Physics Informed Neural Networks (PINNs) for rapid contamination dispersion predictions

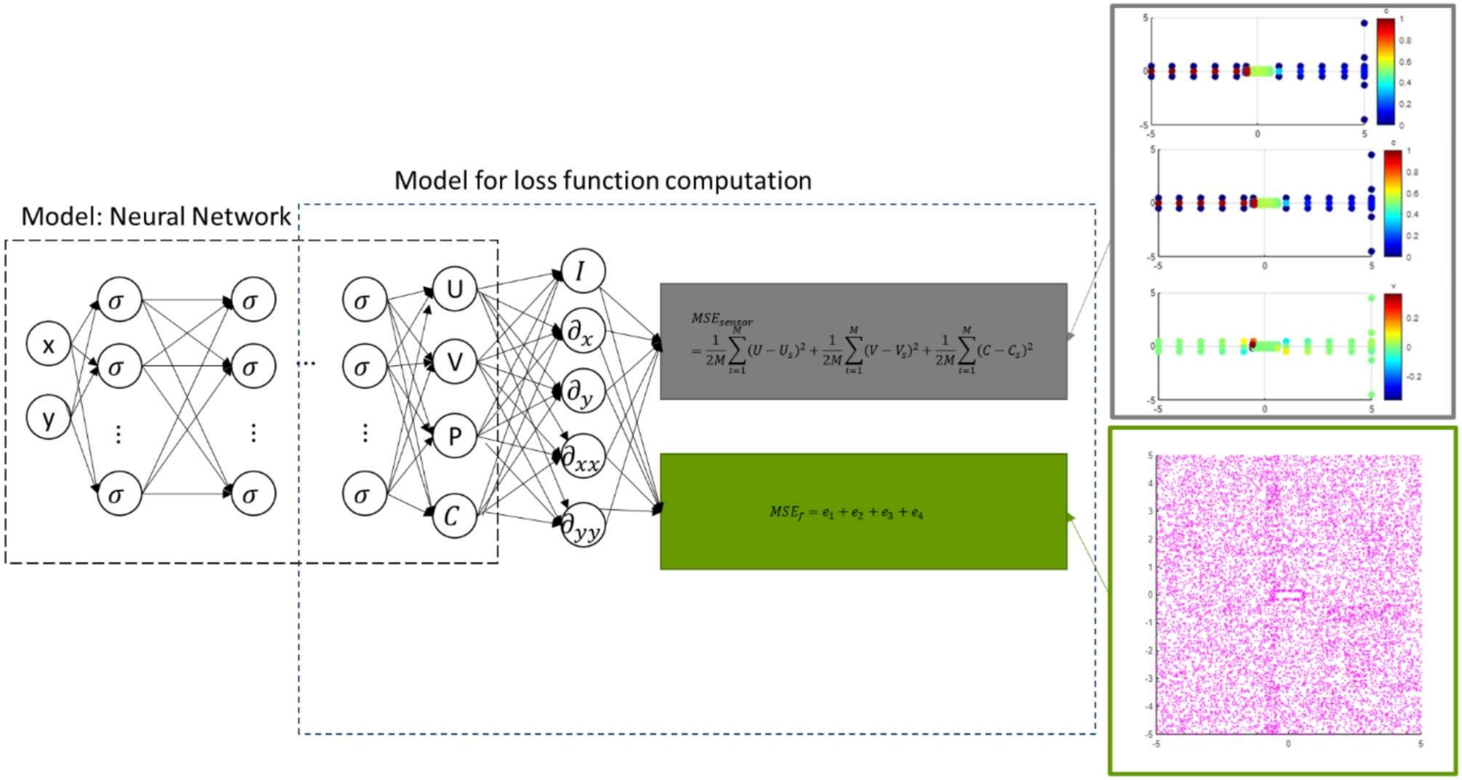
6th May 2024

Feng Zhijing, Elisa Y.M. Ang, Tay Bee Kiat, Koh Wai Heng, Peng Cheng Wang

Introduction

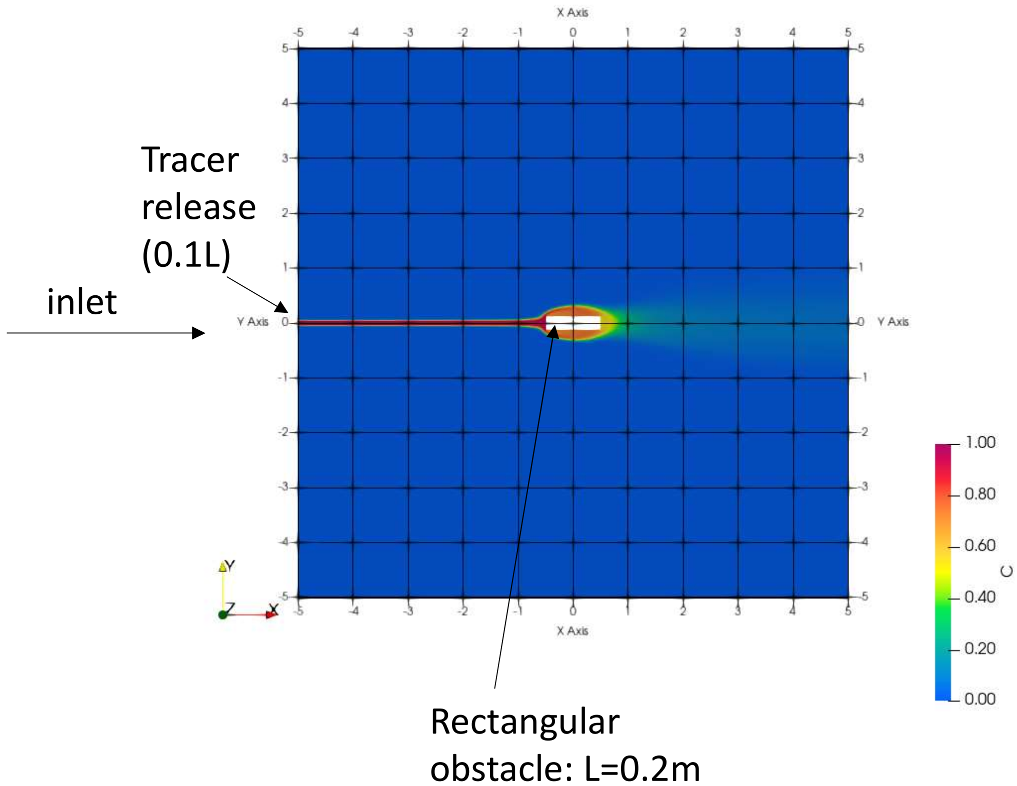
- Contamination dispersion prediction is vital for assessing disease spread, toxic chemical transmission, and indoor air pollutants, and traditionally relies on computational fluid dynamics (CFD)
- CFD requires intensive computation and precise initial conditions, limiting its use in rapid response scenarios.
- Physics Informed Neural Networks (PINN) can be an alternative. PINNs integrate physical laws into their loss functions, allowing for mesh-free solutions to complex equations.
- We can enhance PINN's efficiency by combining it with strategic sensor data, which improve speed and accuracy without detailed environmental data.

