PHY513 Project Report - Fluid Dynamics Approximation through Machine Learning

Ximing Qiao

April 30, 2019

1 Introduction

Although nonlinear systems are difficult to solve analytically, many practical systems exhibits regular patterns. Those patterns are difficult to be described analytically, but are still evident enough for human to recognize. People are constantly solving nonlinear systems in their daily lift with pattern recognition, without even realizing them. After some training, students learning nonlinear dynamics can make accurate predictions of a system's behavior using very limited information. For example, we can project the dynamics to a 2D plane, solve the fixed points and their locally linear properties, then fill the non linear part with our prior knowledge.

On the other hand, recent advances in machine learning and deep neural networks show remarkable results in fitting highly non-linear data, e.g., natural images, and achieves human-level performance. A particularly interesting topic is image-to-image translation. Given enough training data, a neural network can translate images from an input domain to an output domain. The neural network uses its learned knowledge to fill the missing information, such as texture and color, exactly like how we solve nonlinear systems. As such, a natural question rises: Can we train a machine learning model to predict the behavior of some nonlinear systems?

To utilize the successful experience of applying deep neural networks on stationary 2D images, in this project, we focus on studying the steady state fluid flow around 2D obstacles. We craft a set of training images based on conventional computation fluid dynamics method, which numerically simulates the fluid using physical models. A deep neural network will fit the data and predict steady state velocity field around any unseen obstacle shape.

From our experiments, we find that a deep neural network can accurately predict the fluid dynamics in a limited range. This machine learning-based approximation approach cannot be as universal as conventional simulation methods, but provide a good speedup when dealing with repetitive tasks. Examples include boat shape optimization, which requires repetitively simulating the fluid flow around an obstacle, which change little between each optimization steps. As we can easily parallelize the execution of neural network, the neural network-based approximation can achieve over 100× speedup comparing to conventional simulations.

2 Background

2.1 Computational fluid dynamics (CFD)

Computational fluid dynamics (CFD) is a type of methods that use numerical analysis to solve fluid dynamic problems. Usually, a fluid field is described using partial differential equations (Navier-Stokes equations, the mass and energy conservation equations, the turbulence equations, etc.), and then solved using discrete approximations.

Solving a steady state fluid dynamic problem with CFD generally follows the following steps: First, choose an appropriate set of equations according to the problem; Second, discretize the problem space into many small connected triangles called *Mesh*; Third, assign boundary condition; Finally, solve the equations on mesh until the solution converge. On the other hand, solving turbulence problems might require different techniques according the specific problem type to get the best approximation results.

2.2 Image-to-image translation

In 2014, generative adversarial nets (GANs) [1] was introduced as an unsupervised learning method that can generatively model an unknown distribution. For a probabilistic distribution f with support set $\mathcal{X}$, the objective of GAN is to find a generator $G : \mathbb{R}^n \to \mathcal{X}$ such that $G(Z) \sim f$, given $Z \sim \mathcal{N}(0, I_n)$. The goal is achieved by assuming a discriminator $D : \mathcal{X} \to 0, 1$ that discriminates a fake sample generated from G against a real sample obtained from f. When parameterized as neural networks, both G and D can be trained by optimizing the min-max game:

$$\min_G \max_D \mathbb{E}_{x\sim f}[\log D(x)] + \mathbb{E}_{x\sim G(Z)}[\log(1 - D(x))] \tag{1}$$

through gradient descent. After convergence, $G(Z)$ should be identical to f, and D will output $1/2$ everywhere, failing to discriminated fake samples from true samples.

Conditional GANs [2] was later introduced as an extension of the original GAN and forms the basis of image-to-image translation [3]. The idea is to extend the loss function in min-max game with a *condition*, denoted as y:

$$\min_G \max_D \mathbb{E}_{x\sim f}[\log D(x|y)] + \mathbb{E}_{x\sim G(Z|y)}[\log(1 - D(x|y))]. \tag{2}$$

The condition can have many possible forms. When y represent the class label of an image (e.g., cats, dogs, horses, etc.), the generator is trained to generate images of a particular class. When y represents an image, the generator is trained to modify the *conditioning image* and match the *target image* from f.

Figure 1 shows image-to-image translation examples from [3]. Conditioning images can be segmentation information, black-and-white photos or object edges. Target images can be real images with detail texture, colored photos, or object paints. Notice that there is no magic in this image-to-image translation. All the missing information in translation are filled by a neural network trained a dataset containing thousands of image *pairs*. The neural net learns by analyzing the difference between each pair of images. Each type of translation is trained on a separate neural network and a separated dataset, and each neural network works like a "domain expert" that can only translate one type of images.

