Contents

List of Figures

List of Tables

1 Hadronic recoil regression with deep neural networks

In the recent years a significant progress was achieved in the field of big datasets analysis. There is a number of principles available for solving a wide variety of tasks. In this thesis a DNN was used for the regression of the 2-component hadronic recoil vector.

1.1 Deep neural networks

Normally a machine learning problem has a number of ingredients: a dataset $\mathbf{X}$, a set of parameters θ, a model $g(\theta)$ and a loss function $C(\mathbf{X})$ that tells us how well the model $g(\theta)$ describes the dataset. Finding the values of θ that would minimize the loss function we fit the model.

1.1.1 Gradient descent optimization

One of the most powerful and used class of methods in minimizing the loss function is called the *gradient descent*, [1] especially its sub-class, the stochastic gradient descent (SGD) [2], [3]. One of its modifications called ADAM [4] was used as an optimization algorithm in the work presented in this thesis.

Let's assume that a loss function $E(\theta)$ may be estimated as a sum over n data points:

$$E(\theta) = \sum_{i=1}^{n} e_i(x_i, \theta). \tag{1.1}$$

In the simplest case of the gradient descent (GD) algorithm we start looking for the values of parameters θ such that the sum of functions $\sum_{i=1}^{n} e_i$ is minimal. We start with a certain value θ_0 and then iteratively perform the following:

$$\begin{aligned} v_t &= \eta_t \nabla_\theta E(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t, \end{aligned} \tag{1.2}$$

where $\nabla_\theta E(\theta_t)$ is the gradient of $E(\theta)$ with respect to θ; factor η_t is called the *learning rate* and defines the length of the step in the direction of θ performed with every iteration. Balancing learning rate is very important for learning process and convergence. A value too low can make the convergence "stuck" in the local minimum, it also increases the number of iterations. Picking a very high learning

rate we risk to miss the minimum so the algorithm would never converge to a minimum. Also, if the number of data points n is high, calculating the gradient is a costly task in terms of CPU time.
Some of the problems accompanying the use of GD are dealt with by using its modification - the SGD. The idea is the following: instead of using all the available data points n at each iteration of the GD, we split the data into k *minibatches*, each having M data points, such that $k = n/M$. Normally the size of the batch is few hundreds of data points, to provide a certain degree of variance and incorporating stochasticity. The transition to SGD algorithm is done in the following way:

$$\nabla_\theta E(\theta) = \sum_{i=1}^{n} \nabla_\theta e_i(x_i, \theta) \rightarrow \sum_{i \in B_l} \nabla_\theta e_i(x_i, \theta), \tag{1.3}$$

where B_l is a set of data points belonging to a minibatch $l \in 1, ..., n/M$. Now every next iteration of θ parameters update is performed over a different batch, consecutively running over all the batches:

$$\begin{aligned} &\nabla_\theta E^{EM}(\theta) = \textstyle\sum_{i \in B_l} \nabla_\theta e_i(x_i, \theta), \\ &v_t = \eta_t \nabla_\theta E^{EM}(\theta_t), \\ &\theta_{t+1} = \theta_t - v_t. \end{aligned} \tag{1.4}$$

A full iteration over all the n/M batches is called an *epoch*. Now stochasticity prevents the gradient algorithm from getting stuck in a local minimum. Also computing the gradient over fewer data point notably decreases the CPU time spent.
The algorithm may be further improved, adding a "memory", that is to say making every next step t dependent on the direction of the previous step $t-1$:

$$\begin{aligned} &v_t = \gamma v_{t-1} \eta_t \nabla_\theta E^{EM}(\theta_t), \\ &\theta_{t+1} = \theta_t - v_t. \end{aligned} \tag{1.5}$$

Thanks to analogy from physics the parameter γ is called a *momentum*, having $0 \leq \gamma \leq 1$ [5], [6]. This parameter provides a certain "inertia" in the change of the direction of the gradient descent. Introduction of the momentum helps for quicker convergence in the case of a slow but steady change of a certain parameter during the gradient descent.
The convergence of the GD may be significantly improved if the learning rate could be different in different directions, depending on the landscape of the parameter space θ: the steeper the gradient in a certain direction - the smaller the corresponding step. The optimal step could be estimated by obtaining the *Hessian matrix* in the vicinity of a point θ_0, providing a description of the local curvature in a multidimensional space. Although calculating Hessian matrix is complicated and slow-converging process [7]. However, a number of methods use the second moment of the gradient to efficiently estimate the optimal learning rate. One of such methods is called ADAM (ADAptive Momentum) [4],

its iterative relations are the following:

$$
\begin{aligned}
g_t &= \nabla_\theta E(\theta_t) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
s_t &= \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-(\beta_1)^t} \\
\hat{s}_t &= \frac{s_t}{1-(\beta_2)^t} \\
\theta_{t+1} &= \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{s}_t}+\epsilon}.
\end{aligned}
\tag{1.6}
$$

Here the parameters β_1 and β_2 set the memory lifetime for the first and second moment; $\eta$is the learning rate and ϵ is a small regularization constant keeping the denominators from vanishing. Like in other cases of the SGD here the iterations are performed batch-wise. Parameter s_t is linked to the variance of the gradient size. This basically means that the learning rate is proportional to the first momentum of the gradient and inverse proportional to its standard deviation.

