Dipartimento di Fisica e Astronomia Galileo Galilei

1222·2022 800 ANNI

Università degli Studi di Padova

Tree Tensor Network supervised classifier for High Energy Physics

Ardino Rocco Mat. 1231629 rocco.ardino@studenti.unipd.it
Valente Alessandro Mat. 1234429 alessandro.valente.4@studenti.unipd.it

(Dated: March 3, 2021)

In this work we discuss the implementation of a supervised Tree Tensor Network classifier using TensorFlow and TensorNetwork frameworks. To validate the method, we apply it to a classical signal-versus-background classification task on the HIGGS dataset, a standard in Machine Learning researches inside HEP field. So, we treat in detail the necessary preliminary operations, such as data preprocessing and Tree Tensor Network hyperparameters optimisation, needed to boost significantly the classifier performances. The results obtained are then presented, focusing also on the time performances of the algorithm. We conclude with a comparison of our approach with the results obtained by [1], in which a classical Deep Neural Network is used, and discussing also the possible improvements of our method.

CONTENTS

I. INTRODUCTION

a. High Energy Physics and Automatic Learning Understanding the nature of the particles that constitute matter and radiation is one of the main concerns of science and, in particular, of the branch of High Energy Physics (HEP). In this field, the theoretical description of how nature behaves has its climax in the so-called Standard Model. The ladder has proved to be extremely successful in predicting a wide variety of particle phenomena with great accuracy, but it does not give a full view of the intrinsic rules of nature. There are still lots of unanswered questions and still New Physics to discover.

However, due to the increasing complexity of data to analyse and to the more demanding selectivity in the interesting events, the current techniques used in HEP fail to capture all the available information. Moreover, it has already been proved by [1] that Machine Learning techniques such as Artificial Neural Networks can overcome this issue by building powerful classifiers. Among the list of the advantages tied to them, there are their capabilities to solve the curse of high dimensionality of data and their potential to compute in an unbiased and very efficient way arbitrary complex functions.

b. Supervised Learning with Tree Tensor Networks An interesting and promising interpretation of this solution is to exploit Quantum Tensor Networks, which are the quantum-inspired version of Biological Neural Networks. The power of tensor methods is being appreciated in several fields of Physics and more others. In particular, tensor decompositions can be used to solve non-convex optimisation tasks, such as minimising a certain function [3]. Moreover, tensor networks avoid the curse of high dimensionality by incorporating only-low order tensors.

Concerning the context of automatic and supervised learning tasks, several architectures of tensor networks have already been proved to be successful for these purposes, such as the so called Tree Tensor Networks (TTN). In particular, it is possible to apply these techniques to High energy Physics dataset, as showed in [2].

c. Outline Given these premises, in this work we exploit a TTN-based solution to build a classifier for a classical signal-over-background discrimination task on the same dataset employed in [1], so in HEP context. We will take as reference and comparison their results obtained with Deep Neural Networks, showing how with a reduced number of degrees of freedom we are still able to obtain a good accuracy in the task of classification. Moreover, we will deepen on the scalability of this method by showing how training and prediction time behave when increasing the complexity of the tensors. In this sense, we will discuss how an efficient implementation based on TensorFlow and TensorNetwork libraries can be achieved with great flexibility. Last but not least, we conclude the discussion with several possible improvements that can be done for future work in order to boost the classification and computational performances of the method.

II. THEORY

In this Section, we describe all theoretical concepts applied in this work. We begin in Subsection II.1 from a brief introduction on the main kinematic variables considered in HEP, in order to introduce clearly the notation for the features and define without ambiguity the quantities needed for the following discussion. Then, we move to the description of the Monte Carlo simulated dataset in Subsection II.2, pointing out the characteristics of the signal we are searching. The next step is a basic description of the Tensor Network methods needed to construct a Tree Tensor Network as well as the training algorithm, in Subsection II.3. Lastly, in Subsection II.4, we introduce several techniques applied to boost the performances of the learning routine of the standard TTN implementation.

1. Kinematics of High Energy collisions

Before introducing the processes studied for this work, it is necessary to introduce the relevant variables we are going to consider. For this purpose, we refer to the experiment of Large Hadron Collider (LHC) at CERN, where proton-proton (pp) collisions are performed in order to study from their products the fundamental structure of nature. A general collision event is represented in FIG. 1a, where the momentum direction of a product particle is highlighted as well as some relevant kinematic variables. In particular, we remark the projection of the produced particle momentum $\vec{\boldsymbol{p}}$ in the transverse plane (x, y), namely the transverse momentum p_{T}. The ladder is calculated through:

$$p_{\mathrm{T}} = \sqrt{p_x^2 + p_y^2} \quad . \tag{1}$$

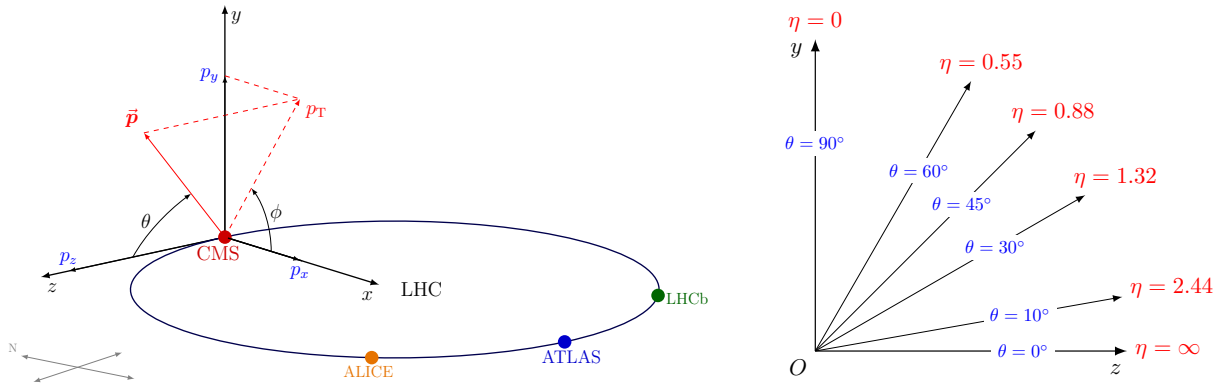
On the other hand, from the polar and azimuth angles θ and ϕ other practical features can be obtained, such as the pseudorapidity η:

$$\eta = -\log\left[\tan\left(\frac{\theta}{2}\right)\right] \quad , \tag{2}$$

and the difference $\Delta\phi$ with the azimuth angle of another product particle:

$$\Delta\phi = \phi_2 - \phi_1 \quad , \tag{3}$$

where the index identifies the first ($i = 1$) or the second ($i = 2$) particle.



(a) Kinematics of a collision

(b) Pseudorapidity visualisation

FIG. 1 LHC structure and kinematics of a product particle, emitted with a polar angle θ and azimuth angle ϕ, in 1a. In 1b, visualisation of the linking between the polar angle and the pseudorapidity η.

Concerning the fundamental particles produced in a collision, we have several possibilities, in particular heavy bosons such as the Higgs (h^0) and the $W^{\pm}$, or leptons, such as electrons (e^-), muons (μ^-) or neutrinos with the corresponding flavour (ν_e, ν_μ). Another important possibility is constituted by gluons (g) or quarks (q). However, the ladders are not directly revealed by a detector, but they undergo to a process of "hadronisation" forming a shower of particles called jet (j) and directed towards the direction of the gluon/quark causing them.

2. HIGGS dataset

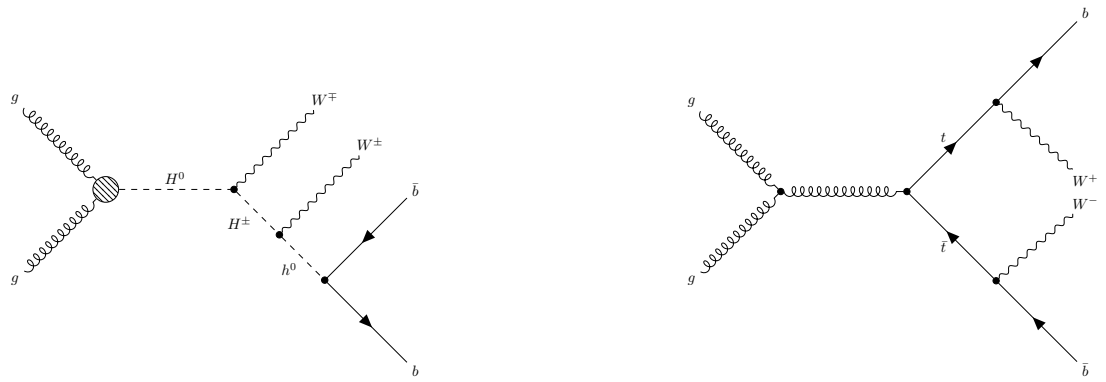
a. Dataset description The dataset taken into account for this work is a Monte Carlo generated sample of a process in which two gluons fuse together. In particular, the events are generated by simulating the "hard process" of proton-proton collisions at $\sqrt{s} = 8$ TeV within the framework MadGraph5. The events are then showered using Pythia, while the detector response is taken into account by Delphes.

In the process, a mass of the Higgs $m_{h^0} = 125$ GeV is assumed, while new exotic Higgs bosons H^0 and $H^\pm$ are introduced with $m_{H^0} = 425$ GeV and $m_{H^\pm} = 325$ GeV masses. The ladders are observed as resonant states over the SM background. Concerning the specific channels simulated for both signal and background events, we have:

$$\text{Signal:} \quad gg \to H^0 \to W^\mp H^\pm \to W^\mp W^\pm h^0 \to W^\mp W^\pm b\bar{b} \tag{4}$$

$$\text{Background:} \quad gg \to g \to t\bar{t} \to W^\mp W^\pm b\bar{b} \tag{5}$$

and a portrait of their development in the formalism of Feynamn diagrams is showed in FIG. 2. As it is possible to observe, the final states of the two channels are identical, so the background process will constitute an irreducible background contribution and the only method to find discrepancies with the signal is to look at the spectra of both low and high-level features distributions.



(a) Signal process (b) Background process

FIG. 2 Feynman diagrams for the Monte Carlo simulated events. In 2a, the diagram of the signal channel is portrayed with the exotic Higgs bosons H^0 and $H^\pm$. In 2b, the background diagram is represented.

