f(S_i) \tag{1.4}$$

where $\phi = S_0 \subset S_1 \subset S_2 \subset \cdots \subset S_n$ is a chain such that $\sum \lambda_i \mathbf{1}_{s_i} = x$ and $\sum \lambda_i = 1, \lambda_i \geq 0$.

An equivalent way to define the Lovász extension is :$f^L(x) = \mathbb{E}[f(\{i, x_i] > \lambda\})]$

Also from the theorem in [5], for every $S \subseteq N$ and any submodular function $f : 2^N \rightarrow \mathbb{R}$, the problem $\min_{S \subseteq N} f(S)$ can be solved in polynomial time with respect to $|N|$. With given oracle access, the Lovász extension is simple to compute and can be widely used in solving submodular minimization problem. For submodular functions, the optimization problem can be solved efficiently by converting the problem into a continuous convex minimization problem using Lovász extension. But this is not true for the case of concave closure, which has been proved as NP-hard to evaluate for submodular functions. The concave closure can be defined as, for $f : 0, 1^N \rightarrow \mathbb{R}$, we define

$$f^+(x) = \max\{\sum_{S \subseteq N} \alpha_S f(S) : \sum_{S \subseteq N} \alpha_s = 1, \alpha_s \geq 0\} \tag{1.5}$$

This is what we are more interested about. The difficulty of evaluating concave closure makes it not suitable for solving submodular maximization problem. This is where multilinear extension is introduced. For a set function $f : 2^N \rightarrow \mathbb{R}$, we define its mutilinear extension $F : [0, 1]^N \rightarrow \mathbb{R}$ by

$$F(x) = \sum_{S \subseteq N} f(S) \prod_{i \in S} x_i \prod_{j \in N \setminus S} (1 - x_j) \tag{1.6}$$

Based on the multilinear relaxation, a relaxation and rounding frame has been developed. This general framework consists of three components including optimizing the multilinear relaxation, dependent randomized rounding and contention resolution schemes [6]. This framework can help solving submodular maximization problems of the form $\max\{F(S) : S \in I\}$, where $f : 2^N \rightarrow \mathbb{R}_+$ is sumbodular and $I \in 2^X$ is a downward-closed family of sets and there is a $(1 - 1/e)$-approximation guarantee for the submodular maximization problem whenever function F is the multilinear extension of a monotone submodular function and I can be any solvable polytope [6]. But the multilinear exten-

sion has a requirement that limits it usage in submodular maximization problem, which is 2^n queries to the value oracle of submodular function f is needed for evaluation. This cannot be possibly afforded. This way, only approximation of the multilinear extension will be evaluated.

For submodular maximization problem, we need to define a certain constraints to make the problem non-trivial. Most commonly used constraints are matroid constraints and cardinality constraints. A matroid is a pair $(V, I$ such that V is a finite set, and $I \subseteq 2^V$ is a collection of subsets of V satisfying the following two properties:

- $A \subseteq B \subseteq V$ and $B \in I$ imples $A \in I$
- $A, B \in I$ and $|B| > |A|$ implies $\exists e \in B \setminus A$ such that $A \cup \{e\} \in I$

Matroids generalize the idea of linear independence where sets in I are named independent in linear algebra [3]. The cardinality constrained submodular optimization problem has the following form:

$$\max_{S:|S|\leq k} f(S) \tag{1.7}$$

where k is the cardinality parameter, the goal is to find a subset $S \subseteq N$ maximizing $f(S)$ such that $|S| \leq k$. The case of S having exactly k elements from N is also considered. Some well-known submodular maximization problems including max-k-coverage, max-Bisection and Max-cut with specific cut size are the optimization problems with cardinality constraints [7].

There are two main types of submodular functions that are available to explore for the purpose of optimization, monotone submodular functions and non-monotone submodular functions. A submodular function f is said to be monotone if for every $S \subseteq N, f(S) \leq f(N)$. And a submodular function that is not monotone is called non-monotone. Some examples of monotone submodular functions include entropy function in information the-

ory, maximum coverage functions and budger-additive functions. Mutual information in information theory can be taken as a standard example of nonmonotone submodular function.

Another property related to submodular functions is curvature. The curvature is a measure of how far the function is to being modular. A function f is said to be modular if $f(S) = \sum_{e \in S} f(e)$, and has curvature $c \in [0, 1]$ if $f_x(e) \geq (1-c)f(e)$ for any $S \subseteq N$[1]. Curvature is crucial while solving submodular maximization problem since hard instances often occur only when the curvature c is close to 1.

α-PMAC learnability is introduced for learning submodular functions using standard models. Another way of saying is a function is PMAC learnable if polynomial times of samples are provided and it is possible to construct a prediction function to fit the original function for where the samples are coming from [3]. Based on what has been proved in [1], for any monotone submodular function with bounded curvature c, there is a $(1 - c)/(1 + c - c^2)$ approximation algorithm for maximization under cardinality constraits when polynomial many samples are selected from the uniform distribution over feasible sets. The PMAC-learning framework of optimization from samples is defined as having a sample $(S, f(S))$ of submodular function $f(\cdot)$, which is a set and its function value, a distribution D where the sets Si are drawn from and some constraint $M \subseteq 2^N$ on possible solution sets. The goal of this framework is to find a set, S, such that $f(S)$ is an α-approximation to the optimal solution $f(S^*)$ [1]. For all the functions F that are α-optimizable from samples under M, there exists an algorithm that have the following guarantee:

$$f(S) \geq \alpha \cdot \max_{T \in M} f(T) \tag{1.8}$$

where $f \in F$.

We want to solve submodular maximization from a different perspective, which is

utilizing different neural networks to form a model specifically to solve submodular maximization problem with cardinality constraint. With a given submodular function, the objectives will be designing a model that consists of the followings:

1. An autoencoder that is able to encode a sequence of numbers and decode them from the encoded representation

2. A predictor that is able to predict the set performance from the encoded representation

3. An optimizer that is able to optimize the encoded representation

This neural network model will be broken down into several parts as listed above and will be talked about individually in the following chapters.

2. AUTOENCODER MODEL

2.1 Introduction of autoencoder

A general autoencoder framework as shown in Figure 2.1 can be derived using a t-uple n, p, m, $\mathbb{F}$, $\mathbb{G}$, A, B, X, Δ where [8]:

1. $\mathbb{F}$ and $\mathbb{G}$ are sets.

2. n and p are positive integers. Here we consider primarily the case where $0 < p < n$.

3. A is a class of functions from $\mathbb{G}^p$ to $\mathbb{F}^n$.

4. B is a class of functions from $\mathbb{F}^n$ to $\mathbb{G}^p$.

5. $X = \{x_1, \cdots, x_m\}$ is a set of m (training) vectors in $\mathbb{F}^n$. When external targets are presents, we let $Y = \{y_1, \cdots, y_m\}$ denote the corresponding set of target vectors in $\mathbb{F}^n$.

6. Δ is a dissimilarity or distortion function (e.g. L_p norm, Hamming distance) defined over $\mathbb{F}^n$.

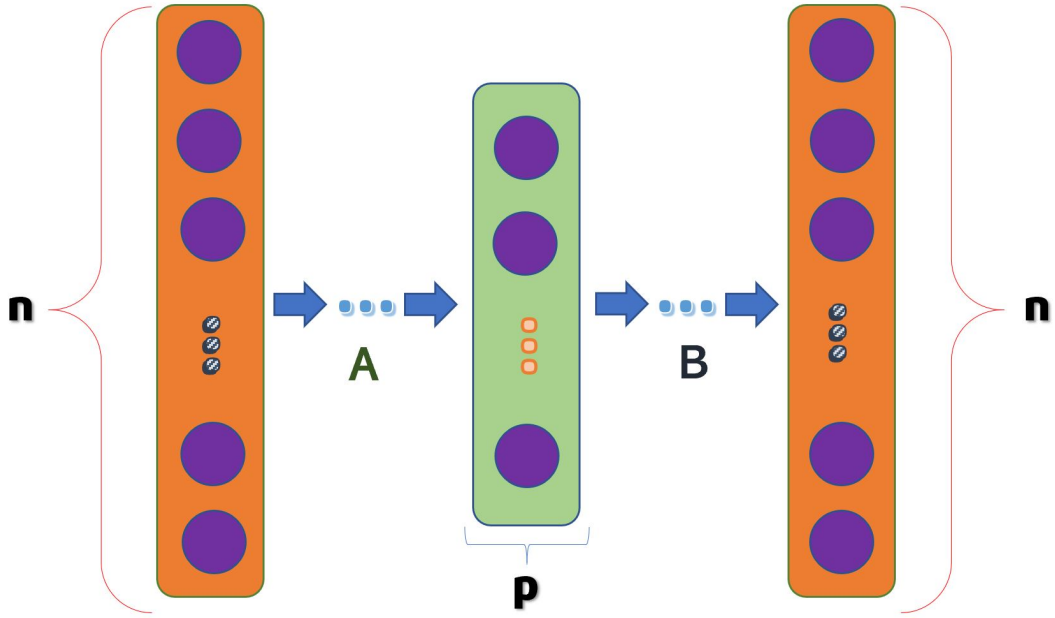

Figure 2.1: Autoencoder architecture

For any set $A \in \mathbb{A}$, $B \in \mathbb{B}$, an $n/p/n$ autoencoder transforms an input vector $x \in \mathbb{F}^n$ into an output vector $A \circ B(x) \in \mathbb{F}^n$. Our expectation for this type of $n/p/n$ autoencoder is to minimize its overall distortion function as indicated in [9]:

$$\min E(A, B) = \min_{A,B} \sum_{t=1}^{m} E(x_t) = \min_{A,B} \sum_{t=1}^{m} \Delta(A \circ B(x_t), x_t) \tag{2.1}$$

This is the same as maximizing the similarities between vector A and vector B.

2.2 RNN

Recurrent Neural Networks (RNNs) have several features that make it stand out among all other neural networks. As illustrated in Figure 2.2, the neural network takes an input vector $\underline{x} = \{x_1, x_2, x_3, \cdots\}$ with no pre-determined length and with contrast to CNNs,

there is an intermediate vector named hidden state, $\underline{h} = \{h_0, h_1, h_2, h_3, \cdots\}$, which updates itself and provides RNNs context of previous inputs to make a better and informative decisions while generating outputs, $\underline{y} = \{y_1, y_2, y_3, \cdots\}$.