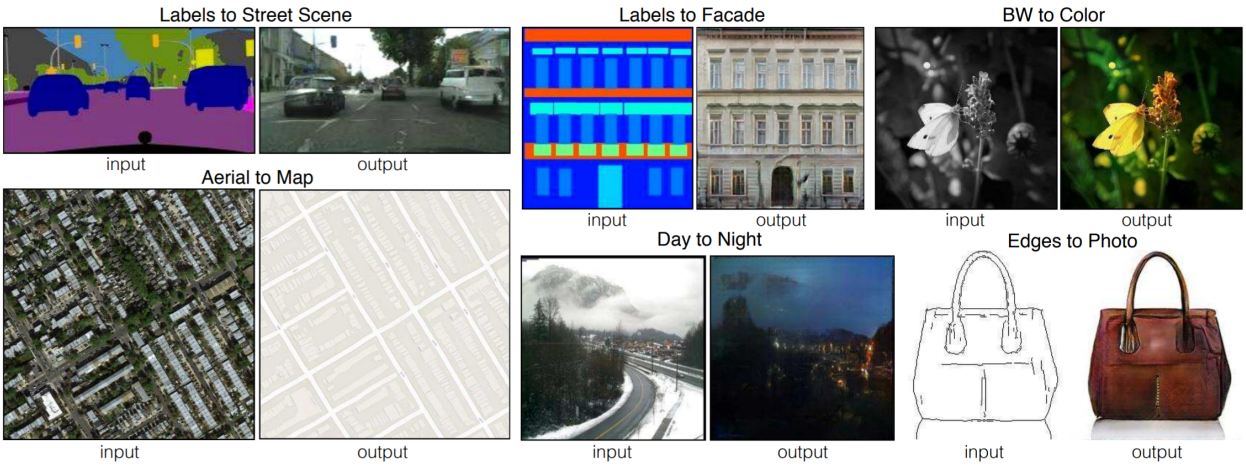


Figure 1: Image-to-image translation examples from [3].

3 Implementation

Implementing the machine learning-based CFD approximation includes multiple steps, each requires a different software.

For shape generation, Mathematica [4] excels at its expressive programming language and powerful visualization.

Mesh generation requires an open-source software Gmsh [5]. Gmsh supports manual mesh creation though its GUI, but it is too slow to create thousands of meshes. Alternatively, we can write a Python script to automate the process using Gmsh's Python API.

After obtaining meshes, we can use a Matlab CFD toolbox QuickerSim [6] for CFD simulation, and continue to use Matlab for some image processing to generate training images.

At last, we use a PyTorch implementation of image-to-image translation [7] to train and test the neural network. The details of each step are described as follows.

3.1 Obstacle shape generation

We divide the shape generation to two cases: simple shapes and complex shapes. Simple shapes are generated in polar coordinates, using a linear combination of several cosine functions. Here we choose 4 fixed frequencies $\omega = 1$, 2, 3, and 4. The amplitudes are sampled from uniform distributions $\mathcal{U}(-1, 1)$, $\mathcal{U}(-1, 1)$, $\mathcal{U}(-0.5, 0.5)$, and $\mathcal{U}(-0.5, 0.5)$. A code snippet of simple shape generation is shown in Figure 2. Those simple shapes are smooth, symmetric, and similar in size.

The complex shapes are still generated in polar coordinates, but using a linear combination of several sine functions, breaking the symmetricy. The complex shapes have 5 different frequencies and larger range of random amplitudes. Several shape examples are shown in Figure 3. Notice how they differ from the simple shapes.

To turn the generated shapes into meshes, we discretize the curves and turn the anchor points into Cartesian coordinates. All the points are exported to a JSON file for next step of processing. The code snippet for discretization and exporting is shown in Figure 4.

```
{numSamples, numPoints} = {12, 32};
{var, bias} = {{0, 2, 2, 1, 1}, {3, 0, 0, 0, 0}};
s[a_] := Function[x, Fold[{#1[[1]] + #2 * Cos[#1[[2]] * x], #1[[2]] + 1} &, {0, 0}, a][[1]]]
coeffs = Map[# * var + bias &, RandomReal[UniformDistribution[Length[var]], numSamples] - 0.5];
shapes = Map[s, coeffs];
Table[PolarPlot[shapes[[i]][x], {x, 0, 2 Pi}], {i, numSamples}]
```

