UNIVERSITÀ DEGLI STUDI DI PADOVA

Tree Tensor Network supervised classifier for High Energy Physics

Rocco Ardino
Mat: 1231629
rocco.ardino@studenti.unipd.it

Alessandro Valente
Mat: 1234429
alessandro.valente.4@studenti.unipd.it

Quantum Information and Computing
a.y. 2020/21

March 19, 2021

Outline

High Energy Physics (HEP):

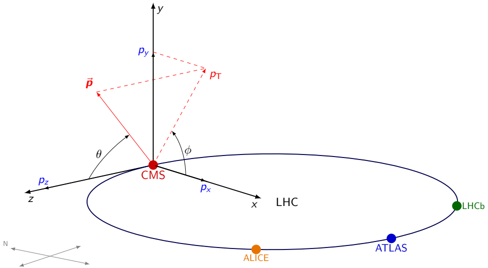
- purpose of understanding the nature of the particles that constitute matter and radiation
- Standard Model as theoretical climax
- still lots of unanswered questions
- $\Rightarrow$ still New Physics to discover

Application of Machine Learning techniques in HEP:

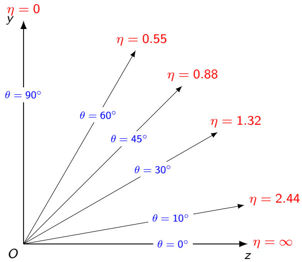
- current techniques used in HEP fail to capture all the available information
- as proved by Baldi et al ([1]), Machine Learning can overcome this issue
- many other advantages, such as solution to the curse of high dimensionality of data
- e.g., Artificial Neural Networks exploited to build powerful classifiers

New approach with **Tree Tensor Networks (TTN)**:

- quantum-inspired version of Biological Neural Networks
- structure based on Tensor Network methods
- possibility to solve non-convex optimisation tasks, such as loss functions minimisation



(a) Kinematics of a collision



(b) Pseudorapidity visualisation

Figure: 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.

Product particles of *pp* collisions

- Leptons ($\ell^{\pm}$, $\overset{(-)}{\nu_\ell}$)
- Bosons (h^0, Z^0, $W^{\pm}$)
- Quarks (q) and Gluons (g)
- Jets from quark/gluon hadronisation (j)

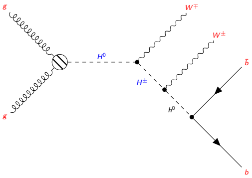
Main features for the analysis

- Transverse momentum p_T
- Pseudorapidity η
- Azimuth angle ϕ
- b-tag (for jets)

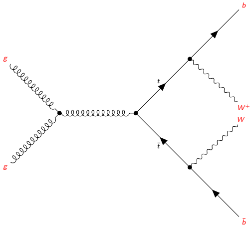
HIGGS dataset

Monte Carlo generated sample of gluon fusion process:

- a mass of the Higgs $m_{h^0} = 125$ GeV is assumed
- new exotic Higgs bosons H^0 and $H^{\pm}$ are introduced ($m_{H^0} = 425$ GeV, $m_{H^{\pm}} = 325$ GeV)
- 21 low level features (such as p_T, η, ϕ, b-tags, ...)
- 7 high level features (invariant masses distributions)



(a) Signal process



(b) Background process

Figure: 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.

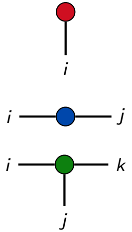
$$\textbf{Signal:} \quad gg \to H^0 \to W^{\mp}H^{\pm} \to W^{\mp}W^{\pm}h^0 \to W^{\mp}W^{\pm}b\bar{b} \tag{1}$$

$$\textbf{Background:} \quad gg \to g \to t\bar{t} \to W^{\mp}W^{\pm}b\bar{b} \tag{2}$$

Tensor Networks

Tensor Networks (TNs) as core of TTNs:

- factorisations of high rank tensors into networks of smaller rank tensors
- intuitive graphical language
  - tensors are notated by solid shapes, tensor indices are notated by lines emanating from these shapes
  - connecting two index lines implies a contraction, or summation over the connected indices

