EMBRY-RIDDLE
Aeronautical University
SCHOLARLY COMMONS

Doctoral Dissertations and Master's Theses

Summer 7-10-2023

Neural Network Models for Generating Synthetic Flight Data

Nathaniel Sisson
Embry-Riddle Aeronautical University, sissonn@my.erau.edu

Follow this and additional works at: https://commons.erau.edu/edt

Part of the Other Aerospace Engineering Commons

Scholarly Commons Citation

Sisson, Nathaniel, "Neural Network Models for Generating Synthetic Flight Data" (2023). *Doctoral Dissertations and Master's Theses*. 762.
https://commons.erau.edu/edt/762

This Thesis - Open Access is brought to you for free and open access by Scholarly Commons. It has been accepted for inclusion in Doctoral Dissertations and Master's Theses by an authorized administrator of Scholarly Commons. For more information, please contact commons@erau.edu.

To an American Pitbull Terrier named King. King, you took a piece of my heart with you when you left. You were my dog, my boy, my best friend. But I hope now that your days are filled with endless running and playing. I long for the day when we will be reunited. Until then, never a day will go by that I do not think about you. I love you King, and I will forever.

ACKNOWLEDGMENTS

I am grateful to my advisor Dr. Hever Moncayo and my committee members Dr. Richard Prazenica and Dr. Richard Stansbury for their role in helping me complete this meaningful endeavour.

ABSTRACT

Flight test data is a valuable resource used in many aerospace applications. However, procuring a sufficiently large database of flight test data poses several challenges. Nominal flight tests can be expensive and time-consuming and require much post-processing depending on the availability of sensors and the quality of the sensor output. Flight test performed outside of nominal flight conditions, or flight tests in which failures are introduced, add to the inherent risk and danger associated with flight tests. The most popular alternative to flight test, numerical simulations, may fail to fully capture all non-linear behavior. While flight tests will always be required, a method for augmenting an existing database of flight test data with synthetically generated data could help alleviate some of the aforementioned challenges. Over the past few decades, generative machine learning has a emerged as a popular tool for data augmentation. In this thesis, several neural network architectures were investigated as methods of generating synthetic flight data that is consistent with the aircraft dynamics.

TABLE OF CONTENTS

ACKNOWLEDGMENTS i

ABSTRACT ii

LIST OF FIGURES vi

LIST OF TABLES vii

NOMENCLATURE viii

1 Introduction 1

1.1 Thesis Objective 2

1.2 Thesis Outline 3

2 Neural Network Background 4

2.1 Neural Network Architecture 4

2.1.1 Multi-Layer Perceptron Neural Network 5

2.1.2 Recurrent Neural Network 7

2.1.3 Convolutional Neural Network 12

2.2 Neural Network Training and Optimization 14

2.2.1 Data Pre-processing 14

2.2.2 Training Process 16

2.2.3 Hyper-parameter Tuning 18

2.3 Composite Neural Network Models 18

2.3.1 Variational Auto-Encoder 19

2.3.2 Generative Adversarial Networks 21

3 Architectures 23

3.1 MLP Generative Adversarial Network without inputs 23

3.2 MLP Generative Adversarial Network with inputs 24
3.3 LSTM Generative Adversarial Network 25
3.4 CNN Generative Adversarial Network 26
3.5 Variational Auto Encoder 27

4 Implementation & Results 29
4.1 Simulink AIRLIB model 29
4.2 Python Code 31
4.3 MLP Generative Adversarial Network without inputs 32
4.4 MLP Generative Adversarial Network with inputs 34
4.5 LSTM Generative Adversarial Network 39
4.6 CNN Generative Adversarial Network 40
4.7 Variational Auto Encoder 42

5 Conclusions and Future Work 44

REFERENCES 46

LIST OF FIGURES

Figure **Page**

2.1 Simple neural network architecture. 4

2.2 Data transformed by neuron. 5

2.3 Data transformation between layers. 6

2.4 Single RNN Cell 8

2.5 RNN cell unrolled 9

2.6 Single LSTM Cell 10

2.7 LSTM forget gate 10

2.8 LSTM input gate 11

2.9 LSTM ouput gate 12

2.10 2-D convolution on 2-D matrix 13

2.11 Structure of basic Auto Encoder 19

2.12 Structure of Variational Auto Encoder 20

2.13 Structure of basic GAN 21

4.1 AIRLIB simulink file used to generate training data. 30

4.2 Example of control inputs given to simulation. 30

4.3 Sample of states from simulation. 31

4.4 Sample of generated data from MLP GAN. 33

4.5 Loss curves from MLP GAN. 34

4.6 Correlations of generated states from MLP GAN. Red is the generated data and the blue is the training data. 35

4.7 Sample of generated data from MLP-GANi. 35

4.8 Loss curves from MLP-GANi. 36

4.9 Correlations of generated states from MLP-GAN. Red is the generated data and the blue is the training data. 37

4.10 Accuracy of the MLP-GANi generated state variables when compared to the true dynamics. 38
4.11 Sample of generated data from LSTM-GAN. 39
4.12 LSTM-GAN loss curves. 40
4.13 Sample of generated data from CNN-GAN. 41
4.14 Loss curves from CNN-GAN. 41
4.15 Sample of generated data from VAE-GAN. 42
4.16 Loss curves from VAE-GAN. 43

LIST OF TABLES

Table		Page
3.1	MLP-GAN Structure	23
3.2	MLP-GAN Hyper-parameters	23
3.3	MLP-GANi structure	25
3.4	MLP-GANi Hyper-parameters	25
3.5	LSTM-GAN structure	26
3.6	LSTM-GAN Hyper-parameters	26
3.7	CNN-GAN structure	27
3.8	CNN-GAN Hyper-parameters	27
3.9	VAE structure	28
3.10	VAE Hyper-parameters	28
4.1	MLP-GAN Generated Data Statistics	33
4.2	MLP-GANi Generated Data Statistics	37

NOMENCLATURE

AE Auto Encoder

ANN Artificial Neural Network

BCE Binary Cross-Entropy

GAN Generative Adversarial Network

GRU Gated Recurrent Unit

$LSTM$ Long Short Term Memory

MAE Mean Absolute Error

ML Machine Learning

$MLP-GAN$ Multi Layer Perceptron Generative Adversarial Network

$MLP-GAN$ Multi Layer Perceptron Generative Adversarial Network

$MLP-GANi$ Multi Layer Perceptron Generative Adversarial Network with inputs

$MLP-GANi$ Multi Layer Perceptron Generative Adversarial Network with inputs

MLP Multi Layer Perceptron

MSE Mean Squared Error

NLP Natural Language Processing

RNN Recurrent Neural Network

SGD Stochastic Gradient Descent

VAE Variational Auto Encoder

$WGAN$ Wasserstein Generative Adversarial Network

$WGAN$ Wasserstein Generative Adversarial Network

1 Introduction

Many published articles and journals underscore the importance and usefulness of flight test data. Aerodynamic parameter estimation [1] is a classic example of this where specific relationships between aerodynamic forces/moments and state variables are extracted from the test data. Whalen and Bragg [2] used flight test data to attempt to predict the onset of wing icing and quantify its effect on aircraft performance. [3] explored using flight test data as a means to calculate aircraft moments of inertia.

Repeated flight testing for the purpose of accumulating a sufficiently large amount of data can become expensive very quickly. In some cases, the value of the flight test data does not justify the cost of performing that flight. Some flight tests are performed specifically for the purpose of obtaining non-nominal or failure flight data. This includes flying outside of the aircraft's flight envelope or introducing small magnitude failures onboard. These types of flights are inherently dangerous such that it is not feasible to perform many of them.

Numerical computer simulations are the most common alternative to performing flight tests. These simulations require a purely mathematical model of the aircraft dynamics that not only include the aircraft equations of motion, but also engine and actuator dynamics modeling, atmospheric modeling, drag and interference modeling, and so on. These mathematical models are often parameterized by approximations of aerodynamic coefficients and derivatives requiring look-up tables to account for changes in flight condition. Many aspects of aircraft flight dynamics are highly non-linear and next to impossible to model mathematically. Considering this, numerical simulations fall short when it comes to providing realistic flight data that is on par with true flight data because it is limited by our ability (or inability) to perfectly model every single physical phenomena that an aircraft experiences in flight.

Considering the difficulty in obtaining a large amount of flight test data, this becomes a candidate data augmentation problem. In general, data augmentation refers to a problem of limited data and the unconventional methods used to obtain more data. The problem of lack of sufficient data affects many fields and disciplines. In the medical field, access

to patient medical data is highly controlled as it is personal and protected [4]. Access to financial data faces similar challenges [5]. In some cases, the quantity of the data is not the problem itself, but it is the imbalance of the data that poses issues and requires a data augmentation solution. Machine learning (ML) is currently the primary focus for data augmentation solutions. Machine learning refers to any algorithm that learns to performs a certain task by being trained strictly on data and not by adhering to a strict set of instructions. Machine learning models have been used to perform tasks such as classifying and clustering, regression, sequence prediction, modeling input/output relations and, more recently, data generation. ML models learn primarily by seeing both input and output data and making inferences on that data. The term ML model refers to the specific architecture of the algorithm with multiple ML models performing similar tasks, so ML models and ML algorithms are not mutually exclusive. One of the oldest ML models is the artificial neural network (ANN). This name is derived from circuit of neurons in the brain (referred to as a biological neural network) from which the ANN is modeled after. Throughout this thesis, the term neural network (NN) will replace the term ANN and does not refer the biological definition. In it's simplest form, a neural network finds a mathematical mapping from the input data to the output data in the form of weights, biases, and activation functions.

1.1 Thesis Objective

This research explores the use of artificial neural networks as a tool for solving the data augmentation problem for flight data. NNs have been shown to be an effective tool for the task generating data such as images, but are greatly challenged by specific types of data, particularly time-dependent data. Various NN architectures are tested and evaluated as means of generating synthetic but realistic and useful flight data. If a NN architecture is effective at learning the complicated dynamic relationships among flight variables and between control inputs and flight variables, this could possibly open the door to generating failure flight data that might not be possible to obtain otherwise. Such methods of flight data generation will never exactly replicate true flight data and should not be considered as

a primary source for such data. At most, flight data generated by machine learning will serve to increase the resolution of existing data sets for the purposes of reducing the frequency of flight testing and thereby saving time, decreasing cost, and mitigating risk.

1.2 Thesis Outline

This thesis will first provide a brief but thorough summary of all neural network architectures explored, Chapter 2. Chapter 3 will cover methods for generating the training data for all NN models. In Chapter 4, the training process of each NN is discussed as well as the results and evaluation for each. Final conclusions and the possibility of further advancement in this research are discussed in Chapter 5.

2 Neural Network Background

As mentioned in the introduction, the neural network is a mapping from input to output. They learn to approximate a function that is the relationship between input data and output data through a particular training algorithm [6]. No prior knowledge of this function's structure is required as a series of weights and hyper parameters are learned from a chosen loss function to provide the most correct mapping from input to output. NNs are algorithms implemented in any high-level programming language. The most popular language used for NN implementation and training is Python. Python is a popular open source, object-oriented programming language. There exists a plethora of libraries and APIs specifically written for the purpose of making NN training efficient and user-friendly [7–9]. This chapter aims to provide a thorough explanation of the NN structure and the training process.

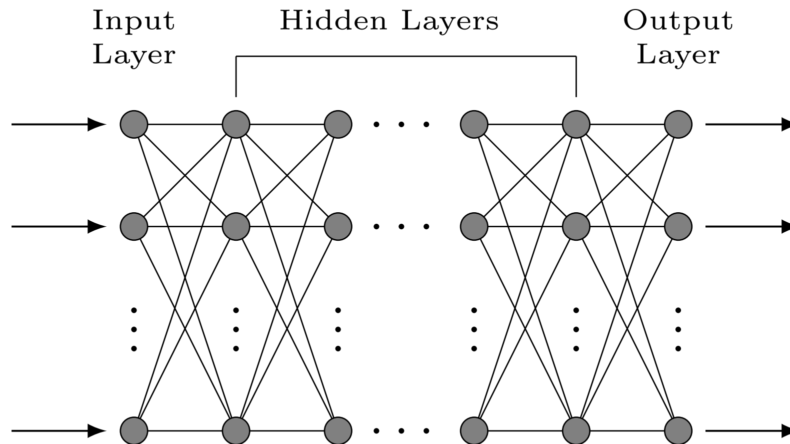

Figure 2.1 Simple neural network architecture.

2.1 Neural Network Architecture

Figure 2.1 illustrates the general architecture of an NN. The NN accepts input data at the input layer and the data is propagated from layer to layer (usually unidirectionally) until it reaches the output layer. During training, this is referred to as the forward pass. A reverse in the direction of data flow also happens during the training process- called back-propagation. When a trained NN is used to make predictions, provided classifications, etc, the data will always propagate from input to output. The NN may have one to many internal "states"

represented by the output of any given hidden layer (any layer that is neither the input layer nor the output layer), however, the output is generally the only useful data. All NNs vary in size, structure, and complexity, however each NN model explored in this thesis follow similar structures with the main differentiators being the type of the layer and the type of transformations performed on the data as it is passed between two layers.

2.1.1 Multi-Layer Perceptron Neural Network

The Multi-Layer Perceptron Neural Network (MLP) is a classical NN architecture and is the original network created. It's layers consist of neurons, or nodes, which are responsible for performing operations on data passed to it from nodes of the previous layer. The data is transformed by applying a weight, w, to the input of each neuron, multiplicatively, and then adding a bias, b, to that result, Equation 2.1. Furthermore this output is passed through an activation function, σ, to limit the neuron output to a specific range. (Note, in the case of Multi-layer Perceptron networks, the σ notation does not strictly represent the sigmoid function. Originally, the sigmoid function was the only activation function used in MLPs, but now, by convention, the notation σ simply means any activation function.) Figure 2.2 illustrates these operations. These parameters, w and b, are continually updated during the training process until their values provide the most accurate input-to-output mapping.

