Adaptive Chaotic Injection to Reduce Overfitting in Artificial Neural Networks

By

Siobhan Reid

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

In partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba, Canada

Abstract

Artificial neural networks (ANNs) have become an integral tool in various fields of research. ANNs are mathematical models which can be trained to perform various prediction tasks. The effectiveness of an ANN can be impacted by overfitting which occurs when the ANN overfits to the training data. As a result, the ANN does not generalize well to novel data. In our research, we assess the feasibility of using a chaotic strange attractor to generate sequences of values to inject into an ANN to reduce overfitting. An adaptive method was developed to scale and inject the values into the neurons throughout training. The chaotic injection (CI) was tested on three benchmark datasets using different ANN models. The results were compared against the baseline ANN, dropout (DO), and Gaussian noise injection (GNI). The CI improved the performance of the ANN and converged faster than DO and GNI.

Acknowledgements

I would like to thank the following individuals and organizations:

1) Professor Ferens for his guidance and supervision throughout my master's degree.

2) Professor Kinsner for his guidance while working on my research and coursework.

3) Canadian Tire and Mitacs for their support and funding throughout my research.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronyms	Description
ACC	Accuracy
ANN	Artificial Neural Network
BERT	Bidirectional Encoder Representations from Transformers
CI	Chaotic Injection
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional Neural Network
DO	Dropout
F1	F1-Score
FN	False Negative
FP	False Positive
GNI	Gaussian Noise Injection
GPU	Graphic Processing Unit
MNIST	Modified National Institute of Standards and Technology
NI	Noise Injection
NPV	Negative-Predictive Value
PDF	Probability Density Function
PPV	Positive-Predictive Value
ReLU	Rectified Linear Unit
RGB	Red Blue Green
SN	Sensitivity
SP	Specificity
TN	True Negative
TP	True Positive

List of Symbols

Symbol	Description
α	Scaling value of the tent map
$a_i^{(k)}$	Activation value of the i^{th}neuron in the output layer of an ANN
α_array	Array containing the α values for each epoch
α_max	Desired maximum scaling value of the tent map
β	Bias value of the tent map
$b_i^{(k-1)}$	Bias connection to the i^{th}neuron in the layer k
$c[n]$	Output of the circular map at iteration n
dy/dx	Derivative
$epoch\_num$	Epoch number
i	Position of i^{th}neuron in the layer k
j	Position of j^{th}neuron in the layer $k-1$
k	Layer number of a neuron in an ANN
K	Bifurcation parameter of the circular map
$l[n]$	Output of the logistic map at iteration n
log	Logarithmic function with base e
max	Maximum function, returns the maximum number in a list
min	Minimum function, returns the minimum number in a list
n	Discrete time-step of an iterative map
N	Number of neurons in layer $k-1$ of an ANN
Ω	Bifurcation parameter of the circular map
π	Value of Pi

r	Bifurcation parameter of the logistic map
$s[n]$	Scaled tent map value at iteration n
$s[n]_i^{(k)}$	Scaled tent map value of the i^{th} neuron of layer k at iteration n
sin	Sine function
t[n]	Output of the tent map at iteration n
$t[n]_i^{(k)}$	Tent map value of the i^{th} neuron of layer k at iteration n
μ	Bifurcation parameter of the tent map
ω	Growth parameter of the logarithmic function
$w_{i,j}^{(k-1)}$	Weight connection from the j^{th}neuron in the layer $k-1$ to the i^{th}neuron in the layer k
$x_i^{(k)}$	Input into the i^{th} neuron of layer k
$y_i^{(k)}$	Activation value of the i^{th} neuron of layer k

1 Introduction

1.1 Motivation

Artificial neural networks (ANNs) are mathematical models inspired by the biological brain [1]. ANNs are used for prediction tasks, such as classification and regression. The use of ANNs has become widespread in various fields. Applications include object detection for self-driving cars [2], disease prediction in medicine [3], and malware detection in cybersecurity [4]. ANNs can be impacted by overfitting, which occurs when an ANN overfits to the training data. As a result, the ANN does not generalize well to novel data [5].

Common techniques to reduce overfitting include early stopping [6], dropout (DO) [7], regularization [8], and noise injection (NI) [9]. Similar to NI, chaotic strange attractors can be used to generate sequences of values, which we will refer to as chaotic values, to inject into an ANN. Injecting chaotic values into an ANN may better reflect the behaviour of the biological brain [10]–[13]. However, there is limited research in this area [14]–[17]. We want to expand this area of research by developing an adaptive method to inject chaotic values into an ANN to reduce overfitting.

1.2 Thesis Statement and Objectives

In this research, we assess the feasibility of using a chaotic strange attractor to generate sequences of values to inject into an ANN to reduce overfitting. We propose an adaptive method to scale and inject the values into the neurons throughout training.

The main objectives of this research include:

1) Developing an adaptive method to inject chaotic values or noise into an ANN.
2) Assessing the effectiveness of the chaotic injection (CI) to prevent overfitting.
3) Comparing the CI to NI.

1.3 Organization of Thesis

The thesis is organized into six main chapters, as described below in Table 1-1.

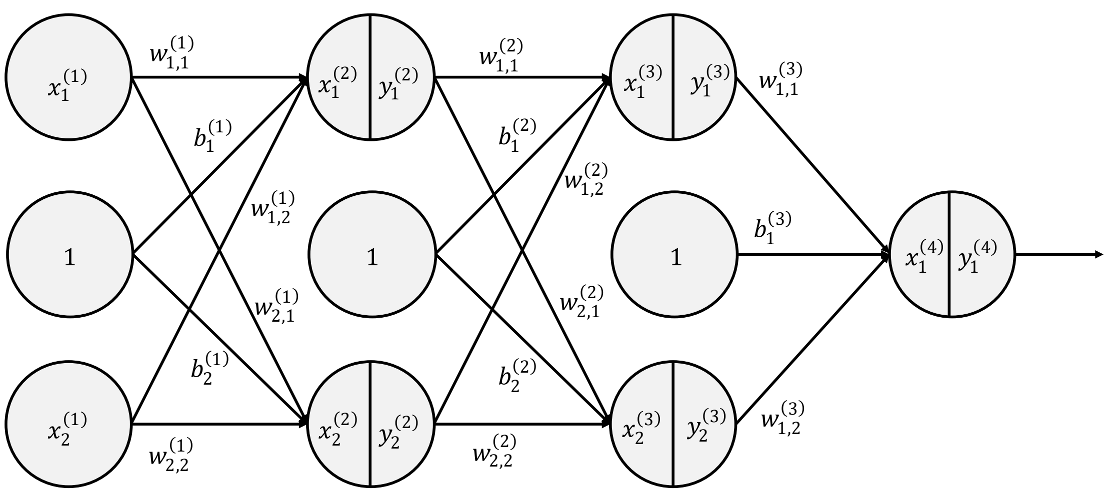
Table 1-1. Organization of Thesis.

Chapter	Description
1: Introduction	Chapter 1 introduces the thesis topic and objectives.
2: Background	Chapter 2 provides background information on ANNs, overfitting, techniques to reduce overfitting, NI, CI, and chaos theory.
3: Implementation	Chapter 3 provides the implementation details of the CI, including the selection of the attractor, the initialization and setup, the adaptive scaling method, the injection method, and the effects on backpropagation.
4: Testing	Chapter 4 provides a description of the datasets and ANN models used for testing the CI.
5: Results	Chapter 5 presents the results, including the ANNs' accuracy and loss per epoch, runtimes, and performance metrics.
6: Conclusion	Chapter 6 provides concluding remarks, recommendations for future work, and a summary of the contributions made to this field of study.

2 Background and Related Work

2.1 Artificial Neural Networks

ANNs are mathematical models used for prediction tasks, such as classification and regression [18]. When input data is passed to an ANN, the ANN processes the data and outputs a prediction. In supervised machine learning, a basic multilayer perceptron ANN consists of layers of artificial neurons connected via parameters referred to as weights. Input data is passed into the first layer of the ANN. In the following layers, the input into a neuron is the sum of outputs from the neurons in the previous layer multiplied by their weight values, in addition to a bias value. A neuron's input is passed through a non-linear activation function and then sent to the next layer. The neurons in the final layer output the predictions. Fig. 2-1 shows the structure of a basic multilayer perceptron ANN with two hidden layers and two neurons per hidden layer. Table 2-1 defines the corresponding symbols.



$$x_i^{(k)} = \sum_{j=1}^{N} y_j^{(k-1)} w_{i,j}^{(k-1)} + b_i^{(k-1)}, \text{ for k} < 1 \qquad (1)$$

$$y_i^{(k)} = activation\_function\left(x_i^{(k)}\right) \qquad (2)$$

Fig. 2-1. Multilayer perceptron ANN.

Table 2-1. ANN symbol definitions.

Symbol	Definition
$x_i^{(k)}$	Input into the i^{th} neuron of layer k
$y_i^{(k)}$	Activation value of the i^{th} neuron of layer k
$w_{i,j}^{(k-1)}$	Weight connection from the j^{th}neuron in the layer $\mathrm{k}-1$ to the i^{th}neuron in the layer k
$b_i^{(k-1)}$	Bias connection to the i^{th}neuron in the layer k
N	Number of neurons in layer $k-1$

During a training phase, the weights and the biases of an ANN are optimized to minimize the error between the ANNs' predictions and the true labels of the input data [18]. Labels are numerical values which can represent a class, regression value, or other types of data. During training, the input data and the labels are passed into the ANN. The input data is propagated through the ANN which then attempts to predict the label for the given input data, in a process referred to as forward propagation. The ANN then updates the weights and biases, in a process referred to as gradient descent. During gradient descent, a loss function is used to calculate the error between the predicted value and the label. The backpropagation algorithm [19] is used to find the partial derivatives of the weights with respect to the loss function. The partial derivatives of the weights are multiplied by a scaling factor, referred to as the learning rate, and then subtracted from the original weight values to update the weights.

An ANN can be trained for multiple epochs. Each epoch, the training dataset is passed into the ANN in batches. The number of training samples in a batch is referred to as the batch size. During training, a separate set of data, referred to as validation data, can be used to assess how well the ANN performs on data it has not trained on [20]. The validation data can be used to fine-tune the hyperparameters of the ANN, such as the learning rate, number of training epochs, and number of neurons per layer in the ANN. After the training process is complete, the ANN is used to perform predictions on data it has not seen before, referred to as test data.

There are many different types of ANNs, such as convolutional neural networks (CNNs) [21], recurrent ANNs [22], and transformer ANNs [23]. Different types of ANNs can be used to solve different types of problems. For example, CNNs are commonly used for image classification and transformer ANNs are commonly used for text classification. More complex ANN architectures can contain millions of trainable parameters. Different types of ANNs have different structures and connections between the neurons and weights. However, many ANNs build upon the ideas of a basic multilayer perceptron ANN and follow a similar training process.

2.2 Overfitting in Artificial Neural Networks

Overfitting is a phenomenon which occurs when an ANN “overfits” to the training data [5]. The ANN learns the distinct characteristics and noise of the training dataset instead of learning a general pattern to solve the problem. As a result, the ANN performs well on the training data, however, the ANN does not generalize well to novel data. The accuracy for the training data is high, whereas the accuracy for the test data is low. The loss per epoch for the validation data increases throughout training. Overfitting is most likely to occur when there is a small training dataset or when the ANN has a very large number of parameters. Fig. 2-2 illustrates an example of overfitting occurring on training data with two classes, where the red line represents a decision boundary created by the ANN.

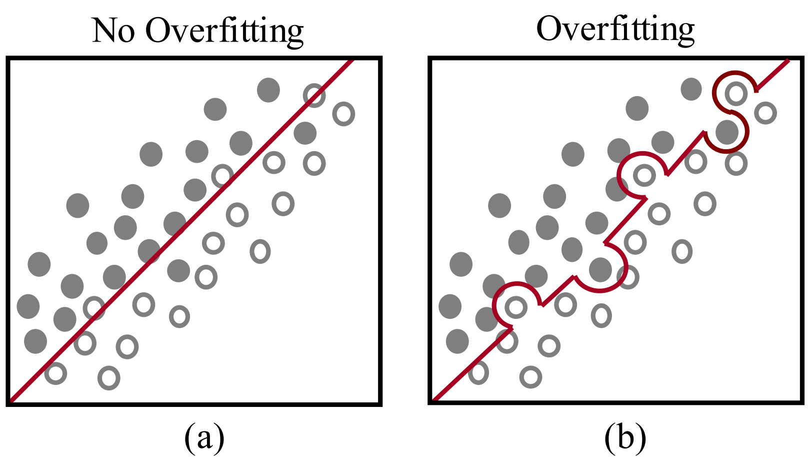


Fig. 2-2. (a) No overfitting versus (b) overfitting.

2.3 Common Techniques to Reduce Overfitting

Overfitting can be improved by increasing the size of the training dataset. However, it can be time-consuming and expensive to collect more data. Therefore, techniques have been developed to reduce overfitting. Common techniques include data augmentation, DO, early stopping, NI, regularization, and weight constraints [5]. Table 2-2 provides a description of each technique.

Table 2-2. Techniques to reduce overfitting.

Technique	Description
Data augmentation	Data augmentation involves performing transformations on the training data to increase the size of the training dataset [24]. For example, data augmentation performed on images could involve cropping, rotating, and adjusting the contrast of the images.
Dropout	DO randomly turns off a specified percent of neurons each iteration during training [7]. DO simulates the effect of training multiple models and then taking the average of the models.
Early stopping	Early stopping is when the training phase is ended before overfitting begins [6]. Overfitting is more likely to occur when an ANN is trained for a long time.
Noise injection	NI involves injecting noise into the ANN [9]. NI has a similar effect to DO. Additional information on NI is provided in Section 2.4.
L1 and L2 regularization	Regularization involves adding a term to the loss function [8]. There are two main regularization techniques: L1-regularization and L2-regularization. L1-regularization adds the sum of the weights, multiplied by a scaling factor, to the loss function. L2- regularization adds the sum of the weights squared, multiplied by a scaling factor, to the loss function. Regularization penalizes large weights and prevents the ANN from focusing too much on one feature.
Weight constraints	Weight constraints can be added to prevent the weights from increasing past a threshold value. Adding weight constraints has a similar effect to regularization.

2.4 Noise Injection to Reduce Overfitting

Researchers have investigated injecting noise into ANNs to improve generalizability. NI adds randomness to an ANN during training, distorting the data, making it difficult for the ANN to overfit. NI can prevent co-adaptation, which causes overfitting. Co-adaptation occurs when neurons learn to make up for errors made by other neurons to improve the accuracy of the training data [25]. NI has been found to perform better than other techniques, such as weight decay and early stopping [9]. NI can make an ANN more resistant to input perturbations [26] and is a form of regularization [26] [27]. NI has also been found to improve the detection of adversarial examples [28], [29]. Adversarial examples are input examples that have been slightly modified, intentionally causing an ANN to misclassify them.

Various NI methods have been proposed, including injecting the noise into the input data [30]–[32], hidden layers [28], [29], [33]–[35], output layer [31], weights [31], [36], and loss function [37]. Noise can be injected additively or multiplicatively. The most common form of NI is Gaussian noise injection (GNI), which uses Gaussian noise [32]–[37]. Recently, adaptive techniques have been proposed to calculate the variance of the Gaussian noise throughout training [33]–[35]. These techniques use the variance of the weights or neurons' inputs.

2.5 Chaotic Injection to Reduce Overfitting

Several researchers have proposed injecting chaotic values into ANNs, as opposed to noise. Chaotic values are bounded, yet non-repeating [38]. Injecting non-repeating values may allow an ANN to search a larger solution space and improve its ability to escape local minimums. As well, chaotic strange attractors have been found in the biological brain [10]–[13]. Modelling an ANN to mimic the behaviour of the biological brain may improve its performance. Additional information on chaotic strange attractors can be found in section 2.6.

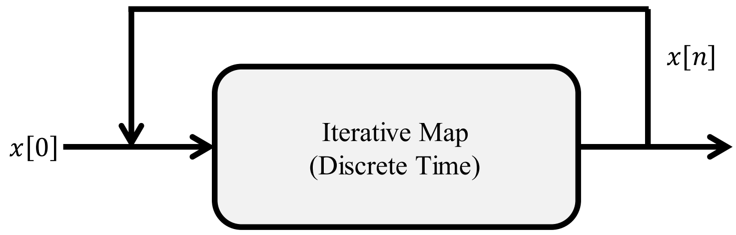
Several CI methods have been proposed. In [16], the neuron's input into the sigmoid activation function [39] is multiplied by a chaotic value produced by a modified version of the logistic map. In [14], the chaotic values are injected into the weight updates during backpropagation and into the sigmoid activation function’s temperature coefficient. Three chaotic strange attractors were tested: the logistic map, the Mackey–Glass equations, and the Lorentz attractor. In [17], the effects of adding chaotic values to the weight updates during backpropagation are analyzed. The logistic map was used to generate the chaotic values. Lastly, in [15], the chaotic values are added to the weight updates during backpropagation. The tent map was used to generate the chaotic values. In these studies, adding chaotic values was found to improve the performance and reduce the convergence times of the ANNs.

