Channel Flow Turbulence Modeling with Tensor Basis Neural Network

Rui Fang

January 25, 2018

1 Introduction

This project aims to utilize a machine learning method proposed by Ling et al., the tensor basis neural network (TBNN), to learn a model for the Reynolds stress anisotropy tensor of a turbulent channel flow from the DNS data.

2 Tensor basis neural network

The TBNN is a deep learning approach to turbulence modeling which uses a multiplicative layer with an invariant tensor basis to embed Galilean invariance into the predicted anisotropy tensor (see Figure 1).

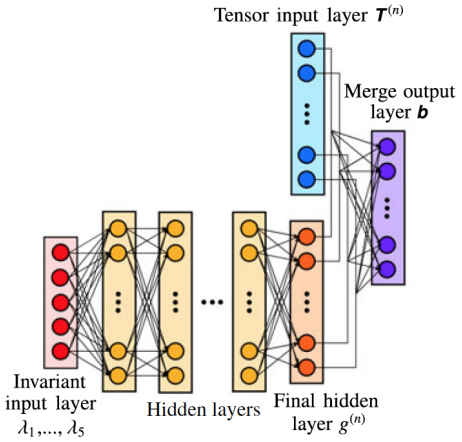


Figure 1: A diagram of the TBNN architecture [2].

The relevant integrity basis was previously derived by Pope [3]. Pope proved that in the most general incompressible case, the normalized Reynolds stress anisotropy tensor $\mathbf{b}$ that is a function of only $\mathbf{S}$ and $\mathbf{R}$ can be expressed as a linear combination of 10 isotropic basis tensors:

$$\mathbf{b} = \sum_{n=1}^{10} g^{(n)}\left(\lambda_1, ..., \lambda_5\right)\mathbf{T}^{(n)}, \tag{2.1}$$

where $\mathbf{T}^{(1)}, ..., \mathbf{T}^{(10)}$ are the ten basis tensors, $g^{(1)}(\lambda_1, ...\lambda_5), ..., g^{(10)}(\lambda_1, ...\lambda_5)$ are the ten scalar coefficients, and $\lambda_1, ...\lambda_5$ are the five scalar invariants. The ten basis tensors and the five scalar invariants are known functions of $\mathbf{S}$ and $\mathbf{R}$ (see Eq.(2.2) and Eq.(2.3)), the non-dimensionalized mean strain rate and the mean rotation rate tensors, which are respectively the symmetric and asymmetric part of the mean velocity gradient $\nabla\langle\mathbf{U}\rangle$ multiplied by k/ϵ.

$$
\begin{aligned}
&\mathbf{T}^{(1)} = \mathbf{S} && \mathbf{T}^{(6)} = \mathbf{R}^2\mathbf{S} + \mathbf{S}\mathbf{R}^2 - \tfrac{2}{3}\mathbf{I}\cdot Tr(\mathbf{S}\mathbf{R}^2)\\
&\mathbf{T}^{(2)} = \mathbf{S}\mathbf{R} - \mathbf{R}\mathbf{S} && \mathbf{T}^{(7)} = \mathbf{R}\mathbf{S}\mathbf{R}^2 - \mathbf{R}^2\mathbf{S}\mathbf{R}\\
&\mathbf{T}^{(3)} = \mathbf{S}^2 - \tfrac{1}{3}\mathbf{I}\cdot Tr(\mathbf{S}^2) && \mathbf{T}^{(8)} = \mathbf{S}\mathbf{R}\mathbf{S}^2 - \mathbf{S}^2\mathbf{R}\mathbf{S}\\
&\mathbf{T}^{(4)} = \mathbf{R}^2 - \tfrac{1}{3}\mathbf{I}\cdot Tr(\mathbf{R}^2) && \mathbf{T}^{(9)} = \mathbf{R}^2\mathbf{S}^2 + \mathbf{S}^2\mathbf{R}^2 - \tfrac{2}{3}\mathbf{I}\cdot Tr(\mathbf{S}^2\mathbf{R}^2)\\
&\mathbf{T}^{(5)} = \mathbf{R}\mathbf{S}^2 - \mathbf{S}^2\mathbf{R} && \mathbf{T}^{(10)} = \mathbf{R}\mathbf{S}^2\mathbf{R}^2 - \mathbf{R}^2\mathbf{S}^2\mathbf{R}
\end{aligned}
\tag{2.2}
$$

$$
\lambda_1 = Tr(\mathbf{S}^2), \quad \lambda_2 = Tr(\mathbf{R}^2), \quad \lambda_3 = Tr(\mathbf{S}^3), \quad \lambda_4 = Tr(\mathbf{R}^2\mathbf{S}), \quad \lambda_5 = Tr(\mathbf{R}^2\mathbf{S}^2) \tag{2.3}
$$