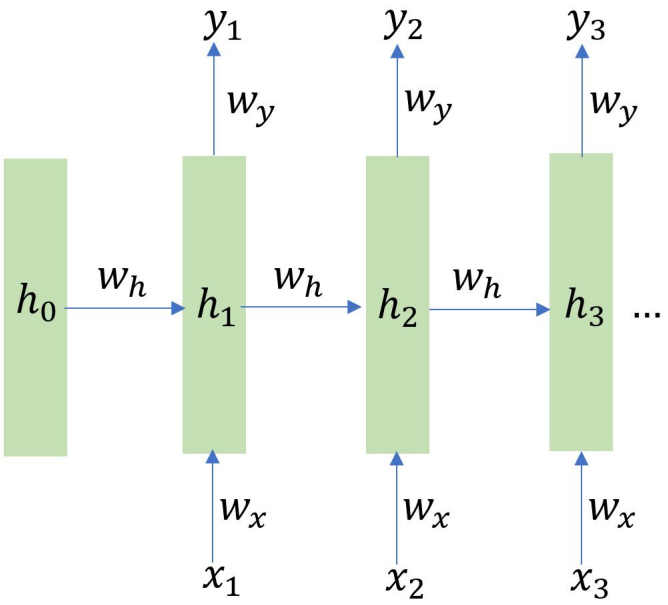

Figure 2.2: RNN architecture

So the key difference between basic NNs and RNNs is memory. Basic NNs only have memories about learned weights during the process of training. RNNs not only have that but also have memories about the original inputs. RNNs can be extremely helpful for preserving information of neighboring inputs. We can see this feature applied really while in translation neural network.

2.3 NMT sequence-to-sequence model

The sequence-to-sequence model in neural machine translation has the outlined autoencoder architecture (Figure 2.1) with detailed internal design. The model is composed of two jointly trained neural networks: a recurrent encoder and a recurrent decoder.

The encoder RNN takes a sequence of feature frames $x_{1..T}$ and transforms it into a sequence of hidden activations, $h_{1..L}$ [9]. The hidden activations can also be called the encoding of the source vector, x_t. Then we have a general form:

$$h_l = f(x_{1..T}) \tag{2.2}$$

where $f(\cdot)$ is an encoding function.

The decoder RNN takes the encoded source vector and produces a sequence of output tokens, $y_{1..K}$ [9]. One output token is produced per step and each output token depends on not only the encoded source vector but also the output token that has been produced in previous time step. The general form of the decoder:

$$y_k = g(y_{k-1}, h_{1..L}) \tag{2.3}$$

where $g(\cdot)$ is a decoding function.

The encoding function often consists of a stack of recurrent layers. The most common recurrent layers that have been used are LSTM layers or GRU layers. Recurrent neural network will not work really well as the source vector gets larger. In the NMT case, the input sequence can be a long paragraph that needs to be translated. Each word might be associated with its neighboring words. RNNs will be having a hard time predicting the correct translated words for all the original words that appear at the beginning of the input sequence and important information could be lost. That is how LSTM and GRU

were created, to solve this problem of RNN. GRU has two gates, reset and update gate. LSTM has three multiplicative gates, input, output and forget gate.

Both of LSTM and GRU utilize gating information to prevent vanishing gradient problem. LSTM has been widely used with RNN architecture. Even though LSTM is computationally more complex than GRU because of the memory unit, LSTM should perform well regardless of the length of input sequence theoretically. Another difference that worth mentioning between LSTM and GRU is the internal state. In LSTM, there are two types of state, cell state c and hidden state h. We will be using h_{t-1} to represent the hidden state computed at time step $t-1$ and h_t to represent the hidden state computed at time step t. Basically, the input x and h_t will be used for computations of input, output and forget gate. The cell state c_t is dependent on not only x and h_{t-1} but also the previous cell state c_{t-1}, input gate and forget gate. The hidden state h_t is then computed using result from c_t and output gate with the help of hyperbolic tangent function in a nonlinear way. The hidden state h is what we interact the most since cell state c serves as the memory and h is passed to the three gates of LSTM to continue processing input. GRU merges these two and only have one hidden state. We will be focusing on LSTM layers for encoder during the thesis. The hidden state of encoder RNN at time step t, h_t, is computed in the form:

$$h_t = f(W^{(hh)}h_{t-1} + W^{(xh)}x_t) \tag{2.4}$$

where $W^{(hh)}$ is the weights matrix based on the previous hidden state and $W^{(xh)}$ is the weights matrix based on the current input.

The decoding function often consists of a stack of recurrent units just like the encoder. The most common recurrent units that have been used are LSTM cells or GRU cells. Cells are contained in layers and we will be talking about LSTM cells instead of GRU cells for consistency with encoding function. The decoding function will use the last state output

from encoder as input and this means the initial hidden state is generated from encoder instead of randomly generated like encoder. Each LSTM Cell accepts a hidden state from previous state and produces an output as well as the current hidden state for the use of next time step. The hidden state of decoder RNN at time step t, h_t, is computed in the form:

$$h_t = f(W^{(hh)} h_{t-1}) \tag{2.5}$$

where $W^{(hh)}$ is the weights matrix based on the previous hidden state.

The output of decoder will be generated one by one using the formula:

$$y_t = soft\max(W^S h_t) \tag{2.6}$$

where W^S is the weights matrix at the current time step. The Softmax activation function f has the form [10]:

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{2.7}$$

The Softmax function is mostly used in output layers for deep learning architecture. The Softmax function will help create a vector of probabilities with respect to each possible output. All the entries of this vector will be in range of values between 0 and 1 and the sum of all entries will be equal to 1. The softmax function will look something similar to the function in Figure 2.3 depending on input values.

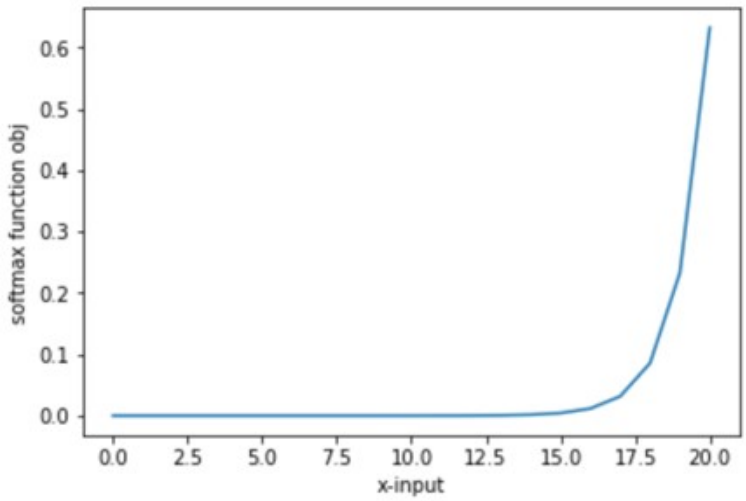

Figure 2.3: Softmax funtion

And the one with highest probability will be selected as the final output at the current time step using $arg\max$ function. This current decoder output will be used as the decoder input for the next time step. The NMT sequence-to sequence model will look something like this in Figure 2.4:

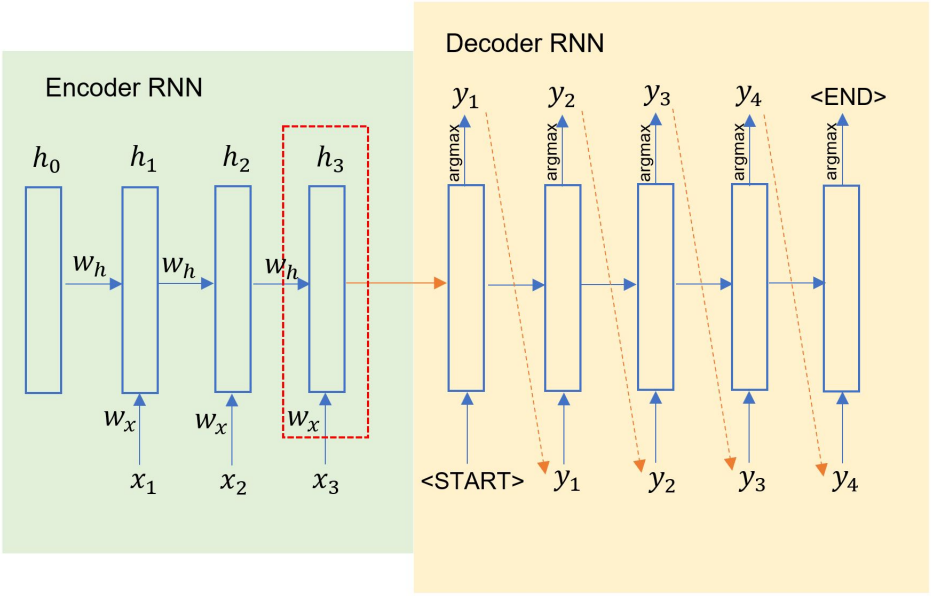

Figure 2.4: Neural Machine Translation model architecture

2.4 Our autoencoder model

Our autoencoder model will be using the same encoder-decoder architecture as the neural machine translation model with some changes.

First, we will introduce the additive Bahdanau attention mechanism developed in [11]. The general purpose of using an attention mechanism on top of the encoder-decoder model is to enhance the memory of original inputs and improve decoding accuracy. The architecture of encoder-decoder model with attention that we will be using is shown in Figure 2.5 below.

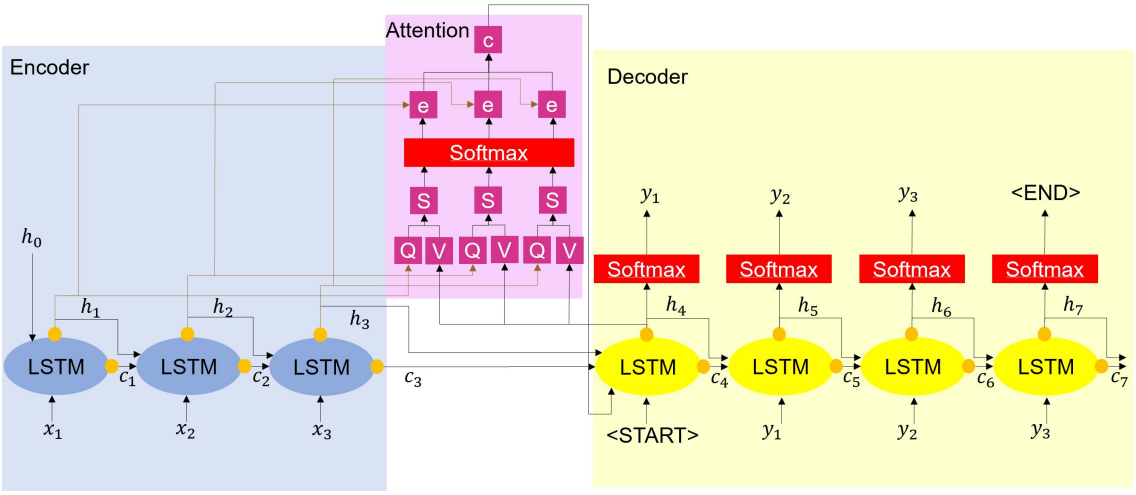

Figure 2.5: Encoder-decoder LSTM model with attention

The goal of this attention mechanism is to produce a context vector c, which is a weighed sum of the encoder output states. As illustrated from the above architecture, at the beginning of decoding, we will need initial information from both the output states of encoder, which is called query, and encoder RNN's output, which is named value in this attention mechanism, to prepare the attention mechanism. After the first decoder LSTM cell output has been produced, we will be using this decoder LSTM cell output state as the new query and previous encoder LSTM output state as value. Each element in the input vector x will be assigned a score to show how much it matters for computing the current wanted output. The score can be calculated using this formula:

$$score(q, v) = v^T tanh(W_1[q]; W_2[v]) \tag{2.8}$$

where V,W_1 and W_2 are learnable weight matrices. After applying softmax function to normalize all scores, the attention weights are ready to be used. Finally the context vector

can be obtained by:

$$c = \sum_{s} \alpha_{ts} \overline{h_S} \tag{2.9}$$

where α_{ts} is the attention weights and $\overline{h_s}$ is the encoder output state.

