Universidad Nacional Autónoma de México
Instituto de Energías Renovables
M.Sc Engineering
Turbulence

TBNN

Autores:
Brian Raymundo Cortés Nava
Jennifer Reyna Guillén

December 7, 2021

1 Introduction

The Tensor Basis Neural Network (TBNN) model developed by (Ling et al., 2016) has shown great promise to improve the momentum equations in Reynolds-averaged Navier Stokes (RANS) solvers. It uses physical insight together with machine learning paradigms to embed rotational invariance into a deep neural network, and then predicts a turbulence anisotropy tensor that obeys this property (Milani et al., 2019).

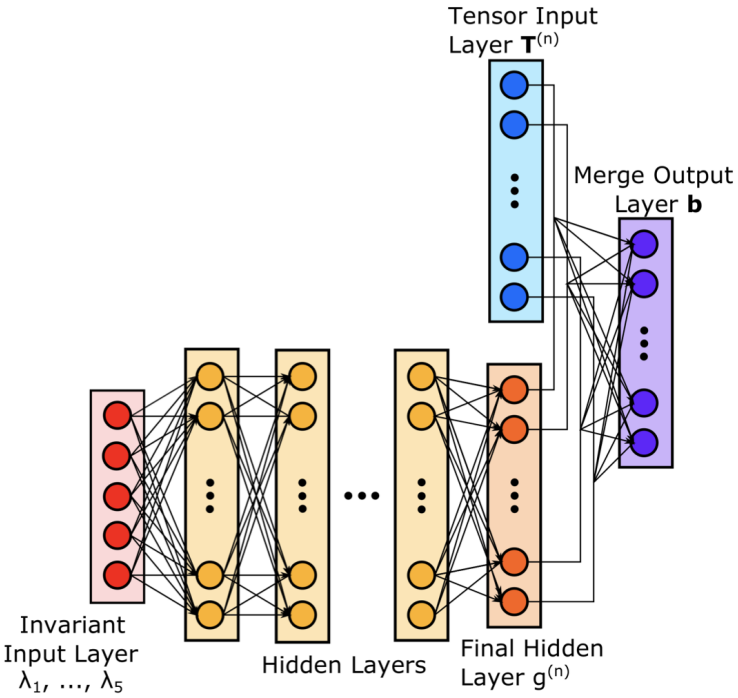

Figure 1: Schematic of Tensor Basis Neural Network

A special network architecture, which will be referred to as the Tensor Basis Neural Network (TBNN), is proposed. This network architecture, shown in Fig. 1, embeds rotational invariance by enforcing that the predicted anisotropy tensor lies on a basis of isotropic tensors. Rotational invariance signifies that the physics of the fluid flow do not depend on the orientation of the coordinate frame of the observer. This is a fundamental physical principle, and it is important that any turbulence closure obeys it. Otherwise, the machine learning model evaluated on identical flows with the axes defined in different directions could yield different predictions (Ling et al., 2016).

Lets say that we want to build a model where the inputs are tensors $\vec{A}$ and $\vec{B}$ and vector $\vec{v}$ and the output is a tensor Y. Furthermore, we know that Y obeys a known invariance constraint. This can be expressed as:

$$\begin{aligned} \text{if} \quad & Y = Y(\vec{A}, \vec{B}, \vec{v}) \\ \text{then} \quad & Y(Q\vec{A}Q^T, Q\vec{B}Q^T, Q\vec{v}) = QY(\vec{A}, \vec{B}, \vec{v})Q^T \end{aligned}$$

In this notation, Q is any matrix from the specified invariance group. Q^T is the transpose of Q. The basic idea is that if the inputs are transformed by some matrix from the invariance group, then the output should be transformed correspondingly (Templenton, 2018).

For an arbitrary set of vectors and tensors and a specified invariance condition, it is possible to construct a finite tensor basis $(S_1, S_2, ..., S_n)$ and a finite scalar basis $(c_1, c_2, ..., c_m)$ (Templenton, 2018).