Mapping Eq.(2.1) into a neural network architecture, the TBNN has two input layers: the scalar invariant input layer and the tensor input layer. The input invariants λ's are transformed through multiple densely connected hidden layers to yield the coefficients $g^{(n)}$. Then the merge layer performs a linear combination of the input tensors $\mathbf{T}^{(n)}$ with the coefficients $g^{(n)}$ and output the anisotropy stress tensor $\mathbf{b}$. This architecture ensures that Eq.(2.1) is satisfied, hence guaranteeing the Galilean invarianve of the predictions.

In the following work, we make use of the TBNN package[1] in python to implement the TBNN approach.

3 Data sets

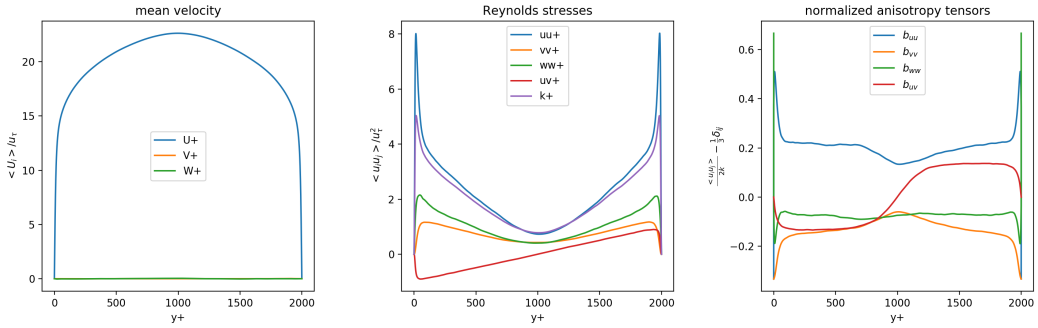
We will train, validate and test the TBNN on a channel flow data set. The channel flow data is from a high-fidelity DNS in the Johns Hopkins Turbulence Database (JHTDB) [1]. In this project, the DNS data is used to provide the truth labels for the anisotropy tensors, and the coarse-grained DNS data are used as inputs to the neural network. This is to imitate the RANS data, which are not available in our situation. If the DNS data are directly used as inputs, then ideally the predicted and true anisotropy tensors should be identical given that the TBNN are successfully trained.

The JHTDB channel flow DNS is performed in a domain of size $8\pi\times2\times3\pi$, using $2048\times512\times1536$ nodes, for approximately a single flow through time (corresponding to 4000 frames). In wall-parallel (x, z) directions, the grid spacing is uniform and the boundary conditions are periodic; in wall-normal (y) direction, the grid spacing is not uniform, and no-slip conditions are applied at the boundaries. Using a python interface[2], we can make queries to obtain velocity and velocity gradient data at the simulation nodes of a particular frame from the database. It is too costly to get the data for all the nodes and frames to perform averaging. Hence we choose to only get data at part of the nodes and frames. A qualitative convergence analysis is carried out to determine how much data may be considered enough.

