Influence of stabilizing strong synapses on "catastrophic forgetting"

Dissertation thesis submitted to the University of Hyderabad towards the partial fulfillment of the requirement for the award of the degree of

Master of Science (M.Sc.)
IN
NEURAL AND COGNITIVE SCIENCES

thesis submitted by

SANTOSH SHUKLA
(21MNMS16)

Conferred by the

Center for Neural and Cognitive Sciences
School of Medical Sciences
University of Hyderabad, Gachibowli
Hyderabad-500046
May 2023

Under the supervision of
Dr. Joby Joseph
Associate Professor
Center for Neural and Cognitive Sciences
School of Medical Sciences
University of Hyderabad

Certification

This is to certify that the thesis entitled “Influence of Stabilizing Strong Synapse on ‘Catastrophic Forgetting’ ' was submitted by Santosh Shukla bearing Reg.No. 21MNMS16, is a partial fulfillment of the requirements for the award of a Master of Science in Neural and Cognitive Sciences and is bona fide work carried out by her under my supervision and guidance.

The thesis has not been submitted previously, in part or in full, to this or any other university or institution for the award of any degree or diploma.

Dr.Joby Joseph

Supervisor

Associate Professor

Centre of Neural and Cognitive

Science University of Hyderabad

Declaration by Student

I, Santosh Shukla (21MNMS16), an M.Sc. in Neural and Cognitive Sciences student at the University of Hyderabad, hereby declares that the dissertation titled "Influence of Stabilizing Strong Synapse on 'Catastrophic Forgetting'" submitted by me under the guidance and supervision of Dr. Joby Joseph is a genuine work. I further certify that it has not previously been submitted in part or in whole to this university or any other university or institution for the granting of any degree or diploma.

I hereby certify that the information given here is correct to the best of my knowledge.

Date: May 8, 2023

Place: Hyderabad

Santosh Shukla
21MNMS16
Center for Neural and Cognitive Science
School of Medical Sciences
University of Hyderabad

Acknowledgements

I am immensely grateful to my project supervisor, Dr. Joby Joseph, whose unwavering support and guidance have been invaluable throughout this journey. I must acknowledge his exceptional patience and understanding, particularly during those moments when my penchant for indiscipline and lack of punctuality threatened to derail our progress. Despite my occasional chaos, Dr. Joseph remained dedicated to my growth and success. His belief in my abilities, even at my lowest points, has left a lasting impact. I am truly fortunate to have had such a patient mentor who saw potential in my disorganized ways and helped me channel my energy towards meaningful progress. Dr. Joseph, thank you for your infinite patience, unwavering support, and all the lessons. Your mentorship has shaped both this project and my character, and I am eternally grateful for the impact you've had on my life.

I would like to extend a big, shout-it-from-the-rooftops kind of thank you to Krishna Pratap Singh for lending me his magical laptop that brought my thesis to life. Your laptop, with all its quirks and superpowers, was my trusty sidekick throughout this adventure. And let's not forget the motivation you provided, whether it was through hilarious memes or enthusiastic pep talks. You truly saved the day (and my procrastinating soul). I am forever grateful for your support and the laughter-filled moments we shared.

I would like to extend my heartfelt thanks to my amazing friends and classmates, Hina, Sagarika, Sumiran, Dileep, Samuel, Rajshree, and Meghna. Your unwavering support, patience, and ability to endure my complaining and whining have been truly remarkable. Thank you for being there to lend an ear, offer encouragement, and inject humour into even the most stressful moments. I am grateful to have shared this adventure with all of you.

I would also like to acknowledge my incredible labmates and seniors, Deepak, Pranay, Shivraju, and Pallavi. Your guidance, expertise, and camaraderie have been invaluable. From sharing insights to helping me navigate our research, you have been an indispensable part of my journey. Thank you for your support and collaboration.

I would be remiss if I didn't acknowledge the fantastic motivation that yours truly, the master of procrastination, provided. Despite my best efforts to delay, my determination prevailed, and I managed to complete this project.

CONTENTS

Abstract

In the realm of machine learning, the phenomenon of catastrophic forgetting in artificial neural networks has long been a problem. The creation of intelligent systems that can adapt and change over time depends on neural networks' capacity to continuously learn new information while maintaining previously acquired knowledge. However, in neural networks, this kind of learning of new mappings disrupts the existing mappings, leading to a phenomenon called catastrophic failure. Many methods have been proposed to address this issue, like EWC (Elastic Weight Consolidation), the Generative Replay Model, etc. In a neural network that is undergoing training, where the weights remain approximately normally distributed. However, it has been reported in the brains of many organisms, ranging from insects to mammals, that continuous learning over a lifetime may lead to an increase in brain connection strengths as measured by the gross volume and spine densities. Given this observation, it is possible that, as learning progresses over a lifetime, certain synapses that have gained strength do not weaken, thus broadening the distribution. There are possibly several modifications to the learning rules that can cause this kind of change to the distribution. In this thesis, we propose to explore modifications to the learning rules in a multilayer network to achieve this change in distribution of weights and measure the performance of these modifications on catastrophic failures in sequential training.

Introduction

In machine learning, catastrophic forgetting is a common problem, especially when used with artificial neural networks. When a neural network that has been taught to execute a certain job is further trained on an unrelated task, it will display this phenomenon and perform significantly worse on the original task. The primary cause of this is that during the training process for the next task, the neural network may forget or replace previously learned information, which might result in an unintentional adjustment to the weights of the neural network. This significant decline in the network's performance on the first task is referred to as "catastrophic forgetting." McCloskey et al. (1989)

Humans and other species can continuously acquire new information without losing their prior knowledge; therefore, catastrophic forgetting is not seen in them. Humans, for instance, are capable of learning new mathematical concepts or identifying new bird species without losing what they have already learned.

Various approaches are being explored to address the issue of catastrophic forgetting in connectionist systems. James Kirkpatrick et al. introduced one such method that involves reducing the learning rate of weights based on their significance in capturing mappings from earlier tasks. This technique aims to overcome the problem of forgetting previously learned information when training for new tasks.

In addition, researchers have drawn inspiration from observations in the brain, particularly the limbic system, where the replay of sequences of neuron activations is believed to play a role in consolidating episodic memory into a cortical-dependent form. Gido M. van de Ven et

al. proposed leveraging this idea to improve performance in scenarios involving catastrophic forgetting within connectionist architectures. These methods often rely on additional information stored outside of synapses, such as in memory in other locations.

In our study, we aim to evaluate the effectiveness of another brain-inspired mechanism in mitigating catastrophic forgetting. This mechanism seeks to minimize or eliminate the reliance on non-local parameters for weight updates. By exploring this approach, we hope to contribute to the development of more efficient solutions for addressing catastrophic forgetting in connectionist systems.

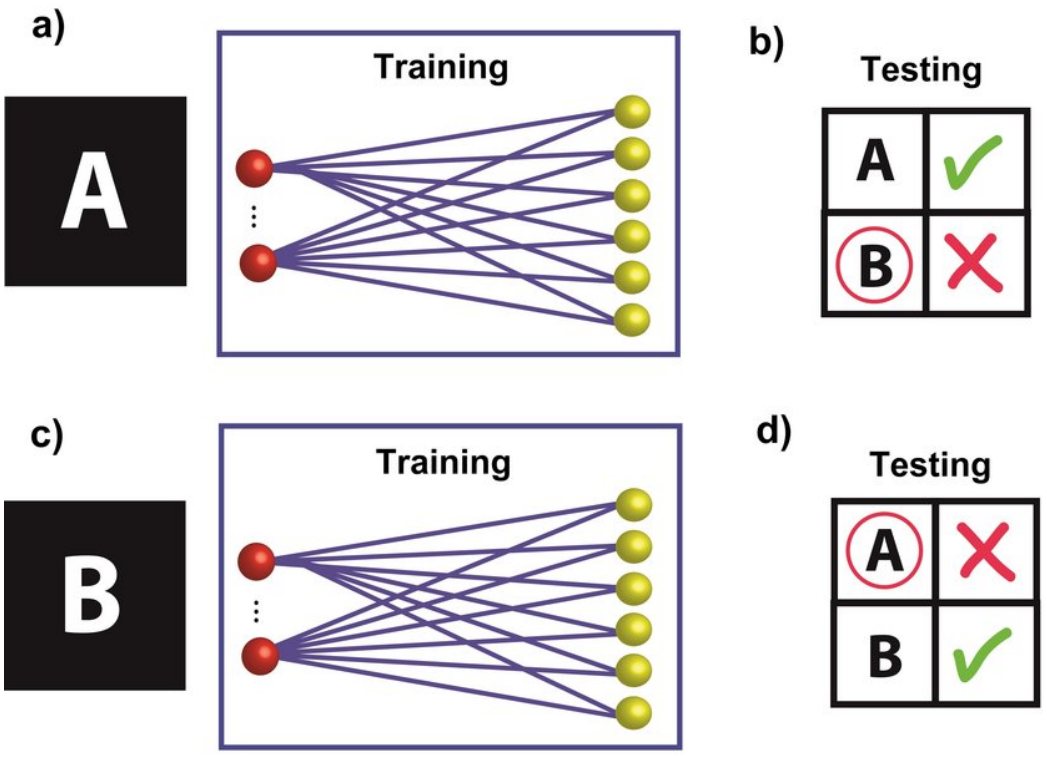


Fig. 1: Illustration of catastrophic forgetting in a multilayer perceptron network:After training a certain pattern A (a), A is recognized, while B is not (b). If the same network is then trained to learn pattern B (c), A is forgotten while pattern B is recognized (d). Source: Unoz-Martin et al., 2019.

Background

Analogy between an artificial neuron and a biological neuron

Scientists have studied the structure and function of biological neurons in order to construct mathematical models of artificial neurons that can mimic the activity of genuine neurons. (2017) Hassabis et al.

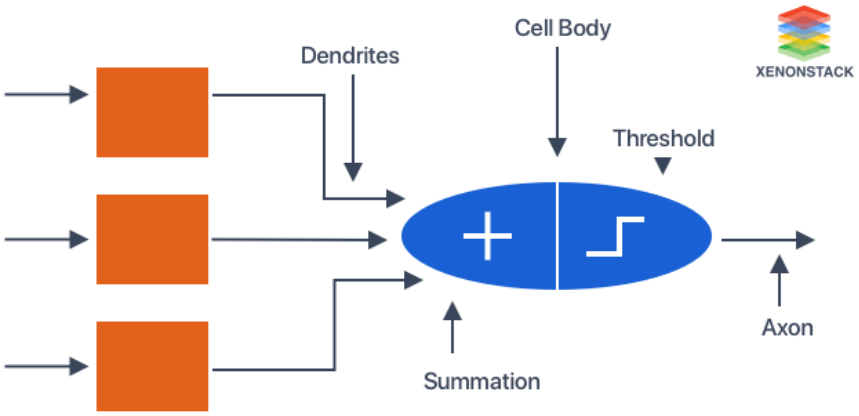


Figure 2: Analogy between an Artificial Neuron and a Biological Neuron, Source-Google Image Search

The model for the artificial neuron was the biological neuron, which serves as the fundamental unit of computation in the brain and nervous system. Scientists have studied the structure and function of biological neurons in order to construct mathematical models of artificial neurons that can mimic the activity of genuine neurons. The biological neuron has had a considerable impact on the artificial neuron, including:

Input: Just like a real neuron, an artificial neuron may receive stimulation from other neurons or sensors. This input is provided to biological neurons as electrical or chemical impulses from other neurons, but it is represented in artificial neurons as a vector of numerical values.

Activation: An artificial neuron, like a real neuron, employs an activation function on its input to generate an output signal. The activation function in a biological neuron is the final

result of the integration of the input signals received by the neuron, but in an artificial neuron, the activation function is a mathematical function that transforms the input to a new value (Goodfellow et al., 2016, pp. 31–202).

Output: The artificial neuron, like the real neuron, generates an output signal that is relayed to other neurons or effectors. In the biological neuron, its output is the release of neurotransmitters that transfer the signal over the synapses to other neurons, but in the artificial neuron, it is a numerical value that is sent across the synapses to other artificial neurons (Russell & Norvig, 2010, pp. 42–727).

