Project TLDR: Standalone desktop application for question answering and summarization using resource-efficient LLMs

Manu Hegde

A Capstone project report
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2025

Committee:

Erika Parsons

Michael Stiber

Shane Steinert-Threlkeld

Program Authorized to Offer Degree:
Computer Science and Software Engineering

University of Washington

Abstract

Project TLDR: Standalone desktop application for question answering and summarization using resource-efficient LLMs

Manu Hegde

Chair of the Supervisory Committee:
Erika Parsons
School of Science, Technology, Engineering & Mathematics

This project presents the design and development of a standalone desktop application for offline question answering and summarization over a user-provided document corpus, using resource-efficient large language models (LLMs). Targeted for Apple's M1/M2 hardware, the application leverages on-device computation via the Apple Neural Engine (ANE) and Metal shaders, exploring the use of the NPU beyond traditional CoreML applications. The application addresses key concerns around data privacy, resource efficiency, and accessibility. Unlike cloud-based services that require constant internet access and raise privacy risks, this application offers a secure, local alternative optimized for researchers and students. It features a graphical interface and supports retrieval-augmented generation (RAG) over the user's corpus, all while utilizing only a fraction of system resources to support seamless multitasking. Evaluation is conducted using both functional metrics (e.g., BERTScore against ChatGPT outputs) and non-functional metrics (e.g., memory and CPU usage). The result is a practical, efficient application that enables interaction with large academic corpora while preserving system responsiveness and data confidentiality.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my gratitude to Prof. Erika Parsons for all the valuable guidance and help during this work. Furthermore, I sincerely thank Prof. Steinert-Threlkeld and Prof. Stiber for accepting my request to be on the committee for this thesis and for providing precise feedback.

Chapter 1

INTRODUCTION

1.1 Background and Motivation

The field of Natural Language Processing (NLP) has undergone a significant transformation with the advent of Large Language Models (LLMs), which are capable of performing complex language understanding and generation tasks. Groundbreaking works such as the Transformer architecture [1], BERT [2], and GPT-family models [3, 4] have paved the way for highly capable models that support applications such as summarization [5], question answering [6], and document understanding [7]. These advances have been further systematized in the concept of foundation models [8], which emphasize the broad applicability and adaptability of pre-trained LLMs.

Despite their success, most widely used LLM applications operate via cloud-based services, which introduce significant limitations when it comes to privacy, data security, and control over computational resources. This is particularly concerning in academic contexts, where students and researchers often deal with sensitive or proprietary content. Recent studies have raised awareness of the risks associated with exposing private data to generative models, including membership inference [9] and data extraction attacks [10]. Moreover, surveys indicate increasing usage of LLMs in research and education, highlighting both the demand for such tools and the concerns around data governance [11, 12].

Simultaneously, the hardware landscape has evolved to enable local deployment of such models. Apple's M1 and M2 chipsets integrate high-performance CPUs, GPUs, and a dedicated Neural Processing Unit (NPU) through the Apple Neural Engine (ANE). These architectures offer a promising platform for efficient, on-device inference of LLMs, provided the models are adapted appropriately to operate under limited memory and compute budgets.

This convergence of high-capability models, growing privacy concerns, and increasingly powerful consumer hardware forms the backdrop for *Project TLDR*—a standalone desktop application for summarization and question answering over a user-specified corpus. The tool is designed to run entirely offline, preserving user privacy while leveraging optimized LLM inference. The project makes use of modern techniques such as quantization [13] and low-rank adaptation (LoRA) [14] to reduce computational overhead and improve deployment feasibility on M1/M2 hardware. Additionally, the use of Retrieval-Augmented Generation (RAG) [15] ensures that answers and summaries are grounded in user-provided text, enhancing both contextual relevance and factual consistency.

In essence, this project is motivated by the goal of empowering academic users with a practical, secure, and efficient means of engaging with large volumes of textual data. By tying together advances in NLP, secure computing practices, and consumer-grade hardware acceleration, Project TLDR aims to demonstrate that high-quality language understanding can be brought directly to the user's device—without compromise.

1.2 Our Contributions

In this project, we present *Project TLDR*, a privacy-preserving, offline, and resource-efficient desktop application that enables users to perform Question Answering (QA) and Summarization over personal document repositories. Designed primarily for MacOS systems powered by Apple's M1 and M2 architectures, the application aims to support academic and research workflows where confidentiality, simplicity, and efficiency are paramount.

Our key contributions are as follows:

- **Novel Utilization of Apple Neural Engine (ANE)**: A significant technical contribution of this project is our investigation into utilizing Apple's underused Neural Processing Unit (ANE), capable of up to 11 TOPS in INT8 precision [16]. While current LLM deployment frameworks such as LLaMA.cpp [17] or Ollama[18] do not harness this co-processor, we demonstrate and document methods to tap into the ANE

for local inference acceleration. We build on the NPU API reverse-engineering work by tinygrad [19] and leverage the learnings to open a new direction for efficient LLM deployment on Apple silicon devices. We hence leverage the NPU outside of traditional CoreML model deployment paradigm and demonstrate how it can be used for various use cases.

- **Retrieval-Augmented Generation (RAG) Architecture**: We implement a lightweight yet effective RAG pipeline [15] for performing QA and summarization tasks over local collections of documents. This enables the application to provide context-grounded, source-aware responses from user-specified corpora while leveraging limited compute resources.

- **Efficient On-Device Inference Using Quantized LLMs**: We leverage quantized transformer models [13], reducing memory and compute demands without compromising output quality. Instead of multi-gigabyte model downloads (as required by tools like Ollama [18] or LLaMA.cpp [17]), we use compact models (50–500MB) that support practical usage scenarios with minimal setup, enhancing portability and usability for non-technical users.

- **User-Friendly and Ready-to-Use Design**: Unlike tools such as Ollama [18] and LLaMAFile[20], which require technical familiarity and understanding the nuances of various models, our application provides a clean graphical interface with ready-to-use capabilities tailored to common academic needs—eliminating the steep learning curve and reducing operational friction.

- **Privacy-Preserving Document Analysis**: By running entirely on-device, our application mitigates the risks associated with uploading sensitive or proprietary documents to third-party services (e.g., ChatGPT, Claude, Gemini), which have raised concerns over data leakage [9, 10]. Users can securely summarize, query, and rephrase

information without network access or cloud APIs.

Through this suite of contributions, *Project TLDR* demonstrates that meaningful and secure LLM-powered applications can be brought directly to end-users without reliance on cloud services or specialized technical knowledge, thereby filling a critical gap in the current LLM applications ecosystem.

1.3 Scope

This project aims to develop a standalone desktop application capable of performing Question Answering (QA) and Summarization over a locally stored corpus of documents using Retrieval-Augmented Generation (RAG). Unlike most current solutions, such as ChatGPT or Gemini, which require users to upload documents each time they wish to query them, this application allows persistent storage and indexing of documents on the user's device. Once added to the local vector store, documents are automatically embedded and made queryable without requiring repeated user intervention.

The application is designed to operate entirely offline, preserving data privacy while minimizing resource consumption. It is optimized for modern Apple Silicon devices (e.g., M1/M2 Macs), with the target of using less than 50% of system resources. By utilizing quantized models ranging between 50MB and 500MB in size and streamlining the entire RAG pipeline—from document ingestion to local inference—this tool ensures meaningful results without the need for large downloads or technical configuration. The scope includes a graphical interface, local embedding and vector database management, and natural language generation capabilities tailored for academic and research use cases.

1.4 Paper overview

This paper is organized as follows. Chapter **??** presents the theoretical foundations relevant to the project, including an overview of Large Language Models (LLMs), key components involved in inference such as the context window and KV cache, and a discussion on the Apple

M1 System-on-Chip (SoC) architecture, with emphasis on the GPU and Neural Processing Unit (NPU). It also covers zero-copy file access techniques, quantization methods, risks of data leakage in LLM usage, and how CoreML leverages the NPU.

Chapter **??** surveys related work, highlighting limitations of existing desktop LLM tools like Ollama and LLaMaFile, prior work on Retrieval-Augmented Generation (RAG), and efforts in reverse engineering NPU APIs by the tinygrad project. Chapter 4 details our methodology and low-level design, including vector dump and memory-mapped reads, interfacing with the NPU, and the internal structure of the RAG pipeline. Furthermore, it describes our software architecture and implementation details, covering our use of SwiftUI, static library linkage, the design of the LLM context pool, and thread management strategies.

Chapter 5 presents our experimentation process and evaluation results using BERTScore across different models and quantization levels. Finally, Chapter 6 concludes the paper, discusses limitations, and outlines future work including the development of an NPU backend for llama.cpp.

Chapter 2

RELATED WORK

This chapter presents an overview of existing technologies and research efforts relevant to the development of efficient, on-device language model-based applications. It focuses on three major areas: Retrieval-Augmented Generation (RAG), efficient LLM runtimes like `llama.cpp`, and lightweight distribution and serving solutions such as Ollama and Llamafile.

2.1 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an architectural paradigm that enhances the factual accuracy and contextual relevance of large language models (LLMs) by incorporating external knowledge retrieval into the generation process. Unlike traditional LLMs that rely solely on internal model weights, RAG allows the model to fetch and condition its output on relevant documents retrieved from a corpus [21].

A typical RAG pipeline consists of the following phases:

1. **Query Formulation:** The input query from the user is either used directly or rephrased using prompt engineering techniques to improve retrieval relevance.

2. **Retrieval:** A vector store or dense retriever (e.g., using FAISS, Milvus, or Elastic-Search) is queried to obtain top-k semantically relevant documents based on vector similarity.

3. **Context Injection:** Retrieved documents are concatenated with the original query and formatted into a prompt.

4. **Generation:** The formatted prompt is passed to the language model to generate a context-aware and informative response.

RAG has become a cornerstone for building knowledge-intensive NLP systems, especially in enterprise search, question answering, and summarization tasks [22, 23].

2.2 `llama.cpp`

`llama.cpp` is a C++ implementation of LLaMA models developed by Meta, optimized for local inference on commodity hardware without GPU requirements. Built upon the GGML tensor library, it provides quantized inference for large models using CPU-friendly formats like 4-bit and 5-bit quantization, making it suitable for running models such as LLaMA, Mistral, and other open-weight transformers on devices ranging from laptops to Raspberry Pi [24].

Key features of `llama.cpp` include:

- Highly efficient CPU inference with quantized models.
- Cross-platform support (macOS, Linux, Windows).
- Integration with popular tooling such as LangChain and Open Interpreter.
- Support for multi-threaded inference and memory-mapped model weights for efficient memory usage.

This project is a foundation for many desktop LLM applications that require local execution and privacy-aware computation.

2.3 Ollama

Ollama is a developer-friendly platform for running LLMs locally with simplified model management and serving. It wraps models like LLaMA 2, Mistral, and Code LLaMA into a

streamlined runtime with a CLI and RESTful API, abstracting away hardware-specific setup and providing a plug-and-play experience for developers [25].