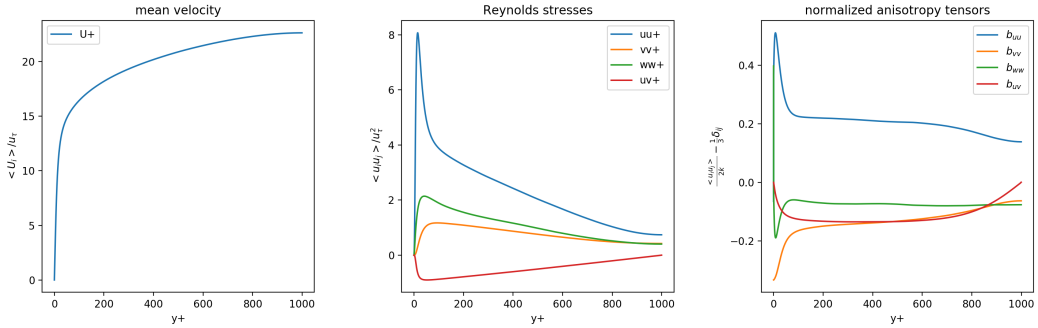
For example, we are currently picking every node in y direction, 1 node per 25 nodes in x direction, and 1 node per 51 nodes in z direction, resulting in a total number of $82 \times 512 \times 31$ nodes. And we query for 40 time frames from frame 0 to 3999 (frame indices: 99, 199, 299, ..., 3899, 3999). All these data are averaged over time and x-z plane to obtain the mean velocity $\langle\mathbf{U}\rangle$ and mean velocity gradient $\nabla\langle\mathbf{U}\rangle$. Based on these, we can calculate the Reynolds stresses $\langle u_i u_j\rangle$, the turbulent kinetic energy k, the normalized anisotropy stress tensors b_{ij}, the dissipation rate ϵ, the wall shear stress τ_w, the friction velocity u_τ and the viscous length scale δ_ν. Profiles of the mean velocity, Reynolds stresses and normalized anisotropy tensors in viscous units are shown in Figure 2a. In comparison, Figure 2b shows the profiles averaged over all time frames and all nodes in x-z plane. The true profiles are directly downloaded from the database website. By comparing the two figures, we observe that the profiles of our current data set are indeed converging to the true profiles. However, the Reynolds stresses profile and the normalized anisotropy tensors profile are not smooth enough. To improve it, we need to accumulate more data in future.

[1] `https://github.com/tbnn/tbnn`

[2] `https://github.com/idies/pyJHTDB`



(a) averaged over 40 frames and $82 \times 512 \times 31$ nodes



(b) averaged over 4000 frames and $2048 \times 512 \times 1536$ nodes

Figure 2: Channel flow profiles of mean velocity, Reynolds stresses and normalized anisotropy stress tensors in viscous units from two sets of data

The inputs to the neural network are the coarse-grained mean velocity gradient $\nabla\langle\mathbf{U}\rangle$, turbulent kinetic energy k and the dissipation rate ϵ. The coarse-graining procedure is basically reducing the data by averaging every n points to represent a single point of the n points. For example, there are originally 512 sample points along y direction. Choosing $n = 3$, then 170 points will remain after coarse-graining. The y positions and truth labels of these points correspond to the original data of the 2nd, 5th, 8th, ... points of the 512 points.

4 Experiment

The loss function associated with the TBNN is the mean squared error evaluated for all training samples and all 9 components of $\mathbf{b}$:

$$\text{Loss} = \frac{1}{9N}\sum_{m=1}^{N}\sum_{i}^{3}\sum_{j}^{3}\left(b_{ij,m,DNS} - b_{ij,m,TBNN}\right)^2. \tag{4.1}$$

In this particular model, the hidden units of the hidden layers are chosen to be leaky ReLU. So at each node, the node input $\mathbf{x}$ is transformed to $f(\mathbf{x}) = \max(0, \mathbf{w}^T\mathbf{x}) + \alpha \cdot \min(0, \mathbf{w}^T\mathbf{x})$, where the leakiness α is set to be 0.1. The parameters to be trained in this model are the node weights w's. The hyperparameters are the number of hidden layers, the number of nodes per hidden layer, the learning rate of the optimization algorithm, etc.

The network is trained using the mini-batch stochastic gradient descent (SGD). The learning rate of this algorithm is controlled by three parameters: the initial learning rate, the decay rate, and the minimum learning rate. At the beginning of each epoch while doing the SGD, the learning rate

is lowered by multiplying with the decay rate, until it hits the minimum learning rate. The batch size of this algorithm is also free to adjust.

The TBNN implementation uses a basic early stopping technique to determine when to stop training. This is described as follows [4]:

1. Split the training data into a training set and a validation set, e.g. in a 2-to-1 proportion.
2. Train only on the training set and evaluate the per-example error on the validation set once in a while, e.g. after every fifth epoch.
3. Stop training as soon as the error on the validation set is higher than it was the last time it was checked.
4. Use the weights the network had in that previous step as the result of the training run.

