Evaluating Robustness for the Tabular Data

Mounir Raki mounir.raki@epfl.ch
Supervisor: Klim Kireev klim.kireev@epfl.ch

June 15, 2022

Abstract

DNNs have been the method of choice in almost every domain for a very long time, due to their great adaptability and coverage in the ML literature. However, for Tabular Data, DNNs were found quite inefficient compared to Decision-Tree based algorithms. Recently, new models like TabNet appeared, which combine ideas from Decision-Tree and Deep Neural Network models into what are called Transformer blocks to aim at improving learning performance on Tabular Data. However, like all DNNs, TabNet is vulnerable to adversarial examples.

We demonstrate in this paper that the most contributing factor to the adversarial robustness of TabNet is the number of non-shared layers in the Feature Transformer blocks, where the robustness decreases as the number of layers gets larger. Finally, we show that the size of the model and the number of shared layers in the Feature Transformers have a marginal impact on the robustness of TabNet.

1 Introduction

Deep Neural Networks-based models have become the go-to in practically every domain, because of their great versatility and the fact that they are very well-documented. However, for Tabular Data, DNNs were found quite inefficient compared to Decision-Tree based algorithms like XGBoost and Random Forests models [CG16, XKL+21].

In 2020, Arık and Pfister [AP20] proposed a new way of learning from Tabular data based on a Neural Network, which aims at improving the performance achieved by Decision-tree algorithms, called TabNet. TabNet is based on building blocks called Transformers, constituted of smaller building blocks called layers, which became very popular in Machine Learning recently due to their very good performance on language tasks [VSP+17, DCLT19, BMR+20] and state-of-the-art results in computer vision [XBK+15, RPV+19, LWU+20], but more importantly due to their main strength: they implement an attention mechanism to be able to select certain features based on some criteria (mostly done by setting a cost value to each feature, with the most important features having a greater cost than the others), and focus on them in the training step to grab as much relevant information as possible to limit bias in the model decisions. However, since TabNet is based on Deep Neural Networks, it inherits a significant drawback of them: their vulnerability to adversarial examples. Indeed, due to the gigantic popularity of Neural Networks, a lot of works in the ML literature have been focused on their adversarial robustness on different kinds of adversarial samples (mainly image and text samples), proposing some methods like RobustBench [CAS+21] to evaluate it for different types of data alterations like rotations or noise application for image samples.

For Transformer robustness, Bhojanapalli et al.'s results [BCG+21] tell us that removing any single layer in the Transformers doesn't alter the robustness, but since the different blocks of layers are highly correlated between each other, removing an excessive amount of them results in occurring misclassifications and thus reduced robustness on adversarial data. In our paper, we seek to gain more information on the factors having an impact on TabNet's robustness on adversarial attacks, and our contributions are as follows:

- We perform a set of exhaustive experiments on the TabNet parameters to find the most relevant ones for the model's robustness, and decide to focus on the size of the model, the number of shared layers, and the number of individual blocks of layers in the Feature Transformers as the most relevant parameters.

- We compare the contributions of each relevant TabNet parameter regarding robustness accuracy, and find that increasing the number of individual blocks of layers in the Feature Transformers leads to a significant decrease in model robustness, while modifying the other two parameters yields a moderate impact on robustness, following the same trend as the most influential parameter.

2 Related works

2.1 Description of TabNet

In a nutshell, TabNet is a model performing Decision Tree (DT)-like classification using conventional Deep Neural Networks (DNN) blocks. To be able to do that, the model selects the relevant features of the input by applying a mask on it, before they pass into a Fully Connected (FC) layer which performs a linear transformation on them before going through a ReLU layer.

TabNet is composed of an encoder and a decoder, like in a conventional DNN. The encoder is structured in what is called "decision steps", where each step contains an Attentive Transformer, a feature selection Mask, a Feature Transformer, a Split block, and a ReLU block. The number of these decision steps is a hyper-parameter of the model, hence it has to be set by the user.

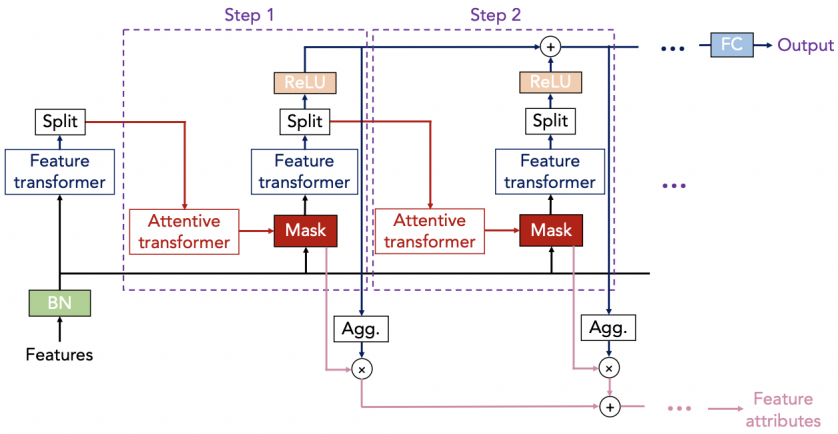


Figure 1: The architecture of TabNet's encoder. [AP20]

