AUTO ENCODER BASED ON TEMPORAL CODING IN A SPIKING NEURAL NETWORK

Kaluhath Dhanushka Niroshan De Abrew

179313A

M.Sc. in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2020

AUTO ENCODER BASED ON TEMPORAL CODING IN A SPIKING NEURAL NETWORK

Kaluhath Dhanushka Niroshan De Abrew

179313A

Thesis/Dissertation submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2020

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

[illegible] ....   29/05/2020 ....

K.D.N. De Abrew   Date

I certify that the declaration above by the candidate is true to the best of my knowledge and that this project report is acceptable for evaluation for the CS6997 MSc Research Project.

....

Dr. Charith Chitraranjan   Date

ABSTRACT

Auto Encoders using Artificial Neural Networks have achieved a high level of regeneration accuracy whereas Auto Encoders using Spiking Neural Networks are still in their early stage and only a few SNN Auto Encoders have been introduced but with lesser accuracies compared to ANN Auto Encoders. Using SNNs for Auto Encoders is desired as SNNs are one step closer to understand the communication and processing in biological neural networks. Sparse discrete events known as Spikes make SNNs energy efficient especially when implemented using Neuro-morphic hardware and Temporal coding scheme with the mapping of 'input value to time of the first generated spike' makes it even more efficient in terms of power consumption and time to generate an output where power consumption and time to encode/decode are the key metrics.

However, the direct application of gradient descent methods is not possible for SNN as the activation functions are non-differentiable. Training an Auto Encoder requires a way to adjusting the network parameters so that the reconstruction loss is minimized. Due to the lack of such training models for SNNs especially with multiple hidden layers, it is a challenging task to implement an Auto Encoder using SNN. In this research, models enable such learning, were analyzed with the aim of selecting a promising model. Based on the selected model for adjust the synaptic weights of the network, an SNN Auto Encoder model is developed which allows the user to configure the network structure and number of neurons in each layer to achieve the desired compression ratio. Considering the demonstrated reconstruction accuracy and convergence rate of the SNN Auto Encoder, it can be concluded that the introduced model is one of the first models which enables to use multilayer Auto Encoder using Spiking Neural Networks.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Dr. Charith Chitraranjan, for his guidance and invaluable assistance by providing me useful knowledge materials, advice, and supervision throughout this research work. His expertise and encouraging guidance enabled me to complete my work successfully.

I am indebted to my parents for their love and untired support throughout my life.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

SNN	Spiking Neural Network
ANN	Artificial Neural Network
STDP	Spike Time Dependent Plasticity
IF	Integrate and Fire
LIF	Leaky Integrate and Fire
SRM	Spike Response Model
PSP	Postsynaptic Potential
SGD	Stochastic Gradient Decent
MSE	Mean Squared Error
PSNR	Peak Signal to Noise Ratio

1. CHAPTER 1

INTRODUCTION

1.1 Auto Encoder

Auto Encoder is implemented using a neural network to find out a compressed representation for a data set through unsupervised learning. Auto Encoder neural network consists of both Encoder and Decoder so that training of the network to Encode and Decode (i.e. Reconstruction) is done simultaneously. Input is clamped to the Output and bottlenecks are enforced at hidden layers as shown in Figure 1.1 so that all the information cannot be propagated from the input layer to the output layer in a linear manner and hidden layers provide a compressed representation for input and noise in the inputs is removed.

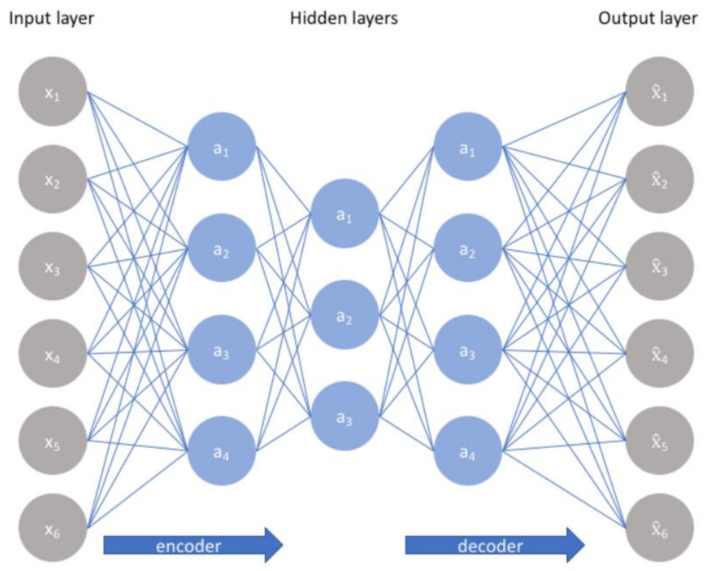


Figure 1.1 : Auto Encoder

Auto Encoders are used in various applications and major applications are provided below.

- Dimensionality reduction
- Feature Extraction
- Anomaly detection
- Noise reduction

1.2 Learning in Auto Encoder

Learning in auto Encoder falls under unsupervised learning as labels are not applicable in this context and the compressed representation which is learned/revealed is completely system decided. However, ultimately the output from the decoder should be the same as the input to the encoder. Since, both the encoder and the decoder are trained simultaneously using a single neural network, the desired output is known for an input. Hence during the training, once an input is fed, parameters in the network need to be adjusted so that actual output is similar to the desired output (For Auto Encoders, the Desired output is same as the input as the aim is to reconstruct the input at the final layer of the neural network). Therefore, unlike other unsupervised learning tasks, the error between the desired output and actual output needs to be calculated and parameters need to be adjusted accordingly to minimize the difference between desired and actual output.

1.3 Auto Encoders using Spiking neural networks

Conventional neural networks are based on neurons with mathematically defined activation functions and continuous static (constant) input/output values of a neuron within a computational clock cycle. Therefore Conventional neural networks are fundamentally different from the mechanism which is used for signal passing in the

Mammalian Brain. SNN mimics the mechanism used in the brain for signal passing where signals are passed as spikes and output from a neuron is generated due to membrane potential difference reaching a threshold due to an input current spike. Information is coded in spike rates and precise timing of spikes. Hence SNNs are considered a closer step for understanding how the biological neural networks operate.

Also, SNNs are considered as low power consuming neural networks when implemented on neuro-morphic hardware due to their sparse nature of spike occurring instead of static continuous signals and maybe faster and lower computational cost than ANNs depending on the underlying mathematical model used for implementation.

However, due to the difficulty in deriving differential equations for the activation functions, models for training SNN using backpropagation are still not well established specially for deep networks. There are several models that enable training SNN without using backpropagation such as STDP, etc. Compared to ANNs, SNNs lag in terms of accuracy.

1.4 Neuron Model for SNN

1.4.1 Leaky Integrate and Fire

In this model, the input current to a neuron is integrated and membrane potential in increased as a result. Once the membrane potential is reached a threshold, output spike is generated and the membrane potential is reset to a resting value and will not be fired until a predefined refectory period. The neuron is modeled as a capacitor in parallel with a resistor as shown in Figure 1.2.

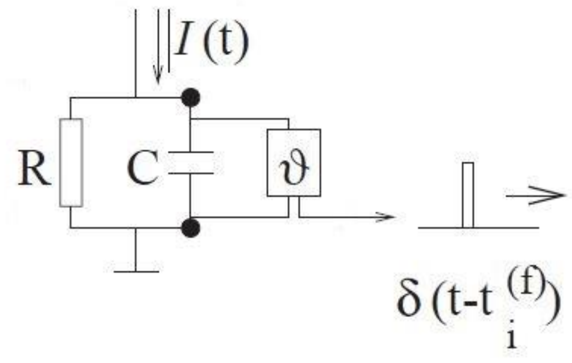


Figure 1.2 : LIF Neuron modeled as a RC circuit

Below linear differential equation can be derived by applying equations in electrical domain to the aforementioned RC circuit.

$$\tau_m \frac{du}{dt} = -U(t) + RI(t) \qquad (1)$$

U(t) is the potential at time t and τ_m is known as the time constant and equal to RC.

This equation can be extended further to derive equations to explain the dynamics of the neuron when a constant input current (I0) is given as shown in Eq. (2) and an input spike current is given (Dirac Delta function) as shown in Eq. (3) where q is the total charge from the input spike current.

$$U(t) = RI_0\left[1 - \exp\left(-\frac{t}{\tau_m}\right)\right] \qquad (2)$$

$$U(t) = q\frac{R}{\tau_m}\exp\left(-\frac{t}{\tau_m}\right) \qquad (3)$$

1.4.2 Integrate and Fire (IF) (Non - leaky)

In contrast to Leaky Integrate and Fire model, only a capacitor is present in the equivalent circuit for Integrate and fire model. Hence, potential is derived by integrating the current over time since last output spike ($\hat{t}$) as shown in equation 4.

$$u(t) = \frac{1}{C}\int_{\hat{t}}^{t} I(t)\,\mathrm{dt} \qquad (4)$$

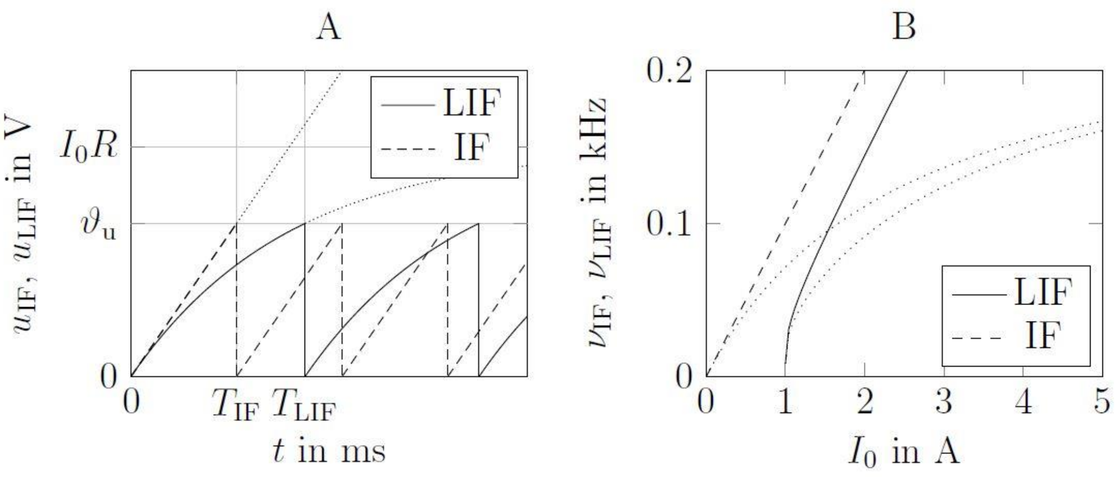


Figure 1.3 : A: Potential increase in LIF and IF models with time for a constant input current, B: Firing rates against input current for LIF and IF models

Figure 1.3 shows how membrane potential is changed when input current is given in LIF and IF neuron model and how firing frequency changes with the magnitude of the constant input current.

1.4.3 Spike Response Model (SRM)

In contrast to Leaky Integrate and Fire model, in SRM, membrane potential depends on the effects of input spikes since the last generated output spike. In this model, 3 Kernel functions are used to model how the membrane potential changes respectively when an input current is given, when a presynaptic neuron is fired and Hyper-polarization after the output spike is generated. Hyper-polarization is shown in figure 1.4. In addition, the increase in firing threshold after and output spike is generated is further discussed.

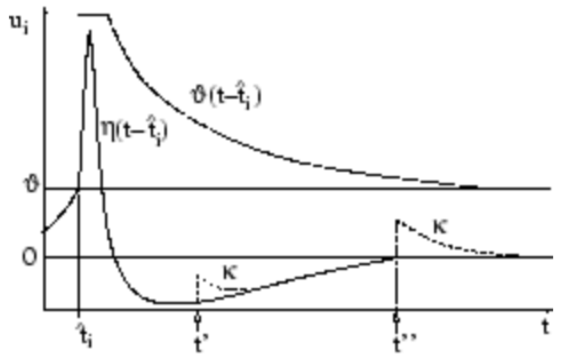


Figure 1.4 : Kernel functions of SRM corresponding to each phase of membrane potential change when an output spike is generated

Eq. (5) represents the membrane potential as a function of combining all three Kernels as follows. η represents the Hyper-polarization kernel and the last term of the equation represents the kernel for the behavior when an input current is given. The second term defines the effect on membrane potential when input spikes are received from presynaptic neurons where w_{ij} represents synaptic weight between output neuron *'i'* and an input neuron *'j'*. ε represents the kernel for membrane potential change due to an input spike occurred at time $t_j^{(f)}$.

$$U_i(t) = \eta(t-\hat{t}_i) + \sum_j W_{ij} \sum_f \varepsilon_{ij}(t-\hat{t}_i, t-t_j^{(f)}) + \int_0^\infty K(t-\hat{t}_i, s).I^{ext}(t-s)ds \qquad (5)$$

1.5 Neural coding

1.5.1 Rate Coding

In this model, it is assumed that information is encoded in the firing rate of spikes. It has been observed that Rate Coding is used in Motor systems in early experiments. Spike rate is scalar value averaged over a time period. Hence, small change spike count due to noise does not impact the output drastically and therefore Rate Coding is considered to be a noise-tolerant model [1].

There are 3 variations of Rate coding based on the way of averaging as follows. [2]

- Spike count rate averaged over a small time period (temporal average). Here, the time period will be enough to capture spikes generated only for one input.
- Rate as a Spike Density Averaged over several runs. This type of coding mechanism is highly unlikely to be used in biological neural networks as the networks should be trained for each input sequence without waiting for repetitions.
- Rate as a Population Activity (Average over several neurons). In this mode, it is assumed that a set of identical neurons are present in a network and Rate is derived for the neurons in the population and the model is backed by parts of the mammalian visual cortex where segments of the cortex are responsive to different types of inputs.

1.5.2 Temporal Coding

In Temporal coding, the main assumption is Information is coded in Precise timing of the spike instead of the rate of spikes generated for an input. Compared to rate base coding, noise may have major adverse effects on the accuracy of an output from a neural network which uses temporal coding as a minor change in the time of spike will change the output drastically. Hence, robustness in temporal coding is low compared to rate-based coding. Some experimental researches suggest that temporal coding may be used in the brain as the recognition of objects is done within few milliseconds which can be done by processing the first spike only and it is unlikely that based coding is used as processing the rate after receiving several spikes is time consuming [2].

1.6 Motivation

Though ANNs are inspired by biological networks, they deviate from biological networks in terms of the mechanism used for passing signals through the network. Even though Auto Encoders have been developed using Conventional Neural Networks, only a few researches have been carried out in the domain of Auto Encoders using Spiking Neural Networks. Especially there have been no researches that use Auto Encoder with multiple hidden layers as per the knowledge of the author.

Also, SNN is considered as the 3rd generation Neural Networks as they are one step closer to understand the mechanism of neural processing of the brain. Also, it is considered that temporal coding may be the dominant coding scheme used in the brain due to very low latency for certain tasks.

Using Spiking Neural Networks for an Auto Encoder instead of Conventional Neural networks provides inherent advantages of SNNs such as low energy consumption (especially for portable devices), faster outputs which are desirable for applications of Auto Encoders such as Compression (Dimensionality reduction) and Noise reduction which might make them useful for real time applications once the network is trained.

Temporal Coding scheme will further improve the speed and the energy efficiency of network as it is not required to wait for all the spikes generated for an input during computational clock cycle / maximum period to be waited (as per the range used in temporal coding scheme) and only a handful of spikes is needed to be processed and output can be provided.

Hence, developing a model for an Auto Encoder using the temporal coding scheme with SNN is an appealing area for research.

1.7 Objective

The main reason for lack of development of models for Auto Encoder using SNN is that usually Auto Encoders are trained using back propagation and back propagation cannot be directly applied for Spiking Neural Networks.

Researches that explores methods of supervised learning for SNN using back propagation and other learning mechanisms are discussed in the literature review section.

The main objective is to come up with a model to use Spiking Neural Network with temporal coding to develop an Auto Encoder with multiple layers.

2 CAHPTER 2

LITERATURE REVIEW

Even though LIF model is widely used for modeling neurons in SNN as it closely approximates the actual behavior of biological neuron, it is difficult to use them for computations (especially for supervised learning) as the equation which defines the neuron dynamics is nonlinear.