We can then say that any function of those vectors and tensors should sit on the tensor basis:

$$Y = \sum [f_i(c_1, ..., c_m) S_i]$$

In other words, our output Y can be expressed as a linear combination of the tensor basis, where the coefficients are given by unknown functions f_i of the scalar invariants. The tensor basis can either be calculated afresh or can often be looked up in the literature, where many such bases have been tabulated (Templenton, 2018).

The Tensor Basis Neural Network uses this concept to ensure that the model obeys the specified invariance constraints.

It has two input layers. The first input layer accepts the scalar invariant basis. These inputs are then transformed through multiple densely connected hidden layers. The second input layer accepts the tensor basis. The final hidden layer is then pair-wise multiplied by this tensor basis input layer and summed to provide the output. Thus, the TBNN directly maps the tensor basis equation into the neural network architecture (Templenton, 2018)

2 Turbulence example

```
def main():
    # Define parameters:
    num_layers = 2  # Number of hidden layers in the TBNN
    num_nodes = 20  # Number of nodes per hidden layer
    max_epochs = 2000  # Max number of epochs during training
    min_epochs = 1000  # Min number of training epochs required
    interval = 100  # Frequency at which convergence is checked
    average_interval = 4  # Number of intervals averaged over for early
        stopping criteria
    split_fraction = 0.8  # Fraction of data to use for training
    enforce_realizability = True  # Whether or not we want to enforce
        realizability constraint on Reynolds stresses
    num_realizability_its = 5  # Number of iterations to enforce realizability
    seed = 12345 # use for reproducibility, set equal to None for no seeding
```

