Machine Learning Approaches for Particle Image Velocimetry

Presented to the S. Daniel Abraham Honor Program

in Partial Fulfillment of the

Requirements for Completion of the Program

Stern College for Women
Yeshiva University
April 27, 2021

Talya H. Stehley

Mentor: Professor Joshua Waxman, Computer Science,

Table of Contents

Machine Learning Approaches for Particle Image Velocimetry *1*

Introduction *3*

Background *5*

Neural Networks 6

Perceptrons 7

Gradient Descent and Backpropagation 12

Loss 15

Convolutional Neural Networks 17

Particle Image Velocimetry 18

CNNs for Optical Flow 19

Comparing Pre-trained Models *21*

Noisy Data 23

Conclusion *27*

Future Work *27*

Training New Models 27

Other Potential Avenues 28

Introduction

Particle image velocimetry, or PIV for short, is a technique in fluid dynamics for visualizing and analyzing complex flow in liquids and gasses. It has many applications, such as optimization of aerodynamics or hydrodynamics in vehicle design. There are also emerging biomedical applications, where it is used to assess blood flow through the heart (Sampath et al., 2018). Generally, the experimental process involves seeding a flow system with particles that are lightweight such that they do not disrupt the system, and reflective, such that they are visible on camera when illuminated by a laser. (Though in the biomedical case, image is obtained using an echocardiogram (Sampath et al., 2018)). Two exposures are taken, and then the task is to determine flow vectors from the displacement of the particles in the two images.

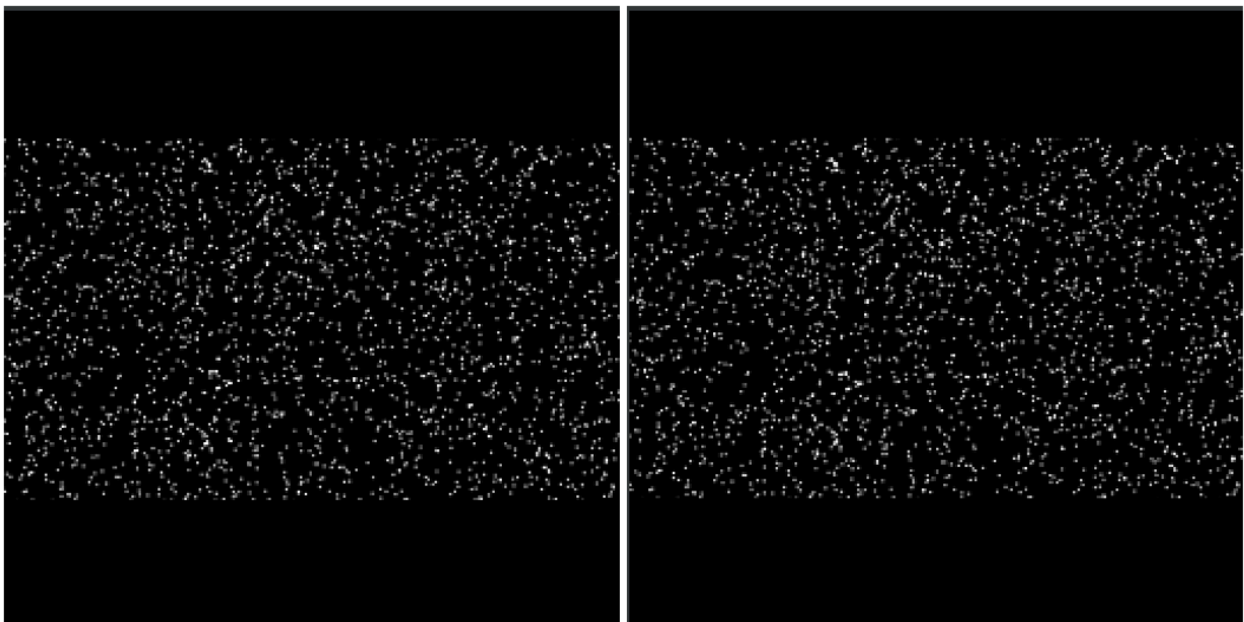

Figure 1: Example of a PIV image pair

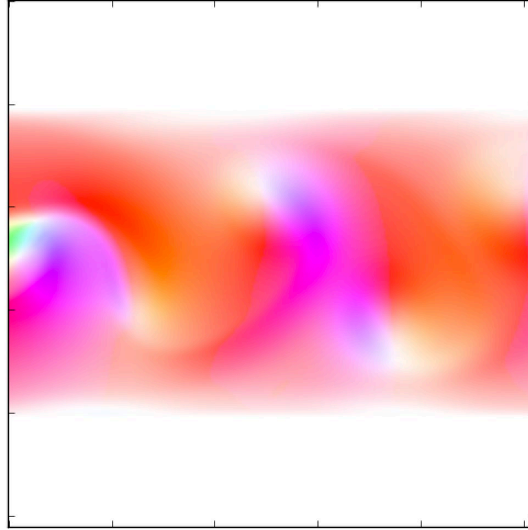

Figure 2: Visualization of the particle flow vectors between the two images in the above pair

Current state-of-the-art methods to determine flow vectors are effective but can be slow and difficult to get right. Recent research has been applying machine learning to the task, in the hopes of getting accurate results quickly. The purpose of my research has been primarily to compare the accuracy of two similar recent methods: one proposed by Cai et al. in 2019 (Cai et al., 2019), and another by Zhang and Piggiot in 2020 (Zhang and Piggott, 2020), with the eventual goal of testing new methods. Both papers train an existing neural network, LiteFlowNet on the same synthetic PIV dataset, but the papers each use different training methods. My intent was to compare the accuracy of the two trained networks, with the eventual goal being to try other training methods to see how they compare. In addition to training LiteFlowNet, the Cai paper proposes a modified version of the network with improved accuracy, but for the purposes of this paper, I have chosen to focus on the models based on the same underlying network in order to evaluate the effects of various methods of training. The code and data from both original papers is available online.

I have been doing this research as part of my internship at the Naval Research Laboratory, with help from Alisha Sharma and Kaushik Sampath. Training and inference for this project have been performed on a compute cluster with a GPU there.

Background

Machine learning is an umbrella term for algorithms that can change the way they solve problems in response to the data they are given. This process is referred to as training. Machine learning is usually done with artificial neural networks, which are complex collections of smaller units for processing data. While machine learning has existed for many years, it has seen an explosion of research and use in the past decade. It has been used to process and find patterns in data that would be difficult, expensive, or impossible to analyze using traditional methods. Increasing availability of large datasets have enabled progress in some domains (Zhou et al, 2017). Another major factor leading to the recent leap forward in machine learning is the use of graphical processing units, or GPUs for general computing. Originally designed for the purpose of 3d rendering, it's become apparent that GPUs are useful for any task that, like the graphical applications for which GPU hardware was originally designed, benefit from significant parallelization of math done on matrices. Cryptocurrency mining is one such application, at one point popular enough to cause a shortage of GPUs (Sexton, 2017). Machine learning, and more specifically, deep learning, involving neural networks with many layers, is the other major application that strongly benefits from use of a GPU. Non-graphical uses of GPUs, known as general-purpose GPU computing, or GPGPU have been facilitated by APIs like NVIDIA's CUDA, OpenCL, and AMD's ROCM, all of which make it relatively simple to write code in high level languages that takes advantage of GPUs. This capability has been highly beneficial for the development of neural network code (Heller, 2017).