Most of the models which are based on spikes use an equivalent version of ANN where weight learning is done by converting spike rates / temporal coding to continuous static value and feeding it to the equivalent ANN or vice versa (i.e. Train equivalent ANN and use those weights for SNN). [3]

2.1 Spike Time Dependent Plasticity (STDP) for learning

2.1.1 Spike Time Dependent Plasticity (STDP)

STDP is believed to be the actual learning mechanism that is used in biological neural networks. In STDP, the Weight of a connection between two neurons is adjusted based on the time gap between spikes from pre-synaptic and post-synaptic neurons. If the postsynaptic neuron is fired briefly after the pre-synaptic neuron is fired, the connection weight is increased (Long Term Potentiation (LTP)) and if the postsynaptic neuron is fired briefly before the presynaptic neuron is fired, the connection weight is decreased (Long Term Depression (LTD)). Figure 2.1. Amount of weight change depends on the time gap between spikes from pre-synaptic and post-synaptic neurons. (5)

$$\Delta w = \begin{cases} A_1 \exp\left[\dfrac{-\Delta t}{\tau_1}\right] & if \quad \Delta t > 0 \\ A_2 \exp\left[\dfrac{-\Delta t}{\tau_2}\right] & otherwise \end{cases} \tag{5}$$

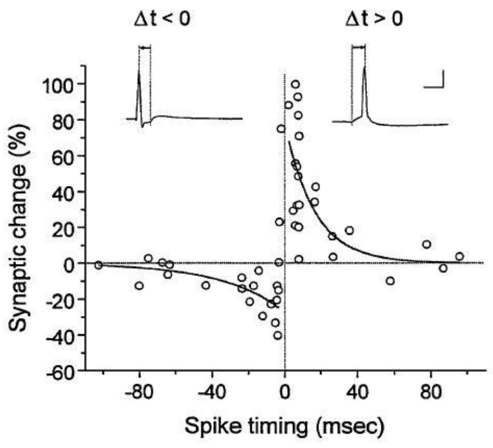


Figure 2.1: Synaptic weight change against time gap between pre/post synaptic neuron spikes

2.1.2 Learning in SNN using STDP

STDP has been used for unsupervised learning and has achieved high accuracy rates compared to other SNN models. And STDP learning rule has been used with other network models such as CNN, with the aim of increasing the accuracy while getting a step closer to biologically plausible neural networks. In [3], deep neural networks using Spiking neurons have been implemented where the input layer of the network consists of temporal coding (which coverts input image to a spike train) and remaining layers correspond to Convolutional and Pooling layers. In convolutional layers, input spikes are integrated and once the threshold is reached, output spikes are generated and learning is done using STDP. This model has achieved an accuracy of 98.4% for MNIST dataset.

Some of the models (especially, in classification) use competition strategy (‘winner takes all’) with STDP to prevent other neurons from firing except for the winning neuron [4], [5].

Due to the nature of learning rule of STDP, basic STDP model is not suitable for learning by comparing actual output and desired output at final layer of the network as in STDP, weights are not adjusted based on the error between expected output and actual output, but based on the exact fire times of pre and postsynaptic neurons. Hence, learning is STDP is considered as a local training mechanism. Hence, the basic form of STDP cannot be used to train an SNN Auto Encoder [6].

However, a model has been developed for Auto Encoder using STDP learning known as mirrored STDP (mSTDP) [5]. In the Mirrored STDP model, only two layers are there. One layer does the encoding and the other layer does the decoding. But in the brain, recognizing visual patterns is done using neural networks which consist of multiple layers. Hence, this model deviates from the neural network that is used in the brain.

Also, few supervised learning models have presented which uses STDP and some of them are discussed below [7], [8].

2.2 Learning in SNN using temporal coding

Learning in Conventional Neural Networks (ANN) is based on back propagation since differential equations which are used can be fairly easily derived for a set of activation functions including Sigmoid, tanh, ReLU, etc whereas it is very difficult to use back propagation for learning in SNNs as deriving differential equations is not straightforward due to the nature of the underlying mechanism used for passing information in SNN. i.e. instead of continuous values, both inputs and outputs are events (spikes) which are occurring within a certain time period determined by the dynamics of the firing model of neurons.

Some of the researches have been carried out using substitute models for biological neuron and derivatives have been obtained and hence enables to use of back propagation for training[9],[10].

Several researches have been done and few models have been developed to enable learning in SNN such as SpikeProp, ReSuMe, BP-STDP, and some well-recognized models are further discussed in below sections.

2.2.1 Spike Prop

SpikeProp model [8] one of the earliest but successful models that enable supervised learning in SNN. In this model, the Spike Response Model is used to model the output spike generation when input spikes are given to a neuron. The kernel used in the Spike Response model is given by equation (6)

$$\varepsilon(t) = \frac{t}{\tau}\exp\left[1 - \frac{t}{\tau}\right] \qquad (6)$$

$$x(t) = \sum_k w_k . \varepsilon(t - t_i - d_k) \qquad (7)$$

Fully connected feed forward networks are used and a connection between a pre-synaptic neuron and postsynaptic neuron is considered to be consist of several sub connections (synaptic terminals) each with different weights and predefined delays as shown in figure 2.2. PSP dynamics of a neuron due to an input from a presynaptic neuron is given by equation (7) where w_k is the weight of a sub connection, t_i is the input spike time and d_k is the delay associated with sub connection k. Also, it is assumed that a neuron can fire only once for an input. This is imposed in the model by using a relatively large postsynaptic time constant compared to the max value used in the temporal coding used in the model.

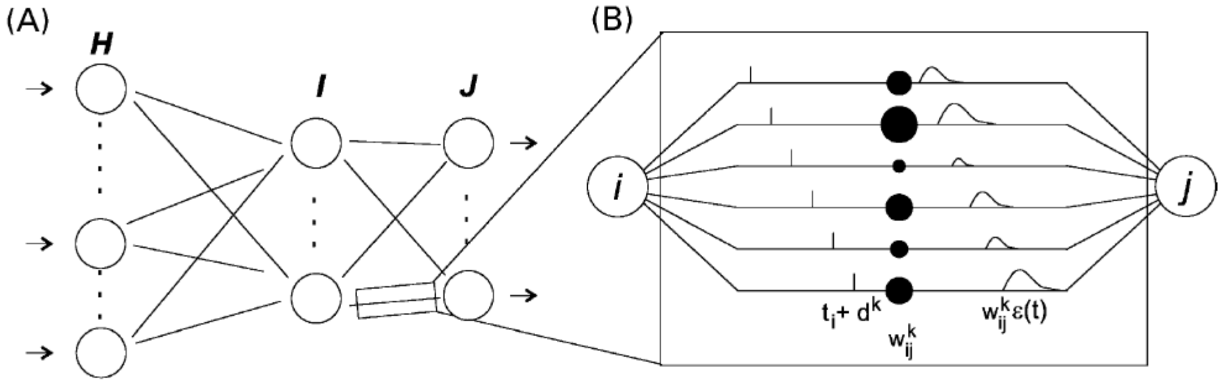


Figure 2.2: Single connection between two neurons consist of different delays and weights

Another important assumption made in this model is that the relationship between the input to postsynaptic neuron and output spike time is linear for a small range around the firing threshold. Due to this assumption, small learning rates must be used when using this model. Using this assumption together with an equation for PSP in the Spike Response model, equations for Error derivate with respect to weights of each sub connection and input spike time from a hidden layer neuron are derived. These two equations enable back propagation for learning in SpikeProp Model.

Spike Prop has achieved accuracies of 96.1% for IRIS data set and 97.0% for Wisconsin breast cancer dataset. However, the assumption of a single connection between two neurons consisting of multiple sub connections with predefined delays contradicts with the actual structure of how biological neurons are connected and spikes are propagated between them.

2.2.2 ReSuMe (Remote Supervised Method)

In 'ReSuMe' model [11] enables supervised learning and it is based on Widrow-Hoff rule (also known as delta rule). Widrow-Hoff rule originates from the idea of gradient descent especially for linear neural networks with the idea of updating the connection weights based on the difference between the desired output and actual output as shown in equation (8).

$$\Delta w = \alpha\, x_i\,(y_d - y_o) = \alpha\,(x_i y_d - x_i y_o) \qquad (8)$$

In 'ReSuMe' model, the Widrow-Hoff rule has been applied by modeling input and output from functions of Spike trains (spikes are modeled using Dirac delta function, hence spike train is denoted by a sequence of delta functions). Hence, Equation 8 can be seen as a combination of STDP processes, where the first term indicates how weight is adjusted according to the correlation between input (Pre-Synaptic) spike train and desired output Spike train. The second term expresses the anti-STDP process where weight is adjusted according to the correlation between input (Pre-Synaptic) spike train and actual output Spike train. The first STDP process derived analytically and has no physical connection between input and desired output spikes, hence term 'Remote' is added to the model name.

Using equations for correlation of Spike trains and assuming similar coefficients and constants for two STDP processes (property of symmetry), learning rule is derived as shown in equation (9). Hence, in ReSuMe model, supervised learning is achieved without deriving the equation for back propagation. Weight adjustment as per the precise time of spikes for an example scenario is shown in Figure 2.3.

$$\frac{dw_{oi}(t)}{dt} = [\mathrm{S_d}(t) - \mathrm{S_o}(t)]\left[\mathrm{a_d} + \int_0^{\infty} a_{di}(s)\, S_i(t-s)\, ds\right] \qquad (9)$$

Where,

$$a_{di}(s) = \begin{cases} A_{di}\, e^{(\frac{s}{\tau_{di}})} & if\ \ s \leq 0 \\ -A_{id}\, e^{(\frac{s}{\tau_{id}})} & otherwise \end{cases}$$

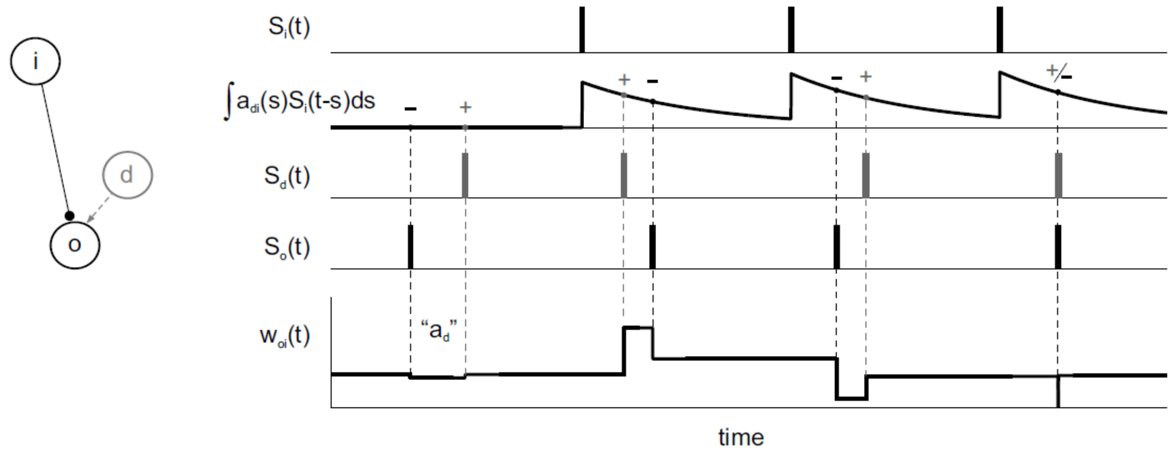


Figure 2.3: Weight adjustment with Spike trains. Si(t) – Input, So(t) – Actual Output, Sd(t) – Desired Output

However, this model is applicable only for single layer neural networks and the basic model is not directly extendable for the multi-layer network. This model can be extended for multi-layer networks. Though the model is simple compared to other gradient decent models, the model has not achieved a high level of accuracies.

2.2.3 BP-STDP

In this model, researchers have approximated the Integrate and Fire (IF) model to ReLU (Rectified Linear) activation function. And then they have shown that function for weight change composed of a STDP process and an anti-STDP process by applying the derivative function of RELU for IF model in SNN [7].

In ReSuMe also (in the previous topic), they have shown that the function for weight change consists of a STDP process and an anti-STDP process for linear neural networks, but the methods of deriving this function for weight change is different. Hence, the importance of BP-STDP model is that it has been shown gradient decent (subjected to the aforementioned assumptions) is related to STDP where STDP is a biologically plausible learning mechanism.

Approximation of the Integrate and Fire (IF) model to activation function is shown as follows.

Activation Function f(y) for ReLU is given by (10) where, x_h are the inputs and w_h are the associated weights.

$$f(y) = \max(0, y)\ ,\ y = \sum_{h} x_h w_h \qquad (10)$$

Potential increase due to a set of input spikes (denoted by Dirac delta functions) to an Integrate and Fire neuron during a short time period (t, t-α] is given by (11). For a given input to the network (during a period T), assume the subject neuron's Potential increase reaches the threshold (θ) R times. i.e. Neuron fires R times. Here the number of Output spikes is proportional to U^{Tot} given in (12) where r(t)=1 indicates where an output spike is generated. If the potential increase does not reach the threshold, no output will be generated. This is very similar to ReLU activation function given in (10) where output value for positive values maps to number of spikes (R) in the Integrate and Fire model.

$$U(t) = \sum_h w_h \left(\int_{t-\alpha}^{t} \sum_{t_h^p} \delta(t' - t_h^p) dt' \right) \qquad (11)$$

$$\mathrm{U}^{\mathrm{Tot}} = \sum_{t^f \in \{r(t)=1\}} U(t^f) \qquad (12)$$

By Applying gradient decent for the networks of linear neurons, the Widrow-Hoff rule is derived. Hence, weight change function can be derived to be consist of STDP and Anti-STDP processes as mentioned in the previous model 'ReSuMe'.

Then the BP-STDP model deviates from ReSuMe model in the way of weight update using STDP learning rules. Short time periods are selected such that at most 1 actual output spike / desired output spike is present. For target neurons (i.e. Neurons from which the output spikes are desired), STDP learning rule is applied to increase the weight whereas anti-STDP rule is applied to non-target neurons to reduce the weight between connections as given in Eq. (13) and Eq. (14) where input last term in Eq. (13) represents the input spikes during time period (t - ε, t) and the learning rate is μ.

$$\Delta w_{ih}(t) = \mu . \xi_i(t) \sum_{t'=t-\varepsilon}^{t} s_h(t') \qquad (13)$$

$$\xi_i(t) = \begin{cases} 1, \ where\ z_i(t) = 1, r_i \neq 1 \\ -1, \ where\ z_i(t) = 0, r_i = 1 \\ 0\ otherwise \end{cases} \qquad (14)$$

To come up with a weight update rule for synapses in hidden layers, initially, the chain rule is applied to the network of ReLU neurons and then in the resulting equation for weight update is modified for the SNN (Integrate and Fire model) by replacing constant static values from integrals of presynaptic spikes with time. Then, the resulting equation is further simplified, by selecting short time periods where at most a single desired / actual postsynaptic spike is expected/generated. At, this stage local rule to update weights between neurons in hidden layers (j and h) is derived as shown in Eq. (15).

$$\Delta w_{hj}(t) = \begin{cases} \mu . \sum_i \xi_i(t) w_{ih}(t) . \sum_{t'=t-\varepsilon}^{t} s_j(t') \text{ , } where\ s_h = 1\ in\ [t-\varepsilon, t] \\ 0\ otherwise \end{cases} \tag{15}$$

Model has achieved high accuracy for MNIST handwritten dataset classification (97.2%).

Though the model has shown the relationship to Property of Biological plausibility using STDP processes, there have been few unrealistic assumptions made. Initially, an assumption to map IF neurons in SNN to ReLU in ANN and assumption is based on the number of spikes generated for an input (much similar to Rate Coding). Then, the Equation for weight change between hidden layer neurons in ANN using ReLU neuron model (based on derivatives) is overridden by applying integrals with time and a simple equation is obtained. This equation is unlikely to be obtained directly using formulas for Spiking Neurons. Hence the provided equations are questionable.

2.2.4 Using back propagation for learning in SNN using temporal coding

In [12] Hesham Mostafa, has come up with a model for SNN using temporal coding and derived differential equations enabling back propagation for learning. However, this approach deviates from the property of biological plausibility as information required for weight adjustment is not locally available.

In this mode, below assumptions are made.

1. Non - leaky integrate and Fire neuron model with exponentially decreasing input current when an input spike is fed to a neuron.
2. A neuron is allowed to fire only once. (this is in line with the assumption made in Model 'Spike Prop', early spikes are more important)

The model has achieved accuracies of 97.55% (2 layer) and 97.14% (3 layer) for the classification of MNIST handwritten dataset. Initially, the input image is converted to black and white from greyscale. Therefore only pixel values 0 and 255 are used for the computation which results in a clear separation in pixel values as well as in z domain. This enhances the accuracy of the classification task.

Since the model is based on, Non – Leaky Integrate and Fire model with exponentially decreasing input current, Membrane potential dynamics can be simply expressed by Eq. (16) where *θ(x)* is the Heaviside step function and t_{ir} denotes the spike time of r^{th} input spike from presynaptic neuron *i*.

$$\frac{dV_j(t)}{dt} = \sum_i W_{ij} \sum_r \Theta(t - t_{ir}).e^{-\frac{(t-t_{ir})}{\tau_{syn}}} \qquad (16)$$

Assuming at most one spike is fed by input neurons to the output neuron (Assumption 2) and Membrane potential increase due to input spikes can be derived by integrating the Eq. (16) w.r.t. time and given by Eq. (17).

$$V_j(t) = \sum_i \Theta(t - t_i).W_{ij}\left[1 - e^{-\frac{(t-t_i)}{\tau_{syn}}}\right] \qquad (17)$$

By setting Membrane potential just before generating an output spike to 1 and denoting the set of input neurons (causal set) which fire before the output neuron by C, Output spike time can be derived as shown in Eq. (18). Further, the time constant (τ_{syn}) is set to 1 to further simplify the equation.

$$e^{(t_{out})} = \frac{\sum_{i \in C} W_i \, e^{t_i}}{\sum_{i \in C} W_i - 1} \qquad (18)$$

Equation is transformed to z-domain by transforming e^t to ‘z’ and Eq. (19) is derived.

$$z_{out} = \frac{\sum_{i \in C} W_i \, z_i}{\sum_{i \in C} W_i - 1} \qquad (19)$$

Hence as per Eq. (19) it can be seen that to fire a neuron, the summation of synaptic weights should be greater than zero. And, another important property of this model, is that input spikes to a neuron from neurons other than the ones in the causal set are insignificant. (After the output neuron is fired, it is not fired for the input spikes received from any input neuron). However, during learning, change in input spike time of a particular presynaptic neuron may result in a drastic change in a causal set of a postsynaptic neuron as shown in Figure 2.4 where the change in input spike time of 3rd neuron has resulted in the removal of neuron 4 from the causal set.

