Robust and Uncertainty-Aware Image Classification using Bayesian Vision Transformer Model

Fazlur Rahman Bin Karim
The University of Texas Rio Grande Valley, fazlurrahmanbin.karim01@utrgv.edu

ROBUST AND UNCERTAINTY-AWARE IMAGE CLASSIFICATION USING BAYESIAN VISION TRANSFORMER MODEL

A Thesis

by

FAZLUR RAHMAN BIN KARIM

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE IN ENGINEERING

Major Subject: Electrical Engineering

The University of Texas Rio Grande Valley

December 2023

ROBUST AND UNCERTAINTY-AWARE IMAGE CLASSIFICATION USING BAYESIAN VISION TRANSFORMER MODEL

A Thesis
by
FAZLUR RAHMAN BIN KARIM

COMMITTEE MEMBERS

Dr. Dimah Dera and Dr. Rogelio Soto
Chair of Committee

Dr. Yong Zhou
Committee Member

Dr. Weidong Kuang
Committee Member

December 2023

ABSTRACT

Karim, Fazlur Rahman Bin, Robust and Uncertainty-Aware Image Classification using Bayesian Vision Transformer Model. Master of Science in Engineering (MSE), December, 2023, 123 pp., 7 tables, 15 figures, references, 115 titles.

Transformer Neural Networks have emerged as the predominant architecture for addressing a wide range of Natural Language Processing (NLP) applications such as machine translation, speech recognition, sentiment analysis, text anomaly detection, etc. This noteworthy achievement of Transformer Neural Networks in the NLP field has sparked a growing interest in integrating and utilizing Transformer models in computer vision tasks. The Vision Transformer (ViT) model efficiently captures long-range dependencies by employing a self-attention mechanism to transform different image data into meaningful, significant representations. Recently, the Vision Transformer (ViT) has exhibited incredible performance in solving image classification problems by utilizing ViT models, thereby surpassing the capabilities of Convolutional Neural Networks (CNN). Deterministic Vision Transformer (ViT) models are prone to noise and adversarial attacks, hence lacking the ability to provide a reliable measure of confidence or uncertainty in their output predictions. However, developing a robust Vision Transformer (ViT) model, which can quantify the confidence (or uncertainty) level in the output predictions for vision applications with high-risk implications, such as autonomous vehicles, medical imaging, etc., has significant importance. To ensure the dependability of Vision Transformer (ViT) in crucial applications, using Bayesian Inference aids in generating probabilistic predictions. In this work, we develop a robust image classification framework using the Bayesian Vision Transformer (Bayes-ViT) and Bayesian Compact Convolutional Transformer (Bayes-CCT) model, which provides output predictions and quantifies uncertainty associated with output predictions. The proposed Bayesian Vision Transformer model incorporates a variational inference framework and optimizes the variational posterior distribution over the model parameters using the evidence lower bound (ELBO) loss function. The propagation of variational moments in Bayesian Vision Transformer's sequential, non-linear layers is achieved using the first-order Taylor series approximation. The output of the proposed architecture consists of a predictive distribution, where the mean represents the output prediction, and the covariance matrix provides information about the uncertainty associated with the prediction. Extensive experiments on benchmark datasets demonstrate (1) the superior robustness of proposed models under noise and adversarial attacks in comparison to the deterministic ViT models and (2) the ability for self-evaluation by utilizing the discernible increase in prediction uncertainty when the developed model encountered high levels of random noise and adversarial attacks, which acts as a warning sign for crucial image classification tasks.

DEDICATION

I would like to dedicate this work to my parents, Md Fazlul Karim and Sanwara Karim, who are my motivators and whose unconditional love and support have helped me to come this far and pursue higher studies. I would also like to thank my wife, Lamia Alam, for her unwavering support throughout this journey. I want to show my gratitude to my sisters Tanjila, Lubna, Farhana and Tazrina for acting as constant motivators.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis supervisor and former chair of my thesis committee, Dr. Dimah Dera, for her invaluable guidance, support, and encouragement throughout my Master's program. Her insightful feedback and constructive criticism have been instrumental in shaping my thesis work. I express my deep gratitude to Dr. Rogelio Soto, current chair of my thesis committee, for all his mentoring and advice.

I extend my appreciation to my thesis committee members, Dr. Yong Zhou and Dr. Weidong Kuang, for their time and feedback. Their invaluable feedback and advice helped to improve the quality of the work. Additionally, I sincerely appreciate the prompt support and guidance of Dr. Hasina Huq, the Chair of the Department of Electrical and Computer Engineering Department. I am grateful for the opportunities and resources provided by the department, which have been instrumental in completing this thesis.

Finally, I would like to thank University of Texas Rio Grande Valley(UTRGV) Library colleagues for helping me with their support in using all the available resources required for this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Motivation: Importance of Vision Transformer

Deep learning approaches have demonstrated superior performance to state-of-the-art machine learning techniques across domains such as Natural Language Processing and Computer Vision Applications in recent years. [27, 100]. With the advent of Artificial Intelligence and computing resources, Computer Vision technology has garnered extensive attention among researchers [78]. The objective of computer vision research is to provide computers with perceptual abilities akin to those of humans, enabling them to see their surroundings and comprehend the information obtained to learn and subsequently enhance their performance through learning [49, 84, 91]. The Convolutional Neural Network (CNN) is a deep learning architecture extensively employed to perform various computer vision tasks, including classification, object detection, and segmentation, with superior performance. [36, 47, 76, 115]. CNN uses convolutional and pooling layers to process shift-invariant data, namely images, effectively. Nevertheless, the capacity of these models to learn long-range dependencies among input sequences is constrained by their localized receptive fields. [59]. For a while, CNNs have been utilized to acquire commendable performance in solving image classification tasks. Classification is the fundamental yet most crucial domain in Computer Vision. On the other hand, Transformer Neural Networks (TNNs) are gaining prominence in solving numerous image classification tasks, drawing upon the achievements gained within the realm of natural language processing (NLP). [62]. Vision Transformer (ViT) architectures solve diverse computer vision problems while employing a self-attention mechanism. [113]. The Vision Transformer (ViT) utilizes a transformer encoder architecture to process non-overlapping image patches for image classification tasks. [7,17,21,45]. The widespread success of Vision Transformers (ViTs) in addressing various image classification challenges has led to increased adoption of ViTs in various classification tasks ranging from biomedical to e-commerce. [6,18,38,42,71,98]. In the subsequent sections, we will demonstrate two important applications in which the performance of solving image classification problems is of utmost importance.

1.1.1 Biomedical Imaging

Biomedical images are crucial in pivotal decision-making procedures, encompassing disease diagnosis, therapy strategizing, and patient surveillance. The precision of the image classification mechanism directly influences the reliability of decisions. Typically, biological images are examined and analyzed by physicians or radiologists. The analysis of different modes of images might provide challenges for humans because of the significant variances observed in diseases. [15]. The advent of advanced imaging techniques and an unparalleled computational capacity presents a unique prospect to analyze and solve biomedical image classification problems in manners that were unattainable before. [1]. Medical imaging techniques have a crucial role in the detection and diagnosis of several diseases, such as cancer, COVID-19, skin disease, tuberculosis, etc., with the implementation of ViT models [4,9,21,95]. While AI has simplified various aspects of our lives, it also introduces certain drawbacks. Patient safety is directly influenced by the decisions based on biomedical image classifications. The presence of erroneous or unreliable classification might potentially result in the incorrect identification of medical conditions and the selection of incorrect treatment options, endangering the well-being of patients; no matter how powerful a deep learning model is, difficult diagnostic cases are inevitable and may lead to potentially severe consequences for patients if the model does not refer them for further inspection. [92]. The deep learning model developed for tumor detection may also face an adversarial attack aimed at fabricating a false tumor to exploit medical billing procedures. [24]. The introduction of imperceptible alterations to medical imaging by an attacker has the potential to lead to erroneous diagnoses. Incorrect output generated by Vision Transformer-based classifiers can have severe consequences, including endangering lives, undermining credibility, administering incorrect treatments, and eroding customer trust. To mitigate the impact of noise and adversarial attacks, it is recommended that medical professionals, such as doctors and radiologists, exclusively utilize robust ViT models for different clinical applications.

1.1.2 Autonomous Driving

The autonomous vehicle (AV) is widely regarded as an essential element of future transportation systems due to its potential advantages, which include the reduction of human labor and associated expenses, improved safety and reliability, and decreased emissions and energy consumption. [25]. Various architectures have been vastly utilized for autonomous driving applications since the introduction of Vision Transformer (ViT) based models. [6, 14, 90]. Object detection technology empowers autonomous cars to effectively recognize and monitor many impediments within their immediate environment, including but not limited to other vehicles, pedestrians, bicycles, and stationary objects. Effective object detection using ViT models helps prevent collision, ensures safe movements of pedestrians, and provides parking and intersection crossing assistance. The ability to detect objects using ViT is crucial for the effective functioning of autonomous driving systems, as it allows vehicles to observe and comprehend their surroundings and navigate precisely in the traffic system. The intentional modification of input data through intelligent adversarial attacks can generate situations in which the autonomous system cannot precisely classify static or moving obstacles. This situation presents a notable safety hazard since the vehicle may fail to execute suitable measures to prevent unavoidable situations. Consequently, strong measures, such as robust object detection, are crucial for preventing self-driving car collisions due to wrong decision-making of deep learning models.

1.2 Background and Problem Statement

Image classification pertains to a computer vision undertaking wherein a machine learning model is trained to identify and classify objects or situations depicted in an image based on training. The objective is to allocate a categorical designation or classification to an input image by analyzing its visual contents. The efficacy of the image classification task depends on selecting a suitable model, the size and variety of the training data, and the meticulous tuning of parameters throughout the training procedure. [65].

Traditional machine learning (ML) algorithms, including Random Forest, K-Nearest Neighbor, Decision Tree, Support Vector Machines, and Naive Bayes classifiers, have been widely employed in addressing image identification and classification applications according to specific needs. [5, 16, 68, 88]. However, these conventional ML algorithms demonstrate subpar performance while solving complex image classification tasks. [74]. In recent years, there has been a notable demonstration of remarkable performance by Deep Neural Network (DNN) based architectures in addressing a range of computer vision challenges, including complicated image recognition tasks [3, 37, 50]. Convolutional neural network (CNN) based models have exhibited promising outcomes and have become the standard approach for addressing diverse image classification tasks. [77, 94, 97, 107]. The Attention-based Transformer Neural Network (TNN) has emerged as a promising solution for addressing heterogeneous classification challenges, owing to its remarkable performance in delivering state-of-the-art results in solving Natural Language Processing (NLP) tasks [11, 26, 99]. One persistent issue is that deep neural networks need more robustness against strong adversarial attacks and noisy settings, manifesting them as unreliable in the face of more diverse and complex attacks. [75]. Relying only on the decisions provided by these vulnerable deep learning models for critical applications may pose challenges.

1.2.1 Problem Statement

The existing machine learning and deep learning algorithms exhibit two significant constraints that impact their performance and dependability. The majority of these algorithms exhibit limited robustness in the presence of noise. While numerous deep learning models have demonstrated superiority in accurate predictions under known circumstances, their performance diminishes when encountered by noises and perturbations. To be precise, they are susceptible to noise added to the input data, and even tiny perturbations might result in erroneous outputs. Hence, these models must be better suited for real-world critical applications where noises or corrupt data can potentially hinder the decision-making of applied mechanisms. Furthermore, these deep learning mechanisms need to quantify certainty, posing challenges in evaluating the dependability of their decision-making processes. Consequently, there needs to be more certainty regarding the reliability of the evaluations conducted by these models, particularly when employed in mission-critical domains. The quantification of uncertainty in robust image classification can notify users about potential noise or perturbation in the output prediction. Quantifying uncertainty is critical for trustworthy decision-making as it outlines the model's confidence and reliability. So, the research community must design a model that can exhibit uncertainty in the output of decision-making algorithms. Failure to recognize instances where models are prone to errors can lead to negative consequences and diminish their effectiveness in vital scenarios.

1.3 Research Objectives and Contributions

In this work, we propose a novel image classification framework employing Bayesian Vision Transformer architectures, referred to as Bayesian Transformer models such as Bayesian Vision Transformer)Bayes-ViT) and Bayesian Compact Convolutional Transformer(Bayes-CCT). Utilizing the Bayesian Formulation, we propose a mathematical framework for uncertainty estimation in model output prediction. Considering ViT and CCT architecture parameters as random variables, Gaussian distribution was initiated over the model parameters as a prior distribution. We use Variational Inference (VI) and minimize the evidence lower bound (ELBO) loss function to approximate the posterior distribution of the model's parameters. Using the first-order Taylor approximation, we propagate the first two moments (mean and covariance) of the variational posterior distribution over the model's parameters through all layers and non-linear activation functions of Bayes-ViT and Bayes-CCT architecture. The mean and Covariance of the predictive distribution were attained at the network output. The covariance matrix represents the degree of uncertainty associated with the output prediction, whereas the mean vector at the output denotes the classification output.
The major contributions of this thesis work are summarized as follows:

1. We propose a Bayesian Vision Transformer Neural Network to make predictions in image classification problems and measure the degree of associated uncertainty with the prediction output.

2. Using the available data, we use variational inference to approximate the posterior distribution of the parameters. We propagate the variational posterior distribution's mean and covariance through the network layers and non-linear functions.

3. Through the non-linear activation functions in the Bayes-ViT architecture, we estimate the first two moments of the variational distribution (mean and covariance) using a first-order Taylor series approximation.

4. We conduct comprehensive experiments on three benchmark datasets, i.e., MNIST [23], Fashion-MNIST [104], CIFAR-10 [54] datasets, which contain labeled images for image classification tasks.

5. By using adversarial attacks and Gaussian Noises to tamper with the test samples and evaluating our Bayes-ViT and Bayes-CCT against the state-of-the-art Deterministic ViT and Deterministic CCT, we thoroughly examine all three datasets in noisy situations.

6. We validate our proposed model's robustness to elevated noise levels. We show that high noise conditions lead to a large rise in uncertainty for image classification tasks.

CHAPTER II

LITERATURE REVIEW

2.1 Image Classification with Deep Neural Networks

The image classification process helps us make important judgments in our daily lives. It improves productivity, security, and ease of use in diverse and important applications. Image classification algorithms enable visual data acquisition, facilitating its application in various domains such as medical diagnosis, traffic analysis and control, security systems, facial recognition, and other areas. For a long while, academics worldwide have been reporting tremendous advancements in the field of image classification [87]. Conventional computer vision methods were the mainstays of early image classification. The image classification process was mostly carried out using models based on feature extractors. Histogram of Oriented Gradients(HOG), Scale-Invariant Feature Transform (SIFT), Content-Based Image Retrieval(CBIR), etc., were some of the primary feature extraction techniques commonly used for image classification tasks [22, 51, 64]. As these conventional techniques primarily bank on handcrafted feature extraction methods, they possess numerous constraints such as obstacles with scaling, managing heterogeneous and complicated data, and adjusting to changing circumstances [56,63]. Their poor scalability and generalization properties have created significant challenges in solving complex image classification tasks.

Certain methodologies emerged in response to the difficulty of efficiently portraying unprocessed data, particularly when dealing with intricate image datasets. The integration of conventional computer vision methodologies with machine learning algorithms, such as Support Vector Machines (SVM), Naive Bayes, Random Forest, etc., frequently yields image classification systems that are characterized by improved performance [12,13,63]. Gao *et al.* proposed an improved SIFT method based on the Bag-of-words(BOW) algorithm to handle intricate local visual features while utilizing the SVM classifier to perform classification [29]. Extracting more global features helps improve the overall performance of this proposed work. Xie *et al.* extended the SIFT algorithm by proposing the MAX-SIFT method, which is an invariant feature that is derived by selecting the maximum value between a SIFT descriptor and its horizontally flipped counterpart [105]. Results demonstrated that the proposed method performs better in image classification tasks without any augmentation. One critical limitation of the SVM algorithm is the higher computational complexity due to its time-consuming optimization process [112]. Zhang *et al.* proposed a hybrid architecture comprised of K-Nearest Neighbor and SVM to triumph over the constraints inherited by SVM in solving Multi-label Classification problems [112]. The fundamental concept involves identifying nearby neighbors to a given query sample and subsequently training a localized support vector machine that maintains the distance function within the set of neighbors. Compared to KNN and SVM approaches, this approach exhibited superior performance when applied to extensive, multi-class datasets. Though Support Vector Machine (SVM) classifiers are well suited for binary classification problems, they are computationally very expensive [89]. On the contrary, Random Forest(RF) classifiers can achieve comparable performance in image classification tasks with lesser computational cost. Sheykhmousa *et al.* provided evidence to support the notion that employing random forests/ferns with a suitable node test can effectively decrease the expenses associated with training and testing, in comparison to a multi-way SVM [10]. Furthermore, the performance achieved by this approach is comparable to that of the SVM. Other conventional Machine Learning algorithms, such as Naive Bayes, Decision Tree, etc., have also been used extensively in diverse image classification applications [108, 110]. However, there are still areas within the field of machine learning that can be further improved [53].

Traditional machine learning techniques frequently struggle to capture intricate patterns. Moreover, these traditional models also face difficulties in selecting features for training. Conventional image feature extraction algorithms primarily emphasize the manual configuration of certain picture properties [19]. With the advent of deep learning, Integrating feature extraction and classifiers inside a learning framework has successfully addressed the challenges associated with traditional methods. The concept of deep learning aims to uncover numerous layers of representation, with the expectation that higher-level characteristics contain more abstract semantic representations of the given data. Like other domains, deep learning has established footprints in image classification tasks using convolution-based architectures [36].

In recent years, there has been a notable surge of deep learning (DL) as a prominent area of research. Convolutional Neural Networks (CNNs) have emerged as the dominant deep learning models in several computer vision tasks, including classification [43, 80, 82]. Lecun *et al.* introduced the first CNN model applied to recognize handwritten digits [57]. Though the model performed well in simple datasets, performance was sub-optimal when applied to larger datasets. Krizhevsky *et al.* introduced AlexNet to perform image classification tasks on a comparatively large dataset named ImageNet [55]. It utilized numerous convolutional layers, a couple of fully connected hidden layers, and one fully connected output layer. Different CNN models have recently been adopted to solve image classification problems in various domains. Sorić *et al.* solved binary classification problems to identify pneumonia using chest x-ray images [93]. Sharma *et al.* employed patch-based CNN architecture to capture the intrinsic relationship between a pixel and its surrounding environment in remote sensing application [86]. Tarmizi *et al.* established a novel CNN framework for enhancing the identification of automobiles in low-light environments and adverse weather conditions in Autonomous Vehicle(AV) applications [96]. Rahman *et al.* proposed a novel framework for identifying unusual behavior, particularly focusing on frequent iris movements using CNN [79]. Xu *et al.* used an attention mechanism with CNN to distinguish threatening objects in airport X-ray images [106]. Shervin *et al.* developed a CNN-based framework for biometric recognition tasks that can achieve higher recognition accuracy in detecting fingerprints [70]. Wang *et al.* introduced a combined CNN-Recurrent Neural Network(RNN) framework, which is capable of acquiring a shared image-label embedding. This enables the characterization of the semantic label dependency and the image-label relevance [102]. Gill *et al.* introduced another hybrid architecture combining CNN, RNN, and long short-term memory(LSTM) in solving multi-label fruit classification problems [32]. LSTM incorporated a memory cell in this framework to encode the learning process at each categorization interval.

Convolutional Neural Networks (CNNs) have emerged as the predominant choice for various computer vision tasks, notably image classification tasks throughout domains. Nevertheless, because of their inherent locality, these models possess a restricted capacity to attain long-range dependencies among diverse input sequences [40]. Conversely, Transformers have gained widespread popularity and emerged as a crucial focal point in contemporary machine learning research following the introduction of "Attention is All You Need" in the Natural Language Processing(NLP) sector [99]. Since its inception, there has been a notable surge in research works dedicated to transformer-based and attention-based approaches within the research community [33, 109]. Inspired by the achievements of Transformer models with self-attention mechanisms in NLP, Dosovitskiy *et al.* introduced the Vision Transformer (ViT) architecture specifically designed for image classification tasks [26]. The maiden model has demonstrated superior performance in image classification applications compared to state-of-the-art CNN architectures when trained on significantly large datasets [30]. ViT architectures are now broadly implemented for solving classification tasks ranging from medical imaging to remote sensing [6, 7, 21, 44]. However, acquiring sizable datasets poses a challenge in certain specialized fields [48].

Moreover, it is not always practical to train the model with large datasets due to limited computational resources. Various hybrid architectures have been put forward in contemporary times to address the constraints associated with pre-training. Lee *et al.* locality-based Self Attention technique to overcome the inherent low inductive bias of ViT [58]. Hassani *et al.* combined ViT and CNN to prevent overfitting and surpass the performance of state-of-the-art CNNs when dealing with small datasets [39]. Liu *et al.* designed a model that motivates ViTs to grasp spatial relationships within images, enhancing the robustness of ViT during training, particularly in scenarios where training data is strictly limited [61]. Ran *et al.* manifested an approach to replace linear-based projections with convolutional projections in the Self-Attention block to acquire a greater number of local spatial dependencies and eliminate ambiguity in local content during the attention process [85].

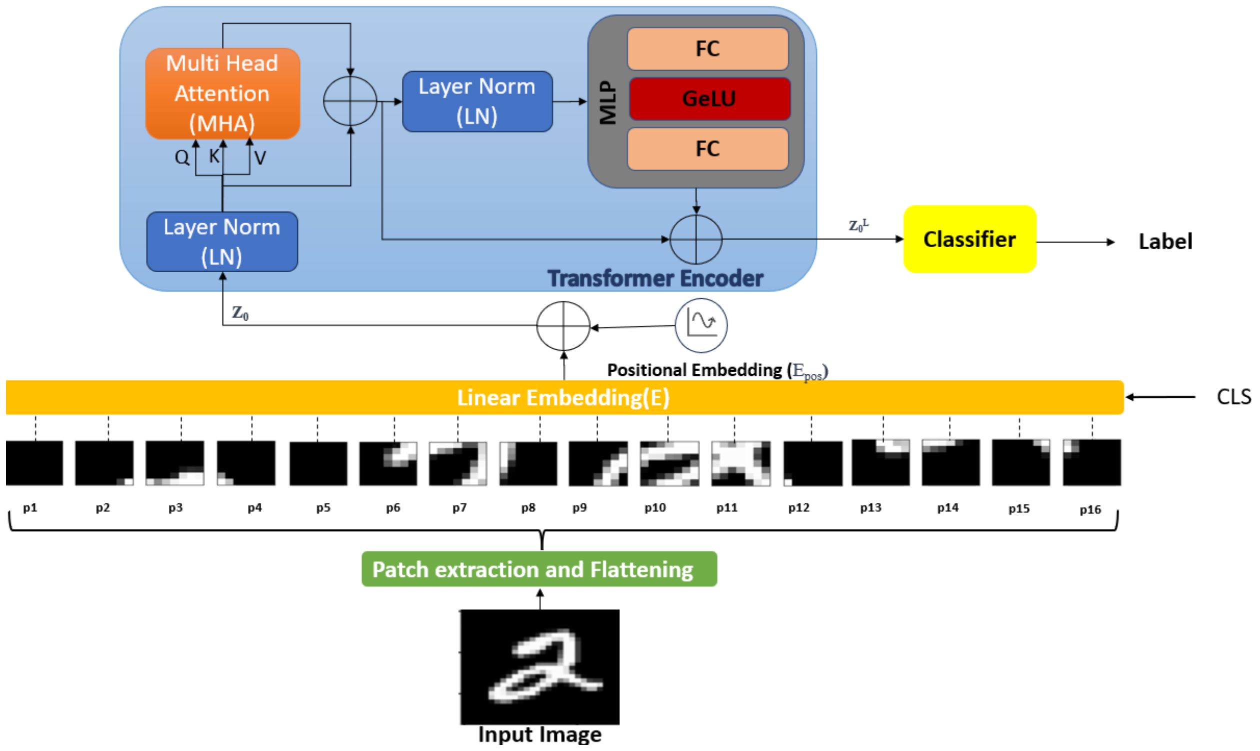


Figure 2.1: Vision Transformer Architecture.

2.2 Vision Transformer

Self-attention-based architectures, specifically the Transformer Neural Network(TNN) model proposed by Vaswani *et al.*, have emerged as the preferred approach in the NLP domain [99]. TNN architectures are widely used in many NLP applications such as machine translation, text summarizing, sentiment analysis, text classification, etc. Transformers were primarily designed to allow the input sequence to be processed in parallel, something that LSTMs and RNNs could not accomplish [7]. Owing to the effectiveness of transformers in NLP, self-attention has been attempted to be implemented in the computer vision domain with the least amount of modifications. To deal with image data, an image is first divided into fixed-size, non-overlapping patches. After patch extraction, all the image patches are flattened and fed to the Transformer encoder layer after converting the patches into patch embedding with the help of linear projection. After that, the linear embedding created from image patches is fed to the transformer encoder in a parallel manner as an input sequence. Figure 2.1 depicts the architecture of the ViT model for image classification tasks. The overall mechanism is described as follows:

2.2.1 Patch Embedding

At first the input image x, where $x \in \mathbb{R}^{H*W*C}$ is reshaped into input sequence of 2D flattened patches $x_p \in \mathbb{R}^{N*(P^2C)}$, where (H,W) is the dimension of input image, C is the channel number, (P,P) is the dimension of fixed size patches and N is the number of patches created for the input sequence where $N = \frac{HW}{P^2}$. One extra learnable embedding has been added at the beginning of the embedded patch series.

2.2.2 Positional Embedding

The spatial information on the locations of patches inside an input image is then obtained by applying the positional embedding. They enable the model to consider the sequential relationships between patch embedding by capturing the position or order information of the tokens in an input sequence. There are different types of positional embedding. Learnable positional embedding was used in most cases for ViT. This learnable positional embedding allows the model to adjust and optimize the positional information during training.

2.2.3 Transformer Encoder

The layers that comprise the Transformer encoder structure comprise the multi-layer perceptron (MLP), the layer normalization, and the self-attention function.

2.2.3.1 Self-Attention. The self-attention mechanism, which determines the correlation between the visual patches within the input sequence, is the central component of the ViT model. The Transformer can ascertain significant correlations between the input image patches with the help of a self-attention mechanism 2.2.

$$Attention(Q,K,V) = SoftMax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

where Query(Q), Key(K), and Value(V) represent the linear projection of the input sequence, and d_k denotes the dimension of the key vector, which is equal to the embedding dimension of the ViT model.

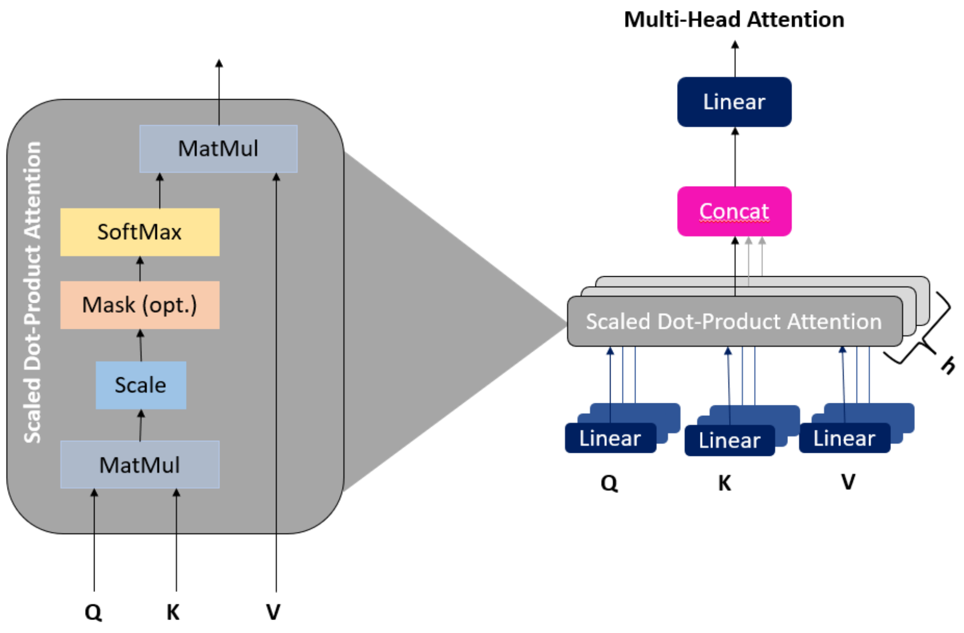


Figure 2.2: Attention Mechanism of ViT.

2.2.3.1.1 Scaled dot product attention. The attention score between them is determined by taking the dot product of each query's and key's respective vectors. This dot product measures the query and key vectors' similarity. The dot products are scaled by the square root of the dimension of the key vectors (d_k) to keep them from growing too big and maintain the learning process. The attention scores are then derived from the scaled dot products. The SoftMax function is used to normalize the attention scores to get attention weights. Figure 2.2 illustrates the operation of the attention mechanism.

In conclusion, the Vision Transformer's self-attention layers employ Scaled Dot-Product Attention to identify dependencies between various patches in an input image. This lets the model pay attention to pertinent details throughout the input image.

2.2.3.1.2 Multi Head Attention. Multi-Headed Attention (MHA) denotes the independent repetition of the self-attention processes for several heads, h. Multi-head attention enables the model to simultaneously focus on input from various representation sub-spaces at distinct positions.

$$Multihead(Q,K,V) = Concat(head_1, head_2, ..., head_h)W^0 \tag{2.2}$$

where, $head_i = Attention(Q\mathbf{W_i}^{\mathbf{Q}}, K\mathbf{W_i}^{\mathbf{K}}, V\mathbf{W_i}^{\mathbf{V}})$ and $\mathbf{W_i}^{\mathbf{Q}}, \mathbf{W_i}^{\mathbf{K}}, \mathbf{W_i}^{\mathbf{V}}$ are weight matrices and $\mathbf{W_i^Q} \in \mathbb{R}^{(d_{model}*d_k)}, \mathbf{W_i}^{K} \in \mathbb{R}^{(d_{model}*d_k)}, \mathbf{W_i}^{\mathbf{V}} \in \mathbb{R}^{(d_{model}*d_k)}$

2.2.3.2 Layer Normalization. Layer normalization (LN) is a widely employed normalization approach in neural networks, specifically in Vision Transformer (ViT) architecture. In the context of Vision Transformer (ViT), the technique of layer normalization is commonly employed in a manner that is individually applied to every token or patch inside the input sequence. Given an input vector $x = [x_1, x_2, ..., x_n]$ the layer normalization is computed as:

$$LayerNorm(x) = \gamma . \frac{(x - \mu)}{\sigma} + \beta \tag{2.3}$$

Where γ is the learnable scale parameter, β is the learnable shift parameter, μ is the mean of input vector x, and σ is the input vector x standard deviation.

The mean, μ, and the standard deviation σ are computed separately for each point (patches) along the feature dimension of the input sequence. Layer Normalization is a technique that stabilizes the training process by standardizing the values at each point, hence reducing sensitivity to the input's scale. This aspect has significance for the general stability and convergence of the model throughout the training process.

2.2.3.3 Multi-Layer Perceptron. The Multi-Layer Perceptron (MLP) within a Vision Transformer (ViT) is crucial in capturing localized patterns, creating non-linearities, and augmenting the model's capacity to acquire intricate representations from the input patches. It collaborates with the attention mechanism to offer a holistic comprehension of the input image's global and local features. MLP applies a feature-wise transformation to the input data, translating it to a higher-dimensional space. This transformation enables the representation of non-linear interactions effectively. The learnable parameters of the MLP, encompassing weights for linear transformations and activation functions, undergo updates during the training process, enabling the model to adjust and accommodate some particular data traits.

The MLP layer comprises two fully connected layers and the Gaussian error Linear Unit

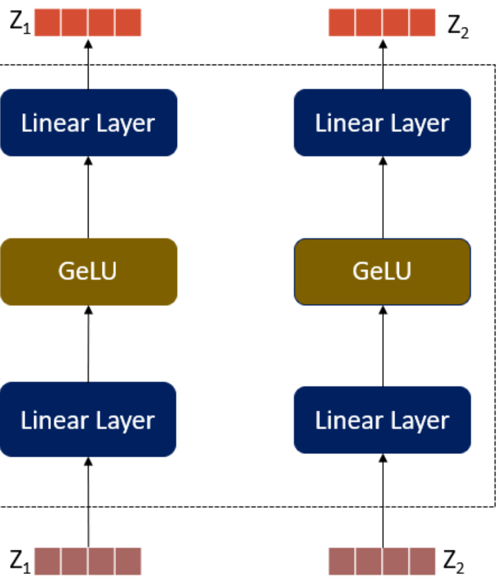


Figure 2.3: Multi Layer Perceptron.

(GeLU) activation function. Figure 2.3 demonstrates the MLP layer consists of two linear layers and the GeLU activation function.

2.2.3.3 Final Embedding Layer. The MLP's output helps create each patch's final embeddings, fed into the classification head for the final predictions.

The overall mechanism is outlined as follows:

$$z_0 = [x_{class}; {x_p}^1 E; {x_p}^2 E, ...{x_p}^N E] + E_{pos}; \quad E \in \mathbb{R}^{(F^2 * C) * D}, E_{pos} \in \mathbb{R}^{(N+1) * D} \tag{2.4}$$

$$z_l^{'} = MSA(LN(z_{l-1})) + z_{l-1}; \quad l = 1......L \tag{2.5}$$

$$z_l = MLP(LN(z_l^{'})) + z_l^{'}; \quad l = 1......L \tag{2.6}$$

$$y = LN({z_L}^0) \tag{2.7}$$

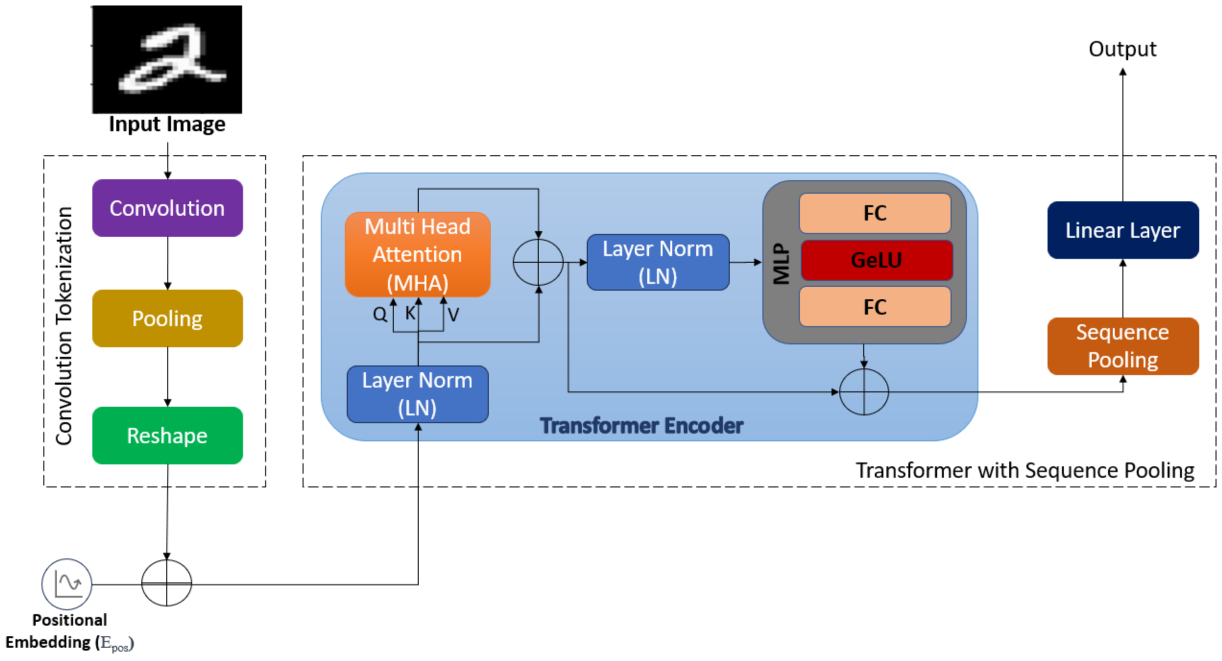


Figure 2.4: Compact Convolutional Transformer Architecture.

2.3 Compact Convolutional Transformer

2.3.1 Convolutional Tokenizer

With Compact Convolutional Transformer (CCT) architecture et al. aimed to propose a mechanism for small-scale learning. This study presented novel findings indicating that when combined with transformers, appropriately sized convolutional tokenization can effectively mitigate overfitting and achieve superior performance when applied to small datasets. It diminishes the reliance on high computing resources for higher performance from transformer-based architectures. Instead of employing a non-overlapping patch-based tokenization approach, this model utilized a convolutional tokenizer, demonstrating superior performance in encoding associations across patches compared to the original ViT.

CCT employs a convolutional approach to tokenization, resulting in the creation of more comprehensive tokens while also retaining important local information. The convolutional tokenizer performs better in encoding interdependencies among patches when compared to the original ViT. The convolution-based tokenizer comprises convolution layers, max pooling, activation function(ReLU), and batch normalization layers. Convolutional Kernel size, kernel stride, pooling size, and pooling stride are adjusted based on dataset image size. Figure 2.4 shows the CCT architecture.

Given the input image, $x \in \mathbb{R}^{H*W*C}$

$$x_o = MaxPool(ReLU(Conv2d(x))) \tag{2.8}$$

The operation is performed using a total of d filters, which is equivalent to the embedding dimension of the Vision Transformer Encoder. In addition, the Conv2d and MaxPooling operations brought inductive bias as these operations were overlapping in nature. This helped in increasing the overall performance of classification accuracy.

2.3.2 Positional Embedding

As a convolution-based tokenizer induces inductive bias in the CCT model, positional embedding is optional for providing spatial information of image patches.

2.3.3 Transformer Encoder

All the layers inside the transformer encoder (Multi et al.) followed the architecture of ViT.