Neural Networks

While many of the ideas underlying modern deep learning with neural networks have been around since the 80s and 90s, only in the last decade has the computational power existed to make significant use of them. StyleGAN (Karras et al., 2020), is a trained neural network that produces human faces that are often indistinguishable from the real thing. Science fiction author Arthur C. Clarke is quoted as saying that “Any sufficiently advanced technology is indistinguishable from magic.” (New Scientist) But it is crucial to remember that machine learning is not magic. Machine learning’s strengths can also be its weaknesses. Neural networks may identify patterns in their training data, but they may not be the patterns we want it to pick up on. The field is littered with tales like that of the dog/wolf differentiator that actually only learned to detect snow in an image (Ribeiro et al, 2016). In one well-publicized instance, Amazon had to stop using a recruiting AI they had developed after it became clear that it was under-rating female applicants (Dastin, 2018). That Amazon AI wasn’t designed to be sexist, but it had been trained on hiring data from a male-dominated workplace in a male-dominated field, and had picked up on that hiring pattern on its own. Still, despite the technology’s pitfalls, it can do impressive things.

So, what is a neural network? Artificial neural networks were originally based on mathematical models for describing the neurons in an animal brain, but they have use beyond and outside their original use in biological modeling (Cios, 2017). In biology, neurons are the major cells making up the brain. A neuron consists of three major parts: the soma, or main body of the cell, the dendrites, which accept input from other neurons, and the axon, which outputs an action potential to other neurons (Animatlab, 2011). Neurons release an action potential only once a certain threshold of input from other neurons is reached (Animatlab, 2011).

Perceptrons

Perceptron was an early example of an artificial neuron, conceived of and implemented in the late 50s by Frank Rosenblatt, then a psychologist at Cornell Aeronautical Laboratory (Lefkowitz, 2019). Unlike the more complex networks that would follow, it consisted of what would be considered a single layer. This consisted of a weighted sum of several inputs, which fed into a step function returning one of two possible values (Wallis). In one demonstration, it was able to distinguish between punch cards marked on the left from punch cards marked on the right, without explicit direction to do so (Lefkowitz, 2019).

At the simplest level a perceptron with one input can be described as output = weights*inputs (Shen)

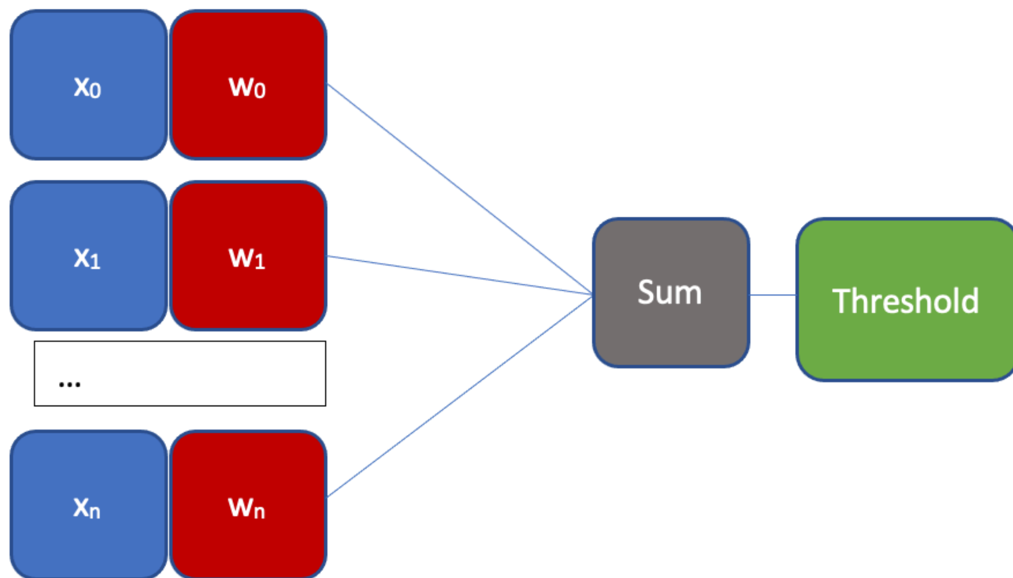

Figure 3: Example of a Perceptron

As you can see from the diagram, regardless of how many inputs a perceptron takes, it can only represent a linear equation. Weights are assigned to various forms of input. Using Rosenblatt's demonstration problem, sorting punch cards, as an example, these parameters could represent places a hole in the card could be. Say x_0 represents whether the card is punched on the

left and x_1 represents whether or not the card is punched on the right. What truly distinguishes the perceptron from an elementary linear equation to solve the same problem is that at the start, the model does not know how the cards should be sorted. Say we assign both weights (w in the diagram) to 1, and set a threshold of 2, meaning that if the weighted sum is greater than or equal to 2, the card is moved.

Now the operator gives the machine a stack of cards. Some are punched on the right, and some are punched on the left. The first card is punched on the left, so $x_0 = 1$, $x_1 = 0$, $wx_0 = 1$ and $wx_1 = 0$. The weighted sum is 1, which is less than 2, and the card is correctly moved to the left. Next, the operator feeds the machine a card punched on the right. This means that $wx_0 = 0$, $wx_1 = 1$, and the weighted sum is 1. This is lower than the threshold so the card is incorrectly moved to the left. We tell the machine that it has made a mistake, and misclassified our card. This is where the perceptron learning rule comes in. The perceptron learning rule is the algorithm for determining the ideal weights for the inputs of a perceptron (AI Learning). The rule is applied here by adding the x values to the weights (If the machine's mistake had been in the opposite direction, meaning we had obtained a value above the threshold that should have been below the threshold, we would subtract the x values from the weights) (Hagan et al., 2016). When we apply the perceptron learning rule, adding the x values to the weights, we add 0 to w_0 and 1 to w_1, meaning that now $w_0 = 1$, and $w_1 = 2$. Now that the perceptron has adjusted its weights, it behaves slightly differently. When a card punched on the right is passed into the machine now, $wx_0 = 0$, and $wx_1 = 2$. The weighted sum is 2, and the card is correctly sorted to the right. The perceptron has been taught to distinguish the cards based on a NAND relationship, Right NAND Left, where putting the card to the right would be equivalent to "true" for that logical operation.

And in fact, many logical relationships can be implemented with a perceptron. And the beauty is that there is no need to specify that relationship explicitly. With the perceptron learning rule, the perceptron can learn from the examples it is given. For another example, say we have the same setup as before, with a perceptron and punch cards. This time the cards may be punched on the left side, the right side, on both sides, or on neither side. This time, the we wish to separate the unpunched cards from the punched cards, meaning the relationship we want to implement is an inclusive OR relationship. We'll call the two possible piles "True" and "False". Once again, we reset all the weights to 1 and set the threshold at 2, so that if the weighted sum is smaller than 2, the card is placed in the "False" pile. First, we give the machine an unpunched card. This means $x_0 = 0$, $x_1 = 0$, so $wx_0 = 0$ and $wx_1 = 0$. The weighted sum is 0, smaller than 1, and the card is correctly placed in the "False" pile. Next, we feed the machine a card punched only on the left. We find that $wx_0 = 1$ and $wx_1 = 0$. The weighted sum is equal to 1, so that card is incorrectly classified as "False". Using the perceptron learning rule, we add the values of x to their corresponding weights, so that $w_0 =2$, and $w_1 =1$. Now, when we run a card punched on both sides through, $wx_0 = 2$ and $wx_1 = 1$, the weighted sum is greater than 2, it is correctly placed in the "True" pile. Finally, we pass in a card punched only on the right. We find that $wx_0 = 0$, $wx_1 = 1$, and the card is incorrectly placed in the "False" pile. We alert the machine to its error, and it once again uses the perceptron learning rule, updating the weights by adding the associated inputs. Now $w_0 = 2$ and $w_1 =2$, and the perceptron is fully trained to sort based on an inclusive OR relationship. With these weights, anything passed in previously will be correctly sorted. Despite starting with the same initial conditions as before, the perceptron now functions following a completely different logical rule, one it uses not because it was told to, but because it determined that it was the requirement based on the data.