The first main building block of TabNet is the Attentive Transformer. Its purpose is to keep track of how many times each feature has been used in the prior decision steps, to make the feature selection more effective. An Attentive Transformer is composed of a Fully Connected Layer, a Batch Normalization Layer, and takes the prior scales computed in the previous decision steps as input to update them with the ones retrieved at the output of the Batch Normalization layer. Updating the coefficients is a matter of multiplying the newly computed coefficients by the prior ones and applying a Sparsemax function on them for normalization, which enables the model to do what is called "sparse selection" of the most salient features, i.e. selection of the features for which the coefficients are the highest.

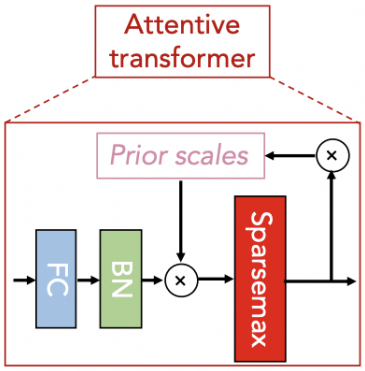


Figure 2: The architecture of an Attentive Transformer. [AP20]

The output of the Attentive Transformer goes directly into what we call a feature-selection Mask. This mask is a matrix of zeros and ones, whose purpose is to select some features and discard the other ones to focus the learning part on the more relevant features. The disposition of the matrix values is specified by the output of the Attentive Transformer.

After the more relevant features have been isolated, they are fed into the second most important block of TabNet, the Feature Transformer. Its purpose is to simulate the behavior of a Decision-Tree model by transforming the most salient input features and learning from them. Each Feature Transformer block is constituted of multiple little blocks linked together by a direct and a bypass connection. In those blocks, there is a Gated-Linear Unit (GLU) layer performing region selection and can either be shared with every decision step or be specific to the actual one. The number of shared GLU layers and the number of individual little blocks in each Feature Transformer are also hyper-parameters of the model. Each of these little blocks also contains a Fully Connected layer to perform a linear transformation of the inputs and a Batch Normalization (BN) layer.

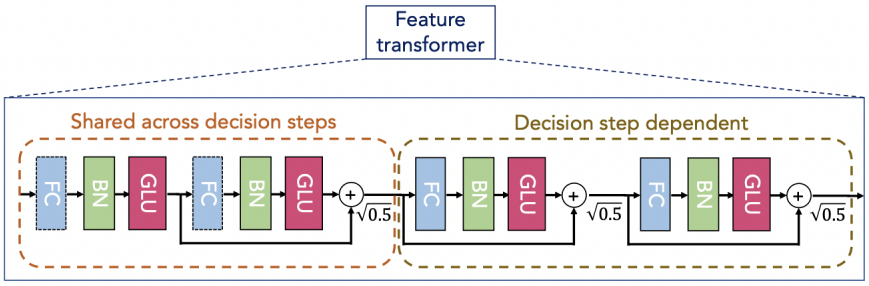


Figure 3: The architecture of a Feature Transformer. [AP20]

After passing through the ReLU block, the output of each step is aggregated before passing through a Fully-Connected layer, constituting the final output of the encoder.

In addition, the model keeps track of each feature attribute by aggregating the result of each decision step and then applying its mask to it (i.e. for decision step i, we take the results of the decision step, aggregate them, and apply the feature-selection mask of decision step i to the result).

The decoder is also structured in "steps", where a decoder contains the same number of steps as the encoder. Each step takes the encoded representation of the data as input, and each is constituted of a Feature Transformer and a Fully Connected layer. Finally, the output of each decoding step is aggregated to reconstruct the original features.

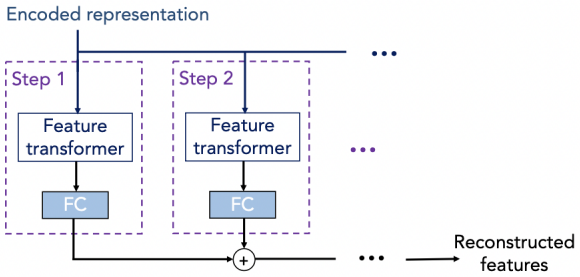


Figure 4: The architecture of TabNet's decoder. [AP20]

2.2 Related works

From Bhojanapalli et al.'s paper [BCG+21], we learn that the Transformers in Transformer-based architectures like TabNet can be impacted by input or model perturbations. What is meant by input perturbations are modifications performed on the data like noise on images, which makes them still look normal to the human eye but different enough for the model that it wrongly classifies the data, whereas model perturbations denote modifications done to the actual architecture of the Transformer parts. An example of that is the famous panda/gibbon example which is pictured below.

