CS230

Deep Learning for Efficient Riverine Bathymetry Inversion

Steven Das
Department of Computer Science
Stanford University
scdas@stanford.edu

Shima Salimi Tari
Department of Computer Science
Stanford University
shsalimi@stanford.edu

Abstract

Existing methods for directly measuring or indirectly inferring the bathymetry (depth) profile of a river suffer from both high cost and slow performance (5). This paper proposes a novel deep learning solution to this problem, using both convolutional and fully-connected neural network architectures to infer a river's bathymetry profile from easily-obtained surface velocity measurements. These networks not only predicted the bathymetry profile of a section of the Savannah River near Atlanta, GA, more accurately than a state-of-the-art numerical method, but also reduced the prediction time by four orders of magnitude.

1 Introduction

Shipping, navigation, flood risk assessment, and other maritime activities for a river are greatly assisted by having an accurate bathymetry (i.e., depth) profile of the river (5).

However, direct measurements of riverine bathymetry are often time-consuming and expensive. Numerical methods which attempt to infer a river's bathymetry based on more easily measurable data (e.g., surface velocity profiles) have been proposed to work around these difficulties, but can suffer from slow performance (5).

This project uses a combination of fully-connected and convolutional neural networks to improve upon both the accuracy and runtime performance of one of the most recently developed numerical methods for riverine bathymetry inversion, PCGA (principal component geostatistical approach) (5). Our neural network uses the exact same inputs and outputs as the PCGA numerical model: it takes in the surface velocity profile (i.e, the X and Y components of the water velocity at various (X, Y) coordinates on the river's surface) and some related boundary conditions and outputs the river's bathymetry profile (i.e., the depth Z at each point (X, Y) in the river).

2 Related Work

Literature review suggests that deep learning has never previously been used to infer riverine bathymetry from surface velocity profiles. The most related papers fall into one of 2 categories:

Riverine bathymetry inference via surface velocity: (5) represents the current state-of-the-art in this problem domain by employing the PCGA (principal component geostatistical analysis) first proposed in (4) to improve upon both the final RMSE and computational cost achieved by the ensemble-based solution in (9). Both (5) and (9) employ numerical analytical methods rather than deep learning or any other kind of machine learning.

Oceanic bathymetry inference via neural networks: As early as 1998, (3) used a relatively small fully-connected, feed-forward neural network to predict oceanic depth from Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) images, with reported RMSE ranging from 0.39m to 0.84m. One interesting limitation of this approach - which this project aimed to improve upon - is that predictions were only made on a pixel-by-pixel basis for the input image, rather than attempting to learn using data from nearby pixels. More recently, (7) claimed to beat the performance of two existing inversion and regression tree models for inferring oceanic

CS230: Deep Learning, Fall 2019, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

bathymetry from satellite data using both an MLP (multi-layer perceptron) network and a GRNN (generalized regression neural network).

We used (5) as the baseline for this project given its superior (but slow) performance in our exact problem domain.

3 Dataset and Features

Identical to (5), our dataset consisted of synthetic data generated by running the U.S. Army Corps of Engineers' AdH library (1) on the actual bathymetry profile of a section of the Savannah River near Atlanta, GA.

Each sample contained the following features:

1. *mesh* ***(20541 x 2 matrix)***: The x and y coordinates of each depth and velocity measurement.
2. Z ***(20541 x 1 vector)***: The depth at each point in *mesh*.
3. $velocity_{prof}$ ***(41082 x 1 vector)***: The x and y surface velocity components at each point in *mesh*, with white Gaussian noise added to simulate real-world measurement error.
4. Q_b ***(scalar)***: Volumetric flow, used as a boundary condition.
5. z_f ***(scalar)***: Free surface elevation, used as a boundary condition.