Before looking at the aforementioned distributions, it is important to remark that the HIGGS dataset has been "cleaned" by [1] in order to keep only the interesting events that can be detected in a real experiment. For instance, we are not interested in particles that are not inside the geometric acceptance of a detector that should reveal them. Therefore, the events available for the classification task satisfy the following conditions:

- exactly one lepton ℓ, which could be an electron or a muon, with transverse momentum $p_\mathrm{T} > 20$ GeV and inside the geometric acceptance of a hypothetical detector, so a cut on the pseudorapidity $|\eta| < 2.5$. Note that the lepton comes from the decay $W^\pm \to \ell^\pm(\overline{\nu_\ell})$;
- at least four jets, each one with a transverse momentum $p_\mathrm{T} > 20$ GeV and with $|\eta| < 2.5$. Again, we remark that they can come from the b quarks or from the products of the $W^\pm$ decay into quarks;
- b-tags on at least two of the jets, indicating that their traces are likely due to b quarks rather than gluons or lighter quarks.

b. Features and their distributions The events satisfying these requirements are described by a set of low-level features, namely basic measurements made by the particle detector. In particular, these are:

- the p_T, η, ϕ and b-tag for each of the four jets;
- the p_T, η and ϕ of the lepton ℓ;
- the missing energy magnitude $\not{E}$ and azimuth angle ϕ, due to the presence of an invisible neutrino from the $W^\pm$ decay.

The plots of their distributions in both the cases of signal and background events are reported for brevity in Appendix **A.1**, in FIG. **13** for the leptonic part and in FIG. **14** for the hadronic part (jets).

On the other hand, we can combine the low-level features to construct new features carrying information of higher level, namely the high-level features. In particular, for signal processes, several resonant decays are theorised and the related invariant masses are computed. By this way, we get several high-level features:

- for signal processes, we expect a peak in the distributions of $m_{\ell\nu}$, m_{jj}, m_{bb}, m_{Wbb} and m_{h^0}, due to the resonant decays in the intermediate channels of the signal process;
- for the background, we expect a peak in the distributions of $m_{\ell\nu}$ and m_{jj}, as for the signal events, and in $m_{\ell\nu b}$ and m_{jjb} due to the resonant decay of the top quarks.

The plots of the seven listed high-level features are showed for brevity in Appendix **A.2**, in FIG. **15**.

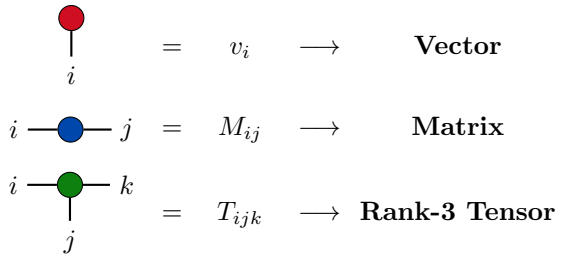
3. Tree Tensor Networks

a. Tensor Networks Before starting with the Machine Learning application of TTNs, we should firstly introduce the core of their structure, namely the Tensor Networks (TN). These can be seen as factorisations of high rank tensors into networks of smaller rank tensors and their application is becoming more and more diffused in scientific research.

Tensor networks come with an intuitive graphical language that can be used in formal reasoning and proofs [6]:

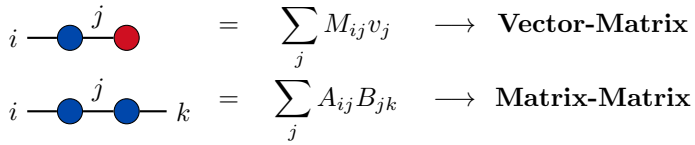
- tensors are notated by solid shapes, and tensor indices are notated by lines emanating from these shapes;
- connecting two index lines implies a contraction, or summation over the connected indices.

In order to have an insight of these graphic rules, let us consider some simple and common examples, such as the graphic notation for vectors, matrices and rank-3 tensors, which are the ones mainly used in this work:



(6)

On the other hand, let us sketch the graphical notation for some of the most common operations between tensors, such as vector-matrix and matrix-matrix multiplications:



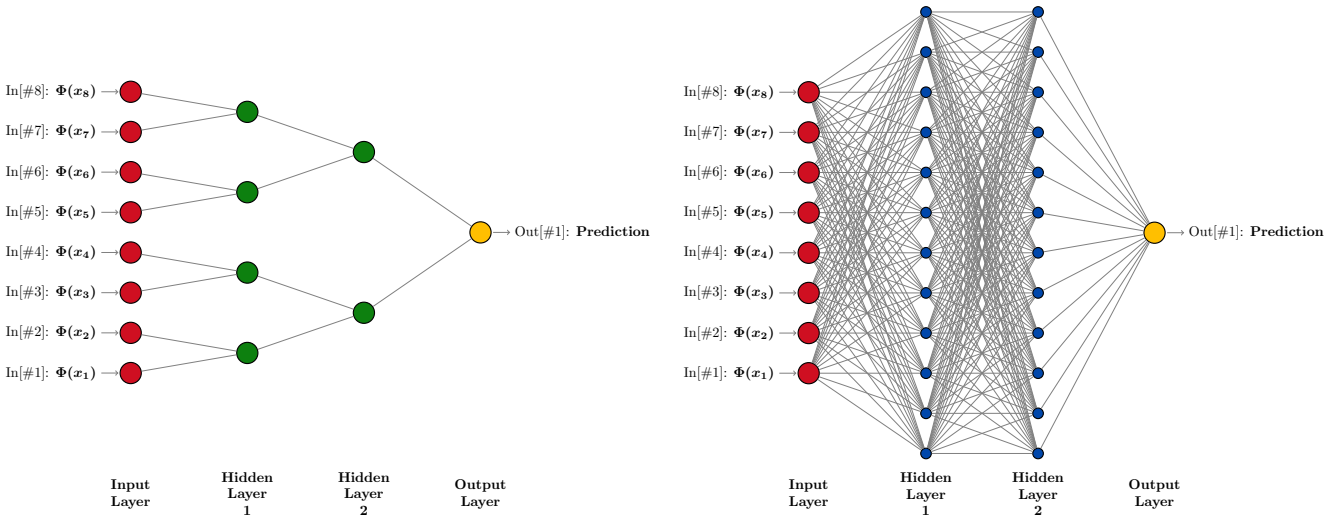
(7)

b. TN for machine learning Given the basic rules for tensor network contractions and composition, we introduce the architecture of a Tree Tensor Network (TTN) using the formalism of Quantum Mechanics. Starting from features x given in input to the network, these can be mapped into $\Phi(x)$ through an apposite feature map. Then, these quantities are fed to a tensor network with a tree-like structure, represented with the notation $T(w;\chi)$, where w are entries of the tensors, namely the weights of the network to be tuned by the learning algorithm, and χ is the so-called bond dimension. The ladder is the dimension of the bonds between tensor nodes in the TTN and it allows to control the complexity of the network in an efficient way: the larger it is, the higher the network weights are. The output of the TTN, which in the simplest case is a scalar quantity, can formally be written as the output of a decision function $f(x)$, namely:

$$f(x;w) = T(w;\chi) \cdot \Phi(x) \tag{8}$$

An example of the overall structure is represented in FIG. **3a**, where we remark the tree structure with the nodes being tensors of rank 3, except the final one being of rank 2. For comparison, a classical structure of Deep Neural Network is showed in FIG. **3b**, where each node represents a neuron in Deep Learning jargon. The main differences between them stands:

- in the approach, in fact DNNs are biological-inspired while TTNs are quantum-inspired;
- in the fact that quantities like correlation and entanglement entropy between the features are easily accessible in the case of TTNs, while this is complex for Neural Networks in general [2].



(a) Example of TTN structure

(b) Example of DNN structure

FIG. 3 Comparison between examples of structures for a TTN, in **3a**, and a DNN, in **3b**. In particular, in the TTN the dimension of the bonds between the tensors in the hidden layers is the bond dimension χ.

c. Learning algorithm Focusing on the TTN structure, our purpose is to make predictions as accurate as possible with respect to the true label associated to the training data. More precisely, given a the i^{th} training sample $x^{(i)}$ with a label $y^{(i)}_{\text{true}}$, we want that the output of the network $y^{(i)}_{\text{pred}}$ is in agreement with $y^{(i)}_{\text{true}}$. Given that this evaluation should be done for the whole training set of samples, we need to quantify how the TTN is wrongly classifying the class of events. Assuming that the true label is a number in the set $\{0, 1\}$, 0 corresponding to background and 1 to signal, and that the prediction is bounded in the interval $(0, 1)$, we can quantify this misclassification through a "loss function" like the binary cross-entropy:

$$L(y_{\text{true}}, y_{\text{pred}}) = \sum_{i}^{n} y_{\text{true}} \log\left(y^{(i)}_{\text{pred}}\right) + (1 - y^{(i)}_{\text{true}}) \log\left(1 - y^{(i)}_{\text{pred}}\right) \quad , \tag{9}$$

where the index i indicates the i^{th} sample of the dataset of n samples on which the loss function is evaluated. An important aspect of EQ. **9** is that it penalises heavily the predictions that are confident but wrong.

Now, we can define our learning algorithm. Our purpose is to reduce the misclassification on a dataset of samples reserved for the training phase, then we can evaluate the trained model on a dataset of validation samples and quantify the performances through a certain metric. Apart from the loss function, useful metrics are represented by:

- the **accuracy**, namely the fraction of correctly classified samples. The correct classification is based on a threshold ξ in $(0, 1)$ over which the prediction is considered signal, under which it is considered background;
- the **AUC**, namely the area under the Receiver Operating Characteristic (ROC) Curve. The ladder is the True Positive Rate (TPR) in function of the False Positive Rate (FPR), which in practice is obtained by repeating the classification for a sweep of threshold values ξ in $(0, 1)$. The advantage of the AUC is that it is directly linked to the concept of discovery significance commonly used in HEP [1].

Returning to the learning task, this is nothing more than an optimisation of the tunable parameters of the TTN, namely the weights w. There exists several methods to accomplish the task, but in general they belong to a class of algorithms called optimisers. Among these, a common choice in Machine Learning field is the ADAM algorithm, which adapts the correction to the weights at each step parameter per parameter. Its routine is sketched in ALG. 1. It is important to remark that the training dataset is divided into batches of dimension m and the weights are updated after having processed the whole batch. When all the batches are processed, we say that a training epoch is ended and we repeat the optimisation for a desired number of epochs or until a certain condition on the variation of the chosen metric is met.

Algorithm 1 ADAM algorithm [5].

Require: Step size ε (suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$ (suggested defaults: $\rho_1 = 0.9$, $\rho_2 = 0.999$)
Require: Small constant δ, usually 10^{-8}, used to stabilise division by small numbers
Require: Initial weights $\boldsymbol{w}$
Require: Minibatch dimension m
procedure ADAM($\{x^{(i)}\}_{i=1,\dots,n}$) ▷ Input: training dataset of n elements
  Initialize 1st and 2nd moment variables, $s = 0$, $r = 0$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample minibatch $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
    $g \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{w}} \sum_i L(f(x^{(i)}; \boldsymbol{w}), y^{(i)})$ ▷ Compute gradient estimate
    $t \leftarrow t + 1$
    $s \leftarrow \rho_1 s + (1 - \rho_1) g$ ▷ Update biased 1st moment estimate
    $s \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$ ▷ Update biased 2nd moment estimate
    $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$ ▷ Correct bias in 1st moment
    $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$ ▷ Correct bias in 2nd moment
    $\Delta \boldsymbol{w} = -\varepsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$ ▷ Compute parameter update
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta \boldsymbol{w}$
  end while
  return $\boldsymbol{w}$
end procedure

The last step is evaluating the generalisation power of the trained model on a validation dataset, which is not processed in the training procedure. Moreover, when tuning the hyperparameters of the network, such as the bond dimension or the feature map, we choose the set of parameters which gives the best results for the metrics on the validation dataset and we finally recompute the generalisation power of the network on another never-seen test dataset.