1.1.2 DNN structure and training

A neural network is composed of single neurons, also called nodes, arranged in layers. The first layer is called the input layer, the last one is called the output layer; all the layers in between are named hidden layers (see Fig. 11).

A single node i takes a vector of k input features $\mathbf{x} = (x_1, x_2, ..., x_k)$ and produces a scalar input $a_i(\mathbf{x})$. Function a_i may have a different form, although it normally can be decomposed into two steps. The first step is a linear transformation of the inputs into a scalar value assigning each input a weight:

$$
z^i = w_k^i \cdot x_k + b^i, \tag{1.7}
$$

where $\mathbf{w}^i = (w_1^i, w_2^i, ..., w_k^i)$ is a set of k weights assigned to corresponding inputs. The weights $\mathbf{w}^i$ are specific to a neuron i, as well as the scalar bias b^i. The next step is where the non-linear function σ_i comes into play: we can express the output function $a_i(\mathbf{x})$ as follows:

$$
a_i(\mathbf{x}) = \sigma_i(z^i). \tag{1.8}
$$

There exists a number of options for the non-linear function σ; in current thesis a tanh is used. When the neurons are arranged in layers in a feed-forward neural network - the outputs from neurons of the previous layer serve as inputs for the succeeding layers neurons. The universal approximation theorem states, that a neural network with a single hidden layer can approximate any continuous multiparametric function with arbitrary accuracy [8], [9]. However, in practice it is easier to reach the possible precision having more hidden layers.

So in terms of a DNN fitting the model means tuning the weights and biases $(\mathbf{w}^i, b^i)$ in such a way that a loss function applied to the new dataset would be minimal. It is reached through iterative process called *training*, that involves the GD with an algorithm called *backpropagation* [10]. The backpropagation algorithm allows to calculate the gradients and adjust the corresponding parameters

in a very computation-efficient way.

Let us assume that there are L layers in the network $l = 1,...,L$, that w^l_{jk} and b^l_j are the weight of an

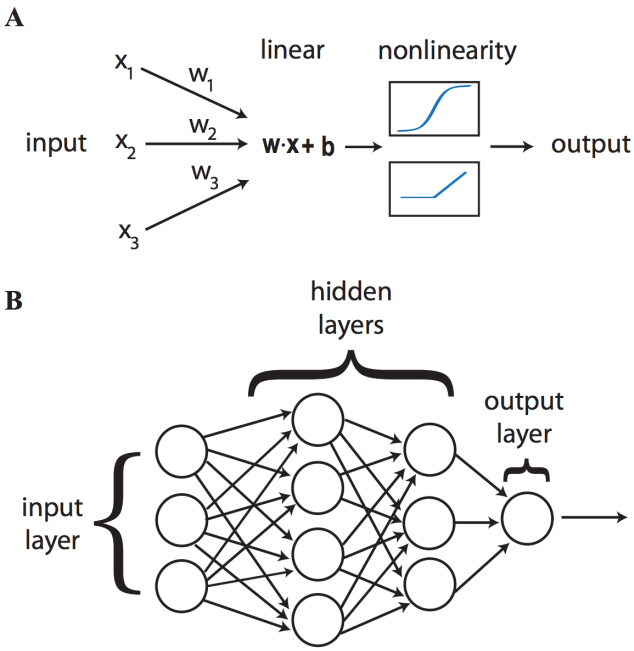

Figure 11: A: The nodes perform a linear transformation of the inputs, then apply a non-linear activation function. B: The architecture of a deep neural network: neurons are arranged into layers [11].

input parameter k and the bias for node k in layer l respectively. The layered structure of the neural network ensures that the inputs for the nodes in layer l depend only on the outputs of the nodes from layer $l-1$, hence:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) = \sigma(z^l_j), \tag{1.9}$$

where the linear weighted sum is denoted as:

$$\sigma(z^l_j) = \sum_k w^l_{jk} a^{l-1}_k + b^l_j. \tag{1.10}$$

The cost function E is computed from the output of the neural network, so it directly depends only on the values of a^L_j. Let us define the error Δ^L_j of the j-th node in the output (L-th) layer as a change in the cost function with respect to the weighted output of the last layer:

$$\Delta^L_j = \frac{\partial E}{\partial z^L_j}. \tag{1.11}$$

At the same time the loss depends indirectly on all the preceding layers, so keeping in mind eq. 1.9 we can define the error of an arbitrary node j in arbitrary layer l as the change in the cost function E with respect to the weighted input z^l_j:

$$\Delta^l_j = \frac{\partial E}{\partial z^l_j} = \frac{\partial E}{\partial a^l_j}\sigma'(z^l_j), \tag{1.12}$$

where $\sigma'(z_j^l)$ is the derivative of the non-linear activation function σ with respect to its input at z_j^l. But on the other hand we can also interpret the error function Δ_j^L in terms of bias partial derivatives:

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l}\cdot \mathbf{1}. \tag{1.13}$$

So starting from the output layer we can compute the error in any layer l, provided we know it for the subsequent layer $l+1$:

$$\begin{aligned}\Delta_j^l &= \frac{\partial E}{\partial z_j^l} = \sum_k \frac{\partial E}{\partial z_j^{l+1}}\frac{\partial z_j^{l+1}}{\partial z_j^l} = \\ &= \sum_k \Delta_j^l \frac{\partial z_j^{l+1}}{\partial z_j^l}\left(\sum_k \Delta_j^l w_{kj}^{l+1}\right)\sigma'(z_j^l).\end{aligned} \tag{1.14}$$

And finally we can get the gradient of the cost function E with respect to a weight of an arbitrary neuron:

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1}. \tag{1.15}$$

Using these four equations (1.11, 1.13, 1.14, 1.15) it is possible to "backpropagate" the error back from the output layer and once we can compute the gradient - we know how we should tune the weights and biases in order to minimize the loss function.