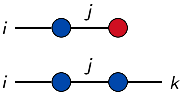


$$= v_i \longrightarrow \textbf{Vector}$$

$$= M_{ij} \longrightarrow \textbf{Matrix} \tag{3}$$

$$= T_{ijk} \longrightarrow \textbf{Rank-3 Tensor}$$

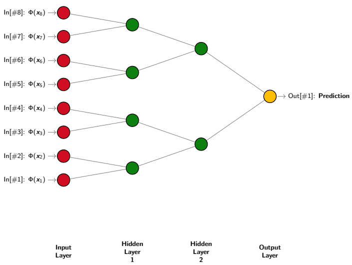
  - graphical notation for some of the most common operations between tensors



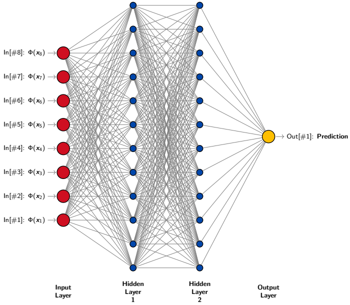
$$\begin{aligned} &= \sum_j M_{ij} v_j \longrightarrow \textbf{Vector-Matrix} \\ &= \sum_j A_{ij} B_{jk} \longrightarrow \textbf{Matrix-Matrix} \end{aligned} \tag{4}$$

TNs for Machine Learning in a nutshell:

- input features x, mapped into $\Phi(x)$ through an apposite feature map
- TN with a tree-like structure $T(w;\chi)$, with:
  - $\boldsymbol{w}$ **the entries of the tensors**, namely the weights to be tuned by the learning algorithm
  - χ **the bond dimension**, which controls the complexity of the structure
- decision function: $f(x;w) = T(w;\chi) \cdot \Phi(x)$



(a) Example of TTN structure



(b) Example of DNN structure

Figure: 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.

TTN Learning algorithm

Loss function:

- need to minimise the ladder on a fraction of dataset reserved for training the TTN
- quantification of TTN misclassification on new fraction of dataset reserved for validation
- binary cross-entropy possible loss for signal-versus-background discrimination with:
  - $y^{(i)}_{\text{true}}$: true label of i^{th} sample of validation set
  - $y^{(i)}_{\text{pred}}$: TTN predicted label for the i^{th} sample of validation set

$$L(y_{\text{true}}, y_{\text{pred}}) = \sum_{i}^{n_{\text{val}}} y^{(i)}_{\text{true}} \log\left(y^{(i)}_{\text{pred}}\right) + (1 - y^{(i)}_{\text{true}}) \log\left(1 - y^{(i)}_{\text{pred}}\right) \tag{5}$$

Metrics for performances quantification:

- accuracy: fraction of correctly classified samples (using a decision threshold $\xi \in (0, 1)$)
- AUC: area under the Receiver Operating Characteristic (ROC) Curve
  - True Positive Rate (TPR) in function of the False Positive Rate (FPR)
  - obtained by sweeping the decision threshold ξ in $(0, 1)$
  - directly linked to concept of discovery significance commonly used in HEP

Optimiser for weights update:

- ADAM: adapts the correction to the weights at each step parameter per parameter
- update after the TTN has processed a mini batch of input training set of dimension m
- after processing all mini batches in the training set, a training epoch has ended

Activation function:

- function applied at the output of a layer
- source of non-linearity to enlarge the space of functions that the TTN can approximate
- for this work, ELU (for inner layers) and sigmoid (for output layer) tested:

$$\mathrm{ELU}(x) = \begin{cases} x & x \geq 0 \\ a(e^x - 1) & x < 0 \end{cases} \qquad \sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

Kernel regularisation:

- add a penalty for large weights to the loss function to avoid overfit
- ℓ_2 regularisation:

$$J(y_{\text{true}}, y_{\text{pred}}) = L(y_{\text{true}}, y_{\text{pred}}) + \lambda \sum_{i=1}^{n_{\text{weights}}} |w_i|^2 \tag{7}$$