How PINN works



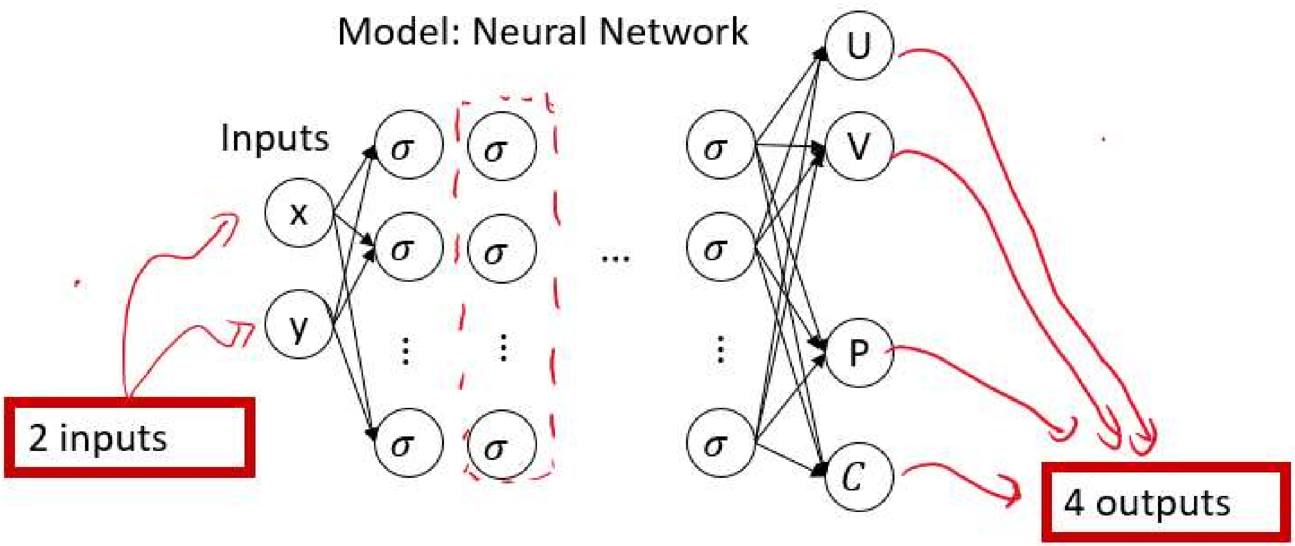
- Input, output and hidden layer
- contain physical equations in the loss function so that we can use both data-driven modelling and domain-specific knowledge

simple 2D scenario



- single release point and a rectangular bluff body placed at the center
- Non-dimensionalized with respect to the x dimension of the obstacle (0.2m)
- The release gas was assumed to have similar properties to air.
- 5 m/s freestream air velocity in the x-direction.

Define neural network model

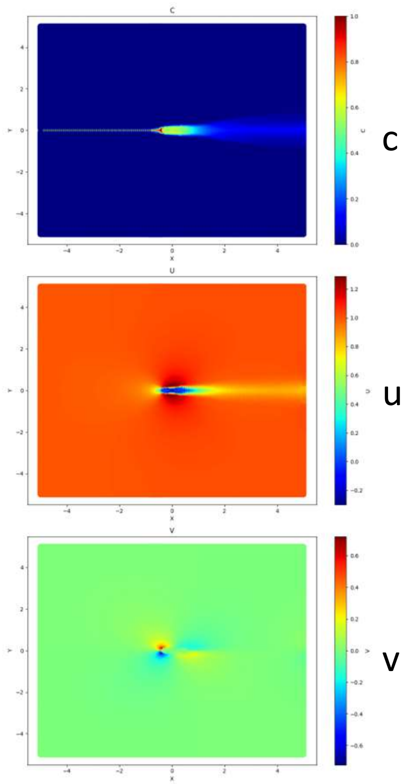


- Inputs: coordinate points, using x,y from 2 datasets:
  - CFD data
  - sensor data
- Output: u,v,p,c
  - U,V: velocity in two directions
  - P: pressure
  - C: concentration
- Focus on U,V,C while p is used for calculating loss function

2 input datasets

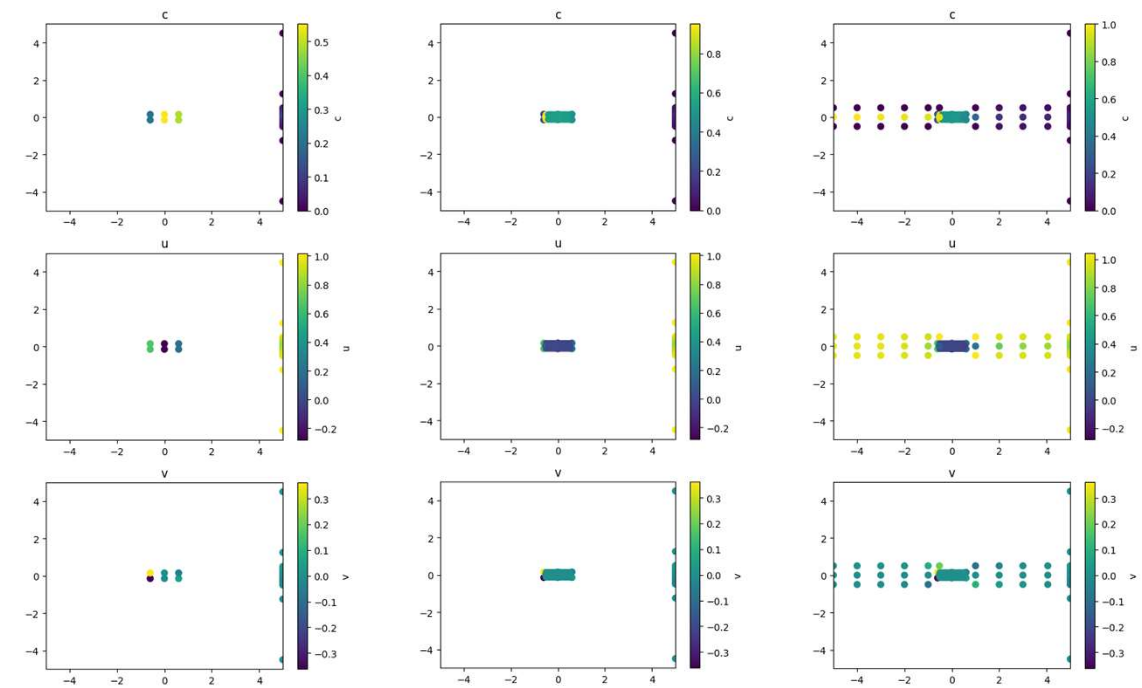
- **CFD data** (45k), obtained from openFoam, coordinates points with u,v,c
- randomly choose 10k for model

openFoam



Sensor data

Sensor configuration 1 (21) | Sensor configuration 2 (45) | Sensor configuration 3 (78)



Define loss function

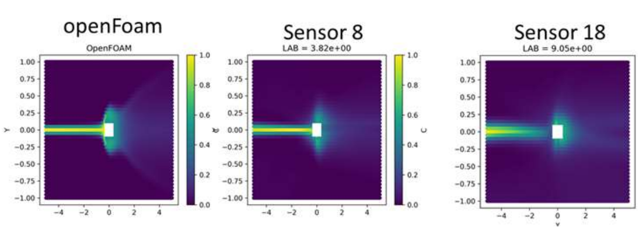
- Loss function contains 2 parts:
  - **loss_CFD**: use CFD datasets, calculate Navier Stokes + Scalar Transport equations, Re refers to the Reynolds number and Pe refers to Peclet number. Re=Pe= 67567.57
    - $e_1 = u_x + v_y = 0$
    - $e_2 = uu_x + vu_y + p_x - \frac{1}{Re}(u_{xx} + u_{yy}) = 0$
    - $e_3 = uv_x + vv_y + p_y - \frac{1}{Re}(v_{xx} + v_{yy}) = 0$
    - $e_4 = uc_x + vc_y - \frac{1}{Pe}(c_{xx} + c_{yy}) = 0$
  - Calculate mse of 4 equations and add 4 losses $MSE_f = e_1 + e_2 + e_3 + e_4$
  - **loss_sensor**: using sensor data, compute mse between predict value and true value

$$MSE_{sensor} = \frac{1}{2M}\sum_{i=1}^{M}\left(\left(U(x_i, y_i) - U_s(x_i, y_i)\right)^2 + \left(V(x_i, y_i) - V_s(x_i, y_i)\right)^2 + \left(C(x_i, y_i) - C_s(x_i, y_i)\right)^2\right)$$

  - **Total loss** = loss_CFD + loss_sensor

Model evaluation metrics: lab

- Transform images from RGB into LAB76 color space
- What's LAB76 color space: L, a, and b represent the 3 parameters which are used to separate out colours
  - L is for lightness. It goes from 0 to 100
  - a is red to green. The negative axis is green and the positive is red.
  - b goes from yellow to blue. Blue lies on the negative side and yellow on the positive one.
- Metrics lab: calculating the average for Mean Delta E in each pixel, $LAB = \frac{1}{n}\sum_{i=1}^{n}\left(\sqrt[2]{\left(L_i - \widehat{L_i}\right)^2 + (a_i - \widehat{a_i})^2 + \left(b_i - \widehat{b_i}\right)^2}\right)$
- Initially, we use Mean Squared Error (MSE) for assessment. However, we observed its inadequacy in certain scenarios, where despite small MSE values, there are substantial differences between predicted and original images.



