ENS 491-492 – Graduation Project

Final Report

Project Title:

Application of autoencoder neural networks for CFD problems

Group Members:

Edin Guso 23435

Güneş Başak Özgün 23521

Supervisor:

Serhat Yeşilyurt

Date: 16/05/2020

Contents

EXECUTIVE SUMMARY

Drag and lift forces are important quantities in systems where fluid dynamics are present. Drag force can cause energy losses in systems such as airplanes and trucks. Our goal was to reduce the energy losses caused by drag force. In order to achieve this goal, we considered several approaches.

We first designed a toy problem for which we could extract simulation data. The toy problem consisted of a cylinder inside a rectangular tube with flow passing over it. We applied two different approaches to reduce the drag force acting on the cylinders. First one was to have two jets located on the cylinder which would be used to control the flow passing over the cylinder. We used COMSOL to generate data for this case. But the data generation step in COMSOL takes a long time. Therefore, we decided to try another approach. The second approach made use of magnus effect by creating a rotation effect on the cylinder instead of utilizing the jets. We used fluid dynamics equations in MATLAB to generate data for this case.

In order to minimize the drag force acting on the cylinder, in both approaches, we developed a machine learning model which would predict lift coefficient and drag coefficient values from the 2D stress field images from the surface of the cylinder. Our model consisted of autoencoder layers followed by a shallow network. The autoencoders were used for local data compression and the shallow network was used for regression.

We managed to accurately predict drag and lift coefficients and reconstruct the encoded images. This model can be used to predict the drag and lift coefficients acting on the system and minimize the drag force acting on the system by optimizing the control parameter.

PROBLEM STATEMENT

The problem we are tackling is reducing the drag force on objects that are subject to a flow. Drag force is a significant factor when it comes to energy losses. Systems that are subject to a flow such as aerodynamic systems must be designed to reduce the effects of drag. The magnitude of the drag force is changed by controlling the parameters associated with the object. But this is a computationally expensive problem due to the presence of many factors.

We are aiming to reducing the cost of computation and effect of drag force at the same time by using machine learning. Machine learning has been gaining attention in computational fluid dynamics field due returning fast prediction results. Simple demonstrations are done by using powerful learning algorithms such as deep reinforcement learning and neural networks to show machine learning’s potential in the field.

This problem is not new at all and there are many different solutions in the literature. Some solutions are boundary layer control, passive control and active flow control. Active flow control is an effective method to reduce drag. However active flow control is a high dimensional and computationally costly method, only control methods for simple inputs (e.g. constant control input) were implemented. To fully appreciate the active flow control, machine learning (ML) approaches are starting to be used. Convolutional Neural Networks (CNN), Deep Lagrangian Networks (DeLaN), Reinforcement Learning (RL), Physics Informed Neural Networks (PINNS) are some examples for ML approaches.

CNNs are mainly used in feature extraction. Inputs signed distance functions (SDF) and CNNs have the advantage encode patterns to the network. The prediction is based on SDF to reduce errors. Boundaries are considered such as velocity in the inside and at the boundaries are zero. Neural network considers the geometric boundaries. [2]

DeLaN approach considers physics of the mechanical system as model prior so that the predictions satisfy physical requirements as well. [3]

RL is about actions and their consequences. The learner will not be told what to do. Instead the learner will have to discover the best actions by trying different actions. Rewards are given based on the learner's actions and the learner will try to maximize the total reward it receives [7]. In a recent research, deep reinforcement learning was used in an active flow control problem. This approach was demonstrated on a cylinder with two jets located on the both side which is placed inside a constant flow in 2D simulation. The goal is to control the jets to reduce the drag [7]. However, RL requires a significant amount of training time which may create time issues on a big scaled problem. One of the approaches for avoiding long training times and the immense need for data in the other ML approaches is to use model-based learners. This approach is more sample efficient and thus it requires less data for reaching near-optimal results [7]. Although model-based learners provide solutions to many problems, in systems that have physical limitations and less data (relative to huge databases), solutions with traditional models do not provide a desired result.

Recently a solution that incorporates physical boundaries to the neural network model as loss functions was introduced [2]. With this technique, physical limitations are imposed, and neural networks can produce more accurate predictions as well as require less data (relative to what traditional machine learning techniques require).

Objectives/Tasks

Define the environment (flow) and the object in COMSOL and MATLAB: This will allow us to model the environment for data generation.

Generate different stress fields by changing parameters: We will find an efficient image size and sample points that will make our model give accurate and relatively fast results.

Build a network with 3 autoencoders and 1 shallow network: Autoencoders will encode and decode the image and shallow network will predict the drag and lift forces from the reconstructed image.

Training the network with simulated data: The model will be trained using the simulated data in MATLAB.

Compare the results with actual field results: Accuracy will be tested.

Improve the network: Accuracy of the network will be increased by using PCA.

Update to network to minimize total drag force: Given the stress field, network will output the parameter values that will result in smallest drag force.

Realistic Constraints

Autoencoder bottleneck and data were common constraints we encountered in two approaches. Both issues are explained with examples below.

Autoencoder Bottleneck

We started with a single autoencoder that would take 44x28 images and encode them to a scalar value. From that scalar value, it would decode it back to 44x28 image. But this was a poor decision since a scalar value was not enough to represent the complexity of a stress field. With different U_{jet} values, the model reconstructed almost identical stress fields as shown in Figure 1. We solved this issue by encoding the 44x28 image into a 16x1 vector which will allow a larger size of information to be stored. We followed the same strategy in the second approach to avoid bottlenecks.

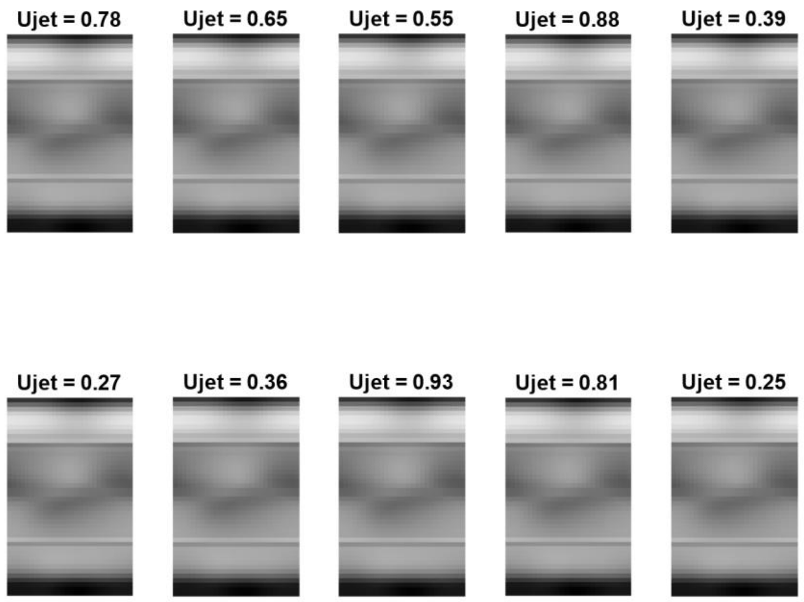

Figure 1: Reconstructed stress fields with their U_jet values

Data Set Size

Having enough data to train our model is crucial. In the first approach we obtained 100 data samples (U_{jet} values) from COMSOL simulation with 7 GB size. Then we divided them into train and test sets for building the model. The model is built with many parameters and small data set; thus, an overfitting issue has occurred. Also having a small data set increased the probability of having consecutive U_{jet} values being selected into the train set. This resulted model to be biased towards these U_{jet} values and increased the error rate. We attempted solving this problem by re-shuffling the dataset and re-training the network 10 times and taking the average error values.

In the second approach, we used MATLAB to produce sufficient amount of data which was much faster than data generation in COMSOL. We were able to generate hundreds of data in a few minutes and the space required was in KBs range.

METHODOLOGY

In our project we considered two approaches for drag reduction. The first one was directly affecting the flow using two jets on the cylinder. And the second one was using the magnus effect.

We divided our project into two phases; data gathering phase and model development phase. First phase is crucial for the model development phase to work. We started obtaining data by designing a simulation environment in COMSOL for the approach where we used the jets. By changing the parameters of the simulation such as flow rate of the jets, we created different cases. For the approach where we utilized magnus effect to reduce the drag force, we used fluid dynamics equations to generate the data in MATLAB. Following the data generation, in both approaches, we processed these data in MATLAB to transform them into a format that we can feed into our machine learning model.

In order to analyze the performance of our solution, we calculated drag coefficients related with each U_{jet} (first approach) and g (second approach) value and compared them with the predicted drag coefficient values coming from the neural network. We will explain each step, in detail in the following parts.

Data Gathering Phase

Approach 1: Jets

Simulation Environment

The simulation environment is a slightly modified version of the one used in the paper [5] since we will be comparing our results to theirs. All quantities that are defined in the simulation are non-dimensionalized. We began by defining the geometry of the environment. The box that will house the cylinder has non-dimensional height of 3 and length of 10. The cylinder with non-dimensional diameter of 1 was placed at the center with an 0.1 offset in y direction. The walls of the cylinder and the upper and lower walls of the box have no slip boundary condition. The jets were placed into the inlets defined perpendicular to the direction of the flow (at $90°$ and $270°$). The inlets are $10°$ wide. Figure 2 shows the geometry we are using.

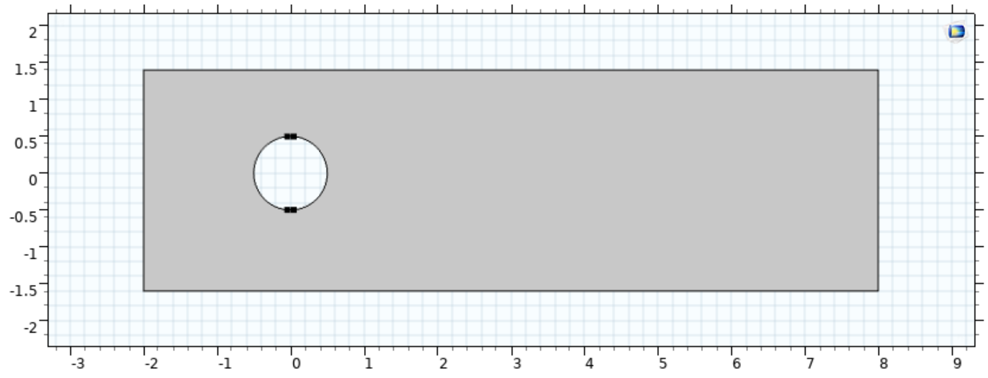

Figure 2: Simulation Geometry

Creating Simulations

We started by defining parameters that the simulation will be run on. We kept almost all of them (except three parameters that we will explain) fixed for different cases. We will begin with the fixed parameters. The Reynolds number was set to 200 and the velocity field was defined as parabolic formula $U(y) = 6y(1-y)$. The jet flow rates are defined as U_{jet1} (placed at 90°) and U_{jet2} (placed at 270°). U_{jet1} and U_{jet2} have the following formulas:

$$U_{jet1}(t) = \max\left(floor(((t - wait\_time\ ))/15) * input\_step\_size\ ),0\right)$$

$$U_{jet2}(t) = \ -U_{jet1}(t)$$

U_{jet2} is defined as the same magnitude and opposite direction of U_{jet1} in order to avoid adding extra mass to the system.

Flow rates are step functions with step size given with the parameter $input\_step\_size$. We began each simulation with a constant flow coming in and jets are idle so that a stable flow distribution is ensured. The duration of the constant flow is defined with $wait\_time$. The simulation will end at $stop\_time$.

We extracted the forces in X and Y directions and the time stamps in one data set. Data points are saved with 0.1 time intervals.

Extracting the Useful Information

Drag and lift force data are then moved to MATLAB to calculate the drag and lift coefficients at given flow rates. Our goal was getting average drag coefficient and lift coefficient during a step input interval by integrating drag and lift forces over the interval and dividing it by the size of the integrated area. We wanted to find these coefficients so that we could use them to connect the flow rates with drag and lift forces later when we will be training our model. The drag and lift coefficients are calculated as below:

$$C_D = \frac{\int F_D\, dA}{A} \qquad\qquad C_L = \frac{\int F_L\, dA}{A}$$

Approach 2: Magnus Effect

In this approach, the data gathering phase was much simpler and quicker compared to the previous one. The reason is that, instead of using COMSOL for data generation, we used fluid dynamics equations and computed our data samples in MATLAB. This meant that we had no data limitations for our machine learning model, which was one of the more important setbacks in the previous approach.

