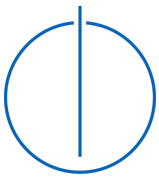
FAKULTÄT FÜR INFORMATIK

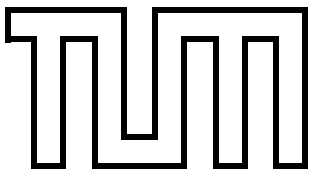
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

Perceptual Losses for Deep Learning on Fluid Simulations

Hanfeng Wu





FAKULTÄT FÜR INFORMATIK

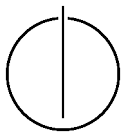
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

Perceptual Losses for Deep Learning on Fluid Simulations

Perceptual Losses für Deep Learning von Flüssigkeitssimulationen

Author:	Hanfeng Wu
Supervisor:	Prof. Dr. Nils Thürey
Advisor:	M.Sc. Georg Kohl
Date:	15.09.2021



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2021 Hanfeng Wu

Acknowledgements

Abstract

This thesis studies the integration of perceptual loss into several models that are related to fluid simulation. Perceptual losses are used to compare high level differences, while traditional loss functions like MSE and MAE only compare the pixel level differences which is more brute force.

Now some famous perceptual loss functions like comparing the intermediate layers of pretrained vgg network already improve its performance in some image related tasks. [JAFF16] We want to show that in the context of fluid simulation, the integration of some pretrained perceptual losses should also outperform the traditional loss functions.

We tested some fluid simulation tasks in SOL [UBF+20] to improve the performance of the percetual loss functions. Meanwhile we also integrate such loss functions into models like autoencoder and superresolution to compare their results with those from the same models but trained by MSE loss functions

Finally we combined the result of SOL and superresolution model to conduct some fluid simulations at a lower cost to show the advantage of integrating perceptual losses in such tasks.

Zusammenfassung

Diese Dissertation untersucht die Integration von Wahrnehmungsverlusten in verschiedene Modelle, die sich auf die Fluidsimulation beziehen. Wahrnehmungsverluste werden verwendet, um Unterschiede auf hohem Niveau zu vergleichen, während herkömmliche Verlustfunktionen wie MSE und MAE nur die Pixelniveauunterschiede vergleichen, was roher ist.

Einige berühmte Wahrnehmungsverlustfunktionen wie der Vergleich der Zwischenschichten eines vortrainierten VGG-Netzwerks verbessern bereits seine Leistung bei einigen bildbezogenen Aufgaben. [JAFF16] Wir wollen zeigen, dass die Integration einiger vortrainierter Wahrnehmungsverluste im Kontext der Fluidsimulation übertreffen auch die traditionellen Verlustfunktionen.

Wir haben einige Fluidsimulationsaufgaben in SOL [UBF+20] getestet, um die Leistung der Percetual Loss Functions zu verbessern. Inzwischen integrieren wir solche Verlustfunktionen auch in Modelle wie Autoencoder und Superresolution, um deren Ergebnisse mit denen aus den gleichen Modellen zu vergleichen, die jedoch mit MSE-Verlustfunktionen trainiert wurden

Schließlich haben wir die Ergebnisse von SOL und Superauflösungsmodell kombiniert, um einige Fluidsimulationen zu geringeren Kosten durchzuführen, um den Vorteil der Integration von Wahrnehmungsverlusten in solchen Aufgaben zu zeigen. *Note: Insert the German translation of the English abstract here.*

Contents

SOL Solver in the Loop

LSIM Learning Similarity Metrics for Numerical Simulations

Chapter 1

Introductions

Our main purpose of this paper is to demonstrate the advantage of adopting the perceptual loss functions in various deep learning tasks on fluid simulation. We try to show statistically and perceptually that by integrating suitable perceptual loss functions in such tasks can outperform the original model trained on traditional loss functions such as MSE and MAE. In the end we will also compare the performance and outputs of running solely the simulation and the simulation combined with some deep learning models.

1.1 Fluid simulation

1.1.1 Navier Stokes Equation

The state of art Fluid simulation is based on the famous incompressible equation Navier-Stokes equation

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \tag{1.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{1.2}$$

where ∇ denotes the gradient, $\nabla \cdot$ denotes the Divergence, $\vec{u}$ denotes the velocity of the fluid, t denotes the time step, ρ denotes the density of the fluid, p denotes the pressure, $\vec{g}$ denotes the gravity and ν denotes the viscosity of the fluid.

The equation (1.1) is actually a transformation of the $\vec{F} = m\vec{a}$ and the equation (1.2) describes the incompressibility of the fluid.

1.1.2 Grid Stucture

The way of storing the velocity and the density is based on two different grid structure. We store the dencity in the center of each cell, we call it centeredgrid or scalar grid. However for storing the velocity, we sample them at the face centers of each cell [HW65]. In such way we can more easily compute the inflow and outflow of each grid for each direction, however as a trade off, the data format would be more complex than normal centeredgrid, and we usually have to stack each verlocity array of each dimension together and also adopt them individually in some computations(e.g. computing loss function in deep learning)