Ollama supports:

- Running quantized models locally with GPU acceleration where available.
- Seamless model downloading and serving.
- Custom model creation using a simple Modelfile syntax.

It is widely used for prototyping private, offline chatbots and assistants.

2.4 Llamafile

Llamafile, developed by Mozilla-Ocho, enables packaging a complete LLM runtime into a single, self-contained executable file [26]. It leverages the `llama.cpp` backend and Cosmopolitan Libc to build universal binaries that run across major operating systems (Windows, macOS, Linux) without requiring dependencies.

Notable features:

- Distributable as a single file under 1GB (depending on model).
- Useful for shipping LLM-based tools with zero-install requirements.
- Integrates with web frontends for local chatbot deployment.

Llamafile simplifies the deployment of on-device LLMs, especially in constrained environments or where installation overhead is a concern.

Chapter 3
RELATED WORK

This chapter presents an overview of existing technologies and research efforts relevant to the development of efficient, on-device language model-based applications. It focuses on three major areas: Retrieval-Augmented Generation (RAG), efficient LLM runtimes like `llama.cpp`, and lightweight distribution and serving solutions such as Ollama and Llamafile. It also looks at the work done by tinygrad project in attempting to reverse engineer the NPU API.

3.1 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an architectural paradigm that enhances the factual accuracy and contextual relevance of large language models (LLMs) by incorporating external knowledge retrieval into the generation process. Unlike traditional LLMs that rely solely on internal model weights, RAG allows the model to fetch and condition its output on relevant documents retrieved from a corpus [21].

A typical RAG pipeline consists of the following phases:

1. **Query Formulation:** The input query from the user is either used directly or rephrased using prompt engineering techniques to improve retrieval relevance.

2. **Retrieval:** A vector store or dense retriever (e.g., using FAISS, Milvus, or Elastic-Search) is queried to obtain top-k semantically relevant documents based on vector similarity.

3. **Context Injection:** Retrieved documents are concatenated with the original query and formatted into a prompt.

4. **Generation:** The formatted prompt is passed to the language model to generate a context-aware and informative response.

RAG has become a cornerstone for building knowledge-intensive NLP systems, especially in enterprise search, question answering, and summarization tasks [22, 23].

3.2 llama.cpp

`llama.cpp` is a C++ implementation of LLaMA models developed by Meta, optimized for local inference on commodity hardware without GPU requirements. Built upon the GGML tensor library, it provides quantized inference for large models using CPU-friendly formats like 4-bit and 5-bit quantization, making it suitable for running models such as LLaMA, Mistral, and other open-weight transformers on devices ranging from laptops to Raspberry Pi [24].

Key features of `llama.cpp` include:

- Highly efficient CPU inference with quantized models.
- Cross-platform support (macOS, Linux, Windows).
- Integration with popular tooling such as LangChain and Open Interpreter.
- Support for multi-threaded inference and memory-mapped model weights for efficient memory usage.

This project is a foundation for many desktop LLM applications that require local execution and privacy-aware computation.

3.3 Ollama

Ollama is a developer-friendly platform for running LLMs locally with simplified model management and serving. It wraps models like LLaMA 2, Mistral, and Code LLaMA into a

streamlined runtime with a CLI and RESTful API, abstracting away hardware-specific setup and providing a plug-and-play experience for developers [25].

Ollama supports:

- Running quantized models locally with GPU acceleration where available.
- Seamless model downloading and serving.
- Custom model creation using a simple Modelfile syntax.

It is widely used for prototyping private, offline chatbots and assistants. However, the users need to be technically savvy in cases of issues downloading or running the models. Furthermore, models often may not be quantized and could lead to downloading of model weights in order of many GBs.

3.4 Llamafile

Llamafile, developed by Mozilla-Ocho, enables packaging a complete LLM runtime into a single, self-contained executable file [26]. It leverages the `llama.cpp` backend and Cosmopolitan Libc to build universal binaries that run across major operating systems (Windows, macOS, Linux) without requiring dependencies.

Notable features:

- Distributable as a single file under 1GB (depending on model).
- Useful for shipping LLM-based tools with zero-install requirements.
- Integrates with web frontends for local chatbot deployment.

Llamafile is widely used for prototyping private, offline chatbots and assistants. However, it often requires users to be technically proficient, especially when encountering issues related to downloading or executing models. Moreover, many models are not pre-quantized, potentially resulting in downloads of several gigabytes of model weights.

3.5 Apple Neural Engine (ANE)

The Apple Neural Engine (ANE) is a custom neural processing unit designed by Apple to accelerate machine learning workloads on its silicon platforms. Introduced with the A11 Bionic chip, the ANE has evolved into a high-performance, low-power DMA-based inference engine embedded in Apple's M-series chips. This section synthesizes insights obtained by the reverse-engineering efforts of the `tinygrad` [19] project, to examine ANE's architecture, capabilities, and compilation flow.

3.5.1 Hardware Overview

The ANE operates primarily as a DMA engine optimized for convolutional operations and supports a wide range of neural network layers and fused operations. Its key hardware features include:

- **16-core architecture:** A 16-wide Kernel DMA engine for parallel computation.
- **5D Tensor Support:** Tensors are structured with width (column), height (row), planes (channels), depth, and group (batch).
- **Supported Data Types:** `UInt8`, `Int8`, and `Float16` (with `Float32` inputs automatically downcast).
- **Manually Managed 4MB L2 Cache:** Applied only to input/output data; weights are embedded in the compiled program.
- **Execution Unit:** Executes up to 0x300 micro-operations per instruction.
- **Performance:** Approximate 11 TOPS throughput, assuming 32×32 MAC at 335 MHz.

All memory strides are constrained to multiples of 0x40 bytes, reflecting hardware alignment requirements.

3.5.2 Software and Compilation Stack

The ANE software stack is heavily abstracted behind Apple's proprietary frameworks but has been reverse-engineered to reveal a structured flow (as shown in Fig 3.1):

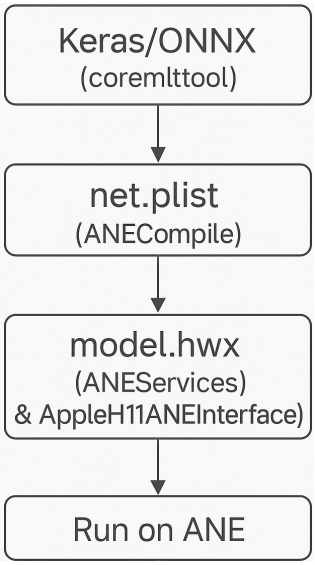


Figure 3.1: Apple Neural Engine Workflow

1. **Model Definition:** Models are authored in Keras or ONNX.

2. **Conversion:** Models are converted to CoreML format using open-source tools such as `coremltools`.

3. **Intermediate Representation:** CoreML is internally converted into `net.plist` by Apple's Espresso framework.

4. **Compilation:** The `ANECompiler` service transforms `net.plist` into a hardware-specific binary (`.hwx`), a Mach-O formatted executable.

5. **Execution:** The `AppleNeuralEngine` and `ANEServices` handle execution via the kernel extension `AppleH11ANEInterface`.

3.5.3 Instruction Format and Operation Structure

Each ANE instruction is 0x300 bytes and comprises multiple segments:

- **Header:** Includes DMA addresses and next-op offset.
- **KernelDMASrc:** Specifies weights, bias, and channel usage.
- **Common:** Describes input/output shapes, types, kernel size, and padding.
- **TileDMASrc/TDMADst:** Layout and stride configurations for input/output tensors.
- **L2 and NE:** L2 cache flags and activation parameters.

3.5.4 Supported Operations and Activations

ANE supports a variety of operations:

- **Core Ops:** CONV, POOL, EW, CONCAT, RESHAPE, MATRIX_MULT, TRANSPOSE
- **Advanced:** SCALE_BIAS, SOFTMAX, INSTANCE_NORM, BROADCAST, L2_NORM
- **Fused Ops:** NEFUSED_CONV, PEFUSED_POOL, etc.
- **Activations:** RELU, SIGMOID, TANH, CLAMPED_RELU, PRELU, LOG2/EXP2, CUSTOM_LUT

Over 30 activation functions are supported in hardware.

3.5.5 tinygrad Implementation

The `tinygrad` project interfaces directly with ANE through a three-stage pipeline:

- **1_build:** Generates CoreML models using `coremltools`.
- **2_compile:** Uses Objective-C and Apple's private ANECompiler framework to compile models into HWX binaries.
- **3_run:** Loads HWX binaries and executes them on ANE using custom Objective-C wrappers around `AppleH11ANEInterface`.

The implementation also includes tools like `hwx_parse.py` for disassembling HWX files and visualizing internal ops.

3.5.6 Security and Access

Execution on ANE requires system entitlements that are typically unavailable to third-party applications:

- `com.apple.ane.iokit-user-access`
- Workarounds: amfid patching, kernel extension modification, or use of provisioning profiles.

3.5.7 Conclusion

The ANE represents a proprietary, highly optimized inference accelerator that is difficult to access and understand due to Apple's closed ecosystem. Reverse engineering, as demonstrated by `tinygrad`, reveals a modular, DMA-centric architecture capable of executing complex neural network operations at high throughput and low latency. As Apple continues to iterate on the ANE, deeper access and tooling may unlock broader ML deployment options on Apple hardware.

Chapter 4

METHODOLOGY-TODO

This chapter covers the procedures involved in this research, from data collection to the proposed DL model's architecture and its training, how it is evaluated, and all the tools used during this study. It provides an explanation of how the solution for this study works and its justification.

4.1 Data Collection

The dataset for this study consists of fluid flows interacting with an obstacle. Its purpose is to train and test the neural network solution proposed in this research. The fluid flows are generated using a numerical method, as is typically done in computational fluid dynamics simulations. The sequences represent the evolution of the fluid flow in space and time; therefore, this dataset could be considered two-dimensional Time Series data (See Section **??**).

Each fluid flow is a sequence of 400 frames representing the state at a given time. The frames are a two-dimensional grid discretization of the simulated space, with a resolution of 200-width by 100-height cells. The values in the cells represent either the fluid's velocity at the position or -1 if it belongs to the obstacle.

All the fluid flows in the dataset have a Reynolds number of 220 and flow from left to right, interacting with an obstacle with several dimensions and positions. These obstacles are either circumferences or ellipses, as shown in Figure 4.1. Circumferences are simple shapes to model and are usually used to demonstrate fluid flow simulations. In contrast, ellipses are an easy simplification of an aircraft wing cross-section with different dimensions and inclinations (known as the "angle of attack").