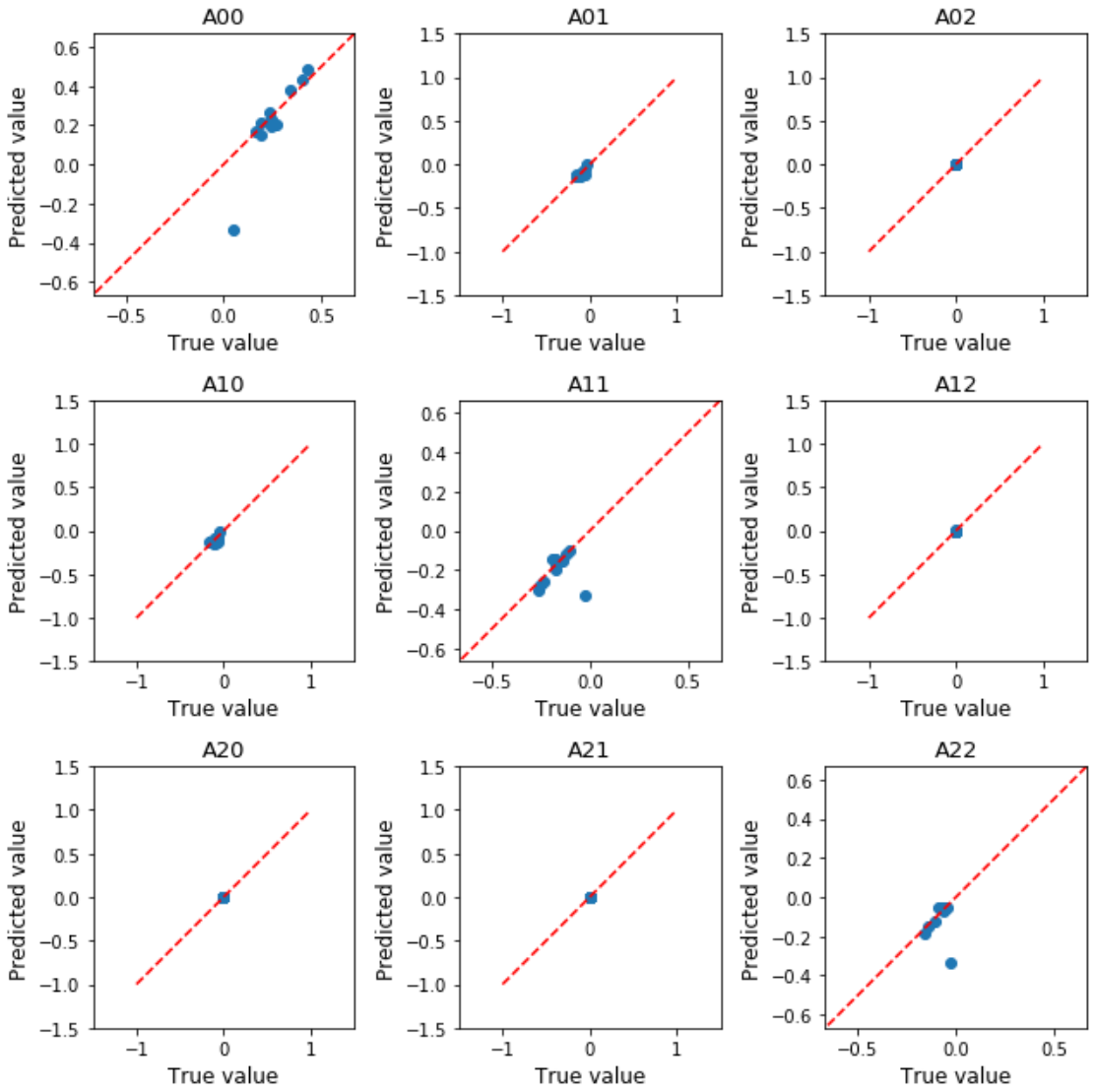

Figure 2: Results obtained from the original code

3 Proposed changes and Results

3.1 Change 1

The first configuration we tried was change the number of layers to 1, also change the neurons per layer to 15 and increase the maximun epochs to 4000, as well as the minimun epochs to 2000. The figure 3 shows us the results of the code described above.