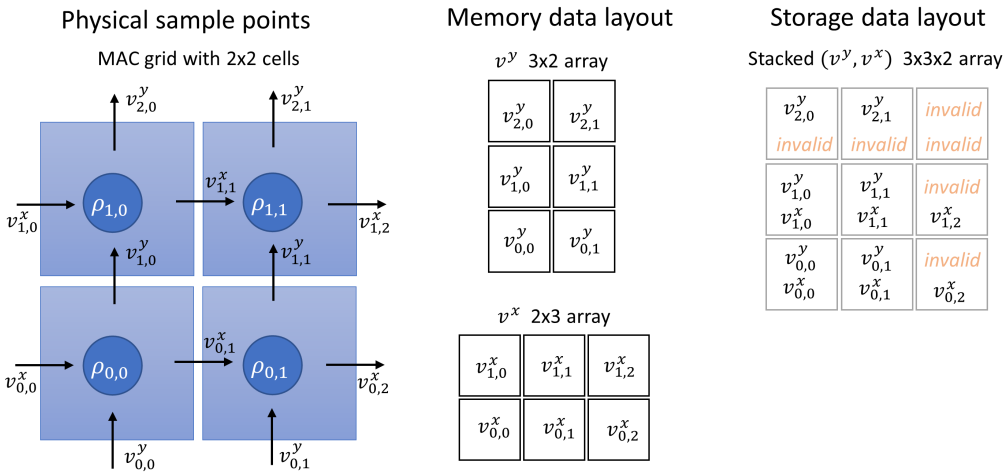


Figure 1.1: Staggered grid format

1.1.3 Advection

With the knowledge base of the NS-equation and grid structures we now need the advection algorithm to run the simulation. We adopt the semi-lagrangian advection algorithm. In the lagrangian point of view, if we want to compute the grid value of the n+1 time step of grid position $\vec{x_G}$, we should figure out which particle actually flows to the grid $\vec{x_G}$ from time step n+1. If we name the grid position of the particle as $\vec{x_P}$, then the new particle at time step n+1 in $\vec{x_G}$ should have the same physical property as the particle at time step n in $\vec{x_P}$

we first use the euler equation to calculate the $\vec{x_P}$ as a step before of $\vec{x_G}$

$$\vec{x}_P = \vec{x}_G - \Delta t \frac{d\vec{x}_G}{dt} \tag{1.3}$$

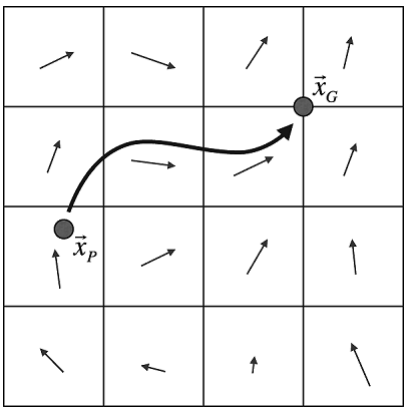


Figure 1.2: advection

because $\frac{d\vec{x}}{dt} = \vec{u}(\vec{x})$ we have

$$\vec{x}_P = \vec{x}_G - \Delta t \vec{u}(\vec{x}_G) \tag{1.4}$$

where $\vec{u}(\vec{x}_G)$ denotes the velocity sampled at the position x_G, with the advection we can update the velocity, pressure and density of the whole vector field.

with all the knowledge above, we can formulate a sequence to conduct the basic fluid simulation.

- start with an initial velocity field $\vec{u}^0$ with the property $\nabla \cdot \vec{u} = 0$
- choose a time step Δt
- For time step n = 0, 1, 2, ...
  - $\vec{u}^A = advect(\vec{u}^n, \vec{u}^n, \Delta t)$
  - add the extra force $\vec{u}^B = \vec{u}^A + \Delta t \vec{g}$
  - $\vec{u}^{n+1} = make\_incompressible(\Delta t, \vec{u}^B)$

1.2 Deep learning tasks

There are also various ways of integrating deep learning models into the fluid simulation.

1.2.1 Autoencoder

The Autoencoder model has a bottle neck shape, which aims to produce the same output as the input. Once trained, the Autoencoder model can be saparated into two parts, namely the encoder and the decoder, which can

compress the image with the encoder and later decompress the image with the decoder to its forer shape.

1.2.2 Superresolution

With the superresolution model, we can upscale the low-resolution images to high-resolution images, which based on that we have fed lots of similar data to the model. In this paper, we apply the Superresolution model at the end of the Solver in the Loop model in order to boost the fluid simulation while still obtain a fairly similar output.

1.2.3 Solver in the Loop

When we want to perform our fluid simulation on a lower-resolution domain to decrease the time cost, we always suffer from the numerical error comparing to the simulation running on the higher-resolution domain. It is reasonable because with the same initial state, the vector field on the higher domain has richer information than the one sampled in the lower domain. Dr. Kiwon Um [UBF+20] developed a mechanisim to learn the difference between the 2 different domains to reduce the numerical error.

1.3 Perceptual loss functions

Perceptual loss functions are used when comparing two images, they are not like traditional loss functions like MSE or MAE which only computes the low level pixel differences. Perceptual losses on the other hand, are normally forward networks trained for specific tasks. For example, VGG-16 network was trained for massive image classification, and LSIM was trained on smoke and fluid simulations. When adopting those networks, we can either compute the mse loss between the intermediate layers of the trained network, or directly passing it through depending on the goal of the chosen perceptual loss network. These perceptual loss networks tries to retrieve the high level differences between the images, hence will usually have a better performance than the traditional loss functions. In a word, traditional loss functions have more general use cases, while perceptual loss functions will achieve better performance on specific tasks. In this paper we mainly focus on comparing the performance between traditional loss function MSE and perceptual loss functions LSIM, VGG-16 and lpips.

Chapter 2

Related Work

similar goal area methods

Chapter 3

Metrics and Perceptual Losses

In this chapter we will cover several loss functions that we adopted in this paper and briefly explain their characteristics and their drawbacks. More percise integrations and results will be convered in the chapter 5.

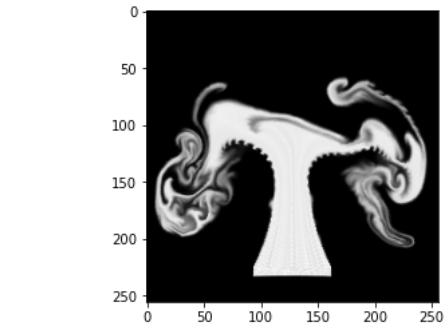
3.1 Mean Squre Error

MSE is one of the most widely used loss functions.