The fluid dynamic equations used to generate the data can be seen below:

$$\sigma_x(\theta, t, g) = cos(\theta).*\, p(\theta, t, g)$$

$$\sigma_y(\theta, t, g) = sin(\theta).*\, p(\theta, t, g)$$

where $.*$ is the elementwise multiplication operator and

$$p(\theta, t, g) = \frac{1}{2}\rho\big(U^2 - \nu_\theta(\theta, t, g).* \nu_\theta(\theta, t, g)\big)$$

where

$$\nu_\theta(\theta, t, g) = -2URsin(\theta) + (sin(\omega\theta)\pi + 0.15)\frac{0.5}{\pi} + gsin\left(\frac{\theta}{3}\right)$$

The code used for data generation can be found in Appendix A.

Approach 3: Magnus Effect with Deep Learning

We used the same data generation approach as in Approach 2.

Model Development Phase

Approach 1: Jets

Generating the Dataset

The first step of developing an accurate model starts from generating a good dataset. In order to do that we had to generate a dataset that captures the relationship between U_{jet} values and stress fields meaningfully.

After every U_{jet} increment, there is some time until the system enters steady oscillation state. In order to catch the most accurate representation, we decided to use the time period between the last two peaks. This resulted in each sample having (number of probes on the cylinder surface * number of time steps between two peaks) image size. As the number of time steps between two peaks is varying for each sample (between 28 and 31), we decided to crop each image's time samples to 28. Also, we had 44 probes on the cylinder surface. So, each image (data sample) had 44x28 size.

As we had simulations for 100 different U_{jet} values, our dataset consisted of 100 images of size 44x28x1 (height*width*channels). An example data sample can be seen in Figure 3.

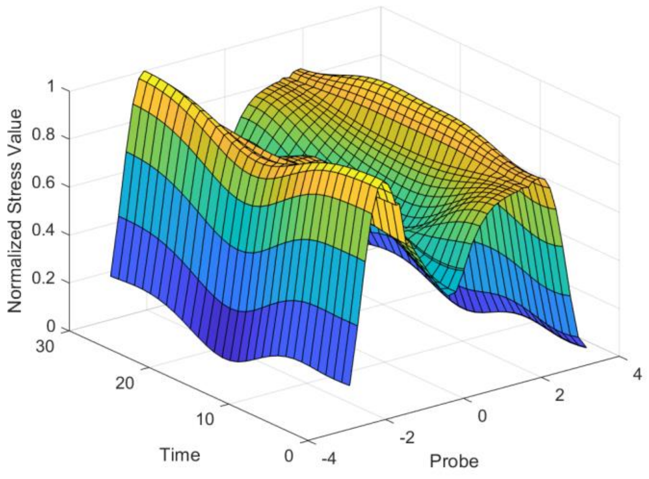

Figure 3: Example Data Sample

Shuffling and Splitting the Dataset

As our dataset is very small, we had to split it carefully. First, we randomly shuffled all the data samples. Then, we split the dataset as 65% training set, 25% validation set and 10% test set.

Network Architecture

We needed to create a network architecture which would allow us to first compress the data with minimal information loss to compensate for the lack of data, and then use this compressed data to accurately predict C_D values for the given stress field. To achieve this, we created a network with 3 stacked autoencoders followed by 3 fully connected layers. Autoencoders can achieve the desired local compression of the data [2]. The visual representation of the network architecture can be seen in Figure 4. The details of each layer will be explained in the following sections.

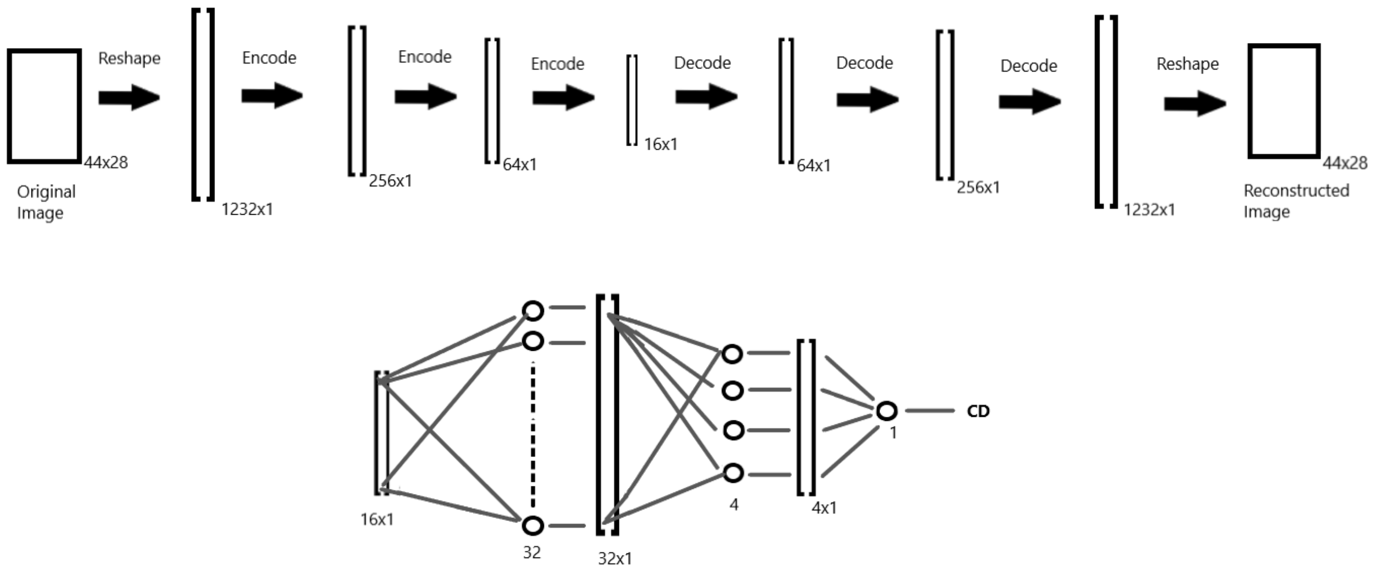

Figure 4: Network Architecture

Autoencoders

Each autoencoder takes a vector as an input and encodes it to a smaller size vector. However, since our input data is a 2D image, there is first a reshaping step that converts 44x28 images into 1232x1 vectors.

It is important not to reduce the dimension of the input by a large factor in a single autoencoder layer to avoid large information loss.

First autoencoder compresses the data into a 256x1 vector. Second autoencoder takes the 256x1 vector and compresses it into a 64x1 vector. The third autoencoder takes the 64x1 vector and compresses it to a 16x1 vector. Each autoencoder roughly decreases the size by a factor of 4.

For each autoencoder we used the same parameters. These were:

- *L2WeightRegularization* = 0.004
- *SparsityRegularization* = 2
- *SparsityProportion* = 0.25
- *MaxEpochs* = 256

Fully Connected Layers

Fully connected layers are used to predict C_D values from the given compressed 16x1 vectors. We created three fully connected layers each with 32, 4 and 1 hidden nodes. We found these values with trial and error. Best prediction accuracies were achieved with these values.

We also optimized many of the neural network parameters during training. These parameters are listed below:

- *MiniBatchSize* = 1
- *MaxEpochs* = 20
- *InitialLearningRate* = 0.001
- *LearnRateSchedule* = piecewise
- *LearnRateDropFactor* = 10
- *Shuffle* = every-epoch
- *ValidationFrequency* = 1

Approach 2: Magnus Effect

Generating the Dataset

We followed a similar approach for data generation as in the previous approach. The algorithm we used for data generation outputted the stress values on the cylinder caused by drag and lift forces. Using these, we created 2D stress field images just like we did before (Figure 3) and computed the C_D and C_L values for corresponding to each image. The images, this time, were of the size 50x25. And we created 500 of these samples corresponding to different g values. Then, we used these samples as inputs for our machine learning model and the C_D and C_L values as targets.

Shuffling and Splitting the Dataset

Again, we randomly shuffled all our data samples. Then, we divided our dataset as 60% training, 30% validation and 10% test set.

Network Architecture

We followed a similar architecture to the one given in Figure 4. We will discuss the main differences below.

Autoencoders

Both approaches take the stress fields as input and have almost identical sizes (44x28 vs 50x25). This resulted in both types of input having similar dimensions (1232 vs 1250). Therefore, we used the same autoencoder structure as in the previous approach.

Fully Connected Layers

The main difference was in the fully connected layers. Even though the input dimensionality is similar, the complexity of the problem was different. Therefore, we had to adjust the complexity of the shallow network.

We again had to predict C_D and C_L values from the encoded 16x1 vector. But due to the increased complexity of the problem, we had to increase the number of neurons. We used three fully connected layers and they had 64, 16 and 1 nodes, respectively.

We also changed the neural network parameters as the dataset changed. Since the number of samples increased a lot in this approach, we increased the mini batch size and decreased the max epochs. The new parameters can be seen below:

- *MiniBatchSize* = 3
- *MaxEpochs* = 15
- *InitialLearningRate* = 0.001
- *LearnRateSchedule* = piecewise
- *LearnRateDropFactor* = 10
- *Shuffle* = every-epoch
- *ValidationFrequency* = 1

Approach 3: Magnus Effect with Deep Learning

Generating the Dataset

We used the same data generation algorithm as in Approach 2. The only difference was that we slightly increased the resolution of the images. Images used in this approach are 64x32.

Shuffling and Splitting the Dataset

Again, we randomly shuffled all our data samples. Then, we divided our dataset as 60% training, 30% validation and 10% test set.

Network Architecture

We developed our autoencoders using a CNN architecture instead of the built-in autoencoders provided by MATLAB. We will discuss the architecture in more detail below.

Autoencoders

We have designed our encoder by constructing three 2D convolutional layers.

First, our input goes through a 2-D convolutional layer consisting of 16 filters with size [3 3], stride of [1 1], and the size of the padding is calculated at training time to keep the output image size same as the input image size. Then, the output goes through a ReLU activation function. Before entering the next convolutional layer, we apply a 2-D max-pooling layer which halves the image width and height.

Then, we apply another a 2-D convolutional layer consisting of 8 filters with size [3 3], stride of [1 1], and the size of the padding is calculated at training time to keep the output image size same as the input image size. Then, the output goes through a ReLU activation function. Before entering the next convolutional layer, we apply a 2-D max-pooling layer which halves the image width and height.

Finally, we repeat the same three layers (2-D convolutional layer, ReLU activation function, and 2-D max-pooling layer).

We have designed our decoder using the createUpsampleTransposeConvLayer helper function. The implementation can be found under MATLAB documentation: Prepare Datastore for Image-to-Image Regression.

First, we apply a createUpsampleTransposeConvLayer with upsampling factor 2 using 8 filters. Then, the output goes through a ReLU activation function.

Following this, the same two layers are applied again.

Then, we apply a createUpsampleTransposeConvLayer with upsampling factor 2 using 16 filters. Then, the output goes through a ReLU activation function.

Finally, we apply another a 2-D convolutional layer consisting of 1 filter with size [3 3], stride of [1 1], and the size of the padding is calculated at training time to keep the output image size same as the input image size. Then, the output goes through a clipped ReLU activation function. Lastly, a regression layer is attached which gives the output image.

RESULTS & DISCUSSION

Approach 1: Jets

Performance Measure

In order to measure the performance of our system, we created 2 different metrics. First one being for the reconstruction accuracy and the second one being for the C_D prediction accuracy.

For reconstruction accuracy, we applied the following formula:

$$e = \frac{\sum_{n=1}^{num} \sum_{i=1}^{x} \sum_{j=1}^{y} abs\left(A_{n_{ij}} - P_{n_{ij}}\right)}{num\_samples \times x\_dimension \times y\_dimension \times max\left(A_{n_{ij}}\right)}$$

where $A_{n_{ij}}$ is the stress at probe i and time j of the actual stress field n, and $P_{n_{ij}}$ is the stress at probe i and time j of the reconstructed stress field n.