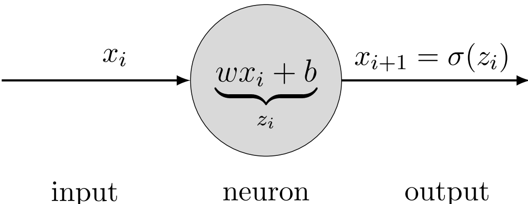

Figure 2.2 Data transformed by neuron.

$$x_{i+1} = \sigma\left(wx_i + b\right) \tag{2.1}$$

Generally, each neuron accepts inputs from all neurons of the previous layer and also sends its outputs to all neurons in the next layer. Because of the matrix-like structure of

the network and the fact that all neurons of adjacent layers are connected, the mathematical operations can be represented as tensor operations. All MLP NNs in this work are two-dimensional, so the tensor operations reduce to matrix operations.

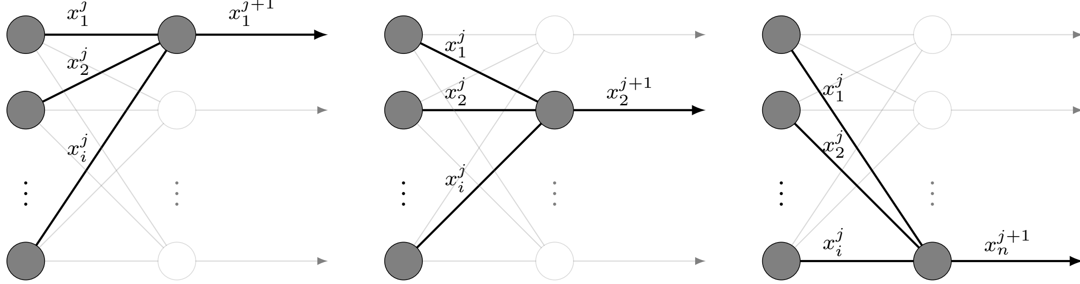

Figure 2.3 Data transformation between layers.

Figure 2.3 illustrates how each neuron accepts inputs from all neurons in the previous layer. This can be expressed mathematically with Equations 2.2.

$$
\begin{aligned}
x_1^{j+1} &= \sigma\left(\begin{bmatrix} w_{1\to 1} & w_{2\to 1} & \dots & w_{i\to 1}\end{bmatrix}\begin{bmatrix} x_1^j \\ x_2^j \\ \vdots \\ x_i^j \end{bmatrix} + b_1\right) \\
x_2^{j+1} &= \sigma\left(\begin{bmatrix} w_{1\to 2} & w_{2\to 2} & \dots & w_{i\to 2}\end{bmatrix}\begin{bmatrix} x_1^j \\ x_2^j \\ \vdots \\ x_i^j \end{bmatrix} + b_2\right) \\
x_n^{j+1} &= \sigma\left(\begin{bmatrix} w_{1\to n} & w_{2\to n} & \dots & w_{i\to n}\end{bmatrix}\begin{bmatrix} x_1^j \\ x_2^j \\ \vdots \\ x_i^j \end{bmatrix} + b_n\right)
\end{aligned}
\tag{2.2}
$$

Combining theses equations together yields a matrix Equation 2.3 which can be succinctly

expressed as Equation 2.4

$$\begin{bmatrix} x_1^{j+1} \\ x_2^{j+1} \\ \vdots \\ x_n^{j+1} \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{1\to1} & w_{2\to1} & \dots & w_{i\to1} \\ w_{1\to2} & w_{2\to2} & \dots & w_{i\to2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1\to n} & w_{2\to n} & \dots & w_{i\to n} \end{bmatrix} \begin{bmatrix} x_1^j \\ x_2^j \\ \vdots \\ x_i^j \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right) \tag{2.3}$$

$$\underline{x}^{j+1} = \sigma \left(W\underline{x}^j + \underline{b} \right) \tag{2.4}$$

In these equations, the superscript, j, indexes the layers, the subscript, i, indexes each neuron in the previous layer, and the subscript, n, indexes the neurons in the next layer.

MLP neural networks are very effective for tasks involving strictly numeric and stationary data (i.e. non-time dependent, non-sequential, etc). However, most real world data is non-stationary data. We are often concerned with how certain data changes over time. MLPs have no way of accounting for non-stationarity because data flows in one direction only during prediction. There are no feedback loops or other types of mechanisms for retaining certain states in memory. Having some sort of memory sate is required for learning sequential data. It is very difficult to make a prediction about the next step of a sequence if there is no knowledge of the previous steps. Therefore, other neural network architectures must be considered.

2.1.2 Recurrent Neural Network

The Recurrent Neural Network [10] (RNN) is an architecture designed to handle sequential or non-stationary data. RNNs are one of the most popular architectures and have seen widespread use primarily in natural language processing[11] (NLP) and speech/text recognition. Anyone who has ever talked to Siri or Google assistant has seen the capability of RNNs. RNNs have layers just like MLPs, however, the layers contain cells which process information quite differently compare to MLP nodes. As described earlier, an MLP node processes data by multiplying the input by a weight, adding a bias, and passing through

an activation function to form the output. An RNN cell performs a similar computation but this is performed recursively with the output of the cell's computation at each iteration being fed back to the cell which is combined with the input data to form the total input of the cell. The cell's output at each iteration is referred to as the hidden state. The number of times the cell preforms this feedback computation is equivalent to the length of the input data sequence. At step t in the sequence, the RNN cell is receiving two inputs, x_t and h_{t-1}, where x_t is the t$^{\text{th}}$ data point in the sequence and h_{t-1} is the output of the cell's previous iteration. Figure 2.4 illustrates this data flow through a cell. This is the feedback mechanism that allows the RNN to consider previous steps in a sequence in order to predict the next step. It is easier to visualize this flow by graphically unrolling the cell. This is illustrated in Figure 2.5. The superscript, j, indexes the layer. If $j = 1$ (i.e. it is the first layer), then h_n^{j-1} is initialized to zero. The subscript n represents the total length of the sequence. Note that Figure 2.5 does not represent multiple cells, but rather a single cell which iterates over a sequence of data. In Recurrent Neural Networks, the output of the last layer or last cell can either be the last hidden state, h_n, or it can be the entire sequence of hidden states, $[h_1, h_2, ..., h_n]$.

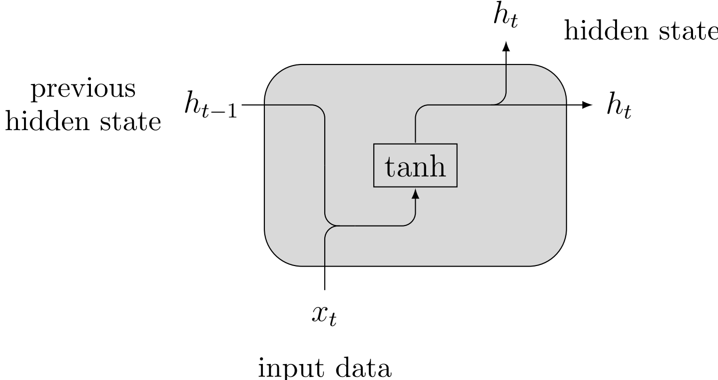

Figure 2.4 Single RNN Cell

Typically, this simple RNN described is not implemented in practice. The reason is while this architecture is capable of retaining previous information through the feedback loop, this is not a true memory because there is no internal memory state in the cell. This

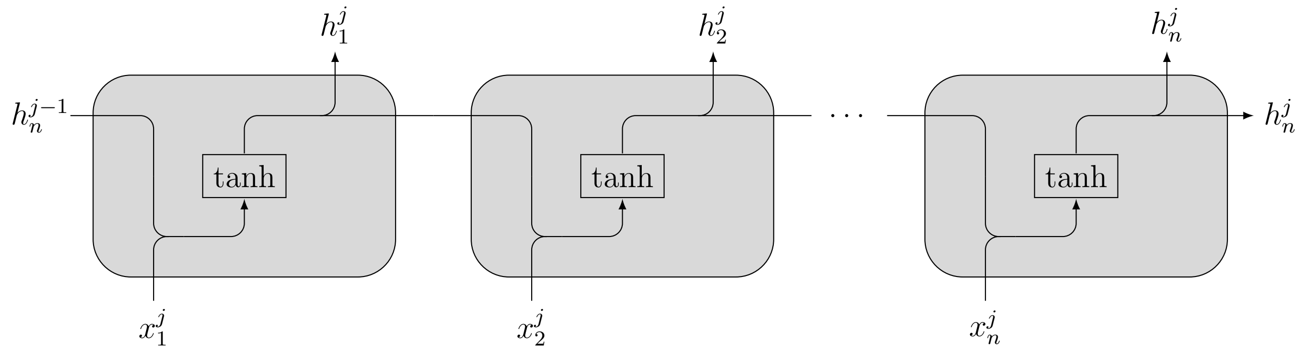

Figure 2.5 RNN cell unrolled

results in a short range memory that is limited to only a few time steps or sequence steps back. This is a disadvantage in some applications where information from many previous steps are required such as in natural language processing. Two adapted RNN models, Long Short Term Memory [12] (LSTM) and Gated Recurrent Unit [13] (GRU), were developed to increase the RNN's ability to store information from many previous sequence steps. These architectures share a similar external structure of a simple RNN cell, however, the internal processing of the data is much different. The research in this thesis only considered LSTMs as the RNN architecture. The LSTM cell is much more effective than a simple RNN cell at retaining information further back in the sequence. Unlike the simple RNN cell, the LSTM cell chooses which information to keep and which new information to add through a series of three "gates". Additionally, the LSTM cell considers a second hidden state, referred to as the cell state. This is the true memory of the LSTM. The cell state is primarily what is affected by the gates and the hidden state (the cell's output) is derived from this. Both are fed back to the cell's input as is the case with the simple RNN cell. Figure 2.6 illustrates the flow of information through the cell.

The task of the forget gate is to determine how much of the previously held information in memory is relevant and should be allowed to continue to propagate. This is accomplished by applying a weight and bias to the previous hidden state and the current input in the same manner as in the MLP neuron and then passing that result through a sigmoid activation

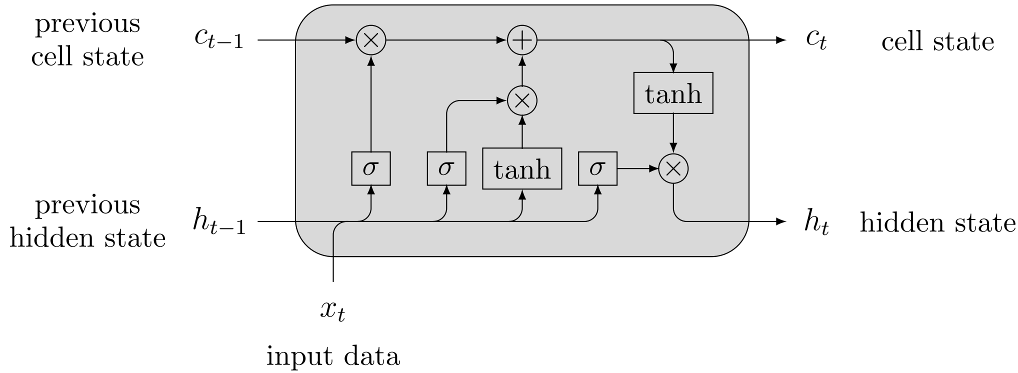

Figure 2.6 Single LSTM Cell

function. The sigmoid activation function restricts the output between 0 and 1 exclusively. This result is then multiplied by the previous cell state which effectively filters out information that the forget gate deems as no longer pertinent. Output values of the forget gate closer to one indicate that the cell state contains mostly relevant information and that is should be retained. Output values closer to 0 indicate that the cell state contains mostly non-useful information and that is should be discarded. Since this gate is accepting two separate inputs, x_t and h_{t-1}, two sets of weights are required, W_f and U_f, Equation 2.5.

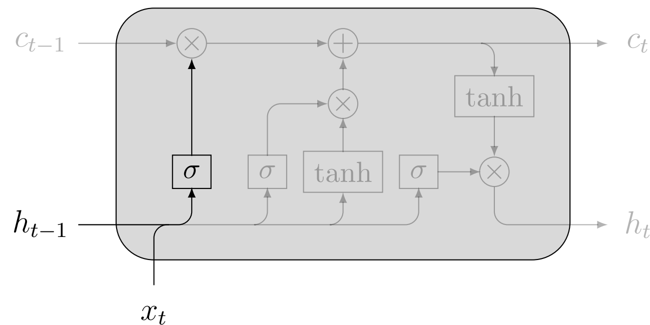

Figure 2.7 LSTM forget gate

$$f_t = \sigma \left(W_f x_t + U_f h_{t-1} + b_f \right) \tag{2.5}$$

The task of the input gate is to determine how much of the current data input is relevant and how much of it gets to be added to the current cell state. This is a two step process as the cell's previous hidden state and current data input is transformed by two operations of

weights, bias, and activation function. In the first operation, weights W_i and U_i are applied to x_t and h_{t-1} followed by a bias, b_i, and a sigmoid activation function, Equation 2.6. In the second operation, weights W_c and U_c are applied to x_t and h_{t-1} followed by a bias, b_c, and a hyperbolic tangent activation function, Equation 2.7. The output of the latter operation is often referred to as the candidate cell state, $\tilde{c}_t$. The hyperbolic tangent activation function restricts the output between -1 and 1 exclusively in manner similar to the sigmoid activation function. The two outputs are multiplied together and the product is what is added to the current cell state.