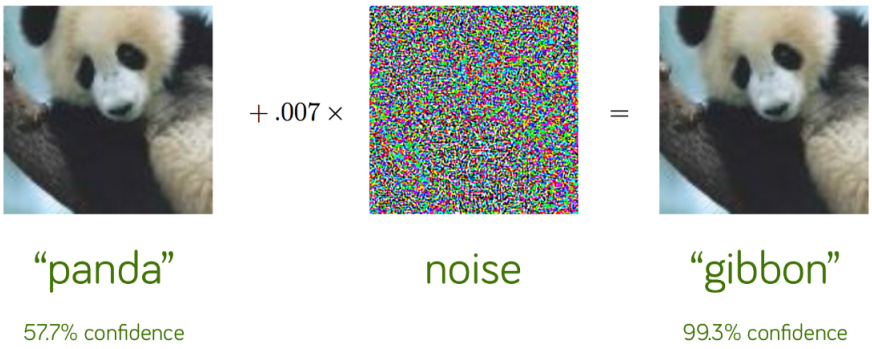


Figure 5: The Panda/Gibbon example. [ea15]

Regarding input perturbations, the paper tells us that the robustness of a model depends on the size of the used dataset and the size of the Transformers present in the model's architecture. Indeed, the robustness of the models gets better when the size of the dataset gets larger or when we add more layers to the Transformers to increase their size.

For model perturbations, the paper tells us that the different blocks of layers have some kind of redundancy, meaning that if we keep the first block of layers intact but remove some of the other ones, even though the Transformer will perform a bit worse, it will not be impacted by a lot depending on its size (i.e. a larger Transformer will perform better than a smaller one in case we remove layers, which is expected). A very important point the paper learns us is that altering the self-attention layers (in our case, the components of the Attentive Transformer) hurts the model a lot more than modifying the other layers, which shows the importance of the Attentive Transformer in our case.

3 Description of the approach

3.1 Software experiments

3.1.1 Description of the setup for experiments

For the project, we were provided with some code performing the training and evaluation of TabNet models by giving parameters as command-line arguments, and a given configuration of a TabNet model with hard-coded hyper-parameters, to familiarize ourselves with the train & eval procedures and set up the work environments. The whole project was centered on the IEEECIS Fraud Detection dataset, which is originally a dataset used for benchmarking machine learning models in ML competitions. At first, we encountered some issues when trying to run the train & eval on local laptops due to a lack of hardware acceleration on macOS with PyTorch, and we had to switch to a server provided by the lab for the rest of the project, to leverage hardware acceleration on GPUs which were installed in the server. The provided TabNet model to be trained as a test was one with the following hyper-parameter values, with an explanation of each hyper-parameter:

Parameter name	Default value	Description of parameter
n_a	16	denotes the dimension of the features exiting the ReLU blocks
n_d	16	denotes the dimension of the features exiting the ReLU blocks
n_steps	4	denotes the number of decision steps
n_shared	2	denotes the number of shared GLU layers across decision steps, in the Feature Transformer
n_ind	2	denotes the number of individual blocks of layers (i.e. not shared across decision steps) in the Feature Transformer
relax	1.2	denotes the upper bound on the number of times a given feature can be used in the different decision steps
vbs	512	the size of batches used to perform the Batch Normalization in the Batch-Normalization layers

After that, the goal was to create many models by varying the hyper-parameters and performing the training procedure on each of them.

3.1.2 TabNet hyperparameter selection

To ensure that the system works properly, we started the grid search on clean models, meaning that no adversarial samples would be fed into the model in the training stage. From Arık and Pfister's paper [AP20], we focused on varying the *n_steps*, *n_shared*, and *n_ind* hyper-parameters while keeping the other ones at their default values. The ranges for these parameters were taken from the TabNet paper [AP20] and an article describing how to implement TabNet in PyTorch [Tha20] based on the paper, but reduced a bit to have fewer models to train and evaluate since these procedures are very time-consuming:

Parameter name	Ideal range	Chosen range for experiments
n_steps	2 to 10	2 to 8
n_shared	2 to 10	1 to 3
n_ind	2 to 10	1 to 3

In the end, we generated a total of **63** models and evaluated them, then retrieve the data from the evaluation and processed it. For that, we created a bash script iterating on all three parameters and training a model for each combination of them, and another script handling the evaluation part. For the eval procedure, we were provided with many different criteria called "utility_types", and we focused on two of them due to the irrelevance of the other criteria, which are the following:

- **Average Attack Cost** → gives us the total cost of features that have to be modified in the training dataset to misclassify a sample, averaged on the whole size of the training dataset.
- **Robustness Accuracy** → accuracy of the model when fed adversarial samples, i.e. the percentage of correctly classified adversarial samples.

