DA.32_TrinhHungCuong_520K0140_520C0001.docx

bởi Khoa Liêu Đăng

Ngày Nộp: 29-thg 3-2024 01:11SA (UTC+0700)
ID Bài Nộp: 2326897595
Tên Tập tin: DA.32_TrinhHungCuong_520K0140_520C0001.docx (3.48M)
Đếm từ: 9658
Đếm ký tự: 57243

VIETNAM GENERAL CONFEDERATION OF LABOR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

LIEU DANG KHOA – 520K0140
VI NGUYEN THANH DAT – 520C0001

CLASSIFICATION OF AIR POLLUTION LEVELS USING DEEP LEARNING MODELS

INFORMATION TECHNOLOGY PROJECT

COMPUTER SCIENCE

HO CHI MINH CITY, 2024

VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

LIEU DANG KHOA – 520K0140
VI NGUYEN THANH DAT – 520C0001

CLASSIFICATION OF AIR POLLUTION LEVELS USING DEEP LEARNING MODELS

INFORMATION TECHNOLOGY PROJECT

COMPUTER SCIENCE

Advised by

Dr. Trinh Hung Cuong

HO CHI MINH CITY, 2024

ACKNOWLEDGEMENT

We would like to express our gratitude to our teacher Mr. Trinh Hung Cuong for his guidance throughout the project and his willingness to provide us with valuable research materials. We are also grateful to our parents for their support and our friends for their advice to complete this study. This project has provided us with valuable skills, and we anticipate applying them effectively in the future.

Ho Chi Minh city, 15th March 2024.

Author

(Signature and full name)

Khoa

Lieu Dang Khoa

Dat

Vi Nguyen Thanh Dat

DECLARATION OF AUTHORSHIP

We hereby declare that this is our own project and is guided by Mr. Trinh Hung Cuong; The content research and results contained herein are central and have not been published in any form before. The data in the tables for analysis, comments and evaluation are collected by the main author from different sources, which are clearly stated in the reference section.

In addition, the project also uses some comments, assessments as well as data of other authors, other organizations with citations and annotated sources.

If something wrong happens, We'll take full responsibility for the content of our project. Ton Duc Thang University is not related to the infringing rights, the copyrights that We give during the implementation process (if any).

Ho Chi Minh city, 15th March 2024

Author

(Signature and full name)

Khoa

Lieu Dang khoa

Dat

Vi Nguyen Thanh Dat

ABSTRACT

The classification of air pollution levels is crucial to the process of public health assessment and environmental monitoring. Due to the complex nature of this task, in this study we utilize five different deep learning models for the accurate classification of air pollution levels. Over the years, deep learning models have demonstrated their abilities to handle complex and high-dimensional data, making them well-suited for this task. Our task requires us to train deep learning models, including RNNs such as LSTM and GRU on a large diverse range of pollutants such as Particulate Matter 2.5 (PM2.5), Particulate Matter 10, (PM10), Carbon Monoxide (CO), Ozone (O3) and Sulfur Dioxide (SO2). We perform evaluation on the performance of our models by using real life air quality data collected from different sources. We also compare the performance of various trained models, evaluating their accuracy and robustness across different air pollution datasets. The findings of this study shall provide valuable insights to help mitigate the impacts of air pollution on human health and overall wellbeing.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

FNN	Feedforward Neural Network
RNN	Recurrent Neural Network
MLP	Multi-layer Perceptron
LSTM	Long Short-Term Memory
BiLSTM	Bidirectional Long Short-Term Memory
GRU	Gated Recurrent Unit
PM	Particulate Matter
CO	Carbon Monoxide
O3	Ozone
SO2	Sulfur Dioxide
NO	Nitric Oxide
NO2	Nitrogen Dioxide
NOx	Nitrogen Oxides
NH3	Ammonia
AMB_TEMP	Ambient Temperature
AQI	Air Quality Index
AQI_Bucket	Air Quality Bucket
SGD	Stochastic Gradient Descent

CNN	Convolutional Neural Network
tanh	hyperbolic tangent
ReLU	Rectified Linear Unit
L1	Lasso Regression
L2	Ridge Regression

CHAPTER 1. INTRODUCTION

In recent decades, the field of deep learning has witnessed significant advancements and support from the computer science community. Considering this perspective, deep learning has been investigated and implemented across various fields, resulting in notable enhancements in both efficiency and accuracy, areas where machines demonstrate proficiency. Henceforth, this approach is also being employed to document and analyze air pollution across diverse urban regions to better understand its severity and make further research. In this report, we developed deep learning models that aimed to resolve classification tasks for air quality dataset as well as implementing Flask for web application.

1.1 The drawbacks encounter of this project

Section: Drawbacks of Implementing Deep Learning Classification on Air Quality Dataset

During our attempts on this deep learning project for the classification task on an air quality dataset, we encountered several challenges and drawbacks, hindering the progress and efficiency of the project. It is essential to be aware of these potential pitfalls to mitigate their impact and ensure the successful completion of the project. Some of the key drawbacks include:

Trouble Dealing with Preprocessed Dataset:

One of the major challenges faced in this project is the preprocessing of the air quality dataset. Cleaning and preparing the dataset for deep learning models can be a time-consuming and complex process. Issues such as missing data, outliers, and noisy observations can significantly impact the accuracy and reliability of the classification results.

Errors When Installing Packages:

Another obstacle we encountered is errors during the installation of required packages and dependencies for deep learning frameworks like TensorFlow or

PyTorch. Compatibility issues, version conflicts, and configuration errors can lead to delays in the project and disrupt the development.

Dealing with Low Morale:

The nature of deep learning projects, with their intricate algorithms and complex models, can sometimes lead to feelings of frustration and low morale among team members. Long training times, unexpected errors, and subpar performance of the models can demotivate individuals, affecting productivity of our group.

1.2 The meanings of this project

There are several objectives when choosing a deep learning classification project particularly for its application and insight into the analysis of the air quality in the urban area. Some key takeaways can be:

Monitoring the environment:

By utilizing deep learning algorithms to classify air quality data, we can monitor and track pollutant levels in real-time. This application is crucial for identifying pollution sources, assessing air quality trends, and implementing targeted interventions to improve overall environmental health.

Health Assessment:

Deep learning classification can help identify patterns and associations between air pollutants and respiratory diseases, enabling early detection and prevention strategies.

City Development:

Deep learning classification of air quality data can assist in identifying pollution hotspots, optimizing transportation routes to reduce emissions, and designing green spaces to improve air quality.

CHAPTER 2. THEORETICAL BASIS

2.1 An introduction to Air Pollution Classification

This task aims to provide helpful information about air quality patterns, pollutant sources and potential health risks by measuring the concentrations of pollutants in the air. Based on the data provided, scientists and environmentalists around the world can make informed decisions on how to counter air pollution effectively.

2.1.1 Overview

In recent years, air pollution has become a hot topic as it poses a serious threat to public health and the environment. As urbanization continues to expand worldwide, the assessment of air pollution has become a critical and important task. By monitoring air pollution, effective strategies can be created to mitigate the harmful effects of pollution.

2.1.2 The significance of air pollution monitoring

By continuously measuring pollutant concentrations in the atmosphere, monitoring systems provide helpful information about air quality patterns, pollutant sources, the most polluted areas and potential health risks to the common population. This data informs policymakers, urban planners, and healthcare professionals, empowering them to make informed decisions aimed at safeguarding public health and preserving the environment.

2.2 Fundamentals of Deep Learning models for classification

2.2.1 An overview of deep learning in classification

Deep learning has changed how we classify things by teaching computers to understand complicated patterns straight from the data. It uses neural networks made up of many layers to figure out important details from the input data. These networks

can be set up to do all sorts of classification jobs, like recognizing images, understanding language, or analyzing data over time.

In classification tasks, the model aims to predict a category or a label based on its features. This process involves the mapping of input features and output features through optimization algorithms such as stochastic gradient descent (SGD). Deep learning's main strength is its ability to learn relevant features from the data without the need for feature engineering.

The flexibility of deep learning architectures, such as feedforward neural networks (FNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs), make them well-suited for a wide range of classification tasks. These models excel at capturing complex patterns and dependencies within the data, enabling highly accurate classification even if the data has noise or outliers.

2.2.2 Feedforward Neural Networks (FNNs)

Feedforward Neural Networks (FNNs), also known as multi-layer perceptrons (MLPs), are the building blocks of deep learning models. They consist of multiple layers of interconnected neurons, where each neuron in a layer is connected to every neuron in the subsequent layer.

The architecture of an FNN typically comprises an input layer, one or more hidden layers, and an output layer. The input layer receives the raw features of the data, while the hidden layers perform nonlinear transformations on the input data, extracting hierarchical representations. The output layer produces the final predictions based on the learned representations.

Common activation functions include the sigmoid function, hyperbolic tangent (tanh) function, and rectified linear unit (ReLU). These functions introduce nonlinearities that enable FNNs to approximate arbitrary functions efficiently.

FNNs are trained using optimization algorithms such as stochastic gradient descent (SGD). During training, the model's parameters (weights and biases) are iteratively updated to minimize a predefined loss function, which measures the discrepancy between predicted and true labels. Backpropagation, a technique for

efficiently computing gradients, is used to propagate error signals backward through the network and adjust the parameters accordingly.

2.2.3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a type of neural network designed to work with data that comes in sequences, like sentences or time series. Unlike regular neural networks, which treat each piece of data separately, RNNs have loops in them, allowing them to remember information over time. This makes RNNs great for tasks involving sequences, such as predicting future values in a time series, understanding language, or recognizing speech.

The basic architecture of an RNN consists of recurrent units that receive input at each time step and produce output as well as an updated internal state. The key characteristic of RNNs is their ability to retain information about previous inputs through recurrent connections, enabling them to model sequences of arbitrary length. However, traditional RNNs suffer from the vanishing gradient problem, which limits their ability to capture long-range dependencies.

2.2.4 Regularization Techniques

In deep learning, overfitting is a common problem where the model learns to memorize the training data instead of understanding the underlying patterns. Regularization techniques are methods used to prevent overfitting by adding constraints to the optimization process.

Types of Regularization Techniques

L1 and L2 Regularization: These methods add a penalty term to the loss function based on the magnitude of the weights. L1 regularization encourages sparsity in the weights, while L2 regularization penalizes large weight values.

Dropout: Dropout is a technique where randomly selected neurons are ignored during training. This forces the network to learn redundant representations and prevents it from relying too heavily on specific neurons.

Batch Normalization: Batch normalization normalizes the activations of each layer, helping to stabilize the training process and reduce internal covariate shift. It acts as a regularizer by reducing the reliance on specific features.

Early Stopping: Early stopping is a simple yet effective regularization technique where training is stopped when the performance on a validation dataset starts to degrade. This prevents the model from overfitting to the training data.

Data Augmentation: Data augmentation involves artificially increasing the size of the training dataset by applying transformations such as rotation, scaling, or flipping to the input data. This helps the model generalize better to unseen data.

Regularization techniques are crucial in deep learning to prevent overfitting and improve the generalization performance of the models. By incorporating regularization methods into the training process, we can build more robust and reliable models that perform well on unseen data.

2.2.5 Evaluation metrics

Accuracy

Accuracy represents the proportion of correctly classified instances out of the total number of instances. It is a fundamental metric for evaluating classification models and provides a general measure of model performance.