4. Advanced techniques for performances boosting

Hereafter we further discuss on several techniques taken from Deep Learning field to improve the performances of the TTN, namely reducing the generalisation error and to speed up the convergence of the training procedure to an optimal solution.

a. Activation function When dealing with a TTN prediction, we should take into account that tensor contractions are linear operations. Therefore, we would have a limit on the space of functions that they are able to approximate. In order to enlarge it, it is necessary to add a source of non-linearity inside the recipe and this can be easily accomplished by adding an activation function after the output of every TTN layer of nodes. Although there are lots of choices whose success has been scientifically proved, we focus just on the expressions of the ones employed in this work. The first one is the Exponential Linear Unit (ELU), a very common choice in Deep Learning algorithms

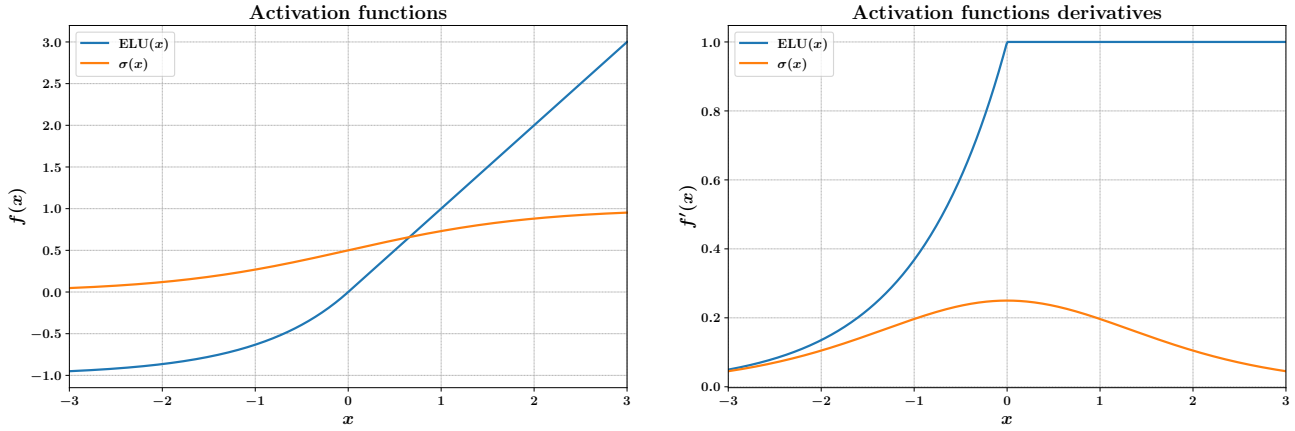
due to its effectiveness and non-linearity, and defined as:

$$\mathrm{ELU}(x) = \begin{cases} x & x \geq 0 \\ a(e^x - 1) & x < 0 \end{cases} \quad . \tag{10}$$

The second one is sigmoid $\sigma(x)$, often used as activation of the output layer since it is bounded in $(0, 1)$ and so it is functional for making predictions. It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad . \tag{11}$$

Their plots are showed in FIG. 4a, while their derivatives are showed in FIG. 4b. For completeness, a detailed treatment of other activation functions aside from ELU and sigmoid is given in [4] along with a discussion on their effectiveness in practical applications.



(a) Activation functions

(b) Derivatives of activation functions

FIG. 4 Comparison between ELU and sigmoid activation functions in 4a, and between their derivatives, in 4b, which are used also for loss function gradient calculation.

b. Kernel regularisation In general, a regularisation is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. In practice, the most common way to regularise a model like $f(x; w)$ is to add a penalty for large weights to the loss function. Keeping them small often leads to more stable and general solutions and allows to train the model for a longer number of epochs before it starts to overfit. An example is the ℓ_2 regularisation, which leads to the minimisation of:

$$J(y_{\mathrm{true}}, y_{\mathrm{pred}}) = L(y_{\mathrm{true}}, y_{\mathrm{pred}}) + \lambda \sum_{i=1}^{n_{\mathrm{weights}}} |w_i|^2 \quad , \tag{12}$$

where λ is a parameter that increases or reduces the relevance of the penalty depending on its magnitude.

c. Batch normalisation Training networks with multiple layers can be more or less challenging. In fact, the model is updated layer-by-layer backward from the output to the input using an estimate of error that assumes the weights in the layers prior to the current layer are fixed. Because all layers are changed during an update, the update procedure is forever chasing a moving target. Batch normalization provides an elegant way of reparameterising almost any Deep Network. The reparameterisation significantly reduces the problem of coordinating updates across many layers and, so, it significantly speed up the training.

Batch normalization can be implemented during training by calculating the mean and standard deviation of the output of each layer, before applying the activation, for every batch of the training dataset. Using these statistics it is possible to perform a standardisation inside the network at the output of each layer.

III. CODE IMPLEMENTATION

In this Section we discuss the practical implementation of a Tree Tensor Network classifier with TensorFlow and TensorNetwork libraries (for their documentation, see [7; 8]). We begin with a brief introduction on the preliminary operations applied before the initialisation of the learning algorithm, such as data preprocessing, in Subsection **III.1**. Then, we present in Susbsection **III.2** the TTN construction with dedicated TensorFlow layers in which tensor contractions are managed by TensorNetwork. Lastly, in Subsections **III.3** and **III.4** we introduce the implementation of a method for flexible TTN building and training.

Before going on, we remark the notation employed in this Section. We refer to the j^{th} feature of the i^{th} sample in the dataset as $x_j^{(i)}$. In particular, we have that $i \in [1, n_{\text{d}}]$ and $j \in [1, n_{\text{f}}]$, with n_{d} and n_{f} defined as the number of samples and features in the dataset, respectively. In our case, we have:

$$\begin{aligned} n_{\text{d}} &= 11 \cdot 10^6 \\ n_{\text{f}} &= 28 \end{aligned} \quad . \tag{13}$$

1. Data preprocessing

The HIGGS dataset provided by [1] comes out in a format already normalised:

- the features with negative values are assumed either normally or uniformly distributed, so their distribution is centred and rescaled by the standard deviation;
- the features with only positive and large values are rescaled by just setting their mean to 1.

This is a common choice in HEP context, since it preserves the intrinsic structure of the features distributions.

a. Rescaling Due to the feature maps we are going to apply, we further rescale the dataset in order to bound the features in $[-1, 1]$ for the ones with negative values, in $[0, 1]$ for the ones positively defined. So, we apply the following transformation before mapping:

$$x_j^{(i)} \longrightarrow x_j^{(i)\prime} = \frac{x_j^{(i)}}{m_j} \quad , \tag{14}$$

with

$$m_j = \max_{i \in \mathcal{D}_{\text{train}}} \left| x_j^{(i)} \right| \quad , \tag{15}$$

where $\mathcal{D}_{\text{train}}$ is the part of dataset reserved for training the TTN[1]. This procedure is performed using the **Standardize** function, sketched in LST. **1**.

```
def Standardize(x, nt):
    """
    Standardize each feature diving by the absolute maximum, distributions will be:
    > in [-1, 1] for negative feature
    > in [ 0, 1] for positive ones.
    """

    for j in range(x.shape[1]):                    # loop over features
        vec      = x  [:, j]                       # get feature vector
        vec_norm = vec[:nt]                        # take only training part
        vec      = vec / np.max(np.abs(vec_norm))  # normalize
        x[:,j]   = vec
    return x
```

LST. 1 Implementation of the standardisation of dataset in the preprocessing phase of rescaling.

[1] In Machine Learning field, in order to obtain a truthful estimation of the goodness of a model, the validation and test set should not be "seen" by the model during an epoch of training. This practical rule holds also during the dataset preprocessing. So, the preprocessing constants such as the means, standard deviations and maxima are obtained from the training set and applied to the full dataset.

b. Padding At the input layer of the TTN, as we will see in the later discussion, we contract at least two features per tensor node and, in general, at least two legs per tensor node in the following layers, keeping constant the number of legs contracted. In order to do this without problems, the total number of input features must be a power of the number of features we want to contract. Since this is not true in general, as we have 28 features, a possible solution is to add fictitious features by padding in order to reach the desired size of input legs. The implementation of this preprocessing operation is sketched in LST. 2 with the function **PadToOrder**. The ladder, given the original dataset $\mathcal{D}$ with n_{f} features and the number of features n_{con} to contract per tensor node, returns a properly padded dataset with a number $\tilde{n}_{\text{f}}$ of features calculated as:

$$\tilde{n}_{\text{f}} = \min_{n\in\mathbb{N}} \{n \geq n_{\text{f}} \ : \ n = (n_{\text{con}})^m \ , \ m \in \mathbb{N}\} \quad . \tag{16}$$

```python
def PadToOrder(x, con_order):
    """
    Pad the dataset with fictitious features in order to reach the minimum:
    \tilde(n)_(f) = (con_order)^(m)
    with m \in N
    """

    # number of padding features
    n_pad = int(
        con_order**(
            math.ceil(
                math.log(x.shape[1], con_order)
            )
        ) - x.shape[1]
    )

    # pad dataset
    x = np.append(x, np.zeros((x.shape[0], n_pad)), axis=1)
    return x
```

LST. 2 Implementation of the padding of dataset in the preprocessing phase of padding.

c. Feature Map As last operation of the preprocessing phase, we apply a feature map to the padded dataset in order to enhance the capability of the classificator to span more types of combinations of the physical quatities and to improve the discrimination of signal from background events. There exists a wide variety of maps in Machine Learning field, however, due to time limitations, we explore only two types. The first one is a polynomial map of order d, whose expression is given by:

$$\Phi_d^{\text{pol}}(x) = \left[1, x, \dots, x^d\right] \quad . \tag{17}$$

The ladder is a very common choice in classification tasks. The second type explored is a quantum-inspired spherical map of order d, whose expression is given by:

$$\Phi_d^{\text{sph}}(x) = \left[\phi_d^{(1)}(x), \dots, \phi_d^{(d)}(x)\right] \quad , \tag{18}$$

with the single vector components expressed as:

$$\phi_d^{(s)}(x) = \sqrt{\binom{d-1}{s-1}} \left(\cos\left(\frac{\pi}{2}x\right)\right)^{d-s} \left(\sin\left(\frac{\pi}{2}x\right)\right)^{s-1} \quad . \tag{19}$$

In order to understand the physical meaning of the spherical maps, let us consider the simplest situation of $d = 2$. In this case, it maps x into a spin vector.