But what if we wanted to sort based on an XOR relationship? What if we wanted cards punched once in the “True” pile, regardless of whether that hole is punched on the left or the, and we wanted cards punched twice or not all in the “False” pile? The perceptron has learned other logical operations, so it seems like it should be able to learn XOR. As before, we set the weights at 1 and threshold at 2, so that any value under the threshold is considered “False”. Then we start sending in our cards. First, we send in a card that is punched on the left, so $wx_0 = 1$, $wx_1 = 0$. The weighted sum is 1, and under the threshold, so the card is incorrectly sorted as “False”. As before, we apply the perceptron learning rule, adding the inputs to the relevant weights. Now $w_0 = 2$, $w_1 = 1$. Next, we send in a card punched on both sides. We find that $wx_0 = 2$, $wx_1 = 1$, so the weighted sum is 3, which is over the threshold. The double-punched card is incorrectly placed in the “True” pile. We apply the perceptron learning rule again. This time, since the output was *over* the threshold, we subtract the inputs from the weights, meaning that now $w_0 = 1$, $w_1 = 0$, and the problem with this tool for this problem has become clear. If we pass in another card punched only on the left, it would again be sorted incorrectly “False”, we would increase w_0 by 1, only to have to decrease it again when the perceptron incorrectly puts a double-punched card in the “True” pile. Perceptrons can do some things, but they can’t do everything.

Rosenblatt demonstrated that a perceptron could function as a binary classifier, but the perceptron has a major limitation. What was proved by Marvin Minsky, a contemporary of Rosenblatt’s, and AI researcher at MIT (Lefkowitz, 2019), was that a singular perceptron could only classify linearly separable data, or any data that when plotted, could be separated using a straight line. Every logical operation except XOR and NXOR (Wallis) is linearly separable, but plenty of potentially classifiable things are not.

Though artificial neurons were proven to have their limits, that did not mean research into their uses was a dead end. In the 1970s and 80s (Hagan et al., 2016) it was discovered that more complex data could be classified by chaining together multiple individual artificial neurons. A perceptron with its input dependent on the output of another perceptron could deal with nonlinear classifications. This approach is known as a multi-layer perceptron or a neural network (AI Learning). Going back to our XOR problem, if we allow ourselves more than one perceptron, we can chain them together to differentiate based on the XOR relationship. A XOR B can also be expressed as (A OR B) AND (!A OR !B) (Hagan et al., 2016).

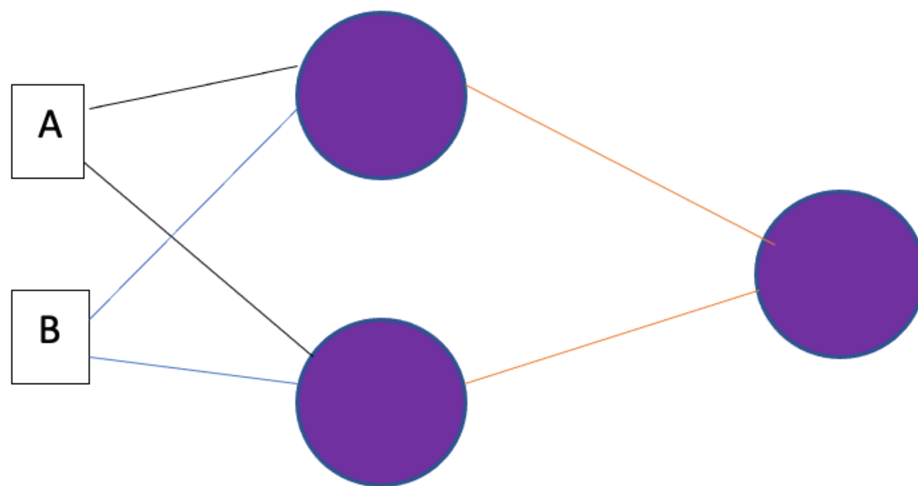

Figure 4: Multilayer Perceptron

We can build that using what we know already. We can see how this might work if we treat trained perceptrons like the logic gates they have learned to imitate. We can hook the input up to two separate perceptrons, train them on the two OR relationships, then hook the output of those perceptrons to a third perceptron, and train that to distinguish an AND relationship. Now we have a multilayer perceptron. What we would consider the first layer would be the part

consisting of the two perceptrons connected to the inputs. The second layer would consist of the single perceptron connected to the output.

Gradient Descent and Backpropagation

Now that it is established that a solution for an XOR classifier exists for this configuration of perceptrons, the next question is how one would train such a thing. Trainability, after all, is one of the big advantages of the neural approach, but if one can only train individual perceptrons, there would seem to be no real benefit to chaining them together. But there is a process for training a multilayer network, known as gradient descent (Nielsen, 2015). To perform gradient descent, we must introduce the concept of loss, or error. Several functions exist for determining this value, and they will be discussed in greater depth further on. Whichever loss function we decide to use, we use it by running input through the network, then finding the loss between the desired output and the actual output, then applying gradient descent (Hagan et al., 2016). Gradient descent is in essence, an optimization. One can conceptualize the loss as a function of the weights and input. In any particular example, the inputs would function as constants, since the input is the input, and the weights are changeable variables. The goal is to find the set of weights that is the minimum of this loss function (Nielsen, 2015). In its simplest form, one would find the the derivative of the loss with respect to each of the weights (Hagan et al., 2016). We would use that derivative to the adjust the parameter weights in the correct directions to minimize future losses (Nielsen, 2015). The appropriate step size is problem-dependent, and the parameter governing the degree to which weights can shift in a single iteration is referred to as the learning rate. This learning rate needs to be low enough that the algorithm doesn't overshoot the minimum it is meant to be locating, but high enough that the training process can be completed before the heat death of the universe. With enough iterations

of this optimization, we should be able to minimize the loss (Nielsen, 2015), meaning that the output of the trained neural network should be more accurate than it would have been before training.

Although the we know the weights affect the loss, actually determining the loss in terms of the weights so that we know how to change them seems difficult. The specific algorithm for obtaining that gradient of the weights is known as backpropagation, and was only invented in the 70s (Nielsen, 2015). The loss function in its simple form seems would seem to be a function of the final network output, since it is determined the gap between the final network output and the expected output, but that output is affected by the network weights, which is how we can have that understanding of the loss as a function of the network weights (Nielsen, 2015). The backpropagation algorithm goes back layer by layer, from the output to the input, finding loss in terms of the preceding layer weights, those weights in terms of the weights before them, and so on and so forth, eventually calculating that necessary gradient in its entirety (Nielsen, 2015), allowing us to train networks of neurons.