Limitations to the previous studies include small datasets and ANN models. Previous research has primarily focused on injecting the chaotic values into the weight updates during backpropagation [14], [15], [17]. There is limited research assessing injecting the chaotic values into the neurons during forward propagation. Only the sigmoid activation function has been tested when injecting chaotic values into the neurons [14], [16]. Also, note that chaotic values have been used in the particle swarm optimization and simulated-annealing algorithms [40]. However, no significant improvements were found when using chaotic values instead of noise.

2.6 Chaos Theory

Chaos is a behaviour that can arise in dynamical systems [41]. Dynamical systems are systems which can exhibit different types of behaviour depending on the parameters of the system. The outputs of a dynamical system exhibiting chaotic behaviour are bounded between a set of values and non-repeating. A small change in the initial conditions of the system will lead to different sequences of outputs. The outputs may appear to be unpredictable and random, however, they are deterministic.

There are two main types of systems which can exhibit chaotic behaviour: iterative maps and differential equations [42]. An iterative map is a function or set of functions used to model discrete-time systems. The outputs from the functions are saved and used as inputs into the functions in the following time-step. Differential equations are used to model continuous-time systems. The outputs of the system can be found given the system's differential equations and initial conditions. Iterative maps directly provide the outputs of the system, whereas differential equations must be solved using analytical or numerical methods to find the outputs of the system, as shown in Fig. 2-3.



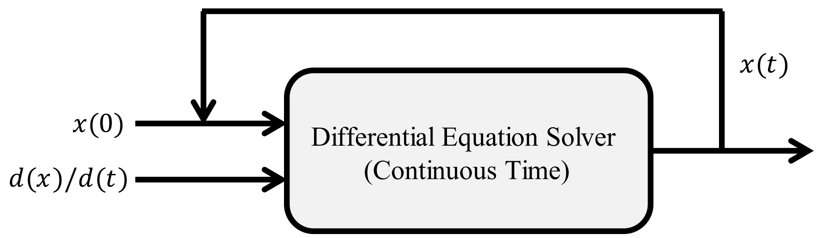
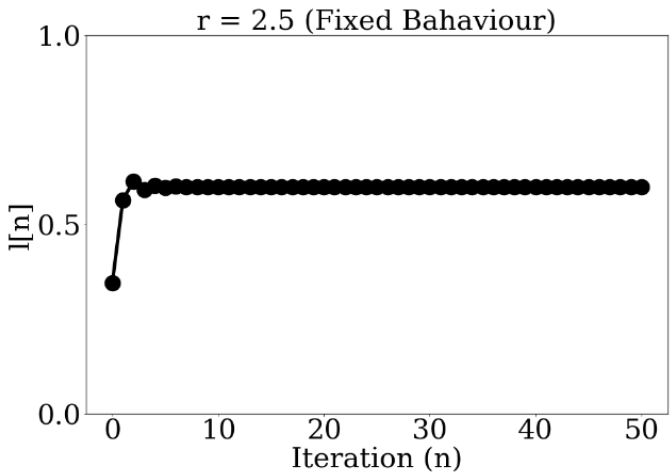


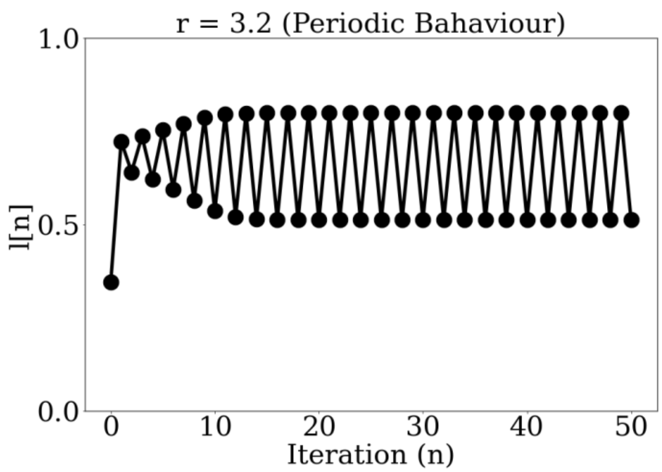
Fig. 2-3. (a) Iterative maps versus (b) differential equations.

In dynamical systems, the parameters which control the behaviour of the system are referred to as bifurcation parameters [41]. These parameters cause the system to converge to either fixed, periodic, cyclic, or chaotic behaviour. When the system's variables are initialized between a given range of values, the system will converge to the state determined by the bifurcation parameters. The state which the system settles into is called the attractor; if the state is chaotic, it is referred to as a chaotic strange attractor. The set of initial values which allow the system to converge to the given state are called the basin of attractors. The system may fluctuate between various values for a given number of iterations before settling into its state; these values are referred to as transient values. Fig. 2-4 illustrates the outputs of the logistic map with different bifurcation values, where r is the bifurcation parameter and n is the iteration number. The logistic map is defined by Equation (3).

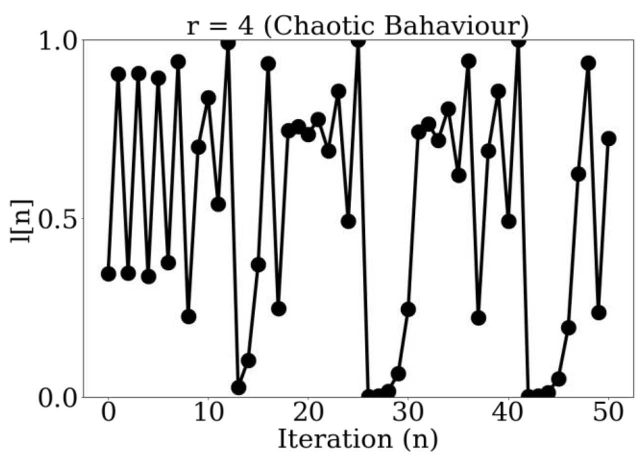
$$l[n+1] \ = r\,(l[n])(1-l[n]) \tag{3}$$



(a)



(b)



(c)

Fig. 2-4. Logistic map with different bifurcation parameters.

A bifurcation diagram can be used to show how different bifurcation parameters affect a dynamical system [38]. The bifurcation diagram plots the value or values which the system has converged to versus the bifurcation parameter. Additionally, a Lyapunov exponent diagram can be used to show the Lyapunov exponents for different bifurcation parameters. The Lyapunov exponent is a measure of how fast two close initial trajectories diverge. A Lyapunov exponent greater than zero is a characteristic of chaos.

Fig. 2-5 illustrates (a) the bifurcation diagram and (b) the Lyapunov exponent diagram for the tent map. The tent map is defined by Equation (4), where μ is the bifurcation parameter. When the bifurcation parameter is between 1.0 and 2.0, exclusive, the system converges to chaotic behaviour. The Lyapunov exponents for these bifurcation parameters are greater than zero. Note that the upper and lower bounds of the chaotic values depend on the bifurcation parameter. As shown in the bifurcation diagram, when the bifurcation parameter is set to 1.5, the chaotic values are bound between [0.35, 0.75]. When the bifurcation parameter is set to 1.99, the chaotic values are bound between [0, 1].

$$t[n+1] = \mu\big(min(t[n], 1 - t[n])\big) = \begin{cases} \mu(t[n]), & t[n] < 0.5 \\ \mu(1 - t[n]), & t[n] \geq 0.5 \end{cases} \tag{4}$$

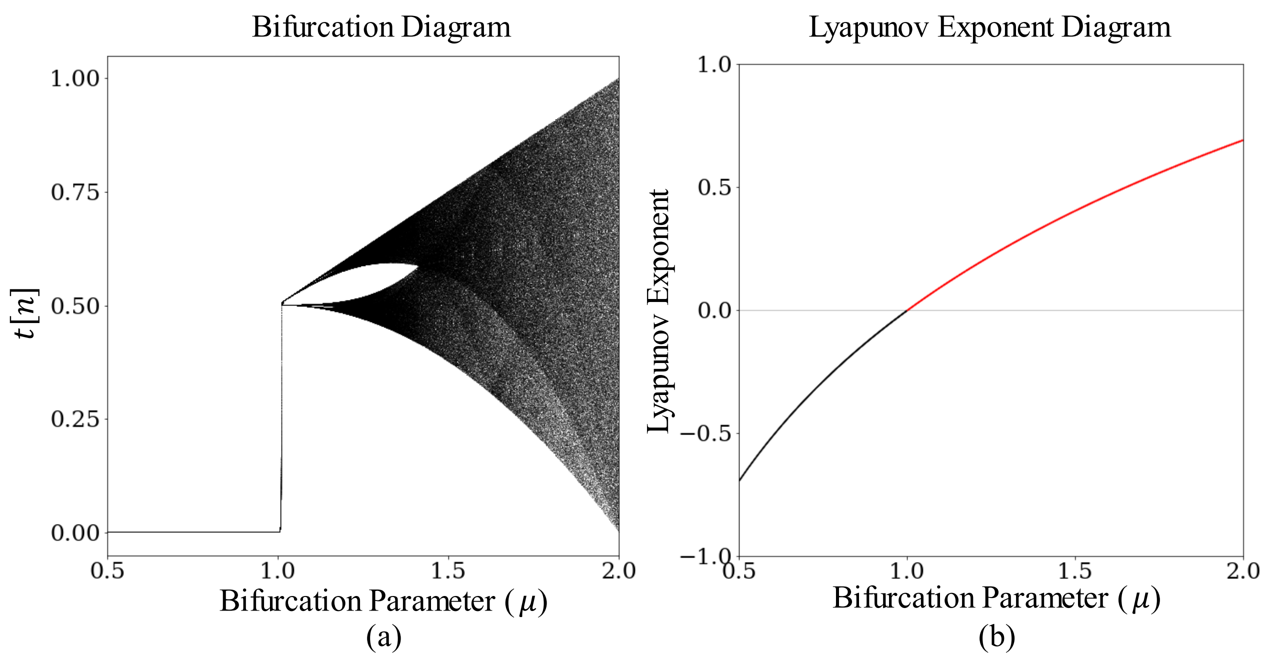
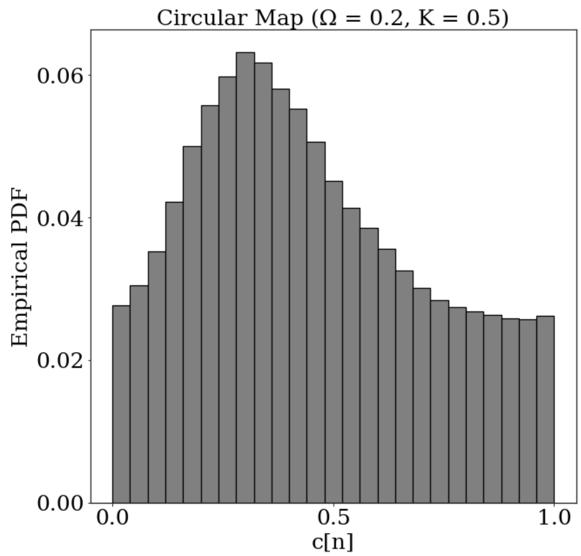


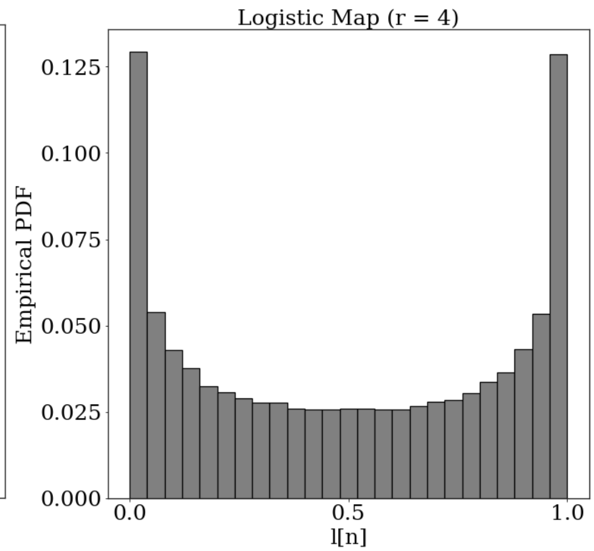
Fig. 2-5. Bifurcation diagram and Lyapunov exponent diagram of the tent map.

Different chaotic strange attractors have different probability density functions (PDFs). Given a specific bifurcation parameter, the sequence of chaotic outputs will follow a unique PDF [40]. Fig. 2-6 shows different chaotic strange attractors and their empirical PDFs. The circular map is defined by Equation (5), where K and Ω are the bifurcation parameters.

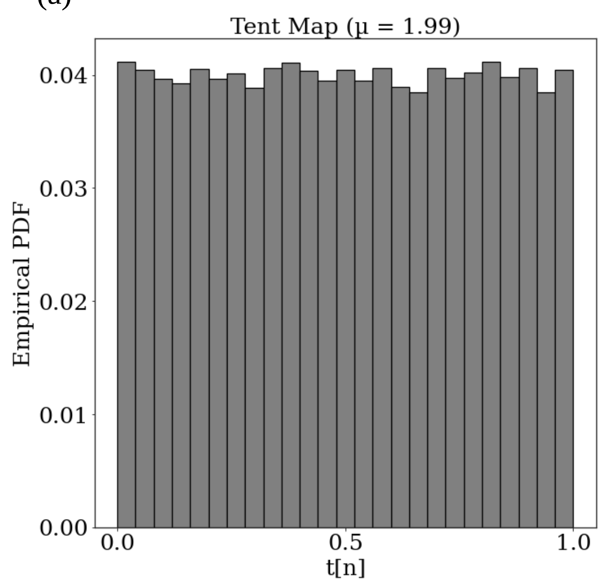
$$c[n+1] = \left(c[n] + \Omega - \frac{K}{2\pi}\sin(2\pi c[n])\right) mod 1 \qquad (5)$$



(a)



(b)



(c)

Fig. 2-6. The PDF of different chaotic iterative maps.

3 Implementation

3.1 Selection of the Attractor

The tent map was selected to generate the chaotic values. The tent map is an iterative map defined by Equation (4), where t represents the tent map value, n represents the time-step, and μ represents the bifurcation parameter. A bifurcation parameter of 1.99 was selected. When the bifurcation parameter is set to 1.99, the tent map becomes a chaotic strange attractor generating a sequence of pseudo-random values between 0 and 1. The tent map was selected for several reasons:

1) The tent map produces a uniform distribution of values between 0 and 1, whereas other iterative maps' PDFs tend to be skewed towards certain values, as shown in Fig. 2-6. Injecting chaotic values which follow a uniform distribution may perform better than other distributions because it allows the neurons to search a broader solution space. Other iterative maps primarily output chaotic values centered around the distribution's peak, potentially narrowing the ANN's search space.
2) The tent map can be computed quickly. The tent map produces the outputs directly, unlike differential equations which must be solved either numerically or analytically.
3) The tent map only contains one variable. Some chaotic strange attractors contain multiple variables. The outputs of the chaotic strange attractor must be saved to be used as input into the attractor in the following time-step. Therefore, memory may be a concern if a large number of neurons are using the CI and multiple variables must be saved.

3.2 Initialization and Setup

In our research, we will assess the feasibility of injecting the chaotic values into neurons in the hidden layers during forward propagation. Each neuron in a layer using the CI has its own tent map. The initial values of the tent maps are initialized randomly between 0 and 1. The tent maps are then iterated for 1000 iterations before training to remove transient values. Each batch iteration during training, the tent maps are iterated to generate a new chaotic value. The chaotic values are saved to be used as input into the tent maps in the following iteration. The chaotic values are multiplied by a scaling factor and then injected into their respective neuron. The scaling factor is an adaptive parameter which changes each epoch. The scaling factors are initialized before training begins. The CI only occurs on the training data. Table 3-1 provides an overview of the algorithm.

Table 3-1. CI algorithm pseudo-code.