1.1.3 Batch normalization

Batch normalization is a regularization scheme that helps to improve the speed and stability of the DNN training. The main idea behind the method is to prevent an *internal covariant shift* - a change in the distribution of network activations due to the change in network parameters during training by means of normalization of the parameters transferred from layer l to layer $l+1$ [12]. So let us consider a layer l that has d inputs $\mathbf{x} = (x^1, x^2, ..., x^d)$, then for every x^k we perform the following transformation:

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}, \tag{1.16}$$

where $E[x^k]$ and $Var[x^k]$ are the expectation and variance of the parameter x, calculated over the training dataset, respectively. Although we have to be sure that we preserve the non-linearity of the activation function output. In order to do this the two additional parameters are introduced:

$$y^k = \gamma \hat{x}^k + \beta^k, \tag{1.17}$$

where the parameters γ and β are trained just like the rest of the network parameters. Practically if the training is performed within the mini-batch scheme with batch size $B = x_1, ..., x_m$ the batch normalization layer is inserted between the DNN layers the transformations for the input x are the

following:

$$\begin{aligned} &\frac{1}{m}\sum_{i=1}^{m} x_i \rightarrow \mu_B \\ &\frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \rightarrow \sigma_B^2 \\ &\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \rightarrow \hat{x_i} \\ &\gamma \hat{x_i} + \beta \rightarrow y_i \equiv BN_{\gamma,\beta}(x_i), \end{aligned} \tag{1.18}$$

where ϵ is a small regularization constant.

1.2 HR regression

Considering that hadronic recoil is an observable what uses many inputs from Inner Detector (ID), electromagnetic calorimeter (EMC) and hadronic calorimeter (HC) it is reasonable to expect improvement of the result using modern MultiVariate Analysis (MVA) techniques.

1.2.1 Input features and model

Training, testing and validation was performed using the MC sample $W^+ \rightarrow \mu\nu$ at 13 TeV with the same selection. From the 3625136 events that have passed the selection 12 734 109 were used for training and 3 034 130 for testing the performance. Below is the list of 38 input features:

- **Hadronic recoil** is possible in a number of definitions. As it was described before, the Hadronic Recoil (HR) may be defined using exclusively charged Particle Flow Objects (PFO), exclusively neutral PFO or both. All three definitions are included to the input features in the with two Cartesian components for each definition, making 6 input features.
- **Transverse energy sum** $\sum E_T$ is also defined in three similar ways, adding three input features.
- **C**artesian components of the two leading jets momenta in the transverse plane. The jets were demanded to have $p_T > 20$ GeV. If one or both jets don't make the cut or there is less than two jets in the event - the corresponding features were assigned zero value.
- Cartesian components of the five leading Neutral Particle Flow Objects (nPFOs) and five leading Charged Particle Flow Objects (cPFOs) momenta in the transverse plane.
- Number of primary vertices in the event.
- Pile-up value μ.
- Total number of jets in the event.
- Total number of nPFOs and cPFOs in the event.

All input features were pre-processed using the StandardScaler module from Scikit Learn package [13]. The model contains 3 dense layers with 256 neurons each, alternated with batch normalization layers (see Fig. 12). Using batch normalization layers has allowed to reduce the training time by the factor of 10. The model has used Adam optimizer with learning step 0.001 and batch size of 4000 data points.

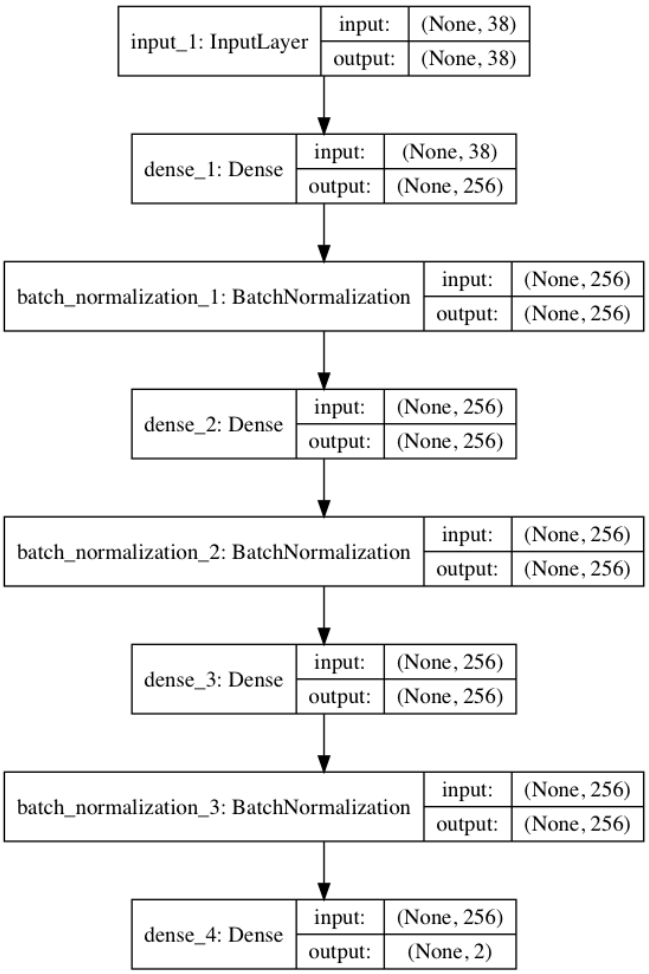

Figure 12: A model of the DNN used in the analysis.

