Learning error in LES

By Xinyi Huang, 09/2026

Caltech

Division of Engineering
and Applied Science

Pytorch training customize – 3

- Autograd dimension
  - can have Jacobian product and Jacobian matrix, use *tensor.backward*(gradient=v), v is the same shape of tensor, leads to a summation of v*J, if multiple outputs are using the same input, it might need to be separately computed.
  - *backward()* populates from the root nodes, which is the final output, and creates gradient at the leaf nodes, which is the original input that requires gradients. → *loss.backward()*, *print(inputs.grad)*
  - Actually, *forward*(ctx, input1, param1) takes any amount of inputs of any shape, and use them as you want to produce any amount of ouputs of any shape. *backward*(ctx, grad_output1, grad_output2) takes grad_outputs the same shape as outputs, and use it as you want to produce grad_input1, grad_param1 of the same shape in *forward()*.
  - *backward*(gradient=v) feeds in grad_output = v. It can used in whatever way, either use summation or only use single value to obtain multiple repeats of the derivatives.
  - Use summation: inputs $n_1 \times n_2 \times \cdots \times n_p$, outputs/grad_output $m_1 \times m_2 \times \cdots \times m_q$, grad_inputs $n_1 \times n_2 \times \cdots \times n_p$ combines all $\sum_{j_1,j_2,\ldots,j_q} v_{j_1,j_2,\ldots,j_q} \cdot \partial out_{i_1,i_2,\ldots,i_p} / \partial in_{j_1,j_2,\ldots,j_q}$
  - Use single value: inputs $n_1 \times n_2 \times \cdots \times n_p \times l_1 \times l_2 \times \cdots \times l_r$, outputs/grad_output $m_1 \times m_2 \times \cdots \times m_q \times l_1 \times l_2 \times \cdots \times l_r$, grad_inputs $n_1 \times n_2 \times \cdots \times n_p \times l_1 \times l_2 \times \cdots \times l_r$combines all $\sum_{j_1,j_2,\ldots,j_q} v_{j_1,j_2,\ldots,j_q,k_1,k_2,\ldots,k_r} \cdot \partial out_{i_1,i_2,\ldots,i_p,k_1,k_2,\ldots,k_r} / \partial in_{j_1,j_2,\ldots,j_q,k_1,k_2,\ldots,k_r}$ for each repetition of $k_1, k_2, \ldots, k_r$.

Parallelization

- Mainly using torch.distributed (CPU is supported by gloo), torch.multiprocessing (compatible to original multiprocessing), torchrun (wrapper of torch.distirbuted)
  - DDP (distributed data-parallel training) is only for training
  - DistributedSampler is for distributed data.
  - https://pytorch.org/docs/stable/distributed.html
  - https://stackoverflow.com/questions/75017751/using-multiple-cpu-in-pytorch
- The default has num_threads = 32, and num_interop_threads=16
  - Does that automatically improve performance?
- [Possible to-do list] Consider parallelization when with memory issue, or speed issue.