Learning: Artificial neurons, like actual neurons, may modify the connections or weights of their connections in response to input, enabling them to learn and adapt to new information. This is achieved in a real neuron by altering the strength of synaptic connections between neurons, while in an artificial neuron, it is done by using mathematical algorithms to change the weights of connections between artificial neurons. The capacity of an artificial neuron to modify its connections or weights in response to input is akin to synaptic plasticity in real neurons (Haykin, 1999, pp. 45–46).

Overall, the biological neuron has inspired the construction of artificial neurons that can mimic its function, allowing researchers to design artificial neural networks capable of image and speech recognition, natural language processing, and autonomous control.

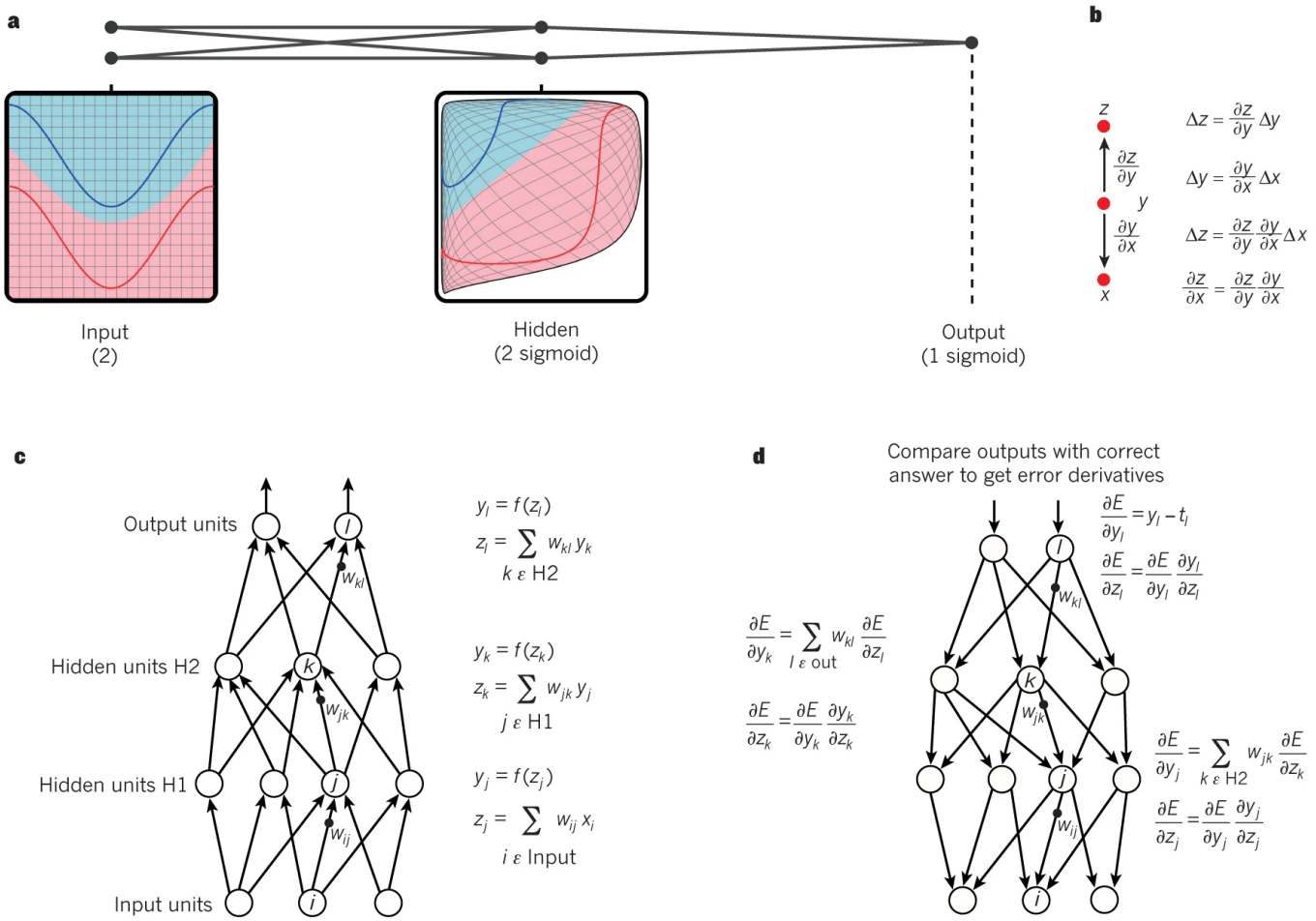


Fig. 3: Mechanism of the Neural Network Source: LeCun et al. (2015)

a) The image shows a multi-layer neural network capable of changing the input space such that distinct classes of data may be segregated linearly. The graphic depicts how the network's hidden units alter the regular grid of the input space. The example network is made up of a single output unit, two hidden units, and two input units.

b) The chain rule of derivatives explains how small changes in two variables, x and y, affect a third variable, z. When a small change Δx occurs in x, it produces a small change Δy in y through multiplication by the partial derivative ∂y/∂x. Similarly, the change Δy produces a change Δz in z. The chain rule expresses how Δx is transformed into Δz by multiplying the partial derivatives ∂y/∂x and ∂z/∂y.

c) A weighted sum of the outputs from the units in the layer below is used in the forward pass of a neural network with two hidden layers and one output layer to calculate the total input z to each unit in the layer. In order to obtain the output of the unit, this input is subsequently processed through a non-linear activation function, such as the often employed ReLU or sigmoid functions. The use of biased terminology has been eliminated in this description for the sake of simplicity.

d) For determining the error derivatives with regard to the outputs of the hidden layers and output layer in a neural network, backward pass equations are utilized. To start, we weight the error derivatives with respect to the total inputs of the units in the layer above and use that sum to compute the error derivative with respect to the output of each unit in a hidden layer. To obtain the error derivative with regard to the unit's input, we multiply this error derivative by the gradient of the non-linear activation function f(z). At the output layer, the cost function is differentiated to calculate the error derivative with respect to a unit's output. The error derivative is (yl tl) if the cost function for unit l is 0.5 (yl tl)2, where tl is the target value. Last but not least, we only multiply the output of unit j, yj, with the error derivative E/zk in order to compute the error derivative for the weight wjk on the link from unit j in the layer beneath to unit k in the present layer.

There are several types of neural networks, each with its own architecture and range of applications, including feedforward neural networks, recurrent neural networks, and convolutional neural networks.

A feedforward neural network is the most basic kind of neural network, consisting of layers of neurons processing inputs in a one-way fashion without feedback. They are commonly used in pattern recognition and classification applications.

Since they incorporate feedback connections, recurrent neural networks can understand

sequential input such as time series, audio, and text. They are particularly useful for operations that need the simulation of long-term reliance.

Convolutional neural networks use filters to extract features from input data since they are especially built for processing grid-like data, such as images. They are often used in computer vision applications such as image segmentation and object recognition. LeCun and colleagues (2015).

Brain-inspired models of catastrophic forgetting

Throughout the years, a number of approaches have been proposed to deal with this problem, each with advantages and disadvantages of its own.

One strategy that Zenke et al. (2017) suggested is synaptic intelligence (SI). In order to regulate the plasticity of each synapse, SI computes a measure of synaptic significance during training. The fact that synaptic strengths in biological brain networks are highly variable and that the relative importance of each synapse in the network controls synaptic plasticity led to the development of this method. Throughout the years, a number of approaches have been proposed to deal with this problem, each with advantages and disadvantages of its own.

One strategy that Zenke et al. (2017) suggested is synaptic intelligence (SI). In order to regulate the plasticity of each synapse, SI computes a measure of synaptic significance during training. The fact that synaptic strengths in biological brain networks are highly variable and that the relative importance of each synapse in the network controls synaptic plasticity led to the development of this method.

Kirkpatrick et al. (2017) proposed Elastic Weight Consolidation (EWC), a brand-new technique that only maintains the network parameters necessary for previously learned tasks.

To do this, each parameter is given a value based on how sensitively it impacts how well the network did on the previous task.

Aljundi et al. (2018) proposed Memory-Aware Synapses (MAS), which offer each synapse a memory vector that remembers its previous importance in the network. This is the third approach. The memory vector that controls each synaptic update safeguards the synapses needed for previously learned tasks during training.

The hippocampus' role in memory consolidation served as the foundation for the fourth technique, which was based on the concept of synaptic stability. This approach allows the network to keep picking up new skills while stabilizing the synapses that are essential for previously learned behaviors. Weight decay or slow learning rates are only two examples of strategies that may successfully avoid catastrophic forgetting, per study (Masse et al., 2018; Bellec et al., 2019).

Each strategy's performance varies depending on the specific use case, but their success in various situations suggests intriguing avenues for further research and development in this area. By better comprehending the mechanisms of learning and memory in the brain, we can continue to develop more effective models that can complete a number of tasks without experiencing catastrophic forgetting.

In conclusion, the creation of several techniques has led to a significant improvement in the study of decreasing catastrophic forgetting in artificial neural networks. By controlling or conserving synapses or characteristics that are essential for previously learned actions, each technique attempts to address the problem of catastrophic forgetting. The success of these techniques emphasizes the need for more research in this area as we continue to develop intelligent systems that can evolve and expand.

Translating Insights from Brain Studies to Improve Neural Network Performance

According to research by Eleanor Maguire et al. titled "Navigation-related structural change in the hippocampus of taxi drivers," licensed London cab drivers exhibit a substantial variation in hippocampal structure compared to control participants. A region of the brain called the hippocampus is responsible for memory and spatial orientation. The posterior hippocampus, which is linked to spatial navigation, was seen to be substantially bigger in taxi drivers than in control participants, although the anterior hippocampus, which is linked to memory, was.

Taxi drivers' considerable training in navigating the intricate streets of London, which necessitates them to continuously update and employ spatial maps, is probably the cause of this anatomical variation in the hippocampus. The work offers proof of the hippocampus' structural flexibility in response to ongoing learning and navigation experiences.

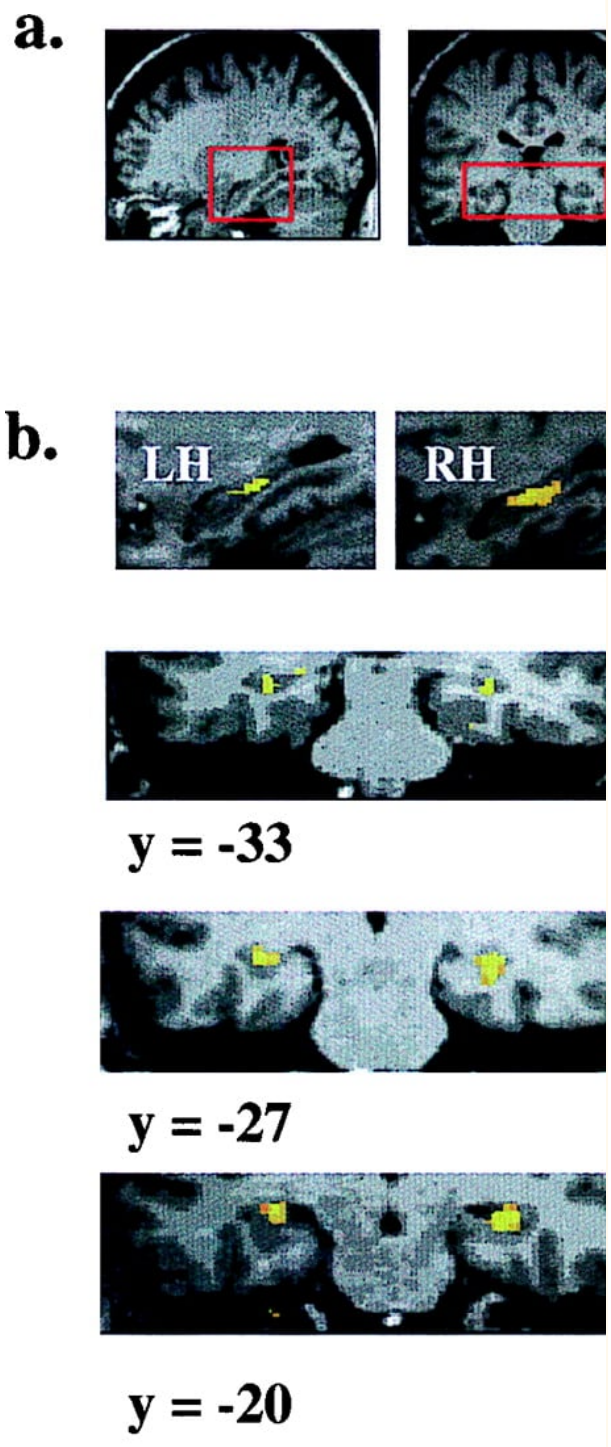
According to a 2021 study by Schmidt et al., the gray matter volume of the brain may increase because of experience-dependent structural plasticity. The major goal of the research was to determine how experience and learning impact brain anatomy. The researchers found that, in addition to other physical changes, learning and experience may result in an increase in the amount of gray matter in the brain. Synaptic and astrocyte remodeling, which occurs when the brain adjusts to changing conditions by changing its structure, results in an increase in gray matter volume. Whereas astrocyte remodeling refers to alterations in the cells that support and feed neurons, synaptic remodeling refers to changes in the connections between neurons. These structural alterations help learning and memory by establishing new channels for information to go through the brain. The study's findings suggest that the adult brain is structurally malleable and that it may alter and grow as a consequence of experience. This mechanism offers one explanation for the observed increase in brain size due to continuous

learning. According to the authors of the research, experience-dependent structural plasticity may lead to the creation of new brain circuits and an increase in the number of synapses, both of which might lead to an overall increase in gray matter.