In aeronautics, several pre-defined airfoils shapes (NACA airfoils) have been studied for

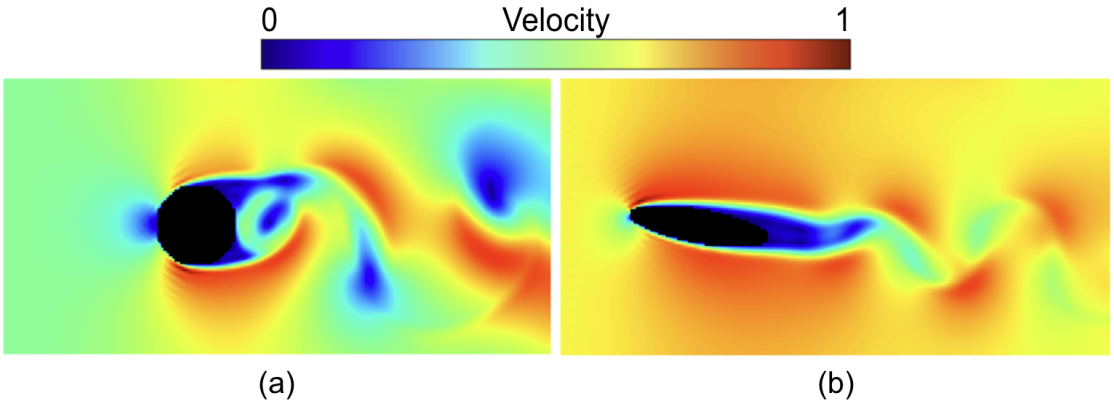


Figure 4.1: Two CFD frames from different sequences showing the (a)circumference and the (b)ellipse obstacles

the design of aircraft wings [**?**]; in particular, the NACA 2412 is used in the popular Cessna 172 Skyhawk. Figure 4.2 shows examples of those airfoils. However, these shapes are very complex to calculate. As an alternative for this work, an ellipse shape was chosen as an approximation to the NACA airfoils. The ellipse geometry has a smooth, curved shape that can resemble the streamlined profile of the airfoil. It can provide a good approximation for the leading and trailing edges, capturing the essential aerodynamic characteristics while simplifying the complex geometry of the airfoil. This approximation is particularly useful for preliminary design and analysis, where exact precision is less critical. It allows for easier mathematical manipulation and analysis compared to the exact airfoil shape.

4.1.1 Definitions

The fluid flow sequence data can be represented in the following mathematical way. Data is taken from velocity observations in a spatial region on a $W \times H$ grid, which consists of W columns and H rows. Each grid cell has a velocity measurement that varies over time T.

Definition 1. Let T be a finite period of time and $t_i \in T$, where $i \in \{0, 1, ..., T\}$, is a time instance.

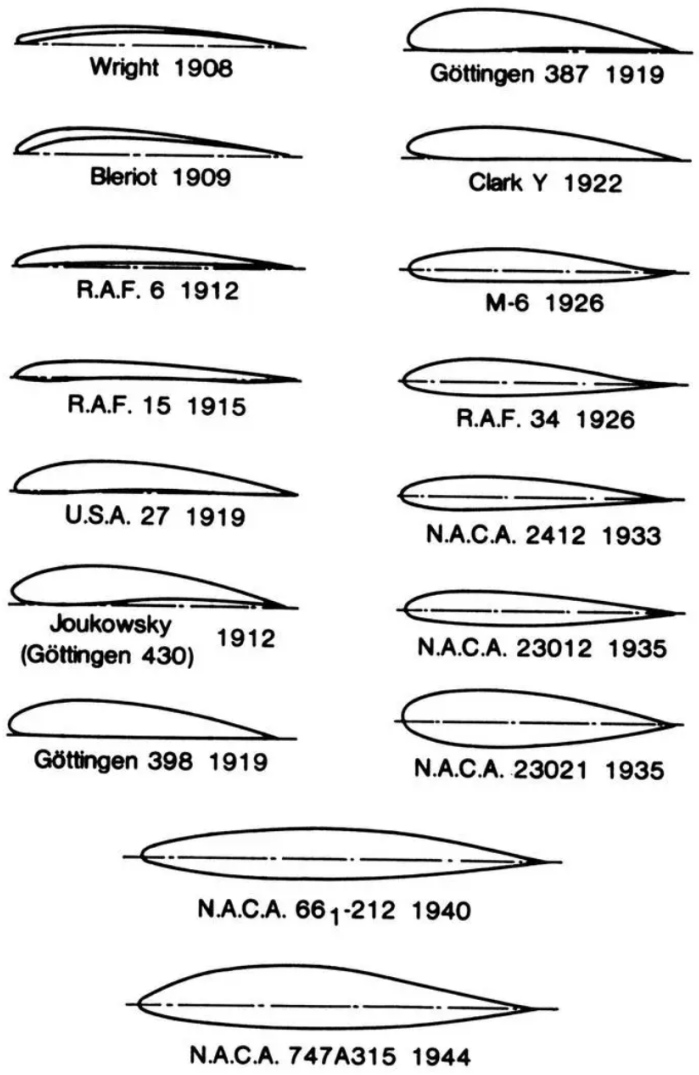


Figure 4.2: The historical evolution of airfoil sections from 1908-1944. Credit: NACA-NASA

We can then mathematically represent a fluid flow sequence as follows:

Definition 2. A fluid flow sequence is represented by a tensor $X \in \mathbb{R}^{T \times W \times H}$, where $x \in X$ and $x \in \mathbb{R}^{W \times H}$ is a velocity observation or fluid state at any given time $t \in T$.

Definition 3. Let $\mathcal{W}$ be a window of time $\subseteq T$, where $\mathcal{W}$ is of size m and $1 < m < T$.

When observations of $x \in X$ are recorded periodically over a time period T, we can think

of them as a sequence of frames $x_0, x_1, x_2, ..., x_i, ..., x_T$. For this research, the spatiotemporal sequence generation problem is to generate the next most likely frame x_i observation, given a window of previous observations $x_{i-m}, ..., x_{i-2}, x_{i-1}$. The problem can be formalized by equation 4.1, where g represents the *generate* function performed by the model.

$$x_i = g(x_{i-m}, ..., x_{i-2}, x_{i-1}) \tag{4.1}$$

4.2 Data Preparation

Before the data is used to train and evaluate the model, the following preprocessing steps are applied to transform the data: 1) data normalization, 2) data slicing, and 3) dividing the dataset into a training and testing set.

1. **Data normalization:** First, the data is normalized by scaling the velocity values between 0 and 1. This data normalization improves the gradient descent optimization during the neural network's training, which is a common requirement for deep learning methods. This scaling is done according to Equation 4.2 below.

$$v_{scaled} = \frac{v - v_{min}}{v_{max} - v_{min}} \tag{4.2}$$

   This scaling technique provides robustness to very small standard deviations in the dataset's velocity values. During this process, the obstacle cells are left with a value of -1 to distinguish them from the fluid while maintaining the velocity values in the 0 to 1 range.

2. **Data slicing:** This step focuses on creating simulated sub-sequences to train the model. The input and output datasets are generated using the original dataset, which consists of the simulated [fluid] sequences. The model takes as input a specific window $\mathcal{W}$ consisting of m frames from the simulated sequence of size T; it then uses this window to generate the next frame. Since $m < T$, the length of the input dataset

elements must be reduced to match the this window. To do this, each original sequence is segmented into sub-sequences of length m (size of $\mathcal{W}$). After segmenting the original sequence of size T, it results in more than one sub-sequence of size m in the input dataset.

Figure 4.3 illustrates this process, where the input dataset to the model (the set of fluid sub-sequences X of size m over a time window $\mathcal{W}$ using definition 3), will be used to generate the output set (the next set of frames). During the training step, the model will learn how to infer x_t from the previous m frames (see Equation 4.1).

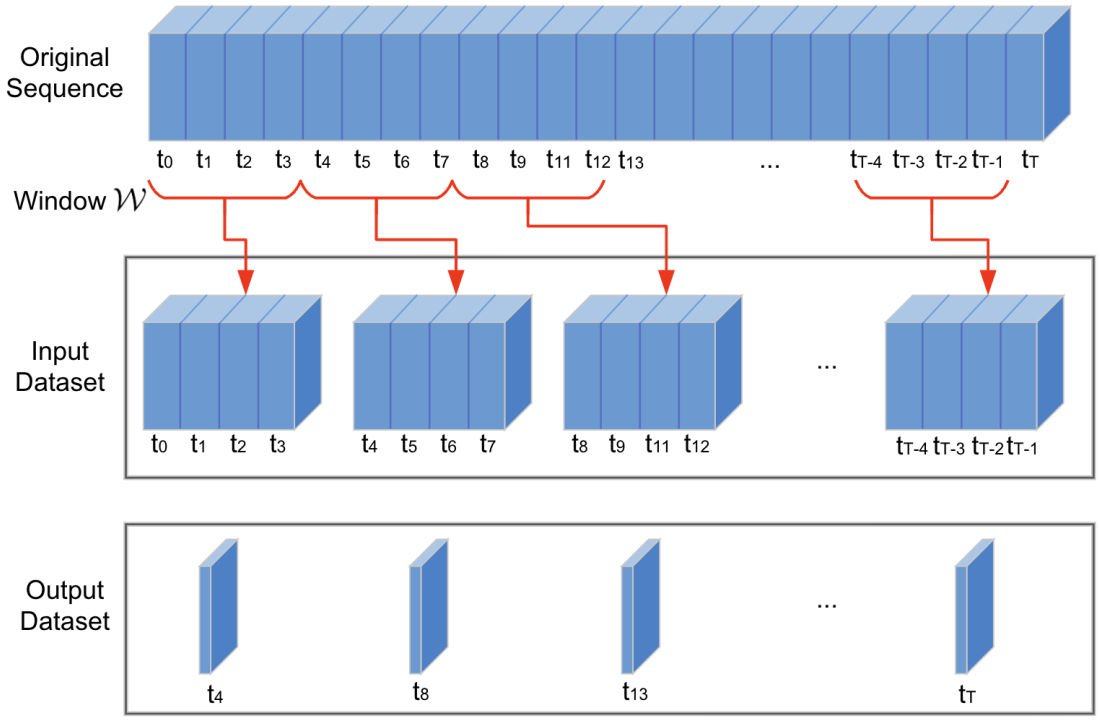


Figure 4.3: Slicing of sequence into $\mathcal{W}$ sub-sequences used to train the model.

3. **Training, Validation, and Testing sets:** Finally, the dataset is shuffled and then randomly divided into training, validation, and testing sets with an 8:1:1 ratio, respectively. This is done to validate and test the model using the cross-validation method

with samples not used during training. This aims to provide an unbiased evaluation of the model's efficacy, which – ideally – should appropriately generalize (for new data) without over-fitting (training data).

4.3 Model Architecture

The DL solution proposed in this research is an end-to-end model, meaning it will perform all the tasks from data input to generating the fluid flow simulation output in one unified process. In contrast, other related research uses machine-learning or DL techniques for only a specific part of the simulation, leaving the rest to a classical numerical method and making the process more complex. A benefit of our simple approach with only one task is eliminating the need for complex pipelines between separate parts in the simulation process, thus reducing development time and potential sources of error. Additionally, end-to-end models can better adapt to diverse datasets and changing environments since they learn directly from raw data, capturing intricate patterns and relationships that might be missed in traditional approaches, like DNS.