Batch normalisation:

- training networks with lots of layer is very challenging
- calculate mean and standard deviation of the output of each layer for every mini batch
- $\Rightarrow$ perform a standardisation
- $\Rightarrow$ speed up of learning algorithm convergence

Input features preprocessing workflow:

- rescaled in $[0, 1]$ or $[-1, 1]$ (depending on the feature)
- padded to match the number of input "legs" of the TTN
- mapped through a feature map to enhance the performances

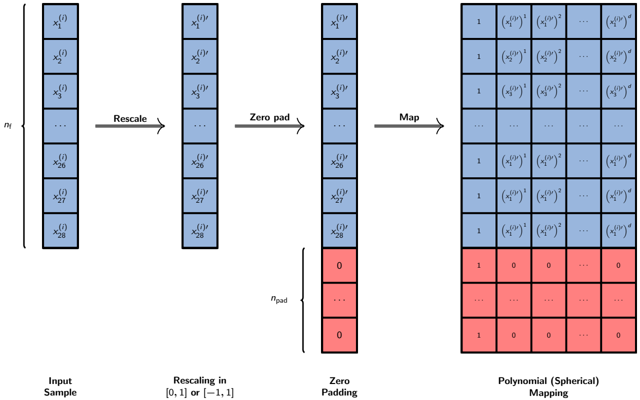


Figure: Workflow of preprocessing precedure, starting from the rescaling of data, going through the zero padding in order to get proper dimensions for the input of the TTN, and lastly the polynomial or spherical mapping.

Università degli Studi di Padova

Rescaling

- rescale "positive" and "negative" features in $[0, 1]$ and $[-1, 1]$, respectively

$$x_j^{(i)} \longrightarrow x_j^{(i)\prime} = \frac{x_j^{(i)}}{m_j} \quad \text{with} \quad m_j = \max_{i \in \mathcal{D}_{\texttt{train}}} \left| x_j^{(i)} \right| \tag{8}$$

Padding

- total number of features n_f must be divisible by features contracted in each site n_con
- dataset is "padded" with fictitious features to reach $\tilde{n}_\mathrm{f}$ features

$$\tilde{n}_\mathrm{f} = \min_{n \in \mathbb{N}} \{ n \geq n_\mathrm{f} \ : \ n = (n_\mathrm{con})^m \ , \ m \in \mathbb{N} \} \tag{9}$$

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

# Padding
def PadToOrder(x, con_order):
    # compute 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
```

Input data preprocessing: feature map

Feature map:

- applying a map to the input features enhances the classifier performances
- polynomial map, common in standard ML classification tasks

$$\Phi_d^{\text{pol}}(x) = \left[1, x, \ldots, x^d\right] \tag{10}$$

- spherical map, quantum inspired, at order 2 maps features to spins:

$$\Phi_d^{\text{sph}}(x) = \left[\phi_d^{(1)}(x), \ldots, \phi_d^{(d)}(x)\right] \tag{11}$$

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

```python
# Spherical map
def SphericalMap(x, order=2, dtype=np.float32):
    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