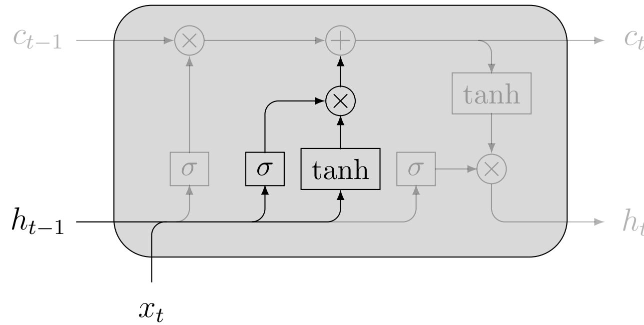

Figure 2.8 LSTM input gate

$$i_t = \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) \tag{2.6}$$

$$\tilde{c}_t = \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) \tag{2.7}$$

The output of the forget gate filters out unnecessary information from the previous cell state and the input gate filters out new information to add to the previous cell state. This results in the updated, current cell state, or cell memory, which will again be modified during the next iteration, Equation 2.8.

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{2.8}$$

In a manner similar to the other gates, the output gate determines what information the cell output should contain. Weights W_o and U_o are applied to x_t and h_{t-1} followed by a bias, b_o, and a sigmoid activation function, Equation 2.9. This is then multiplied by the cell state passed through a hyperbolic tangent activation function to form the cell's current hidden

state or current output.

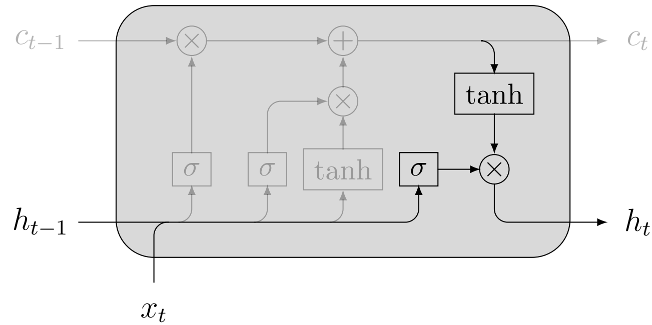

Figure 2.9 LSTM ouput gate

$$h_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) \tanh\left(c_t\right) \tag{2.9}$$

2.1.3 Convolutional Neural Network

The Convolutional Neural Network[14] (CNN) is an neural network architecture that has seen applications dealing almost exclusively with images. CNNs excel at learning spacial features of images. Variants of the CNN have been developed to perform all kinds of image processing and synthesis task. [–] Some CNNs have less popularly been applied to sequential or time-series data. Each layer in a CNN hold a weight tensor, usually a two-dimensional matrix referred to as a kernel or filter, that performs a convolving operation on the input data. These convolutional layers are often followed by other data processing layers such as pooling layers or reshaping layers. These layers are usually meant for reducing or increasing dimensions and do not involve trainable weights or biases. The input to each convolutional layer is a tensor of data. In image applications, this data tensor is at most three-dimensional (e.g an RGB image) and is larger than the weight matrix. The weight matrix slides, convolves, along the height, H_{in}, and width, W_{in}, of the input tensor performing a Hadamard product, element-wise multiplication, of the weight matrix and a submatrix of the data tensor with equivalent dimensions, H_k and W_k. The sum of all elements in the Hadamard product, plus a bias, passes through an activation function and becomes a new data value in the output tensor. Figure 2.10 illustrates how the weight matrix convloves over the data matrix.

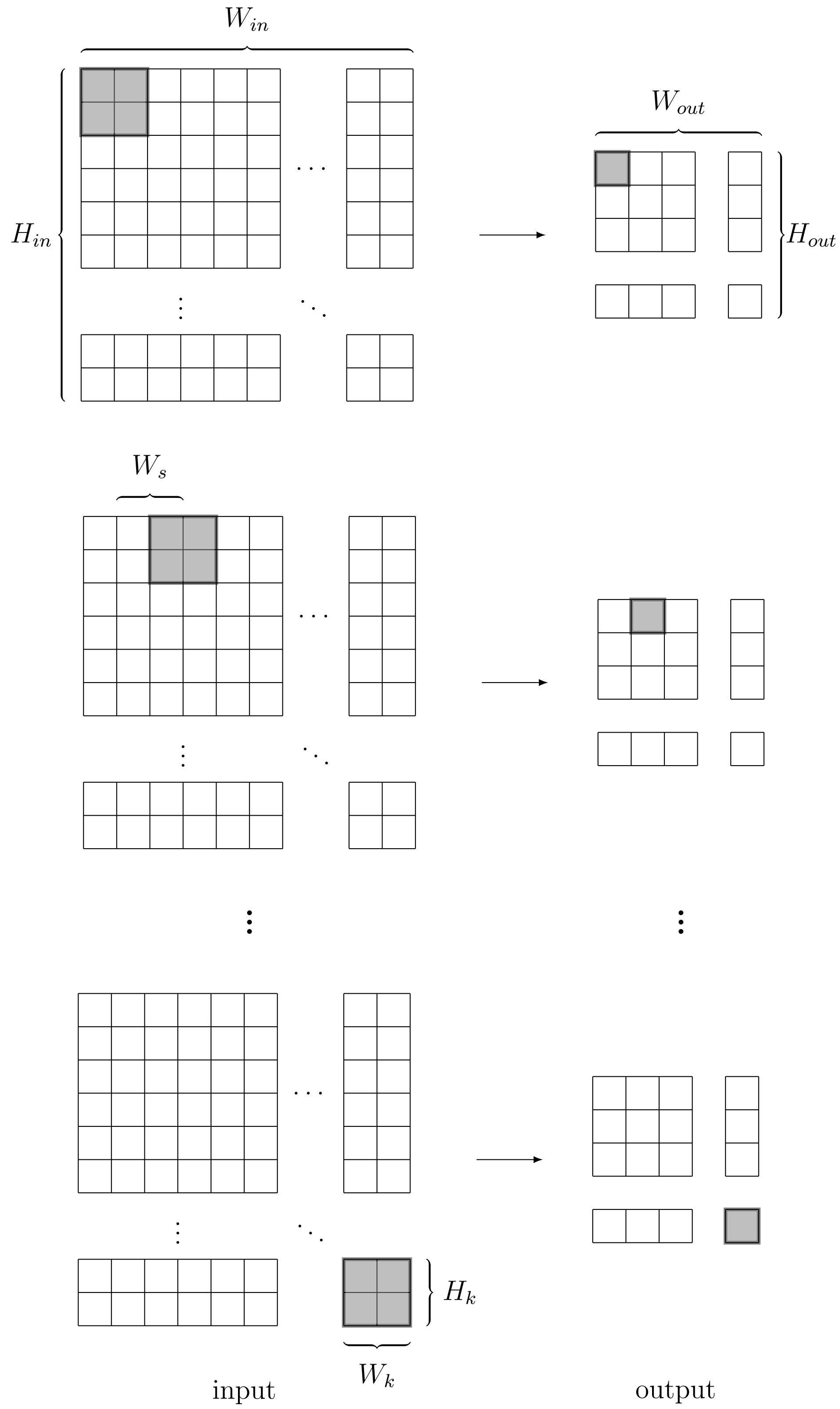

Figure 2.10 2-D convolution on 2-D matrix

The number of dimensions that the weight matrix slides along the input matrix in between each convolution is a varying parameter called the stride, H_s and W_s. The size of the weight matrix and the length of the stride determines the size of the output matrix, Equation 2.10. The notations H_p and W_p represent an optional padding (i.e. the adding of columns and rows of zeros to the input tensor) prior to the convolution. This padding operation is usually done for consistency of dimensions. The mathematical operations performed at each layer can be expressed in Equation 2.11. X^{in} represents the input data matrix, X^{out} represents the output matrix, K is the weight matrix for that layer, b represents the bias, and $\circ$ denotes the Hadamard product.

$$\begin{aligned} H_{out} &= \frac{H_{in} - H_k + 2H_p}{H_s} + 1 \\ W_{out} &= \frac{W_{in} - W_k + 2W_p}{W_s} + 1 \end{aligned} \tag{2.10}$$

$$\begin{gathered} X^{out}_{I,J} = \sigma\left(b + \sum_{m=1}^{H_k}\sum_{n=1}^{W_k} X^{in}_{(i:i+H_k-1,\, j:j+W_k-1)} \circ K\right) \\ i \in \{H_s I - (H_s - 1) \,|\, I \in \{1, 2, ..., H_{out}\}\} \\ j \in \{W_s J - (W_s - 1) \,|\, J \in \{1, 2, ..., W_{out}\}\} \end{gathered} \tag{2.11}$$

2.2 Neural Network Training and Optimization

2.2.1 Data Pre-processing

Before a training algorithm is implemented, the data must be pre-processed. The primary pre-processing technique is categorized based on the type of learning- supervised or unsupervised. Supervised learning requires the data set to be divided and labeled. Data is typically divided into three sets: training data, testing data, and validation data. Each one these sets generally contain two types of data, the input data and its corresponding labels or targets. Each of these three set serves a different purpose during the training. Unsupervised learning does not require the separation of the data into these three categories, but still

may require labels for the input data. Further pre-processing is dependent on the chosen architecture and task. For example, RGB images are stored as 3 two-dimensional matrices of numbers ranging from 0 to 255. Typically, these matrices are normalized between 0 and 1. For NLP tasks, words must be encoded to numbers.

For data that is already numeric, it is either normalized or standardized. Normalizing and standardizing serves two purposes. The first purpose is to map all features of the data to common scale, whereas before they could have been on different orders of magnitude. Neural network models generally learn faster because of this pre-processing technique. The second purpose is to speed up computation time. The computer will process operations on small numbers much faster than larger numbers. Although normalization and standardization are often conflated, they are very much different techniques. Normalization involves mapping an existing range of numbers to another, much smaller, range. This range is usually [0,1] or [-1,1] but can be any range. Equation 2.12 is shows how to normalize data between 0 and 1 and Equation 2.13 shows how to normalize between any arbitrary numbers $z_{\min}$ and $z_{\max}$. Standardization involves forcing the data set to have a normal distribution (i.e. mean of 0 and standard deviation of 1). Equation 2.14 shows how to standardize a dataset. The notation μ represents the mean of the data and σ represents the standard deviation of the data. The key difference between these two methods is that the shape of the data distribution remains the same in normalization and changes in standardization.

$$z = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{2.12}$$

$$z = \frac{z_{\max} - z_{\min}}{x_{\max} - x_{\min}} (x - x_{\min}) + z_{\min} \tag{2.13}$$

$$z = \frac{x - \mu}{\sigma} \tag{2.14}$$

Training data is also commonly batched. Batching refers to splitting the data into smaller "mini-batches". Instead of feeding the NN the entire dataset, the network is provided one

of these mini-batches. The entire training cycle is performed on this one mini-batch. The training cycle is repeated for the remainder of the mini-batches. The number of samples in each mini-batch is called the batch size. Adjusting the batch size is known to have a significant affect on training [15].

2.2.2 Training Process

The neural network training algorithm differs slightly from architecture to architecture, however, the basic algorithm is the same for all NN models. This process centers around the chosen loss function, sometimes referred to as a cost function. The first step of the training process involves a forward pass of a batch of the input data through the model to produce an output. Depending on the architecture, this output can be a prediction, classification, etc. When performing supervised training, this output is directly compared to some type of label, target, or ground truth. The loss function is performing this comparison. The most common examples of supervised loss functions include Mean Squared Error (MSE) and Mean Absolute Error (MAE). Generally the loss function should be a convex function, i.e. at least some local minima exists [16–18]. Quadratic loss functions such as MSE satisfy this requirement although it is not a strict requirement. When performing unsupervised training, the output is not directly compared to a ground truth. Instead the loss function makes inferences on the output, particularly about the distribution of the output data. Such loss functions involve statistical concepts such as maximum likelihood and cross-entropy. The most common of these loss function is the Binary Cross-Entropy (BCE) loss function [19]. In either case, supervised or unsupervised, the loss function provides a metric of how well the NN model fit the input data or how well the model made a prediction. Mathematically, this metric measures how accurate the values of the weights and biases in the model are (i.e. how accurate is the mapping from input to output).

The goal of the training is to find the values of the model parameters, the weights and biases, that provide the most accurate output which generally corresponds to finding a minimum in the loss function. The concept of finding this minimum, and the model parameters

associated with it, is no different than finding the minimum of a simple elementary math function. The gradient of the loss with respect to the weights and biases must be calculated and a process such as gradient descent is used to adjust the parameters in the direction of minimizing the loss. In models with many hidden layers, this derivative is a complicated, multi-dimensional function that requires the chain rule from calculus since there are many operations that separate the parameters from the loss value. The loss as a function of the model parameters is often referred to as the "loss landscape" and is impossible is visualize because of the high degree of dimensionality. Many papers are dedicated to understanding this landsacpe better [20–22]. Additionally, in the presence of nonlinear activation functions with non-continuous derivatives and with the number of trainable parameters possibly totalling in the millions, this is an intractable task to perform analytically. The solution to this problem is a computer science algorithm developed called automatic differentiation [23]. For every single calculation in the forward pass, numeric gradients are calculated and stored as metadata. Since the chain rule still applies, all of these recorded gradients are simply multiplied together in the appropriate order to produce the originally sought after gradients. This part of the training process is referred to as back propagation [24] because the multiplication of all of the chained gradients starts from the output layer and progresses in the opposite direction compared to the forward pass.

Usually, in practice, a complete gradient descent process is never used. Instead, a process called stochastic gradient descent [23] (SGD), or some variation of it, is implemented. The SGD is different because it only considers some of the weights and biases when calculating gradients, not all. The parameters that are considered are randomly chosen. The benefit for using SGD is usually faster training time. SGD and its variants are referred to as optimizers. Once the model parameters have been updated, the process repeats, beginning with forward pass of the next mini-batch of input data. This process repeats until the entire training dataset has been used. The number of cycles required to completely pass through the entire training dataset is referred to as a training epoch, and as many as thousands of training

epochs may be required to properly train the model.