	Algorithm
1	Initialize the tent maps and remove transient values, initialize the scaling values
2	For each epoch during training:
3	Update the scaling value
4	For each batch in the epoch:
5	Update and save the state of the chaotic values
6	Scale the chaotic values
7	Inject the scaled chaotic values into the neurons during forward propagation
8	Perform backpropagation and update the weights

3.3 Offset and Adaptive Scaling

Before a chaotic value is injected into a neuron, an offset value is added, and it is scaled. An offset value, β, of 0.5 is added to shift the chaotic value from the range [0,1] to the range [-0.5, 0.5]. The value is then multiplied by a scaling factor, α, to either amplify or diminish its effect. α is an adaptive parameter which starts at zero and is logarithmically increased each epoch during training. α initially dampens the chaotic values allowing the ANN to converge. α is then increased to allow the ANN to explore a larger solution space and prevent overfitting. Equation (6) shows how the scaled chaotic value, $s[n]$, is calculated.

$$s[n] \ = \alpha(t[n] - \beta)\ , \beta = 0.5 \tag{6}$$

The values of α are calculated and initialized into an array before training begins. The α values are calculated in two steps. Firstly, the α value for each epoch is calculated using Equation (7), where ω is a hyperparameter which controls the growth rate of the log function. The $epoch\_num$ ranges from [0, number of epochs-1]. Secondly, the array of α values is rescaled between [0, α_max] using Equation (8), where α_max is a hyperparameter which sets the maximum value of α. Fig. 3-1 shows an example of the α values throughout training, when ω is set to 25, α_max is set to 5, and the number of epochs is set to 50.

$$\alpha\_array[epoch\_num] = log(\,\omega \times epoch\_num + 1) \tag{7}$$

$$\alpha\_array[:] = \frac{\alpha\_array[:]}{\max\,(\alpha\_array[:])} \times \alpha\_max \tag{8}$$

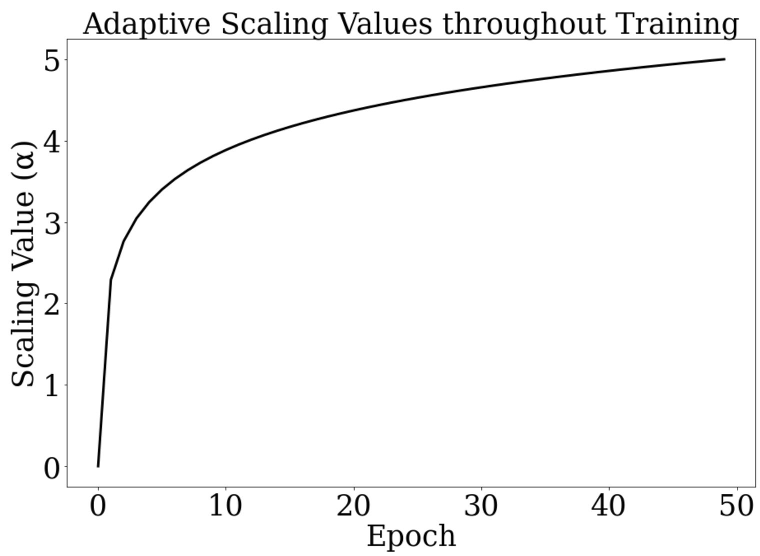


Fig. 3-1. Adaptive scaling parameter.

3.4 Method of Injection

After the chaotic value is scaled, it is injected into the neuron. There are various ways to inject the chaotic value into the neuron. The chaotic value can be added or multiplied into the neuron, before or after the activation function. Fig. 3-2 illustrates how the various injection methods can affect the rectified linear unit (ReLU) activation function. The ReLU activation function [43], defined by Equation (9), was selected because it is commonly used in practice and it has a simple derivative, as shown in Equation (10).

$$ReLU(x) = \begin{cases} 0, & if\ x < 0 \\ x, & if\ x > 0 \end{cases} \tag{9}$$

$$\frac{d\big(ReLU(x)\big)}{dx} = \begin{cases} 0, & if\ x < \ 0 \\ 1, & if\ x > 0 \end{cases} \tag{10}$$

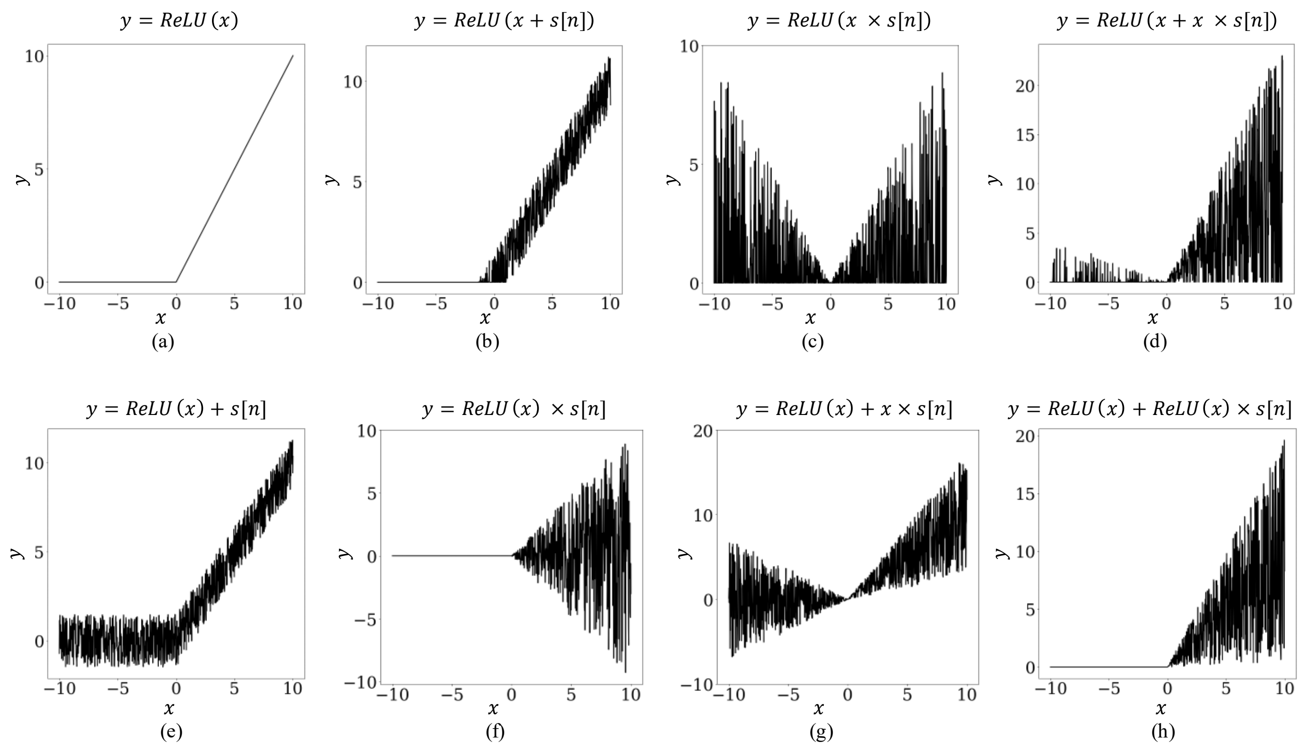


Fig. 3-2. Various injection methods.

Option (h) from Fig. 3-2 was selected for the final implementation, which is defined by Equation (11). In this injection method, the additive and multiplicative injection approaches are combined. The chaotic value is first multiplied by the activation value to scale its effect; it will have a larger effect on neurons with a large activation value and it will not affect neurons with an activation value less than zero. This method is similar to the adaptive methods proposed by [33]–[35], where either the weights or neurons' inputs were used to determine the variance of the Gaussian noise. Fig. 3-3 (a) illustrates the setup for the CI. Table 3-2 defines the corresponding symbols. Fig. 3-3 (b) illustrates a multilayer perceptron ANN using the CI. The ANN contains two hidden layers, with two neurons per hidden layer.

$$y = ReLU(x) + ReLU(x) \times s[n] \tag{11}$$

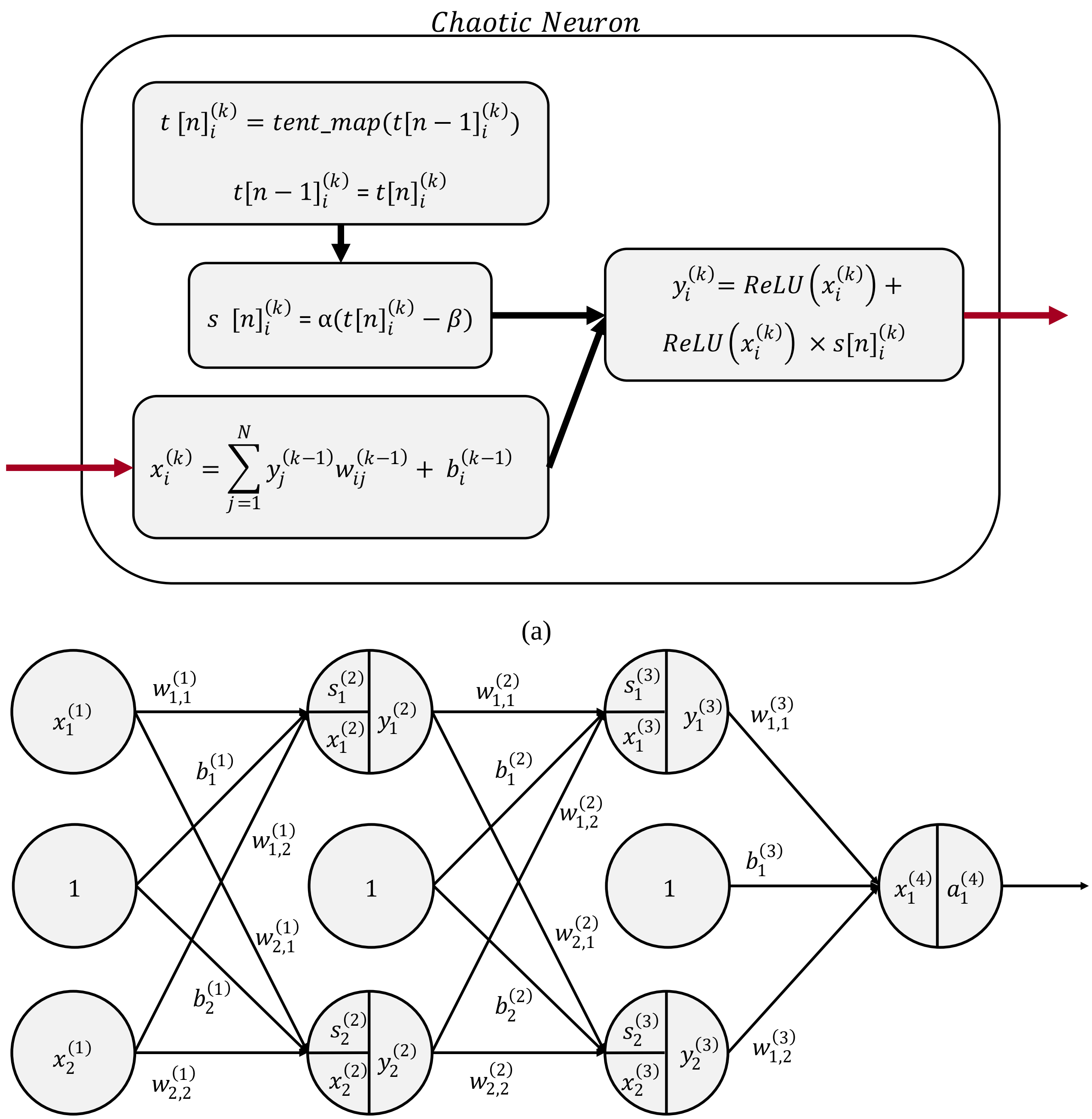


Fig. 3-3. (a) Neuron using the CI and (b) ANN using the CI.

Table 3-2. CI ANN symbol definitions.

Symbol	Definition
$x_i^{(k)}$	Input into the i^{th} neuron of layer k
$y_i^{(k)}$	Output of the i^{th} neuron of layer k
$t[n]_i^{(k)}$	Tent map value of the i^{th} neuron of layer k at iteration n
$s[n]_i^{(k)}$	Scaled tent map value of the i^{th} neuron of layer k at iteration n
$a_i^{(k)}$	Activation value of the i^{th}neuron in the output layer
$w_{i,j}^{(k-1)}$	Weight connection from the j^{th}neuron in the layer $k-1$ to the i^{th}neuron in the layer k
$b_i^{(k-1)}$	Bias connection to the i^{th}neuron in the layer k
N	Number of neurons in layer $\mathrm{k}-1$

3.5 Effects on Backpropagation

During backpropagation, the CI affects the derivative of neurons with a positive activation value. The CI does not affect the derivative of neurons with a negative activation value. Equation (12) shows the derivative of a neuron using the CI. If a neuron has a positive activation value, the derivative is $1+\ s[n]$. The extent to which $s[n]$ affects the weights depends on the tent map scaling factor (α) and the overall structure of the ANN. The CI adds pseudo-randomness to the ANN, causing the weights to be slightly increased or decreased throughout training.

$$\frac{dy_i^{(k)}}{dx_i^{(k)}} = \frac{d\left(ReLU\left(x_i^{(k)}\right) +\ ReLU\left(x_i^{(k)}\right)\ \times s[n]_i^{(k)}\right)}{dx_i^{(k)}} = \begin{cases} 0, & if\ x_i^{(k)} < 0 \\ 1 +\ s[n]_i^{(k)}, & if\ x_i^{(k)} > 0 \end{cases} \tag{12}$$

4 Testing

The code used for implementation and testing was developed using Python (version 3.7.13) [44]. The code was developed in Google Colab Pro+ [45]. All code can be found in Appendix A.

4.1 Datasets and Data Preprocessing

Three open-source classification datasets were used for testing: Fashion-MNIST (Modified National Institute of Standards and Technology database) [46], CIFAR-10 (Canadian Institute for Advanced Research) [47], and Stanford Cars [48]. The datasets were obtained and preprocessed using the TorchVision library (version 0.13.0+cu113) [49], which is a Python library used for image processing and computer vision tasks.

4.1.1 Fashion-MNIST

The Fashion-MNIST dataset contains 70,000 greyscale images. The images are of the size 28x28 pixels. The dataset contains 10 classes, consisting of the following articles of clothing: t-shirts, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags, and ankle boots. Fig. 4-1 shows sample images from the dataset. Prior to training, the pixel values of the images were normalized between [-1,1] and the images were flattened to the size 784x1 pixels.

Fig. 4-1. Sample images from the Fashion-MNIST dataset.

4.1.2 CIFAR-10

The CIFAR-10 dataset contains 60,000 RGB images. The images are of the size 3x32x32 pixels. The dataset contains 10 classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Fig. 4-2 shows sample images from the dataset. Prior to training, the pixel values of the images were normalized between [-1,1].

Fig. 4-2. Sample images from the CIFAR-10 dataset.

4.1.3 Stanford Cars

The Stanford Cars dataset contains 16,185 RGB images of varying sizes. The dataset contains 196 classes of different types of cars. Fig. 4-3 shows sample images from the dataset. The images were resized to 224x224 pixels. The pixel values were rescaled between 0 and 1. The RGB channels were normalized using the following parameters:

mean=[0.485, 0.456, 0.406], standard deviation=[0.229, 0.224, 0.225].

Fig. 4-3. Sample images from the Stanford Cars dataset.

4.2 Models

The models were developed using the PyTorch machine learning library (version 1.12.0+cu113) [50]. The CI was compared against the baseline ANNs, DO, GNI without adaptive scaling, and CI without adaptive scaling. The CI was compared against DO because it is commonly used in practice. The CI was compared against GNI due to their similar mechanisms of action. The Gaussian noise used a mean of zero and variance of one. The GNI used the same injection method as the CI, as described in Section 3.4. The CI was tested with and without adaptive scaling to assess its effects. When adaptive scaling wasn't used, the α value was set to a constant value throughout training. The CI, DO, and GNI were used in the hidden dense layers of the ANNs. The hidden dense layers were selected for the CI because DO is commonly used in these layers to prevent overfitting. The PyTorch cross-entropy loss function [51] was used as the loss function for all models.

4.2.1 Multilayer Perceptron Model

The Fashion-MNIST dataset was tested using a multilayer perception ANN. The ANN contained 2 hidden layers. Each hidden layer contained 512 neurons. Fig. 4-4 illustrates the model.

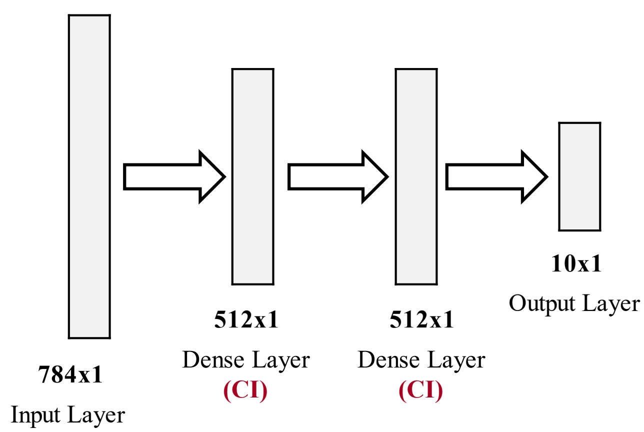


Fig. 4-4. Multilayer perceptron ANN used for testing the Fashion-MNIST dataset.