To simplify the interpretation of the data and speed up the evaluation step on the Average Attack Cost metric, we decided to set the three main features of the IEEECIS Fraud Detection dataset to a cost equal to 1 while setting the cost of the other features to 0. This way, the value obtained can also be interpreted as the average number of features to alter to have a misclassification, rather than the average cost only.

3.1.3 Train step-specific hyperparameter selection

We decided to set the number of epochs for the train part to an arbitrarily large value of **1000** since we implemented an early-stopping mechanism preventing the model from over-fitting on the train data, which would be enabled starting from epoch **15** with a delay before stopping the training part set to **10** epochs. To compare, we also decided to run the train part on all models with a delay threshold (i.e. the number of epochs before which the early-stopping mechanism is disabled) equal to 100 to let the models train for longer, such that we can see if we can reach a better overall accuracy for all models or not. The results of the evaluation procedure are the following:

Avg. Test Accuracy (Delay Thres.=15)	**Avg. Test Accuracy (Delay Thres.=100)**
0.7088655063640176	0.720583808846792

On average we expected to reach a test accuracy of around 75 to 77%, but as we can see we reached an average of 71% accuracy for a delay threshold equal to 15, and an average accuracy of 72% for a delay threshold equal to 100, quite a bit lower than expected. This also means that for a clean model, increasing the delay threshold doesn't improve the accuracy by a lot, but it can be very helpful for models with adversarial training, hence for the rest of the experiments we used a delay threshold value of **100** epochs.

3.2 Experiments on TabNet's Adversarial Robustness

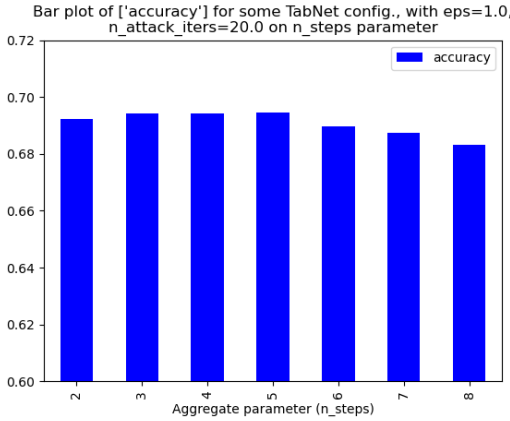
Knowing the little issues arising after performing the first iteration of the grid search, we then fixed them by adding the delay threshold and delay parameters as command-line arguments of the training procedure and we decided to set the delay threshold to 100 while keeping the delay to 10 for the rest of the project.

Then, we set up the training part to perform the first iteration of adversarial training. To do this, we were provided with two command-line arguments, **eps** and **attack_iters**. The eps argument (denoted by the ϵ symbol throughout this report) sets a bound for the distance (difference) between a sample and its adversarial copy, and the attack_iters argument denotes the number of adversarial samples to feed the training step at each epoch (or iteration). After some discussions, we chose to focus ourselves on $\epsilon = 1.0$ and an attack_iters value of 20 due to time constraints, and because TabNet usually doesn't use all the features of the data to classify a sample but focuses on the most important ones, a value of $\epsilon = 1.0$ is great to see if the models are semi-robust.

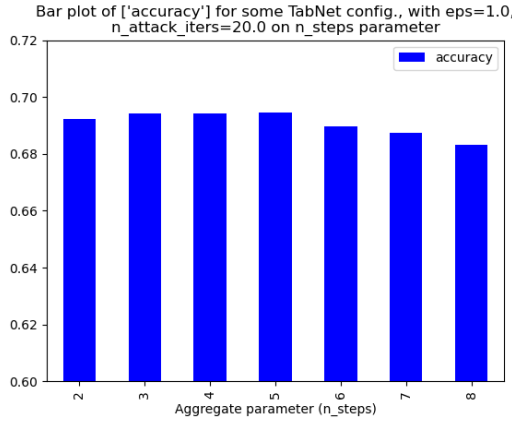
We then performed two runs of the train & eval steps, once using a very neat command-line parameter of the eval step called the cost_bound, and once without it. This parameter, taking a float value, sets an upper bound on the distance (i.e. the difference) between samples and their adversarial copies which will be fed into the model in the evaluation step, hence discarding all the other evaluation samples for which the distance is outside the bound and thus speeding up the evaluation process. We decided to set the cost_bound value to 1.1, to allow adversarial samples at a slightly higher distance than the ϵ value to be fed into the model to see the difference without the cost bound.