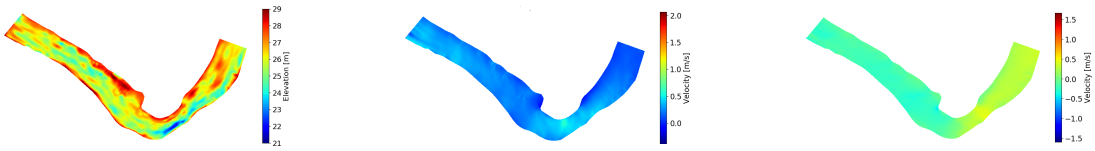

Figure 1: From left to right: 1) True depth Z, 2) Surface velocity (x-component), 3) Surface velocity (y-component).

The high computational cost of generating this synthetic data resulted in our only having 851 samples by the end of the project. As such, we employed a traditional 60/20/20 train/dev/test split of our data, rather than a more lopsided train/dev/test split (e.g., 98/1/1) as might be expected for larger datasets.

Our preprocessing consisted mainly of reshaping the data to be suitable for input to each of our candidate architectures, the details of which will be discussed in the **Methods** section. Some input normalization was attempted, but was not found to substantially improve results.

4 Methods

4.1 Metrics and Loss Function

Similar to (5), we aimed to optimize two metrics: 1) RMSE for measuring the accuracy as following 2) Runtime performance in terms of both training time and prediction time.

$$J(x) = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{m}} \tag{1}$$

We initially used an MSE loss function for our network, as this enabled gradient descent to directly reduce our final RMSE metric on each iteration. To our surprise, we discovered that in some cases an MAE loss function as described in (8) outperformed an MSE loss function in one of two ways:

1. It resulted in final lower RMSE values (all other hyperparameters held constant).
2. It produced meaningful numerical results in situations where MSE output *NaN*s.

4.2 Architectures and Algorithms

The following three approaches were investigated in the most depth:

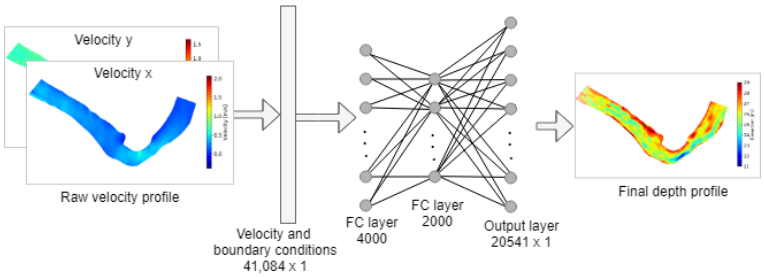

Figure 2: Fully connected architecture over the vectorized velocity and boundary conditions.

4.2.1 Fully Connected

Our simplest approach involved concatenating each sample's velocity vector with the boundary condition scalars, and then running the combined vector through a series of fully-connected layers until reaching the output layer:

The primary appeal of this architecture, besides its simplicity, was how it theoretically enabled every point at which we needed to predict the depth to learn from every point on the river's surface. Due to the large dimensions of the input and output vectors, it was necessary for the hidden layers to have much smaller dimensions than the input and output layers (typically by an order of magnitude). Larger hidden layers tended to result in out-of-memory errors.

4.2.2 2D Convolution

This architecture capitalizes on a crucial detail from (5) explaining that the points in *mesh* can be deterministically mapped to a 41 x 501 rectangular grid for which 1) the *x*-axis represents distance traveled longitudinally down the river and 2) the *y*-axis represents transverse distance traveled across the river. We realized that this transformation made it possible to reshape each *(41082 x 1)* velocity vector into a *(41 x 501 x 2)* tensor in which each channel held a single velocity component, as shown in Figure 3.