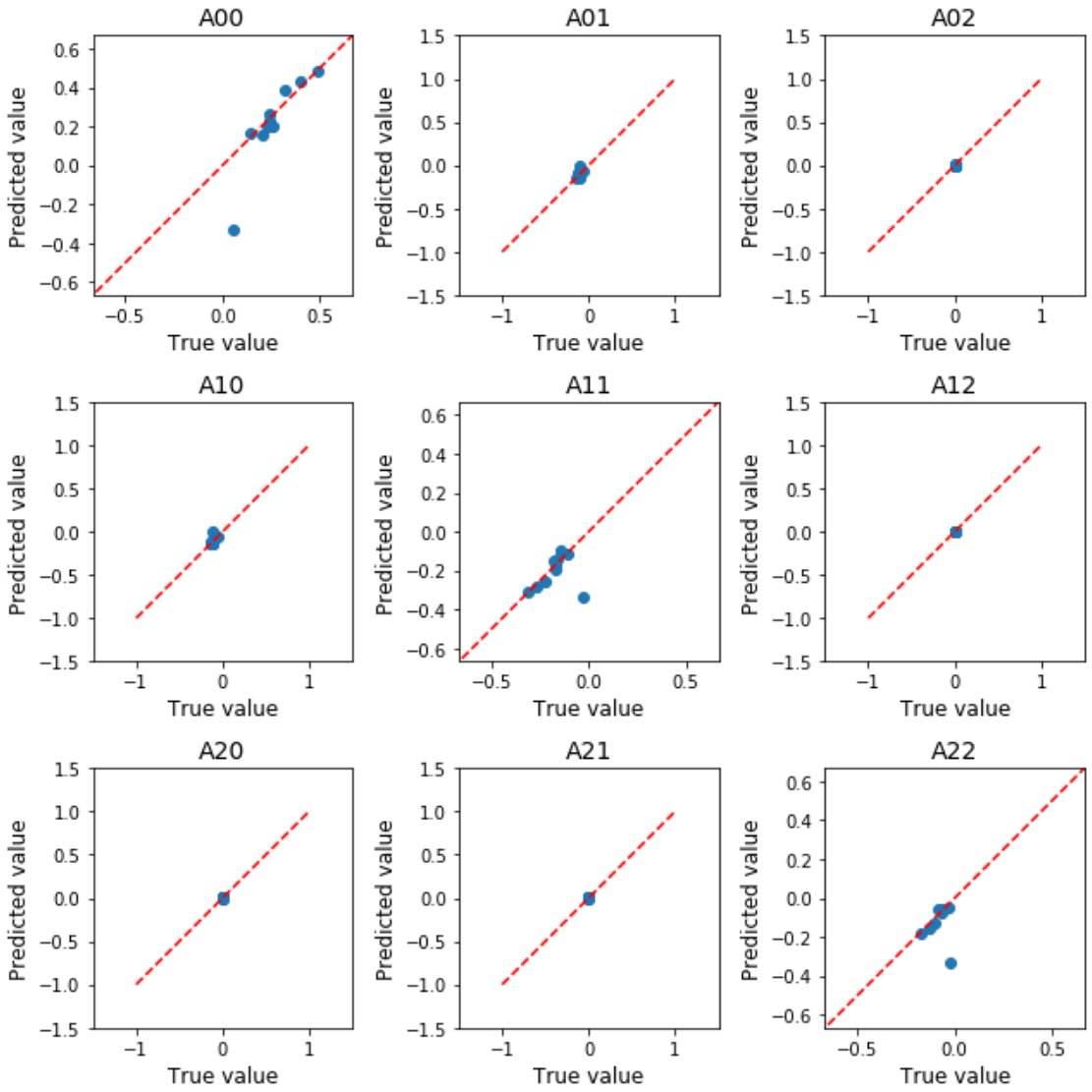

Figure 3: Results obtained from the first configuration made

Making a comparison with the original results, it can be seen that there are no significant changes, some points are simply improved very little and others move further away from the line

3.2 Change 2

The second configuration was change the number of hidden layers to 10, increase the maximun and minimun epochs to 5000 and 2000 respectively, and also increase the split fraction to 0.9. The figure 4 shows us the results of the code described above.