The study by Schmidt et al. concludes that experience and education may alter the brain in ways that increase the volume of gray matter. The research underlines the value of lifelong learning and the benefits it could provide for maintaining brain health and cognitive function.

Ott SR and colleagues investigated the effects of the migratory and non-migratory phases of the locusts on the mushroom body (MB), an area of the brain connected to learning and memory. The MB's volume expanded throughout the course of the migratory period, the researchers discovered, suggesting that it may act as a substantial data processing hub for data on migration behavior. The researchers first watched how locusts behaved in the wild and made a distinction between those that were moving and those that were not in order to perform the study. After measuring the MB volume using magnetic resonance imaging, the outcomes for the two groups were compared (MRI). The volume of the MB was much greater than that of non-migratory locusts, according to the researchers.

Fig 6:



(a left) A sagittal section of an MRI scan with the hippocampus indicated by the red box.

(a right) coronal section through the MRI scan, again with the hippocampi indicated.

(b) The group results are shown superimposed onto the scan of an individual subject selected at random. The bar to the right indicates the Z score level. increased gray matter volume in the posterior of the left and right hippocampus (LH and RH, respectively) of taxi drivers relative to those of controls, shown at the top of the figure in the sagittal section. Underneath, the areas of gray matter difference are shown in coronal sections at three different coordinates on the y axis to illustrate the extent of the difference down the long axis of the hippocampus.

(c) Increased gray matter volume in the anterior of the left and right hippocampus of controls relative to those of taxi drivers, shown in the sagittal section. (Adap from a paper by **Eleanor A. Maget et al.**)

Recent research has shown that in biological neural networks, the volume of the mushroom body, a specialized brain region, increases during the acquisition of new memories (Ott et al., 2018). In contrast, typical connectionist architectures in artificial neural networks do not exhibit this behavior, as the distribution of synaptic weights remains unchanged with no preference for retaining particular synaptic strengths. However, the observed increase in volume with learning in biological neural networks suggests that there may be a need to modify learning algorithms in artificial neural networks to better reflect this phenomenon.

To achieve this, we propose to modify the learning algorithm at the synapses. The modification of learning algorithms in artificial neural networks to better reflect the behavior of biological neural networks is a promising avenue for improving the performance and applicability of these systems. By incorporating a more biologically plausible approach to learning, we can develop more robust and effective artificial neural networks that can better model the complex and dynamic processes of the brain.

Methodology

Experimental Overview

To provide a comprehensive understanding of the experimental process and its outcomes, the following is an overview of the experiment's phases and their results.

In the initial phase of the experimental study, a basic neural network model is developed to classify input data into distinct categories. This model is then trained on a dataset, and the accuracy of the model is measured during training for both the training and validation datasets. The results indicate that the model exhibits catastrophic forgetting, which refers to the phenomenon where the model is unable to retain the knowledge learned during the training phase when presented with new data.

The second phase of the experiment involves building another neural network model based on a hypothesis. This hypothesis aims to address the issue of catastrophic forgetting by incorporating certain mechanisms into the neural network architecture. This new model is trained and tested on the same dataset as the basic model.

The accuracy and other statistical measures are then compared between the two models to see if the model based on the hypothesis is working or not. If the accuracy and other measures are significantly better for the model based on the hypothesis, it can be concluded that the hypothesis is valid and that the new model is an improvement over the basic model. On the other hand, if the results are not significantly better, the hypothesis may need to be modified or re-evaluated, and the experiment may need to be repeated.

The workflow for this experiment can be summarized as follows:

1. Data Preparation: Load the Fashion-MNIST and MNIST datasets, normalize the pixel values, and split the data into training and test sets.
2. Model Training: Develop a basic neural network model and train it on the Fashion-MNIST dataset. Evaluate the accuracy of the model during training for both the training and validation datasets. Save the validation accuracy history for Fashion-MNIST.
3. Model Testing: Evaluate the performance of the basic model on the test set of Fashion-MNIST. Save the model weights.
4. Hypothesis Development: Develop a hypothesis to address the issue of catastrophic forgetting by incorporating certain mechanisms in the neural network architecture.
5. Model Training Based on Hypothesis: Build a new neural network model based on the hypothesis and train it on the Fashion-MNIST dataset. Evaluate the accuracy of the new model during training for both the training and validation datasets.
6. Model Testing Based on Hypothesis: Evaluate the performance of the new model on the test set of Fashion-MNIST. Save the model weights.
7. Comparison of Results: Compare the accuracy and other statistical measures between the two models to see if the model based on the hypothesis is working or not. If the results are significantly better for the model based on the hypothesis, conclude that the hypothesis is valid, and the new model is an improvement over the basic model. Otherwise, modify or re-evaluate the hypothesis and repeat the experiment if necessary.
8. Statistical Analysis: Perform a statistical analysis of the results to determine if the observed differences are significant.

9. Visualization: Visualize the model weights for each layer at each epoch to observe how they change over time.
10. Conclusion: Summarize the findings and draw conclusions about the effectiveness of the hypothesis and the proposed model.

Datasets Used

In this experiment, MNIST was the dataset used.

The MNIST database, often called the Modified National Institute of Standards and Technology database, is a well-known collection of handwritten numbers that is frequently used for teaching different image processing systems and machine learning models. Since the training dataset was obtained from American Census Bureau employees while the testing dataset was obtained from American high school students, the database's creators felt that it was not suitable for machine learning experiments. As a result, they "re-mixed" the samples from NIST's original datasets to create the database. Additionally, the NIST black and white photos were anti-aliased and normalized so that they would fit inside a 28x28 pixel bounding box.

Half of each set, which together make up the MNIST database's 60,000 training and 10,000 test pictures, came from the training and testing datasets of NIST. Each picture is linked to one of the labels that are assigned to the numbers, which range from 0 to 9.

In order to conduct the experiment, the MNIST dataset is divided into two tasks. The numerals 0 through 4 are part of Task 1, whereas the digits 5 through 9 are part of Task 2. LeCun and others (1998)

By splitting the MNIST dataset into two tasks, the experiment aims to assess the ability of a neural network to learn and retain information in a sequential learning setting. Specifically, the experiment aims to investigate whether a neural network can learn task 1 (digits 0 to 4) without forgetting this information when learning task 2 (digits 5 to 9).

Fig. 7: MNIST dataset used in the experiment.
Source:https://www.tensorflow.org/datasets/catalog/mnist

Hardware and Software Implementation:

The whole process of the experiment, including the data processing, model training, testing, analysis, visualization, etc., has been done on the following machines:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro18,3
Model Number:	Z15J0017XHN/A
Chip:	Apple M1 Pro
Total Number of Cores:	8 (6 performance and 2 efficiency)
Memory:	16 GB

In this research, we utilized various software tools and environments for building and training the neural networks. In the beginning, we used the Spyder IDE for writing and debugging the code, which provided a convenient interface with features such as syntax highlighting, code

completion, and debugging, but because of some errors in the operating system, I switched to Google Colab.

Both the basic model architecture and the novel architecture with the hypothesis were implemented using a Google Colaboratory environment, which offers the convenience of running.ipynb files on Jupyter Notebook. The Google Colaboratory environment utilizes remote runtime access, which allows users to execute code on powerful machines without using their own system resources. Additionally, this access is free for a limited time (T. Carneiro et al., 2018).

To achieve optimal performance during the training, validation, and testing of both networks, the GPU of the Google Colaboratory server was utilized. This allowed for faster processing times and improved performance in deep learning implementations. By leveraging the power of the GPU, the training times for these large neural networks were significantly reduced, enabling a more efficient experimentation process.

Library and API Installation

In this research, the Python programming language was the primary tool used for implementing the machine learning models and conducting data analysis. Python is widely used among data scientists and machine learning practitioners because of its simplicity, readability, and the large number of available libraries that facilitate data manipulation, visualization, and modeling.

The specific version of Python used in this research was 3.9.7. This version was chosen due to its stability, security, and compatibility with the various libraries and tools used in the research. It is crucial to note that various versions of Python may have subtle changes in the

syntax and functionality, and as a result, it is essential to make sure that you select a version that is suitable with the research project you are working on.

The choice of Python as the primary programming language for this research provides several advantages, such as ease of use, readability, and a large community of developers who contribute to its development and maintenance. Additionally, the availability of numerous libraries for machine learning, such as TensorFlow, Keras, and Scikit-Learn, made it possible to easily implement and experiment with different machine learning models. The following sections provide an overview of these tools and the installation process.

Libraries	**Version**
tensorflow	2.12.0
keras	2.12.0
scipy	1.10.1
numpy	1.22.4
pip	23.1.2
matplotlib	3.7.1

- Tensorflow and Keras are deep learning libraries for building and training neural networks.
- mnist from keras.datasets is used to load the MNIST dataset.
- Sequential from keras.models is a type of neural network model where layers are stacked sequentially on top of each other.
- Dense and Flatten from keras.layers are different types of layers used in the model architecture.

- Adam from keras.optimizers is an optimization algorithm used to update the weights of the neural network during training.
- K from keras.backend provides backend support for TensorFlow operations.
- Matplotlib.Pyplot is a plotting library used to visualize the results.
- Numpy is a library for working with arrays and mathematical operations on arrays.
- The scipy.stats module is part of the scipy library, which is a Python library used for scientific and technical computing. The scipy.stats module provides a large number of probability distributions and statistical functions, including continuous and discrete probability distributions, statistical tests, and descriptive statistics. It allows for easy calculation of various statistical metrics and tests and is commonly used in data analysis, machine learning, and scientific research.

Network used in the experiment:

A multilayer perceptron (MLP) is a sort of feedforward artificial neural network made up of several layers of sequentially organized neurons, also known as nodes. Each neuron in a layer gets inputs from the layer below, adds the inputs together while applying an activation function, and then sends the output to the layer below.

Three layers are commonly included in an MLP: the input layer, one or more hidden levels, and the output layer. The network's output layer is made up of neurons that create the network's output, while the input layer is made up of neurons that accept input data. The hidden layers are intermediary layers that process incoming data between the input and output levels.

The dimensionality of the input data determines the number of neurons in the input layer,

while the kind of job the network is intended to carry out determines the number of neurons in the output layer. For instance, the output layer of a classification job with n classes will contain n neurons, each of which represents a class and generates a probability distribution across the classes.

The activation function used in each neuron can be a sigmoid function, a hyperbolic tangent function, or a rectified linear unit (ReLU) function. The weights between the neurons in each layer are learned through a process called backpropagation, which uses gradient descent to minimize a cost function that measures the error between the predicted output and the actual output.