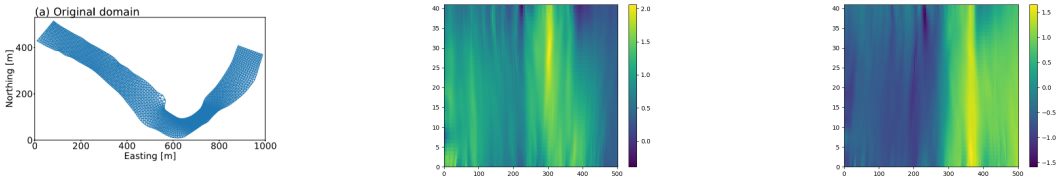

Figure 3: From left to right: 1) The original triangular mesh. 2) Rectangular transformation of the *x* velocity components. 3) Rectangular transformation of the *y* velocity components. The reader is encouraged to compare these plots to the original triangular mesh plots in Figure 6.

At this point, we were able to build a 2D convolutional neural network (Figure 4) accepting these reshaped tensors as inputs. Boundary conditions were input to the network by concatenating them to the **Flatten** layer.

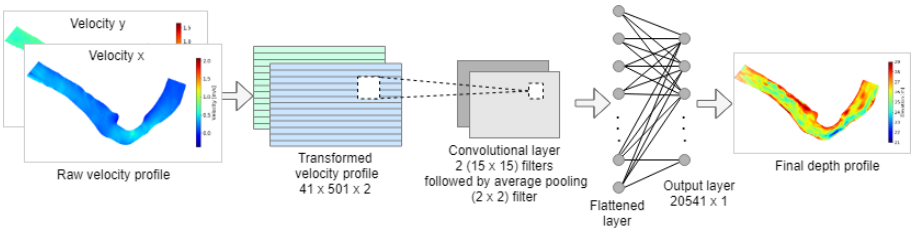

Figure 4: 2D convolutional architecture over a rectangular mesh.

Relative to the fully-connected approach, this approach enabled us to maximize the odds of training on spatial locality effects in our data while reducing the overall number of parameters to train.

4.2.3 1D Convolution

Intuitively, we expected the surface velocity components parallel to the flow of the river (i.e., the x velocity components in the rectangular mesh) to contribute more significantly to the final predictions than the surface velocity components perpendicular to the river's flow.

Curious whether this might improve training or prediction speed at the possible expense of some accuracy, we attempted to train a 1D convolutional neural network (Figure 5) by reshaping each *(41082 x 1)* velocity vector into a *(20541 x 1 x 2)* tensor using velocity components as channels.

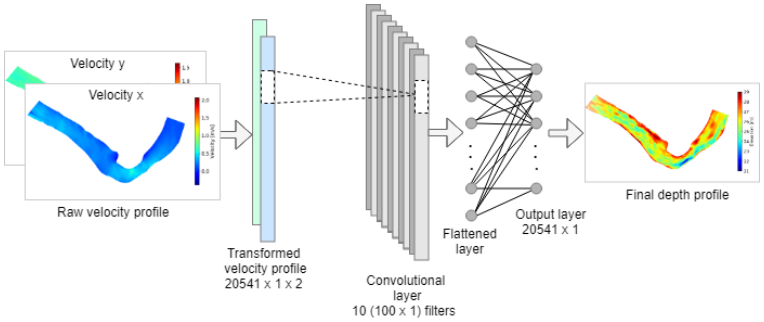

Figure 5: 1D convolutional architecture over the vectorized mesh.

An understood weakness of this architecture is that depending on the choices of filter size and stride, it may "wrap around" and convolve over sections of the river which are not physically contiguous.

5 Experiments/Results/Discussion

5.0.1 Final Results

Table 1 shows the best results for each architecture implemented using Keras library in Tensorflow 2 (6). The results were produced using an Amazon Web Services (AWS) p3.2x large VM running version 25.3 of the Deep Learning AMI (Ubuntu 18.04) with 61 GB of RAM, 8 VCores, and an NVIDIA Tesla V100 GPU.

Architecture	Train RMSE (m)	Dev RMSE (m)	Test RMSE (m)	Training Time (s)	Prediction Time Per Sample (s)
Fully Connected	0.388	0.584	0.268	593.701	0.121
2D Convolutional	0.378	0.570	0.258	911.767	0.139
1D Convolutional	0.254	0.563	0.271	1131.783	0.133
PCGA (baseline)	-	-	0.7	-	1 hour

Table 1: Final results for each architecture.