In actual implementation, extra parameters such as minimum and maximum number of epochs are also used.

After the model is trained, we need to validate the model/optimize hyperparameters using the validation data set. Currently, this search is done by hand. The entire training and validation procedure is summarized below.

Before training

1. For each sample: calculate S_{ij}, R_{ij} from $\nabla\langle\mathbf{U}\rangle$, k, ϵ; calculate scalar invariants, tensor basis from S_{ij}, R_{ij}; calculate true anisotropy tensor b_{ij} from Reynolds stress tensor.
2. Split data into training, validation, test sets.

Training

1. Define network structure: `num_layers`, `num_nodes`.
2. Specify loss function. Define `batch_size`, `init_learning_rate`, `learning_rate_decay`, `min_learning_rate`, `max_epochs`, `min_epochs`, `interval`, and `average_interval`.
3. Perform SGD. In each epoch:
   - Set learning rate.
   - Do a full pass over the training data, update the weights of the neural networks.
   - Check for early stopping criteria once per `interval` epochs.
     - If epoch $\leq$ `min_epochs`, continue.
     - If epoch $>$ `max_epochs`, stop.
     - If `min_epochs` $<$ epoch $\leq$ `max_epochs`, check if the averaged validation error over `average_interval` starts rising. If rises, then stop.

Validation

1. Tune hyperparameters and train the network until validation set yields best results.

5 Results and Discussion

We repeat the experiment for two groups of data: 1) 512 samples, without coarse-graining; 2) 170 samples, with coarse-graining. The lists of hyperparameters used are presented in Table 1 and Table 2 respectively for group 1 and group 2.

For each group, we plot the training and validation error curves to visualize how the RMSE evolves over epochs for training and validation sets (Figure 3a, 4a). The RMSE is defined as the squared root of the loss:

$$\text{RMSE} = \sqrt{\frac{1}{9N}\sum_{m=1}^{N}\sum_{i}^{3}\sum_{j}^{3}\left(b_{ij,m,DNS} - b_{ij,m,TBNN}\right)^2}. \tag{5.1}$$

Through a final RMSE report (Figure 3b, 4b), the performance of the TBNN model is compared with a baseline linear eddy viscosity model (LEVM). We can conclude from the test data RMSE that the TBNN predictions are more accurate than the LEVM predictions for both groups of data. In addition, the discrepancy between the accuracy of two models is larger in group 1 than that in group 2.

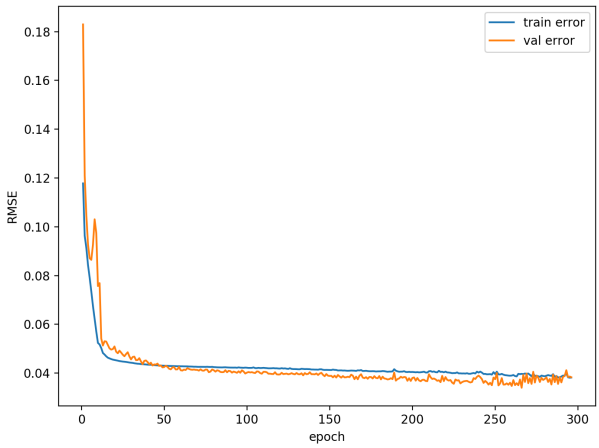
We can discover more details by looking at individual components of $\mathbf{b}$. In Figure 5 and Figure 6, the predictions and true values of four non-zero components of $\mathbf{b}$ are shown: $\mathbf{b}_{uu}$, $\mathbf{b}_{vv}$, $\mathbf{b}_{ww}$, $\mathbf{b}_{uv}$. Overall, the TBNN performs better than the LEVM. For uu, vv, and ww components, the LEVM fails to capture the shape of the anisotropy completely, while the TBNN predicts the near wall portions pretty well but not so well in the center of the channel. For uv component, the TBNN predictions look completely overlapping with the true values, showing very high accuracy, while the LEVM predicts the correct trend but the magnitudes are off as it gets closer to the walls. These observations hold for both groups. Comparing groups, the TBNN performs a lot better in group 1 than in group 2.