Also, during learning, it has been observed that summation of weights around value 1, may result in drastic changes in output as per Eq. (19), a small change in numerator can change the output value drastically when the numerator value is closer to 0.

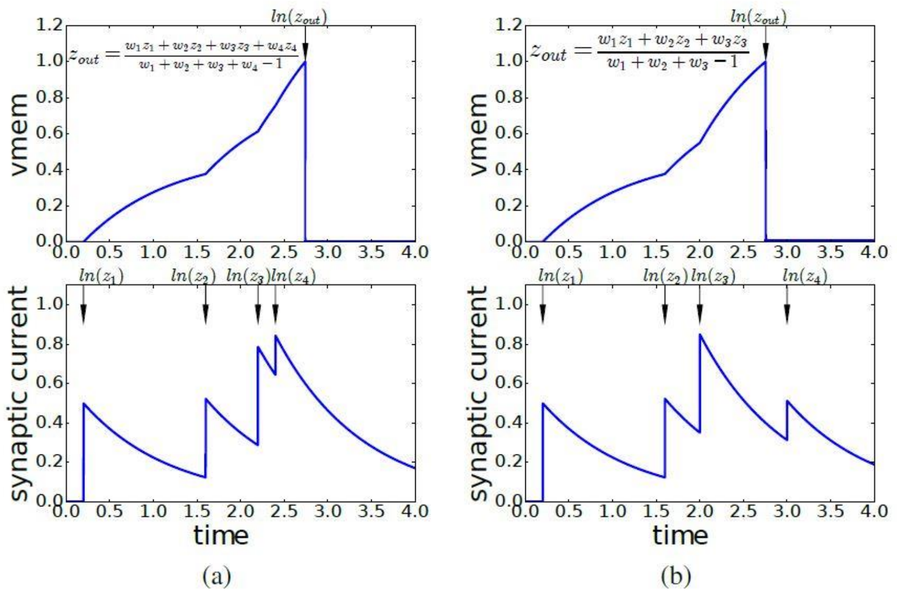


Figure 2.4: Membrane potential and summation of input synaptic current of a neuron changes as per the input spike times

The linear relationship between output spike time with input spikes and synaptic weights enables deriving simple differential equations for output spike time with respect to synaptic weight and input spike time of input neuron '*p*' as shown in Eq. (20) and Eq. (21) which enables learning using back propagation in a multilayer network.

$$\frac{dz_{out}}{dw_p} = \begin{cases} \dfrac{z_p - z_{out}}{\sum\limits_{i \in C} w_i - 1} & if \;\; p \in C \\ 0 \;\; otherwise \end{cases} \tag{20}$$

$$\frac{dz_{out}}{dz_p} = \begin{cases} \dfrac{w_p}{\sum\limits_{i \in C} w_i - 1} & if \;\; p \in C \\ 0 \;\; otherwise \end{cases} \tag{21}$$

2.3 Comparison of SNN Models

Comparison of Candidate Models discussed in the above section in terms of accuracy and the suitability in using them to implement an Auto Encoder with ability learn using back propagation or similar mechanisms are provided in Table 2.1. Different models have been tested on different data sets, hence evaluating those by comparing the accuracy figures may not be ideal.

Model / Research Paper	**Classification Accuracy**	**Supports learning using error back propagation**	**Comments**
Train Equivalent ANNs [3]	MNIST – 98% - 99.42%	YES	
SpikeProp [13]	IRIS - 96.1% Wisconsin breast cancer dataset – 97%	YES	
ReSuMe [11]		YES	Only one layer is trained, but model can be extended for multiple layer network
BP-STDP [7]	IRIS – 96% MNIST - 97.2 % (3 layer)	YES	
Convolutional Spiking Neural Network with STDP learning [14]	MNIST - 98.4 %	NO	
Simplified SNN with SRM and STDP learning algorithm [4]	-	NO	Only two layers have been used.
Mirrored STDP [15]	-	NO	Only two layers have been used.
Mostafa (2017) Supervised learning based on Temporal coding in SNN [12]	MNIST – 97.14% (3 layer) 97.55% (2 layer)	YES	

Table 2.1: Comparison of SNN Models

3 CAHPTER 3

METHODOLOGY AND IMPLEMENTATION

3.1 Using a SNN Model for Auto Encoder

As discussed in the previous chapter, Auto Encoders using SNNs with high accuracy of reconstruction have not been proposed and developed. Out of the several supervised learning models for SNN based on temporal coding discussed in the previous chapter, some of the models can be used to develop an Auto Encoder.

The selected model should be able to train the SNN by adjusting the weights for each input (or for a set of inputs in each mini-batch) so that the reconstruction loss is minimized. Suitability of models to adjusted weights in this manner is first considered. Thereafter, the accuracy of the model and the simplicity in implementation and computation efficiency are considered as the parameter to evaluate models.

BP-STDP [10] and Model introduced by Mostafa [11] are the best candidates for the model for Auto Encoder as back propagation is supported by both models. Accuracy in both these models is high and approximately the same. And the complexity in implementing the Auto Encoder using any of these two models will be the same. However, as mentioned in the literature review, equations derived for weight change are questionable in BP-STDP model. In Mostafa's model, a neuron is fired only once for an input and encoding input value to the time of the first spike can be done directly. But in BP-STDP, encoding the input may need to be done as per Rate Coding (spike train should be generated at similar time intervals for an input and frequency of the spikes is based on the input value) and computations and weight updates will involve for each of these spikes. Hence, computational complexity/power consumption of BP-STDP will be higher compared to Mostafa's model.

Compared to other models, this model has the advantage of very rapid output generation (for classification task) and very low power consumption if it is implemented on Neuromorphic hardware as few numbers of neurons will be fired before the classification is completed for a given input. The model introduced by Mostafa [12] is selected for the Auto Encoder.

3.2 Overview

The flow of the Auto Encoder is shown in Figure 3.1. Once the image data set is loaded, images will be resized as required. Any preprocessing steps can be applied here. Preprocessed image is converted to input spike times and corresponding z-domain values are derived as described in section3.2. Input spike times in z-domain are fed to the Auto Encoder for training. During the training, reconstruction error is calculated in z-domain and synaptic weights are adjusted accordingly as described in section 3.5.

After the Auto Encoder is trained, when the training data set is fed to the Auto Encoder, similarly, input values are converted to spike times z-domain and output spike times in z-domain with reconstruction error are provided. Finally, output spike times in z-domain are converted back to pixel values and reconstructed images are displayed.

Before starting the training, Auto Encoder’s network structure (i.e. Number of hidden layers and number of neurons in each layer) should be defined and fed to the Auto Encoder model. Based on the network structure, synaptic weights are initialized in a random manner.

Auto Encoder Model was implemented such that the model supports mini batch wise training. Also, the number of times the model to traverse through all training data (number of epochs) during the trained can be configured. Several other hyper parameters are involved with the model and they are shown in the Table 3.1.

<u>Hyper parameter</u>	**<u>Usage</u>**
Learning Rate	Weight adjustment factor (Similar to conventional Neural Networks)
L2 Regularization Factor	L2 Regularization factor to prevent controlling the spike generation time of the target neuron by one source neuron alone. This done normalizing weights by applying L2 Regularization so that one input synaptic weight becoming too large compared to all other input synaptic weights of the target neuron.
Frobenius norm	Regularization of weight modification values. If weight modification value matrix between two layers exceeds a predefined threshold, matrix values are normalized so that drastic changes are not applied to the network.
Small weight Adjustment factor	If the summation of the synaptic weights of a target neuron is less than 1, the neuron will not fire for any input pattern. To eliminate such redundant neurons, all weights of a target neuron are adjusted according to this hyperparameter.
Initial Sum of weights	Initial synaptic weighs should be random. But the summation of input weights of all neurons should exceed 1. Hence, this parameter is used when generating the initial weight matrix.
Max Spike time	Maximum allowed time for the neuron to fire. Typically set to a large value compared to z-domain spike generation time for black pixel (the allowed lowest input value).

Table 3.1 : Hyper Parameters of the Auto Encoder Model

Once the SNN Auto Encoder is trained, it is possible to extract the synaptic weights of the whole network and create the Encoder and Decoder networks separately by applying the relevant synaptic weights accordingly.

Outputs of the final layer neurons (i.e. spike generation times in z-domain) of the Encoder network provide the compressed representation for the provided input image. When the compressed representation (i.e. Spike generation times in z-domain) are fed to the Decoder network, spike generation times of final layer neurons provide corresponding values for the reconstruction.

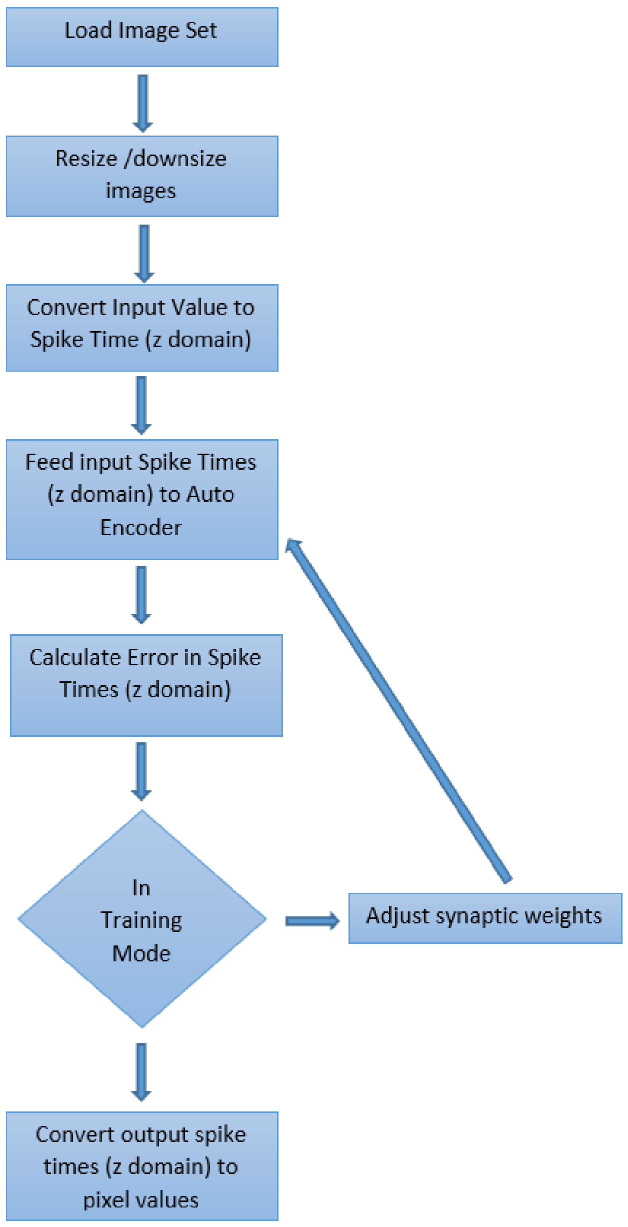

Load Image Set
Resize /downsize images
Convert Input Value to Spike Time (z domain)
Feed input Spike Times (z domain) to Auto Encoder
Calculate Error in Spike Times (z domain)
In Training Mode
Adjust synaptic weights
Convert output spike times (z domain) to pixel values

Figure 3.1 : Task wise flow of Auto Encoder model

3.3 Convert input/output values to spike times

As the first step, input values should be encoded to spike times in a manner that supports the regeneration of output values from spike times. As primarily the research is developed focusing on the image data, a suitable conversion is required.

Some of the models (which were developed for classification task) coverts greyscale pixel value to Black (0-pixel value) and White (255-pixel value) and the input spikes are generated based on Black and white pixel value. This conversion may have caused in the high classification accuracy. However, such conversions are not possible for an Auto Encoder.

For image data, where pixel values are in the range of 0 to 255, the spike time range should be defined. Hence, the corresponding spike time range is set to 3 to 0 (Higher the value, lower the time to generate the spike). In the Auto Encoder, all the calculations are done on the 'z' dimension where conversion from input spike time to corresponding 'z' domain value is derived by taking the exponential value of the input spike time. This conversion is illustrated in Table 3.1.

Input Value (Pixel Value)	Spike Time	Z domain value ($e^{(\text{Spike Time})}$)
0	3	20.0855
255	0	1

Table 3.2 Input value to spike time conversion

Similarly, the Auto Encoder's output is converted back to the pixel value applying the aforementioned conversion method.

3.4 Structure of the network

Implementation of Auto Encoder model is based on a fully connected feed-forward network and it enables the user to configure the number of hidden layers and the number of neurons in each layer. User can add the bottleneck to the network by setting a small number of neurons in the hidden layers such that linear relationship cannot be established in the network during learning.

3.5 Error function for Auto Encoder

All candidate models for SNNs discussed in the literature review, were developed for classification tasks where the output layer consist of few neurons and error functions used in those models are inappropriate for an Auto Encoder. Error function for Auto Encoder is the summation of mean squared error for each output neuron given by Eq. (22).

$$E = \frac{1}{2}\sum_{i}(d_{i,actual} - d_{i,\mathrm{exp}ected})^2 \qquad (22)$$

The above error function will be differentiated with respect to synaptic weights and synaptic weights will be adjusted accordingly as shown in Eq. (23). Derivative functions that were mentioned in the literature survey will be used to construct the algorithm for learning using back propagation.

$$w_{k+1} = w_k + \alpha\ \Delta w_k \quad where\ \Delta w_k = -\left.\frac{dE}{dW}\right|_k \qquad (23)$$

A suitable adjustment factor is desirable to be used when comparing the expected and actual output spike times to eliminate or minimize the effect of propagation delay in the actual output spike time output. However, there is no direct forward method to calculate the propagation error during the training phase of the Auto Encoder. i.e. Propagation delay is incorporated in training error. Hence, when computing the error to adjust synaptic weights using Eq. (22) and Eq. (23), the propagation delay is not considered.

However, once the Auto Encoder is fully trained, propagation error can be determined by feeding an all-white image to the network. The minimum of the output spike generation time (in z-domain) from all output neurons can be considered as the propagation delay.

3.6 Python Implementation

Using the Mostafas model [1] for spiking neural networks, Auto Encoder was developed using python from scratch. Below python Libraries were used.

- Numpy
- Scipy
- Random
- Matplotlib
- imageio
- inspect

Auto Encoder was developed function-wise where each function carries a specific task allocated. Initially, to decide whether a neuron generates an output spike when the input spike generation times from source neurons and the corresponding synaptic weights are given, function ‘GetCausalSetWithOuputSpikeTime’ is implemented based on mathematical Eq. (19). If the output neuron generates an output spike, the time of the spike generation and the causal set is returned by the function. The causal set is the set of input neurons which causes the output neurons to generate the output spike.

To calculate spike generation times of all neurons in the network, function ‘GetCausalSetWithOuputSpikeTime’ is called for each neuron in each layer in the sequence of order of layers of the network. (i.e. Start with first hidden layer, and then continue to next immediate layer). This is done by the function ‘GetOutputVectorSpikeTimes’. Besides calculating the output spike generation times, derivative values of output spike generation time with respect to synaptic weight and input spike time using Eq. (20) and Eq. (21) for each neuron in the network is also calculated within this function to minimize the computation time as all the required parameters are available, instead of implementing a new function to traverse through the whole network once again to calculate derivatives.

To calculate the synaptic weight adjustment values using Eq. (23), it is required to calculate the derivative values of final layer neurons’ output spike generation time with respect to each synaptic weight in the network. This is behavior is implemented in the function ‘CalculateFinalderivativesWRTWeights’.