Sensor config 8 is more accurate than Sensor config 18

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

sensor	MSEc	LABc
sensor 8	0.060106	5.500113
sensor 18	0.052693	9.053966

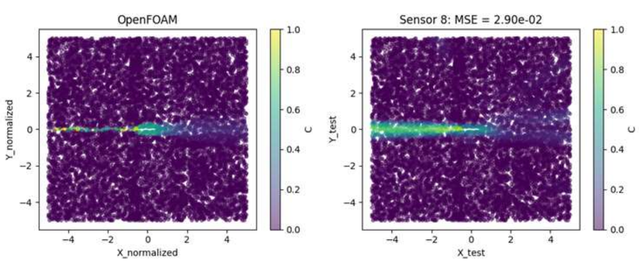
However the MSE for Sensor config 18 is lower than Sensor config 8. LAB provides a better measure

Model optimization

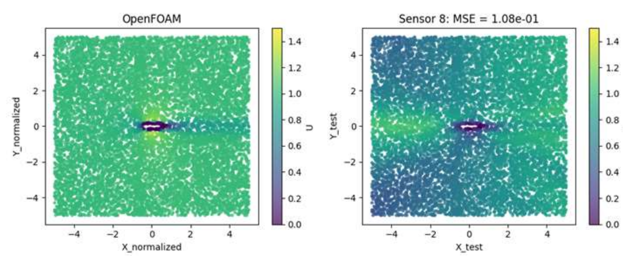
- Do grid search (a method to find best parameters) on 4 parameters:
  initialLearnRate, decayRate, numNeurons and numLayers
- Choose one candidate value for each parameter
  - 'initialLearnRate': [0.0005, 0.001],
  - ‘decayRate’: [0.0005, 0.001],
  - 'numNeurons': [64, 128, 256],
  - 'numLayers': [8, 12, 16, 20]
- Get 2*2*3*4=48 possible parameter combinations
  - No.1: initialLearnRate = 0.0005, decayRate = 0.0005, numNeurons = 64, numLayers = 8;
  - No 2: initialLearnRate = 0.0005, decayRate = 0.0005, numNeurons = 64, numLayers = 12
  - ……
- For each combination, run 2k epochs, use sensor configuration 3 as sensor data, randomly choose 10k as CFD data

How to choose candidates for each parameter

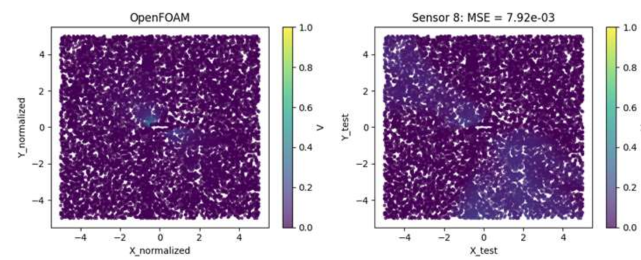
- Do basic optimizations while optimizing one parameter and keeping other parameters constant.
- Before optimization:
  - numlayers: 11
  - initialLearnRate: 10^-4
  - decay_rate: 0.0001
  - optimizer: Adam
  - batch_size: 1000
- openFoam and test plots before optimization:



c

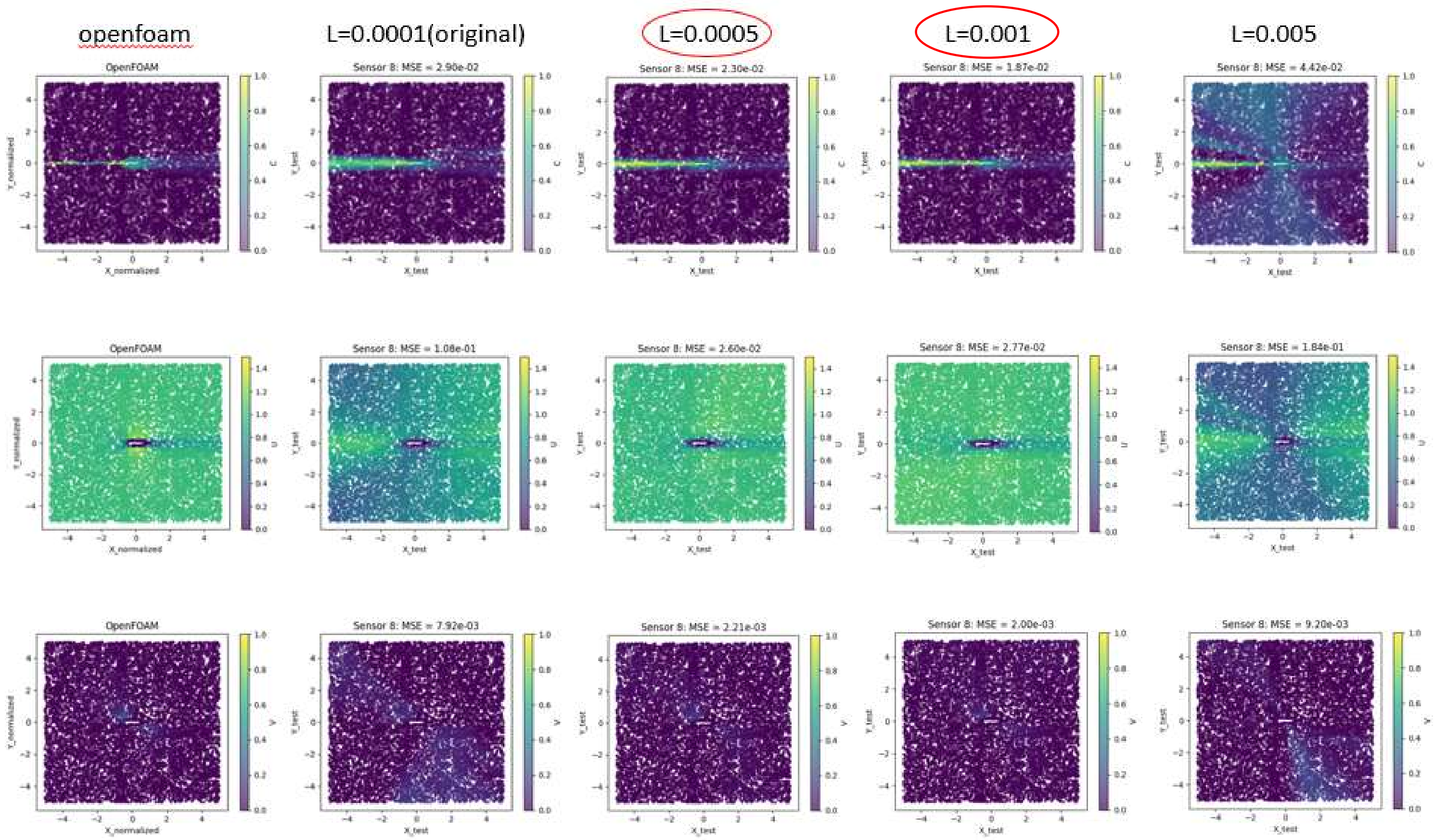


u



v

Example: only optimize learning_rate



- the model performs best when learning rate is 0.0005 and 0.001.
- in subsequent optimizations, we chose these two values as the candidate values for optimizing the parameters.