4.2.2 Convolutional Model

The CIFAR-10 dataset was tested using a CNN model. The CNN consisted of three convolutional layers, three 2D-max-pooling layers, followed by two hidden dense layers, and the output layer. The convolutional layers used a filter size of 3x3 and a padding size of one. The first convolutional layer used 16 filters, the second convolutional layer used 32 filters, and the last convolutional layer used 64 filters. A 2D-max-pooling layer followed each convolutional layer. The 2D-max-pooling layers used a kernel size of two and a stride of two. The two dense layers each contained 512 neurons. Fig. 4-5 illustrates the model.

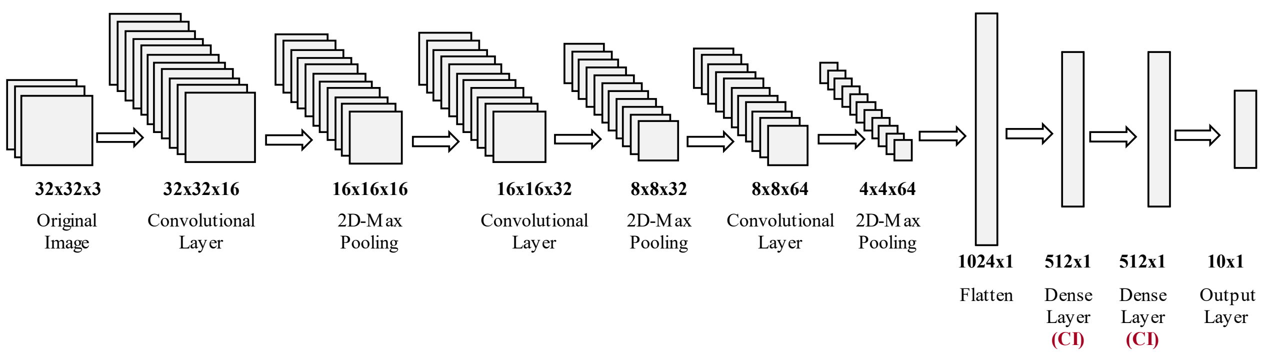


Fig. 4-5. CNN used for testing the CIFAR-10 dataset.

4.2.3 EfficientNet-B7 Model

The Stanford Cars dataset was tested using the EfficientNet-B7 model [52]. EfficientNet-B7 is a state-of-the-art CNN architecture, containing ~66 million trainable parameters. The output layer of the model was removed and replaced by two dense layers containing 512 neurons and an output layer containing 196 neurons. The weights of the model were pre-trained on ImageNet [53], which is a large dataset, containing thousands of classes. The pre-trained weights were loaded into the model prior to training. The Adam optimizer [54] was used during training with an initial learning rate of 0.0001. Fig. 4-6 illustrates the model.

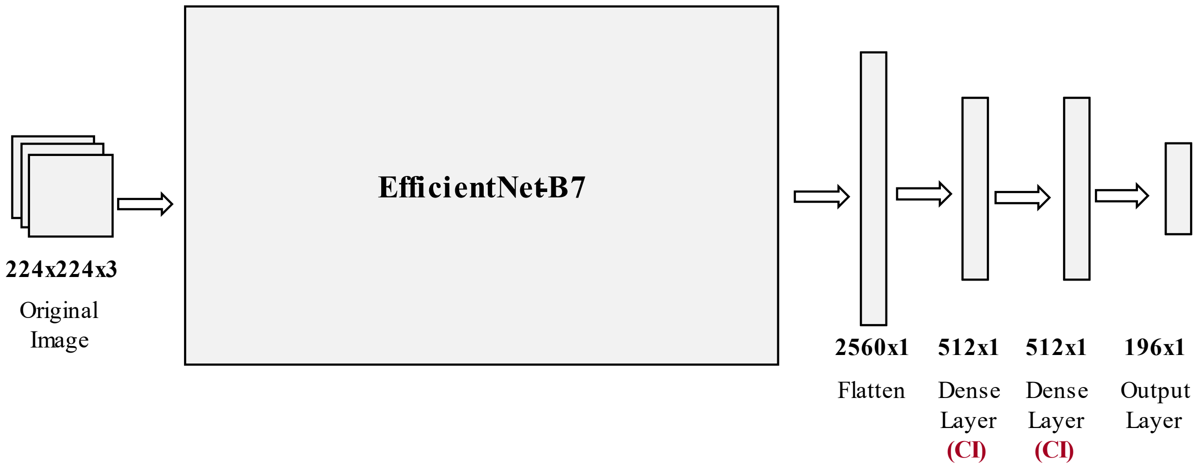


Fig. 4-6. EfficientNet-B7 model used for testing the Stanford Cars dataset.

4.3 Cross-Validation

Ten-fold cross-validation was used for fine-tuning the models and selecting the hyperparameters [20]. In ten-fold cross-validation, the training data is randomly separated into ten folds. Ten training runs are performed. For each training run, a different fold is selected as the validation data. A portion of the data was excluded from cross-validation to be used as the test data. The Scikit-Learn library (version 1.0.2) [55] was used for implementing the cross-validation. Fig. 4-7 illustrates the ten-fold cross-validation.

During each training run, the accuracy and loss per epoch for the training and validation data were saved. As well, the model was saved at the epoch when the validation data had the lowest loss value. After the ten training runs were completed, the average accuracy and loss per epoch for the training and validation data were found to produce the overall results. As well, the ten saved models were used to get the average performance metrics of the test data.

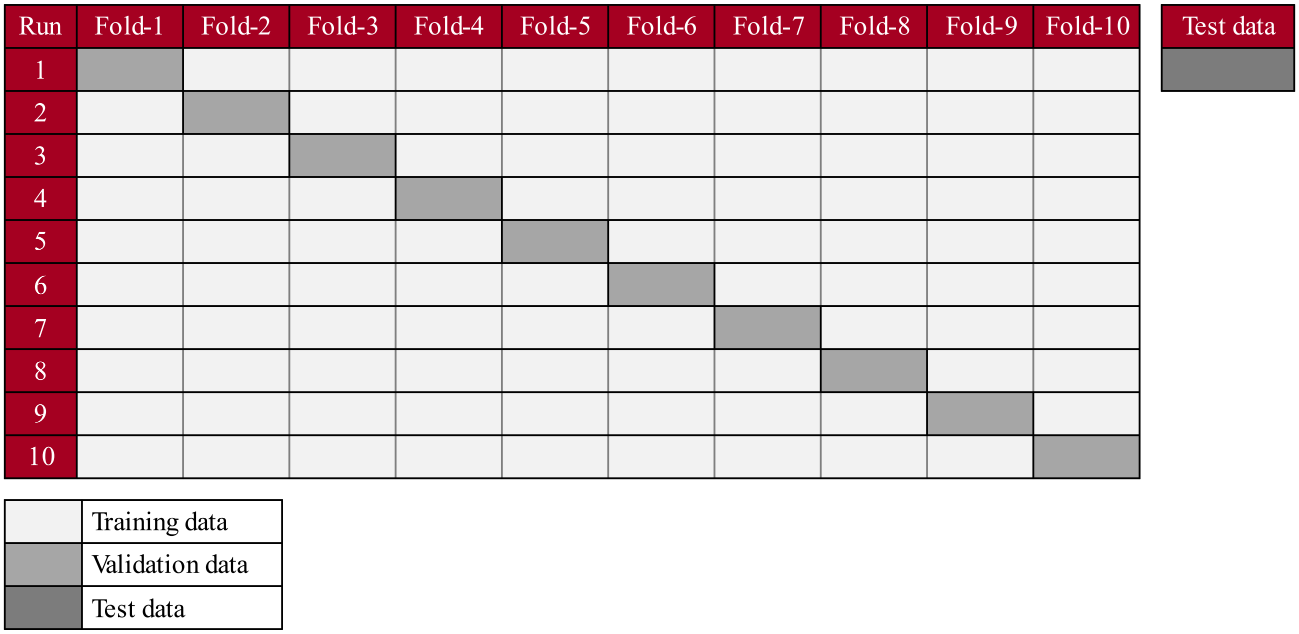


Fig. 4-7. Ten-fold cross-validation setup.

The training and test data were well-balanced with respect to their classes for all datasets. The datasets were separated into training and test data using the train/test splits created by the authors of the datasets. For the Fashion-MNIST dataset, 60,000 images were used for cross-validation and 10,000 images were used for testing. For the CIFAR-10 dataset, 50,000 images were used for cross-validation and 10,000 images were used for testing. For the Stanford Cars dataset, 8144 images were used for cross-validation, and 8041 images were used for testing.

4.4 Hyperparameter Selection

The training and validation data were used to finetune the hyperparameters, such as the learning rate, the batch size, and the CI parameters. Table 4-1 shows the hyperparameter values selected. The models were trained for a set number of epochs. Early stopping was not used because it is a technique to prevent overfitting; we wanted to assess how well the CI performed without using other overfitting techniques.

Table 4-1. Hyperparameter Selection.

Parameter	Fashion-MNIST	CIFAR-10	Stanford Cars
Batch size	100	100	20
CI bias value (β)	0.5	0.5	0.5
CI bifurcation parameter (μ)	1.99	1.99	1.99
CI scale value (constant α)	3.0	5.0	5.5
CI maximum scale value (α_max)	3.0	5.5	6.5
CI scale growth rate (ω)	25	25	25
DO value	0.5	0.7	0.6
GNI bias value (β)	0	0	0
GNI scale value (constant α)	0.9	1.5	1.5
Learning rate	0.05	0.05	0.0001
Number of epochs	50	50	20
Number of test images	10,000	10,000	8041
Number of weights and biases	669,706	816,170	65,461,396
Number of training images	60,000	50,000	8144
Optimizer	None	None	Adam

5 Results

Two methods were used to assess the CI's performance. Firstly, the training convergences of the models were analyzed. The average accuracies and losses per epoch were plotted, and the runtimes of the models were compared. Secondly, the average results of the test data were analyzed, using the following performance metrics: accuracy (ACC), F1-score (F1), negative-predictive value (NPV), positive-predictive value (PPV), sensitivity (SN), and specificity (SP).

5.1 Training Convergence

5.1.1 Accuracy and Loss Per Epoch

Fig. 5-1, Fig. 5-2, and Fig. 5-3 show (a) the loss per epoch for the training data, (b) the loss per epoch for the validation data, (c) the accuracy per epoch for the training data, and (d) the accuracy per epoch for the validation data for the three datasets. Table 5-1, Table 5-2, and Table 5-3 show the accuracy and loss at the end of training for the three datasets. The accuracy is the number of correctly classified true-positive samples versus the total number of samples, and the loss is the cross-entropy loss function.

The baseline ANNs produce the highest accuracy and lowest loss for the training data. However, the baseline ANNs produce the lowest accuracy and highest loss for the validation data. As well, the validation data's loss for the baseline ANNs increases throughout training. These characteristics indicate the baseline ANNs are overfitting to the training data. When the ANNs are trained for a long time, they begin learning the distinct characteristics and noise of the training data. As a result, the ANNs' performance on the validation data begins decreasing, causing the loss to increase.

The CI, DO and GNI methods reduce overfitting. These methods add randomness to the ANNs, making it difficult for the ANNs to overfit to the training data. As a result, the accuracy is lower and the loss is higher for the training data compared to the baseline ANNs. However, the accuracy is higher and the loss is lower for the validation data. These methods allow the ANNs to generalize better to novel data.

The CI with adaptive scaling reduces the final loss of the validation data compared to the baseline ANNs by 21.85%, 65.42%, and 29.77% for the Fashion-MNIST, CIFAR-10, and Stanford Cars datasets, respectively. Likewise, the CI with adaptive scaling increases the final accuracy of the validation data by 0.53%, 1.70%, and 5.55% for the Fashion-MNIST, CIFAR-10, and Stanford Cars datasets, respectively.

The baseline ANNs converge the fastest. The CI, DO, and GNI models take longer to converge because they decrease the accuracy of the training data, in exchange for better generalizability. The CI with adaptive scaling converges faster than DO and GNI, as shown on the Stanford Cars dataset. The adaptive scaling method initially dampens the chaotic values allowing the ANNs to converge, and then amplifies the chaotic values allowing the ANNs to explore a larger solution space.

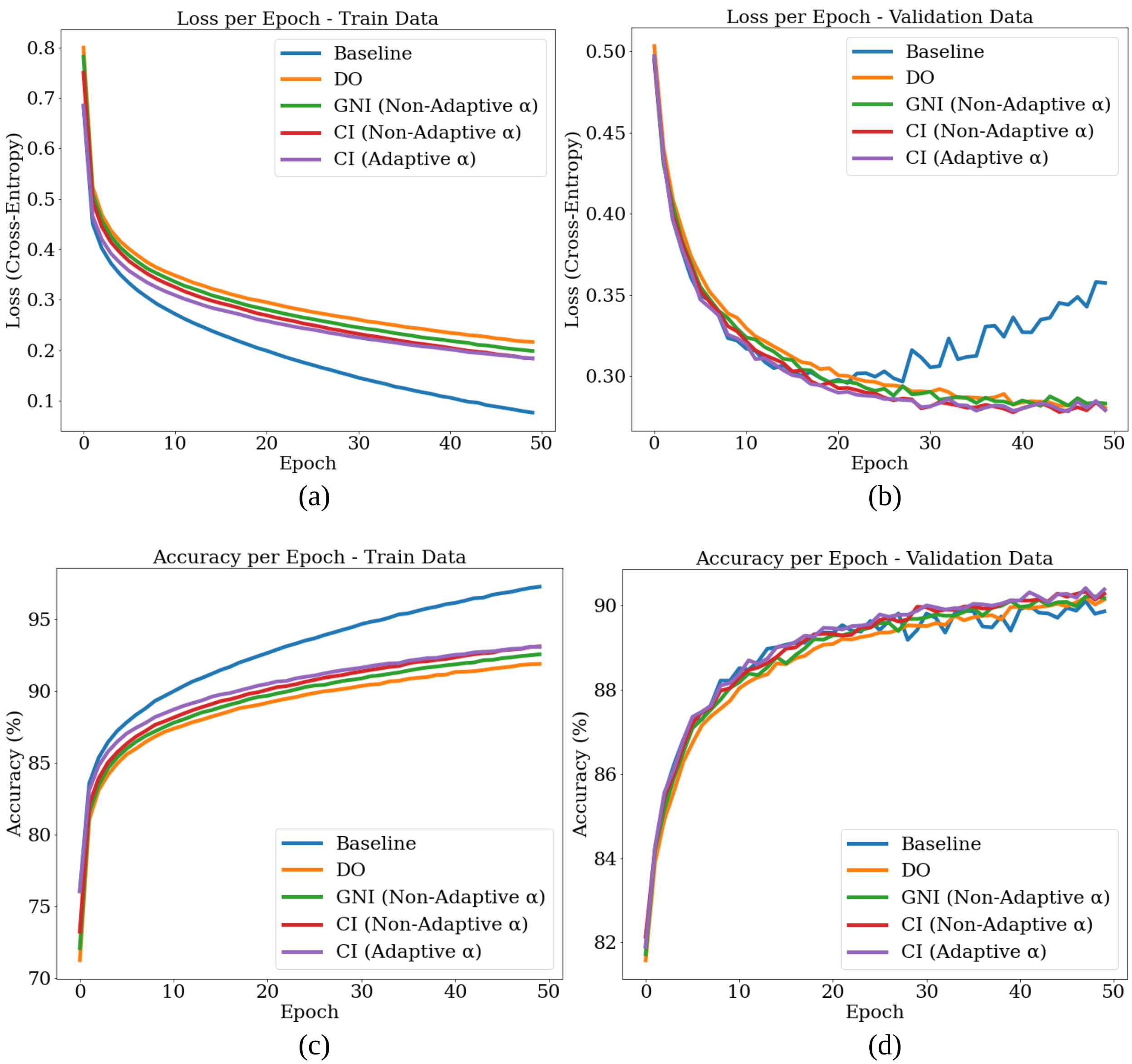
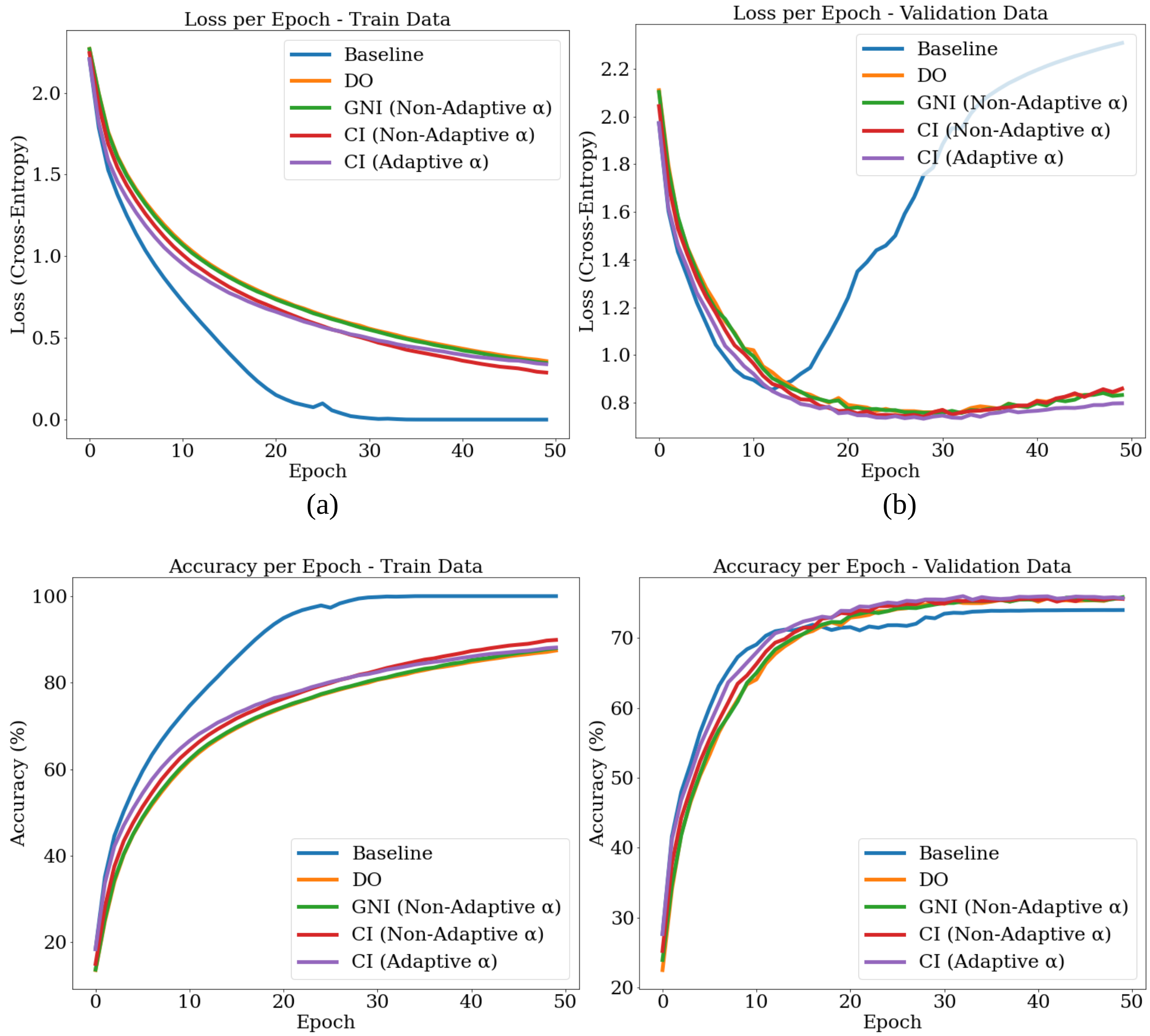