By chaining artificial neurons together, we would seem to get closer to how a biological brain might work. However, a criticism of neural networks research is that since so little is understood about the way that human and animal brains operate, there is no reason to assume that any artificial neural network will be able to truly understand things the way humans can (Watson, 2019). Regardless of whether or not any hypothetical artificial neural network could be said to be truly intelligent, neural networks have, at the very least, proved to be useful, and to be an improvement over models that existed in the past for certain tasks.

When artificial neurons are chained together, several factors affect how the resulting network functions. One major task when designing a neural network is determining what data

neurons pass on to the other neurons in the network. On the individual neuron level, this is done through what is known as the activation function. Activation functions is what determines the ultimate output of an artificial neuron based on the weighted sum of the input (Sharma, 2017). Early artificial neurons used a step function (Wallis), where it would fire if the weighted sum of the inputs met a certain threshold (Sharma, 2017). This Boolean output replicates the behavior of a biological neuron, which either generates an action potential, or does not generate an action potential. This functions like the threshold in our examples of individual perceptrons, where either the output is true, or the output is false. This type of activation function is useful for binary classifiers, where two mutually exclusive results are possible. However, it is not suited to more complex classification tasks, where one wants one result out of multiple possibilities (Sharma, 2017), which is why many other activation functions exist. The linear activation function is proportional to the weighted sum of the neuron's inputs, which allows for a range of outputs (Sharma, 2017). In the example of a classifier, the outcome with the largest output would be chosen as the result. Where a step function could be used to classify between two categories (assuming separability under limitations established by Minsky), a linear function could classify between several possibilities. However, the derivative of the linear function is a constant (Sharma, 2017), so when weights are adjusted using the gradient descent algorithms discussed earlier, the change in weight will be constant, rather than proportional to the error, meaning that even if multiple neurons are used, the network will only be able to classify linearly separable data, because multiple linear functions chained together is still a linear function. Only by using a nonlinear activation function in the neural network can this limitation be avoided. An example of a nonlinear activation function is the sigmoid function. It can be graphed as a curve which is steep towards the middle of the range, and less steep near the extremes. This means it tends to

make a clear distinction (Sharma, 2017) (though nothing will ever be quite as clearly distinct as the step function). Like a step function, the sigmoid activation is bounded between 0 and 1. Another nonlinear activation function, particularly common in the convolutional neural networks used for many computer vision tasks, is known as a rectified linear unit, or ReLu. A ReLu activation function is linear when input is positive, and 0 if when the input from the neuron is negative (Sharma, 2017). This makes for more efficient networks than some other activation functions since it means the relatively high chance of a 0 output means fewer inputs to subsequent layers, an important consideration in computer vision where there can be a nearly unmanageable number of parameters. However, the gradient of the zero section of the ReLu activation is zero, meaning the partial derivatives of those weights will be 0, meaning that when gradient descent is applied, there will be no change to those particular weights regardless of error, which means that the network could essentially train itself into a corner, where it can no longer improve its predictions, because the gradients will not change the weights that need to change. One solution to this problem is known as a leaky ReLu, which makes the response to negative input a small non-horizontal line, reintroducing a small gradient into the backpropagation process, allowing the weights to slowly shift if they are in that condition. Leaky ReLu is the activation function used by LiteFlowNet, the neural network with which the experimental portion of this paper is concerned (Hui et al., 2017).

Loss

As networks and the data fed to them grow more complex, so does the process of determining error. While with our single perceptron, all we needed to do was to identify the whether the error was positive or negative (true instead of false, or false instead of true), and then apply the perceptron learning rule, determining error in order to adjust weights is not as simple

for larger networks. Even when the same general idea of gradient descent and backpropagation is being used, there are many ways to approach learning. Some components that make up these variable approaches are the loss function, also known as an error function or objective function (Nielsen, 2015), and the optimization function. The loss function determines how far the output is from what it should be. The optimizer determines how and when gradient descent should be applied, shifting the weights in response to the loss (Nielsen, 2015). For a network that only needs to output a single number, the loss may very well be something as simple as the difference between the network's output and the correct output. One common loss function, known as Least Absolute Deviations, or L1 loss, is that idea applied to larger output data. In many cases the output of a neural network will be an array of values. L1 loss is the sum of the differences between each individual point and the corresponding value in reference array (Shekhar, 2019). L2 loss, also known as Mean Squared Error, is similar, but each individual error is squared before being summed up (Shekhar, 2019). Average End Point Error, or EPE, is used for validation purposes by Zhang and Piggott (Zhang and Piggott, 2020). It seems to be fairly common for evaluating optical flow calculations (Baker et al.), and it is one of the metrics used for evaluating performance on the Middlebury optical flow benchmark. It involves finding the average difference between values in the network output, and values of the ground truth, but does this from the perspective of vectors rather than pixels (Zhang et al., 2017).

Once we know the loss, we can change the weights to hopefully improve the network. Stochastic Gradient Descent, or SGD is a significantly less computationally intensive variant of the gradient descent algorithm, which calculates gradient based on a randomly chosen sample of the output (Bottou, 2010), at the expense of adding more randomness into the training process. Adam is another stochastic optimization method, which has variable step sizes for each

parameter (Ruder, 2016). This allows training to progress quickly at first, and slow down as it approaches the minimum. (Kingma and Ba, 2014). Adam keeps an average of past squared gradients that decays exponentially, functioning as a form of momentum (there are also SGD variants that use momentum) (Ruder, 2016). Amsgrad is a variant of Adam that determines momentum using the maximum of past squared gradients rather than the average, leading to better convergence in some cases (Reddi et al., 2018).

Convolutional Neural Networks

While there are many different neural network architectures in common use, the one most relevant to PIV applications is a type known as a convolutional neural network. Convolutional neural networks (or CNNs for short) are designed to take images as input (O'Shea and Nash, 2015) have been used to great success for computer vision tasks (Cios, 2017). CNNs generally contain three types of layer: convolutional layers, pooling layers, and fully connected layers. Convolutional layers produce output based on specified parts of the input image, usually using ReLu as the activation function (O'Shea and Nash, 2015) The convolution operation involves a sort of matrix, significantly smaller than the input image, known as a convolutional kernel. The convolutional kernel will slide across the input image, calculating a weighted sum for each position, of the data there along with the kernel's activation map (O'Shea and Nash, 2015). This is a relatively efficient way to extract identifiable features from raw image data.

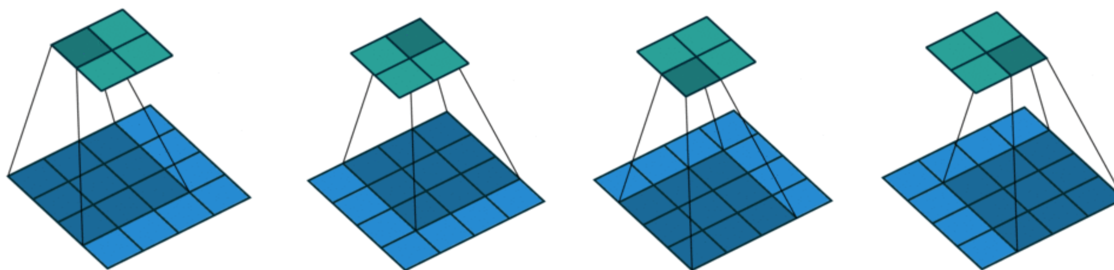

Figure 5: Visual Representation of Convolution from source (Dumolin and Visin, 2016)

The activation map is how convolution kernels are tuned to pick up on specific features. The pooling layers downsample the input they've been given, reducing the complexity of the model (O'Shea and Nash, 2015), a necessary step for dealing with large complex data like images, as well as allowing for aggregation (Dosovitskiy et al., 2015). Max pooling is the most common type (O'Shea and Nash, 2015), but more complex pooling methods also exist. The fully connected layers then produce output based on the totality of what they're given from the preceding layers, functioning like a more typical neural network in the final stage of inference. Unlike the prior layers, each neuron in a fully connected layer is connected to every neuron in the layer that follows. The fully connected layers of a CNN convert the representation of the image from the convolution and pooling layers to network output. In the case of a classifier, that would be the classification. In the case of the optical flow networks be discussed later in this paper, that would be a velocity matrix.