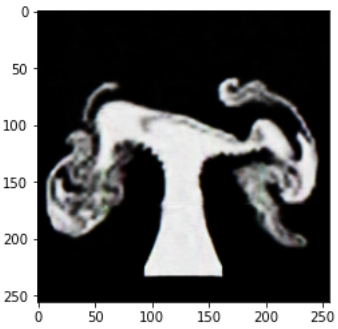
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.1}$$

Where Y is the reference tensor and $\hat{Y}$ is the predicted tensor based on the trainable parameters. With this loss function being adopted, the network will try to do the gradient decent on the weights in a way that can achieve minimum square error. The optimal case is that the predicted tensor $\hat{Y}$ is exactly the same is the reference tensor Y. We can see that the core idea for the MSE is very general so that this loss function can be applied almost for any models, however the MSE loss function is not suitable for some image related tasks. If the two inputs of the MSE loss function is merely a image with its core content shifted by some pixels, the MSE will produce a very high value, which is not representitive, in another word, the MSE does not have shift or rotate invariance.

Besides, MSE computes the squre error of the two tensors, which means that MSE is also prone to the outliers, beacause when it has a very large outlier in the tensor, the result of MSE loss function will be strongly influenced, so in most image construction cases, in order to achieve a lower square error, the output will be a relative blur one compared with the original.



(a) ref



(b) MSE output

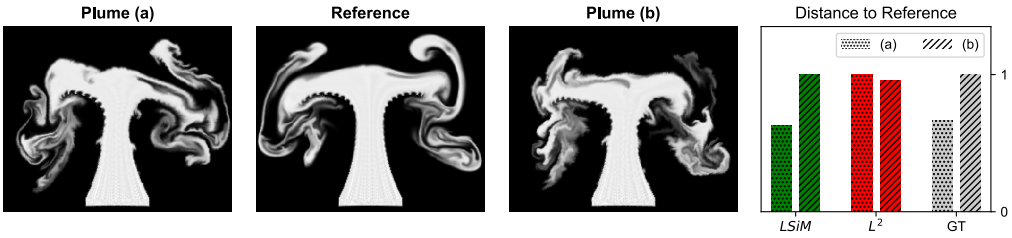


Figure 3.2: The loss computed between two different plume images and the reference with LSiM and L^2 metrics. While the GT is the ground truth distance determined by the data generation.

3.2 Learning Similarity Metrics for Numerical Simulations

LSiM [KUT20] is a neural network-based approach that computes a stable and generalizing metric. Due to the unreliability of the MSE loss in numerical simulation tasks, LSiM has been established. It aims to demonstrate the performance of CNN-based evaluations on numerical simulation tasks. The data used to train LSiM are generated with known partial differential equations (PDEs).

With figure 3.2, we can tell that the L^2 loss function consider plume b a closer image to the reference, while the fact is that plume a has a smaller distance to the reference image, which is exactly predicted by the LSiM.

The LSiM itself is also a CNN-based metric, see figure 3.3. The input of the network must be two batches of images with 3 channels

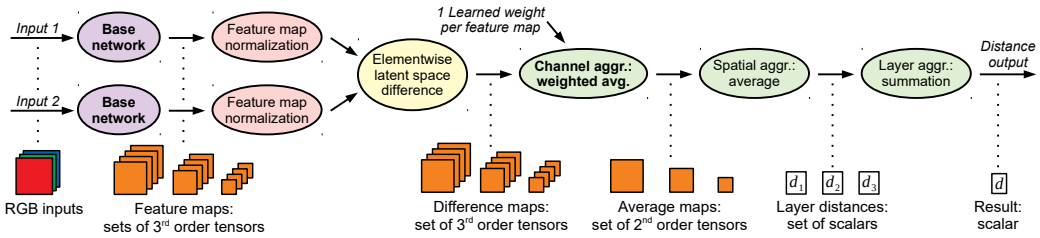


Figure 3.3: The Network structure of LSiM

As a metric, it also holds the metric properties $\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{I}$:

$$m(\boldsymbol{x}, \boldsymbol{y}) \geq 0 \qquad \text{non-negativity} \tag{3.2}$$

$$m(\boldsymbol{x}, \boldsymbol{y}) = m(\boldsymbol{y}, \boldsymbol{x}) \qquad \text{symmetry} \tag{3.3}$$

$$m(\boldsymbol{x}, \boldsymbol{y}) \leq m(\boldsymbol{x}, \boldsymbol{z}) + m(\boldsymbol{z}, \boldsymbol{y}) \qquad \text{triangle ineq.} \tag{3.4}$$

$$m(\boldsymbol{x}, \boldsymbol{y}) = 0 \iff \boldsymbol{x} = \boldsymbol{y} \qquad \text{identity of indisc.} \tag{3.5}$$

3.3 VGG-16 loss network

VGG-16 [SZ14] is originally a CNN-based network trained for image classification problems. The author proposed 6 versions of vgg networks with different numbers of CNN layers but applied 3 fully connected layers for all of them at the end to realize the classification functionality. We adopted in the paper only the pretrained vgg-16 network(see figure 3.4) and apply it as a perceptual loss function.This pretrained model achieves 92.7% top-5 test accuracy in ILSVRC-2014, which is challenge based on a dataset of over 14 million images belonging to 1000 classes. Although the networks was not originally trained for computing the distance, we can still use transfer learning to use it as a perceptual loss. In the paper [JAFF16], they chose to extract the third intermediate layer outputs and compute the MSE loss between them to realize the perceptual loss functionality(shown in figure 3.5)

Apartfrom that we have also tested extracting the first, second or the fourth intermediate layers to calculate the mse loss in order to compare the performance with the current method. We found out that computing mse between the third one has the best performance in our training task.