The implementation of this preprocessing operation is sketched in LST. 3 with the functions **PolynomialMap** and **SphericalMap**. The ladders, given the padded dataset $\mathcal{D}_{\text{pad}}$ with $\tilde{n}_{\text{f}}$ features, return a mapped dataset with dimensions:

- $n_{\text{samples}} \times \tilde{n}_{\text{f}} \times (d+1)$ for the polynomial map of order d;
- $n_{\text{samples}} \times \tilde{n}_{\text{f}} \times d$ for the spherical map of order d.

```python
def SphericalMap(x, order=2, dtype=np.float32):
    """
    Apply spherical map of order d=order to input dataset x
    """

    x_map = np.zeros((x.shape[0],x.shape[1],order), dtype=dtype)
    for i in range(order):
        comb_coef    = np.sqrt(scipy.special.comb(order-1,i))
        x_map[:,:,i] = comb_coef * np.power(np.cos(x),order-1-i) * np.power(np.sin(x),i)

    return x_map


def PolynomialMap(x, order=2, dtype=np.float32):
    """
    Apply polynomial map of order d=order to input dataset x
    """

    x_map = np.zeros((x.shape[0],x.shape[1],order+1), dtype=dtype)
    for i in range(order+1):
        x_map[:,:,i] = np.power(x,i)

    return x_map
```

LST. 3 Implementations of the spherical and polynomial mappings of dataset in the preprocessing phase of mapping.

Given all the discussed phases of input data preprocessing, we summarise them in FIG. 5, where they are represented in logical order.

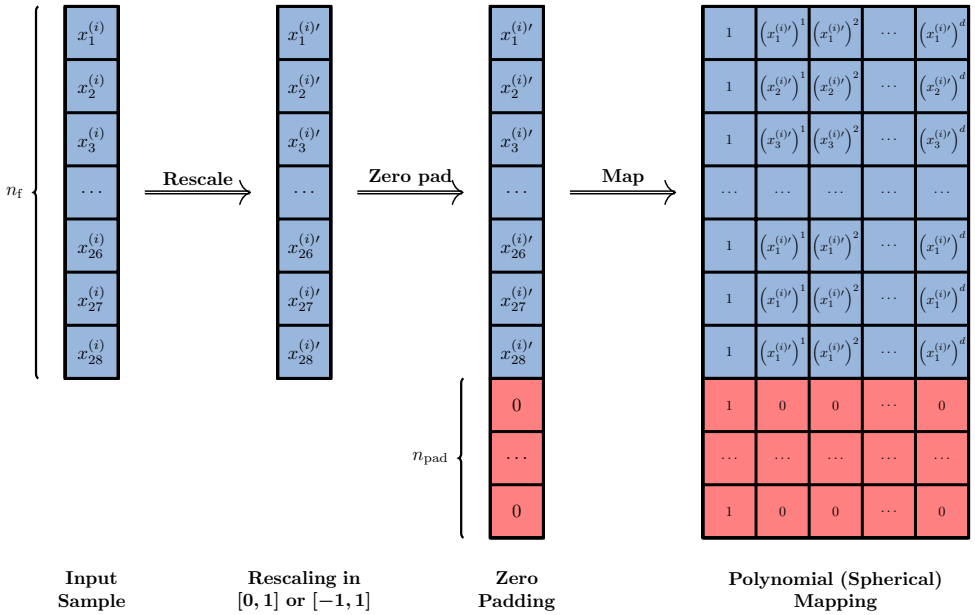


FIG. 5 Workflow of preprocessing precedure, starting from the rescaling of data, going through the zero padding in order to get proper dimensions for the input of the TTN, and lastly the polynomial or spherical mapping.

2. TTN layers in TensorFlow framework

The TTN model is created exploiting the power of TensorFlow, a widely diffuse Machine Learning framework, for automatic differentiation and of TensorNetwork for contractions between tensor nodes. In particular, we choose the Keras API for TensorFlow, in order to improve the code readability. Given these premises, here we discuss in detail the implementation of a layer of tensor nodes, denoted as **`TTN_SingleNode`**. By the concatenation of such layers, we are able to build a TTN model with high performances, due also to the highly optimised frameworks employed. Moreover, its training and evaluation can be easily managed by CPU or GPU by simply switching an apposite flag, with the possibility of running in parallel on both of them. Now, let us focus on the core points of the layer implementation.

a. Layer initialisation The layer `TTN_SingleNode` is implemented as a Python class and each layer of the final TTN model is an istance of it. As every class, when a layer object is instantiated, an apposite initialisation function is run with different input parameters carrying information on the characteristics of the layer itself. Some of the most important input parameters are:

- **`n_contraction`**: the number of features to contract in each node site, namely the previously defined n_{con};
- **`bond_dim`**: the dimension χ of the bonds between the tensor nodes inside the TTN;
- **`input_shape`**: the shape of the input samples, namely $\tilde{n}_{\text{f}} \times (d+1)$ for the polynomial map and $\tilde{n}_{\text{f}} \times d$ for the spherical map;
- **`activation`**: string carrying the name of the activation function to use, if specified;
- **`use_bias`**: boolean variable that introduces a bias weight vector to sum in every tensor node of the TTN, if true is specified;
- **`use_batch_norm`**: boolean variable to introduce the batch normalization of the layers, if true is specified.

Hence, starting from the number of features $\tilde{n}_{\text{f}}$ and the number of features to contract n_{con}, we initialise $\frac{\tilde{n}_{\text{f}}}{n_{\text{con}}}$ tensors of rank $n_{\text{con}}+1$ for the input layer. The ladders will have d or $(d+1$-length along the first n_{con} dimensions and χ along the last one. Moreover, if specified, a rank-2 tensor of dimension $\frac{n_{\text{f}}}{n_{\text{con}}} \times \chi$ is also initialised. This is the so-called bias, which will be summed to the output of the layer. The core code for the instantiation of the layer is sketched in LST. 4.

```python
class TTN_SingleNode(Layer):

    def __init__(
        self,
        n_contraction      : int,
        bond_dim           : int,
        use_bias           : Optional[bool] = True,
        activation         : Optional[Text] = None,
        kernel_initializer : Optional[Text] = 'glorot_uniform',
        bias_initializer   : Optional[Text] = 'zeros',
        kernel_regularizer                  = None,
        **kwargs
    ) -> None:

        # Allow specification of input_dim instead of input_shape
        if 'input_shape' not in kwargs and 'input_dim' in kwargs:
            kwargs['input_shape'] = (kwargs.pop('input_dim'),)

        # initialize parent class
        super().__init__(**kwargs)

        self.n_contraction      = n_contraction
        self.bond_dim           = bond_dim
        self.nodes              = []
        self.use_bias           = use_bias
```

```
        self.activation         = activations.get(activation)
        self.kernel_initializer = initializers.get(kernel_initializer)
        self.bias_initializer   = initializers.get(bias_initializer)
        self.kernel_regularizer = regularizers.get(kernel_regularizer)
```

LST. 4 Implementation of the tensor nodes initialisation in the `TTN_SingleNode` layer.

b. Edges connection After their initialisation, the legs of tensor nodes are connected to the tensors for the input samples. This operation is accomplished by casting the tensor objects with weights from TensorFlow (`Tensor` types) to TensorNetwork (`Node` types). The advantage of this method is the possibility of an easy selection of the tensor edges that should be be connected. So, it allows to select without ambiguity on which index the contraction should be performed. In order to do this selection for every leg, two for loops are used:

- in the first one, the `Node` objects are instantiated;
- in the second one, their edges are connected using the built-in TensorNetwork ^ operator.

This passages are sketched in LST. 5, namely a piece of code from the `TTN_SingleNode` class definition.

```
x_nodes  = []
tn_nodes = []

for i in range(len(nodes)):
    for j in range(n_contr):
        # create feature nodes
        x_nodes.append(tn.Node(x[n_contr*i+j], name='xnode', backend="tensorflow"))
    # create ttn node
    tn_nodes.append(tn.Node(nodes[i] , name=f'node_{i}', backend="tensorflow"))

for i in range(len(nodes)):
    for j in range(n_contr):
        # make connections
        x_nodes[n_contr*i+j][0] ^ tn_nodes[i][j]
```

LST. 5 Implementation of the connections between tensor nodes in the `TTN_SingleNode` layer.

It is important to remark that the computations needed to perform the contractions are not executed in this step. In fact, the real contraction is executed in a separated loop. Moreover, at first sight the use of three loops operating on the same object may not seem efficient. However, this choice enhances the parallelisation of the full computational task thanks to the inner structure of TensorNetwork library.

c. Contractions After connecting all the legs of the tensor nodes in every layer of the TTN model, the real contraction is performed using the `greedy` contractor from the TensorNetwork library. Lastly, after the contraction computations the bias vector is added to the result, if explicitly specified at the layers instantiation. The code implementing this part of the algorithm is sketched in LST. 6.

```
result = []

for i in range(len(nodes)):
    result.append(
        tn.contractors.greedy([x_nodes[n_contr*i+j] for j in range(n_contr)]+[tn_nodes[i]])
    )

result = tf.convert_to_tensor([r.tensor for r in result])

if use_bias:
    result += bias_var
```

LST. 6 Implementation of the contractions between tensor nodes in the `TTN_SingleNode` layer.

Note that other types of contractors are present inside the TensorNetwork library and, in particular, the mostly optimised one is not employed. The reason under this choice is related to the fact that the use of an optimised contractor, despite allowing a faster contraction, does not allow the vectorisation of the computations inside the layer[2].

d. Input vectorisation The contractions of the weight tensors with the input samples are parallelised in order to speed up both the training and the evaluation speed. This operation is performed using the **`vectorized_map`** function from TensorFlow library. In particular, it allows the complete parallelisation of the workflow during the training and prediction procedures. After this part, the activation function is applied, if specified. A sketch of how these methods are implemented is showed in LST. 7.

```
# prepare input data for the vectorization of the contract function
input_shape = list(inputs.shape)
inputs      = tf.reshape(inputs, (-1, input_shape[1], input_shape[2]))
# vectorize the contraction over all the input samples
result = tf.vectorized_map(  # vectorize
    lambda vec: contract(    # create a lambda function to be vectorized
        vec                , # input sample
        self.nodes         , # weight tensors
        self.n_contraction , # number of feaqture to contract
        self.use_bias      ,
        self.bias_var        # bias tensor
    ),
    inputs                   # input dataset over which vectorize the lambda function
)

# apply activation, if specified
if self.activation is not None:
    result = self.activation(result)
```

LST. 7 Implementation of the input vectorisation through `vectorized_map` in the `TTN_SingleNode` layer.

Now, given all the discussed phases of `TTN_SingleNode` implementation, we summarise them in FIG. 6, where they are represented in logical order with also the main phases of preprocessing procedure. Note that the previous discussion is focused on the input layer of the TTN. The workflow and implementation for the following layers is the same with the only exception that the tensor nodes in the middle layers are connected with the output of the previous layer.