3.2.1 Focusing on the *n_steps* parameter

We plotted the aggregated values of the eval step on the *n_steps* parameter for each eval criterion, and the results are the following:

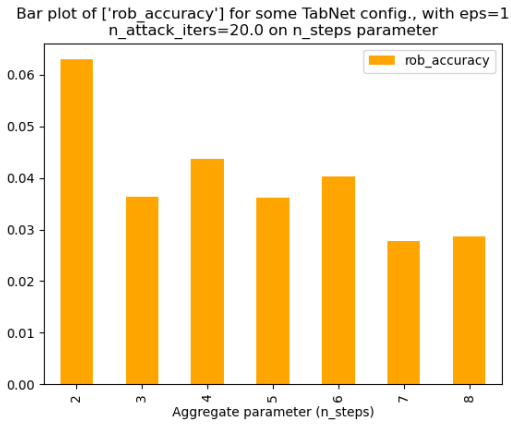


(a) Without cost_bound parameter

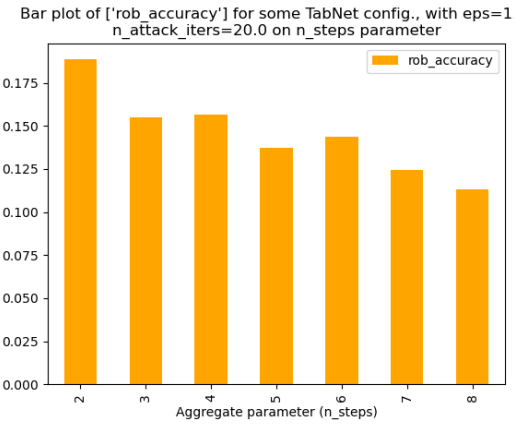


(b) With cost_bound=1.1

Figure 6: Bar plots of Test Accuracy metric, results averaged for each model on the *n_steps* parameter

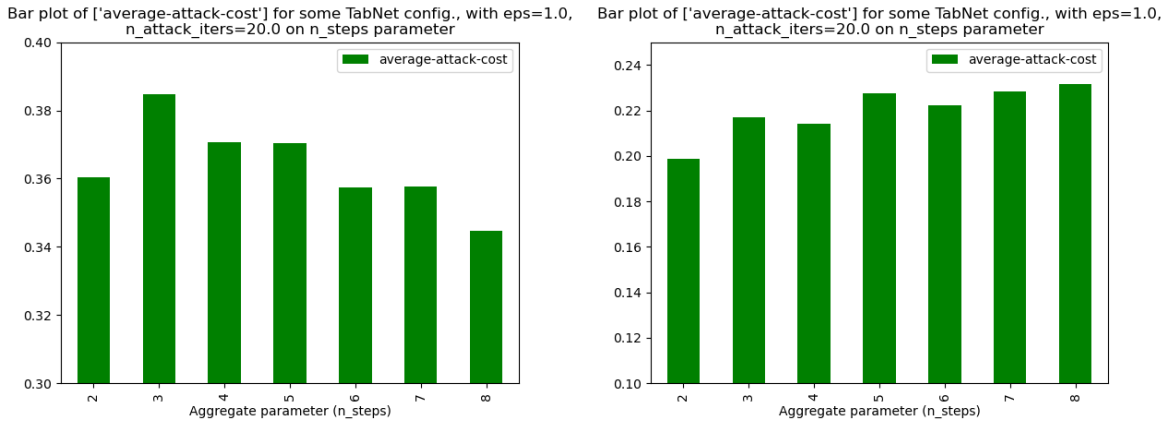


(a) Without cost_bound parameter



(b) With cost_bound=1.1

Figure 7: Bar plots of Robustness Accuracy metric, results averaged for each model on the *n_steps* parameter



(a) Without cost_bound parameter

(b) With cost_bound=1.1

Figure 8: Bar plots of Average Attack Cost metric, results averaged for each model on the *n_steps* parameter

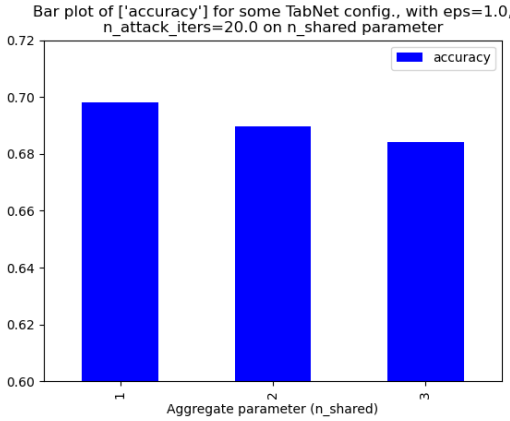
As we can see, the robustness accuracy is lower for every model size on the non-cost-bounded graph compared to the cost-bounded graph, which is normal since the cost-bound parameter prevents feeding inputs for which the adversarial copies are too different from the original sample (i.e. the distance is bigger than the threshold set by the cost bound parameter). On both plots of Figure 7, we see a decrease in the robustness accuracy for bigger models, with the smallest model being the best on average. This also follows the decrease in test accuracy we can see on the graphs in Figure 6, where bigger models reach a lower accuracy. These observations contradict what was observed by Bhojanapalli et al. [BCG+21], since on average it was observed that bigger models tend to have better robustness accuracy, and increasing the size of the model seemed to help increase its robustness.