Parameters matrix

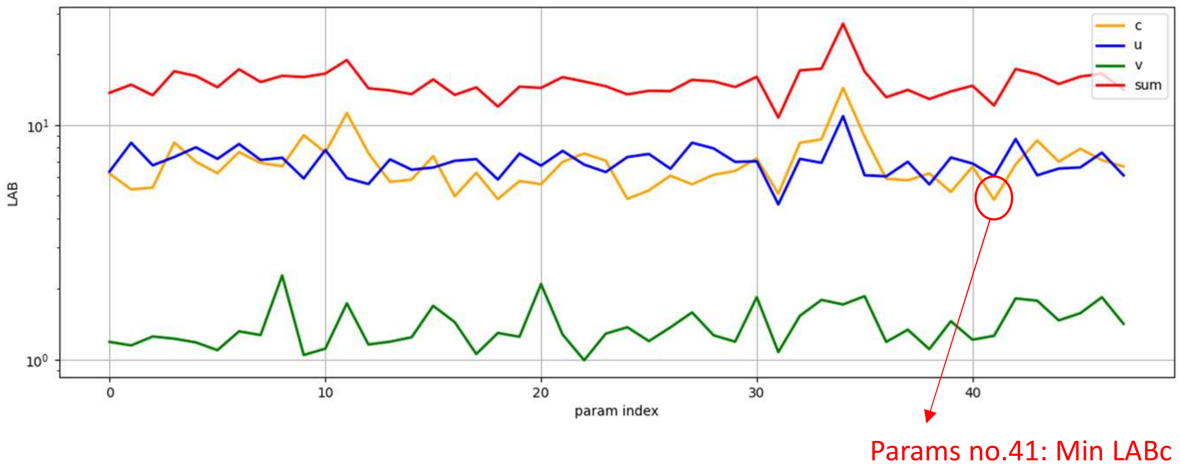
48 combinations in total

Params no.41: Min LABc

optimization

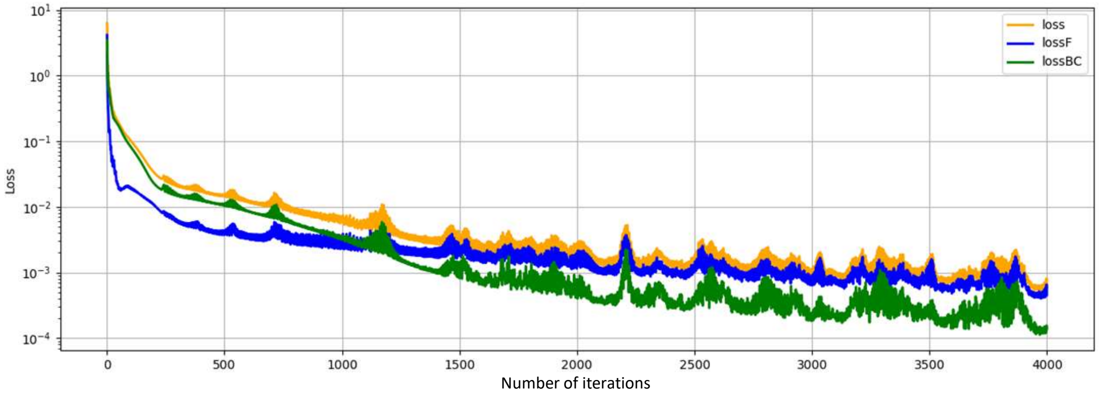
LABc	LABu	LABv	LAB_sum	param
6.20295988878797	6.3373194489428900	1.188631746431170	13.728911084162000	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 8, 'l2_regularizer': 0.01}
5.316474673811500	8.398679356316220	1.1487197919278200	14.863873822055500	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 12, 'l2_regularizer': 0.01}
5.409471965915670	6.735318775570120	1.253540170636090	13.39833091212190	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 16, 'l2_regularizer': 0.01}
8.422441683555540	7.309129119495430	1.2269652731153300	16.9585360761663	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 20, 'l2_regularizer': 0.01}
6.974807729831900	8.035911236354930	1.1831259601166300	16.19384492630350	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 8, 'l2_regularizer': 0.01}
6.230658354544630	7.175031259645030	1.095892765959700	14.501582380149300	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 12, 'l2_regularizer': 0.01}
7.667837642338380	8.300892666362460	1.3187560136466000	17.287486322347400	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 16, 'l2_regularizer': 0.01}
6.892923815711000	7.09395800367604	1.2715081137593200	15.258389933146400	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 20, 'l2_regularizer': 0.01}
6.674393154306540	7.254890281333810	2.281608246067770	16.210891681708100	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 8, 'l2_regularizer': 0.01}
9.058464878237780	5.917532841247250	1.0452267460397900	16.021224465524800	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 12, 'l2_regularizer': 0.01}
7.631127156352910	7.827667232981400	1.112628286985990	16.5714226763203	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 16, 'l2_regularizer': 0.01}
11.257795635381800	5.938416508318940	1.7363931682882700	18.932605311989	{'initialLearnRate': 0.0005, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 20, 'l2_regularizer': 0.01}
7.586737846707310	5.602935815160600	1.159079220245470	14.348752882113400	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 8, 'l2_regularizer': 0.01}
5.723168847096460	7.137094173990160	1.1914243305582500	14.051687351644900	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 12, 'l2_regularizer': 0.01}
5.854882360432510	6.450559427486100	1.2442199281464700	13.549661716065100	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 16, 'l2_regularizer': 0.01}
7.369041961682470	6.589426021575510	1.695024432582850	15.653492415840800	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 20, 'l2_regularizer': 0.01}
4.967914278851520	7.038157050224870	1.4428777949865200	13.44894912406290	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 8, 'l2_regularizer': 0.01}
6.257865110613410	7.155954603987130	1.0556311358082500	14.469450850408800	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 12, 'l2_regularizer': 0.01}
4.832875632244230	5.8553030456989800	1.2979285886827100	11.986107266625900	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 16, 'l2_regularizer': 0.01}
5.768122819609500	7.556048591533830	1.2503186545324500	14.574490065675800	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 20, 'l2_regularizer': 0.01}
5.595760333839160	6.70982323706249	2.0980944275704800	14.403677998472100	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 8, 'l2_regularizer': 0.01}
6.966899199487600	7.759685277742970	1.278648837717180	16.005233314947800	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 12, 'l2_regularizer': 0.01}
7.562298696114880	6.779055097882690	0.9925916781514610	15.333945472149000	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 16, 'l2_regularizer': 0.01}
7.045634275439020	6.304095622203450	1.2890510950576700	14.638780992700100	{'initialLearnRate': 0.0005, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 20, 'l2_regularizer': 0.01}
4.8418286744639400	7.302392411313240	1.3731257051223900	13.517346790899600	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 8, 'l2_regularizer': 0.01}
5.2671170093174	7.529256106828880	1.197991277544560	13.99436439369080	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 12, 'l2_regularizer': 0.01}
6.082135445492520	6.510781035893220	1.3715825306987600	13.964499012084500	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 16, 'l2_regularizer': 0.01}
5.589681191880800	8.406918488250390	1.588354187849770	15.584953867981000	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 64, 'numLayers': 20, 'l2_regularizer': 0.01}
6.13536893339695	7.950592454078830	1.268462983844950	15.354424371320700	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 8, 'l2_regularizer': 0.01}
6.370026924556800	6.963418158188610	1.1913082174276300	14.524753300173000	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 12, 'l2_regularizer': 0.01}
7.209895690829060	7.004410410327620	1.8452496650936900	16.059555766250400	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 16, 'l2_regularizer': 0.01}
5.0938159022017200	4.587680192123540	1.0757709866668000	10.757267080992100	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 128, 'numLayers': 20, 'l2_regularizer': 0.01}
8.39527370338769	7.174237682057080	1.536120092776150	17.105631478220900	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 8, 'l2_regularizer': 0.01}
8.692408314311510	6.900813572215480	1.795875884032470	17.389097770559500	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 12, 'l2_regularizer': 0.01}
14.405338380794000	10.919055169588600	1.7169540036500300	27.041347554032600	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 16, 'l2_regularizer': 0.01}
8.931943771679430	6.107339393171980	1.8629790204224100	16.90226218527380	{'initialLearnRate': 0.001, 'decayRate': 0.0005, 'numNeurons': 256, 'numLayers': 20, 'l2_regularizer': 0.01}
5.905186740719480	6.043330009319470	1.1887924064582400	13.13730915649720	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 8, 'l2_regularizer': 0.01}
5.809610969883030	6.970655321082400	1.3401474264434600	14.120413717408900	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 12, 'l2_regularizer': 0.01}
6.221803231582470	5.583753708677660	1.1072368274476100	12.912793767707700	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 16, 'l2_regularizer': 0.01}
5.1877189927068500	7.269365897786520	1.4566125105959000	13.913697401089300	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 64, 'numLayers': 20, 'l2_regularizer': 0.01}
6.639391880675560	6.852662260974420	1.2149559852533500	14.707010126903300	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 8, 'l2_regularizer': 0.01}
4.803653331364180	6.055593463237610	1.2619094936904200	12.121156288292200	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 12, 'l2_regularizer': 0.01}
6.7936984022758300	8.716929262040460	1.8221358684375900	17.332763532753900	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 16, 'l2_regularizer': 0.01}
8.591904097462360	6.105983591460230	1.7822557680750600	16.48014345699760	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 20, 'l2_regularizer': 0.01}
6.978147655901990	6.522867024665960	1.4683837373984500	14.9693984179664	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 8, 'l2_regularizer': 0.01}
7.926349687562960	6.595575063797560	1.576685352393860	16.098610103754400	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 12, 'l2_regularizer': 0.01}
7.089424137568450	7.624429135165880	1.8438754008118300	16.55772867354620	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 16, 'l2_regularizer': 0.01}
6.6565195274845700	6.100933513866320	1.420129395513530	14.177582436864400	{'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 256, 'numLayers': 20, 'l2_regularizer': 0.01}