2.3.4 Sequence Pooling

Sequence Pooling (SeqPool) is an attention-based approach that performs pooling over the output sequence of patches. The objective is to ensure the output sequence retains relevant information from various input image areas. By retaining this information, an improvement in overall performance was observed without the need for additional parameters. This mechanism also diminished overall computational expenses. The operation is given as follows:

$$x_L = f(x_0) \in \mathbb{R}^{b*n*d} \tag{2.9}$$

Where L is the Transformer encoder layer, x_L is the output of the encoder, b is the input batch size, n is the corresponding input sequence, and d is the embedding dimension. x_L is given input to the dense layer. The output of the dense layer becomes:

$$g(x_L) \in \mathbb{R}^{d*1} \tag{2.10}$$

After applying the SoftMax activation function, the output becomes:

$$x_L{}' = SoftmMax(g(x_L)^T \in \mathbb{R}^{b*1*d} \tag{2.11}$$

Equation (2.11) shows the process that calculates the weight of importance for every token in the input in the following way:

$$z = x_L{}' x_L = SoftMax(g(x_L)^T * x_L \in \mathbb{R}^{b*1*d} \tag{2.12}$$

Output $z \in \mathbb{R}^{b*d}$ is produced after flattening the equation (2.12).
Final prediction is made by passing z through the final classifier consisting of a fully connected layer and SoftMax activation for solving multilabel classification problems.

2.4 Bayesian Inference in ViT

An initial method for integrating Bayesian inference into neural networks was Hamiltonian Monte Carlo (HMC), a Markov chain Monte Carlo (MCMC) technique used to generate samples from the posterior distribution. However, this method encountered significant computational complexities as it uses the whole dataset for posterior distribution estimation [73]. A variant of The MCMC technique, known as Stochastic Gradient MCMC, was introduced to extend sampling methods to accommodate significantly large datasets and Deep Neural Networks (DNNs) by utilizing subsets of the dataset. This approach demonstrates more scalability as it utilizes subsets of the whole data [20, 103]. Another approach to integrating Bayesian Inference with Deep Neural Networks has involved utilizing the Laplace approximation method. This mechanism assumes that the posterior distribution can be approximated as a Gaussian distribution [66]v. Ritter *et al.* utilized a second-order optimization technique for neural networks in order to create a Kronecker factored Laplace approximation to the posterior distribution of the weights in a trained network [81]. The utilization of the maximum a posteriori (MAP) estimate was employed to calculate the mean of the posterior distribution. Based on the data that was observed, a point estimate was developed to represent the parameter values that are considered most probable. However, this method is intractable in terms of computational cost. Expectation Propagation (EP) and assumed density filtering (ADF) are two posterior approximation techniques that employ local computations on an iterative basis to approximate posterior distribution factors for individual data points. [31, 41, 60]. In solving regression problems, Hernandez-Lobato and Adams presented the probabilistic back-propagation (PBP) method to enhance the Gaussian posterior approximation [41]. In subsequent work, Ghosh *et al.* developed PBP further in solving multi-label classification problems with the proposed work [31]. The ADF approximation, as proposed by Hernandez-Lobato and Adams, effectively eliminated the need for ordering by conducting several ADF iterations on the dataset. However, full execution of EP was found to be impractical for deep neural networks (DNNs) due to the massive computing and storage demands associated with it. [60].In contemporary times, scholars are directing their attention towards assessing uncertainty in vision transformers by utilizing Bayesian inference [72, 83, 114].

Variational inference (VI) is a well-established method for approximating posterior distribution. In recent years, researchers have successfully used VI to deep neural networks (DNNs) with effective scaling [8, 35]. In an extended work, Shridhar et al. expanded upon the concept of Bayesian formulation in CNN architecture (Bayes-CNN) by introducing a fully factorized Gaussian distribution on top of convolutional kernels [28]. However, all the VI-based methods emphasize estimating uncertainty at the model output by following a frequentist approach. Propagation of variational distribution moments, defining over ViT network parameters, does not occur from one layer to subsequent layers.

CHAPTER III

METHODOLOGY

3.1 Input image Preprocessing

Image data preprocessing is an essential and critical phase in preparing data for image categorization using deep neural networks. It encompasses a sequence of procedures aimed at improving the quality of the input images, enabling efficient model training, and enhancing the overall performance of the deep neural network. By implementing these preprocessing processes, the input image data is processed to optimize its suitability for input into a deep neural network. This, in turn, enhances the learning process and results in a strong classification performance robust to errors. The resolution of an image is determined by the total number of pixels it possesses. Images with higher resolution have more pixels, which typically results in a more comprehensive representation of visual data. Resolution is commonly denoted by its width and height, such as 28×28 pixels for images in the MNIST dataset. At first, fixed-size, non-overlapping patches are created from the input image.

We treat the input patches as sequential data, similar to how sentences are formed in natural language processing tasks. The interaction between patches and the entire input image can be compared to the construction of sentences in natural language processing tasks. This enables the utilization of Transformer-based models, originally employed for NLP tasks. The same approach with little modification can be employed in computer vision to examine image patches and extract significant information from them. Natural language sentences consist of words organized in a specific sequence and possess semantic connections. Similarly, patches derived from the input image typically exhibit a certain degree of correlation with the patches situated near them.

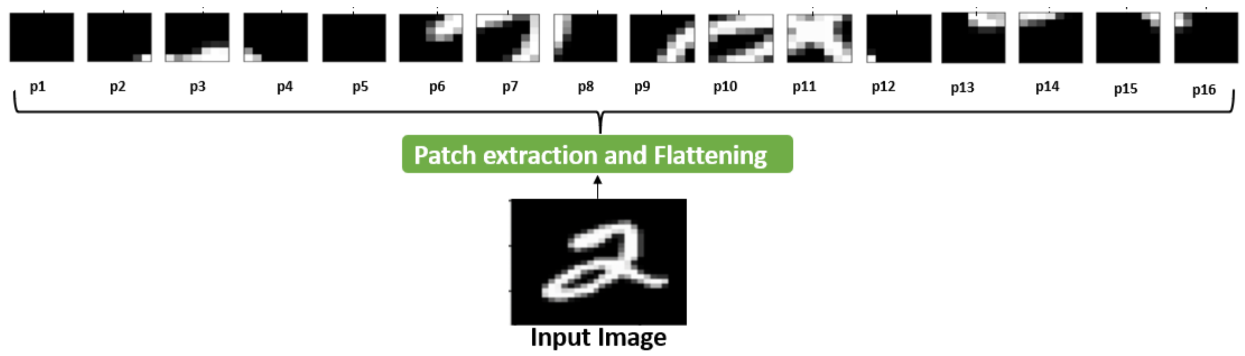


Figure 3.1: Tokenization and embedding of input image patches.

When working with image data, resizing images to a constant resolution guarantees uniformity in input dimensions. It is crucial to follow this step, especially when dealing with deep learning models that necessitate fixed input sizes. We omitted this section as the datasets utilized have consistent image dimensions across the whole dataset. We had to perform a normalization operation to re-scale pixel values to a standardized interval, commonly ranging from 0 to 1. This is accomplished by dividing the values of each pixel by the highest possible pixel value, such as 255 for photos with 8 bits per pixel. Normalization facilitates accelerated convergence during the training process. After that, we implemented data augmentation techniques to artificially enhance the training dataset's variety. Typical enhancements such as random rotations, flips, shifts, changes in brightness, and zooming were executed. Data augmentation played a crucial role in enhancing the generalization of the Bayesian transformer models.

3.1.1 Patch based Tokenization

Image tokenization is the process of dividing images into tokens in order to utilize the self-attention processes of transformers. These tokens generated from the input image are fixed size and non-overlapping. Each image patch is considered a separate input token to the transformer encoder. Figure 3.1 shows the image patch extraction and tokenization step. After creating 2D image patches from the input image, these patches are flattened into a 1D vector. Each token represents a specific part of the input image. After completing the flattening operation, linear projection transforms the flatted 1D vectors into lower dimensional representation while preserving important information and relationships along the vectors. It is done by multiplying each element of

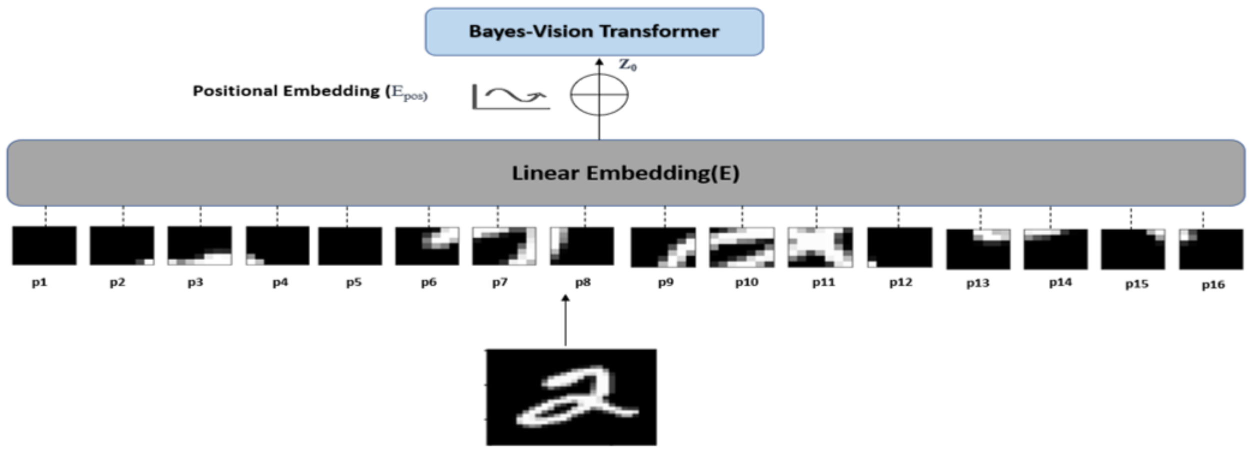


Figure 3.2: Patch and positional embedding.

the flattened sequence by weight matrix W. The training process involves acquiring knowledge for the embedding matrix through the use of back-propagation. [111]. Linear projection helps reduce parameters, subsequently making the model less computationally expensive. The aim lines capture the most important input image features for the classification task. Before linear projection, one extra cls token is added to the input sequence. This learnable token is being used to make the final classification.

After linear projection, positional embedding is added to the input sequence to provide inductive bias for the ViT network. Tokenization helps transformer-based architecture process the input sequence in a parallel manner, unlike convolution-based architectures. This input sequence is provided as an input to the transformer encoder layer.

As illustrated in 3.2, positional embeddings are added to patch embeddings before providing the input sequence to the transformer encoder layer.

3.2 Image Classification using Bayesian Transformer Models

We are examining a vision transformer comprising a total of L Encoder layers. The encoder layer comprises self-attention, layer normalization, and an MLP layer.

3.2.1 Bayesian Formulation

Bayesian neural networks are a type of neural network that can be employed to quantify the level of uncertainty in predictions. In Bayesian neural networks, the network weights are regarded as random variables that follow a prior probability distribution. Before data observation, the prior distribution reflects our first assumptions about the weights. Upon analyzing the data, we form our judgments regarding the weights specified by the posterior distribution. The posterior distribution is directly proportional to the product of the prior distribution and the likelihood distribution of the data multiplied by the weights.

In the suggested Bayesian transformer models, namely Bayes-ViT and Bayes-CCT, the Bayesian estimate is conducted by considering the model's parameters $\mathscr{W}$ as random variables that follow a Gaussian prior distribution $p(\mathscr{W})$. All these factors are considered independent inside and across the network levels. The independent assumption enables the extraction of independent features across different network levels. It facilitates the establishment of a manageable optimization problem, as computing the joint distribution of all layers is computationally demanding.

The true posterior distribution $p(\mathscr{W}|\mathscr{D})$, which represents the complete knowledge about the network parameters after seeing the training dataset, $\mathscr{D}$ can be calculated using Bayes' algorithm in Equation (3.1).

$$p(\mathscr{W}|\mathscr{D}) = \frac{p(\mathscr{D}|\mathscr{W})p(\mathscr{W})}{\int p(\mathscr{D}|\mathscr{W})p(\mathscr{W})d\mathscr{W}} \tag{3.1}$$

On the right-hand side, we observe the probability distribution of the data given the weights(likelihood) $p(\mathscr{D}|\mathscr{W})$, multiplied by the prior $p(\mathscr{W})$, and divided by the marginal probability distribution of the data. Nevertheless, the denominator of the equation necessitates the integration of all possible values for the parameters in the model. The model parameters in deep neural networks (DNNs) typically have many dimensions, and the presence of non-linearities further complicates the process of integrating them. Consequently, due to the excessively costly computation required to calculate this integral, employing the Bayesian technique for directly estimating the posterior distribution is no longer feasible. Therefore, it is not possible to accurately estimate the true distribution using Bayesian methods. Approximation Bayesian inference techniques, such as variational inference and Markov Chain Monte Carlo (MCMC), have been created to tackle the intricacy of Deep Neural Networks (DNNs) and facilitate Bayesian inference.

3.2.2 Variational inference

Variational inference (VI) is a method used to approximate Bayesian inference. It involves estimating the posterior distribution of the latent variables in a latent variable model when the true posterior is not directly available. It is a machine-learning technique that uses optimization approaches to estimate intricate probability distributions. As a result of this attribute, Variational Inference (VI) exhibits faster convergence compared to conventional methods such as Markov Chain Monte Carlo (MCMC) sampling. Variational inference (VI) attempts to estimate the posterior distribution by employing a distribution that is considered to be well-behaved. This suggests that integrals are computed in such a way that the level of accuracy increases as the precision of the approximation improves.

Variational Inference (VI) entails considering a parameterized variational posterior distribution $q_\phi(\mathscr{W})$ and subsequently estimating the actual posterior $p(\mathscr{W}|\mathscr{D})$.The optimization process involves decreasing the Kullback-Leibler (KL) divergence between the variational posterior distribution $q_\phi(\mathscr{W})$ and the genuine unknown posterior distribution $p(\mathscr{W}|\mathscr{D})$ [101].

$$\mathrm{KL}\left[q_\phi(\mathscr{W})\|p(\mathscr{W}|\mathscr{D})\right] = \int q_\phi(\mathscr{W}) \log \frac{q_\phi(\mathscr{W})}{p(\mathscr{W})p(\mathscr{D}|\mathscr{W})} d\mathscr{W}, \tag{3.2}$$

The evidence lower bound (ELBO) loss function $\mathfrak{L}(\phi;\mathscr{D})$ is minimized on the right-hand side of Equation 3.2 while training Bayesian transformer models using the gradient descent update method to optimize the variational parameters ϕ.

$$\mathfrak{L}(\phi;\mathscr{D}) = -E_{q_\phi(\mathscr{W})}\{\log p(\mathscr{D}|\mathscr{W})\} + \mathrm{KL}\left[q_\phi(\mathscr{W})\|p(\mathscr{W})\right]. \tag{3.3}$$

The ELBO loss function in Equation 3.3 consists of two components: the predicted log-likelihood of the training data given the model parameters and a regularization factor defined as the KL-divergence between the proposed variational distribution $q_\phi(\mathscr{W})$ and the prior distribution $p(\mathscr{W})$. Through the process of marginalizing the model parameters, we calculate the predictive distribution.

$$p(\hat{\mathbf{y}}|\hat{\mathbf{X}},\mathscr{D}) = \int p(\hat{\mathbf{y}}|\hat{\mathbf{X}},\mathscr{W})\, p(\mathscr{W}|\mathscr{D})\, d\mathscr{W}. \tag{3.4}$$

The Bayesian transformer architecture utilizes the mean and covariance matrix of the variational distribution, $q_\phi(\mathscr{W})$ to convey information. This information is then used to obtain the mean and covariance matrix of the predictive distribution, $p(\hat{\mathbf{y}}|\hat{\mathbf{X}},\mathscr{D})$ at the output of the model.

The average value of the predictive distribution signifies the image classification, while the covariance matrix depicts the level of uncertainty associated with the predicted output. The objective of the proposed models is to acquire uncertainty information regarding the output decision to enhance the reliability and dependability of the image classification task. This is achieved by employing Bayesian transformer models to ensure a secure implementation of sequence machine learning models in critical applications.

3.3 Mathematical Basis of the Image Classification Methods

This section presents the mathematical foundation of the proposed transformer models, namely Bayes-ViT and Bayes-CCT. The basic layout of a Bayes-ViT network is depicted in Fig. 3.3. We provide a mathematical derivation for quantifying uncertainty in Transformer models for the Self-attention layer. The same derivation can be applied to all layers of Bayes-ViT. A similar approach can be applied to Bayes-CCT. The key difference between these approaches is the initial tokenization method. The transformer encoder layer used in both architectures is identical in nature and mechanism.

3.3.1 Image Classification with Bayesian Vision Transformer, Bayes-ViT

The proposed Bayes-ViT model takes a sequence of non-overlapping image patches as input and linearly projects these patches into vectors, i.e., $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n \in \mathbb{R}^p$, where p is the size of each patch. The positional embedding is then applied to provide spatial information about the location of image patches within an input image. The embedding output is fed to the encoder structure of the Bayes-ViT model. The encoder structure consists of several layers, including the self-

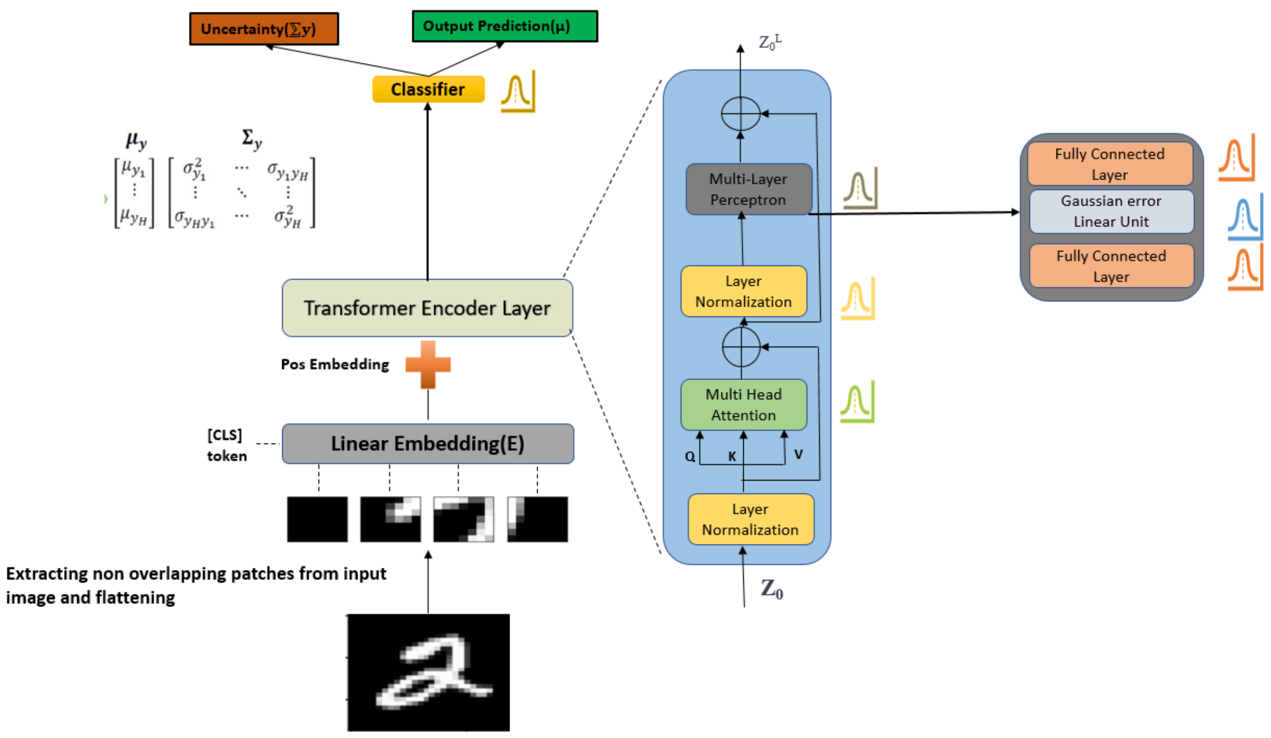


Figure 3.3: Illustration of the proposed image classification approach based on Bayes-ViT architecture.

attention function, the multi-layer perceptron (MLP), and the layer normalization. The self-attention mechanism is the core of the Bayes-ViT model because it ascertains the correlation between the image patches within the input sequence. Given a set of n query vectors $\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_n \in \mathbb{R}^{d_k}$, n key vectors $\mathbf{k}_1, \mathbf{k}_2, \cdots, \mathbf{k}_n \in \mathbb{R}^{d_k}$, and n value vectors $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n \in \mathbb{R}^n$, the attention mechanism maps the query vector, $\mathbf{q}_i$, the key vector, $\mathbf{k}_j$, and the value vector, $\mathbf{v}_j$ and computes a set of output vectors $\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_n \in \mathbb{R}^q$, such that

$$a_{ij} = \frac{\mathbf{k}_j^T \mathbf{q}_i}{\sqrt{d_k}}, \text{ and } \tilde{\mathbf{a}}_i = \varphi(\mathbf{a}_i) = \frac{\exp(\mathbf{a}_i)}{\sum_{j=1}^n \exp(a_{ij})}, \tag{3.5}$$

$$\mathbf{z}_i = \sum_{j=1}^{n} \tilde{\mathbf{a}}_i \odot \mathbf{v}_j, \text{ for } i, j = 1, \cdots, n. \tag{3.6}$$

where $\mathbf{q}_i = \mathbf{W}^{(\mathbf{q})}\, \mathbf{x}_i$, $\mathbf{k}_j = \mathbf{W}^{(\mathbf{k})}\, \mathbf{x}_j$, and $\mathbf{v}_j = \mathbf{W}^{(\mathbf{v})}\, \mathbf{x}_j$, and $\mathbf{W}^{(\mathbf{q})}$, $\mathbf{W}^{(\mathbf{k})}$, and $\mathbf{W}^{(\mathbf{v})}$ are the weight matrices. The query, key, and value vectors represent the linear projection of the input sequence, d_k denotes the dimension of the key vector, φ is the softmax function, and $\odot$ is the Hadamard product.

The MLP consists of two fully connected layers and a Gaussian error linear unit (GeLU) activation function.

We propagate the moments of the variational distributions, $q_(\mathcal{W})$, i.e., the mean and covariance matrix, through all layers of the Bayes-ViT model. In our proposed model, all the learnable parameters are random variables. In the self-attention function, we have inner products between two random vectors, Hadamard products between random vectors, and non-linear functions applied to random vectors. We will formulate the moment propagation for the self-attention function, and the mathematical relations can then be generalized to all layers.

Let, $\mathbf{w}_h^{(\mathbf{q})}$ be the h^{th} row vector of the weight matrix $\mathbf{W}^{(\mathbf{q})}$ where $h = 1, 2, \cdots, H$ and H is number of hidden nodes. The variational distribution is $\mathbf{w}_h^{(\mathbf{q})} \sim \mathscr{N}(\boldsymbol{\mu}_{\mathbf{w}_h^{(\mathbf{q})}}, \boldsymbol{\Sigma}_{\mathbf{w}_h^{(\mathbf{q})}})$. We assume the weight vectors and the input vector $\mathbf{x}_i$ are independent. Each row of the matrix $\mathbf{W}^{(\mathbf{q})}$ multiplies the vector $\mathbf{x}_i$ in the matrix-vector multiplication $\mathbf{q}_i = \mathbf{W}^{(\mathbf{q})}\mathbf{x}_i$. Thus, the inner product between each pair of independent random vectors, $\boldsymbol{\mu}_{\mathbf{w}_h^{(\mathbf{q})}}$ and $\mathbf{x}_i$ can be written as $q_i = (\mathbf{w}_h^{(\mathbf{q})})^T\ \mathbf{x}_i$. The mean and covariance of $\mathbf{q}_i$ can be derived as the following,

$$\boldsymbol{\mu}_{\mathbf{q}_i} = \mathbf{M}^{(\mathbf{q})}\boldsymbol{\mu}_{\mathbf{x}_i}, \text{ where } \mathbf{M}^{(\mathbf{q})} = [\boldsymbol{\mu}^T_{\mathbf{w}_{h_1}^{(\mathbf{q})}}] \tag{3.7}$$

$$\boldsymbol{\Sigma}_{\mathbf{q}_i} = \begin{cases} \operatorname{tr}\left(\boldsymbol{\Sigma}_{\mathbf{w}_{h_1}^{(\mathbf{q})}}\boldsymbol{\Sigma}_{\mathbf{x}_i}\right) + \boldsymbol{\mu}^T_{\mathbf{w}_{h_1}^{(\mathbf{q})}}\ \boldsymbol{\Sigma}_{\mathbf{x}_i}\ \boldsymbol{\mu}_{\mathbf{w}_{h_2}^{(\mathbf{q})}} + \boldsymbol{\mu}^T_{\mathbf{x}_i}\boldsymbol{\Sigma}_{\mathbf{w}_{h_1}^{\mathbf{q}}}\ \boldsymbol{\mu}_{\mathbf{x}_i}, & h_1 = h_2 \\ \boldsymbol{\mu}^T_{\mathbf{w}_{h_1}^{(\mathbf{q})}}\ \boldsymbol{\Sigma}_{\mathbf{x}_i}\ \boldsymbol{\mu}^T_{\mathbf{w}_{h_2}^{(\mathbf{q})}}. & h_1 \neq h_2 \end{cases}$$

Similarly, the mean and covariance matrix of the key vector $\mathbf{k}_i$, the value vector $\mathbf{v}_i$ can be derived as follows:

$$\boldsymbol{\mu}_{\mathbf{k}_i} = \mathbf{M}^{(\mathbf{k})}\boldsymbol{\mu}_{\mathbf{x}_i}, \text{ where } \mathbf{M}^{(\mathbf{k})} = [\boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{k})}_{h_1}}] \tag{3.8}$$

$$\boldsymbol{\Sigma}_{\mathbf{k}_i} = \begin{cases} \operatorname{tr}\left(\boldsymbol{\Sigma}_{\mathbf{w}^{(\mathbf{k})}_{h_1}}\boldsymbol{\Sigma}_{\mathbf{x}_i}\right) + \boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{k})}_{h_1}} \boldsymbol{\Sigma}_{\mathbf{x}_i} \boldsymbol{\mu}_{\mathbf{w}^{(\mathbf{k})}_{h_2}} + \boldsymbol{\mu}^T_{\mathbf{x}_i}\boldsymbol{\Sigma}_{\mathbf{w}^{\mathbf{k}}_{h_1}} \boldsymbol{\mu}_{\mathbf{x}_i}, & h_1 = h_2 \\ \boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{k})}_{h_1}} \boldsymbol{\Sigma}_{\mathbf{x}_i} \boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{k})}_{h_2}}. & h_1 \neq h_2 \end{cases}$$

$$\boldsymbol{\mu}_{\mathbf{v}_i} = \mathbf{M}^{(\mathbf{v})}\boldsymbol{\mu}_{\mathbf{x}_i}, \text{ where } \mathbf{M}^{(\mathbf{v})} = [\boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{v})}_{h_1}}] \tag{3.9}$$

$$\boldsymbol{\Sigma}_{\mathbf{v}_i} = \begin{cases} \operatorname{tr}\left(\boldsymbol{\Sigma}_{\mathbf{w}^{(\mathbf{v})}_{h_1}}\boldsymbol{\Sigma}_{\mathbf{x}_i}\right) + \boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{v})}_{h_1}} \boldsymbol{\Sigma}_{\mathbf{x}_i} \boldsymbol{\mu}_{\mathbf{w}^{(\mathbf{v})}_{h_2}} + \boldsymbol{\mu}^T_{\mathbf{x}_i}\boldsymbol{\Sigma}_{\mathbf{w}^{\mathbf{v}}_{h_1}} \boldsymbol{\mu}_{\mathbf{x}_i}, & h_1 = h_2 \\ \boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{v})}_{h_1}} \boldsymbol{\Sigma}_{\mathbf{x}_i} \boldsymbol{\mu}^T_{\mathbf{w}^{(\mathbf{v})}_{h_2}}. & h_1 \neq h_2 \end{cases}$$

The first-order Taylor approximation estimates the mean and covariance matrix after the non-linear activation functions in the model, including the softmax function. Thus, the mean and covariance of $\tilde{\mathbf{a}}_i$ in Equation 3.5 are derived as follows.

$$\boldsymbol{\mu}_{\tilde{\mathbf{a}}_i} \approx \varphi(\boldsymbol{\mu}_{\mathbf{a}_i}), \quad \boldsymbol{\Sigma}_{\tilde{\mathbf{a}}_i} \approx \mathbf{J}_\varphi \boldsymbol{\Sigma}_{\mathbf{a}_i} \mathbf{J}_\varphi^T, \tag{3.10}$$

where $\mathbf{J}_\varphi$ denotes the Jacobian matrix of $\tilde{\mathbf{a}}_i$ with respect to $\mathbf{a}_i$ evaluated at $\boldsymbol{\mu}_{\mathbf{a}_i}$. The results presented in Equation 3.10 hold for any non-linear activation function, including hyperbolic tangent (Tanh), sigmoid, or rectified linear unit (ReLU). The mean and covariance matrix of the element-wise multiplication in Equation 3.6, i.e., $\tilde{\mathbf{z}}_i = \tilde{\mathbf{a}}_i \odot \mathbf{v}_j$, are derived as follows,

$$\begin{aligned} \boldsymbol{\mu}_{\tilde{\mathbf{z}}_i} &= \boldsymbol{\mu}_{\tilde{\mathbf{a}}_i} \odot \boldsymbol{\mu}_{\mathbf{v}_j}, \\ \boldsymbol{\Sigma}_{\tilde{\mathbf{z}}_i} &= \boldsymbol{\Sigma}_{\tilde{\mathbf{a}}_i} \odot \boldsymbol{\Sigma}_{\mathbf{v}_j} + D(\boldsymbol{\mu}_{\mathbf{v}_j})\boldsymbol{\Sigma}_{\tilde{\mathbf{a}}_i} D(\boldsymbol{\mu}_{\mathbf{v}_j}) + D(\boldsymbol{\mu}_{\tilde{\mathbf{a}}_i})\boldsymbol{\Sigma}_{\mathbf{v}_j} D(\boldsymbol{\mu}_{\tilde{\mathbf{a}}_i}), \end{aligned} \tag{3.11}$$

where $D(\boldsymbol{\mu}_{\mathbf{v}_j})$ represents the diagonal matrix whose entries are given by the column vector $\boldsymbol{\mu}_{\mathbf{v}_j}$.

Consider d-dimensional input vectors $\boldsymbol{x}_i \in \mathbb{R}$ of a layer. Then, the output of a layer normalizing transform per i^{th} data sample, $\boldsymbol{y}_i^{LN}$, (assuming independent data samples), is given by,

$$\boldsymbol{y}_i^{LN} = \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}_i + \boldsymbol{\beta}, \hat{\boldsymbol{x}}_i = (\boldsymbol{x}_i - \boldsymbol{\mu}) \odot \frac{1}{\sqrt{\boldsymbol{\sigma}^2 + \varepsilon}} \tag{3.12}$$

where, $\boldsymbol{\mu}$, $\boldsymbol{\sigma}^2$ are the sample mean and sample variance , $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$ are hyper-parameters for scaling and shifting the input and ε is a constant to ensure numerical stability.

Since $\boldsymbol{\mu}$, $\boldsymbol{\sigma}^2$, $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$ and ε are deterministic quantities, then the propagation of the first two moments through the LN is derived as,

$$\boldsymbol{\mu}_{\mathbf{y}_i}^{LN} = \frac{\boldsymbol{\gamma}}{\sqrt{\boldsymbol{\sigma}^2 + \varepsilon}} \odot (\boldsymbol{\mu}_{x_i} - \boldsymbol{\mu}) + \boldsymbol{\beta} \tag{3.13}$$

$$\boldsymbol{\Sigma}_{\mathbf{y}_i}^{LN} = D(\frac{\boldsymbol{\gamma}}{\sqrt{\boldsymbol{\sigma}^2 + \varepsilon}}) \boldsymbol{\Sigma}_{\boldsymbol{x}_\mathbf{i}} D(\frac{\boldsymbol{\gamma}}{\sqrt{\boldsymbol{\sigma}^2 + \varepsilon}}) \tag{3.14}$$

where $D(x)$ is a diagonal matrix, whose diagonal entries are the entries of the column vector x, and $\odot$ is a Hadamard product.

Similarly, for the MLP layer, propagation of mean and covariance will be:

$$\boldsymbol{\mu}_{\mathbf{MLP}_i} = \mathbf{M}^{(\mathbf{MLP})} \boldsymbol{\mu}_{\mathbf{x}_i}, \text{ where } \mathbf{M}^{(\mathbf{MLP})} = [\boldsymbol{\mu}^T_{\mathbf{w}_{h_1}^{(\mathbf{MLP})}}] \tag{3.15}$$

$$\boldsymbol{\Sigma}_{\mathbf{MLP}_i} = \begin{cases} \operatorname{tr}\left(\boldsymbol{\Sigma}_{\mathbf{w}_{h_1}^{(\mathbf{MLP})}} \boldsymbol{\Sigma}_{\mathbf{x}_i}\right) + \boldsymbol{\mu}^T_{\mathbf{w}_{h_1}^{(\mathbf{MLP})}} \boldsymbol{\Sigma}_{\mathbf{x}_i} \boldsymbol{\mu}_{\mathbf{w}_{h_2}^{(\mathbf{MLP})}} + \boldsymbol{\mu}^T_{\mathbf{x}_i} \boldsymbol{\Sigma}_{\mathbf{w}_{h_1}^{(\mathbf{MLP})}} \boldsymbol{\mu}_{\mathbf{x}_i}, & h_1 = h_2 \\ \boldsymbol{\mu}^T_{\mathbf{w}_{h_1}^{(\mathbf{MLP})}} \boldsymbol{\Sigma}_{\mathbf{x}_i} \boldsymbol{\mu}^T_{\mathbf{w}_{h_2}^{(\mathbf{MLP})}}. & h_1 \neq h_2 \end{cases}$$

By propagating the variational moments through all layers, we obtain the moments of the predictive distribution, $p(\mathbf{y}|\mathbf{X}, \mathscr{D})$. The mean of $p(\mathbf{y}|\mathbf{X}, \mathscr{D})$, i.e., $\boldsymbol{\mu}_{\mathbf{y}}$ represents the network's prediction, while the covariance matrix, $\boldsymbol{\Sigma}_{\mathbf{y}}$, reflects the uncertainty associated with the output classification.

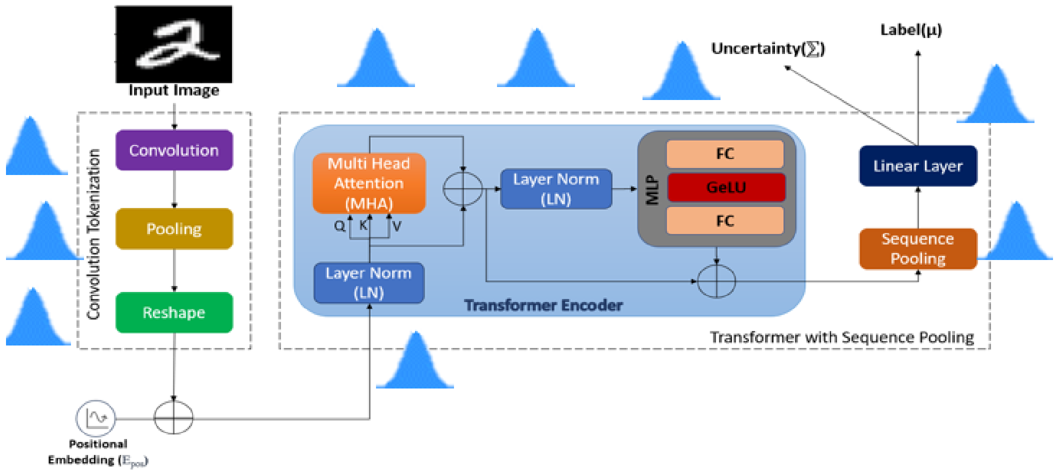


Figure 3.4: Illustration of the proposed image classification approach based on Bayes-CCT architecture.

3.3.2 Image Classification with Bayesian Compact Convolutional Transformer, Bayes-CCT

In this transformer-based model, convolution along with Rectified Linear Unit(ReLU) and Max-pooling were applied instead of patch-based tokenization. So, the density propagation through the transformer encoder follows the Bayes-ViT model. Figure 3.4 illustrates the block diagram of Bayes-CCT.

Convolutional Layer: The convolution operation between a set of kernels and the input image is formulated as a matrix-vector multiplication. We first form sub-tensors $\chi_{i:i=r_1-1,j:j=r_2-1}$ from the input tensor χ, having the same size as the kernels $W^{(k_c)} \in \mathbb{R}^{(r_1\times r_2\times K)}$. These sub-tensors are subsequently vectorized and arranged as the rows of a matrix $\hat{\mathbf{X}}$. Thus, convolving χ with the $k_c{}^{th}$ kernel $W^{(k_c)}$ is equivalent to multiplication of $\hat{\mathbf{X}}$ with $vec(W^{(k_c)})$,

$$z^{(k_c)} = \chi * W^{(k_c)} = \hat{\mathbf{X}} \times vec(W^{(k_c)}) \tag{3.16}$$

where $*$ denotes the convolution operation. We have defined variational distribution over kernels,$\boldsymbol{\Sigma}^{(k_c)})$, where $\mathbf{m}^{(k_c)} = vec(M^{(k_c)})$ and $\boldsymbol{\Sigma}^{(k_c)}) = \mathbf{U}^{(1,k_c)} \otimes \mathbf{U}^{(2,k_c)} \otimes \mathbf{U}^{(3,k_c)}$. It follows that the output of the convolution is derived by,

$$z^{(k_c)} \sim N(\mu_{z^{(k_c)}} = \hat{\mathbf{X}}\mathbf{m}^{(k_c)}, \boldsymbol{\Sigma}_{z^{(k_c)}} = \hat{\mathbf{X}}\boldsymbol{\Sigma}^{(k_c)}\hat{\mathbf{X}}^T) \tag{3.17}$$

In the first convolution layer, we assume that the input tensor χ is deterministic for simplicity.