The Basic Model Architecture

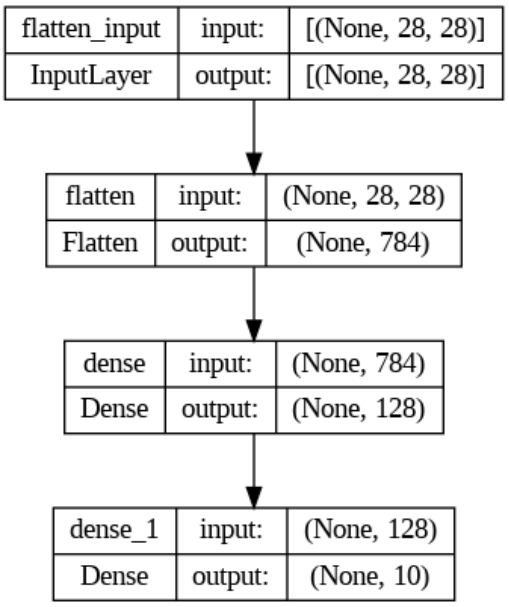


Fig. 8: :Architecture of the Network Used in the Experiment, Source: Code Used in the Experiment

A 28x28 picture that has been compressed into a 784-dimensional vector serves as the experiment's input layer. The first hidden layer, which consists of 128 units with the ReLU activation function and 10 units with the softmax activation function, generates the probability distribution across 10 classes.

Model 1 (with a fixed learning rate)

The code trains a neural network model to perform two different classification tasks sequentially on the MNIST dataset. The MNIST dataset contains images of handwritten digits, and each image is labeled with the corresponding digit from 0 to 9.

The first task is to classify images into five classes: 0, 1, 2, 3, or 4. The second task is to classify images into the remaining five classes: 5, 6, 7, 8, or 9.

The code splits the MNIST dataset into two parts for the two tasks. It also normalizes the pixel values of the images to be between 0 and 1.

The neural network model used in the code is a simple one with two layers. The first layer is a flattening layer that converts the two-dimensional image data into a one-dimensional vector. The second layer is a dense layer with 128 units and a ReLU activation function. The final layer is another dense layer with 10 units and a softmax activation function, which outputs the probability of each class.

The code compiles the model with the Adam optimizer and sparse categorical cross-entropy loss function. It also evaluates the initial accuracy of the model on both tasks using the evaluate method.

After compiling the model, the code saves the initial weights of the model and then trains the model on the first task for ten epochs using the fit method. The code saves the weights of the model after each epoch and also records the validation accuracy of the model on both tasks after each epoch.

After training on the first task, the code saves the weights of the model and then loads the saved weights to initialize the model for training on the second task. The code then trains the model on the second task for a few epochs, saves the weights of the model after each epoch, and records the validation accuracy on both tasks after each epoch.

In this model, the learning rate is fixed at 0.001. The learning rate controls the step size that the optimizer takes during gradient descent in order to update the model's weights.

The model is compiled with certain configurations. Here is what each argument does:

Optimization:During training, the weights of the model are updated using an optimization method called the optimizer. In this instance, the adaptive learning rate optimization technique known as the Adam optimizer, which is often used in deep learning, is set as the default.

Loss: During training, the model will attempt to minimize the objective function. In this instance, we use the widely used "sparse categorical crossentropy" loss function for multiclass classification issues.

Metrics: These are the evaluation metrics that the model will use to determine how well it is performing during training and testing. In this instance, the evaluation measure is "accuracy."

Equations used in the model:

The Adam optimizer is a stochastic gradient descent optimization algorithm that computes adaptive learning rates for each parameter. It combines the advantages of two other SGD extensions: the Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSprop).

The Adam optimizer uses the following equations to update the parameters during training:

1. Compute gradient: Calculate the gradient of the loss function with respect to the model parameters.
2. Compute momentum: Update the moving average of the first and second moments of the gradients. The momentum is computed separately for each parameter.

- First moment: the exponentially weighted average of the gradients (also known as the mean)
- Second moment: the exponentially weighted average of the squared gradients

3. Update parameters: Adjust the parameters based on the gradients and momentum.
    - Calculate the bias-corrected first and second moments.
    - Update the parameters by subtracting the learning rate times the first moment, dividing by the square root of the second moment, and adding a small epsilon value to prevent division by zero.

The equations are as follows:

- g_t Gradient of the objective function at time t
- β_1, β_2 Exponential decay rates for the first and second moment estimates
- $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
- $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
- $m_t^{corrected} = \frac{m_t}{1-\beta_1^t}$
- $v_t^{corrected} = \frac{v_t}{1-\beta_2^t}$
- $\theta_t = \theta_{t-1} - \alpha \cdot \frac{m_t^{corrected}}{\sqrt{v_t^{corrected}}+\epsilon}$

where:

- θ_t= model parameters at time t
-
- v_t = second-moment estimate (uncentered variance) at time t
- α learning rate
- ϵ = small value to avoid division by zero

The Adam optimizer updates the parameters using these equations in each training iteration, with the gradient of the loss function being calculated for each batch of training data.

In this case, the learning rate has an entropy of 0.001, which means that the optimizer will update the network weights by taking small steps of size 0.001 in the direction of the gradient. This small step size ensures that the optimizer converges slowly and helps prevent overshooting the optimal weights.

During training, the optimizer calculates the gradient of the loss function with respect to the network weights and then updates the weights based on the magnitude of the gradient and the learning rate. (Kingma & Ba, J. 2014)

Equation for Sparse Categorical Cross-Entropy

When performing multiclass classification tasks where the classes are mutually exclusive, the sparse categorical cross-entropy loss function is frequently used. The sparse categorical cross-entropy calculates the difference between each sample's real probability distribution and anticipated probability distribution given a set of N samples and K potential classes.

$$L = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} log(p_{ij})$$

where:

- N is the number of samples in the dataset.
- C is the number of classes.
- y_{ij} is a binary indicator (0 or 1) for whether class j is the true class for sample i.
- p_{ij} is the predicted probability that sample i belongs to class j.

Note that the indicator y_{ij} is sparse since it only takes the value 1 for the true class of each sample and 0 for all other classes. Hence the name "sparse" categorical cross-entropy.

The loss L is minimized when the predicted probability distribution for each sample matches the true distribution. This loss function is commonly used in multiclass classification tasks where the classes are mutually exclusive.

Model 2 (variable learning rate)

Keeping the rest of the model the same as above, we introduced a special optimizer class named "Modified Learning Rate Optimizer" in order to investigate a different learning strategy. This new optimizer, which derives from the well-known "keras.optimizers.Adam" class while introducing significant changes to the learning rate, maintains the majority of the model's architecture.

The method through which we determine the learning rate for each parameter has been modified most significantly. We now calculate the learning rate based on the absolute value of the parameter itself rather than a preset number. We generate a dynamic learning rate that changes in response to the magnitude of the parameter by using the formula new lr = 1 - K. tanh (5*x), where x is the absolute value of the parameter.

This adaptive learning rate calculation seeks to reconcile stability in areas where the parameter values are larger with sensitivity to major parameter adjustments. The hyperbolic tangent function (tanh), which is a non-linear activation, reduces the learning rate to a value between -1 and 1.

The optimizer then updates the parameter values after figuring out the new learning rate for each parameter. The new parameter value is obtained by subtracting the gradient from the product of the new learning rate. By making these changes to the parameters, we want to lead the optimization process in a way that makes use of the knowledge gleaned from the changed learning rate.

Our specialized optimizer will hopefully open up a fresh way of looking at neural network optimization. We want to improve the model's capability to cope with different magnitudes of updates by taking into consideration the absolute values of parameters and modifying the learning rate appropriately. This method may enhance convergence and fine-tuning in challenging optimization environments.

The ability to tailor optimizers gives us the chance to improve the training process and find fresh methods for enhancing model performance as we continue to investigate many directions in the area of machine learning.

Results

In this section, we present the findings of our study, which aimed to investigate the question of whether we can effectively stabilize weights with high magnitudes in the context of our model. Through a thorough analysis of the data collected from two distinct models, each subjected to 10 epochs of training, we have explored several key aspects pertaining to the behavior and performance of the models.

To ensure a comprehensive evaluation, we employed two separate models, each with its own distinctively analyzed statistical measures for results. By comparing the results from these models, we aimed to elucidate the effectiveness of weight stabilization methods.

Model 1

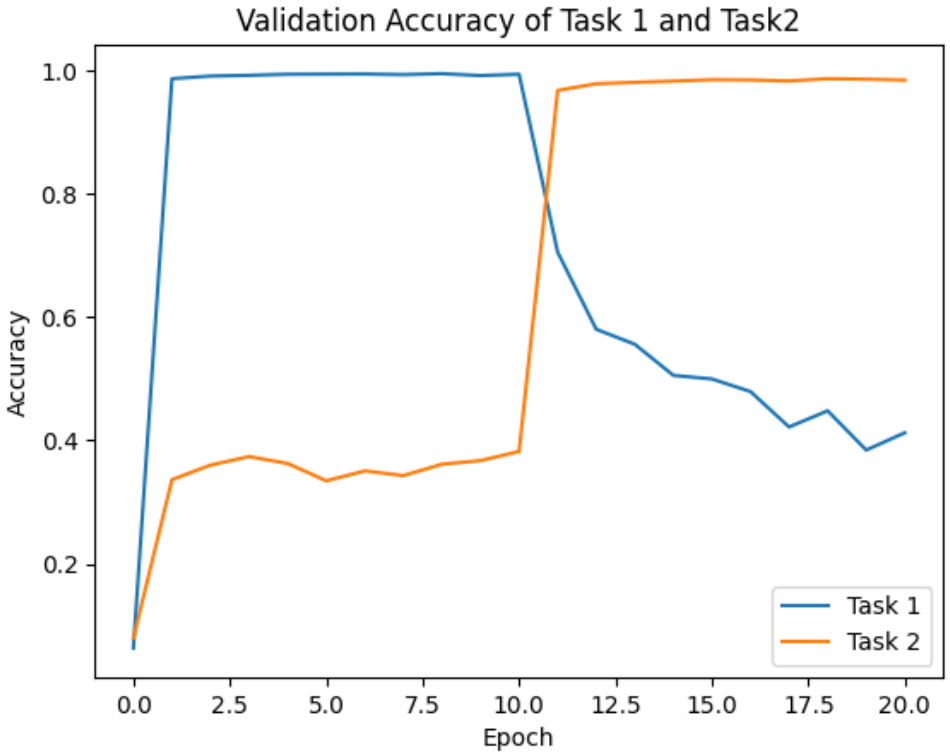


Fig. 9: Accuracy of tasks 1 and 2 obtained from model 1

Upon analyzing the accuracy plot of our model 1, we observed a clear pattern of catastrophic forgetting. Initially, as the model was trained on the initial task, it achieved high accuracy, effectively capturing the underlying patterns in the data. However, as subsequent tasks were introduced, the model's accuracy on previously learned tasks dramatically decreased, indicating a case of catastrophic forgetting.