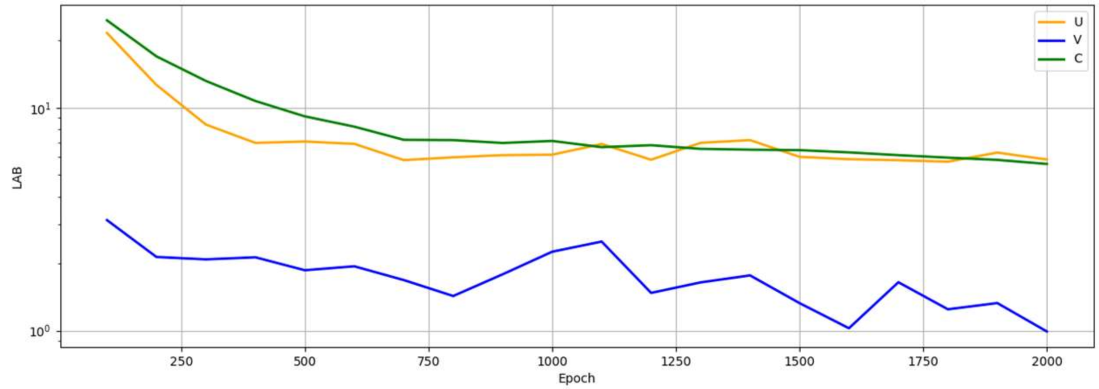
Model optimization



- The horizontal axis represents the combination number, while the vertical axis represents the LAB values.
- Focus on concentration, find params which has **min LABc (combination no.41)**
- {'initialLearnRate': 0.001, 'decayRate': 0.001, 'numNeurons': 128, 'numLayers': 12}
- minLABc: 4.803653

Train model

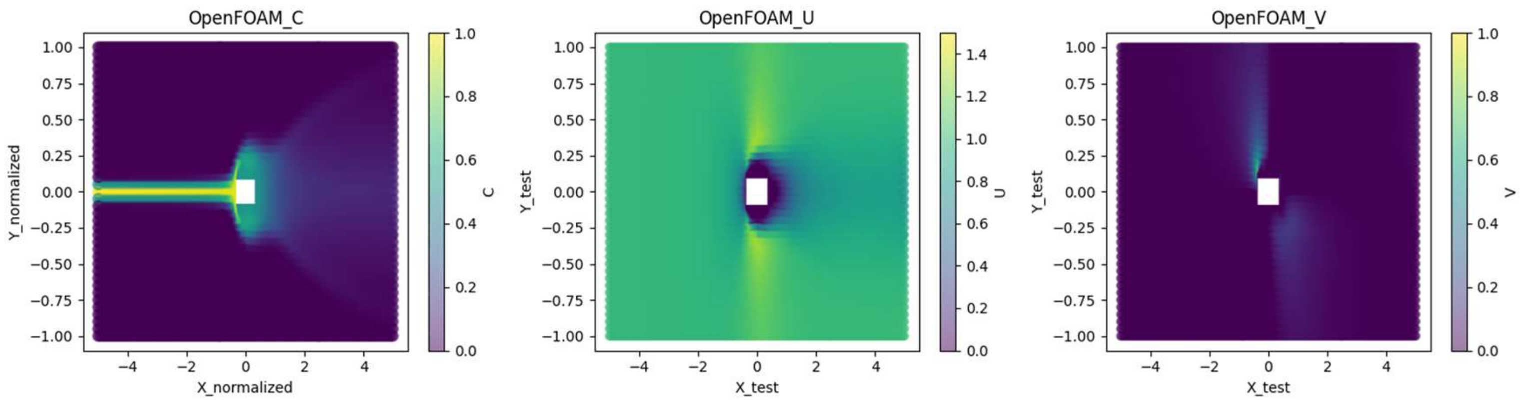




- Using 3 sensors and run each model for 2k epochs (enough for convergence and prevent over-fitting)
- Loss and LAB plot using sensor configuration 3

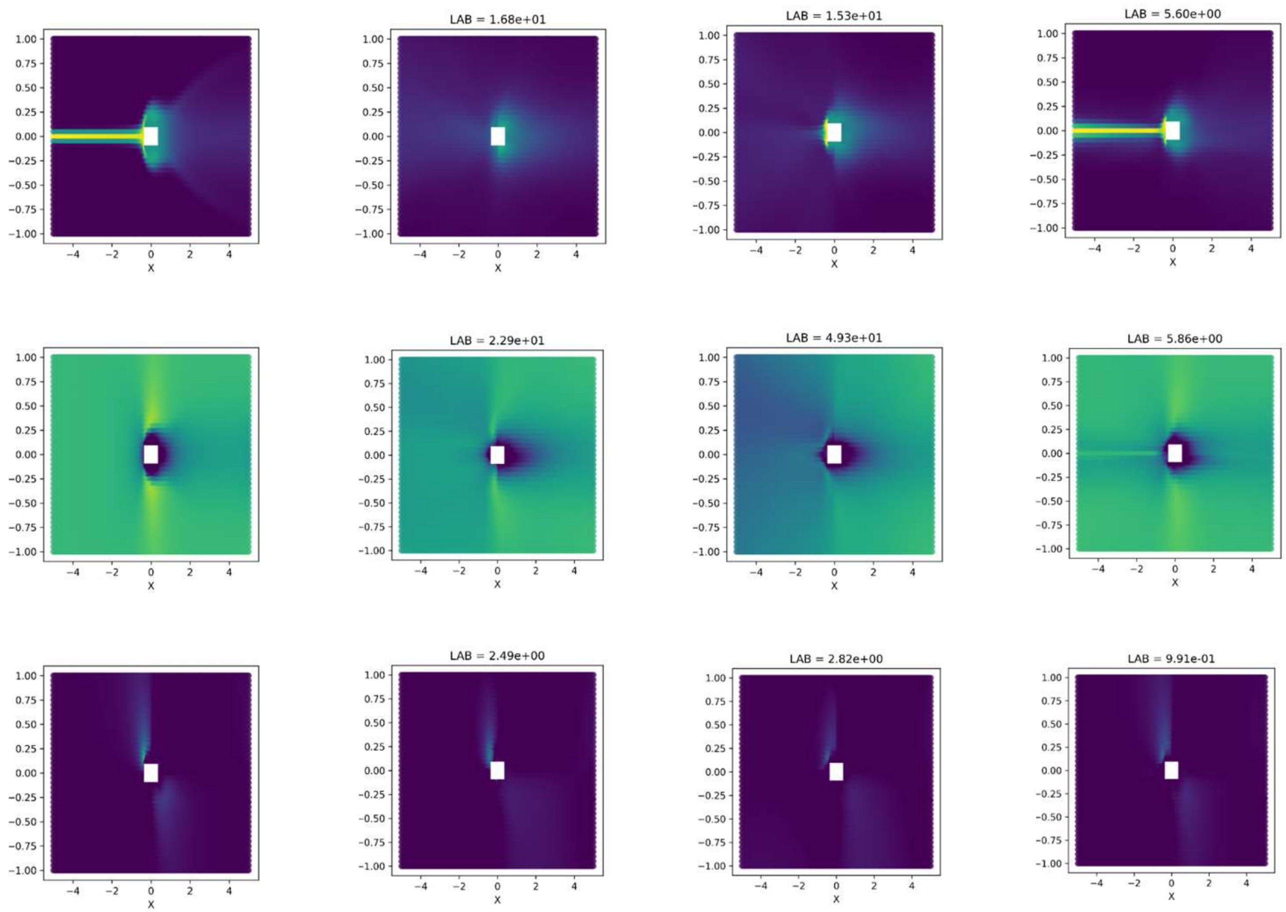
Test model

- Using test data to predict model.
- Test data: within the range of -1<y<1 segment of the entire CFD dataset, which have fewer 0 and more valid data as our aim is to predict the contour plot of the gas
- test data may have data which is not used for training which can prevent overfitting problem (CFD randomly choose from entire dataset, not from -1<y<1 segment)