Fig. 5-1. Accuracy and loss per epoch for the Fashion-MNIST dataset.

Table 5-1. Accuracy and loss for the Fashion-MNIST dataset.

Metric	**Base**		**CI (α constant)**		**CI (α adaptive)**		**DO**		**GNI (α constant)**	
	Train	**Valid**	**Train**	**Valid**	**Train**	**Valid**	**Train**	**Valid**	**Train**	**Valid**
Accuracy (%)	97.28	89.86	93.13	90.28	93.09	90.39	91.91	90.14	92.58	90.18
Loss	0.077	0.357	0.184	0.279	0.184	0.279	0.217	0.281	0.199	0.283



(a) (b)

(c) (d)

Fig. 5-2. Accuracy and loss per epoch for the CIFAR-10 dataset.

Table 5-2. Accuracy and loss for the CIFAR-10 dataset.

Metric	Base		CI (α constant)		CI (α adaptive)		DO		GNI (α constant)	
	Train	Valid	Train	Valid	Train	Valid	Train	Valid	Train	Valid
Accuracy (%)	100.0	73.99	89.88	75.58	88.12	75.69	87.43	75.59	87.97	75.83
Loss	0.000	2.308	0.288	0.860	0.339	0.798	0.358	0.859	0.347	0.833

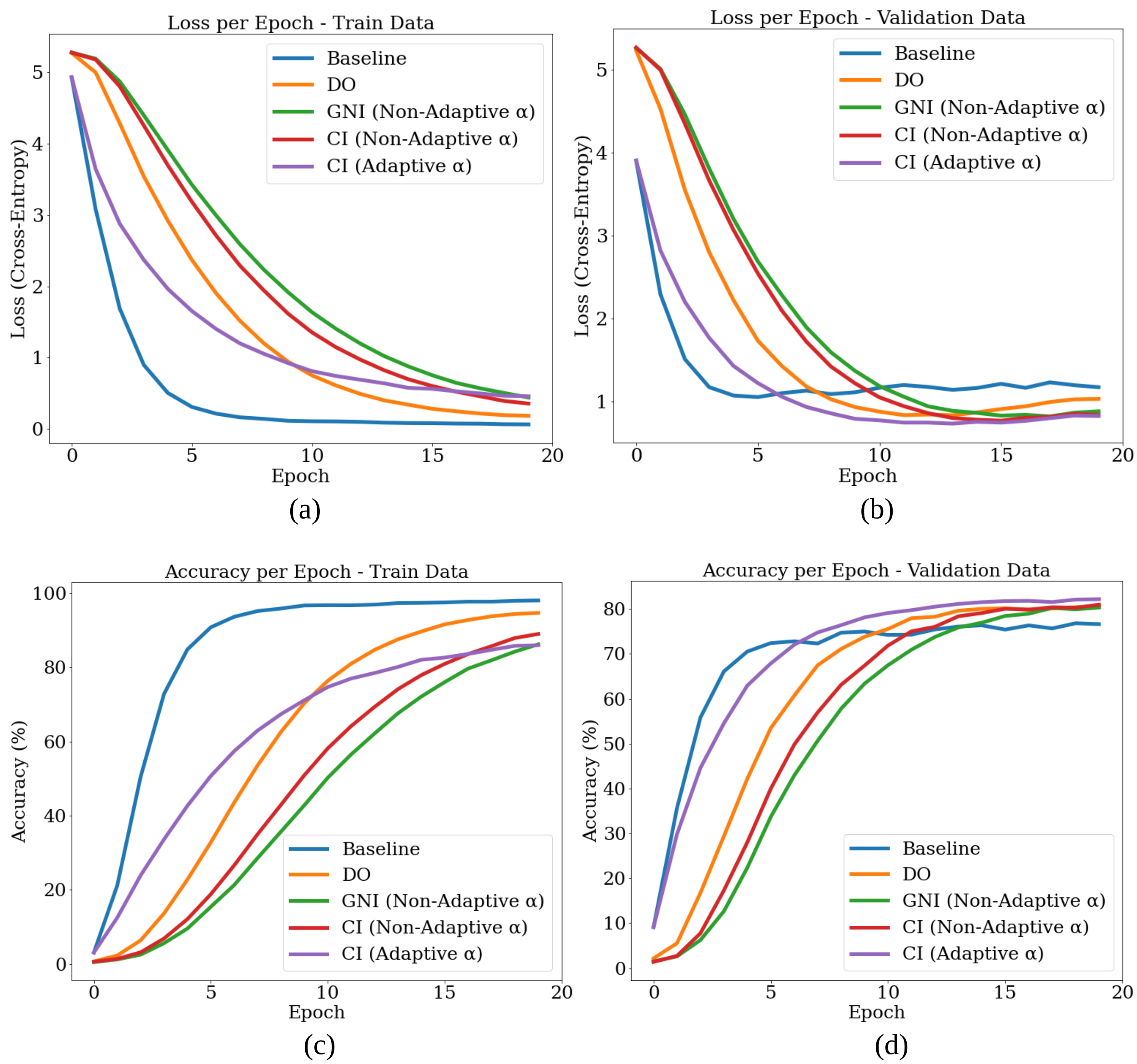


Fig. 5-3. Accuracy and loss per epoch for the Stanford Cars dataset.

Table 5-3. Accuracy and loss for the Stanford Cars dataset.

Metric	Base		CI (α constant)		CI (α adaptive)		DO		GNI (α constant)	
	Train	Valid	Train	Valid	Train	Valid	Train	Valid	Train	Valid
Accuracy (%)	98.08	76.61	89.03	80.94	86.00	82.16	94.71	80.49	86.23	80.29
Loss	0.065	1.169	0.357	0.834	0.459	0.821	0.186	1.028	0.442	0.878

5.1.2 Runtime

Table 5-4 shows the average runtimes of the models. Empirically, the results show that the CI does not have a significant impact on the runtime. The CI adds three computations to the training algorithm: (1) generating the chaotic values, (2) scaling the chaotic values, and (3) injecting the chaotic values into the neurons. Note that the models were run in Google Colab Pro+, therefore the runtimes may vary based on GPU (Graphics Processing Unit) availability.

Table 5-4. Average runtimes (s) of the models.

Dataset	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
Fashion-MNIST	501.71	512.89	509.59	493.92	529.22
CIFAR-10	600.32	595.25	839.93	581.96	630.97
Stanford Cars	6146.46	5954.48	6711.64	4108.73	4355.58

5.2 Performance Metrics

The test data was used to assess the models' performances. The models were assessed using the following metrics: (1) accuracy, (2) F1-score, (3) negative-predictive value, (4) positive-predictive value, (5) sensitivity, and (6) specificity. The metrics were calculated for each class using the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) samples. Table 5-5 provides the corresponding formulas. After the metrics were found for each class, the averages were taken.

Table 5-5. Performance metric formulas.

Metric	Formula
Sensitivity	$SN = \frac{TP}{(TP + FN)}$ (13)
Specificity	$SP = \frac{TN}{(TN + FP)}$ (14)
Positive-Predictive Value	$PPV = \frac{TP}{(TP + FP)}$ (15)
Negative-Predictive Value	$NPV = \frac{TN}{(TN + FN)}$ (16)
F1-Score	$F1 = \frac{2 \times SN \times PPV}{(SN + PPV)}$ (17)
Accuracy	$ACC = \frac{TP + TN}{(TP + FP + TN + FN)}$ (18)

Table 5-6, Table 5-7, and Table 5-8 show the results of the test data for the three datasets. Appendix B shows the results of the validation data for the three datasets. The CI with adaptive scaling achieves the highest performance metrics on the test data, with results similar to DO and GNI. The CI's improvements over the baseline ANNs range between 0.04% and 7.36% for various performance metrics. The CI's improvements over DO and GNI range between 0.01% and 2.40% for various performance metrics. The greatest improvements are seen on the F1-score, sensitivity, and positive-predictive value metrics.

The results indicate the CI is more effective on difficult datasets and large ANN models. The Stanford Cars dataset contains the smallest number of training samples and uses the largest ANN model, containing ~66 million trainable parameters. Whereas, the Fashion-MNIST dataset contains the largest number of training samples and uses the smallest ANN model, containing less than one million trainable parameters. Therefore, the Stanford Cars model is more likely to suffer from overfitting than the Fashion-MNIST model. Consequently, the Stanford Cars model likely benefits more from the CI than the Fashion-MNIST model. Additional testing on large ANN models could be performed to confirm these findings.

Table 5-6. Performance metrics of the test data for the Fashion-MNIST dataset.

Metric	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
ACC	97.79	**97.92**	**97.92**	97.88	97.90
F1	88.95	89.58	**89.59**	89.39	89.48
NPV	98.77	**98.85**	**98.85**	98.83	98.83
PPV	89.06	**89.63**	89.62	89.42	89.51
SN	88.95	89.60	**89.62**	89.42	89.50
SP	98.77	98.84	**98.85**	98.82	98.83

Table 5-7. Performance metrics of the test data for the CIFAR-10 dataset.

Metric	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
ACC	94.32	95.04	**95.11**	94.98	95.02
F1	71.39	75.17	**75.50**	74.84	75.11
NPV	96.86	97.25	**97.29**	97.21	97.24
PPV	72.10	75.38	**75.77**	74.94	75.32
SN	71.59	75.20	**75.54**	74.90	75.10
SP	96.84	97.24	**97.28**	97.21	97.23

Table 5-8. Performance metrics of the test data for the Stanford Cars dataset.

Metric	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
ACC	99.74	99.79	**99.82**	99.79	99.80
F1	74.45	79.40	**81.78**	79.39	79.80
NPV	99.87	99.90	**99.91**	99.90	99.90
PPV	76.94	81.13	**83.05**	80.97	81.31
SN	74.53	79.54	**81.89**	79.49	79.95
SP	99.87	99.90	**99.91**	99.90	99.90

6 Conclusion

6.1 Thesis Conclusions

This thesis presented a method to inject chaotic values into the neurons of an ANN. In Chapter 3, the injection method is presented. The chaotic values are generated using the tent map, which is a chaotic strange attractor when the bifurcation parameter is set to 1.99. Each neuron in a layer using the CI has its own tent map. The chaotic values are scaled and then injected into the neurons using a combined additive and multiplicative approach. An adaptive scaling parameter was developed to increase the effect of the chaotic values throughout training. In Chapter 4, the models used for testing were presented. A variety of different datasets and models were used to assess the performance of the CI. Three datasets were used for testing: Fashion-MNIST, CIFAR-10, and Stanford Cars. In Chapter 5, the results were presented. The CI was able to reduce overfitting and improve the performance of the ANNs. The CI achieves higher accuracy than the baseline ANN on all datasets. The CI converges faster than DO and GNI using the adaptive scaling method.

6.2 Future Work

Recommendations for future work are listed below:

1) A method could be developed to determine the optimal maximum scaling value, α_max. If α_max is too large, the ANN will not learn. If α_max is too small, it will not have an effect on the ANN. α_max is not a trainable parameter because the ANN may learn to set it to zero to increase the accuracy of the training data, however, then overfitting would not be improved.

2) Additional testing could be performed. Firstly, the CI was only injected into the dense layers of the ANNs. Further testing is required to determine its effects on other layers, such as convolutional layers. Secondly, the CI could be tested on other large ANN models, such as BERT (Bidirectional Encoder Representations from Transformers) [56]. Our results indicate the CI has the greatest impact on large ANN models. Lastly, the CI could be compared against other adaptive injection methods [33]–[35] which have recently been proposed.
3) Additional research could be performed to determine the optimal distribution of values used for the injection. In this research, the tent map was used which follows a uniform distribution. Previous work has primarily focused on NI using a Gaussian distribution [32]–[37]. An adaptive method could be developed to determine the optimal distribution of values for each individual neuron throughout training.

6.3 Thesis Contributions

In this thesis, several contributions have been made to this area of research:

1) A method for injecting chaotic values or noise into an ANN was developed, which combines the previous additive and multiplicative injection methods.
2) An adaptive method was developed for scaling the chaotic values. This method uses a logarithmic function to scale the values, allowing the ANN to initially converge and then explore a larger solution space. This method can be applied to the CI and NI.
3) The effectiveness of using a chaotic strange attractor to generate sequences of values to inject into the neurons of an ANN was assessed. The CI successfully reduces overfitting and improves the performance of ANNs.

References

[1] J. Schmidhuber, "Deep Learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015, doi: 10.1016/j.neunet.2014.09.003.

[2] M. R. Bachute and J. M. Subhedar, "Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms," *Mach. Learn. with Appl.*, vol. 6, p. 100164, 2021, doi: https://doi.org/10.1016/j.mlwa.2021.100164.

[3] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights Imaging*, vol. 9, no. 4, pp. 611–629, 2018, doi: 10.1007/s13244-018-0639-9.

[4] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *J. Netw. Comput. Appl.*, vol. 153, p. 102526, 2020, doi: https://doi.org/10.1016/j.jnca.2019.102526.

[5] X. Ying, "An Overview of Overfitting and its Solutions," *J. Phys. Conf. Ser.*, vol. 1168, no. 2, pp. 0–6, 2019, doi: 10.1088/1742-6596/1168/2/022022.

[6] L. Prechelt, "Early Stopping — But When? BT - Neural Networks: Tricks of the Trade: Second Edition," G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 53–67.

[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[8] A. Y. Ng, "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance," in

Proceedings of the Twenty-First International Conference on Machine Learning, 2004, p. 78, doi: 10.1145/1015330.1015435.

[9] R. M. Zur, Y. Jiang, L. L. Pesce, and K. Drukker, “Noise injection for training artificial neural networks: A comparison with weight decay and early stopping,” *Med. Phys.*, vol. 36, no. 10, pp. 4810–4818, 2009, doi: 10.1118/1.3213517.

[10] H. Korn and P. Faure, “Is there chaos in the brain? II. Experimental evidence and related models,” *Comptes Rendus - Biol.*, vol. 326, no. 9, pp. 787–840, 2003, doi: 10.1016/j.crvi.2003.09.011.

[11] M. Small, H. P. C. Robinson, I. C. Kleppe, and C. K. Tse, “Uncovering bifurcation patterns in cortical synapses,” *J. Math. Biol.*, vol. 61, no. 4, pp. 501–526, 2010, doi: 10.1007/s00285-009-0312-5.