While accuracy is intuitive and easy to interpret, it may not be suitable for imbalanced datasets, where one class dominates the others. In such cases, accuracy alone may not provide a comprehensive assessment of model performance.

Precision

Precision measures the proportion of true positive predictions out of all positive predictions made by the model. It focuses on the accuracy of positive predictions and is particularly useful when the cost of false positives is high.

A high precision indicates that the model makes fewer false positive predictions, which is desirable in applications where the emphasis is on minimizing false alarms.

Recall (Sensitivity)

Recall, also known as sensitivity or true positive rate, measures the proportion of actual positive instances that are correctly identified by the model. It assesses the model's ability to capture all positive instances and is crucial when the cost of false negatives is high. Recall is calculated as:

A high recall indicates that the model effectively captures most positive instances, minimizing the number of false negatives.

F1 Score

The F1 score is the harmonic mean of precision and recall and provides a balanced measure of a model's performance. It considers both false positives and false negatives and is especially useful when there is an imbalance between classes or when both precision and recall are equally important. The F1 score is calculated as:

The F1 score ranges from 0 to 1, where a higher score indicates better model performance in terms of both precision and recall.

2.3 Models Overview

2.3.1 Model 1 - Multilayer Perceptron (MLP)

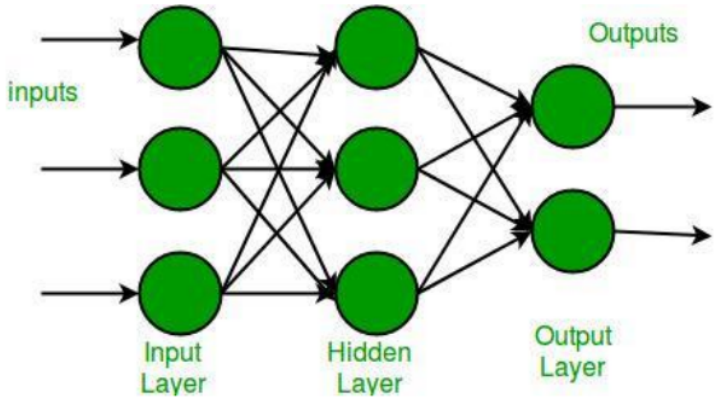


Figure 2.1: In the depicted multi-layer perceptron (MLP) diagram, a neural network architecture is illustrated. This MLP comprises three distinct layers: an input layer, a hidden layer, and an output layer. Each layer consists of nodes, also known as neurons, which process and transmit information

Input Layer: The input layer receives the features of the data being fed into the model. Each neuron in the input layer corresponds to a feature in the dataset. The number of neurons in the input layer is equal to the number of features in the dataset.

Hidden Layers: Each hidden layer consists of multiple neurons, and the number of hidden layers and neurons in each layer is a design choice. The activation function is applied to the weighted sum of inputs at each neuron to allow the model to learn complex patterns in the data. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, and tanh.

Output Layer: The output layer produces the predictions or outputs of the model. For regression tasks, the output layer typically consists of a single neuron with a linear activation function. For binary classification tasks, the output layer usually consists of a single neuron with a sigmoid activation function, which squashes the output to the range [0, 1], representing the probability of the positive class. For multiclass classification tasks, the output layer typically has multiple neurons, one for each class, with a softmax activation function to produce a probability distribution over the classes.

Regularization Layers: Regularization techniques like dropout and batch normalization can be applied to the hidden layers to prevent overfitting. Dropout randomly sets a fraction of the neurons' outputs to zero during training, which helps prevent the model from relying too heavily on any individual neuron. Batch normalization normalizes the activations of each layer, making training more stable and accelerating convergence.

Loss Function and Optimization: The choice of loss function depends on the nature of the required task: mean squared error for regression, binary cross-entropy for binary classification, categorical cross-entropy for multiclass classification, etc. Optimization algorithms like Adam, RMSprop, or SGD are used to minimize the loss function during training.

Activation Functions: Activation functions introduce non-linearity into the model, enabling it to learn complex mappings between the input and output. Common activation functions used in hidden layers include ReLU (Rectified Linear Unit), sigmoid, tanh, and Leaky ReLU

2.3.2 Model 2 - Long Short-Term Memory (LSTM)

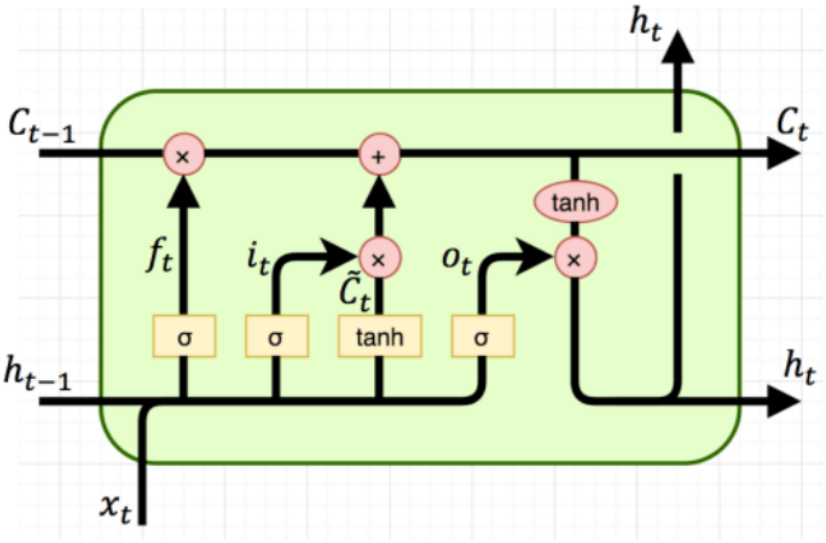


Figure 2.2: Long Short Term Memory model (LSTM) model consists of forget gate (ft), input gate (it), output gate (ot), cell state (ct), hidden state (ht)

Input Layer: The input layer of an LSTM model is designed to accept sequential data, such as time series or text data. It takes input in the form of sequences with variable lengths. Each input sample consists of a sequence of features.

LSTM Layers: The LSTM layers are the key components responsible for capturing long-term dependencies in sequential data. Each LSTM unit contains a cell state and three gates: forget gate, input gate, and output gate. These gates control the flow of information through the cell state, allowing the model to retain or discard information at each time step. LSTM layers can be stacked to create deeper architectures.

Output Layer: The output layer of the LSTM model produces predictions or outputs based on the information learned from the input sequences. Its configuration

depends on the task at hand. For sequence prediction tasks, such as time series forecasting or text generation, the output layer may consist of multiple neurons, one for each time step in the output sequence. For sequence classification tasks, such as sentiment analysis or named entity recognition, the output layer typically consists of a single neuron with a sigmoid or softmax activation function.

Regularization Layers: To prevent overfitting, LSTM models can utilize regularization techniques such as dropout and recurrent dropout. Dropout randomly drops connections between units during training to reduce the model's reliance on specific paths through the sequence. Batch normalization can also be applied to stabilize the training process and accelerate convergence.

Loss Function and Optimization: The choice of loss function and optimization algorithm depends on the specific task. Mean squared error (MSE) is commonly used for regression tasks, while categorical cross-entropy is used for classification tasks. Optimization algorithms like Adam, RMSprop, or SGD are employed to minimize the loss function during training.

Activation Functions: LSTM units typically use activation functions such as sigmoid and tanh to control the flow of information through the gates and regulate the cell state. These activation functions introduce non-linearity into the model, enabling it to learn complex patterns in sequential data.

Components of LSTM model

Forget gate: (f_t) allows LSTM model to retain and discard unnecessary information from previous cell states. Input of forget gate is a combination of the previous hidden state h_{t-1} and current input of x_t. It produces an output value between 0 and 1, with the model discarding information based on a value of 0 and retaining it otherwise.

Forget gate: $ft = \sigma(Wf \cdot [ht - 1, xt] + bf)$

Input gate: (i_t) This gate decides what new information should be stored in the cell state. This gate consists of two parts: a sigmoid layer which decides how much information needed to be updated, and a tanh layer that generates a new vector of new candidate values to be added to the state. The sigmoid layer is called the "input gate" because it determines how much of the candidate values we'll actually use.

- Calculate the input gate itit using the sigmoid activation function:

$$it = \sigma(Wi \cdot [ht - 1, xt] + bi)$$

- Calculate the candidate cell state C~t using the tanh activation function:

$$C\sim t = tanh(Wc \cdot [ht - 1, xt] + bc)$$

Cell state: The memory of the LSTM model. It passes through time steps and is modified by the forget and input gates.

$$Ct = ft \odot Ct - 1 \; + \; it \odot C\sim t$$

Output Gate Operation:

- Calculate the output gate otot using the sigmoid activation function:

$$ot = \sigma(Wo \cdot [ht - 1, xt] + bo)$$

- Calculate the output htht using the updated cell state CtCt and the tanh activation function:

$$ht = ot \odot tanh(Ct)$$

2.3.3 Model 3 - Bidirectional LSTM Model (BiLSTM)

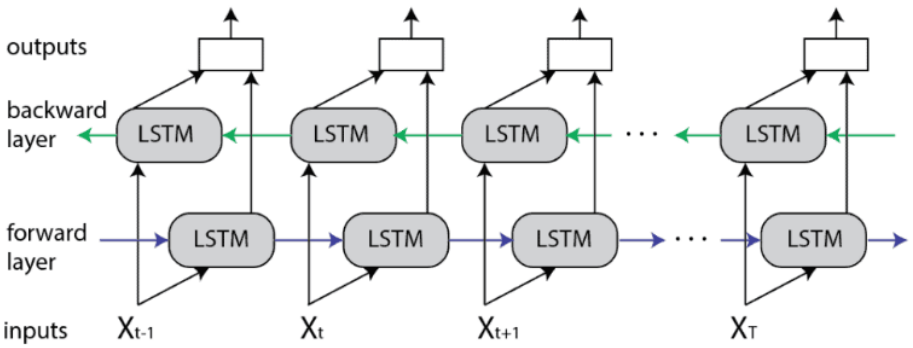


Figure 2.3: Bidirectional LSTM (BiLSTM) Model has two LSTM networks in opposite direction: one works on input sequences forward, while the other works on it in the backward direction. The outputs of these two networks are then concatenated at each time step to provide a comprehensive representation of the input sequence

Input Layer: Similar to the LSTM model, the input layer of a Bidirectional LSTM (BLSTM) model is designed to accept sequential data. It takes input in the form of sequences with variable lengths, just like the unidirectional LSTM.

Bidirectional LSTM Layers: The main difference between a BLSTM and a unidirectional LSTM lies in the architecture of the LSTM layers. In a BLSTM, each LSTM layer is split into two separate layers: one processing the input sequence in the forward direction, and the other processing the sequence in the reverse direction. This bidirectional processing allows the model to capture information from both past and future context at each time step, enabling it to better understand the temporal dependencies in the data.

Concatenation or Merge Layer: After processing the input sequence in both forward and backward directions through the bidirectional LSTM layers, the outputs from these two directions are combined. This can be done using a concatenation layer, where the outputs from the forward and backward LSTM layers are concatenated

along the feature axis. Alternatively, a merge layer, such as addition or averaging, can be used to combine the outputs.