All 3 algorithms beat the 0.7 m RMSE figure reported in (5). Far more significant were the improvements in prediction speed: whereas PCGA required roughly 1 hour to predict a single sample, our models predicted the same in roughly a tenth of a second (a speedup of 4 orders of magnitude).

Two important qualifications on comparing these results to (5) are as follows:

1. (5) employed a Linux workstation with more RAM (128 GB) and cores (36) than our project; however, (5) does not appear to have used a GPU.
2. (5) used only 4% (816 of 20,541) of the points in *mesh*, whereas our project used them all.

5.0.2 Hyperparameter Choices

The exact architectures used for each approach were displayed in the **Methods** section. Other hyperparameters selected for each of these approaches were as follows:

1. **Fully-Connected:** Minibatches of size 32 for 200 epochs, followed by full batch for 5000 epochs. Dropout rate of 0.2 in hidden layers.

2. **2D Convolutional:** 10 15 x 15 filters with stride 3 and same padding in the convolutional layer, filter size 2 and stride 2 in the average pooling layer. Minibatch size of 32, 3000 epochs. Dropout rate of 0.4 in hidden layers.
3. **1D Convolutional:** 10 100 x 1 filters with stride 10 and same padding. Minibatches of size 32 for the first 200 epochs, followed by full batch for the final 5000 epochs. 3000 epochs. Dropout rate of 0.4 in hidden layers. Training set included error samples from dev set.

All 3 architectures provided an initial learning rate of 0.2 to the Adam optimizer, and used batch normalization on all layers except the output layer.

Generally speaking, we tuned our hyperparameters via *ceterus parabus*: one hyperparameter at a time was varied while holding all others constant, and the results on both the train and dev set were used to determine whether to retain the modified hyperparameter.

Some of the more significant discoveries during hyper-parameter tuning were as follows:

1. It was critical to pick the right batch size to avoid divergence, even with Adam's variable learning rate. Ultimately, we switched from small mini-batches to full batches after only 200 epochs.
2. We also experimented with custom learning rate decay to avoid divergence and speed up training. Ultimately, this performed no better than Adam's built-in learning rate decay.
3. Due to our large input and output vectors, memory limits heavily constrained our hyperparameter choices. For example: slight increases in fully-connected layer size or convolutional filter count, or slight decreases in stride, all resulted in out-of-memory errors.
4. Other hyperparameters with nontrivial effects on our results included 1) number of epochs 2) presence of batch normalization, 3) convolutional filter size (larger worked better), and 4) dropout percentage.
5. Hyperparameters which proved mostly irrelevant included 1) whether we or not we included the boundary conditions (in contrast to the expectations of many methods of numerical analysis), 2) weight initialization, and 3) input normalization.

Some hyperparameters we did not bother to tune, such as the choice of activation function or optimizer (i.e., we always used ReLU and Adam, respectively).

5.0.3 Error Analysis

Error analysis consisted of manually inspecting the dev set for inaccurate predictions similar to Figure 6 and adding them to the training set to achieve a better prediction with lower RMSE for those samples.

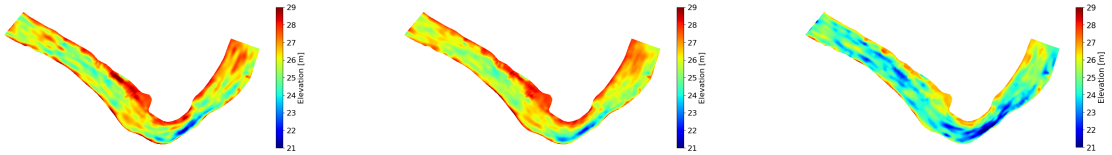

Figure 6: From left to right: 1) true depth and 2) prediction with low RMSE, 3) prediction with high RMSE.