[12] A. Celletti and A. E. P. Villa, “Determination of chaotic attractors in the rat brain,” *J. Stat. Phys.*, vol. 84, no. 5–6, pp. 1379–1385, 1996, doi: 10.1007/BF02174137.

[13] G. Rodríguez-Bermúdez and P. J. García-Laencina, “Analysis of EEG signals using nonlinear dynamics and chaos: A review,” *Appl. Math. Inf. Sci.*, vol. 9, no. 5, pp. 2309–2321, 2015, doi: 10.12785/amis/090512.

[14] S. U. Ahmed, M. Shahjahan, and K. Murase, “Injecting chaos in feedforward neural networks,” *Neural Process. Lett.*, vol. 34, no. 1, pp. 87–100, 2011, doi: 10.1007/s11063-011-9185-x.

[15] A. Azamimi, Y. Uwate, and Y. Nishio, “An Analysis of Chaotic Noise Injected to Backpropagation Algorithm in Feedforward Neural Network,” in *International Workshop*

on Vision, Communications and Circuits, 2008, vol. 1, no. 1, pp. 70–73.

[16] Y. Uwate and Y. Nishio, "Chaotically Oscillating Sigmoid Function in Feedforward Neural Network," in *Proceedings of International Symposium on Nonlinear Theory and its Applications (NOLTA'06)*, 2006, pp. 215–218.

[17] H. Zhang, Y. Zhang, D. Xu, and X. Liu, "Deterministic convergence of chaos injection-based gradient method for training feedforward neural networks," *Cogn. Neurodyn.*, vol. 9, no. 3, pp. 331–340, 2015, doi: 10.1007/s11571-014-9323-z.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, doi: 10.1038/323533a0.

[20] S. Raschka, "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning," 2018. [Online]. Available: http://arxiv.org/abs/1811.12808.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, vol. 25, [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[22] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network," *Phys. D Nonlinear Phenom.*, vol. 404, no. March, pp. 1–43, 2020, doi: 10.1016/j.physd.2019.132306.

[23] A. Vaswani *et al.*, “Attention is All You Need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010.

[24] C. Shorten and T. M. Khoshgoftaar, “A survey on Image Data Augmentation for Deep Learning,” *J. Big Data*, vol. 6, no. 1, 2019, doi: 10.1186/s40537-019-0197-0.

[25] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *CoRR*, vol. abs/1207.0, 2012, [Online]. Available: http://arxiv.org/abs/1207.0580.

[26] A. Camuto, M. Willetts, C. Holmes, and S. Roberts, “Explicit regularisation in Gaussian noise injections,” *NeurIPS*, no. NeurIPS, pp. 1–12, 2020, doi: https://doi.org/10.48550/arXiv.2007.07368.

[27] C. M. Bishop, “Training with Noise is Equivalent to Tikhonov Regularization,” *Neural Comput.*, vol. 7, no. 1, pp. 108–116, Jan. 1995, doi: 10.1162/neco.1995.7.1.108.

[28] S. Wang, W. Liu, and C.-H. Chang, “Detecting Adversarial Examples for Deep Neural Networks via Layer Directed Discriminative Noise Injection,” in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2019, pp. 1–6, doi: 10.1109/AsianHOST47458.2019.9006702.

[29] A. Liu, X. Liu, H. Yu, C. Zhang, Q. Liu, and D. Tao, “Training Robust Deep Neural Networks via Adversarial Noise Propagation,” *IEEE Trans. Image Process.*, vol. 30, pp. 5769–5781, 2021, doi: 10.1109/TIP.2021.3082317.

[30] X. Meng, C. Liu, Z. Zhang, and D. Wang, “Noisy training for deep neural networks,” in *2014 IEEE China Summit & International Conference on Signal and Information*

Processing (ChinaSIP), 2014, pp. 16–20, doi: 10.1109/ChinaSIP.2014.6889193.

[31] G. An, "The Effects of Adding Noise During Backpropagation Training on a Generalization Performance," *Neural Comput.*, vol. 8, no. 3, pp. 643–674, 1996, doi: 10.1162/neco.1996.8.3.643.

[32] S. Zheng, Y. Song, T. Leung, and I. Goodfellow, "Improving the robustness of deep neural networks via stability training," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-Decem, pp. 4480–4488, 2016, doi: 10.1109/CVPR.2016.485.

[33] B. Khalfaoui, J. Boyd, and J.-P. Vert, "ASNI: Adaptive Structured Noise Injection for shallow and deep neural networks," pp. 1–16, 2019, [Online]. Available: http://arxiv.org/abs/1909.09819.

[34] Y. Li and F. Liu, "Adaptive Gaussian Noise Injection Regularization for Neural Networks," in *Advances in Neural Networks – ISNN 2020*, 2020, pp. 176–189, doi: https://doi.org/10.1007/978-3-030-64221-1_16.

[35] Y. X. Marcus Tan, Y. Elovici, and A. Binder, "Adaptive Noise Injection for Training Stochastic Student Networks from Deterministic Teachers," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 7587–7594, doi: 10.1109/ICPR48806.2021.9412385.

[36] L. Adilova, N. Paul, and P. Schlicht, "Introducing Noise in Decentralized Training of Neural Networks," 2018, doi: 10.1007/978-3-030-13453-2_21.

[37] Z. Cai, C. Peng, and S. Du, "Jitter: Random Jittering Loss Function," *Proc. Int. Jt. Conf. Neural Networks*, vol. 2021-July, no. 1, 2021, doi: 10.1109/IJCNN52387.2021.9533462.

[38] J. C. Sprott and R. H. Abraham, “Strange Attractors: Creating Patterns in Chaos,” *Am. J. Phys.*, vol. 63, no. 5, pp. 477–477, 1995, doi: 10.1119/1.17885.

[39] B. Ding, H. Qian, and J. Zhou, “Activation functions and their characteristics in deep neural networks,” in *2018 Chinese Control And Decision Conference (CCDC)*, 2018, pp. 1836–1841, doi: 10.1109/CCDC.2018.8407425.

[40] I. Gagnon, A. April, and A. Abran, “An investigation of the effects of chaotic maps on the performance of metaheuristics,” *Eng. Reports*, vol. 3, no. 8, pp. 1–14, 2021, doi: 10.1002/eng2.12369.

[41] S. H. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*, 2nd ed. CRC Press, 2015.

[42] A. S. Mikhailov and A. Y. Loskutov, *Foundations of Synergetics II: Chaos and Noise*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[43] V. Nair and G. E. Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines,” in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 2010, pp. 807–814.

[44] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[45] Google, “Google Colab,” 2022. https://colab.research.google.com/?utm_source=scs-index (accessed Jul. 01, 2022).

[46] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms,” pp. 1–6, 2017, [Online]. Available:

http://arxiv.org/abs/1708.07747.

[47] Alex Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[48] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, “3D object representations for fine-grained categorization,” *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 554–561, 2013, doi: 10.1109/ICCVW.2013.77.

[49] S. Marcel and Y. Rodriguez, “Torchvision the Machine-Vision Package of Torch,” in *Proceedings of the 18th ACM International Conference on Multimedia*, 2010, pp. 1485–1488, doi: 10.1145/1873951.1874254.

[50] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” *Adv. Neural Inf. Process. Syst.*, vol. 32, no. NeurIPS, 2019.

[51] PyTorch, “CrossEntropyLoss,” 2022. https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (accessed Jul. 01, 2022).

[52] M. Tan and Q. V. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” *36th Int. Conf. Mach. Learn. ICML 2019*, vol. 2019-June, pp. 10691–10700, 2019.

[53] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, “ImageNet: a Large-Scale Hierarchical Image Database,” in *IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255, doi: 10.1109/CVPR.2009.5206848.

[54] D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *Int. Conf. Learn.*

Represent., Dec. 2014.

[55] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011, [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html.

[56] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” *NAACL HLT 2019 - 2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf.*, vol. 1, no. Mlm, pp. 4171–4186, 2019.

Appendix A

Appendix A provides the code used for all the experiments. The code consists of 6 modules, as listed below:

1) imports.ipynb
2) main.ipynb
3) load_dataset.ipynb
4) create_model.ipynb
5) train_model.ipynb
6) display_results.ipynb

Note: To run the code, the user should update the "selected_dataset" variable in main.ipynb to select either the Fashion-MNIST, CIFAR-10, or Stanford Cars dataset. The user must also update the number of epochs, CI, DO, and GNI hyperparameters accordingly. Lastly, the user must update the paths to where their code, models, and results are stored.

A.1 Imports

```
#******************************************************************************************
# MODULE:    Imports
# PURPOSE:   Imports the libraries and .ipynb files, sets the seed values, and
#            connects to the GPUs.
# AUTHOR:    Siobhan Reid
# VERSION:   2
# DATE:      August 1, 2022
#******************************************************************************************

# Download PyTorch if not already installed
# !pip install torch
# !pip install torchvision

# PyTorch imports
```

```
import torch
import torch.nn as nn
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.autograd import Variable
from torch.utils.data import DataLoader
from torchvision.utils import make_grid

# Other imports
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold
import autoreload
from matplotlib import pyplot as plt
import numpy as np
import timeit
import os
import random
import math
import gc
import seaborn as sns
import pandas as pd

# Set Pandas display options
pd.set_option('display.max_columns', None)

# Set seed values
def set_seed() :
  seed = 0
  torch.manual_seed(seed)
  torch.cuda.manual_seed_all(seed)
  torch.cuda.manual_seed(seed)
  np.random.seed(seed)
  random.seed(seed)
  torch.backends.cudnn.deterministic = True
  torch.backends.cudnn.benchmark = False

set_seed()

# Connect to GPU
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("Running on the GPU")
else:
    device = torch.device("cpu")
```

```
    print("Running on the CPU")
torch.cuda.device_count()

# Run .ipynb files
%run load_dataset.ipynb
%run create_model.ipynb
%run train_model.ipynb
%run display_results.ipynb
```

A.2 Main

```
#*******************************************************************************************
# MODULE:    Main
# PURPOSE:   Loads the dataset, trains the ANN models, and displays the results.
# AUTHOR:    Siobhan Reid
# VERSION:   2
# DATE:      August 1, 2022
#*******************************************************************************************

# Connect to Google Drive
from google.colab import drive
drive.mount('/content/drive/', force_remount=True)
%cd "/content/drive/My Drive/ColabNotebooks/v2/code"

# Import libraries
%run imports.ipynb

# Select the dataset
fashion_mnist = 1
cifar_10 = 2
stanford_cars = 3

load_dataset = Load_Dataset()
dataset_download_path = "Downloads"
selected_dataset = fashion_mnist

# Download and load the dataset
if (selected_dataset == fashion_mnist):
  train_data, test_data, test_gen, input_size, num_classes, batch_size = load_dataset.load_fashio
n_mnist_dataset(dataset_download_path)
  save_model_path = "/content/drive/My Drive/ColabNotebooks/v2/models/fashion_mnist/"
  train_data_shape = np.zeros((len(train_data), input_size))
if (selected_dataset == cifar_10):
```

```
  train_data, test_data, test_gen, input_size, num_classes, batch_size = load_dataset.load_cifar_
10_dataset(dataset_download_path)
  save_model_path = "/content/drive/My Drive/ColabNotebooks/v2/models/cifar_10/"
  train_data_shape = np.zeros((len(train_data), input_size[0], input_size[1], input_size[2]))
if (selected_dataset == stanford_cars):
  train_data, test_data, test_gen, input_size, num_classes, batch_size = load_dataset.load_stanfo
rd_cars_dataset(dataset_download_path)
  save_model_path = "/content/drive/My Drive/ColabNotebooks/v2/models/stanford_cars/"
  train_data_shape = np.zeros((len(train_data), input_size[0], input_size[1], input_size[2]))

# Path to where the models and results will be saved
aci_save_model_path = save_model_path + "/aci/"
ci_save_model_path = save_model_path + "/ci/"
base_save_model_path = save_model_path + "/base/"
do_save_model_path = save_model_path + "/do/"
gni_save_model_path = save_model_path + "/gni/"

# Show batch of images from the dataset
load_dataset.show_batch(test_gen)

# Create a k-fold object using Sklearn
# Used to perform the 10-fold cross-validation; separates the training data into ten folds
num_kfolds = 10
rand_state = 1
kf = KFold(n_splits = num_kfolds, random_state = rand_state, shuffle = True)

# Specify which ANN model should be used
# The Fashion-MNIST dataset uses a MLP model
# The CIFAR-10 dataset uses a CNN model
# The Stanford Cars dataset uses the EfficientNet-B7 model
if (selected_dataset == stanford_cars) :
  use_optim = True    # Boolean which specifies whether the Adam optimizer should be used
  use_conv = False    # Boolean which specifies whether the CNN model should be used
  use_eff_net = True  # Boolean which specifies whether the Efficient-B7 model should be used
if (selected_dataset == cifar_10) :
  use_optim = False
  use_conv = True
  use_eff_net = False
if (selected_dataset == fashion_mnist) :
  use_optim = False
  use_conv = False
  use_eff_net = False

# Specify the ANN models' hyperparameters depending on whether the MLP, CNN, or EfficientNet-B7
model was selected
```

```
num_epochs = 50                     # Number of training epochs
learning_rate = 0.05                # Learning rate
hidden_size_layer1 = 512            # Size of first dense layer
hidden_size_layer2 = 512            # Size of second dense layer
ci_bifur_val = 1.99                 # Tent map bifurcation parameter
ci_offset_val = 0.5                 # Tent map offset value
do_val = 0.6                        # Dropout value
gni_scale_val = 0.9                 # Gaussian noise scaling value
loss_function = nn.CrossEntropyLoss() # Loss function

# Five different ANN models are trained: baseline ANN, CI with adaptive scaling, CI without adapt
ive scaling, GNI without adaptive scaling, and Dropout
# Boolean values (use_ci, use_do, use_gni, and use adapt_scale) are used to specify which model i
s being trained
# Each model is trained using 10-fold cross-validation

# Train the CI model without adaptive scaling
use_ci = True
use_gni = False
use_do = False
ci_scale_val = 3.0 # Tent map scaling value
use_adapt_scale = False

set_seed()

train_model = Train_Model(input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_
size,
               ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
               do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
               num_epochs, learning_rate, use_optim, loss_function, kf, train_data_shape, train_d
ata)

train_model.kfold_train_model(ci_save_model_path)

# Train the CI model with adaptive scaling
use_ci = True
use_gni = False
use_do = False
ci_scale_val = 3.0
use_adapt_scale = True

set_seed()

train_model = Train_Model(input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_
size,
```

```
              ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
              do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
              num_epochs, learning_rate, use_optim, loss_function, kf, train_data_shape, train_d
ata)

train_model.kfold_train_model(aci_save_model_path)

# Train the baseline model
use_ci = False
use_gni = False
use_do = False
use_adapt_scale = False

set_seed()

train_model = Train_Model(input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_
size,
              ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
              do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
              num_epochs, learning_rate, use_optim, loss_function, kf, train_data_shape, train_d
ata)

train_model.kfold_train_model(base_save_model_path)

# Train the Dropout model
use_ci = False
use_gni = False
use_do = True
use_adapt_scale = False

set_seed()

train_model = Train_Model(input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_
size,
              ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
              do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
              num_epochs, learning_rate, use_optim, loss_function, kf, train_data_shape, train_d
ata)

train_model.kfold_train_model(do_save_model_path)

# Train the GNI model
use_ci = False
use_gni = True
use_do = False
```

```
use_adapt_scale = False

set_seed()

train_model = Train_Model(input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_
size,
               ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
               do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
               num_epochs, learning_rate, use_optim, loss_function, kf, train_data_shape, train_d
ata)

train_model.kfold_train_model(gni_save_model_path)

# After all models are trained, the results are displayed
use_ci = False
use_gni = False
use_do = False
use_adapt_scale = False

display_results = Display_Results(input_size, hidden_size_layer1, hidden_size_layer2, num_classes
, batch_size,
               ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
               do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
               num_epochs, kf, train_data_shape, train_data, test_gen, save_model_path)

# Plot the loss and accuracy per epoch for the training and validation data
aci_plots_df = pd.read_csv(aci_save_model_path + "/plots.csv")
ci_plots_df = pd.read_csv(ci_save_model_path + "/plots.csv")
gni_plots_df = pd.read_csv(gni_save_model_path + "/plots.csv")
base_plots_df = pd.read_csv(base_save_model_path + "/plots.csv")
do_plots_df = pd.read_csv(do_save_model_path + "/plots.csv")

display_results.plot_loss_and_acc_per_epoch(aci_plots_df.valid_loss, ci_plots_df.valid_loss, base
_plots_df.valid_loss, do_plots_df.valid_loss, gni_plots_df.valid_loss, 'upper right', "Loss per E
poch - Validation Data", "Loss (Cross-Entropy)")
display_results.plot_loss_and_acc_per_epoch(aci_plots_df.train_loss, ci_plots_df.train_loss, base
_plots_df.train_loss, do_plots_df.train_loss, gni_plots_df.train_loss, 'upper right', "Loss per E
poch - Train Data", "Loss (Cross-Entropy)")
display_results.plot_loss_and_acc_per_epoch(aci_plots_df.valid_acc, ci_plots_df.valid_acc, base_p
lots_df.valid_acc, do_plots_df.valid_acc, gni_plots_df.valid_acc, 'lower right', "Accuracy per Ep
och - Validation Data", "Accuracy (%)")
display_results.plot_loss_and_acc_per_epoch(aci_plots_df.train_acc, ci_plots_df.train_acc, base_p
lots_df.train_acc, do_plots_df.train_acc, gni_plots_df.train_acc, 'lower right', "Accuracy per Ep
och - Train Data", "Accuracy (%)")
```

```
# Display the performance metrics for the validation data
get_valid_results = True
display_results.kfold_display_metrics(get_valid_results)

# Display the performance metrics for the test data
get_valid_results = False
display_results.kfold_display_metrics(get_valid_results)
```