Particle Image Velocimetry

While there has been recent research using neural networks for PIV analysis, there are some other more established existing methods for the task. Cross-correlation is one of the primary methods for deriving underlying flow vectors from PIV images. The output of that method is a sparse vector field (Dabiri). A velocity vector is determined by applying a cross-correlation algorithm to a specified interrogation area (O'Shea and Nash, 2015). The cross-correlation operation can be done directly or using a Fast Fourier Transform (O'Shea and Nash, 2015). Cross-correlation determines the displacement, and along with the known change in time between exposures, thus yielding the velocity.

Optical flow estimation is the other major technique for getting velocity vectors from PIV images. As a general concept, optical flow is defined by the way the structure of the light hitting

a sensor changes (Raudies, 2013). An advantage optical flow estimation methods have over cross-correlation is that they are able to produce dense vector output, since analysis is not limited by the windowing technique used in cross-correlation. Optical flow estimation has a number of applications in the broader world of computer vision. This is beneficial for PIV research, since it means that research done for broader vision tasks using optical flow techniques can then be turned to PIV, for which optical flow is already an established mode of analysis. Several methods exist for optical flow analysis (Raudies, 2013), including methods based on convolutional neural networks.

CNNs for Optical Flow

Enter FlowNet. The abstract to Fischer et al's landmark 2015 paper states that "Optical Flow estimation has not been among the tasks where CNNs were successful", but their work changed that (Dosovitskiy et al., 2015). Unlike earlier computer vision tasks to which CNNs were applied, any optical flow application requires not just identifying image features, but finding the same features in a subsequent image (Dosovitskiy et al., 2015). The FlowNet paper was a landmark use of neural networks for optical flow analysis, and while it didn't outperform the variational methods that formed the state-of-the-art at the time (Dosovitskiy et al., 2015), from it sprang all manner of sequels and derivatives. The original FlowNet paper proposes two architectures for the task, FlowNetSimple, and FlowNetCorr (Dosovitskiy et al., 2015), referred to in later works a FlowNetS and FlowNetC. FlowNetSimple stacks the images together and feeds them to what the authors describe as a fairly typical CNN. FlowNetCorr initially processes the two input images separately, generating meaningful representations individually, before using a "correlation layer", which makes comparisons between the respective feature maps. Notably, FlowNet lacks fully-connected layers.

In 2017, a different team of researches attempted to build an improved FlowNet, calling it FlowNet2.0. They found that they could make significant improvements just by modifying the training methods. They also made some architectural improvements, introducing image warping and an iterative refinement process that involved stacking multiple subnetworks, each made up of an entire FlowNetC or FlowNetS network. In these stacked models, every subnetwork after the firsts would get both the input images and the previous network's flow estimate.

The LiteFlowNet paper, published in 2018 (Hui et al., 2017), sought to achieve comparable performance to FlowNet2.0 using a smaller model size. Like FlowNet2.0, it warps the second image towards the first based on earlier flow estimates, but applies warping to extracted features rather than to the image as a whole, and does so several times. (Hui et al., 2017). LiteFlowNet's two big ideas are pyramidal feature extraction and feature warping. With pyramidal feature extraction, details are determined using a course-to-fine framework (Hui et al, 2). In the feature warping stage, the extracted features of the second image will be warped to the first image, to improve the flow estimate(ibid). This is done several times to refine the flow estimate(ibid).

There has been some recent, promising research on the use of machine learning to analyze PIV data. Cai et al.'s 2019 paper, "Particle Image Velocimetry Based on a Deep Learning Motion Estimator" (Cai et al., 2019), and Zhang and Piggott's 2020 paper, "Unsupervised Learning of Particle Image Velocimetry" (Zhang and Piggott, 2020) both build from LiteFlowNet (Hui et al., 2017) and get results comparable to current state of the art. The crucial difference between them is that the Cai paper uses a supervised learning method, while Zhang and Piggott's paper uses an unsupervised learning method, where comparison to ground truth is not used to train the model. From a high-level point of view, that unsupervised method

essentially determines error by comparing the what the first image would look like after the predicted flow is applied to it, and comparing that to the second image (and repeating this in the opposite direction). While the supervised paper's method of determining error is in more familiar territory, comparing actual output to expected output, the exact loss function it used is more complex than the loss functions discussed earlier. LiteFlowNet makes predictions at several points in the process of generating its final output, the loss function is a weighted sum of all of the losses from all of the predictions (Cai et al., 2019). The unsupervised paper, based on the same network, uses a similarly multi-scale approach with its loss function, which makes sense given that the LiteFlowNet architecture enables this approach.

Comparing Pre-trained Models

My first goal experimentally, was to test the performance of the existing models myself. Code from both networks is publicly available on GitHub. Though both models are based on the same underlying network, PIV-LiteFlowNet is based on the original Caffe version of LiteFlowNet while UnLiteFlowNet-PIV is based on a PyTorch port of LiteFlowNet (Niklaus, 2019). I ran that code on a cluster with a GPU, within Singularity containers for each network's requirements. This was itself an undertaking. At first, I had attempted to install an off-the-shelf version of Caffe, using the apt package manager, but found that I could not load Cai's trained model. Eventually, it became clear that their model included custom layers, and to use it, their modified version of Caffe needed to be built from their code. While Zhang and Piggott's PyTorch model did not require custom compilation, I had initially assumed that I would be able test it out on a CPU. However, their correlation layer includes a custom CUDA kernel, with no CPU alternative. This means their model cannot be run without an Nvidia GPU, at least not without significant modification. Neither of those difficulties are particularly surprising. Code

from research papers is shared as a courtesy, not as a product. Neither team made any pretense of offering their code as a finished PIV solution. It is not overly surprising that research code may not always be easy to use without tweaking.

Both models were trained with the same dataset (Cai, Zhou, et al., 2019). Each item in the dataset consists of two images of a fluid system in motion, and a ground truth. All images are 256x256 pixel grayscale tiff images. The ground truth is in the Middlebury .flo format, which lists the vertical and horizontal components of the flow vector at each point in the image (Scharstien, 2007). For a previous paper (Cai, Zhou, et al., 2019), Cai and co-authors had simulated some of the flow types directly, including uniform flow, backstep flow (where recirculation occurs due to the fluid encountering a step), and flow over a cylinder. Others were taken from existing sources, including the 2D DNS turbulent flow (flow with frequent changes in pressure and velocity), a surface quasi geographic model (flow of atmosphere across multiple isobars, or values of pressure), as well as channel flow, isotropic flow (flow with no directional preference,), and magnetohydrodynamic flow (affected by both fluid mechanics and electromagnetic forces) from the Johns Hopkins Turbulence Database (JHTDB). This dataset comes pre-partitioned into a set for training and a set for testing, both spanning all flow conditions included.