6 Conclusions/Future Work

All 3 algorithms surpassed RMSE values obtained via PCGA in (5), while improving per-sample prediction time from roughly one hour to around a tenth of a second. The test set results were close enough that it is not yet clear which of these 3 algorithms will perform best on accuracy in the long run with further iterations of the architecture and hyperparameters.

Given that hyperparameter tuning was sharply constrained by memory, an obvious next step will be to iterate these architectures on a machine with more memory. It will be especially interesting to see how this benefits the fully-connected architecture: even with its layer sizes heavily constrained by memory, this architecture essentially matched the convolutional networks on accuracy and clearly beat them on training and prediction speed.

Other obvious next steps include 1) generating and training on larger synthetic datasets, 2) training on noisier real-world data, and 3) deploying models generated by the latter into the field, where the maritime industries and communities which benefit from accurate riverine bathymetry surveys may most readily benefit from their speedy predictions.

7 Contributions

Contributions to this project were essentially equal among group members. Some details on contributions are as follows:

Steven: Virtualenv setup, Linux and GPU administration, Tensorflow 2 prototyping, data plotting and visualization, fully-connected implementation, 2D convolution implementations and associated data preprocessing, literature review, report, code review, proposal, milestone.

Shima: Keras prototyping, 1D convolution implementation and associated data preprocessing, 2D convolution implementations and associated data preprocessing, hyperparameter tuning (layers, batch normalization, filter size, learning rate, etc.), error analysis, report, poster, code review, proposal, milestone, literature review.

8 Acknowledgments

We would like to thank Hojat Ghorbanidehno for sponsoring this project, for his detailed mentorship, and for generating the synthetic data used in this project.

We would also like to thank Huizi Mao and Mohammed Mahmoud for their detailed and enthusiastic mentorship throughout the quarter.

Finally, we would like to thank Jonghyun Lee for offering suggestions on the final paper, answering questions about the source code of (5), and introducing us to other authors of the paper.

9 Code

The source code for this project can be found in a private GitHub repository at https://github.com/DasAllFolks/CS230.

Please contact the authors if you would like to review the repository.

References

[1] Adaptive Hydraulics, https://chl.erdc.dren.mil/chladh, Accessed: October 6, 2019.

[2] Ghorbanidehno, H., J. Lee, M. Farthing, T. Hesser, P.K. Kitanidis, and E.F. Darve, 2019: Novel Data Assimilation Algorithm for Nearshore Bathymetry. J. Atmos. Oceanic Technol., 36, 699–715, https://doi.org/10.1175/JTECH-D-18-0067.1

[3] Juanita C. Sandidge, Ronald J. Holyer, Coastal Bathymetry from Hyperspectral Observations of Water Radiance, Remote Sensing of Environment, Volume 65, Issue 3, 1998, Pages 341-352, ISSN 0034-4257, https://doi.org/10.1016/S0034-4257(98)00043-1.

[4] Kitanidis, P. K., and Lee, J. (2014), Principal Component Geostatistical Approach for large-dimensional inverse problems, Water Resour. Res., 50, 5428– 5443, doi:10.1002/2013WR014630.

[5] Lee, Jonghyun Ghorbanidehno, Hojat Farthing, Matthew Hesser, Tyler Darve, Eric Kitanidis, Peter. (2018). Riverine Bathymetry Imaging With Indirect Observations. Water Resources Research. 54. 10.1029/2017WR021649.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Shan Liu, Yong Gao, Wenfeng Zheng Xiaolu Li (2015) Performance of two neural network models in bathymetry, Remote Sensing Letters, 6:4, 321-330, DOI:10.1080/2150704X.2015.1034885

[8] MAE and RSMEhttps://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d, Accessed: December 8, 2019.

[9] Wilson, G., Ozkan-Haller, H. T. (2012). Ensemble-based data assimilation for estimation of river depths. Journal of Atmospheric and Oceanic Technology, 29(10), 1558–1568.