We will place a "start" token at the beginning of decoder's input to initialize the input of decoder to start the decoding process and generate decoder hidden state one by one for attention mechanism to evaluate. This attention network will be jointly trained with encoder RNN and decoder RNN.

3. PREDICTOR AND OPTIMIZER MODEL

3.1 Fully-connected neural network

Fully-connected neural network has been widely used for many applications in the field of machine learning due to its "input agnostic" feature. This means that we do not necessarily need to provide any assumptions about the input data. A fully-connected neural network usually consists of a series of fully-connected layers. Each fully-connected layer has the functionality of transforming any representation from $\mathbb{R}^n$ to $\mathbb{R}^m$. Each neuron in a fully-connected layer is connected to every neuron in the previous layer. The dimensionality is dependent on the previous layer as well. A generalized fully-connected neural network has the architecture in Figure 3.1.

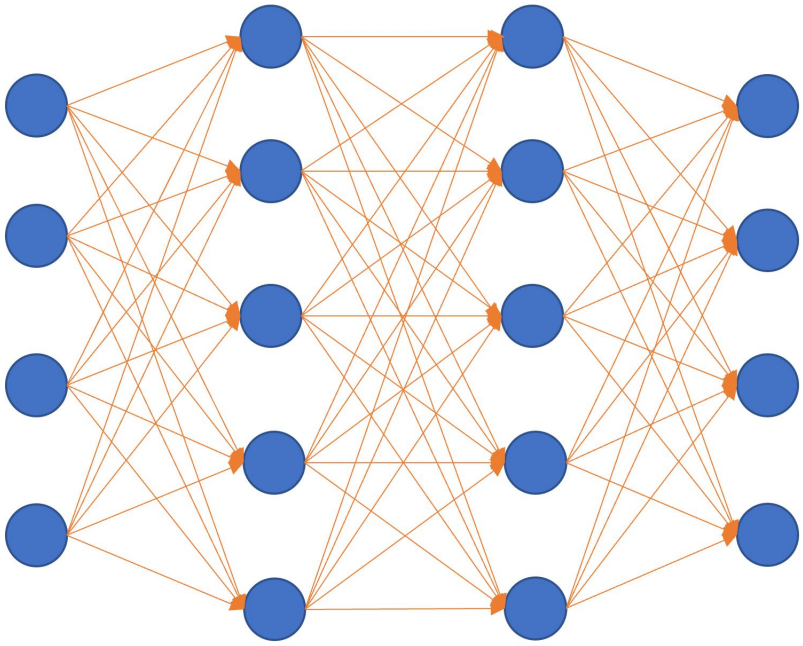

Figure 3.1: Fully-connected neural network

In our case, we expect the input of our predictor to be the last output of encoder RNN. And our fully-connected neural network will use this representation to predict the performance of the original input x in submodular set function. Instead of having multiple classes as output, we will ask the fully-connected neural network to generate a single output as the prediction of submodular function value. The number of hidden fully-connected layers is dependent on the size of the ground set N as well as the cardinality constraint on k. The larger these numbers are, the more complex the encoder output representation is. And a deeper fully-connected neural network will be needed to fit this submodular function over a complex but informative input. The example predictor model in Figure 3.2 has two dense layers between input layer and output layer.

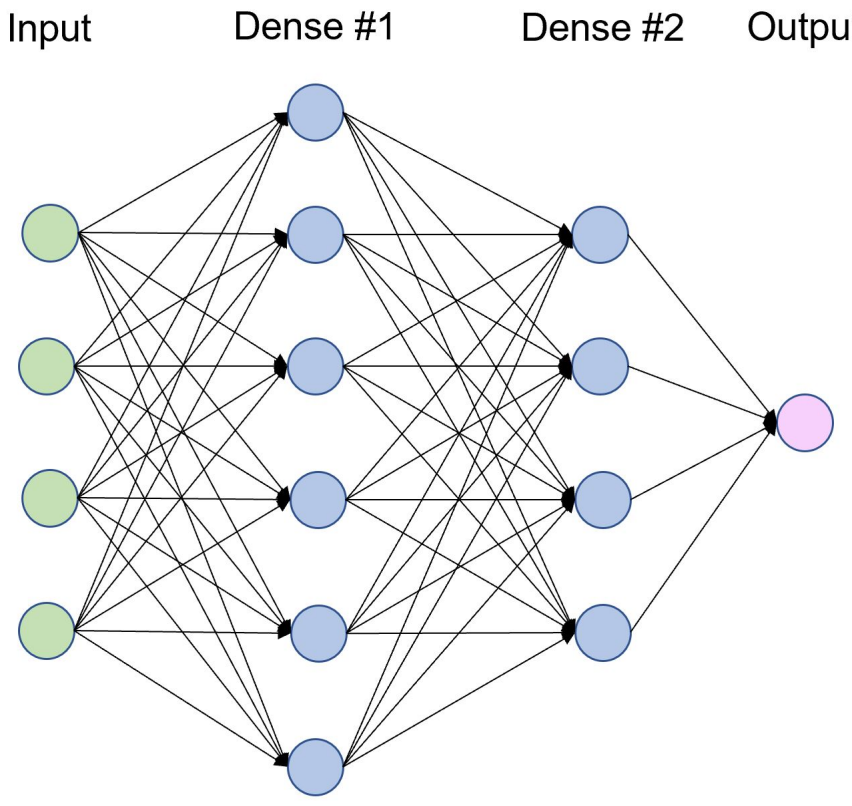

Figure 3.2: Predictor model architecture

Without changing the batch size, a flatten layer will be introduced before this fully-connected predictor model to help prepare input data. A flatten layer will be in charge of reshaping the representation of last LSTM output states of encoder together and reducing its dimensionality to make it suitable for this fully-connected predictor model.

We have talked about Softmax activation function, which is also used by the last layer in the predictor model. For the layers before output layer, the activation function we will be using is called ReLU activation function. These two activation functions will perform non-linear transformation in the neural network and help the model to learn complex functions. ReLU function is half rectified and converts all numbers that are less than zero to be zero. All other numbers including zero stay the same. A ReLU function plotted using Python can be found in Figure 3.3.

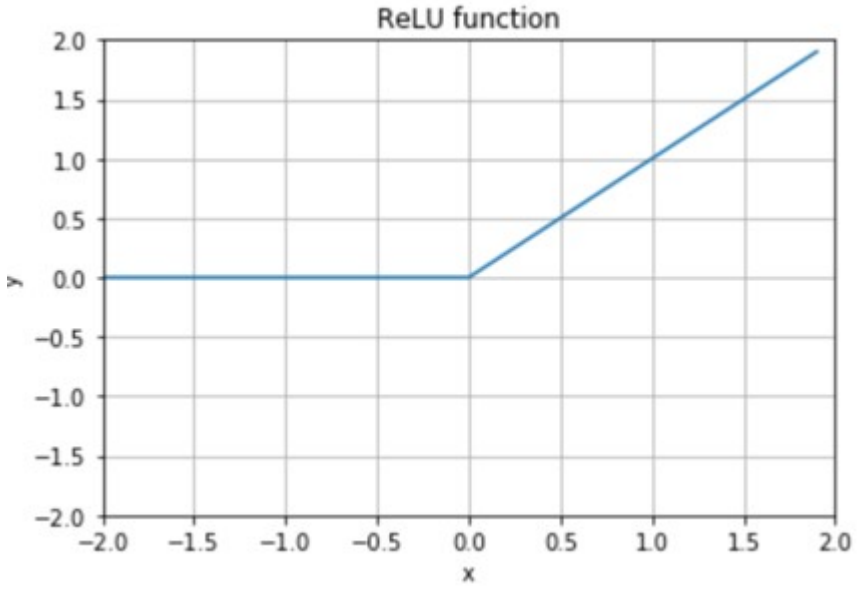

Figure 3.3: ReLU activation function

3.2 Our optimizer

After predicting the submodular function performance using the above fully-connected neural network model, we will use the same input for optimizer. This input is also the output state of encoder RNN. The weights and bias parameters will be saved from the previous well-trained predictor. And we will calculate the negative gradient direction of these parameters with respect to objective value, which is the only output from predictor, over the input representation at each step. The total step size can be observed by running different experiments.

4.1 Training data

Before generating data for training, we need to be specific of which submodular function we will be using. The submodular function, which serves as the objective function, is a simple synthetic function f from [1] defined as following:

$$f(S) = \begin{cases} |S \cap (G \cup B)| & \text{if } |S \cap B| \leq 5 \\ |S \cap G| + |S \cap B| \cdot (1 - c) - 10c & \text{otherwise} \end{cases} \tag{4.1}$$

where S is the subset of N, $|\cdot|$ is the cardinality function, c is the bounded curvature, both G and B are fixed sets.

This is also a monotone submodular function. We will prepare the training data sets for autoencoder and predictor separately since our training strategy is to train the encoder-decoder with attention first and then use this pre-trained autoencoder model to jointly train with predictor model.

4.1.1 Encoder-decoder model with attention

Since we want to make the encoding process order invariant for sequence-to-sequence model, we will sort all input elements into an ascending order and use them as the inputs for encoder. This is because the representation of output states from encoder will carry information of the original order. This matters a lot in the NMT case since a sentence can have varying meanings when words are placed in different positions. But in our case, we want the representation, for example, set $S_1 = \{1, 2, 3\}$, set $S_2 = \{3, 1, 3\}$ and set $S_3 = \{3, 2, 1\}$, look as similar as possible. Because these three sets are identical in the set level and we are avoiding the neural network processing them differently. The encoder

learns a powerful representation of input set by stacking two LSTM layers together. But these LSTM layers do not interact with the integer sets directly. All the training input sets will go through embedding layer first so the true input of encoder is actually an embedded representation. One-hot embedding commonly existed in NMT ror similar machine learning problems. But this one-hot embedding can be inefficient when the input vector gets longer. And in the long input vector case, the vector is going to be sparse and hard to be understood by neural networks since most of the elements in one-hot vector are zeros. We want the embedded vector to be dense and preserve more information besides just being distinguished from others. The embedding layer will contain a trainable embedding matrix, which can be randomly initialized. This embedding matrix will be gradually updated during training by backpropogation. The embedded representation will be the dot product of embedding matrix and input vector. This embedded representation will then be used as the real input of LSTM layer. With the help of attention mechanism, we expect the decoder perform really well after training. We will focus on the case when the size of ground set N is 600 and k is 20. This is basically a 600 chooses 20 problem and we have approximately 10^{37} number of subsets.