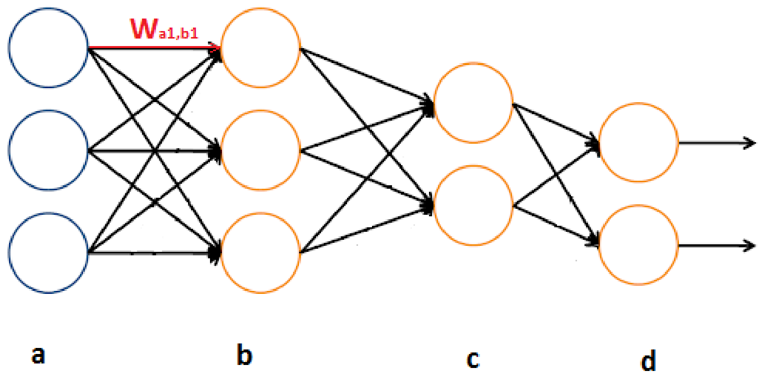


Figure 3.2: Calculating derivative of output spike time w.r.t. synaptic weight

To adjust synaptic weights as per the reconstruction error, backpropagation is used by calculating the derivative of Output Spike time w.r.t. a selected synaptic weight variable. By applying the Multivariable chain rule (Eq. (24)) to the output spike generation time function given in Eq. (19), it is possible to derive equations for desired derivative functions of Output Spike time with respect to a given synaptic weight variable as illustrated using Eq.(25) and Eq.(26) for the sample scenario represented in Figure 3.2.

$$\mathrm{dz/dt} = \partial \mathrm{z}/\partial \mathrm{x.dx/dt} + \partial \mathrm{z}/\partial \mathrm{y.dy/dt} \qquad (24)$$

For the scenario represented in Figure 3.2, the output spike time of a neuron in layer "d" can be written as a multivariable function of spike times of neurons in immediate previous layer (i.e. Layer "c") using equation (19). Hence, Eq. (25) is derived by applying Multivariable chain rule (Eq. (24)). Similarly, Eq. (26) is derived by applying the same concept to 1st neuron in layer "c". Likewise, applying the concept for each neuron in the layer, the final equation can be derived.

$$\frac{dz_{d1}}{dw_{a1,b1}} = \frac{dz_{d1}}{dz_{c1}} \cdot \frac{dz_{c1}}{dw_{a1,b1}} + \frac{dz_{d1}}{dz_{c2}} \cdot \frac{dz_{c2}}{dw_{a1,b1}} \qquad (24)$$

$$\frac{dz_{c1}}{dw_{a1,b1}} = \frac{dz_{c1}}{dz_{b1}} \cdot \frac{dz_{b1}}{dw_{a1,b1}} + \frac{dz_{c1}}{dz_{b2}} \cdot \frac{dz_{b2}}{dw_{a1,b1}} + \frac{dz_{c1}}{dz_{b3}} \cdot \frac{dz_{b3}}{dw_{a1,b1}} \qquad (25)$$

As described above, for a multilayer network, the final derivative equation consists, primarily, multiple number of derivatives values corresponding to output spike time w.r.t. input spike times (of neurons in succeeding layers) and one derivative value corresponding to output spike time w.r.t given synaptic weight. Therefore, for each neuron in the network, derivative corresponding to final layer neuron's output spike time w.r.t. spike time of the subject neuron is calculated. This functionality is implemented in python function 'CalculateFinalOutputDerivativeWRTInputSpikeTimes'. This function is called from the function 'CalculateFinalderivativesWRTWeights' to calculate final derivative values.

Once the final derivative values are derived, to calculate the weight adjustment values for each synaptic weight, function ‘GetWeightAdjustmentArrayMap’ is used. Also as described in the Mostafs’s model, the Frobenius norm threshold is used to avoid large weight adjustments. When the summation of synaptic weights of a target neuron is closer to 1, weight adjustment values can be very large which causes spikes in training error where such non-smooth learning is discouraged. Hence weight adjustment values are normalized using the Frobenius norm threshold. Also, if the summation of the synaptic weights of a target neuron is less than 1, then a penalty value is adjusted to the error function in the Mostafa’s model. This feature is used in the SNN Auto Encoder model implementation also.

Also, the function ‘RegularizeWeight’ was implemented to enforce the L2 regularization of synaptic weight array of a target neuron to avoid the scenario where one source neuron purely determining the output spike generation time of a target neuron.

Additionally, functions were implemented to generate the initial synaptic weight matrix when the network structure is given, convert image to spike time matrix (and vice versa).

Example code snippet of training the network is provided below. Initially, all configuration values of the model are set. Then the network structure is defined and the initial synaptic weight matrix is generated (or if the network is already trained to a certain extent, weight matrix snapshot can be loaded from disk). Then the input dataset is loaded (in this context, MNIST handwritten digit data set) and the selected image set is downsized. Then, iteratively trained the Auto Encoder. (Without explicitly calling train data function, by setting a desired epoch value to the configuration ‘NUMBER_OF_ITERATIONS_OVER_TRAINNIG_DATA’, user can allow the network to train over and over on the same data set).

4 CAHPTER 4

EXPERIMENTAL ANALYSIS & MODEL EVALUATION

4.1 Input Dataset

MNIST handwritten digit data set is used to evaluate the performance of the SNN Auto Encoder. MNIST handwritten digit data set consists a total of 70,000 elements where 60,000 of them belong to the training data set and rest 10,000 elements are unlabeled. The label of the data set is irrelevant in this context as Auto Encoders do not fall under supervised training tasks.

Each image represents a digit and the size of the image is 28x28. As the data flow graph technique is not used for the computations in the SNN Auto Encoder, the complexity, time consumption (the number of clock cycles) for the computations and memory consumption during the computations increases exponentially when the number of input neurons increases.

For example, when the original image (28x28) is fed to the Auto Encoder with 1 Hidden layer consisting of 196 neurons, dimensions of arrays that are used for computations in SNN Auto Encoder will be 784x196. Hence the number of parameters to be trained will be multiples of 784x196 (i.e. 153,664). However, if the original image downsized to 12x12 image, and Auto Encoder is used with 1 hidden layer with the same compression factor (i.e. 4), the number of neurons in the hidden layer will be 36. Corresponding array dimension will be 144x36 and number of parameters to be trained will be relatively very low (i.e. 5184 – nearly 30 times lesser)

Hence, instead of using the original image which is 28x28 (i.e. 784 neurons in the input and the output layers), each input image is downsized to a 12x12 image which is adequate to represent the image. So the number of input/output neurons of the Auto Encoder is set to 144.

4.2 Performance Comparison with ANN AutoEncoder developed using Keras

To demonstrate that the SNN Auto Encoder reconstructs the image comparatively to a conventional Auto Encoder which has the same network structure, experiments were carried out by setting similar values to common parameters as follows.

- Mini batch size = 5
- Number of epochs = 100
- Dataset size = First 100 images in MNIST handwritten digit data set (downsized to 12x12)

A comparison between reconstructions by 3 layer SNN Auto Encoder and an equivalent conventional Auto Encoder using Keras for a network structure 144x24x144 is shown in Table 4.1. A similar comparisons are provided as follows. 5 layer Auto Encoder with network structure 144x48x24x48x144 shown in Table 4.2, 6 layer Auto Encoder with network structure 144x48x24x24x48x144 shown Table 4.3 in and 7 layer Auto Encoder with network structure 144x48x24x16x24x48x144 shown in Table 4.4.

However, it should be noted that the comparative reconstructions by conventional Auto Encoder (using Keras) are achieved using the optimizer ‘adadelta’. Reconstruction error of conventional Auto Encoder with ‘SGD’ optimizer is significantly higher. The ‘SGD’ optimizer with a batch size (> 1) is equivalent to the training mechanism used in SNN Auto Encoder whereas ‘adadelta’ is an improved optimization mechanism used by Keras where the convergence rate is high (Accuracy can be less than the accuracy of ‘SGD’) [16], [17]. Conventional Auto Encoder with ‘SGD’ optimizer requires 10,000 epochs to generate an output with a decent reconstruction error. Therefore it can be concluded that the SNN Auto Encoder outperforms equivalent conventional Auto Encoder in this case by the convergence rate.

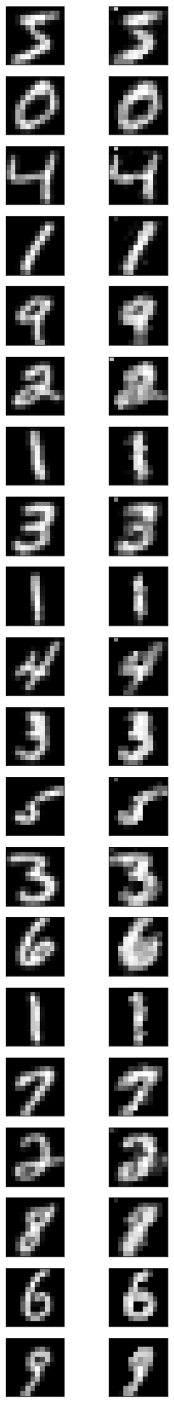
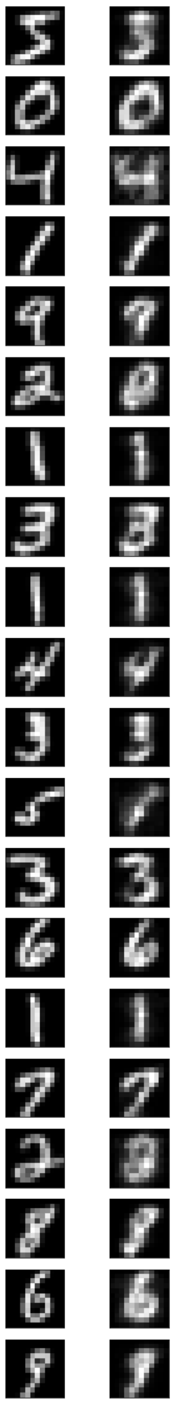
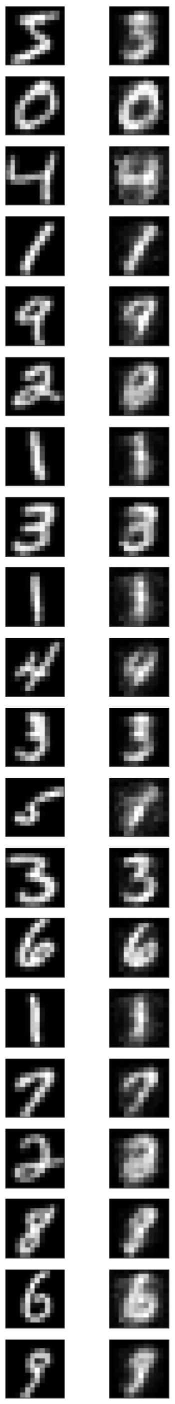
SNN Auto Encoder	Keras-‘adadelta’-100 epochs	Keras-‘SGD’-10,000 epochs
		

Table 4.1: [144 x 24 x 144] Reconstruction comparison

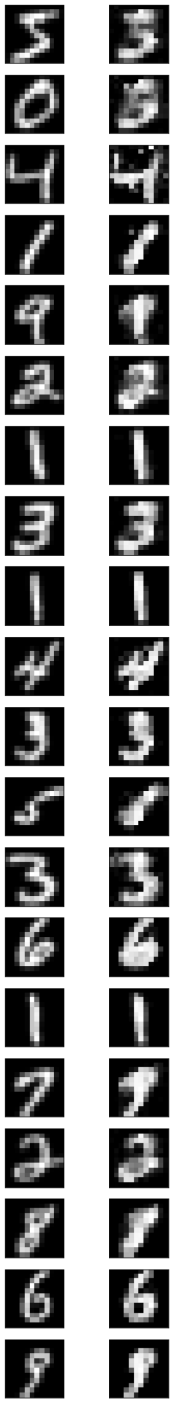
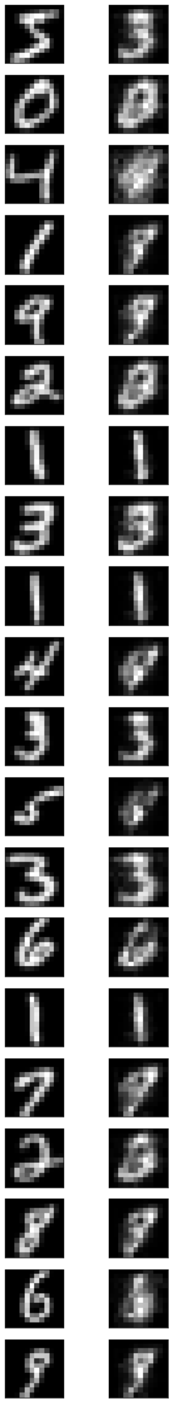
SNN Auto Encoder reconstruction	**Conventional Auto Encoder using KERAs reconstruction**
	

Table 4.2 : [144 x 48 x 24 x 48 x144] Reconstruction comparison

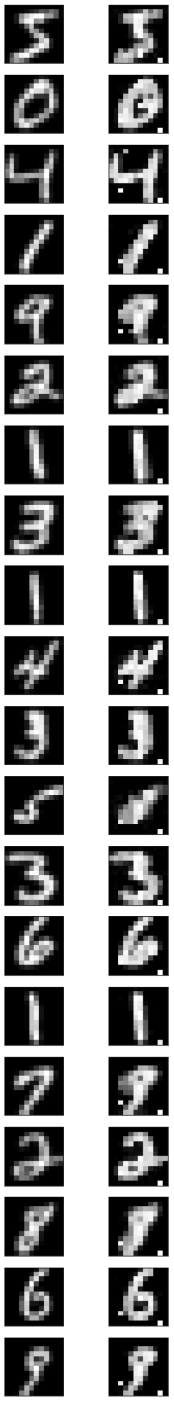
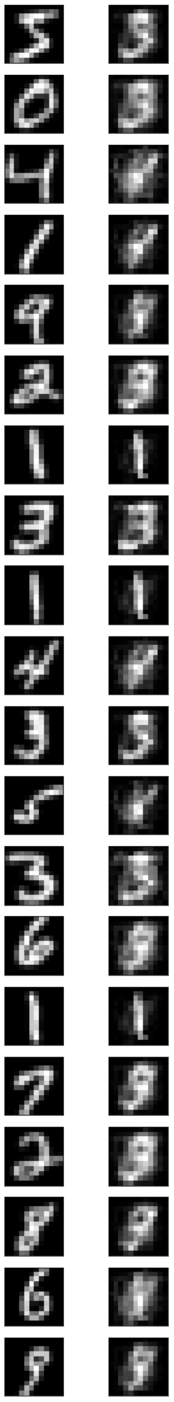
SNN Auto Encoder reconstruction	**Conventional Auto Encoder using KERAs reconstruction**
	

Table 4.3: [144 x 48 x 24 x 24 x 48 x144] Reconstruction comparison

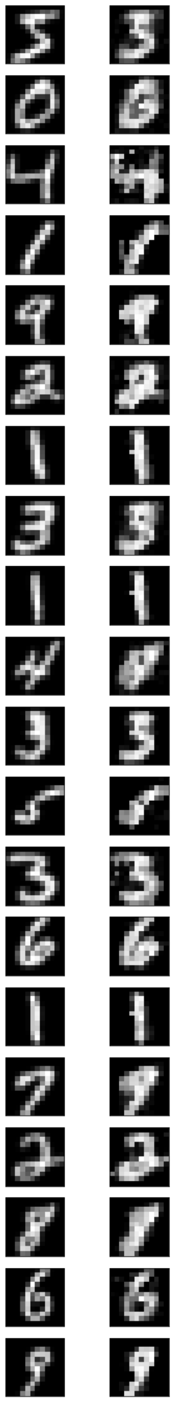
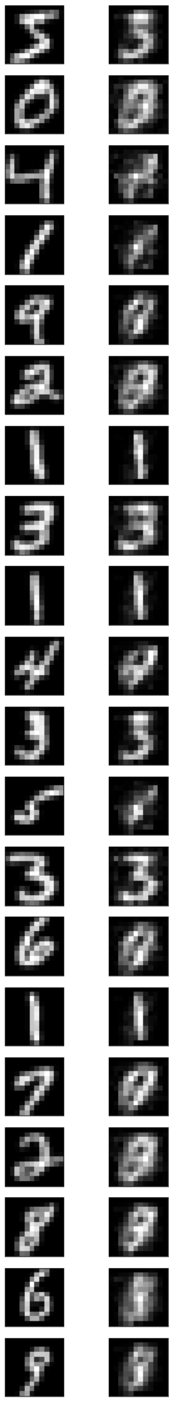
<u>**SNN Auto Encoder reconstruction**</u>	<u>**Conventional Auto Encoder using KERAs reconstruction**</u>
	

Table 4.4 : [144 x 48 x 24 x 16 x 24 x 48 x144] Reconstruction comparison

Even though it is not possible to directly compare the error between SNN Auto Encoder and equivalent conventional Auto Encoder as the outputs of SNN Auto Encoder are in z-domain of output spike times, once the SNN Auto Encoder's output is converted to pixel values, reconstruction errors can be compared. Comparisons of reconstruction errors in pixel values between SNN Auto Encoder and equivalent conventional Auto Encoder are shown in Table 4.5 and Table 4.6. For the reconstruction error comparison, 2 metrics were used.

- Mean Squared Error (MSE) in pixel values (where pixel value range is 0-255)
- Peak Signal-To-Noise Ratio (PSNR) – Higher PSNR values means lesser the reconstruction error.

  i.e. $PSNR\ value = 20.log[\frac{255}{\sqrt{\mathrm{MSE}}}]$

Provided MSE value is the average MSE across all images used for the testing. Minimum, maximum and average PSNR values provided (Minimum PSNR value is for the most distorted image). From the MSE and PSNR values of the reconstructions, it can be concluded that SNN Auto Encoder outperforms equivalent standard Auto Encoder.