Non-Linear Activation Function: We approximate the mean and covariance after a non-linear activation function ψ using the first-order Taylor series approximation [38]). Let $g^{(k_c)} = \psi[z^{(k_c)}]$, then the mean and covariance of $g^{(k_c)}$ are derived as follows:

$$\mu_{g^{(kc)}} \approx \psi(\mu_{\mathbf{z}^{(\mathbf{kc})}}), \tag{3.18}$$

$$\boldsymbol{\Sigma}_{z^{(kc)}} \approx \boldsymbol{\Sigma}_{z^{(kc)}} \odot (\nabla\psi(\mu_{\mathbf{z}^{(\mathbf{kc})}})\nabla\psi(\mu_{\mathbf{z}^{(\mathbf{kc})}})^T) \tag{3.19}$$

where ∇ is the gradient with respect to $z^{(k_c)}$ and $\odot$ is the Hadamard product. The state-of-the-art activation functions in DNNs, i.e., the Rectified Linear Unit (ReLU), and its variations can be approximately considered piece-wise linear. Thus, the first-order approximation may provide satisfactory results when propagating the first two moments of the variational distribution through these activation functions.

Max-Pooling Layer: For the max-pooling, $\mu_{p^{(kc)}} = pool(\mu_{g^{(kc)}})$ and $\boldsymbol{\Sigma}_{p^{(kc)}} = co-pool(\boldsymbol{\Sigma}_{g^{(kc)}})$, where pool represents the max-pooling operation on the mean and co-pool represents down-sampling the covariance, i.e., keeping only the rows and columns of $\boldsymbol{\Sigma}_{g^{(kc)}}$ that correspond to the pooled mean elements.

The encoder block of Bayes-CCT is similar to Bayes-ViT, and it will follow the same derivation for layer normalization, multi-head attention and MLP layer.

By propagating the variational moments through all layers, we obtain the moments of the predictive distribution, $p(\mathbf{y}|\mathbf{X}, \mathscr{D})$. The mean of $p(\mathbf{y}|\mathbf{X}, \mathscr{D})$, i.e., $\boldsymbol{\mu}_\mathbf{y}$ represents the network's prediction, while the covariance matrix, $\boldsymbol{\Sigma}_\mathbf{y}$, reflects the uncertainty associated with the output classification.

3.4 Algorithm of proposed model

3.4.1 Algorithm of proposed Bayes-ViT

Algorithm 1 Proposed Image Classification with Bayes-ViT

```
Require: Total number of training epoch $Max-epoch$, initial learning rate $\eta$, batch size and
    weight decay $\beta$
 1: Initializing the variational parameters $\phi = \{\mu, \sum\}$, where $\mu$ is the mean and $\sum$ is the covariance
    matrix of the Bayes-CCT random parameters.
 2: for $epoch < Max-epoch$ do
 3:   for $t < batchsize$ do
 4:     Observe the input image samples $N$.
 5:     Apply patch-based tokenization to obtain the sequence of image patches.
 6:     Add Positional Embedding(optional) to each patch entering Transformer Encoder.
 7:     The training dataset is $D = \{X^{(n)}, y^{(n)}\}_{n=1}^{N}$, where $y^{(n)}$ represents true image labels.
 8:     Propagate $\mu$ and $\sum$ through the Bayes-ViT architecture, including different layers and
    non-linear activation functions.
 9:     Calculate loss $\mathscr{L}(\phi;D) = -E_{q_\phi(\Omega)}\{logp(D|\Omega)\} + \beta KL[q_\phi(\Omega)||p(\Omega)]$.
10:     Calculate the gradient of the loss $\Delta\mathscr{L}(\phi;D)$.
11:     Use ADAM optimizer with decaying learning rate $\eta$.
12:     Optimize the ELBO objective function and update the Bayes-CCT variational parameters
    using the gradient descent update rule:
                $\phi \leftarrow \phi - \eta\Delta\mathscr{L}(\phi;D)$
13:   end for
14: end for
```

3.4.2 Algorithm of proposed Bayes-CCT

Algorithm 2 Proposed Image Classification with Bayes-CCT

```
Require: Total number of training epoch $Max-epoch$, initial learning rate $\eta$, batch size and weight decay $\beta$
 1: Initializing the variational parameters $\phi = \{\mu, \sum\}$, where $\mu$ is the mean and $\sum$ is the covariance matrix of the Bayes-CCT random parameters.
 2: for $epoch < Max-epoch$ do
 3:   for $t < batchsize$ do
 4:     Observe the input image samples $N$.
 5:     Apply convolutional tokenization to obtain the sequence of image patches.
 6:     Add Positional Embedding(optional) to each convolutional patch entering Transformer Encoder.
 7:     The training dataset is $D = \{X^{(n)}, y^{(n)}\}_{n=1}^{N}$, where $y^{(n)}$ represents true image labels.
 8:     Propagate $\mu$ and $\sum$ through the Bayes-CCT architecture, including different layers and non-linear activation functions.
 9:     Calculate loss $\mathscr{L}(\phi; D) = -E_{q_\phi(\Omega)}\{logp(D|\Omega)\} + \beta KL[q_\phi(\Omega)||p(\Omega)]$.
10:     Calculate the gradient of the loss $\Delta\mathscr{L}(\phi; D)$.
11:     Use ADAM optimizer with decaying learning rate $\eta$.
12:     Optimize the ELBO objective function and update the Bayes-CCT variational parameters using the gradient descent update rule:
            $\phi \leftarrow \phi - \eta\Delta\mathscr{L}(\phi; D)$
13:   end for
14: end for
```

CHAPTER IV

EXPERIMENTAL RESULTS AND ANALYSIS

This section compares Bayes-ViT and Bayes-CCT architectures with their deterministic counterparts while solving multi-label classification problems. The Bayes-ViT model comprises three key parts: (1) patch-based tokenization and embedding along with positional encoding, (2) transformer encoder layer, and (3) the final classifier at the output. The encoder layer contains (GeLU) and softmax activation function, whereas the final classification head consists of softmax activation. On the other hand, the Bayes-CCT model contains four important sections: (1) The convolution-based tokenization along with positional encoding; (2) the transformer encoder layer; (3) Sequence Pooling; and (4) Final Classifier at the output. The encoder layer contains (GeLU) and softmax activation function, whereas sequence pooling and final classifier contain softmax activation. Both Bayes-ViT and Bayes-CCT models were trained using the Adam optimizer [52] along with a decaying learning rate and polynomial schedule [2]. The Bayes-ViT, Bayes-CCT, Deterministic ViT, and Deterministic CCT are both trained and fine-tuned for three distinct image classification datasets(Bayes-ViT trained and fine-tuned on two datasets whereas, Bayes-CCT trained and fine-tuned on one dataset). Table 4.1 and Table 4.2 present hyperparameters utilized in the optimized models for each dataset. We utilized the same input size, sequence length, no. of layers, head number, and embedding dimension to demonstrate fair comparison. For deterministic CCT and Bayes-CCT, convolutional kernel size, pooling size, kernel, and pooling stride are kept the same. In order to optimize the performance of each model, the remaining hyperparameters, including batch size, epoch number, initial learning rate, final learning rates, and KL weighting factor, are provided for both deterministic and Bayesian models.

Table 4.1: Hyperparameter for ViT

	Dataset	Input Size	Patch Size	No. of Encoder Layer	Hidden Units	Batch Size	Epoch N o.	Initial LR	Final LR	KL Weight Factor
Deterministic ViT	MNIST	28	4	5	64	20	300	0.001	0.00001	0.00001
	Fashion-MNIST	28	8	7	64	50	500	0.001	0.00001	0.001
Bayesian VIT	MNIST	28	4	5	64	20	300	0.001	0.00001	0.00001
	Fashion-MNIST	28	8	7	64	50	500	0.001	0.00001	0.001

Table 4.2: Hyperparameter for CCT

	Dataset	Input Size	Kernel Size	No. of Conv layers	Kernel No.	Hidden Units	Pooling Size	Pooling Stride	No. of Encoder Layer	Batch Size	Epoch No.	Initial LR	Final LR	KL Weight Factor
Deterministic CCT	CIFAR-10	32	5	5	128	128	2	2	3	30	350	0.001	1E-04	0.001
Bayesian CCT	CIFAR-10	32	5	5	128	128	2	2	3	30	350	0.001	1E-04	0.001

4.1 Experimental Setup.

Three distinct experiments or prediction tasks are conducted to solve the image classification task to assess the performance of the proposed Bayesian Transformer models.

4.1.1 Multi Label Classification Problem.

The strategy involves solving multi-label classification problems based on image data provided during testing. It is noteworthy to emphasize that in all of the datasets utilized for experiments, each image belongs to one particular class. Therefore, the training procedures were conducted independently for images within a specific dataset, utilizing the supervised learning technique. Each dataset is divided into training and validation sets containing images to corresponding class labels by default. The learning and assessment of the model are adjusted to the attributes and trends associated with the objects within the training data as a result of this division.

4.2 Dataset Selection for Model Development

Three publicly available multi-label image classification datasets, namely MNIST [23], Fashion-MNIST [104], and CIFAR-10 [54], consisting of labeled and unlabeled images for training and testing purpose, were utilized in our experimental analysis analysis.

4.2.1 MNIST dataset.

The Modified National Institute of Standards and Technology (MNIST) database is a substantial collection of handwritten digits frequently employed to train several image processing

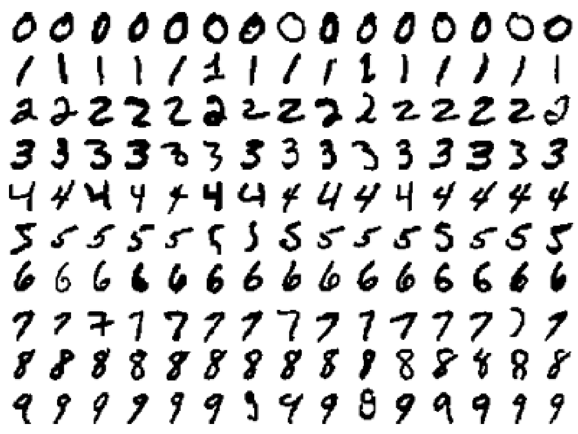
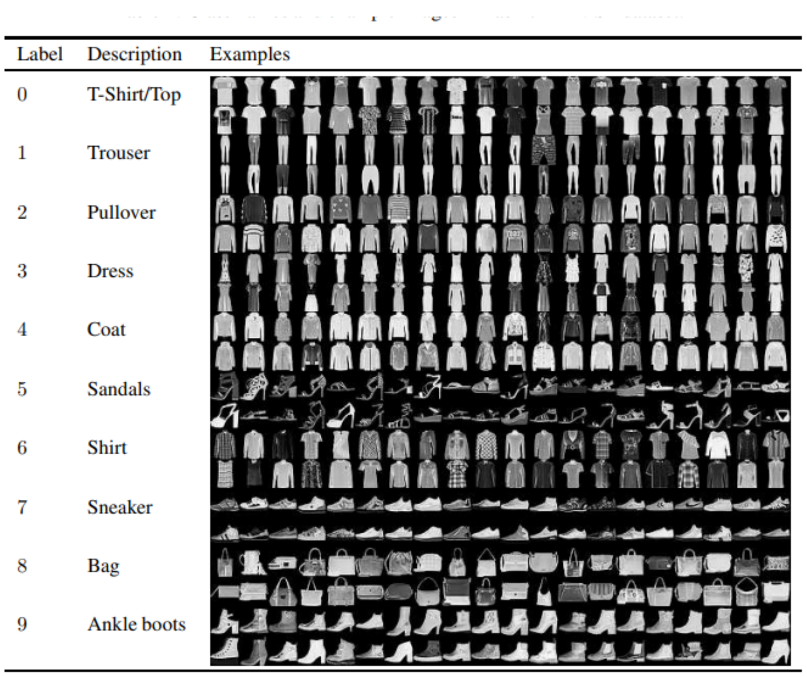


Figure 4.1: Some sample images of MNIST [23].

algorithms. This dataset is integrated into TensorFlow and can be accessed easily. The training dataset comprises 60,000 images, while the validation dataset contains 10,000 images. Each image in both datasets corresponds to a single handwritten digit, resulting in 10 classes. Figure 4.1 demonstrates some sample images of MNIST. Each class has 7,000 images, with roughly 6,000 images allocated for training and 1,000 for testing. The numerical characters have undergone a standardization process in size and have been positioned at the center of a predetermined image size. The images were aligned in the center of a 28x28 image by determining the center of mass of the pixels. Subsequently, the image was shifted to position this point precisely at the center of the 28x28 field. The objective of utilizing this dataset is to categorize a provided image of a handwritten numeral into one of ten categories, which correspond to integer values ranging from 0 to 9, inclusively. The machine learning community widely uses this dataset for testing image classification models.

4.2.2 Fashion-MNIST dataset

The Fashion-MNIST dataset comprises a training set including 60,000 samples and a test set containing 10,000 examples. Each image in both sets corresponds to a single digit, resulting in 10 classes. Each instance is a gray-scale image with dimensions of 28x28 and is assigned a label from 10 classes. Figure 4.2 demonstrates some sample images of Fashion-MNIST. Each image has dimensions of 28 pixels in height and 28 pixels in width, resulting in a total of 784 pixels. Every pixel is assigned a pixel value, which represents the brightness level of that pixel. Higher pixel



Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Figure 4.2: Some sample images of Fashion-MNIST [104].

values correspond to deeper shades. The pixel value is a discrete numerical value ranging from 0 to 255. This dataset is integrated into TensorFlow and can be accessed easily.

4.2.3 CIFAR-10 dataset

Lastly, The CIFAR-10(Canadian Institute for Advanced Research) is another popular benchmark dataset comprising 60,000 color images, each measuring 32x32 pixels. These images are categorized into ten distinct classes, each including 6,000 images. The dataset consists of 50,000 training images and 10,000 test images. The dataset is extensively utilized within machine learning research for solving computer vision problems. The classes exhibit perfect mutual exclusivity. The set of ten distinct categories encompasses various objects: airplanes, vehicles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each class consists of a total of 6,000 photos. Figure 4.3 demonstrates some sample images of Fashion-MNIST. Like MNIST and Fashion-MNIST, this dataset is integrated into TensorFlow and is easily accessed. The images comprising CIFAR-10 depict commonplace objects, enhancing the dataset's applicability to practical scenarios. In contrast to certain datasets (e.g., MNIST and Fashion-MNIST), which offer gray-scale images, CIFAR-10 comprises three channels.

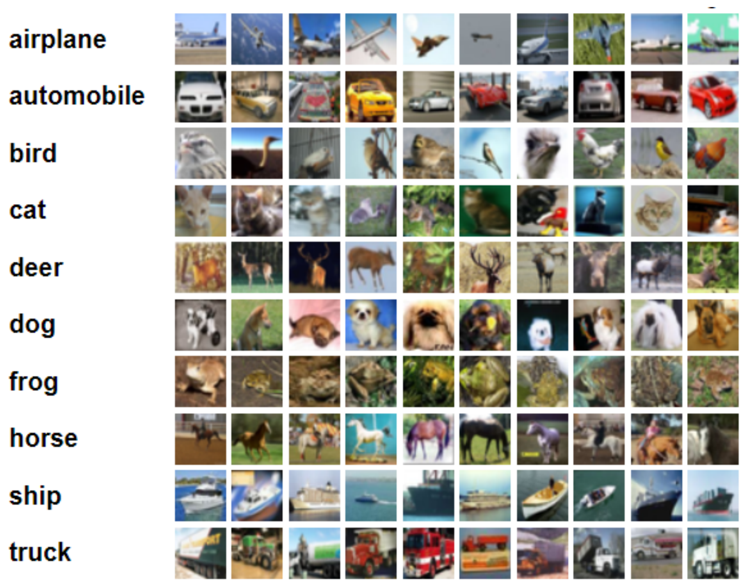


Figure 4.3: Some sample images of CIFAR-10 [54].

4.3 Performance Evaluation

4.3.1 Robustness and Noise Analysis

The robustness of the proposed Bayesian Transformer models is assessed through a comparative analysis of their performance with that of the deterministic models across a range of noise scenarios. Initial training for each model is conducted using noise-free image examples. Then, numerous adversarial attacks and Gaussian noise of varying magnitudes are introduced during testing as part of the preprocessing phases. The standard deviation (SD) serves as a noise intensity metric for Gaussian noise. We modify the standard deviation value and utilize varying noise levels (low, medium, and high) to introduce randomness into the test data adequately. The adversarial examples are generated by implementing two distinct methodologies: the projected gradient descent (PGD) and the fast gradient sign method (FGSM). [34, 67]. The noise utilized in the FGSM adversarial attack is generated through the multiplication of the test samples by ε, i.e., $\varepsilon \text{sign}\left[\nabla_{\mathbf{X}}\mathfrak{L}(\phi;\mathbf{X},\mathbf{y})\right]$, and $\nabla_{\mathbf{X}}$ is the gradient of the loss function (ELBO loss in the Bayesian Transformer models) represents the gradient with respect to the input image [34]. In order to generate FGSM attacks against the deterministic ViT and CCT, the gradient of the cross-entropy loss function of the deterministic network has been computed with respect to the input image. For

Table 4.3: The level of Gaussian noise (standard deviation (SD)) and the strength of adversarial attacks (ε) applied for MNIST and Fashion MNIST using Bayes-ViT

Bayesian Vision transformer models	Noise Type	Noise Level	MNIST	F-MNIST
Bayesian ViT	Gaussian	Low	0.05	0.05
		Medium	0.1	0.1
		High	0.2	0.2
	FGSM	Low	0.001	0.001
		Medium	0.005	0.005
		High	0.05	0.05
	PGD	Low	0.001	0.001
		Medium	0.005	0.005
		High	0.05	0.05

FGSM to generate the perturbation, the sign of the gradient of the loss with regard to the input is directly taken after the gradient has been computed as a one-step operation. In contrast, PGD is an attack based on iterative optimization that utilizes numerous iterations of FGSM. With a small step size, α, PGD is an iterative assault in which FGSM is applied repeatedly.

The number of iterations is 20, and the step size, α, is configured to 1 for our simulation. The clipping operation ensures that the adversarial cases (ε-neighborhood) closely resemble the original data. The selection of the three levels of adversarial attacks (Low, Medium, and High) for both FGSM and PGD is accomplished by modifying the ε value. The Table also provides the ε values corresponding to each level of adversarial attack, in addition to the SD used to generate distinct levels of Gaussian noise. Along with the SD used to produce different levels of Gaussian noise, the ε values for each level of adversarial attack are given in Table 4.3 and Table 4.4. The SD and ε values may exhibit variability across numerous datasets due to noise.

4.4 Results and Discussion

The performance of proposed Bayesian transformer models in comparison to their deterministic counterparts across various noise levels, as measured by the MNIST, Fashion-MNIST, and

Table 4.4: The level of Gaussian noise (standard deviation (SD)) and the strength of adversarial attacks (ε) applied for CIFAR-10 using Bayes-CCT

Bayesian Vision transformer model	Noise Type	Noise Level	CIFAR-10
Bayesian CCT	Gaussian	Low	0.05
		Medium	0.1
		High	0.2
	FGSM	Low	0.001
		Medium	0.01
		High	0.1
	PGD	Low	0.001
		Medium	0.01
		High	0.1

CIFAR-10 datasets, is presented in the following tables. As shown in Table 4.5, the accuracy of the proposed Bayes-ViT and Bayes-CCT models for the MNIST dataset remains significantly higher than that of corresponding deterministic models in different noise settings. The Bayesian Transformer models demonstrate robustness in accurately identifying the appropriate class in prediction tasks even when the input images from the test set are subjected to various forms of distortion, such as random noise and adversarial perturbations.

The accuracy of the proposed Bayes-ViT is equivalent to that of the deterministic transformer-based models when evaluated on noise-free test data of the MNIST dataset. Nevertheless, the precision of deterministic models significantly diminishes when noise levels progressively escalate, particularly in the presence of strong adversarial attacks. Gaussian noise at a moderate level does not have any discernible effect on either model. However, a high level of Gaussian noise significantly reduces the accuracy of the deterministic model. The Bayes-ViT model, as proposed, exhibits robust accuracy even when confronted with significant noise levels. The highest accuracy for the two models is emphasized for the highest noise level. As an illustration, the Bayes-ViT model achieves accuracy rates of 87.01% and 88.59% when subjected to the most severe FGSM and PGD

Table 4.5: Classification accuracy of the proposed Bayes-ViT and Deterministic ViT models using MNIST dataset for various levels of Random noise and FGSM and PGD adversarial attacks.

Noise Type	Noise level	Bayes-ViT	Deterministic ViT
No Noise		90.08	88.1
Gaussian	Low	90.01	85.57
	Medium	89.53	84.57
	High	**86.50**	**78.22**
FGSM	Low	89.43	85.7
	Medium	89.43	84.88
	High	**87.01**	**56.52**
PGD	Low	89.59	86.07
	Medium	89.53	85.12
	High	**88.59**	**80.92**

adversarial noise levels, respectively. In contrast, the deterministic ViT model achieves accuracy rates of 56.52% and 80.92% under the same conditions on the MNIST dataset. The reaction to FGSM and PGD attacks exhibits comparable patterns, wherein the deterministic models experience a more pronounced degradation in accuracy over time.

Table 4.6 showcases the test accuracy of the Fashion-MNIST dataset across various noise levels. When the test data is devoid of noise, both the suggested Bayesian and deterministic transformer models exhibit comparable levels of accuracy. However, the classification accuracy of deterministic models diminishes when subjected to Gaussian noise or an adversarial attack containing substantial noise. The accuracy of the deterministic model experiences a significant decrease when subjected to adversarial noise generated by FGSM and PGD techniques. Compared to the deterministic model, the proposed model demonstrates higher accuracy rates of 51.5% and 68.00% when subjected to the most intense levels of FGSM and PGD adversarial noise, respectively, on the Fashion-MNIST dataset. In contrast, the deterministic model achieves accuracy rates of 28.9% and 45.9% under the same conditions. As a result, their performance is less susceptible to being impacted by high-noise environments.

Table 4.6: Classification accuracy of the proposed Bayes-ViT and Deterministic ViT models using Fashion-MNIST for various levels of Random noise and FGSM and PGD adversarial attacks.

Noise Type	Noise level	Bayes-ViT	Deterministic ViT
No Noise		82.44	79.9
Gaussian	Low	81.60	79.2
	Medium	75.40	72.4
	High	**52.20**	**47.7**
FGSM	Low	81.10	79.23
	Medium	77.97	76.05
	High	**51.50**	**28.9**
PGD	Low	82.00	79.01
	Medium	80.30	77.2
	High	**68.00**	**45.9**

The test accuracy for the Bayes-CCT architecture in addressing the image classification problem using the CIFAR-10 dataset is depicted in Table 4.7. In the absence of noise, it is evident that the accuracy of the proposed model is approximately similar to that of its deterministic counterpart. The accuracy values produced by the deterministic CCT model are 38.93%, 46.55%, and 31.65% for the high levels of Gaussian, FGSM, and PGD attacks, respectively, as presented in Table 4.4.On the contrary, the Bayes-CCT model generates 49.03%, 49.83%, and 66.34% accuracy, respectively. Similarly, in comparison to a deterministic model, the Bayes-CCT exhibits a higher level of accuracy even when subjected to heavy adversarial attacks.

4.5 Uncertainty Analysis for Self-Awareness

This section examines the uncertainty of the Bayesian transformer models in the presence of noise in various contexts, including Gaussian noise, FGSM, and PGD adversarial attacks. The quantification of the noise level is accomplished by utilizing the signal-to-noise ratio (SNR) at each respective noise level. The expected variance of the Bayesian Transformer models is calculated in relation to the signal-to-noise ratio (SNR), resulting in a variance-vs-SNR curve. This analysis is

Table 4.7: Classification accuracy of the proposed Bayes-CCT and Deterministic CCT models using CIFAR-10 for various levels of Random noise and FGSM and PGD adversarial attacks.

Noise Type	Noise level	Bayes-CCT	Deterministic CCT
No Noise		85.92	88.80
Gaussian	Low	77.82	77.19
	Medium	67.10	61.40
	High	**49.03**	**38.93**
FGSM	Low	84.69	83.08
	Medium	78.91	78.61
	High	**49.83**	**46.55**
PGD	Low	86.53	86.19
	Medium	85.22	76.39
	High	**66.34**	**31.65**

conducted for each of the three datasets, encompassing different prediction tasks. The anticipated variance is utilized as a metric to quantify the level of uncertainty in the models, particularly in the presence of noise.

The results of this study demonstrate a consistent pattern across all classes and prediction tasks, indicating a notable rise in predictive variance as the signal-to-noise ratio (SNR) decreases. It is important to note that the x-axis in the graphs should be read from right to left.

For all classes and all prediction tasks, the findings indicate an increase in predictive variance (the x-axis is read from right to left) with decreasing SNR values. Hence, it can be observed that the proposed Bayes-ViT and Bayes-CCT models exhibit a rise in uncertainty with an increase in noise level (or a corresponding decrease in signal-to-noise ratio), leading to a degradation in model accuracy. This behavior is referred to as "*self-assessment*" as it involves the model evaluating its own performance and identifying its failure mode in response to a considerable increase in noise level.

In contrast to the Fast Gradient Sign Method (FGSM), the Projected Gradient Descent (PGD) approach iteratively generates noise. The quick increase in model uncertainty caused by FGSM

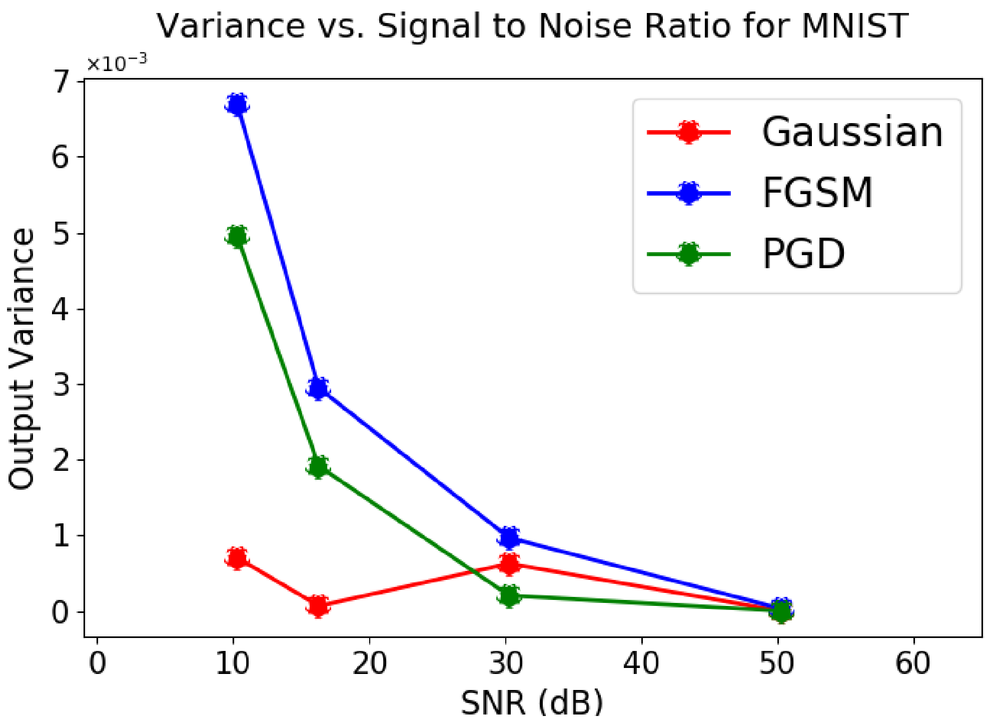


Figure 4.4: Average predictive variance plotted against SNR under Gaussian noise, FGSM, and PGD adversarial attack for Bayes-ViT for MNIST. The statistical increase in the variance can be observed in respective points. A significant increase in the variance can serve as a "red flag" and initiate the process of manual review of the input.

and PGD adversarial noise is substantially greater than that of Gaussian noise. This observation is made by examining the uncertainty of the proposed Bayesian transformer models across all classes under noisy conditions. This assertion is grounded in the observation that adversarial attacks are intentionally crafted to impact models' performance detrimentally.

The variance-versus-SNR curve for the MNIST dataset is depicted in Figure 4.4. The Bayes-ViT model demonstrates a significant rise in model uncertainty with higher noise levels. The escalation of uncertainty is shown to transpire faster in adversarial attacks than in the presence of Gaussian noise. The progressive escalation of uncertainty directly impacts the learning mechanism of the Bayes-ViT model, enhancing its robustness and overall performance. Uncertainty is crucial in maintaining the integrity of significant data attributes while reducing vulnerable and redundant aspects that attacks may significantly impact.

Figure 4.5 presents the noise analysis conducted on the Bayes-ViT model using the Fashion-MNIST dataset. We investigate the impact of various types of noise on the level of uncertainty

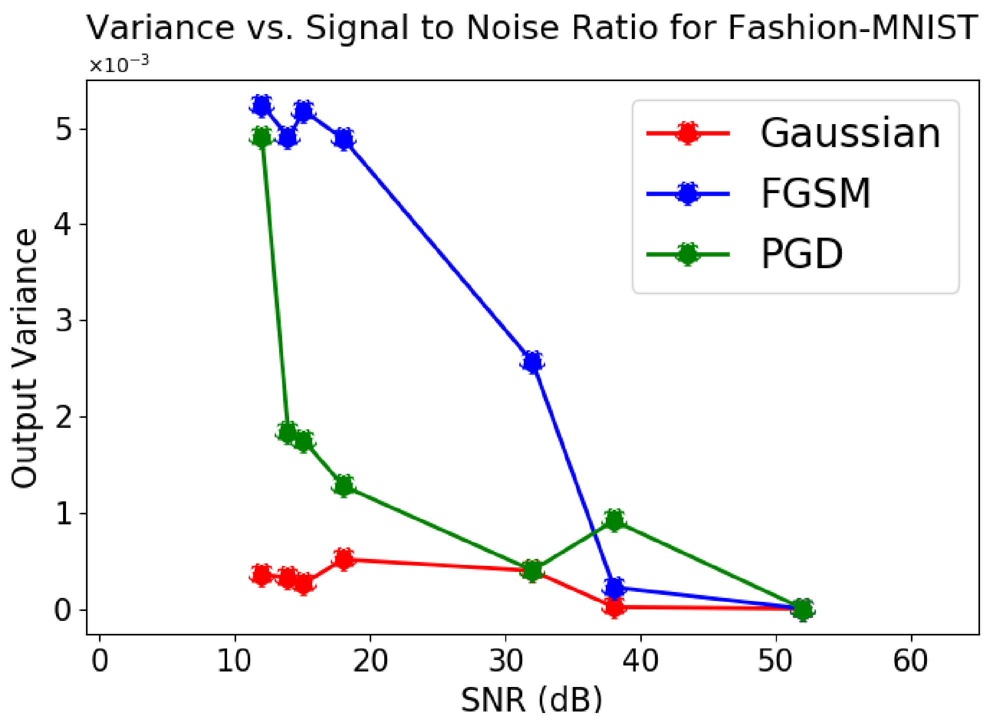


Figure 4.5: Average predictive variance plotted against SNR under Gaussian noise, FGSM, and PGD adversarial attack for Bayes-ViT for Fashion-MNIST. The statistical increase in the variance can be observed in respective points. A significant increase in the variance can serve as a “red flag” and initiate the process of manual review of the input.

exhibited by the model in the context of a prediction job. The Bayes-ViT model uncertainty has a similar pattern to that observed in the MNIST dataset. The pace at which uncertainty increases is higher in the context of adversarial attacks compared to the presence of Gaussian noise. Escalation of variance has been observed to occur consistently when both Gaussian noise and adversarial attacks are present.

The noise analysis of the Bayes-CCT model using the CIFAR-10 dataset is illustrated in Figure 4.6. Similarly, our study aims to examine the effects of different forms of noise on the degree of uncertainty displayed by the model within the framework of a prediction task. Unlike Bayes-ViT, the Bayes-CCT model depicts the increase of uncertainty(variance) with a higher Gaussian Noise, FGSM, and PGD adversarial attack. Higher variance values have been witnessed for both Gaussian noise and adversarial attacks for the Bayes-CCT model, unlike Bayes-ViT.

Consequently, using Bayesian Transformer models presents a promising approach to address the challenge of picture prediction, irrespective of the presence of many forms of noise or the

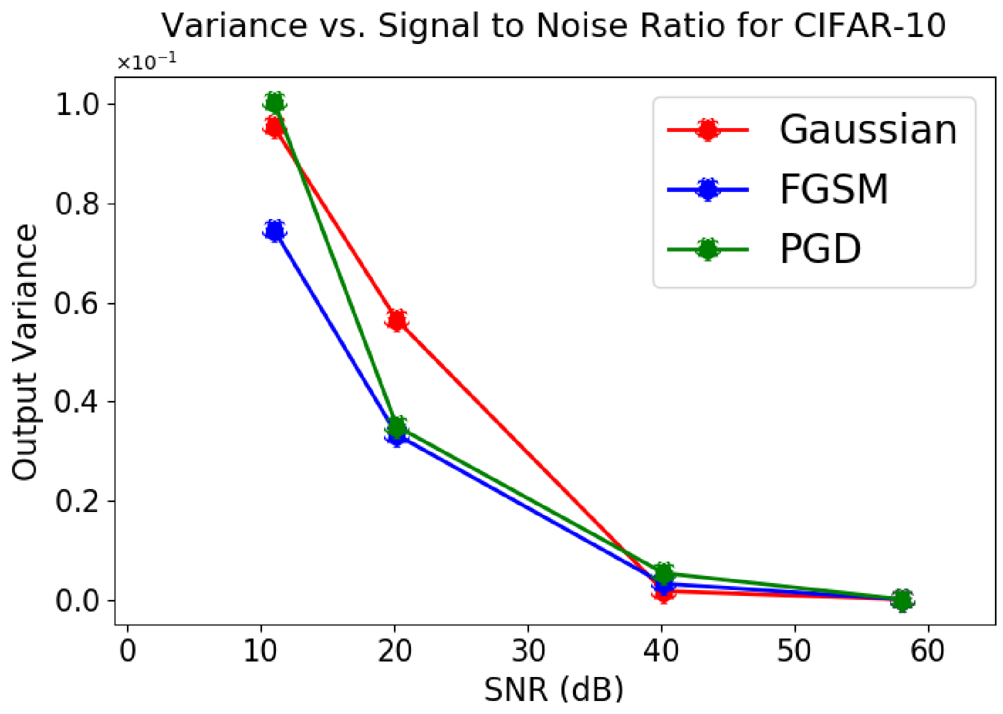


Figure 4.6: Average predictive variance plotted against SNR under Gaussian noise, FGSM, and PGD adversarial attack for Bayes-CCT for CIFAR-10. The statistical increase in the variance can be observed in respective points. A significant increase in the variance can serve as a "red flag" and initiate the process of manual review of the input.

susceptibility to malicious attacks. The Bayesian transformer models can differentiate between input images that are noisy or assaulted, as evidenced by the discernible rise in uncertainty observed in the projected variance as the level of noise or severity of the attack escalates.

CHAPTER V

FUTURE WORK AND CONCLUSION

5.1 Future Work

The integration of robustness and Bayesian approaches within the framework of Vision Transformer (ViT) entails the development of a model that not only exhibits high performance on the given task but also incorporates the consideration of uncertainty and variability inherent in the data. Two critical fields of computer vision where these Bayes-ViT and Bayes-CCT models can be applied.

5.1.1 Detection of Disease in Biomedical Imaging

The efficient detection of diseases by biomedical imaging plays a crucial role in the early diagnosis and successful treatment of various medical problems. The utilization of Vision Transformer (ViT) based models has exhibited promising results. In addition to making predictions, Bayes-ViT and Bayes-CCT architectures can provide estimations of uncertainty in conjunction with these predictions. In the usual practice, medical professionals such as physicians or radiologists are responsible for examining and analyzing biomedical images. The emergence of sophisticated imaging methods, combined with an unprecedented degree of processing power, offers a distinctive opportunity to analyze and address biomedical image classification challenges in previously unachievable ways. The decisions made regarding biomedical image classifications directly impact patient safety. If these classifications contain errors or are unreliable, there is a risk of misidentifying medical conditions and selecting incorrect treatment options. This poses a potential threat to the well-being of patients. Understanding the level of uncertainty associated with a diagnosis in medical imaging is of utmost importance. This may lead to enhanced decision-making abilities among

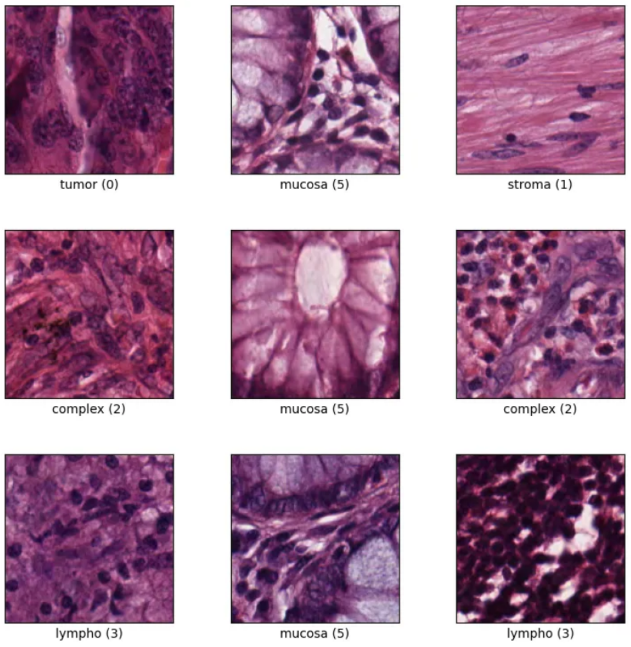


Figure 5.1: Some sample images of Colorectal Histology [46].

radiologists, particularly in scenarios characterized by unpredictable or ambiguous patterns seen by the model.