The goal of this model is to generate predictions of a fluid flow's evolution. To accomplish this, the model looks at past states in the flow and generates the following future state. The future state can then be used as an input to continue generating new states in the simulation. This step can be repeated as many times as necessary as a feedback loop shown in Figure 4.4 below. As a result of this feedback loop, the model can produce a long fluid flow sequence. In Figure 4.4.a we can see how the resulting frame is put at the end of the input sequence to generate a new frame (see Section 4.3.2). Figure 4.4.b shows how the model looks at a certain window of the frame and moves forward in time to generate the rest of the sequence. At the beginning of a simulation, the model uses an initial condition or "ground truth" represented by a number of frames equal to the sliding window ($\mathcal{W}$). As $\mathcal{W}$ slides, new frames are generated, which are in turn used to generate more subsequent frames. Eventually, an entire sequence can be generated, knowing only the first frames from the initial fluid's condition, and the rest are completely generated by the model.

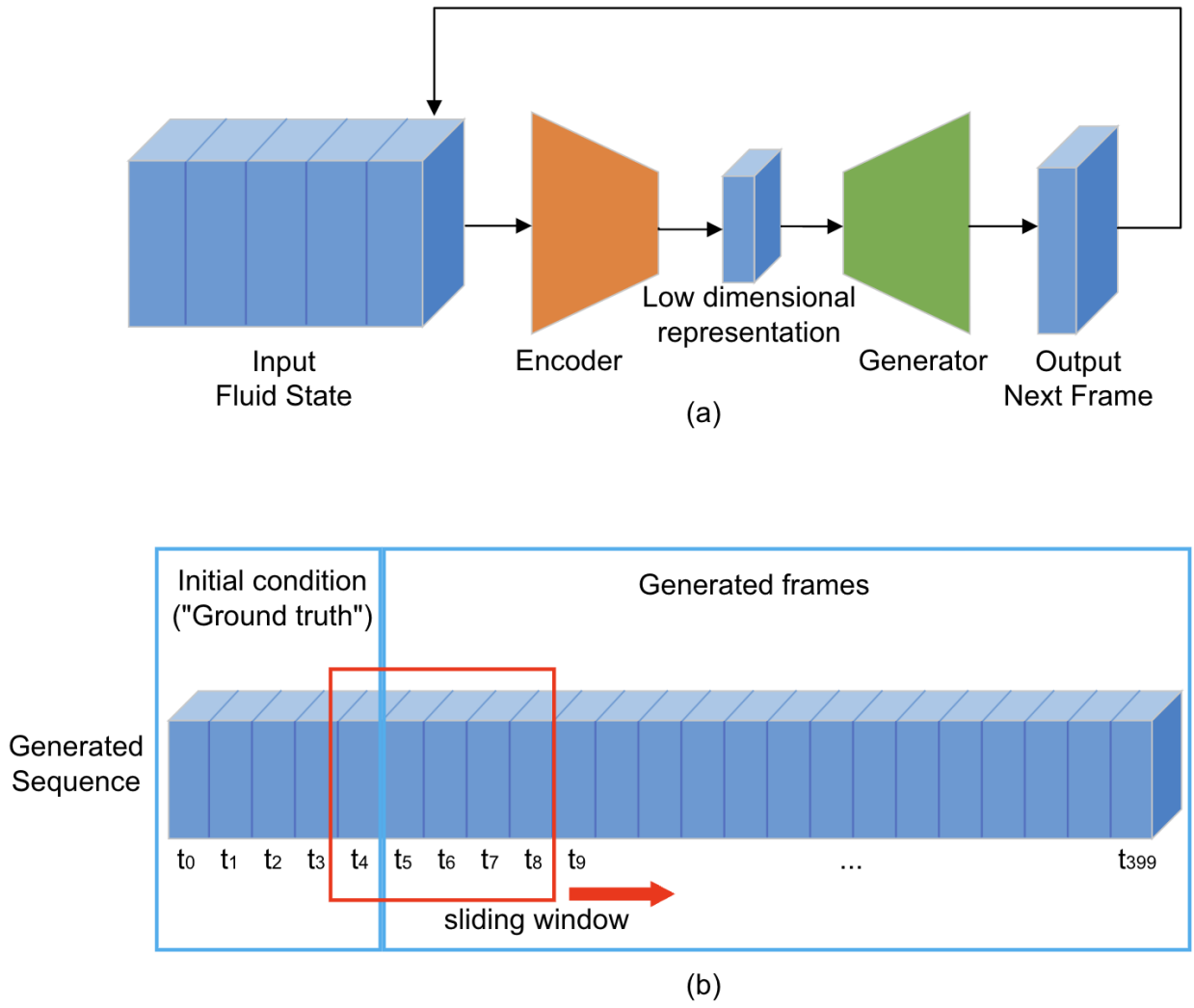


Figure 4.4: Diagram showing a) the flow between the model's main components and b) sliding window approach for prediction.

Because of the data's spatiotemporal characteristics, the neural network has to be capable of analyzing and learning the evolution of fluid flows in two dimensions: space and time. A CNN (See Section **??**) can help understand the spatial structure of the fluid to extract the flow's features and patterns in space. Additionally, the model needs to "remember" what happened in the past to produce the next state, so it needs a memory mechanism that can be provided by a recurrent neural network such as an LSTM (See Section **??**), commonly used in Natural Language Processing tasks. As mentioned before, these two types of neural networks have been combined to create the ConvLSTM (See Section **??**) as an extension of the LSTM network that can also "look" for features in space and time. For this reason, the ConvLSTM network was chosen to implement the neural network architecture proposed in this study.

Because this data has many dimensions and complexities, a dimensionality reduction is applied to capture the principal components of the flow before generating the next frame. This ensures that the model will rely on a minimal representation of the fluid flow that accurately describes its behavior. For this model architecture, the dimensionality reduction is implemented by an Autoencoder (See Section **??**) neural network that can produce it as part of the same model.

In summary, the model has two main parts, as shown in Figure 4.5: 1) an Encoder that reduces the data's dimensionality and 2) a Generator that gets a representation of the past frames and creates the next one.

4.3.1 Encoder

The Autoencoder is a neural network architecture composed of an Encoder and a Decoder, and it is used for dimensionality reduction (See section **??**). It takes data with many dimensions and creates a representation of this data in a lower-dimensional space. It is used similarly to classical statistical methods, such as singular value decomposition or principal component analysis. As explained in the related works chapter **??**, autoencoders were previously used for fluid flow analysis, such as identifying the main components in a fluid flow

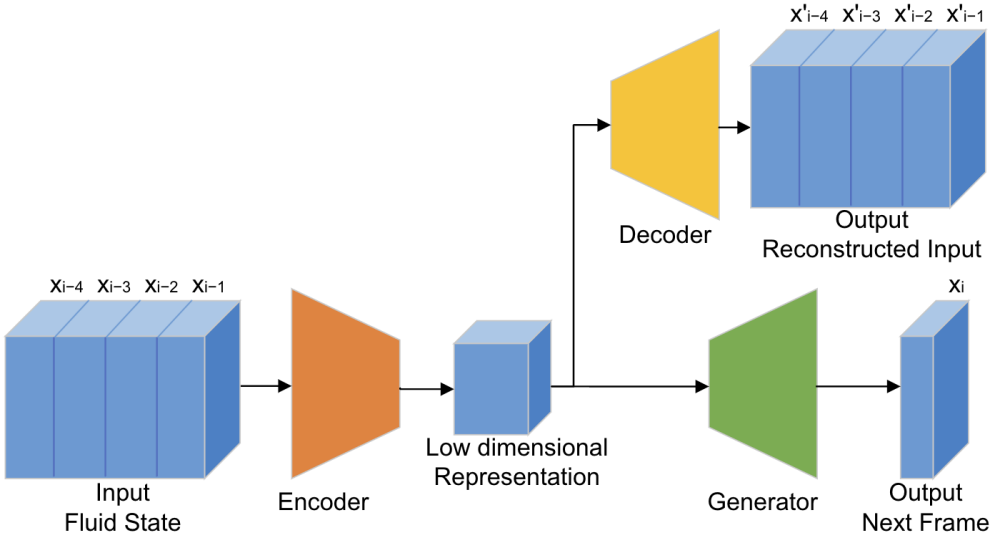


Figure 4.5: Schematic diagram of the model architecture

and identifying eddies in the flows. It has also been used for dimensionality reduction for other methods that are not deep learning models.

4.3.2 Generator

The generator takes the lower-dimensional representation of the fluid flow state and generates a new frame in the sequence. Using a lower representation of the data instead of all the original dimensions makes this work easier because the generator will get as input the main components that can describe the flow.

4.4 Model training

The Encoder is trained in conjunction with the Decoder component, which takes the lower dimensionality representation and tries to reconstruct the original input. If the decoder can reconstruct the original data using the reduced representation from the encoder, this means that the representation captures the main elements of the sequence, and the encoder works correctly. This decoder is auxiliary and discarded once the model's training is completed.

Cross-validation [**?**] is used to ensure that the model performance generalizes well to unseen data during training. This method, commonly used in machine learning, involves splitting the dataset into three parts: a training set, a validation set, and a testing set. The training set is used to train the model, and simultaneously, the validation set is used to evaluate the model's performance during training. Once the model is trained and optimized, it is tested on a separate and unseen test dataset to assess its generalization performance. This technique also helps prevent overfitting by providing an independent dataset for model evaluation. It ensures that the model's performance estimates are more reliable and indicate its performance on new data.

The neural network training job was distributed across the two GPUs available on the server. This was done using the Mirrored Strategy, a synchronous data parallelism algorithm for neural network training. This algorithm replicates the model on both GPUs and splits the dataset between devices. The CPU prepares and sends the data batches to the GPUs. During training, each GPU performs a forward pass over the model on different input data to compute the loss function; subsequently, gradients are calculated on each device based on that loss. Both sets of gradients are then combined with an all-reduce operation by averaging them and re-distributing them across devices to update the model parameters on each replica, synchronizing them. This process is repeated for each data batch for every training epoch.

4.5 Hyperparameter Optimization

The model architecture and training have specific characteristics or hyperparameters that can take various possible values. These hyperparameters impact the model's performance, i.e., different combinations of such values may result in different model efficacy; consequently, it is important to find a suitable hyperparameter combination to optimize the model efficacy.

Hyperparameter optimization with Random Grid Search [**?**] in ML involves systematically exploring a predefined hyperparameter space by randomly sampling combinations of hyperparameters, rather than exhaustively searching through all possible such combina-

tions. This approach helps efficiently find an optimal set of hyperparameters for the model by balancing computational resources and exploring the parameter space. Random grid search randomly selects hyperparameter values from specified distributions and evaluates each combination using cross-validation to determine the set that yields the best performance metric. This technique can effectively search a large hyperparameter space, improving model performance without exhaustive computational costs.