Network structure	**SNN AutoEncoder**				**Standard AutoEncoder (‘AdaDelta’)**			
	Mean Squared Error	**PSNR**			**Mean Squared Error**	**PSNR**		
		Min	**Max**	**Avg.**		**Min**	**Max**	**Avg.**
3 layers 144 x 24 x 144 (100 epochs)	353.92	19.94	26.96	23.03	2560.46	11.36	17.05	14.21
5 layers 144 x 48 x 24 x 48x 144 (150 epochs)	1097.15	13.90	24.76	18.49	2350.32	12.20	18.08	14.63
6 layers 144 x 48 x 24 x 24x 48x 144 (350 epochs)	1086.88	15.12	22.44	18.21	1246.21	14.21	24.90	17.84
7 layers 144 x 48 x 24 x 16 x 24x 48x 144 (350 epochs)	1170.20	14.61	22.21	17.99	1494.07	13.11	24.66	17.05

Table 4.5 : Reconstruction Error comparison of SNN AutoEncoder vs. Standard AutoEncoder (Optimizer - 'AdaDelata')

Network structure	SNN AutoEncoder				Standard AutoEncoder ('sgd')			
	Mean Squared Error	PSNR			Mean Squared Error	PSNR		
		Min	Max	Avg		Min	Max	Avg
3 layers 144 x 24 x 144 (100 epochs)	353.92	19.94	26.96	23.03	12051.05	6.70	7.89	7.33
5 layers 144 x 48 x 24 x 48x 144 (150 epochs)	1097.15	13.90	24.76	18.49	11617.34	6.80	7.99	7.49
6 layers 144 x 48 x 24 x 24x 48x 144 (350 epochs)	1086.88	15.12	22.44	18.21	11563.91	6.85	8.07	7.54
7 layers 144 x 48 x 24 x 16 x 24x 48x 144 (350 epochs)	1170.20	14.61	22.21	17.99	11655.50	6.80	8.05	7.48

Table 4.6 : Reconstruction Error comparison of SNN AutoEncoder vs. Standard AutoEncoder (Optimizer -'SGD')

4.3 Performance of Auto Encoder vs network structure

During the experiments carried out, training error is in z-domain value for spike generation time. As the minimum spike generation time and maximum spike generation times are respectively set to 1 (i.e. exp(0)) and 20.0855 (i.e. exp(3)), percentage error can be calculated as follows.

$$training\ error\ as\ a\ percentage = \frac{Average\ training\ error\ in\ spike\ generation}{(e^3 - e^0)}\ \% \qquad (26)$$

4.3.1 Performance vs Number of Hidden layers

To analyze the impact of number of layers of the network on performance in terms of the reconstruction error and the number of epochs to train the network such that the training reconstruction error reaches an acceptable level, experiments were carried out using Auto Encoders with different number of layers. As shown in the Figure 4.1 and Table 4.8, number of iterations to train the network increases drastically with the number of layers of the Auto Encoder. This behavior is in line with training of conventional artificial neural networks with any training mechanism based on backpropagation (i.e. gradient decent, stochastic gradient decent and Mini Batch) due to the increased number of parameters to be learned and other inherent issues in backpropagation such as vanishing gradient problem.

Network structure	**SNN AutoEncoder**			
	Mean Squared Error	**PSNR**		
		Min	**Max**	**Avg**
3 layers 144 x 24 x 144 (100 epochs)	353.92	19.94	26.96	23.03
5 layers 144 x 48 x 24 x 48x 144 (150 epochs)	1097.15	13.90	24.76	18.49
6 layers 144 x 48 x 24 x 24x 48x 144 (350 epochs)	1086.88	15.12	22.44	18.21
7 layers 144 x 48 x 24 x 16 x 24x 48x 144 (350 epochs)	1170.20	14.61	22.21	17.99

Table 4.7 : Reconstruction Error with number of layers

Also from Table 4.7, it can be seen that reconstruction error increases from MSE and PSNR values when the number of layers is increased. SNN Auto Encoders with higher number could be trying to learn general representations (higher order features) and hence the reconstruction error could be higher. However to analyze this behavior, extensive tests will have to be carried to out.

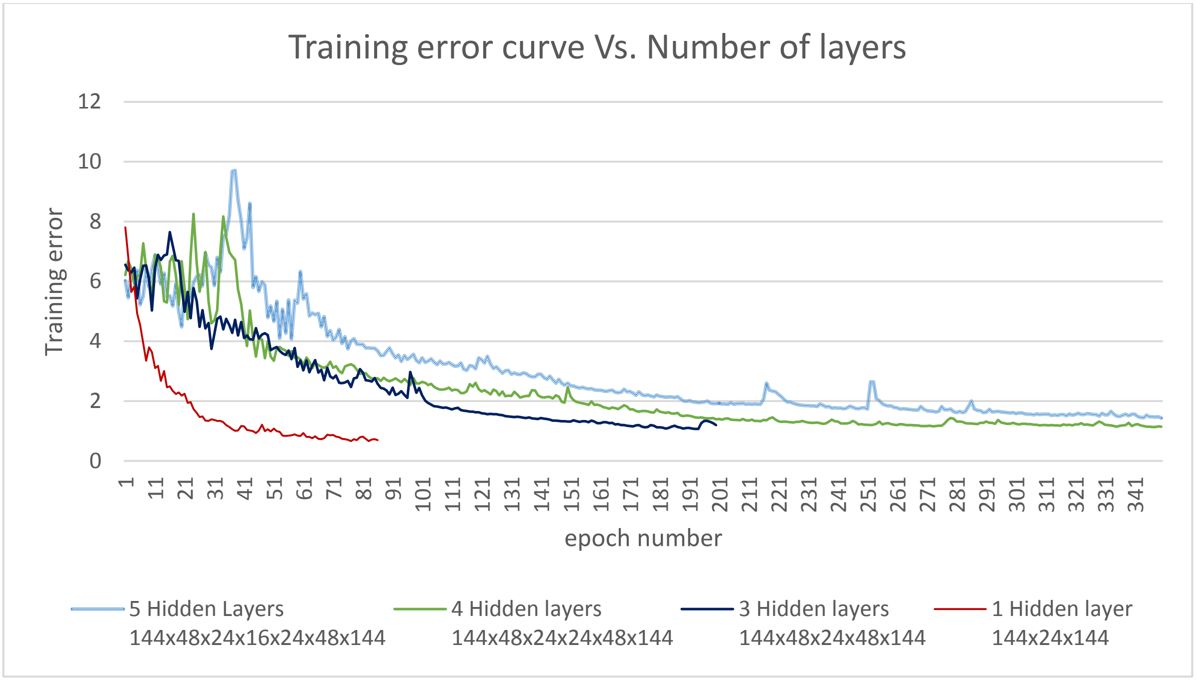


Figure 4.1: Training error curve Vs. Number of layers

Network Structure	**Number of epochs to Achieve acceptable training error**	**Training error at the end of the training phase**
144x24x144	100	0.697021465
144x48x24x48x144	200	1.197019799
144x48x24x24x48x144	350	1.150954263
144x48x24x16x24x48x144	350	1.431811091

Table 4.8 : Network Structure vs. Training Error

4.3.2 Performance vs compression ratio

To analyze the impact of Compression ratio on performance of the SNN Auto Encoder in terms of the number of epochs to train the network such that the training reconstruction error reaches an acceptable level, experiments were carried out by setting the number of neurons in the hidden layer to achieve the desired compression ratio as shown in Table 4.9. Auto Encoder with 1 hidden layer is used for the test. (200 images were used for the test)

From Table 4.9 and Figure 4.2 / Figure 4.3, it can be seen that when the compression ratio increases, the training reconstruction error increases. This is in line with the general fact that in lossy compressions, the reconstruction error increases with the compression ratio.

Compression ratio	Number of Neurons in Hidden Layer	Training error after 50 epochs	
		Mini Batch size = 5	Mini Batch size = 10
9:1	16	1.200582669 (6.29%)	1.566901714 (8.21%)
6:1	24	0.982066042 (5.15%)	1.174818221 (6.15%)
4.5:1	32	0.879138981 (4.60%)	1.139022948 (5.97%)
3.6:1	40	1.05927208 (5.55%)	1.078116341 (5.65%)
3:1	48	0.883807026 (4.63%)	1.575804122 (8.26%)

Table 4.9 : Compression ratio vs Training error

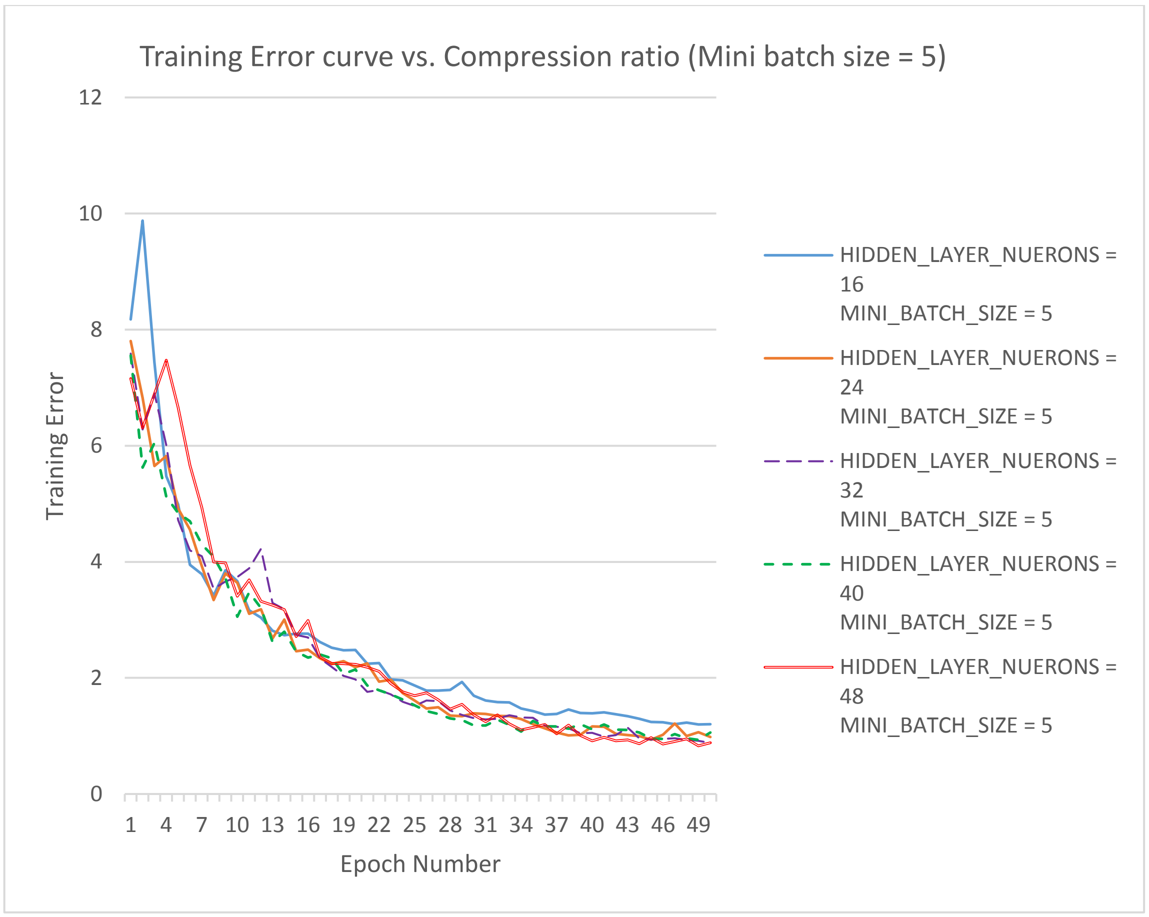


Figure 4.2 : Training Error curve vs. Compression ratio 1

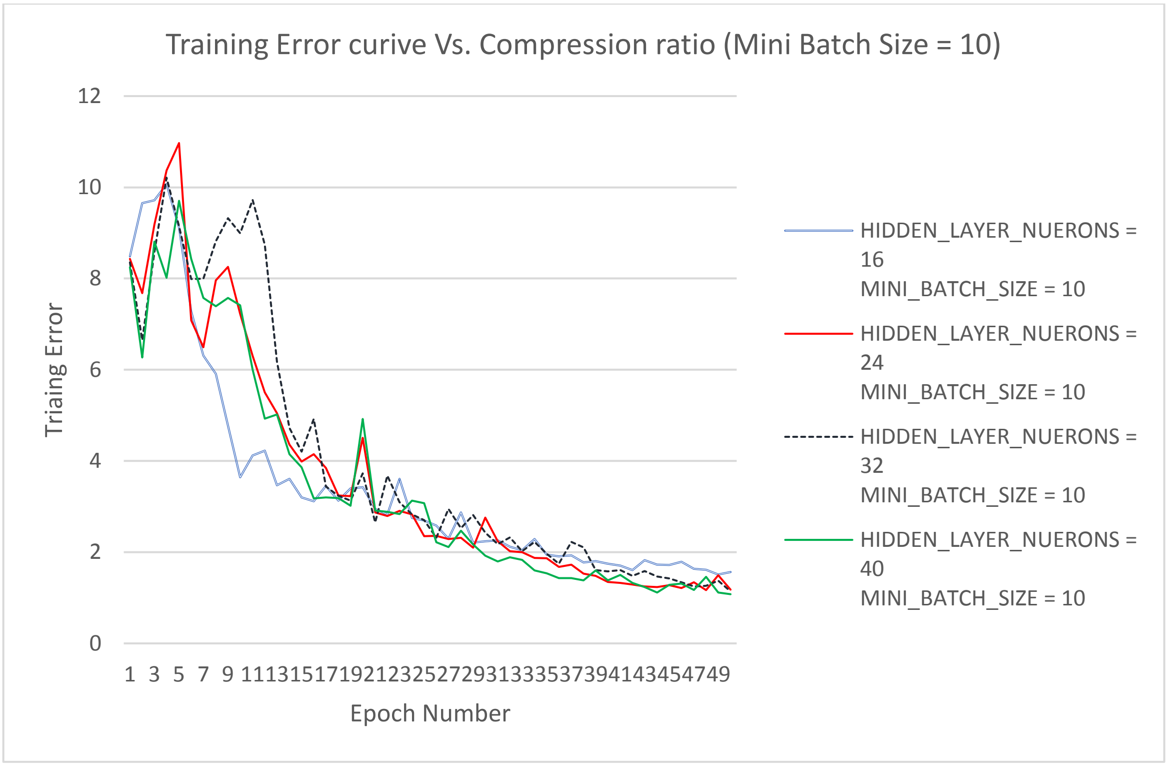


Figure 4.3 : Training Error curve vs. Compression ratio 2

4.4 Hyper parameter analysis

The training performance of Auto Encoder with respect to each hyperparameter is discussed below. All experiments were carried out with the below configurations.

- 3 Layer SNN Auto Encoder (1 hidden layer)
- SNN Auto Encoder with network structure – 144x24x144 (Compression ratio 6:1)
- 200 images were used for the training.

4.4.1 Mini Batch Size

To analyze the impact of Mini Batch Size on the training performance of the SNN Auto Encoder, individual tests were carried out with different Mini Batch Sizes. Training error at each epoch number is shown for Mini Batch sizes 2, 5, and 8 in Figure 4.4. It is observed that training error minimizes quickly for lower Mini Batch sizes (For lower Mini Batch sizes, learning is achieved quickly). However, in all cases, after 100 epochs training error is recovered to acceptable values as shown in Table 4.10.

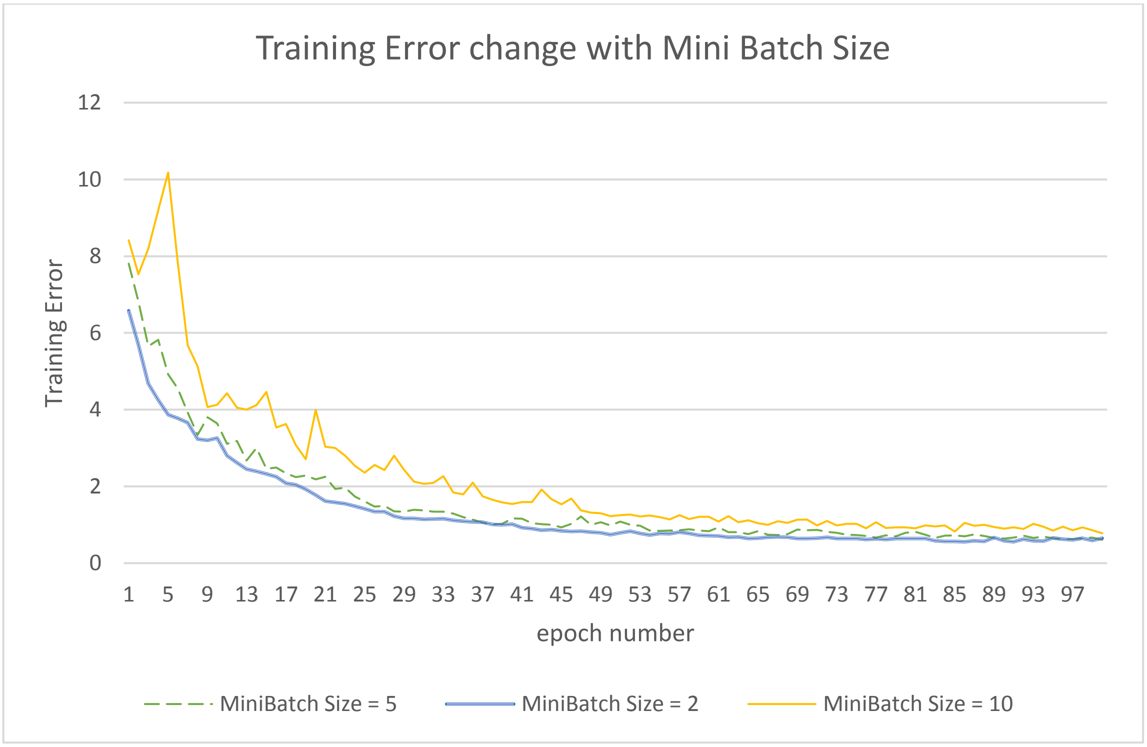


Figure 4.4 : Training Error curve with Mini Batch size

Mini Batch size	**Training Error after 100 epochs**
2	0.609811117 (3.20%)
5	0.648047928 (3.40%)
10	0.777276331 (4.07%)

Table 4.10 Training error after 100 epochs for different mini batch sizes

4.4.2 Learning Rate

To analyze the impact of Mini Batch Size on the training performance of the SNN Auto Encoder, individual tests were carried out with different learning rates. Training error at epoch number is shown in Figure 4.5 for learning rates from 0.0005 to 0.1. It is observed that the training error smoothly diminished for learning rates in range 0.0005 to 0.001. For higher learning rates, peaks in the training error can be observed. Such peaks in the training error can be caused by large adjustments of synaptic weights so that the drastic changes in input spike times and causal sets associated with neurons in the layers in the latter part of the network.