Bayesian Transformer models exhibit enhanced robustness towards variations in noise, imaging circumstances, and other forms of variability due to their inherent ability to manage and simulate uncertainty effectively. One particular application field where our developed Bayesian Transformer models can be effectively utilized is the categorization of textures of colorectal cancer using the "colorectal histology dataset" [46]. The dataset comprises 5000 image data, with an equal distribution of 625 images for each class(in total- 8 classes). Figure 5.1 demonstrates some image samples of the dataset. All of the input images have dimensions of 150x150 pixels and three channels. Eight classes include: 0) tumor epithelium, 1) simple stroma, 2) complex stroma (comprising single cancer cells and single immune cells), 3) immune cell conglomerates, 4) debris and mucus, 5) mucosal glands, 6) adipose tissue, and 7) background. With advanced pre-processing and regularization techniques, Bayes-CCT architecture can be effectively utilized for detecting cancer. The robustness and self-evaluation properties of Bayesian transformer models will help detect cancer during both favorable and unfavorable conditions and plan proactive treatment procedures.

5.1.2 Object Detection in autonomous driving / Self-driving cars

Autonomous driving systems often operate inside complex and dynamic environments. The system can assess its confidence level in the identified things or aspects of a scene by using the Bayesian Transformer designs, which can produce uncertainty estimation for its predictions. The utilization of knowledge becomes crucial in the process of decision-making, especially in situations when the environment is subject to high levels of noise or adversarial attacks. Robustness is a crucial element in the development of autonomous driving systems since it necessitates the constant functioning of the system across diverse environmental contexts, including factors such as fluctuating lighting conditions, variable weather patterns, and diverse road surfaces. Robustness and Self-Awareness properties in Bayes-Vit and Bayes-CCT have the potential to enhance the operational capabilities of autonomous vehicles by effectively handling uncertainties and unanticipated events. Over the course of time, Bayesian transformer models can facilitate continuous learning and adaptation. In autonomous driving, this characteristic confers a notable benefit as the system can acquire knowledge from novel data and encounters, augmenting its capabilities over time and adapting to alterations in the surrounding environment.

Moreover, the Bayes-ViT architecture can be integrated with U-net architecture to create a framework for robust semantic segmentation tasks. The Bayes-ViT architecture can capture both global context and uncertainty information. On the other hand, the U-Net-like structure is designed to preserve spatial information and effectively handle local details. One particular application can be brain tumor segmentation using the BraTS dataset [69]. The BraTS dataset has consistently prioritized assessing cutting-edge techniques for segmenting brain tumors in multi-modal magnetic resonance imaging (MRI) scans. However, robust image segmentation would require advanced pre-processing and data augmentation techniques to perform the segmentation task efficiently.

5.2 Conclusion

In this work, we have developed a new image classification technique using the Bayes-Vit and Bayes-CCT architecture. The Bayesian inference enables estimating the variational distribution specified over the model’s parameters. The prediction result is obtained from the mean of the predictive distribution in the output of the Bayesian transformer models. On the other hand, the uncertainty in the predicted label is captured by the covariance matrix. The Bayesian transformer models under consideration are trained and tested on three datasets, each consisting of 60,000 image data. Experimental results demonstrate the superiority of the proposed Bayesian models’ robustness compared to deterministic counterparts during Gaussian noise and strong adversarial attacks. Proposed transformer models exhibit a substantial augmentation in the predictive uncertainty, also known as predictive variance, when subjected to elevated amounts of noise or more potent adversarial attacks. The model can utilize this behavior to evaluate its performance and notify the user of any degradation in performance caused by noise or attacks. This self-assessment method is particularly valuable when precise and reliable predictions are essential, especially in mission-critical domain applications. These models can promptly identify any decline in performance caused by excessive noise or hostile attacks by consistently monitoring the associated uncertainty with the label prediction task. This feature allows them to notify users of potential dangers or compromised data. Bayesian transformer models are suitable for applications with common disruptive factors (prevalent noise or adversarial attacks). Consequently, these models can earn the user’s trust and confidence by making well-informed output predictions regardless of noise and other factors.

REFERENCES

[1] *Chapter eight - deep learning in biomedical image analysis*, in Biomedical Information Technology (Second Edition), D. D. Feng, ed., Biomedical Engineering, Academic Press, second edition ed., 2020, pp. 239–263.

[2] M. ABADI, , ET AL., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[3] S. A. AHMED, S. DEY, AND K. K. SARMA, *Image texture classification using artificial neural network (ann)*, in 2011 2nd National Conference on Emerging Trends and Applications in Computer Science, IEEE, 2011, pp. 1–4.

[4] M. ALORAINI, A. KHAN, S. ALADHADH, S. HABIB, M. F. ALSHAREKH, AND M. ISLAM, *Combining the transformer and convolution for effective brain tumor classification using mri images*, Applied Sciences, 13 (2023), p. 3680.

[5] G. AMATO AND F. FALCHI, *knn based image classification relying on local feature similarity*, in Proceedings of the Third International Conference on Similarity Search and Applications, 2010, pp. 101–108.

[6] A. ANDO, S. GIDARIS, A. BURSUC, G. PUY, A. BOULCH, AND R. MARLET, *Rangevit: Towards vision transformers for 3d semantic segmentation in autonomous driving*, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 5240–5250.

[7] Y. BAZI, L. BASHMAL, M. M. A. RAHHAL, R. A. DAYIL, AND N. A. AJLAN, *Vision transformers for remote sensing image classification*, Remote Sensing, 13 (2021), p. 516.

[8] C. BLUNDELL, J. CORNEBISE, K. KAVUKCUOGLU, AND D. WIERSTRA, *Weight uncertainty in neural network*, in International conference on machine learning, PMLR, 2015, pp. 1613–1622.

[9] N. BORAH, P. S. P. VARMA, A. DATTA, A. KUMAR, U. BARUAH, AND P. GHOSAL, *Performance analysis of breast cancer classification from mammogram images using vision transformer*, in 2022 IEEE Calcutta Conference (CALCON), IEEE, 2022, pp. 238–243.

[10] A. BOSCH, A. ZISSERMAN, AND X. MUNOZ, *Image classification using random forests and ferns*, in 2007 IEEE 11th International Conference on Computer Vision, 2007, pp. 1–8.

[11] A. M. BRAŞOVEANU AND R. ANDONIE, *Visualizing transformers for nlp: a brief survey*, in 2020 24th International Conference Information Visualisation (IV), IEEE, 2020, pp. 270–279.

[12] L. BREIMAN, *Random forests*, Machine learning, 45 (2001), pp. 5–32.

[13] C. J. BURGES, *A tutorial on support vector machines for pattern recognition*, Data mining and knowledge discovery, 2 (1998), pp. 121–167.

[14] N. CARION, F. MASSA, G. SYNNAEVE, N. USUNIER, A. KIRILLOV, AND S. ZAGORUYKO, *End-to-end object detection with transformers*, in European conference on computer vision, Springer, 2020, pp. 213–229.

[15] S. CHAKRABORTY AND K. MALI, *An overview of biomedical image analysis from the deep learning perspective*, Research Anthology on Improving Medical Imaging Techniques for Analysis and Intervention, (2023), pp. 43–59.

[16] M. A. CHANDRA AND S. BEDI, *Survey on svm and their application in image classification*, International Journal of Information Technology, 13 (2021), pp. 1–11.

[17] C.-F. R. CHEN, Q. FAN, AND R. PANDA, *Crossvit: Cross-attention multi-scale vision transformer for image classification*, in Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 357–366.

[18] L. CHEN, H. CHOU, Y. XIA, AND H. MIYAKE, *Multimodal item categorization fully based on transformer*, in Proceedings of The 4th Workshop on e-Commerce and NLP, 2021, pp. 111–115.

[19] L. CHEN, S. LI, Q. BAI, J. YANG, S. JIANG, AND Y. MIAO, *Review of image classification algorithms based on convolutional neural networks*, Remote Sensing, 13 (2021), p. 4712.

[20] T. CHEN, E. FOX, AND C. GUESTRIN, *Stochastic gradient hamiltonian monte carlo*, in International conference on machine learning, PMLR, 2014, pp. 1683–1691.

[21] Y. DAI, Y. GAO, AND F. LIU, *Transmed: Transformers advance multi-modal medical image classification*, Diagnostics, 11 (2021), p. 1384.

[22] N. DALAL AND B. TRIGGS, *Histograms of oriented gradients for human detection*, in 2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05), vol. 1, Ieee, 2005, pp. 886–893.

[23] L. DENG, *The mnist database of handwritten digit images for machine learning research [best of the web]*, IEEE signal processing magazine, 29 (2012), pp. 141–142.

[24] D. DERA, N. C. BOUAYNAYA, G. RASOOL, R. SHTERENBERG, AND H. M. FATHALLAH-SHAYKH, *Premium-cnn: Propagating uncertainty towards robust convolutional neural networks*, IEEE Transactions on Signal Processing, 69 (2021), pp. 4669–4684.

[25] J. DONG, S. CHEN, S. ZONG, T. CHEN, AND S. LABI, *Image transformer for explainable autonomous driving system*, in 2021 IEEE International Intelligent Transportation Systems Conference (ITSC), 2021, pp. 2732–2737.

[26] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al., *An image is worth 16x16 words: Transformers for image recognition at scale*, arXiv preprint arXiv:2010.11929, (2020).

[27] S. A. Fahad and A. E. Yahya, *Inflectional review of deep learning on natural language processing*, in 2018 International Conference on Smart Computing and Electronic Enterprise (ICSCEE), 2018, pp. 1–4.

[28] Y. Gal and Z. Ghahramani, *Bayesian convolutional neural networks with bernoulli approximate variational inference*, arXiv preprint arXiv:1506.02158, (2015).

[29] H. Gao, L. Dou, W. Chen, and J. Sun, *Image classification with bag-of-words model based on improved sift algorithm*, in 2013 9th Asian Control Conference (ASCC), 2013, pp. 1–6.

[30] B. Gheflati and H. Rivaz, *Vision transformers for classification of breast ultrasound images*, in 2022 44th Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC), IEEE, 2022, pp. 480–483.

[31] S. Ghosh, F. Delle Fave, and J. Yedidia, *Assumed density filtering methods for learning bayesian neural networks*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30, 2016.

[32] H. S. Gill and B. S. Khehra, *An integrated approach using cnn-rnn-lstm for classification of fruit images*, Materials Today: Proceedings, 51 (2022), pp. 591–595. CMAE'21.

[33] A. Gillioz, J. Casas, E. Mugellini, and O. Abou Khaled, *Overview of the transformer-based models for nlp tasks*, in 2020 15th Conference on Computer Science and Information Systems (FedCSIS), IEEE, 2020, pp. 179–183.

[34] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, in Proceedings of 3rd International Conference on Learning Representations, (ICLR), 2015.

[35] A. Graves, *Practical variational inference for neural networks*, Advances in neural information processing systems, 24 (2011).

[36] T. Guo, J. Dong, H. Li, and Y. Gao, *Simple convolutional neural network on image classification*, in 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), 2017, pp. 721–724.

[37] ——, *Simple convolutional neural network on image classification*, in 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), IEEE, 2017, pp. 721–724.

[38] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, Z. Yang, Y. Zhang, and D. Tao, *A survey on vision transformer*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 45 (2023), pp. 87–110.

[39] A. Hassani, S. Walton, N. Shah, A. Abuduweili, J. Li, and H. Shi, *Escaping the big data paradigm with compact transformers*, arXiv preprint arXiv:2104.05704, (2021).

[40] A. Hatamizadeh, Y. Tang, V. Nath, D. Yang, A. Myronenko, B. Landman, H. R. Roth, and D. Xu, *Unetr: Transformers for 3d medical image segmentation*, in Proceedings of the IEEE/CVF winter conference on applications of computer vision, 2022, pp. 574–584.

[41] J. M. Hernández-Lobato and R. Adams, *Probabilistic backpropagation for scalable learning of bayesian neural networks*, in International conference on machine learning, PMLR, 2015, pp. 1861–1869.

[42] N. Hütten, R. Meyes, and T. Meisen, *Vision transformer in industrial visual inspection*, Applied Sciences, 12 (2022), p. 11981.

[43] N. Jmour, S. Zayen, and A. Abdelkrim, *Convolutional neural networks for image classification*, in 2018 international conference on advanced systems and electric technologies (IC_ASET), IEEE, 2018, pp. 397–402.

[44] Z. Kang, J. Xue, C. S. Lai, Y. Wang, H. Yuan, and F. Xu, *Vision transformer-based photovoltaic prediction model*, Energies, 16 (2023), p. 4737.

[45] M. Kaselimi, A. Voulodimos, I. Daskalopoulos, N. Doulamis, and A. Doulamis, *A vision transformer model for convolution-free multilabel classification of satellite imagery in deforestation monitoring*, IEEE Transactions on Neural Networks and Learning Systems, (2022).

[46] J. N. Kather, C.-A. Weis, F. Bianconi, S. M. Melchers, L. R. Schad, T. Gaiser, A. Marx, and F. G. Z"ollner, *Multi-class texture analysis in colorectal cancer histology*, Scientific reports, 6 (2016), p. 27988.

[47] B. Kayalibay, G. Jensen, and P. van der Smagt, *Cnn-based segmentation of medical imaging data*, arXiv preprint arXiv:1701.03056, (2017).

[48] A. Ke, W. Ellsworth, O. Banerjee, A. Y. Ng, and P. Rajpurkar, *Chextransfer: performance and parameter efficiency of imagenet models for chest x-ray interpretation*, in Proceedings of the conference on health, inference, and learning, 2021, pp. 116–124.

[49] A. I. Khan and S. Al-Habsi, *Machine learning in computer vision*, Procedia Computer Science, 167 (2020), pp. 1444–1451. International Conference on Computational Intelligence and Data Science.

[50] S. Khan, H. Rahmani, S. A. A. Shah, M. Bennamoun, G. Medioni, and S. Dickinson, *A guide to convolutional neural networks for computer vision*, vol. 8, Springer, 2018.

[51] S. M. H. Khan, A. Hussain, and I. F. T. Alshaikhli, *Comparative study on content-based image retrieval (cbir)*, in 2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT), IEEE, 2012, pp. 61–66.

[52] D. P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, in Proceedings of 3th International Conference on Learning Representations, (ICLR), 2015.

[53] J. D. KOTHARI, *A case study of image classification based on deep learning using tensorflow*, Jubin Dipakkumar Kothari (2018). A Case Study of Image Classification Based on Deep Learning Using Tensorflow. International Journal of Innovative Research in Computer and Communication Engineering, 6 (2018), pp. 3888–3892.

[54] A. KRIZHEVSKY AND G. HINTON, *Convolutional deep belief networks on cifar-10*, Unpublished manuscript, 40 (2010), pp. 1–9.

[55] A. KRIZHEVSKY, I. SUTSKEVER, AND G. E. HINTON, *Imagenet classification with deep convolutional neural networks*, Advances in neural information processing systems, 25 (2012).

[56] S. LAZEBNIK, C. SCHMID, AND J. PONCE, *Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories*, in 2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06), vol. 2, IEEE, 2006, pp. 2169–2178.

[57] Y. LECUN, L. BOTTOU, Y. BENGIO, AND P. HAFFNER, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86 (1998), pp. 2278–2324.

[58] S. H. LEE, S. LEE, AND B. C. SONG, *Vision transformer for small-size datasets*, arXiv preprint arXiv:2112.13492, (2021).

[59] X. LI, Z. YANG, Q. WANG, Y. SUN, AND A. LIU, *Vision transformer for cell tumor image classification*, in 2023 3rd International Conference on Frontiers of Electronics, Information and Computation Technologies (ICFEICT), 2023, pp. 176–180.

[60] Y. LI, J. M. HERNÁNDEZ-LOBATO, AND R. E. TURNER, *Stochastic expectation propagation*, Advances in neural information processing systems, 28 (2015).

[61] Y. LIU, E. SANGINETO, W. BI, N. SEBE, B. LEPRI, AND M. NADAI, *Efficient training of visual transformers with small datasets*, Advances in Neural Information Processing Systems, 34 (2021), pp. 23818–23830.

[62] Z. LIU, Y. LIN, Y. CAO, H. HU, Y. WEI, Z. ZHANG, S. LIN, AND B. GUO, *Swin transformer: Hierarchical vision transformer using shifted windows*, in 2021 IEEE/CVF International Conference on Computer Vision (ICCV), 2021, pp. 9992–10002.

[63] D. G. LOWE, *Object recognition from local scale-invariant features*, in Proceedings of the seventh IEEE international conference on computer vision, vol. 2, Ieee, 1999, pp. 1150–1157.

[64] ——, *Distinctive image features from scale-invariant keypoints*, International journal of computer vision, 60 (2004), pp. 91–110.

[65] D. LU AND Q. WENG, *A survey of image classification methods and techniques for improving classification performance*, International Journal of Remote Sensing, 28 (2007), pp. 823–870.

[66] D. J. MacKay, *A practical bayesian framework for backpropagation networks*, Neural computation, 4 (1992), pp. 448–472.

[67] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, *Towards deep learning models resistant to adversarial attacks*, arXiv preprint arXiv:1706.06083, (2017).

[68] S. McCann and D. G. Lowe, *Local naive bayes nearest neighbor for image classification*, in 2012 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2012, pp. 3650–3656.

[69] B. H. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer, K. Farahani, J. Kirby, Y. Burren, N. Porz, J. Slotboom, R. Wiest, et al., *The multimodal brain tumor image segmentation benchmark (brats)*, IEEE transactions on medical imaging, 34 (2014), pp. 1993–2024.

[70] S. Minaee, E. Azimi, and A. Abdolrashidi, *Fingernet: Pushing the limits of fingerprint recognition using convolutional neural network*, arXiv preprint arXiv:1907.12956, (2019).

[71] J. N. Mogan, C. P. Lee, K. M. Lim, and K. S. Muthu, *Gait-vit: Gait recognition with vision transformer*, Sensors, 22 (2022), p. 7362.

[72] S. Müller, N. Hollmann, S. P. Arango, J. Grabocka, and F. Hutter, *Transformers can do bayesian inference*, arXiv preprint arXiv:2112.10510, (2021).

[73] R. M. Neal, *Bayesian learning for neural networks*, vol. 118, Springer Science & Business Media, 2012.

[74] N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh, *Deep learning vs. traditional computer vision*, in Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1 1, Springer, 2020, pp. 128–144.

[75] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, *The limitations of deep learning in adversarial settings*, in 2016 IEEE European symposium on security and privacy (EuroS&P), IEEE, 2016, pp. 372–387.

[76] R. Patel and S. Patel, *A comprehensive study of applying convolutional neural network for computer vision*, International Journal of Advanced Science and Technology, 6 (2020), pp. 2161–2174.

[77] J. Qin, W. Pan, X. Xiang, Y. Tan, and G. Hou, *A biological image classification method based on improved cnn*, Ecological Informatics, 58 (2020), p. 101093.

[78] J. Qiu, J. Liu, and Y. Shen, *Computer vision technology based on deep learning*, in 2021 IEEE 2nd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA), vol. 2, 2021, pp. 1126–1130.

[79] M. M. U. Rahman, M. H. Robin, and A. M. Taief, *A new framework for video-based frequent iris movement analysis towards anomaly observer detection*, International Journal of Image, Graphics and Signal Processing (IJIGSP), 13 (2021), pp. 13–27.

[80] W. Rawat and Z. Wang, *Deep convolutional neural networks for image classification: A comprehensive review*, Neural computation, 29 (2017), pp. 2352–2449.

[81] H. Ritter, A. Botev, and D. Barber, *A scalable laplace approximation for neural networks*, in 6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings, vol. 6, International Conference on Representation Learning, 2018.

[82] M. H. Robin, M. M. U. Rahman, A. M. Taief, and Q. N. Eity, *Improvement of face and eye detection performance by using multi-task cascaded convolutional networks*, in 2020 IEEE Region 10 Symposium (TENSYMP), IEEE, 2020, pp. 977–980.

[83] K. A. Sankararaman, S. Wang, and H. Fang, *Bayesformer: Transformer with uncertainty estimation*, arXiv preprint arXiv:2206.00826, (2022).

[84] N. Sebe, *Machine learning in computer vision*, vol. 29, Springer Science & Business Media, 2005.

[85] R. Shao and X.-J. Bi, *Transformers meet small datasets*, IEEE Access, 10 (2022), pp. 118454–118464.

[86] A. Sharma, X. Liu, X. Yang, and D. Shi, *A patch-based convolutional neural network for remote sensing image classification*, Neural Networks, 95 (2017), pp. 19–28.

[87] N. Sharma, V. Jain, and A. Mishra, *An analysis of convolutional neural networks for image classification*, Procedia Computer Science, 132 (2018), pp. 377–384. International Conference on Computational Intelligence and Data Science.

[88] B. Shepherd, *An appraisal of a decision tree approach to image classification.*, in IJCAI, Citeseer, 1983, pp. 473–475.

[89] M. Sheykhmousa, M. Mahdianpari, H. Ghanbari, F. Mohammadimanesh, P. Ghamisi, and S. Homayouni, *Support vector machine versus random forest for remote sensing image classification: A meta-analysis and systematic review*, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 13 (2020), pp. 6308–6325.

[90] A. Singh, *Training strategies for vision transformers for object detection*, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, June 2023, pp. 110–118.

[91] R. K. Sinha, R. Pandey, and R. Pattnaik, *Deep learning for computer vision tasks: A review*, 2018.

[92] B. Song, S. Sunny, S. Li, K. Gurushanth, P. Mendonca, N. Mukhia, S. Patrick, S. Gurudath, S. Raghavan, I. Tsusennaro, S. T. Leivon, T. Kolur, V. Shetty, V. R. Bushan, R. Ramesh, T. Peterson, V. Pillai, P. Wilder-Smith, A. Sigamani, A. Suresh, moni Abraham Kuriakose, P. Birur, and R. Liang, *Bayesian deep learning for reliable oral cancer image classification*, Biomed. Opt. Express, 12 (2021), pp. 6422–6430.

[93] M. Sorić, D. Pongrac, and I. Inza, *Using convolutional neural network for chest x-ray image classification*, in 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), 2020, pp. 1771–1776.

[94] F. Sultana, A. Sufian, and P. Dutta, *Advancements in image classification using convolutional neural network*, in 2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN), IEEE, 2018, pp. 122–129.

[95] L. Tanzi, A. Audisio, G. Cirrincione, A. Aprato, and E. Vezzetti, *Vision transformer for femur fracture classification*, Injury, 53 (2022), pp. 2625–2634.

[96] I. A. Tarmizi and A. A. Aziz, *Vehicle detection using convolutional neural network for autonomous vehicles*, in 2018 International Conference on Intelligent and Advanced System (ICIAS), 2018, pp. 1–5.

[97] M. Tripathi, *Analysis of convolutional neural network based image classification techniques*, Journal of Innovative Image Processing (JIIP), 3 (2021), pp. 100–117.

[98] S. Tummala, S. Kadry, S. A. C. Bukhari, and H. T. Rauf, *Classification of brain tumor from magnetic resonance imaging using vision transformers ensembling*, Current Oncology, 29 (2022), pp. 7498–7511.

[99] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is all you need*, Advances in neural information processing systems, 30 (2017).

[100] A. Voulodimos, N. Doulamis, G. Bebis, and T. Stathaki, *Recent developments in deep learning for engineering applications*, Computational intelligence and neuroscience, 2018 (2018).

[101] B. Wang, J. Lu, Z. Yan, H. Luo, T. Li, Y. Zheng, and G. Zhang, *Deep uncertainty quantification: A machine learning approach for weather forecasting*, in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 2087–2095.

[102] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, *Cnn-rnn: A unified framework for multi-label image classification*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016.

[103] M. Welling and Y. W. Teh, *Bayesian learning via stochastic gradient langevin dynamics*, in Proceedings of the 28th international conference on machine learning (ICML-11), 2011, pp. 681–688.

[104] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms*, arXiv preprint arXiv:1708.07747, (2017).

[105] L. Xie, Q. Tian, J. Wang, and B. Zhang, *Image classification with max-sift descriptors*, in International conference on acoustics, speech and signal processing, 2015.

[106] M. Xu, H. Zhang, and J. Yang, *Prohibited item detection in airport x-ray security images via attention mechanism based cnn*, in Pattern Recognition and Computer Vision: First Chinese Conference, PRCV 2018, Guangzhou, China, November 23-26, 2018, Proceedings, Part II 1, Springer, 2018, pp. 429–439.

[107] S. S. Yadav and S. M. Jadhav, *Deep convolutional neural network based medical image classification for disease diagnosis*, Journal of Big data, 6 (2019), pp. 1–18.

[108] C.-C. Yang, S. O. Prasher, P. Enright, C. Madramootoo, M. Burgess, P. K. Goel, and I. Callum, *Application of decision tree technology for image classification using remote sensing data*, Agricultural Systems, 76 (2003), pp. 1101–1117.

[109] L. Yuan, Y. Chen, T. Wang, W. Yu, Y. Shi, Z.-H. Jiang, F. E. Tay, J. Feng, and S. Yan, *Tokens-to-token vit: Training vision transformers from scratch on imagenet*, in Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 558–567.

[110] H. T. Zaw, N. Maneerat, and K. Y. Win, *Brain tumor detection based on naïve bayes classification*, in 2019 5th International Conference on engineering, applied sciences and technology (ICEAST), IEEE, 2019, pp. 1–4.

[111] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into deep learning*, arXiv preprint arXiv:2106.11342, (2021).

[112] H. Zhang, A. Berg, M. Maire, and J. Malik, *Svm-knn: Discriminative nearest neighbor classification for visual category recognition*, in 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), vol. 2, 2006, pp. 2126–2136.

[113] H. Zhao, J. Jia, and V. Koltun, *Exploring self-attention for image recognition*, in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2020, pp. 10076–10085.

[114] H. Zhao, Q. Wang, Z. Jia, Y. Chen, and J. Zhang, *Bayesian based facial expression recognition transformer model in uncertainty*, in 2021 International Conference on Digital Society and Intelligent Systems (DSInS), IEEE, 2021, pp. 157–161.

[115] W. Zhiqiang and L. Jun, *A review of object detection based on convolutional neural network*, in 2017 36th Chinese control conference (CCC), IEEE, 2017, pp. 11104–11109.