A notable observation can be made regarding the Average Attack Cost metric, which reaches its highest value for $n\_steps = 3$ before decreasing for bigger models on the non-cost-bounded graph, but increases on the cost-bounded one. This means that if we limit the set of adversarial samples to those for which the difference with their original copy doesn't exceed the cost-bound (1.1 in our case), we need to alter more features to have a misclassification the bigger the model gets. However, if we don't specify any cost-bound at the evaluation step, we see that we need to alter fewer features to get a misclassification on average, the bigger the model gets.

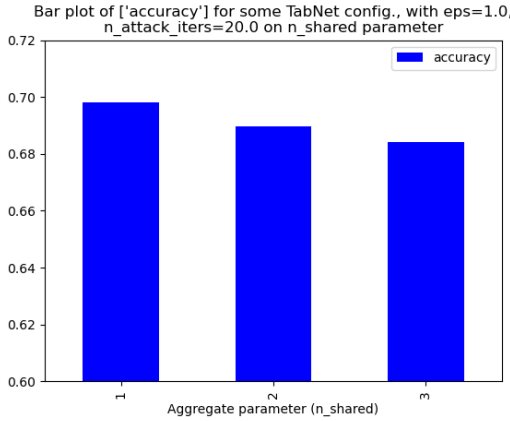
The second observation stays coherent with the observed trend with the accuracies, but the first one leads us to a contradiction in the data we have, going in the same direction as the results obtained by Bhojanapalli et al. [BCG+21].

3.2.2 Focusing on the *n_shared* parameter

Focusing on the Feature Transformers, we get the following plots for the *n_shared* parameter:

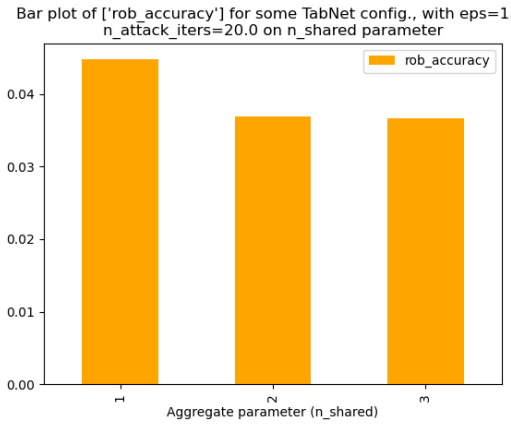


(a) Without cost_bound parameter

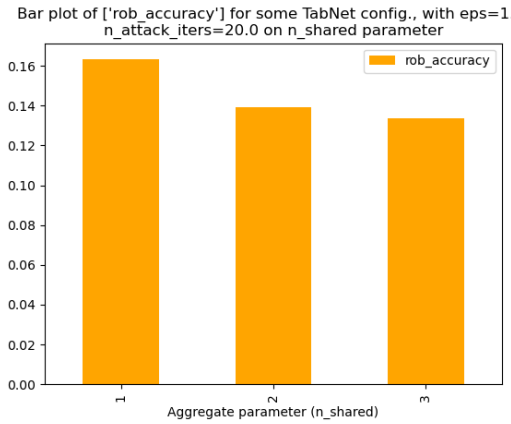


(b) With cost_bound=1.1

Figure 9: Bar plots of Test Accuracy metric, results averaged for each model on the *n_shared* parameter

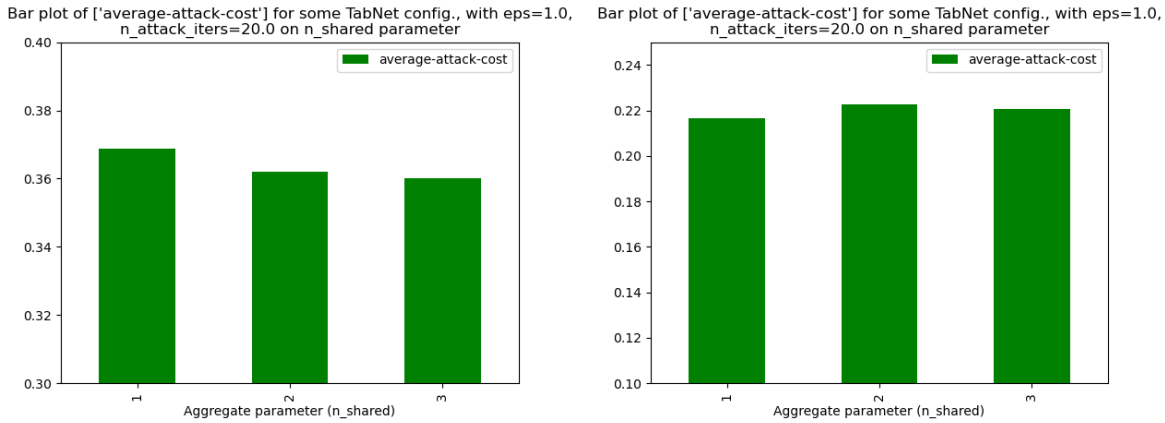


(a) Without cost_bound parameter



(b) With cost_bound=1.1

Figure 10: Bar plots of Robustness Accuracy metric, results averaged for each model on the *n_shared* parameter