# Polynomial map
def PolynomialMap(x, order=2, dtype=np.float32):
    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
```

Framework and workflow:

- TTN classifier is a series of layers built using TensorFlow and TensorNetwork libraries
- possibility of running on both CPU and GPU through Keras API
- layer object main input parameters:
  - **n_contraction**: number of features to contract in each node site, namely 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_batch_norm**: introduce the batch normalisation of the layers if true is specified

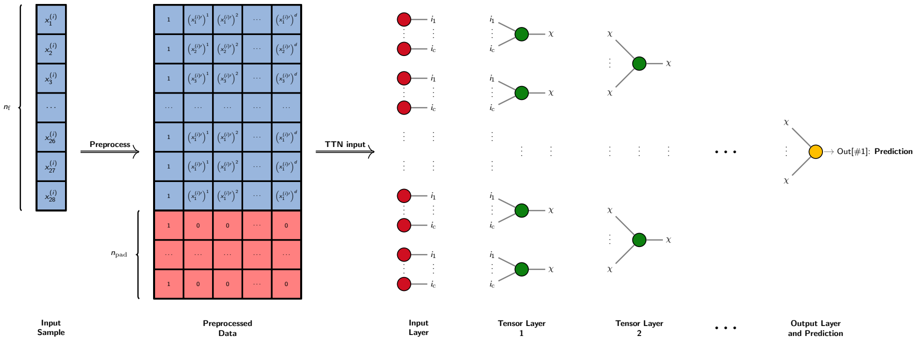


Figure: Schematic representation of workflow of the data preprocessing and of the TTN structure, with $i_c = i_{n_{\text{con}}}$.

Node creation and contraction:

- the computations inside a TTN layer are divided in three logical steps:
  - **nodes creations**: input and weight tensors are initialised as TensorNetwork nodes
  - **edge connection**: edges of each tensor node are connected to the corresponding input node
  - **edge contraction**: node contractions along connected edges are executed
- In each step the same loop structure is used but separating allows better GPU parallelisation

```
# tensor nodes initialisation
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"))
```

```
# tensor nodes edges connection
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]
```

```
# tensor nodes edges contraction
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]])
    )
```

TTN framework: model building

Model structure:

- using the TTN layers, a classifier can be created using the Sequential Keras API
- when instantiating the layers, it is possible to specify many parameters, such as:
  - **input_shape**: shape of input samples
  - **n_contraction**: number of features to contract at each weight node
  - **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**: introduce bias weights if true is specified
  - **use_batch_norm**: introduce the batch normalisation of the layers if true is specified
  - **kernel_regulariser**: TensorFlow regulariser object to introduce the regularisation

```
# create keras sequential model
tn_model = Sequential()

# first layer, input shape must be specified
tn_model.add( TTN_SingleNode(bond_dim=10, activation='elu',     n_contraction=2,
                             input_shape=(x_train.shape[1:])                    ) )

# intemediate layers, input shape computed from previous layers output
tn_model.add( TTN_SingleNode(bond_dim=10, activation='elu',     n_contraction=2 ) )
tn_model.add( TTN_SingleNode(bond_dim=10, activation='elu',     n_contraction=2 ) )
tn_model.add( TTN_SingleNode(bond_dim=10, activation='elu',     n_contraction=2 ) )

# last layer, bond dim 1 and sigmoid function to interpret output as probability
tn_model.add( TTN_SingleNode(bond_dim=1,  activation='sigmoid', n_contraction=2 ) )
```

TTN framework: model training

Model compilation:

- after layers instantiation, the model is compiled inside TensorFlow framework
- optimiser, loss function and metrics are specified at compilation time
  - ADAM optimiser
  - binary cross entropy loss
  - accuracy and AUC metrics

Model training:

- TTN classifier is trained using TensorFlow framework
- training and validation sets are provided to the training function, alongside with training parameters:
  - number of epochs of training
  - batch size, namely after how many samples processed the weights should be updated by the optimiser

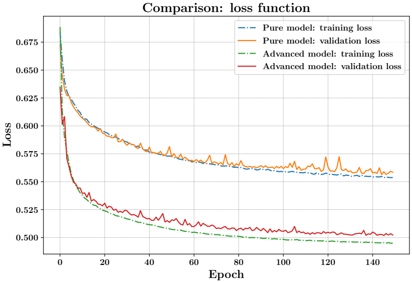
```
# model compilation
tn_model.compile(
    optimizer = 'adam',                   # optimizer for training
    loss      = 'binary_crossentropy',    # loss function to minimize
    metrics   = ['accuracy', 'AUC']       # metrics to monitor
)

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

“Pure” TTN model

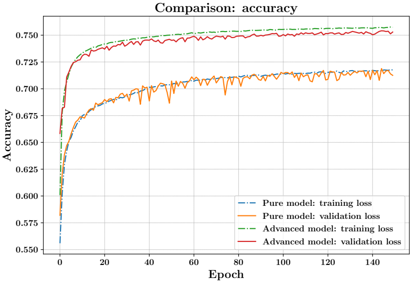
- Standard TTN structure
  - no advanced techniques
  - no batch normalisation, no regularisation
  - no activation for inner layers
  - only sigmoid activation for final prediction

Advanced TTN model

- ML optimisations added:
  - batch normalisation
  - ℓ_2 regularisation
  - ELU activation function for inner layers
  - sigmoid activation for final prediction



(a) Loss metric over training epochs



(b) Accuracy metric over training epochs

Figure: 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 **6a** and **6b**, respectively.