Improve training

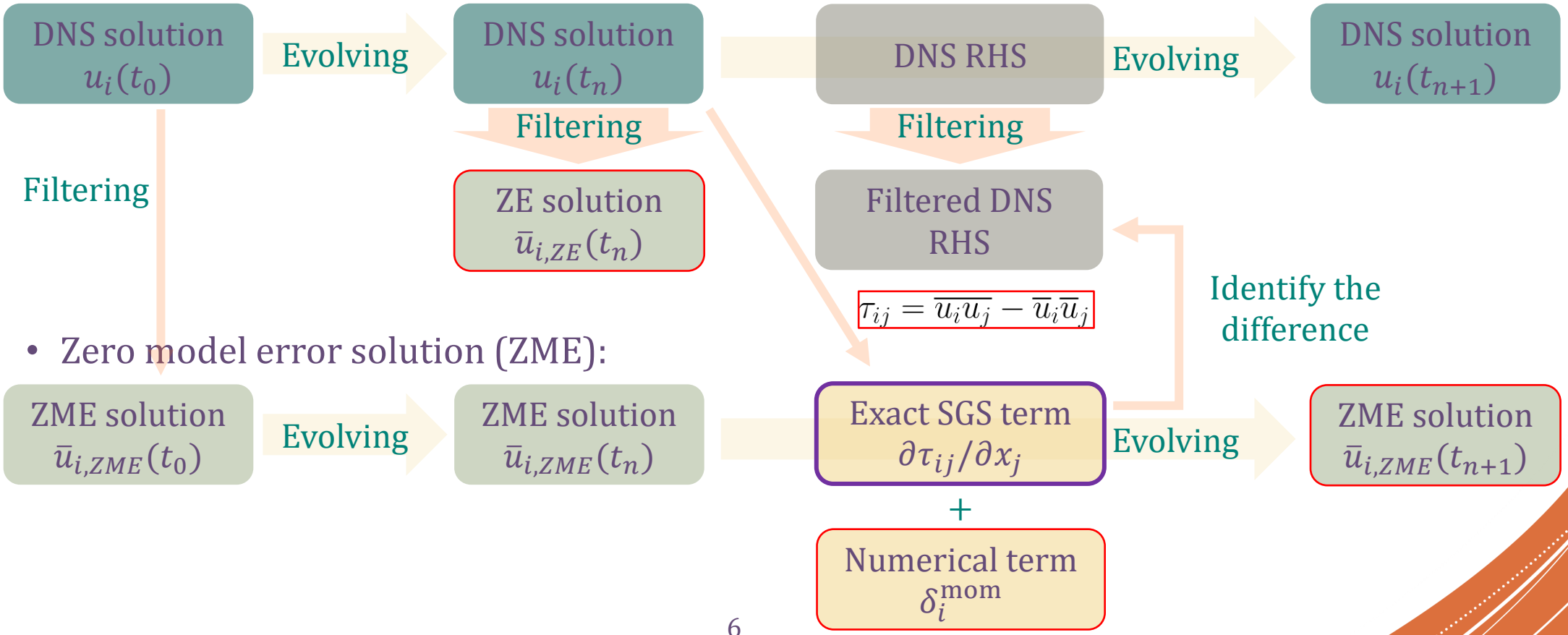
- Adam operator:
  - Benefit: https://www.kdnuggets.com/2022/12/tuning-adam-optimizer-parameters-pytorch.html
- L-BFGS:
  - Reference performance: https://johaupt.github.io/blog/pytorch_lbfgs.html
  - Batch ver: https://sagecal.sourceforge.net/pytorch/index.html
- Speed test:
  - For Pytorch: https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2
  - fast.ai: https://www.fast.ai/posts/2018-07-02-adam-weight-decay.html

Results of known functions

- Redid the training for known functions as linear relationship using Adam in Pytorch, to understand the behavior of training when the data has specific characteristics.
- Some conclusions:
  - Using Adam has a much better performance than SGD, but still worse than second-order training methods as Levenberg-Marquardt algorithm as in Matlab.
    - $x \in [0, 0.4]$ and other 5 irrelevant inputs. All with white noise ~2%
    - Use $y = \sin(p_1 x)$ is better, as in Matlab LM $y = \sqrt{x}$ can use linear function to approximate well, with err ~ 3% and corr ~ 0.989; For $y = \sin(10x)$ the err ~ 11% and corr ~ 0.916, because $x > 0.2$ is below 0.1% of dataset.
    - e.g. Adam converges for $y = \sin(10x)$ and $y = \sqrt{x}$, err~0.01, while SGD has 0.2 error rate for $y = \sqrt{x}$.
    - e.g. Adam does not converge for $y = \sin(100x)$ while Matlab LM converges and err ~ 1e-4, though Matlab LM cannot predict $y = \sin(1000x)$
  - In general, NNs are not good at predicting periodicity
    - e.g., Adam does not converge for $y = \sin(100x)$ and Matlab LM not converge for $y = \sin(1000x)$
    - e.g., Matlab does not extrapolate for predicted $y = \sin(100x)$
  - inorm=3 works the best, which has I_1 and I_2 generally the largest values.
    - e.g. for inorm=1, other I are comparative to I_1; for inorm=2, other I are larger magnitude than I_1. Neither of them can train a working NN. Their being large ($I_{3,4,5,6}$) will be a sensitive factor for training and predicting.
  - The impact of itar, with inorm=3, $y = \sin(10x)$; all predicting well and with high corr of cost function quantity and other quantities.
    - itar = 1, stress tensor as target, $L(\tau_{ij})$~0.010, while $L(S_{ij}\tau_{ij})$~0.0070; ($L(Q) = \sqrt{MSE(Q_M, Q_T)}/||Q_T||_2$ is the relative loss function)
    - itar = 2, $\nu_{T,\text{effective}}$ as target, $L(\nu_T)$~0.0070, and $L(S_{ij}\tau_{ij})$~0.0070;
    - itar = 3, $S_{ij}\tau_{ij}$ as target, $L(S_{ij}\tau_{ij})$~0.0071, and $L(\nu_T)$~0.0080.
  - The impact of LR scheduler has not been explored.
    - Current results are not stable, as it does not converge at all in some settings and L~1 for it, instead of somewhere between 0 and 1.