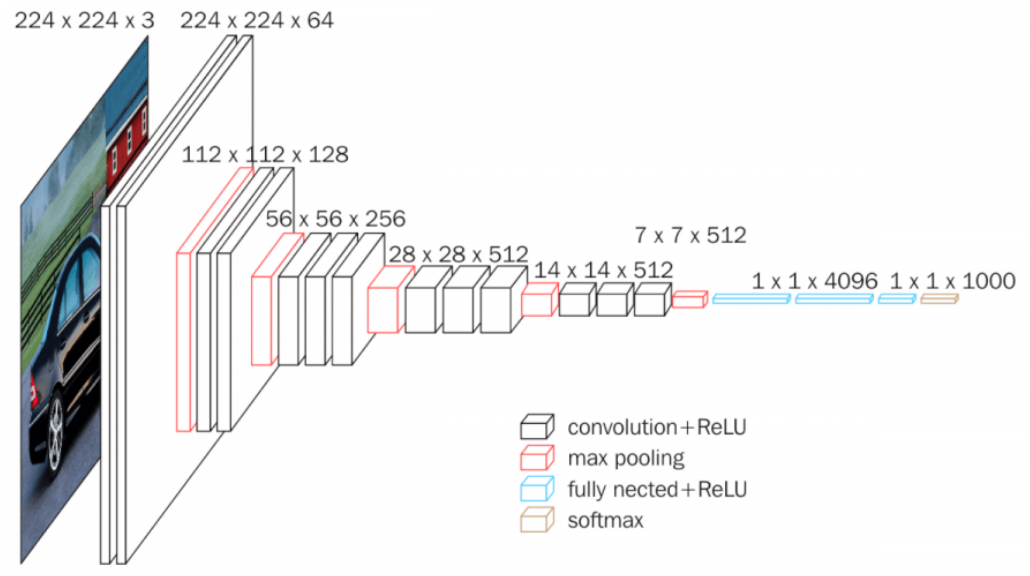


Figure 3.4: The network structure of vgg-16

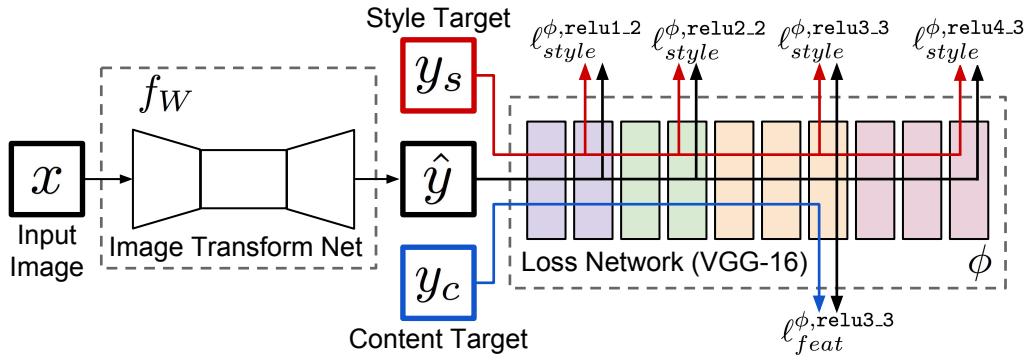


Figure 3.5: The system figure of their style transfer learning, we only focus on how they compute the loss function based on the vgg-16 network

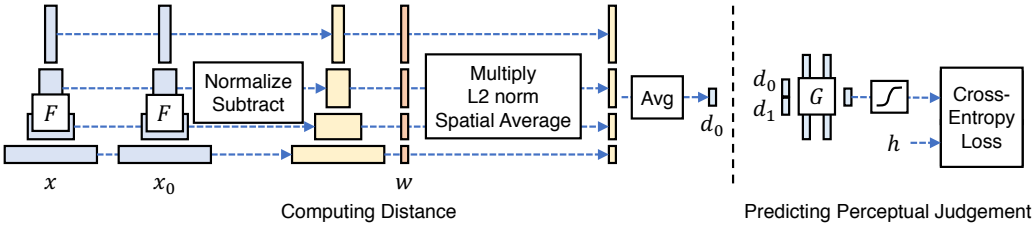


Figure 3.6: The lpips training setup: The distance is computed between the ground truth patch x and the distorted patch x_0 or x_1 based on the chosen network architecture. The perceptual distance is then trained on a small network G towards the Psychophysical Similarity Measurement h as the reference based on the Cross Entrophy Loss [ZIE+18]

3.4 Lpips

Lpips as in Learned Perceptual Image Patch Similarity [ZIE+18] is also a trained network to compare the high level image features based on various distortions. According to the author, they adopted distortions like CNN-based distortions, superresolution, frame interpolation and so on for the dataset. Based on that, they use several psychophysical similarity measurements on the dataset, trying to approach the human judgements to the images. For the network architecture, they evaluate the SqueezeNet, AlexNet, and VGG architectures. We in this project choose to use the AlexNet based lpips.

Chapter 4

Datasets

In this chapter we will cover the various dataset that we have adopted for training validation and testing in the following three tasks.

4.1 Autoencoder

We set up the Autoencoder mainly to have a rough overview of the performance of the perceptual loss functions. Hence, we choose to use the nuerical simulation data with the size of 256*256, on which the LSiM model was trained.