For C_D prediction accuracy, we applied the following formula:

$$e = \frac{\sum_{n=1}^{num} abs(a_n - p_n)}{num\_samples \times max(a_n)}$$

where a_n is the actual C_D for stress field n, and p_n is the predicted C_D for stress field n.

Results

After the first autoencoder, we reconstructed stress fields corresponding to C_D values in the test set. The autoencoder was able to reconstruct stress fields with a relative mean error of 0.74376%. Reconstructed stress fields with their C_D values are shown in Figure 6 and actual stress fields with their C_D values are shown in Figure 5.

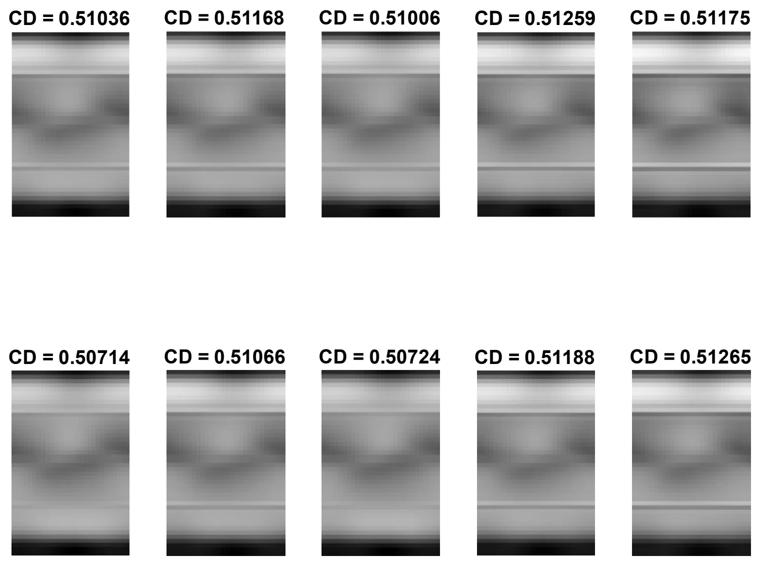

Figure 5: Actual Stress Fields

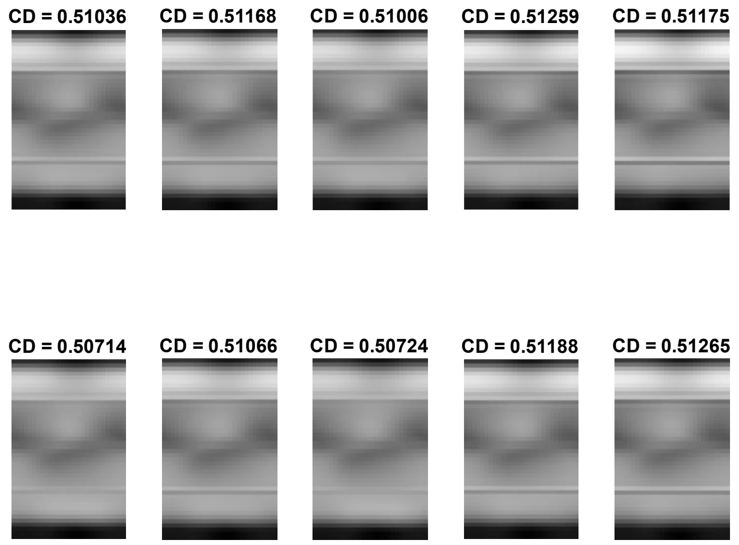

Figure 6: Reconstructed Stress Field After the first Layer

Then we did the same plotting after the second autoencoder, we reconstructed stress fields corresponding to C_D values in the test set. The autoencoder was able to reconstruct stress fields with a relative mean error of 0.83783%. Reconstructed stress fields with their C_D values are shown in Figure 7 and actual stress fields with their C_D values are shown in Figure 5.

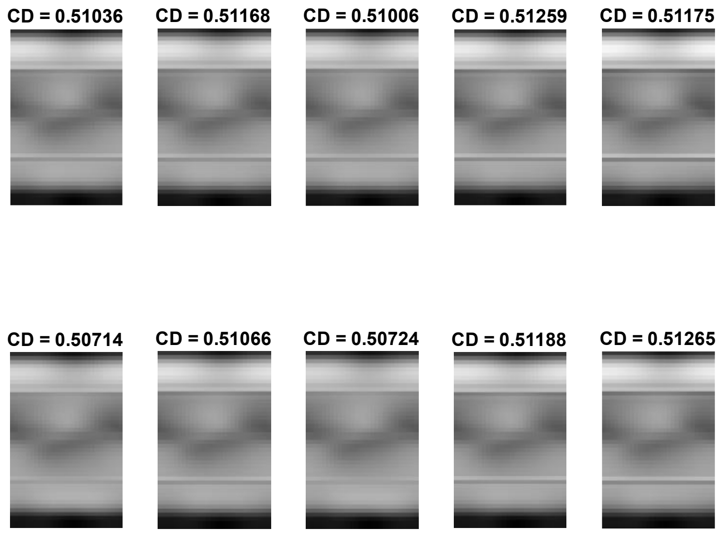

Figure 7: Reconstructed Stress Field After the second Layer

We reconstructed stress fields corresponding to C_D values in the test set after the third autoencoder. The autoencoder was able to reconstruct stress fields with a relative mean error of 1.0685%. Reconstructed stress fields with their C_D values are shown in Figure 8 and actual stress fields with their C_D values are shown in Figure 5.

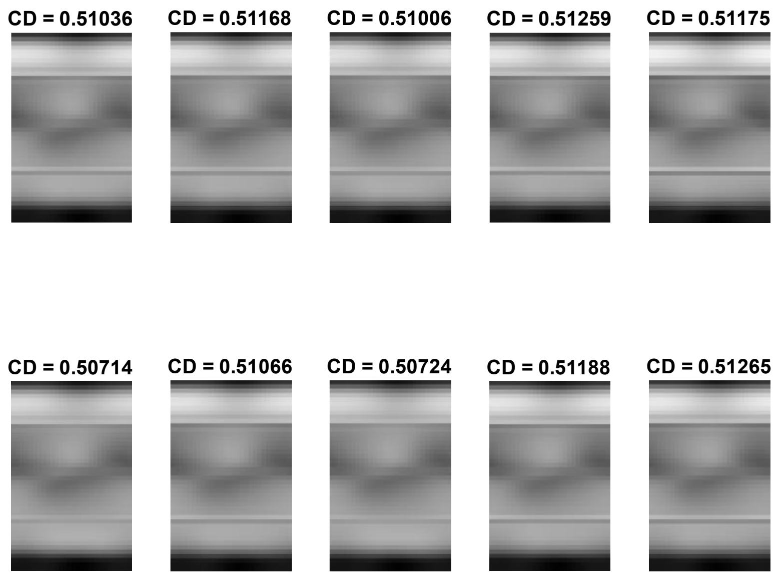

Figure 8: Reconstructed Stress Field After the third Layer

After the shallow network, C_D is predicted with relative mean error of 1.8614%: Then we plotted the actual and predicted C_D values in the same plot as shown in Figure 9.

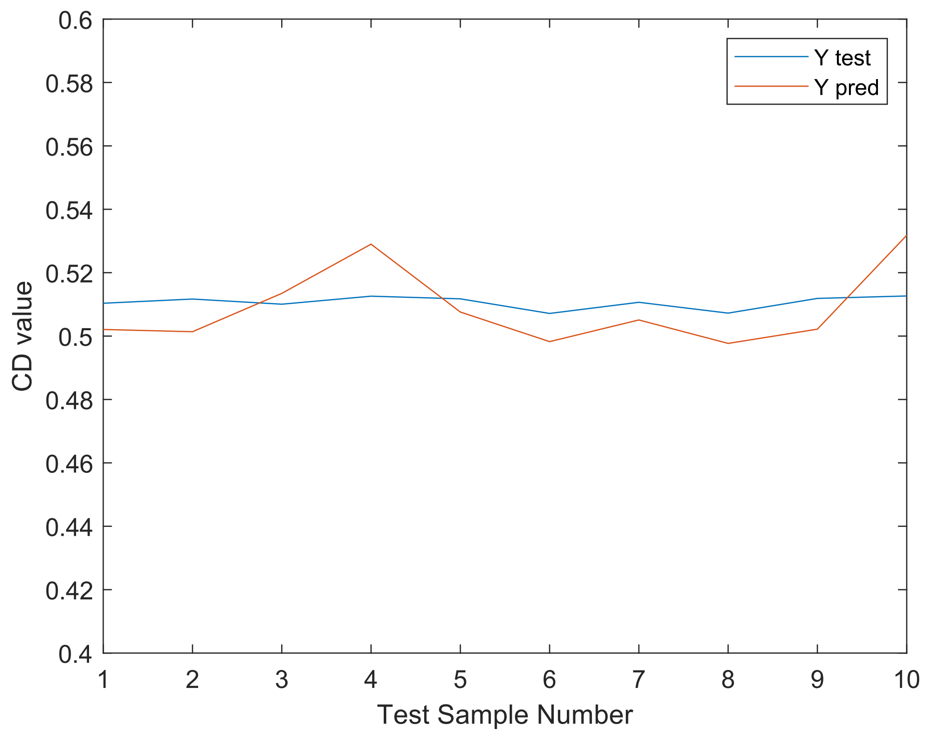

Figure 9: Real CD values vs Predicted CD values

Approach 2: Magnus Effect

Performance Measure

In order to measure the performance of our system, we created 3 different metrics. First one being for the reconstruction accuracy and the second one being for the C_D prediction accuracy.

For reconstruction accuracy, we applied the following formula:

$$e = \sqrt{\frac{\sum_{n=1}^{num} \sum_{i=1}^{x} \sum_{j=1}^{y} \left(A_{n_{ij}} - P_{n_{ij}}\right)^2}{num\_samples \times x\_dimension \times y\_dimension}}$$

where $A_{n_{ij}}$ is the stress at probe i and time j of the actual stress field n, and $P_{n_{ij}}$ is the stress at probe i and time j of the reconstructed stress field n.

For C_D prediction accuracy, we applied the following formula:

$$e = \sqrt{\frac{\sum_{n=1}^{num} (a_n - p_n)^2}{num\_samples \times max(a_n)}}$$

where a_n is the actual C_D for stress field n, and p_n is the predicted C_D for stress field n.

For C_L prediction accuracy we used the same metric as we did for C_D.

Results

After the first autoencoder, we reconstructed 10 (out of 50) of the stress fields corresponding to C_D and C_L values in the test set. The autoencoder was able to reconstruct stress fields with a root mean squared error of 2.6186% and 2.5807% for C_D and C_L, respectively. Reconstructed stress fields with their C_D and C_L values are shown in Figure 12 and Figure 13 and actual stress fields with their C_D and C_L values are shown in Figure 10 and Figure 11.