APPENDIX A

APPENDIX A

1.1 Source Code of Bayes-ViT

```
# -*- coding: utf-8 -*-
import tensorflow as tf
from tensorflow import keras
import os
from tensorflow.keras.layers import (Dense, Dropout, LayerNormalization, )
# os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
# For multiple devices (GPUs: 4, 5, 6, 7)
# os.environ["CUDA_VISIBLE_DEVICES"] = "1,4,5,6,7"
# import imageio
import matplotlib
import shutil

matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import math
import time, sys
import pickle
import timeit
import xlsxwriter

from scipy.interpolate import make_interp_spline, BSpline
from tensorflow.keras.layers.experimental.preprocessing import Rescaling
from tensorflow.keras import layers
# import tensorflow_addons as tfa
from keras.optimizers import SGD
import keras.backend as K
import pandas as pd
import wandb
from keras.preprocessing.image import ImageDataGenerator

os.environ["WANDB_API_KEY"] = "3df171be2d23f8aaf89ddc494bb7116af7a1ec9b"

import numpy as np
# !pip install tensorflow_addons
import tensorflow as tf
from tensorflow import keras
import math
from tensorflow.keras import layers
```

```
# import tensorflow_addons as tfa
from keras.optimizers import Adam
import keras.backend as K
import pandas as pd

plt.ioff()

mnist = tf.keras.datasets.mnist


# update_progress() : Displays or updates a console progress bar
## Accepts a float between 0 and 1. Any int will be converted to a float.
## A value under 0 represents a 'halt'.
## A value at 1 or bigger represents 100%
def update_progress(progress):
    barLength = 10  # Modify this to change the length of the progress bar
    status = ""
    if isinstance(progress, int):
        progress = float(progress)
    if not isinstance(progress, float):
        progress = 0
        status = "error: progress var must be float\r\n"
    if progress < 0:
        progress = 0
        status = "Halt...\r\n"
    if progress >= 1:
        progress = 1
        status = "Done...\r\n"
    block = int(round(barLength * progress))
    text = "\rPercent: [{0}] {1}% {2}".format("#" * block + "-" * (barLength - block), progress * 100, status)
    sys.stdout.write(text)
    sys.stdout.flush()


# Auxilary Functions


def x_Sigma_w_x_T(x, W_Sigma):
    batch_sz = x.shape[0]
    xx_t = tf.reduce_sum(tf.multiply(x, x), axis=-1,
                         keepdims=True)  # [50, 17, 64]  -> [50, 17, 1] or [50, 64] - > [50, 1]
    # xx_t_e = tf.expand_dims(xx_t,axis=2)
    return tf.multiply(xx_t, W_Sigma)  # [50,17,64] or [50, 64] or [50, 10]


def w_t_Sigma_i_w(w_mu, in_Sigma):  # [64, 64]  , [50, 17, 64] or [64, 10], [50, 64]
    Sigma_1 = tf.matmul(in_Sigma, tf.multiply(w_mu, w_mu))  # [50, 17, 64] or [50, 10]
    return Sigma_1


def tr_Sigma_w_Sigma_in(in_Sigma, W_Sigma):
    Sigma = tf.reduce_sum(in_Sigma, axis=-1, keepdims=True)  # [50,17, 1]
    return tf.multiply(Sigma, W_Sigma)  # [50,17, 64]


def sigma_regularizer(x):
    input_size = 1.0
    f_s = tf.math.softplus(x)  # tf.math.log(1. + tf.math.exp(x))
    return input_size * tf.reduce_mean(-1. - tf.math.log(f_s) + f_s)


# Bayesian Fully Connected Layers


class LinearFirst(keras.layers.Layer):
    """y = w.x + b"""

    def __init__(self, units):
        super(LinearFirst, self).__init__()
        self.units = units

    def build(self, input_shape):
        tau = 0.001  # 1. /input_shape[-1]
        ini_sigma = -6.5
        self.w_mu = self.add_weight(name='w_mu',
                                    shape=(input_shape[-1], self.units),
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.05, seed=None),
                                    regularizer=tf.keras.regularizers.l2(tau),
                                    trainable=True)
        self.w_sigma = self.add_weight(name='w_sigma',
                                       shape=(self.units,),
                                       initializer=tf.constant_initializer(ini_sigma),
                                       # initializer=tf.random_uniform_initializer(minval= -12., maxval=-2.2, seed=None),
                                       regularizer=sigma_regularizer,
                                       trainable=True)

    def call(self, inputs):  # [50,17,64]
        # Mean
        # print(self.w_mu.shape)
        mu_out = tf.matmul(inputs, self.w_mu)  # + self.b_mu       [50, 17, 64]            # Mean of the output
        # Varinace
        W_Sigma = tf.math.log(
            1. + tf.math.exp(self.w_sigma))  # [64]                        # Construct W_Sigma from w_sigmas
        Sigma_out = x_Sigma_w_x_T(inputs,
                                  W_Sigma)  # [50, 17, 64]            + tf.math.log(1. + tf.math.exp(self.b_sigma)) #tf.linalg.diag(self.b_sigma)
        Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.abs(Sigma_out)
        return mu_out, Sigma_out


class LinearNotFirst(keras.layers.Layer):
    """y = w.x + b"""

    def __init__(self, units):
        super(LinearNotFirst, self).__init__()
        self.units = units

    def build(self, input_shape):
        ini_sigma = -6.5
        # min_sigma = -4.5
        tau = 0.001  # 1. /input_shape[-1]
        self.w_mu = self.add_weight(name='w_mu', shape=(input_shape[-1], self.units),
                                    # [64 , 64] or or [64, 10] or [10, 10]
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.05, seed=None),
                                    regularizer=tf.keras.regularizers.l2(tau),
                                    # tau/self.units), #tf.keras.regularizers.l2(0.5*0.001),
                                    trainable=True, )
        self.w_sigma = self.add_weight(name='w_sigma',
                                       shape=(self.units,),
                                       initializer=tf.constant_initializer(ini_sigma),
                                       # tf.random_uniform_initializer(minval= min_sigma, maxval=ini_sigma, seed=None) ,
                                       regularizer=sigma_regularizer, # tf.constant_initializer(ini_sigma)
                                       trainable=True, )

    def call(self, mu_in, Sigma_in): # [50,17,64], [50,17,64]  or [50, 64] or [50, 10]
        mu_out = tf.matmul(mu_in, self.w_mu) # + self.b_mu  [50, 17, 64]

        W_Sigma = tf.math.log(1. + tf.math.exp(self.w_sigma)) # [64]
        Sigma_1 = w_t_Sigma_i_w(self.w_mu, Sigma_in) # [50,17,64]
        Sigma_2 = x_Sigma_w_x_T(mu_in, W_Sigma) # [50, 17, 64]
        Sigma_3 = tr_Sigma_w_Sigma_in(Sigma_in, W_Sigma) # [50, 17, 64]
        Sigma_out = Sigma_1 + Sigma_2 + Sigma_3 # + tf.linalg.diag(tf.math.log(1. + tf.math.exp(self.b_sigma))) #[50, 17, 64]

        Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out) # [50,2,17,64,64]
        Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out) # [50,2,17,64,64]
        Sigma_out = tf.abs(Sigma_out)
        return mu_out, Sigma_out # mu_out=[50,17,64], Sigma_out = [50,17,64]


# Bayesian Activation Functions

class VDP_GeLU(keras.layers.Layer):
    def __init__(self):
        super(VDP_GeLU, self).__init__()

    def call(self, mu_in, Sigma_in): # mu_in = [50,17,64], Sigma_in= [50,17,64]
        mu_out = tf.nn.gelu(mu_in) # [50,17,64]
        with tf.GradientTape() as g:
            g.watch(mu_in)
            out = tf.nn.gelu(mu_in)
        gradi = g.gradient(out, mu_in) # [50,17,64]
        Sigma_out = activation_Sigma(gradi, Sigma_in)
        return mu_out, Sigma_out # [50,2,17,64], [50,2,17,64,64]


def activation_Sigma(gradi, Sigma_in):
    grad1 = tf.multiply(gradi, gradi) # [50,17,64] or [50, 10]
    return tf.multiply(Sigma_in, grad1) # [50,17,64] or [50, 10]


class VDP_ReLU(keras.layers.Layer):
    """ReLU"""

    def __init__(self):
        super(VDP_ReLU, self).__init__()
    def call(self, mu_in, Sigma_in):
        mu_out = tf.nn.relu(mu_in)
        with tf.GradientTape() as g:
            g.watch(mu_in)
            out = tf.nn.relu(mu_in)
        gradi = g.gradient(out, mu_in)
        Sigma_out = activation_Sigma(gradi, Sigma_in)
        return mu_out, Sigma_out


# Bayesian Dropout

class VDP_Dropout(keras.layers.Layer):
    def __init__(self, drop_prop):
        super(VDP_Dropout, self).__init__()
        self.drop_prop = drop_prop

    def call(self, mu_in, Sigma_in, Training=True):
        # shape=[batch_size, seq length, embedding_dim]
        scale_sigma = 1.0 / (1 - self.drop_prop)
        if Training:
            mu_out = tf.nn.dropout(mu_in, rate=self.drop_prop)  # [50,17,64] or [50, 10]
            # print('shape in dropout',mu_out.shape)
            non_zero = tf.not_equal(mu_out, tf.zeros_like(mu_out))  # [50,17,64]
            non_zero_sigma_mask = tf.boolean_mask(Sigma_in, non_zero)
            idx_sigma = tf.dtypes.cast(tf.where(non_zero), tf.int32)
            Sigma_out = (scale_sigma ** 2) * tf.scatter_nd(idx_sigma, non_zero_sigma_mask, tf.shape(non_zero))
            # print('sigma shape in dropout',Sigma_out.shape)
        else:
            mu_out = mu_in
            Sigma_out = Sigma_in
        return mu_out, Sigma_out  # [50,17,64], [50,17,64]


# Bayesian Multi Layer Perceptron

class VDP_MLP(tf.keras.layers.Layer):
    def __init__(self, hidden_features, out_features, dropout_rate=0.1):
        super(VDP_MLP, self).__init__()
        self.dense1 = LinearNotFirst(hidden_features)
        # self.dense1 = LinearNotFirst(mlp_dim)
        self.dense2 = LinearNotFirst(out_features)
        # self.dense2 = LinearNotFirst(embed_dim)
        self.dropout1 = VDP_Dropout(dropout_rate)
        self.gelu_1 = VDP_GeLU()

    def call(self, mu_in, sigma_in):
        mu_out, sigma_out = self.dense1(mu_in, sigma_in)
        # print('shape of x(MLP layer) :',mu_out.shape)
        mu_out, sigma_out = self.gelu_1(mu_out, sigma_out)
        # print('shape of x through GeLU :',mu_out.shape)
        mu_out, sigma_out = self.dropout1(mu_out, sigma_out)
        # print('shape of x after dropout :',mu_out.shape)
        mu_out, sigma_out = self.dense2(mu_out, sigma_out)
        # print('shape of x after 2nd dense :',mu_out.shape)
        mu_out, sigma_out = self.dropout1(mu_out, sigma_out)
        # print('shape of mu_out after MLP layer', mu_out.shape)
        return mu_out, sigma_out


# Deterministic Layernorm

class LayerNorm(tf.keras.layers.Layer):
    def __init__(self, eps=1e-6, **kwargs):
        self.eps = eps
        super(LayerNorm, self).__init__(**kwargs)

    def build(self, input_shape):
        self.gamma = self.add_weight(name='gamma', shape=input_shape[-1:],
                                     initializer=tf.keras.initializers.Ones(), trainable=True)
        self.beta = self.add_weight(name='beta', shape=input_shape[-1:],
                                    initializer=tf.keras.initializers.Zeros(), trainable=True)
        super(LayerNorm, self).build(input_shape)

    def call(self, x):
        mean = K.mean(x, axis=-1, keepdims=True)
        std = K.std(x, axis=-1, keepdims=True)
        # print( "mean of LN",mean.shape)
        # print("std of LN",std.shape)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta

    def compute_output_shape(self, input_shape):
        return input_shape


# Bayesian Layernorm
class Bayesian_LayerNorm(layers.Layer):

    def __init__(self, eps=1e-6, **kwargs):
        self.eps = eps
        super(Bayesian_LayerNorm, self).__init__(**kwargs)

    def build(self, input_shape):
        self.gamma = self.add_weight(name='gamma', shape=input_shape[-1:],
                                     initializer=tf.keras.initializers.Ones(), trainable=True)
        self.beta = self.add_weight(name='beta', shape=input_shape[-1:],
                                    initializer=tf.keras.initializers.Zeros(), trainable=True)
        super(Bayesian_LayerNorm, self).build(input_shape)

    def call(self, mu_x,
             sigma_x):  # (batch_size, sequence_length, embedding_dim),  (batch_size, sequence_length, embedding_dim)[50,17, 64],[50,17,64]
        mean = K.mean(mu_x, axis=-1, keepdims=True)  # [50,17,1]
        std = K.std(mu_x, axis=-1, keepdims=True)  # [50,17,1]
        # print('std = ' , std.shape)
        #  print('gamma = ',self.gamma)
        out_mu = self.gamma * (mu_x - mean) / (std + self.eps) + self.beta
        a = (self.gamma / (std + self.eps)) ** 2  # [50,17,64]

        out_sigma = tf.math.multiply(a, sigma_x)  # [50,17,64]
        return out_mu, out_sigma

    def compute_output_shape(self, input_shape):
        return input_shape


# Bayesian Multi Head Attention

class Bayesian_MultiHeadSelfAttention_First(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads):
        super(Bayesian_MultiHeadSelfAttention_First, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        if embed_dim % num_heads != 0:
            raise ValueError(
                f"embedding dimension = {embed_dim} should be divisible by number of heads = {num_heads}"
            )
        self.projection_dim = embed_dim // num_heads
        self.query_dense = LinearFirst(embed_dim)
        self.key_dense = LinearFirst(embed_dim)
        self.value_dense = LinearFirst(embed_dim)
        self.combine_heads = LinearNotFirst(embed_dim)

    def attention(self, mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value, input_dimension):
        mu_score = tf.matmul(mu_query, mu_key, transpose_b=True)  # [50, 2, 17, 32] x [50, 2, 32, 17] = [50, 2, 17, 17]

        # print('mu_score',mu_score.shape)

        a = tf.reduce_sum(tf.math.multiply(mu_query ** 2, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        print('a', a.shape)
        b = tf.transpose(tf.reduce_sum(tf.math.multiply(mu_key ** 2, sigma_query), axis=-1, keepdims=True),
                         perm=[0, 1, 3, 2])  # [50, 2, 1, 17 ]
        a_b = a + b  # [50, 2, 17, 17]

        c1 = tf.reduce_sum(tf.math.multiply(sigma_query, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        # print('c1',c1.shape)
        c2 = tf.transpose(c1, perm=[0, 1, 3, 2])  # [50, 2, 1, 17]
        c = c1 + c2  # [50, 2, 17, 17]
        # print('c1+c2',c.shape)
        sigma_score = a_b + c  # [50, 2, 17, 17]
        # print('sigma score',sigma_score.shape)
        dim_key = tf.cast(tf.shape(mu_key)[-1], tf.float32)
        mu_scaled_score = mu_score / tf.math.sqrt(dim_key)  # [50, 2, 17, 17]
        # print('mu scaled score',mu_scaled_score.shape)
        sigma_scaled_score = sigma_score * dim_key  # [50, 2, 17, 17]

        mu_weights = tf.nn.softmax(mu_scaled_score, axis=-1)  # [50, 2, 17, 17]
        # Sigma for softmax function
        pp1 = tf.expand_dims(mu_weights, axis=-1)  # [50, 2, 17, 17,1]
        pp2 = tf.expand_dims(mu_weights, axis=3)  # [50, 2, 17,1, 17]
        ppT = tf.matmul(pp1, pp2)  # # [50, 2, 17, 17,17]
        p_diag = tf.linalg.diag(mu_weights)  # [50, 2, 17, 17,17]
        grad = (p_diag - ppT) ** 2  # # [50, 2, 17, 17,17]
        Sigma_weights = tf.squeeze(tf.matmul(grad, tf.expand_dims(sigma_scaled_score, axis=-1)))  # [50, 2, 17, 17]
        Sigma_weights = tf.where(tf.math.is_nan(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        Sigma_weights = tf.where(tf.math.is_inf(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        # Sigma_weights = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))

        mu_output = tf.matmul(mu_weights, mu_value)  # [50,2,17,17] X [50,2,17,32]=  [50,2,17,32]
        # print('mu output',mu_output.shape)
        d = tf.matmul(mu_weights ** 2, sigma_value)  # [50,2,17,32]
        e = tf.matmul(Sigma_weights, mu_value ** 2)  # [50,2,17,32]
        f = tf.matmul(Sigma_weights, sigma_value)  # [50, 2, 17, 17]x[50,2,17,32]=  [50,2,17,32]
        output_sigma = d + e + f
        return mu_output, output_sigma  # , mu_weights, Sigma_weights

    def separate_heads(self, mu_x, sigma_x, batch_size):  # [50, 17,64], [50, 17, 64]
        mu_x = tf.reshape(mu_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 ,32]
        # print('mu_x',mu_x.shape)
        sigma_x = tf.reshape(sigma_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 32]
        mu_x = tf.transpose(mu_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        sigma_x = tf.transpose(sigma_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        return mu_x, sigma_x  # [50,2,17,32],[50,2,17,32]

    def call(self, inputs):
        batch_size = tf.shape(inputs)[0]
        mu_query, sigma_query = self.query_dense(inputs)  # [50, 17,64]  , [50, 17,64]
        mu_key, sigma_key = self.key_dense(inputs)  # [50, 17,64] , [50, 17,64]
        mu_value, sigma_value = self.value_dense(inputs)  # [50, 17,64], [50, 17,64]

        mu_query, sigma_query = self.separate_heads(mu_query, sigma_query, batch_size)
        mu_key, sigma_key = self.separate_heads(mu_key, sigma_key, batch_size)
        mu_value, sigma_value = self.separate_heads(mu_value, sigma_value, batch_size)
        # print('query2 in MHA after passing through separate heads=',mu_query.shape)

        mu_attention, sigma_attention = self.attention(mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value,
                                                       tf.shape(inputs)[1])
        mu_attention = tf.transpose(mu_attention, perm=[0, 2, 1, 3])  # [50,17,2,32]
        # print('mu attention',mu_attention.shape)
        sigma_attention = tf.transpose(sigma_attention, perm=[0, 2, 1, 3])
        # print('sigma attention',sigma_attention.shape)

        mu_concat_attention = tf.reshape(mu_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention:",mu_concat_attention.shape) #[50,17,64]
        sigma_concat_attention = tf.reshape(sigma_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention sigma:",sigma_concat_attention.shape) #[50,17,64]

        mu_output, sigma_output = self.combine_heads(mu_concat_attention, sigma_concat_attention)
        # print('shape after combine head mu', mu_output.shape)
        # print('shape after combine head sigma', sigma_output.shape)
        # sigma_output = self.combine_heads(sigma_concat_attention)
        return mu_output, sigma_output


class Bayesian_MultiHeadSelfAttention_Intermediate(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads):
        super(Bayesian_MultiHeadSelfAttention_Intermediate, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        if embed_dim % num_heads != 0:
            raise ValueError(
                f"embedding dimension = {embed_dim} should be divisible by number of heads = {num_heads}"
            )
        self.projection_dim = embed_dim // num_heads
        self.query_dense = LinearNotFirst(embed_dim)
        self.key_dense = LinearNotFirst(embed_dim)
        self.value_dense = LinearNotFirst(embed_dim)
        self.combine_heads = LinearNotFirst(embed_dim)

    def attention(self, mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value):
        mu_score = tf.matmul(mu_query, mu_key, transpose_b=True)  # [50, 2, 17, 32] x [50, 2, 32, 17] = [50, 2, 17, 17]

        # print('mu_score',mu_score.shape)

        a = tf.reduce_sum(tf.math.multiply(mu_query ** 2, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        print('a', a.shape)
        b = tf.transpose(tf.reduce_sum(tf.math.multiply(mu_key ** 2, sigma_query), axis=-1, keepdims=True),
                         perm=[0, 1, 3, 2])  # [50, 2, 1, 17 ]
        a_b = a + b  # [50, 2, 17, 17]

        c1 = tf.reduce_sum(tf.math.multiply(sigma_query, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        # print('c1',c1.shape)
        c2 = tf.transpose(c1, perm=[0, 1, 3, 2])  # [50, 2, 1, 17]
        c = c1 + c2  # [50, 2, 17, 17]
        # print('c1+c2',c.shape)
        sigma_score = a_b + c  # [50, 2, 17, 17]
        # print('sigma score',sigma_score.shape)
        dim_key = tf.cast(tf.shape(mu_key)[-1], tf.float32)
        mu_scaled_score = mu_score / tf.math.sqrt(dim_key)  # [50, 2, 17, 17]
        # print('mu scaled score',mu_scaled_score.shape)
        sigma_scaled_score = sigma_score * dim_key  # [50, 2, 17, 17]

        mu_weights = tf.nn.softmax(mu_scaled_score, axis=-1)  # [50, 2, 17, 17]
        # Sigma for softmax function
        pp1 = tf.expand_dims(mu_weights, axis=-1)  # [50, 2, 17, 17,1]
        pp2 = tf.expand_dims(mu_weights, axis=3)  # [50, 2, 17,1, 17]
        ppT = tf.matmul(pp1, pp2)  # # [50, 2, 17, 17,17]
        p_diag = tf.linalg.diag(mu_weights)  # [50, 2, 17, 17,17]
        grad = (p_diag - ppT) ** 2  # # [50, 2, 17, 17,17]
        Sigma_weights = tf.squeeze(tf.matmul(grad, tf.expand_dims(sigma_scaled_score, axis=-1)))  # [50, 2, 17, 17]
        Sigma_weights = tf.where(tf.math.is_nan(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        Sigma_weights = tf.where(tf.math.is_inf(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        # Sigma_weights = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))

        mu_output = tf.matmul(mu_weights, mu_value)  # [50,2,17,17] X [50,2,17,32]=  [50,2,17,32]
        # print('mu output',mu_output.shape)
        d = tf.matmul(mu_weights ** 2, sigma_value)  # [50,2,17,32]
        e = tf.matmul(Sigma_weights, mu_value ** 2)  # [50,2,17,32]
        f = tf.matmul(Sigma_weights, sigma_value)  # [50, 2, 17, 17]x[50,2,17,32]=  [50,2,17,32]
        output_sigma = d + e + f
        return mu_output, output_sigma  # , mu_weights, Sigma_weights

    def separate_heads(self, mu_x, sigma_x, batch_size):  # [50, 17,64], [50, 17, 64]
        mu_x = tf.reshape(mu_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 ,32]
        # print('mu_x',mu_x.shape)
        sigma_x = tf.reshape(sigma_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 32]
        mu_x = tf.transpose(mu_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        sigma_x = tf.transpose(sigma_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        return mu_x, sigma_x  # [50,2,17,32],[50,2,17,32]

    def call(self, mu_inputs, sigma_inputs):
        batch_size = tf.shape(mu_inputs)[0]
        mu_query, sigma_query = self.query_dense(mu_inputs, sigma_inputs)  # [50, 17,64]  , [50, 17,64]
        mu_key, sigma_key = self.key_dense(mu_inputs, sigma_inputs)  # [50, 17,64] , [50, 17,64]
        mu_value, sigma_value = self.value_dense(mu_inputs, sigma_inputs)  # [50, 17,64], [50, 17,64]

        mu_query, sigma_query = self.separate_heads(mu_query, sigma_query, batch_size)
        mu_key, sigma_key = self.separate_heads(mu_key, sigma_key, batch_size)
        mu_value, sigma_value = self.separate_heads(mu_value, sigma_value, batch_size)
        # print('query2 in MHA after passing through separate heads=',mu_query.shape)

        mu_attention, sigma_attention = self.attention(mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value)
        mu_attention = tf.transpose(mu_attention, perm=[0, 2, 1, 3])  # [50,17,2,32]
        # print('mu attention ',mu_attention.shape)
        sigma_attention = tf.transpose(sigma_attention, perm=[0, 2, 1, 3])
        # print('sigma attention ',sigma_attention.shape)

        mu_concat_attention = tf.reshape(mu_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention:",mu_concat_attention.shape) #[50,17,64]
        sigma_concat_attention = tf.reshape(sigma_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention sigma:",sigma_concat_attention.shape) #[50,17,64]

        mu_output, sigma_output = self.combine_heads(mu_concat_attention, sigma_concat_attention)
        # print('shape after combine head mu', mu_output.shape)
        # print('shape after combine head sigma', sigma_output.shape)
        # sigma_output = self.combine_heads(sigma_concat_attention)
        return mu_output, sigma_output


# Bayesian Transformer Block

class VDP_TransformerBlock_first(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, mlp_dim, dropout=0.1):
        super(VDP_TransformerBlock_first, self).__init__()
        self.att = Bayesian_MultiHeadSelfAttention_First(embed_dim, num_heads)  # [64,2]
        self.mlp = VDP_MLP(mlp_dim * 2, mlp_dim, dropout)  # [64*2,64,dropout]
        self.layernorm1 = LayerNorm(eps=1e-6)
        self.layernorm2 = Bayesian_LayerNorm(eps=1e-6)
        self.dropout1 = VDP_Dropout(dropout)

    def call(self, inputs, training):
        inputs_norm = self.layernorm1(inputs)  # [50,17,64]
        # print("output of first LN before MHA",inputs_norm.shape) #[50,17,64]
        mu_output, sigma_out = self.att(inputs_norm)  # [50,17,64]
        # print("output of  MHA in TB",mu_output.shape)
        mu_output, sigma_out1 = self.dropout1(mu_output, sigma_out, training=training)  # [50,17,64]
        # print("output of  MHA in TB after dropout",mu_output.shape)
        mu_out1 = mu_output + inputs  # [50,17,64]
        # print('output of of MHA before entering to MLP',mu_out1.shape)

        mu_out1_norm, sigma_out1_norm = self.layernorm2(mu_out1, sigma_out1)
        mu_mlp_output, sigma_mlp_output = self.mlp(mu_out1_norm, sigma_out1_norm)
        mu_mlp_output, sigma_mlp_output = self.dropout1(mu_mlp_output, sigma_mlp_output, training=training)
        # print('2nd LN and MLP output ',mu_mlp_output.shape)
        mu_output = mu_mlp_output + mu_out1
        with tf.GradientTape() as g:
            g.watch(mu_out1)
            out = mu_mlp_output + mu_out1
        gradi = g.gradient(out, mu_out1)
        sigma_output = tf.math.multiply(tf.math.multiply(gradi, gradi), sigma_out1)
        return mu_output, sigma_output


class VDP_TransformerBlock_Intermediate(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, mlp_dim, dropout=0.1):
        super(VDP_TransformerBlock_Intermediate, self).__init__()
        self.att = Bayesian_MultiHeadSelfAttention_Intermediate(embed_dim, num_heads)  # [64,2]
        self.mlp = VDP_MLP(mlp_dim * 2, mlp_dim, dropout)  # [64*2,64,dropout]
        self.layernorm1 = Bayesian_LayerNorm(eps=1e-6)
        self.layernorm2 = Bayesian_LayerNorm(eps=1e-6)
        self.dropout1 = VDP_Dropout(dropout)

    def call(self, mu_inputs, sigma_inputs, training):
        mu_norm, sigma_norm = self.layernorm2(mu_inputs, sigma_inputs)  # [50,17,64]
        # print("output of first LN before MHA",inputs_norm.shape) #[50,17,64]
        mu_output, sigma_out = self.att(mu_norm, sigma_norm)  # [50,17,64]
        # print("output of  MHA in TB",mu_output.shape)
        mu_output, sigma_out1 = self.dropout1(mu_output, sigma_out, training=training)  # [50,17,64]
        # print("output of  MHA in TB after dropout",mu_output.shape)
        mu_out1 = mu_output + mu_inputs  # [50,17,64]
        # print('output of of MHA before entering to MLP',mu_out1.shape)

        mu_out1_norm, sigma_out1_norm = self.layernorm2(mu_out1, sigma_out1)
        mu_mlp_output, sigma_mlp_output = self.mlp(mu_out1_norm, sigma_out1_norm)
        mu_mlp_output, sigma_mlp_output = self.dropout1(mu_mlp_output, sigma_mlp_output, training=training)
        # print('2nd LN and MLP output',mu_mlp_output.shape)
        mu_output = mu_mlp_output + mu_out1
        with tf.GradientTape() as g:
            g.watch(mu_out1)
            out = mu_mlp_output + mu_out1
        gradi = g.gradient(out, mu_out1)
        sigma_output = tf.math.multiply(tf.math.multiply(gradi, gradi), sigma_out1)
        return mu_output, sigma_output


# Bayesian Fully Connected Layer

class DDense(keras.layers.Layer):
    def __init__(self, units=32):
        '''
        Initialize the instance attributes
        '''
        super(DDense, self).__init__()
        self.units = units

    def build(self, input_shape):
        '''
        Create the state of the layer (weights)
        '''
        w_init = tf.random_normal_initializer()
        self.w = tf.Variable(name='kernel',
                             initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
                             trainable=True)
        # initialize bias
        b_init = tf.zeros_initializer()
        self.b = tf.Variable(name='bias',
                             initial_value=b_init(shape=(self.units,), dtype='float32'),
                             trainable=True)

    def call(self, inputs):
        '''
        Defines the computation from inputs to outputs
        '''
        return tf.matmul(inputs, self.w) + self.b


# Bayesian Vision Transformer


class VDP_ViT(tf.keras.Model):
    def __init__(
            self,
            image_size,
            patch_size,
            num_layers,
            num_classes,
            embed_dim,
            num_heads,
            mlp_dim,
            channels=1,
            dropout=0.1,
            name=None
    ):
        super(VDP_ViT, self).__init__()
        num_patches = (image_size // patch_size) ** 2
        self.patch_dim = channels * (patch_size ** 2)

        self.patch_size = patch_size
        self.embed_dim = embed_dim
        self.num_layers = num_layers
        self.mlp_dim = mlp_dim
        self.rescale = Rescaling(1.0 / 255)
        self.pos_emb = self.add_weight(
            "pos_emb", shape=(1, num_patches + 1, embed_dim)
        )
        self.class_emb = self.add_weight("class_emb", shape=(1, 1, embed_dim))
        self.patch_proj = DDense(embed_dim)
        # self.enc_layers = VDP_TransformerBlock_first(d_model, num_heads, mlp_dim, dropout)  # for _ in range(num_layers)]
        # self.enc_layers =  [VDP_TransformerBlock(d_model, num_heads, mlp_dim, dropout)
        # for _ in range(num_layers) ]

        self.enc_layers1 = VDP_TransformerBlock_first(embed_dim, num_heads, mlp_dim, dropout)
        self.enc_layers = [
            VDP_TransformerBlock_Intermediate(embed_dim, num_heads, mlp_dim, dropout)
            for _ in range(num_layers)
        ]

        # self.mlp_head = VDP_MLP(mlp_dim, num_classes)
        self.mlp_head = VDP_MLP(mlp_dim, num_classes)

    def extract_patches(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID")
        patches = tf.reshape(patches, [batch_size, -1, self.patch_dim])
        return patches

    def call(self, x, training):
        print('Input dimension :', x.shape)
        batch_size = tf.shape(x)[0]
        # x = self.rescale(x)
        # print('Input dimension after rescale :', x.shape)
        patches = self.extract_patches(x)
        # print('Input dimension after extract patch :',patches.shape)
        x = self.patch_proj(patches)
        # print('Input dimension after patch projection :',x.shape)
        class_emb = tf.broadcast_to(self.class_emb, [batch_size, 1, self.embed_dim])
        x = tf.concat([class_emb, x], axis=1)
        # print('Input dimension after concat :',x.shape)
        x = x + self.pos_emb
        # print('Input dimension after x = x + self.pos_emb :',x.shape)
        # mu_out, sigma_out = self.enc_layers(x)
        # for layer in self.enc_layers:
        # x = layer(x, training)
        # for layer in self.enc_layers:

        # mu_out, sigma_out = layer(x)
        mu_out, sigma_out = self.enc_layers1(x)

        for layer in self.enc_layers:
            mu_out, sigma_out = layer(mu_out, sigma_out, training)

        # First (class token) is used for classification
        mu, sigma = self.mlp_head(mu_out[:, 0], sigma_out[:, 0])
        print('shape of mu', mu.shape)
        # print('shape of sigma',sigma.shape)
        return mu, sigma


# Loss Function(Modified)


def nll_gaussian(y_test, y_pred_mean, y_pred_sd):
    mu = y_test - y_pred_mean
    mu_2 = mu ** 2
    y_pred_sd = y_pred_sd + 1e-4
    s = tf.math.divide_no_nan(1., y_pred_sd)
    loss1 = tf.math.reduce_mean(tf.math.reduce_sum(tf.math.multiply(mu_2, s), axis=-1))
    loss2 = tf.math.reduce_mean(tf.math.reduce_sum(tf.math.log(y_pred_sd), axis=-1))
    loss = tf.math.reduce_mean(tf.math.add(loss1, loss2))
    loss = tf.where(tf.math.is_nan(loss), tf.zeros_like(loss), loss)
    loss = tf.where(tf.math.is_inf(loss), tf.zeros_like(loss), loss)
    return loss


# Main Function

# def main_function(input_dim=28, num_kernels=[32], kernels_size=[5], maxpooling_size=[2], maxpooling_stride=[2], maxpooling_pad='SAME', class_num=10
#          epochs =20, lr=0.001, lr_end = 0.0001, kl_factor = 0.01,
#          Random_noise=True, gaussain_noise_std=0.5, Adversarial_noise=False, epsilon = 0, adversary_target_cls=3, Targeted=False,
#          Training = False, continue_training = False,  saved_model_epochs=50):

def main_function(image_size=28, patch_size=4, num_layers=7, num_classes=10, embed_dim=64, num_heads=4, mlp_dim=64,
                  channels=1, drop_prob=0.1, batch_size=20, epochs=300, lr=0.001, lr_end=0.00001, kl_factor=0.0001,
                  Targeted=False, Random_noise=True, gaussain_noise_std=0.1, epsilon=0.001,Training=False, Testing=True,
                  Adversarial_noise=False, HCV=0.5, adversary_target_cls=3, PGD_Adversarial_noise=False, stepSize=1,
                  maxAdvStep=20, continue_training=False, saved_model_epochs=30):
    PATH = './VDP_cnn_epoch_{}/'.format(epochs)


    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