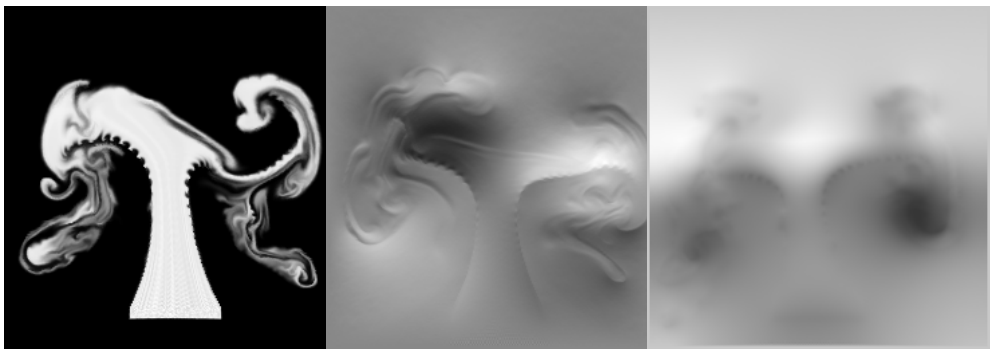
Autoencoder can be trained without the supervision, because the training data is exactly the reference, so we do not need to do any modification to complete the training. However, in order to boost the training we made 2 following modifications:

- since the numerical simulation data are mostly grayscale, which means we can sample them with one channel to boost our training and replicate the channel when certain loss functions demand their inputs in RBG channels. Such techniques can also be applied on any models with grayscale data, you can see more details about it in the superresolution apply loss function section.

- in order to reduce the GPU usage, we downsample the 256*256 training data to the size of 128*128

4.2 Superresolution

The data we used in the superresolution task came from various sources. We first used the training data of LSiM model, because these are all numerical



(a) density data **(b)** velocity data **(c)** pressure data

Figure 4.1: Training set of LSiM model

simulation data including the verlocity, density and pressure images, so we recon that LSiM can outperform the other metrics.

As we also want to test the performance of combining Solver in the loop and superresolution together to compete against the time cost with performing the simulation purely on the higher domain, we therefore also feed the training data of the Solver in the Loop to the superresolution model, so that our model can learn to super sampling the plume density images when we feed the corrected simulation images on the lower domain, and we are able to abtain the plume images with the same resolution as the higher domain.

Apart from that, we also test the our model performance on some random videos data which are not related to numerical simulations but still can be of some assistance to our model. To create the data set, we form a dictionary for each image. We first downsample each training image I_{high} to a lower resolution image $I_{downsampled}$(e.g. images of 256*256 to image of 64*64), then we resize the $I_{downsampled}$ to the same image size of the previous I_{high}, then we get a image I_{low} with the same image size but a lower resolution as the I_{high}. In the training process we treat the I_{low} as the input data of the model, I_{high} as the groud truth to compute the loss with the model output.

4.3 Solver in the Loop

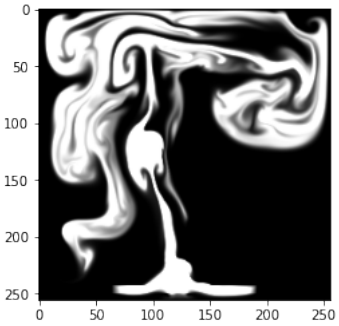
4.3.1 Simulation tasks

In the solver in the loop tasks, we have trained and tested out our model with 2 different fluid simulations, which are the 2-D karman flow and 2-D buoyancy driven flow. The author of the solver in the loop has mainly trained their model on the karmanflow, based on which we extend the model to a

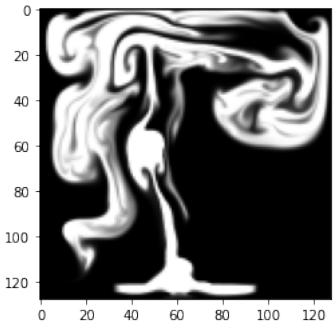
(a) Karmanflow example

(b) Buoyancy driven flow example

Figure 4.2: Two simulation examples for Solver in the Loop



(a) Buoyancy driven flow image with 256*256



(b) Downsampled buoyancy driven flow image with 128*128

Figure 4.3: Reference images generation and downsampling

more complex simulation i.e. the buoyancy driven flow.

For both simulation tasks, we have prepared two dataset with different resolutions in order to compare the performance of different loss functions on different resolutions. The two resolutions we chose for the Karmanflow are 128*64 and 64*32, and for buoyancy driven flow tasks we picked 64*64 and 128*128. As we want to reduce the numerical error for the model trained on a lower domain, for each resolution ratio we have to firstly generate a higher resolution dataset as the reference dataset, but we can not compare two datasets with different resolution therefore we have to downsample the higher resolution training set to a lower one that matches our training domain, and the downsampled dataset with rich simulation details from higher domain is our actual reference set.

4.3.2 Phiflow set up

The tool we used to perform the fluid simulation is Phiflow [HKT20].

For the karmanflow generation, we define an open domain with the size of 200(height) * 100(width). **(notice that domain size is not the same concept as the domain resolution. The domain size helps to locate the position of the inflow and obstacles of the simulation, while the resolution is used to define how percise we should sample our domain. Hence we could perform different resolution formats on the same domain size.)** we set the inflow as a 5 * 50 rectangle at the top center, and the obstacle as a circle with radius 10 positioned in the very center of the upper half of the domain. The step size Δt we chose here is 1.0. For each simulation we chose to generate 1500 frames for the karman flow.

For the buoyancy driven flow generation, we define a 200 * 200 closed domain and set the inflow as a 5 * 100 rectangle at the bottom center. We generate 250 frames for each simulation. Why we only chose 250 steps for the buoyancy driven flow is because we defined a closed boundary for the buoyancy flow so the simulation after 250 steps are extremely complex, and we also set the timestep to 0.1 to slow down the simulation so we will not have a chaotic frame of plumes. Which helps to make our training easier.

For both tasks, we have a total of 8 simulations with their renolds numbers ranging from $1 * 10^5$ to $8 * 10^5$ as the training set. As for the test set we can alter the renolds number and the seed of the simulation to generate a sets of simulation frames that are overall similar but in details deviated from the used training set.

Chapter 5

Tasks and Experiments Setup

5.1 Autoencoder

5.1.1 Network

We used a very simple bottle-neck like and fully convolutional neural network for the Autoencoder.