The hyperparameters considered for this model are:

- Learning rate: it determines the update step size of the weight in the neural network at each iteration during the optimization process, i.e., the loss function moving towards a minimum. A high learning rate can lead to rapid convergence at the risk of overshooting the optimal solution, causing the model to diverge. On the other hand, a low learning rate ensures steady convergence at the sake of a slower training process which can get stuck in local minima. This parameter significantly impacts the efficiency and effectiveness of the model training. The best training results were achieved with a learning rate of 0.001.

- Number of layers: the number of neural network layers affects the *capacity* of the model to learn complex representations. In the context of this research, more layers can enable the model to capture more hierarchical features and intricate patterns in the fluid-flows. This can improve the model's performance. However, having more layers makes the model more complex and more susceptible to overfitting the training data. The model achieved the best results using 3 ConvLSTM layers on each Encoder and Decoder component and 4 ConvLSTM layers in the Generator component.

- Number of filters or kernels on each layer: convolutional filters detect spatial features in the input data. These filters enable the model to capture both spatial and temporal dependencies in the fluid flow. The number of filters impacts the model's ability to extract relevant features. Having more filters can capture more complex data patterns

but at the cost of increasing computational cost. This parameter affects the accuracy and efficiency of the model. In the ConvLSTM layers of the Autoencoder, this model architecture got the best results using 64 filters in the first and last layers and 32 in the other ones. In the Generator component, the first layer uses 32 filters and 64 in the rest.

- Size of the filter: this refers to the convolutional filters' dimension to capture features of the input data. The filter size determines the scope of the local spatial region examined by the model. Larger filters can capture broader spatial patterns but may miss fine-grained details, while smaller filters focus on more localized features, potentially overlooking broader context. This impacts the model's ability to detect relevant features. The following filter sizes in the ConvLSTM layers of the architecture were used to get the best results, in order from the first to last layer of each component: the Encoder has filter sizes of 4 by 4, 3 by 3, and 2 by 2; the Decoder has filters of 2 by 2, 3 by 3, and 4 by 4; and the Generator filter sizes are 3 by 3 for all of its layers.

In order to define a search space of hyperparameter combinations, a range of values is set for each of these hyperparameters. Potential combinations are selected by random sampling, then a model is trained using those combinations, and the one with the best performance is selected. Once the best combination is found, the model is trained during more epochs to get the final version of the model.

When choosing the m size of the window $\mathcal{W}$, several aspects were considered, e.g., the amount of "historical" data used to generate a prediction. Similar to the Exponential Moving Average (EMA) heuristic [**?**] [**?**], which can be used in time-series forecasting scenarios and LSTM models to weight the amount and relevance of historic measurements vs. predicted ones. In our case, we can think of m as the amount of historical information that the model will receive to make a future prediction. In this sense, using a small window will not give enough information to the model. On the other hand, having a bigger window size provides more information and is better for the model, but if m is too big, it will need too much

historical information, making the model pointless. Additionally, a bigger window expands the input dimensions, so more computations would be required to compress the data to create the lower dimensional representation. This increase in computations makes the model slower to execute and train, which given our limited computational resources, could render our model impractical. In addition, since the sequences have 400 frames each, the window size m has to be a divisor of 400 to split them into sub-sequences for data preparation, as explained in 4.2. Overall, since a window of size 2 would be too small to provide the model with enough information, a window size of 4 was chosen since it is the smallest m that allows for the even division of sub-sequences while keeping the model practical.

4.6 Proposed Architecture Details

The final model architecture has two components: the **Autoencoder** and the **Generator**. The Autoencoder is divided into the Encoder and Decoder. The window $\mathcal{W}$ of $m = 4$ frames is first input into the Decoder, which creates a low-resolution representation of the simulation. Next, this low-resolution representation is input into the Generator to create the next frame state of the fluid flow. All these components are implemented in the neural network by a set of ConvLSTM layers followed by either MaxPooling or UpSampling layers. ConvLSTM layers extract features in the data by using convolutional operators called filters or kernels. MaxPooling and UpSampling layers compress or uncompress the intermediate data outputs between ConvLSTM layers. MaxPooling downsamples the input by taking the maximum value in the kernel window along the spatial dimension. UpSampling layers resize the input by interpolating its values. Figure 4.6 shows a detailed diagram of this architecture where we can see the order of these layers. In the Encoder, successive layers of ConvLSTM and MaxPooling layers downsample the data, resulting in a representation that is half the input size. This representation is then reconstructed in the Decoder by upsampling it using UpSampling and ConvLSTM layers. Each of these layers has a different amount of filters or pool windows of different sizes, respectively. Figure 4.6 also shows the output dimension of each layer.

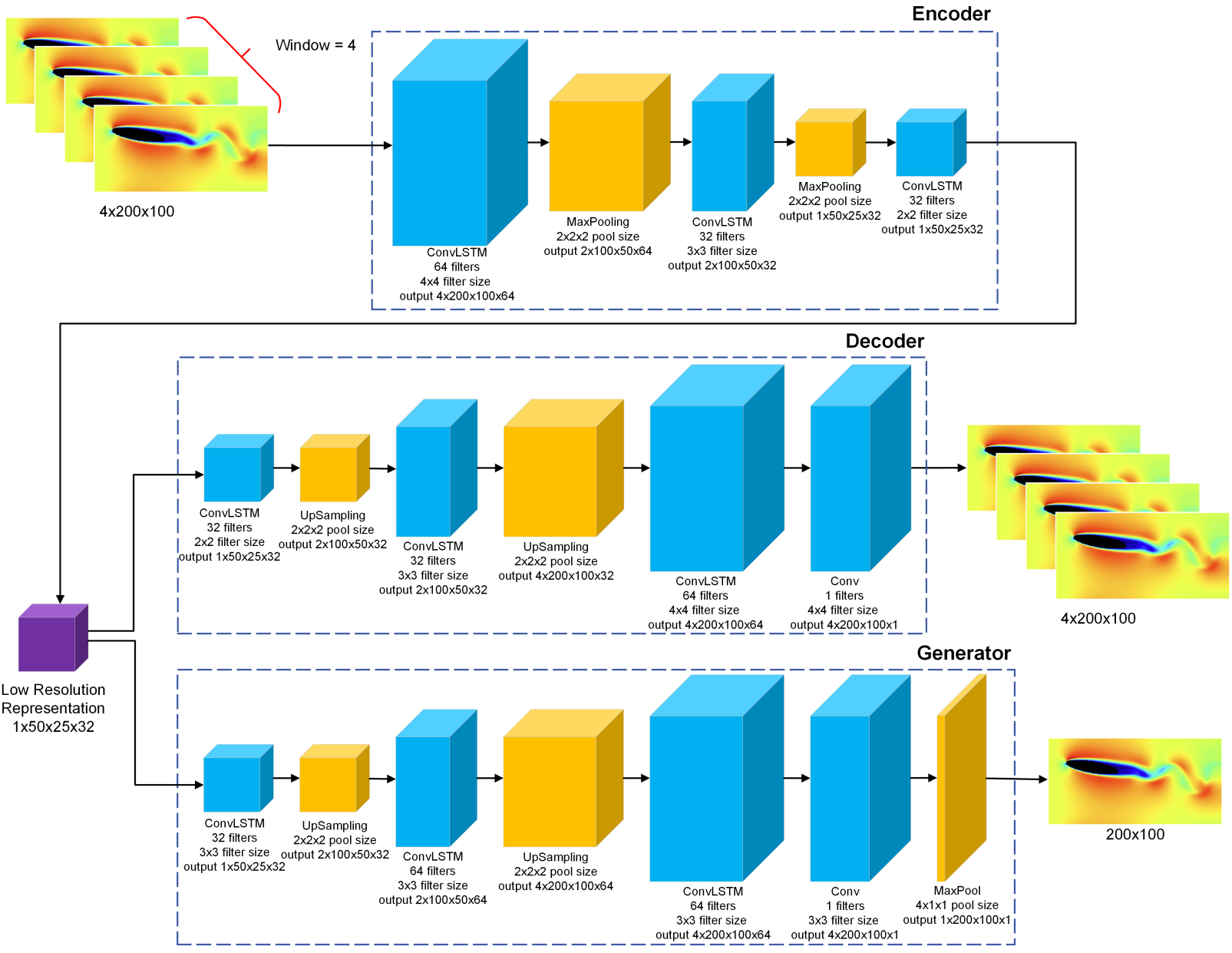


Figure 4.6: Detail diagram of the model architecture

4.7 Evaluation Methods

The model evaluation is done using three strategies:

1. Calculating the Mean Square Error between the original and generated simulations.

2. Visually inspecting the simulation result by rendering an image with the generated data and comparing it with the expected data (the ground truth).

3. Creating a histogram of frame values for the original and the generated data, these will be compared "side-by-side" to analyze similarities and identify possible drastic differences.

4.8 Software and Tools

For the implementation of all the methods described in the previous section, the Python [**?**] scripting language with the following libraries: TensorFlow [**?**] and Keras [**?**], for the neural network implementations and training; Numpy [**?**] and scikit-learn [**?**], for data manipulation and preprocessing; and Matplotlib [**?**], to generate the plots and visualizations.

VS Code was used as an IDE for code implementation, and data analysis of the results was done using Jupyter Notebook [**?**].

The server used for training and evaluation of the neural network model has the hardware specifications described in Table 4.1 below.

Table 4.1: Server Hardware specifications

CPU	Intel(R) Xeon(R) Gold 5118 CPU 2.30GHz of frequency 12 cores
RAM	192 GB
GPU	2x Nvidia Tesla V100 with 16 GB or RAM each Nvidia Volta Microarchitecture Compute Capability 7.0

Chapter 5

EXPERIMENTATION-TODO

This chapter covers the experiment design and setup, describing in detail results obtained during different phases and tests. This chapter is organized as follows: first, we discuss the results of the training and its validation in Section 5.1, then in Section 5.2 and Section 5.3 we explore the DL model performance from the perspective of each component individually using the methods defined in Section 4.7. Lastly, in Section 5.4, we analyze the performance of the DL model based on the two main evaluation metrics defined in Section **??** (MSE and execution time).

Results from experiments in this chapter are based on the dataset described in Section 4.1. The experiments performed can be categorized into two main types based on the obstacle shape used:

1. Circular obstacle: the obstacle is randomly positioned in the simulation space, and its size is given by radius $\mathbf{r} = qH$, where $q \in [\frac{1}{9}, \frac{1}{5}]$ and H is the height of the simulated region.

2. Elliptical obstacle: the obstacle is also randomly positioned in the space, and its size is given by semi-major axis $\mathbf{a} = qH$, where $q \in [\frac{1}{5}, \frac{1}{3}]$ and H is the simulated region height, and the semi-minor axis $\mathbf{b} = p\mathbf{a}$, where $p \in [\frac{1}{5}, \frac{1}{4}]$. The ellipse is also tilted with an angle $\boldsymbol{\alpha} \in [-30°, 30°]$ with respect to $\mathbf{a}$ and the Cartesian x-axis.

The ranges of the obstacle dimensions and positions were chosen to fully fit the shape inside the simulated space without touching its perimeter. The dimensions of the Ellipse object were chosen to maintain a shape similar to that of an airfoil as described in Section 4.1,

and its inclination range to represent common wing "angle of attack" including the critical or stalling "angle of attack" (typically between 18° and 25° degrees)[**?**].