Twenty percent of events were used for validation. The two target values were Cartesian components of the truth HR vector. Figure 13 shows the training loss diagram. Eventually the model with weights obtained after 38 epochs of training was used in the analysis.

1.2.2 Kinematic distributions

The results presented here demonstrate the regression plots obtained with the trained DNN. The regression was tested for the four W channels in MC and for the two Z channels in both data and MC. Below the plots for $W^- \to \mu\nu$ channel are presented, the rest of the results are presented in Appendix B.

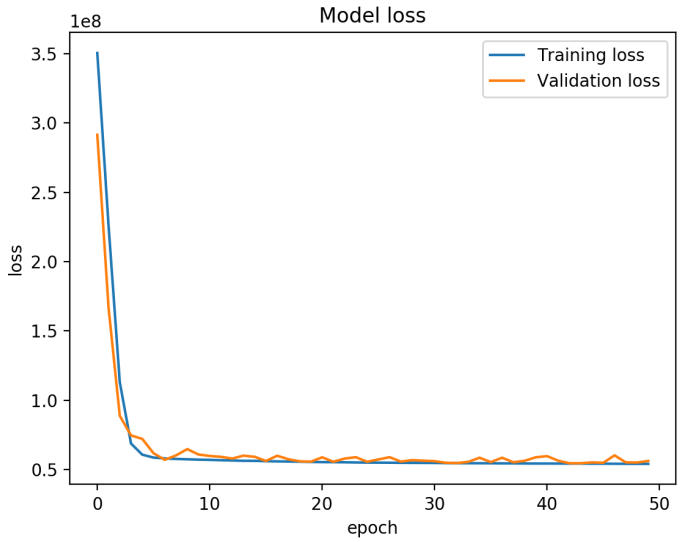

Figure 13: Learning curve of the model.

In Figure 14 the difference between the hadronic recoil and truth distributions are shown. For the two target components u_X and u_Y a sharper peak centred at zero is observed in the MVA-reconstructed recoil comparing to the standard algorithm. Similar comparison is shown in Fig. **??** for HR vector components in polar coordinates. The polar angle u_ϕ of the HR vector show a very small, nearly negligible improvement. The u_T vector magnitude demonstrates a shift from zero for the u_T^{MVA}, indicating a bias. Figure 16 shows that the MVA-reconstructed recoil has a softer spectrum comparing to both p_T^{truth} and standard recoil spectra.

The bias, defined as $1 - \frac{u_\parallel}{p_T^{truth}}$, is shown in Fig. **??** together with the u_{perp} component. Indeed the u_T^{MVA} demonstrates a larger bias comparing to the standard algorithm. At the same time the $u_\perp$ component indicates the improvement in the resolution.

The dependence of the bias and $u_\perp$ on the momentum is studied in Fig. 18. The u_T^{MVA} recoil demonstrates improvement in the resolution and a larger bias in the region of $p_T < 80$GeV. However, for a quantitative resolution comparison we need to make sure that $u_\perp$ and $u_\perp^{MVA}$ are on the same scale. A possible way to achieve this is to normalize them to average recoil $< u_T >$. The resulting normalized curves on Fig. 19 show that the MVA provides 5-10% resolution improvement at $p_T < 10$GeV and a bit more than 10% in $10 < p_T < 30$GeV transverse momentum region.

The same neural network was applied for the HR regression for both Z channels, for data and MC events. In case of data the p_T^{truth} vector was replaced with $p_T^{\ell\ell}$. A qualitatively similar picture holds for both Z channels, confirming recoil universality for W and Z events (see Fig. 110). The complete set of u_T vs u_T^{MVA} comparison plots for all W and Z channels can be found in Appendix B.