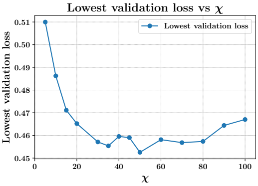
Model Characterisation: parameters and metrics

Bond dimension χ:

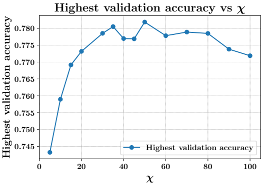
- determines the number of parameters and consequently the model complexity

χ	Parameters	χ	Parameters	χ	Parameters	χ	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	$\approx 1.40 \cdot 10^7$

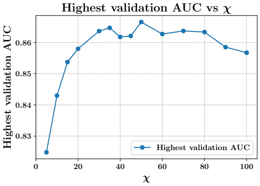
Table: Number of parameters of the TTN depending on χ, namely the bond dimension of the tensor nodes.



(a) Model lowest loss value



(b) Model highest accuracy score



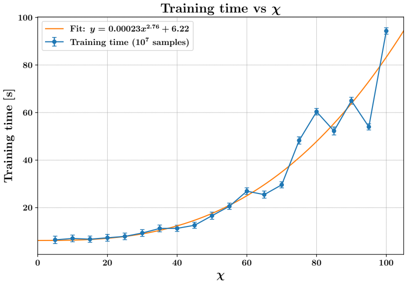
(c) Model highest AUC score

Figure: Performances 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 **7a** the lowest loss, in **7b** the highest accuracy and in **7c** the highest AUC, depending on the bond dimension χ.

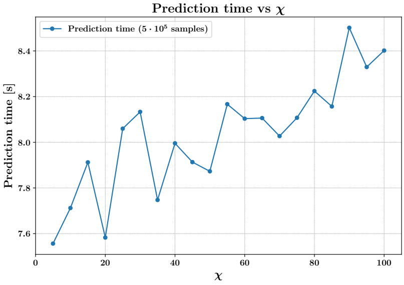
Model Characterisation: time scaling with χ

Time scaling with χ:

- number of parameters grows with χ as $O(\chi^3)$
- training and prediction time will be higher for more complex models
  - training time follows a power law with exponent $k = 2.76$
  - expectation for the latter is $k_{\text{th}} = 3$ (for rank-3 tensor-vector multiplication)
  - prediction time grows linearly (with a limited number of samples)



(a) Model training time per epoch for 10^7 samples



(b) Model prediction time for $5 \cdot 10^5$ samples

Figure: Timing analysis and scaling of TTN classifier depending on the bond dimension χ. In **8a** the training time per epoch for 10^7 samples is showed, while in **8b** the prediction time for $5 \cdot 10^5$ samples is visualised.

Model Characterisation: feature map

Feature map and order:

- performance dependence on the applied map during data preprocessing and on map order
- polynomial map, common in standard ML classification tasks

$$\Phi_d^{\text{pol}}(x) = \left[1, x, \ldots, x^d\right] \tag{12}$$

- spherical map, quantum inspired, at order 2 maps features to spins:

$$\Phi_d^{\text{sph}}(x) = \left[\phi_d^{(1)}(x), \ldots, \phi_d^{(d)}(x)\right] \tag{13}$$

- map orders $d \in \{2, 3, 4, 5\}$ tested

	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

Table: 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.