Output Layer: Similar to the unidirectional LSTM model, the output layer of a BLSTM model produces predictions or outputs based on the information learned from both the forward and backward directions. The configuration of the output layer depends on the specific task, with multiple neurons for sequence prediction tasks or a single neuron for sequence classification tasks.

Regularization Layers: To prevent overfitting, regularization techniques like dropout and batch normalization can be applied to the BLSTM model, similar to the unidirectional LSTM. These layers help improve the generalization performance of the model by reducing overfitting and stabilizing the training process.

Loss Function and Optimization: The choice of loss function and optimization algorithm remains the same as in the unidirectional LSTM model, depending on the task at hand. Common choices include mean squared error (MSE) for regression tasks and categorical cross-entropy for classification tasks, along with optimization algorithms like Adam, RMSprop, or SGD.

Activation Functions: The activation functions used in the BLSTM model are typically the same as those in the unidirectional LSTM, including sigmoid and tanh functions. These functions introduce non-linearity into the model, enabling it to capture complex patterns in sequential data.

Components of BiLSTM model

Forward layer: This LSTM network processes the input sequence from left to right, capturing information from the past context..

Backward LSTM: This LSTM network processes the input sequence from right to left, capturing information from the future context.

Concatenation: The outputs of the forward and backward LSTMs are combined at each time step.

Output Layer: The combined representations from the forward and backward LSTMs are often passed through additional layers (e.g., fully connected layers) to produce the final output of the BLSTM model. The specific architecture of the output layer depends on the task being performed (e.g., classification, regression, sequence labeling).

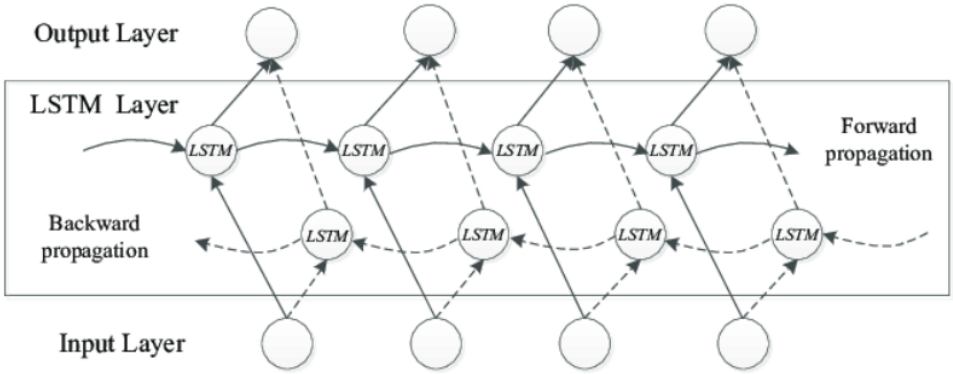


Figure 2.4: The architecture of a Bidirectional LSTM (BiLSTM) model depicted in another image

2.3.4 Model 4 - Gated Recurrent Unit (GRU)

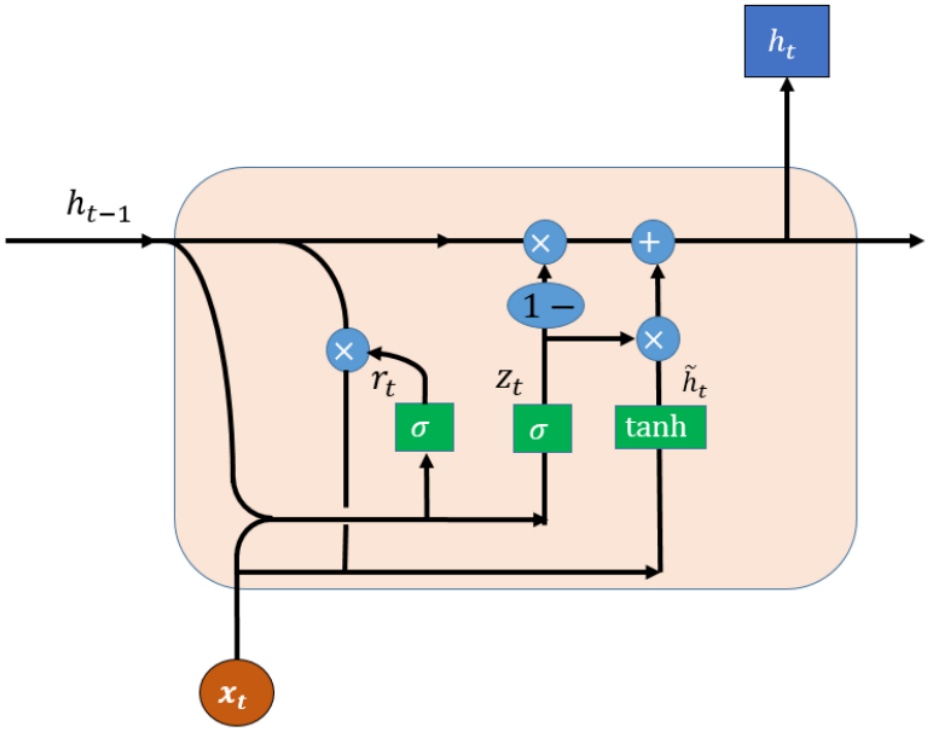


Figure 2.5: Generative Recurrent Unit (GRU) model has similar structure compared to LSTM with an additional gate mechanism to input or remove features.

Input Layer: Similar to the LSTM and Bidirectional LSTM (BLSTM) models, the input layer of a Gated Recurrent Unit (GRU) model is designed to accept sequential data. It receives input sequences of variable lengths, accommodating the dynamic nature of sequential data.

GRU Layer: The GRU layer consists of recurrent units with gating mechanisms, similar to LSTM layers. However, the GRU architecture is simpler, with fewer gating parameters. Each GRU unit processes the input sequence and updates

its hidden state, controlling the flow of information through gating operations. This simplicity often leads to faster training times compared to LSTM models.

Output Layer: The output layer of a GRU model generates predictions based on the learned representations from the input sequence. The configuration of the output layer depends on the specific task, such as classification or sequence prediction. For classification tasks, there may be multiple neurons representing different classes, while for sequence prediction tasks, a single neuron or multiple neurons may be used to predict future values.

Regularization Layers: To prevent overfitting, regularization techniques like dropout and batch normalization can be applied to the GRU model. These layers help improve the model's generalization performance by reducing overfitting and promoting robustness.

Loss Function and Optimization: The choice of loss function and optimization algorithm in a GRU model is similar to that of LSTM and BLSTM models, depending on the task requirements. Common loss functions include mean squared error (MSE) for regression tasks and categorical cross-entropy for classification tasks. Optimization algorithms like Adam, RMSprop, or SGD are used to train the model parameters effectively.

Activation Functions: GRU models typically employ activation functions such as sigmoid and hyperbolic tangent (tanh) functions within the recurrent units. These activation functions introduce non-linearity, enabling the model to capture complex patterns and dependencies in sequential data.

Components of a GRU model

Update Gate (z_t):

This gate determines how much of the past information to keep and how much of the new information to let through. It is computed using the sigmoid activation function.

The update gate z_t decides what portion of the previous hidden state h_{t-1} to keep and what portion of the new candidate activation h~t to use.

Mathematically, the update gate is computed as:

$$zt = \sigma(Wz \cdot [ht - 1, xt])$$

Wz is the weight matrix specific to the update gate.

Reset Gate (r_t):

This gate decides how much of the past information to forget.

It is computed similarly to the update gate but with a different weight matrix:

$$rt = \sigma(Wr \cdot [ht - 1, xt])$$

Candidate Activation (h~th~t):

This is the new candidate activation that will be added to the hidden state.

It is computed similarly to the candidate cell state in LSTM:

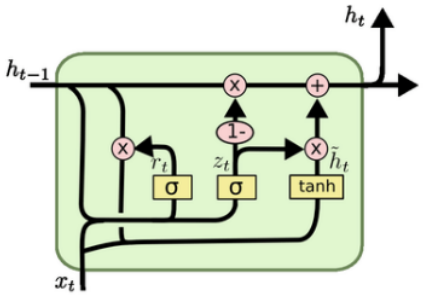
$$h\sim t = tanh(Wh \cdot [rt \odot ht - 1, xt])$$

Wh is the weight matrix specific to the candidate activation.

Hidden State Update (h_t):

The new hidden state h_t is a combination of the previous hidden state h_{t-1} and the candidate activation h~t, controlled by the update gate z_t:

$$ht = (1 - zt) \odot ht - 1 + zt \odot h\sim t$$



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2.6: The GRU model combines the "forget" gate and "input" gate into a single "update" gate

2.3.5 Model 5 – Autoencoder

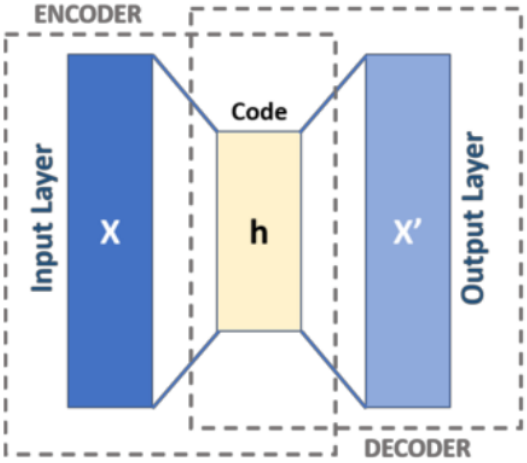


Figure 2.7: Autoencoder model. This model consists of encoder and decoder layers

Input Layer: The input layer of an autoencoder receives the raw input data, which could be images, text, or any other type of data. It represents the initial stage of the autoencoder architecture, where the input is fed into the network for encoding.

Encoder: The encoder component of the autoencoder compresses the input data into a lower-dimensional latent space representation. This is achieved through a series of hidden layers that progressively reduce the dimensionality of the input data. Each layer in the encoder applies transformations to the input, extracting relevant features and patterns.

Latent Space: The latent space is a low-dimensional representation of the input data that captures its essential characteristics. It serves as an encoded version of the input and is often much smaller in dimensionality compared to the original input space. The latent space representation is learned by the autoencoder during the training process.

Decoder: The decoder component of the autoencoder reconstructs the original input data from the compressed latent space representation. It consists of one or more layers that gradually upsample the latent space representation to match the dimensionality of the original input. The decoder's goal is to generate output that closely resembles the input data.

Output Layer: The output layer of the autoencoder produces the reconstructed data, which ideally should closely resemble the input data. The output layer typically uses an activation function appropriate for the type of data being reconstructed, such as sigmoid for binary data or linear for continuous data.

Loss Function: The loss function of the autoencoder measures the discrepancy between the input data and the reconstructed output. Common loss functions used in autoencoders include mean squared error (MSE) for continuous data and binary cross-entropy for binary data. The autoencoder aims to minimize this loss function during training to improve the quality of the reconstructed output.

Training: During the training process, the autoencoder learns to encode the input data into a compact latent space representation and decode it back to reconstruct the original data. This is achieved through an iterative optimization process, where the network's parameters are adjusted to minimize the reconstruction error.

Applications: Autoencoders have various applications across domains such as image processing, natural language processing, and anomaly detection. They are used for tasks such as image denoising, dimensionality reduction, feature learning, and generating synthetic data.

