Supplemental Material for Accelerated Probabilistic Marching Cubes by Deep Learning for Time-Varying Scalar Ensembles

Mengjiao Han[*]
Scientific Computing and Imaging Institute

Tushar M. Athawale[†]
Oak Ridge National Laboratory

David Pugmire[‡]
Oak Ridge National Laboratory

Chris R. Johnson[§]
Scientific Computing and Imaging Institute

1 Introduction

In this supplemental material, we first explain the training and inference process of our deep learning-based method in Sect. 2. We further add more qualitative experiments to compare the range of normalization (Sect. 3.2) and the activation functions (Sect. 3.1).

2 Training and Inference Process

The training process is shown in algorithm 1. We exploited the k-fold cross-validation procedure during the training process to split the data sets into training and testing sets for a more stable and robust result. We normalized the input *vector_mean* and *vector_cov* to the range of $[0,1]$ separately while loading the training set. The normalization speeds up the learning for the network as well as leads to faster convergence.

```
Input: Data set shown in Equation 2
       Initial weights of the network $w$
Output: Optimized weights $w$
Load and normalize training samples
Divide data set into $K$ folds
for fold $k_i$ in the $K$ folds do
    Set fold $k_i$ as the test set
    Set remaining $K-1$ folds as training set
    for each epoch do
        for each batch of training set do
            model.train()
            $pred = model(vector\_m, vector\_cov, vector\_iso)$
            $loss = L2\_Loss(pred, target)$
            Backpropagation and update weight $w$
        end
        for each batch of validation samples do
            model.eval()
            $pred = model(vector\_m, vector\_cov, vector\_iso)$
            $loss = L2\_Loss(pred, target)$
        end
        Call the learning rate scheduler
        Adjust the learning rate, if needed
    end
end
```

Algorithm 1: Training Process

We implemented the neural network using PyTorch [4]. The backpropagation process and the weights optimization are done by PyTorch automatically. We used the Adam optimizer [3] with the hyperparameters of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 1\text{e}{-6}$. Furthermore, we set the initial learning rate as $1\text{e}{-5}$ and used a learning rate scheduler [1] to reduce the learning rate by a factor of 2 if the testing loss has not decreased for five epochs. We used the mean square error (MSE) as the loss function.

```
Input: single time step 2D ensemble data
Input: isovalue $s$
Input: $min\_mean$, $max\_mean$, $min\_cov$, and $max\_cov$ for normalization
Output: predicted level-crossing probabilities $out$
Step 1: Load Data
List $mean\_list$; Empty list for saving mean values for each cell
List $cov\_list$; Empty list for saving variances and covariances for each cell
List $indices$ Empty list for saving indices of cells that contain the isovalue
for each 2D cell do
    Get values of $Y_0, Y_1, Y_2, Y_3$ at each grid vertex of a cell
    $min\_value$ = MIN($Y_0, Y_1, Y_2, Y_3$)
    $max\_value$ = MAX($Y_0, Y_1, Y_2, Y_3$)
    if $min\_value \le s \le max\_value$ then
        Calculate mean values and covariance matrix
        Normalize mean values using $min\_mean$ and $max\_mean$
        Normalize variance and covariance values using $min\_cov$ and $max\_cov$
        Push mean values to the $mean\_list$
        Push variances and covariances to the $cov\_list$
        Push index of the cell to $indices$
    end
end
Construct PyTorch dataloader using the $mean\_list$, $cov\_list$ and $s$
Step 2: Load Pre-trained Model
Step 3: Compute
for each batch from the dataloader do
    $pred = model(vector\_m, vector\_cov, vector\_iso)$
end
Initialize all values in $out$ to be zeros
Predict the probabilities for cells that contain the isovalue using the saved $indices$.
```

Algorithm 2: Inference Process

During the inference process (algorithm 2), to predict the level-crossing probabilities for a single time step ensemble data, the first step is to compute and normalize the vector of means (*vector_mean*) and the vector of variances and covariances (*vector_cov*) for each cell. After loading the trained model, the *vector_mean*, *vector_cov*,

[*]e-mail: mengjiao@sci.utah.edu
[†]e-mail: athawaletm@ornl.gov
[‡]e-mail: pugmire@ornl.gov
[§]e-mail: crj@sci.utah.edu

[1]`https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html`

and *vector_iso* are converted to the PyTorch tensor to feed into the trained model.

3 More Qualitative Results

We compared the network performance with a different activation function (Sect. 3.1) and normalization range (Sect. 3.2).