5.1 Training Results

The neural network was trained for 500 epochs, across which, the Mean Squared Error (MSE) loss functions results were collected for both the Autoencoder and the Generator. The training error is captured in Both plots in Figure 5.1 show that the is generally lower than the validation error; this is to be expected given the proposed model was evaluated with sequences used during the training process.

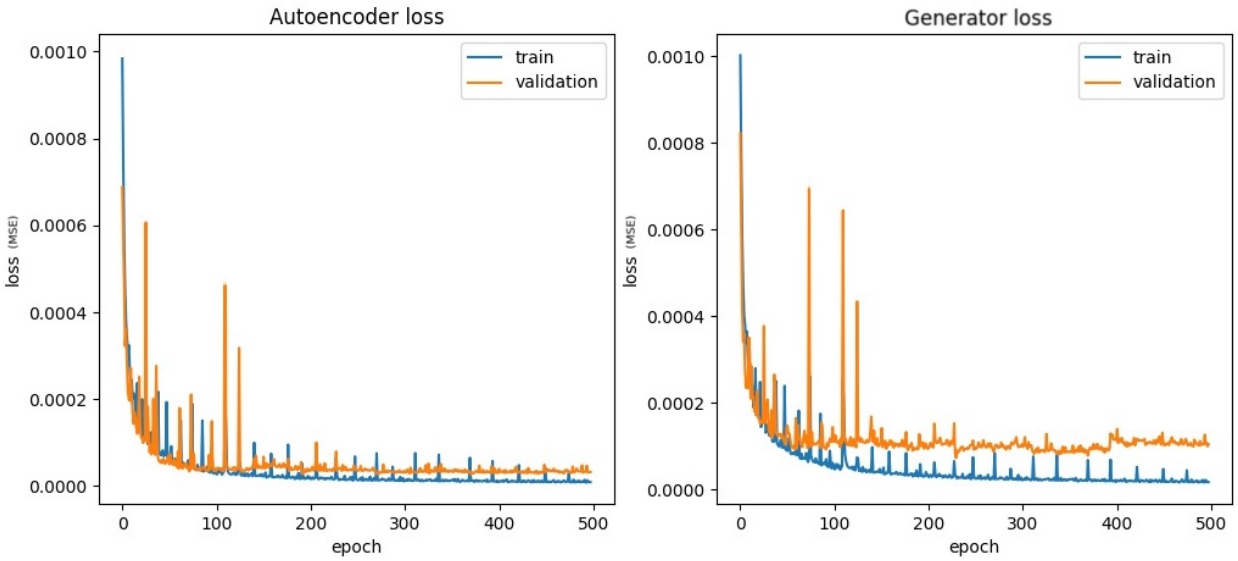


Figure 5.1: Autoencoder and Generator loss evolution during training

Figure 5.1 demonstrates the stability of the MSE loss function's evolution during training. The loss trend quickly descends, and after 200 epochs, the error stabilizes and converges to its final value. In addition, note that the slight difference between the training and validation errors signifies that the model does not excessively over-fit the training data. Regular error spikes appear during training, representing instances where the model encountered local

maximums before finding a local minimum. This happens when, during parameter optimization, a new combination of the network's weights is worse than the previous one, but then it improves again. This further underscores the model's reliability. These spikes decrease as the training advances, indicating a stable and reliable training process. Another critical aspect to note is that those spikes happen almost on the same epoch number and with a similar intensity for both of the model's components (Autoencoder and Generator), meaning that these components are working in collaboration to find the best result. This supports the election of the model architecture.

Table 5.1: Training and Testing errors (MSE)

	Autoencoder	Generator
Training	7.7727×10^{-06}	1.5429×10^{-05}
Validation	3.2173×10^{-05}	2.0414×10^{-05}

A comparison between training and validation errors is displayed in Table 5.1 for both the Autoencoder and Generator components of the model. The low error rate in all cases indicates the model's good performance. It is important to remember that validation samples were not used during training, which is noticeable in how lower the training errors are compared to the validation errors. Since the model has already "seen" the training samples to optimize its weights, it learns how to approximate the output based on those values.

In the following sections, we analyze the model's performance more deeply, examining each component's performance individually.

5.2 Autoencoder Results

To evaluate the Autoencoder, we need to verify that the Decoder can reconstruct the original information using the low-resolution representation of the data created by the Encoder. Because the input to the model is a set of frames according to the window described in

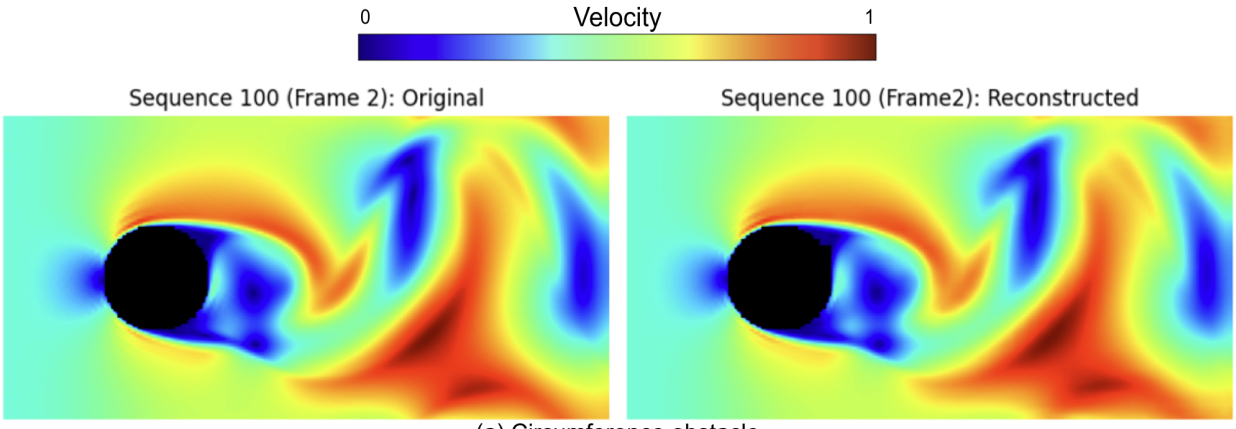
Chapter **??**, the Decoder output will also have those dimensions. This means that to evaluate the Decoder's output, we must compare all the frames in the set.

We did this evaluation with two methods:

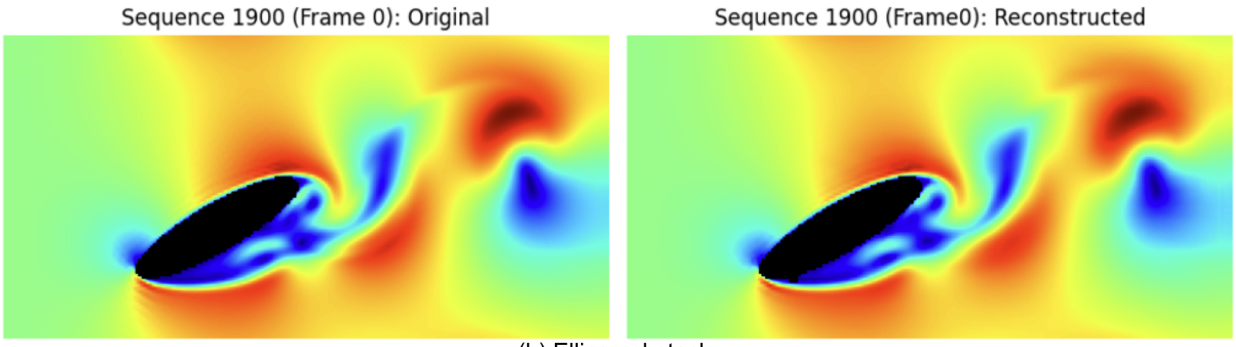
1. Comparing the reconstructed frames with the originals.
2. Comparing the histogram of velocity values.

Using the Decoder's output sequence representing the fluid state, a *heatmap* of fluid velocity values was rendered to compare the results visually. Figure 5.2 shows examples of frames with each type of obstacle. In each case, we show the Original frame to the left and the Reconstructed frame output by the Decoder to the right. We can see that although there are some minor differences, both frames are almost identical. Similar results were obtained for the rest of the sequences. The differences found can be seen in small changes of intensity of the *heatmap* colors, which may indicate that the velocity values approximate the original ones but are either lower or higher than expected. Although there are some minor differences, the similarities between the Original and Reconstructed frames give us a clue that the Autoencoder is working as intended.

For the following evaluation method, we compare the velocity values between the Original and Reconstructed frames to verify their similarity. This is done by creating a histogram of velocities, i.e., a frequency count of velocity values on each grid cell in the frame. Figure 5.3 shows examples of those histograms for Origianl and Reconstructed frames containing each type of obstacle. The frame examples are the same as Figure 5.2 used in the previous evaluation method. On the x-axis, we have the range of all the velocity values in the frame. These values are between 0 and 1 because the data was previously normalized, as explained in Section 4.2. On the y-axis, the frequency or occurrence of each velocity value is represented. To compare all the velocity histograms, we calculated the Jensen-Shannon distance between each frequency distribution. The resulting average distance between the original and reconstructed frames was 0.021, with a standard deviation of 0.014.