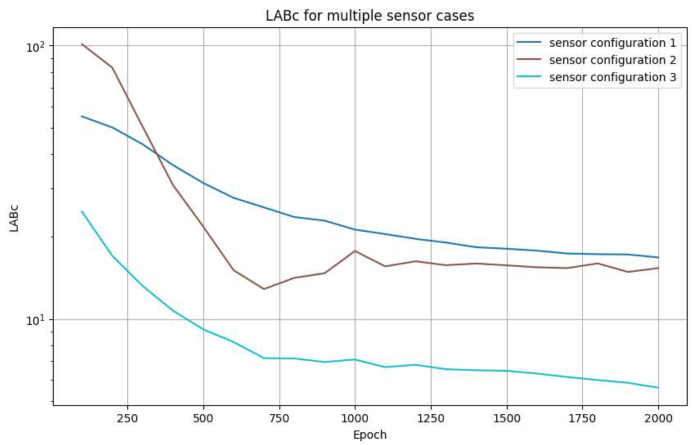
Test model on sensor configurations (1,2,3)

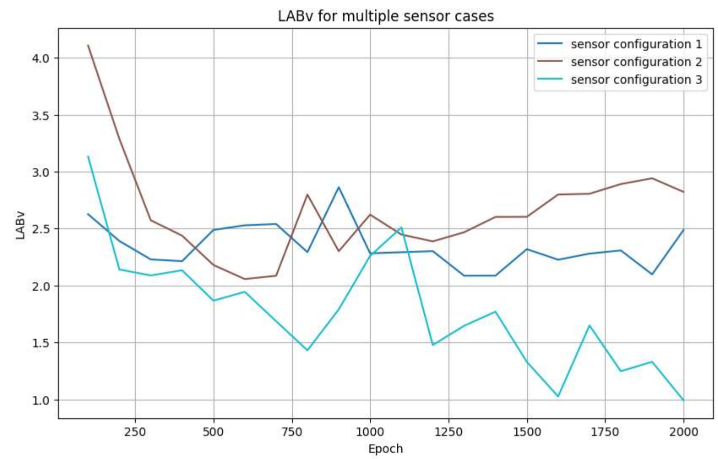
Sensor configuration 1 | Sensor configuration 2 | Sensor configuration 3

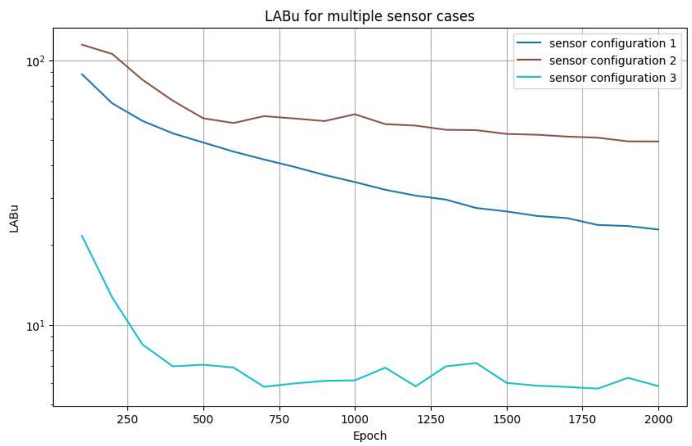


- first column: openFoam plots, true values
- the remaining three columns: predicted values using sensor configuration 1,2,3
- Focus on concentration, so plot c is what we concern
- Performs good when using sensor configuration 3, which means choosing the right location to place sensors is crucial to model performance

LAB for sensor configurations (1,2,3)



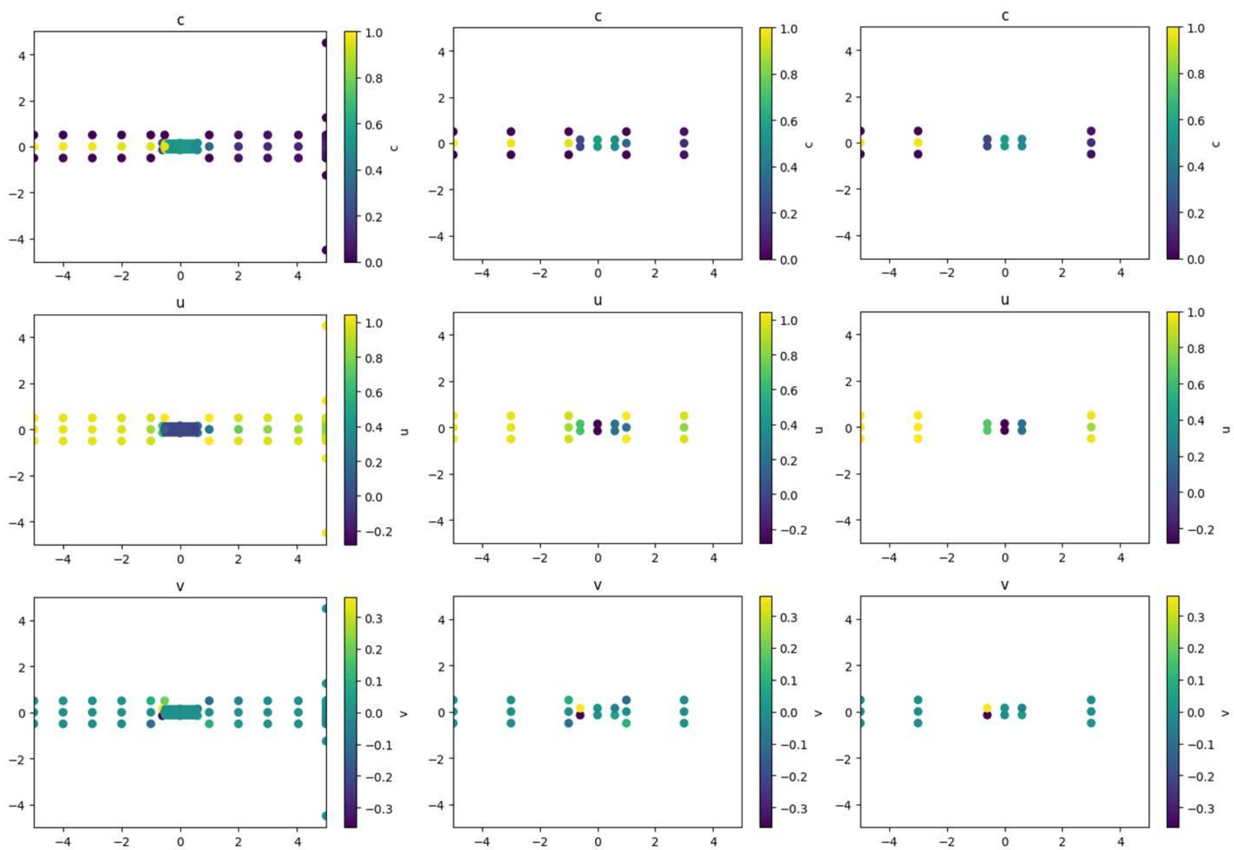




- Sensor 3 has the lowest LAB for c,u and v
- the closest predicted images compared to the OpenFOAM images
- LAB is a good metrics for this project.