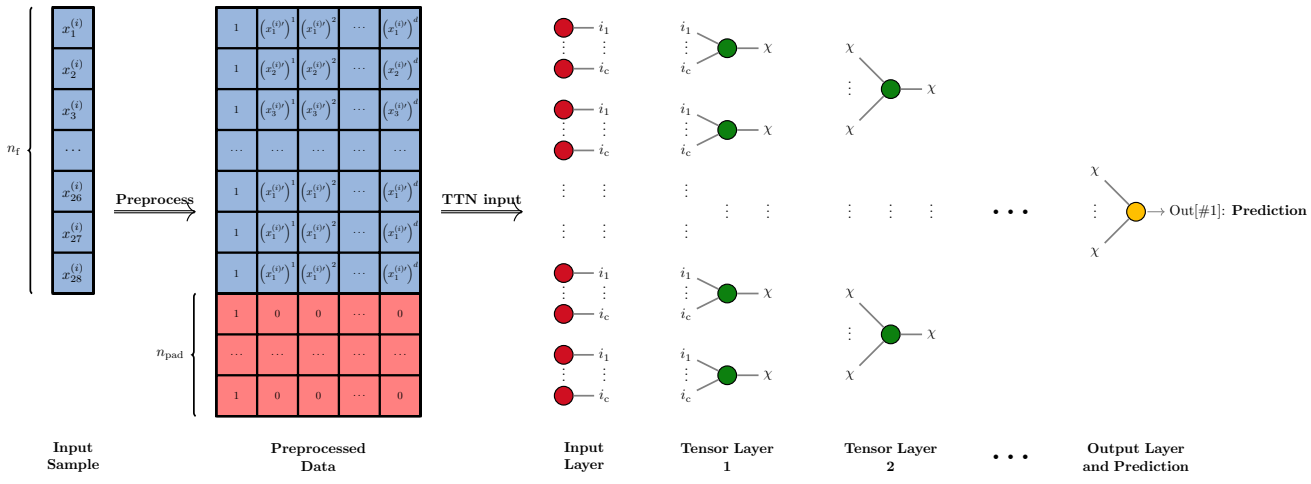


FIG. 6 Schematic representation of workflow of the data preprocessing and of the TTN structure, with $i_c = i_{n_{\text{con}}}$.

[2] The reason for this behaviour is that inside the TensorFlow library the operations are converted to a graph and vectorised. On the other hand, optimised contractors can not be converted to a graph since their contraction order is not predefined (it depends on the tensors themselves), resulting in a single faster step but slowing down significantly the whole execution.

3. Model building

The model building is divided into two steps:

- in the first step we create the structure of the TTN using a series of `TTN_SingleNode` layers described before;
- the second step is the compilation of the model into TensorFlow framework, needed to get fast and optimised code.

Hereafter we describe in detail both of these steps.

a. Model structure In order to automatise the model structure construction with `TTN_SingleNode` layers, an apposite function **`Make_SingleNode_Model`** is implemented. The ladder accepts as input parameters:

- **`input_shape`**: the shape of the input samples, namely $\tilde{n}_\mathrm{f} \times (d+1)$ for the polynomial map and $\tilde{n}_\mathrm{f} \times d$ for the spherical map;
- **`n_contraction`**: the number of features to contract in each node site, namely the previously defined n_con;
- **`bond_dim`**: the dimension χ of the bonds between the tensor nodes inside the TTN;
- **`activation`**: string carrying the name of the activation function to use, if specified;
- **`use_bias`**: boolean variable that introduces a bias weight, if true is specified;
- **`use_batch_norm`**: boolean variable to introduce the batch normalization of the layers, if true is specified;
- **`kernel_regulariser`**: a TensorFlow regulariser object to introduce the regularisation;
- **`verbose`**: a flag to enable intermediate information printing, useful for debug.

In order to give an idea of the internal structure of the model constructor, we report in Appendix **A.3** in LST. **10** the core part of the function to construct a "pure" model, namely without the application of advanced techniques like normalisation and regularisation. On the other hand, we report in the same Appendix in LST. **11** the core part capable of constructing more sophisticated models with the employment of advanced techniques.

b. Model compilation The second and last step in the definition of a TTN model structure, namely its compilation inside TensorFlow framework, is needed before beginning model training. In this phase it is possible to specify hyperparameters like the loss function and the optimisation algorithm to use. For this work, only the `binary_crossentropy` loss function and the `adam` optimiser are tested. Other arguments are the metrics of interest, such as the accuracy and the AUC, which are monitored during the training procedure on both training and validation sets. An example of the implementation of the compilation instructions is sketched in LST. **8**.

```
tn_model.compile(
    optimizer = 'adam',                  # optimizer for training
    loss      = 'binary_crossentropy',   # loss function to minimize
    metrics   = ['accuracy', 'AUC']      # metrics to monitor
)
```

LST. 8 Implementation of the compilation of the TTN model inside TensorFlow framework.

4. Model training

In the previous Subsections we have discussed the implementations of input dataset preprocessing and of the `TTN_SingleNode` layer, as well as the construction of a TTN model structure. After these operations, it is possible to move to the training of the TTN. The training procedure is done using the standard TensorFlow automatic differentiation methods for weights update. In particular, one can specify two hyperparameters aside from the training samples to use, namely:

- the number of **epochs** of training;
- the **batch size**, namely after how many samples processed the weights should be updated.

It is important to remark that, depending on the available memory of the hardware, a rule of thumb is to use bigger batch sizes when training on GPU. This choice allows to speed up the computations due to a more efficient exploiting of the higher number of GPU cores with respect to the CPU ones. However, as a trade-off, the model requires a larger number of training epochs in order to reach the comparable performances.

As last remark, a full example of implementation of all the previously discussed steps, from data preprocessing to model training, is sketched in LST. **9**.

```
# load N samples from dataset
data = pd.read_csv(
    DATA_PATH + 'HIGGS.csv.gz',
    nrows           = N,
    compression     = 'gzip',
    error_bad_lines = False,
    header          = None
)

# preprocess the data and split in train, validation and test sets
# preprocess applying polynomial order 3 map and pad to contract 2 feratures at time
x_train, x_val, x_test, y_train, y_val, y_test = preprocess.Preprocess(
    data,
    feature_map = 'polynomial',
    map_order   = 2,
    con_order   = 2,
    N_train     = N_train,
    N_val       = N_val,
    N_test      = N_test,
    verbose     = True
)

# create TTN model
tn_model = Make_SingleNode_Model(
    input_shape    = (x_train.shape[1:]),
    bond_dim       = 25,
    n_contr        = 2
    activation     = 'elu',
    use_batch_norm = True
)

# compile model
tn_model.compile(
    optimizer = 'adam',
    loss      = 'binary_crossentropy',
    metrics   = ['accuracy', 'AUC']
)

# train model on GPU
with tf.device('/device:gpu:0'):
    history = tn_model.fit(
        x_train, y_train,
        validation_data = (x_val,y_val),
        epochs          = 150,
        batch_size      = 5000
    )
```

LST. 9 Example of a complete workflow, from data preprocessing to model training on GPU.

IV. RESULTS AND DISCUSSION

We move now to the results obtained from the previously discussed implementation of the TTN structure and learning algorithm. So, in this part, in Subsection **IV.1** we begin with a rapid comparison between the performances of a "pure" TTN model, previously defined, with a more advanced one with regularisation and batch normalisation techniques. After this point, in Subsection **IV.2** we go on with the characterisation of the advanced structure, in particular with a timing and performance analysis depending on the bond dimension χ, map type and map order d. To conclude, in Subsection **IV.3** we treat in detail the analysis of the results for the most successful architecture of the TTN trained for this work.

Last but not least, we remark that due to the dimension of the dataset and to the type of computational task, the computational load of the problem is faced using the resources of CloudVeneto computing infrastructure [9]. In particular, the following results are obtained using a Virtual Machine (VM) with 15 virtual cores, 90 GB of RAM and with an NVIDIA Tesla T4 Graphic Processor Unit for TTN training.

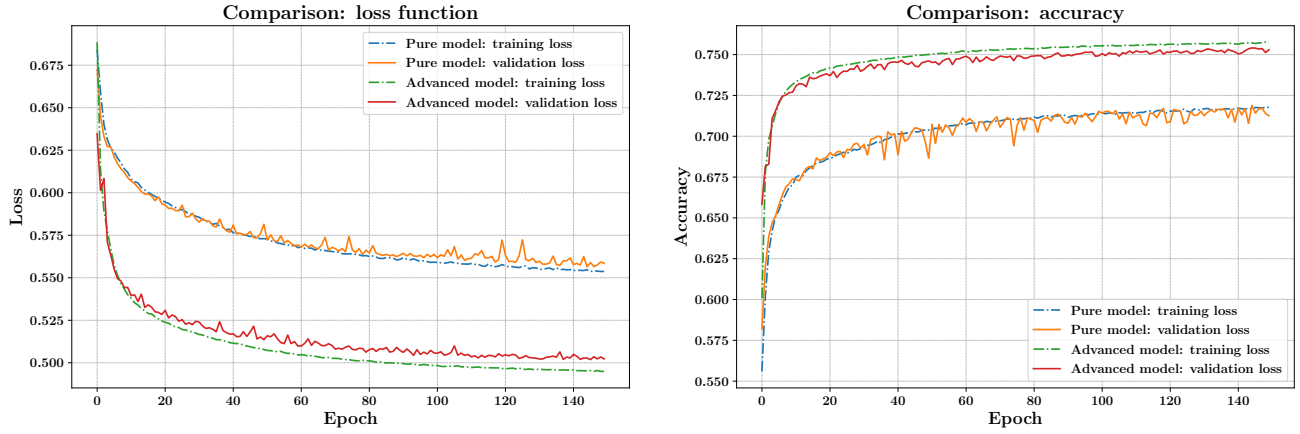
1. Comparison between "pure" and advanced TTN models

The first study performed is a comparison between the a "pure" TTN model, namely a model without regularisation, batch normalisation and activation functions in the inner layers, and a model in which some of the most common Machine Learning techniques are included. In particular, we describe more in detail the additions:

- after each layer the Exponential Linear Unit activation function is applied to the output data;
- the ℓ_2 regularisation is imposed to the weights of the tensor nodes;
- batch normalisation is employed after each tensor layer.

The dataset is preprocessed using an order 2 spherical map, which corresponds to having each input features represented by a spin; the models on the other hand are built using a bond dimension of 10 and with a 2-feature contraction architecture.

In order to understand the behaviour of the two aforementioned models, these are trained on a smaller training set of $n_{\text{train}} = 10^6$ samples. Concerning their performances, these are quantified by the metrics of accuracy and loss, which are monitored over the epochs on both training and validation sets, with the ladder composed of $n_{\text{val}} = 5 \cdot 10^5$ samples. The comparison for the loss metric is showed in FIG. **7a**, while for the accuracy metric it is showed in FIG. **7b**.