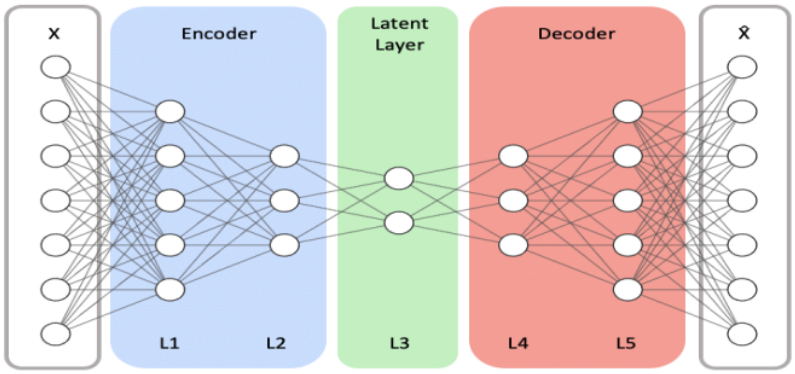


Figure 2.8: A typical autoencoder model comprises symmetric encoder and decoder networks. Here, 'X' denotes the model's input, while '^X' represents its reconstructed output.

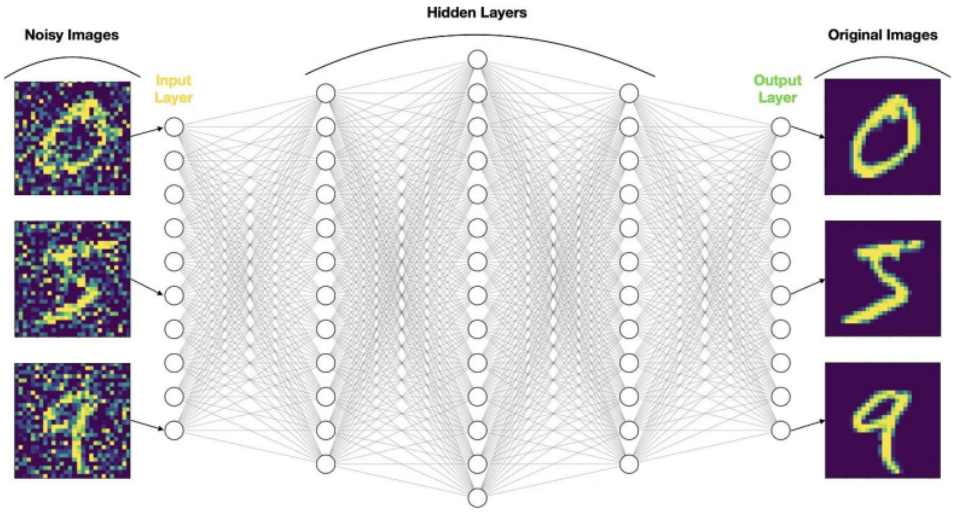


Figure 2. 1: An example of a Denoising Autoencoder (DAE) used to denoise MNIST handwritten digits

CHAPTER 3. DESIGNING AND IMPLEMENTING

3.1 Data Collection and Preprocessing

In this section, we will highlight two datasets that had been applied for training for our air quality classification deep learning models.

3.1.1 Data collection

The first dataset we chose is the dataset city_day on Kaggle. This is the dataset that records air quality based on many factors that affect the air environment of different cities in India, it is recorded daily in a span of 5 years from 2015 till 2020. Some notable features that can be described in the city_day dataset include:

PM2.5: Particulate Matter 2.5 (PM2.5) refers to the particles and droplets in the air that measure less or equal than 2.5 micrometers. These particles can be dust, dirt, soot, droplets.

PM10: Particulate Matter 10 (PM10) refers to the particles and droplets in the air that measure less or equal than 10 micrometers. These particles can be dust, pollen, mold, and particles from vehicle exhaust that are much larger.

NO: Nitric Oxide (NO) is gas produced during the combustion process from vehicle and industrial activities.

NO2: Nitrogen Dioxide (NO2) is a type of gas and a significant air pollutant.

NOx: Nitrogen Oxides (NOx) is a collective term referring to a group of nitrogen-containing gasses, including nitric oxide (NO) and nitrogen dioxide (NO2).

NH3: Ammonia (NH3) is a pungent, colorless gas, a combination of nitrogen and hydrogen. It is released from agricultural activities, livestock waste, and industrial processes.

CO: Carbon Monoxide (CO) is a colorless, odorless gas produced by incomplete combustion of carbon-containing fuels such as gasoline, wood, and natural gas.

SO2: Sulfur Dioxide (SO2) is a pungent, colorless gas produced by burning sulfur-containing fuels like coal and oil, as well as industrial processes such as metal smelting.

O3: Ozone (O3) is a reactive gas composed of three oxygen atoms.

Benzene: Benzene is a colorless, aromatic hydrocarbon compound found in gasoline and other petroleum-based products.

Toluene: Toluene is a clear, colorless liquid with a sweet smell commonly used as a solvent in paints, coatings, and adhesives.

Xylene: Xylene is a group of three isomeric aromatic hydrocarbon compounds (ortho-, meta-, and para-xylene) commonly used as solvents in paints, varnishes, and adhesives.

AQI: Air Quality Index (AQI) is a measurement that calculates the concentration of pollutants in the air and provides information about health risk in the area affected.

AQI_Bucket: AQI Bucket categorizes the AQI value into predefined ranges or buckets, indicating the overall air quality status. The bucket includes these evaluations: 'Moderate', 'Poor', 'Very Poor', 'Satisfactory', 'Good', 'Severe'.

The next dataset that we used is the taiwan dataset. This dataset records air quality of different stations in Taiwan in the year 2015. In contrast to the first dataset, this dataset measures the changes of features of the air environment on an hourly basis. Some notable features that can be described in the taiwan dataset include:

AMB_TEMP: Ambient Temperature (AMB_TEMP) refers to the temperature of the surrounding air in the environment where the air quality measurements are taken.

CO: Carbon Monoxide

NOx: Nitrogen Oxides

O3: Ozone

PM10: Particulate Matter 10

PM2.5: Particulate Matter 2.5

RAINFALL: rainfall measured in millimeters.

SO2: Sulfur Dioxide

WIND_SPEED: refers to the speed at which air molecules move measured in m/s

3.1.2 Data Preprocessing

Prior to training the model, we performed extensive preprocessing on the dataset to ensure compatibility with the chosen model architecture and to enhance model performance. The preprocessing pipeline consisted of the following steps:

For the dataset preparation phase using the "city_day.csv" and “taiwan2015.csv” datasets, the following steps are taken:

Cleaning Data:

- Remove rows where the target column "AQI" is null, as these instances cannot be used for training.
- Drop unnecessary features such as "Date", "AQI", and "City" to streamline the dataset and remove redundant information.

Feature Selection:

Select relevant features necessary for training by retaining only the essential columns in the dataset after removing unnecessary features. We use up to 12 features that are “PM2.5, PM10, NO, NO2, NOx, NH3, CO, SO2, O3, Benzene, Toluene and Xylene” for the first dataset while only using six of them for the second dataset.

Splitting Data:

Separate the dataset into input features (X) and target variable (y), where X contains the selected features and y contains the target variable "AQI".

Apply label encoding to the target variable y to convert categorical data into numerical labels, making it suitable for model training.

Handling Missing Values:

Fill in missing values in the input features (X) using a technique such as SimpleImputer(). This ensures that the dataset is complete and ready for training, as missing values can adversely affect model performance.

Scaling Features:

Scale the input features (X) using MinMaxScaler to normalize the data within a specific range. Scaling ensures that all features contribute equally to model training, preventing features with larger magnitudes from dominating the learning process.

By following these dataset preparation steps, we ensure that the "city_day.csv" and “taiwan2015.csv” datasets is properly cleaned, relevant features are selected, missing values are handled appropriately, and the data is scaled for effective training of machine learning models. This process sets the foundation for building accurate and robust predictive models for air quality prediction.

3.2 Model Training

3.2.1 Designing Models

We implemented 5 models (MLP, LSTM, BLSTM, Autoencoder) with various structures:

Model 1 - Multilayer Perceptron (MLP)

```
126] import tensorflow as tf
     from keras import regularizers
     MLP_model = tf.keras.models.Sequential([
         #tf.keras.layers.Input(shape=(100,7)),
         tf.keras.layers.Dense(512, activation='relu', input_shape=input_shape, kernel_regularizer=regularizers.l2(0.001)),
         #tf.keras.layers.BatchNormalization(),
         tf.keras.layers.Dropout(0.125),

         tf.keras.layers.Dense(256, activation='relu'),
         #tf.keras.layers.BatchNormalization(),
         tf.keras.layers.Dropout(0.125),

         tf.keras.layers.Dense(128, activation='relu'),
         #tf.keras.layers.BatchNormalization(),
         tf.keras.layers.Dropout(0.125),

         tf.keras.layers.Dense(64, activation='relu'),
         #tf.keras.layers.BatchNormalization(),
         tf.keras.layers.Dropout(0.125),


         tf.keras.layers.Flatten(),  # Add Flatten layer to flatten the output
         tf.keras.layers.Dense(num_classes, activation='softmax')
     ])
```

Figure 3.1: The architecture of the first model - MLP

Architecture

Input Layer: The input layer takes the shape of the input features. In this case, the input shape is determined based on the dimensions of the input data of one of the datasets.

Hidden Layers: The MLP consists of multiple dense fully connected layers. Each hidden layer contains a configurable number of neurons and utilizes the rectified linear unit (ReLU) activation function to introduce non-linearity.

Dense Layer 1: This layer has 512 neurons with ReLU activation. It takes input from the input layer (or the previous hidden layer) and applies a regularization penalty of L2 with a coefficient of 0.001 to the kernel weights. Following the first dense layer, a dropout layer with a dropout rate of 0.125 is applied. Dropout layers help in preventing overfitting by randomly setting a fraction of input units to 0 during training.

Dense Layer 2: This layer has 256 neurons with ReLU activation.Dropout Layer 2: Following the second dense layer, another dropout layer with a dropout rate of 0.125 is applied.

Dense Layer 3: This layer has 128 neurons with ReLU activation.Dropout Layer 3: Following the third dense layer, another dropout layer with a dropout rate of 0.125 is applied.

Dense Layer 4: This layer has 64 neurons with ReLU activation, another dropout layer with a dropout rate of 0.125 is applied.

Output Layer: The output layer is a Dense layer with num_classes neurons, where num_classes represents the number of classes in the classification task, depending on the dataset. It uses softmax activation, which is typical for multi-class classification problems, to output probabilities for each class.

Model 2 - Long Short-Term Memory (LSTM)

```
[ ]  LSTM_model = Sequential([
        LSTM(units=128, return_sequences=True, input_shape=input_shape, kernel_regularizer=regularizers.l2(0.001)),
        #BatchNormalization(),
        Dropout(0.2),

        LSTM(units=64, return_sequences=False, kernel_regularizer=regularizers.l2(0.001)),
        #BatchNormalization(),
        Dropout(0.2),

        Dense(units=32, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
        #BatchNormalization(),
        Dropout(0.2),

        Dense(units=num_classes, activation='softmax')
    ])
```

Figure 3.2: The architecture of the second model - LSTM

Architecture

Input Layer: The input layer of the LSTM_model plays a crucial role in processing the sequential nature of the input data. It accommodates input sequences with a shape defined as (time_step, features), where "time_step" represents the number of time steps or observations in each input sequence, and "features" denotes the number of features or variables associated with each time step.