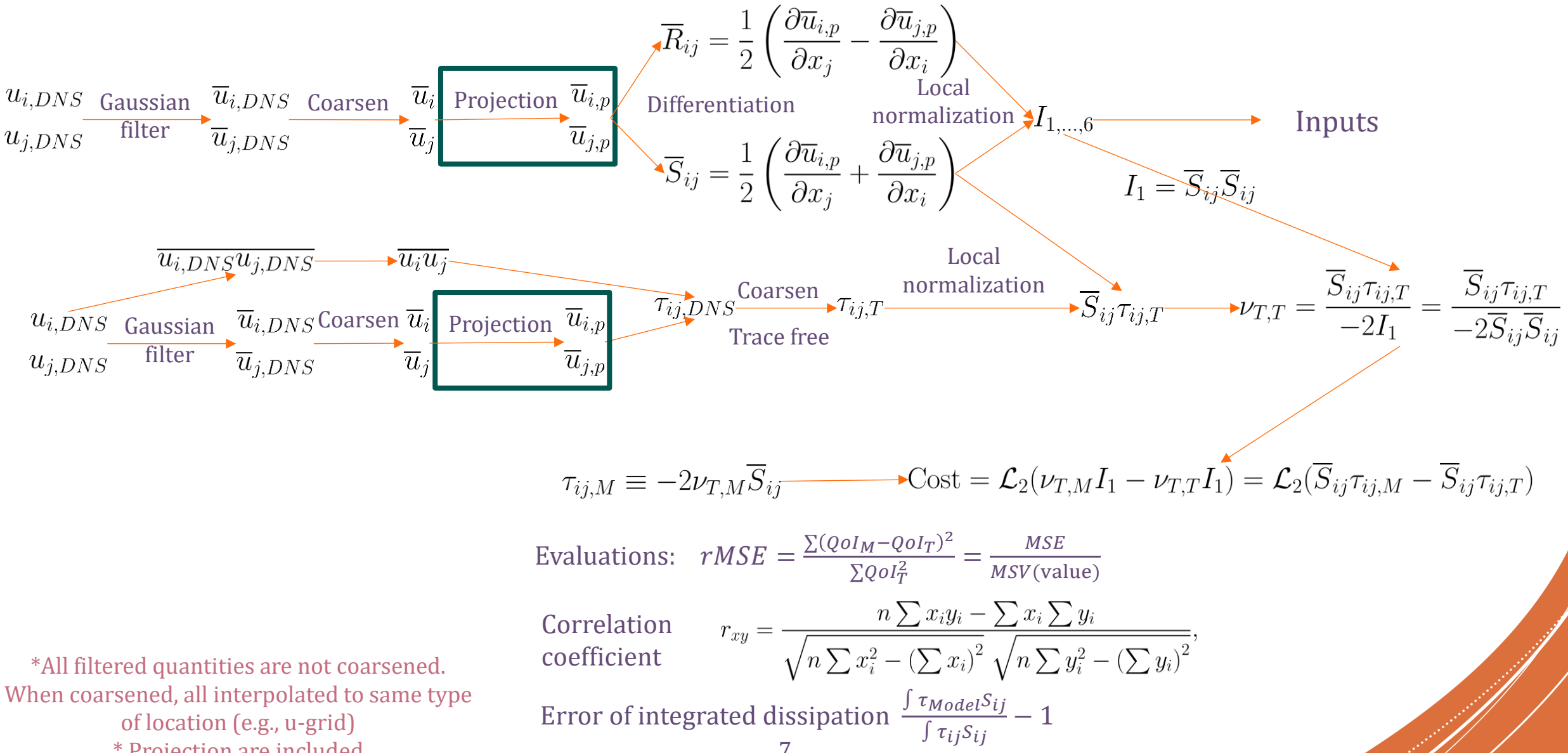
Datasets available for modeling

- Input data: R & S
- Output data:
  - Zero error solution (ZE):



  - Zero model error solution (ZME):