3.1 Comparison of the Activation Function

Our neural network architecture was adapted from the approach proposed by Han et al. [2]. Their network architecture used ReLU [1] as the activation function. Sitzmann et al. [5] demonstrated the sinusoidal activation function to be more accurate and faster. In our network, we applied the sinusoidal activation function after each fully connected layer except the last layer of the latent decoder D. We evaluated the accuracy of using the ReLU activation function with the sinusoidal activation function for our task. As Fig. 2 shows, the ReLU provides a more accurate result for the **Red Sea** data set only compared to the sinusoidal activation function, which performs better for the **Wind** and **Temperature** data set.

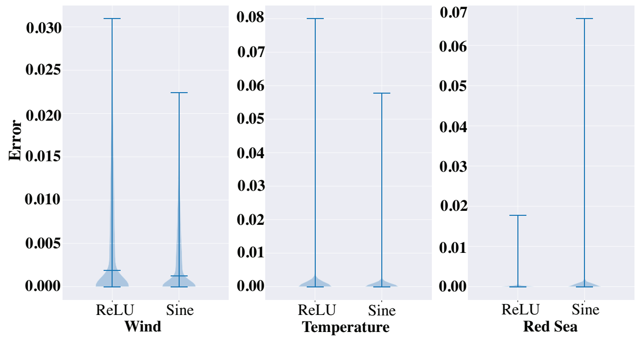

Figure 1: The Violin plots visualize the errors for the model trained with ReLU and sinusoidal activation functions. The errors are calculated as the pixel-wise absolute differences between model-predicted results and the ground truth. The model was trained with about 60% of each data set. The evaluations are performed using one-time step testing data for each data set. The outlier error values are not displayed in the plots. The results show that the sinusoidal activation function is more accurate for the **Wind** and **Temperature** data sets, and ReLU is more accurate for the **Red Sea** data set. In our method, we used the sinusoidal activation function.

3.2 Comparison of the Normalization Range

Scaling normalization is a common method in deep learning to normalize the range of data features. This process is essential to make the optimization faster because the weights are restricted in range. Two typical normalization ranges are $[0,1]$ and $[-1,1]$. Fig. 2 presents the errors resulting from applying these two ranges in the normalization process. Normalizing to the range $[0,1]$ is better for the **Wind** and **Temperature** data set, and the range $[-1,1]$ is more accurate for the **Red Sea** data set.

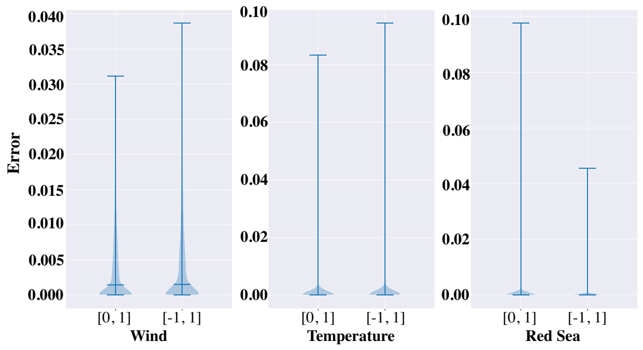

Figure 2: Violin plots visualizing the errors for the model trained with data normalization to ranges $[0,1]$ and $[-1,1]$. The errors are calculated as the pixel-wise absolute differences between model-predicted results and the ground truth. The model was trained with about 60% of each data set. The evaluations are performed using one-time step testing data for each data set. The outlier error values are not displayed in the plots. The results show that the range $[0,1]$ performs better than $[-1,1]$ for the **Wind** and **Temperature** data sets, and $[-1,1]$ is more accurate for the **Red Sea** data set. We chose range $[0,1]$ as the normalization range in our experiments.

References

[1] A. F. Agarap. Deep Learning using Rectified Linear Units (ReLU). *arXiv preprint arXiv:1803.08375*, 2018.

[2] M. Han, S. Sane, and C. R. Johnson. Exploratory Lagrangian-Based Particle Tracing Using Deep Learning. *Journal of Flow Visualization and Image Processing*, 2022. doi: 10.1615/JFlowVisImageProc.2022041197

[3] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. doi: 10.48550/arXiv.1412.6980

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in neural information processing systems*, 32, 2019. doi: 10.48550/arXiv.1912.01703

[5] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. Implicit Neural Representations with Periodic Activation Functions. *Advances in Neural Information Processing Systems*, 33, 2020. doi: 10.48550/arXiv.2006.09661