We have mentioned that we will be using Softmax function at the output of decoder. So we have chosen cross-entropy as the loss function. Softmax function outputs probabilities and cross-entropy measures performance of a model whose output is a probability value. The cross-entropy loss is also called log loss and has the general equation as following [12]:

$$-\sum_{j} y^{(j)} log \sigma(o)^{(j)} \tag{4.2}$$

where $\sigma(\cdot)$ is the probability estimate, y is the true label and o is the output from last layer.

For the decoder part, we will be using an algorithm called teacher forcing to help improve the decoding accuracy. Basically, the decoder RNN is fed with input data as

well. The decoder RNN will be supplied by observed sequence values as inputs during training and use its own one-step-ahead predictions for future steps [13]. The input of decoder's first LSTM Cell will always be a "start" token and the output sequence will always end with an "end" token to mark the stop of decoding process.

Since RNNs are mainly used in our model, we will be applying layer normalization instead of batch normalization to speed up the learning process. The comparison between layer normalization and batch normalization is illustrated in Figure 4.1 below.

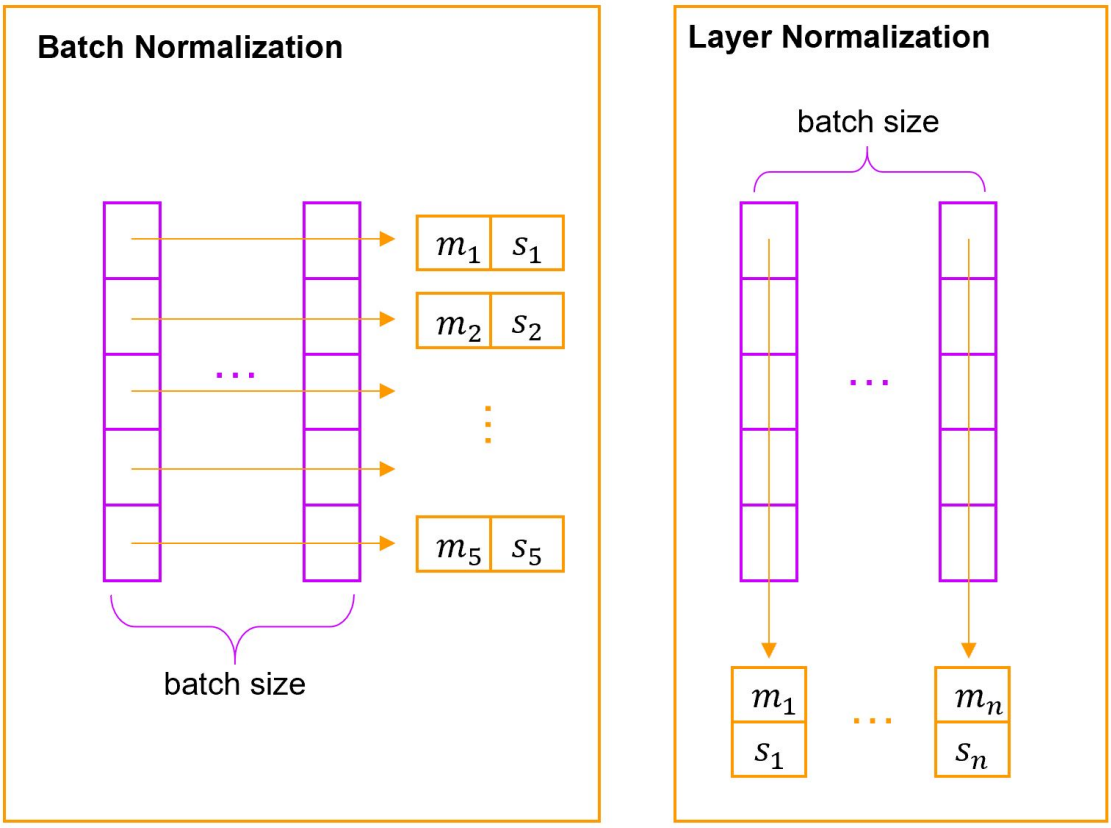

Figure 4.1: Comparison between batch and layer normalization

The idea of this type of normalization is to standardize each summed input using its

mean and standard deviation so that the feedforward neural networks converge faster [14]. There are some limitations on using batch normalization. The biggest limitation related to our model is batch normalization is not compatible with recurrent connections because the activations at each time step will have different mean and standard deviation. This requires a different batch normalization layer at each time step and this is not anything ideal to do. We decided to use layer normalization, which intended to eliminate the limitations that batch normalization has. The layer normalization statistics is computed over all the hidden units in the same layer as follows:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \tag{4.3}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{4.4}$$

where H denotes the number of hidden units in a layer [14].

4.1.2 Predictor and Optimizer

The training data for predictor will depend on which submodular function we are interested about. The training labels will be the same as the objective values of submodular function. There is one part we should take special care of. Since we will be using gradient descent algorithm to find the steepest decrease direction of predictor parameters, weights and bias, we need to make all the submodular function objective values to be negative so that our goal is to find a gradient direction where a smaller label output can be obtained.

The inputs that are passed into our optimizer are encoder RNN's final output states and well-trained predictor parameters, weights and bias. The output of the optimizer will be an optimized representation of the original input set. And we will then send this optimized representation to inference decoder model to decode it back in set form. The

optimized objective value can be then calculated using the submodular function.

4.2 Training result

4.2.1 Encoder-Decoder model with attention

First, we need to be clear about how to set hyperparameters for our training of this encoder-decoder model. Please see Table 4.1 for differences between hyperparameter batch and epoch. These hyperparameters are different for different cases.

Batch	Epoch
number of samples to run before updating model parameters	number of times the learning algorithm goes through the entire training samples
training samples are divided into one or more batches	comprised of one ore more batches

Table 4.1: Comparison between batch and epoch

The number of epochs was not defined during the training of our encoder-decoder model. We generated the batch size of training data randomly every time and iterate this over and over again until the training loss converges. We have run 3500 iterations with batch size of 100 and we can see the process of how training loss approaches 0 throughout training with only two outlier points in Figure 4.2.

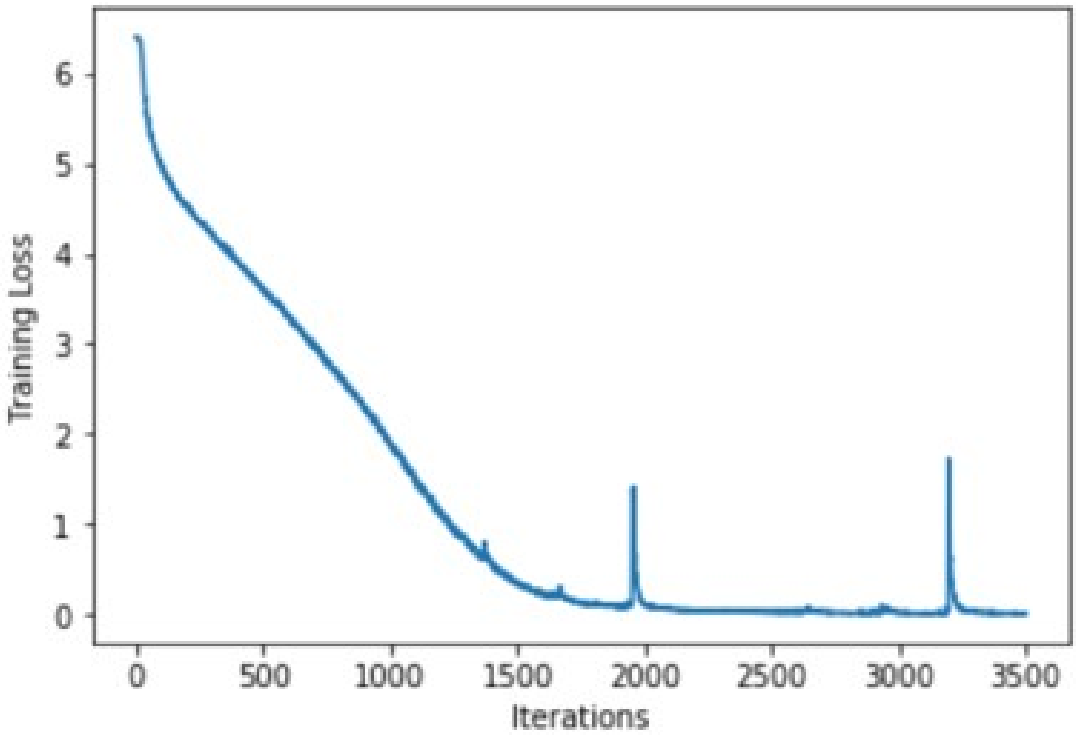

Figure 4.2: Training loss for autoencoder

Validation datasets were generated the same way as the training datasets. We used 1000 sample sets and compared the expected outputs with the outputs decoded from our autoencoder with attention model from elementwise. The validation accuracy can be visualized in Figure 4.8 below.

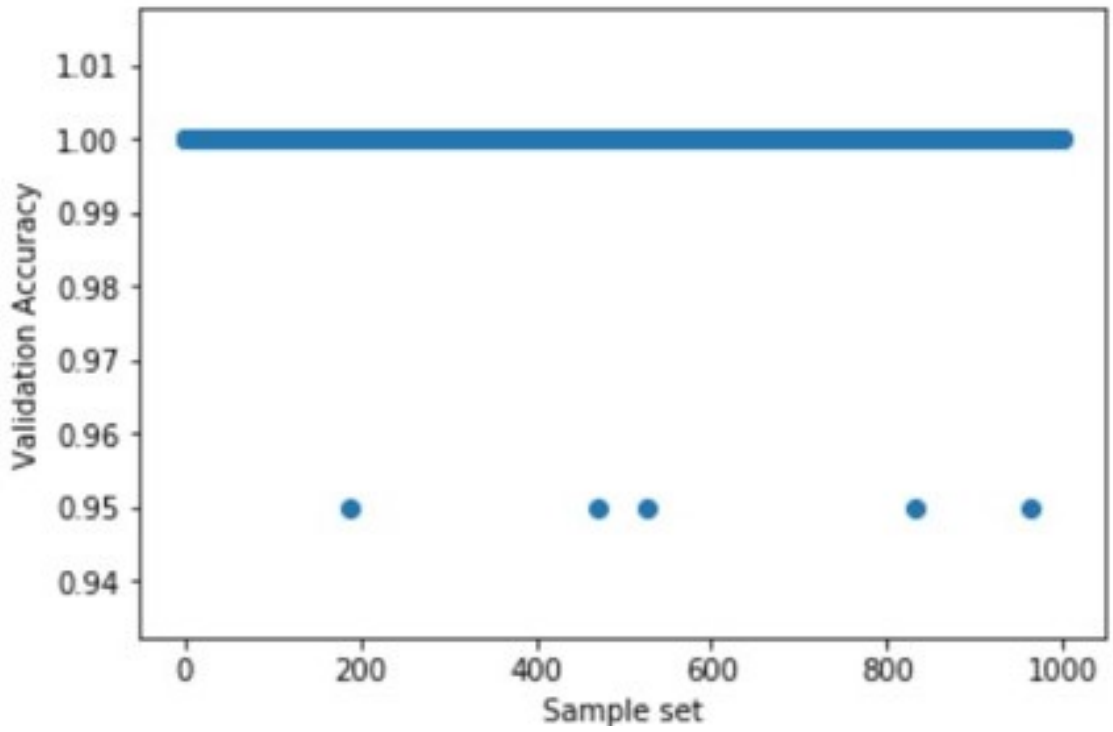

Figure 4.3: Validation accuracy for autoencoder

From the plot in Figure 4.8, we can observe that there were only 10 out of 1000 sample sets that were not decoded with 100% accuracy. But the least accuracy from validation data was still above 90%.