Trainable dataset – All from DNS



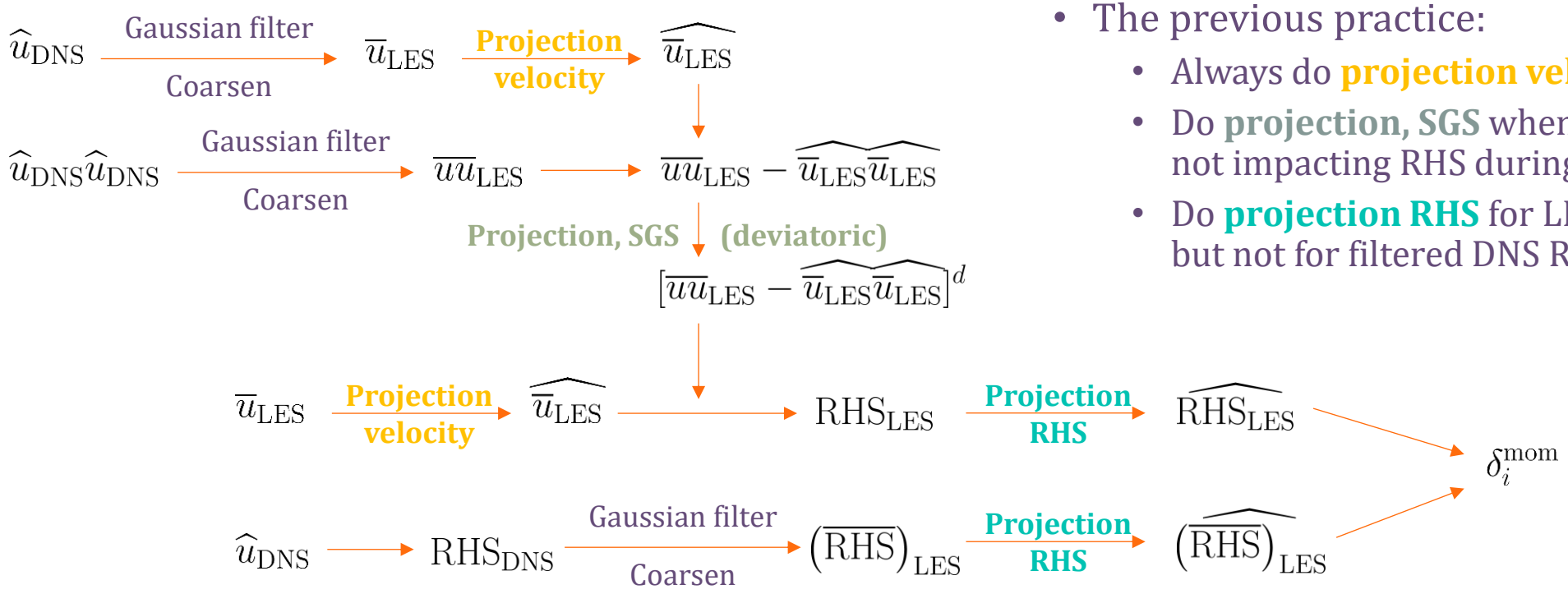
Evaluations: $rMSE = \frac{\sum(QoI_M - QoI_T)^2}{\sum QoI_T^2} = \frac{MSE}{MSV(\text{value})}$

Correlation coefficient $r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$,

Error of integrated dissipation $\frac{\int \tau_{Model} S_{ij}}{\int \tau_{ij} S_{ij}} - 1$

*All filtered quantities are not coarsened.
When coarsened, all interpolated to same type of location (e.g., u-grid)
* Projection are included

Projection process



- The previous practice:
  - Always do **projection velocity**.
  - Do **projection, SGS** when postprocessing, not impacting RHS during data generation
  - Do **projection RHS** for LES eqn (top line) but not for filtered DNS RHS (bottom line).

- Xinyi's current proposed practice:
  - Always do **projection velocity**.
  - Always do **projection, SGS** when data generation
  - Not do **projection RHS** for both steps.

Other impacts of filtering

- For S_{ij},
  - Divergence free: $S_{ij,DNS}$ and $\bar{S}_{ij,DNS}$ has $S_{ii} \approx 0$, but when coarsened, $\bar{S}_{ij,LES}$ is not. Interpolation to center or not does not impact much.
  - For spectral cutoff filter, coarsening method $\widehat{(\cdot)}$ (both $\widehat{(\cdot)}_{interpolation}$ or both $\widehat{(\cdot)}_{spectral}$) needs to be consistent, otherwise $\frac{\partial_{DNS}\bar{u}_i}{\partial x_j}$ and $\frac{\partial_{DNS}u_i}{\partial x_j}$ will also be different.
  - However, $\frac{\partial_{LES}\widehat{\bar{u}_i}}{\partial x_j}$ which is coarsened before derivative will be very different from $\frac{\partial_{DNS}\bar{u}_i}{\partial x_j}$.
  - interpolation to center or not does not change results much.
- Three operators: the derivative $\frac{\partial}{\partial x_j}$, Gaussian/spectral cutoff filter $\overline{(\cdot)}$, and grid filter $\widehat{(\cdot)}$ i.e. coarsening.
  - $\frac{\partial}{\partial x_j}$ and $\overline{(\cdot)}$ are commutative for both Gaussian and spectral cutoff filter
  - Gaussian/spectral filter needs to be done before grid filter in practice, so only three ways are available
  - 1) Gaussian/spectral filter -> grid filter -> derivative (Derivative of filtered $\frac{\partial_{LES}\widehat{\bar{u}_i}}{\partial x_j}$ in LES)