    '''for i in range(10):
        print(f"Label {i + 1}: {fashion_mnist_labels[y_train[i]]}") '''

    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255

    # Make sure images have shape (, 28, 1)
    x_train = np.expand_dims(x_train, -1)
    x_test = np.expand_dims(x_test, -1)
    # print(len(x_train))

    #x_train = tf.image.resize(x_train, [64, 64])  # resizing image shape to 64 X 64
    # print('shape after resizing image',x_train.shape)

    #x_test = tf.image.resize(x_test, [64, 64])

    datagen = ImageDataGenerator(rotation_range=10,  # Rotating randomly the images up to 25
                                 width_shift_range=0.2,  # Moving the images from left to right
                                 height_shift_range=0.2,  # Then from top to bottom
                                 shear_range=0.10,
                                 zoom_range=0.05,  # Zooming randomly up to 20%
                                 zca_whitening=False,
                                 horizontal_flip=False,
                                 vertical_flip=False,
                                 fill_mode='nearest')

    #datagen.fit(x_train)  # Very important to fit the Generator on the data
    print(len(x_train))

    #  (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # x_train, x_test = x_train / 255.0, x_test / 255.0
    # x_train = x_train.astype('float32')
    # x_test = x_test.astype('float32')
    # x_train = tf.expand_dims(x_train, -1)
    # x_test = tf.expand_dims(x_test, -1)
    one_hot_y_train = tf.one_hot(y_train.astype(np.float32), depth=num_classes)
    one_hot_y_test = tf.one_hot(y_test.astype(np.float32), depth=num_classes)
    tr_dataset = tf.data.Dataset.from_tensor_slices((x_train, one_hot_y_train)).batch(batch_size)
    val_dataset = tf.data.Dataset.from_tensor_slices((x_test, one_hot_y_test)).batch(batch_size)

    trans_model = VDP_ViT(image_size=image_size, patch_size=patch_size, num_layers=num_layers, num_classes=num_classes,
                          embed_dim=embed_dim, num_heads=num_heads, mlp_dim=mlp_dim,
                          channels=channels, dropout=drop_prob, name='vdp_trans')

    num_train_steps = epochs * int(x_train.shape[0] / batch_size)
    #     step = min(step, decay_steps)
    #     ((initial_learning_rate - end_learning_rate) * (1 - step / decay_steps) ^ (power) ) + end_learning_rate

    learning_rate_fn = tf.keras.optimizers.schedules.PolynomialDecay(initial_learning_rate=lr,
                                                                     decay_steps=num_train_steps,
                                                                     end_learning_rate=lr_end, power=3.)
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate_fn)  # , clipnorm=1.0)

    @tf.function  # Make it fast.
    def train_on_batch(x, y):
        with tf.GradientTape() as tape:
            mu_out, sigma = trans_model(x, training=True)
            print("shape of mu_out", mu_out.shape)
            trans_model.trainable = True
            trans_model.summary()
            loss_final = nll_gaussian(y, mu_out, tf.clip_by_value(t=sigma, clip_value_min=tf.constant(1e-2),
                                                                  clip_value_max=tf.constant(1e+8)))
            regularization_loss = tf.math.add_n(trans_model.losses)
            loss = 0.5 * (loss_final + kl_factor * regularization_loss)
        gradients = tape.gradient(loss, trans_model.trainable_weights)
        gradients = [(tf.where(tf.math.is_nan(grad), tf.constant(1.0e-5, shape=grad.shape), grad)) for grad in
                     gradients]
        gradients = [(tf.where(tf.math.is_inf(grad), tf.constant(1.0e-5, shape=grad.shape), grad)) for grad in
                     gradients]
        optimizer.apply_gradients(zip(gradients, trans_model.trainable_weights))
        return loss, mu_out, sigma, gradients, regularization_loss, loss_final

    @tf.function
    def validation_on_batch(x, y):
        mu_out, sigma = trans_model(x, training=False)
        # cnn_model.trainable = False
        vloss = nll_gaussian(y, mu_out, tf.clip_by_value(t=sigma, clip_value_min=tf.constant(1e-2),
                                                         clip_value_max=tf.constant(1e+8)))
        regularization_loss = tf.math.add_n(trans_model.losses)
        total_vloss = 0.5 * (vloss + kl_factor * regularization_loss)
        return total_vloss, mu_out, sigma

    @tf.function
    def test_on_batch(x, y):
        trans_model.trainable = False
        mu_out, sigma = trans_model(x, training=False)
        return mu_out, sigma

    @tf.function
```

```
def create_adversarial_pattern(input_image, input_label):
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        trans_model.trainable = False
        prediction, sigma = trans_model(input_image)
        loss_final = nll_gaussian(input_label, prediction,
                                  tf.clip_by_value(t=sigma, clip_value_min=tf.constant(1e-4),
                                                   clip_value_max=tf.constant(1e+3)))
        # clip_value_max=tf.constant(1e+3)), num_classes, batch_size)

        loss = 0.5 * loss_final
        # Get the gradients of the loss w.r.t to the input image.
    gradient = tape.gradient(loss, input_image)
    # Get the sign of the gradients to create the perturbation
    signed_grad = tf.sign(gradient)
    return signed_grad

wandb.init(entity="fazlur7512",
           project="VDP_Trans_mnist_epochs_{}_layer_{}_lr_{}_kl_factor_{}_batch_size_{}_dimension_{}_patch_size_{}_head_{}_input_{}".format(
               epochs, num_layers, lr, kl_factor, batch_size, embed_dim, patch_size, num_heads, image_size))

if Training:
    wandb.init(entity="fazlur7512",
               project="VDP_Trans_mnist_epochs_{}_layer_{}_lr_{}_kl_factor_{}_batch_size_{}_dimension_{}_patch_size_{}_head_{}_input_{}".format(
                   epochs, num_layers, lr, kl_factor, batch_size, embed_dim, patch_size, num_heads, image_size))
    if continue_training:
        saved_model_path = './saved_models/VDP_cnn_epoch_{}/'.format(saved_model_epochs)
        trans_model.load_weights(saved_model_path + 'vdp_cnn_model')
    train_acc = np.zeros(epochs)
    valid_acc = np.zeros(epochs)
    train_err = np.zeros(epochs)
    valid_error = np.zeros(epochs)

    start = timeit.default_timer()
    for epoch in range(epochs):
        print('Epoch: ', epoch + 1, '/', epochs)
        acc1 = 0
        acc_valid1 = 0
        err1 = 0
        err_valid1 = 0
        tr_no_steps = 0
        va_no_steps = 0
        # -------------Training--------------------
        acc_training = np.zeros(int(x_train.shape[0] / (batch_size)))
        err_training = np.zeros(int(x_train.shape[0] / (batch_size)))
        for step, (x, y) in enumerate(tr_dataset):
            update_progress(step / int(x_train.shape[0] / (batch_size)))
            #  print(y.shape)
            loss, mu_out, sigma, gradients, regularization_loss, loss_final = train_on_batch(x, y)
            #  print(mu_out.shape)
            err1 += loss.numpy()
            corr = tf.equal(tf.math.argmax(mu_out, axis=1), tf.math.argmax(y, axis=1))
            accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
            acc1 += accuracy.numpy()
            if step % 100 == 0:
                print('\n gradient', np.mean(gradients[0].numpy()))
                #   print('\n Matrix Norm', np.mean(sigma))
                print("\n Step:", step, "Loss:", float(err1 / (tr_no_steps + 1.)))
                print("Total Training accuracy so far: %.3f" % float(acc1 / (tr_no_steps + 1.)))
            tr_no_steps += 1
            wandb.log({"Average Variance value": tf.reduce_mean(sigma).numpy(),
                       "Total Training Loss": loss.numpy(),
                       "Training Accuracy per minibatch": accuracy.numpy(),
                       "gradient per minibatch": np.mean(gradients[0]),
                       'epoch': epoch,
                       "Regularization_loss": regularization_loss.numpy(),
                       "Log-Likelihood Loss": np.mean(loss_final.numpy())
                       })
        train_acc[epoch] = acc1 / tr_no_steps
        train_err[epoch] = err1 / tr_no_steps
        print('Training Acc  ', train_acc[epoch])
        print('Training error  ', train_err[epoch])
        # ---------------Validation----------------------
        for step, (x, y) in enumerate(val_dataset):
            update_progress(step / int(x_test.shape[0] / (batch_size)))
            total_vloss, mu_out, sigma = validation_on_batch(x, y)
            err_valid1 += total_vloss.numpy()
            corr = tf.equal(tf.math.argmax(mu_out, axis=-1), tf.math.argmax(y, axis=-1))
            va_accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
            acc_valid1 += va_accuracy.numpy()

            if step % 50 == 0:
                print("Step:", step, "Loss:", float(total_vloss))
                print("Total validation accuracy so far: %.3f" % va_accuracy)
            va_no_steps += 1
           # wandb.log({"Average Variance value (validation Set)": tf.reduce_mean(sigma).numpy(),
           #            "Total Validation Loss": total_vloss.numpy(),
           #            "Validation Acuracy per minibatch": va_accuracy.numpy()
           #            })
        valid_acc[epoch] = acc_valid1 / va_no_steps
        valid_error[epoch] = err_valid1 / va_no_steps
        stop = timeit.default_timer()
        trans_model.save_weights(PATH + 'vdp_transfm_model')
        wandb.log({"Average Training Loss": train_err[epoch],
                   "Average Training Accuracy": train_acc[epoch],
                   #"Average Validation Loss": valid_error[epoch],
                   #"Average Validation Accuracy": valid_acc[epoch],
                   'epoch': epoch
                   })

        # wandb.log({"Average Training Loss": train_err[epoch],
        #            "Average Training Accuracy": train_acc[epoch],
        #           'epoch': epoch
        #          })

        print('Total Training Time: ', stop - start)
        print('Training Acc  ', train_acc[epoch])
        print('Validation Acc  ', valid_acc[epoch])
        print('------------------------------------')
        print('Training error  ', train_err[epoch])
        print('Validation error  ', valid_error[epoch])
        # -----------------End Training-------------------------
```

```
trans_model.save_weights(PATH + 'vdp_cnn_model')

if (epochs > 1):
    fig = plt.figure(figsize=(15, 7))
    plt.plot(train_acc, 'b', label='Training acc')
    plt.plot(valid_acc, 'r', label='Validation acc')
    plt.ylim(0, 1.1)
    plt.title("Density Propagation Trans on Fashion MNIST Data")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend(loc='lower right')
    plt.savefig(PATH + 'VDP_Trans_on_Fashion_MNIST_Data_acc.png')
    plt.close(fig)

    fig = plt.figure(figsize=(15, 7))
    plt.plot(train_err, 'b', label='Training error')
    plt.plot(valid_error, 'r', label='Validation error')
    plt.title("Density Propagation Trans on Fashion MNIST Data")
    plt.xlabel("Epochs")
    plt.ylabel("Error")
    plt.legend(loc='upper right')
    plt.savefig(PATH + 'VDP_Trans_on_FMNIST_Data_error.png')
    plt.close(fig)

f = open(PATH + 'training_validation_acc_error.pkl', 'wb')
pickle.dump([train_acc, valid_acc, train_err, valid_error], f)
f.close()

textfile = open(PATH + 'Related_hyperparameters.txt', 'w')
textfile.write(' Input Dimension : ' + str(image_size))
textfile.write('\n Hidden units : ' + str(mlp_dim))
textfile.write('\n Number of Classes : ' + str(num_classes))
textfile.write('\n No of epochs : ' + str(epochs))
textfile.write('\n Initial Learning rate : ' + str(lr))
textfile.write('\n Ending Learning rate : ' + str(lr_end))
#  textfile.write('\n kernels Size : ' +str(kernels_size))
#  textfile.write('\n Max pooling Size : ' +str(maxpooling_size))
#  textfile.write('\n Max pooling stride : ' +str(maxpooling_stride))
textfile.write('\n batch size : ' + str(batch_size))
textfile.write('\n KL term factor : ' + str(kl_factor))
textfile.write("\n---------------------------------")
if Training:
    textfile.write('\n Total run time in sec : ' + str(stop - start))
    if (epochs == 1):
        textfile.write("\n Averaged Training  Accuracy : " + str(train_acc))
        textfile.write("\n Averaged Validation Accuracy : " + str(valid_acc))

        textfile.write("\n Averaged Training  error : " + str(train_err))
        textfile.write("\n Averaged Validation error : " + str(valid_error))
    else:
        textfile.write("\n Averaged Training  Accuracy : " + str(np.mean(train_acc[epoch])))
        textfile.write("\n Averaged Validation Accuracy : " + str(np.mean(valid_acc[epoch])))

        textfile.write("\n Averaged Training  error : " + str(np.mean(train_err[epoch])))
        textfile.write("\n Averaged Validation error : " + str(np.mean(valid_error[epoch])))
textfile.write("\n---------------------------------")
```

```
    textfile.write("\n---------------------------------")
    textfile.close()

# if (Testing):
#     test_path = 'test_results/'
# if Random_noise:
# test_path = 'test_random_noise_{}/'.format(gaussain_noise_std)
#  os.makedirs(PATH + test_path)
# trans_model.load_weights(PATH + 'vdp_cnn_model')

if Testing:
    test_path = 'test_results/'
    if Random_noise:
        test_path = 'test_results_random_noise_{}/'.format(gaussain_noise_std)
    full_test_path = PATH + test_path
    if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
        shutil.rmtree(full_test_path)
    os.makedirs(PATH + test_path)

    trans_model.load_weights(PATH + 'vdp_cnn_model')

    test_no_steps = 0
    true_x = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, 1])
    true_y = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    mu_out_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    # sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes,num_classes])
    acc_test = np.zeros(int(x_test.shape[0] / (batch_size)))
    for step, (x, y) in enumerate(val_dataset):
        update_progress(step / int(x_test.shape[0] / (batch_size)))
        true_x[test_no_steps, :, :, :, :] = x
        true_y[test_no_steps, :, :] = y
        if Random_noise:
            noise = tf.random.normal(shape=[batch_size, image_size, image_size, 1], mean=0.0,
                                     stddev=gaussain_noise_std, dtype=x.dtype)
            x = x + noise
        mu_out, sigma = test_on_batch(x, y)
        mu_out_[test_no_steps, :, :] = mu_out
        # sigma_[test_no_steps, :, :,:]= sigma
        sigma_[test_no_steps, :, :] = sigma
        corr = tf.equal(tf.math.argmax(mu_out, axis=1), tf.math.argmax(y, axis=1))
        accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
        acc_test[test_no_steps] = accuracy.numpy()
        if step % 100 == 0:
            print("Total running accuracy so far: %.3f" % acc_test[test_no_steps])
        test_no_steps += 1
        # New added line
        wandb.log({"Testing Accuracy per minibatch": accuracy.numpy()
                   })

    test_acc = np.mean(acc_test)
    print('Test accuracy : ', test_acc)
    # print("Best Test Accuracy :", np.amax(acc_test))
    # New added line
    wandb.log({"Testing Accuracy": (test_acc)})
```

```
pf = open(PATH + test_path + 'uncertainty_info.pkl', 'wb')
pickle.dump([mu_out_, sigma_, true_x, true_y, test_acc], pf)
pf.close()




var = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size])

if Random_noise:
    snr_signal = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size])
    for i in range(int(x_test.shape[0] / (batch_size))):
        for j in range(batch_size):
            noise = tf.random.normal(shape=[image_size, image_size, 1], mean=0.0, stddev=gaussain_noise_std,
                                     dtype=x.dtype)
            snr_signal[i, j] = 10 * np.log10(
                np.sum(np.square(true_x[i, j, :, :, :])) / np.sum(np.square(noise)))

        predicted_out = np.argmax(mu_out_[i, j, :])
        var[i, j] = sigma_[i, j, int(predicted_out)]
    print('SNR', np.mean(snr_signal))

#sigma_1 = np.reshape(sigma_, int(x_test.shape[0] / (batch_size)) , batch_size)
#var = np.zeros([int(test_X.shape[0] / (batch_size)) , batch_size])
#for i in range(int(test_X.shape[0] / (batch_size)), batch_size):
# for i in range(int(test_X.shape[0] / (batch_size))):
    #s = np.abs(sigma_1[i])
    #if (i != 0):
      #  if (np.abs(s) > 10000):
     #       var[i] = 0.0  # np.abs(sigma_1[i-1])
    #    else:
   #         var[i] = s
  #  else:
 #      var[i] = s
#data_mean, data_std = np.mean(np.abs(sigma_1)), np.std(np.abs(sigma_1))
# identify outliers
#cut_off = data_std * 3
#lower, upper = data_mean - cut_off, data_mean + cut_off
#outliers = [x for x in np.abs(sigma_1) if x < lower or x > upper]
#outliers_removed = [x for x in np.abs(sigma_1) if x > lower and x < upper]
#print('outliers_removed', np.mean(outliers_removed))
#writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
#df = pd.DataFrame(np.abs(sigma_1))
# Write your DataFrame to a file
#df.to_excel(writer, "Sheet")
#writer.save()
#print('Output Variance without outlier', np.mean(np.abs(var)))
#print('Output Variance', np.mean(np.abs(sigma_)))

valid_size = x_test.shape[0]
pred_var = np.zeros(int(valid_size))
true_var = np.zeros(int(valid_size))
correct_classification = np.zeros(int(valid_size))
misclassification_pred = np.zeros(int(valid_size))
misclassification_true = np.zeros(int(valid_size))
predicted_out = np.zeros(int(valid_size))
```

```
true_out = np.zeros(int(valid_size))
k = 0
k1 = 0
k2 = 0
for i in range(int(valid_size / batch_size)):
    for j in range(batch_size):
        predicted_out[k] = np.argmax(mu_out_[i, j, :])
        true_out[k] = np.argmax(true_y[i, j, :])
        pred_var[k] = sigma_[i, j, int(predicted_out[k])]
        true_var[k] = sigma_[i, j, int(true_out[k])]
        if (predicted_out[k] == true_out[k]):
            correct_classification[k1] = sigma_[i, j, int(predicted_out[k])]
            k1 = k1 + 1
        if (predicted_out[k] != true_out[k]):
            misclassification_pred[k2] = sigma_[i, j, int(predicted_out[k])]
            misclassification_true[k2] = sigma_[i, j, int(true_out[k])]
            k2 = k2 + 1
        k = k + 1
print('Average Output Variance', np.mean(pred_var))

var1 = pred_var  # np.reshape(var, int(x_test.shape[0]/(batch_size))* batch_size)
writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
df = pd.DataFrame(np.abs(var1))
# Write your DataFrame to a file
df.to_excel(writer, "Sheet")

df1 = pd.DataFrame(predicted_out)
df1.to_excel(writer, 'Sheet', startcol=4)

df2 = pd.DataFrame(true_out)
df2.to_excel(writer, 'Sheet', startcol=7)

df3 = pd.DataFrame(correct_classification)
df3.to_excel(writer, 'Sheet', startcol=10)

df4 = pd.DataFrame(misclassification_pred)
df4.to_excel(writer, 'Sheet', startcol=13)

df5 = pd.DataFrame(misclassification_true)
df5.to_excel(writer, 'Sheet', startcol=16)
writer.save()

pf = open(PATH + test_path + 'var_info.pkl', 'wb')
pickle.dump([correct_classification, misclassification_true, pred_var], pf)
#if Random_noise:

textfile = open(PATH + test_path + 'Related_hyperparameters.txt', 'w')
textfile.write(' Input Dimension : ' + str(image_size))
# textfile.write('\n No of Kernels : ' +str(num_kernels))
textfile.write('\n Number of Classes : ' + str(num_classes))
textfile.write('\n No of epochs : ' + str(epochs))
textfile.write('\n Initial Learning rate : ' + str(lr))
textfile.write('\n Ending Learning rate : ' + str(lr_end))
# textfile.write('\n kernels Size : ' +str(kernels_size))
# textfile.write('\n Max pooling Size : ' +str(maxpooling_size))
# textfile.write('\n Max pooling stride : ' +str(maxpooling_stride))
```

```
    textfile.write('\n batch size : ' + str(batch_size))
    textfile.write('\n KL term factor : ' + str(kl_factor))
    textfile.write("\n---------------------------------")
    textfile.write("\n Test Accuracy : " + str(test_acc))
    textfile.write("\n Output Variance: " + str(np.mean(np.abs(var))))
    textfile.write("\n Correct Classification Variance: " + str(np.mean(correct_classification)))
    textfile.write("\n MisClassification Variance: " + str(np.mean(misclassification_pred)))

    textfile.write("\n---------------------------------")
    if Random_noise:
        textfile.write('\n Random Noise std: ' + str(gaussain_noise_std))
        # textfile.write('\n Random Noise HCV: ' + str(HCV))
        textfile.write("\n SNR: " + str(np.mean(snr_signal)))
    textfile.write("\n---------------------------------")
    textfile.close()

    # if (Adversarial_noise):
    # elif(Adversarial_noise):

if (Adversarial_noise):
    if Targeted:
        test_path = 'test_results_targeted_adversarial_noise_{}/'.format(epsilon)
        full_test_path = PATH + test_path
        if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
          shutil.rmtree(full_test_path)
        os.makedirs(PATH + test_path)
    else:
        test_path = 'test_results_non_targeted_adversarial_noise_{}/'.format(epsilon)
        full_test_path = PATH + test_path
        if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
          shutil.rmtree(full_test_path)
        os.makedirs(PATH + test_path)
    trans_model.load_weights(PATH + 'vdp_cnn_model')
    test_no_steps = 0

    true_x = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, 1])
    adv_perturbations = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, 1])
    true_y = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    mu_out_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    #sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, class_num, class_num])
    sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])

    acc_test = np.zeros(int(x_test.shape[0] / (batch_size)))
    for step, (x, y) in enumerate(val_dataset):
        update_progress(step / int(x_test.shape[0] / (batch_size)))
        true_x[test_no_steps, :, :, :, :] = x
        true_y[test_no_steps, :, :] = y

        if Targeted:
            y_true_batch = np.zeros_like(y)
            y_true_batch[:, adversary_target_cls] = 1.0
            adv_perturbations[test_no_steps, :, :, :, :] = create_adversarial_pattern(x, y_true_batch)
        else:
            adv_perturbations[test_no_steps, :, :, :, :] = create_adversarial_pattern(x, y)
    adv_x = x + epsilon * adv_perturbations[test_no_steps, :, :, :, :]
    adv_x = tf.clip_by_value(adv_x, 0.0, 1.0)

    mu_out, sigma = test_on_batch(adv_x, y)
    mu_out_[test_no_steps, :, :] = mu_out
    sigma_[test_no_steps, :, :] = sigma
    #sigma_[test_no_steps, :, :, :] = sigma
    corr = tf.equal(tf.math.argmax(mu_out, axis=1), tf.math.argmax(y, axis=1))
    accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
    acc_test[test_no_steps] = accuracy.numpy()
    if step % 10 == 0:
        print("Total running accuracy so far: %.3f" % accuracy.numpy())
    test_no_steps += 1

test_acc = np.mean(acc_test)
print('Test accuracy : ', test_acc)

pf = open(PATH + test_path + 'uncertainty_info.pkl', 'wb')
pickle.dump([mu_out_, sigma_, adv_perturbations, test_acc], pf)
pf.close()

var = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
snr_signal = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
for i in range(int(x_test.shape[0] / batch_size)):
    for j in range(batch_size):
        predicted_out = np.argmax(mu_out_[i, j, :])
        var[i, j] = sigma_[i, j, int(predicted_out)]
        snr_signal[i, j] = 10 * np.log10(np.sum(np.square(true_x[i, j, :, :, :])) / np.sum(
            np.square(epsilon * adv_perturbations[i, j, :, :, :])))

print('Output Variance', np.mean(var))
print('SNR', np.mean(snr_signal))
valid_size = x_test.shape[0]
pred_var = np.zeros(int(valid_size))
true_var = np.zeros(int(valid_size))
correct_classification = np.zeros(int(valid_size))
misclassification_pred = np.zeros(int(valid_size))
misclassification_true = np.zeros(int(valid_size))
predicted_out = np.zeros(int(valid_size))
true_out = np.zeros(int(valid_size))
k = 0
k1 = 0
k2 = 0
for i in range(int(valid_size / batch_size)):
    for j in range(batch_size):
        predicted_out[k] = np.argmax(mu_out_[i, j, :])
        true_out[k] = np.argmax(true_y[i, j, :])
        pred_var[k] = sigma_[i, j, int(predicted_out[k])]
        true_var[k] = sigma_[i, j, int(true_out[k])]
        if (predicted_out[k] == true_out[k]):
            correct_classification[k1] = sigma_[i, j, int(predicted_out[k])]
            k1 = k1 + 1
        if (predicted_out[k] != true_out[k]):
            misclassification_pred[k2] = sigma_[i, j, int(predicted_out[k])]
            misclassification_true[k2] = sigma_[i, j, int(true_out[k])]
            k2 = k2 + 1
        k = k + 1
print('Average Output Variance', np.mean(pred_var))

var1 = pred_var  # np.reshape(var, int(x_test.shape[0]/(batch_size))* batch_size)
# print(var1)
writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
df = pd.DataFrame(np.abs(var1))
# Write your DataFrame to a file
df.to_excel(writer, "Sheet")

df1 = pd.DataFrame(predicted_out)
df1.to_excel(writer, 'Sheet', startcol=4)

df2 = pd.DataFrame(true_out)
df2.to_excel(writer, 'Sheet', startcol=7)

df3 = pd.DataFrame(correct_classification)
df3.to_excel(writer, 'Sheet', startcol=10)

df4 = pd.DataFrame(misclassification_pred)
df4.to_excel(writer, 'Sheet', startcol=13)

df5 = pd.DataFrame(misclassification_true)
df5.to_excel(writer, 'Sheet', startcol=16)
writer.save()

pf = open(PATH + test_path + 'var_info.pkl', 'wb')
pickle.dump([correct_classification, misclassification_true, pred_var], pf)
pf.close()

textfile = open(PATH + test_path + 'Related_hyperparameters.txt', 'w')
textfile.write(' Input Dimension : ' + str(image_size))
#textfile.write('\n No of Kernels : ' + str(num_kernels))
textfile.write('\n Number of Classes : ' + str(num_classes))
textfile.write('\n No of epochs : ' + str(epochs))
textfile.write('\n Initial Learning rate : ' + str(lr))
textfile.write('\n Ending Learning rate : ' + str(lr_end))
#textfile.write('\n kernels Size : ' + str(kernels_size))
#textfile.write('\n Max pooling Size : ' + str(maxpooling_size))
#textfile.write('\n Max pooling stride : ' + str(maxpooling_stride))
textfile.write('\n batch size : ' + str(batch_size))
textfile.write('\n KL term factor : ' + str(kl_factor))
textfile.write("\n---------------------------------")
textfile.write("\n Averaged Test Accuracy : " + str(test_acc))
textfile.write("\n Output Variance: " + str(np.mean(np.abs(var))))
textfile.write("\n Correct Classification Variance: " + str(np.mean(correct_classification)))
textfile.write("\n MisClassification Variance: " + str(np.mean(misclassification_pred)))

textfile.write("\n---------------------------------")
if Adversarial_noise:
    if Targeted:
        textfile.write('\n Adversarial attack: TARGETED')
        textfile.write('\n The targeted attack class: ' + str(adversary_target_cls))
    else:
        textfile.write('\n Adversarial attack: Non-TARGETED')
    textfile.write('\n Adversarial Noise epsilon: ' + str(epsilon))
```

```
            textfile.write("\n SNR: " + str(np.mean(snr_signal)))
        textfile.write("\n----------------------------------")
        textfile.close()

if (PGD_Adversarial_noise):
    if Targeted:
      test_path = 'test_results_targeted_PGDadversarial_noise_{}_max_iter_{}_{}/'.format(HCV, maxAdvStep, stepSize)
      full_test_path = PATH + test_path
      if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
          shutil.rmtree(full_test_path)
      os.makedirs(PATH + test_path)
    else:
      test_path = 'test_results_non_targeted_PGDadversarial_noise_{}/'.format(HCV)
      full_test_path = PATH + test_path
      if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
          shutil.rmtree(full_test_path)
      os.makedirs(PATH + test_path)



    trans_model.load_weights(PATH + 'vdp_cnn_model')
    trans_model.trainable = False

    test_no_steps = 0
    true_x = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, channels])
    adv_perturbations = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, channels])
    true_y = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    mu_out_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size,  num_classes])
    #sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size,  num_classes,  num_classes])
    sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size,  num_classes])

    acc_test = np.zeros(int(x_test.shape[0] / (batch_size)))
    epsilon = HCV / 3
    for step, (x, y) in enumerate(val_dataset):
        update_progress(step / int(x_test.shape[0] / (batch_size)))
        true_x[test_no_steps, :, :, :] = x
        true_y[test_no_steps, :, :] = y

        adv_x = x + tf.random.uniform(x.shape, minval=-epsilon, maxval=epsilon)
        adv_x = tf.clip_by_value(adv_x, 0.0, 1.0)
        for advStep in range(maxAdvStep):
            if Targeted:
                y_true_batch = np.zeros_like(y)
                y_true_batch[:, adversary_target_cls] = 1.0
                adv_perturbations[test_no_steps, :, :, :] = create_adversarial_pattern(adv_x, y_true_batch)
            else:
                adv_perturbations[test_no_steps, :, :, :] = create_adversarial_pattern(adv_x, y)
            adv_x = adv_x + stepSize * adv_perturbations[test_no_steps, :, :, :]
            pgdTotalNoise = tf.clip_by_value(adv_x - x, -epsilon, epsilon)
            adv_x = tf.clip_by_value(x + pgdTotalNoise, 0.0, 1.0)

        mu_out, sigma = test_on_batch(adv_x, y)
        mu_out_[test_no_steps, :, :] = mu_out
        #sigma_[test_no_steps, :, :, :] = sigma
        sigma_[test_no_steps, :, :] = sigma
    corr = tf.equal(tf.math.argmax(mu_out, axis=-1), tf.math.argmax(y, axis=-1))
    accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
    acc_test[test_no_steps] = accuracy.numpy()
    if step % 50 == 0:
        print("Total running accuracy so far: %.4f" % acc_test[test_no_steps])
    test_no_steps += 1
test_acc = np.mean(acc_test)
print('Test accuracy : ', test_acc)
print('Best Test accuracy : ', np.amax(acc_test))

pf = open(PATH + test_path + 'uncertainty_info.pkl', 'wb')
pickle.dump([mu_out_, sigma_, true_x, true_y, adv_perturbations, test_acc], pf)
pf.close()

var = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
snr_signal = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
for i in range(int(x_test.shape[0] / batch_size)):
    for j in range(batch_size):
        predicted_out = np.argmax(mu_out_[i, j, :])
        var[i, j] = sigma_[i, j, int(predicted_out)]
        snr_signal[i, j] = 10 * np.log10(
            np.sum(np.square(true_x[i, j, :, :, :])) / np.sum(np.square(epsilon * adv_perturbations[i, j, :, :, :])))

print('Output Variance', np.mean(var))
print('SNR', np.mean(snr_signal))

##        var1 = np.reshape(var, int(x_test.shape[0]/(batch_size))* batch_size)
##        #print(var1)
##        writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
##        df = pd.DataFrame(np.abs(var1) )
##        # Write your DataFrame to a file
##        df.to_excel(writer, "Sheet")
##        writer.save()

textfile = open(PATH + test_path + 'Related_hyperparameters.txt', 'w')
textfile.write(' Input Dimension : ' + str(image_size))
#textfile.write('\n No of Kernels : ' + str(num_kernels))
textfile.write('\n Number of Classes : ' + str(num_classes))
textfile.write('\n No of epochs : ' + str(epochs))
textfile.write('\n Initial Learning rate : ' + str(lr))
textfile.write('\n Ending Learning rate : ' + str(lr_end))
#textfile.write('\n kernels Size : ' + str(kernels_size))
#textfile.write('\n Max pooling Size : ' + str(maxpooling_size))
#textfile.write('\n Max pooling stride : ' + str(maxpooling_stride))
textfile.write('\n batch size : ' + str(batch_size))
textfile.write('\n KL term factor : ' + str(kl_factor))
textfile.write("\n---------------------------------")
textfile.write("\n Test Accuracy : " + str(test_acc))
textfile.write("\n Output Variance: " + str(np.mean(np.abs(var))))
textfile.write("\n---------------------------------")
if PGD_Adversarial_noise:
    if Targeted:
        textfile.write('\n Adversarial attack: TARGETED')
        textfile.write('\n The targeted attack class: ' + str(adversary_target_cls))
    else:
        textfile.write('\n Adversarial attack: Non-TARGETED')
```

```
            textfile.write('\n Adversarial Noise epsilon: ' + str(epsilon))
            textfile.write('\n Adversarial Noise HCV: ' + str(HCV))
            textfile.write("\n SNR: " + str(np.mean(snr_signal)))
            textfile.write("\n stepSize: " + str(stepSize))
            textfile.write("\n Maximum number of iterations: " + str(maxAdvStep))
        textfile.write("\n---------------------------------")
        textfile.close()

if __name__ == '__main__':
    main_function()
```