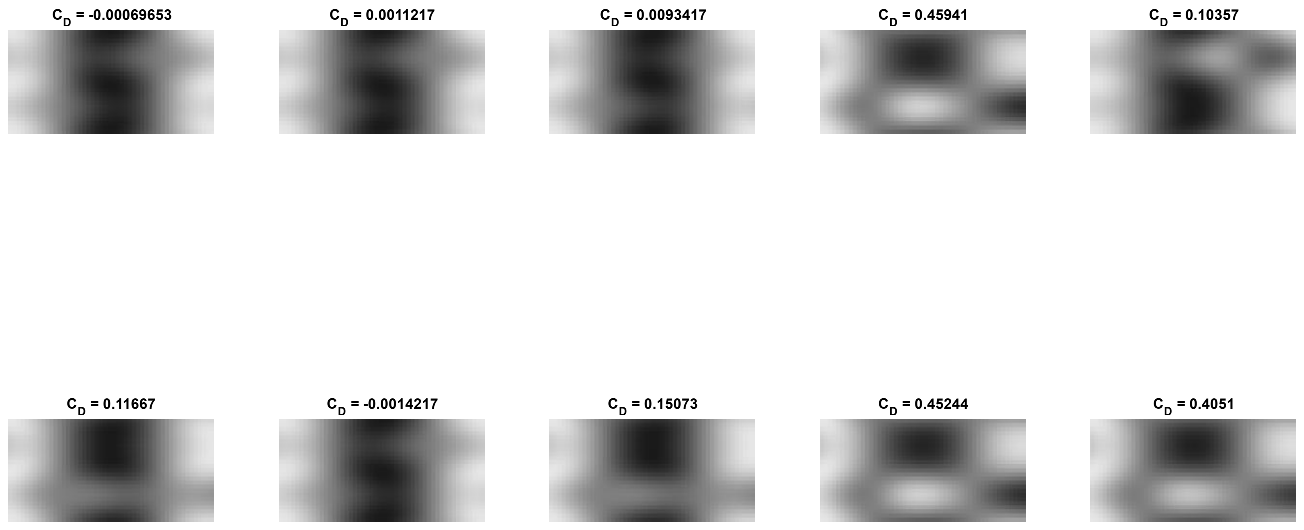

Figure 10: Actual Drag Stress Fields

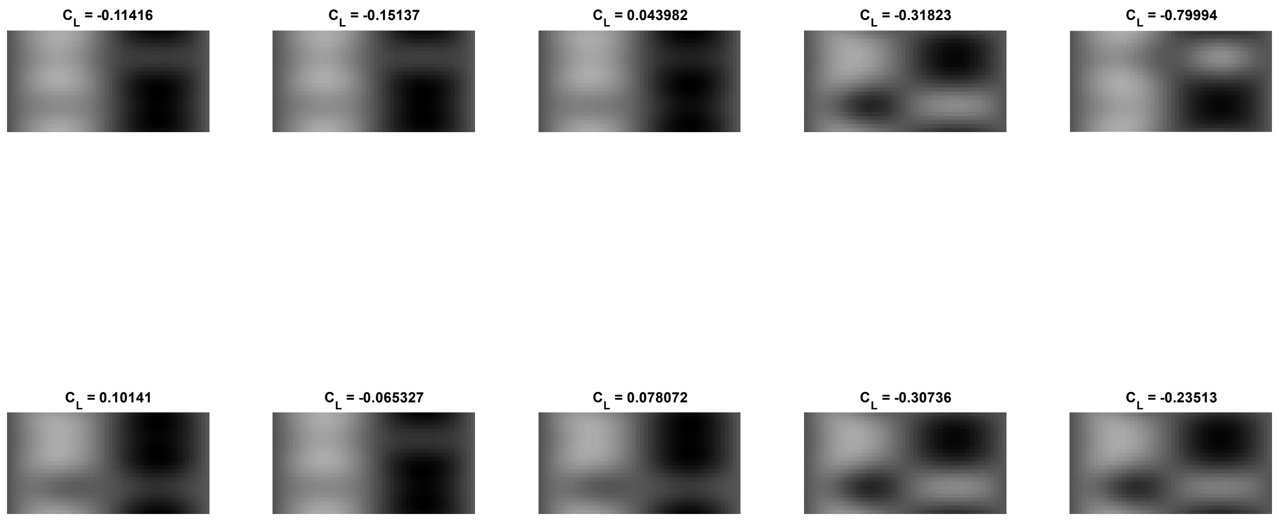

Figure 11: Actual Lift Stress Fields

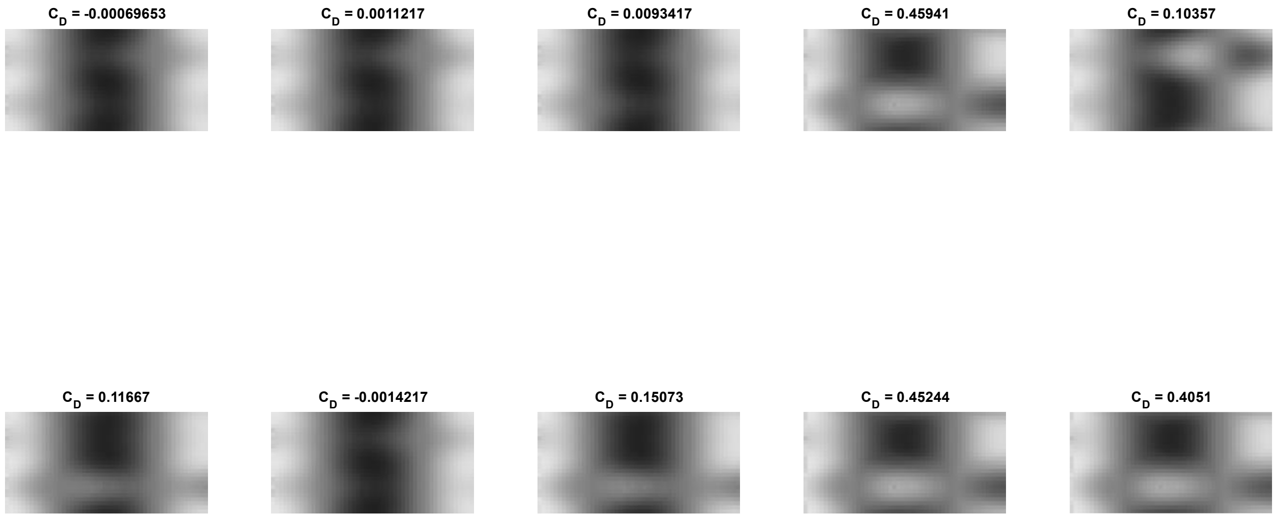

Figure 12: Reconstructed Drag Stress Field After the first Layer

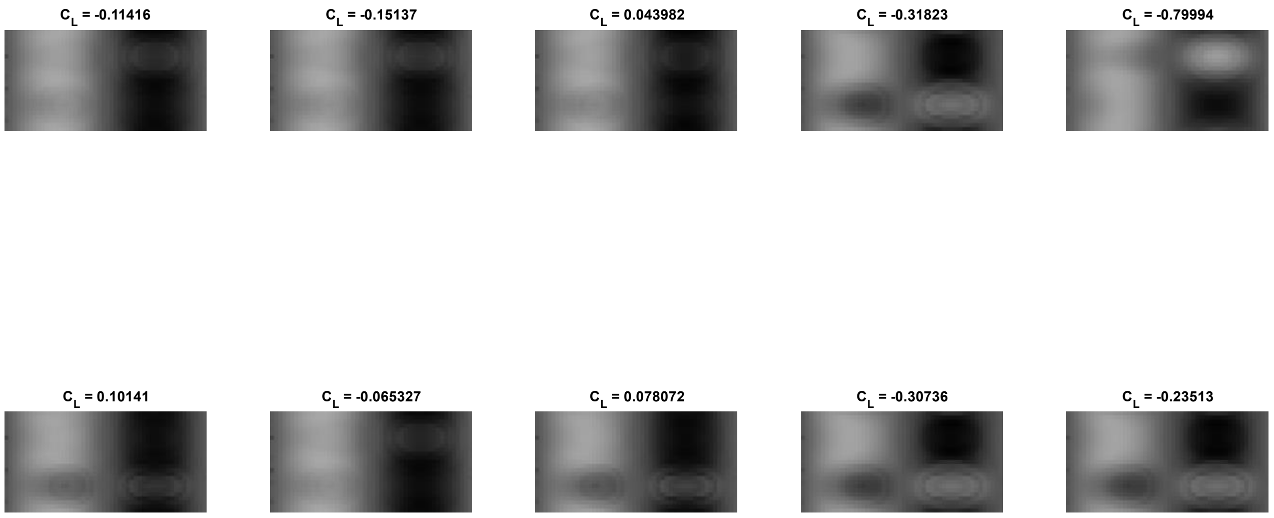

Figure 13: Reconstructed Lift Stress Field After the first Layer

Then we did the same plotting after the second autoencoder, we reconstructed stress fields corresponding to C_D and C_D values in the test set. The autoencoder was able to reconstruct stress fields with a root mean squared of 4.1605% and 3.7656% for C_D and C_L, respectively. Reconstructed stress fields with their C_D and C_D values are shown in Figure 14 and Figure 15 and actual stress fields with their C_D and C_L values are shown in Figure 10 and Figure 11.

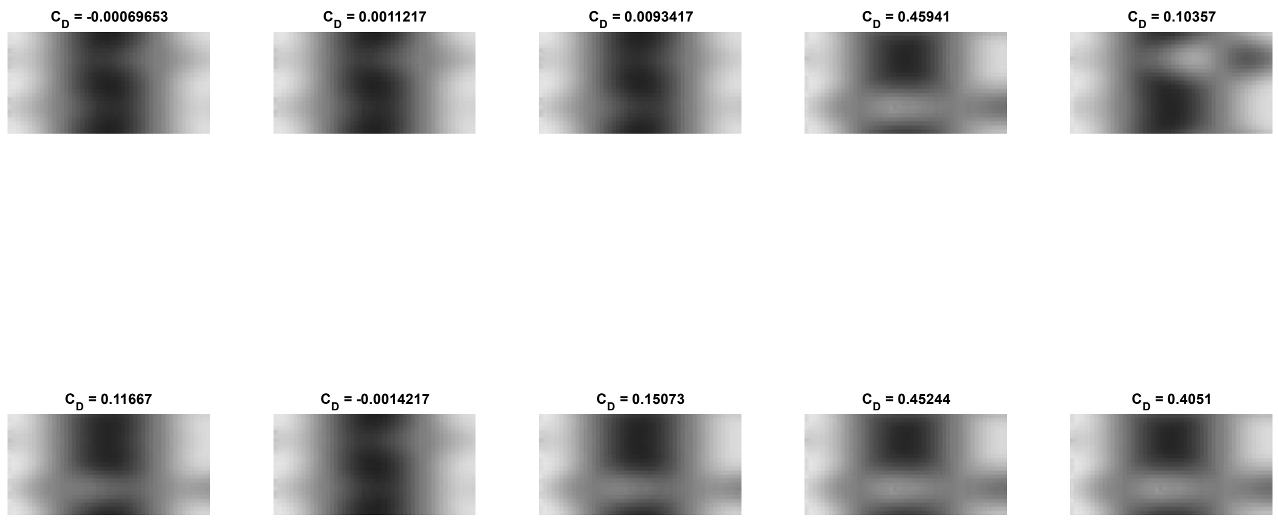

Figure 14: Reconstructed Drag Stress Field After the second Layer

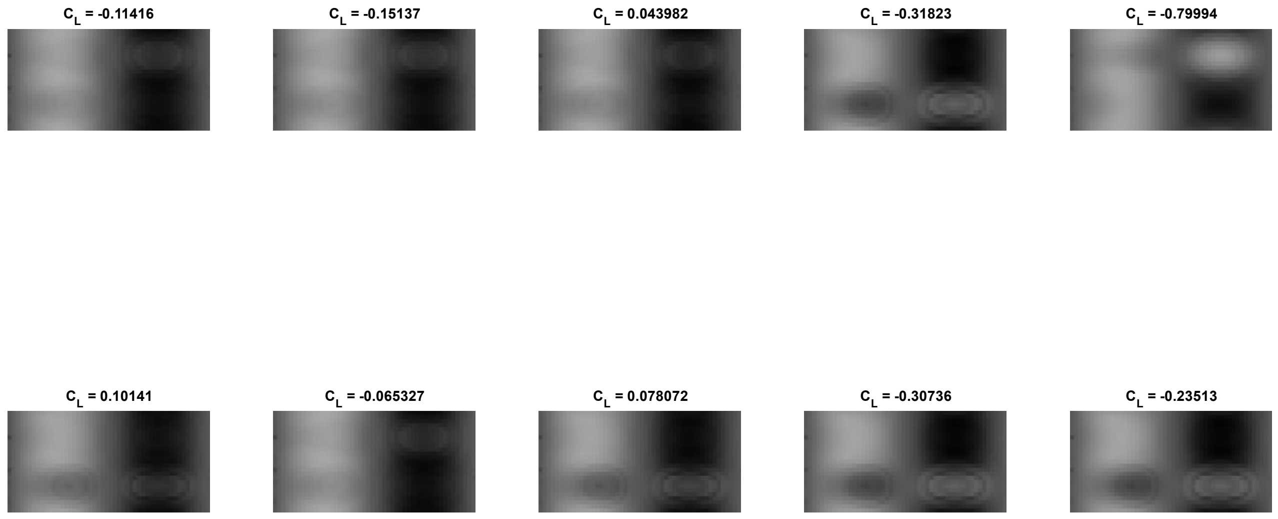

Figure 15: Reconstructed Lift Stress Field After the second Layer

We reconstructed stress fields corresponding to C_D and C_L values in the test set after the third autoencoder. The autoencoder was able to reconstruct stress fields with a root mean squared error of 5.6558% and 5.1226% for C_D and C_L, respectively. Reconstructed stress fields with their C_D and C_L values are shown in Figure 16 and Figure 17 and actual stress fields with their C_D and C_L values are shown in Figure 10 and Figure 11.