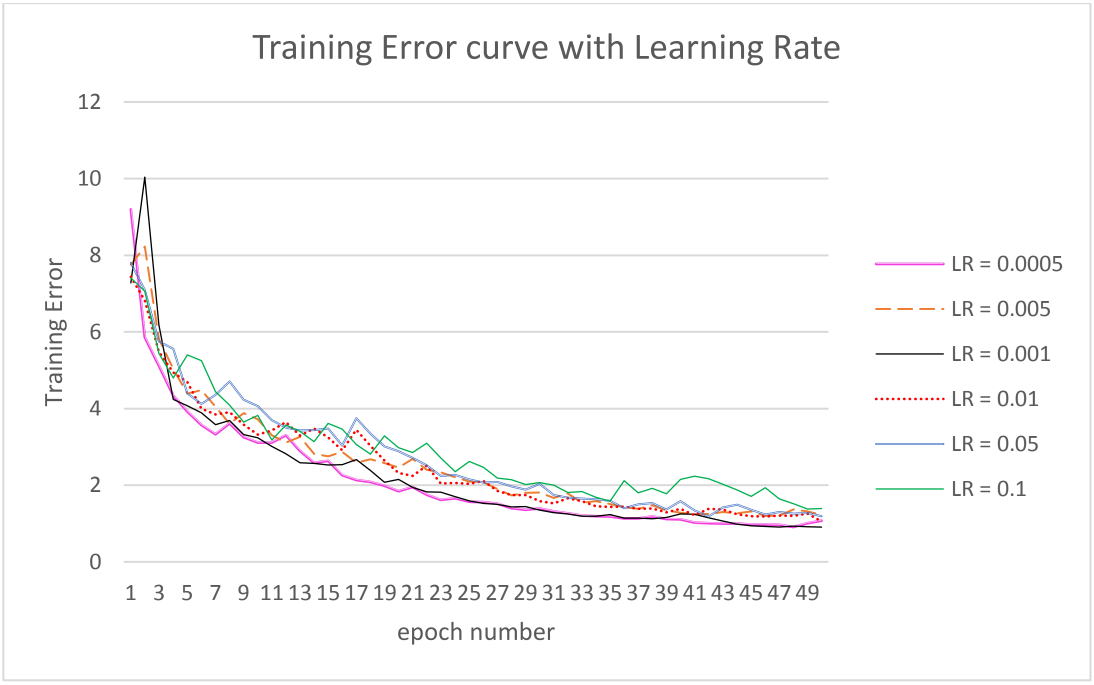


Figure 4.5 : Training Error curve with Learning Rate

4.4.3 Frobenius norm threshold

To analyze the impact of Frobenius norm threshold on the training performance of the SNN Auto Encoder, individual tests were carried out with different Frobenius norm threshold values as shown in Figure 4.6 . To avoid applying large adjustment values to synaptic weights, Frobenius normalization is applied as discussed in section 3.6. (When summation of synaptic weights of a target neuron is closer to 1, weight adjustment values will large). From Figure 4.6, it is observed that when small adjustments to synaptic weights are allowed, the Auto Encoder network's weights are changed drastically which leads to unsuccessful learning.

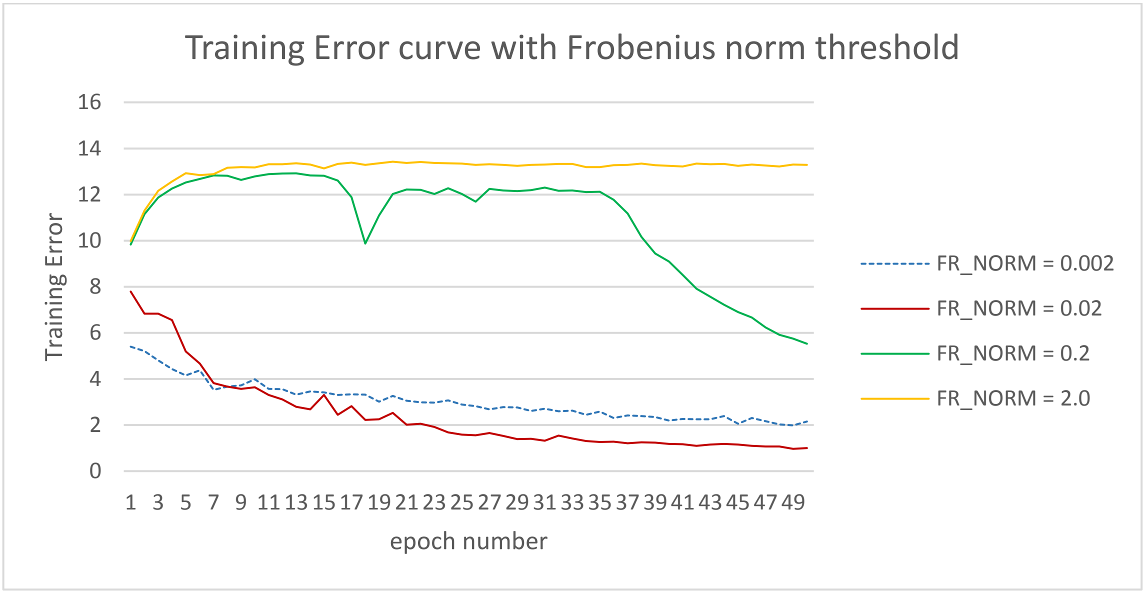


Figure 4.6 : Training Error curve with Frobenius norm threshold

4.5 Hidden Layer Spike Time Analysis

Figure 4.8 shows a heat map of the hidden layer spike generation times of an SNN Auto Encoder with 1 hidden layer consisting of 24 neurons for 10 input images shown in

Figure 4.7. The SNN Auto Encoder was trained for 100 input images. By analyzing the heat map, it can be identified that for this set of input images, some of the neurons in hidden layers do not represent descriptive information or they represent redundant information. (E.g. Neuron index 14 has generated spikes for inputs early for all input images). This supposition can be confirmed by analyzing the average spike generation time for each neuron in the hidden layer across all 100 training input images is shown in Table 4.11 (Average spike generation time for Neuron index 14 is at a minimum level and this neuron generates an early spike for input images, hence the neuron can be considered as redundant).

Also, it can be identified that hidden layer spike generation times (compressed representation) have a higher correlation for similar input images from the heat map shown in Figure 4.8.

Neuron Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Avg. Spike Time	9.3	7.3	18	12	20	12	7.4	12	11	12	7.9	9.9	13	1.8	18	7.8	21	12	15	18	16	21	12	7.7

Table 4.11 : Avg. Spike generation time for hidden layer neurons

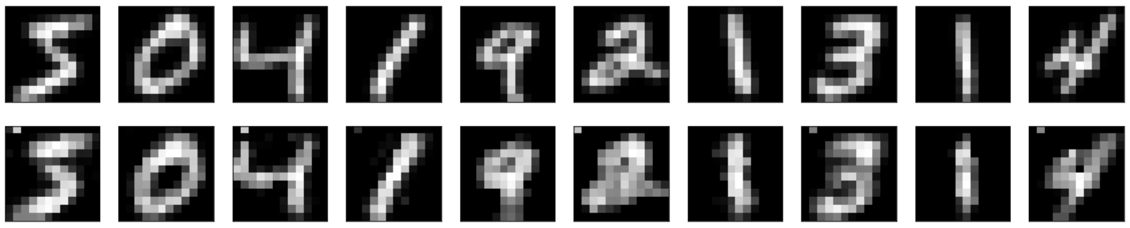


Figure 4.7: Reconstructions for sample 10 images:

Input Image Index	0	1	2	3	4	5	6	7	8	9
Neuron Index										
1	3.041	7.382	8.117	10.86	8.079	2.642	8.562	1.319	17.18	20.26
2	1.802	1.865	21.62	16.56	6.691	20.25	10.29	5.029	10.67	8.564
3	8.472	15.51	18.59	19.94	22.67	3.491	21.19	10.95	21.4	20.37
4	20.38	4.84	2.674	20.64	2.324	4.976	21.79	6.907	22.18	1.973
5	20.28	21.05	17.49	20.34	22	21.85	20.45	21.12	21.18	21.62
6	2.536	3.689	19.69	2.791	9.33	14.63	17.35	6.841	20.3	21.2
7	11.08	2.213	20.4	6.612	3.066	1.867	13.17	1.391	18.27	3.778
8	14.12	1.866	22.5	13.79	24.17	2.162	14.47	5.082	20.93	21.71
9	5.202	17.99	2.447	11.31	20.48	21.46	15.04	6.503	20.22	4.242
10	14.35	16.92	14.42	21.07	5.169	3.534	1.635	21.08	2.608	20.72
11	20.54	14.23	3.457	1.654	2.517	4.216	21.33	5.786	21.22	3.962
12	15.61	4.528	16.2	1.754	13.74	6.776	19.17	10.41	15.29	4.606
13	15.68	2.386	19.74	20.11	7.829	14.07	21.72	13.69	21.03	11.08
14	1.713	1.721	2.954	1.647	1.716	1.811	1.384	1.32	2.445	1.653
15	14.68	9.506	15.87	18.55	21.53	20.85	19.52	20.6	20.66	20.43
16	10.65	16.32	17.37	9.561	2.408	2.695	1.317	9.772	2.199	1.503
17	22.38	2.541	3.32	20.67	23.62	22.4	26.04	21.55	24.86	21.78
18	2.046	6.051	20.49	6.069	23.22	5.307	20.7	15.27	21.41	3.977
19	16.88	22.22	13.02	20.61	2.926	22.45	2.146	18.02	3.285	22.4
20	3.52	17.17	11.08	9.83	23.37	21.21	21.77	5.478	21.56	4.938
21	4.474	20.75	2.777	16.01	23.45	16.57	22.29	2.758	22.93	20.68
22	10.34	10.67	15.93	23.08	24.85	26.89	21.47	21.66	21.08	18.6
23	11.99	10.24	3.558	14.15	20.79	20.79	8.579	7.004	20.38	10.51
24	1.924	6.689	20.93	1.509	23.37	10.97	1.794	1.54	5.333	16.46

Figure 4.8 : Heat map of Hidden layer spike time for different input images

4.6 Hardware information

Tests were executed on Redhat Linux 7.2 operating system. Hardware specifications are shown in Table 4.12. Python 3.6 version is used.

Property	**Value**
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	1
Core(s) per socket:	2
Socket(s):	2
NUMA node(s):	1
CPU family:	6
CPU MHz:	3465.680
BogoMIPS:	6931.36
Virtualization type:	full
L1d cache:	32K
L1i cache:	32K
L2 cache:	4096K
NUMA node0 CPU(s):	0-3

Table 4.12 : Hardware information

5 CAHPTER 5

CONCLUSION

5.1 Contribution

As discussed in the literature survey, there are only a handful of Auto Encoder models developed using Spiking Neural Networks. None of them supports Auto Encoders with multiple layers as per the knowledge of the author. In this research, a model has been introduced for Auto Encoder and implemented which supports multiple hidden layers and the model is based on spiking neural networks using the temporal coding scheme. Experimental evidence shows that the performance (in terms of reconstruction error) of the new SNN Auto Encoder is in par with the performance of conventional Auto Encoder with a similar network structure and in fact in certain experiments, SNN Auto Encoder shows superior performance compared to the equivalent conventional Auto Encoder in terms of convergence rate.

Also, it suggests that the power consumption could be low when implemented on neuro-morphic hardware as the SNN model uses Temporal Coding Scheme and one neuron is allowed fire only once.

5.2 Limitations

The required number of passes through the data set (epochs) to train the network increases with the number of layers of the network. Even though the model enables the user to configure an Auto Encoder with any number of layers, a high number of training epochs will make the user reluctant to go for deep Auto Encoders. This behavior is similar to the training in conventional neural networks that use backpropagation to adjust network parameters where issues such as vanishing gradient come into play [18].

Also, as per the current implementation, the time required for computations increases exponentially with the dimension of input images which urges the user to resize the input image to an acceptable level before feeding to the Auto Encoder.

5.3 Future work

Calculations in the python implementation of the SNN Auto Encoder model are basically based on multi-dimensional ‘numpy’ arrays. Support of ‘numpy’ library for mathematical operations on multi-dimensional arrays and linear algebra functions such as Frobenius norm has leveraged the implementation of the SNN Auto Encoder network. However, computation time increases exponentially with the size of the ‘numpy’ multi-dimensional arrays. This is the primary reason to downsize the input image before feeding to the SNN Auto Encoder during the experiments. Also, when the number of layers in the Auto Encoder is increased, training time increases exponentially as the number of multi-dimensional arrays involved in computations increases. As a result, even though the model is capable of creating Deep SNN Auto Encoders (as the implementation enables the user to provide the number of layers as a parameter when declaring the Auto Encoder instance), a significant time period is required to train such Deep SNN Auto Encoders compared to the training time of Conventional Deep Auto Encoders implemented using libraries such as Keras.

These bottlenecks can be avoided by incorporating libraries such as ‘Theano’ and ‘TenserFlow’ to optimize the multi-dimensional array calculations using data flow / tenser programming techniques [19]. Current python implementation will need significant changes to incorporate such data flow programming techniques enhanced with GPU support.

Also, to resolve aforementioned scalability issues, it is possible to use services provided by AWS (such as EC2 / Amazon Machine Image) and Google Cloud AI (with tenser flow) where GPUs, and TPUs / ASIC (Application Specific Integrated Circuits) can be used which will speed up the computations when training the AutoEncoder for large data sets.

When using the AutoEncoder model for different image datasets, several preprocessing steps will be required such resizing, removing noise (or add noise for De-noising AutoEncoders) and morphing. Also, when applying the AutoEncoder model for data sets other than image data, suitable coding mechanism is needed.

In addition, other optimizers such as NAG, ‘adagrad’, ‘adadelta’/‘RMSProp’ etc. can be implemented for the Auto Encoder model to support fast training at the cost of the reconstruction error.

REFERENCES

[1] M. Kiselev, “Rate coding vs. temporal coding - Is optimum between?,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2016-Octob, no. November, pp. 1355–1359, 2016, doi: 10.1109/IJCNN.2016.7727355.

[2] Tim Utz Krause, “Rate Coding and Temporal Coding in a Neural Network,” *Thesis Diss.*, 2014, [Online]. Available: http://www.ini.rub.de/PEOPLE/rolf/articles/krausetim-msc.pdf.

[3] E. Hunsberger and C. Eliasmith, “Spiking Deep Networks with LIF Neurons,” pp. 1–9, 2015, [Online]. Available: http://arxiv.org/abs/1510.08829.

[4] T. Iakymchuk, A. Rosado-Muñoz, J. F. Guerrero-Martínez, M. Bataller-Mompeán, and J. V. Francés-Víllora, “Simplified spiking neural network architecture and STDP learning algorithm applied to image classification,” *Eurasip J. Image Video Process.*, 2015, doi: 10.1186/s13640-015-0059-4.

[5] J. H. Lee, T. Delbruck, and M. Pfeiffer, “Training deep spiking neural networks using backpropagation,” *Front. Neurosci.*, vol. 10, no. NOV, pp. 1–10, 2016, doi: 10.3389/fnins.2016.00508.

[6] M. Kiselev, “A synaptic plasticity rule providing a unified approach to supervised and unsupervised learning,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2017-May, no. May 2017, pp. 3806–3813, 2017, doi: 10.1109/IJCNN.2017.7966336.

[7] A. Tavanaei and A. Maida, “BP-STDP: Approximating backpropagation using spike timing dependent plasticity,” *Neurocomputing*, 2019, doi: 10.1016/j.neucom.2018.11.014.

[8] H. Paugam-Moisy, R. Martinez, and S. Bengio, “A supervised learning approach

based on STDP and polychronization in spiking neuron networks," *ESANN 2007 Proc. - 15th Eur. Symp. Artif. Neural Networks*, pp. 427–432, 2007.

[9] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*. 2019, doi: 10.1016/j.neunet.2018.12.002.

[10] A. Kasiński and F. Ponulak, "Comparison of supervised learning methods for spike time coding in spiking neural networks," *International Journal of Applied Mathematics and Computer Science*. 2006.

[11] F. Ponulak and A. Kasiński, "Supervised learning in spiking neural networks with ReSuMe: Sequence learning, classification, and spike shifting," *Neural Computation*. 2010, doi: 10.1162/neco.2009.11-08-901.