This structure allows the model to effectively capture temporal dependencies and patterns present in the input data over time. By incorporating this sequential information, the LSTM_model can learn from the temporal dynamics inherent in the dataset, making it suitable for tasks involving time-series data analysis, such as air quality prediction.

Hidden Layers: The LSTM_model comprises multiple layers, including LSTM, and Dense layers, each serving a specific purpose in the model's architecture.

LSTM Layer 1: This LSTM layer has 128 units and is configured to return sequences (return_sequences=True). It receives input from the input layer and applies L2 regularization with a coefficient of 0.001 to the kernel weights..A dropout layer with a dropout rate of 0.3 is added after the layer. The increased dropout rate helps in regularizing the model and preventing overfitting.

LSTM Layer 2: This LSTM layer has 64 units and is configured not to return sequences (return_sequences=False). It receives input from the first LSTM layer and applies L2 regularization with a coefficient of 0.001 to the kernel weights.Another dropout layer with a dropout rate of 0.3 is added after the second LSTM layer to avoid overfitting.

Dense Layer: This dense layer has 32 units with ReLU activation and applies L2 regularization with a coefficient of 0.001 to the kernel weights. Another dropout layer with a dropout rate of 0.3 is added after to avoid overfitting.

Output Layer:The output layer consists of neurons corresponding to the six (or three) classes of air pollution levels. It utilizes the softmax activation function to generate probability distributions over the classes, enabling multi-class classification.

Model 3 - Bidirectional LSTM Model

```
[ ]  BLSTM_model = Sequential([
        Bidirectional(LSTM(units=256, return_sequences=True, kernel_regularizer=regularizers.l2(0.001)), input_shape=input_shape),
        #BatchNormalization(),
        Dropout(0.2),

        Bidirectional(LSTM(units=128, return_sequences=False, kernel_regularizer=regularizers.l2(0.001))),
        #BatchNormalization(),
        Dropout(0.2),

        Dense(units=64, activation='relu'),
        #BatchNormalization(),
        Dropout(0.2),

        # Dense(units=32, activation='relu'),
        # #BatchNormalization(),
        # Dropout(0.2),

        Dense(units=num_classes, activation='softmax')
    ])
```

Figure 3.3: The architecture of the third model - BiLSTM

Architecture

Input Layer:

The input layer of the BLSTM_model operates similarly to that of the LSTM_model. It is instrumental in processing the sequential nature of the input data, accommodating input sequences structured as (time_step, features). Here, "time_step" represents the number of time steps or observations within each input sequence, while "features" indicates the number of attributes or variables associated with each time step. This design parallels that of the LSTM_model, allowing both models to effectively capture temporal dependencies and patterns inherent in the input data over time

Hidden Layers:

Bidirectional LSTM Layer 1: This layer employs a Bidirectional LSTM with 256 units, configured to return sequences, and applies L2 regularization with a coefficient of 0.001 to the kernel weights. It receives input from the input layer and processes the sequential data bidirectionally, capturing dependencies in both forward and backward directions. Dropout regularization with a rate of 0.2 is

subsequently applied to mitigate overfitting by randomly dropping 20% of the input units during training.

Bidirectional LSTM Layer 2: Similar to the first layer, this Bidirectional LSTM has 128 units but is configured not to return sequences. It also applies L2 regularization to the kernel weights.

Dense Layer: This dense layer consists of 64 units with ReLU activation and L2 regularization on the kernel weights. It processes the output from the Bidirectional LSTM layers. And then, a dropout layer with a rate of 0.2 is subsequently applied.

Output Layer: The output layer is a dense layer with softmax activation, producing the final output probabilities for each class in the classification task. No regularization is applied to this layer.

Model 4 - Gated Recurrent Unit (GRU)

```
[ ]  GRU_model = Sequential([
        GRU(256, return_sequences=True, input_shape=input_shape, kernel_regularizer=regularizers.l2(0.01)),
        BatchNormalization(),
        Dropout(0.2),
        GRU(128, return_sequences=False, kernel_regularizer=regularizers.l2(0.001)),
        BatchNormalization(),
        Dropout(0.2),
        Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
        BatchNormalization(),
        Dropout(0.2),
        Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
        BatchNormalization(),
        Dropout(0.2),

        Dense(num_classes, activation='softmax')
    ])
```

Figure 3.4: The architecture of the fourth model - GRU

Input Layer: The input layer of the GRU_model is designed to handle sequential data with a shape defined as (time_step, features), where "time_step" represents the number of time steps or observations in each input sequence, and "features" denote the number of features or variables associated with each time step. This structure enables the model to process temporal dependencies and patterns present in the input data over time.

Hidden Layers:

GRU Layer 1: The GRU layer with 256 units applies L2 regularization to the kernel weights with a coefficient of 0.01. L2 regularization helps to prevent overfitting by penalizing large weights, which can lead to complex models that generalize poorly to new data. It helps in improving the training speed and stability. Dropout regularization with a rate of 0.2 is applied. Dropout randomly sets a fraction of input units to zero during training, which helps prevent overfitting by forcing the model to learn more robust features.

GRU Layer 2: This GRU layer with 128 units also applies L2 regularization to the kernel weights, but with a smaller coefficient of 0.001. A smaller regularization coefficient is used here because this layer follows a layer with already regularized weights, and overly strong regularization might hinder learning.

Dense Layer 1: The first dense layer with 64 units applies L2 regularization to the kernel weights with a coefficient of 0.001. Regularization helps prevent overfitting and encourages the model to learn simpler patterns.

Dense Layer 2: The second dense layer with 32 units applies L2 regularization to the kernel weights with a coefficient of 0.001.

Output Layer: The output layer is a dense layer with softmax activation, producing the final output probabilities for each class in the classification task. No regularization is applied to this layer.

Model 5 – MLP Autoencoder

```
[ ]  # Define the encoder architecture
     encoder_input = Input(shape=input_shape)
     encoded = Dense(256, activation='relu')(encoder_input)
     encoded = Dense(128, activation='relu')(encoded)

     # Define the decoder architecture
     decoded = Dense(256, activation='relu')(encoded)
     decoded = Dense(input_dim, activation='sigmoid')(decoded)

     # Build the autoencoder model
     autoencoder = Model(encoder_input, decoded)
```

Figure 3.5: The architecture of the fifth model - MLP AutoEncoder

Input Layer:The input layer of the autoencoder model receives data with a shape defined by input_shape. It serves as the entry point for the data into the model.

Hidden Layers:

The encoder architecture consists of two dense (fully connected) layers.

The first dense layer has 256 units and applies the ReLU activation function, which introduces non-linearity to the network.

The second dense layer has 128 units and also uses the ReLU activation function.The decoder architecture mirrors the encoder's architecture.It begins with a dense layer with 256 units and ReLU activation.This is followed by a dense layer with a number of units equal to the input dimension (defined by input_dim) and uses the sigmoid activation function.

Output Layer:The output layer of the autoencoder model is the result of the decoder architecture. It produces output data with the same dimensionality as the input data. The activation function used in the output layer is the sigmoid function, which scales the output values between 0 and 1.Model Building:The autoencoder model is constructed using the Model class from Keras. It takes encoder_input as the input layer and decoded as the output layer, forming the structure of the autoencoder.

3.2.2 Model Training

In this section, we will detail the training procedure for deep learning air quality classification models. Each model undergoes training using two distinct datasets: one derived from city_day records and the other from the Taiwan dataset. Consequently, two versions of each model were generated. After designing models has been achieved the models were compiled and some parameters like optimizer, loss function and metrics were given to finalize the models. In this project, we applied adam as an optimizer, sparse categorical cross entropy for the loss function and accuracy as the models metric. Most models were trained for 100 epochs and they fed 32 batches for training for every step of the training process to optimize the accuracy from the predictions. The details of training parameters of the models can be explained:

- The MLP model was trained for 100 epochs and with 32 batch size and 0.2 validation loss.
- The LSTM model was trained for 100 epochs and with 32 batch size and 0.2 validation loss.
- The BLSTM model was trained for 80 epochs and with 32 batch size and 0.2 validation loss.

- The GRU was trained for 50 epochs and with 32 batch size and 0.2 validation loss.
- The Autoencoder model was trained for 100 epochs and with 32 batch size and 0.2 validation loss.

3.2.3 Model Evaluation and Validation

After the training procedure had been conducted, the models were evaluated and validated based on their performance and we adjusted parameters based on the evaluations to avoid overfitting and underfitting to ensure the models performed ideally. The graphs below depict the loss and accuracy of each model:

MLP model:

Accuracy:

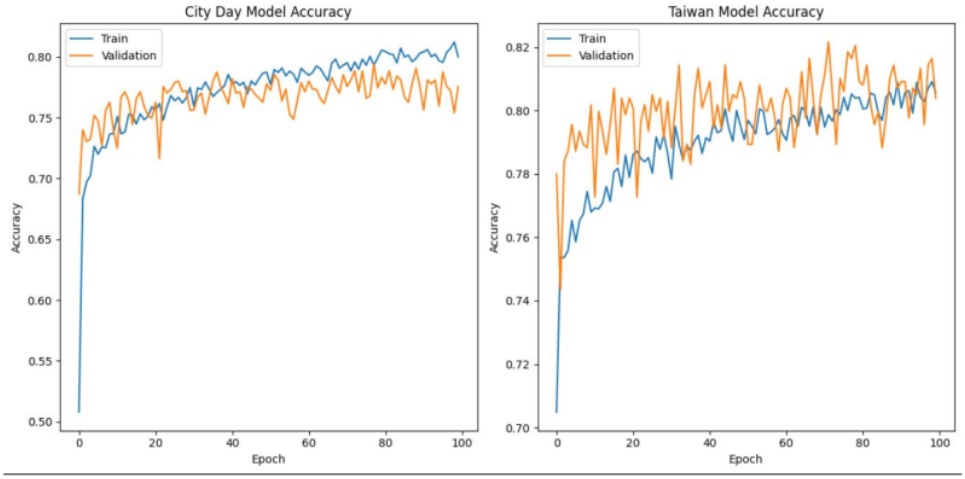


Figure 3.6: Accuracy of MLP model – City day & Taiwan

Loss:

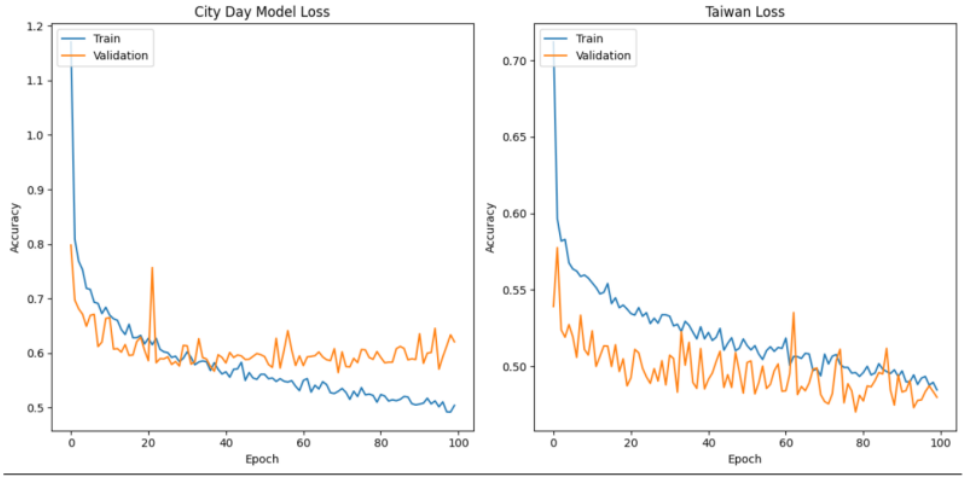


Figure 3.7: Loss of MLP model – City day & Taiwan

LSTM model:

Accuracy:

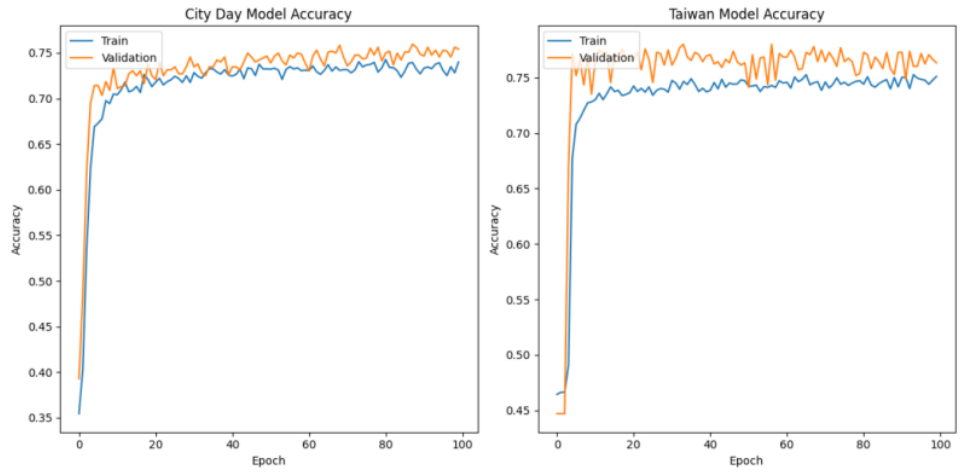


Figure 3.8: Accuracy of LSTM model – City day & Taiwan

Loss:

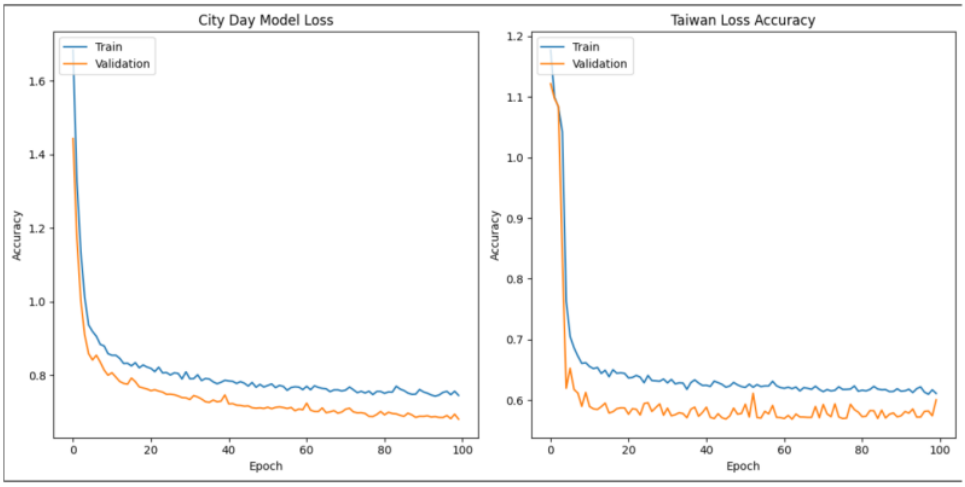


Figure 3.9: Loss of LSTM model – City day & Taiwan

BLSTM model:

Accuracy:

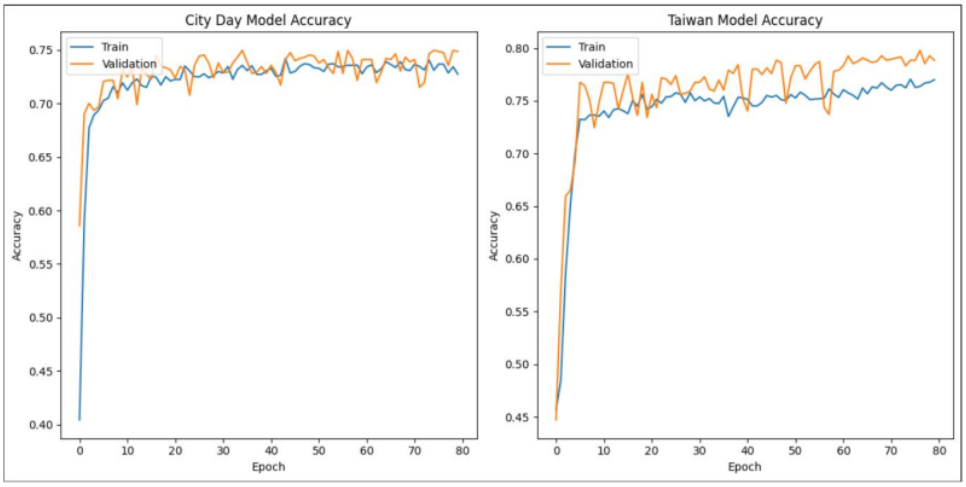


Figure 3.10: Accuracy of BiLSTM model – City day & Taiwan

Loss:

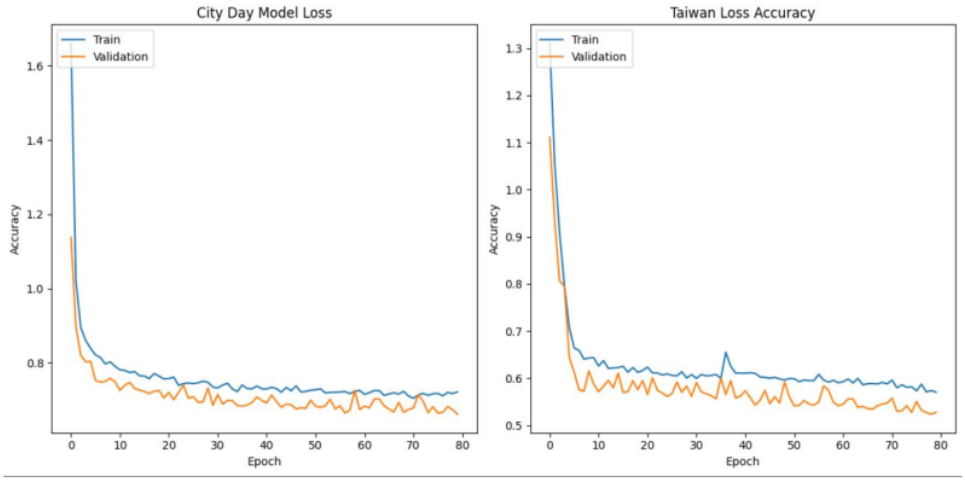


Figure 3.11: Loss of BiLSTM model – City day & Taiwan

GRU Model:

Accuracy:

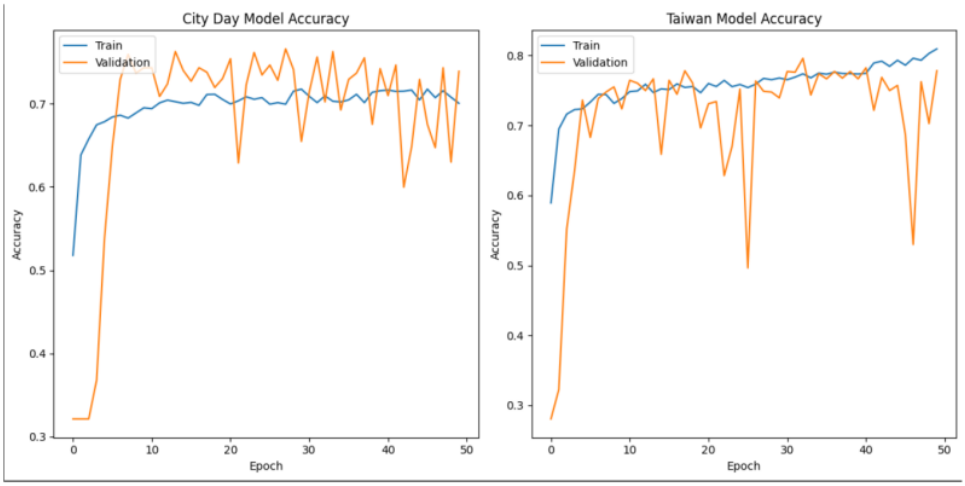


Figure 3.12: Accuracy of GRU model – City day & Taiwan

Loss:

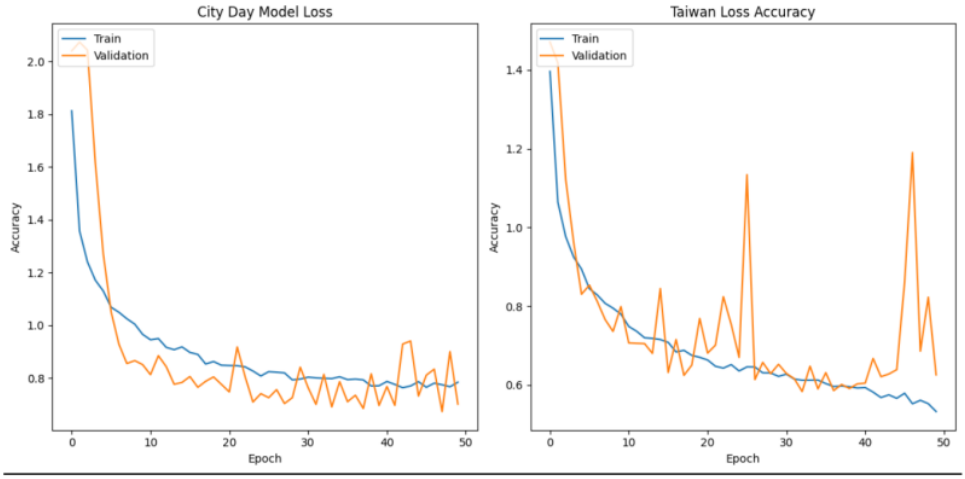


Figure 3.13: Loss of GRU model – City day & Taiwan

MLP Autoencoder:

Accuracy:

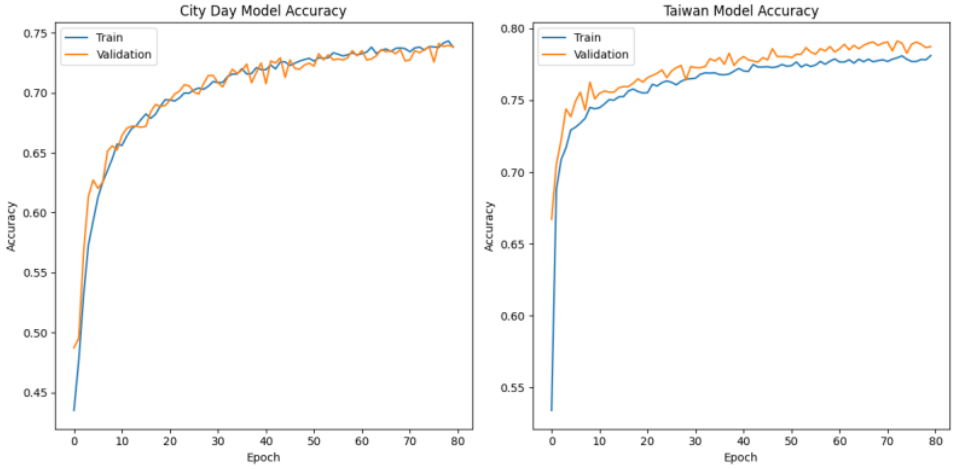


Figure 3.14: Accuracy of AutoEncoder model – City day & Taiwan

Loss:

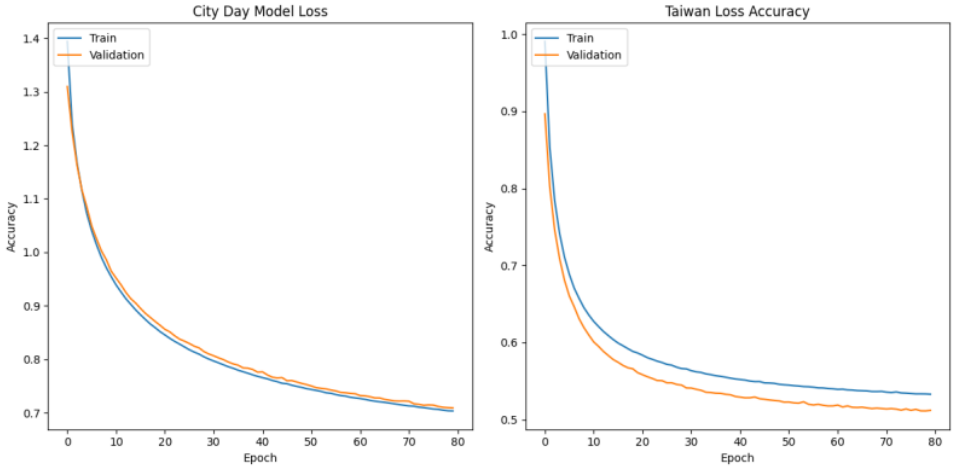


Figure 3.15: Loss of AutoEncoder model – City day & Taiwan

Evaluation of 5 models with 2 different datasets:

Table 3. 1: Evaluation of five models on dataset City day

City Day				
	precision	recall	f1-score	support
MLP	0.77	0.76	0.76	1159
LSTM	0.74	0.76	0.75	1159
BiLSTM	0.73	0.75	0.73	1159
GRU	0.75	0.74	0.73	1159
Autoencoder	0.74	0.74	0.73	1159

Table 3. 2: Evaluation of five models on dataset Taiwan

Taiwan				
	precision	recall	f1-score	support
MLP	0.80	0.79	0.78	1199
LSTM	0.75	0.75	0.75	1176
BiLSTM	0.73	0.75	0.73	1159
GRU	0.75	0.74	0.73	1159
Autoencoder	0.74	0.74	0.73	1159

CHAPTER 4. COMMAND LINE APPLICATION

4.1 Command Line Interface

4.1.1 Interface

The command line interface for testing the air pollution classification models is designed in a straightforward way to offer users a way to test the models with ease without having to interact with the models themselves. Upon execution, the CLI prompts the user to select the desired model they want to use.

```
Please select the model you wish to evaluate:
Type 1 for MLP
Type 2 for LSTM
Type 3 for BiLSTM
Type 4 for GRU
Type 5 for MLP AutoEncoder
Enter the number corresponding to the model:
```

Figure 4.1: The interface to select a model

And then, after selecting the desired model, the user is asked to input a number according to the dataset in which the selected model is trained on, and depending on the chosen dataset, there are different sample files located in different folders for the user to test.

Afterwards, the user must choose one of the available sample files to test the models on, there are 20 sample files for each dataset to test out.

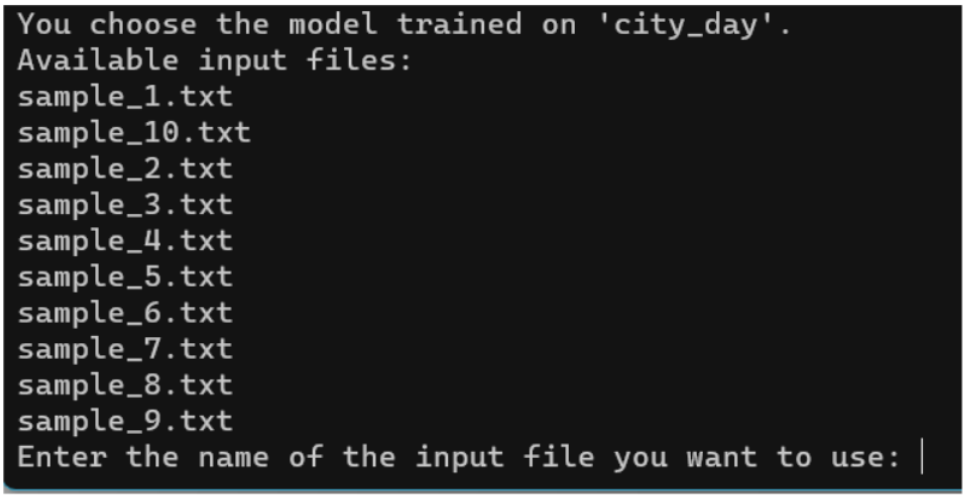


Figure 4.2: The interface to select a sample file for evaluation

4.1.2 Implementation Detail

After the user selects the desired model to evaluate through the command line interface (CLI), the corresponding Python script responsible for executing the model is invoked. This process is facilitated by the execute_script() function defined within the CLI script.The execute_script() function takes the selected model number as input and retrieves the corresponding Python script file from the specified folder (scripts_folder). The mapping between model numbers and script filenames is stored in a dictionary named script_files. Upon obtaining the filename, the function constructs the full path to the script file and executes it using the os.system() function.For instance, if the user chooses the first model (MLP) by typing '1', the execute_script() function will locate the MLP_script.py file in the scripts folder and execute it.

Following the execution of the selected model script, the user is prompted to select a sample file for evaluation. This interaction is facilitated by the script corresponding to the chosen model (e.g., MLP_script.py). Within this script, the user is presented with a list of available input files to choose from, based on the dataset the model was trained on. Subsequently, the selected input file is processed, and the model makes predictions on the provided data.

4.2 Deployment And Testing

4.2.1 Requirements

Before deploying and testing the models, it's essential to ensure that all necessary requirements are met. These requirements encompass both software and hardware components to ensure the smooth operation of the system.

Software Requirements:

Operating System Compatibility: The system should be compatible with the targeted operating system(s), such as Windows, macOS, or Linux distributions. Specify the minimum version of the operating system supported if applicable.

Python Environment: The system relies on specific Python libraries and dependencies. Ensure that the required Python version and libraries are installed. List the dependencies along with their versions for reproducibility.

TensorFlow and Keras: Install TensorFlow and Keras libraries to support machine learning model development and execution.

NumPy and Pandas: Ensure the presence of NumPy and Pandas libraries for data manipulation and preprocessing.Scikit-learn: If applicable, install Scikit-learn library for machine learning utilities and algorithms.

Hardware Acceleration Dependencies: If utilizing GPU acceleration, ensure the installation of CUDA and cuDNN libraries for TensorFlow GPU support.

Hardware Requirements:Processing Power: Specify the minimum hardware specifications required for model execution, including CPU speed, number of cores,

and RAM capacity. Consider the computational demands of the models and algorithms.

Storage Space: Estimate the required storage space for storing datasets, model files, logs, and other system-related data. Specify both temporary storage (RAM) and persistent storage (hard disk or SSD).

Testing Environment:

Test Data: Prepare a diverse set of test data representative of real-world scenarios to evaluate the models' performance comprehensively. We have prepared a sufficient amount of data for this purpose

4.2.2 Testing and Validation

In this section, we describe the testing and validation process conducted to evaluate the performance of the trained models. Three samples from each model's respective dataset were selected for testing. For each sample, the model's predictions were compared against the ground truth labels to assess its accuracy and effectiveness in classifying air quality levels.

The testing procedure involved the following steps:Model Selection: Five models were considered for testing, including Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), Bidirectional LSTM (BiLSTM), Gated Recurrent Unit (GRU), and Autoencoder.

Sample Selection: Two samples were randomly selected from each model's dataset. To simplify our process, only MLP and LSTM models are chosen for evaluation. These samples represent diverse scenarios and variations in air quality conditions.

Prediction Evaluation: Each selected sample was fed into its corresponding model for prediction. The model's predictions were compared against the actual air quality levels to determine the accuracy of the predictions.

Here are our testing results:

Dataset 1: city_day.csv:

MLP

Sample 1:

0.02270201,0.06965243,0.02248719,0.07873024,0.07558870,0.02435221,0.015
81028,0.07157984,0.10754201,0.00163315,0.00345062,0.06179508

True value:

Satisfactory

Result:

Figure 4.3: The result of the first sample tested on MLP (city_day.csv)

Sample 2:

0.03775379,0.08103732,0.01210500,0.12067771,0.05953689,0.11874789,0.019
76285,0.07002682,0.18499414,0.00239869,0.00221131,0.06179508

True value: Satisfactory

Result:

```
Choose MLP model (1 for model trained on data 'city_day.csv', 2 for model trained on data 'taiwan2015.csv'): 1
You choose the model trained on 'city_day'.
Available input files:
sample_1.txt
sample_10.txt
sample_2.txt
sample_3.txt
sample_4.txt
sample_5.txt
sample_6.txt
sample_7.txt
sample_8.txt
sample_9.txt
Enter the name of the input file you want to use: sample_3.txt
You chose input file: sample_3.txt
True class: Satisfactory
1/1 [==============================] - 0s 69ms/step
Predicted class index: Satisfactory
```

Figure 4.4: The result of the second sample tested on MLP (city_day.csv)

LSTM

Sample 1:

0.11318823,0.31545056,0.14340119,0.33104491,0.27095761,0.10509157,0.17807364,0.25737682,0.22762798,0.01806676,0.01730171,0.06179508

True value: Poor

Result:

```
Choose LSTM model (1 for model trained on data 'city_day.csv', 2 for model trained on data 'taiwan2015.csv'): 1
You choose the model trained on 'city_day'.
Available input files:
sample_1.txt
sample_10.txt
sample_2.txt
sample_3.txt
sample_4.txt
sample_5.txt
sample_6.txt
sample_7.txt
sample_8.txt
sample_9.txt
Enter the name of the input file you want to use: sample_6.txt
You chose input file: sample_6.txt
True class: Poor
1/1 [==============================] - 0s 397ms/step
Predicted class index: Poor
```

Figure 4.5: The result of the first sample tested on LSTM (city_day.csv)

Sample 2:

0.56569247,0.73297601,0.28172462,0.44013353,0.37943485,0.16482395,0.05387976,0.22010448,0.18733880,0.01814331,0.13469576,0.13090129