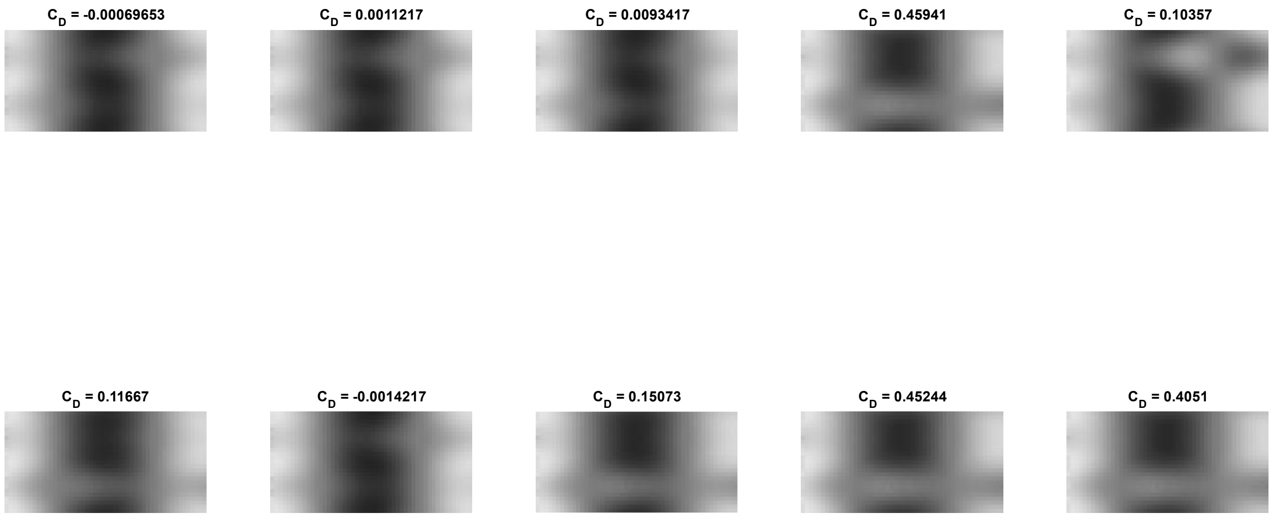

Figure 16: Reconstructed Drag Stress Field After the third Layer

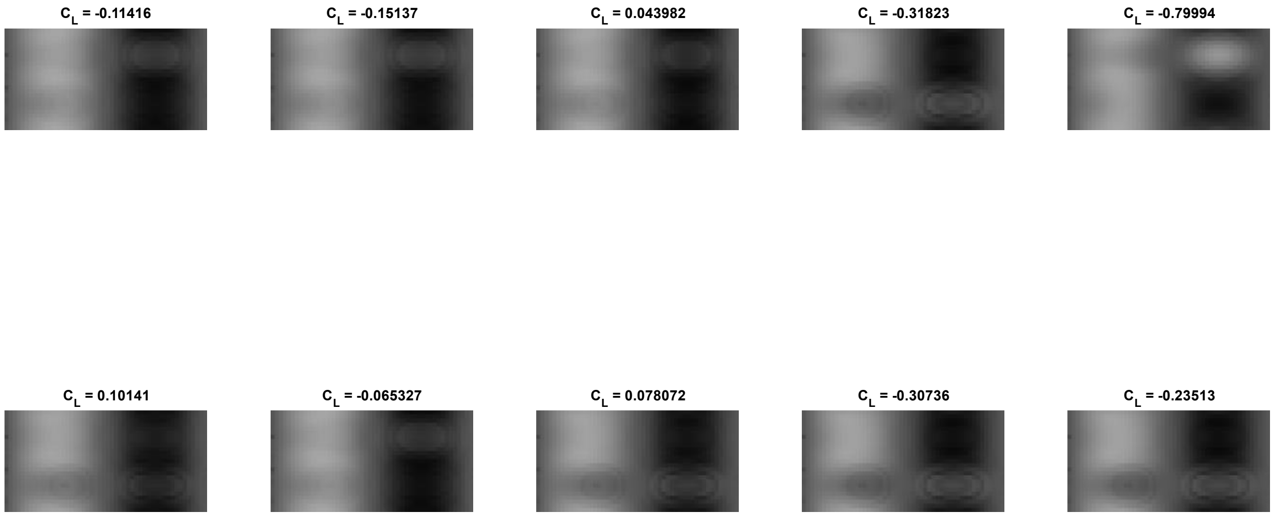

Figure 17: Reconstructed Lift Stress Field After the third Layer

After the shallow network, C_D and C_L is predicted with root mean squared error of 6.5124% and 18.6012%, respectively. Then we plotted the actual and predicted C_D and C_L values in the same plot as shown in Figure 18.

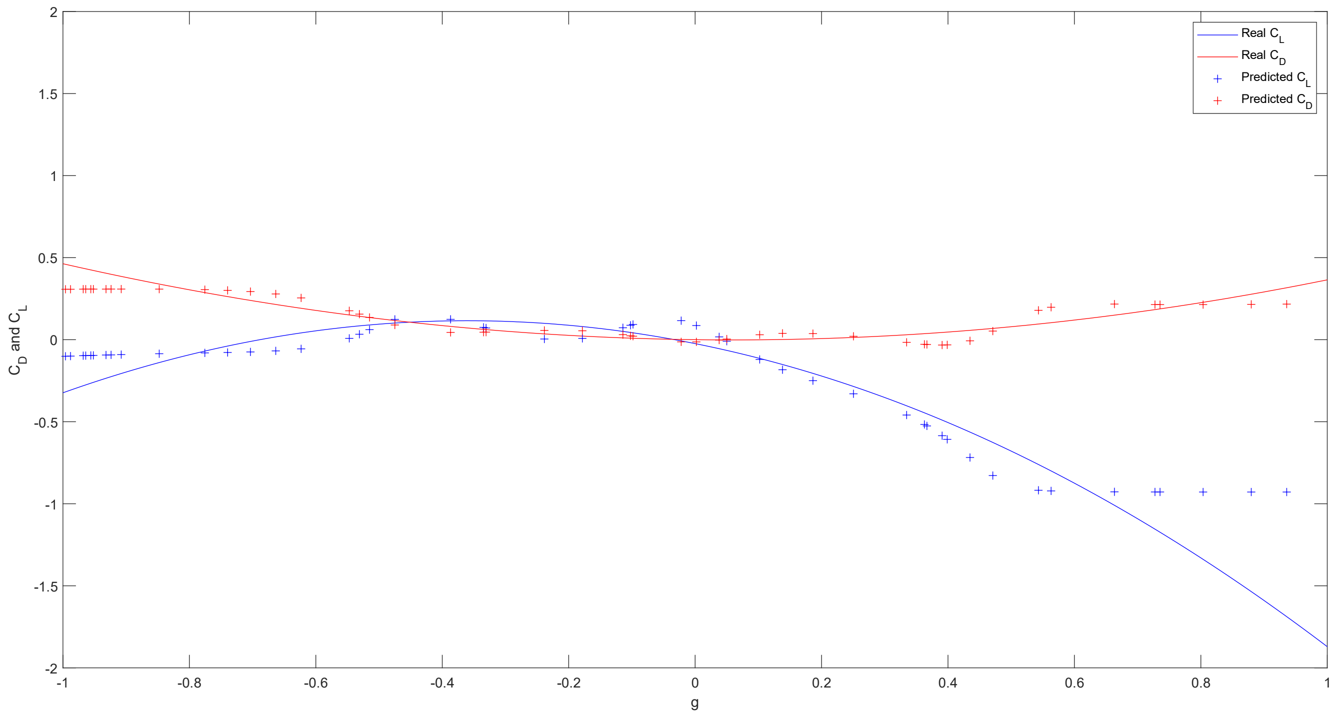

Figure 18: Real CD and CL values vs Predicted CD and CL values

Approach 3: Magnus Effect with Deep Learning

Performance Measure

In order to measure the performance of our system, we created 2 metrics. First one being for the reconstructed C_L accuracy and the other one for the reconstructed C_D accuracy.

We applied the following formula:

$$e = \sqrt{\frac{\sum_{n=1}^{num}(a_n - p_n)^2}{num\_samples}}$$

where a_n is the actual C_L for stress field n, and p_n is the reconstructed C_L for stress field n.

For C_D prediction accuracy we used the same metric as we did for C_L.

Results

The autoencoder was able to reconstruct stress fields with a root mean squared error of 3.2% and 1.83% for C_D and C_L, respectively. Comparison of sample reconstructed and real stress fields for drag and lift forces are shown in Figure 19 and Figure 20. Also, the comparison of reconstructed and real C_D and C_L values can be seen in Figure 21 and Figure 22.