Model 2

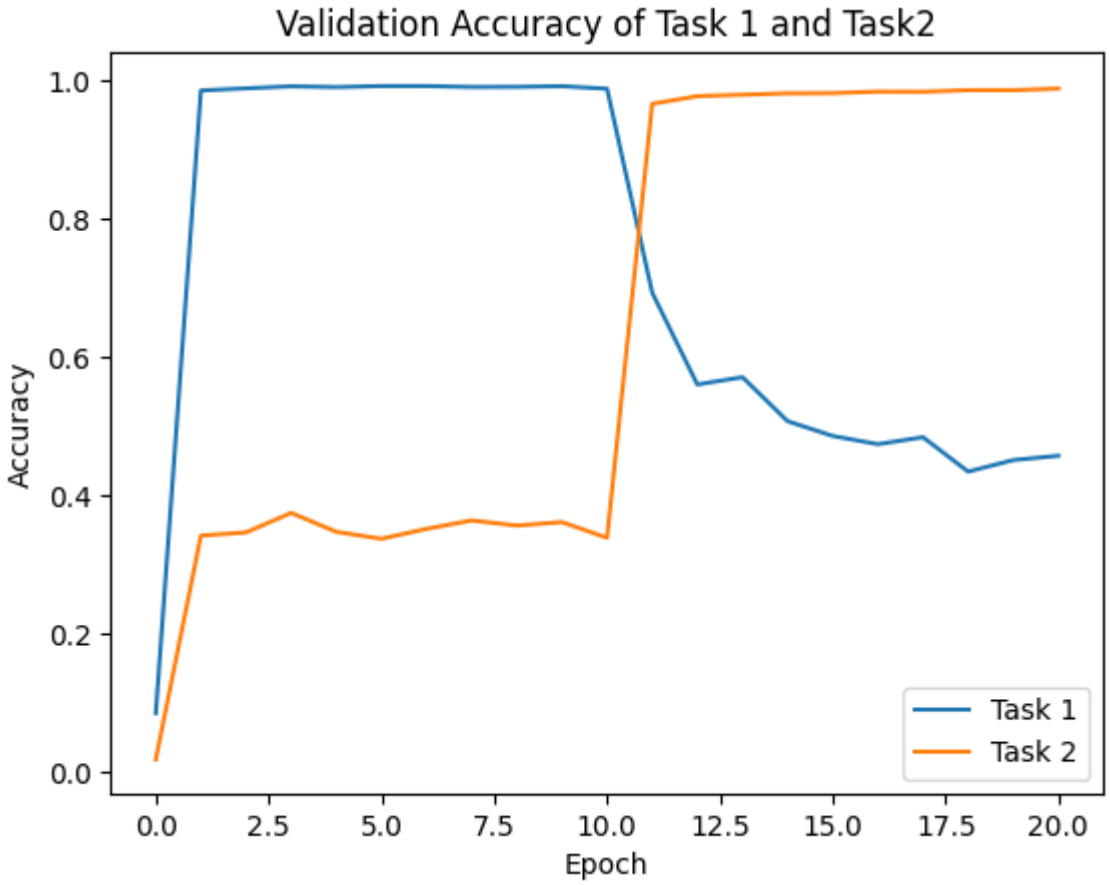


Fig. 10: Accuracy of tasks 1 and 2 obtained from model 2

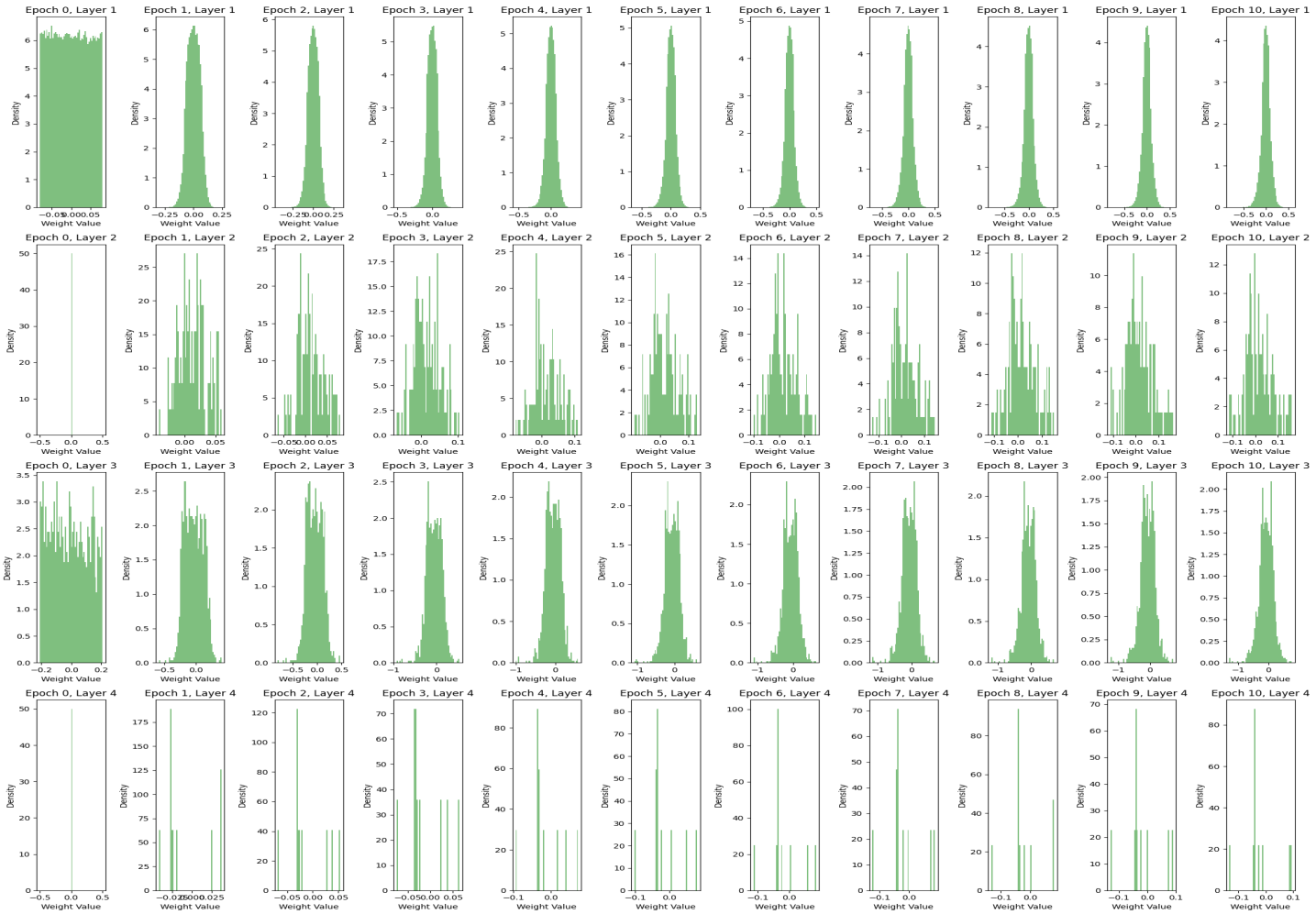
Figure 10 presents the accuracy plot, which displays the model's performance over multiple epochs. As depicted in the figure, the accuracy initially exhibited a steady increase up to 10 epochs, indicating the model's ability to learn and improve its predictions. However, there was a sudden decline in accuracy around the halfway point of training, and then there was a period of fluctuation.

After comparing the accuracy plots of both models, it became apparent that there was no significant improvement in mitigating catastrophic forgetting. This outcome suggests that the lack of improvement could be attributed to either an incorrect hypothesis or an inadequate implementation of the hypothesis within the network.

One possible explanation for the lack of improvement is that the hypothesis itself was not correct. The assumption that the modified learning rate would effectively address catastrophic forgetting and lead to improved performance may not hold true in the context of our study. This highlights the importance of critically evaluating the initial hypothesis and considering alternative explanations for the observed outcomes.

Another potential reason for the limited success in mitigating catastrophic forgetting could be an insufficient implementation of the hypothesis within the network. Factors such as the specific configuration of the modified learning rate, the choice of hyperparameters, or the interaction between the modification and other components of the network architecture might have played a role in the suboptimal performance.

Weights Distributions Model 1



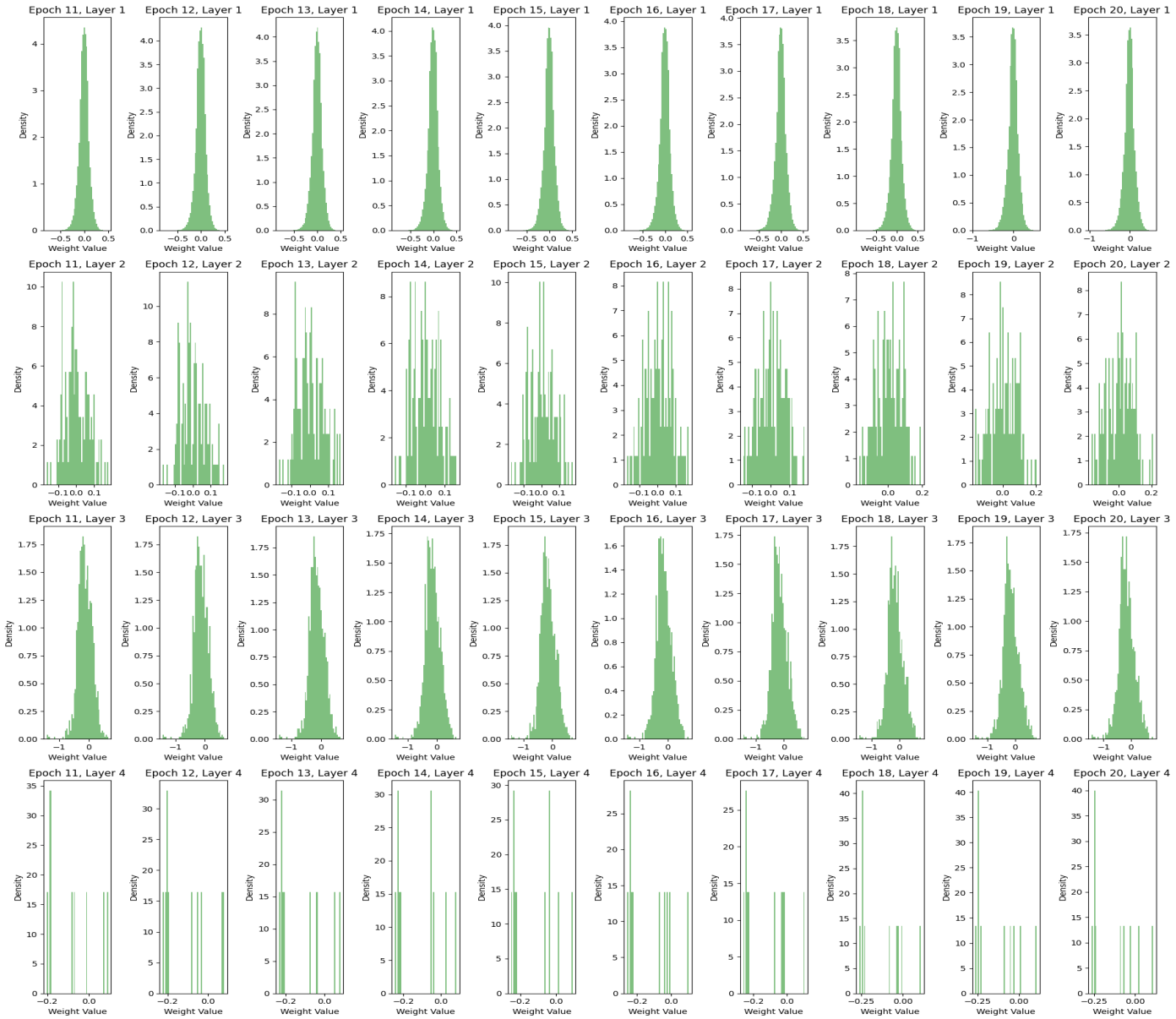
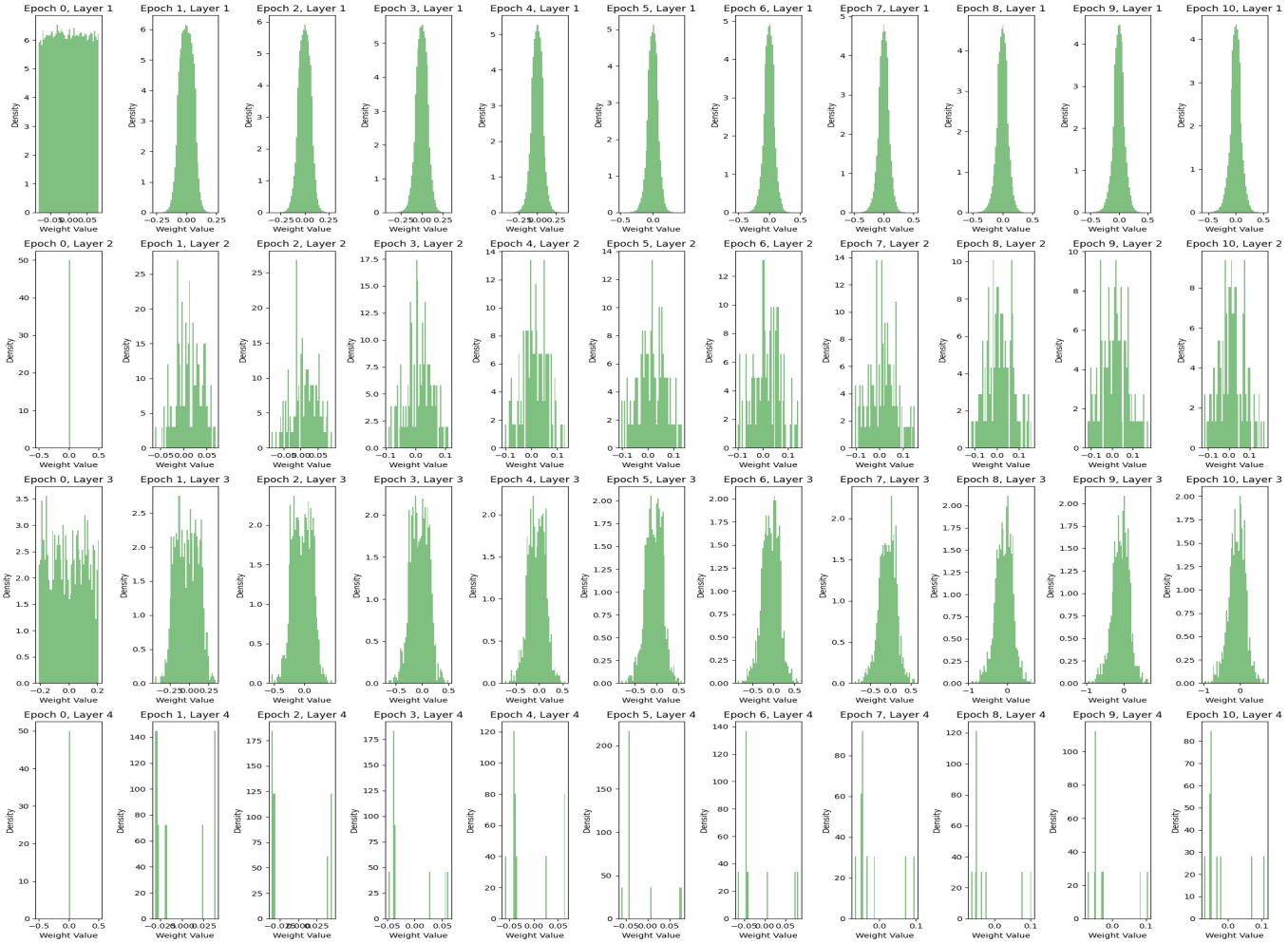