    2) Gaussian/spectral filter -> derivative -> grid filter

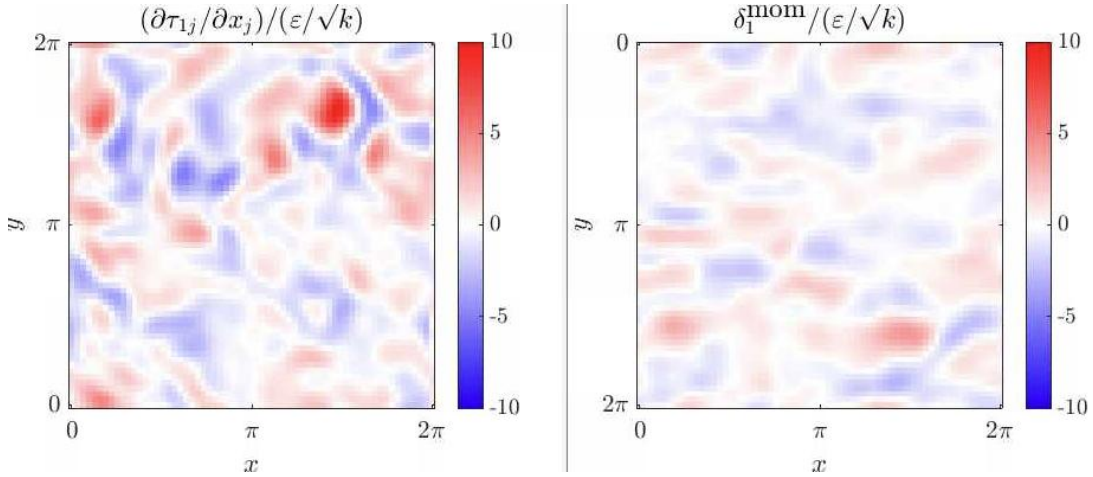
    3) derivative -> Gaussian/spectral filter -> grid filter (Filtered DNS derivative $\frac{\partial_{DNS}u_i}{\partial x_j}$ in DNS)
  - 2) and 3) have similar results when using the same grid filter, but 1) is different from the rest. 1) and 3) are commonly used.

Python wrapper in Fortran

- References:
  - General: https://fortranwiki.org/fortran/show/Python
  - Pytorch-Fortran: https://github.com/alexeedm/pytorch-fortran/tree/main
  - [Will use] FTorch: https://github.com/Cambridge-ICCS/FTorch?tab=readme-ov-file
  - [Will use] TorchScript: https://pytorch.org/docs/stable/generated/torch.jit.save.html#torch.jit.save
- Use FTorch since it is more updated and seems better document
- Process: major issues
  - [Done] Installation on HPC: *Intel compiler has compatibility issue*
  - [Done] Example on HPC
  - [Done] Test with HIT compilation setting: *MPI options require explicit library path*
  - Test the example case with HIT code
  - Coding with HIT code

Details: data generation parameters

- Previous setup for Gaussian $\sigma = 2\Delta_{LES}$ filtered data
  - Linear interpolation during coarsening
  - Invariant calculation, all centered invariants
  - Projection, only projected for initialization
  - Using $\Delta, \nu, |S|$ for normalization
  - No explicit filtering
- Current setup, hopefully standardized in future
  - Spectral interpolation during coarsening
  - Invariant calculation, all staggered invariants
  - Projection, projected initialization, SGS stress (trace free) and all RHS.
  - Using η and u_η for normalization
  - Keep no explicit filtering



- Impacts of parameters
  - Interpolation mainly impacts initialization during a posteriori test
  - Staggered invariant calculation make interpolation reduced from twice to once
  - RHS projection will greatly reduce δ_i absolute value. Current setup still see comparable values
  - Using Δ for normalization has non-unique non-dimensionalization.
  - Explicit filtering impacts greatly so not added at this moment

Summary: what is happening in the constant coefficient?

- The dimensionalization of $\nu_{T,1}$ is ν and for $\nu_{T,2/3/4}$ is $\nu[S]$. Actual NN leads to 4 constants.
  - Examined the predicted output of NN. It is indeed constant, not a mistake of IO.
  - The prediction by the evaluation code and the training code is the same for a single data pt.
  - Examined the prediction process. It will reproduce the same result.
  - The dimension of the data generation is correct, not a problem of shape manipulation.
    (e.g., a post on Reddit mention the auto-broadcasting in training, https://stackoverflow.com/questions/4493554/neural-network-always-produces-same-similar-outputs-for-any-input)
- Curious: Is the training process not correct?
  - Modified based on the verified case, major modification is the IO process and the dimensionalization. No training logic is changed.
  - Try to reproduce some of the training results using artificial functions I had before, i.e., have low error rate with linear function, sqrt, and sine wave. Not even able to train linear function.
  - Current temporary conclusion: change of the dimensionalization leads to difficulties in training (maybe due to the vanishing gradient problem and the inconsistent input range).
    Reasoning: switching from |S| to $\sqrt{|S^2 + R^2|}$ in DIM2 leads to artificial functions trainable; switching from Sigmoid to ReLU leads to artificial functions trainable, etc.