(a) Circumference obstacle



(b) Ellipse obstacle

Figure 5.2: Original vs Reconstructed frames

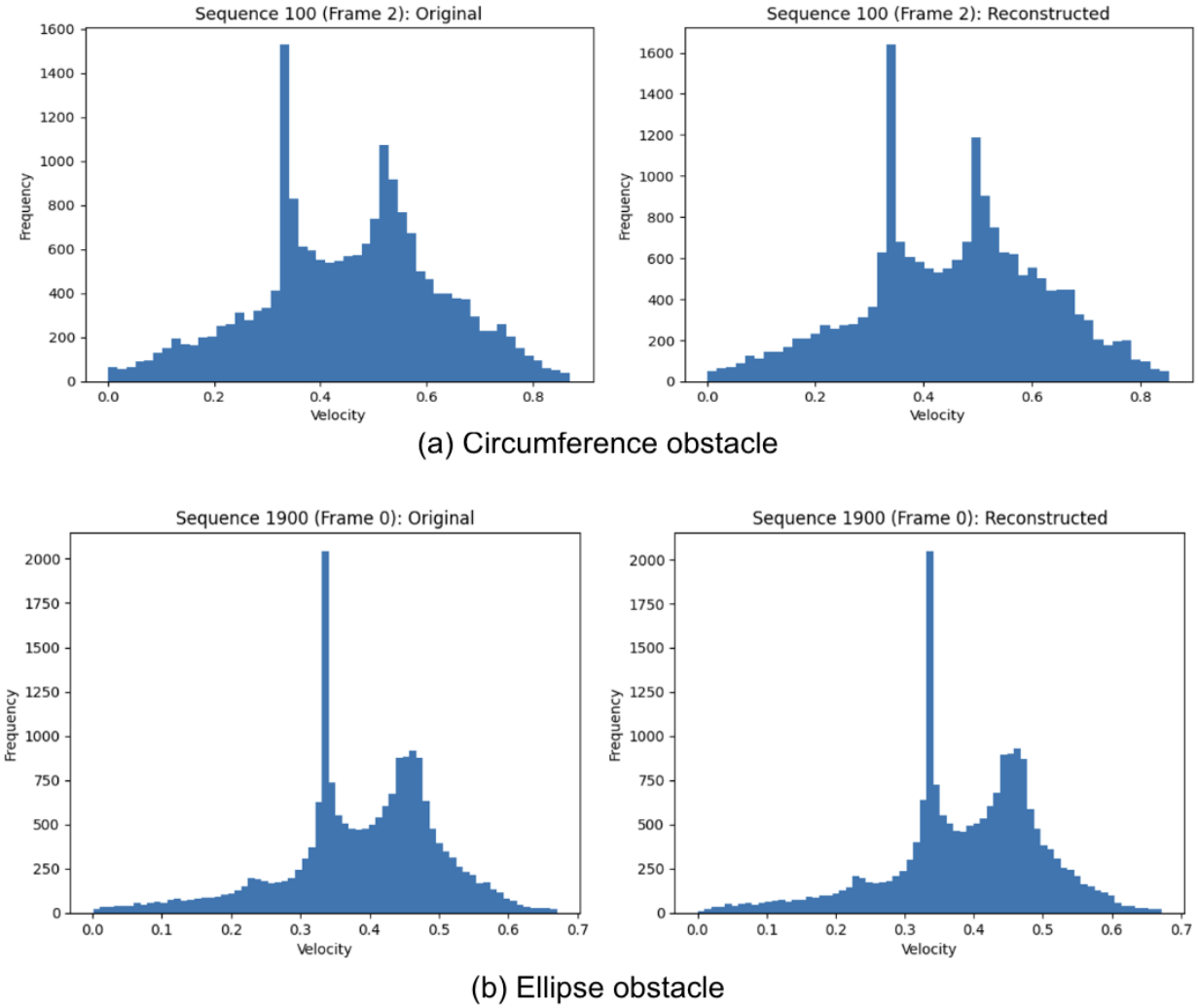


Figure 5.3: Original vs Reconstructed frames velocity histograms

Both evaluations tell us that the Autoencoder successfully reduces the data's dimensionality so that the original data can be reconstructed using that low-resolution representation. This means that the training of this model's component was successful. Although there are some minor errors, the fluid flow structure remains correct, and the approximations of the velocities are very close.

The Autoencoder is a vital component of the model because it guarantees that the model

successfully extracts enough information to represent the original data. This process of reducing the amount of information with a lower representation is important to support the next phase, which is the generation of the next fluid state.

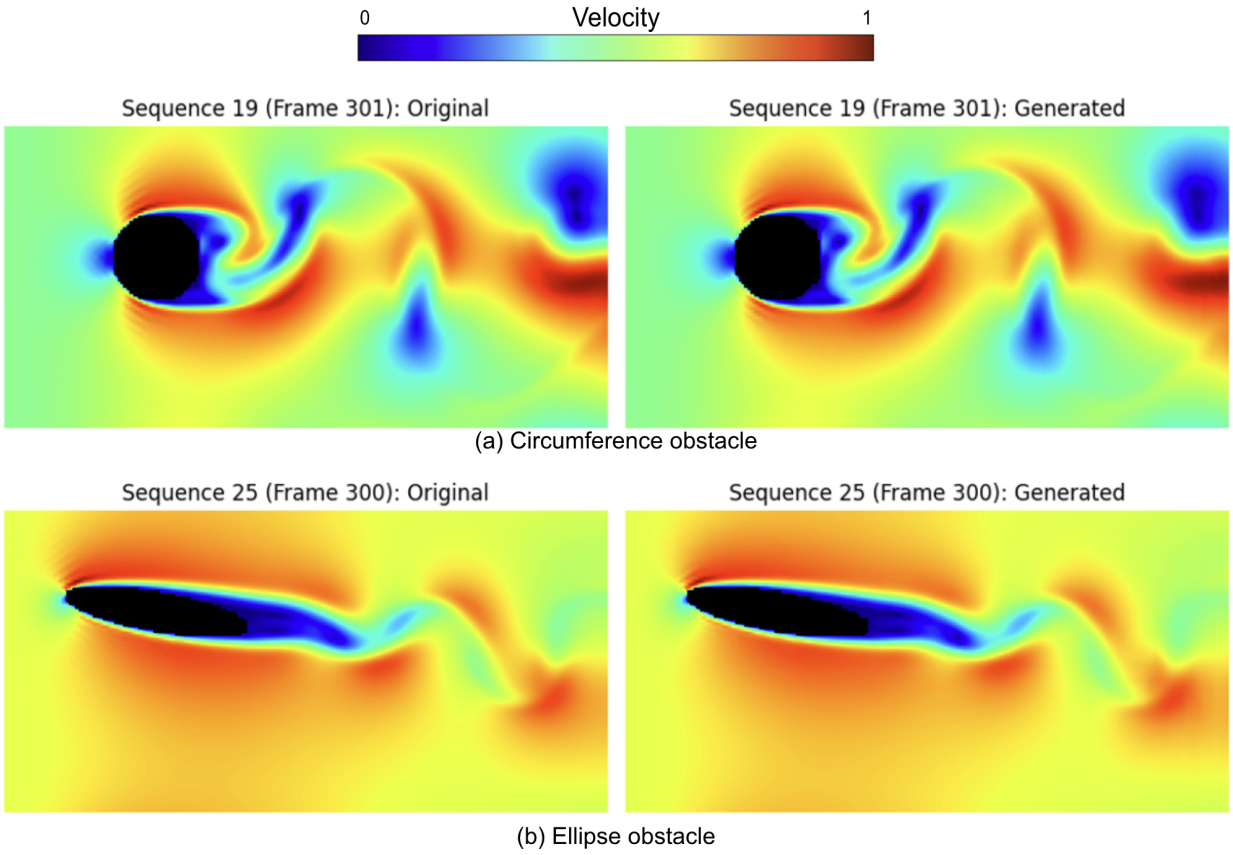
5.3 Generator Results

The Generator's goal is to generate the next fluid flow state using as an input the low-resolution representation created by the Encoder. To evaluate this component, the resulting frame is compared against the expected frame taken from the dataset created by the numerical simulation. For this evaluation we used similar methods than the Autoencoder evaluation.

Figure 5.4 shows a comparison between a generated frame velocity *heatmap* at the right, and the expected frame at the left. The images show a result example for each of the possible obstacle types. Similar to the Autoencoder results, very small differences appear in the *heatmap* colors, however, both images are almost similar.

We created the velocity histogram for each generated frame and compared it to the original frame. Figure 5.5 compares 2 examples of the velocity histograms. Then, we calculated the Jensen-Shannon distance between the velocity distributions of the frames in all the sequences. This results in an average distance between the expected and generated frames of 0.043 with a standard deviation of 0.010. The low average value of the distance metric between both distributions indicates that the original and generated frames are very similar.

These evaluations show that the model can successfully approximate the next state in the fluid flow sequence. The same level of accuracy in both evaluation methods was observed across all the cases in the testing dataset. Although there are some differences between the expected and generated frames, the model can replicate the evolution of the fluid flow structure across the sequence simulation time.



(a) Circumference obstacle

(b) Ellipse obstacle

Figure 5.4: Original vs Generated frames

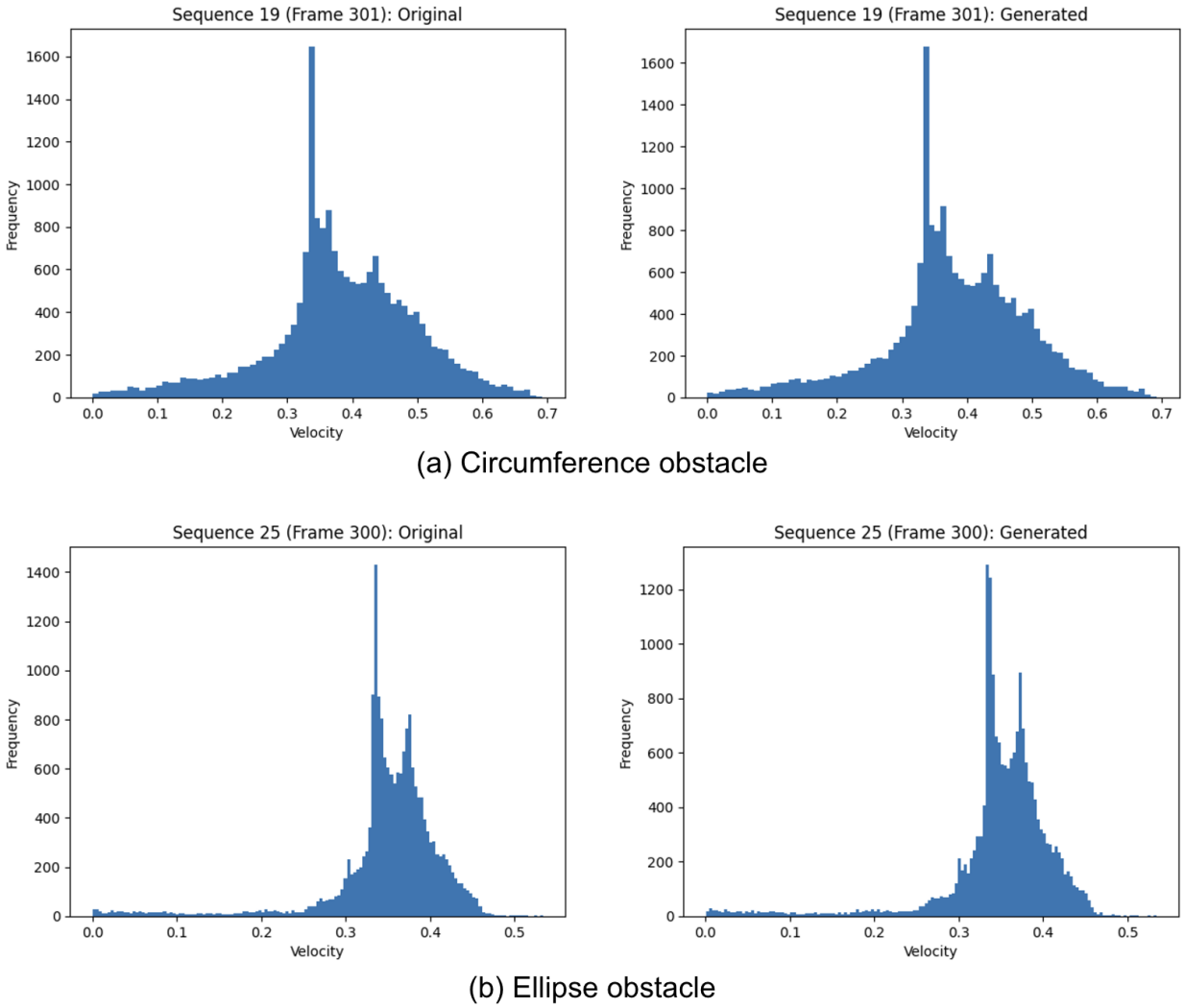


Figure 5.5: Original vs Generated frames velocity histograms

5.4 Model Performance

In this section we explore the results of the two main metrics chosen to evaluate this models performance as explained in Section **??**. These metrics are: the Error measured with MSE, and the execution time of the simulation measured in seconds.