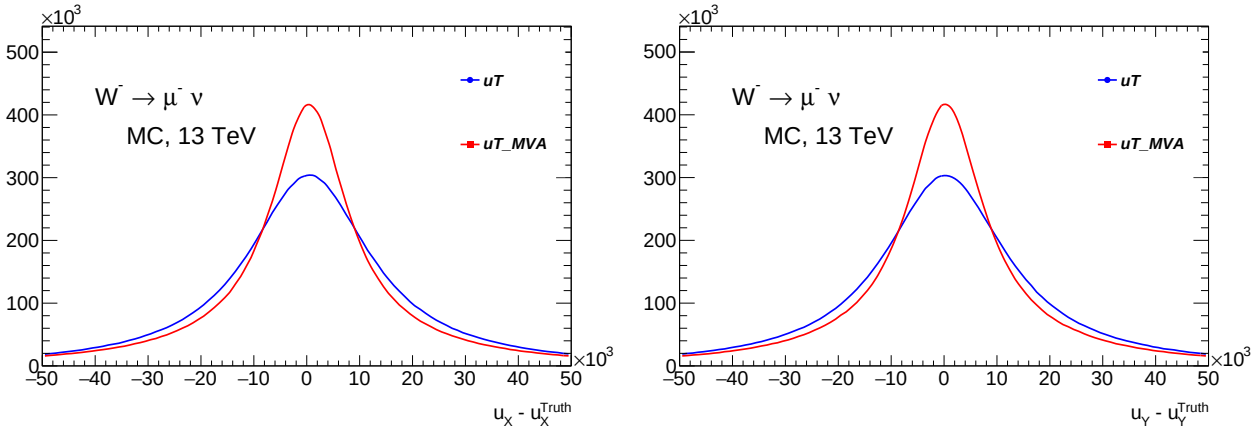

Figure 14: The difference between the Cartesian components of the HR vector for the standard u_T and u_T^{MVA}.

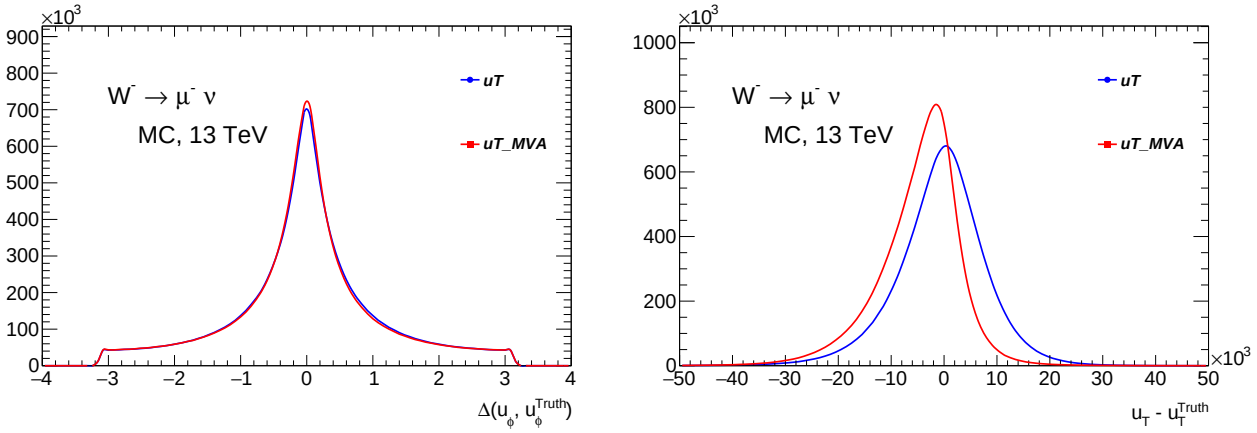

Figure 15: The difference between the polar components of the HR vector for the standard u_T and u_T^{MVA}.

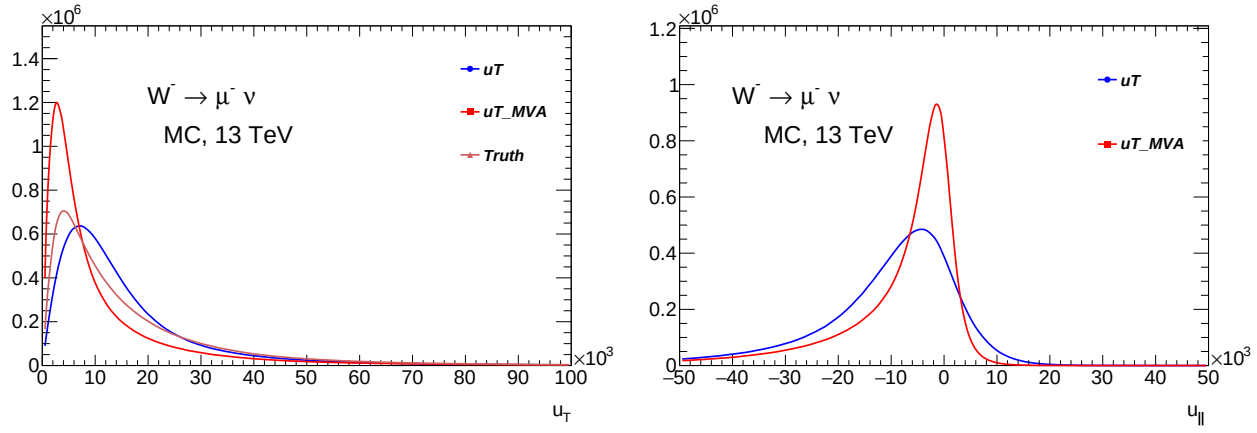

Figure 16: Comparison of u_T and u_T^{MVA} spectra and response.