Try reducing sensor

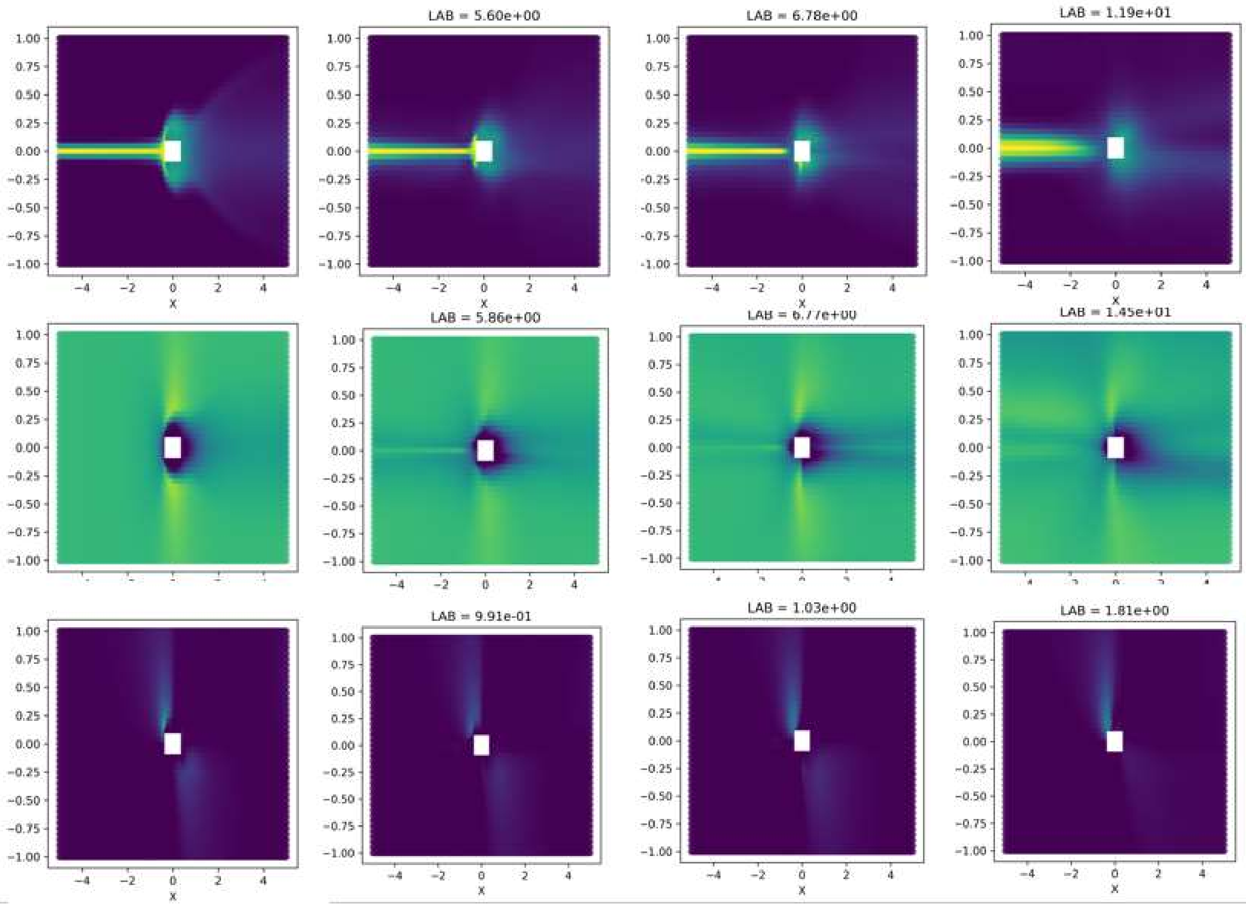
Sensor configuration 3 | Sensor configuration 4 | Sensor configuration 5



- find the sensor placement rules as well as try to minimize the usage of sensors as much as possible as it can reduce costs
- sensor configuration 3 is the benchmark
- For sensor configuration 4, we remove the long strip sensor in the right side, and make it sparse around the obstacle as well as along the path of the gas.
- For sensor configuration 5, we remove more sensors around the obstacle.

Test model on sensor configurations (3,4,5)

Sensor configuration 3 | Sensor configuration 4 | Sensor configuration 5



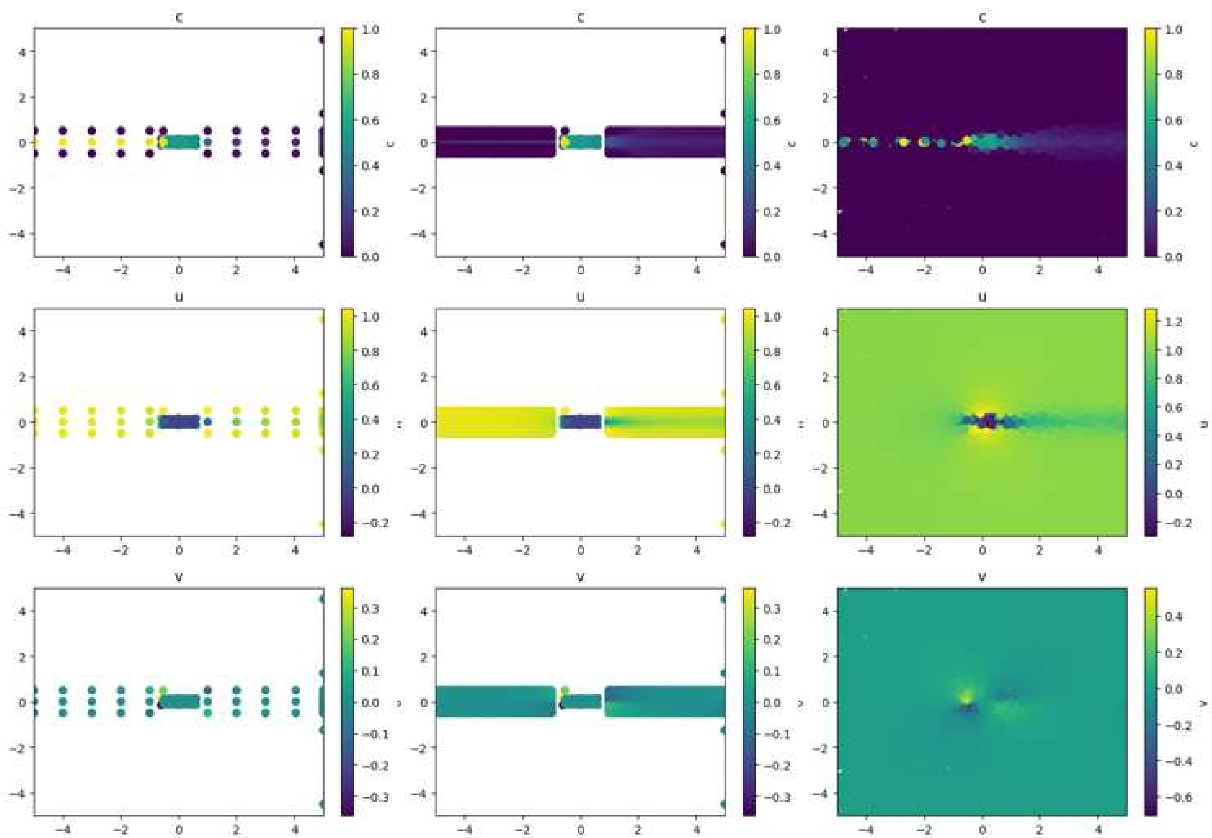
- sensor 4 still have a good performance on concentration
- sensor 5 can’t get good results for this project.

Sensor placement rules

- sensors placement should primarily be concentrated along the path of contamination diffusion (which would depend on gas release location and wind direction) and around obstacles.
- sparse placement, when appropriately implemented, does not compromise accuracy while reducing costs.
- it is essential to avoid excessive sparsity around obstacles.