To test the models, I used the test partition included in the Cai dataset. Using both models, I ran inference on all image pairs in the test set, then compared normalized output to the ground truth, comparing mean squared error between flow conditions and models.

Flow Type	Supervised Avg MSE	Unsupervised Avg MSE	Image Source
Uniform	4.26E-05	7.57E-2	Cai
DNS Turbulence	8.94E-3	3.53E-2	http://fluid.irisa.fr/data-eng.htm (Memin et al.)
SQG	1.00E-2	3.5E-2	http://vressegu.github.io/sqgmu/ (Resseguir et al., 2016)
Backstep	1.49E-4	2.00E-3	Cai
Channel	8.68E-4	3.24E-3	JHTDB
Channel HD	1.62E-2	8.08E-2	JHTDB
Isotropic	9.02E-3	3.5E-2	JHTDB
Magnetohydrodynamic	6.45E-3	2.52E-2	JHTDB
Cylinder	2.23E-2	1.08E-2	Cai

Table 1

For most of the flow conditions contained in the Cai dataset, Cai et al.'s supervised model produced more accurate output than Zhang and Piggott's unsupervised model, with one exception. In the case of cylinder flow the unsupervised model performs slightly better. The physics of flow over a cylinder mean that very small changes to initial conditions have a massive effect of the overall flow. Perhaps the seeming discontinuity of this flow type makes it the type of condition where an unsupervised model can perform better than a supervised model.

Noisy Data

One of the advantages claimed by Zhang and Piggott (Zhang and Piggott, 2020) is that the unsupervised network should be able to be more robust to noise and other elements that make inference on real world data more difficult than it is for idealized synthetic data. We sought to test that against a new set of data. Set 001 was generated in house at NRL by Kaushik Sampath. All images in this set are generated from the same ground truth, but vary the amount of gaussian noise (making up a fraction between 0.0001 an 0.1 of the whole image), mean particle diameter (ranging from 3-5 pixels), variation in particle diameter (ranging from 0.5 to 2 pixels), particles

per pixel (from 0.01 to 0.1) and particle intensity change from frame to frame (between 0 and 1). A model that is robust to these variables should be more effective in real-world use, where outside conditions and image quality may make an image noisier than simulated images would be. Over the set as a whole, the average MSE for PIV-LiteFlowNet is 11.32, and the average MSE for UnLiteFlowNet-PIV is 17.07. However, if we only look at examples with greater amounts of noise, UnLiteFlowNet-PIV starts to pull ahead. When considering noise only above 0.06, the average PIV-LiteFlowNet error rises to 23.6, while UnLiteFlowNet-PIV's average mean squared error at that level of noise is only 19.3. Though under ideal conditions, PIV-LiteFlowNet is more accurate, UnLiteFlowNet-PIV may be more accurate once significant noise is added to the image. The UnLiteFlowNet-PIV's average error rises more slowly than PIV-LiteFlowNet's average error, and its standard deviation is relatively stable. UnLiteFlowNet-PIV's average mean squared error is better than PIV-LiteFlowNet's average mean squared error at higher levels of noise. Once I had run inference through the models, Kaushik Sampath also did some analysis of his own, using L2 loss, as well as calculating standard deviations. When compared using L2 loss, UnLiteFlowNet-PIV's average error is slightly higher than that of PIV-LiteFlowNet, even at high levels of noise, but its standard deviations are consistently smaller once sufficient noise is introduced. It seems when the image is noisy, the supervised model may perform better, but may also perform significantly worse than the unsupervised model. The unsupervised model by contrast, seems to have a relatively consistent amount of error, in a relatively small range. It seems likely that there are applications where the latter behavior would be preferable. One may want to have flow data that can be relied on to be at a certain level of accuracy, as opposed to data that may be variably accurate and more sharply dependent on conditions. However, a researcher using PIV in a more ideal situation where things like noise are

less of a concern may prefer the unsupervised model, if it would be more accurate in that specific scenario. For scenarios with consistently very low noise, PIV-LiteFlowNet would seem to be the clear choice between the two.