True value: Severe

Result:

```
Choose LSTM model (1 for model trained on data 'city_day.csv', 2 for model trained on data 'taiwan2015.csv'): 1
You choose the model trained on 'city_day'.
Available input files:
sample_1.txt
sample_10.txt
sample_2.txt
sample_3.txt
sample_4.txt
sample_5.txt
sample_6.txt
sample_7.txt
sample_8.txt
sample_9.txt
Enter the name of the input file you want to use: sample_10.txt
You chose input file: sample_10.txt
True class: Severe
1/1 [==============================] - 0s 406ms/step
Predicted class index: Severe
```

Figure 4.6: The result of the second sample tested on LSTM (city_day.csv)

Dataset 2: taiwan2015.csv

MLP

Sample 1:

0.01449275,0.02758621,0.19623461,0.01017355,0.02513661,0.07014028

True value: Good

Result:

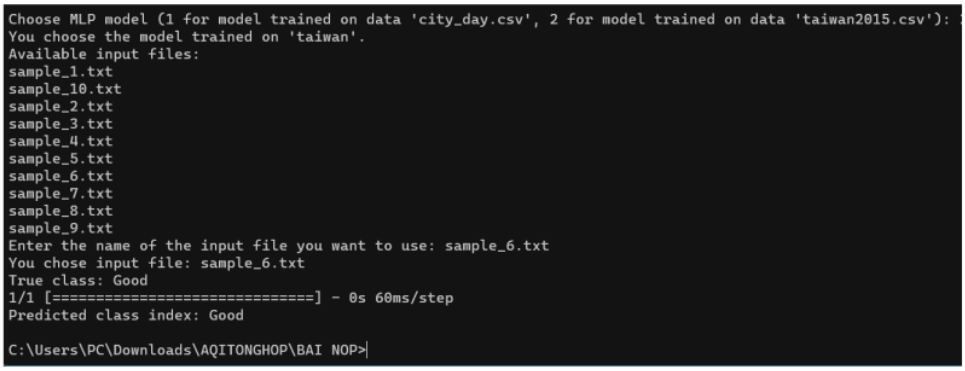


Figure 4.7: The result of the first sample tested on MLP (taiwan2015.csv)

Sample 2:

0.05636071,0.07910751,0.50760319,0.00837822,0.03497268,0.12424850

True value: Satisfactory

Result:

Figure 4.8: The result of the second sample tested on MLP (taiwan2015.csv)

LSTM

Sample 1:

0.07085346,0.13995943,0.11658219,0.00937562,0.03169399,0.20440882

0.03220612,0.06693712,0.28312817,0.00957510,0.02950820,0.14428858

0.28985507,0.41582150,0.84069515,0.01555955,0.05027322,0.11222445

0.01449275,0.02758621,0.19623461,0.01017355,0.02513661,0.07014028

0.10305958,0.12373225,0.06734251,0.00498703,0.01748634,0.08617234

0.05636071,0.07910751,0.50760319,0.00837822,0.03497268,0.12424850

0.11272142,0.10344828,0.23244026,0.00797925,0.01639344,0.07214429

0.05152979,0.10344828,0.01810282,0.00917614,0.02513661,0.19038076

0.33655395,0.47261663,0.84793628,0.01176940,0.02950820,0.09819639

0.08051530,0.10344828,0.01882694,0.00658288,0.02185792,0.10220441

0.04830918,0.06288032,0.28312817,0.00738081,0.03278689,0.10020040

0.08051530,0.11156187,0.01448226,0.00718133,0.02076503,0.10821643

0.14975845,0.22515213,0.02244750,0.01037303,0.01420765,0.09018036

0.06763285,0.06693712,0.14554671,0.00797925,0.01967213,0.06012024

0.58615137,0.78904665,0.23968139,0.01236784,0.03387978,0.16232465

0.01288245,0.02758621,0.15278783,0.00578496,0.02185792,0.06613226

0.04830918,0.03894523,0.39898624,0.01356473,0.03715847,0.06613226

0.29468599,0.32251521,0.65242578,0.01196888,0.03278689,0.11422846

0.42512077,0.51724138,0.71759594,0.01416318,0.03497268,0.13627255

0.19323671,0.22109533,0.03113686,0.01396369,0.04262295,0.15430862

0.02093398,0.02880325,0.34829833,0.00997407,0.02841530,0.07414830

0.28985507,0.29411765,0.21071687,0.01137044,0.02841530,0.17034068

0.06280193,0.14807302,0.26864591,0.00658288,0.01311475,0.08416834

0.15942029,0.35496957,0.53656770,0.01595851,0.04153005,0.08617234

True value: Moderate

Result:

```
Choose LSTM model (1 for model trained on data 'city_day.csv', 2 for model trained on data 'taiwan2015.csv'): 2
You choose the model trained on 'taiwan'.
Available input files:
sample_1.txt
sample_10.txt
sample_2.txt
sample_3.txt
sample_4.txt
sample_5.txt
sample_6.txt
sample_7.txt
sample_8.txt
sample_9.txt
Enter the name of the input file you want to use: sample_3.txt
You chose input file: sample_3.txt
True class: Moderate
1/1 [==============================] - 0s 397ms/step
Predicted class index: Moderate
```

Figure 4.9: The result of the first sample tested on LSTM (taiwan2015.csv)

Sample 2:

0.01449275,0.02758621,0.19623461,0.01017355,0.02513661,0.07014028

0.10305958,0.12373225,0.06734251,0.00498703,0.01748634,0.08617234

0.05636071,0.07910751,0.50760319,0.00837822,0.03497268,0.12424850

0.11272142,0.10344828,0.23244026,0.00797925,0.01639344,0.07214429

0.05152979,0.10344828,0.01810282,0.00917614,0.02513661,0.19038076

0.33655395,0.47261663,0.84793628,0.01176940,0.02950820,0.09819639

0.08051530,0.10344828,0.01882694,0.00658288,0.02185792,0.10220441

0.04830918,0.06288032,0.28312817,0.00738081,0.03278689,0.10020040

0.08051530,0.11156187,0.01448226,0.00718133,0.02076503,0.10821643

0.14975845,0.22515213,0.02244750,0.01037303,0.01420765,0.09018036

0.06763285,0.06693712,0.14554671,0.00797925,0.01967213,0.06012024

0.58615137,0.78904665,0.23968139,0.01236784,0.03387978,0.16232465

0.01288245,0.02758621,0.15278783,0.00578496,0.02185792,0.06613226

0.04830918,0.03894523,0.39898624,0.01356473,0.03715847,0.06613226

0.29468599,0.32251521,0.65242578,0.01196888,0.03278689,0.11422846

0.42512077,0.51724138,0.71759594,0.01416318,0.03497268,0.13627255

0.19323671,0.22109533,0.03113686,0.01396369,0.04262295,0.15430862

0.02093398,0.02880325,0.34829833,0.00997407,0.02841530,0.07414830

0.28985507,0.29411765,0.21071687,0.01137044,0.02841530,0.17034068

0.06280193,0.14807302,0.26864591,0.00658288,0.01311475,0.08416834

0.15942029,0.35496957,0.53656770,0.01595851,0.04153005,0.08617234

0.02737520,0.05476673,0.10209993,0.01037303,0.03169399,0.09819639

0.05636071,0.06693712,0.03475742,0.00538600,0.01311475,0.05811623

0.06924316,0.05882353,0.24692252,0.00758029,0.03825137,0.05010020

True value: Good

Result:

```
Choose LSTM model (1 for model trained on data 'city_day.csv', 2 for model trained on data 'taiwan2015.csv'): 2
You choose the model trained on 'taiwan'.
Available input files:
sample_1.txt
sample_10.txt
sample_2.txt
sample_3.txt
sample_4.txt
sample_5.txt
sample_6.txt
sample_7.txt
sample_8.txt
sample_9.txt
Enter the name of the input file you want to use: sample_6.txt
You chose input file: sample_6.txt
True class: Good
1/1 [==============================] - 0s 399ms/step
Predicted class index: Good
```

Figure 4.10: The result of the second sample tested on LSTM (taiwan2015.csv)

CHAPTER 5. CONCLUSION

5.1 Conclusion

We have created and assessed many machine learning models for the classification of air quality in this work. Real-world air quality data was used for training and testing a variety of models, including the Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), Bidirectional LSTM (BiLSTM), Gated Recurrent Unit (GRU), and Autoencoder. We have learned more about the capabilities and efficacy of each model in determining the various levels of air quality through extensive testing and validation.

The results obtained from the testing phase indicate that all models perform relatively the same, with the exception of the GRU model, which is our worst model in terms of performance. Overall, the developed models demonstrate promising performance in accurately predicting air quality levels.

However, it's important to recognize the constraints and difficulties that arose during the process of developing and testing. First, choosing appropriate datasets for training and assessment proved to be difficult. There are very few high-quality air quality datasets available.

Preprocessing the chosen datasets presented still another important difficulty. Additional data cleaning and preparation procedures were necessary because the raw air quality data frequently had outliers, missing values, and inconsistencies.

Furthermore, it took a lot of work to choose and refine five deep learning models for the classification of air quality. The development process was made more complex by the need to carefully evaluate the architecture, hyperparameters, and training procedures for each model.

5.2 Future Development

We are committed to further refining our existing models to improve their performance and robustness. This refinement process will involve conducting in-depth analyses of model architectures, hyperparameters, and training strategies to optimize predictive accuracy and generalization capabilities.

In line with our commitment to democratizing air quality monitoring, we plan to develop a comprehensive application dedicated to air quality classification. This application will feature a user-friendly interface, allowing users to easily input air quality data and obtain accurate classification results. Furthermore, the application will encompass a diverse array of machine learning models, providing users with the flexibility to test and compare different models for their specific use cases.

REFERENCES

Sitarz, M. (2011). *Extending F1 metric, probabilistic approach*. Retrieved from arxiv: https://arxiv.org/abs/2210.11997

Yann, L. (2015). *Deep Learning*. Retrieved from arxiv: https://www.researchgate.net/publication/277411157_Deep_Learning

DA.32_TrinhHungCuong_520K0140_520C0001.docx

BÁO CÁO ĐỘC SÁNG

6%	5%	10%	5%
CHỈ SỐ TƯƠNG ĐỒNG	NGUỒN INTERNET	ẤN PHẨM XUẤT BẢN	BÀI CỦA HỌC SINH

NGUỒN CHÍNH

1	fastercapital.com Nguồn Internet	1%
2	opendata.uni-halle.de Nguồn Internet	1%
3	Gypsy Nandi. "Principles of Soft Computing Using Python Programming", Wiley, 2023 Xuất bản	1%
4	Submitted to Ton Duc Thang University Bài của Học sinh	1%
5	Submitted to University of KwaZulu-Natal Bài của Học sinh	1%
6	repository.unika.ac.id Nguồn Internet	1%
7	Submitted to University of Melbourne Bài của Học sinh	1%
8	iq.opengenus.org Nguồn Internet	1%
9	"Artificial Neural Networks and Machine Learning – ICANN 2018", Springer Science	1%

and Business Media LLC, 2018
Xuất bản

10 Submitted to Institute of Aeronautical Engineering (IARE) 1%
Bài của Học sinh

Loại trừ Trích dẫn	Mở	Loại trừ trùng khớp	< 1%
Loại trừ mục lục tham khảo	Mở		