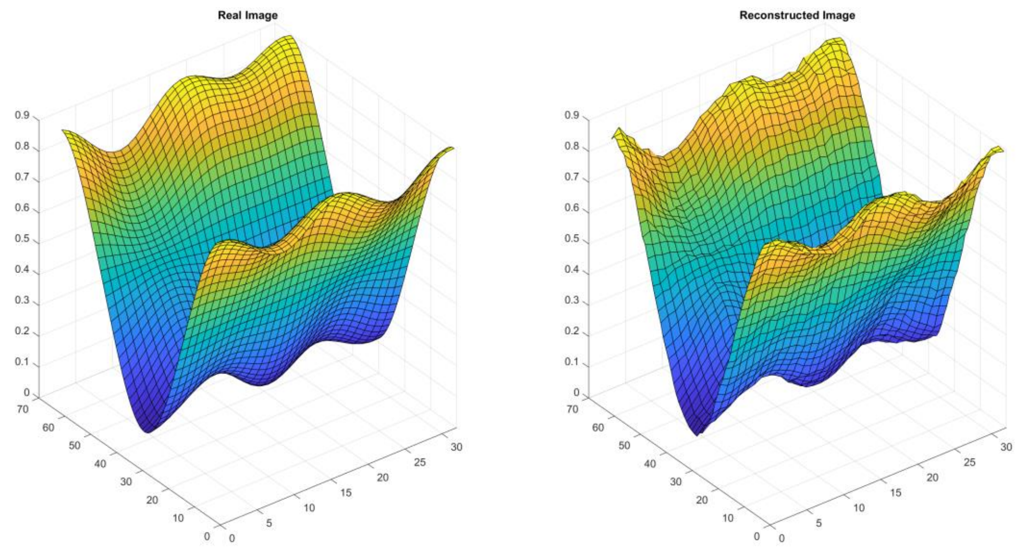

Figure 19: Sample Drag Field Comparison

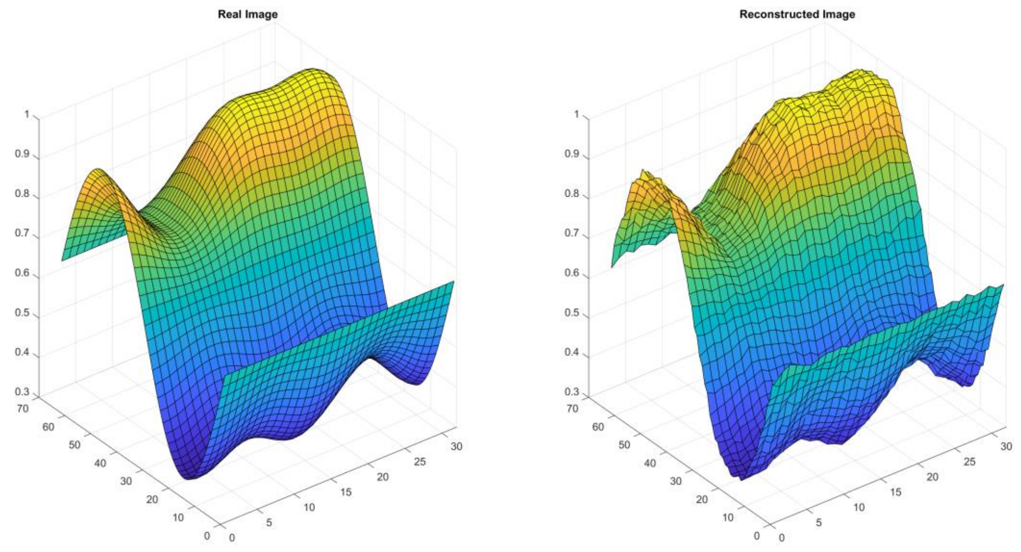

Figure 20: Sample Lift Field Comparison

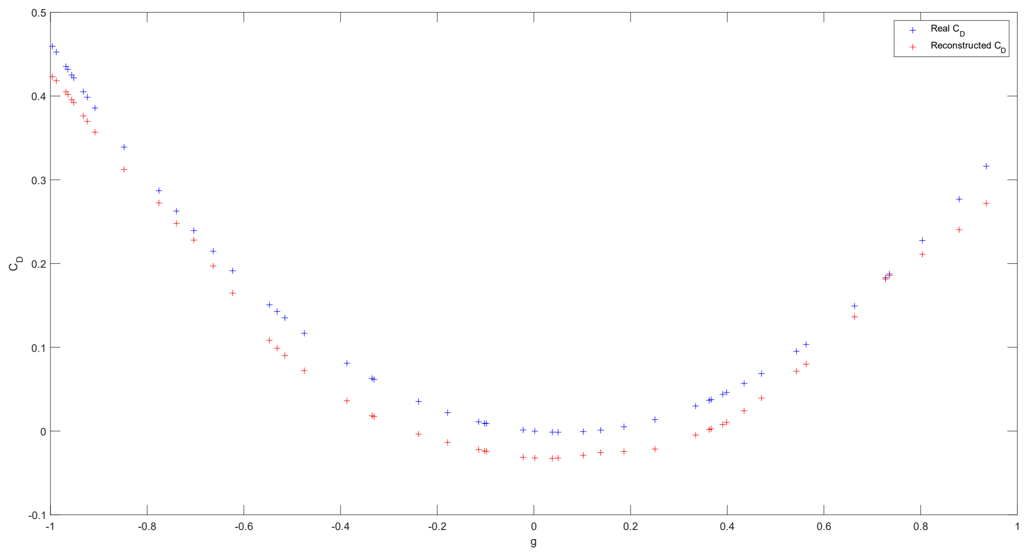

Figure 21: Drag Coefficient Comparison

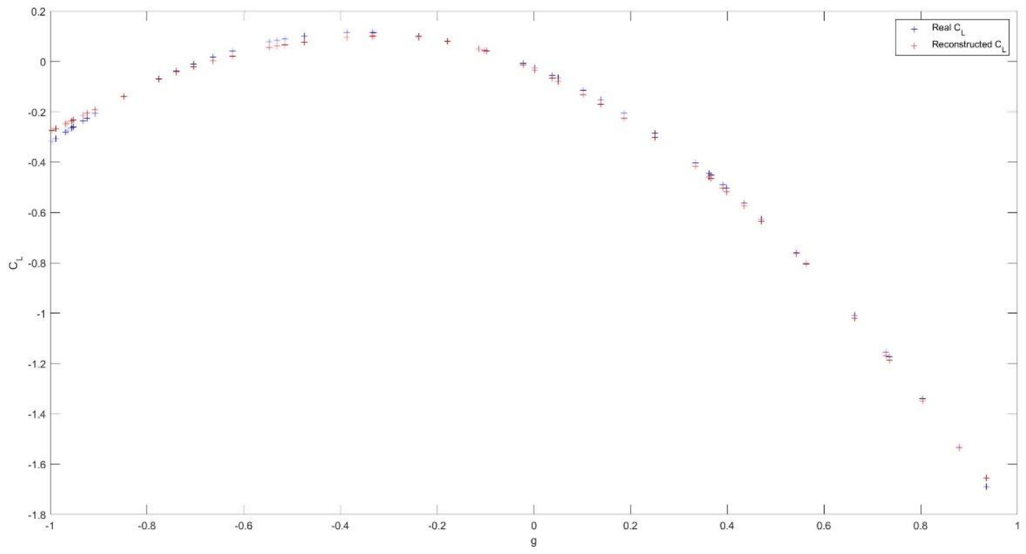

Figure 22: Lift Coefficient Comparison

Discussion

We managed to achieve the initial objectives of the project. Using our model, one can predict C_D and C_L values. It is also possible to reconstruct the images from the encoded vectors. This creates the possibility to minimize the drag force by optimizing the control parameter.

We decided to focus on the toy problem for the whole duration of the project instead of moving to a more complex problem because, if we manage to improve our model on the toy problem, it can be easily adapted to the more complex problems later.

We observed that our model was more accurate when used with the first approach. It had difficulties in predicting C_L in the second approach. We failed to improve the accuracies for the boundary cases.

In order to improve the reconstruction accuracies, we used deep learning algorithms in our autoencoders. We achieved better reconstruction accuracies compared to the previous method especially for C_L.

We managed to show that C_D and C_L values do not necessarily need to be predicted from large velocity fields of the entire domain, they can also be predicted from much smaller stress fields of the object. This is an important addition to the previous state-of-the-art due to the decreased amount of simulation and training times as well as decreased storage requirement.

IMPACT

We developed a model that will predict the values of drag and lift coefficient of a given stress field. This will allow users to optimize their systems without using any computationally costly model. There is no FTU issues, anyone can use our model to build something on to and make it more domain specific.

ETHICAL ISSUES

There are no ethical issues present in our solution to the problem stated.

PROJECT MANAGEMENT

Initially our goal was to apply this model to a more complex problem (such as a wind turbine blade) after applying it to a toy example. However, we have realized that it will be better to improve the model using the toy example as all the steps (data generation, training, testing, etc.) are much quicker. This allowed us to make great improvements in short amounts of time.

Another reason we changed our plans in this regard is that after improving the model for the toy example, it is relatively easy to adapt it to the complex problem. It turned out that our decision to change plans was good because we managed to improve our model a lot.

We have learned that even though it is important to have a thorough project plan, it is also important to know when to change these plans. By changing our data generation method from COMSOL to MATLAB, we managed to avoid setbacks in the progress.

CONCLUSION AND FUTURE WORK

Our work has shown that it is possible to predict C_D and C_L values from the stress field on the object rather than the velocity field. This greatly improves the opportunities to improve this field as it saves a lot of time and space.

The main limitation in this project was the amount of data generated by COMSOL. As CFD simulations take a long time, this is the biggest setback as it decreases the available data, and data is the most important part of machine learning. By using MATLAB for data generation, we managed to save time on data generation and spend it on building more accurate models.

There are multiple things that can be done from now on. Greater accuracies might be achieved by using PCA for compression instead of autoencoders. Additionally, an optimization problem for the control variable could be solved to minimize the drag force. The same algorithm could be slightly modified and applied to more complex problems than our toy example.