number of hidden layers	10
number of nodes per hidden layer	15
batch size	50
initial learning rate	0.005
learning rate decay	1
minimum learning rate	10^{-6}
minimum epoch	100
maximum epoch	2000
interval	5
average interval	10

Table 1: Group 1 hyperparameters

number of hidden layers	10
number of nodes per hidden layer	15
batch size	20
initial learning rate	0.001
learning rate decay	1
minimum learning rate	10^{-6}
minimum epoch	50
maximum epoch	2000
interval	2
average interval	4

Table 2: Group 2 hyperparameters

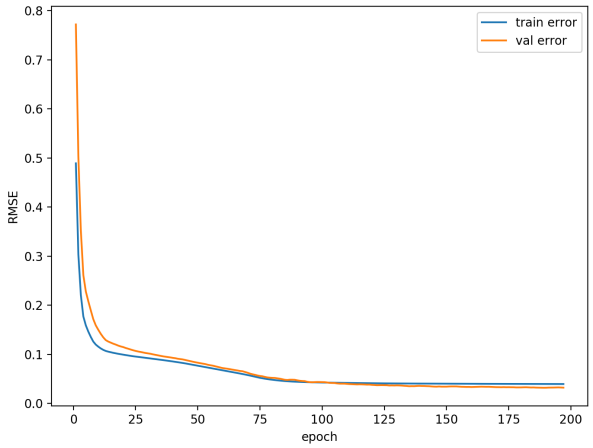


(a) Training and validation error curves

```
Re-setting normalization constants
Epoch 1 took 0.009s
  rmse training loss:       0.117769
  rmse validation loss:     0.182943
Epoch 51 took 0.004s
  rmse training loss:       0.042998
  rmse validation loss:     0.043004
Epoch 101 took 0.006s
  rmse training loss:       0.042120
  rmse validation loss:     0.040434
Epoch 151 took 0.008s
  rmse training loss:       0.041249
  rmse validation loss:     0.038881
Epoch 201 took 0.006s
  rmse training loss:       0.040396
  rmse validation loss:     0.038214
Epoch 251 took 0.007s
  rmse training loss:       0.040574
  rmse validation loss:     0.040021
Total number of epochs:  296
Final rmse validation error:  0.0382962125079
==============TBNN==============
RMSE of training data: 0.0381747873127
RMSE of validation data: 0.0382962125079
RMSE of test data: 0.0355616028099
==============LEVM==============
RMSE of training data: 0.133533374904
RMSE of validation data: 0.166606578235
RMSE of test data: 0.135584040722
```

(b) Final RMSE report

Figure 3: Group 1 results



(a) Training and validation error curves

```
Re-setting normalization constants
Epoch 1 took 0.005s
  rmse training loss:       0.488873
  rmse validation loss:     0.771807
Epoch 51 took 0.003s
  rmse training loss:       0.075621
  rmse validation loss:     0.081742
Epoch 101 took 0.003s
  rmse training loss:       0.042275
  rmse validation loss:     0.043088
Epoch 151 took 0.003s
  rmse training loss:       0.039986
  rmse validation loss:     0.034299
Total number of epochs:  197
Final rmse validation error:  0.0319514978317
==============TBNN==============
RMSE of training data: 0.0393012953925
RMSE of validation data: 0.0319514978317
RMSE of test data: 0.0562918622957
==============LEVM==============
RMSE of training data: 0.146425254871
RMSE of validation data: 0.14158536234
RMSE of test data: 0.11237045414
```

(b) Final RMSE report

Figure 4: Group 2 results

6 Conclusion

Based on the results, the capability of the TBNN to make relatively good predictions of the anisotropy tensors is confirmed. However, the fact that the TBNN tends to predict wrong values in the center of channel for uu, vv, and ww components raises up new questions.

The next steps focus on improving the neural network performance from two aspects. First, since we partition the data set into three sets, the training set may not be big enough for the neural network to learn the true model and the results may largely depend on a particular random choice of the training and validation sets. To resolve this, techniques such as cross validation will be helpful. Second, we hope to improve the process of optimizing hyperparameters. It is important to automate the search especially when the hyperparameter space is huge. We will try to implement Bayesian optimization to accomplish this.

References

[1] Yi Li, Eric Perlman, Minping Wan, Yunke Yang, Charles Meneveau, Randal Burns, Shiyi Chen, Alexander Szalay, and Gregory Eyink. A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, (9):N31, 2008.