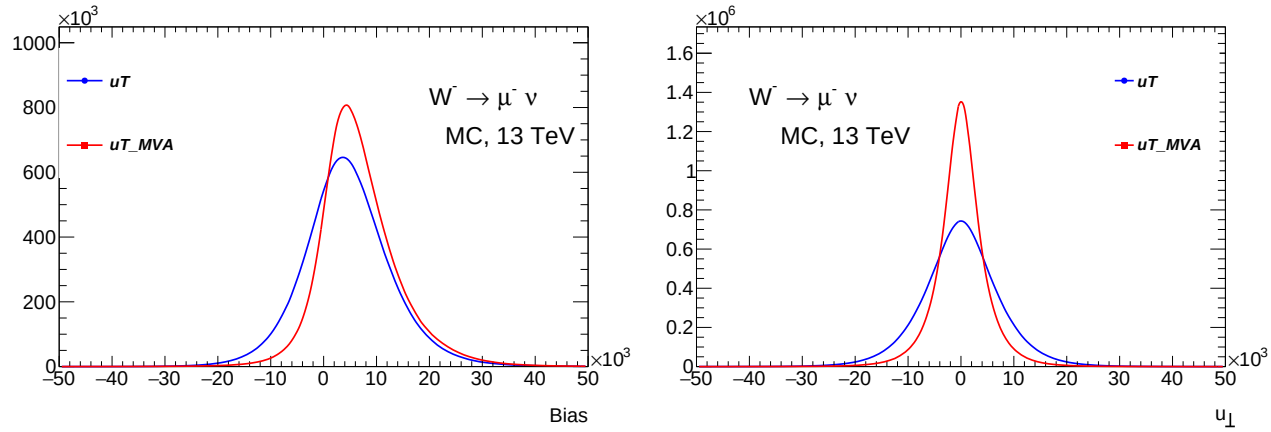

Figure 17: Bias and $u_\perp$ for the standard u_T and u_T^{MVA}.

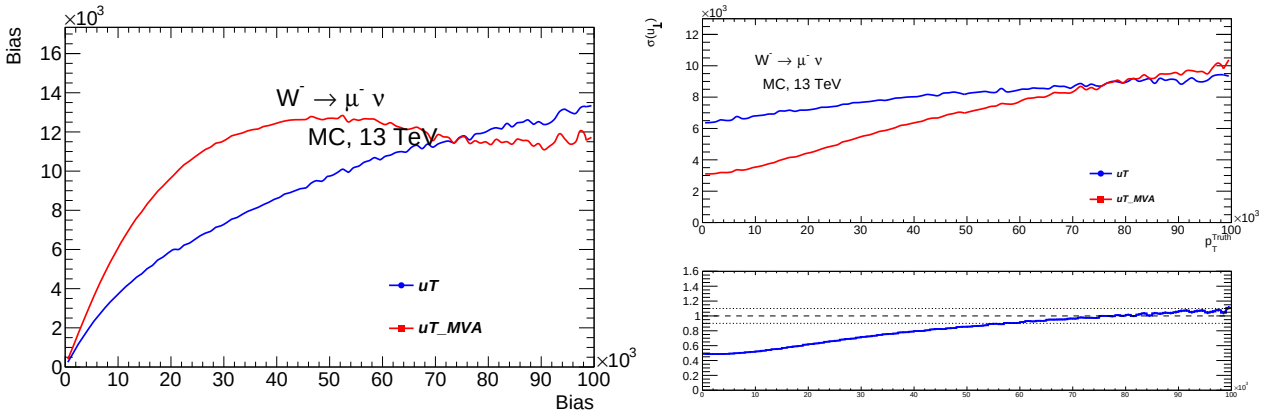

Figure 18: Bias and $u_\perp$ as a function of p_T^{truth}.

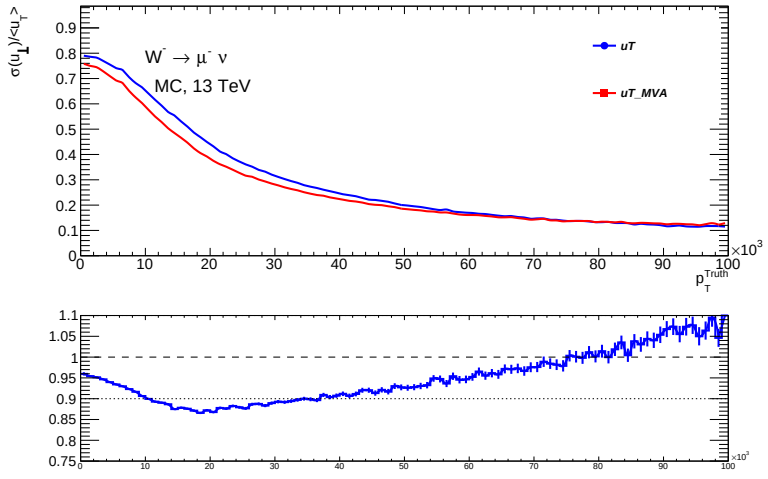

Figure 19: Normalized $\frac{u_\perp}{<u_T>}$ as a function of p_T^{truth}.