5.4.1 Error Measurements

Figure 5.6 shows the results of the MSE metric. The results are divided into three groups, one for all the shapes in the dataset together, one with only Circumferences obstacles, and another for Ellipses obstacles. The minimum, maximum, and average error values are plotted for each group. It is important to mention that the dataset is balanced, meaning that the amount of examples with each obstacle type is the same, which is essential to ensure fairness in the results.

The following analysis can be done by looking at the MSE plots in Figure 5.6. We can see that the minimum and maximum errors across the entire dataset are both in simulations with an ellipse obstacle. Additionally, the difference between the minimum and maximum error is lower for the circumference than the ellipse obstacle. This could be caused by circumference obstacles presenting less variability in their shapes, with only a change in the radius, while ellipses obstacles have more diversity in their shapes. This variability in the obstacle shapes makes it more challenging for the model to learn how to simulate the ellipse objects. However, because the average errors are similar between the two types of obstacles, we can conclude that no specific obstacle shape is significantly more difficult for the model to simulate.

5.4.2 Execution time

As explained in Section **??**, the goal of this model is to reduce the execution time of the simulation while maintaining a low error to preserve the pattern structure of the fluid flow in the generated sequence. Table 5.2 compares execution time between the simulation and the

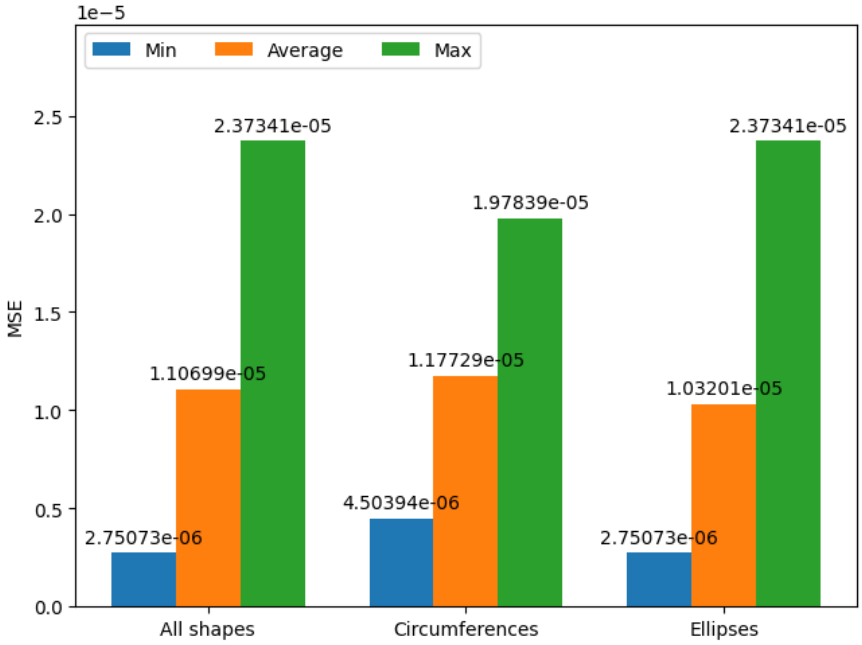


Figure 5.6: Model MSE error metric

DL Model. The simulation took, on average, 191 seconds (3.2 minutes), while the DL model took, on average, only 42 seconds. This represents a 4.5 times improvement in execution speed over the numerical simulation. This result shows that using this DL model improves the simulation's performance, reducing the total execution time while successfully simulating the evolution of the fluid flow.

Table 5.2: CFD simulation vs DL Model execution time

	Average Execution Time
CFD Simulation	191
DL Model	42

Chapter 6

CONCLUSION-TODO

6.1 Conclusion

In this research, we analyze the use of ConvLSTM to implement a deep-learning model for turbulent fluid flow simulations. We propose a neural network architecture to create a Reduced Order Model (ROM) and generate a fluid flow by predicting its spatiotemporal dynamics. This architecture is implemented completely using the ConvLSTM network for all its components. The model was trained and tested using a dataset of fluid flows interacting with circumference and ellipses obstacles of diverse sizes and positions in a two-dimensional space. The data was generated using a Direct Numerical Simulation (DNS). The performance was evaluated using the MSE error metric, achieving results comparable to a DNS simulation. When it was evaluated using the training and testing data, the model achieved similar error metrics, meaning that it was able to generalize well to unknown data. It is also worth mentioning that the error was balanced across the two types of obstacles. Additionally, the simulation execution time with the DL model was four times faster than the DNS. These results demonstrate that a neural network model proposed, combining an autoencoder and a generator implemented with a ConvLSTM, is suitable for predicting turbulent fluid flows and can accelerate Computational Fluid Dynamics simulations.

6.2 Contributions

6.2.1 Data preparation and training methods

The dataset we created for this research builds upon others used in CFD research by including a greater diversity of obstacles with different shapes and positions in the simulated space. Additionally, it takes into account a common and important shape in fluid dynamics, which

is the airfoil shape, by using a simple approximation of it.

The methods defined to preprocess the raw data of simulated fluid flows are essential to prepare the dataset to be used for the model's training process. This greatly influences the final model performance. The methods we describe in this study can be easily extended and adapted to other applied cases, like longer simulated sequences or different window $\mathcal{W}$ sizes. Additionally, this could be applied to any neural network training problem that relies on a multidimensional time-series dataset.

6.2.2 Model arquitecture

This research demonstrates the effectiveness of the ConvLSTM type of neural network in CFD by presenting a model architecture that is totally implemented with ConvLSTM and can achieve similar results to a DNS of fluid dynamics. This architecture was successful in other domains involving multivariate time series, like weather prediction, and we show how a solution in CFD can completely rely on this neural network.

The DL model presented further demonstrates how a data-driven approach with an end-to-end model can be used to generate fluid flow simulations in CFD applications. Additionally, the model architecture shows how useful is to include the creation of ROMs with an Autoencoder as a component of the DL model. The proposed model could represent the input data with half of the initial dimension and still reconstruct the original data. By including the dimensionality reduction as part of a unified model, the entire CFD process is simplified because it reduces the steps in the process to only one. Additionally, this research showed how much faster is to execute the neural network model compared with the DNS. By obtaining an improvement in the execution time of about 4 times faster, the research proves that this model can accelerate scientific computing simulations, including CFD.

6.3 Limitations

Some limitations exist in this work that are worth mentioning:

- Available hardware resources: the available hardware had 2 GPUs with 16GB of RAM each. This limits the size of the model and the dataset that could be loaded in memory. Because the model and the dataset have to share the GPU memory simultaneously, it limits the outcome of the model training. While training, the process used about 95% of the available memory on each GPU.

- Dataset: The dataset only has a single obstacle of two possible types with a simple shape. Furthermore, the shapes are static during the simulation. Additionally, all the fluid flows considered have the same Reynolds number.

- Comparing the results to other solutions: because the dataset was different than equivalent methods, there was no way to compare the results without implementing all the other solutions again.

6.4 Future work

This work opens plenty of opportunities to extend this research. These opportunities are related to scope limitations that exist in this work and possible improvements to this research. Future work could focus on extending the dataset to consider a wider range of shape types and complexity. Additionally, moving obstacles could be considered for the dataset. Including fluid flow sequences with multiple obstacles or complex obstacles that use simple ones as building blocks could also be interesting to test. Additional work could consider more complex turbulent flows with different Reynolds numbers and evaluate how well the model results generalize to those cases. In future research, it might be worthwhile to investigate how the window size m affects the accuracy and performance of the model; this could involve determining the point of diminishing returns when increasing the size of the window while giving the model a small amount of input information.

Computational Fluid Dynamics is already a mature field. It utilizes principles from fluid dynamics that are at least 300 years old and relies on computer simulation algorithms and

methods that are about 60 years old. However, recent research like this one shows that progress can still be made with the introduction of Deep Learning to the field.

BIBLIOGRAPHY

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008, 2017.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[4] OpenAI. Gpt-4 technical report. `https://openai.com/research/gpt-4`, 2023. Accessed 2024.

[5] Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*, 2019.

[6] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2021.

[7] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[8] Rishi Bommasani et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[9] Philipp Mattern, Felix Sattler, Hartmut Schmeck, and Bastian Pfitzner. Membership inference attacks against language models via neighbourhood comparison. *arXiv preprint arXiv:2306.04554*, 2023.

[10] Milad Nasr, Nicholas Carlini, Florian Tramer, and Reza Shokri. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023.

[11] Amy Deschenes and Meg McMahon. A survey on student use of generative ai chatbots for academic research. *Evidence Based Library and Information Practice*, 19(2):2–22, 2024.

[12] Mohammad Hosseini, Serge Horbach, Tanja Van den Broek, Boudewijn De Bruin, and Sietse Wieringa. An exploratory survey about using chatgpt in education, healthcare, and research. *medRxiv*, 2023.

[13] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

[14] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kulkarni, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[16] Apple Inc. Apple neural engine (ane) performance across devices, 2024. Accessed May 13, 2025. Reports ANE performance ranging from 0.6 TOPS (A11) to 38 TOPS (M4).

[17] Georgi Gerganov. LLaMa.cpp. https://github.com/ggerganov/llama.cpp, 2023. C++ framework to run open source LLMs on local hardware.

[18] Ollama. https://ollama.com, 2023. Desktop application to download and run a specific LLM.

[19] Tinygrad. Apple neural engine reverse engineered for c++. https://github.com/geohot/tinygrad/tree/master/accel/ane, 2023. GitHub repository documenting reverse engineering of Apple's ANE.

[20] LLamaFile. https://github.com/Mozilla-Ocho/llamafile, 2023. Command-line app to package and run an LLM.

[21] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Transactions of the Association for Computational Linguistics*, 9:447–466, 2021.

[22] Stackademic Blog. Mastering retrieval-augmented generation (rag) architecture. https://blog.stackademic.com/mastering-retrieval-augmented-generation-rag-architecture-unleash-the-power-of-larg 2023. Accessed May 2025.

[23] LangChain. Rag tutorial. https://python.langchain.com/docs/tutorials/rag/, 2023. Accessed May 2025.

[24] ggml org. llama.cpp. https://github.com/ggml-org/llama.cpp, 2023. GitHub repository. Accessed May 2025.

[25] Ollama. Ollama. https://ollama.com/, 2023. Accessed May 2025.

[26] Mozilla-Ocho. Llamafile. https://github.com/Mozilla-Ocho/llamafile, 2023. Accessed May 2025.