1.2 Source Code of Bayes-CCT

```
# -*- coding: utf-8 -*-
import tensorflow as tf
# tf.config.run_functions_eagerly(True)
from tensorflow import keras
import os
from tensorflow.keras.layers import (Dense, Dropout, LayerNormalization, )
# os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
# For multiple devices (GPUs: 4, 5, 6, 7)
# os.environ["CUDA_VISIBLE_DEVICES"] = "1,4,5,6,7"
# import imageio
import matplotlib
import shutil

matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import math
import time, sys
import pickle
import timeit
import xlsxwriter

from scipy.interpolate import make_interp_spline, BSpline
from tensorflow.keras.layers.experimental.preprocessing import Rescaling
from tensorflow.keras import layers
# import tensorflow_addons as tfa
from keras.optimizers import SGD
import keras.backend as K
import pandas as pd
import wandb
from keras.preprocessing.image import ImageDataGenerator

os.environ["WANDB_API_KEY"] = "3df171be2d23f8aaf89ddc494bb7116af7a1ec9b"

import numpy as np
# !pip install tensorflow_addons
import tensorflow as tf
from tensorflow import keras
import math
from tensorflow.keras import layers

# import tensorflow_addons as tfa
```

```
from keras.optimizers import Adam
import keras.backend as K
import pandas as pd

plt.ioff()

cifar10 = tf.keras.datasets.cifar10



# update_progress() : Displays or updates a console progress bar
## Accepts a float between 0 and 1. Any int will be converted to a float.
## A value under 0 represents a 'halt'.
## A value at 1 or bigger represents 100%
def update_progress(progress):
    barLength = 10  # Modify this to change the length of the progress bar
    status = ""
    if isinstance(progress, int):
        progress = float(progress)
    if not isinstance(progress, float):
        progress = 0
        status = "error: progress var must be float\r\n"
    if progress < 0:
        progress = 0
        status = "Halt...\r\n"
    if progress >= 1:
        progress = 1
        status = "Done...\r\n"
    block = int(round(barLength * progress))
    text = "\rPercent: [{0}] {1}% {2}".format("#" * block + "-" * (barLength - block), progress * 100, status)
    sys.stdout.write(text)
    sys.stdout.flush()



# Auxilary Functions



def x_Sigma_w_x_T(x, W_Sigma):
    batch_sz = x.shape[0]
    xx_t = tf.reduce_sum(tf.multiply(x, x), axis=-1,
                         keepdims=True)  # [50, 17, 64]  -> [50, 17, 1] or [50, 64] - > [50, 1]
    # xx_t_e = tf.expand_dims(xx_t,axis=2)
    return tf.multiply(xx_t, W_Sigma)  # [50,17,64] or [50, 64] or [50, 10]



def w_t_Sigma_i_w(w_mu, in_Sigma):  # [64, 64]  , [50, 17, 64] or [64, 10], [50, 64]
    Sigma_1 = tf.matmul(in_Sigma, tf.multiply(w_mu, w_mu))  # [50, 17, 64] or [50, 10]
    return Sigma_1



def tr_Sigma_w_Sigma_in(in_Sigma, W_Sigma):
    Sigma = tf.reduce_sum(in_Sigma, axis=-1, keepdims=True)  # [50,17, 1]
    return tf.multiply(Sigma, W_Sigma)  # [50,17, 64]



def sigma_regularizer(x):
    input_size = 1.0
    f_s = tf.math.softplus(x)  # tf.math.log(1. + tf.math.exp(x))
    return input_size * tf.reduce_mean(-1. - tf.math.log(f_s) + f_s)



# Bayesian 1st ConV

class VDP_first_Conv(keras.layers.Layer):
    def __init__(self, kernel_size, kernel_num, kernel_stride, padding="VALID"):
        super(VDP_first_Conv, self).__init__()
        self.kernel_size = kernel_size
        self.kernel_num = kernel_num
        self.kernel_stride = kernel_stride
        self.padding = padding

    def build(self, input_shape):
        def sigma_regularizer_conv(x):
            f_s = tf.math.softplus(x)  # tf.math.log(1. + tf.math.exp(x))
            return (self.kernel_size * self.kernel_size * input_shape[-1]) * tf.reduce_mean(f_s - tf.math.log(f_s) - 1.)

        ini_sigma = -6.9
        tau = 1.  # / (self.kernel_size* self.kernel_size*  input_shape[-1])
        # ini_sigma = -6.9
        # min_sigma = -4.5
        # tau = 1.  # / (self.kernel_size * self.kernel_size * input_shape[-1])
        self.w_mu = self.add_weight(name='w_mu',
                                    shape=(self.kernel_size, self.kernel_size, input_shape[-1], self.kernel_num),
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.05, seed=None),
                                    regularizer=tf.keras.regularizers.l2(tau),  # l1_l2(l1=tau, l2=tau)
                                    trainable=True,
                                    )
        self.w_sigma = self.add_weight(name='w_sigma',
                                       shape=(self.kernel_num,),
                                       initializer=tf.constant_initializer(ini_sigma),
                                       # tf.random_uniform_initializer(minval=min_sigma, maxval=ini_sigma,  seed=None),
                                       regularizer=sigma_regularizer_conv,
                                       trainable=True,
                                       )

    def call(self, mu_in):
        batch_size = mu_in.shape[0]
        num_channel = mu_in.shape[-1]
        mu_out = tf.nn.conv2d(mu_in, self.w_mu, strides=[1, self.kernel_stride, self.kernel_stride, 1],
                              padding=self.padding, data_format='NHWC')
        x_train_patches = tf.image.extract_patches(mu_in, sizes=[1, self.kernel_size, self.kernel_size, 1],
                                                   strides=[1, self.kernel_stride, self.kernel_stride, 1],
                                                   rates=[1, 1, 1, 1],
                                                   padding=self.padding)  # shape=[batch_size, image_size, image_size, kernel_size*kernel_size*num_cha
        x_train_matrix = tf.reshape(x_train_patches, [batch_size, -1,
                                                      self.kernel_size * self.kernel_size * num_channel])  # shape=[batch_size, image_size*image_size,
        x_train_matrix = tf.math.reduce_sum(tf.math.square(x_train_matrix),
                                            axis=-1)  # shape=[batch_size, image_size*image_size] = [16, 576]

        X_XTranspose = tf.ones([1, 1, self.kernel_num]) * tf.expand_dims(x_train_matrix, axis=-1)
        Sigma_out = tf.multiply(tf.math.log(1. + tf.math.exp(self.w_sigma)),
                                X_XTranspose)  # shape=[batch_size,image_size*image_size, kernel_num]
        Sigma_out = tf.reshape(Sigma_out, [batch_size, mu_out.shape[1], mu_out.shape[1], self.kernel_num])
        #         Sigma_out = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))
        return mu_out, Sigma_out  # mu_shape= [batch_size, image_size, image_size, kernel_num], sigma_shape=[batch_size, image_size, image_size, kernel


# Bayesian Intermediate Conv


class VDP_intermediate_Conv(keras.layers.Layer):
    def __init__(self, kernel_size=5, kernel_num=16, kernel_stride=1, padding="VALID"):
        super(VDP_intermediate_Conv, self).__init__()
        self.kernel_size = kernel_size
        self.kernel_num = kernel_num
        self.kernel_stride = kernel_stride
        self.padding = padding

    def build(self, input_shape):
        ini_sigma = -6.9
        # min_sigma = -4.5
        tau = 1.  # / (self.kernel_size * self.kernel_size * input_shape[-1])
        self.w_mu = self.add_weight(name='w_mu',
                                    shape=(self.kernel_size, self.kernel_size, input_shape[-1], self.kernel_num),
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.05, seed=None),
                                    regularizer=tf.keras.regularizers.l2(tau),  # l1_l2(l1=tau, l2=tau)
                                    trainable=True,
                                    )
        self.w_sigma = self.add_weight(name='w_sigma',
                                       shape=(self.kernel_num,),
                                       initializer=tf.constant_initializer(ini_sigma),
                                       # tf.random_uniform_initializer(minval=min_sigma, maxval=ini_sigma,  seed=None),
                                       regularizer=sigma_regularizer,
                                       trainable=True,
                                       )

    def call(self, mu_in, Sigma_in):  # [batch_size, image_size , image_size, channel]
        batch_size = mu_in.shape[0]
        num_channel = mu_in.shape[-1]  # shape=[batch_size, im_size, im_size, num_channel]
        mu_out = tf.nn.conv2d(mu_in, self.w_mu, strides=[1, self.kernel_stride, self.kernel_stride, 1],
                              padding=self.padding, data_format='NHWC')

        diag_sigma_patches = tf.image.extract_patches(Sigma_in, sizes=[1, self.kernel_size, self.kernel_size, 1],
                                                      strides=[1, self.kernel_stride, self.kernel_stride, 1],
                                                      rates=[1, 1, 1, 1],
                                                      padding=self.padding)  # shape=[batch_size, new_im_size, new_im_size, kernel_size*kernel_size*nu

        diag_sigma_g = tf.reshape(diag_sigma_patches, [batch_size, -1,
                                                       self.kernel_size * self.kernel_size * num_channel])  # shape=[batch_size, new_im_size*new_im_si
self.kernel_size*self.kernel_size*num_channel ]
        mu_cov_square = tf.reshape(tf.math.multiply(self.w_mu, self.w_mu),
                                   [self.kernel_size * self.kernel_size * num_channel,
                                    self.kernel_num])  # shape[ kernel_size*kernel_size*num_channel,   kernel_num]

        mu_wT_sigmags_mu_w = tf.matmul(diag_sigma_g,
                                       mu_cov_square)  # shape=[batch_size, new_im_size*new_im_size , kernel_num   ]
        trace = tf.math.reduce_sum(diag_sigma_g, 2, keepdims=True)  # shape=[batch_size,  new_im_size* new_im_size, 1]
        trace = tf.ones([1, 1, self.kernel_num]) * trace  # shape=[batch_size,  new_im_size*new_im_size, kernel_num]
        trace = tf.multiply(tf.math.log(1. + tf.math.exp(self.w_sigma)),
                            trace)  # shape=[batch_size, , new_im_size*new_im_size, kernel_num]

        mu_in_patches = tf.reshape(tf.image.extract_patches(mu_in, sizes=[1, self.kernel_size, self.kernel_size, 1],
                                                            strides=[1, self.kernel_stride, self.kernel_stride, 1],
                                                            rates=[1, 1, 1, 1], padding=self.padding),
                                   [batch_size, -1,
                                    self.kernel_size * self.kernel_size * num_channel])  # shape=[batch_size, new_im_size*new_im_size, self.kernel_size

        mu_gT_mu_g = tf.math.reduce_sum(tf.math.multiply(mu_in_patches, mu_in_patches),
                                        axis=-1)  # shape=[batch_size, new_im_size*new_im_size]
        mu_gT_mu_g1 = tf.ones([1, 1, self.kernel_num]) * tf.expand_dims(mu_gT_mu_g,
                                                                       axis=-1)  # shape=[batch_size, new_im_size*new_im_size, kernel_num]
        sigmaw_mu_gT_mu_g = tf.multiply(tf.math.log(1. + tf.math.exp(self.w_sigma)),
                                        mu_gT_mu_g1)  # shape=[batch_size, new_im_size*new_im_size, kernel_num]

        Sigma_out = trace + mu_wT_sigmags_mu_w + sigmaw_mu_gT_mu_g  # # shape=[batch_size, new_im_size*new_im_size, kernel_num]
        Sigma_out = tf.reshape(Sigma_out, [batch_size, mu_out.shape[1], mu_out.shape[1], self.kernel_num])
        return mu_out, Sigma_out


# Bayesian Maxpooling

class VDP_MaxPooling(keras.layers.Layer):
    """VDP_MaxPooling"""

    def __init__(self, pooling_size=2, pooling_stride=2, pooling_pad='SAME'):
        super(VDP_MaxPooling, self).__init__()
        self.pooling_size = pooling_size
        self.pooling_stride = pooling_stride
        self.pooling_pad = pooling_pad

    def call(self, mu_in, Sigma_in):  # shape=[batch_size,,im_size, im_size, num_channel]
        batch_size = mu_in.shape[0]  # shape=[batch_size, im_size, im_size, num_channel]
        hw_in = mu_in.shape[1]
        num_channel = mu_in.shape[-1]
        mu_out, argmax_out = tf.nn.max_pool_with_argmax(mu_in, ksize=[1, self.pooling_size, self.pooling_size, 1],
                                                        strides=[1, self.pooling_stride, self.pooling_stride, 1],
                                                        padding=self.pooling_pad)  # shape=[batch_zise, new_size,new_size,num_channel]
        hw_out = mu_out.shape[1]
        argmax1 = tf.transpose(argmax_out, [0, 3, 1, 2])
        argmax2 = tf.reshape(argmax1, [batch_size, num_channel,
                                       -1])  # shape=[batch_size, num_channel, new_size*new_size]
        x_index = tf.math.floormod(tf.compat.v1.floor_div(argmax2, tf.constant(num_channel,
                                                                              shape=[batch_size, num_channel,
                                                                                     hw_out * hw_out], dtype='int64')),
                                   tf.constant(hw_in, shape=[batch_size, num_channel, hw_out * hw_out], dtype='int64'))
        aux = tf.compat.v1.floor_div(tf.compat.v1.floor_div(argmax2, tf.constant(num_channel,
                                                                                shape=[batch_size, num_channel,
                                                                                       hw_out * hw_out],
                                                                                dtype='int64')),
                                     tf.constant(hw_in, shape=[batch_size, num_channel, hw_out * hw_out],
                                                 dtype='int64'))
        y_index = tf.math.floormod(aux,
                                    tf.constant(hw_in, shape=[batch_size, num_channel, hw_out * hw_out], dtype='int64'))
        index = tf.multiply(y_index, hw_in) + x_index  # shape=[batch_size, num_channel,new_size*new_size]
        Sigma_in1 = tf.transpose(tf.reshape(Sigma_in, [batch_size, -1, num_channel]), [0, 2, 1])
        Sigma_out = tf.gather(Sigma_in1, index, batch_dims=2,
                              axis=-1)  # shape=[batch_size,num_channel,new_size*new_size]
        Sigma_out = tf.reshape(tf.transpose(Sigma_out, [0, 2, 1]), [batch_size, mu_out.shape[1], mu_out.shape[1],
                                                                     num_channel])  # shape=[batch_size,new_size, new_size, num_channel]
        return mu_out, Sigma_out


# Bayesian BatchNorm

class VDPBatch_Normalization(keras.layers.Layer):
    def __init__(self, var_epsilon):
        super(VDPBatch_Normalization, self).__init__()
        self.var_epsilon = var_epsilon

    def call(self, mu_in, Sigma_in):
        mean, variance = tf.nn.moments(mu_in, [0, 1, 2])
        mu_out = tf.nn.batch_normalization(mu_in, mean, variance, offset=None, scale=None,
                                           variance_epsilon=self.var_epsilon)
        Sigma_out = tf.multiply(Sigma_in, 1 / (variance + self.var_epsilon))
        return mu_out, Sigma_out


# Bayesian Fully Connected Layers


class LinearFirst(keras.layers.Layer):
    """y = w.x + b"""

    def __init__(self, units):
        super(LinearFirst, self).__init__()
        self.units = units

    def build(self, input_shape):
        tau = 0.01  # 1. /input_shape[-1]
        ini_sigma = -6.5
        self.w_mu = self.add_weight(name='w_mu',
                                    shape=(input_shape[-1], self.units),
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.05, seed=None),
                                    regularizer=tf.keras.regularizers.l2(tau),
                                    trainable=True)
        self.w_sigma = self.add_weight(name='w_sigma',
                                       shape=(self.units,),
                                       initializer=tf.constant_initializer(ini_sigma),
                                       # initializer=tf.random_uniform_initializer(minval= -12., maxval=-2.2, seed=None),
                                       regularizer=sigma_regularizer,
                                       trainable=True)

    def call(self, inputs):  # [50,17,64]
        # Mean
        # print(self.w_mu.shape)
        mu_out = tf.matmul(inputs, self.w_mu)  # + self.b_mu      [50, 17, 64]          # Mean of the output
        # Varinace
        W_Sigma = tf.math.log(
            1. + tf.math.exp(self.w_sigma))  # [64]                          # Construct W_Sigma from w_sigmas
        Sigma_out = x_Sigma_w_x_T(inputs,
                                  W_Sigma)  # [50, 17, 64]            + tf.math.log(1. + tf.math.exp(self.b_sigma)) #tf.linalg.diag(self.b_sigma)
        Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.abs(Sigma_out)
        return mu_out, Sigma_out


class LinearNotFirst(keras.layers.Layer):
    """y = w.x + b"""

    def __init__(self, units):
        super(LinearNotFirst, self).__init__()
        self.units = units

    def build(self, input_shape):
        ini_sigma = -6.9
        # min_sigma = -4.5
        tau = 1.  # 1. /input_shape[-1]

        self.w_mu = self.add_weight(name='w_mu', shape=(input_shape[-1], self.units),
                                    # [64 , 64] or or [64, 10] or [10, 10]
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.05, seed=None),
                                    regularizer=tf.keras.regularizers.l2(tau),
                                    # tau/self.units), #tf.keras.regularizers.l2(0.5*0.001),
                                    trainable=True, )
        self.w_sigma = self.add_weight(name='w_sigma',
                                       shape=(self.units,),
                                       initializer=tf.constant_initializer(ini_sigma),
                                       # tf.random_uniform_initializer(minval= min_sigma, maxval=ini_sigma, seed=None) ,
                                       regularizer=sigma_regularizer,  # tf.constant_initializer(ini_sigma)
                                       trainable=True, )

    def call(self, mu_in, Sigma_in):  # [50,17,64],  [50,17,64]   or [50, 64] or [50, 10]
        mu_out = tf.matmul(mu_in, self.w_mu)  # + self.b_mu  [50, 17, 64]

        W_Sigma = tf.math.log(1. + tf.math.exp(self.w_sigma))  # [64]
        Sigma_1 = w_t_Sigma_i_w(self.w_mu, Sigma_in)  # [50,17,64]
        Sigma_2 = x_Sigma_w_x_T(mu_in, W_Sigma)  # [50, 17, 64]
        Sigma_3 = tr_Sigma_w_Sigma_in(Sigma_in, W_Sigma)  # [50, 17, 64]
        Sigma_out = Sigma_1 + Sigma_2 + Sigma_3  # + tf.linalg.diag(tf.math.log(1. + tf.math.exp(self.b_sigma)))  #[50, 17, 64]

        Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)  # [50,2,17,64,64]
        Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)  # [50,2,17,64,64]
        Sigma_out = tf.abs(Sigma_out)
        return mu_out, Sigma_out  # mu_out=[50,17,64], Sigma_out = [50,17,64]


# Bayesian Activation Functions

class VDP_GeLU(keras.layers.Layer):
    def __init__(self):
        super(VDP_GeLU, self).__init__()

    def call(self, mu_in, Sigma_in):  # mu_in = [50,17,64], Sigma_in= [50,17,64]
```

```
        mu_out = tf.nn.gelu(mu_in)  # [50,17,64]
        with tf.GradientTape() as g:
            g.watch(mu_in)
            out = tf.nn.gelu(mu_in)
        gradi = g.gradient(out, mu_in)  # [50,17,64]
        Sigma_out = activation_Sigma(gradi, Sigma_in)
        return mu_out, Sigma_out  # [50,2,17,64], [50,2,17,64,64]


def activation_Sigma(gradi, Sigma_in):
    grad1 = tf.multiply(gradi, gradi)  # [50,17,64] or [50, 10]
    return tf.multiply(Sigma_in, grad1)  # [50,17,64] or [50, 10]


class VDP_ReLU(keras.layers.Layer):
    """ReLU"""

    def __init__(self):
        super(VDP_ReLU, self).__init__()

    def call(self, mu_in, Sigma_in):
        mu_out = tf.nn.relu(mu_in)
        with tf.GradientTape() as g:
            g.watch(mu_in)
            out = tf.nn.relu(mu_in)
        gradi = g.gradient(out, mu_in)
        Sigma_out = activation_Sigma(gradi, Sigma_in)
        return mu_out, Sigma_out


# Bayesian Dropout

class VDP_Dropout(keras.layers.Layer):
    def __init__(self, drop_prop):
        super(VDP_Dropout, self).__init__()
        self.drop_prop = drop_prop

    def call(self, mu_in, Sigma_in, Training=True):
        # shape=[batch_size, seq length, embedding_dim]
        scale_sigma = 1.0 / (1 - self.drop_prop)
        if Training:
            mu_out = tf.nn.dropout(mu_in, rate=self.drop_prop)  # [50,17,64] or [50, 10]
            # print('shape in dropout',mu_out.shape)
            non_zero = tf.not_equal(mu_out, tf.zeros_like(mu_out))  # [50,17,64]
            non_zero_sigma_mask = tf.boolean_mask(Sigma_in, non_zero)
            idx_sigma = tf.dtypes.cast(tf.where(non_zero), tf.int32)
            Sigma_out = (scale_sigma ** 2) * tf.scatter_nd(idx_sigma, non_zero_sigma_mask, tf.shape(non_zero))
            # print('sigma shape in dropout',Sigma_out.shape)
        else:
            mu_out = mu_in
            Sigma_out = Sigma_in
        return mu_out, Sigma_out  # [50,17,64], [50,17,64]


# Bayesian Multi Layer Perceptron
```

```
class VDP_MLP(tf.keras.layers.Layer):
    def __init__(self, hidden_features, out_features, dropout_rate=0.1):
        super(VDP_MLP, self).__init__()
        self.dense1 = LinearNotFirst(hidden_features)
        # self.dense1 = LinearNotFirst(mlp_dim)
        self.dense2 = LinearNotFirst(out_features)
        # self.dense2 = LinearNotFirst(embed_dim)
        self.dropout1 = VDP_Dropout(dropout_rate)
        self.gelu_1 = VDP_GeLU()

    def call(self, mu_in, sigma_in):
        mu_out, sigma_out = self.dense1(mu_in, sigma_in)
        # print('shape of x(MLP layer) :',mu_out.shape)
        mu_out, sigma_out = self.gelu_1(mu_out, sigma_out)
        # print('shape of x through GeLU :',mu_out.shape)
        mu_out, sigma_out = self.dropout1(mu_out, sigma_out)
        # print('shape of x after dropout :',mu_out.shape)
        mu_out, sigma_out = self.dense2(mu_out, sigma_out)
        # print('shape of x after 2nd dense :',mu_out.shape)
        mu_out, sigma_out = self.dropout1(mu_out, sigma_out)
        # print('shape of mu_out after MLP layer', mu_out.shape)
        return mu_out, sigma_out


# Deterministic Layernorm

class LayerNorm(tf.keras.layers.Layer):
    def __init__(self, eps=1e-4, **kwargs):
        self.eps = eps
        super(LayerNorm, self).__init__(**kwargs)

    def build(self, input_shape):
        self.gamma = self.add_weight(name='gamma', shape=input_shape[-1:],
                                     initializer=tf.keras.initializers.Ones(), trainable=True)
        self.beta = self.add_weight(name='beta', shape=input_shape[-1:],
                                    initializer=tf.keras.initializers.Zeros(), trainable=True)
        super(LayerNorm, self).build(input_shape)

    def call(self, x):
        mean = K.mean(x, axis=-1, keepdims=True)
        std = K.std(x, axis=-1, keepdims=True)
        # print( "mean of LN",mean.shape)
        # print("std of LN",std.shape)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta

    def compute_output_shape(self, input_shape):
        return input_shape


# Bayesian Layernorm
class Bayesian_LayerNorm(layers.Layer):

    def __init__(self, eps=1e-5, **kwargs):
        self.eps = eps
        super(Bayesian_LayerNorm, self).__init__(**kwargs)
    def build(self, input_shape):
        self.gamma = self.add_weight(name='gamma', shape=input_shape[-1:],
                                     initializer=tf.keras.initializers.Ones(), trainable=True)
        self.beta = self.add_weight(name='beta', shape=input_shape[-1:],
                                    initializer=tf.keras.initializers.Zeros(), trainable=True)
        super(Bayesian_LayerNorm, self).build(input_shape)

    def call(self, mu_x,
             sigma_x):  # (batch_size, sequence_length, embedding_dim), (batch_size, sequence_length, embedding_dim)[50,17, 64],[50,17,64]
        mean = K.mean(mu_x, axis=-1, keepdims=True)  # [50,17,1]
        std = K.std(mu_x, axis=-1, keepdims=True)  # [50,17,1]
        # print('std = ', std.shape)
        #  print('gamma = ',self.gamma)
        out_mu = self.gamma * (mu_x - mean) / (std + self.eps) + self.beta
        a = (self.gamma / (std + self.eps)) ** 2  # [50,17,64]

        out_sigma = tf.math.multiply(a, sigma_x)  # [50,17,64]
        return out_mu, out_sigma

    def compute_output_shape(self, input_shape):
        return input_shape


# Bayesian Multi Head Attention

class Bayesian_MultiHeadSelfAttention_First(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads):
        super(Bayesian_MultiHeadSelfAttention_First, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        if embed_dim % num_heads != 0:
            raise ValueError(
                f"embedding dimension = {embed_dim} should be divisible by number of heads = {num_heads}"
            )
        self.projection_dim = embed_dim // num_heads
        self.query_dense = LinearNotFirst(embed_dim)
        self.key_dense = LinearNotFirst(embed_dim)
        self.value_dense = LinearNotFirst(embed_dim)
        self.combine_heads = LinearNotFirst(embed_dim)

    def attention(self, mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value, input_dimension):
        mu_score = tf.matmul(mu_query, mu_key, transpose_b=True)  # [50, 2, 17, 32] x [50, 2, 32, 17] = [50, 2, 17, 17]

        # print('mu_score',mu_score.shape)

        a = tf.reduce_sum(tf.math.multiply(mu_query ** 2, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        print('a', a.shape)
        b = tf.transpose(tf.reduce_sum(tf.math.multiply(mu_key ** 2, sigma_query), axis=-1, keepdims=True),
                         perm=[0, 1, 3, 2])  # [50, 2, 1, 17 ]
        a_b = a + b  # [50, 2, 17, 17]

        c = tf.matmul(sigma_query, sigma_key, transpose_b=True)  # [50, 2, 17, 17]
        #  c1 = tf.reduce_sum(tf.math.multiply(sigma_query, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        # print('c1',c1.shape)
        #  c2 = tf.transpose(c1, perm=[0, 1, 3, 2])  # [50, 2, 1, 17]
        #  c = c1 + c2  # [50, 2, 17, 17]
        # print('c1+c2',c.shape)
        sigma_score = a_b + c  # [50, 2, 17, 17]
        # print('sigma score',sigma_score.shape)
        dim_key = tf.cast(tf.shape(mu_key)[-1], tf.float32)
        mu_scaled_score = mu_score / tf.math.sqrt(dim_key)  # [50, 2, 17, 17]
        # print('mu scaled score',mu_scaled_score.shape)
        sigma_scaled_score = sigma_score * dim_key  # [50, 2, 17, 17]

        mu_weights = tf.nn.softmax(mu_scaled_score, axis=-1)  # [50, 2, 17, 17]
        # Sigma for softmax function
        pp1 = tf.expand_dims(mu_weights, axis=-1)  # [50, 2, 17, 17,1]
        pp2 = tf.expand_dims(mu_weights, axis=3)  # [50, 2, 17,1, 17]
        ppT = tf.matmul(pp1, pp2)  # # [50, 2, 17, 17,17]
        p_diag = tf.linalg.diag(mu_weights)  # [50, 2, 17, 17,17]
        grad = (p_diag - ppT) ** 2  # # [50, 2, 17, 17,17]
        Sigma_weights = tf.squeeze(tf.matmul(grad, tf.expand_dims(sigma_scaled_score, axis=-1)))  # [50, 2, 17, 17]
        Sigma_weights = tf.where(tf.math.is_nan(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        Sigma_weights = tf.where(tf.math.is_inf(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        # Sigma_weights = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))

        mu_output = tf.matmul(mu_weights, mu_value)  # [50,2,17,17] X [50,2,17,32]=  [50,2,17,32]
        # print('mu output',mu_output.shape)
        d = tf.matmul(mu_weights ** 2, sigma_value)  # [50,2,17,32]
        e = tf.matmul(Sigma_weights, mu_value ** 2)  # [50,2,17,32]
        f = tf.matmul(Sigma_weights, sigma_value)  # [50, 2, 17, 17]x[50,2,17,32]=  [50,2,17,32]
        output_sigma = d + e + f
        return mu_output, output_sigma  # , mu_weights, Sigma_weights

    def separate_heads(self, mu_x, sigma_x, batch_size):  # [50, 17,64], [50, 17, 64]
        mu_x = tf.reshape(mu_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 ,32]
        # print('mu_x',mu_x.shape)
        sigma_x = tf.reshape(sigma_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 32]
        mu_x = tf.transpose(mu_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        sigma_x = tf.transpose(sigma_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        return mu_x, sigma_x  # [50,2,17,32],[50,2,17,32]

    def call(self, mu_inputs, sigma_inputs):
        batch_size = tf.shape(mu_inputs)[0]
        mu_query, sigma_query = self.query_dense(mu_inputs, sigma_inputs)  # [50, 17,64]  , [50, 17,64]
        mu_key, sigma_key = self.key_dense(mu_inputs, sigma_inputs)  # [50, 17,64] , [50, 17,64]
        mu_value, sigma_value = self.value_dense(mu_inputs, sigma_inputs)  # [50, 17,64], [50, 17,64]

        mu_query, sigma_query = self.separate_heads(mu_query, sigma_query, batch_size)
        mu_key, sigma_key = self.separate_heads(mu_key, sigma_key, batch_size)
        mu_value, sigma_value = self.separate_heads(mu_value, sigma_value, batch_size)
        # print('query2 in MHA after passing through separate heads=',mu_query.shape)

        # mu_attention, sigma_attention = self.attention(mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value)
        mu_attention, sigma_attention = self.attention(mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value,
                                                       tf.shape(mu_inputs)[1])
        mu_attention = tf.transpose(mu_attention, perm=[0, 2, 1, 3])  # [50,17,2,32]
        # print('mu attention',mu_attention.shape)
        sigma_attention = tf.transpose(sigma_attention, perm=[0, 2, 1, 3])
        # print('sigma attention',sigma_attention.shape)

        mu_concat_attention = tf.reshape(mu_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention:",mu_concat_attention.shape) #[50,17,64]
        sigma_concat_attention = tf.reshape(sigma_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention sigma:",sigma_concat_attention.shape) #[50,17,64]

        mu_output, sigma_output = self.combine_heads(mu_concat_attention, sigma_concat_attention)
        print('shape after combine head mu after first MHA', mu_output.shape)
        print('shape after combine head sigma after first MHA', sigma_output.shape)
        # sigma_output = self.combine_heads(sigma_concat_attention)
        return mu_output, sigma_output


class Bayesian_MultiHeadSelfAttention_Intermediate(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads):
        super(Bayesian_MultiHeadSelfAttention_Intermediate, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        if embed_dim % num_heads != 0:
            raise ValueError(
                f"embedding dimension = {embed_dim} should be divisible by number of heads = {num_heads}"
            )
        self.projection_dim = embed_dim // num_heads
        self.query_dense = LinearNotFirst(embed_dim)
        self.key_dense = LinearNotFirst(embed_dim)
        self.value_dense = LinearNotFirst(embed_dim)
        self.combine_heads = LinearNotFirst(embed_dim)

    def attention(self, mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value):
        mu_score = tf.matmul(mu_query, mu_key, transpose_b=True)  # [50, 2, 17, 32] x [50, 2, 32, 17] = [50, 2, 17, 17]

        # print('mu_score',mu_score.shape)

        a = tf.reduce_sum(tf.math.multiply(mu_query ** 2, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        print('a', a.shape)
        b = tf.transpose(tf.reduce_sum(tf.math.multiply(mu_key ** 2, sigma_query), axis=-1, keepdims=True),
                         perm=[0, 1, 3, 2])  # [50, 2, 1, 17 ]
        a_b = a + b  # [50, 2, 17, 17]

        c1 = tf.reduce_sum(tf.math.multiply(sigma_query, sigma_key), axis=-1, keepdims=True)  # [50, 2, 17, 1]
        # print('c1',c1.shape)
        c2 = tf.transpose(c1, perm=[0, 1, 3, 2])  # [50, 2, 1, 17]
        c = c1 + c2  # [50, 2, 17, 17]
        # print('c1+c2',c.shape)
        sigma_score = a_b + c  # [50, 2, 17, 17]
        # print('sigma score',sigma_score.shape)
        dim_key = tf.cast(tf.shape(mu_key)[-1], tf.float32)
        mu_scaled_score = mu_score / tf.math.sqrt(dim_key)  # [50, 2, 17, 17]
        # print('mu scaled score',mu_scaled_score.shape)
        sigma_scaled_score = sigma_score * dim_key  # [50, 2, 17, 17]

        mu_weights = tf.nn.softmax(mu_scaled_score, axis=-1)  # [50, 2, 17, 17]
        # Sigma for softmax function
        pp1 = tf.expand_dims(mu_weights, axis=-1)  # [50, 2, 17, 17,1]
        pp2 = tf.expand_dims(mu_weights, axis=3)  # [50, 2, 17,1, 17]
        ppT = tf.matmul(pp1, pp2)  # # [50, 2, 17, 17,17]
        p_diag = tf.linalg.diag(mu_weights)  # [50, 2, 17, 17,17]
        grad = (p_diag - ppT) ** 2  # # [50, 2, 17, 17,17]
        Sigma_weights = tf.squeeze(tf.matmul(grad, tf.expand_dims(sigma_scaled_score, axis=-1)))  # [50, 2, 17, 17]
        Sigma_weights = tf.where(tf.math.is_nan(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        Sigma_weights = tf.where(tf.math.is_inf(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
        # Sigma_weights = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))

        mu_output = tf.matmul(mu_weights, mu_value)  # [50,2,17,17] X [50,2,17,32]=  [50,2,17,32]
        # print('mu output',mu_output.shape)
        d = tf.matmul(mu_weights ** 2, sigma_value)  # [50,2,17,32]
        e = tf.matmul(Sigma_weights, mu_value ** 2)  # [50,2,17,32]
        f = tf.matmul(Sigma_weights, sigma_value)  # [50, 2, 17, 17]x[50,2,17,32]=  [50,2,17,32]
        output_sigma = d + e + f
        return mu_output, output_sigma  # , mu_weights, Sigma_weights

    def separate_heads(self, mu_x, sigma_x, batch_size):  # [50, 17,64], [50, 17, 64]
        mu_x = tf.reshape(mu_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 ,32]
        # print('mu_x',mu_x.shape)
        sigma_x = tf.reshape(sigma_x, (batch_size, -1, self.num_heads, self.projection_dim))  # [50, 17, 2 32]
        mu_x = tf.transpose(mu_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        sigma_x = tf.transpose(sigma_x, perm=[0, 2, 1, 3])  # [50, 2, 17, 32]
        return mu_x, sigma_x  # [50,2,17,32],[50,2,17,32]

    def call(self, mu_inputs, sigma_inputs):
        batch_size = tf.shape(mu_inputs)[0]
        mu_query, sigma_query = self.query_dense(mu_inputs, sigma_inputs)  # [50, 17,64]  , [50, 17,64]
        mu_key, sigma_key = self.key_dense(mu_inputs, sigma_inputs)  # [50, 17,64] , [50, 17,64]
        mu_value, sigma_value = self.value_dense(mu_inputs, sigma_inputs)  # [50, 17,64], [50, 17,64]

        mu_query, sigma_query = self.separate_heads(mu_query, sigma_query, batch_size)
        mu_key, sigma_key = self.separate_heads(mu_key, sigma_key, batch_size)
        mu_value, sigma_value = self.separate_heads(mu_value, sigma_value, batch_size)
        # print('query2 in MHA after passing through separate heads=',mu_query.shape)

        mu_attention, sigma_attention = self.attention(mu_query, sigma_query, mu_key, sigma_key, mu_value, sigma_value)
        mu_attention = tf.transpose(mu_attention, perm=[0, 2, 1, 3])  # [50,17,2,32]
        # print('mu attention',mu_attention.shape)
        sigma_attention = tf.transpose(sigma_attention, perm=[0, 2, 1, 3])
        # print('sigma attention',sigma_attention.shape)

        mu_concat_attention = tf.reshape(mu_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention:",mu_concat_attention.shape) #[50,17,64]
        sigma_concat_attention = tf.reshape(sigma_attention, (batch_size, -1, self.embed_dim))
        # print("shape after concat_attention sigma:",sigma_concat_attention.shape) #[50,17,64]

        mu_output, sigma_output = self.combine_heads(mu_concat_attention, sigma_concat_attention)
        # print('shape after combine head mu', mu_output.shape)
        # print('shape after combine head sigma', sigma_output.shape)
        # sigma_output = self.combine_heads(sigma_concat_attention)
        return mu_output, sigma_output


# Bayesian Transformer Block

class VDP_TransformerBlock_first(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, mlp_dim, dropout=0.1):
        super(VDP_TransformerBlock_first, self).__init__()
        self.att = Bayesian_MultiHeadSelfAttention_First(embed_dim, num_heads)  # [64,2]
        self.mlp = VDP_MLP(mlp_dim * 2, mlp_dim, dropout)  # [64*2,64,dropout]
        self.layernorm1 = Bayesian_LayerNorm(eps=1e-6)
        self.layernorm2 = Bayesian_LayerNorm(eps=1e-6)
        self.dropout1 = VDP_Dropout(dropout)

    def call(self, mu_input, sigma_input, training):
        mu_output, sigma_out = self.layernorm1(mu_input, sigma_input)  # [50,17,64]
        # print("output of first LN before MHA",inputs_norm.shape) #[50,17,64]
        mu_output, sigma_out = self.att(mu_output, sigma_out)  # [50,17,64]
        # print("output of  MHA in TB",mu_output.shape)
        mu_output, sigma_out1 = self.dropout1(mu_output, sigma_out, training=training)  # [50,17,64]
        # print("output of  MHA in TB after dropout",mu_output.shape)
        mu_out1 = mu_output + mu_input  # [50,17,64]
        # print('output of of MHA before entering to MLP',mu_out1.shape)

        mu_out1_norm, sigma_out1_norm = self.layernorm2(mu_out1, sigma_out1)
        mu_mlp_output, sigma_mlp_output = self.mlp(mu_out1_norm, sigma_out1_norm)
        mu_mlp_output, sigma_mlp_output = self.dropout1(mu_mlp_output, sigma_mlp_output, training=training)
        # print('2nd LN and MLP output',mu_mlp_output.shape)
        mu_output = mu_mlp_output + mu_out1
        with tf.GradientTape() as g:
            g.watch(mu_out1)
            out = mu_mlp_output + mu_out1
        gradi = g.gradient(out, mu_out1)
        sigma_output = tf.math.multiply(tf.math.multiply(gradi, gradi), sigma_out1)
        return mu_output, sigma_output


class VDP_TransformerBlock_Intermediate(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, mlp_dim, dropout=0.1):
        super(VDP_TransformerBlock_Intermediate, self).__init__()
        self.att = Bayesian_MultiHeadSelfAttention_Intermediate(embed_dim, num_heads)  # [64,2]
        self.mlp = VDP_MLP(mlp_dim * 2, mlp_dim, dropout)  # [64*2,64,dropout]
        self.layernorm1 = Bayesian_LayerNorm(eps=1e-6)
        self.layernorm2 = Bayesian_LayerNorm(eps=1e-6)
        self.dropout1 = VDP_Dropout(dropout)

    def call(self, mu_inputs, sigma_inputs, training):
        mu_norm, sigma_norm = self.layernorm2(mu_inputs, sigma_inputs)  # [50,17,64]
        # print("output of first LN before MHA",inputs_norm.shape) #[50,17,64]
        mu_output, sigma_out = self.att(mu_norm, sigma_norm)  # [50,17,64]
        # print("output of  MHA in TB",mu_output.shape)
        mu_output, sigma_out1 = self.dropout1(mu_output, sigma_out, training=training)  # [50,17,64]
        # print("output of  MHA in TB after dropout",mu_output.shape)
        mu_out1 = mu_output + mu_inputs  # [50,17,64]
        # print('output of of MHA before entering to MLP',mu_out1.shape)

        mu_out1_norm, sigma_out1_norm = self.layernorm2(mu_out1, sigma_out1)
        mu_mlp_output, sigma_mlp_output = self.mlp(mu_out1_norm, sigma_out1_norm)
        mu_mlp_output, sigma_mlp_output = self.dropout1(mu_mlp_output, sigma_mlp_output, training=training)
        # print('2nd LN and MLP output',mu_mlp_output.shape)
        mu_output = mu_mlp_output + mu_out1
        with tf.GradientTape() as g:
            g.watch(mu_out1)
            out = mu_mlp_output + mu_out1
        gradi = g.gradient(out, mu_out1)
        sigma_output = tf.math.multiply(tf.math.multiply(gradi, gradi), sigma_out1)
        return mu_output, sigma_output


# Bayesian Fully Connected Layer

class DDense(keras.layers.Layer):
    def __init__(self, units=32):
        '''
        Initialize the instance attributes
        '''
        super(DDense, self).__init__()
        self.units = units

    def build(self, input_shape):
        '''
        Create the state of the layer (weights)
        '''
        w_init = tf.random_normal_initializer()
        self.w = tf.Variable(name='kernel',
                             initial_value=w_init(shape=(input_shape[-1], self.units), dtype='float32'),
                             trainable=True)

        # initialize bias
        b_init = tf.zeros_initializer()
        self.b = tf.Variable(name='bias',
                             initial_value=b_init(shape=(self.units,), dtype='float32'),
                             trainable=True)

    def call(self, inputs):
        '''
        Defines the computation from inputs to outputs
        '''
        return tf.matmul(inputs, self.w) + self.b


class mysoftmax(keras.layers.Layer):
    def __init__(self):
        super(mysoftmax, self).__init__()

    def call(self, mu_in, Sigma_in):
        mu_out = tf.nn.softmax(mu_in)
        print('shape of mu in softmax', mu_out.shape)
        pp1 = tf.expand_dims(mu_out, axis=2)
        # print('shape of pp1',pp1.shape)
        # pp1 = tf.expand_dims(tf.expand_dims(mu_out, axis=2), axis=3)
        pp2 = tf.expand_dims(mu_out, axis=3)
        print('shape of pp2',pp2.shape)
        # a = tf.transpose(mu_out,perm = [0,2,1])

        ppT = tf.matmul(pp1, pp2)
        # ppT = tf.matmul(mu_out, a)
        # ppT = tf.matmul(mu_out,tf.transpose(mu_out, perm=[0, 2, 1]))
        print('shape of ppT', ppT.shape)

        p_diag = tf.linalg.diag(mu_out)
        grad = p_diag - ppT
        # grad = mu_out - ppT
        print('shape of grad', grad.shape)
        print('shape of sigma_in', Sigma_in.shape)

        Sigma_out = tf.squeeze(tf.matmul(grad, tf.expand_dims(Sigma_in, axis=-1)), axis=3)
        print('sigma_out in softmax', Sigma_out.shape)

        # Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        # Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))
        return mu_out, Sigma_out


class mysoftmax_diag(keras.layers.Layer):
    def __init__(self):
        super(mysoftmax_diag, self).__init__()

    def call(self, mu_in, Sigma_in):     #[50,256]
        mu_out = tf.nn.softmax(mu_in) #[50,256]
        print('shape of mu in softmax', mu_out.shape)
        pp1 = tf.expand_dims(mu_out, axis=2) #[50,256, 1]
        # print('shape of pp1',pp1.shape)
        # pp1 = tf.expand_dims(tf.expand_dims(mu_out, axis=2), axis=3)
       # pp2 = tf.expand_dims(mu_out, axis=3)
        #print('shape of pp2',pp2.shape)
        # a = tf.transpose(mu_out,perm = [0,2,1])

        ppT = tf.matmul(pp1, pp1,transpose_b=True ) # [50,256,1]X[50,1,256] =  [50,10,10]
        # ppT = tf.matmul(mu_out, a)
        # ppT = tf.matmul(mu_out,tf.transpose(mu_out, perm=[0, 2, 1]))
        print('shape of ppT', ppT.shape)

        p_diag = tf.linalg.diag(mu_out)  #[50,256, 256]
        grad = tf.math.square(p_diag - ppT)  #[50,256, 256]
        # grad = mu_out - ppT
        print('shape of grad', grad.shape)
        print('shape of sigma_in', Sigma_in.shape)

        Sigma_out = tf.squeeze(tf.matmul(grad, tf.expand_dims(Sigma_in, axis=-1))) #[50,256]
        print('sigma_out in softmax', Sigma_out.shape)

        # Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        # Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))
        return mu_out, Sigma_out


class mysoftmax_1(keras.layers.Layer):
    def __init__(self):
        super(mysoftmax_1, self).__init__()

    def call(self, mu_in, Sigma_in):
        """
        mu_out = tf.nn.softmax(mu_in)
        print('shape of mu in last softmax', mu_out.shape)
        print('shape of sigma in last softmax', Sigma_in.shape)
        pp1 = tf.expand_dims(mu_out, axis=2)
        print('shape of pp1 in last softmax', pp1.shape)
        pp2 = tf.expand_dims(mu_out, axis=1)
        print('shape of pp2 in last softmax', pp2.shape)
        ppT = tf.matmul(pp1, pp2)
        print('shape of ppT in last softmax', ppT.shape)
        p_diag = tf.linalg.diag(mu_out)
        grad = p_diag - ppT
        print('shape of grad in last softmax', grad.shape)
        print(tf.matmul(Sigma_in, tf.transpose(grad, perm=[0, 2, 1])).shape)
        # Sigma_out = tf.matmul(grad, tf.matmul(Sigma_in, tf.transpose(grad, perm=[0, 2, 1])))
        Sigma_out = tf.squeeze(tf.matmul(grad, tf.expand_dims(Sigma_in, axis=2)), axis=2)
        # Sigma_out = tf.matmul(grad,tf.transpose(tf.matmul(Sigma_in, tf.transpose(grad, perm=[0, 2, 1])),perm=[0,2,1]))
        #         Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        #         Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))
        return mu_out, Sigma_out
        """
    # Softmax for mu
        mu_out = tf.nn.softmax(mu_in, axis=-1)
        #Sigma_out = Sigma_in * tf.expand_dims(tf.linalg.diag_part(mu_out), axis=-1)
        Sigma_out = Sigma_in * mu_out
        return mu_out, Sigma_out

        """
       # Outer product of mu_out for Sigma_out
        pp1 = tf.expand_dims(mu_out, axis=-1)
        pp2 = tf.expand_dims(mu_out, axis=-2)
        ppT = pp1 * tf.transpose(pp2, perm=[0, 2, 1])
        p_diag = tf.linalg.diag(mu_out)

        # Ensure non-negativity and avoid NaN/Inf
        Sigma_out = tf.matmul(ppT, tf.matmul(Sigma_in, ppT, transpose_b=True))

        Sigma_out = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))
        """
        Sigma_out = tf.where(tf.math.is_nan(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        Sigma_out = tf.where(tf.math.is_inf(Sigma_out), tf.zeros_like(Sigma_out), Sigma_out)
        return mu_out, Sigma_out

# Bayesian Vision Transformer



class VDP_ViT(tf.keras.Model):
    def __init__(
            self,
            image_size,
            patch_size,
            kernel_size,
            kernel_num,
            kernel_stride,
            pooling_size,
            pooling_stride,
            pooling_pad,
            num_layers,
            num_classes,
            embed_dim,
            var_epsilon,
            num_heads,
            mlp_dim,
            # units,
            channels=3,
            drop_prop=0.1,
            name=None
    ):
        super(VDP_ViT, self).__init__()
        num_patches = (image_size // patch_size) ** 2
        self.patch_dim = channels * (patch_size ** 2)

        self.patch_size = patch_size
        self.embed_dim = embed_dim
        self.num_layers = num_layers
        self.kernel_size = kernel_size
        self.num_heads = num_heads
        self.kernel_num = kernel_num
        self.kernel_stride = kernel_stride
        self.pooling_size = pooling_size
        self.pooling_stride = pooling_stride
        self.pooling_pad = pooling_pad
        self.num_classes = num_classes
        self.var_epsilon = var_epsilon
        self.drop_prop = drop_prop

        # self.units = units

        self.rescale = Rescaling(1.0 / 255)
        # pos_embed, seq_length = self.positional_embedding(image_size)
        # positions = tf.range(start=0, limit=seq_length, delta=1)