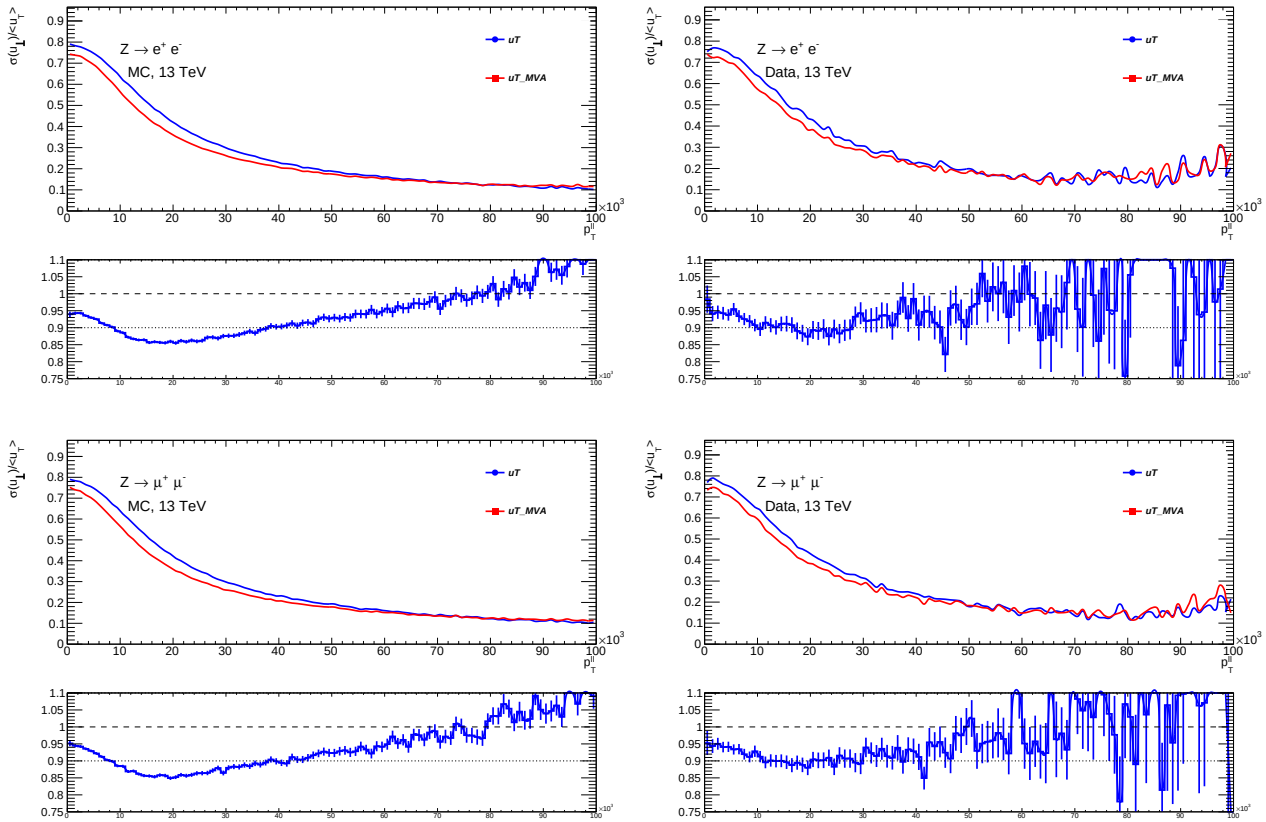

Figure 110

1.3 Control plots and spectrum

A dedicated calibration was derived for u_T^{MVA} in the same way as it has been done for the standard HR. The control plots for the u_T^{MVA} observable are shown in Fig. 111.

The MVA unfolded spectrum is presented in comparison with other predictions in Fig. **??**.

1.4 Conclusions

The application of contemporary analysis methods like the regression using DNNs shows a promising potential for improving the measurements precision.