(a) Without cost_bound parameter

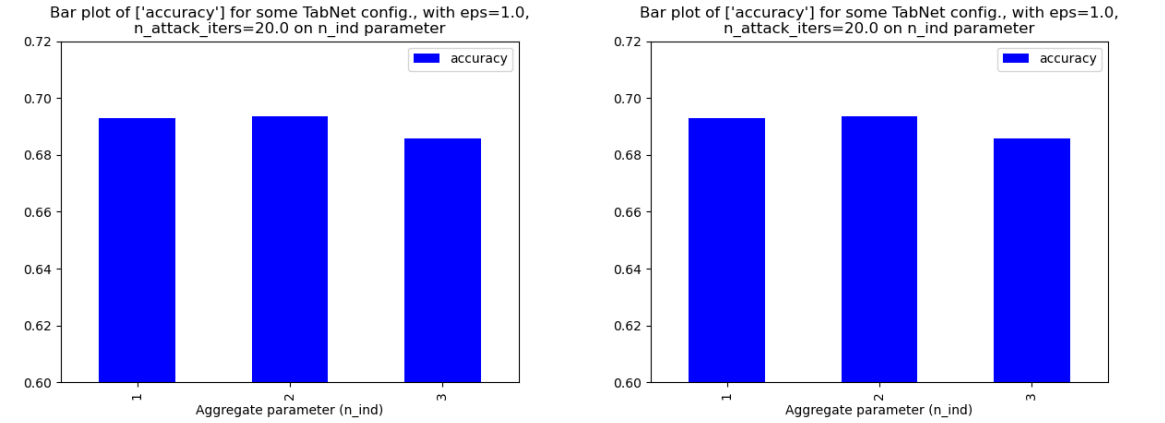
(b) With cost_bound=1.1

Figure 11: Bar plots of Average Attack Cost metric, results averaged for each model on the *n_shared* parameter

As we can see, we get the same trends on each metric as on the *n_steps* graphs, but the variations are much less pronounced, meaning that increasing the number of shared GLU layers in the Feature Transformers is pretty insignificant in regards to model robustness in our case. This joins what was found in Bhojanapalli et al.'s paper [BCG+21], where the removal of single layers did not show any significant decrease in model accuracy or robustness.

3.2.3 Focusing on the *n_ind* parameter

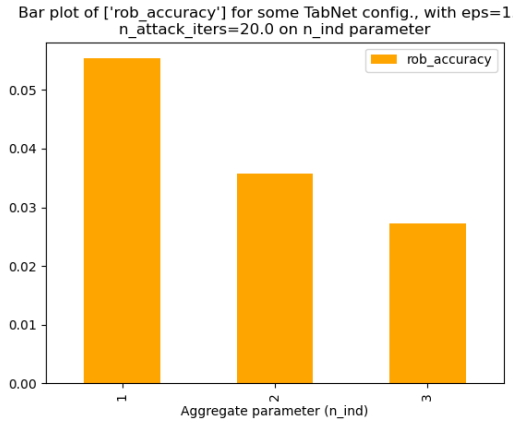
Finally, we get the following plots for the *n_ind* parameter:



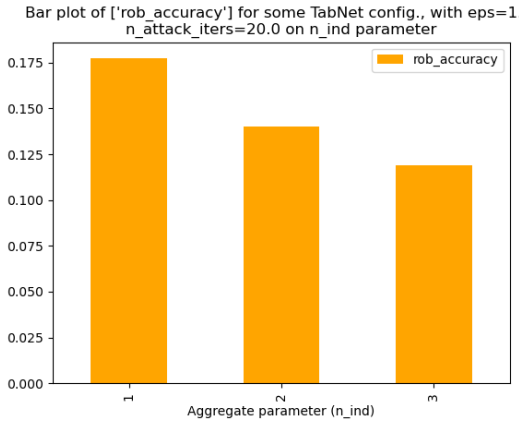
(a) Without cost_bound parameter

(b) With cost_bound=1.1

Figure 12: Bar plots of Test Accuracy metric, results averaged for each model on the *n_ind* parameter