4.2.2 Autoencoder with predictor

After we were satisfied with the training result of encoder-decoder model, we saved this pre-trained model to disk and we started jointly training encoder-decoder model with predictor network. We set the hyperparameter epoch of size 40 and batch of size 100. The training loss of this joint model with respect to the number of epoch can be visualized in Figure 4.4 below.

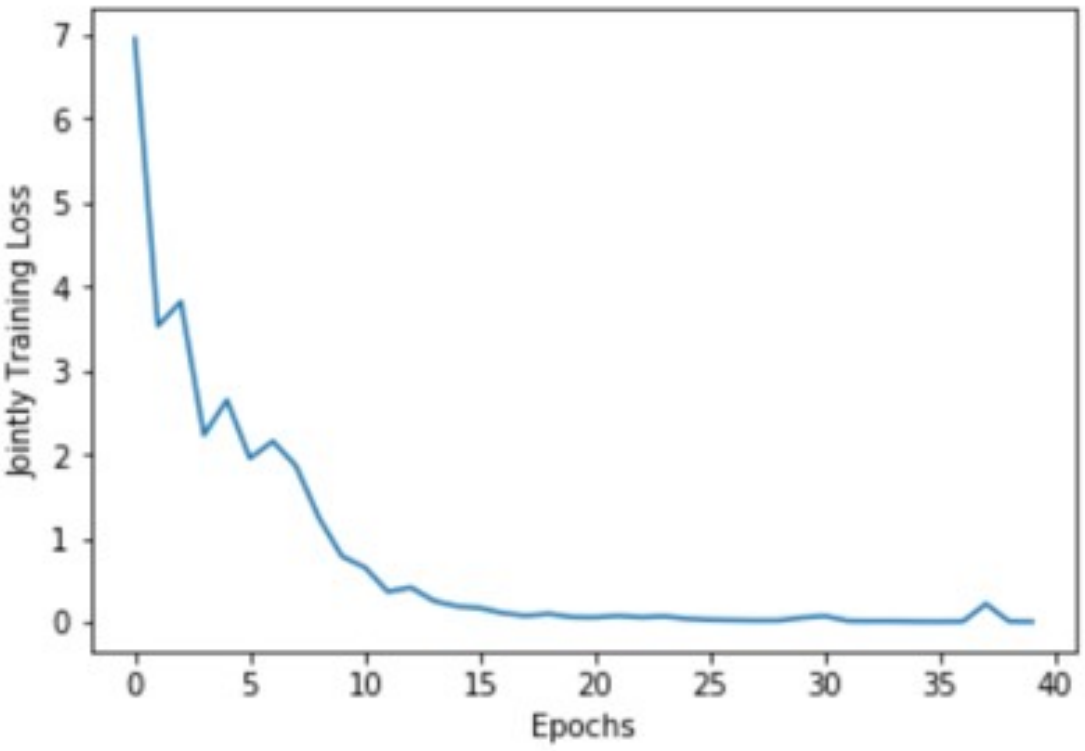

Figure 4.4: Joint training loss

After 40 epochs of joint training, the training loss has converged to a value < 0.008 and the meantime, we can also look at the validation loss for this model in Figure 4.5.

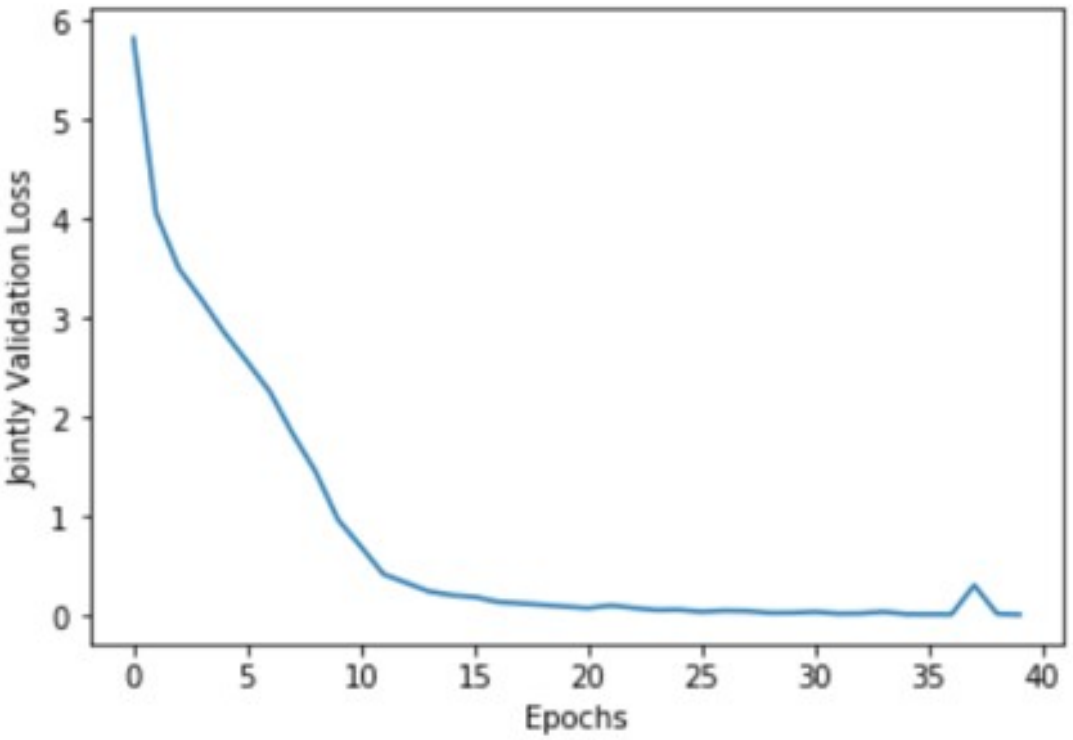

Figure 4.5: Joint validation loss

The validation loss plot for this autoencoder with predictor model shared the same trending of convergence as the training loss. This means our model not only performed well on the training datasets but also the validation datasets.

Since the encoder-decoder with attention is a pre-trained model, we assign a twice-larger weight parameter for predictor loss comparing with encoder-decoder loss so that we can emphasize the predictor-training portion. For training loss of encoder-decoder with attention, we have used the same cross-entropy (log) loss function.

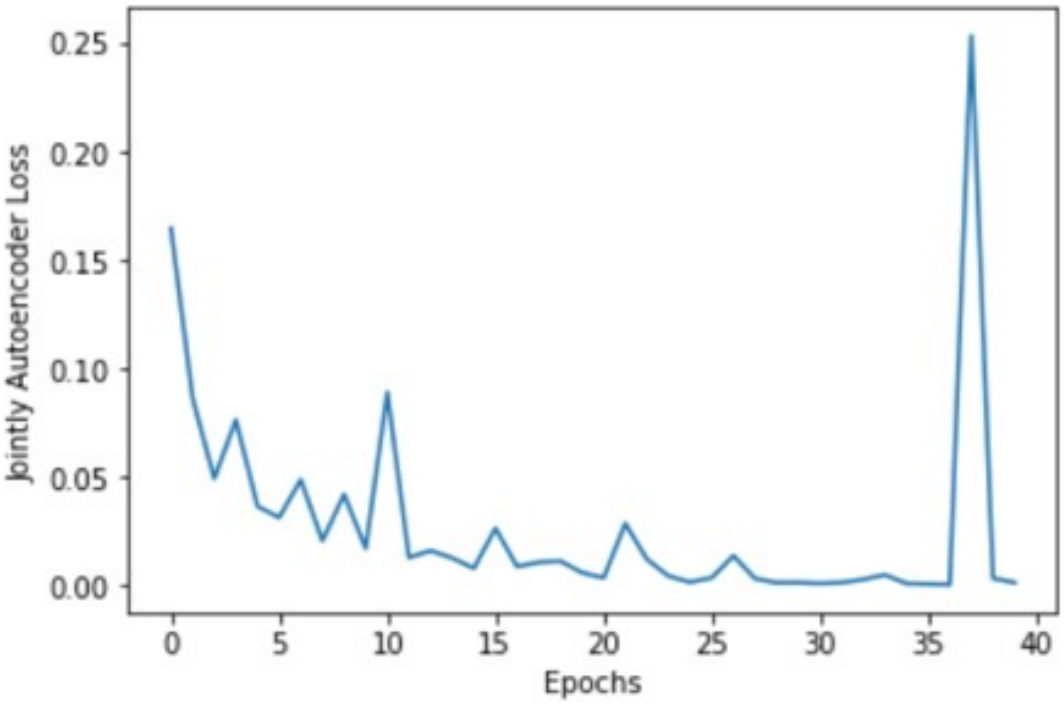

Figure 4.6: Encoder-decoder joint loss

Comparing with the joint loss, the encoder-decoder model started with a fairly low loss value as expected. There was an outlier point occurred between epoch 35 and epoch 40 but the majority of epoch results is close to zero. Unlike the encoder-decoder loss, we have chosen MSE regression loss function for predictor for single value output.

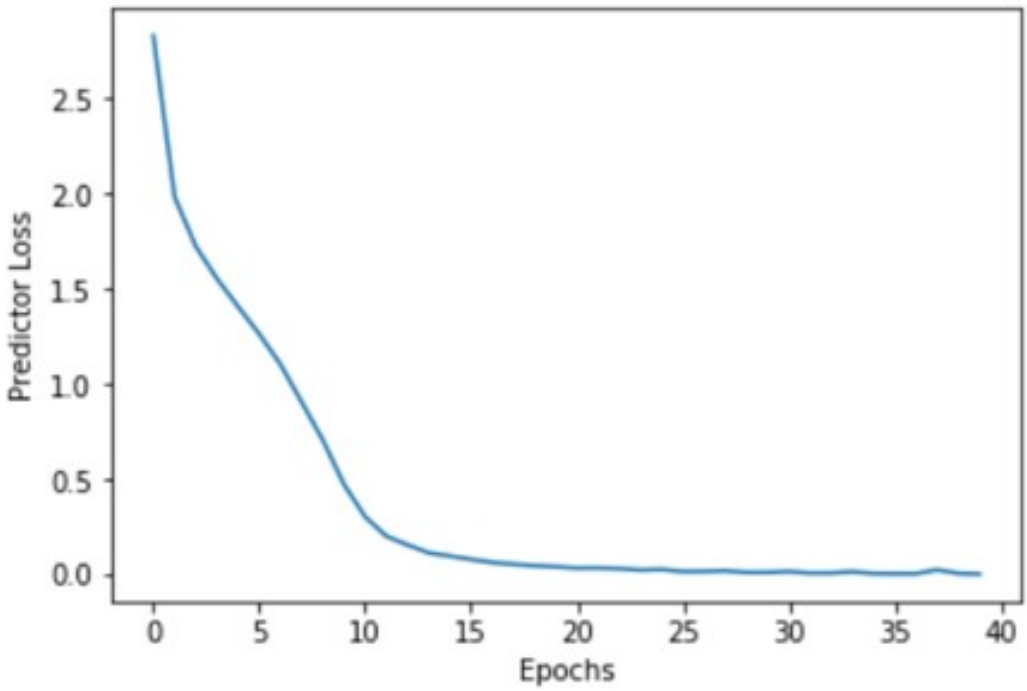

Figure 4.7: Predictor joint loss

Even though the predictor has not been pre-trained, the starting loss after one epoch was not too bad and it converged approximately after 10 epochs. From both the training loss and individual loss shown above, we expect the joint model predict the objective value of submodular function with high accuracy and also able to decode any sequence of numbers with high rate of success.