(a) Loss metric over training epochs

(b) Accuracy metric over training epochs

FIG. 7 Comparison between the "pure" TTN model and the advanced one. The trend of the cost function and of the accuracy score during the training are showed in **7a** and **7b**, respectively.

It is possible to observe how both the models, after the training procedure, have learned to discern signal from background in a way not exactly accurate, but significant. In particular, the advanced model performances overcome the performances of the more simple "pure" TTN model on both the training and validation sets. Another thing to

remark is the higher stability of the advanced model, visible in the less pronounced fluctuations in the metrics. This fact is of paramount importance when performing hyperparameters tuning, as it leads to more stable and reliable results.

2. Characterisation of advanced TTN model

Let us focus now on the advanced TTN structure. During the whole learning procedure, from the preprocessing of the input dataset up to the training itself, many hyperparameters have to be chosen. Each of them has a certain impact on the final performance the TTN classifier. Therefore, it is of particular interest to further study the procedure in this sense, characterising the TTN by varying some of its crucial features:

- **bond dimension χ**, which establishes the complexity of the model;
- **feature map**, including its type and the order d, offering the possibility to enhance the capability of the model to recognise specific non-linear combinations of the input features;
- **batch size**, which can enhance the parallelisation of the computational task.

a. Bond dimension χ Let us begin from the most important parameter, namely the bond dimension χ. The ladder plays a pivotal role in the definition of the architecture since the level of complexity of the model is proportional to its value. In particular a higher bond dimension will result in a higher number of parameters, which may enhance the performance of the network. However, the bond dimension has to be selected carefully since using a too large value enlightens his drawbacks:

- the more parameters the model has, the more computationally intensive training and evaluation are;
- if the model is too complex for input dataset, it may perform worse than a simpler one due to overfitting.

For instance, the number of parameters of the model depending on χ is showed in TAB. I.

Bond dimension	Model parameters	Bond dimension	Model parameters	Bond dimension	Model parameters	Bond dimension	Model parameters
5	$\approx 2.73 \cdot 10^3$	30	$\approx 3.84 \cdot 10^5$	55	$\approx 2.34 \cdot 10^6$	80	$\approx 7.18 \cdot 10^6$
10	$\approx 1.60 \cdot 10^4$	35	$\approx 6.08 \cdot 10^5$	60	$\approx 3.03 \cdot 10^6$	85	$\approx 8.62 \cdot 10^6$
15	$\approx 5.03 \cdot 10^4$	40	$\approx 9.05 \cdot 10^5$	65	$\approx 3.86 \cdot 10^6$	90	$\approx 1.02 \cdot 10^6$
20	$\approx 1.16 \cdot 10^5$	45	$\approx 1.28 \cdot 10^6$	70	$\approx 4.82 \cdot 10^6$	95	$\approx 1.20 \cdot 10^7$
25	$\approx 2.24 \cdot 10^5$	50	$\approx 1.76 \cdot 10^6$	75	$\approx 5.92 \cdot 10^6$	100	$\approx 1.40 \cdot 10^7$

TAB. I Number of parameters of the TTN depending on χ, namely the bond dimension of the tensor nodes.

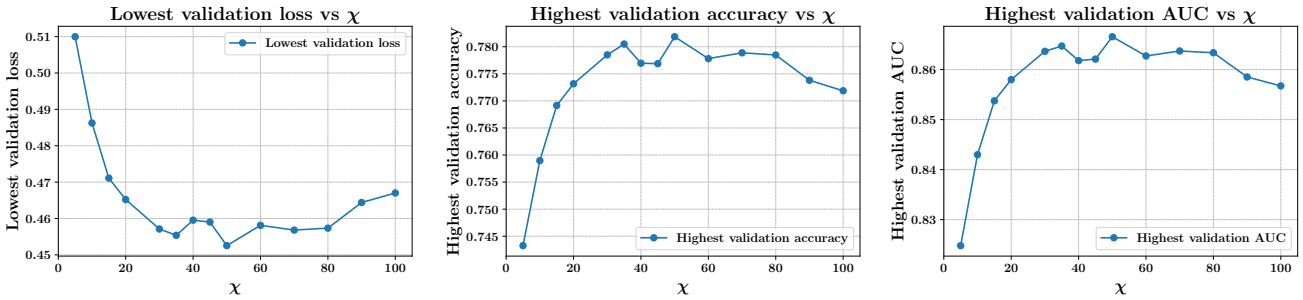
In order to study the behaviour of the model depending on χ, a series of models with different values of χ is trained. The best results over training epochs for the metrics of loss, accuracy and AUC are showed in FIG. 8. Concerning the time performances and scaling with χ, these are visualised in FIG. 9.

From the plots of the metrics, it is possible to observe how the performances of the model increase with χ up to a value of about $\chi \approx 50$, after which the validation accuracy and AUC start to decrease. This behaviour indicates that after that value the model complexity is too high for the input dataset dimension and so it shows overfitting symptoms.

Concerning the timing plots, the training time per epoch for 10^7 input samples is showed in FIG. 9a, while the prediction time for $5 \cdot 10^5$ input samples is showed in FIG. 9b. As expected, the training time per epoch increases with χ. The data points are interpolated using a power law distribution function, namely:

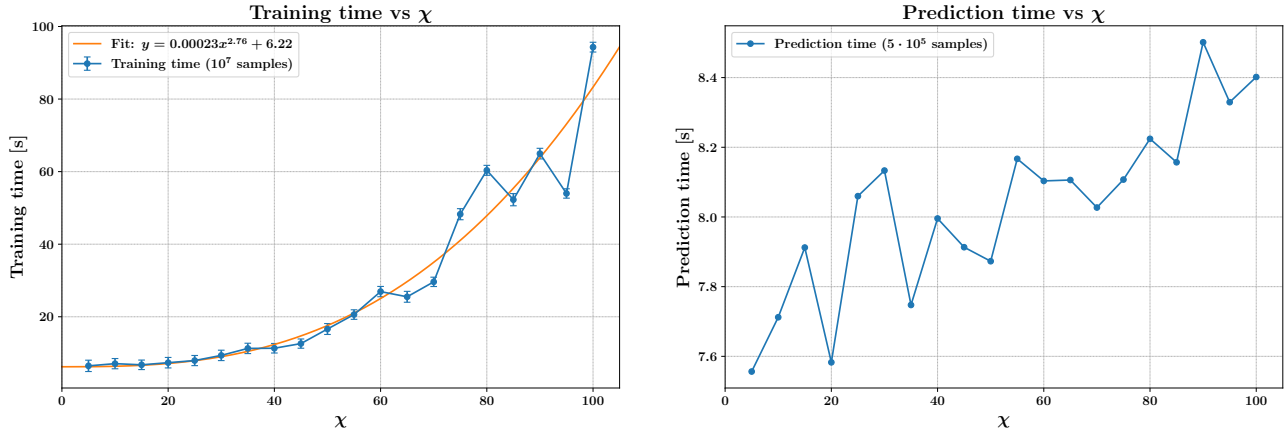
$$y = ax^k + c \quad , \tag{20}$$

obtaining as exponent parameter a value of $k = 2.76$. From a theoretical point of view, the expected value is $k = 3$, since we are dealing with rank-3 tensors and in the middle layers the tensor nodes have dimension equal to χ^3. On the other hand, the TTN model is trained on a GPU, so the discrepancy can be due to the parallelisation of the computations on the hardware. Concerning the prediction time for a fixed number of input samples, visualised in



(a) Model lowest loss value (b) Model highest accuracy score (c) Model highest AUC score

FIG. 8 Performance of TTN model depending on the bond dimension. The results are obtained from the validation set, on which the best values over training epochs for the metrics are computed. In particular, in **8a** the lowest loss, in **8b** the highest accuracy and in **8c** the highest AUC, depending on the bond dimension χ.



(a) Model training time per epoch for 10^7 samples (b) Model prediction time for $5 \cdot 10^5$ samples

FIG. 9 Timing analysis and scaling of TTN classifier depending on the bond dimension χ. In FIG. **9a** the training time per epoch for 10^7 samples is showed, while in FIG. **9b** the prediction time for $5 \cdot 10^5$ samples is visualised.

FIG. **9b**, it exhibits a linear trend with higher fluctuations. The ladders can be explained again by the fact that the computations are performed on a GPU, so the overhead time needed for data transfer on the hardware, which is not constant and depends on the system status, has also to be taken into account.

b. Feature map and map order We focus now on the characterisation of the TTN model depending on the feature map applied in the preprocessing and on its order d. The results for this phase are obtained using $\chi = 35$, which is the second best result for the bond dimension analysis. We explicitly do not choose the best one with $\chi = 50$ due to its computational effort and to time limitations. Moreover, the differences in performances between them are not too significant.

Similarly to what already done for the bond dimension analysis, for the map characterisation the same TTN structure is trained for both spherical and polynomial feature maps and for an order $d \in \{2, 3, 4, 5\}$.

The results for this study are presented in TAB. **II**. As it is possible to see, the TTN classifier performances are not significantly affected by the choice of the feature map and by its order. In fact, it is important to remark that the order of the feature map affects only the dimensions and number of parameters in the input layer. This explains the small rise in the training and prediction time for a map of higher order. On the other hand, in terms of absolute performances, it is possible to observe a slight improvement with a rising map order, in particular with a spherical map of order $d = 5$.

c. Batch size The last characterisation study involves the hyperparameter of batch size, namely the number of input samples to process before a weight update by the optimiser. This quantity has a crucial role in time needed

	Spherical Map				Polynomial Map			
Map Order *d*	2	3	4	5	2	3	4	5
Min loss	0.532	0.523	0.512	0.511	0.526	0.519	0.512	0.514
Max accuracy	0.777	0.781	0.781	0.782	0.781	0.781	0.781	0.781
Max AUC	0.862	0.866	0.866	0.867	0.866	0.866	0.866	0.865
Training time [s]	120	127	134	140	125	133	141	150
Prediction time [s]	8.65	8.80	8.91	9.13	8.61	8.80	8.93	9.30

TAB. II Results of the feature map and map order characterisation analysis. The training time is obtained for 10^7 input samples, while the prediction time for $5 \cdot 10^5$ input samples.

for the TTN training, as already explained. In fact, a bigger batch size allows to process more samples in parallel, which results in a significant speed enhancement, especially when the training is performed on a GPU.

Passing to the results, in FIG. 10 the dependency of the training time on the batch size is visualised. As it is possible to see, smaller values of the batch size translate into very long training times, making the training procedure extremely inefficient and infeasible. Therefore, an optimal choice would be a batch size that can be handled in a single parallel run to maximise the efficiency. In particular, the hardware employed for this work allows a batch size of $\sim 10^4$. For this reason, all the tests with this parameter fixed are done with a minibatch of 10^4 samples.