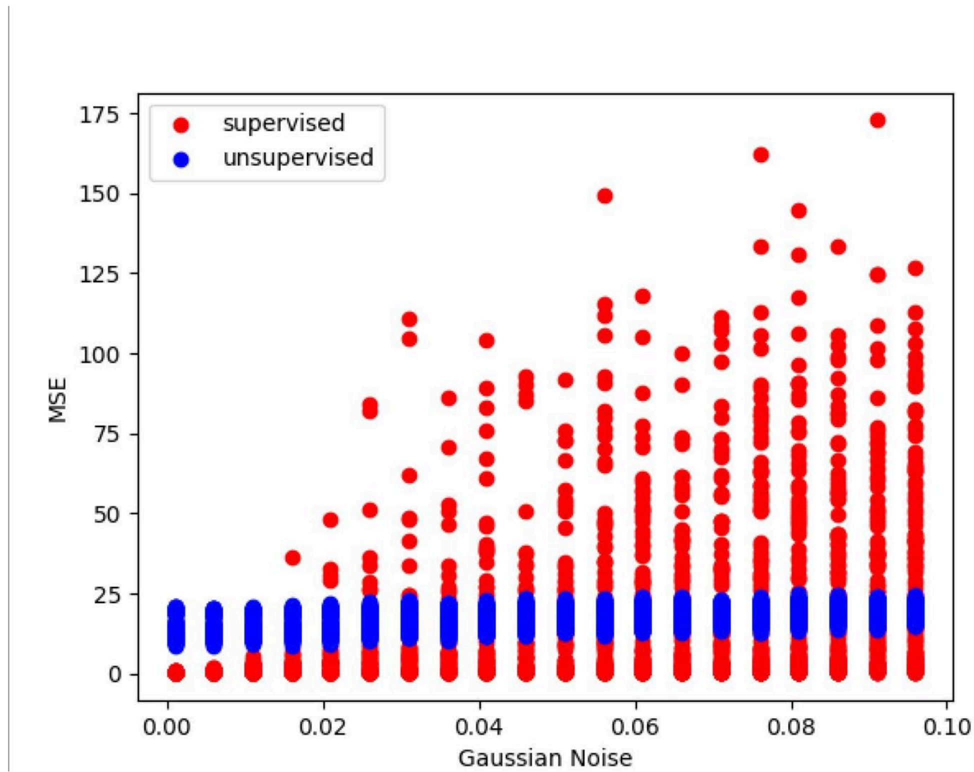

Figure 6: Scatter plot of MSE vs gaussian noise

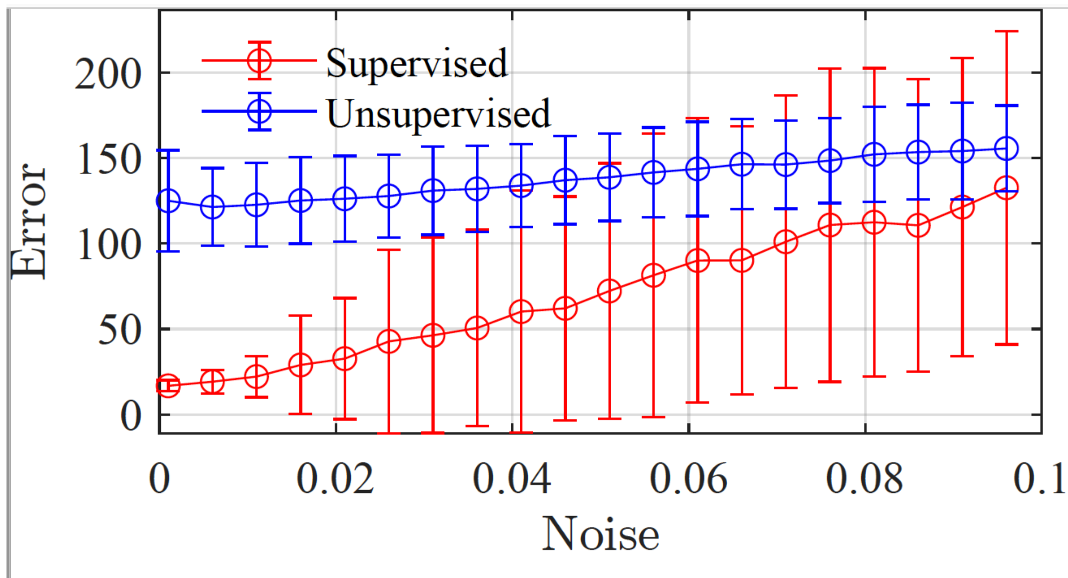

Figure 7: Same images compared using L2 Norm, averaged, and with standard deviation. Plot by Kaushik Sampath

Noise is not the only factor that impacts the accuracy of PIV analysis. Another factor towards which the supervised model seems to be more sensitive is particles per pixel. Both models seem to do better with more particle density, but once again we see a much greater degree of variation from the supervised model, especially when the density is low.

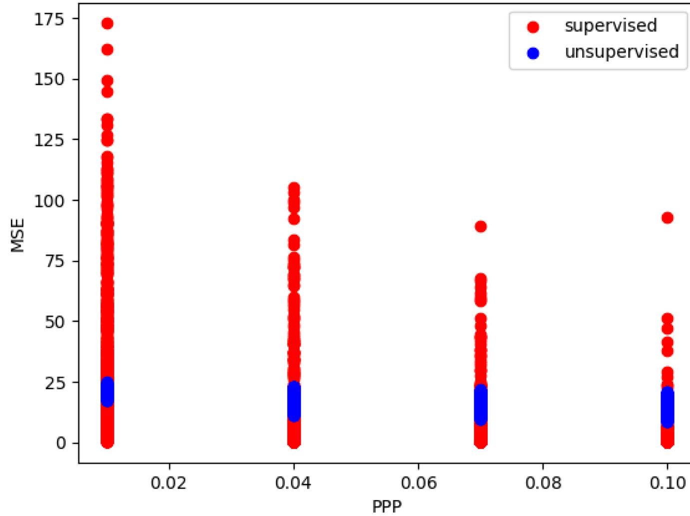

Figure 8: MSE vs. Particles Per Pixel

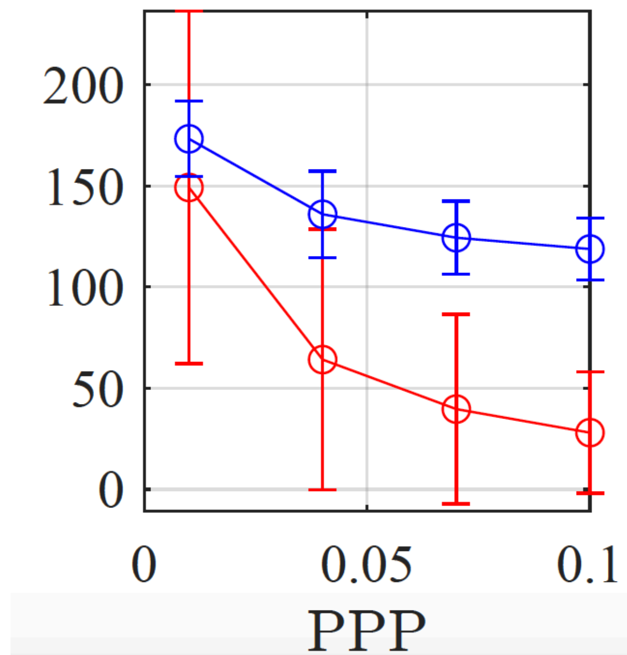

Figure 9: L2 Error vs. Particles Per Pixel. Plot by Kaushik Sampath

Conclusion

Machine learning has the potential to make PIV analysis faster, easier, and more robust, but the best possible way to perform this task is still an open question. Comparing various approaches to learning on the same model, multi-scale supervised learning is more accurate under ideal circumstances, but unsupervised learning may perform better in certain flow conditions, and especially in noisy conditions. Though the supervised approach may be more accurate in many cases, under sub-optimal conditions, its level of accuracy can be far more variable than that of the unsupervised model. Both approaches would seem to have their advantages. Whether one could build a model combining the accuracy of the supervised model with the robustness of the second remains to be seen.

Future Work

Training New Models

Seeing these results, which showed that both the supervised and unsupervised approaches had different advantages with regards to this problem, I wanted to attempt a hybrid training method, where I trained a model partially in a supervised fashion, and partially in an unsupervised fashion. To do this, I first built a supervised training loop on PyTorch. For simplicity's sake, I started off with a basic EPE loss, because I had noticed that Zhang and Piggiot had used the same loss function for validation (and had kindly provided the code in their files), and I was also curious to see how a simple loss function stacked up to the more complex LiteFlowNet specific loss. I do also intend to implement a hybrid loss with the multi-scale loss. This work is still ongoing. I hope to train models using both types of loss, either starting with supervised loss, then switching to unsupervised loss, or starting with supervised and switching to

unsupervised loss. I would also have to investigate to determine if there is an optimal point in the training process to switch loss functions.

Other Potential Avenues

In general, there is still considerable work to be done in this area. As the FlowNet2.0 paper makes clear, existing models can be improved significantly, merely by improving the data on which they are trained (Ilg et al., 2017). It would be interesting to see if the multi-scale supervised model would perform differently if trained with noisier data. In addition to training LiteFlowNet, the Cai paper (Cai et al., 2019) also made some enhancements to the underlying network which improved its accuracy. It would be interesting to see how the enhanced network responds to noise in comparison with the original network, as well as to re-train that enhanced network in an unsupervised or semi-supervised fashion. In general, training networks on more realistic synthetic data may produce better results. Some optical flow papers mention using a "curriculum learning method" where better results are achieved by training the network on simpler data before training it on more complex data (Ilg et al., 2017). Perhaps similar techniques could be used to lend these PIV models better robustness to noise.

Another potential avenue for future research is to incorporate a physics-informed approach, embedding more constraints into the network based on the equations that govern fluid systems. It seems likely that more accurate results may be achieved by tailoring the network to the specific problem of PIV.

The study of machine learning based methods for PIV analysis has progressed considerably over the past few years, but there may very well still be room for continued improvement. Machine learning methods may be more robust to conditions where traditional analysis methods struggle. Though training is time and resource intensive, a trained model

analyzes an image pair quickly, on the order of hundredths of a second per image pair (when running on a GPU), which is significantly faster than conventional methods. This could be beneficial in applications like medicine, where time may be crucial. Another downside of conventional methods, is that while effective, they are complicated to implement as the correct parameters for certain things are problem dependent. A mature machine learning solution could potentially work more consistently “out-of-the-box”, with fewer adjustments required to obtain acceptable output for one’s specific problem. Though existing code can be difficult to get running, with its specific dependencies, this is understandable, as that is research code, meant to test an idea. Assuming no insurmountable flaws with the method are discovered, it seems likely that machine learning methods will become more popular for PIV, and as that happens, we will likely start seeing implementations that are more portable and user-friendly, as it shifts from being an experimental technique to a new state-of-the art method. It will be interesting to see how the technique develops.