After this algorithm is applied to more complex problems successfully and the optimization problem is solved, it would be possible to apply this to a real-world problem where we could minimize drag force to maximize energy efficiency.

```
clear all
clc
U = 1;
R = .1;
rho = 1;
omega = 2*pi;
% we are modeling the forces on the cylinder using the potential flow
% theory
%  g: is theta dependent circulation! (as if there are wholes on the
%  cylinder and they disturb the flow and change its direction).
%
vth = @(th,t,g) -2*U*R*sin(th) + (sin(omega*t)*pi + .15)*.5/pi + g*sin(th/3) ;
p   = @(th,t,g) 1/2*rho*(U^2 - vth(th,t,g).*vth(th,t,g));
sigmax  = @(th,t,g) cos(th).*p(th,t,g);
sigmay  = @(th,t,g) sin(th).*p(th,t,g);
Nth = 50;
Nt  = 25;
th  = [0:2*pi/(Nth-1):2*pi];
t = [0:1/(Nt-1):1]';
Ng = 1000;
gg = [-1:2/(Ng-1):1];

for j=1:Ng
    g = gg(j);
    for k=1:Nt
        fx(k) = trapz(th,sigmax(th,t(k),g))/(2*pi);
        fy(k) = trapz(th,sigmay(th,t(k),g))/(2*pi);
    end
    fybar(j) = trapz(t,fy)/(rho*U*U*R);
    fxbar(j) = trapz(t,fx)/(rho*U*U*R);
end
%%
figure(1)
plot(gg,fxbar,gg,-fybar);
%grid on
%%
%figure
%contourf(sigmay(th,t,1));
%%
%figure
%contourf(sigmax(th,t,0.5))
%%
my_dir = pwd;
backslashes = strfind(my_dir,filesep);
data_dir = my_dir(1:backslashes(end)-1) + "\MATLAB data";

writematrix(fxbar, data_dir + "\Magnus_Train_Data_1000s_50th_25t\CD.dat");
writematrix(-fybar, data_dir + "\Magnus_Train_Data_1000s_50th_25t\CL.dat");
```

```
for i = 1:Ng
    writematrix(sigmax(th,t,gg(i)), data_dir +↙
"\Magnus_Train_Data_1000s_50th_25t\sampleCD_" + int2str(i) + ".dat");
    writematrix(sigmay(th,t,gg(i)), data_dir +↙
"\Magnus_Train_Data_1000s_50th_25t\sampleCL_" + int2str(i) + ".dat");
end
```

```
clear all; close all;

%% PARAMETER INITIALIZATIONS
num_samples = 500;
train_size = num_samples * 0.60;
val_size = num_samples * 0.30;
test_size = num_samples * 0.1;
sample_x_size = 25;
sample_y_size = 50;
Ng = num_samples;

bookKeeping_CL = zeros(3,Ng+1);
bookKeeping_CD = zeros(3,Ng+1);
bookKeeping_CL(3,:) = 5 * ones(1,Ng+1);
bookKeeping_CD(3,:) = 5 * ones(1,Ng+1);

a1_errors = [];
a2_errors = [];
a3_errors = [];

%% LOAD DATA (X)
my_dir = pwd;
backslashes = strfind(my_dir,filesep);
data_dir = my_dir(1:backslashes(end)-1) + "\MATLAB data\Magnus_Train_Data_" + ...
    int2str(num_samples) + "s_" + num2str(sample_y_size) + "th_" + num2str ↙
(sample_x_size) + "t";
data = cell(1,num_samples);

CL_val = load(data_dir + "\CL.dat");
CD_val = load(data_dir + "\CD.dat");

bookKeeping_CL(2,1:Ng) = CL_val;
bookKeeping_CD(2,1:Ng) = CD_val;

CL = num2cell(CL_val(1:Ng), [1 2]);
CD = num2cell(CD_val(1:Ng), [1 2]);
for i = 1:num_samples
    sample_lift = load(data_dir + "\sampleCL_" + i + ".dat");
    sample_drag = load(data_dir + "\sampleCD_" + i + ".dat");

    data_lift(1,i) = num2cell(sample_lift(1:sample_x_size,1:sample_y_size), [1 2]);
    data_drag(1,i) = num2cell(sample_drag(1:sample_x_size,1:sample_y_size), [1 2]);
end

%% NORMALIZATION
max_lift_stress = intmin;
min_lift_stress = intmax;
max_drag_stress = intmin;
min_drag_stress = intmax;
```

```
for i = 1:num_samples
    if (max_lift_stress < max(data_lift{i},[],'all'))
        max_lift_stress = max(data_lift{i},[],'all');
    end
    if (min_lift_stress > min(data_lift{i},[],'all'))
        min_lift_stress = min(data_lift{i},[],'all');
    end
    if (max_drag_stress < max(data_drag{i},[],'all'))
        max_drag_stress = max(data_drag{i},[],'all');
    end
    if (min_drag_stress > min(data_drag{i},[],'all'))
        min_drag_stress = min(data_drag{i},[],'all');
    end
end

for i = 1:num_samples
    cur_lift = data_lift{i};
    cur_drag = data_drag{i};
    for j = 1:sample_x_size
        for k = 1:sample_y_size
            cur_lift(j,k) = (cur_lift(j,k) - min_lift_stress) / (max_lift_stress - ↙
min_lift_stress);
            cur_drag(j,k) = (cur_drag(j,k) - min_drag_stress) / (max_drag_stress - ↙
min_drag_stress);
        end
    end
    data_lift{i} = cur_lift;
    data_drag{i} = cur_drag;
end

%% SHUFFLE AND SPLIT
indexes = randperm(size(data, 2));

X_CL = data_lift(:,indexes);
X_CD = data_drag(:,indexes);
Y_CL = CL_val(indexes);
Y_CD = CD_val(indexes);
g = [-1:2/(Ng-1):1];

bookKeeping_CL(1,1:Ng) = g;
bookKeeping_CD(1,1:Ng) = g;

g = g(indexes);

X_CL_train = X_CL(:,1:train_size);
Y_CL_train = Y_CL(1:train_size);
X_CD_train = X_CD(:,1:train_size);
Y_CD_train = Y_CD(1:train_size);
```

```
X_CL_val = X_CL(:,train_size+1:train_size+val_size);
Y_CL_val = Y_CL(train_size+1:train_size+val_size);
X_CD_val = X_CD(:,train_size+1:train_size+val_size);
Y_CD_val = Y_CD(train_size+1:train_size+val_size);

X_CL_test = X_CL(:,train_size+val_size+1:num_samples);
Y_CL_test = Y_CL(train_size+val_size+1:num_samples);
X_CD_test = X_CD(:,train_size+val_size+1:num_samples);
Y_CD_test = Y_CD(train_size+val_size+1:num_samples);
g_test = g(train_size+val_size+1:num_samples);
%%
 for i = 1:2

     if i == 1
        X_train = X_CL_train;
        X_val = X_CL_val;
        X_test = X_CL_test;

        Y_train = Y_CL_train;
        Y_val = Y_CL_val;
        Y_test = Y_CL_test;
     elseif i == 2
        X_train = X_CD_train;
        X_val = X_CD_val;
        X_test = X_CD_test;

        Y_train = Y_CD_train;
        Y_val = Y_CD_val;
        Y_test = Y_CD_test;
     end

    %% AUTOENCODER LAYER 1
    %Training
    hiddenSize_1 = 256;
    autoenc_1 = trainAutoencoder(X_train,hiddenSize_1,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',2,...
        'SparsityProportion',0.25,...
        'MaxEpochs', 256,...
        'ShowProgressWindow',true);

    %Testing
    X_reconstructed = predict(autoenc_1,X_test);

    %Comparisson of real images and reconstructed images
    if i == 1
        figure('Name','Real Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_test{n});
            title(['C_L = ', num2str(Y_test(n))])
        end
        figure('Name','Reconstructed Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_reconstructed{n});
            title(['C_L = ', num2str(Y_test(n))])
        end
    end
    if i == 2
        figure('Name','Real Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_test{n});
            title(['C_D = ', num2str(Y_test(n))])
        end
        figure('Name','Reconstructed Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_reconstructed{n});
            title(['C_D = ', num2str(Y_test(n))])
        end
    end


    %Error calculation
    err = 0;
    max_val = intmin;
    min_val = intmax;
    for j = 1:test_size
        err = err + sum(sum((X_reconstructed{j} - X_test{j}).^2));
        if (max_val < max(max(X_test{j})))
            max_val = max(max(X_test{j}));
        end
        if (min_val > min(min(X_test{j})))
            min_val = min(min(X_test{j}));
        end
    end
    merr = err / test_size;
    mmerr = sqrt(merr / (sample_x_size*sample_y_size));
    autoencoder_layer_1_relative_mean_error = 100 * mmerr;
    %autoencoder_layer_1_relative_mean_error = 100 * (mmerr - min_val) / (max_val - ↙
min_val);

    %% AUTOENCODER LAYER 2
    %Encode the data
    X_train_encoded = num2cell(encode(autoenc_1, X_train), 1);
    X_val_encoded = num2cell(encode(autoenc_1, X_val), 1);
    X_test_encoded = num2cell(encode(autoenc_1, X_test), 1);

    %Training
    hiddenSize_2 = 64;
    autoenc_2 = trainAutoencoder(X_train_encoded,hiddenSize_2,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',2,...
        'SparsityProportion',0.25,...
        'MaxEpochs', 256,...
        'ShowProgressWindow',true);

    %Testing
    X_encoded_reconstructed = predict(autoenc_2,X_test_encoded);
    X_reconstructed_reconstructed = decode(autoenc_1,cell2mat ↙
(X_encoded_reconstructed));

    %Comparisson of real images and reconstructed images
    if i == 1
        figure('Name','Real Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_test{n});
            title(['C_L = ', num2str(Y_test(n))])
        end
        figure('Name','Reconstructed Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_reconstructed_reconstructed{n});
            title(['C_L = ', num2str(Y_test(n))])
        end
    end
    if i == 2
        figure('Name','Real Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_test{n});
            title(['C_D = ', num2str(Y_test(n))])
        end
        figure('Name','Reconstructed Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_reconstructed_reconstructed{n});
            title(['C_D = ', num2str(Y_test(n))])
        end
    end


    %Error calculation
    err = 0;
    max_val = intmin;
    min_val = intmax;
    for k = 1:test_size
        err = err + sum(sum((X_reconstructed_reconstructed{k} - X_test{k}).^2));
        if (max_val < max(max(X_test{k})))
            max_val = max(max(X_test{k}));
        end
        if (min_val > min(min(X_test{k})))
            min_val = min(min(X_test{k}));
        end
    end
    merr = err / test_size;
    mmerr = sqrt(merr / (sample_x_size*sample_y_size));
    autoencoder_layer_2_relative_mean_error = 100 * mmerr;
    %autoencoder_layer_2_relative_mean_error = 100 * (mmerr - min_val) / (max_val - ↙
min_val);

    %% AUTOENCODER LAYER 3
    %Encode the data
    X_train_encoded_encoded = num2cell(encode(autoenc_2, X_train_encoded), 1);
    X_val_encoded_encoded = num2cell(encode(autoenc_2, X_val_encoded), 1);
    X_test_encoded_encoded = num2cell(encode(autoenc_2, X_test_encoded), 1);

    %Training
    hiddenSize_3 = 16;
    autoenc_3 = trainAutoencoder(X_train_encoded_encoded,hiddenSize_3,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',2,...
        'SparsityProportion',0.25,...
        'MaxEpochs', 256,...
        'ShowProgressWindow',true);

    %Testing
    X_encoded_encoded_reconstructed = predict(autoenc_3,X_test_encoded_encoded);
    X_reconstructed_reconstructed_reconstructed = decode(autoenc_1,cell2mat(decode ↙
(autoenc_2,cell2mat(X_encoded_encoded_reconstructed))));

        %Comparisson of real images and reconstructed images
    if i == 1
        figure('Name','Real Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_test{n});
            title(['C_L = ', num2str(Y_test(n))])
        end
        figure('Name','Reconstructed Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_reconstructed_reconstructed_reconstructed{n});
            title(['C_L = ', num2str(Y_test(n))])
        end
    end
    if i == 2
        figure('Name','Real Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_test{n});
            title(['C_D = ', num2str(Y_test(n))])
        end
        figure('Name','Reconstructed Test Data');
        for n = 1:10
            subplot(2,10/2,n);
            imshow(X_reconstructed_reconstructed_reconstructed{n});
            title(['C_D = ', num2str(Y_test(n))])
        end
    end


    %Error calculation
    err = 0;
    max_val = intmin;
    min_val = intmax;
    for m = 1:test_size
        err = err + sum(sum((X_reconstructed_reconstructed_reconstructed{m} - X_test ↙
{m}).^2));
        if (max_val < max(max(X_test{m})))
            max_val = max(max(X_test{m}));
        end
        if (min_val > min(min(X_test{m})))
            min_val = min(min(X_test{m}));
        end
    end
    merr = err / test_size;
    mmerr = sqrt(merr / (sample_x_size*sample_y_size));
    autoencoder_layer_3_relative_mean_error = 100 * mmerr;
    %autoencoder_layer_3_relative_mean_error = 100 * (mmerr - min_val) / (max_val - ↙
min_val);

    %% PCA
    %asda = cell2mat(X_train_encoded_encoded_encoded)';
    %coef = pca(asda)


    %% SHALLOW NETWORK
    %Encode the data
    X_train_encoded_encoded_encoded = num2cell(encode(autoenc_3, ↙
X_train_encoded_encoded), 1);
    X_val_encoded_encoded_encoded = num2cell(encode(autoenc_3, ↙
X_val_encoded_encoded), 1);
    X_test_encoded_encoded_encoded = num2cell(encode(autoenc_3, ↙
X_test_encoded_encoded), 1);

    %Define the NN architecture
    layers = [
        sequenceInputLayer(hiddenSize_3,'Name','Encoded Data Input Layer')
        fullyConnectedLayer(64,'Name','Fully Connected Regression Layer-1')
        tanhLayer('Name','Activation Function Layer-1')
        fullyConnectedLayer(16,'Name','Fully Connected Regression Layer-2')
        tanhLayer('Name','Activation Function Layer-2')
        fullyConnectedLayer(1,'Name','Fully Connected Regression Layer-3')
        tanhLayer('Name','Activation Function Layer-3')