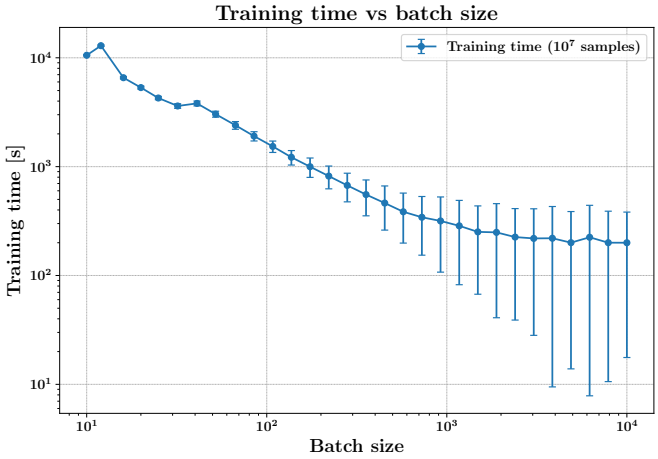


FIG. 10 TTN classifier training time per epoch for 10^7 input samples depending on the batch size.

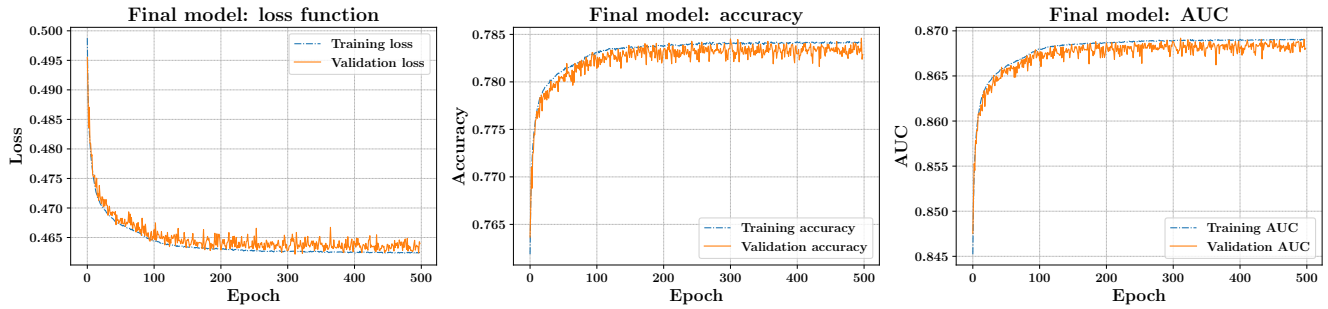
3. Final model results

After having characterised some of the main hyperparameters of the TTN classifier, we combine all the information from the previous analyses to produce the final structure treated. The ladder is trained for a higher number of epochs in order to extract the most promising results of this work. For the sake of completeness, the main parameters used for this last run are given in TAB. III. On the other hand, the results obtained for the metrics on training and validation sets are visualised in FIG. 11.

At the end of the training procedure the best model is saved in a `hdf5` file format in order retrieve the model with the best validation AUC over the training epochs. In particular, this is possible through a series of "callbacks" defined inside TensorFlow framework, so that the model saved is not the one at the last epoch, but the most performing one. In order to quantify the goodness of this model, we extract from it its ROC curve and "confusion matrix" calculated on a test set of $5 \cdot 10^5$ samples. Then, the ladders are visualised in FIG. 12a and FIG. 12b, respectively.

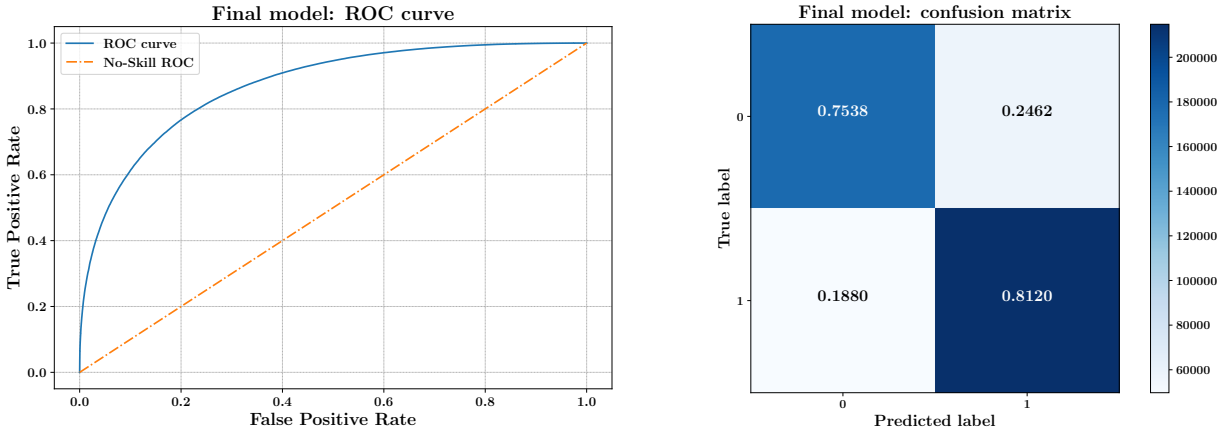
Feature Map		**Model architecture**	
Type	`Spherical`	χ	50
Order	5	Activation	`Elu`
Training Parameters		**Number of samples**	
Batch size	10^4	Train set	10^7
Epochs	500	Validation set	$5 \cdot 10^5$
Optimizer	`Adam`	Test set	$5 \cdot 10^5$

TAB. III Hyperparameters for final TTN model, from which the best results obtained in this work are extracted.



(a) Loss over training epochs (b) Accuracy over training epochs (c) AUC over training epochs

FIG. 11 Final TTN model metrics computed on training and validation sets over training epochs, with the loss in **11a**, the accuracy in **11b** and the AUC in **11c**.



(a) Final model ROC curve (b) Final model confusion matrix

FIG. 12 Final TTN classifier predictions on test set. In **12a**, the ROC curve of the model prediction is showed, in **12b** the confusion matrix of the predicted classes is visualised. In particular, in the ladder the colormap represents the number of samples belonging to a certain prediction class.

From this analysis we can extrapolate the final test accuracy and AUC:

$$\begin{aligned} \text{Accuracy}_{\text{final}} &= 78.46\% \\ \text{AUC}_{\text{final}} &= 0.8694 \end{aligned} \quad . \tag{21}$$

The ladder is below the best result obtained by [1], namely an AUC value of 0.885. However, the model trained in [1] derives from a different approach and from a larger number of training epochs. Moreover, several improvements on the TTN best model are possible, such as a finer tuning of other hyperparameters and higher training epochs, but they go beyond the purpose of this work. These possibilities will be explored and tested for future work.

V. CONCLUSIONS

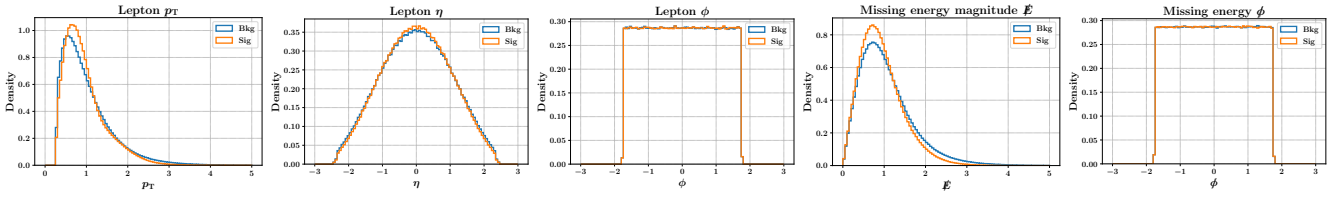
a. Summary In this work we have faced the problem of a Tree Tensor Network classifier implementation and we have validated it with a High Energy Physics dataset, namely the "HIGGS" dataset of [1]. Moreover, we have showed all the preliminary operations for this purpose, like data preprocessing and the implementation of a learning algorithm inside the frameworks of TensorFlow and TensorNetwork. After this part, the results have been presented, showing how, with several optimisations, the TTN model is capable of discriminating signal from background with good performances. In particular, after the tuning of crucial TTN hyperparameters like the bond dimension, the best results of this work are obtained with a longer training procedure, getting values of the metrics employed not so far from the best result of [1]. Last but not least, we have remarked the difference of our approach with [1] and how there is still room for improvement.

b. Technical issues It is necessary to point out the issues encountered during the build up of this work. In particular, the time limitation imposed by the long training procedures, needed to obtain significant results, and by the hardware specifics are among the main problems faced. Moreover, the creation of a custom layer inside TensorFlow is technically difficult if one wants to get state-of-the-art performances. In fact, this operation requires the employment of many optimisation paradigms and advanced functions. However, there are also good reasons to still choose it, such as the facility to run the learning and evaluation algorithms on GPU or CPU with intrinsic parallelisation. This translates into a huge speed up in timing performances for this type of task.

c. Possible improvements The approach presented in this work has proved to be effective and well performaning. However, as discussed before, there is still room for improvement, since the architecture can be further tuned. In this sense, it would be interesting to analyse the performances derived from the use of other optimisers or new regularisation techniques, so that the problem of overfitting can be avoided. An analysis of this type requires long technical times for the optimisation and also a more advanced hardware on which the training can be run, such as more performing GPUs or also the newer Application Specific Integrated Circuits (ASIC). These would carry another significant boost in training speed and so they would allow to explore many more possibilities and the results of new architectures or tensor layers.