5.2 Superresolution

5.2.1 Network

We adopt the same ImageTranformNetwork as in the paper [JAFF16], which is a fully convolutional network that has the same output tensor as the input. The network consists of three Convlayers, three UpsampleConvlayers and five residual blocks.(Notice that we do not use the deconvolutinal layer in the UpsampleConvlayers as suggested in the paper [ODO16], that deconvolutional layers will likely result into checkerboards artifacts due to the overlapping when doing the reverse convolution operations. So we prefer to fistly do the traditional upsampling either with Nearest-neighbor interpolation or with

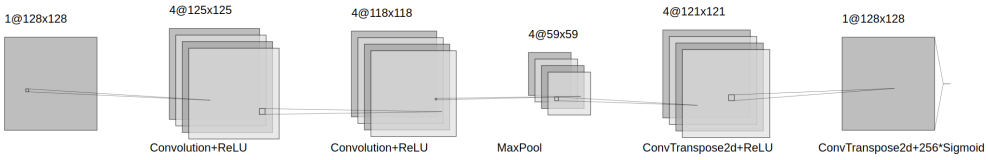


Figure 5.1: The structure of Autoencoder

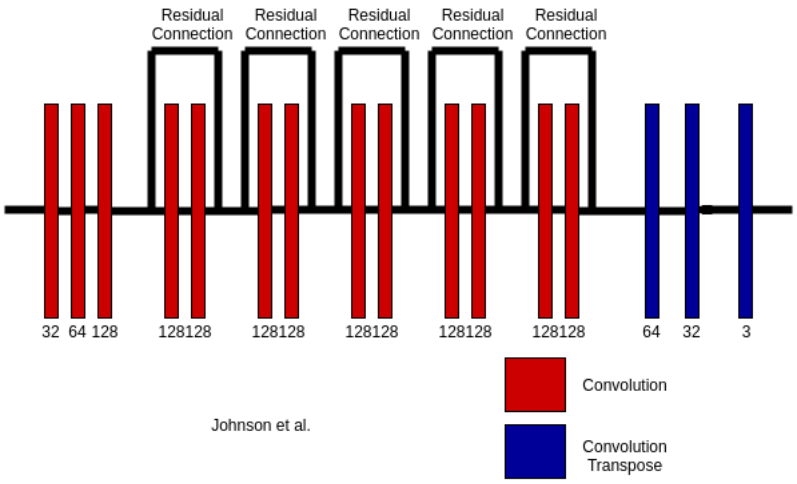


Figure 5.2: The structure of ImageTranformationNetwork of [JAFF16]

Bilinear interpolation then we apply the convlayer afterwards to obtain the expected upsampled size.)

Since the outputs of our model is RBG images, which means the tensor values lie in the range of [0, 255], so we apply at the end of the last upsampling layer a sigmoid function and times 256. In such way, we can ensure that our output tensor value won't exceed the range. Since the input tensor and the output tensor share the same shape, so before we feed the low resolution data to the model, we need to first do the upsampling with some interpolation methods(we choose bilinear here) to resize our input images to the expected high resolution, then feed them to the model.

5.2.2 Applying loss functions

Before feeding the input image to the model and compute the loss function, we can still add on some modification to make our training eaiser. Since most of our data are smoke images or verlocity fields, so they are in grey scale. In this case, we can modify our network in the way that the new input channel size and the output channel size are 1, because the grey scale images only have one channel. With this modification, our model does not need to learn to align all the three RBG channels to the same value, hence it will strongly reduce the training difficulty. However, in the process of applying the loss function, there will be the problem that many of the perceptual loss functions only accept RBG images that the input tensors should have three channels. In order to fix this, we choose to replicate the channel size to three with the same values, so the loss functions will regard our grey scale images

(a) Using deconvolution. Heavy checkerboard artifacts.

(b) Using resize-convolution. No checkerboard artifacts.

Figure 5.3: The comparison between two upsample methods [ODO16]

as RBG images. In this way we keep the tensors in the training process as one-channel tensors however in the loss function computation process as three-channels tensors, which increase our model performance.

In order to compute the loss functions, we need to define the reference and the model output. Now we know our first input is the model output with replicated channels, and the reference input should be the I_{high}. The rest is to define the respective metrics. For MSE loss we can directly compute it between two tensors. For vgg-16 loss functions, we can feed each tensor into the model and compute MSE loss between their third intermediate outputs. For LSiM we can form a dictionary of these two tensors and pass it to the forward function of the LSiM model. For lpips we can directly use the pretrained model and feed the two tensors into the function.

5.3 Solver in the Loop

After we have generated the training set and its downsampled version, we can start the training. We will first cover the basics of the network and its training goal, then we will go into details of how to apply the loss function.

5.3.1 Network

The training network we used is called MarsMoon, which is the same network that Prof. Um adopted in the his project(see figure 5.3) we can see that the