Works Cited

AI Learning. What is Perceptron | Simplilearn. https://www.simplilearn.com/what-is-perceptron-tutorial

Animatlab, 2011. Basic explanation of how a neuron works. http://animatlab.com/Help/Documentation/Neural-Network-Editor/Neural-Simulation-Plug-ins/Firing-Rate-Neural-Plug-in/Neuron-Basics

Simon Baker, Daniel Scharstein, JP Lewis, Stefan Roth, Michael Black and Richard Szeliski. Optical flow evaluation results: Average endpoint error. https://vision.middlebury.edu/flow/eval/results/results-e1.php

Léon Bottou. 2010. Large-Scale Machine Learning with Stochastic Gradient Descent. *Proceedings of COMPSTAT'2010*:177-186. http://khalilghorbal.info/assets/spa/papers/ML_GradDescent.pdf

Shengze Cai, Jiaming Liang, Qi Gao, Chao Xu and Runjie Wei. 2020. Particle Image Velocimetry Based on a Deep Learning Motion Estimator. *IEEE Transactions on Instrumentation and Measurement*, 69(6):3538-3554. https://ieeexplore.ieee.org/document/8793167

Shengze Cai, Shichao Zhou, Chao Xu and Qi Gao. 2019. Dense motion estimation of particle images via a convolutional neural network. *Experiments in Fluids*, 60(4). https://github.com/shengzesnail/PIV_dataset

Krzysztof J. Cios. 2017. Deep Neural Networks—A Brief History. *Advances in Data Analysis with Computational Intelligence Methods*:183-200. https://arxiv.org/ftp/arxiv/papers/1701/1701.05549.pdf

Dana Dabiri. Cross-Correlation Digital Particle Image Velocimetry – A Review. https://www.aa.washington.edu/sites/aa/files/faculty/dabiri/pubs/piV.Review.Paper.final.pdf

Jeffrey Dastin. 2018. Amazon scraps secret AI recruiting tool that showed bias against women. https://www.reuters.com/article/us-amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G

Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers and Thomas Brox. 2015. FlowNet: Learning Optical Flow with Convolutional Networks. *2015 IEEE International Conference on Computer Vision (ICCV)*. https://arxiv.org/pdf/1504.06852.pdf

Vincent Dumolin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. https://github.com/vdumoulin/conv_arithmetic (Source for convolution images)

Martin T Hagan, Howard B Demuth, Mark Hudson Beale and Orlando De Jesús. 2016. *Neural network design*. 2nd edition.k http://hagan.okstate.edu/NNDesign.pdf

Martin Heller. 2018. What is CUDA? Parallel programming for GPUs. https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html

Tak-Wai Hui, Xiaoou Tang and Chen Change Loy. 2018. LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/1805.07036

Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy and Thomas Brox. 2017. FlowNet 2.0: Evolution of Optical Flow Estimation with Deep Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://ieeexplore.ieee.org/document/8099662

Terro Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen and Timo Aila. 2020. Analyzing and Improving the Image Quality of StyleGAN. *roceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*:8110-8119. https://arxiv.org/abs/1912.04958

Deiderik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *3rd International Conference for Learning Representations*. https://arxiv.org/abs/1412.6980

Melanie Lefkowitz. 2019. Professor's perceptron paved the way for AI – 60 years too soon. https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon

Etienne Memin, J Carlier, B Wieneke and F Sarcano. 2021. FLUID Project. http://fluid.irisa.fr/data-eng.htm (DNS Turbulence used in Cai Dataset)

New Scientist. Clarke's three laws. https://www.newscientist.com/definition/clarkes-three-laws/

Michael A Nielsen. 2015. *Neural networks and deep learning*. Determination Press. http://neuralnetworksanddeeplearning.com/chap1.html#learning_with_gradient_descent, http://neuralnetworksanddeeplearning.com/chap2.html)

Simon Niklaus. 2019. A Reimplementation of LiteFlowNet Using PyTorch. https://github.com/sniklaus/pytorch-liteflownet

Keiron O'Shea and Ryan Nash. 2015. An Introduction to Convolutional Neural Networks. https://arxiv.org/pdf/1511.08458.pdf

Florian Raudies. 2013. Optic flow. http://www.scholarpedia.org/article/Optic_flow

Shashank Reddi, Satyen Kale and Sanjiv Kumar. 2018. On the Convergence of Adam and Beyond. *International Conference on Learning Representations*. https://paperswithcode.com/method/amsgrad

V. Resseguier and P. Derian. 2016. Surface Quasi-Geostrophic model under Moderate Uncertainty by vressegu. http://vressegu.github.io/sqgmu/

Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/

Marco Tulio Ribeiro, Sameer Singh and Carlos Guestrin. 2016. "Why Should I Trust You?." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. https://arxiv.org/pdf/1602.04938.pdf

Kaushik Sampath, Thura T. Harfi, Richard T. George and Joseph Katz. 2018. Optimized Time-Resolved Echo Particle Image Velocimetry– Particle Tracking Velocimetry Measurements Elucidate Blood Flow in Patients With Left Ventricular Thrombus. *Journal of Biomechanical Engineering*, 140(4). https://asmedigitalcollection.asme.org/biomechanical/article/140/4/041010/371418/Optimized-Time-Resolved-Echo-Particle-Image?searchresult=1

Daniel Scharstein. 2007. https://vision.middlebury.edu/flow/code/flow-code/README.txt

Avinash Sharma. 2017. Understanding Activation Functions in Neural Networks. https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

Amit Shekhar. 2019. What Are L1 and L2 Loss Functions? https://afteracademy.com/blog/what-are-l1-and-l2-loss-functions

Owen Shen. Perceptrons Explained. https://owenshen24.github.io/perceptron/

Michael Sexton. 2017. All Eyes On Graphics Card Shortage, Few Answers Forthcoming. https://www.tomshardware.com/news/gpu-shortage-2017,34964.html

Wallis. History of the Perceptron. https://web.csulb.edu/~cwallis/artificialn/History.htm

David Watson. 2019. The Rhetoric and Reality of Anthropomorphism in Artificial Intelligence. *Minds and Machines*, 29(3):417-440. https://link.springer.com/article/10.1007/s11023-019-09506-6

Mingrui Zhang and Matthew D. Piggott. 2020. Unsupervised Learning of Particle Image Velocimetry. *Lecture Notes in Computer Science*:102-115. https://arxiv.org/pdf/2007.14487.pdf

Cong-xuan Zhang, Ling-ling Zhu, Zhen Chen, Ding-ding Kong and Xuan Shang. 2017. An Improved Evaluation Method for Optical Flow of Endpoint Err. *Advances in Computer Science Research (ACRS)*, 54:313. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjK0J31nZ_wAhWiMVkFHX49DysQFjABegQIAhAD&url=https%3A%2F%2Fdownload.atlantis-press.com%2Farticle%2F25870830.pdf&usg=AOvVaw10xXUOynUrbYsLV3QUJ9xk

Lina Zhou, Shimei Pan, Jianwu Wang and Athanasios V. Vasilakos. 2017. Machine learning on big data: Opportunities and challenges. *Neurocomputing*, 237:350-361. https://www.sciencedirect.com/science/article/pii/S0925231217300577