A.3 Load Dataset

```
#*********************************************************************************************
# MODULE:    Load_Dataset
# PURPOSE:   Loads the training and test data for the Fashion-MNIST, CIFAR-10, and Stanford
#            Cars datasets, and preprocesses the images.
# AUTHOR:    Siobhan Reid
# VERSION:   2
# DATE:      August 1, 2022
#*********************************************************************************************

class Load_Dataset():

  #*******************************************************************************************
  # FUNCTION:  show_batch
  # PURPOSE:   Displays a batch of samples images from a data generator.
  # PARAMS:    The data generator.
  # RETURNS:   NA.
  #*******************************************************************************************

  def show_batch(self, gen):
    for images, labels in gen:
        fig, ax = plt.subplots(figsize=(12, 6))
        ax.set_xticks([]); ax.set_yticks([])
        ax.imshow(make_grid(images, nrow=16).permute(1, 2, 0))
        break
    return

  #*******************************************************************************************
  # FUNCTION:  load_cifar_10_dataset
  # PURPOSE:   Downloads the CIFAR-10 dataset and preprocesses the images.
  # PARAMS:    The download path.
  # RETURNS:   The training data, the test data, the test data loaded into a generator,
  #            the image input size, the number of classes, and the selected batch size.
  #*******************************************************************************************
```

```
  def load_cifar_10_dataset(self, dataset_download_path):
    num_classes = 10
    input_size = (3,32,32)
    batch_size = 100
    transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,0.5,0.5), (0.
5,0.5,0.5)),])
    train_data = datasets.CIFAR10(root = dataset_download_path, train=True, download=True, transf
orm=transform)
    test_data = datasets.CIFAR10(root = dataset_download_path, train=False, download=True, transf
orm=transform)
    test_gen = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False)
    return train_data, test_data, test_gen, input_size, num_classes, batch_size

  #*****************************************************************************************
  # FUNCTION:  load_stanford_cars_dataset
  # PURPOSE:   Downloads the Stanford Cars dataset and preprocesses the images.
  # PARAMS:    The download path.
  # RETURNS:   The training data, the test data, the test data loaded into a generator,
  #            the image input size, the number of classes, and the selected batch size.
  #*****************************************************************************************

  def load_stanford_cars_dataset(self, dataset_download_path):
    num_classes = 196
    input_size = (3,224,224)
    batch_size = 20
    transform=transforms.Compose([transforms.ToTensor(), transforms.Resize((224,224)), transforms
.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])
    train_data = datasets.StanfordCars(root = dataset_download_path, split = "train", download =
True, transform=transform)
    test_data = datasets.StanfordCars(root = dataset_download_path, split = "test", download = Tr
ue, transform=transform)
    test_gen = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False)
    return train_data, test_data, test_gen, input_size, num_classes, batch_size

  #*****************************************************************************************
  # FUNCTION:  load_fashion_mnist_dataset
  # PURPOSE:   Downloads the Fashion-MNIST dataset and preprocesses the images.
  # PARAMS:    The download path.
  # RETURNS:   The training data, the test data, the test data loaded into a generator,
  #            the image input size, the number of classes, and the selected batch size.
  #*****************************************************************************************

  def load_fashion_mnist_dataset(self, dataset_download_path):
    num_classes = 10
    input_size = 28*28
```

```
    batch_size = 100
    transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
    train_data = datasets.FashionMNIST(dataset_download_path, download = True, train = True, tran
sform = transform)
    test_data = datasets.FashionMNIST(dataset_download_path, download = True, train = False, tran
sform = transform)
    test_gen = torch.utils.data.DataLoader(test_data, batch_size = batch_size, shuffle = True)
    return train_data, test_data, test_gen, input_size, num_classes, batch_size

  #******************************************************************************************
  # FUNCTION:  seed_worker
  # PURPOSE:   Used to set the seed for the training data generator.
  # PARAMS:    The worker ID for the generator.
  # RETURNS:   NA.
  # REFERENCE: https://discuss.pytorch.org/t/reproducibility-with-all-the-bells-and-
whistles/81097
  #******************************************************************************************

  def seed_worker(self, worker_id):
    worker_seed = torch.initial_seed() % 2**32
    np.random.seed(worker_seed)
    random.seed(worker_seed)
```

A.4 Create Model

```
#******************************************************************************************
# MODULE:    Net
# PURPOSE:   Creates the ANN model.
#            The Fashion-MNIST dataset uses a MLP model.
#            The CIFAR-10 dataset uses a CNN model.
#            The Stanford Cars datset uses the EfficientNet-B7 model.
# AUTHOR:    Siobhan Reid
# VERSION:   2
# DATE:      August 1, 2022
#******************************************************************************************

class Net(nn.Module):

  #******************************************************************************************
  # FUNCTION:  __init__
  # PURPOSE:   Initializes the class variables and ANN layers. Inherits from the PyTorch
  #            class.
  # PARAMS:    The hyperparameters of the ANN (described in main.ipynb).
  # RETURNS:   NA.
```

```
#*****************************************************************************************

def __init__(self, input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_size,

             ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
             do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale, num_epoch
s):

  super(Net, self).__init__()

  self.num_epochs = num_epochs
  self.batch_size = batch_size
  self.hidden_size_layer1 = hidden_size_layer1
  self.hidden_size_layer2 = hidden_size_layer2
  self.use_conv = use_conv
  self.use_eff_net = use_eff_net
  self.do = nn.Dropout(do_val)
  self.relu = nn.ReLU()
  self.use_ci = use_ci
  self.use_gni = use_gni
  self.use_do = use_do
  self.ci_bifur_val = ci_bifur_val
  self.ci_offset_val = ci_offset_val
  self.ci_scale_val = self.init_scale_val(ci_scale_val, num_epochs, use_adapt_scale)
  self.gni_scale_val = self.init_scale_val(gni_scale_val,num_epochs,use_adapt_scale)
  self.layer1_ci_vals = self.init_tent_map((batch_size, hidden_size_layer1))
  self.layer2_ci_vals = self.init_tent_map((batch_size, hidden_size_layer2))

  if (use_conv == True) :
    self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
    self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
    self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
    self.pool = nn.MaxPool2d(2, 2)
    self.fc1 = nn.Linear(1024, hidden_size_layer1)
    self.fc2 = nn.Linear(hidden_size_layer1, hidden_size_layer2)
    self.fc3 = nn.Linear(hidden_size_layer2, num_classes)
  elif (use_eff_net == True) :
    self.model = torchvision.models.efficientnet_b7(pretrained = True)
    self.model = nn.Sequential(*list(self.model.children())[:-1])
    self.fc1 = nn.Linear(2560, hidden_size_layer1)
    self.fc2 = nn.Linear(hidden_size_layer1, hidden_size_layer2)
    self.fc3 = nn.Linear(hidden_size_layer2, num_classes)
  else:
    self.fc1 = nn.Linear(input_size, hidden_size_layer1)
    self.fc2 = nn.Linear(hidden_size_layer1, hidden_size_layer2)
```

```
    self.fc3 = nn.Linear(hidden_size_layer2, num_classes)

#*******************************************************************************************
# FUNCTION:  forward
# PURPOSE:   Passes the input to the selected ANN model (MLP model, CNN model, or
#            EfficientNet-B7 model) to perform forward propagation.
# PARAMS:    The input images, the boolean parameters which select whether the CI, GNI,
#            or DO should be used, and the epoch number.
# RETURNS:   The output.
#*******************************************************************************************

def forward(self, x, use_ci, use_gni, use_do, epoch_num):

  self.use_ci = use_ci
  self.use_gni = use_gni
  self.use_do = use_do
  self.epoch_num = epoch_num

  if (self.use_conv == True) :
    x = self.forward_conv(x)
  elif (self.use_eff_net == True) :
    x = self.forward_eff_net(x)
  else :
    x = self.forward_mlp(x)
  return x

#*******************************************************************************************
# FUNCTION:  forward_conv
# PURPOSE:   Performs forward propagation for the CNN model.
# PARAMS:    The input images.
# RETURNS:   The output.
# REFERENCE: https://shonit2096.medium.com/cnn-on-cifar10-data-set-using-pytorch-34be87e09844
#*******************************************************************************************

def forward_conv(self,x):
  x = self.pool(self.relu(self.conv1(x)))
  x = self.pool(self.relu(self.conv2(x)))
  x = self.pool(self.relu(self.conv3(x)))
  x = x.view(-1, 64 * 4 * 4)
  x = self.forward_mlp(x)
  return x

#*******************************************************************************************
# FUNCTION:  forward_eff_net
# PURPOSE:   Performs forward propagation for the EfficientNet-B7 model.
```

```
  # PARAMS:    The input images.
  # RETURNS:   The output.
  #*******************************************************************************************

  def forward_eff_net(self, x) :
    x = self.model(x)
    x = x.view(-1, 2560)
    x = self.forward_mlp(x)
    return x

  #*******************************************************************************************
  # FUNCTION:  forward_mlp
  # PURPOSE:   Performs forward propagation for the mlp model containing the 2 dense layers.
  #            The dense layers either use the CI, GNI, DO or none.
  # PARAMS:    The input images.
  # RETURNS:   The output.
  #*******************************************************************************************

  def forward_mlp(self,x):
    x = self.fc1(x)
    x = self.relu(x)

    if (self.use_gni == True):
      x = x + x * self.gni_scale_val[self.epoch_num] * Variable(torch.randn(self.batch_size, self
.hidden_size_layer1)).cuda()
    if (self.use_ci == True):
      self.layer1_ci_vals = self.tent_map(self.layer1_ci_vals)
      temp = self.ci_scale_val[self.epoch_num] * (self.layer1_ci_vals - self.ci_offset_val)
      x = x + x * temp
    if (self.use_do == True) :
      x = self.do(x)

    x = self.fc2(x)
    x = self.relu(x)

    if (self.use_gni == True):
      x = x + x * self.gni_scale_val[self.epoch_num] * Variable(torch.randn(self.batch_size, self
.hidden_size_layer2)).cuda()
    if (self.use_ci == True):
      self.layer2_ci_vals = self.tent_map(self.layer2_ci_vals)
      temp = self.ci_scale_val[self.epoch_num] * (self.layer2_ci_vals - self.ci_offset_val)
      x = x + x * temp
    if (self.use_do == True) :
      x = self.do(x)
```

```
  x = self.fc3(x)
  return x

#*****************************************************************************************
# FUNCTION:  tent_map
# PURPOSE:   Iterates the tent map function.
# PARAMS:    The input into the tent map.
# RETURNS:   The output of the tent map.
#*****************************************************************************************

def tent_map(self, chaotic_input) :
  chaotic_input = self.ci_bifur_val * torch.min(chaotic_input, 1 - chaotic_input)
  return chaotic_input

#*****************************************************************************************
# FUNCTION:  init_tent_map
# PURPOSE:   Initializes the tent map values between 0 and 1, and iterates them for
#            1000 iterations to remove transient values.
# PARAMS:    The size of the hidden layers using the tent maps (batch size, hidden size).
# RETURNS:   The output of the tent maps.
#*****************************************************************************************

def init_tent_map(self,hidden_size) :
  chaotic_input = Variable(torch.rand(hidden_size)).cuda()
  for i in range(1000):
    chaotic_input = self.ci_bifur_val * torch.min(chaotic_input, 1 - chaotic_input)
  return chaotic_input

#*****************************************************************************************
# FUNCTION:  init_scale_val
# PURPOSE:   Initializes the scaling values of the tent maps. If use_adapt_scale if true,
#            the scaling value changes depending on the epoch number, otherwise the
#            scaling value is constant.
# PARAMS:    The maximum scaling value, the number of epochs, and the boolean value
#            used to select adaptive scaling.
# RETURNS:   An array containing the scaling values for each epoch.
#*****************************************************************************************

def init_scale_val(self, max_scale_val, num_epochs, use_adapt_scale) :

  scale_val_arr = []

  if (use_adapt_scale == True) :
    for n in range(num_epochs):
      scale_val = (math.log(25 * n + 1)) # w = 25, growth rate parameter
```

```
        scale_val_arr.append(scale_val)
      scale_val_arr = np.array(scale_val_arr)
      scale_val_arr = ((scale_val_arr) / np.max(scale_val_arr)) * max_scale_val
    else :
      for n in range(num_epochs):
        scale_val_arr.append(max_scale_val)
      scale_val_arr = np.array(scale_val_arr)

    return scale_val_arr
```