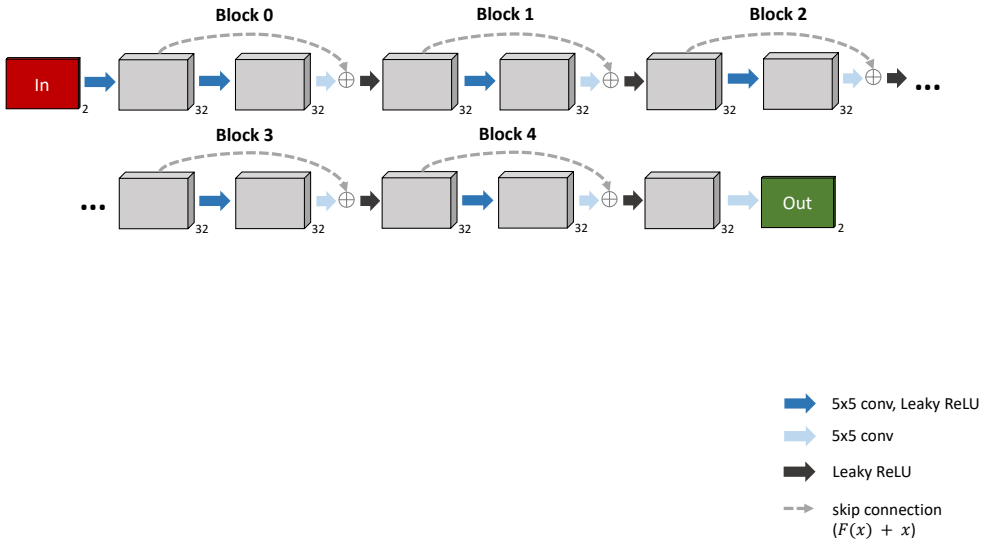


Figure 5.4: The network architecture [UBF+20]

number of channels for both input and output is 2. This is because the input is the staggered tensor of the verlocity which has two directions one represents the y direction and the other represents the x direction. The output of the network is also a staggered tensor however it can not be directly addressed as the verlocity but a correction of the verlocity, which can be added to the verlocity calculated on the lower domain, then the corrected verlocity can be applied for the next simulation step. The reason why we do not train the output towards the verlocity for the next simulation step is because we want to take advantage of the physical information stored in those tensors, if we should simply get the next verlocity from passing the former verlocity through the nerwork, then we would not need to perform the physical simulation, which is very unstable. Thus, adding the trained correction to the physical simulation output of the former verlocity would at least make sure that our simulation is performing on the right track. That means our consective steps of the simulation are based on valid physical simulation, which can not be ensured by purely passing through the neural network.

5.3.2 Training Set

In karman flow simulation, we choose not to use the whole training set to train our model, as in the first several steps of karman flow do not have much difference between high and low domains. So we only try to train the last 1000 steps of the total 1500 steps. When we train the lower domain

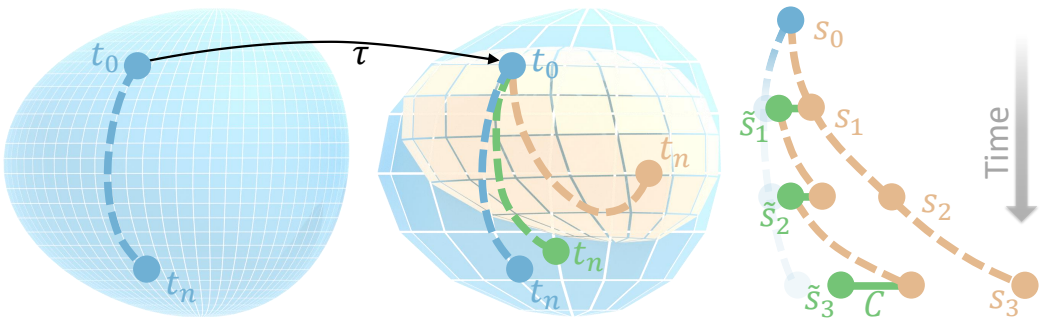
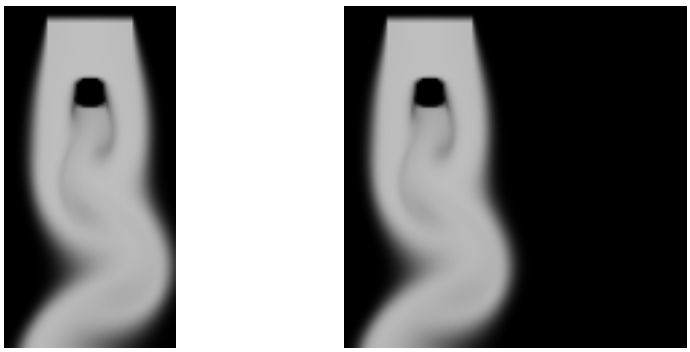


Figure 5.5: The training goal is to obtain the suitable correction, with which the simulation on the lower domain can reduce the numerical error. The brown one represents the source simulation of the lower domain without any corrections, the blue one represents the simulation on the higher domain which is our target, the green one is the brown one plus the trained correction, which aims to apporach the blue one [UBF+20]



(a) **(b)**

Figure 5.6: Padding plume image to match the input size of LSiM model

64*32 karmanflow, we also find out that the 64*32 image for LSiM model is too small, so we decide to alter the domain to the resolution of 64*64, correspondingly the training set also need to be retrained on the domain of 128*128. However the size of the plume remains unchanged, we only pad the domain such that they can be applied to LSiM loss function.(see figure 5.3) For the buoyancy driven flow simulation, it only consists of 250 steps and each step has rich numerical information, so we regard the whole simulation as the training set. Besides, since the buoyancy simulation on the lower domain already has the shape of 64*64 we don't need to add extra padding to suit the LSiM model.

5.3.3 Applying loss functions

In order to compute the loss we have to define the reference and the model output. We know our model output is the correction, however it is unreasonable to directly compare the correction with the reference simulation. So the two inputs of the loss functions should be

- V: The sum between the verlocity tensor on the lower-domain simulation and our trained correction .
- $\hat{V}$: The downsampled verlocity tensor from the higher-domain simulation(reference).