[2] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[3] Stephen B Pope. A more general effective-viscosity hypothesis. *Journal of Fluid Mechanics*, 72(2):331–340, 1975.

[4] Lutz Prechelt. Early stopping-but when? *Neural Networks: Tricks of the trade*, pages 553–553, 1998.

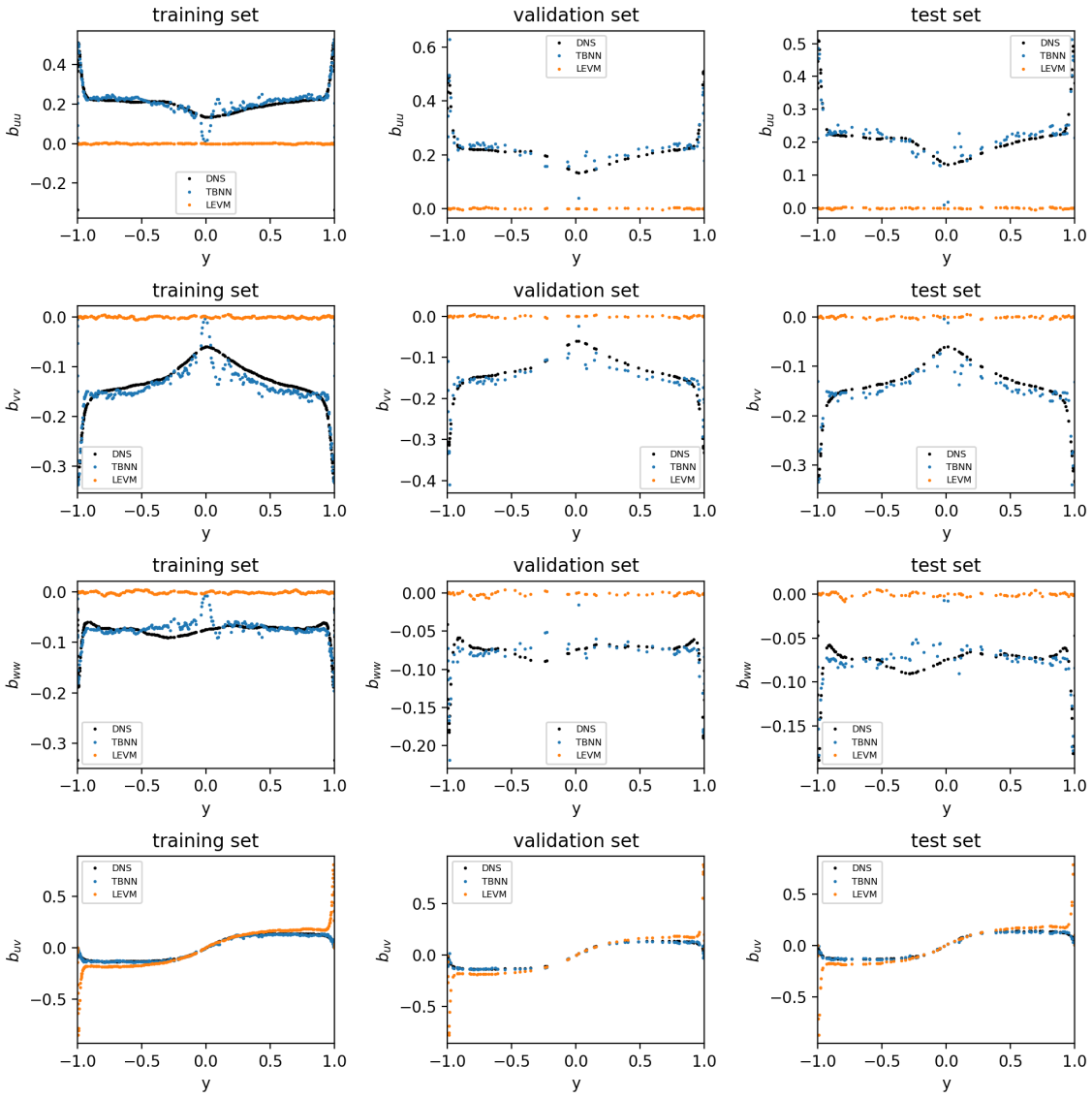


Figure 5: Group 1: True values, TBNN and LEVM predictions of anisotropy tensors vs y position (uu, vv, ww, and uv components of $\mathbf{b}$)