[12] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *IEEE Trans. Neural Networks Learn. Syst.*, 2018, doi: 10.1109/TNNLS.2017.2726060.

[13] S. M. Bohte, H. La Poutré, and J. N. Kok, "Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons," *Neurocomputing*, 2000.

[14] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "STDP-based spiking deep convolutional neural networks for object recognition," *Neural Networks*, 2018, doi: 10.1016/j.neunet.2017.12.005.

[15] K. S. Burbank, "Mirrored STDP Implements Autoencoder Learning in a Network of Spiking Neurons," *PLoS Comput. Biol.*, 2015, doi: 10.1371/journal.pcbi.1004566.

[16] S. Ruder, "An overview of gradient descent optimization algorithms," pp. 1–14, 2016, [Online]. Available: http://arxiv.org/abs/1609.04747.

[17] S. Sun, Z. Cao, H. Zhu, and J. Zhao, "A Survey of Optimization Methods From a Machine Learning Perspective," *IEEE Trans. Cybern.*, pp. 1–14, 2019, doi:

10.1109/tcyb.2019.2950779.

[18] R. Grosse, “Exploding and Vanishing Gradients,” *Cs.Toronto.Edu*, pp. 1–11, 2017, [Online]. Available: http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15 Exploding and Vanishing Gradients.pdf.

[19] A. Shatnawi, G. Al-Bdour, R. Al-Qurran, and M. Al-Ayyoub, “A comparative study of open source deep learning frameworks,” *2018 9th Int. Conf. Inf. Commun. Syst. ICICS 2018*, vol. 2018-Janua, no. April, pp. 72–77, 2018, doi: 10.1109/IACS.2018.8355444.

APPENDIX

Python implementations for functions mentioned in section 3.6 are provided below.

```
#!/usr/bin/env python

#imports
import sys
import inspect
import imageio
import os

import numpy as np
from numpy import linalg as LA
from random import shuffle
import matplotlib
from matplotlib import pyplot as plt
import scipy
import scipy.misc
from PIL import Image

EXP_MAX_SPIKE_TIME = np.exp(3)

FUNCTION_LOG_LEVELS=    {
  "GetCausalSetWithOuputSpikeTime": 0,
  "GenerateWeightMatrix": 0,
  "GetOutputVectorSpikeTimes": 0,
  "CalculateFinalOutputDerivativeWRTInputSpikeTimes": 0,
  "CalculateFinalderivativesWRTWeights":0,
  "GetWeightAdjustmentArrayMap":0,
  "regularizeWeight":0
}
```

Figure A: 1 Imports and Definitions

```
def DEBUGPRINT(LogLevel, callerFunctionName, *args, **kwargs):
    allowewdLogLevel = 2
    allowewdLogLevel = FUNCTION_LOG_LEVELS.get(callerFunctionName, 2)
    if allowewdLogLevel >= LogLevel:
        print( "__",callerFunctionName,"__"+" \t".join(map(str,args))+"", **kwargs)
```

Figure A: 2 Log print function

```
def GenerateWeightMatrix(neuronCountArray):
    numberOfLayers = len(neuronCountArray)
    weghitMatrixList =  [None] * (numberOfLayers - 1)

    for layerIndex in range(1,numberOfLayers):      #first layer is input layer, so no weights associated
        currentLayerNeuronCount = neuronCountArray[layerIndex]
        previousLayerNeuronCount = neuronCountArray[layerIndex - 1]

        initialWeightPerNeuron = INITIAL_SUM_OF_WEIGHTS_PER_NEURON / previousLayerNeuronCount
        currentLayerWeightMatrix = np.full((previousLayerNeuronCount,currentLayerNeuronCount), initialWeightPerNeuron)
        for prevNeuronIndex in range(0,previousLayerNeuronCount):
            for currNeuronIndex in range(0,currentLayerNeuronCount):
                perturbationFactor = ((np.random.rand(1)[0]) / 10)
                currentLayerWeightMatrix[prevNeuronIndex][currNeuronIndex] = (currentLayerWeightMatrix[prevNeuronIn-
dex][currNeuronIndex])*perturbationFactor

        weghitMatrixList[layerIndex - 1] = currentLayerWeightMatrix
    return weghitMatrixList
```

Figure A: 3 Function to generate initial weight matrix

```
def GetCausalSetWithOuputSpikeTime(inputSpikeTimeArr,weightVector):
    causalIndexSet = np.empty(shape=(0, 0))
    estimatedOutputSpTime = MAX_SPIKE_TIME
    totalWeight = 0.0

    inputTimeColIndex = 1
    inputWeightColIndex = 2
    ArrayIndexColIndex = 0

    if len(inputSpikeTimeArr) != len(weightVector):
        raise Exception('Mismathced array lengths in GetCausalSet function!')

    numberOfInputNeurons = len(inputSpikeTimeArr)
    neuronIndexSet = np.arange(numberOfInputNeurons)
    neuronIndexSet = np.reshape(neuronIndexSet, (numberOfInputNeurons, 1))
    mergedArray = np.concatenate((neuronIndexSet,inputSpikeTimeArr,weightVector),axis=1)
    sortedinputSpikeTimeArr = 
mergedArray[np.lexsort((mergedArray[:,ArrayIndexColIndex],mergedArray[:,inputWeightColIndex],mergedArray[:,inputTimeColIn
dex]))]

    for nueronIndex in range(0, numberOfInputNeurons):
        if nueronIndex == (numberOfInputNeurons - 1):
            nextInputSpikeTime = MAX_SPIKE_TIME
        else:
            nextInputSpikeTime = sortedinputSpikeTimeArr[(nueronIndex+1),inputTimeColIndex]


        totalWeight = sortedinputSpikeTimeArr[:(nueronIndex+1),inputWeightColIndex].sum()
        if totalWeight > 1:
            inputSpikeTimeBywieght = 0
            for neuronInnerIndex in range(0, (nueronIndex+1)):
                inputSpikeTimeBywieght += 
sortedinputSpikeTimeArr[neuronInnerIndex,inputTimeColIndex]*sortedinputSpikeTimeArr[neuronInnerIndex,inputWeightColIndex]

            estimatedOutputSpTime = inputSpikeTimeBywieght / (totalWeight - 1)
            if estimatedOutputSpTime < nextInputSpikeTime:
                causalIndexSet = sortedinputSpikeTimeArr[0:(nueronIndex+1),ArrayIndexColIndex]
                return causalIndexSet,estimatedOutputSpTime,totalWeight

    estimatedOutputSpTime = MAX_SPIKE_TIME
    totalWeight = 0.0
    return causalIndexSet,estimatedOutputSpTime,totalWeight
```

Figure A: 4 Function to get output spike time with causal set

```python
def GetOutputVectorSpikeTimes(inputTimesArray, neuronCountArray,weightMatrixList):

    numberOfProcessingLayers = len(weightMatrixList)
    neuronCountInOutputLayer = neuronCountArray[len(neuronCountArray) - 1]
    outputSpikeTimeArray = np.full((1, neuronCountInOutputLayer), MAX_SPIKE_TIME)

    #Variables to store data required for differentiation calculations
    ds_causelSetForNN = []
    ds_outputSpikeTimeForNN = [None]*(numberOfProcessingLayers)
    ds_DERWRTspikeTime = [None]*(numberOfProcessingLayers)
    ds_DERWRTspikeTime = [None]*(numberOfProcessingLayers)


    currentLayerOutputSpikeTimeArray = np.asarray(inputTimesArray)

    for layerIndex in range (1,(numberOfProcessingLayers+1)):   #Lets ignore input layer. Start from 1st hidden layer

        numberOfNeuronsInPrevLayer = neuronCountArray[layerIndex - 1]
        numberOfNeuronsInCurrentLayer = neuronCountArray[layerIndex]

        DEBUGPRINT(2, 'GetOutputVectorSpikeTimes','In GetOutputVectorSpikeTimes function, Processing the layer : ',
layerIndex,', numberOfNeuronsInPrevLayer : ',numberOfNeuronsInPrevLayer
                ,'numberOfNeuronsInCurrentLayer',numberOfNeuronsInCurrentLayer)

        ds_DERWRTspikeTime[layerIndex - 1] = np.full((numberOfNeuronsInPrevLayer, numberOfNeuronsInCurrentLayer), 0.0)
        ds_DERWRTspikeTime[layerIndex - 1] = np.full((numberOfNeuronsInPrevLayer, numberOfNeuronsInCurrentLayer), 0.0)
        ds_outputSpikeTimeForNN[layerIndex - 1] = np.full((1, numberOfNeuronsInCurrentLayer), MAX_SPIKE_TIME)



        CurLyrInputTimeArr = currentLayerOutputSpikeTimeArray
        currentLayerOutputSpikeTimeArray = np.full((1, neuronCountArray[layerIndex]), MAX_SPIKE_TIME)

        for neuronIndex in range (0,numberOfNeuronsInCurrentLayer):
            weightMatrixForCurrentLayer = weightMatrixList[layerIndex - 1]
            curNeuronWeightArr = (weightMatrixForCurrentLayer[:,neuronIndex]).reshape(numberOfNeuronsInPrevLayer,1)
            CurLyrInputTimeArr = CurLyrInputTimeArr.reshape(numberOfNeuronsInPrevLayer,1)
            CauselSetWithOutputSpikeTime = GetCausalSetWithOuputSpikeTime(CurLyrInputTimeArr,curNeuronWeightArr)
            currNeuronSpikeTime = CauselSetWithOutputSpikeTime[1]
            currentLayerOutputSpikeTimeArray[0][neuronIndex] = currNeuronSpikeTime
            causalSetForNeuron = (CauselSetWithOutputSpikeTime[0]).astype(int)
            totalWeightOfInputNuerons = CauselSetWithOutputSpikeTime[2]

            #Store derivative values
            #Derivate WRT weights and input spike time
            if (currNeuronSpikeTime < MAX_SPIKE_TIME and len(causalSetForNeuron) != 0 and totalWeightOfInputNuerons > 1):
                denomonatorVal = totalWeightOfInputNuerons - 1  #TODO - Use Tensors

                for subjectNueronIndex in causalSetForNeuron:
                    spikeTimeDiff = (CurLyrInputTimeArr[subjectNueronIndex][0] - currNeuronSpikeTime)

                    (ds_DERWRTspikeTime[layerIndex - 1])[subjectNueronIndex][neuronIndex] =  ( spikeTimeDiff
/denomonatorVal)
                    (ds_DERWRTspikeTime[layerIndex - 1])[subjectNueronIndex][neuronIndex] =
(curNeuronWeightArr[subjectNueronIndex][[0]] / denomonatorVal)


            #Lets calculate and return derivatives for future weight adjustment

        ds_outputSpikeTimeForNN[layerIndex - 1] = currentLayerOutputSpikeTimeArray

        if layerIndex == numberOfProcessingLayers:
            outputSpikeTimeArray = currentLayerOutputSpikeTimeArray

    DEBUGPRINT(1, 'GetOutputVectorSpikeTimes',' \n__outputSpikeTimeArray__: \n', outputSpikeTimeArray)
    return outputSpikeTimeArray,ds_outputSpikeTimeForNN,ds_DERWRTspikeTime,ds_DERWRTspikeTime
```

Figure A: 5 Function to calculate output spike times

```
def CalculateFinalOutputDerivativeWRTInputSpikeTimes(derivativesWRTspikeTime):

    noOfLayers = len(derivativesWRTspikeTime) + 1

    outputSpikeTimeDerivativeWRTAllinputSpikeTimeMap = {}
    lastLayerIndex = (noOfLayers - 1) - 1
    numberOfNueronsInLastLayer =  ((derivativesWRTspikeTime[lastLayerIndex]).shape)[1]
    for layerIndex in range (lastLayerIndex, -1, -1):
        if layerIndex == lastLayerIndex:
            outputSpikeTimeDerivativeWRTAllinputSpikeTimeMap[layerIndex] = derivativesWRTspikeTime[lastLayerIndex]
        else:
            nextLayerfinalDERArray = outputSpikeTimeDerivativeWRTAllinputSpikeTimeMap[layerIndex + 1]
            CurrLyrLocalDERArray = derivativesWRTspikeTime[layerIndex]

            numberOfNueronsInCurrentLayer = ((derivativesWRTspikeTime[layerIndex]).shape)[0]
            derivativesMatrixForCurrentLayer = np.full((numberOfNueronsInCurrentLayer, numberOfNueronsInLastLayer), 0.0)
            for currLayerNeuronIdx in range (0,numberOfNueronsInCurrentLayer):

                currNeuronFinalDER = np.full((1,numberOfNueronsInLastLayer),0.0)
                nextLyrNeuronCount = ((derivativesWRTspikeTime[layerIndex]).shape)[1]
                for nextLayerNeuronIdx in range (0,nextLyrNeuronCount):
                    currNeuronFinalDER +=
(CurrLyrLocalDERArray[currLayerNeuronIdx][nextLayerNeuronIdx]*nextLayerfinalDERArray[nextLayerNeuronIdx])

                derivativesMatrixForCurrentLayer[currLayerNeuronIdx] = currNeuronFinalDER

            outputSpikeTimeDerivativeWRTAllinputSpikeTimeMap[layerIndex] = derivativesMatrixForCurrentLayer
    return outputSpikeTimeDerivativeWRTAllinputSpikeTimeMap
```

Figure A: 6 Function to calculate derivatives of output spike time w.r.t. input spike times

```python
def GetWeightAdjustmentArrayMap(outputArr, expectedOutputArr,FinalDERMap):

    if len(expectedOutputArr) != (outputArr.shape)[1]:
        raise Exception('In CalculateErrorDerivateWRTWeight. outputArr and expectedOutputArr lengths mismatch')

    errorDerivateWRTToWeightMap = {}
    numberOfNueronsInLastLayer = len(expectedOutputArr)

    for key, value in FinalDERMap.items():
        lyrIdx = key
        ouputDerivative3DArr = value

        numberOfNueronsInCurrentLayer = (ouputDerivative3DArr.shape)[0]
        numberOfNueronsInNextLayer = (ouputDerivative3DArr.shape)[1]
        errorDerivateWRTToWeightMap[lyrIdx] = np.full((numberOfNueronsInCurrentLayer,numberOfNueronsInNextLayer), 0.0)

        if (ouputDerivative3DArr.shape)[2] != numberOfNueronsInLastLayer:
            raise Exception('In CalculateErrorDerivateWRTWeight. ouputDerivative3DArr length  does not match with
numberOfNueronsInLastLayer')

        for currLyrNeuronIdx in  range(0, numberOfNueronsInCurrentLayer):
            for nextLyrNeuronIdx in  range(0, numberOfNueronsInNextLayer):
                weightChangeVal = 0.0
                for lasttLyrNeuronIdx in  range(0, numberOfNueronsInLastLayer):
                    if outputArr[0][lasttLyrNeuronIdx] < EXP_MAX_SPIKE_TIME:
                        weightChangeVal += (outputArr[0][lasttLyrNeuronIdx] -
expectedOutputArr[lasttLyrNeuronIdx])*(FinalDERMap[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx][lasttLyrNeuronIdx])
                    else:
                        weightChangeVal += (EXP_MAX_SPIKE_TIME -
expectedOutputArr[lasttLyrNeuronIdx])*(FinalDERMap[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx][lasttLyrNeuronIdx])



                totalWeightAdjustement = (-1*WEIGHT_ADJUSTMENT_FACTOR)*weightChangeVal
                errorDerivateWRTToWeightMap[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx] = totalWeightAdjustement

    ##FrobeniusNorm to avoid large jumps in weights due to small denominator values
    for lyrIdx, errorDerivateWRTToWeightMatrix in errorDerivateWRTToWeightMap.items():
        FrobeniusNormForCurrentMatrix =  LA.norm(errorDerivateWRTToWeightMatrix)
        numberOfSourceNeurons = (errorDerivateWRTToWeightMatrix.shape)[0]
        if (FrobeniusNormForCurrentMatrix / numberOfSourceNeurons) > FROBENIUS_NORM_THREASHOLD_PER_SOURCE_NEURON:
            normalizationValue = FrobeniusNormForCurrentMatrix /
(numberOfSourceNeurons*FROBENIUS_NORM_THREASHOLD_PER_SOURCE_NEURON)
            errorDerivateWRTToWeightMap[lyrIdx] = (errorDerivateWRTToWeightMap[lyrIdx] / normalizationValue)

    return errorDerivateWRTToWeightMap
```