$$\mathbf{m}_1 = \mathbf{S} \qquad \mathbf{m}_3 = \mathbf{R}^2$$
$$\mathbf{m}_2 = \mathbf{S}^2 \qquad \mathbf{m}_4 = \mathbf{SR} - \mathbf{RS}$$

- DIM0: (Verified version)
  - $u_{char} = 10\sqrt{\nu\sqrt{2S_{ij}S_{ij}}}$
  - $l_{char} = 0.2\sqrt{3}\Delta_{LES}$
- DIM1: (sometimes constant)
  (Produce some positive results)
  - $u_{char} = 10\sqrt{\nu\sqrt{2(S_{ij}S_{ij} + R_{ij}R_{ij})}}$
  - $l_{char} = 10\sqrt{\nu/\sqrt{2(S_{ij}S_{ij} + R_{ij}R_{ij})}}$
- DIM2: (sometimes constant)
  - $u_{char} = \Delta_{LES}\sqrt{S_{ij}S_{ij}}$
  - $l_{char} = \sqrt{3}\Delta_{LES} = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$

Though not proved, but I think it could be proved that

$|tr(ABC\cdots)| \le \|A\|\|B\|\|C\|\cdots$ $|tr(I_5)| \le \|S\|^2\|R\|^2 \le (\|S\|^2 + \|R\|^2)^2$ $|tr(I_6)| \le \|S\|^3\|R\|^3 \le (\|S\|^2 + \|R\|^2)^3$

https://math.stackexchange.com/questions/2241879/reference-for-trace-norm-inequality

https://mathoverflow.net/questions/78330/bounding-the-trace-of-a-matrix-product-by-the-operator-norms-generalized-h%C3%B6lder

Combining Holder inequality further for the final conclusion.

Details: Debugging process on and before Feb. 14th

- Previously achieved:
  - Possibly due to the constant input of $I_1 = 1$ lead to difficulty in convergence.
  - The constant value of output can be reproduced
  - The error rate is tested for small dataset, and uses easy cases as sine wave and linear value with no hidden layers.
  - Increasing dataset size does not change much.
  - The testing and the evaluation data in the training code are compatible with each other. They are also compatible with the artificial values.
  - The testing data in the training code and the evaluation data in the evaluation code is the same.
- Each hr conclusion:
  - Hr #1: a 1% err for pure linear regression. The constant input maybe the issue.
  - Hr #2: For linear regression, the change of the coeffs is not dependent on gradient for first step and also a bit on next steps.
  - Hr #3: It is Adam optimizer that leads to const gradient for the first step, and it is known that it might not converge.
    - https://novakov-alexey.github.io/adam-optimizer/
      https://openreview.net/forum?id=ryQu7f-RZ
  - Hr #4: Most of the hyperparameter change does not change the performance of the Adam optimizer. It is a correct implementation. Includes using more data, using AMSgrad, using higher tolerance (5->10), using large eps (1e-4)
  - Hr #5: All testings are actually almost the same err rate as the theoretical optimal, 0.04*1/sqrt(12)*6/9 = 0.0077; Using larger dataset size and higher tolerance is slightly better.
  - Hr #6: Found that the training speed is the fastest when last layer neuron #=60 and batch size = 64 or 32.
  - Hr #7: Found that when using more layers, the activation function to be ReLU is much better.

Details: Training testing further

- Training speed: (using 55k data pts)
  - Using 60 neurons in the output layer, batch size 64, 12 workers on 16 nodes, ~18s/epoch
    usually much faster than 20 neurons in output layer, batch size 64, 12 workers on 16 nodes, ~ 34s/epoch (rare) to 110s/epoch
  - In some cases, some nodes are slow (but not sure if I enter one and not sure how to avoid one)
  - Using 20 neurons in the output layer, batch size 24, 12 workers on 8 nodes are fine. ~ 35s/epoch to 45s/epoch
- Improving training results:
  - https://stats.stackexchange.com/questions/352036/what-should-i-do-when-my-neural-network-doesnt-learn
  - https://stats.stackexchange.com/questions/365778/what-should-i-do-when-my-neural-network-doesnt-generalize-well
- Some discussions
  - Optimal Learning rate: Adam is usually the fastest training technique, and it has its best learning rate range large (0.0001 to 0.01 for CNN); Learning rate can be fixed, decaying from large learning rate may be bad, requires testing based on the situation.
    https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2
    https://stackoverflow.com/questions/39517431/should-we-do-learning-rate-decay-for-adam-optimizer
  - Effect of batch size:
    https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e
  - Batch scaling? Residual neural network?