The best way to examine our model is to generate random validation datasets to see how the joint neural networks perform on them. Just like the individual testing, we generated 1000 sample sets with each size of 20 and obtained the validation accuracy as follows.

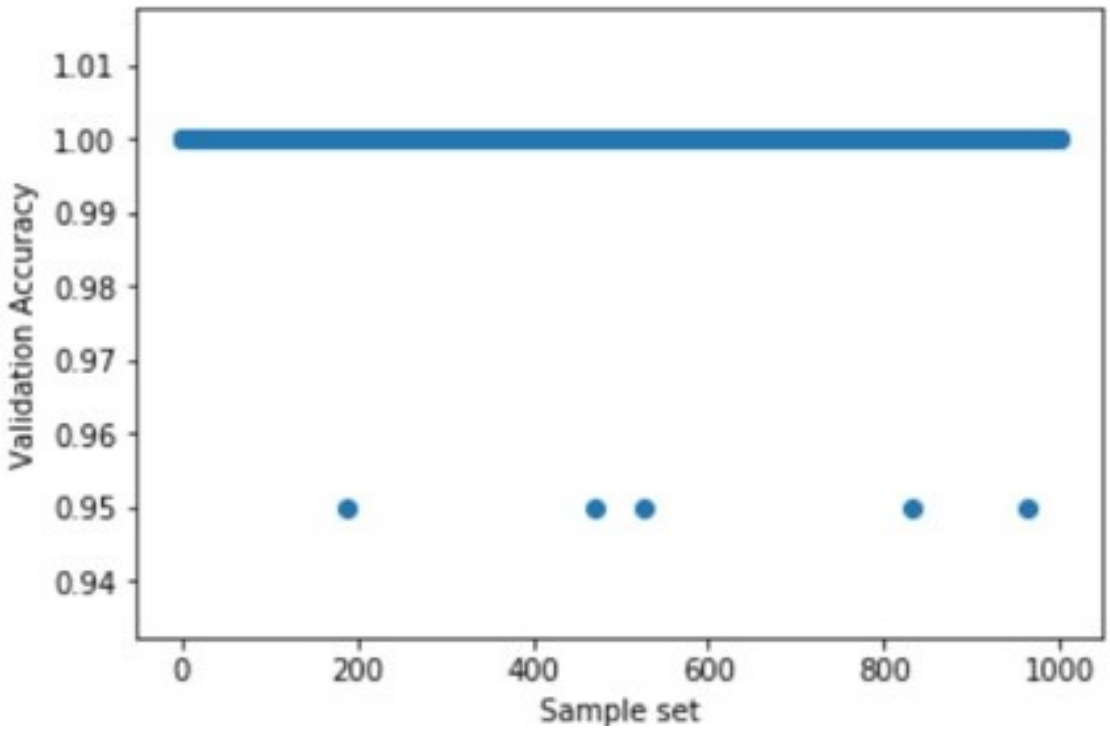

Figure 4.8: Joint validation accuracy

This time, there were only 5 sample sets received validation accuracies that are not 100%. This only represents the joint accuracy and we still want to take a look at the prediction accuracy on testing datasets to ensure the functionality of this newly trained predictor.

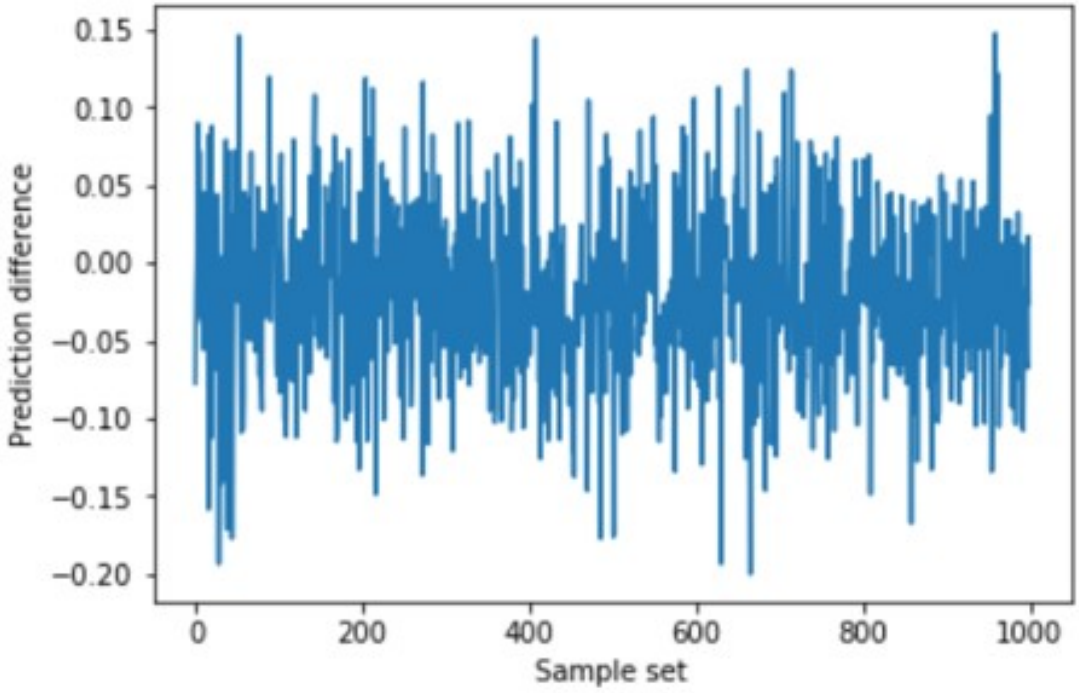

Figure 4.9: Prediction difference plot

We generated a plot of prediction difference, which indicated the difference between the expected objective value and prediction value from predictor. The difference ranged from -0.2 +0.15. We considered this range as qualified for fitting our target submodular function. It is determined that this model has been well-trained and we will then use it with optimizer together for final testing.

4.2.3 Optimizer

We have successfully trained and validated our encoder-decoder with predictor model for optimizing purposes. The final step of testing was to extract the input features from our encoder model and optimize that representation to achieve a higher objective value of submodular function after being processed by optimizer.

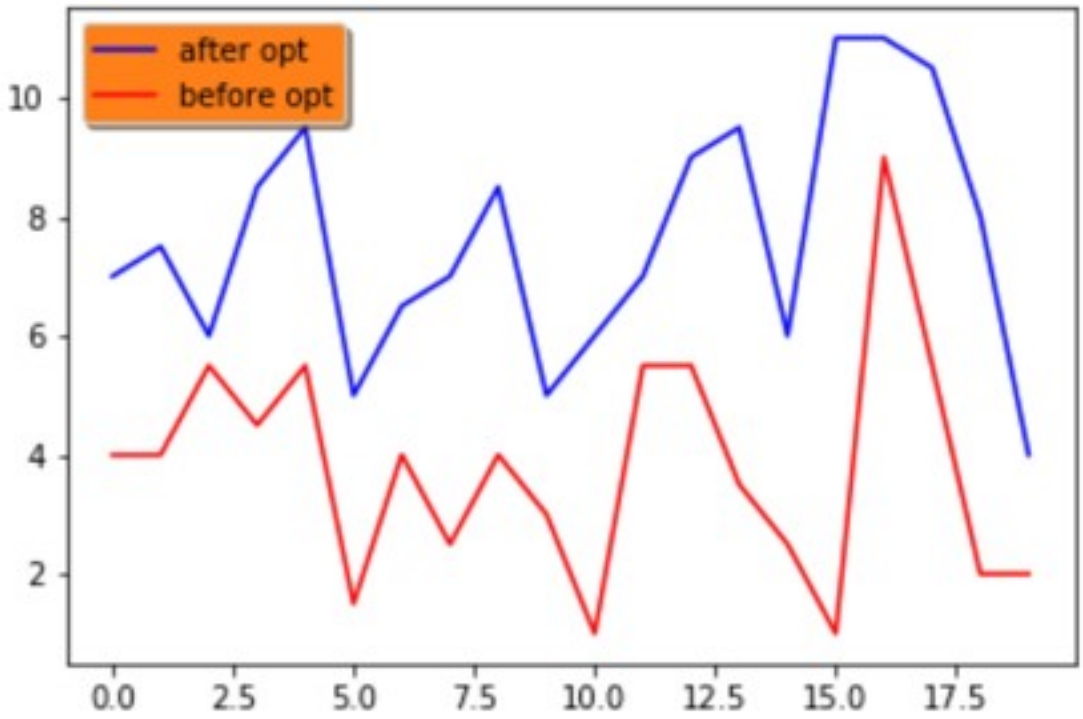

Figure 4.10: Function value comparison

The graph in Figure 4.10 was based on the first 20 sample sets out of 1000. It is clear to see that there was a significant improvement in objective value after going through optimizer with well-trained model. There was no overlapping between the two colored plots. We have also looked at the best sample value from 1000 testing datasets. The best sample has an objective value of 8.5. After this best sample got optimized, it was updated to 11.5.

We are satisfied with the overall result but there are definitely more things we considered for future work.

5. CONCLUSION

In this thesis, we have focused on proposing a different method on solving submodular maximization problem. We have combined different types of neural networks and made them work both effectively and efficiently. Our neural network now is able to take in a random set of distinct integers, predict its performance in simple submodular function, optimize it and decode the optimized representation back in numeric set form. We have clearly explained the structure of our model, which is comprised of an autoencoder, a predictor and an optimizer. The autoencoder has two separate recurrent neural networks consisting of stacks of LSTM layers and LSTM Cells. With the help of attention mechanism, which carries the information of encoder's input data with different identification scores, the autoencoder can efficiently learn a powerful representation for a set of data. The fully-connected neural networks, predictor and optimizer, can recognize the encoded representation generated from the encoder and predict its objective value and optimize such representation accordingly. And we can always take the optimized representation as decoder's input state to decode it back into a set form for future use. We have talked about the training strategy and parameter tuning for this model. We evaluated the model performance from the training and testing aspects and we have showed the results to prove the stability and liability of this model. We definitely have considered applying this model into different situations or into a larger scale and we have thought about where this model can be improved.