Figure A: 7 Function to calculate final weight adjustment values

```
def CalculateFinalderivativesWRTWeights(finalDERWRTInputTimes, DERWRTweight):

    if len(finalDERWRTInputTimes) !=    len(DERWRTweight):
        raise Exception('Mismathced array lengths in CalculateFinalderivativesWRTWeights function!')

    finalWeightDERMap = {} # each element will have a 3D array

    numberOfLayers = len(finalDERWRTInputTimes) + 1
    lastLayerIndex = (numberOfLayers - 1) - 1
    numberOfNueronsInLastLayer = ((finalDERWRTInputTimes[lastLayerIndex]).shape)[1]


    for lyrIdx in range (lastLayerIndex, -1, -1):

        CurLayerNueronCount = ((DERWRTweight[lyrIdx]).shape)[0]
        numberOfNueronsInNextLayer = ((DERWRTweight[lyrIdx]).shape)[1]
        finalWeightDERMap[lyrIdx] = np.full((CurLayerNueronCount,numberOfNueronsInNextLayer,numberOfNueronsInLastLayer),
0.0)

        if lyrIdx == lastLayerIndex:

            for currLyrNeuronIdx in range (0,CurLayerNueronCount):
                for nextLyrNeuronIdx in range (0,numberOfNueronsInNextLayer):
                    for lastLayerNeuronIndex in range (0,numberOfNueronsInLastLayer):
                        if lastLayerNeuronIndex == nextLyrNeuronIdx:
                            finalWeightDERMap[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx][lastLayerNeuronIndex] =
DERWRTweight[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx]

        else:

            if (((finalDERWRTInputTimes[lyrIdx+1]).shape[0]) !=  ((DERWRTweight[lyrIdx]).shape[1])):
                raise Exception('Mismathced matrix shapes in CalculateFinalderivativesWRTWeights function!')


            for currLyrNeuronIdx in range (0,CurLayerNueronCount):
                for nextLyrNeuronIdx in range (0,numberOfNueronsInNextLayer):
                    finalWeightDERMap[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx] =
DERWRTweight[lyrIdx][currLyrNeuronIdx][nextLyrNeuronIdx] *finalDERWRTInputTimes[lyrIdx+1][nextLyrNeuronIdx]

    return finalWeightDERMap
```

Figure A: 8 Function to calculate final derivatives w.r.t. synaptic weights

```
def regularizeWeight(weightMAtrixList,L2adjustementFactor, smallWeightAdjustFactor):

    regularizationMap = {}

    numberOfLayers = len(weightMAtrixList)

    for lyrIdx in range (0,numberOfLayers):
        regularizationMap[lyrIdx] = np.full((weightMAtrixList[lyrIdx].shape),0.0)
        ArrayOfWeightSumForEachNueron = np.sum(weightMAtrixList[lyrIdx], axis=0)
        if (weightMAtrixList[lyrIdx].shape)[1] != len(ArrayOfWeightSumForEachNueron):
            raise Exception('In regularizeWeight. ArrayOfWeightSumForEachNueron does not match with weightMAtrixList
shape')

        numberOfNeurons = len(ArrayOfWeightSumForEachNueron)
        for neuronIdx in range (0,numberOfNeurons):
            if ArrayOfWeightSumForEachNueron[neuronIdx] < 1:        ##Sum Weights should be > 1, otherwise neuron will
not fire
                regularizationMap[lyrIdx][:,neuronIdx] =
smallWeightAdjustFactor*abs(weightMAtrixList[lyrIdx][:,neuronIdx])

            regularizationMap[lyrIdx][:,neuronIdx] += -1*L2adjustementFactor*weightMAtrixList[lyrIdx][:,neuronIdx]

    return regularizationMap
```

Figure A: 9 Function to regularize weights

```
def ConvertPixelValueToSpikeTime(inputData, revertConversion = False):

    maxPixelValue = 255
    minSpikeTime = 0.0
    maxSpikeTime = 3.0

    if revertConversion == False:   ##Convert Pixel Value To Spike Time
        inputPixelValues = inputData
        multilyingFactor = ((maxSpikeTime - minSpikeTime) / maxPixelValue)
        inputSpikeTimes = [(maxSpikeTime - PixelVal * multilyingFactor) for PixelVal in inputPixelValues]

        inputSpikeTimesInZDomain = np.exp(inputSpikeTimes)

        return inputSpikeTimesInZDomain

    else:   ##Convert SpikeTime to PixelValue
        inputSpikeTimes = inputData
        inputSpikeTimes = np.log(inputSpikeTimes)
        multilyingFactor = (maxPixelValue / (maxSpikeTime - minSpikeTime))
        inputPixelValues = [(maxSpikeTime - spikeTime) * multilyingFactor for spikeTime in inputSpikeTimes]

        print("shape : ", (inputPixelValues[0]).shape[0])

        for Index in range((inputPixelValues[0].shape[0])):
            if (inputPixelValues[0])[Index] < 0.0:
                print("Val : ", (inputPixelValues[0])[Index])
                (inputPixelValues[0])[Index] = 0.0


        return inputPixelValues
```

Figure A: 10 Mapping between pixel value and z-domain spike time

```python
def TrainAutoEncoder(Train_data,givenNetworkStruct,givenweightMAtrix):
    NumberOfTrainingData = len(Train_data)
    NumberOfRowsInTrainingData = Train_data[0].shape[0]
    NumberOfColumnsInTrainingData = Train_data[0].shape[1]
    totalNumberOfInputNuerons = NumberOfRowsInTrainingData*NumberOfColumnsInTrainingData
    totalNumberOfOutputNuerons = totalNumberOfInputNuerons

    NetworkStructure = givenNetworkStruct
    weightMatrix = givenweightMAtrix
    numbeOfLayers = len(weightMatrix)
    shuffledInputDataIndexList = list(range(NumberOfTrainingData))
    ##shuffle(shuffledInputDataIndexList) ##Lets do shuffling only once, All epochs use same shuffled data order
    for epochIteator in range (0,NUMBER_OF_ITERATIONS_OVER_TRAINNIG_DATA):
        miniBatchIndex = 0
        weightAdjustmentValueMap = {}
        curBatchNueronWiseErr = np.full((1,totalNumberOfOutputNuerons),0.0)
        ouputNueronWiseErrorForwholeDataSet = np.full((1,totalNumberOfOutputNuerons),0.0)
        for curDataIdx in range (0,NumberOfTrainingData):
            miniBatchIndex = miniBatchIndex +   1
            shufflednputDataIndex = shuffledInputDataIndexList[curDataIdx]
            rearrangedInputData = (Train_data[shufflednputDataIndex]).reshape(1,(NumberOfRowsInTrainingData*NumberOfCol-
umnsInTrainingData))
            inputSpikeTimesArr = ConvertPixelValueToSpikeTime((rearrangedInputData.reshape(-1,)).tolist())

            K = GetOutputVectorSpikeTimes(inputSpikeTimesArr,NetworkStructure,weightMatrix)
            outputSpikeTimeDerivativeWRTEachinputSpikeTimeMap = CalculateFinalOutputDerivativeWRTInputSpikeTimes(K[2])
            outputDerivateWRTToCurrWeightMap = CalculateFinalderivativesWRTWeights(outputSpikeTimeDerivativeWRTEachin-
putSpikeTimeMap, K[3])
            weightAdjustmentMap = GetWeightAdjustmentArrayMap(K[0], inputSpikeTimesArr,outputDerivateWRTToCurrWeight-
Map)
            AdjustedOutputtimeArr = np.array(K[0],copy=True)
            for indexOutSpikeTimeArr in range(0, AdjustedOutputtimeArr.shape[1]):
                if AdjustedOutputtimeArr[0][indexOutSpikeTimeArr] > EXP_MAX_SPIKE_TIME:
                    AdjustedOutputtimeArr[0][indexOutSpikeTimeArr] = EXP_MAX_SPIKE_TIME

            weightAdjustmentValueMap[miniBatchIndex] = weightAdjustmentMap
            curBatchNueronWiseErr = curBatchNueronWiseErr + abs(AdjustedOutputtimeArr - inputSpikeTimesArr)
            ouputNueronWiseErrorForwholeDataSet = ouputNueronWiseErrorForwholeDataSet + abs(AdjustedOutputtimeArr - in-
putSpikeTimesArr)

            if ((miniBatchIndex == MINI_BATCH_SIZE) or (curDataIdx == (NumberOfTrainingData - 1))):
                weightRegularizationArr = regularizeWeight(weightMatrix, L2_REGULARIZATION_ADJUSTMENT_FACTOR,
SMALL_WEIGHTS_REGULARIZATION_ADJUSTMENT_FACTOR)
                curBatchWeighAdjustmets = {}
                for layerIndex3 in range (0,numbeOfLayers):
                    curBatchWeighAdjustmets[layerIndex3] = np.full(((weightRegularizationArr[layerIndex3]).shape),0.0)

                for inputDataIndex, weightAdjustmentVal in weightAdjustmentValueMap.items():
                    if len(weightMatrix) != len(weightAdjustmentVal):
                        raise Exception('Invalid lengths (number of layers) in weightMatrix and weightAdjustmentMap')
                    for lyrIdx in range (0,numbeOfLayers):
                        if len(weightMatrix[lyrIdx].shape) != len(weightAdjustmentVal[lyrIdx].shape) != (weightRegulari-
zationArr[lyrIdx].shape) != (curBatchWeighAdjustmets[lyrIdx].shape):
                            raise Exception('Invalid lengths in weightMatrix and weightAdjustmentMap')
                        curBatchWeighAdjustmets[lyrIdx] += weightAdjustmentVal[lyrIdx]

                for lyrIdx2 in range (0,numbeOfLayers):
                    weightMatrix[lyrIdx2] = weightMatrix[lyrIdx2] + curBatchWeighAdjustmets[lyrIdx2] + weightRegulariza-
tionArr[lyrIdx2]

                detailWeightAdjustmentMap = weightAdjustmentValueMap
                weightAdjustmentValueMap = {}

                curBatchNueronWiseErr = (curBatchNueronWiseErr / miniBatchIndex)
                DEBUGPRINT(2,'TrainAutoEncoder','AverageError for current Mini Batch : ', curBatchNueronWiseErr)
                totalErrorForCurrentMiniBatch =  np.sum(curBatchNueronWiseErr) / totalNumberOfOutputNuerons
                DEBUGPRINT(0,'TrainAutoEncoder','OverAll Error for current Mini Batch : ', totalErrorForCurrentMiniBatch)
                curBatchNueronWiseErr = np.full((1,totalNumberOfOutputNuerons),0.0)
                miniBatchIndex = 0
        detailWeightAdjustmentMapOld = {}
        weightMAtrixListOld = {}
        if(epochIteator != (NUMBER_OF_ITERATIONS_OVER_TRAINNIG_DATA - 1)):
            detailWeightAdjustmentMapOld = detailWeightAdjustmentMap
            weightMAtrixListOld = weightMatrix
        ouputNueronWiseErrorForwholeDataSet = (ouputNueronWiseErrorForwholeDataSet / NumberOfTrainingData)
        totalErrorForCurrentMiniBatch =  np.sum(ouputNueronWiseErrorForwholeDataSet) / totalNumberOfOutputNuerons
    return weightMatrix, K, detailWeightAdjustmentMap,detailWeightAdjustmentMapOld,weightMAtrixListOld, weightRegulariza-
tionArr
```

Figure A: 11 Auto Encoder training function

```
def GenerateReconstruction(givenNetworkStruct,givenWeightMAtrixList,TestDataSet):

    NetworkStructure = givenNetworkStruct
    weightMAtrixList = givenWeightMAtrixList

    NumberOfRowsInTestDataSet = TestDataSet[0].shape[0]
    NumberOfColumnsInTestDataSet = TestDataSet[0].shape[1]
    totalNumberOfInputNuerons = NumberOfColumnsInTestDataSet*NumberOfColumnsInTestDataSet
    totalNumberOfOutputNuerons = totalNumberOfInputNuerons

    NumberOfTestData = len(TestDataSet)
    print('NumberOfTestData : ', NumberOfTestData)

    for trainingDataInstanceIndex in range (0,NumberOfTestData):
        rearrangedInputData =
(TestDataSet[trainingDataInstanceIndex]).reshape(1,(NumberOfRowsInTestDataSet*NumberOfColumnsInTestDataSet))
        inputSpikeTimesArr = ConvertPixelValueToSpikeTime((rearrangedInputData.reshape(-1,)).tolist())
        K = GetOutputVectorSpikeTimes(inputSpikeTimesArr,NetworkStructure,weightMAtrixList)
        genratedoutputspiketimeArr = K[0]
        for indexOutSpikeTimeArr in range(0, genratedoutputspiketimeArr.shape[1]):
            if genratedoutputspiketimeArr[0][indexOutSpikeTimeArr] > EXP_MAX_SPIKE_TIME:
                genratedoutputspiketimeArr[0][indexOutSpikeTimeArr] = EXP_MAX_SPIKE_TIME
        DEBUGPRINT(2,'TestOnArtifialData_2','TestDataSetInstanceIndex : ',trainingDataInstanceIndex,', Err : \n\n',(K[0]
- inputSpikeTimesArr))
        ouputNueronWiseError = abs(genratedoutputspiketimeArr - inputSpikeTimesArr)
        AvgErrPerNeuron =  np.sum(ouputNueronWiseError) / totalNumberOfOutputNuerons
        DEBUGPRINT(0,'TestOnArtifialData_2','Average Error for current TestDataSetInstanceIndex : ', AvgErrPerNeuron)

    return K[0]
```

Figure A: 12 Function for Generating Reconstruction

```
#!/usr/bin/env python
import SNNAutoEncoder

SNNAutoEncoder.WEIGHT_ADJUSTMENT_FACTOR  = 0.005
SNNAutoEncoder.L2_REGULARIZATION_ADJUSTMENT_FACTOR = 0.0
SNNAutoEncoder.FROBENIUS_NORM_THREASHOLD_PER_SOURCE_NEURON = 0.02
SNNAutoEncoder.INITIAL_SUM_OF_WEIGHTS_PER_NEURON =100.1
SNNAutoEncoder.SMALL_WEIGHTS_REGULARIZATION_ADJUSTMENT_FACTOR = 2.1
SNNAutoEncoder.MAX_SPIKE_TIME = 100000000.0
SNNAutoEncoder.MAX_ITERATION_COUNT_FOR_A_INPUT = 1000

SNNAutoEncoder.MINI_BATCH_SIZE = 5
SNNAutoEncoder.NUMBER_OF_ITERATIONS_OVER_TRAINNIG_DATA = 1



NetworkStructure = [(IMAGE_HIGHT*IMAGE_WIDTH),48,24,48,(IMAGE_HIGHT*IMAGE_WIDTH)]

##Resized input image dimensions a
IMAGE_HIGHT = 12
IMAGE_WIDTH = 12
TOTAL_NUMBER_OF_SAMPLES_TO_CONSIDER  = 1000
NUMBER_OF_EPOCHS = 100
STARTING_EPOCH = 0
if (len(sys.argv) - 1) > 0:
    STARTING_EPOCH = int(sys.argv[1])



###########     LOAD  INITIAL WIGHT MATIX FROM DISK   ###########
if STARTING_EPOCH == 0:
    initialWeightMatrix = SNNAutoEncoder.GenerateWeightMatrix(NetworkStructure)
else:
    lastSavedArrIndex = STARTING_EPOCH - 1
    loadedInitWeightArr = np.load('SavedWeightSnapShot.npz', allow_pickle=True)
    initialWeightMatrix = loadedInitWeightArr['arr_0']



SelectedSampleSet = np.zeros((TOTAL_NUMBER_OF_SAMPLES_TO_CONSIDER,IMAGE_HIGHT, IMAGE_WIDTH))
mnistDataIndex = 0

##load mnist data set and initiate arrays
Mnistdataset= np.load('mnist.npz')
SelectedSampleSet = np.zeros((TOTAL_NUMBER_OF_SAMPLES_TO_CONSIDER,IMAGE_HIGHT, IMAGE_WIDTH))
for ministDataIndex in range (0,TOTAL_NUMBER_OF_SAMPLES_TO_CONSIDER):
    resizedImage =
np.array(Image.fromarray(Mnistdataset['x_train'][ministDataIndex]).resize((IMAGE_HIGHT,IMAGE_WIDTH),Image.BILINEAR))

    if (SelectedSampleSet[ministDataIndex].shape[0] != IMAGE_HIGHT) or (SelectedSampleSet[ministDataIndex].shape[1] !=
IMAGE_WIDTH):
         sys.exit('Resized image size is incorrect')

##Train the network
for epochNumber in range (STARTING_EPOCH,(STARTING_EPOCH+NUMBER_OF_EPOCHS)):
    if  epochNumber == STARTING_EPOCH:
        print("First iteration - using initial weights")
        M = SNNAutoEncoder.TrainAutoEncoder(SelectedSampleSet,SelectedSampleSet,NetworkStructure,initialWeightMatrix)
    else:
        print("subsequent iteration - using adjusted weights. Iteration NUMBER : ",epochNumber)
        M = SNNAutoEncoder.TrainAutoEncoder(SelectedSampleSet,SelectedSampleSet,NetworkStructure,M[0])


## Save synaptic weight snapshot - for future loading purposes -> retrain the network
np.savez('SavedWeightSnapShot',M[0])
```

Figure A: 13 Sample code for training Auto Encoder

```
from keras.layers import Input, Dense
from keras.models import Model
from keras import losses

inputImageSize
hiddenLayerNueronCount = 24

inputData = Input(shape=(inputImageSize,))
hiddenLayer = Dense(hiddenLayerNueronCount, activation='relu')(input_img)
outptutLayer = Dense(inputImageSize, activation='sigmoid')(hiddenLayer)
autoEncoderInstance = Model(inputData, outptutLayer)
autoEncoderInstance.compile(optimizer='sgd', loss=losses.mean_squared_error)

autoEncoderInstance.fit(inputTrainDataSet, inputTrainDataSet,
                epochs=100,
                batch_size=5,
                shuffle=True,
                validation_data=(inputTestDataSet, inputTestDataSet))

reconstructedoutput = autoEncoderInstance.predict(unseenDataSet)
```

Figure A: 14 Equivalent Conventional Auto Encoder