For the MSE loss, which is exactly used by Prof. Um in his project, we can simply calculate their pixel wise square difference and compute their mean. This traditional loss function does not rely on the structure of the tensor as long as they share the same shape. So the MSE loss between two staggered tensors can be esaily computed. However the perceptual loss function has some demands on our tensor shape due to its CNN structure. For example, LSiM requires that the input tensor should consist of 3 channels to represent the RBG channels, while the size of the input tensor should be large enough to pass through several conv-layers. In our case, the output of the model is a staggered tensor with 2 channels which respectively represent x and y directions. Our solution is, that we compute the loss over each direction separately and sum them up. We firstly both extract the x direction tensor X and y direction tensor Y from our model output V, then we also extract the x direction tensor $\hat{X}$ and y direction tensor $\hat{Y}$ from the reference verlocity tensor $\hat{V}$. Our final loss function representation looks like:

$$Loss(V, \hat{V}) = F(X, \hat{X}) + F(Y, \hat{Y}) \tag{5.1}$$

where F is the respective perceptual loss function. In this way, we are able to treat the verlocity tensor on each direction as an image(or triplicate the channel to RBG image) and compare them with their ground truth.

Chapter 6

Results and Analysis

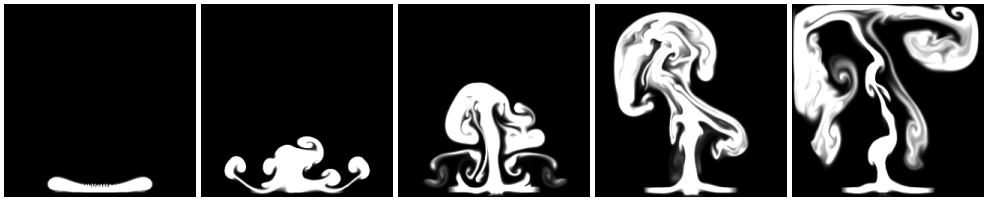
6.1 Comparison

6.1.1 Solver in the loop

After training lots of models on both buoyancy driven flow and the karman-flow of different resolutions, we have following results. In terms of numerical evaluation, we compute the distance of various metrics(l1, l2, mse, lsim, ssim, lpips) between the simulation corrected by the models trained by different loss functions, and the downsampled reference simulation. We compute the distance between each corresponding frames pair and compute their average as the result.

- Karmanflow
- Buoyancy driven flow

We first pick a dataset from our training set as the testset with renolds number of 2e5(note the renolds numbers of out trainingset range from 1e5 to 8e5)



(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.1: Reference simulation

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.2: Source simulation

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.3: simulation with lsim

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.4: simulation with mse

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.5: simulation with vgg

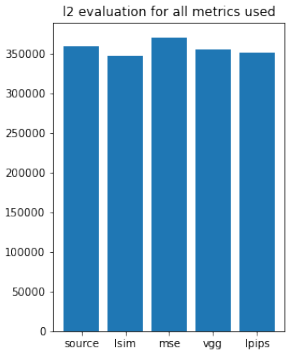
(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.6: simulation with lpips

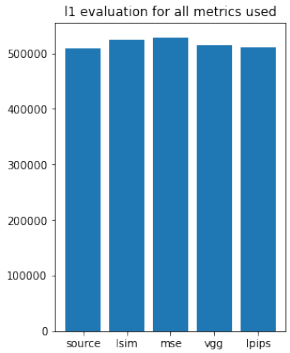
However when we generate a test set with the renolds number of 2.5e5 which was not included in the training set, the results are then different.

6.1.2 Time cost evaluation

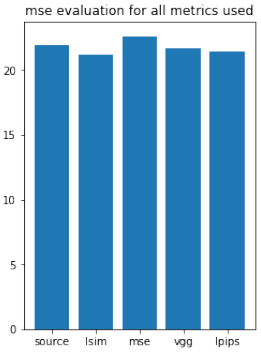
6.2 Limitaions



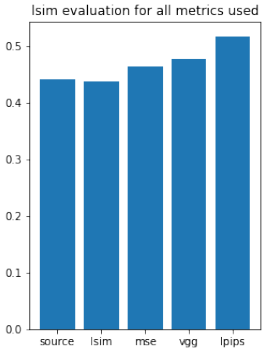
(a) l2 evaluation



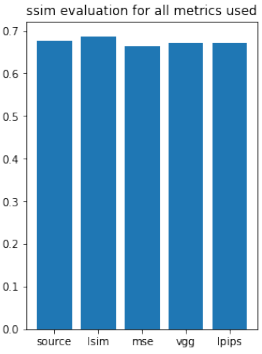
(b) l1 evaluation



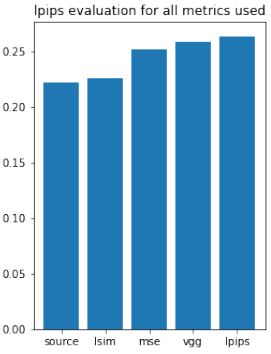
(c) mse evaluation



(d) lsim evaluation



(e) ssim evaluation



(f) lpips evaluation

Figure 6.7: Numerical evaluation of each metrics on training set

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.8: Reference simulation

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.9: Source simulation

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.10: simulation with lsim

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.11: simulation with mse

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.12: simulation with vgg

(a) $t = 50$ **(b)** $t = 100$ **(c)** $t = 150$ **(d)** $t = 200$ **(e)** $t = 250$

Figure 6.13: simulation with lpips

Chapter 7

Conclusion

7.1 Future work

7.2 Summary

Appendix A

e.g. Questionnaire

Note: If you have large models, additional evaluation data like questionnaires or non summarized results, put them into the appendix.

List of Figures

List of Tables

Bibliography

[HKT20] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. *arXiv preprint arXiv:2001.07457*, 2020.

[HW65] Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, 8(12):2182–2189, 1965.

[JAFF16] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[KUT20] Georg Kohl, Kiwon Um, and Nils Thuerey. Learning similarity metrics for numerical simulations. In *International Conference on Machine Learning*, pages 5349–5360. PMLR, 2020.

[ODO16] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 1(10):e3, 2016.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[UBF+20] Kiwon Um, Robert Brand, Yun Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. *Advances in Neural Information Processing Systems*, 2020.

[ZIE+18] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.