        regressionLayer('Name','Output Layer')];

    options = trainingOptions('sgdm', ...
        'MiniBatchSize',3, ...
        'MaxEpochs',15, ...
        'InitialLearnRate',1e-3, ...
        'LearnRateSchedule','piecewise', ...
        'LearnRateDropFactor',0.1, ...
        'LearnRateDropPeriod',10, ...
        'Shuffle','every-epoch', ...
        'ValidationData',{X_val_encoded_encoded_encoded,num2cell(Y_val)}, ...
        'ValidationFrequency',1, ...
        'Plots','training-progress', ...
        'Verbose',false);
        %'Plots','none', ...
    %Train and test
    net = trainNetwork(X_train_encoded_encoded_encoded,num2cell(Y_train),layers, ↙
options);
    Y_pred = predict(net,X_test_encoded_encoded_encoded);

    comp=[Y_test', cell2mat(Y_pred)];

    %% BOOKKEEPING
    %Error calculation
    err = 0;
    max_val = intmin;
    min_val = intmax;
    for n = 1:test_size
        err = err + (Y_pred{n} - Y_test(n))^2;
        if(i == 1) % if it is CL
            col = findPlace(g_test(n),bookKeeping_CL);
            bookKeeping_CL(3,col) = Y_pred{n};
        else
            col = findPlace(g_test(n),bookKeeping_CD);
            bookKeeping_CD(3,col) = Y_pred{n};
        end
        if (max_val < max(Y_test(n)))
            max_val = max(Y_test(n));
        end
        if (min_val > min(Y_test(n)))
            min_val = min(Y_test(n));
        end
    end
    val_range = max_val - min_val;

    merr = sqrt(err / test_size);
    %%
    if(i == 1)
        CL_prediction_relative_mean_error = 100 * merr;
        bookKeeping_CL(3,Ng+1) = CL_prediction_relative_mean_error;
    else
        CD_prediction_relative_mean_error = 100 * merr;
        bookKeeping_CD(3,Ng+1) = CD_prediction_relative_mean_error;
    end
    a1_errors(i) = autoencoder_layer_1_relative_mean_error;
    a2_errors(i) = autoencoder_layer_2_relative_mean_error;
    a3_errors(i) = autoencoder_layer_3_relative_mean_error;
end

%% PLOTTING
figure
plot(bookKeeping_CL(1,1:num_samples),bookKeeping_CL(2,1:num_samples),'-b')
hold on
plot(bookKeeping_CD(1,1:num_samples),bookKeeping_CD(2,1:num_samples),'-r')

yplot= bookKeeping_CL(3,1:num_samples);      % make a copy of the data specifically ↙
for plotting
yplot(yplot==0)=nan;                         % replace 0 elements with NaN
plot(bookKeeping_CL(1,1:num_samples),yplot,'+b')

yplot= bookKeeping_CD(3,1:num_samples);      % make a copy of the data specifically ↙
for plotting
yplot(yplot==0)=nan;                         % replace 0 elements with NaN
plot(bookKeeping_CD(1,1:num_samples),yplot,'+r')

ylim([-2,2])
xticks(-1:0.2:1)
xlabel('g')
ylabel('C_D and C_L')
legend('Real C_L','Real C_D','Predicted C_L','Predicted C_D')

hold off

%% STATS
```

```
errors_CL =bookKeeping_CL(3,Ng+1) ;
%stdE_CL = std(errors_CL);
%meanE_CL = mean(errors_CL);

errors_CD =bookKeeping_CD(3,Ng+1) ;
%stdE_CD = std(errors_CD);
%meanE_CD = mean(errors_CD);

% std_a1 = std(a1_errors);
% mean_a1 = mean(a1_errors);
%
% std_a2 = std(a2_errors);
% mean_a2 = mean(a2_errors);
%
% std_a3 = std(a3_errors);
% mean_a3 = mean(a3_errors);
```

```
function col = findPlace(test_g,bookKeeping)
    r = 1;
    term = false;
    col = 0;
    sz = size(bookKeeping);
    while r < sz(2) && ~term
        if(round(bookKeeping(1,r)-test_g,3) == 0)
            col = r;
            term = true;
        end
        if ~term
        r = r+1;
        end
    end

end
```

```
clear all; close all;

%% PARAMETER INITIALIZATIONS
num_samples = 500;
train_size = num_samples * 0.60;
val_size = num_samples * 0.30;
test_size = num_samples * 0.1;
sample_x_size = 32;
sample_y_size = 64;
Ng = num_samples;

bookKeeping_CL = zeros(3,Ng+1);
bookKeeping_CD = zeros(3,Ng+1);
bookKeeping_CL(3,:) = 5 * ones(1,Ng+1);
bookKeeping_CD(3,:) = 5 * ones(1,Ng+1);

reconstructed_CL = zeros(2,test_size);
reconstructed_CD = zeros(2,test_size);
th  = [0:2*pi/(sample_y_size-1):2*pi];
t = [0:1/(sample_x_size-1):1]';
U = 1;
R = .1;
rho = 1;


%% LOAD DATA (X)
my_dir = pwd;
backslashes = strfind(my_dir,filesep);
data_dir = my_dir(1:backslashes(end)-1) + "\MATLAB data\Magnus_Train_Data_" + ...
    int2str(num_samples) + "s_" + num2str(sample_y_size) + "th_" + num2str ↙
(sample_x_size) + "t";
data = cell(1,num_samples);

CL_val = load(data_dir + "\CL.dat");
CD_val = load(data_dir + "\CD.dat");

bookKeeping_CL(2,1:Ng) = CL_val;
bookKeeping_CD(2,1:Ng) = CD_val;

CL = num2cell(CL_val(1:Ng), [1 2]);
CD = num2cell(CD_val(1:Ng), [1 2]);
for i = 1:num_samples
    sample_lift = load(data_dir + "\sampleCL_" + i + ".dat");
    sample_drag = load(data_dir + "\sampleCD_" + i + ".dat");

    data_lift(1,i) = num2cell(sample_lift(1:sample_x_size,1:sample_y_size), [1 2]);
    data_drag(1,i) = num2cell(sample_drag(1:sample_x_size,1:sample_y_size), [1 2]);
end
```

```
%% NORMALIZATION
max_lift_stress = intmin;
min_lift_stress = intmax;
max_drag_stress = intmin;
min_drag_stress = intmax;
for i = 1:num_samples
    if (max_lift_stress < max(data_lift{i},[],'all'))
        max_lift_stress = max(data_lift{i},[],'all');
    end
    if (min_lift_stress > min(data_lift{i},[],'all'))
        min_lift_stress = min(data_lift{i},[],'all');
    end
    if (max_drag_stress < max(data_drag{i},[],'all'))
        max_drag_stress = max(data_drag{i},[],'all');
    end
    if (min_drag_stress > min(data_drag{i},[],'all'))
        min_drag_stress = min(data_drag{i},[],'all');
    end
end

for i = 1:num_samples
    cur_lift = data_lift{i};
    cur_drag = data_drag{i};
    for j = 1:sample_x_size
        for k = 1:sample_y_size
            cur_lift(j,k) = (cur_lift(j,k) - min_lift_stress) / (max_lift_stress - ↙
min_lift_stress);
            cur_drag(j,k) = (cur_drag(j,k) - min_drag_stress) / (max_drag_stress - ↙
min_drag_stress);
        end
    end
    data_lift{i} = cur_lift;
    data_drag{i} = cur_drag;
end

%% SHUFFLE AND SPLIT
indexes = randperm(size(data, 2));

X_CL = data_lift(:,indexes);
X_CD = data_drag(:,indexes);
Y_CL = CL_val(indexes);
Y_CD = CD_val(indexes);
g = [-1:2/(Ng-1):1];

bookKeeping_CL(1,1:Ng) = g;
bookKeeping_CD(1,1:Ng) = g;

g = g(indexes);
```

```
X_CL_train = X_CL(:,1:train_size);
Y_CL_train = Y_CL(1:train_size);
X_CD_train = X_CD(:,1:train_size);
Y_CD_train = Y_CD(1:train_size);

X_CL_val = X_CL(:,train_size+1:train_size+val_size);
Y_CL_val = Y_CL(train_size+1:train_size+val_size);
X_CD_val = X_CD(:,train_size+1:train_size+val_size);
Y_CD_val = Y_CD(train_size+1:train_size+val_size);

X_CL_test = X_CL(:,train_size+val_size+1:num_samples);
Y_CL_test = Y_CL(train_size+val_size+1:num_samples);
X_CD_test = X_CD(:,train_size+val_size+1:num_samples);
Y_CD_test = Y_CD(train_size+val_size+1:num_samples);
g_test = g(train_size+val_size+1:num_samples);
%%
for i = 1:2

     if i == 1
        X_train = X_CL_train;
        X_val = X_CL_val;
        X_test = X_CL_test;

        Y_train = Y_CL_train;
        Y_val = Y_CL_val;
        Y_test = Y_CL_test;
     elseif i == 2
        X_train = X_CD_train;
        X_val = X_CD_val;
        X_test = X_CD_test;

        Y_train = Y_CD_train;
        Y_val = Y_CD_val;
        Y_test = Y_CD_test;
     end

     %% CELL2MAT
     for j = 1:train_size
         X_train_mat(:,:,1,j) =  X_train{j};
     end
     Y_train_mat = X_train_mat;

     for j = 1:val_size
         X_val_mat(:,:,1,j) =  X_val{j};
     end
     Y_val_mat = X_val_mat;

     for j = 1:test_size
         X_test_mat(:,:,1,j) =  X_test{j};
```

```
 end
 Y_test_mat = X_test_mat;



 %% MANUAL AUTOENCODER
 imageLayer = imageInputLayer([sample_x_size,sample_y_size,1]);

 encodingLayers = [ ...
    convolution2dLayer(3,16,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2), ...
    convolution2dLayer(3,8,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2), ...
    convolution2dLayer(3,8,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2)];

 decodingLayers = [ ...
    createUpsampleTransponseConvLayer(2,8), ...
    reluLayer, ...
    createUpsampleTransponseConvLayer(2,8), ...
    reluLayer, ...
    createUpsampleTransponseConvLayer(2,16), ...
    reluLayer, ...
    convolution2dLayer(3,1,'Padding','same'), ...
    clippedReluLayer(1.0), ...
    regressionLayer];

layers = [imageLayer,encodingLayers,decodingLayers];

options = trainingOptions('adam', ...
    'MaxEpochs',100, ...
    'MiniBatchSize',15, ...
    'ValidationData',{X_val_mat,Y_val_mat}, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false);

net = trainNetwork(X_train_mat,Y_train_mat,layers,options);

Y_pred = predict(net,X_test_mat);
%%
figure
subplot(1,2,1);
%imshow(X_test_mat(:,:,:,1));
surf(1:sample_x_size,1:sample_y_size,X_test_mat(:,:,:,1)');
title('Real Image')
```

```
    subplot(1,2,2);
    %imshow(Y_pred(:,:,:,1));
    surf(1:sample_x_size,1:sample_y_size,Y_pred(:,:,:,1)');
    title('Reconstructed Image')

    %% CD CL CALCULATION FROM RECONSTRUCTED IMAGE
    if i==1
        for j=1:test_size
            decoded_field = reshape(Y_pred(:,:,:,j), sample_x_size, sample_y_size);
            for k = 1:sample_x_size
                for l = 1:sample_y_size
                    decoded_field(k,l,1) = decoded_field(k,l,1) * (max_lift_stress - ↙
min_lift_stress) + min_lift_stress;
                end
            end
            col = findPlace(g_test(j),bookKeeping_CL);

            fy = trapz(th,decoded_field')/(2*pi);

            reconstructed_CL(1,j) = bookKeeping_CL(2,col);
            reconstructed_CL(2,j) = trapz(t,fy)/(rho*U*U*R);
        end
    elseif i==2
        for j=1:test_size
            decoded_field = reshape(Y_pred(:,:,:,j), sample_x_size, sample_y_size);
            for k = 1:sample_x_size
                for l = 1:sample_y_size
                    decoded_field(k,l,1) = decoded_field(k,l,1) * (max_drag_stress - ↙
min_drag_stress) + min_drag_stress;
                end
            end
            col = findPlace(g_test(j),bookKeeping_CD);

            fx = trapz(th,decoded_field')/(2*pi);

            reconstructed_CD(1,j) = bookKeeping_CD(2,col);
            reconstructed_CD(2,j) = trapz(t,fx)/(rho*U*U*R);
        end
    end

    %%

end


figure
plot(g_test,reconstructed_CL(1,:),'+b')
hold on
plot(g_test,reconstructed_CL(2,:),'+r')
```

```
hold off
xticks(-1:0.2:1)
xlabel('g')
ylabel('C_L')
legend('Real C_L','Reconstructed C_L')

figure
plot(g_test,reconstructed_CD(1,:),'+b')
hold on
plot(g_test,reconstructed_CD(2,:),'+r')
hold off
xticks(-1:0.2:1)
xlabel('g')
ylabel('C_D')
legend('Real C_D','Reconstructed C_D')

%% CD CL error calc
CL_error = 0;
CD_error = 0;
for i = 1:test_size
    CL_error = CL_error + (reconstructed_CL(1,i) - reconstructed_CL(2,i))^2;
    CD_error = CD_error + (reconstructed_CD(1,i) - reconstructed_CD(2,i))^2;
end
CL_error = CL_error / test_size;
CD_error = CD_error / test_size;

CL_error = sqrt(CL_error);
CD_error = sqrt(CD_error);

function out = createUpsampleTransponseConvLayer(factor,numFilters)

filterSize = 2*factor - mod(factor,2);
cropping = (factor-mod(factor,2))/2;
numChannels = 1;

out = transposedConv2dLayer(filterSize,numFilters, ...
    'NumChannels',numChannels,'Stride',factor,'Cropping',cropping);
end
```

REFERENCES

[1] Brunton, S. L., Noack, B. R., & Koumoutsakos, P. (2019). Machine Learning for Fluid Mechanics. *Annual Review of Fluid Mechanics*, *52*(1).

[2] Carlberg, K. T., Jameson, A., Kochenderfer, M. J., Morton, J., Peng, L., & Witherden, F. D. (2019). Recovering missing CFD data for high-order discretizations using deep neural networks and dynamics learning. *Journal of Computational Physics*, 395, 105–124

[3] Guo, X., Li, W., & Iorio, F. (2016). Convolutional Neural Networks for Steady Flow Approximation. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD 16*

[4] Hinton, G. E. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science,* 313(5786), 504–507

[5] Lutter, M., Ritter, C., & Peters, J. (2019). Deep Lagrangian Networks: Using Physics as Model Prior for Deep Learning.

[6] Nagabandi, A., Kahn, G., Fearing, R. S., & Levine, S. (2018). Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*

[7] Rabault, J., Kuchta, M., Jensen, A., Réglade, U., & Cerardi, N. (2019). Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control. *Journal of Fluid Mechanics*, 865, 281–302.

[8] Raissi, M., Perdikaris, P., & Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for olving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707