Fig. 11: Weight distribution of each layer over the epochs for Model 1 (this was plotted for fair comparison of weight distribution and for the calculation of skewness and kurtosis)

Weight Distribution Model 2



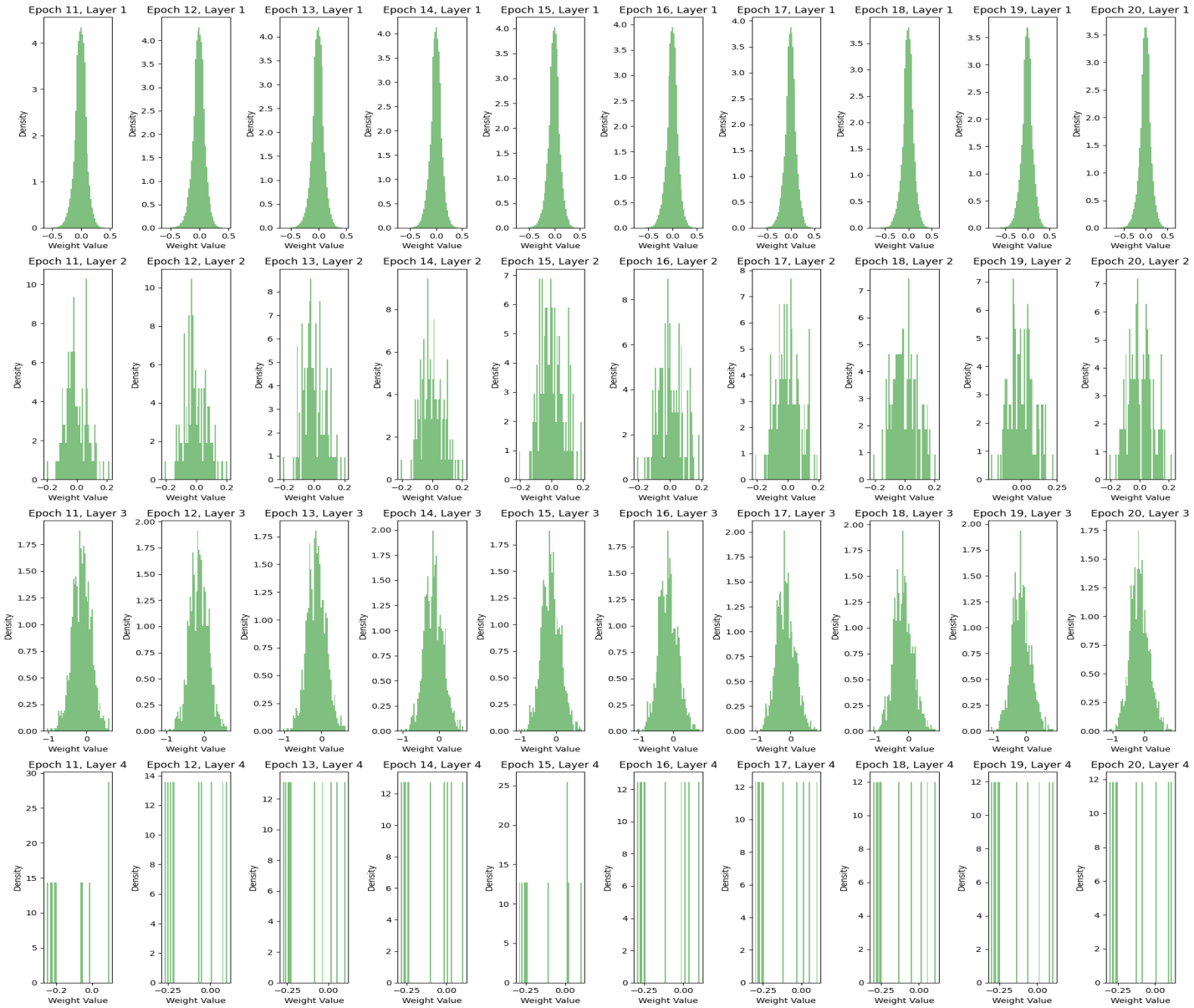


Fig. 12: Weight distribution of each layer over the epochs for Model 2 (this was plotted for fair comparison of weight distribution and for the calculation of skewness and kurtosis)

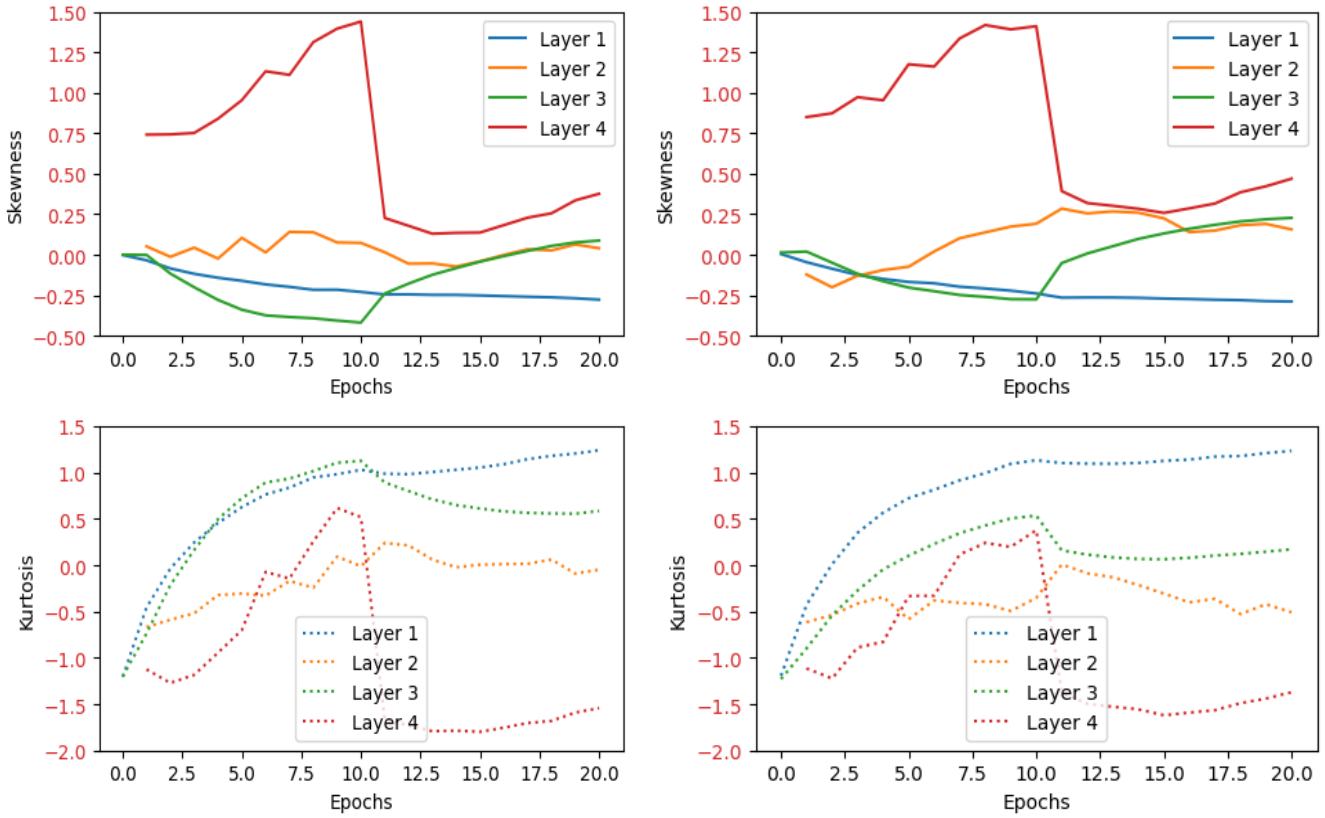


Fig. 13: Illustration of skewness and kurtosis for models 1 and 2. On the left is model 1 and on the right is model 2.)

It became apparent that the hypothesis had not been applied appropriately after further examination of statistical indicators such as skewness and kurtosis for both models over traile of training. These metrics are helpful because they provide information about the behavior and properties of the network, namely the distribution and shape of the weight values.

We hypothesized that the adjusted learning rate, which was based on the absolute value of the parameters, would lead to a distribution with reduced skewness and a negative kurtosis by stabilizing the weights. However, our investigation revealed that the hypothesis's predicted outcomes did not materialize.

In particular, we discovered that the skewness of the weight distribution did not show the predicted decrease when compared to the original model. Similarly, negative values for kurtosis, which would have indicated a flatter and less peaked distribution, were not seen in the weight data. Instead, the kurtosis values maintained their elevated levels relative to the baseline model, indicating that the weight distribution does not support the null.

Here we present the findings of our study, which aimed to investigate the regularization of weights with high magnitudes in our model. Despite our efforts, we were unable to effectively achieve weight regularization within the scope of this study. We conducted an in-depth analysis of how each regularization technique affected the model's weight distribution and overall performance indicators during the trial period. To determine if the regularization methods were successful, we kept a careful eye on metrics including accuracy, skewness, and kurtosis.

The strategies we used were not successful in stabilizing the weights of high magnitudes, despite our extensive testing and study. Regularization attempts had little effect on the model's performance, and the weights maintained their significant magnitudes.

It's possible that several other things played a role in causing this result. It is possible that the complexity of the network design, the features of the dataset, or the interplay of other training parameters altered the efficiency of the changed learning rate in obtaining the necessary weight stabilization and modifying the distribution properties.

These results demonstrate the difficulty with which innovative neural network techniques and theories must be implemented. It is essential to understand the constraints and modify research methods accordingly. To further address the problems seen in this study, future research might look at different approaches or hone the hypothesis.

In conclusion, the skewness and kurtosis analysis showed that the modified learning rate

hypothesis was not realized in the network. These findings shed light on the challenges inherent in weight stabilization methods and suggest avenues for further exploration and methodological improvement in the field of neural networks.

Discussion

The purpose of this research was to test the idea that changing the learning rate at synapses in artificial neural networks, especially lowering it when synaptic strength converges to a high magnitude, will increase network performance. In this discussion, we will look at the weaknesses in the hypothesis and probable errors in its execution, as well as other circumstances that may have led to the absence of meaningful findings.

The hypothesis predicted that slowing down the learning rate as synaptic strength increased would improve network performance in general. It did not, however, take into account the intricate link between synapse strength and network function. While increased synaptic strength may suggest poor network functioning, this is not necessarily the case. Strong synapses are critical for maintaining robust and precise neural network operation, especially in activities requiring high signal-to-noise ratios or persistent memory storage. This omission in the hypothesis design might have led to mistaken expectations and damaged the hypothesis's validity.