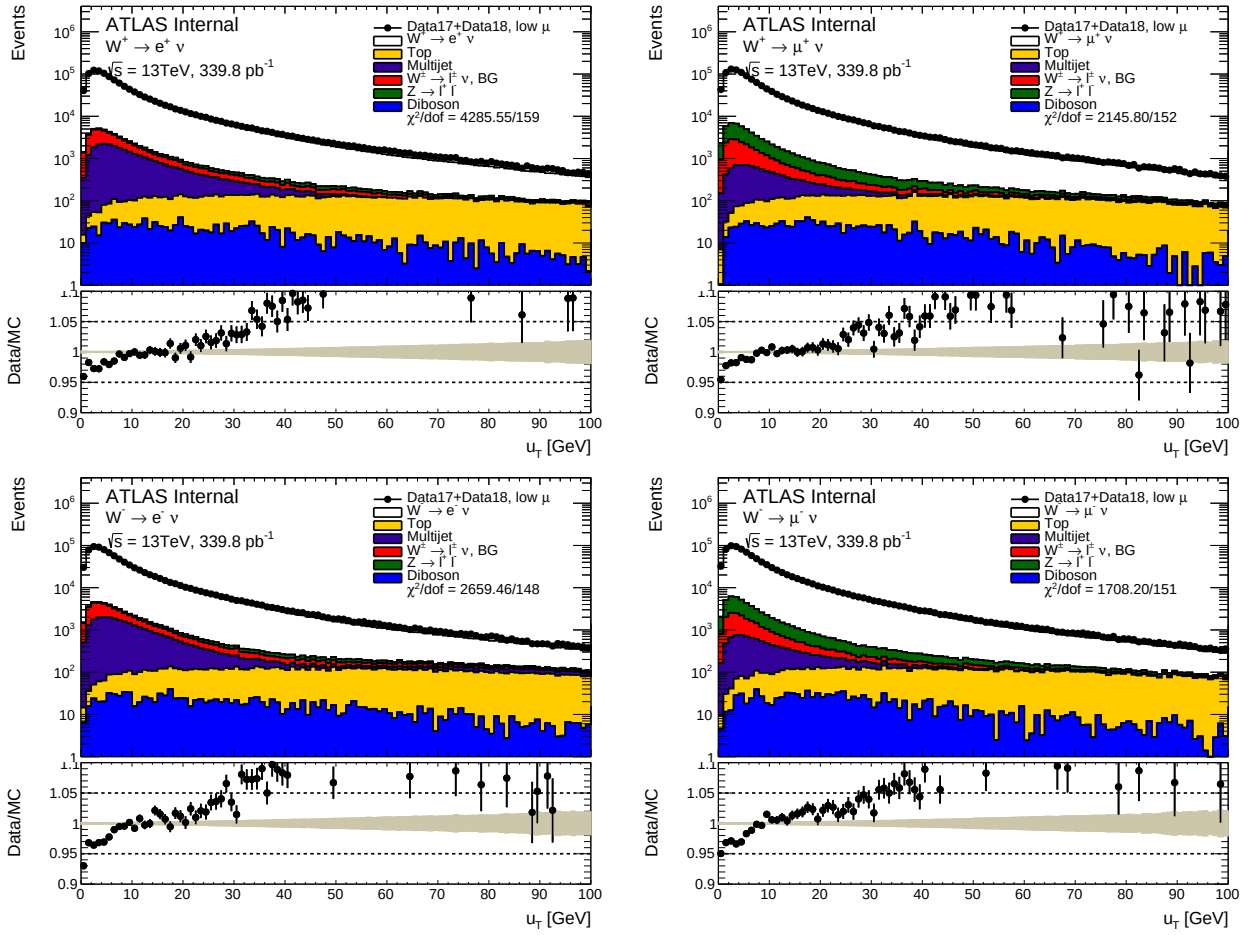

Figure 111: Control plots for u_T^{MVA} at 13 TeV.

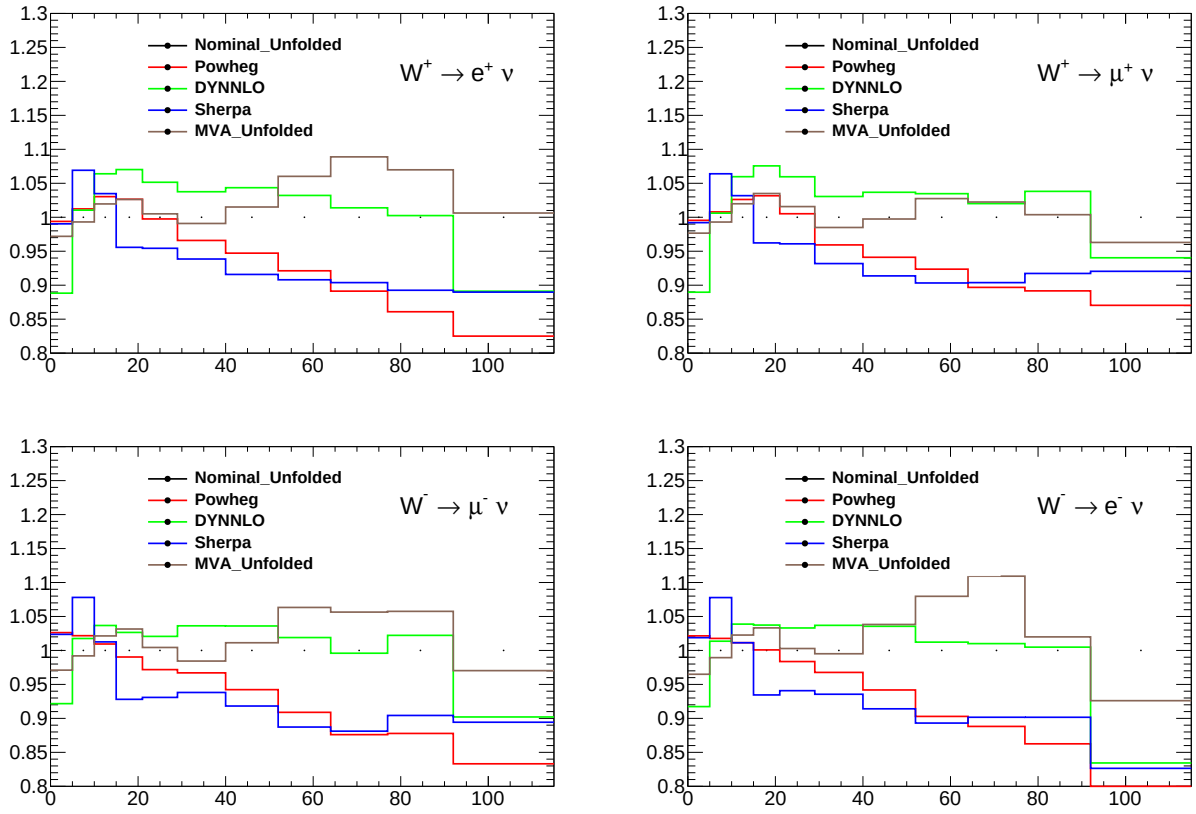

Figure 112: Unfolded spectrum predictions comparison at 13 TeV.