Model Characterisation: time scaling with batch size

Time scaling with batch size *m*:

- *m* is the number of samples processed before weight update by optimiser
- strongly influences training time
- a higher batch size needs more training epochs to reach same performances
- bigger batch sizes more efficient with GPU acceleration

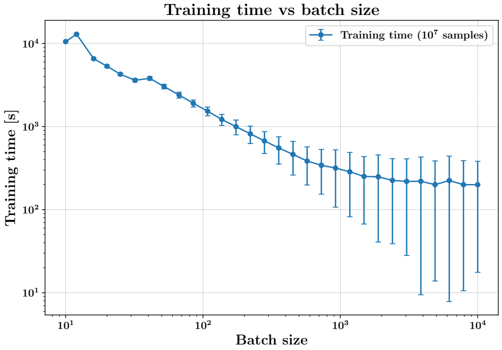


Figure: TTN classifier training time per epoch for 10^7 input samples depending on the batch size.

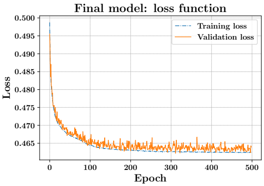
Università degli Studi di Padova

Best model creation:

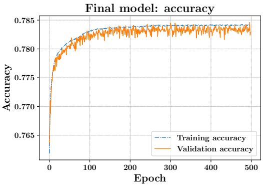
- optimal configuration (from characterisation) trained and tested using the full dataset

Feature Map		Architecture	
Type	Spherical	χ	50
Order	5	Activation	Elu
Training		**Dataset size**	
Batch size	10^4	Train	10^7
Epochs	500	Validation	$5 \cdot 10^5$
Optimizer	Adam	Test	$5 \cdot 10^5$

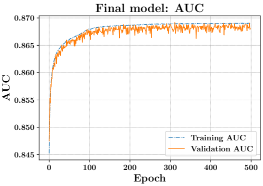
Table: Hyperparameters for final TTN model, from which the best results obtained in this work are extracted.



(a) Loss over training epochs



(b) Accuracy over training epochs



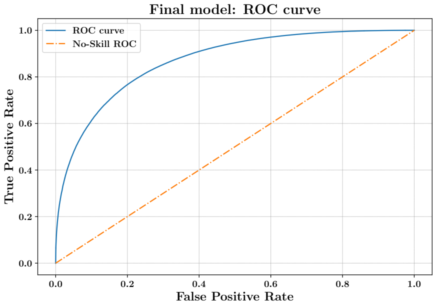
(c) AUC over training epochs

Figure: Final TTN model metrics computed on training and validation sets over training epochs, with the loss in **10a**, the accuracy in **10b** and the AUC in **10c**.

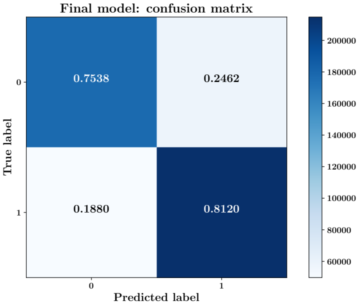
Final results: ROC curve and confusion matrix

Performance evaluation:

- final performance evaluation done on a test set of $5 \cdot 10^5$ samples
- from ROC curve: $\mathrm{AUC}_{\mathrm{final}} = 0.8694$
- from confusion matrix: $\mathrm{Accuracy}_{\mathrm{final}} = 78.46\%$



(a) Final model ROC curve



(b) Final model confusion matrix

Figure: Final TTN classifier predictions on test set. In **11a**, the ROC curve of the model prediction is showed, in **11b** 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.

Conclusions

HIGGS dataset from [1] taken as benchmark:

- common machine learning dataset for HEP classification
- signal-over-background discrimination task

We have seen how to properly preprocess the dataset for the TTN:

- features are rescaled, padded and mapped through a feature map

We have seen how to implement a TTN classifier:

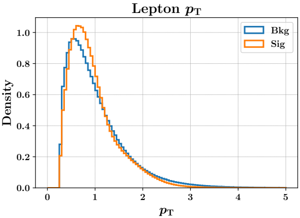
- using TensorFlow and TensorNetwork, a TTN layer has been implemented
- Keras API is used to create a full model starting from the TTN layer
- many advanced techniques also implemented, e.g. batch normalisation, regularisation, ...