Furthermore, the idea oversimplifies the complex link between brain network learning rates and synaptic plasticity processes. It ignored the temporal components of learning, the effects of other network parameters, and the non-linear interactions that occur throughout the learning process by just altering the learning rate depending on synapse strength. These

omissions compromised the hypothesis' usefulness by undermining its ability to reflect the dynamic nature of learning and adaptation.

Moving on to the hypothesis's execution, many faults may have led to the absence of substantial outcomes. One key problem was the failure to account for network architectural complexity. Modifying the learning rate at a single synapse may have had limited influence on the entire network's behavior since artificial neural networks contain a complicated and linked architecture. A more in-depth examination of the network design and its implications for the hypothesis might have yielded useful insights into the link between learning rate modification and network performance.

The implementation may have overlooked important components within the chosen optimizer, such as the Adam optimizer, which incorporates adaptive learning rate techniques by considering momentum, second-moment estimation, and hyperparameters. Neglecting to consider the interactions between these components and the modified learning rate could have affected the effectiveness of the implementation. Future studies should explore these interactions to optimize the approach's performance and convergence behavior. Although neurological studies that link learning and brain volume in biological systems served as the inspiration for this research, it is crucial to acknowledge the challenges of directly applying these findings to artificial neural networks. Biological studies shed light on learning and adaptation, but artificial neural networks lack the sophisticated biological machinery and extensive interconnectedness present in the human brain.

According to biological research, learning causes structural changes in the brain, such as increased synaptic connections, dendritic development, and neurogenesis. These modifications lead to better cognitive capacities and information processing. Although the idea was founded on these discoveries, owing to the fundamental differences between

biological and artificial systems, the translation of these notions to artificial neural networks required careful attention.

Conclusion

In conclusion, this study aimed to investigate the hypothesis that lowering the learning rate of synapses when their strength converges to a large magnitude would enhance the performance of artificial neural networks. However, due to implementation flaws and unforeseen challenges, the results do not provide conclusive evidence to either support or disprove the hypothesis.

The findings of this study highlight the importance of careful implementation and consideration of various factors in modifying the learning rate of synapses. It is evident that the model did not perform as expected due to implementation flaws, which may have hindered the accurate evaluation of the hypothesis. Therefore, caution should be exercised in drawing definitive conclusions regarding the validity of the hypothesis.

While the hypothesis itself may have inherent limitations, such as oversimplification of the complex relationship between synaptic strength and network performance, it is important to acknowledge the potential impact of implementation errors on the outcomes. The limitations identified in this study, including the neglect of network architecture complexity and failure to account for crucial optimizer components, warrant further investigation and refinement in future research.

Additionally, the study highlights the challenges of translating findings from neurological research, which indicate a correlation between learning and brain volume in biological systems, to artificial neural networks. The fundamental differences between biological and

artificial systems necessitate a cautious and nuanced approach to applying concepts derived from neural studies to artificial neural networks.

In conclusion, while the results of this study do not provide strong evidence in support of the hypothesis, it is crucial to recognize the limitations imposed by the implementation flaws. To learn more about how changing the learning rate affects network performance in artificial In neural networks, more research needs to be done to improve implementation, fix the problems that have been found, and get a better understanding of network dynamics and optimizer parts.

References

1. McCloskey, Michael, and Neal J. Cohen. "Catastrophic interference in connectionist networks: The sequential learning problem." Psychology of learning and motivation. Vol. 24. Academic Press, 1989, pp. 109-165.
2. Zenke, F., Poole, B., & Ganguli, S. (2017). Continual learning through synaptic intelligence. Proceedings of the International Conference on Machine Learning, 3987-3995.
3. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.
4. Maguire, E. A., Gadian, D. G., Johnsrude, I. S., Good, C. D., Ashburner, J., Frackowiak, R. S., & Frith, C. D. (2000). navigation-related structural change in the hippocampi of taxi drivers. Proceedings of the National Academy of Sciences, 97(8), 4398-4403. doi: 10.1073/pnas.070039597
5. Zenke, F., Poole, B., & Ganguli, S. (2017). Continual learning through synaptic intelligence. In Proceedings of the 34th International Conference on Machine Learning-Volume 70 (pp. 3987-3995). JMLR.
6. Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. Proceedings of the National Academy of Sciences, 114(13), 3521-3526.
7. Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., & Tuytelaars, T. (2018). Memory aware synapses: learning what not to forget. in the European Conference on Computer Vision (pp. 139-154). Springer, Cham.
8. Masse, N., Grant, A., & Caruana, R. (2018). Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization. Proceedings of the AAAI Conference on Artificial Intelligence, 32(1).
9. Bellec, G., Kappel, D., Lundqvist, M., Maass, W., & Legenstein, R. (2019). A solution to the learning dilemma for recurrent networks of spiking neurons. Nature Communications, 10(1), 1-14.
10. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444. doi: 10.1038/nature14539.

11. Munoz-Martin, Irene & Bianchi, Stefano & Pedretti, Giacomo & Melnic, Octavian & Ambrogio, Stefano & Ielmini, D.. (2019). Unsupervised Learning to Overcome Catastrophic Forgetting in Neural Networks. IEEE Journal on Exploratory Solid-State Computational Devices and Circuits. PP. 1-1. 10.1109/JXCDC.2019.2911135.
12. Maguire, E. A., Gadian, D. G., Johnsrude, I. S., Good, C. D., Ashburner, J., Frackowiak, R. S., & Frith, C. D. (2000). Navigation-related structural change in the hippocampi of taxi drivers. Proceedings of the National Academy of Sciences, 97(8), 4398-4403. https://doi.org/10.1073/pnas.070039597
13. Schmidt, R., De Simoni, S., Cortese, R., Lennartsson, A., Veronese, M., Vadini, F., ... & Nichols, T. E. (2021). Gray matter changes associated with deficit-driven use-dependent plasticity are attenuated by experience. Proceedings of the National Academy of Sciences, 118(10), e2016039118. https://doi.org/10.1073/pnas.2016039118
14. Ott, S. R., Rogers, S. M., Greggers, U., Menzel, R., & Galizia, C. G. (2010). Formation of short-term memory and processing of multiple stimuli by the mushroom bodies of the honeybee (Apis mellifera): localization, physiology, and behavior. Journal of Comparative Physiology A, 196(6), 485-497. https://doi.org/10.1007/s00359-010-0535-
15. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
16. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.
17. van de Ven, Gido M., Hava T. Siegelmann, and Andreas S. Tolias. "Brain-inspired replay for continual learning with artificial neural networks." Nature communications 11.1 (2020): 1-14.
18. //www.tensorflow.org/datasets/catalog/mnist

Appendix

Code for Model 1

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
import tensorflow.keras.backend as K
import matplotlib.pyplot as plt
import numpy as np

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Split the dataset into two tasks
task1_classes = [0, 1, 2, 3, 4]
task2_classes = [5, 6, 7, 8, 9]
task1_train_idx = np.isin(y_train, task1_classes)
task1_test_idx = np.isin(y_test, task1_classes)
task2_train_idx = np.isin(y_train, task2_classes)
task2_test_idx = np.isin(y_test, task2_classes)
x_train_task1,      y_train_task1      =      x_train[task1_train_idx],
y_train[task1_train_idx]
x_test_task1,       y_test_task1       =       x_test[task1_test_idx],
y_test[task1_test_idx]
x_train_task2,      y_train_task2      =      x_train[task2_train_idx],
y_train[task2_train_idx]
x_test_task2,       y_test_task2       =       x_test[task2_test_idx],
y_test[task2_test_idx]
y_train_task2 = y_train_task2 - 5
y_test_task2 = y_test_task2 - 5


# Normalize pixel values
x_train_task1 = x_train_task1.astype('float32') / 255.
x_test_task1 = x_test_task1.astype('float32') / 255.
x_train_task2 = x_train_task2.astype('float32') / 255.
x_test_task2 = x_test_task2.astype('float32') / 255.

# Define neural network model
model = Sequential([
```

```
 Flatten(input_shape=(28, 28)),
 Dense(128, activation='relu'),
 Dense(10, activation='softmax')
])

lr=0.001
optimizer = Adam(lr=lr)
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Get the initial weights of the model
weights = model.get_weights()

test_loss_task1,    test_acc_task1    =    model.evaluate(x_test_task1,
y_test_task1, verbose=2)
print("Task 1 Accuracy at epoch 0:", test_acc_task1)

test_loss_task2,    test_acc_task2    =    model.evaluate(x_test_task2,
y_test_task2, verbose=2)
print("Task 2 Accuracy at epoch 0:", test_acc_task2)

# Save the model weights after each epoch
task1_val_acc = [test_acc_task1]
weights_list = [weights]
accuracy_list = []
task2_val_acc = [test_acc_task2]
for epoch in range(10):
     history_task1 = model.fit(x_train_task1, y_train_task1, epochs=1,
validation_data=(x_test_task1, y_test_task1))
        accuracy_task1   =   model.evaluate(x_test_task1,   y_test_task1,
verbose=0)[1]
        accuracy_task2   =   model.evaluate(x_test_task2,   y_test_task2,
verbose=0)[1]
   accuracy_list.append([accuracy_task1, accuracy_task2])
     print("Epoch {}, Accuracy on Task 1: {}, Accuracy on Task 2:
{}".format(epoch+1, accuracy_task1, accuracy_task2))

   weights_list.append(model.get_weights())

   task1_val_acc.append(accuracy_task1)
   task2_val_acc.append(accuracy_task2)
```

```
# Plot the weight distribution of each layer for each epoch
num_layers = len(weights_list[0])
fig,  axs  =  plt.subplots(nrows=num_layers,  ncols=len(weights_list),
figsize=(20, 20))
for i in range(len(weights_list)):
   for j in range(num_layers):
       w = weights_list[i][j].flatten()
                axs[j,  i].hist(w,  bins=50,  alpha=0.5,color='green',
density=True)
       axs[j, i].set_title(f"Epoch {i}, Layer {j+1}")
       axs[j, i].set_xlabel("Weight Value")
       axs[j, i].set_ylabel("Density")
plt.tight_layout()
plt.show()

# Save the model weights
model.save_weights('task1_model_epoch_4.h5')
#Load weights
model.load_weights('task1_model_epoch_4.h5')

optimizer = Adam(lr=lr)
model.compile(optimizer=optimizer,
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])

# Initialize the list to save the validation accuracy for MNIST
weights_list_task2=[]
for epoch in range(10):
              model.fit(x_train_task2,     y_train_task2,     epochs=1,
validation_data=(x_test_task2, y_test_task2))
       accuracy_task1  =  model.evaluate(x_test_task1,  y_test_task1,
verbose=0)[1]
       accuracy_task2  =  model.evaluate(x_test_task2,  y_test_task2,
verbose=0)[1]
   accuracy_list.append([accuracy_task1, accuracy_task2])
    print("Epoch {}, Accuracy on Task 1: {}, Accuracy on Task 2:
{}".format(epoch+1, accuracy_task1, accuracy_task2))

    # Save the validation accuracy for Fashion-MNIST and MNIST during
this epoch
   task1_val_acc.append(accuracy_task1)
   task2_val_acc.append(accuracy_task2)
```

```
    # Get the weights of the model
    weights = model.get_weights()
    weights_list_task2.append(model.get_weights())

# Plot the weight distribution of each layer for each epoch
num_layers = len(weights_list_task2[0])
fig,  axs  =  plt.subplots(nrows=num_layers,  ncols=len(weights_list),
figsize=(20, 20))
for i, weights in enumerate(weights_list_task2):
    for j in range(num_layers):
                        axs[j,   i].hist(weights[j].flatten(),   bins=50,
alpha=0.5,color='green', density=True)
        axs[j, i].set_title(f"Epoch {i+11}, Layer {j+1}")
        axs[j, i].set_xlabel("Weight Value")
        axs[j, i].set_ylabel("Density")
plt.tight_layout()
plt.show()
```

Plotting Accuracy

```
import matplotlib.pyplot as plt
plt.plot(task1_val_acc, label='Task 1')
plt.plot(task2_val_acc, label='Task 2')
plt.title('Validation Accuracy of Task 1 and Task2')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