2.2.3 Hyper-parameter Tuning

While the training algorithm focuses on adjusting the weights and biases of the model, other parameters relating to the training process or the structure of the model itself, called hyper-parameters, often need to be tuned. This tuning, however, is performed manually after each training session, i.e. complete number of training epochs performed. A list of hyper-parameters include the optimizer learning rate, data batch size, number and size of hidden layers, etc. A hyper-parameter is basically anything in your model or training algorithm that is tunable and not automatically learned from the training process. Proper hyper-parameter tuning is key to successful NN training. However there is very little intuition behind the tuning process and often these values are simply tuned by guessing until a positive effect is noticed after training. Many published papers provide tips on hyper-paramter tuning with most involving heuristic-based techniques [25, 26]. Very little research has been able to successfully explain hyper-parameter tuning in a theoretical manner and virtually no pragmatic methods of tuning exist. The research in this thesis does not focus much on hyper-parameter tuning and the values for the hyper-parameter were chosen through trial and error.

2.3 Composite Neural Network Models

Section 2.1 explained some of the common neural network architectures. These homogeneous models all have multiple layers of the same type. It is not uncommon, however, to see NN models that are comprised of two or more of these fundamental architectures combined. Such models are called composite models and may include a single model contain mixed layer types or multiple models sharing input and output data. A classic example regarding mixed layers is an image classification model. The input image is passed through several convolutional layers before passing through one or more MLP layers to produce a single numeric output. Other examples of composite models include Auto-Encoders (AE), Generative Adversarial Networks (GAN), and Time-GAN.

2.3.1 Variational Auto-Encoder

The Variational Auto Encoder [27] (VAE) is a variant of the Auto Encoder. Auto Encoders are composite models primarily used for dimension reduction. They are trained to find a mapping from the data space to a lower dimensional space, called the latent space, and also find the inverse of that mapping, the latent space back to the data space. Two neural networks, an encoder and decoder, are jointly trained to minimize a reconstruction loss. The input to the encoder is the training data and it's output is a lower dimensional space that is a compressed representation of the original data. The decoder takes a random sampling of the latent space as input and produces a reconstruction of the original data. The reconstruction loss is based on the difference between the original data and the reconstructed data. The two most common Auto Encoder loss functions are Mean Squared Error, Equation 2.15, and Cross-Entropy, Equation 2.16. Once an Auto Encoder is trained the latent space is sampled and passed through the decoder to provide and output.

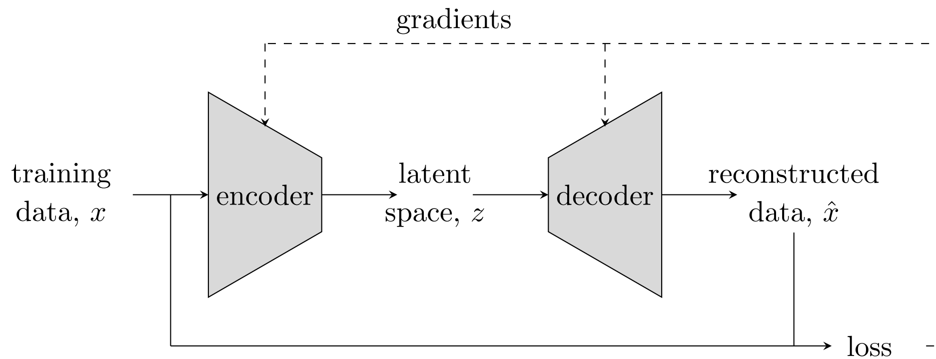

Figure 2.11 Structure of basic Auto Encoder

$$||x - \hat{x}||_2^2 \tag{2.15}$$

$$-\sum x \cdot \log(\hat{x}) \tag{2.16}$$

A major issue with the vanilla Auto Encoder lies is how the latent space is created. The structure of the latent space is not predetermined before training and could take on any type

of distribution depending on the training. With out a prior knowledge of the latent space, it is not possible to effectively sample from it to produce data. In fact, the Auto Encoder cannot produce new data- only data it has seen from the training dataset. The alternative is to regulate the structure of the latent space during training as in the Variational Auto Encoder.

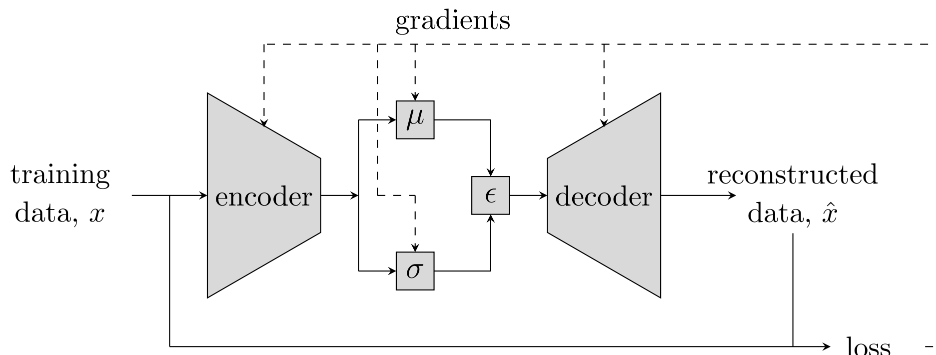

Figure 2.12 Structure of Variational Auto Encoder

$$z = \mu + \sigma\epsilon \tag{2.17}$$

$$||x - \hat{x}||_2^2 - \frac{1}{2}\sum\left(1 + \log\left(\sigma^2\right) - \mu^2 - \sigma^2\right) \tag{2.18}$$

$$-\sum x \cdot \log(\hat{x}) - \frac{1}{2}\sum\left(1 + \log\left(\sigma^2\right) - \mu^2 - \sigma^2\right) \tag{2.19}$$

In the VAE, the mean and variance of the latent space are trainable variables and are encouraged to match the mean and variance of the standard normal distribution. This is accomplished by the addition of a second loss term known as the Kullback-Leibler divergence term. The total loss for the VAE is the sum of the reconstruction loss (unchanged from the vanilla auto encoder) and the Kullback-Leibler loss, Equations 2.18 and 2.19. After a VAE is properly trained, a random vector produced by Equation 2.17, where epsilon is from a normal distribution, is provided as input to the decoder which produced data similar, but not exactly the same, as the original training data.

2.3.2 Generative Adversarial Networks

Another popular composite Neural Network model is the Generative Adversarial Network [28] (GAN). This generative model has been mostly successful with image generation. The GAN is composed of two neural network with different inputs and outputs. The first network, called the generator, $(G(\theta))$, is the generative portion of the model and finds a mapping from a latent space of random numbers to the distribution of of the training data. The second network, called the discriminator or critic, $(D(\theta))$, is a classifier and classifies each input as real or fake data. The goal of the GAN training algorithm is to learn the distribution of the training data and find a mapping from a latent space to that distribution. Once the GAN is successfully trained, the generator is sampled to provide the generated data.

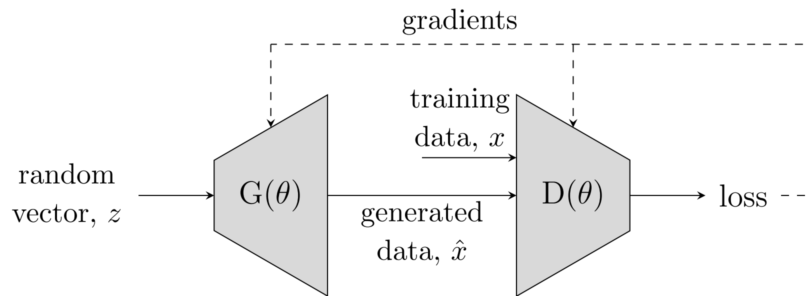

Figure 2.13 Structure of basic GAN

The generator's input is a sampling from a random space of numbers, z, that is typically from a standard normal distribution. The dimension of this space is an important hyper-parameter [29], and is typically around 100. The output of the generator is the generated data, $\hat{x}$. If the generator is trained well, the generated data should look like the real data, and it's distribution should match the true data distribution. The discriminator's input is both the generated data, the generator's output, and the training data, x. The discriminator outputs classification values between 0 and 1. Output values closer to zero indicate that the input data does not belong to the training data distribution, while values closer to 1 indicate that the input data likely came from training data distribution. Training for both network happens independently. The discriminator is the first to be trained with its

weights updated based on how accurate it's classification was. The discriminator receives a batch of both training data and generated data. Both the training data and the generated data are assigned target labels, 1 for the training data and 0 for the generated data. The generator is trained second and it's weight are updated based how the discriminator classifies the generators output. During this part of the training, the discriminator only accepts the generated data as input.

$$\min_{G} \max_{D} \ \mathbb{E}_{x\sim p_{data}(x)} \left[\log D(x)\right] + \mathbb{E}_{z\sim p_{z}(z)} \left[\log\left(1 - D(G(z))\right)\right] \tag{2.20}$$

$$\begin{gathered}
\min_{D} \ \frac{1}{2}\mathbb{E}_{x\sim p_{data}(x)} \left[(D(x) - b)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p_{z}(z)} \left[(D(G(z)) - a)^2\right] \\
\min_{G} \ \frac{1}{2}\mathbb{E}_{z\sim p_{z}(z)} \left[(D(G(z)) - c)^2\right] \\
b - c = 1 \\
b - a = 2
\end{gathered} \tag{2.21}$$

$$\min_{G} \max_{D} \ \mathbb{E}_{x\sim p_{data}} \left[D(x)\right] - \mathbb{E}_{\hat{x}\sim p_{generated}} \left[D(\hat{x})\right] \tag{2.22}$$

The original loss function of the GAN is the Binary Cross-Entropy (BCE) loss Equation 2.20. This loss function is based on entropy, an information theory concept, that measures the disorder in a given data set. When two datasets are present, cross-entropy measures the difference between both. Many well known training problems, such as mode collapse and vanishing gradients, exist with this loss function. Alternative loss functions such as the Wasserstein loss [30], Equation 2.22, or the Least Squares Loss function [31], Equation 2.21, have been developed to address these issues.

3 Architectures

3.1 MLP Generative Adversarial Network without inputs

The first architecture explored was the MLP Generative Adversarial Network (MLP-GAN). In this model, both the generator and discriminator are MLP neural networks. Instead of the Binary Cross-Entropy loss explained in Section 2.3.2, the Wasserstein Loss with gradient penalty was chosen as this provided the better results compared with the BCE loss function and the least-squares loss function. GANs that utilize this loss function are called WGANs. Because of the better results that it provides, all GAN architectures in this work used the Wasserstein loss. The number of layers and layer sizes for the MLP-GAN are summarized in Table 3.1. Final values for all hyper-parameters are summarized in Table 3.2.

Table 3.1 MLP-GAN Structure

Layer	**Generator**	**Critic**
input layer	100 nodes	10800 nodes
hidden layer 1	200 nodes	6400 nodes
activation	ReLU	ReLU
hidden layer 2	800 nodes	1600 nodes
activation	ReLU	ReLU
hidden layer 3	3200 nodes	400 nodes
activation	ReLU	ReLU
output layer	10800 nodes	1 node
activation	none	none

Table 3.2 MLP-GAN Hyper-parameters

Parameter	**Generator**	**Critic**
optimizer	RMS Prop	RMS Prop
learning rate	0.0004	0.0004
batch size	100	
latent dimension	100	
gradient penalty weight	10	
critic train factor	5	

The training data consisted of nine features (V, α, β, p, q, r, ϕ, θ, and ψ). Each feature was a 1200 point long sequence representing one minute of flight data. The data was pre-processed using standardization as some research suggests that GANs learn a normal

distribution more easily. Because of the way Keras structures the kernels for two-dimensional dense layers, keeping the input data in this form (a 1200 by 9 matrix) provided very poor results. When provided a two-dimensional input, the Keras dense layer will use the same kernel for each of the features. To get around this, each mini-batch of training data was reshaped by stacking each of the features on top of one another to form a one-dimensional matrix, specifically 1 by 10800. A major downside of reshaping the input in this way is that it drastically increased the number nodes, and thereby the number of weights and biases, required in each layer. Increasing the model complexity in this way consequently increases training times and increases the chances of training instability. The output of the generator was also structured to be a one-dimensional matrix and was reshaped to form the 1200 by 9 data matrix of generated flight variables.

3.2 MLP Generative Adversarial Network with inputs

The previous MLP-GAN model was modified slightly to generate a control input sequence for all control surfaces as well as the sequence of state variable responses. The training data for this model, MLP Generative Adversarial Network with inputs (MLP-GANi), consisted of 12 features (V, α, β, p, q, r, ϕ, θ, ψ, δe, δa, and δr). The first 9 features of this dataset are the same as the dataset used in the previous MLP-GAN model and the control inputs (elevator, aileron, and rudder) corresponding to that dataset were concatenated to the end. The structure of the MLP-GAN needed to be modified due to the increased size of the training data. Now incorporating 12 feature of length 1200, the reshaped mini-batch of training data formed a 1 by 14400 matrix. The number of hidden layers and nodes in each layer were adjusted as outlined in Table 3.3.

Table 3.3 MLP-GANi structure

Layer	Generator	Critic
input layer	100 nodes	14400 nodes
hidden layer 1	200 nodes	6400 nodes
activation	ReLU	ReLU
hidden layer 2	400 nodes	3200 nodes
activation	ReLU	ReLU
hidden layer 3	800 nodes	1600 nodes
activation	ReLU	ReLU
hidden layer 4	1600 nodes	800 nodes
activation	ReLU	ReLU
hidden layer 5	3200 nodes	400 nodes
activation	ReLU	ReLU
hidden layer 6	6400 nodes	200 nodes
activation	ReLU	ReLU
output layer	14400 nodes	1 node
activation	none	none

Table 3.4 MLP-GANi Hyper-parameters

Parameter	Generator	Critic
optimizer	RMS Prop	RMS Prop
learning rate	0.0004	0.0004
batch size	120	
latent dimension	100	
gradient penalty weight	10	
critic train factor	5	

The batch size was increased to 120, mainly to speed up training time, but all other hyper-parameters remained the same. Table 3.4 lists the final hyper-parameter values for this modified model.