5.1 Future research and work

There are different parts of the model that we can alter to make it better and more general. The first thing that came into our mind was the limitation on input data. Our encoder-decoder is more suitable for inputs with an order; in other words, we expect

the input data to be a sequence of numbers. In our case, we have to give it an either descending or ascending order to make the output representation consistent with all possible inputs. If we want to make it more general for sets not just sequences, we have to consider not using the recurrent autoencoder neural network anymore. We need to design a permutation invariant neural network as the one proposed in [15]. Another part we think is worth exploring is the optimizer. Right now with our model, we do not have any approximation guarantee comparing with other classic way of solving submodular maximization problem. We wanted to focus on proposing an innovative idea of solving submodular maximization problem. But in the future research, we want to improve the optimizer to make it more precise and promising. We have thought about using input convex neural network and Q-learning algorithm to make this optimizer stay in feasible space during optimizing [16]. Because the problem we observed with our model was, when we had a large step along the gradient direction, we might step out of our feasible space. For example, after we optimize the encoded representation of a set, we found repeated elements when we decoded it. A set with repeated elements does not belong to our feasible space. And we have tested our model only on one simple submodualr function. We see the possibility of applying our model into more complex submodular functions and maybe just set functions in general.

REFERENCES

[1] E. Balkanski, A. Rubinstein, and Y. Singer, "The power of optimization from samples," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, p. 4024–4032, December 2016.

[2] V. N. Vapnik, *Statistical Learning Theory*. Wiley-Interscience, 1998.

[3] A. Krause and D. Golovin, "Submodular function maximization," in *Tractability* (L. Bordeaux, Y. Hamadi, and P. Kohli, eds.), ch. 3, pp. 71–104, Cambridge University Press, 2014.

[4] S. Dughmi, "Submodular functions: Extensions, distributions, and algorithms a survey," in *CoRR*, 2009.

[5] L. L., "Submodular functions and convexity," in *Mathematical Programming The State of the Art* (B. A., K. B., and G. M., eds.), pp. 235–257, 1983.

[6] J. Vondrák, C. Chekuri, and R. Zenklusen, "Submodular function maximization via the multilinear relaxation and contention resolution schemes," in *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pp. 7783–792, June 2011.

[7] N. Buchbinder, M. Feldman, J. S. Naor, and R. Schwartz, "Submodular maximization with cardinality constraints," in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, p. 1433–1452, January 2014.

[8] P. Baldi, "Autoencoders, unsupervised learning and deep architectures," in *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning workshop*, vol. 27, pp. 37–50, July 2011.

[9] R. J. Weiss, J. Chorowski, N. Jaitly, Y. Wu, and Z. Chen, "Sequence-to-sequence models can directly translate foreign speech," in *Interspeech*, August 2017.

[10] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *ArXiv*, vol. abs/1811.03378, 2018.

[11] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR*, 2015.

[12] K. Janocha and W. Czarnecki, "On loss functions for deep neural networks in classification," *ArXiv*, vol. abs/1702.05659, 2017.

[13] A. Goyal, A. Lamb, Y. Zhang, S. Zhang, A. C. Courville, and Y. Bengio, "Professor forcing: A new algorithm for training recurrent networks," in *NIPS*, 2016.

[14] J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *ArXiv*, vol. abs/1607.06450, 2016.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.