For calculation of skewness and kurtosis

```
# Combine the weights_list and weights_list_mnist
weights_list.extend(weights_list_task2)

import scipy.stats as stats

weights_array  =  np.array(weights_list)   # convert list of weights to
numpy array
num_epochs,  num_layers  =  weights_array.shape[:2]    #  get  number  of
epochs and layers
skewness = np.zeros((num_epochs, num_layers))
```

```
kurtosis = np.zeros((num_epochs, num_layers))

for i in range(num_epochs):
   for j in range(num_layers):
         weights = weights_array[i, j].flatten()  # flatten weights for
this layer and epoch
       skewness[i, j] = stats.skew(weights)
       kurtosis[i, j] = stats.kurtosis(weights)

skewness_list = []
kurtosis_list = []

for i in range(len(weights_list)):
   for j in range(len(weights_list[0])):
       weights = weights_list[i][j].flatten()
       skew = scipy.stats.skew(weights)
       kurtosis = scipy.stats.kurtosis(weights)
       skewness_list.append(skew)
       kurtosis_list.append(kurtosis)
         print("Epoch {}, Layer {}: Skewness={}, Kurtosis={}".format(i,
j+1, skew, kurtosis))

print("Skewness values:", skewness_list)
print("Kurtosis values:", kurtosis_list)

# Separate kurtosis values by layer
num_layers = len(weights_list[1])
kurtosis_by_layer = [[] for _ in range(num_layers)]
for i in range(len(kurtosis_list)):
   layer_index = i % num_layers
   kurtosis_by_layer[layer_index].append(kurtosis_list[i])

print("Kurtosis values by layer:", kurtosis_by_layer)

# Separate skewness values by layer
num_layers = len(weights_list[0])
skewness_by_layer = [[] for _ in range(num_layers)]
for i in range(len(skewness_list)):
   layer_index = i % num_layers
   skewness_by_layer[layer_index].append(skewness_list[i])

print("Skewness values by layer:", skewness_by_layer)
```

```
# Set color palette
colors = ["tab:blue", "tab:orange", "tab:green", "tab:red"]

# Plot skewness and kurtosis values for each layer
num_layers = len(skewness_by_layer)
epochs = range(len(skewness_by_layer[0]))
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(5, 6))

for layer in range(num_layers):
    color = colors[layer % len(colors)]
    ax1.plot(epochs, skewness_by_layer[layer], color=color, label="Layer
{}".format(layer+1))
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Skewness")
    ax1.tick_params(axis='y', labelcolor=color)
    ax1.set_ylim([-0.5, 1.5])  # set y-axis limits here

        ax2.plot(epochs,  kurtosis_by_layer[layer],  linestyle='dotted',
color=color, label="Layer {}".format(layer+1))
    ax2.set_xlabel("Epochs")
    ax2.set_ylabel("Kurtosis")
    ax2.tick_params(axis='y', labelcolor=color)
    ax2.set_ylim([-2, 1.5])  # set y-axis limits here

ax1.legend()
ax2.legend()
plt.tight_layout()
plt.show()
```

Code for Model 2

Data preparation

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
import tensorflow.keras.backend as K
```

```
import matplotlib.pyplot as plt
import numpy as np

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Split the dataset into two tasks
task1_classes = [0, 1, 2, 3, 4]
task2_classes = [5, 6, 7, 8, 9]
task1_train_idx = np.isin(y_train, task1_classes)
task1_test_idx = np.isin(y_test, task1_classes)
task2_train_idx = np.isin(y_train, task2_classes)
task2_test_idx = np.isin(y_test, task2_classes)
x_train_task1,      y_train_task1      =      x_train[task1_train_idx],
y_train[task1_train_idx]
x_test_task1,       y_test_task1       =       x_test[task1_test_idx],
y_test[task1_test_idx]
x_train_task2,      y_train_task2      =      x_train[task2_train_idx],
y_train[task2_train_idx]
x_test_task2,       y_test_task2       =       x_test[task2_test_idx],
y_test[task2_test_idx]
y_train_task2 = y_train_task2 - 5
y_test_task2 = y_test_task2 - 5


# Normalize pixel values
x_train_task1 = x_train_task1.astype('float32') / 255.
x_test_task1 = x_test_task1.astype('float32') / 255.
x_train_task2 = x_train_task2.astype('float32') / 255.
x_test_task2 = x_test_task2.astype('float32') / 255.
```

Implementation of the Hypothesis(Per parameter learning rate calculation as per the magnitude)

```
# Define the custom optimizer with modified learning rate equation
class ModifiedLearningRateOptimizer(keras.optimizers.Adam):
    def __init__(self, initial_lr=0.001, **kwargs):
        super().__init__(**kwargs)
        self.initial_lr = initial_lr

    def get_updates(self, loss, params):
        grads = self.get_gradients(loss, params)
```

```
        self.updates = [K.update_add(self.iterations, 1)]
         sum_weights = np.sum(K.abs(p) for p in params if 'bias' not in
p.name)
        for p, g in zip(params, grads):
            if 'bias' not in p.name:
                x = K.abs(p)
                new_lr = 1 - K.tanh(5*x)
                self.updates.append(K.update(p, p - new_lr * g))
        return self.updates

# Define neural network model
model = Sequential([
 Flatten(input_shape=(28, 28)),
 Dense(128, activation='relu'),
 Dense(10, activation='softmax')
])

# Compile the model using the custom optimizer
model.compile(optimizer=ModifiedLearningRateOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Get the initial weights of the model
weights = model.get_weights()
```

Compilation and training of model

```
test_loss_task1,    test_acc_task1    =    model.evaluate(x_test_task1,
y_test_task1, verbose=2)
print("Task 1 Accuracy at epoch 0:", test_acc_task1)

test_loss_task2,    test_acc_task2    =    model.evaluate(x_test_task2,
y_test_task2, verbose=2)
print("Task 2 Accuracy at epoch 0:", test_acc_task2)

# Save the model weights after each epoch
task1_val_acc = [test_acc_task1]
weights_list = [weights]
accuracy_list = []
task2_val_acc = [test_acc_task2]
for epoch in range(10):
    history_task1 = model.fit(x_train_task1, y_train_task1, epochs=1,
validation_data=(x_test_task1, y_test_task1))
```

```
        accuracy_task1 = model.evaluate(x_test_task1, y_test_task1,
verbose=0)[1]
        accuracy_task2 = model.evaluate(x_test_task2, y_test_task2,
verbose=0)[1]
   accuracy_list.append([accuracy_task1, accuracy_task2])
     print("Epoch {}, Accuracy on Task 1: {}, Accuracy on Task 2:
{}".format(epoch+1, accuracy_task1, accuracy_task2))

   weights_list.append(model.get_weights())

   task1_val_acc.append(accuracy_task1)
   task2_val_acc.append(accuracy_task2)

# Plot the weight distribution of each layer for each epoch
num_layers = len(weights_list[0])
fig, axs = plt.subplots(nrows=num_layers, ncols=len(weights_list),
figsize=(20, 20))
for i in range(len(weights_list)):
   for j in range(num_layers):
       w = weights_list[i][j].flatten()
                axs[j, i].hist(w, bins=50, alpha=0.5,color='green',
density=True)
       axs[j, i].set_title(f"Epoch {i}, Layer {j+1}")
       axs[j, i].set_xlabel("Weight Value")
       axs[j, i].set_ylabel("Density")
plt.tight_layout()
plt.show()


# Save the model weights
model.save_weights('task1_model_epoch_4.h5')
#Load weights
model.load_weights('task1_model_epoch_4.h5')

# Compile the model using the custom optimizer
model.compile(optimizer=ModifiedLearningRateOptimizer(),
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])
# Initialize the list to save the validation accuracy for MNIST
weights_list_task2=[]
for epoch in range(10):
              model.fit(x_train_task2, y_train_task2, epochs=1,
validation_data=(x_test_task2, y_test_task2))
```

```
        accuracy_task1 = model.evaluate(x_test_task1, y_test_task1,
verbose=0)[1]
        accuracy_task2 = model.evaluate(x_test_task2, y_test_task2,
verbose=0)[1]
   accuracy_list.append([accuracy_task1, accuracy_task2])
     print("Epoch {}, Accuracy on Task 1: {}, Accuracy on Task 2:
{}".format(epoch+1, accuracy_task1, accuracy_task2))

    # Save the validation accuracy for Fashion-MNIST and MNIST during
this epoch
   task1_val_acc.append(accuracy_task1)
   task2_val_acc.append(accuracy_task2)

   # Get the weights of the model
   weights = model.get_weights()
   weights_list_task2.append(model.get_weights())

# Plot the weight distribution of each layer for each epoch
num_layers = len(weights_list_task2[0])
fig, axs = plt.subplots(nrows=num_layers, ncols=len(weights_list),
figsize=(20, 20))
for i, weights in enumerate(weights_list_task2):
   for j in range(num_layers):
                        axs[j, i].hist(weights[j].flatten(), bins=50,
alpha=0.5,color='green', density=True)
       axs[j, i].set_title(f"Epoch {i+11}, Layer {j+1}")
       axs[j, i].set_xlabel("Weight Value")
       axs[j, i].set_ylabel("Density")
plt.tight_layout()
plt.show()


Accuracy plot over the epochs for model 2

import matplotlib.pyplot as plt
plt.plot(task1_val_acc, label='Task 1')
plt.plot(task2_val_acc, label='Task 2')
plt.title('Validation Accuracy of Task 1 and Task2')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
#calculation and plotting of skewness and kurtosis

# Combine the weights_list and weights_list_mnist
weights_list.extend(weights_list_task2)

import scipy.stats as stats

weights_array = np.array(weights_list)  # convert list of weights to
numpy array
num_epochs, num_layers = weights_array.shape[:2]  # get number of
epochs and layers
skewness = np.zeros((num_epochs, num_layers))
kurtosis = np.zeros((num_epochs, num_layers))

for i in range(num_epochs):
   for j in range(num_layers):
        weights = weights_array[i, j].flatten()  # flatten weights for
this layer and epoch
       skewness[i, j] = stats.skew(weights)
       kurtosis[i, j] = stats.kurtosis(weights)



skewness_list = []
kurtosis_list = []

for i in range(len(weights_list)):
   for j in range(len(weights_list[0])):
       weights = weights_list[i][j].flatten()
       skew = scipy.stats.skew(weights)
       kurtosis = scipy.stats.kurtosis(weights)
       skewness_list.append(skew)
       kurtosis_list.append(kurtosis)
        print("Epoch {}, Layer {}: Skewness={}, Kurtosis={}".format(i,
j+1, skew, kurtosis))

print("Skewness values:", skewness_list)
print("Kurtosis values:", kurtosis_list)

# Separate kurtosis values by layer
num_layers = len(weights_list[1])
```

```
kurtosis_by_layer = [[] for _ in range(num_layers)]
for i in range(len(kurtosis_list)):
    layer_index = i % num_layers
    kurtosis_by_layer[layer_index].append(kurtosis_list[i])

print("Kurtosis values by layer:", kurtosis_by_layer)

# Separate skewness values by layer
num_layers = len(weights_list[0])
skewness_by_layer = [[] for _ in range(num_layers)]
for i in range(len(skewness_list)):
    layer_index = i % num_layers
    skewness_by_layer[layer_index].append(skewness_list[i])

print("Skewness values by layer:", skewness_by_layer)

# Set color palette
colors = ["tab:blue", "tab:orange", "tab:green", "tab:red"]

# Plot skewness and kurtosis values for each layer
num_layers = len(skewness_by_layer)
epochs = range(len(skewness_by_layer[0]))
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(5, 6))

for layer in range(num_layers):
    color = colors[layer % len(colors)]
     ax1.plot(epochs, skewness_by_layer[layer], color=color, label="Layer
{}".format(layer+1))
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Skewness")
    ax1.tick_params(axis='y', labelcolor=color)
    ax1.set_ylim([-0.5, 1.5])  # set y-axis limits here

        ax2.plot(epochs,  kurtosis_by_layer[layer],  linestyle='dotted',
color=color, label="Layer {}".format(layer+1))
    ax2.set_xlabel("Epochs")
    ax2.set_ylabel("Kurtosis")
    ax2.tick_params(axis='y', labelcolor=color)
    ax2.set_ylim([-2, 1.5])  # set y-axis limits here

ax1.legend()
ax2.legend()
plt.tight_layout()
```

```
plt.show()
```