3.3 LSTM Generative Adversarial Network

A Generative Adversarial Network using LSTM layers was created to address the issues with MLP layers and time dependencies. The Wasserstein loss function was retained along with the same optimizers as the MLP-GAN models. Because the dimension of the LSTM layer output is dependent on the number of cells in the layer, there was no need to perform reshaping on the training data. The generator contained a single layer of 12 LSTM cells (each

cell outputs a feature of generated data.) Because the the LSTM cell generates a sequence the same length as the input, the latent dimension was changed to match the sequence length. The batch size was increased to 150 to speed up computation time. The training data was normalized, between 0 and 1, instead of standardized. The reason is because the output gate of an LSTM cell has a range of (-1,1) due to the hyperbolic tangent activation function. Standardizing a dataset will not limit the data between the same range. In order to have the generator output range match the range of the training data, the output of the LSTM layer was passed through a sigmoid activation function. The LSTM architecture resulted in considerable longer training times. Each cell was tasked with iterating through a sequence of length 1200.

Table 3.5 LSTM-GAN structure

Layer	**Generator**	**Critic**
input size	(1200×1)	(1200×12)
hidden layer 1	12 cells	12 cells
activation	sigmoid	none
hidden layer 2	- -	12 cells
activation	- -	ReLU
output size	(1200×12)	(1×1)

Table 3.6 LSTM-GAN Hyper-parameters

Parameter	**Generator**	**Critic**
optimizer	Adam	Adam
learning rate	0.001	0.001
batch size	150	
latent dimension	1200	
gradient penalty weight	10	
critic train factor	5	

3.4 CNN Generative Adversarial Network

Convolutional Neural Networks are primarily used with image data that contains many spacial features. While flight data is sequential data with temporal features, it could be presented to a CNN as an 'image'. Images in data form are nothing but two-dimensional

or three-dimensional tensors, so the training data could be thought of as a 1200 by 12 image. The structure of the CNN-GAN is summarized in Table 3.7. A dense hidden layer in the generator increases the dimension from the latent space so it can be reshaped as a small "image" matrix with multiple channels. The reverse operation of a convolution (called transpose convolution) is used to form the 1200 by 12 matrix. The critic uses the conventional convolutional layers to reduce the data to a single value.

Table 3.7 CNN-GAN structure

Layer	**Generator**	**Critic**
input size	(1000×1)	(1200×12)
hidden layer 1	2000 nodes	12 kernels
activation	ReLU	sigmoid
hidden layer 2	10 kernels	6 kernels
activation	sigmoid	sigmoid
hidden layer 3	3 kernels	3 kernels
activation	sigmoid	sigmoid
hidden layer 4	1 kernel	1 kernel
activation	sigmoid	sigmoid
output size	(1200×12)	(1×1)

Table 3.8 CNN-GAN Hyper-parameters

Parameter	**Generator**	**Critic**
optimizer	RMSProp	RMSProp
learning rate	0.001	0.001
batch size	150	
latent dimension	100	
gradient penalty weight	10	
critic train factor	5	

The loss function for this architecture was the Wasserstein loss with the same gradient penalty parameters. The optimizer was switched to RMSProp. The batch size was set to 150 and the latent dimension remained unchanged at 100.

3.5 Variational Auto Encoder

The last architecture explored was the Variational Auto Encoder. This architecture used MLP layers in both the encoder and decoder and the layer sizes are similar to the MLP-

GANi model. Table 3.9 shows a summary of of this structure. The reconstruction loss was chosen to be the Binary Cross-Entropy.

Table 3.9 VAE structure

Layer	**Encoder**	**Decoder**
input layer	14400 nodes	50 nodes
hidden layer 1	6400 nodes	400 nodes
activation	ReLU	ReLU
hidden layer 2	3200 nodes	800 nodes
activation	ReLU	ReLU
hidden layer 3	1600 nodes	1600 nodes
activation	ReLU	ReLU
hidden layer 4	800 nodes	3200 nodes
activation	ReLU	ReLU
hidden layer 5	400 nodes	6400 nodes
activation	ReLU	ReLU
output layer 6	50 nodes	14400 nodes
activation	none	none

Table 3.10 VAE Hyper-parameters

Parameter	**Encoder**	**Decoder**
optimizer	Adam	Adam
learning rate	0.0001	0.0001
batch size	120	
latent dimension	50	

4 Implementation & Results

The purpose of the research in this thesis is to explore generating flight data using machine learning. Naturally, the training data set should consist of actual flight data. However, one the biggest problems with using recorded flight data is the noise. While adding random noise to training data is often cited as having a regularizing effect, thus improving training [32]; the noise distribution of each sensor may not be the same. For this reason, in the initial state of this research, numerical aircraft simulation data was used. Using simulation data allows for greater control over the dataset. We can have access to any state variable, perform any arbitrary input control sequences, and choose any flight conditions desired. Once a viable ML model has been developed, additional steps can be taken to alter the model to accommodate noise in the data.

4.1 Simulink AIRLIB model

The flight simulation was performed in MATLAB/Simulink using the freely available Airlib library. The Airlib library provides comprehensive non-linear models of several popular aircraft. The continuous-time, open-loop model of the Beech 99 aircraft was used. The control inputs for elevator, aileron, rudder were all selected to be sequence of periodic doublet inputs. The timing of the doublets as well as the magnitude was chosen randomly. This was mainly for convenience as the simulation could be run very quickly with a pre-defined sequence of inputs instead of being run in real time. The simulation was performed for a total of 20 hours of flight data. The simulation recorded the Euler angles (ϕ, θ, ψ), angular velocities (p, q, r), forward velocity (V), side-slip angle (β), angle of attack (α), and 3-coordinate position (x, y, h).

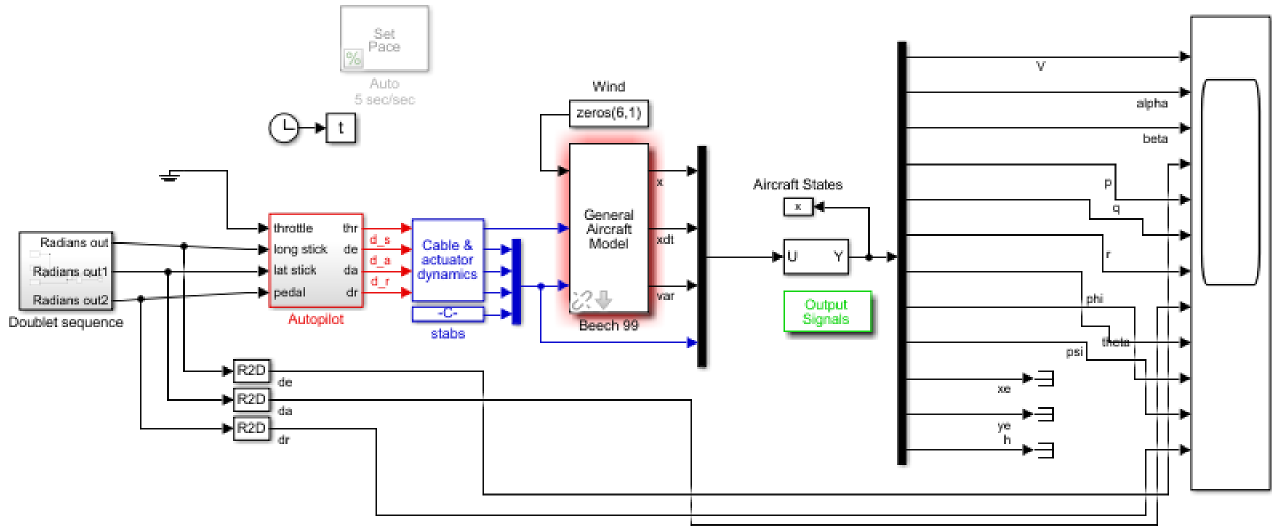

Figure 4.1 AIRLIB simulink file used to generate training data.

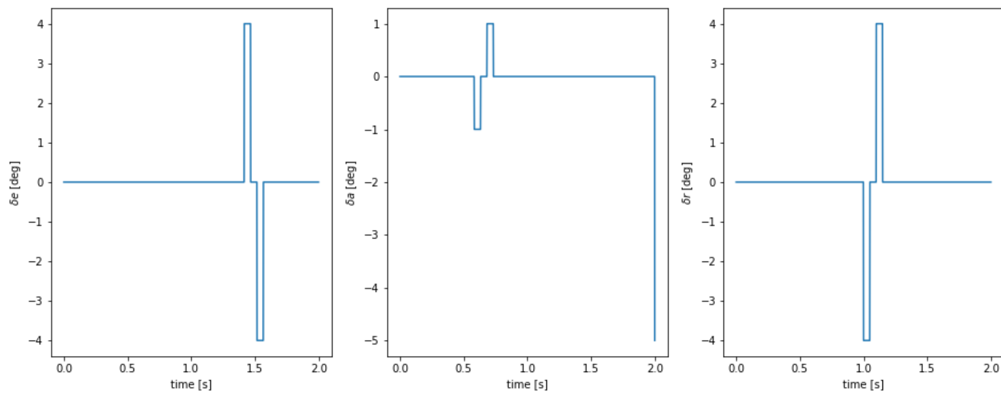

Figure 4.2 Example of control inputs given to simulation.

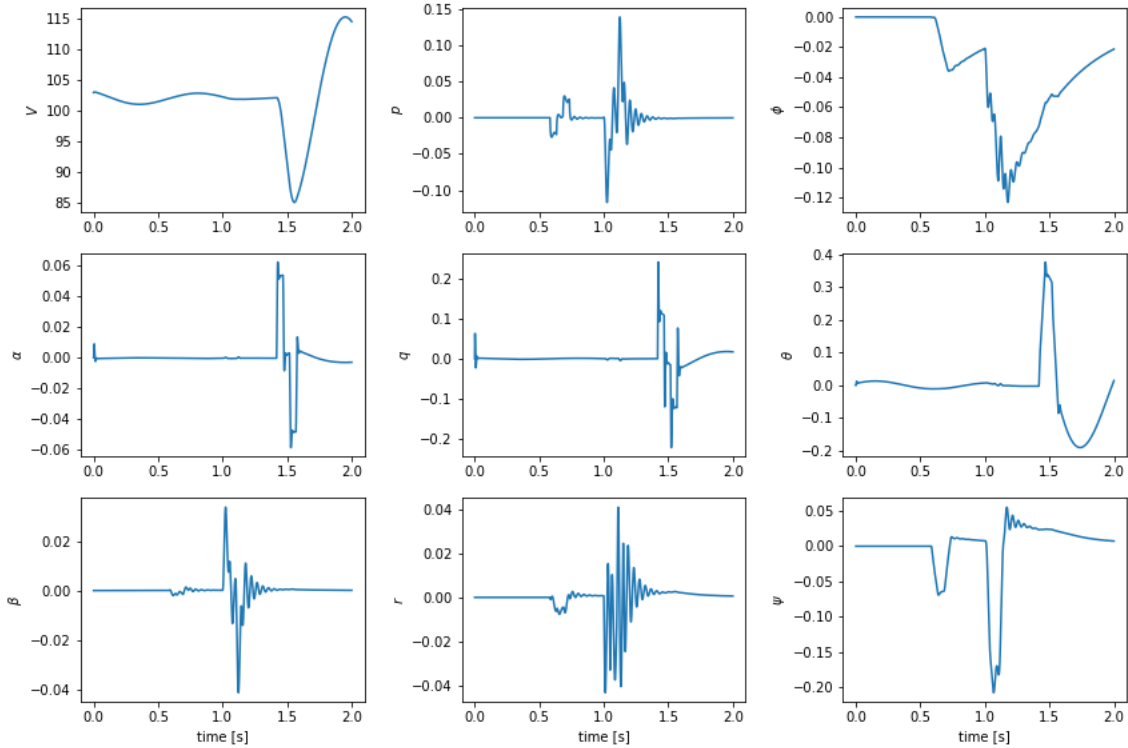

Figure 4.3 Sample of states from simulation.

The Beech 99 aircraft model was trimmed at a cruise flight condition. The trim values were 103 m/s forward velocity and an altitude of 1524 meters. These and other initial state values were taken from [33]. Other flight conditions were tried with the intention of creating a training database with varying flight conditions, however the results from each flight condition could not be easily distinguished apart. An example of the doublet sequence provided to the simulation is shown in 4.2. A sample of the simulation output is shown in 4.3.

4.2 Python Code

Python is the most popular programming language for machine learning. Python code has been optimized for machine learning purposes through freely available libraries and APIs. One of the most popular of these libraries is TensorFlow. The TensorFlow library provides many modules for creating neural network architectures, implementing training algorithms, GPU utilization, and much more.

4.3 MLP Generative Adversarial Network without inputs

Figure 4.4 shows a cherry-picked sample of generated data from this model. Most of the generated samples were very noisy and only about one out of every ten generator samplings provided clean results. The generated state ϕ, however, was an exception as it always contained noise. The generated data contains many details representative of true flight data. The shapes of the plots show damped oscillations and quick changes in direction as is consistent with natural aircraft dynamics and responses to control input. The directions of the responses are also consistent with type of control input. For example, all four longitudinal variables (V, α, q, θ) show a response at approximately 15 seconds. The forward velocity decreases rapidly, while the angle of attack, pitch angle, and pitch rate all increase sharply. These responses are consistent with a sudden upward deflection of the elevator. At approximately 20 seconds, the trends reverses consistent with a downward elevator deflection. The data for the lateral-directional variable show similar trends. The flight variables (β, p, r, ϕ, and ψ) all show a response at 50 seconds. It is unclear whether it could represent a response from an aileron or rudder deflection, however, all five responses are consistent. The roll and yaw angles decrease indicating a left bank maneuver. The angular rate responses, p and r also reflect this.

Figure 4.5 shows a non-ideal loss curve trajectories. Initially, the critic loss approaches zero very quickly, but after 1000 training epochs the critic loss improves very little. The generator's loss increases in the first 500 training epoch, which is not abnormal, then decreases at a very slow rate throughout the remainder of training. Another important detail to note about the loss curves is the magnitude of the oscillations. Most training loss curves will oscillate, particularly initially, however, these loss curves indicate some type of instability resulting in sub-optimal training.