Model trained on the full training set:

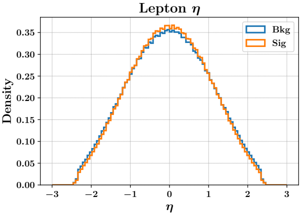
- performance and time scaling studied depending on:
  - bond dimension χ
  - feature map
  - batch size m
- best achieved model:
  - best test accuracy of 78.46%
  - best test AUC of 0.8694

Thank you for the attention!

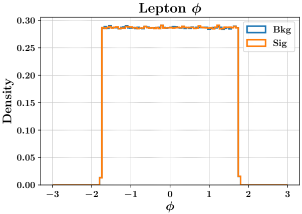
Back-up: low level leptonic features



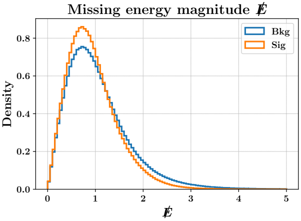
(a) Lepton p_T



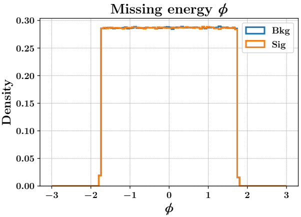
(b) Lepton η



(c) Lepton ϕ



(d) Missing energy $\not{E}$

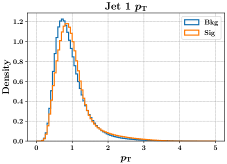


(e) Missing energy ϕ

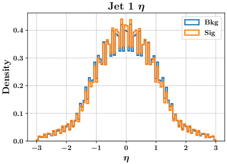
Figure: Distributions of low-level features of leptonic part for background (blue line) and signal (orange line) events. The physical units of measure are omitted due to the fact that the dataset is not available in non-standardised form.

Back-up: low level hadronic features (part 1)

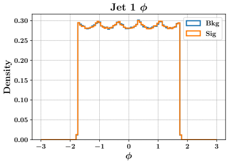
Università degli Studi di Padova



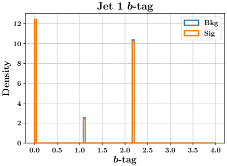
(a) Jet 1: p_T



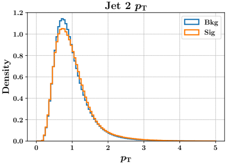
(b) Jet 1: η



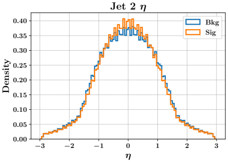
(c) Jet 1: ϕ



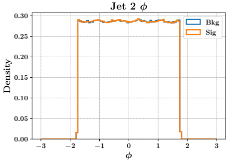
(d) Jet 1: b-tag



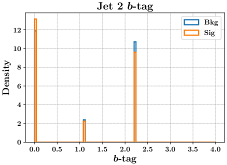
(e) Jet 2: p_T



(f) Jet 2: η



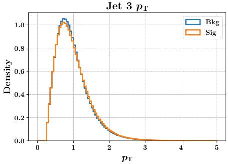
(g) Jet 2: ϕ



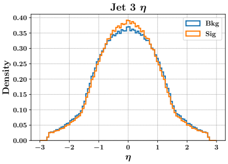
(h) Jet 2: b-tag

Figure: 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.

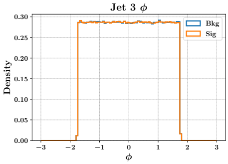
Back-up: low level hadronic features (part 2)