[16] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

APPENDIX A

TRAINING RESULT FOR $N = 300, K = 15$

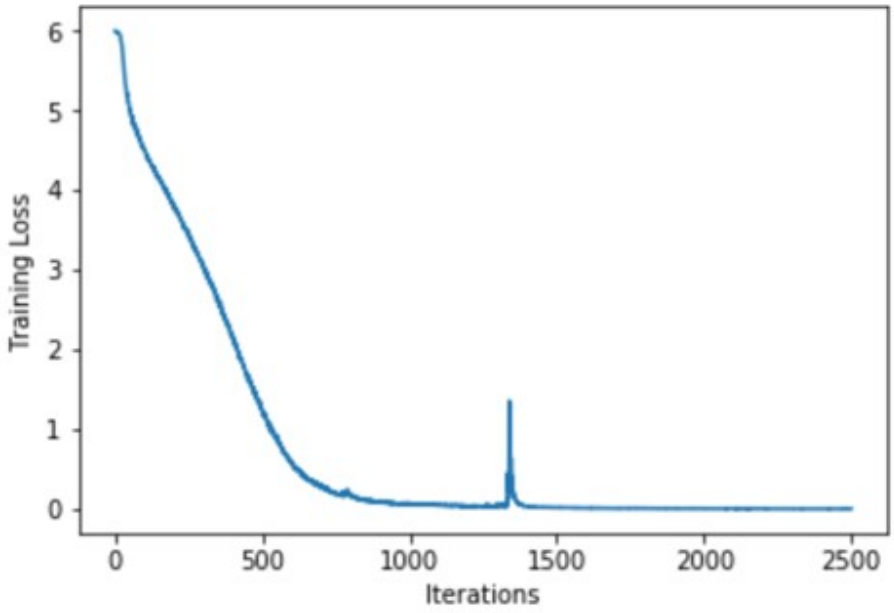

Figure A.1: Training loss vs. Iterations

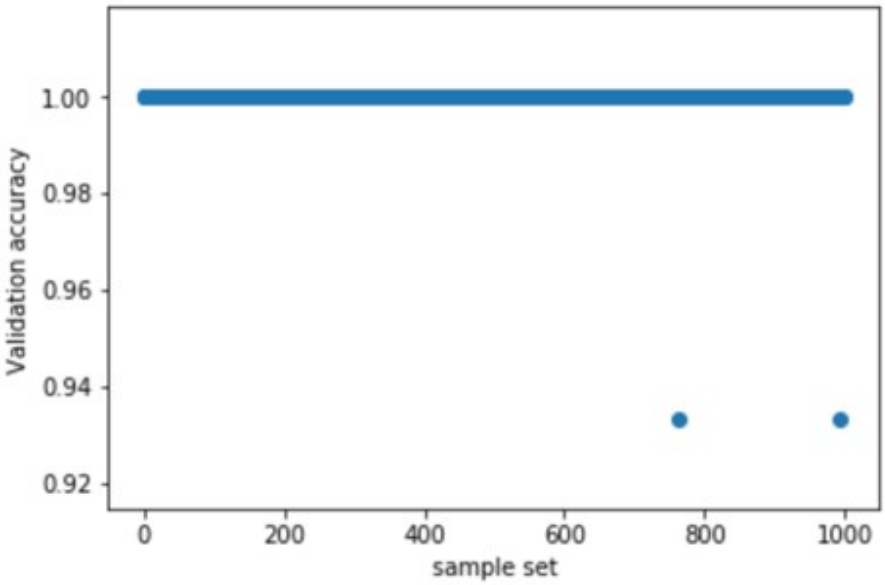

Figure A.2: Validation accuracy vs. Sample set index

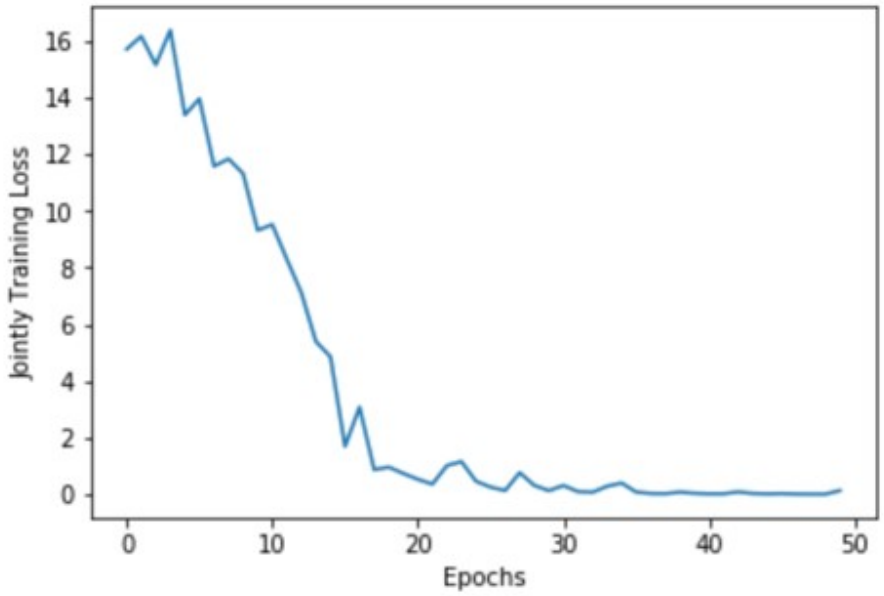

Figure A.3: Joint training Loss vs. Epochs

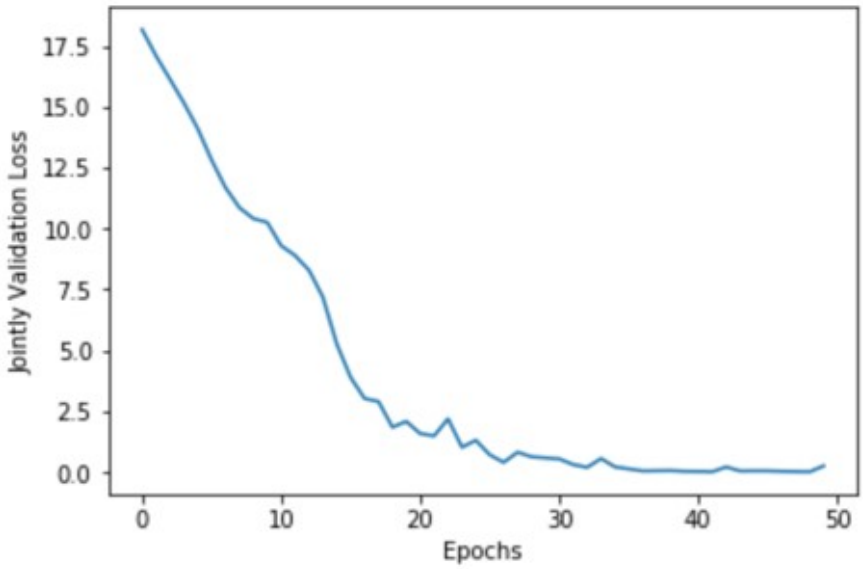

Figure A.4: Joint Validation loss vs. Epochs

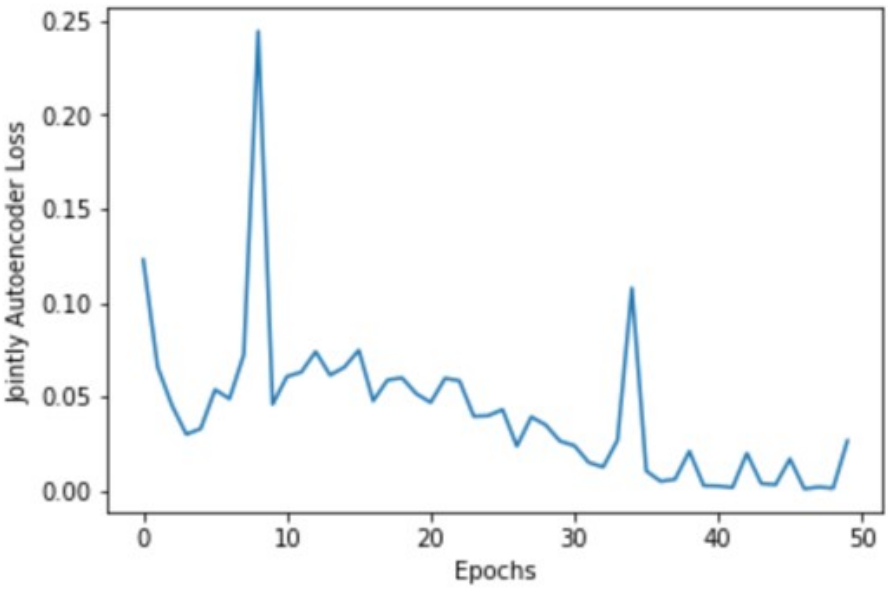

Figure A.5: Joint autoencoder loss vs. Epochs

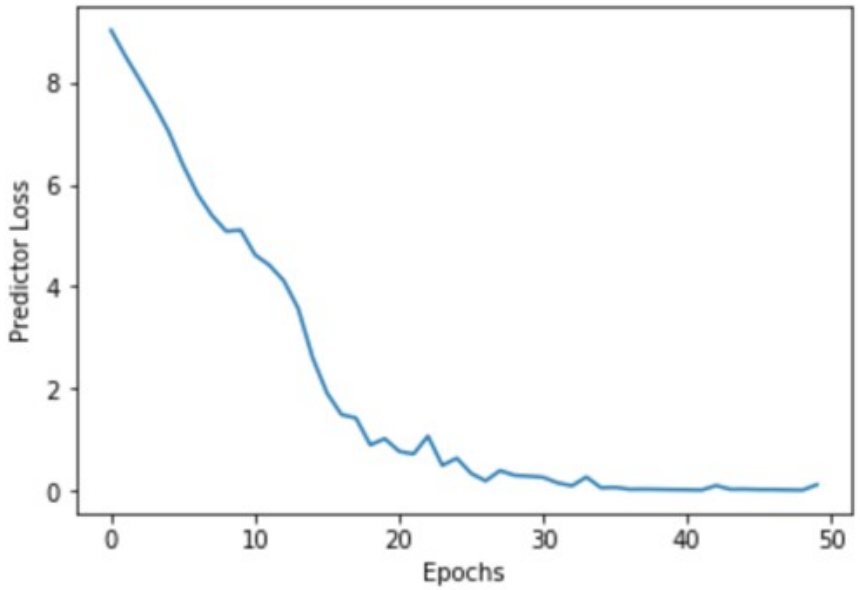

Figure A.6: Predictor loss vs. Epochs

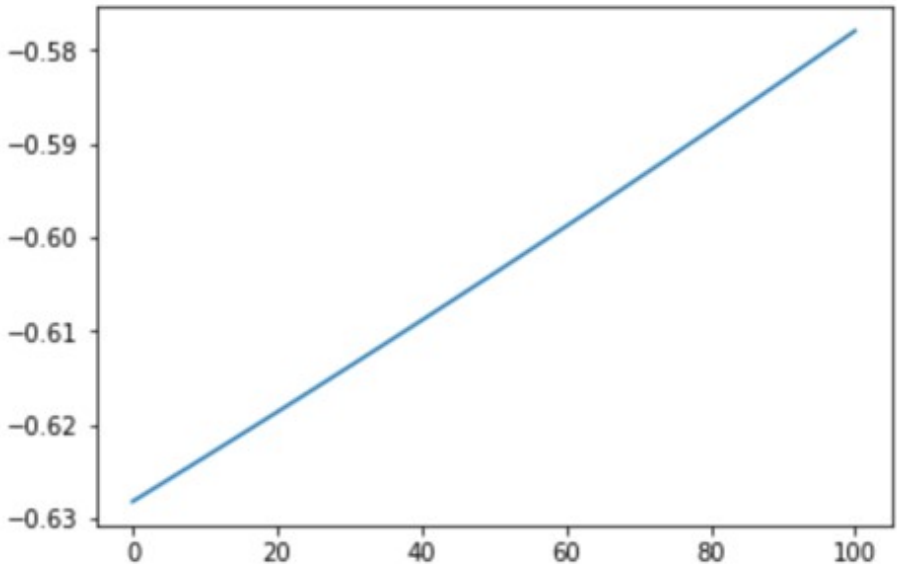

Figure A.7: Optimization vs. Step size

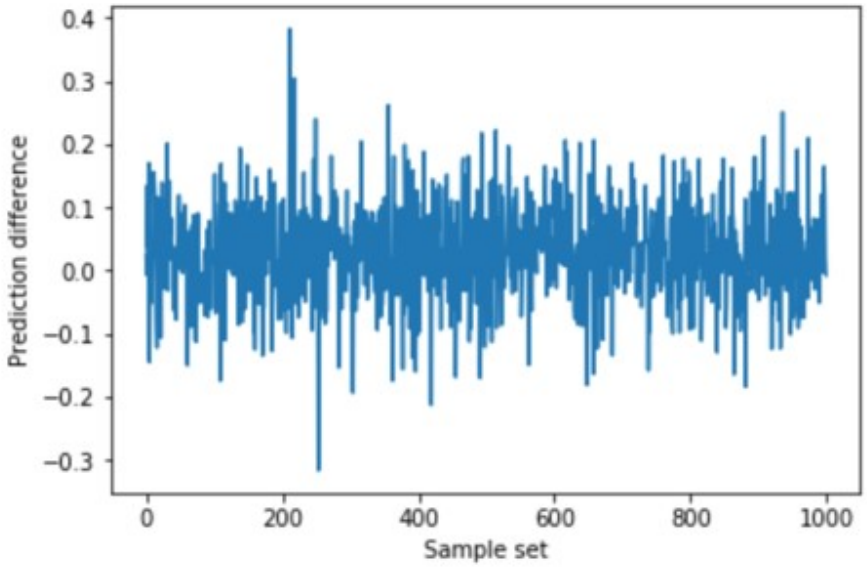

Figure A.8: Prediction difference vs. Sample set index

APPENDIX B

EPOCHS TRAINING RESULT FROM PYTHON

$epoch1 : train_loss : 6.956 val_loss : 5.826 ae_loss : 0.165 pp_loss : 2.830 epoch2 : train_loss : 3.533 val_loss : 4.056 ae_loss : 0.087 pp_loss : 1.985 epoch3 : train_loss : 3.820 val_loss : 3.498 ae_loss : 0.049 pp_loss : 1.724 epoch4 : train_loss : 2.232 val_loss : 3.186 ae_loss : 0.076 pp_loss : 1.555 epoch5 : train_loss : 2.648 val_loss : 2.852 ae_loss : 0.037 pp_loss : 1.408 epoch6 : train_loss : 1.955 val_loss : 2.560 ae_loss : 0.032 pp_loss : 1.264 epoch7 : train_loss : 2.157 val_loss : 2.258 ae_loss : 0.049 pp_loss : 1.105 epoch8 : train_loss : 1.865 val_loss : 1.835 ae_loss : 0.021 pp_loss : 0.907 epoch9 : train_loss : 1.246 val_loss : 1.457 ae_loss : 0.042 pp_loss : 0.707 epoch10 : train_loss : 0.787 val_loss : 0.966 ae_loss : 0.017 pp_loss : 0.474 epoch11 : train_loss : 0.654 val_loss : 0.697 ae_loss : 0.089 pp_loss : 0.304 epoch12 : train_loss : 0.364 val_loss : 0.416 ae_loss : 0.013 pp_loss : 0.202 epoch13 : train_loss : 0.415 val_loss : 0.329 ae_loss : 0.016 pp_loss : 0.157 epoch14 : train_loss : 0.256 val_loss : 0.242 ae_loss : 0.013 pp_loss : 0.114 epoch15 : train_loss : 0.192 val_loss : 0.204 ae_loss : 0.008 pp_loss : 0.098 epoch16 : train_loss : 0.171 val_loss : 0.187 ae_loss : 0.027 pp_loss : 0.080 epoch17 : train_loss : 0.110 val_loss : 0.137 ae_loss : 0.009 pp_loss : 0.064 epoch18 : train_loss : 0.079 val_loss : 0.121 ae_loss : 0.011 pp_loss : 0.055 epoch19 : train_loss : 0.102 val_loss : 0.107 ae_loss : 0.011 pp_loss : 0.048 epoch20 : train_loss : 0.062 val_loss : 0.089 ae_loss : 0.006 pp_loss : 0.042 epoch21 : train_loss : 0.059 val_loss : 0.073 ae_loss : 0.004 pp_loss : 0.035 epoch22 : train_loss : 0.078 val_loss : 0.101 ae_loss : 0.029 pp_loss : 0.036 epoch23 : train_loss : 0.060 val_loss : 0.076 ae_loss : 0.012 pp_loss : 0.032 epoch24 : train_loss : 0.071 val_loss : 0.053 ae_loss : 0.004 pp_loss : 0.024 epoch25 : train_loss : 0.043 val_loss : 0.056 ae_loss : 0.002 pp_loss : 0.027 epoch26 :$

$train_loss$: 0.030val_loss : 0.037ae_loss : 0.004pp_loss : 0.017$epoch$27 : $train_loss$: 0.025val_loss : 0.048ae_loss : 0.014pp_loss : 0.017$epoch$28 : $train_loss$: 0.022val_loss : 0.044ae_loss : 0.003pp_loss : 0.021$epoch$29 : $train_loss$: 0.022val_loss : 0.025ae_loss : 0.001pp_loss : 0.012$epoch$30 : $train_loss$: 0.055val_loss : 0.025ae_loss : 0.002pp_loss : 0.012$epoch$31 : $train_loss$: 0.075val_loss : 0.034ae_loss : 0.001pp_loss : 0.017$epoch$32 : $train_loss$: 0.018val_loss : 0.018ae_loss : 0.002pp_loss : 0.008$epoch$33 : $train_loss$: 0.015val_loss : 0.020ae_loss : 0.003pp_loss : 0.008$epoch$34 : $train_loss$: 0.016val_loss : 0.036ae_loss : 0.005pp_loss : 0.015$epoch$35 : $train_loss$: 0.010val_loss : 0.013ae_loss : 0.001pp_loss : 0.006$epoch$36 : $train_loss$: 0.008val_loss : 0.011ae_loss : 0.001pp_loss : 0.005$epoch$37 : $train_loss$: 0.012val_loss : 0.011ae_loss : 0.001pp_loss : 0.005$epoch$38 : $train_loss$: 0.219val_loss : 0.305ae_loss : 0.254pp_loss : 0.026$epoch$39 : $train_loss$: 0.012val_loss : 0.017ae_loss : 0.003pp_loss : 0.007$epoch$40 : $train_loss$: 0.008val_loss : 0.008ae_loss : 0.001pp_loss : 0.003