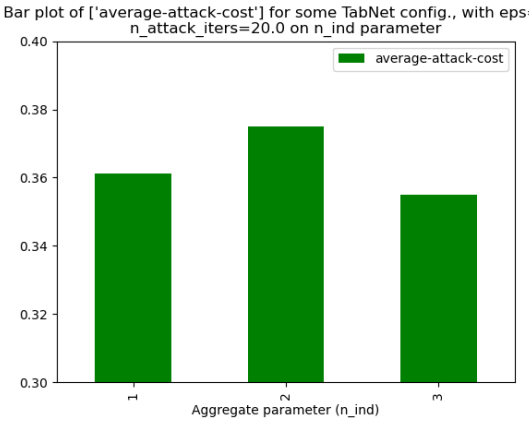


(a) Without cost_bound parameter

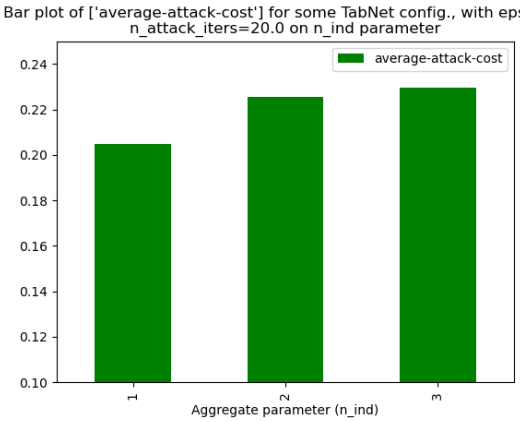


(b) With cost_bound=1.1

Figure 13: Bar plots of Robustness Accuracy metric, results averaged for each model on the *n_ind* parameter



(a) Without cost_bound parameter



(b) With cost_bound=1.1

Figure 14: Bar plots of Average Attack Cost metric, results averaged for each model on the *n_ind* parameter

Here, the observed trends resemble more those of the *n_steps* parameters than of the *n_shared* ones, with an exception being that the decrease in test accuracy is the least pronounced of the two parameters on both plots of Figure 12, meaning that the number of individual blocks of layers doesn't affect much the test accuracy of the model and that the number of individual blocks of layers in the Feature Transformers have a non-negligible impact on the overall robustness of the model.

However, regarding the Average Attack Cost metric, we also see that the maximum value is reached for the second value of the *n_ind* parameter at $n\_ind = 2$, exactly like what we saw in the *n_steps* case. This means that even though we get a decrease in robustness accuracy, it seems that the model configurations being the most robust to feature alteration are those for which $n\_steps = 3$ and $n\_ind = 2$, regardless of the value of the *n_shared* parameter.

4 Conclusion

We learned about TabNet, a new model whose purpose is to improve the performance of models on Tabular Data, for which Decision Tree models are the most popular choice, and also about the state of research on robustness for Transformer-based architectures like TabNet. We then trained and evaluated some models by leveraging the grid search method, and implemented a validation set

with early stopping to avoid over-fitting on models for our experiments. We also found out that for $\epsilon = 1.0$, the robustness of TabNet models decreases as the model size gets larger (i.e. as we add more and more decision steps), following the same trend as the test accuracy. Finally, we learned that the number of individual blocks of layers in the Feature Transformers had a significant impact on the robustness accuracy of models, following the same trend as the model size. To increase the coverage of the experiments, we should extend the set of values for each parameter or train for more parameters like the n_a and n_d ones, and also varied the ϵ parameter to see if for smaller and larger values of ϵ we still get the same results, or if the larger the ϵ value gets, the more robust the models get since more adversarial samples are properly classified. But due to time constraints, this couldn't be done before completing this report, and it is something that will be partially treated in the oral presentation of the project.

References

[AP20] Sercan O. Arık and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. *arXiv*, 2020.

[BCG+21] Srinadh Bhojanapalli, Ayan Chakrabarti, Daniel Glasner, Daliang Li, Thomas Unterthiner, and Andreas Veit. Understanding robustness of transformers for image classification. *arXiv*, 2021.

[BMR+20] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Amanda Askell et al. Language models are few-shot learners. *arXiv*, 2020.

[CAS+21] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Edoardo Debenedetti, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. Robustbench: a standardized adversarial robustness benchmark. *arXiv*, 2021.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *arXiv*, 2016.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

[ea15] Goodfellow et al. Explaining and harnessing adversarial examples. *ICLR*, 2015.

[LWU+20] Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. *Advances in Neural Information Processing Systems*, 2020.

[RPV+19] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. *Advances in Neural Information Processing Systems*, 2019.

[Tha20] Samrat Thapa. Implementing tabnet in pytorch. *Towards Data Science*, 2020.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[XBK+15] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[XKL+21] Haoyin Xu, Kaleab A. Kinfu, Will LeVine, Sambit Panda, Jayanta Dey, Michael Ainsworth, Yu-Chung Peng, Madi Kusmanov, Florian Engert, Christopher M. White, Joshua T. Vogelstein, and Carey E. Priebe. When are deep networks really better than decision forests at small sample sizes, and how? *arXiv*, 2021.