Details: input normalization

- For all training, no bias for 1st layer as I1 and I2 will always have some constant value.
- After normalization of $|S|$, will have I_3 small ($I_1 = 1$) while others large.
- After normalization of $|S|^2 + |R|^2$, will have all input features aligned.
- Using an artificial linear relationship $\nu_T = 0.5I_2$ will lead to not able to train NN for $|S|$ normalization.

$$u_{char} = \Delta_{LES}\sqrt{S_{ij}S_{ij}}$$
$$l_{char} = \sqrt{3}\Delta_{LES} = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

```
Max of abs I1_center, tensor([503.621477455306])
Max of abs I2_center, tensor([895.944804849226])
Max of abs I3_center, tensor([4241.723171796345])
Max of abs I4_center, tensor([2236.616544855686])
Max of abs I5_center, tensor([57971.771654909884])
Max of abs I6_center, tensor([1615153.743596519344])
Max of abs I1_center, tensor([1.000000000000])
Max of abs I2_center, tensor([505.248620568936])
Max of abs I3_center, tensor([0.408248214472])
Max of abs I4_center, tensor([110.517511205051])
Max of abs I5_center, tensor([154.582463170783])
Max of abs I6_center, tensor([311.855051250920])
```

Apart from changing normalization…

- Factors tested, helpful
  - Using ReLU for training linear relationship
  - Using 60 neurons in the last layer and worker = 32 or 64, much faster training.
- Factors tested, not important
  - Using different tolerance (5 or 10), using different initial learning rate / plateau learning rate
- Factors not tested, possibly helpful
  - Using batch normalization, it depends on batch statistics. https://stackoverflow.com/questions/63799763/questions-about-batch-normalization-in-pytorch

$$u_{char} = \Delta_{LES}\sqrt{2(S_{ij}S_{ij} + R_{ij}R_{ij})/9}$$
$$l_{char} = \sqrt{3}\Delta_{LES} = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

```
Max of abs I1_center, tensor([503.621477455306])
Max of abs I2_center, tensor([895.944804849226])
Max of abs I3_center, tensor([4241.723171796345])
Max of abs I4_center, tensor([2236.616544855686])
Max of abs I5_center, tensor([57971.771654909884])
Max of abs I6_center, tensor([1615153.743596519344])
After normalization
Max of abs I1_center, tensor([4.498773898571])
Max of abs I2_center, tensor([4.491111086891])
Max of abs I3_center, tensor([3.885706186808])
Max of abs I4_center, tensor([1.485730164569])
Max of abs I5_center, tensor([2.530847238079])
Max of abs I6_center, tensor([0.543985974335])
```

Details: Specific hyperparameter

- Effect of batch size: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e
  - Large batch size is supposed to easier reach global optimal, actual performance has lower training accuracy; Small batch size is supposed to have noises and overfitting, leading to local optimal.
  - Ramp up learning rate compensate for large batch size, since usual loss values are default averaged gradient in the batch, e.g., torch.nn.MSELoss, leading to difficulty in reaching weight far from initial. (not due to the hypothesis of gradient competition)
  - The large batch size actually has large norm of gradient steps (i.e., travels further $\sim \sqrt{\text{batch size}}$), and larger variance in (relative) gradient steps (due to the large tail in gradient distribution)
  - Using small batch size and ramp up learning rate or train much longer (the same number of stepping number), increases the training accuracy. It can be done at later time of training and still will improve training result.
  - Mostly, ADAM is much less sensitive to initial weight (e.g., weight travel distance grows linearly with the epoch #). Usually, ADAM is considered to have adaptive learning rate, but it is not uncommon to decay learning rate (some theory papers is based on $1/\sqrt{t}$ decaying learning rate). It is mainly trial and error, highly dependent on training dataset.
- Other hyperparameter tuning: optimizer, batch size (decrease it), learning rate (cyclic LR), layers, neuron numbers