        # self.position_embeddings = pos_embed(positions)

        self.class_emb = self.add_weight("class_emb", shape=(1, 1, embed_dim))
        # self.patch_proj = DDense(embed_dim)
        # self.conv=Deterministic_Conv(kernel_size, kernel_num, kernel_stride, padding="VALID")
        self.conv_1 = VDP_first_Conv(kernel_size=self.kernel_size, kernel_num=self.kernel_num,
                                     kernel_stride=self.kernel_stride, padding='VALID')
        self.conv_2 = VDP_intermediate_Conv(kernel_size=self.kernel_size, kernel_num=self.kernel_num,
                                            kernel_stride=self.kernel_stride, padding='SAME')
        self.conv_3 = VDP_intermediate_Conv(kernel_size=self.kernel_size, kernel_num=self.kernel_num,
                                            kernel_stride=self.kernel_stride, padding='SAME')
        self.conv_4 = VDP_intermediate_Conv(kernel_size=self.kernel_size, kernel_num=self.kernel_num,
                                            kernel_stride=self.kernel_stride, padding='SAME')
        self.conv_5 = VDP_intermediate_Conv(kernel_size=self.kernel_size, kernel_num=self.kernel_num,
                                            kernel_stride=self.kernel_stride, padding='SAME')
        self.conv_6 = VDP_intermediate_Conv(kernel_size=self.kernel_size, kernel_num=self.kernel_num,
                                            kernel_stride=self.kernel_stride, padding='SAME')
        self.relu = VDP_ReLU()
        # self.maxpooling_1 = DMaxPooling(pooling_size, pooling_stride, pooling_pad)
        self.maxpooling_11 = VDP_MaxPooling(pooling_size=self.pooling_size, pooling_stride=self.pooling_stride,
                                            pooling_pad=self.pooling_pad)
        self.dropout_1 = VDP_Dropout(self.drop_prop)
        self.batch_norm = VDPBatch_Normalization(self.var_epsilon)
        self.class_emb = self.add_weight("class_emb", shape=(1, 1, embed_dim))
        self.patch_proj = DDense(embed_dim)
        # self.enc_layers = VDP_TransformerBlock_first(d_model, num_heads, mlp_dim, dropout)  # for _ in range(num_layers)]
        # self.enc_layers =  [VDP_TransformerBlock(d_model, num_heads, mlp_dim, dropout)
        # for _ in range(num_layers) ]

        self.enc_layers1 = VDP_TransformerBlock_first(embed_dim, num_heads, mlp_dim, drop_prop)
        self.enc_layers = [
            VDP_TransformerBlock_Intermediate(embed_dim, num_heads, mlp_dim, drop_prop)
            for _ in range(num_layers)
        ]
        self.layernorm1 = Bayesian_LayerNorm(eps=1e-6)
        self.mysoftma = mysoftmax()
        self.mysoftm = mysoftmax_diag()
        # self.fc_1 = LinearNotFirst(self.units)
        # self.fc_1 = LinearNotFirst(units=self.kernel_num)
        self.fc_1 = LinearNotFirst(units=1)
        self.fc_2 = LinearNotFirst(units=self.num_classes)
        # self.mlp_head = VDP_MLP(mlp_dim, num_classes)
        # self.mlp_head = VDP_MLP(mlp_dim, num_classes)

    def call(self, x, training):
        print('Input dimension :', x.shape)
        batch_size = tf.shape(x)[0]

        print('shape of x before conv', x.shape)
        # x = self.rescale(x)
        # patches = self.extract_patches(x)
        # x = self.patch_proj(patches)
        mu, sigma = self.conv_1(x)
        # print('shape of x after conv', x.shape)

        mu, sigma = self.relu(mu, sigma)
        mu, sigma = self.batch_norm(mu, sigma)
        # x=self.maxpooling_11(x)
        # print('shape of x after maxpool',x.shape)

        # x=tf.reshape(x, [batch_size, -1, self.kernel_num])

        mu, sigma = self.conv_2(mu, sigma)
        print('shape of x after conv', mu.shape)

        mu, sigma = self.relu(mu, sigma)
        mu, sigma = self.batch_norm(mu, sigma)

        mu, sigma = self.conv_3(mu, sigma)
        mu, sigma = self.relu(mu, sigma)
        mu, sigma = self.batch_norm(mu, sigma)
        mu, sigma = self.maxpooling_11(mu, sigma)
        mu, sigma = self.dropout_1(mu, sigma, Training=training)
        print('shape of x after maxpool', mu.shape)

        mu, sigma = self.conv_4(mu, sigma)
        mu, sigma = self.relu(mu, sigma)
        mu, sigma = self.batch_norm(mu, sigma)
    mu, sigma = self.conv_5(mu, sigma)
    mu, sigma = self.relu(mu, sigma)
    mu, sigma = self.batch_norm(mu, sigma)
    mu, sigma = self.maxpooling_11(mu, sigma)
    mu, sigma = self.dropout_1(mu, sigma, Training=training)

    mu, sigma = self.conv_6(mu, sigma)
    mu, sigma = self.relu(mu, sigma)
    mu, sigma = self.batch_norm(mu, sigma)
    mu, sigma = self.maxpooling_11(mu, sigma)
    mu, sigma = self.dropout_1(mu, sigma, Training=training)

    mu = tf.reshape(mu, [batch_size, -1, self.kernel_num])
    sigma = tf.reshape(sigma, [batch_size, -1, self.kernel_num])
    print('shape of x after reshape', mu.shape)

    mu, sigma = self.enc_layers1(mu, sigma)

    for layer in self.enc_layers:
        mu, sigma = layer(mu, sigma, training)

    # Bayesian Sequence Pooling

    mu_1, sigma_1 = self.layernorm1(mu, sigma)
    print('shape of mu after LN in seq pool', mu_1.shape)
    mu, sigma = self.fc_1(mu_1, sigma_1)    #[50,49,1]
    print('shape of mu in fc', mu.shape)
    # x1=layers.Dense(1)(x)
    # print(x1.shape)

    mu_weights = tf.nn.softmax(mu, axis =1 )  # [50,49,1]
    # Sigma for softmax function


#    pp1 = tf.expand_dims(mu_weights, axis=-1)  # [50, 49,1,1]
 #   pp2 = tf.expand_dims(mu_weights, axis=3)  # [50, 49,1,1]
    ppT = tf.matmul(mu_weights, mu_weights,transpose_b=True )  # #[50,49,49]
    print('shape of ppT',ppT.shape)
    p_diag = tf.linalg.diag(tf.squeeze(mu_weights) ) # [50,49,49]
    print( 'shape of p_diag',p_diag.shape)
    grad = tf.math.square(p_diag - ppT)  # # [50,49,49]
    print('shape of grad',grad.shape)
    Sigma_weights = tf.matmul(grad, sigma) #[50,49,49]X[50,49,1]=[50,49,1]
    print('sigma_out in softmax', Sigma_weights.shape) #[50,49,1]
    #Sigma_weights = tf.squeeze(tf.matmul(grad, tf.expand_dims(sigma_scaled_score, axis=-1)))
    Sigma_weights = tf.where(tf.math.is_nan(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
    Sigma_weights= tf.where(tf.math.is_inf(Sigma_weights), tf.zeros_like(Sigma_weights), Sigma_weights)
    # Sigma_weights = tf.linalg.set_diag(Sigma_out, tf.abs(tf.linalg.diag_part(Sigma_out)))


    #mu_output = tf.matmul(mu_weights, mu_1,transpose_a=True) [50,1,49] X [ 50,49,256] = [50,1,256]




    #mu_xbar, sigma_xbar = self.mysoftma(mu,sigma)
```

```
        #print('shape of mu_xbar',mu_xbar.shape) #[50,225,1]
        #print('shape of sigma_xbar',sigma_xbar.shape) #[50,225,1]
        mu_score = tf.matmul(mu_weights, mu_1, transpose_a=True)  # [50,1,49] X [50,49,256] = [50,1,256]
        print('shape of mu_score',mu_score.shape)#[50,1,1]
        a = tf.matmul(Sigma_weights, mu_1 ** 2, transpose_a=True)  # [50, 1, 256]
        print('shape of a in seq pool', a.shape) #[50,1,128]
        b = tf.matmul(mu_weights ** 2, sigma_1, transpose_a=True)  # [50,1,49]  X [50,49,256] = [50,1,256]
        print('shape of b in seq pool', b.shape)
        c = tf.matmul(Sigma_weights, sigma_1, transpose_a=True)  ## [50,1,49] X [50,49,256] = [50,1,256]
        sigma_score = a + b + c  # [16, 1, 128]/ [50,1,256]

        #mu_output = tf.matmul(mu_weights, mu_value)  # [50,2,17,17] X [50,2,17,32]=  [50,2,17,32]
        # print('mu output',mu_output.shape)
        #d = tf.matmul(mu_weights ** 2, sigma_value)  # [50,2,17,32]
        #e = tf.matmul(Sigma_weights, mu_value ** 2)  # [50,2,17,32]
        #f = tf.matmul(Sigma_weights, sigma_value)  # [50, 2, 17, 17]x[50,2,17,32]=  [50,2,17,32]
        #output_sigma = d + e + f




        # mu= mu_score
        # sigma= sigma_score
        mu = tf.squeeze(mu_score, -2)
        print('shape of mu after squeeze seq pool', mu.shape)
        sigma = tf.squeeze(sigma_score, -2)
        print('shape of sigma after seq pool', sigma.shape)
        # mu,sigma

        # Final Classification

        mu, sigma = self.fc_2(mu, sigma)  #[50,10]
        print('shape of mu in fc during final classification', mu.shape)
        mu_out, sigma_out = self.mysoftm(mu, sigma)
        print('shape of mu after last softmax',mu.shape)
        #mu_out = tf.squeeze(mu, -2)
        #sigma_out = tf.squeeze(sigma, -2)

        #print('shape of mu during final classification after passing through dense and softmax', mu_out.shape)
        #print('shape of sigma during final classification', sigma_out.shape)
        sigma_out = tf.where(tf.math.is_nan(sigma_out), tf.zeros_like(sigma_out), sigma_out)
        sigma_out = tf.where(tf.math.is_inf(sigma_out), tf.zeros_like(sigma_out), sigma_out)

        return mu_out, sigma_out

        # First (class token) is used for classification
        # mu, sigma = self.mlp_head(mu_out[:, 0], sigma_out[:, 0])
        # print('shape of mu', mu.shape)
        # print('shape of sigma',sigma.shape)
        # return mu_out, sigma


# Loss Function(Modified)

def nll_gaussian(y_test, y_pred_mean, y_pred_sd):
    mu = y_test - y_pred_mean
    mu_2 = mu ** 2
    y_pred_sd = y_pred_sd + 1e-5
    s = tf.math.divide_no_nan(1., y_pred_sd)
    loss1 = tf.math.reduce_mean(tf.math.reduce_sum(tf.math.multiply(mu_2, s), axis=-1))
    loss2 = tf.math.reduce_mean(tf.math.reduce_sum(tf.math.log(y_pred_sd), axis=-1))
    loss = tf.math.reduce_mean(tf.math.add(loss1, loss2))
    loss = tf.where(tf.math.is_nan(loss), tf.zeros_like(loss), loss)
    loss = tf.where(tf.math.is_inf(loss), tf.zeros_like(loss), loss)
    return loss


# convert images to float32 format and convert labels to int32
def preprocess(image, label):
    image = tf.image.convert_image_dtype(image, tf.float32)
    # label = tf.cast(label, tf.int32)
    label = tf.cast(label, tf.float32)
    return image, label


# Peform augmentations on training data
def augmentation(image, label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_brightness(image, max_delta=0.5)  # Random brightness
    return image, label


# Main Function

# def main_function(input_dim=28, num_kernels=[32], kernels_size=[5], maxpooling_size=[2], maxpooling_stride=[2], maxpooling_pad='SAME', class_num=10 ,
#          epochs =20, lr=0.001, lr_end = 0.0001, kl_factor = 0.01,
#          Random_noise=True, gaussain_noise_std=0.5, Adversarial_noise=False, epsilon = 0, adversary_target_cls=3, Targeted=False,
#          Training = False, continue_training = False,  saved_model_epochs=50):

def main_function(image_size=32, patch_size=8, num_layers=2, num_classes=10, embed_dim=128, num_heads=4, mlp_dim=128,
                  channels=3, drop_prop=0.1, batch_size=50, epochs=435, lr=0.001, lr_end=0.0001, kl_factor=0.001,
                  kernel_size=5, kernel_num=128, pooling_size=2, pooling_stride=2, kernel_stride=1, pooling_pad='VALID',
                  Targeted=False, Random_noise=False, gaussain_noise_std=0.5, epsilon=0.5, Training=False, Testing=True,
                  Adversarial_noise=False, HCV=0.5, adversary_target_cls=3, PGD_Adversarial_noise=True, stepSize=1,
                  maxAdvStep=20, continue_training=False, saved_model_epochs=30):
    PATH = './VDP_cnn_epoch_{}/'.format(epochs)

    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

    x_train, x_test = x_train / 255.0, x_test / 255.0
    x_train = x_train.astype("float32")
    x_test = x_test.astype("float32")

    one_hot_y_train = tf.one_hot(np.squeeze(y_train).astype(np.float32), depth=num_classes)
    one_hot_y_test = tf.one_hot(np.squeeze(y_test).astype(np.float32), depth=num_classes)

    # x_train = tf.image.resize(x_train, [64, 64])  # resizing image shape to 64 X 64
    # print('shape after resizing image',x_train.shape)
    # tr_dataset = tf.data.Dataset.from_tensor_slices((x_train, one_hot_y_train))
    # val_dataset = tf.data.Dataset.from_tensor_slices((x_test, one_hot_y_test))

    tr_dataset = tf.data.Dataset.from_tensor_slices((x_train, one_hot_y_train)).batch(batch_size)
```

```
val_dataset = tf.data.Dataset.from_tensor_slices((x_test, one_hot_y_test)).batch(batch_size)

# x_test = tf.image.resize(x_test, [64, 64])

# tr_dataset = tf.data.Dataset.from_tensor_slices((x_train, one_hot_y_train))

# val_dataset = tf.data.Dataset.from_tensor_slices((x_test, one_hot_y_test))

# tr_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
# val_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))

AUTO = tf.data.AUTOTUNE
# applying transformations
tr_dataset = tr_dataset.shuffle(1024)  # shuffle the images
tr_dataset = tr_dataset.map(preprocess, num_parallel_calls=AUTO)  # mapping our preprocess function to train_data
tr_dataset = tr_dataset.map(augmentation, num_parallel_calls=AUTO)  # mapping our augmentation funtion to train_data
dataset_size = tf.data.experimental.cardinality(tr_dataset).numpy()
print("Size of tr_dataset:", dataset_size)

#tr_dataset = tr_dataset.batch(batch_size)  # Converting train_data to batches
tr_dataset = tr_dataset.prefetch(
    AUTO)  # using prefetch which prepares subsequent batches of data while other batches are being computed.
dataset_size = tf.data.experimental.cardinality(tr_dataset).numpy()
print("Size of tr_dataset:", dataset_size)
# val_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
# applying transformations
val_dataset = val_dataset.map(preprocess, num_parallel_calls=AUTO)  # mapping our preprocess function test_data
#val_dataset = val_dataset.batch(batch_size)
val_dataset = val_dataset.prefetch(
    AUTO)  # using prefetch which prepares subsequent batches of data while other batches are being computed.



trans_model = VDP_ViT(image_size=image_size, patch_size=patch_size, num_layers=num_layers, num_classes=num_classes,
                      embed_dim=embed_dim,
                      num_heads=num_heads, mlp_dim=mlp_dim, kernel_size=kernel_size, kernel_num=kernel_num,
                      kernel_stride=kernel_stride, pooling_size=pooling_size,
                      pooling_stride=pooling_stride, pooling_pad=pooling_pad, var_epsilon=1e-4, channels=channels,
                      drop_prop=drop_prop, name='vdp_trans')

num_train_steps = epochs * int(x_train.shape[0] / batch_size)
#     step = min(step, decay_steps)
#     ((initial_learning_rate - end_learning_rate) * (1 - step / decay_steps) ^ (power) ) + end_learning_rate

learning_rate_fn = tf.keras.optimizers.schedules.PolynomialDecay(initial_learning_rate=lr,
                                                                 decay_steps=num_train_steps,
                                                                 end_learning_rate=lr_end, power=3.)
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate_fn)  # , clipnorm=1.0)

@tf.function  # Make it fast.
def train_on_batch(x, y):
    with tf.GradientTape() as tape:
        mu_out, sigma = trans_model(x, training=True)
        print("shape of mu_out", mu_out.shape)
        trans_model.trainable = True
        trans_model.summary()
        print(' y in train on batch', y)
        print(' mu_out in train on batch', mu_out)
        # y = tf.cast(y, tf.float32)
        print('y after converting to float', y)
        loss_final = nll_gaussian(y, mu_out, tf.clip_by_value(t=sigma, clip_value_min=tf.constant(1e-3),
                                                           clip_value_max=tf.constant(1e+8)))

        regularization_loss = tf.math.add_n(trans_model.losses)
        loss = 0.5 * (loss_final + kl_factor * regularization_loss)
        print(loss)
        print(mu_out)
        print(sigma)
        print(loss_final)
        print(regularization_loss)
    gradients = tape.gradient(loss, trans_model.trainable_weights)
    gradients = [grad if grad is not None else tf.zeros_like(var) for grad, var in
                 zip(gradients, trans_model.trainable_weights)]
    # if len(gradients) != len(trans_model.trainable_weights):
    #   print("Some gradients are None.")
    # else:
    gradients = [(tf.where(tf.math.is_nan(grad), tf.constant(1.0e-5, shape=grad.shape), grad)) for grad in
                 gradients]
    gradients = [(tf.where(tf.math.is_inf(grad), tf.constant(1.0e-5, shape=grad.shape), grad)) for grad in
                 gradients]

    # Handle None gradients

    optimizer.apply_gradients(zip(gradients, trans_model.trainable_weights))
    print('sjshshshs')
    print(gradients)
    return loss, mu_out, sigma, gradients, regularization_loss, loss_final

@tf.function
def validation_on_batch(x, y):
    mu_out, sigma = trans_model(x, training=False)
    # cnn_model.trainable = False
    vloss = nll_gaussian(y, mu_out, tf.clip_by_value(t=sigma, clip_value_min=tf.constant(1e-3),
                                                       clip_value_max=tf.constant(1e+8)))
    regularization_loss = tf.math.add_n(trans_model.losses)
    total_vloss = 0.5 * (vloss + kl_factor * regularization_loss)
    return total_vloss, mu_out, sigma

@tf.function
def test_on_batch(x, y):
    trans_model.trainable = False
    mu_out, sigma = trans_model(x, training=False)
    return mu_out, sigma

@tf.function
def create_adversarial_pattern(input_image, input_label):
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        trans_model.trainable = False
        prediction, sigma = trans_model(input_image)
        loss_final = nll_gaussian(input_label, prediction,
                                  tf.clip_by_value(t=sigma, clip_value_min=tf.constant(1e-4),
                                                   clip_value_max=tf.constant(1e+3)))
        # clip_value_max=tf.constant(1e+3)), num_classes, batch_size)

        loss = 0.5 * loss_final
        # Get the gradients of the loss w.r.t to the input image.
    gradient = tape.gradient(loss, input_image)
    # Get the sign of the gradients to create the perturbation
    signed_grad = tf.sign(gradient)
    return signed_grad

wandb.init(entity="fazlur7512",
           project="VDP_cct_cifar10_epochs_{}_layer_{}_lr_{}_kl_factor_{}_batch_size_{}_dimension_{}_patch_size_{}_head_{}_input_{}".format(
               epochs, num_layers, lr, kl_factor, batch_size, embed_dim, patch_size, num_heads, image_size))

if Training:
    wandb.init(entity="fazlur7512",
               project="VDP_cct_cifar10_epochs_{}_layer_{}_lr_{}_kl_factor_{}_batch_size_{}_dimension_{}_patch_size_{}_head_{}_input_{}".format(
                   epochs, num_layers, lr, kl_factor, batch_size, embed_dim, patch_size, num_heads, image_size))
    if continue_training:
        saved_model_path = './saved_models/VDP_cnn_epoch_{}/'.format(saved_model_epochs)
        trans_model.load_weights(saved_model_path + 'vdp_cnn_model')
    train_acc = np.zeros(epochs)
    valid_acc = np.zeros(epochs)
    train_err = np.zeros(epochs)
    valid_error = np.zeros(epochs)

    start = timeit.default_timer()
    for epoch in range(epochs):
        print('Epoch: ', epoch + 1, '/', epochs)
        acc1 = 0
        acc_valid1 = 0
        err1 = 0
        err_valid1 = 0
        tr_no_steps = 0
        va_no_steps = 0
        # -------------Training--------------------
        acc_training = np.zeros(int(x_train.shape[0] / (batch_size)))
        err_training = np.zeros(int(x_train.shape[0] / (batch_size)))
        for step, (x, y) in enumerate(tr_dataset):
            update_progress(step / int(x_train.shape[0] / (batch_size)))
            #  print(y.shape)
            loss, mu_out, sigma, gradients, regularization_loss, loss_final = train_on_batch(x, y)
            print('mu_out shape in train on batch', mu_out.shape)
            err1 += loss.numpy()
            corr = tf.equal(tf.math.argmax(mu_out, axis=1), tf.math.argmax(y, axis=1))
            # corr = tf.equal(mu_out,y)
            print('i am here', corr)
            accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
            acc1 += accuracy.numpy()
            if step % 100 == 0:
                print('\n gradient', np.mean(gradients[0].numpy()))
                #   print('\n Matrix Norm', np.mean(sigma))
                print("\n Step:", step, "Loss:", float(err1 / (tr_no_steps + 1.)))
                print("Total Training accuracy so far: %.3f" % float(acc1 / (tr_no_steps + 1.)))
            tr_no_steps += 1
            wandb.log({"Average Variance value": tf.reduce_mean(sigma).numpy(),
                       "Total Training Loss": loss.numpy(),
                "Training Accuracy per minibatch": accuracy.numpy(),
                "gradient per minibatch": np.mean(gradients[0]),
                'epoch': epoch,
                "Regularization_loss": regularization_loss.numpy(),
                "Log-Likelihood Loss": np.mean(loss_final.numpy())
                })
    train_acc[epoch] = acc1 / tr_no_steps
    train_err[epoch] = err1 / tr_no_steps
    print('Training Acc  ', train_acc[epoch])
    print('Training error  ', train_err[epoch])
    # ---------------Validation----------------------
    for step, (x, y) in enumerate(val_dataset):
        update_progress(step / int(x_test.shape[0] / (batch_size)))
        total_vloss, mu_out, sigma = validation_on_batch(x, y)
        err_valid1 += total_vloss.numpy()
        corr = tf.equal(tf.math.argmax(mu_out, axis=-1), tf.math.argmax(y, axis=-1))
        va_accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
        acc_valid1 += va_accuracy.numpy()

        if step % 50 == 0:
            print("Step:", step, "Loss:", float(total_vloss))
            print("Total validation accuracy so far: %.3f" % va_accuracy)
        va_no_steps += 1
    # wandb.log({"Average Variance value (validation Set)": tf.reduce_mean(sigma).numpy(),
    #            "Total Validation Loss": total_vloss.numpy(),
    #            "Validation Acuracy per minibatch": va_accuracy.numpy()
    #            })
    valid_acc[epoch] = acc_valid1 / va_no_steps
    valid_error[epoch] = err_valid1 / va_no_steps
    stop = timeit.default_timer()
    trans_model.save_weights(PATH + 'vdp_transfm_model')
    wandb.log({"Average Training Loss": train_err[epoch],
               "Average Training Accuracy": train_acc[epoch],
               # "Average Validation Loss": valid_error[epoch],
               "Average Validation Accuracy": valid_acc[epoch],
               'epoch': epoch
               })

    # wandb.log({"Average Training Loss": train_err[epoch],
    #            "Average Training Accuracy": train_acc[epoch],
    #           'epoch': epoch
    #          })

    print('Total Training Time: ', stop - start)
    print('Training Acc  ', train_acc[epoch])
    print('Validation Acc  ', valid_acc[epoch])
    print('------------------------------------')
    print('Training error  ', train_err[epoch])
    print('Validation error  ', valid_error[epoch])
    # -----------------End Training--------------------------
trans_model.save_weights(PATH + 'vdp_cnn_model')

if (epochs > 1):
    fig = plt.figure(figsize=(15, 7))
    plt.plot(train_acc, 'b', label='Training acc')
    plt.plot(valid_acc, 'r', label='Validation acc')
```

```
        plt.ylim(0, 1.1)
        plt.title("Density Propagation Trans on Fashion MNIST Data")
        plt.xlabel("Epochs")
        plt.ylabel("Accuracy")
        plt.legend(loc='lower right')
        plt.savefig(PATH + 'VDP_Trans_on_Fashion_MNIST_Data_acc.png')
        plt.close(fig)

        fig = plt.figure(figsize=(15, 7))
        plt.plot(train_err, 'b', label='Training error')
        plt.plot(valid_error, 'r', label='Validation error')
        plt.title("Density Propagation Trans on Fashion MNIST Data")
        plt.xlabel("Epochs")
        plt.ylabel("Error")
        plt.legend(loc='upper right')
        plt.savefig(PATH + 'VDP_Trans_on_FMNIST_Data_error.png')
        plt.close(fig)

    f = open(PATH + 'training_validation_acc_error.pkl', 'wb')
    pickle.dump([train_acc, valid_acc, train_err, valid_error], f)
    f.close()

    textfile = open(PATH + 'Related_hyperparameters.txt', 'w')
    textfile.write(' Input Dimension : ' + str(image_size))
    textfile.write('\n Hidden units : ' + str(mlp_dim))
    textfile.write('\n Number of Classes : ' + str(num_classes))
    textfile.write('\n No of epochs : ' + str(epochs))
    textfile.write('\n Initial Learning rate : ' + str(lr))
    textfile.write('\n Ending Learning rate : ' + str(lr_end))
    #  textfile.write('\n kernels Size : ' +str(kernels_size))
    #  textfile.write('\n Max pooling Size : ' +str(maxpooling_size))
    #  textfile.write('\n Max pooling stride : ' +str(maxpooling_stride))
    textfile.write('\n batch size : ' + str(batch_size))
    textfile.write('\n KL term factor : ' + str(kl_factor))
    textfile.write("\n---------------------------------")
    if Training:
        textfile.write('\n Total run time in sec : ' + str(stop - start))
        if (epochs == 1):
            textfile.write("\n Averaged Training  Accuracy : " + str(train_acc))
            textfile.write("\n Averaged Validation Accuracy : " + str(valid_acc))

            textfile.write("\n Averaged Training  error : " + str(train_err))
            textfile.write("\n Averaged Validation error : " + str(valid_error))
        else:
            textfile.write("\n Averaged Training  Accuracy : " + str(np.mean(train_acc[epoch])))
            textfile.write("\n Averaged Validation Accuracy : " + str(np.mean(valid_acc[epoch])))

            textfile.write("\n Averaged Training  error : " + str(np.mean(train_err[epoch])))
            textfile.write("\n Averaged Validation error : " + str(np.mean(valid_error[epoch])))
    textfile.write("\n---------------------------------")
    textfile.write("\n---------------------------------")
    textfile.close()

# if (Testing):
#   test_path = 'test_results/'
# if Random_noise:
```

```
# test_path = 'test_random_noise_{}/'.format(gaussain_noise_std)
#  os.makedirs(PATH + test_path)
# trans_model.load_weights(PATH + 'vdp_cnn_model')

if Testing:
    test_path = 'test_results/'
    if Random_noise:
        test_path = 'test_results_random_noise_{}/'.format(gaussain_noise_std)
    full_test_path = PATH + test_path
    if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
        shutil.rmtree(full_test_path)
    os.makedirs(PATH + test_path)

    trans_model.load_weights(PATH + 'vdp_cnn_model')

    test_no_steps = 0
    true_x = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, 3])
    true_y = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    mu_out_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    # sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes,num_classes])
    acc_test = np.zeros(int(x_test.shape[0] / (batch_size)))
    for step, (x, y) in enumerate(val_dataset):
        update_progress(step / int(x_test.shape[0] / (batch_size)))
        true_x[test_no_steps, :, :, :, :] = x
        true_y[test_no_steps, :, :] = y
        if Random_noise:
            noise = tf.random.normal(shape=[batch_size, image_size, image_size, 1], mean=0.0,
                                     stddev=gaussain_noise_std, dtype=x.dtype)
            x = x + noise
        mu_out, sigma = test_on_batch(x, y)
        mu_out_[test_no_steps, :, :] = mu_out
        # sigma_[test_no_steps, :, :,:]= sigma
        sigma_[test_no_steps, :, :] = sigma
        corr = tf.equal(tf.math.argmax(mu_out, axis=1), tf.math.argmax(y, axis=1))
        accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
        acc_test[test_no_steps] = accuracy.numpy()
        if step % 100 == 0:
            print("Total running accuracy so far: %.3f" % acc_test[test_no_steps])
        test_no_steps += 1
        # New added line
        wandb.log({"Testing Accuracy per minibatch": accuracy.numpy()
                   })

    test_acc = np.mean(acc_test)
    print('Test accuracy : ', test_acc)
    # print("Best Test Accuracy :", np.amax(acc_test))
    # New added line
    wandb.log({"Testing Accuracy": (test_acc)})
    pf = open(PATH + test_path + 'uncertainty_info.pkl', 'wb')
    pickle.dump([mu_out_, sigma_, true_x, true_y, test_acc], pf)
    pf.close()

    var = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size])
```

```
if Random_noise:
    snr_signal = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size])
    for i in range(int(x_test.shape[0] / (batch_size))):
        for j in range(batch_size):
            noise = tf.random.normal(shape=[image_size, image_size, 1], mean=0.0, stddev=gaussain_noise_std,
                                     dtype=x.dtype)
            snr_signal[i, j] = 10 * np.log10(
                np.sum(np.square(true_x[i, j, :, :, :])) / np.sum(np.square(noise)))

        predicted_out = np.argmax(mu_out_[i, j, :])
        var[i, j] = sigma_[i, j, int(predicted_out)]
    print('SNR', np.mean(snr_signal))

# sigma_1 = np.reshape(sigma_, int(x_test.shape[0] / (batch_size)) , batch_size)
# var = np.zeros([int(test_X.shape[0] / (batch_size)) , batch_size])
# for i in range(int(test_X.shape[0] / (batch_size)), batch_size):
# for i in range(int(test_X.shape[0] / (batch_size))):
# s = np.abs(sigma_1[i])
# if (i != 0):
#   if (np.abs(s) > 10000):
#        var[i] = 0.0  # np.abs(sigma_1[i-1])
#     else:
#          var[i] = s
#   else:
#        var[i] = s
# data_mean, data_std = np.mean(np.abs(sigma_1)), np.std(np.abs(sigma_1))
# identify outliers
# cut_off = data_std * 3
# lower, upper = data_mean - cut_off, data_mean + cut_off
# outliers = [x for x in np.abs(sigma_1) if x < lower or x > upper]
# outliers_removed = [x for x in np.abs(sigma_1) if x > lower and x < upper]
# print('outliers_removed', np.mean(outliers_removed))
# writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
# df = pd.DataFrame(np.abs(sigma_1))
# Write your DataFrame to a file
# df.to_excel(writer, "Sheet")
# writer.save()
# print('Output Variance without outlier', np.mean(np.abs(var)))
# print('Output Variance', np.mean(np.abs(sigma_)))

valid_size = x_test.shape[0]
pred_var = np.zeros(int(valid_size))
true_var = np.zeros(int(valid_size))
correct_classification = np.zeros(int(valid_size))
misclassification_pred = np.zeros(int(valid_size))
misclassification_true = np.zeros(int(valid_size))
predicted_out = np.zeros(int(valid_size))
true_out = np.zeros(int(valid_size))
k = 0
k1 = 0
k2 = 0
for i in range(int(valid_size / batch_size)):
    for j in range(batch_size):
        predicted_out[k] = np.argmax(mu_out_[i, j, :])
        true_out[k] = np.argmax(true_y[i, j, :])
        pred_var[k] = sigma_[i, j, int(predicted_out[k])]
        true_var[k] = sigma_[i, j, int(true_out[k])]
        if (predicted_out[k] == true_out[k]):
            correct_classification[k1] = sigma_[i, j, int(predicted_out[k])]
            k1 = k1 + 1
        if (predicted_out[k] != true_out[k]):
            misclassification_pred[k2] = sigma_[i, j, int(predicted_out[k])]
            misclassification_true[k2] = sigma_[i, j, int(true_out[k])]
            k2 = k2 + 1
        k = k + 1
print('Average Output Variance', np.mean(pred_var))

var1 = pred_var  # np.reshape(var, int(x_test.shape[0]/(batch_size))* batch_size)
writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
df = pd.DataFrame(np.abs(var1))
# Write your DataFrame to a file
df.to_excel(writer, "Sheet")

df1 = pd.DataFrame(predicted_out)
df1.to_excel(writer, 'Sheet', startcol=4)

df2 = pd.DataFrame(true_out)
df2.to_excel(writer, 'Sheet', startcol=7)

df3 = pd.DataFrame(correct_classification)
df3.to_excel(writer, 'Sheet', startcol=10)

df4 = pd.DataFrame(misclassification_pred)
df4.to_excel(writer, 'Sheet', startcol=13)

df5 = pd.DataFrame(misclassification_true)
df5.to_excel(writer, 'Sheet', startcol=16)
writer.save()

pf = open(PATH + test_path + 'var_info.pkl', 'wb')
pickle.dump([correct_classification, misclassification_true, pred_var], pf)
# if Random_noise:

textfile = open(PATH + test_path + 'Related_hyperparameters.txt', 'w')
textfile.write(' Input Dimension : ' + str(image_size))
# textfile.write('\n No of Kernels : ' +str(num_kernels))
textfile.write('\n Number of Classes : ' + str(num_classes))
textfile.write('\n No of epochs : ' + str(epochs))
textfile.write('\n Initial Learning rate : ' + str(lr))
textfile.write('\n Ending Learning rate : ' + str(lr_end))
# textfile.write('\n kernels Size : ' +str(kernels_size))
# textfile.write('\n Max pooling Size : ' +str(maxpooling_size))
# textfile.write('\n Max pooling stride : ' +str(maxpooling_stride))
textfile.write('\n batch size : ' + str(batch_size))
textfile.write('\n KL term factor : ' + str(kl_factor))
textfile.write("\n---------------------------------")
textfile.write("\n Test Accuracy : " + str(test_acc))
textfile.write("\n Output Variance: " + str(np.mean(np.abs(var))))
textfile.write("\n Correct Classification Variance: " + str(np.mean(correct_classification)))
textfile.write("\n MisClassification Variance: " + str(np.mean(misclassification_pred)))

textfile.write("\n---------------------------------")
```

```
    if Random_noise:
        textfile.write('\n Random Noise std: ' + str(gaussain_noise_std))
        # textfile.write('\n Random Noise HCV: ' + str(HCV))
        textfile.write("\n SNR: " + str(np.mean(snr_signal)))
    textfile.write("\n---------------------------------")
    textfile.close()

    # if (Adversarial_noise):
    # elif(Adversarial_noise):

if (Adversarial_noise):
    if Targeted:
        test_path = 'test_results_targeted_adversarial_noise_{}/'.format(epsilon)
        full_test_path = PATH + test_path
        if os.path.exists(full_test_path):
            # Remove the existing test path and its contents
            shutil.rmtree(full_test_path)
        os.makedirs(PATH + test_path)
    else:
        test_path = 'test_results_non_targeted_adversarial_noise_{}/'.format(epsilon)
        full_test_path = PATH + test_path
        if os.path.exists(full_test_path):
            # Remove the existing test path and its contents
            shutil.rmtree(full_test_path)
        os.makedirs(PATH + test_path)
    trans_model.load_weights(PATH + 'vdp_cnn_model')
    test_no_steps = 0

    true_x = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, 3])
    adv_perturbations = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, 3])
    true_y = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    mu_out_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
    # sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, class_num, class_num])
    sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])

    acc_test = np.zeros(int(x_test.shape[0] / (batch_size)))
    for step, (x, y) in enumerate(val_dataset):
        update_progress(step / int(x_test.shape[0] / (batch_size)))
        true_x[test_no_steps, :, :, :, :] = x
        true_y[test_no_steps, :, :] = y

        if Targeted:
            y_true_batch = np.zeros_like(y)
            y_true_batch[:, adversary_target_cls] = 1.0
            adv_perturbations[test_no_steps, :, :, :, :] = create_adversarial_pattern(x, y_true_batch)
        else:
            adv_perturbations[test_no_steps, :, :, :, :] = create_adversarial_pattern(x, y)
        adv_x = x + epsilon * adv_perturbations[test_no_steps, :, :, :, :]
        adv_x = tf.clip_by_value(adv_x, 0.0, 1.0)

        mu_out, sigma = test_on_batch(adv_x, y)
        mu_out_[test_no_steps, :, :] = mu_out
        sigma_[test_no_steps, :, :] = sigma
        # sigma_[test_no_steps, :, :, :] = sigma
        corr = tf.equal(tf.math.argmax(mu_out, axis=1), tf.math.argmax(y, axis=1))
        accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
    acc_test[test_no_steps] = accuracy.numpy()
    if step % 10 == 0:
        print("Total running accuracy so far: %.3f" % accuracy.numpy())
    test_no_steps += 1

test_acc = np.mean(acc_test)
print('Test accuracy : ', test_acc)

pf = open(PATH + test_path + 'uncertainty_info.pkl', 'wb')
pickle.dump([mu_out_, sigma_, adv_perturbations, test_acc], pf)
pf.close()

var = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
snr_signal = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
for i in range(int(x_test.shape[0] / batch_size)):
    for j in range(batch_size):
        predicted_out = np.argmax(mu_out_[i, j, :])
        var[i, j] = sigma_[i, j, int(predicted_out)]
        snr_signal[i, j] = 10 * np.log10(np.sum(np.square(true_x[i, j, :, :, :])) / np.sum(
            np.square(epsilon * adv_perturbations[i, j, :, :, :])))

print('Output Variance', np.mean(var))
print('SNR', np.mean(snr_signal))
valid_size = x_test.shape[0]
pred_var = np.zeros(int(valid_size))
true_var = np.zeros(int(valid_size))
correct_classification = np.zeros(int(valid_size))
misclassification_pred = np.zeros(int(valid_size))
misclassification_true = np.zeros(int(valid_size))
predicted_out = np.zeros(int(valid_size))
true_out = np.zeros(int(valid_size))
k = 0
k1 = 0
k2 = 0
for i in range(int(valid_size / batch_size)):
    for j in range(batch_size):
        predicted_out[k] = np.argmax(mu_out_[i, j, :])
        true_out[k] = np.argmax(true_y[i, j, :])
        pred_var[k] = sigma_[i, j, int(predicted_out[k])]
        true_var[k] = sigma_[i, j, int(true_out[k])]
        if (predicted_out[k] == true_out[k]):
            correct_classification[k1] = sigma_[i, j, int(predicted_out[k])]
            k1 = k1 + 1
        if (predicted_out[k] != true_out[k]):
            misclassification_pred[k2] = sigma_[i, j, int(predicted_out[k])]
            misclassification_true[k2] = sigma_[i, j, int(true_out[k])]
            k2 = k2 + 1
        k = k + 1
print('Average Output Variance', np.mean(pred_var))

var1 = pred_var  # np.reshape(var, int(x_test.shape[0]/(batch_size))* batch_size)
# print(var1)
writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
df = pd.DataFrame(np.abs(var1))
# Write your DataFrame to a file
df.to_excel(writer, "Sheet")
```

```
    df1 = pd.DataFrame(predicted_out)
    df1.to_excel(writer, 'Sheet', startcol=4)

    df2 = pd.DataFrame(true_out)
    df2.to_excel(writer, 'Sheet', startcol=7)

    df3 = pd.DataFrame(correct_classification)
    df3.to_excel(writer, 'Sheet', startcol=10)

    df4 = pd.DataFrame(misclassification_pred)
    df4.to_excel(writer, 'Sheet', startcol=13)

    df5 = pd.DataFrame(misclassification_true)
    df5.to_excel(writer, 'Sheet', startcol=16)
    writer.save()

    pf = open(PATH + test_path + 'var_info.pkl', 'wb')
    pickle.dump([correct_classification, misclassification_true, pred_var], pf)
    pf.close()

    textfile = open(PATH + test_path + 'Related_hyperparameters.txt', 'w')
    textfile.write(' Input Dimension : ' + str(image_size))
    # textfile.write('\n No of Kernels : ' + str(num_kernels))
    textfile.write('\n Number of Classes : ' + str(num_classes))
    textfile.write('\n No of epochs : ' + str(epochs))
    textfile.write('\n Initial Learning rate : ' + str(lr))
    textfile.write('\n Ending Learning rate : ' + str(lr_end))
    # textfile.write('\n kernels Size : ' + str(kernels_size))
    # textfile.write('\n Max pooling Size : ' + str(maxpooling_size))
    # textfile.write('\n Max pooling stride : ' + str(maxpooling_stride))
    textfile.write('\n batch size : ' + str(batch_size))
    textfile.write('\n KL term factor : ' + str(kl_factor))
    textfile.write("\n---------------------------------")
    textfile.write("\n Averaged Test Accuracy : " + str(test_acc))
    textfile.write("\n Output Variance: " + str(np.mean(np.abs(var))))
    textfile.write("\n Correct Classification Variance: " + str(np.mean(correct_classification)))
    textfile.write("\n MisClassification Variance: " + str(np.mean(misclassification_pred)))

    textfile.write("\n---------------------------------")
    if Adversarial_noise:
        if Targeted:
            textfile.write('\n Adversarial attack: TARGETED')
            textfile.write('\n The targeted attack class: ' + str(adversary_target_cls))
        else:
            textfile.write('\n Adversarial attack: Non-TARGETED')
        textfile.write('\n Adversarial Noise epsilon: ' + str(epsilon))
        textfile.write("\n SNR: " + str(np.mean(snr_signal)))
    textfile.write("\n---------------------------------")
    textfile.close()

if (PGD_Adversarial_noise):
    if Targeted:
        test_path = 'test_results_targeted_PGDadversarial_noise_{}_max_iter_{}_{}/'.format(HCV, maxAdvStep,
                                                                                            stepSize)
        full_test_path = PATH + test_path
    if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
        shutil.rmtree(full_test_path)
    os.makedirs(PATH + test_path)
else:
    test_path = 'test_results_non_targeted_PGDadversarial_noise_{}/'.format(HCV)
    full_test_path = PATH + test_path
    if os.path.exists(full_test_path):
        # Remove the existing test path and its contents
        shutil.rmtree(full_test_path)
    os.makedirs(PATH + test_path)

trans_model.load_weights(PATH + 'vdp_cnn_model')
trans_model.trainable = False

test_no_steps = 0
true_x = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, channels])
adv_perturbations = np.zeros(
    [int(x_test.shape[0] / (batch_size)), batch_size, image_size, image_size, channels])
true_y = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
mu_out_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])
# sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size,  num_classes,  num_classes])
sigma_ = np.zeros([int(x_test.shape[0] / (batch_size)), batch_size, num_classes])

acc_test = np.zeros(int(x_test.shape[0] / (batch_size)))
epsilon = HCV / 3
for step, (x, y) in enumerate(val_dataset):
    update_progress(step / int(x_test.shape[0] / (batch_size)))
    true_x[test_no_steps, :, :, :] = x
    true_y[test_no_steps, :, :] = y

    adv_x = x + tf.random.uniform(x.shape, minval=-epsilon, maxval=epsilon)
    adv_x = tf.clip_by_value(adv_x, 0.0, 1.0)
    for advStep in range(maxAdvStep):
        if Targeted:
            y_true_batch = np.zeros_like(y)
            y_true_batch[:, adversary_target_cls] = 1.0
            adv_perturbations[test_no_steps, :, :, :] = create_adversarial_pattern(adv_x, y_true_batch)
        else:
            adv_perturbations[test_no_steps, :, :, :] = create_adversarial_pattern(adv_x, y)
        adv_x = adv_x + stepSize * adv_perturbations[test_no_steps, :, :, :]
        pgdTotalNoise = tf.clip_by_value(adv_x - x, -epsilon, epsilon)
        adv_x = tf.clip_by_value(x + pgdTotalNoise, 0.0, 1.0)

    mu_out, sigma = test_on_batch(adv_x, y)
    mu_out_[test_no_steps, :, :] = mu_out
    # sigma_[test_no_steps, :, :, :] = sigma
    sigma_[test_no_steps, :, :] = sigma
    corr = tf.equal(tf.math.argmax(mu_out, axis=-1), tf.math.argmax(y, axis=-1))
    accuracy = tf.reduce_mean(tf.cast(corr, tf.float32))
    acc_test[test_no_steps] = accuracy.numpy()
    if step % 50 == 0:
        print("Total running accuracy so far: %.4f" % acc_test[test_no_steps])
    test_no_steps += 1
test_acc = np.mean(acc_test)
print('Test accuracy : ', test_acc)
print('Best Test accuracy : ', np.amax(acc_test))

pf = open(PATH + test_path + 'uncertainty_info.pkl', 'wb')
pickle.dump([mu_out_, sigma_, true_x, true_y, adv_perturbations, test_acc], pf)
pf.close()

var = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
snr_signal = np.zeros([int(x_test.shape[0] / batch_size), batch_size])
for i in range(int(x_test.shape[0] / batch_size)):
    for j in range(batch_size):
        predicted_out = np.argmax(mu_out_[i, j, :])
        var[i, j] = sigma_[i, j, int(predicted_out)]
        snr_signal[i, j] = 10 * np.log10(
            np.sum(np.square(true_x[i, j, :, :, :])) / np.sum(
                np.square(epsilon * adv_perturbations[i, j, :, :, :])))

print('Output Variance', np.mean(var))
print('SNR', np.mean(snr_signal))

##        var1 = np.reshape(var, int(x_test.shape[0]/(batch_size))* batch_size)
##        #print(var1)
##        writer = pd.ExcelWriter(PATH + test_path + 'variance.xlsx', engine='xlsxwriter')
##        df = pd.DataFrame(np.abs(var1) )
##        # Write your DataFrame to a file
##        df.to_excel(writer, "Sheet")
##        writer.save()

textfile = open(PATH + test_path + 'Related_hyperparameters.txt', 'w')
textfile.write(' Input Dimension : ' + str(image_size))
# textfile.write('\n No of Kernels : ' + str(num_kernels))
textfile.write('\n Number of Classes : ' + str(num_classes))
textfile.write('\n No of epochs : ' + str(epochs))
textfile.write('\n Initial Learning rate : ' + str(lr))
textfile.write('\n Ending Learning rate : ' + str(lr_end))
# textfile.write('\n kernels Size : ' + str(kernels_size))
# textfile.write('\n Max pooling Size : ' + str(maxpooling_size))
# textfile.write('\n Max pooling stride : ' + str(maxpooling_stride))
textfile.write('\n batch size : ' + str(batch_size))
textfile.write('\n KL term factor : ' + str(kl_factor))
textfile.write("\n---------------------------------")
textfile.write("\n Test Accuracy : " + str(test_acc))
textfile.write("\n Output Variance: " + str(np.mean(np.abs(var))))
textfile.write("\n---------------------------------")
if PGD_Adversarial_noise:
    if Targeted:
        textfile.write('\n Adversarial attack: TARGETED')
        textfile.write('\n The targeted attack class: ' + str(adversary_target_cls))
    else:
        textfile.write('\n Adversarial attack: Non-TARGETED')
    textfile.write('\n Adversarial Noise epsilon: ' + str(epsilon))
    textfile.write('\n Adversarial Noise HCV: ' + str(HCV))
    textfile.write("\n SNR: " + str(np.mean(snr_signal)))
    textfile.write("\n stepSize: " + str(stepSize))
    textfile.write("\n Maximum number of iterations: " + str(maxAdvStep))
textfile.write("\n---------------------------------")
textfile.close()
```

```
if __name__ == '__main__':
    main_function()
```

BIOGRAPHICAL SKETCH

Fazlur Rahman Bin Karim is from Chattogram, Bangladesh. He finished his Bachelor of Science in Electrical and Electronic Engineering from Chittagong University of Engineering and Technology in December 2014. After completing graduation, he worked in the industrial sector of Bangladesh for five years as an Operations Engineer. He moved to the USA to pursue a Master of Science in Electrical Engineering (EE) at the University of Texas Rio Grande Valley (UTRGV) in January 2022. He started his job as a Graduate Research Assistant (GRA) and researched Probabilistic Machine Learning with Dr. Dimah Dera.

Under the guidance of Dr. Dimah Dera, an accomplished researcher in machine learning, he pursued his Master's thesis. His research focuses on advancing Bayesian deep neural networks for sequential data, and he applied this work in practical contexts that include healthcare and optimization.

He obtained his Master of Science in Engineering degree from the University of Texas Rio Grande Valley (UTRGV), USA, in December 2023. After completing his master's, he is going to join the University of Texas at Dallas, Richardson, Texas, as a Graduate Research Assistant for his PhD in Electrical Engineering. He can be reached at fazlur0902033@gmail.com.