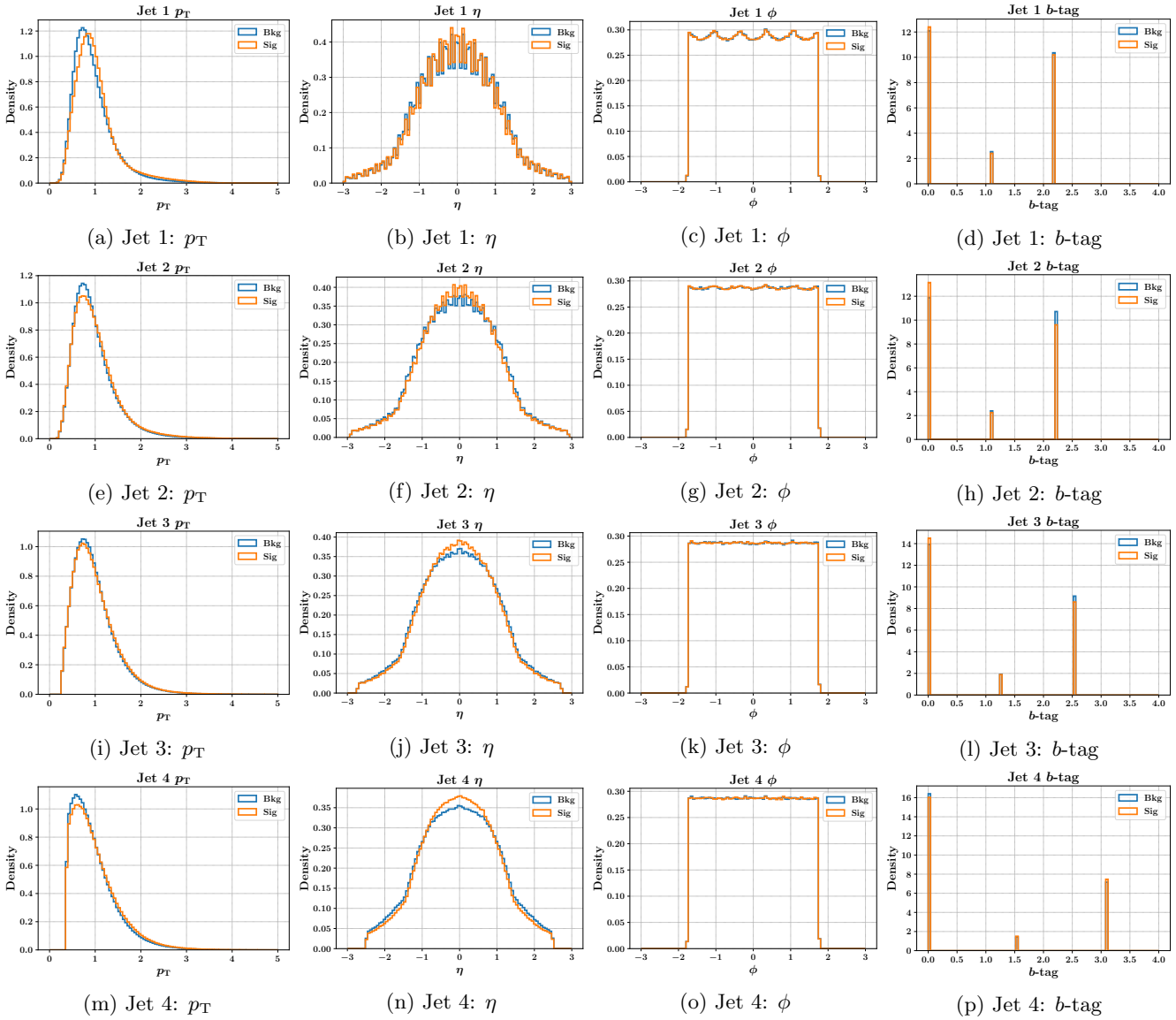
A. APPENDIX

1. Low-level features distributions



(a) Lepton p_T (b) Lepton η (c) Lepton ϕ (d) Missing energy $\not{E}$ (e) Missing energy ϕ

FIG. 13 Distributions of low-level features of leptonic part for background (blue) and signal (orange) events. The physical units of measure are omitted due to the fact that the dataset is not available in non-standardised form.



(a) Jet 1: p_T (b) Jet 1: η (c) Jet 1: ϕ (d) Jet 1: b-tag

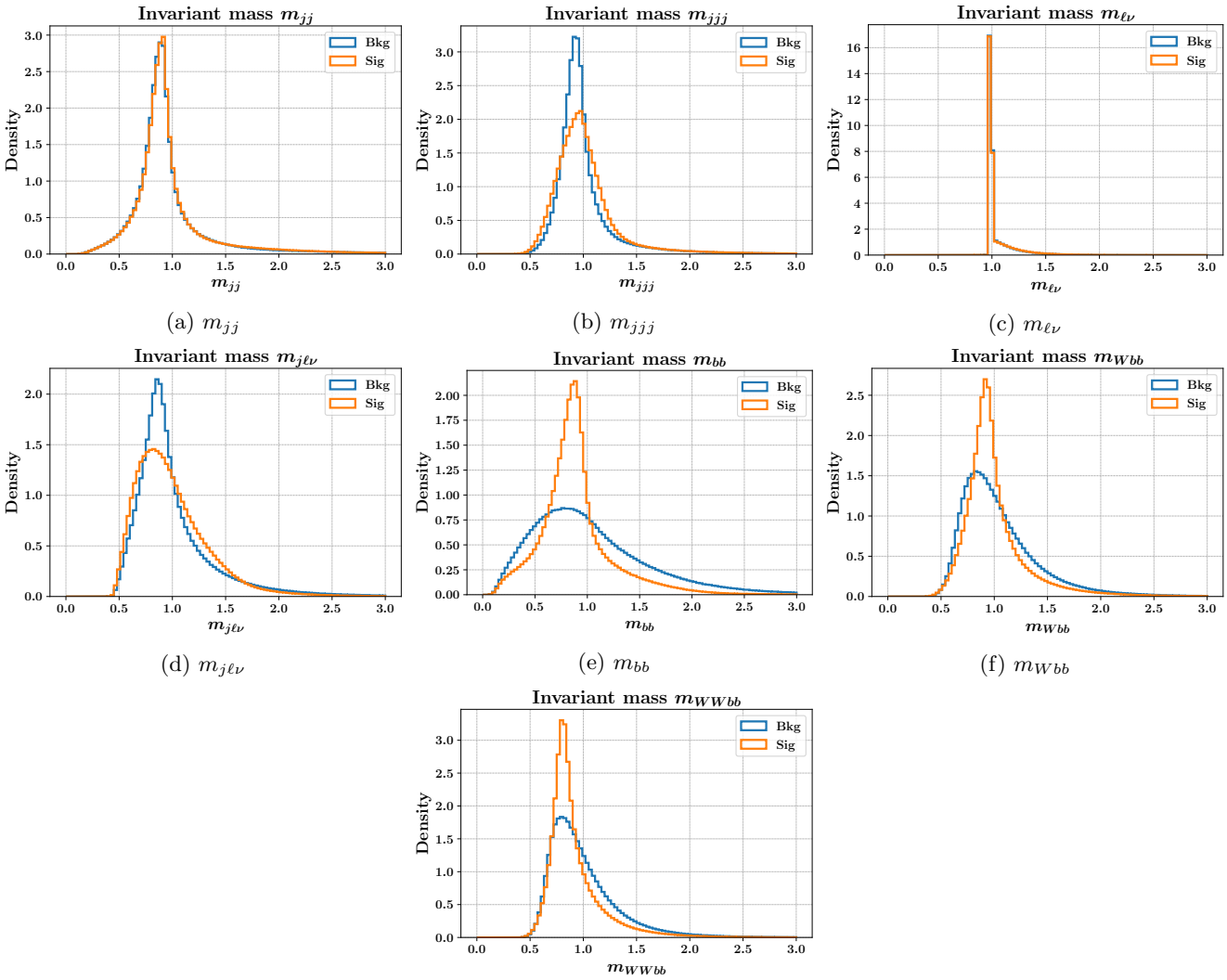
(e) Jet 2: p_T (f) Jet 2: η (g) Jet 2: ϕ (h) Jet 2: b-tag

(i) Jet 3: p_T (j) Jet 3: η (k) Jet 3: ϕ (l) Jet 3: b-tag

(m) Jet 4: p_T (n) Jet 4: η (o) Jet 4: ϕ (p) Jet 4: b-tag

FIG. 14 Distributions of low-level features of hadronic part for background (blue) and signal (orange) events. The physical units of measure are omitted due to the fact that the dataset is not available in non-standardised form.

2. High-level features distributions



(a) m_{jj} (b) m_{jjj} (c) $m_{\ell\nu}$

(d) $m_{j\ell\nu}$ (e) m_{bb} (f) m_{Wbb}

(g) m_{WWbb}

FIG. 15 Distributions of high-level features for background (blue) and signal (orange) events. The physical units of measure are omitted due to the fact that the dataset is not available in non-standardised form.

3. TTN constructors

```
# first layer
tn_model.add(
    TTN_SingleNode(
        bond_dim      = bond_dim,
        n_contraction = n_contr
        use_bias      = True,
        activation    = activation,
        input_shape   = input_shape,
    )
)

# intermediate layers
for _ in range(n_layers-2):
    tn_model.add(
        TTN_SingleNode(
            bond_dim      = bond_dim,
            n_contraction = n_contr
            use_bias      = True,
            activation    = activation
        )
    )

# last layer
tn_model.add(
    TTN_SingleNode(
        bond_dim      = 1,
        n_contraction = n_contr
        use_bias      = True,
        activation    = 'sigmoid'
    )
)
```

LST. 10 Core implementation of a "pure" TTN structure constructor without the employment of advanced techniques, such as batch normalisation and regularisation.

```
# first layer
tn_model.add(
    TTN_SingleNode(
        bond_dim           = bond_dim,
        n_contraction      = n_contr,
        use_bias           = use_bias,
        kernel_regularizer = kernel_regulariser
        input_shape        = input_shape,
    )
)
tn_model.add(
    BatchNormalization(
        epsilon  = 1e-06,
        momentum = 0.9,
        weights  = None
    )
)
if activation is not None:
    tn_model.add(Activation(activation))

```

```
# intermediate layers
for _ in range(n_layers-2):
    tn_model.add(
        TTN_SingleNode(
            bond_dim           = bond_dim,
            n_contraction      = n_contr,
            use_bias           = use_bias,
            kernel_regularizer = kernel_regulariser
        )
    )
    tn_model.add(
        BatchNormalization(
            epsilon  = 1e-06,
            momentum = 0.9,
            weights  = None
        )
    )
    if activation is not None:
        tn_model.add(Activation(activation))

# last layer with bond dimension 1
tn_model.add(
    TTN_SingleNode(
        bond_dim           = 1,
        use_bias           = True,
        n_contraction      = n_contr,
        input_shape        = input_shape,
        kernel_regularizer = kernel_regulariser
    )
)
tn_model.add(
    BatchNormalization(
        epsilon  = 1e-06,
        momentum = 0.9,
        weights  = None
    )
)
tn_model.add(Activation('sigmoid'))
```

LST. 11 Core implementation of a more sophisticated TTN structure constructor with the employment of advanced techniques, such as batch normalisation and regularisation.

REFERENCES

[1] P. Baldi, P. Sadowski, and D. Whiteson, *Searching for Exotic Particles in High-Energy Physics with Deep Learning*, https://arxiv.org/pdf/1402.4735.pdf

[2] M. Trenti et al, *Quantum-inspired Machine Learning on high-energy physics data*, https://arxiv.org/pdf/2004.13747.pdf

[3] E. Miles Stoudenmire and David J. Schwab, *Supervised Learning With Quantum-Inspired Tensor Networks*, https://arxiv.org/abs/1605.05775

[4] Machine Learning From Scratch, *Activation Functions Explained - GELU, SELU, ELU, ReLU and more*, https://mlfromscratch.com/activation-functions-explained/#/

[5] Ian Goodfellow and Yoshua Bengio and Aaron Courville, *Deep Learning Book*, http://www.deeplearningbook.org

[6] Tensor Network, *Tensor Diagram Notation*, https://tensornetwork.org/diagrams/

[7] TensorFlow, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, url=http://tensorflow.org/

[8] TensorNetwork, *TensorNetwork: A Library for Physics and Machine Learning*, https://arxiv.org/pdf/1905.01330.pdf

[9] CloudVeneto, *CloudVeneto.it User Guide*, http://userguide.cloudveneto.it/en/latest/