Correlation was used as a metric to compare the generated data distribution to the training data distribution. This is an appropriate metric because, fundamentally, the GAN learns the distribution of the training data. Equally sized sets of training data and generated

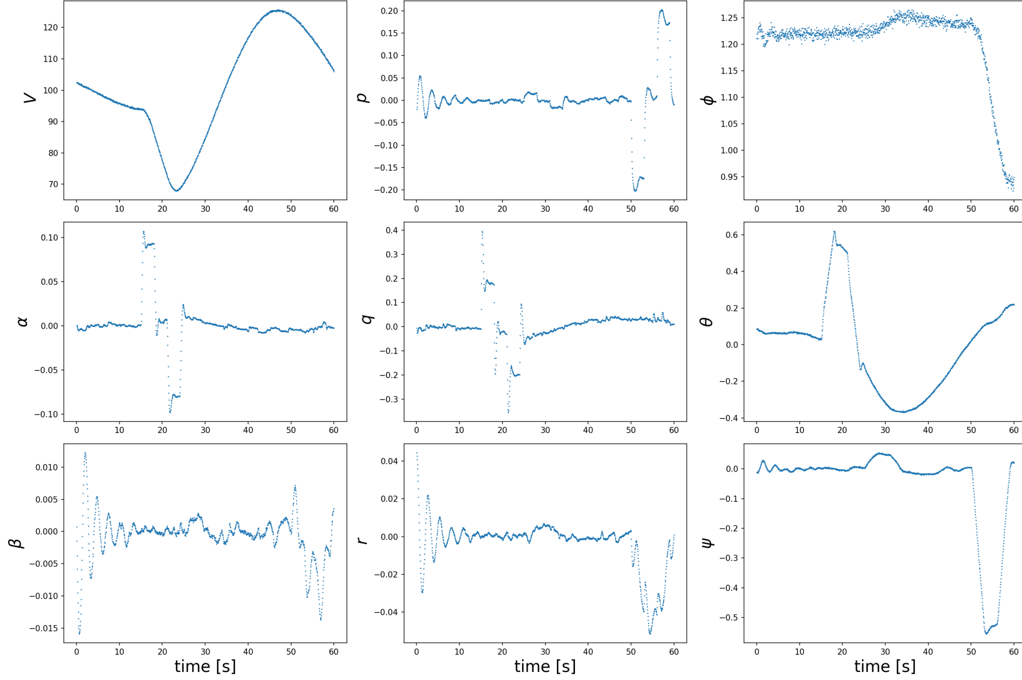

Figure 4.4 Sample of generated data from MLP GAN.

data were plotted in together in Figure 4.6 to provide a method of visualizing the correlations and distributions of each flight variable. From these plots, it appears that the MLP-GAN was able to effectively capture the training data distribution. Table 4.1 lists the mean and standard deviation of each feature for both datasets which is a more concrete method for comparing similarity between two datasets.

Table 4.1 MLP-GAN Generated Data Statistics

	Generated Dataset		**Training Dataset**	
	mean	std dev	mean	std dev
V	$1.023 \cdot 10^{2}$	6.551	$1.022 \cdot 10^{2}$	6.612
α	$-3.929 \cdot 10^{-4}$	$1.428 \cdot 10^{-2}$	$-3.430 \cdot 10^{-4}$	$1.402 \cdot 10^{-2}$
β	$-2.245 \cdot 10^{-5}$	$6.321 \cdot 10^{-3}$	$-3.976 \cdot 10^{-6}$	$6.659 \cdot 10^{-3}$
p	$1.508 \cdot 10^{-5}$	$3.098 \cdot 10^{-2}$	$-4.559 \cdot 10^{-6}$	$3.181 \cdot 10^{-2}$
q	$2.832 \cdot 10^{-4}$	$4.274 \cdot 10^{-2}$	$1.694 \cdot 10^{-4}$	$4.211 \cdot 10^{-2}$
r	$-1.513 \cdot 10^{-4}$	$1.050 \cdot 10^{-2}$	$2.307 \cdot 10^{-5}$	$1.102 \cdot 10^{-2}$
ϕ	$5.201 \cdot 10^{-2}$	$2.712 \cdot 10^{-1}$	$5.922 \cdot 10^{-2}$	$2.775 \cdot 10^{-1}$
θ	$-7.065 \cdot 10^{-4}$	$9.644 \cdot 10^{-2}$	$-7.321 \cdot 10^{-4}$	$9.733 \cdot 10^{-2}$
ψ	$-1.471 \cdot 10^{-3}$	$5.951 \cdot 10^{-2}$	$2.856 \cdot 10^{-4}$	$5.788 \cdot 10^{-2}$

This model produced convincing data. However, there isn't a way to verify if the gen-

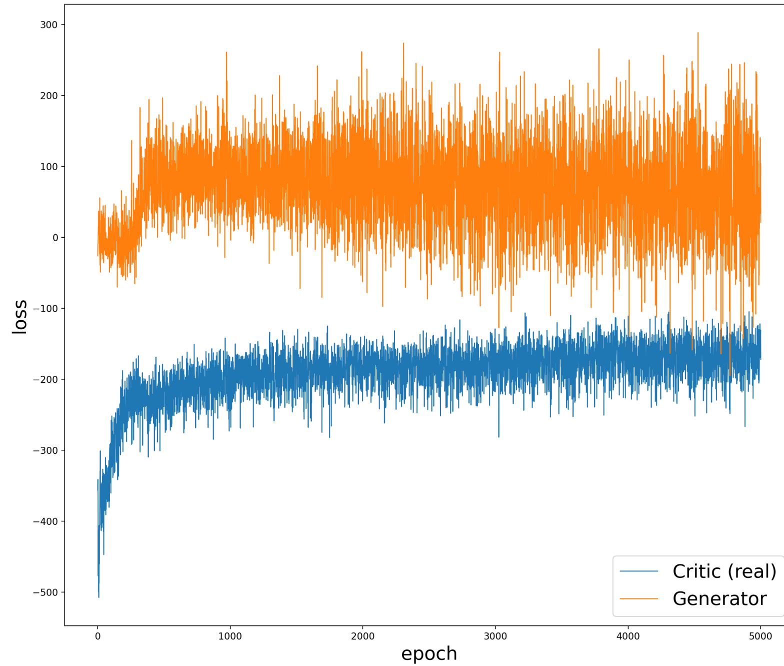

Figure 4.5 Loss curves from MLP GAN.

erated data satisfies the aircraft dynamics model because it only produced the generated outputs, not the control inputs. The control inputs were not even considered in the training data set. The next model accounts for this by including the control surface deflections in the training data.

4.4 MLP Generative Adversarial Network with inputs

The results from the MLP-GANi, Figure 4.7, were similar to the MLP-GAN with exception of the generated control inputs. The same evidence of the kinematic consistency is evident in these results. However, with the presence of generated control inputs, it's clear to see that the timing of the responses in the generated flight variables are consistent with the timing of the control surface doublets.

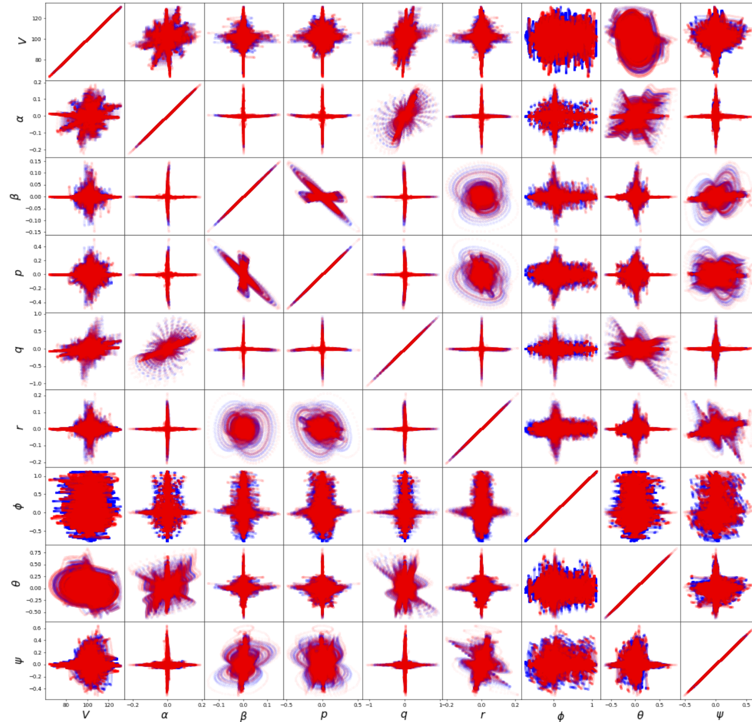

Figure 4.6 Correlations of generated states from MLP GAN. Red is the generated data and the blue is the training data.

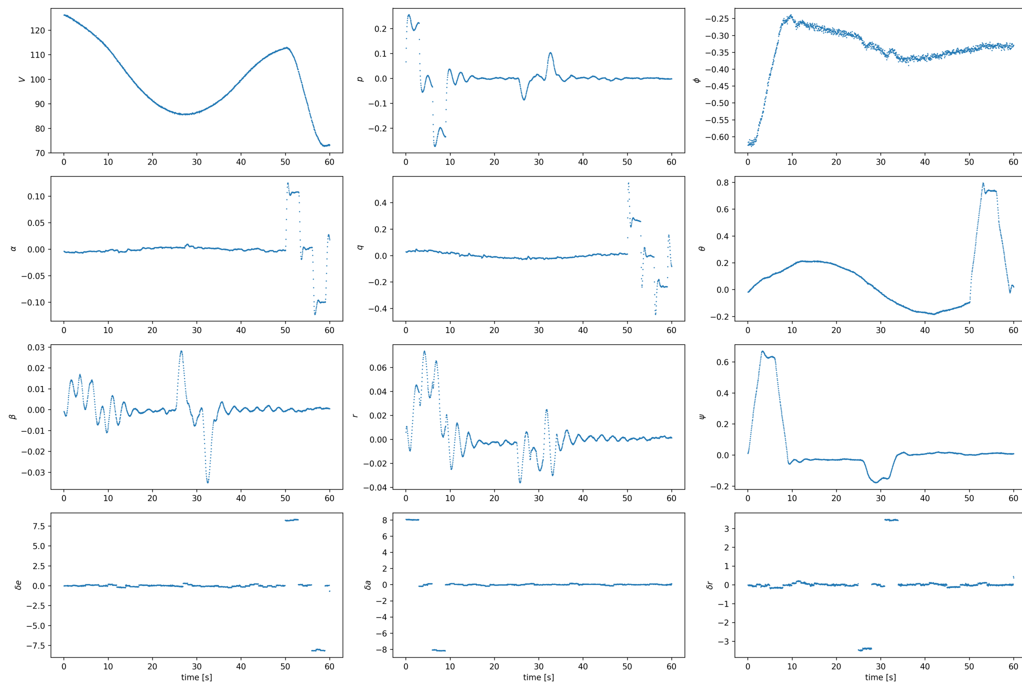

Figure 4.7 Sample of generated data from MLP-GANi.

The trends of the loss curves were very identical as well. This was expected as all hyperparameters, except the batch size, remained the same as those in the MLP-GAN. The batch size increased only slightly and was not significant enough to improve the training. However, the increase in batch size did cause the oscillations in the curves to grow. The Wasserstein loss is an expectation of the critic output over the batch size, so a larger batch size will generally result in a larger expectation.

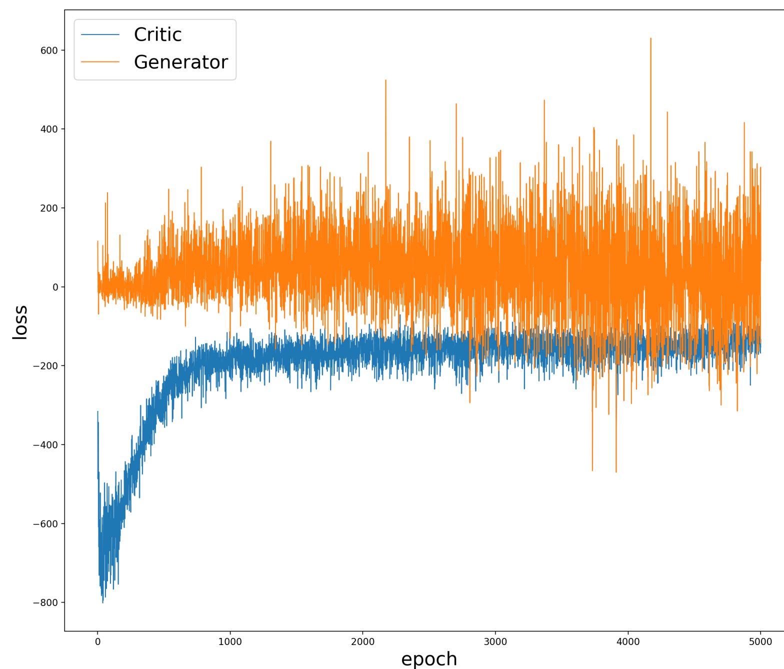

Figure 4.8 Loss curves from MLP-GANi.

Figure 4.9 shows similar correlations and distributions among the generated flight variables. As with the MLP-GAN, the MLP-GANi does seem to capture the training data distribution well. The mean and standard deviation values listed in Table 4.2 show smaller errors between the two datasets compared to the results of the MLP-GAN indicating that the MLP-GANi actually learned better. One possible reason for this could be that the MLP-GANi has a larger structure with more trainable parameters and therefore was more effective

at creating the input to output mapping.