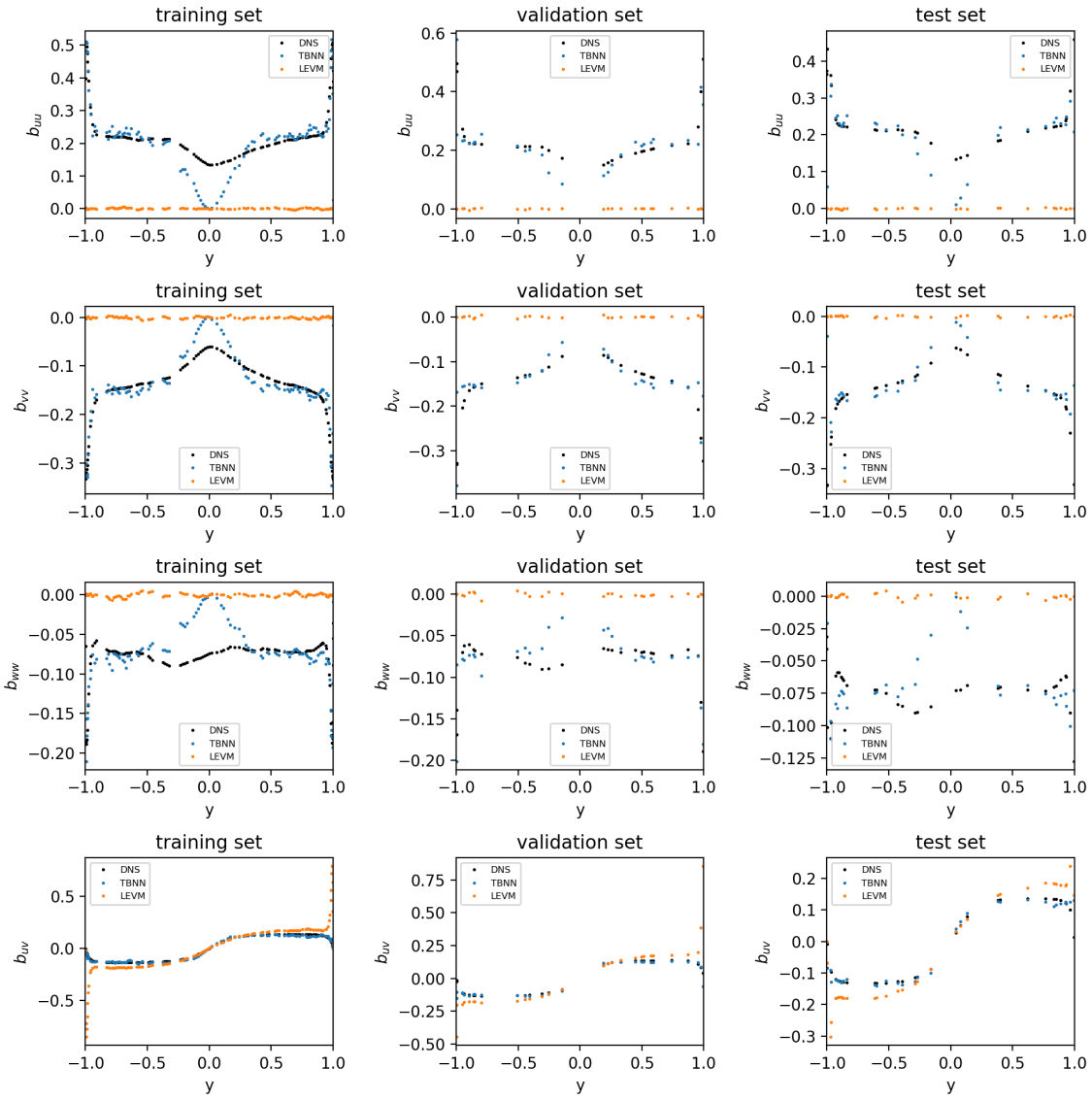


Figure 6: Group 2: True values, TBNN and LEVM predictions of anisotropy tensors vs y position (uu, vv, ww, and uv components of $\mathbf{b}$)