Details: General techs to verify training and testing

- How to train well?
  https://stats.stackexchange.com/questions/352036/what-should-i-do-when-my-neural-network-doesnt-learn
  - Avoid bug, build unit testing.
  - Scale the data, and unscale the prediction; standardize to small interval as [-0.5, 0.5]
  - Batch normalization (to have converging accelerated training)
  - Choose correct number of neurons, hidden layers, and network wiring; pay attention to configurations, as initialization, activation functions (ReLU or leaky ReLU), residual connection, etc. Check the training metric makes sense.
  - Not easy to handle non-convex optimization, not easy to use high-order methods (>1), methods as correct learning rate range, gradient clipping, learning rate rescheduling, choose minibatch size, stochastic gradient descent with momentum / adaptive learning rates (ADAM, but it can converge to a much different local minimum)
  - No regularization for test purpose.
  - Test with 1 pt or 2 pts. Test the shuffled output (should reach the random change loss on the test set.). Check that initial loss is reasonable. Check masked data (e.g., LSTM). Check individual layers (visualization tool TensorBoard). Build a simple model (in case library error).
  - Check the training data distribution (input / output not heavily stewed toward 0, etc). Be consistent in preprocessing. Check a few existing data points. Try random shuffle. Try another dataset.
  - Try linear regression before training neural network. Provide easier dataset first, and then use a reduced LR for the whole dataset.
  - In full training, check training loss and validation loss, check test accuracy.
  - Hyperparameter tuning: optimizer, batch size (decrease it), learning rate (cyclic LR), layers, neuron numbers
- How to prevent overfitting / reduce generalization error?
  https://stats.stackexchange.com/questions/365778/what-should-i-do-when-my-neural-network-doesnt-generalize-well
  - Prerequisite: the model shall train well before dealing with overfitting.
  - Early techniques: parameter norm penalties (to limit model capability), **3)** early stopping (before validation error increases)
  - Neural network specific: **1)** batch normalization (image-related, not used with dropout simultaneously), **2)** dropout (fully connected layer), transfer learning (fine tuning after trained with larger but general dataset), reduce batch size, use SGD instead of adaptive method (as ADAM), remove outlier (in data, or in error in the model, namely influence); superconvergence (with cyclical learning rate and large learning rates)
  - Less used: tune hyperparameters, feature selection; adding random noises to weight, data augmentation (image-related), stratified/over/under sampling

Details: possible insights of regular training task

- Testing dataset:
  - Commonly used UCI dataset: https://archive.ics.uci.edu/datasets; MNIST dataset, etc
  - For a regression task, commonly seen that R^2=0.92 to be a good value, usually rRMSE ~ 25%.
- Known issues to take care of in neural network
  - Sensitive to the quality of data
  - Sensitive to feature scaling
  - L2 norm (MSE) is sensitive to the outliers; L1 norm (MAE) is less sensitive

Details: training setup that should work

- Tips:
  - Speed: batch size matching neuron number of output layer
    - Using 20 neurons in the output layer, batch size 24, 12 workers on 8 nodes are fine.
  - Avoid overlapping training as the data preprocessing takes time.
- Using decaying learning rate with Adam optimizer
- Test cases with artificial function
  - For nondimensionalized I1, I2 $\sim$ [0, 4.5], ($S_{char} = \sqrt{2(I_1 - I_2)/9}$), Loss L_2 error in absolute value.
    $\sin(x)$ is error rate of $\sim$ 0.005, with LR = 0.001; $\sin(10x)$ is error rate of $\sim$ 0.11, with LR = 0.001. Similar to the previous test set.
  - Loss L_2 error in absolute value (itar=(0,1)), with 2% noise, $\sin(x)$ has error rate $\sim$ 0.01; $\sin(10x)$ has error rate $\sim$ 0.041
    ($\sin(x)$ has error rate even lower than 2%, not checking if it is overfitting, $\sin(10x)$ even better than no noise)
  - Loss L_2 error in dissipation rate with noise (itar=(0,2)), $\sin(x)$ has error rate $\sim$ 0.008; $\sin(10x)$ has error rate $\sim$ 0.033
- Test cases with actual data, $S\tau$ as target
  - Use a very small true dataset $\sim$ 10 pts, 100pts, 1kpts, see if the model can overfit. (Only trained for 20min ~750 epochs)
  - LR $\sim$ 0.1 will learn, instead of LR $\sim$ 0.001. Because artificial functions -> loss function $\sim O(1)$, but the true data has $\tau \sim O(100)$. Or further modify l_{char} for a better scaled output.
  - 10pts data: training err $\sim$ 0.068, testing err $\sim$ 0.62. Overfitting $\surd$
  - 100 pts data: training err $\sim$ 0.15, testing err $\sim$ 0.67. requires LR plateau $\sim$ 0.01. Overfitting $\surd$
  - 1k pts data: training err $\sim$ 0.39, testing err $\sim$ 0.849. LR starts 0.1, plateau $\sim$ 0.01. Overfitting $\surd$
- Layers:
  - Current standard: Use 3 hidden layers (20,20,20); The previous built ones in DIM1 uses 4 hidden layers (20,20,20,60) (should make no difference)
  - Remove the bias for the first layer, as $I_1 + I_2 = const$ by definition.

Thank you for listening!

Xinyi Huang, xinyih@caltech.edu

Caltech

Division of Engineering
and Applied Science