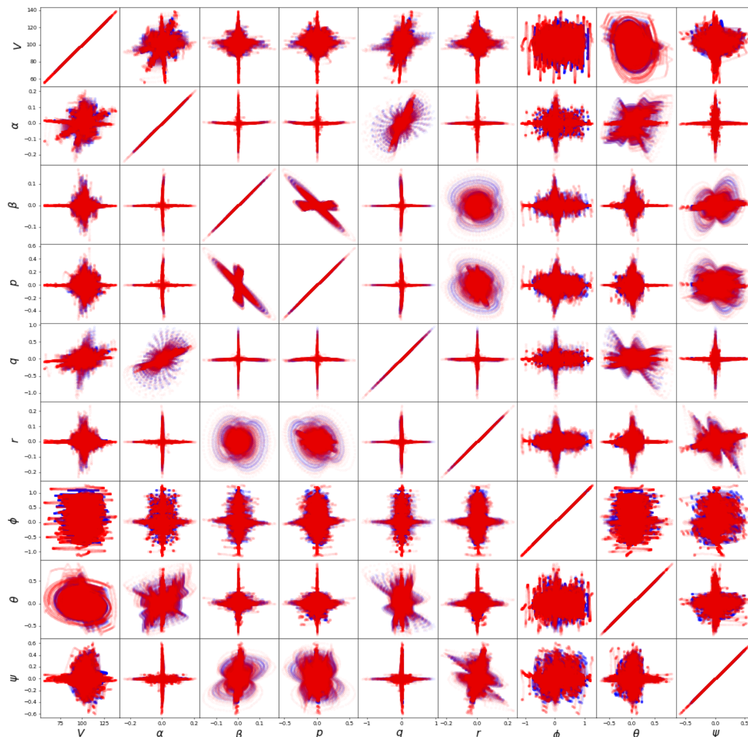

Figure 4.9 Correlations of generated states from MLP-GAN. Red is the generated data and the blue is the training data.

Table 4.2 MLP-GANi Generated Data Statistics

	Generated Dataset		**Training Dataset**	
	mean	std dev	mean	std dev
V	$1.021 \cdot 10^{2}$	6.777	$1.022 \cdot 10^{2}$	6.612
α	$-2.781 \cdot 10^{-4}$	$1.434 \cdot 10^{-2}$	$-3.430 \cdot 10^{-4}$	$1.402 \cdot 10^{-2}$
β	$-2.746 \cdot 10^{-5}$	$7.510 \cdot 10^{-3}$	$-3.976 \cdot 10^{-6}$	$6.659 \cdot 10^{-3}$
p	$-1.092 \cdot 10^{-4}$	$3.423 \cdot 10^{-2}$	$-4.559 \cdot 10^{-6}$	$3.181 \cdot 10^{-2}$
q	$1.496 \cdot 10^{-4}$	$4.267 \cdot 10^{-2}$	$1.694 \cdot 10^{-4}$	$4.211 \cdot 10^{-2}$
r	$-3.578 \cdot 10^{-5}$	$1.227 \cdot 10^{-2}$	$2.307 \cdot 10^{-5}$	$1.102 \cdot 10^{-2}$
ϕ	$6.413 \cdot 10^{-2}$	$2.830 \cdot 10^{-1}$	$5.922 \cdot 10^{-2}$	$2.775 \cdot 10^{-1}$
θ	$-3.419 \cdot 10^{-4}$	$1.008 \cdot 10^{-1}$	$-7.321 \cdot 10^{-4}$	$9.733 \cdot 10^{-2}$
ψ	$-4.412 \cdot 10^{-4}$	$5.924 \cdot 10^{-2}$	$2.856 \cdot 10^{-4}$	$5.788 \cdot 10^{-2}$

The data for the generated control inputs was saved and provided to the simulation to obtain the actual aircraft response to those generated inputs. The comparison of the true and generates responses are plotted in Figure 4.10. The simulation results align very well with most of the generated results. The generated results seemed to struggle with the lower frequency responses as compared to the higher frequency responses. It's also worth pointing out that the consistently noisy generated state, ϕ, was the least accurate when compared to the true state.

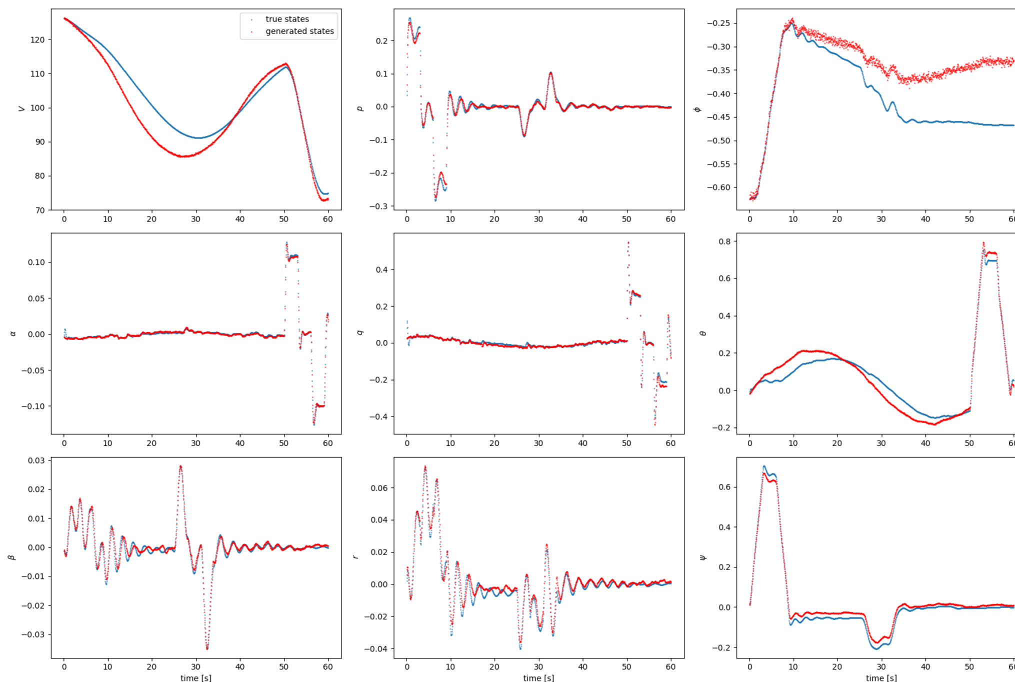

Figure 4.10 Accuracy of the MLP-GANi generated state variables when compared to the true dynamics.

As explained in Section 2.1.1 MLP models are not built for handling time-dependent data. The batches of training data were provided to the model all at once, not in a sequential manner. Therefore, it is not possible for the MLP neural network to learn the time dependencies in the data. It is likely that the MLP architecture is simply learning the shapes of the data and is not learning anything about the dynamics of the aircraft.

4.5 LSTM Generative Adversarial Network

Despite much hyper-parameter tuning, the LSTM-GAN could not produce satisfactory results. Figure 4.11 shows data generated by the LSTM-GAN. All states and control inputs appear to favor the extremes of the range of possible values. The loss curves shown in Figure 4.12 show no converging trends in the loss. Shortly after 1550 training epochs, the training becomes very unstable. An extremely large spike in the generator loss occurs and then the loss starts to slowly diverge. Both the results and the loss curves clearly indicate a lack of training. The final layer of the LSTM-GAN is returning a sequence with many repeated values instead of sequence of dynamically changing states. The LSTM layer may not be the best layer for the generator of a GAN. The purpose of an LSTM is to consider temporal features in the input sequence. However, in the case of a GAN, the generator's input is a sampling of random noise. There are no temporal features contained in this latent space. The LSTM is more appropriate for the critic since it's input is sequential data. After 1500 training epochs the generator loss temporarily spikes and then slowly starts to diverge indicating unstable training for the generator.

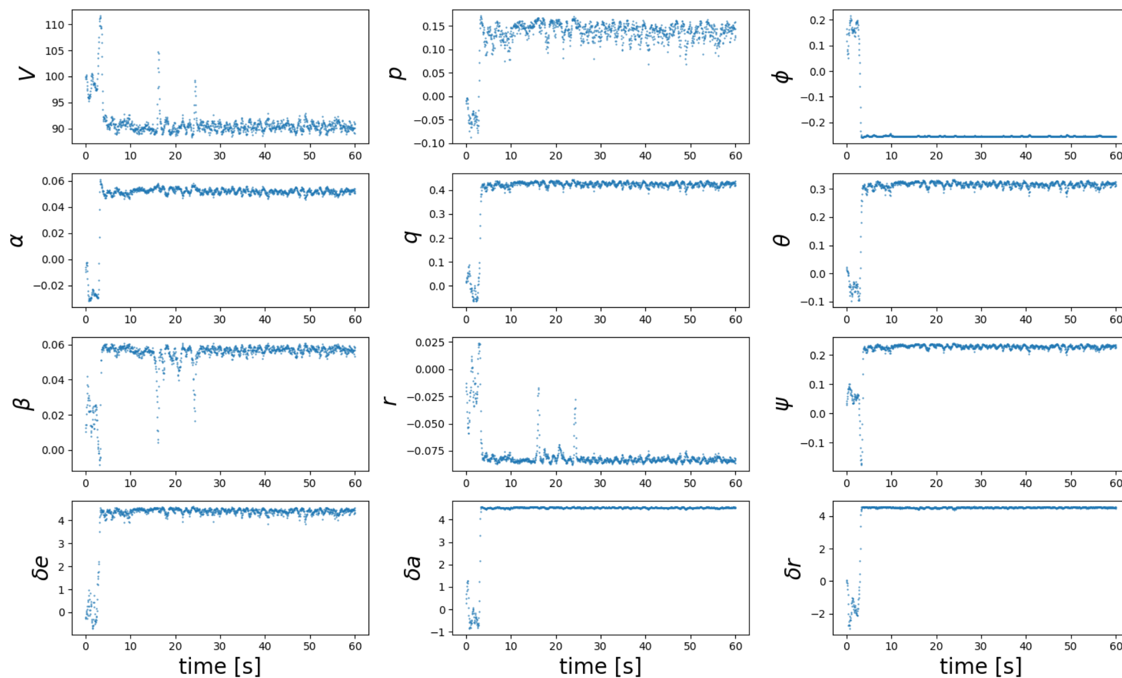

Figure 4.11 Sample of generated data from LSTM-GAN.

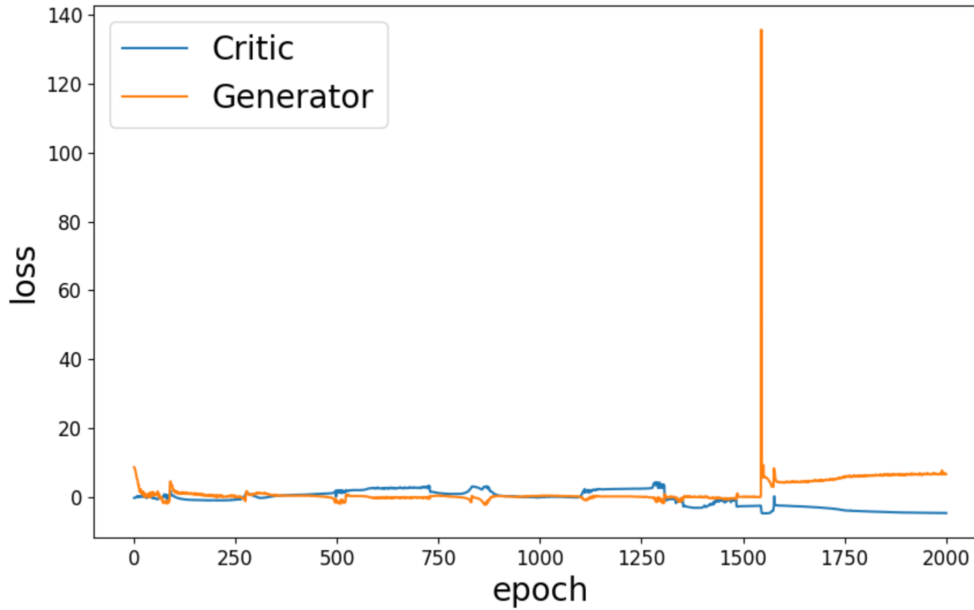

Figure 4.12 LSTM-GAN loss curves.

4.6 CNN Generative Adversarial Network

The CNN-GAN generated data much closer to flight data compared to the LSTM-GAN but not as well defined as results from the MLP architectures. Although very noisy, the generates states show some form of dynamic responses and control surface deflection sequences roughly form doublets. It is possible that the training was very slow and more training epochs would show improved results. However, attempts to train the model longer than 8000 epochs resulted in issues with the Google Colab GPU access. Figure 4.14 shows unstable training for both the generator and critic throughout the first half of the training. While the generator loss eventually stabilizes, the critic loss continues to oscillate and diverge.

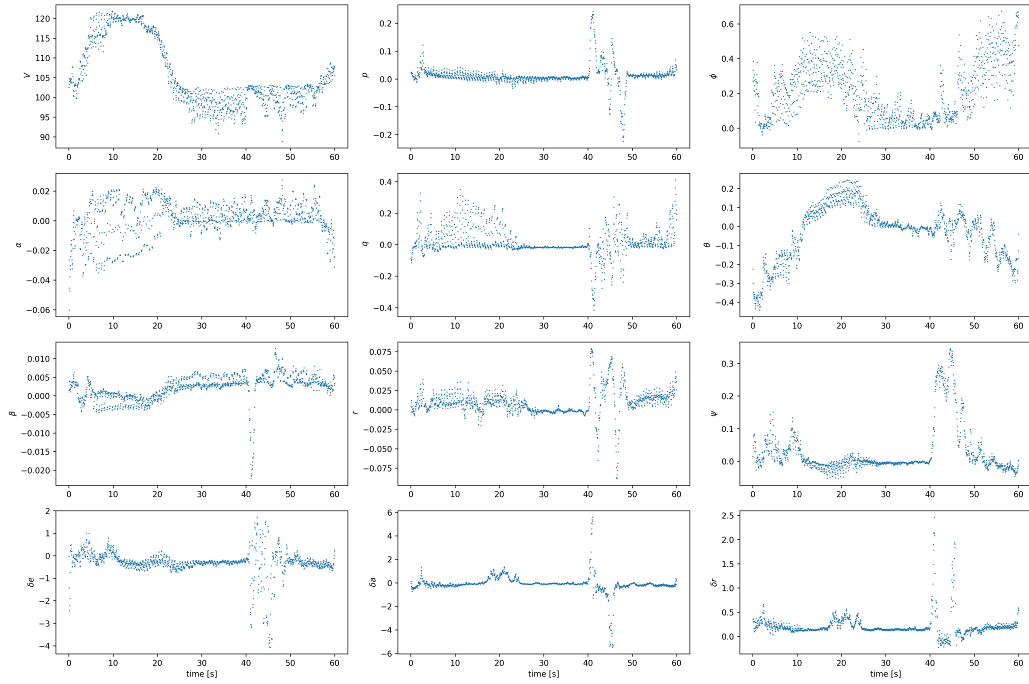

Figure 4.13 Sample of generated data from CNN-GAN.