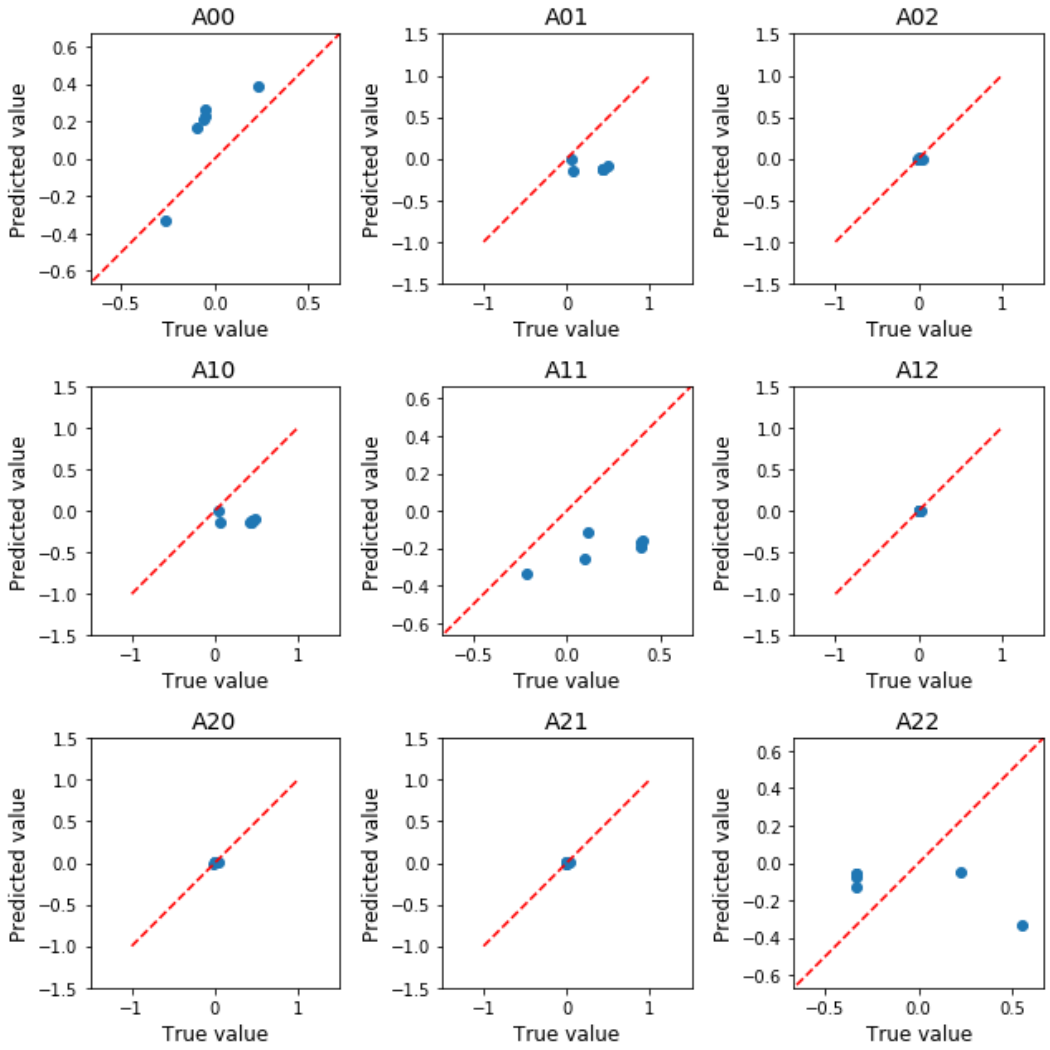

Figure 4: Results obtained from the second configuration made

Making a comparison between the results obtained in 2 and 4, we can notice how the values are farther away from the red line in this second configuration for all the plots except for those which have a zero value.

3.3 Change 3

The third configuration was change the number of layers to 20, the nodes per layer to 10 and decrease the split fraction to 0.7; The figure 5 show us the results of the code described above.