A.5 Train Model

```
#*********************************************************************************************
# MODULE:     Train_Model
# PURPOSE:    Train the ANN model.
# AUTHOR:     Siobhan Reid
# VERSION:    2
# DATE:       August 1, 2022
#*********************************************************************************************

class Train_Model():

  #******************************************************************************************
  # FUNCTION:   __init__
  # PURPOSE:    Initializes the class variables.
  # PARAMS:     The hyperparameters of the ANN (described in main.ipynb) and the parameters
  #             used to train the ANN, such as the loss function, the k-fold object used to
  #             perform 10-fold cross-validation, and the training data.
  # RETURNS:    NA.
  #******************************************************************************************

  def __init__(self, input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_size,

               ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
               do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
               num_epochs, learning_rate, use_optim, loss_function, kf, train_data_shape, train_d
ata):

    self.input_size = input_size
    self.hidden_size_layer1 = hidden_size_layer1
    self.hidden_size_layer2 = hidden_size_layer2
    self.num_classes = num_classes
    self.batch_size = batch_size
    self.ci_scale_val = ci_scale_val
```

```
  self.gni_scale_val = gni_scale_val
  self.ci_bifur_val = ci_bifur_val
  self.ci_offset_val = ci_offset_val
  self.do_val = do_val
  self.use_conv = use_conv
  self.use_eff_net = use_eff_net
  self.use_ci = use_ci
  self.use_gni = use_gni
  self.use_do = use_do
  self.use_adapt_scale = use_adapt_scale
  self.num_epochs = num_epochs
  self.use_optim = use_optim
  self.loss_function = loss_function
  self.kf = kf
  self.train_data_shape = train_data_shape
  self.train_data = train_data
  self.learning_rate = learning_rate

#******************************************************************************************
# FUNCTION:  calc_valid_loss_and_acc
# PURPOSE:   Calculate the loss and accuracy of the validation data.
# PARAMS:    The validation data generator and the ANN model.
# RETURNS:   The loss and accuracy.
#******************************************************************************************

def calc_valid_loss_and_acc(self, gen, net):

  net.eval()
  running_loss=0
  correct=0
  total=0

  with torch.no_grad():

    for i ,(images,labels) in enumerate(gen):
      if (selected_dataset == fashion_mnist):
        images = Variable(images.view(-1, self.input_size)).cuda()
        labels = Variable(labels).cuda()
      else :
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()

      outputs = net(images, False, False, False, 0)

      loss = self.loss_function(outputs,labels)
```

```
        running_loss += loss.item()

        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    loss = running_loss/len(gen)
    acc = 100.*correct/total

    return loss, acc

#*******************************************************************************************
# FUNCTION:  train_model
# PURPOSE:   Trains the ANN model. Saves the ANN model at the epoch with the lowest
#            validation loss.
# PARAMS:    The path where to save the model, and the training and validation data.
# RETURNS:   The loss and accuracy per epoch for the training and validation data.
# REFERENCE: Reference: https://towardsdatascience.com/how-to-save-and-load-a-model-
#            in-pytorch-with-a-complete-example-c2920e617dee
#*******************************************************************************************

def train_model(self, best_model_path, net, train_gen, valid_gen):

    print("Learning rate: ", self.learning_rate)
    print("Num epochs: ", self.num_epochs)
    train_loss_arr = []
    train_acc_arr = []
    valid_loss_arr = []
    valid_acc_arr = []

    min_valid_loss = 1000000

    if (self.use_optim == True) :
      lr = 1e-4
      optimizer = torch.optim.Adam(
          (p for p in net.parameters() if p.requires_grad), lr=lr
      )

    for epoch in range(self.num_epochs):

      print('\nEpoch : %d'%epoch)
      net.train()
      running_loss=0
      correct=0
      total=0
```

```
for i, (images,labels) in enumerate(train_gen):
  if (selected_dataset == fashion_mnist):
    images = Variable(images.view(-1, self.input_size)).cuda()
    labels = Variable(labels).cuda()
  else :
    images = Variable(images).cuda()
    labels = Variable(labels).cuda()

  outputs = net(images, self.use_ci, self.use_gni, self.use_do, epoch)

  if (self.use_optim == True) :
    optimizer.zero_grad()
    loss = self.loss_function(outputs, labels)
    loss.backward()
    optimizer.step()

  else :
    net.zero_grad()
    loss = self.loss_function(outputs, labels)
    loss.backward()
    for name, param in net.named_parameters():
      if (param.requires_grad) :
        param.data -= self.learning_rate * param.grad.data

  running_loss += loss.item()
  _, predicted = outputs.max(1)
  total += labels.size(0)
  correct += predicted.eq(labels).sum().item()

train_loss = running_loss/len(train_gen)
train_acc = 100.*correct/total
valid_loss, valid_acc = self.calc_valid_loss_and_acc(valid_gen, net)

print('Train Loss: %.3f | Accuracy: %.3f'%(train_loss,train_acc))
print('Valid Loss: %.3f | Accuracy: %.3f'%(valid_loss,valid_acc))

train_loss_arr.append(train_loss)
train_acc_arr.append(train_acc)
valid_loss_arr.append(valid_loss)
valid_acc_arr.append(valid_acc)

if (valid_loss < min_valid_loss) :
  print("best " , min_valid_loss)
  print("curr ", valid_loss)
```

```
        checkpoint = { 'epoch': epoch, 'valid_loss_min': valid_loss, 'state_dict': net.state_dict
() }
        min_valid_loss = valid_loss
        torch.save(checkpoint, best_model_path)

    return train_loss_arr, train_acc_arr, valid_loss_arr, valid_acc_arr

  #******************************************************************************************
  # FUNCTION:  kfold_train_model
  # PURPOSE:   Performs 10-fold cross validation. The kf object separates the training data
  #            into 10 folds. Each training run, a different fold is used as the
  #            validation data. The training and validation data are passed to the
  #            train_model() function. After training, the average loss and accuracy per
  #            epoch are calculated for the 10 training runs.
  # PARAMS:    The path where to save the model.
  # RETURNS:   NA.
  #******************************************************************************************

  def kfold_train_model(self, save_model_path) :

    start = timeit.default_timer()
    kfold_num = 0
    kfold_train_loss_arr = []
    kfold_train_acc_arr = []
    kfold_valid_loss_arr = []
    kfold_valid_acc_arr = []

    for train_indexes, valid_indexes in self.kf.split(self.train_data_shape) :

      print("*****************************************************************")
      print("Kfold Number: ", kfold_num)

      train_set = torch.utils.data.Subset(self.train_data, train_indexes)
      valid_set = torch.utils.data.Subset(self.train_data, valid_indexes)
      train_gen = torch.utils.data.DataLoader(train_set, batch_size = self.batch_size,num_workers
 = 0, worker_init_fn = load_dataset.seed_worker, drop_last = True, shuffle = True)
      valid_gen = torch.utils.data.DataLoader(valid_set, batch_size = self.batch_size, shuffle=Fa
lse)

      net = Net(self.input_size, self.hidden_size_layer1, self.hidden_size_layer2, self.num_class
es, self.batch_size,
                self.ci_scale_val, self.gni_scale_val, self.ci_bifur_val, self.ci_offset_val, sel
f.do_val,
                self.use_conv, self.use_eff_net, self.use_ci, self.use_gni, self.use_do, self.use
_adapt_scale, self.num_epochs)
```

```
      if torch.cuda.is_available():
        net.cuda()

      best_model_path = save_model_path + "kfold_" + str(kfold_num) + "_best_model.pt"
      train_loss_arr, train_acc_arr, valid_loss_arr, valid_acc_arr = self.train_model(best_model_
path, net, train_gen, valid_gen)

      df = pd.DataFrame({"train_loss": train_loss_arr, "train_acc": train_acc_arr, "valid_loss":
valid_loss_arr, "valid_acc": valid_acc_arr})
      df_save_path = save_model_path + "/plots_" + str(kfold_num) + ".csv"
      df.to_csv(df_save_path)

      kfold_train_loss_arr.append(train_loss_arr)
      kfold_train_acc_arr.append(train_acc_arr)
      kfold_valid_loss_arr.append(valid_loss_arr)
      kfold_valid_acc_arr.append(valid_acc_arr)

      net = None
      torch.cuda.empty_cache()
      gc.collect()
      kfold_num += 1

    kfold_train_loss_arr = np.mean(np.array(kfold_train_loss_arr), axis = 0)
    kfold_train_acc_arr = np.mean(np.array(kfold_train_acc_arr), axis = 0)
    kfold_valid_loss_arr = np.mean(np.array(kfold_valid_loss_arr), axis = 0)
    kfold_valid_acc_arr = np.mean(np.array(kfold_valid_acc_arr), axis = 0)
    kfold_df = pd.DataFrame({"train_loss" : kfold_train_loss_arr, "train_acc" : kfold_train_acc_a
rr, "valid_loss" : kfold_valid_loss_arr, "valid_acc" : kfold_valid_acc_arr})
    kfold_df_save_path = save_model_path + "/plots.csv"
    kfold_df.to_csv(kfold_df_save_path)

    stop = timeit.default_timer()
    print('Time: ', stop - start)

    return
```

A.6 Display Results

```
#*********************************************************************************************
# MODULE:     Display_Results
# PURPOSE:    Plots the accuracy and loss per epoch, and calculates the performance metrics
#             (accuracy, sensitivity, specificity, etc) of the ANN models.
# AUTHOR:     Siobhan Reid
```

```
# VERSION:    2
# DATE:       August 1, 2022
#*********************************************************************************************

class Display_Results():

  #*******************************************************************************************
  # FUNCTION:   __init__
  # PURPOSE:    Initializes the class variables.
  # PARAMS:     The hyperparameters of the ANN (described in main.ipynb) and the parameters
  #             used to display the ANN results, such as the loss function and the k-fold
  #             object.
  # RETURNS:    NA.
  #*******************************************************************************************

  def __init__(self, input_size, hidden_size_layer1, hidden_size_layer2, num_classes, batch_size,

               ci_scale_val, gni_scale_val, ci_bifur_val, ci_offset_val,
               do_val, use_conv, use_eff_net, use_ci, use_gni, use_do, use_adapt_scale,
               num_epochs, kf, train_data_shape, train_data, test_gen, save_model_path):

    self.input_size = input_size
    self.hidden_size_layer1 = hidden_size_layer1
    self.hidden_size_layer2 = hidden_size_layer2
    self.num_classes = num_classes
    self.batch_size = batch_size
    self.ci_scale_val = ci_scale_val
    self.gni_scale_val = gni_scale_val
    self.ci_bifur_val = ci_bifur_val
    self.ci_offset_val = ci_offset_val
    self.do_val = do_val
    self.use_conv = use_conv
    self.use_eff_net = use_eff_net
    self.use_ci = use_ci
    self.use_gni = use_gni
    self.use_do = use_do
    self.use_adapt_scale = use_adapt_scale
    self.num_epochs = num_epochs
    self.kf = kf
    self.train_data_shape = train_data_shape
    self.train_data = train_data
    self.test_gen = test_gen
    self.save_model_path = save_model_path

  #*******************************************************************************************
```

```
# FUNCTION:  plot_loss_and_acc_per_epoch
# PURPOSE:   Used to plot the loss or accuracy per epoch for the training and validation
#            data.
# PARAMS:    The average loss or accuracy arrays for the ANNs, the location
#            where the legend should be placed on the plot, the title of the plot,
#            and the title of the y axis.
# RETURNS:   NA.
#*******************************************************************************************

def plot_loss_and_acc_per_epoch(self, aci, ci, base, drop, gni, text_loc, title, y_label) :

  plt.figure()
  plt.rcParams["figure.figsize"] = (12,10)
  plt.rcParams['font.family'] = 'serif'
  plt.rcParams['font.serif'] = ['Times New Roman'] + plt.rcParams['font.serif']

  plt.plot(base, linewidth=5)
  plt.plot(drop, linewidth=5)
  plt.plot(gni, linewidth=5)
  plt.plot(ci, linewidth=5)
  plt.plot(aci, linewidth=5)

  plt.xticks([0,10,20,30,40,50], fontsize = 25)
  # plt.xticks([0,5,10,15,20], fontsize = 25)
  plt.yticks(fontsize = 25)
  plt.legend(["Baseline", "DO", "GNI (Non-Adaptive α)", "CI (Non-
Adaptive α)", "CI (Adaptive α)"], loc=text_loc, prop={'size': 25})
  plt.title(title, fontsize=25)
  plt.xlabel("Epoch", fontsize=25)
  plt.ylabel(y_label, fontsize=25)

  return

#*******************************************************************************************
# FUNCTION:  get_preds
# PURPOSE:   Get the predictions from the ANN model for the validation or test data.
# PARAMS:    The ANN model and the validation or test data generator.
# RETURNS:   The predicted classes and the labels.
#*******************************************************************************************

def get_preds(self, net, gen) :

  preds_tensor = torch.Tensor().cuda()
  labels_tensor = torch.Tensor().cuda()
```

```
  net.eval()

  with torch.no_grad():

    for i ,(images, labels) in enumerate(gen):
      if (selected_dataset == fashion_mnist):
        images = Variable(images.view(-1, self.input_size)).cuda()
        labels = Variable(labels).cuda()
      else :
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()

      outputs = net(images, False, False, False, 0)
      preds_tensor = torch.cat((preds_tensor, outputs), dim=0)
      labels_tensor = torch.cat((labels_tensor, labels), dim=0)

  preds = list(preds_tensor.detach().cpu().numpy())
  preds = np.stack(preds)
  preds = preds.argmax(axis=1)
  labels = list(labels_tensor.detach().cpu().numpy())
  labels = np.stack(labels)

  return preds, labels

#*********************************************************************************************
# FUNCTION:  display_metrics
# PURPOSE:   Calculates the performance metrics (accuracy, sensitivity, specificity,
#            positive-predictive value, negative-predictive value, and f1-score)
#            of the ANN models for the validation and test data.
# PARAMS:    The ANN model, the data generator for the validation or test data,
#            the path to where the trained weights are stored for the ANN model,
#            and the name of the model (either baseline, CIA (adaptive), CI, DO, or GNI).
# RETURNS:   A dataframe containing the calculated metrics for each class and the
#            averages.
# REFERENCE: https://stackoverflow.com/questions/31324218/scikit-learn-how-to-obtain-
#            true-positive-true-negative-false-positive-and-fal
#*********************************************************************************************

def display_metrics(self, net, gen, best_model_path, name) :

  checkpoint = torch.load(best_model_path)
  net.load_state_dict(checkpoint['state_dict'])
  preds, labels = self.get_preds(net, gen)
  conf_mat=confusion_matrix(labels, preds)
  class_accuracy=100*conf_mat.diagonal()/conf_mat.sum(1)
```

```
    fp = conf_mat.sum(axis=0) - np.diag(conf_mat)
    fn = conf_mat.sum(axis=1) - np.diag(conf_mat)
    tp = np.diag(conf_mat)
    tn = conf_mat.sum() - (fp+fn+tp)

    sn = tp/(tp+fn)
    sp = tn/(tn+fp)
    ppv = tp/(tp+fp)
    npv = tn/(tn+fn)
    f1 = 2*(sn*ppv)/(sn+ppv)
    acc = (tp+tn)/(tp+fp+fn+tn)

    df = pd.DataFrame({"sn" + name : sn, "sp" + name: sp, "ppv" + name: ppv, "npv" + name: npv, "
acc" + name : acc, "f1" + name : f1})
    df.loc['mean'] = df.mean()

    return df

  #*******************************************************************************************
  # FUNCTION:  kfold_display_metrics
  # PURPOSE:   Calculates the performance metrics for each of the ten models created during
  #            cross-validation, and then takes the average. Displays the average cross-
  #            validation metrics and saves them into a csv file. This is done for each of
  #            ANN models (baseline, CI, CIA, DO, and GNI).
  # PARAMS:    A boolean value used to determine whether the results should be calculated
  #            for the validation or test data.
  # RETURNS:   NA.
  #*******************************************************************************************

  def kfold_display_metrics(self, get_valid_results):

    kfold_df = pd.DataFrame()
    kfold_num = 0

    pd.set_option('display.max_columns', None)

    for train_indexes, valid_indexes in self.kf.split(self.train_data_shape) :

      set_seed()
      net = Net(self.input_size, self.hidden_size_layer1, self.hidden_size_layer2, self.num_class
es, self.batch_size,
                self.ci_scale_val, self.gni_scale_val, self.ci_bifur_val, self.ci_offset_val, sel
f.do_val,
```

```
                self.use_conv, self.use_eff_net, self.use_ci, self.use_gni, self.use_do, self.use
_adapt_scale, self.num_epochs)

      if torch.cuda.is_available():
        net.cuda()

      train_set = torch.utils.data.Subset(self.train_data, train_indexes)
      valid_set = torch.utils.data.Subset(self.train_data, valid_indexes)
      train_gen = torch.utils.data.DataLoader(train_set, batch_size = self.batch_size,num_workers
 = 0, worker_init_fn = load_dataset.seed_worker, drop_last = True, shuffle = True)
      valid_gen = torch.utils.data.DataLoader(valid_set, batch_size = self.batch_size, shuffle =
False)

      if (get_valid_results == True) :
        gen = valid_gen
      else :
        gen = self.test_gen

      model_path = self.save_model_path + "/aci/kfold_" + str(kfold_num) + "_best_model.pt"
      aci_df = self.display_metrics(net, gen, model_path, "_cia")

      model_path = self.save_model_path + "/base/kfold_" + str(kfold_num) + "_best_model.pt"
      base_df = self.display_metrics(net, gen, model_path, "_base")

      model_path = self.save_model_path + "/ci/kfold_" + str(kfold_num) + "_best_model.pt"
      ci_df = self.display_metrics(net, gen, model_path, "_ci")

      model_path = self.save_model_path + "/do/kfold_" + str(kfold_num) + "_best_model.pt"
      do_df = self.display_metrics(net, gen, model_path, "_do")

      model_path = self.save_model_path + "/gni/kfold_" + str(kfold_num) + "_best_model.pt"
      gni_df = self.display_metrics(net, gen, model_path, "_gni")

      df = pd.concat([base_df, ci_df, aci_df, do_df, gni_df], axis = 1)
      df = df.reindex(sorted(df.columns), axis=1)

      kfold_df = pd.concat([kfold_df, df])

      net = None
      torch.cuda.empty_cache()
      gc.collect()
      kfold_num += 1

    kfold_df = kfold_df.groupby(level=0).mean()
    kfold_df = kfold_df * 100
```

```
kfold_df = kfold_df.round(2)
display(kfold_df)

if (get_valid_results == True):
  kfold_df.to_csv(self.save_model_path + "/results/valid_results.csv")
else :
  kfold_df.to_csv(self.save_model_path + "/results/test_results.csv")

return
```

Appendix B

Table B-1. Performance metrics of the validation data for the Fashion-MNIST dataset.

Metric	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
ACC	97.98	98.06	**98.08**	98.04	98.05
F1	89.88	90.26	**90.35**	90.15	90.22
NPV	98.88	98.92	**98.93**	98.91	98.92
PPV	89.99	90.31	**90.37**	90.18	90.24
SN	89.90	90.30	**90.38**	90.19	90.25
SP	98.88	98.92	**98.93**	98.91	98.92

Table B-2. Performance metrics of the validation data for the CIFAR-10 dataset.

Metric	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
ACC	94.35	95.11	**95.14**	95.04	95.04
F1	71.57	75.54	**75.65**	75.17	75.23
NPV	96.87	97.29	**97.30**	97.25	97.25
PPV	72.32	75.78	**75.93**	75.32	75.48
SN	71.72	75.56	**75.71**	75.21	75.20
SP	96.86	97.28	**97.30**	97.25	97.24

Table B-3. Performance metrics of the validation data for the Stanford Cars dataset.

Metric	Base	CI (α constant)	CI (α adaptive)	DO	GNI (α constant)
ACC	99.74	99.80	**99.82**	99.79	99.80
F1	75.11	80.71	**82.32**	79.98	80.78
NPV	99.87	99.90	**99.91**	99.89	99.90
PPV	76.95	82.24	**83.81**	80.90	81.39
SN	75.04	80.48	**82.27**	79.92	80.20
SP	99.87	99.90	**99.91**	99.89	99.90