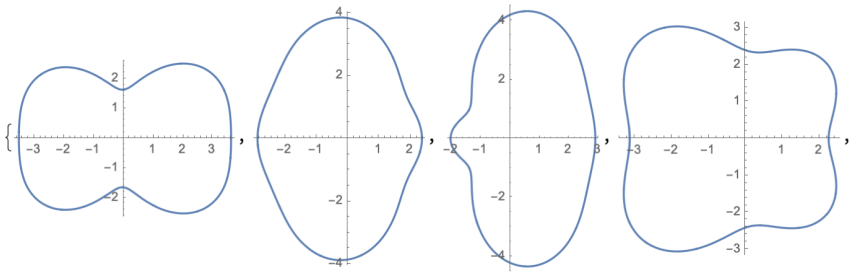


Figure 2: Mathematica code for simple shape generation.

```
{numSamples, numPoints} = {2000, 64};
f[a_] := Function[x, Min[5, Max[0.5, Fold[{#1[[1]] + #2 * Sin[#1[[2]] * x], #1[[2]] + 1} &, {3, 0}, a][[1]]]]]
{var, bias} = {{0, 5, 5, 5, 5, 5}, {0, 0, 0, 0, 0, 0}};
coeff = Map[# * var + bias &, RandomReal[UniformDistribution[Length[var]], numSamples] - 0.5];
s = Map[f, coeff];
Table[PolarPlot[s[[i]][x], {x, 0, 2 Pi}], {i, 12}]
```

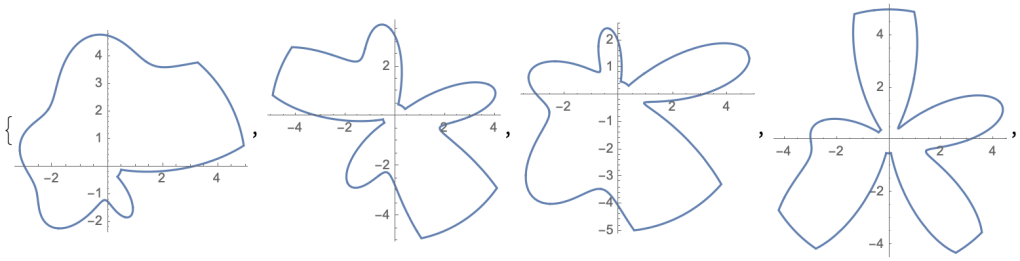
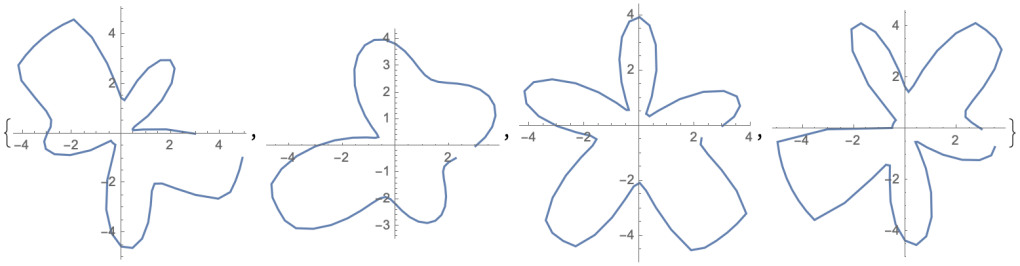


Figure 3: Mathematica code for complex shape generation.

```
polarPoints = Table[{s[[i]][x], x}, {i, numSamples}, {x, 0.001, 2 Pi * (1 - 1 / numPoints), 2 Pi / numPoints}];
cartePoints = Map[Map[{#[[1]] Cos[#[[2]]], #[[1]] Sin[#[[2]]]} &, #] &, polarPoints];
Table[ListLinePlot[cartePoints[[i]], AspectRatio → Automatic], {i, 4}]
```



```
SetDirectory[NotebookDirectory[]];
Export["data/points.json", cartePoints]
```

Figure 4: Discretize the generated shapes and export to a file.

3.2 Mesh generation

3.3 CFD simulation

Each training sample requires about 2 seconds of CPU time to compute. The whole process is done within 2 hours.

3.4 Neural network training

4 Results

4.1 Simple shapes

4.2 Complex shapes

5 Discussion

6 Conclusion and future works

For future works, an interesting direction is to integrate the machine learning approximation and numerical simulation. We can use the approximated solution as an initial value for the numerical solver and then obtain an accurate solution faster.

References

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[2] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[3] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[4] Mathematica. `https://www.wolfram.com/mathematica/`.

[5] Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. `https://gmsh.info/`.

[6] Quickersim: a cfd toolbox for matlab. `https://quickersim.com/cfdtoolbox/`.

[7] Image-to-image translation in pytorch. `https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/`.