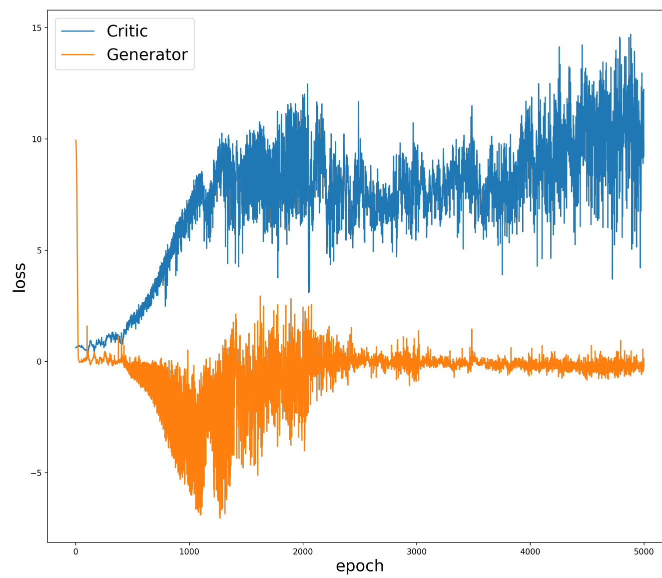

Figure 4.14 Loss curves from CNN-GAN.

4.7 Variational Auto Encoder

After 5000 training epochs the VAE failed to produce reasonable data. It appears that the decoder is heavily favoring the mean values of each flight variable. Although the reconstruction loss does improve throughout the training, it does not reflect in the results. The KL loss does not improve indicating that the structure of the latent space is not being effectively learned during training. It is possible based on Figure 4.16 to conclude that the VAE is learning to encode and reconstruct the data and the latent space that the being created is not the normal distribution. Because of this, it is difficult to sample from the latent space to show examples of regenerated data.

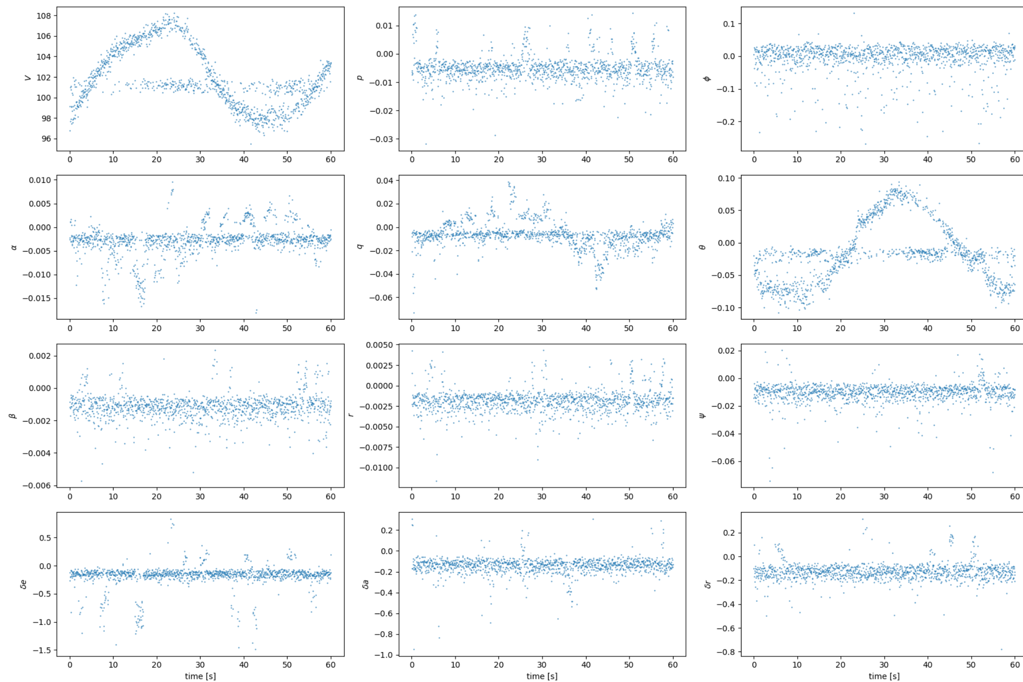

Figure 4.15 Sample of generated data from VAE-GAN.

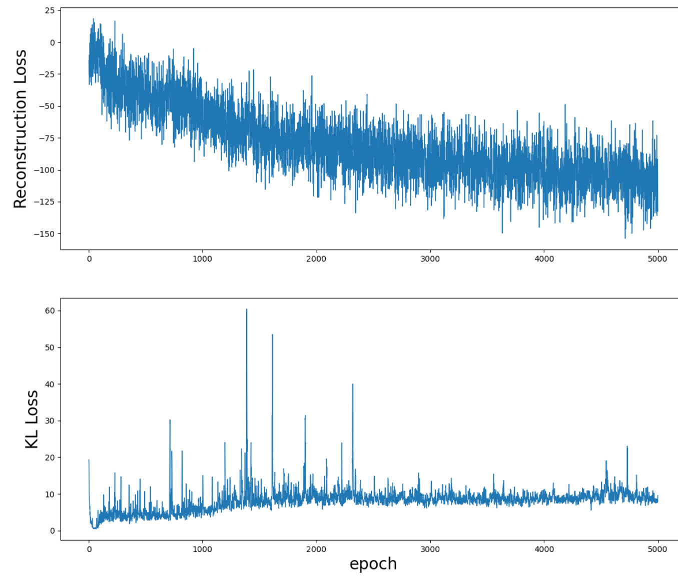

Figure 4.16 Loss curves from VAE-GAN.

5 Conclusions and Future Work

Of the architectures explored, only the MLP-GAN and MLP-GANi were successful at generating results resembling valid flight data. In the latter case, it was shown that the generated control inputs and generated states were, overall, consistent with the aircraft dynamics model. Basic statistical analysis showed that the generated data distribution matched the training data distribution fairly well. Results from the CNN-GAN architecture show that it has some potential to generate valid data, but the model would need to be adjusted and hyper-parameters would need be fine tuned because, as the loss curves indicate, the training process was very poor and unstable. The other two architectures, LSTM-GAN and the Variational Auto Encoder, could not produce any meaningful results. With the LSTM-GAN, it is likely that the LSTM layers are not well suited for this architecture-particularly the generator. The Variational Auto Encoder was unable to learn the desired structure of the latent space and therefore generated poor results.

Although the results of the MLP-GAN/MLP-GANi look promising, there are several drawbacks associated with it. As mentioned earlier, only a small fraction of the MLP-GAN's generated data provided clean results. The consistency check with the aircraft dynamics model wasn't perfect. Also this model only produced data for one aircraft at one flight condition. The first two problems mentioned could be remedied by more hyper-parameter and structure tuning. The third issue might be addressed by using a conditional neural network. However, the dynamic responses of an aircraft at different flight conditions all share very similar characteristics. In neural network terms, the datasets for each flight condition would share very similar distributions. This would make it challenging for the neural network to distinguish among them. Another problem comes from the fact that the model will only generate 60 second long sequences of data. If a longer sequence is desired, multiple 60 second sequences cannot be strung together.

The desired data to be generated is likely better suited for more complicated composite models such as the TimeGAN [34]. Instead of considering the entire training dataset as

one distribution, the TimeGAN considers the distribution at each time step of the data and constructs a dynamic latent space that also changes at each time step. Neural ODEs [35] were created with the intention of mimicking the behaviour of ordinary differential equations. The current machine learning models explored all have one major flaw. That flaw relates to way it maps input data to output data A neural network's success will be completely dependent on the training data. This is a major hindrance because the neural networks are not capable of modeling the full range of a parametric math model's output. If the networks are successfully trained on nominal flight data and are able to generate valid flight data, they will not be able to generate non-nominal flight data. Therefore a machine leaning model with a different mapping method (something suitable for leaning dynamic behaviour) is likely a better way to generate aircraft flight data synthetically. This could perhaps be a model that directly maps control inputs to flight variables.

REFERENCES

[1] Hale, L. E., Patil, M., and Roy, C. J., "Aerodynamic parameter identification and uncertainty quantification for small unmanned aircraft," *Journal of Guidance, Control, and Dynamics*, Vol. 40, No. 3, 2017, pp. 680–691.

[2] Whalen, E., and Bragg, M., "Aircraft characterization in icing using flight test data," *Journal of Aircraft*, Vol. 42, No. 3, 2005, pp. 792–794.

[3] Morelli, E. A., "Determining aircraft moments of inertia from flight test data," *Journal of Guidance, Control, and Dynamics*, Vol. 45, No. 1, 2022, pp. 4–14.

[4] Esteban, C., Hyland, S. L., and Rätsch, G., "Real-valued (medical) time series generation with recurrent conditional gans," *arXiv preprint arXiv:1706.02633*, 2017.

[5] Eckerli, F., and Osterrieder, J., "Generative adversarial networks in finance: an overview," *arXiv preprint arXiv:2106.06364*, 2021.

[6] Hornik, K., Stinchcombe, M., and White, H., "Multilayer feedforward networks are universal approximators," *Neural networks*, Vol. 2, No. 5, 1989, pp. 359–366.

[7] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," , 2015. URL `https://www.tensorflow.org/`, software available from tensorflow.org.

[8] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai,

J., and Chintala, S., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[9] Chollet, F., et al., "Keras," `https://keras.io`, 2015.

[10] Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al., "Learning internal representations by error propagation," , 1985.

[11] Tarwani, K. M., and Edem, S., "Survey on recurrent neural network in natural language processing," *Int. J. Eng. Trends Technol*, Vol. 48, No. 6, 2017, pp. 301–304.

[12] Hochreiter, S., and Schmidhuber, J., "Long short-term memory," *Neural computation*, Vol. 9, No. 8, 1997, pp. 1735–1780.

[13] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y., "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[14] O'Shea, K., and Nash, R., "An Introduction to Convolutional Neural Networks," *CoRR*, Vol. abs/1511.08458, 2015.

[15] Kandel, I., and Castelli, M., "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset," *ICT express*, Vol. 6, No. 4, 2020, pp. 312–315.

[16] Janocha, K., and Czarnecki, W. M., "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.

[17] Zhao, H., Gallo, O., Frosio, I., and Kautz, J., "Loss functions for neural networks for image processing," *arXiv preprint arXiv:1511.08861*, 2015.

[18] Zhao, H., Gallo, O., Frosio, I., and Kautz, J., "Loss functions for image restoration with neural networks," *IEEE Transactions on computational imaging*, Vol. 3, No. 1, 2016, pp. 47–57.

[19] Ruby, U., and Yendapalli, V., "Binary cross entropy with deep learning technique for image classification," *Int. J. Adv. Trends Comput. Sci. Eng*, Vol. 9, No. 10, 2020.

[20] Cooper, Y., "The loss landscape of overparameterized neural networks," *arXiv preprint arXiv:1804.10200*, 2018.

[21] Sun, R., Li, D., Liang, S., Ding, T., and Srikant, R., "The global landscape of neural networks: An overview," *IEEE Signal Processing Magazine*, Vol. 37, No. 5, 2020, pp. 95–108.

[22] Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T., "Visualizing the loss landscape of neural nets," *Advances in neural information processing systems*, Vol. 31, 2018.

[23] Amari, S.-i., "Backpropagation and stochastic gradient descent method," *Neurocomputing*, Vol. 5, No. 4-5, 1993, pp. 185–196.

[24] Werbos, P. J., "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, Vol. 78, No. 10, 1990, pp. 1550–1560.

[25] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X., "Improved techniques for training gans," *Advances in neural information processing systems*, Vol. 29, 2016.

[26] Yadav, A. K., and Chandel, S., "Solar radiation prediction using Artificial Neural Network techniques: A review," *Renewable and sustainable energy reviews*, Vol. 33, 2014, pp. 772–781.

[27] Kingma, D. P., and Welling, M., "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[28] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y., "Generative adversarial networks," *Communications of the ACM*, Vol. 63, No. 11, 2020, pp. 139–144.

[29] Bojanowski, P., Joulin, A., Lopez-Paz, D., and Szlam, A., "Optimizing the latent space of generative networks," *arXiv preprint arXiv:1707.05776*, 2017.

[30] Arjovsky, M., Chintala, S., and Bottou, L., "Wasserstein generative adversarial networks," *International conference on machine learning*, PMLR, 2017, pp. 214–223.

[31] Mao, X., Li, Q., Xie, H., Lau, R. Y., Wang, Z., and Paul Smolley, S., "Least squares generative adversarial networks," *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2794–2802.

[32] You, Z., Ye, J., Li, K., Xu, Z., and Wang, P., "Adversarial noise layer: Regularize neural network by adding noise," *2019 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2019, pp. 909–913.

[33] Poole, R., "Aircraft Dynamics: From Modeling to Simulation, MR Napolitano, John Wiley and Sons, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK. 2012. 706pp. Illustrated.£ 49.99. ISBN 978-0-470-62667-2." *The Aeronautical Journal*, Vol. 116, No. 1180, 2012, pp. 680–680.

[34] Yoon, J., Jarrett, D., and Van der Schaar, M., "Time-series generative adversarial networks," *Advances in neural information processing systems*, Vol. 32, 2019.

[35] Chen, R. T., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K., "Neural ordinary differential equations," *Advances in neural information processing systems*, Vol. 31, 2018.