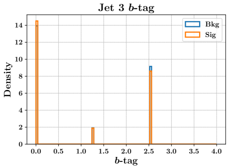
(a) Jet 3: p_T



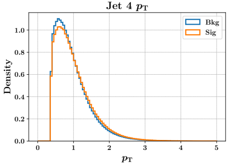
(b) Jet 3: η



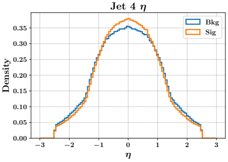
(c) Jet 3: ϕ



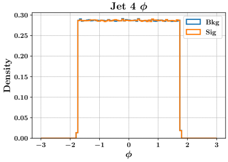
(d) Jet 3: b-tag



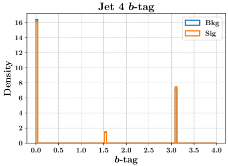
(e) Jet 4: p_T



(f) Jet 4: η



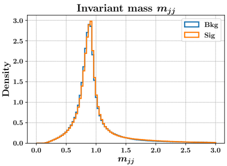
(g) Jet 4: ϕ



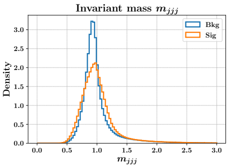
(h) Jet 4: b-tag

Figure: 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.

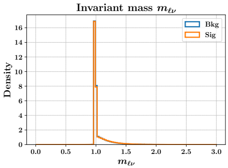
Back-up: high level features



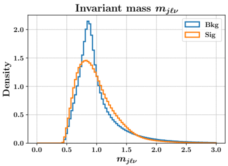
(a) m_{jj}



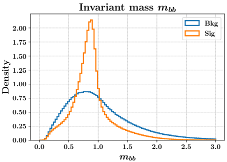
(b) m_{jjj}



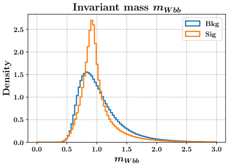
(c) $m_{\ell\nu}$



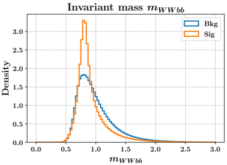
(d) $m_{j\ell\nu}$



(e) m_{bb}



(f) m_{Wbb}



(g) m_{WWbb}

Figure: 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.

Back-up: ADAM algorithm

Algorithm ADAM algorithm

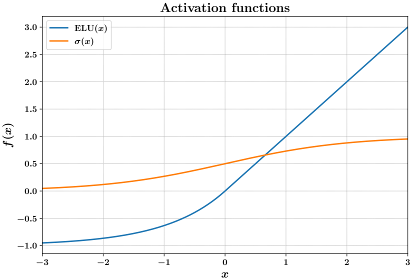
Require: Step size ε (suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$ (suggested defaults: $\rho_1 = 0.9$, $\rho_2 = 0.999$)
Require: Small constant δ, usually 10^{-8}, used to stabilise division by small numbers
Require: Initial weights $\boldsymbol{w}$
Require: Minibatch dimension m

procedure $\mathrm{ADAM}(\{x^{(i)}\}_{i=1,\ldots,n})$ ▷ Input: training dataset of n elements
    Initialize 1^{st} and 2^{nd} moment variables, $s = 0$, $r = 0$
    Initialize time step $t = 0$
    while stopping criterion not met **do**
        Sample minibatch $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
        $g \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{w}} \sum_i L(f(x^{(i)}; \boldsymbol{w}), y^{(i)})$ ▷ Compute gradient estimate
        $t \leftarrow t + 1$
        $s \leftarrow \rho_1 s + (1 - \rho_1)g$ ▷ Update biased 1^{st} moment estimate
        $s \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$ ▷ Update biased 2^{nd} moment estimate
        $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$ ▷ Correct bias in 1^{st} moment
        $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$ ▷ Correct bias in 2^{nd} 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

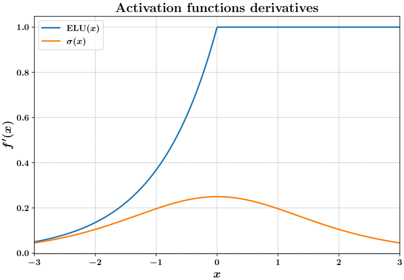
Back-up: activation functions

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

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



(a) Activation functions



(b) Derivatives of activation functions

Figure: Comparison between ELU and sigmoid activation functions in **16a**, and between their derivatives, in **16b**, which are used also for loss function gradient calculation.

Back-up: input vectorisation code


```
# 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)
```