Try adding sensors

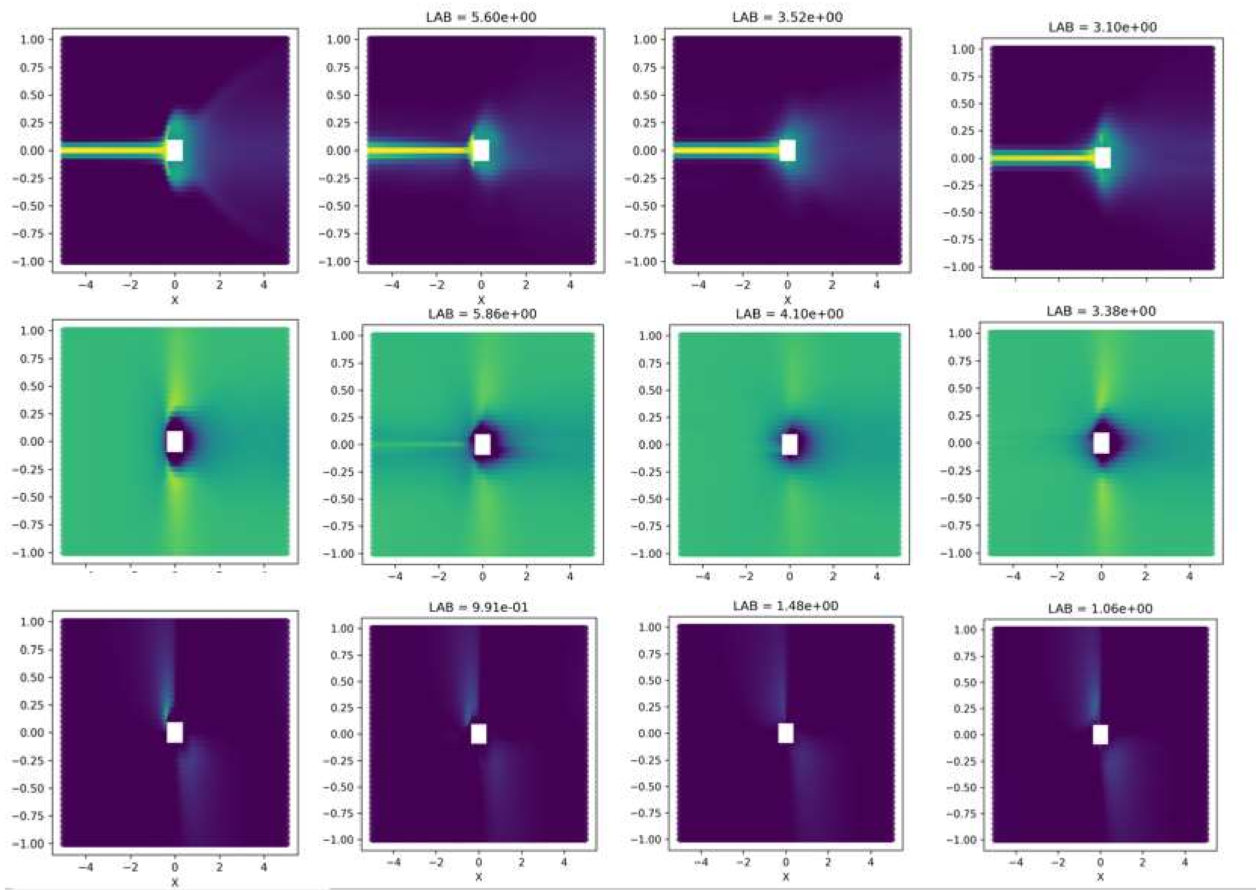
Sensor configuration 3 | Sensor configuration 6 (3.7k) | Sensor configuration 7 (10k)



- investigated the impact of the number of sensors on the convergence speed of the model
- Sensor 6: increased the number of sensors within the gas path range
- Sensor 7: randomly choose 10k from CFD
- Stop when LABc<4.5 last for 30 consecutive epochs
- Why 4.5: Based on extensive prior experiments, when LABc is less than or equal to 4.5, the prediction results for c are quite good.

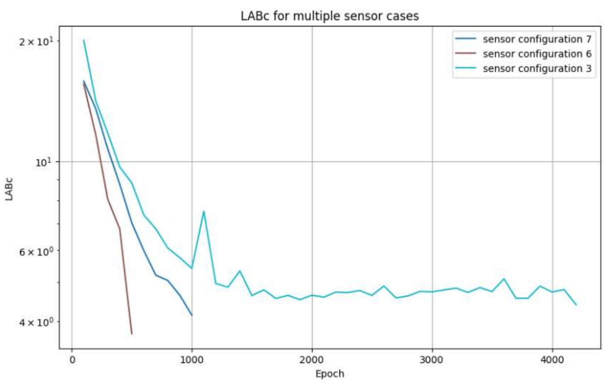
Test model on sensor configurations (3,6,7)

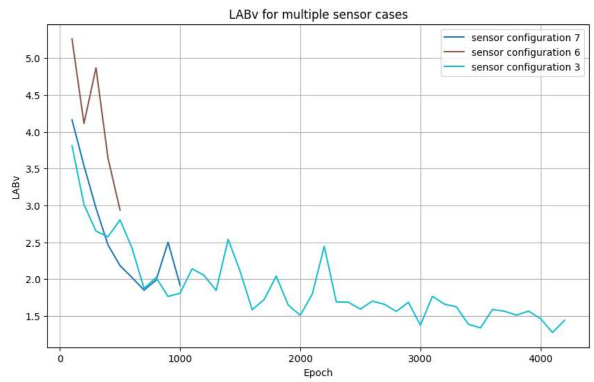
Sensor configuration 3 Sensor configuration 6 (3.7k) Sensor configuration 7 (10k)

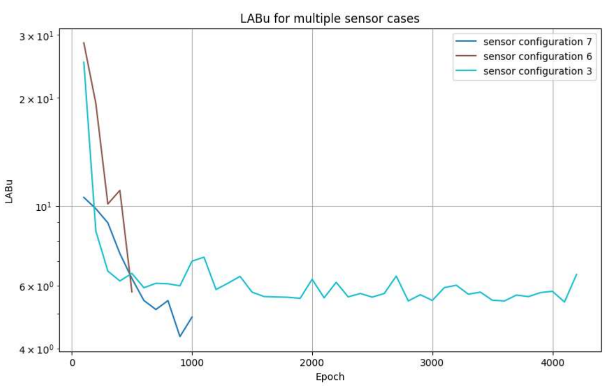


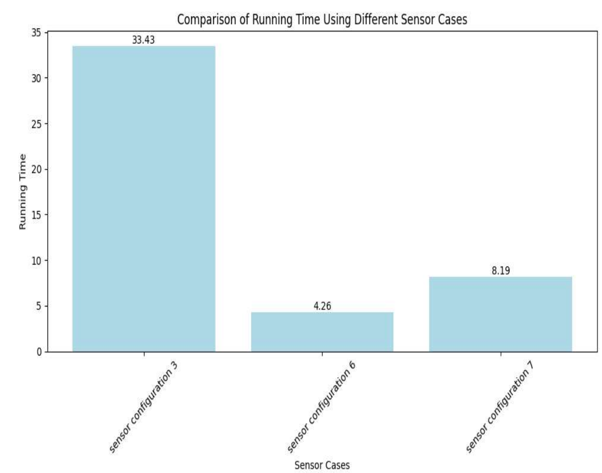
all 3 sensor configurations can have good performance.

Compare convergence time





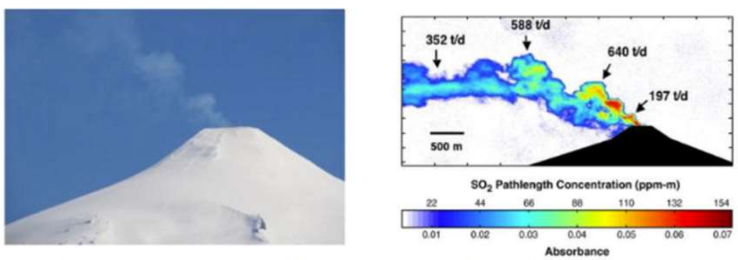




- Sensor 6 has the fastest convergence speed, indicating that appropriately increasing sensors can enhance model velocity, although increasing the cost.
- sensor 7 run longer than sensor 6, because excessive number of sensors cause heightened computational pressure, thus slow down the speed.
- Balance has also to be strike as point sensors can be expensive to install and maintain

Future work

- investigate whether the sensor placement rules we found are applicable to other 2D cases, such as shifting or rotating rectangular obstacles.
- Can we identify a universal sensor placement location for all 2D cases?
- extend the problem to 3D domain and incorporate time series data
- Consider other types of sensors other than point sensors (gas cameras that can create 2D concentration map)



A visible picture of a volcano in Chile is shown above left, followed by a false-colour representation of SO_2 concentration on the right. This image exemplifies the result tha will be obtained in real-time from the SO_2 camera.
Image source: "Development of an ultra-violet digital camera for volcanic SO_2 imaging" by G.J.S. Bluth, J.M. Shannon, I.M. Watson, A.J. Prata, and V.J. Realmuto.

From: https://resonance.on.ca/gas_camera.htm