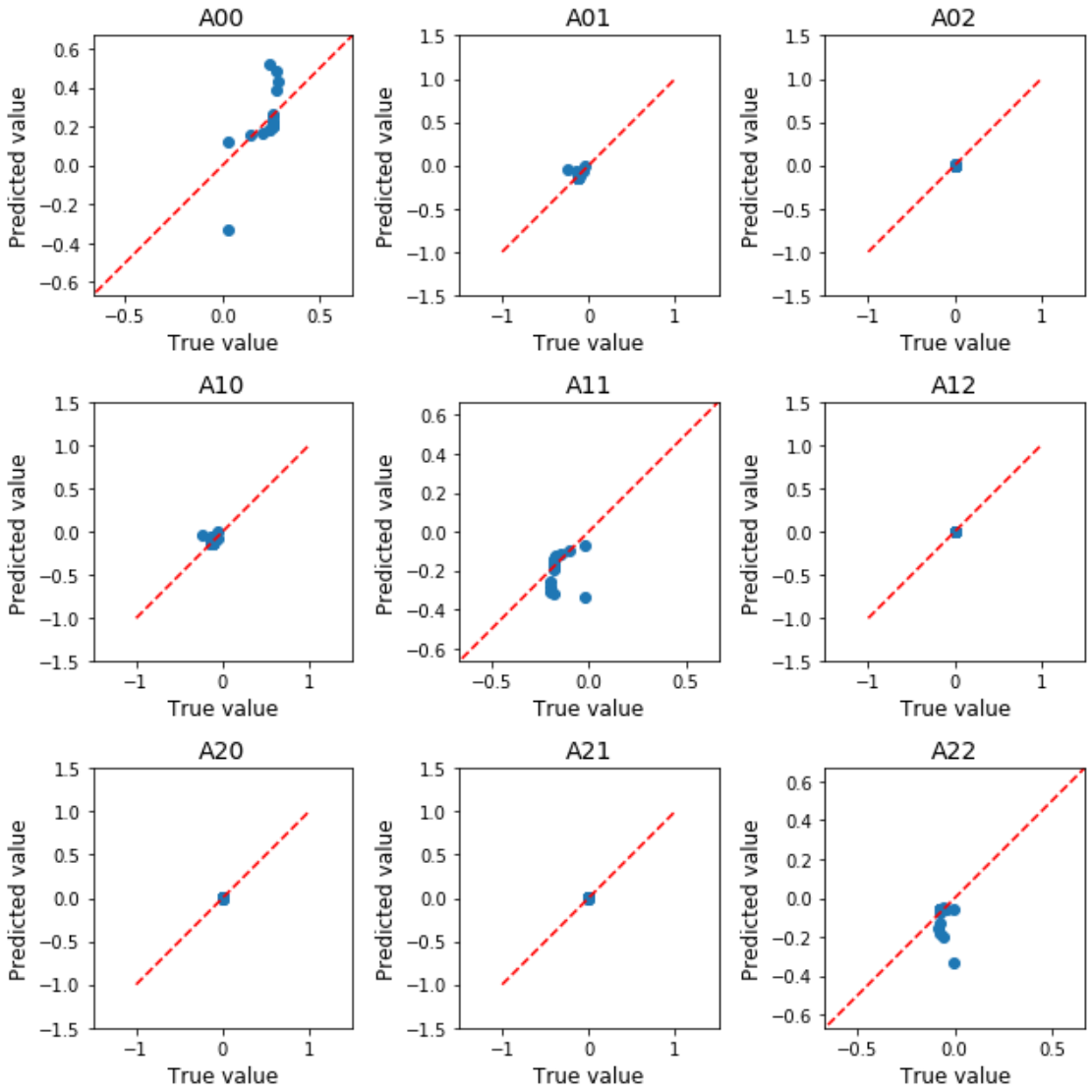

Figure 5: Results obtained from the third configuration made

In this case, making the comparison between the figure 2 and the figure 5, we can observe that we have deviations from the red line but they are not as large as the ones that we can see in the figure 4, also in this case, we can notice that in the ones that the true value is zero, the prediction value is also zero.

4 Conclusions

In summary, we made 3 configurations in the first one we decreased the number of layers and in the other ones we increased to 10 and 20 layers and also we made changes in the number of neurons per layer, in some cases we decrease the neurons and in others we increase the nodes.

The results obtained from the first configuration 3 compared to 4 and 5, showed that it is not necessary use more than 10 hidden layers with more than 15 neurons, because if we look for the graph, the original code and the one where we only decrease the layers to one, the results are too similar, so this show us, that adding more layers maybe is not necessary, because with that we are adding equations and operations that are not necessary. With this, we can conclude that to get an accurate result, we can use one or two layers with a range of 15 to 20 neurons per layer; if we want to use more than 10 layers, the results show us that we need to use less than 10 neurons because the difference between the real value and the prediction starts to grow.

Bibliography

Ling, J., Kurzawski, A., Templeton, J., 2016. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. Journal of Fluid Mechanics 807, 155–166. doi:10.1017/jfm.2016.615.

Milani, P.M., Ling, J., Eaton, J.K., 2019. Tensor Basis Neural Networks for Turbulent Scalar Flux Modeling, in: APS Division of Fluid Dynamics Meeting Abstracts, p. G17.002.

Templenton, J., 2018. tbnn. URL: https://github.com/tbnn/tbnn.