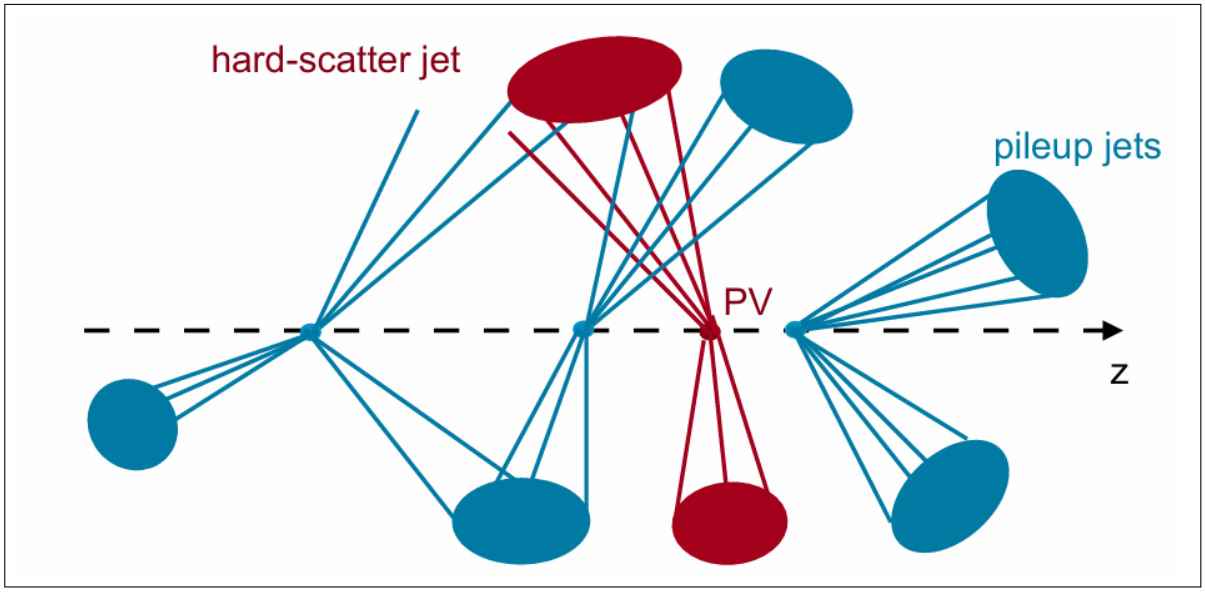
Exploring new avenues for pileup jet rejection using machine learning

CERN SUMMER STUDENT PROJECT

Floris Peter Meijvis



Supervisors:

M. Holzbock
ATLAS

T.J.Khoo
ATLAS

August 29, 2025

Abstract

In this project we train a neural network for the distinction between hard-scatter and pileup jets. In particular, we are interested if certain changes in the networks architecture and input features can lead to an increase in the pile-up rejection rate. We consider both simple deep neural networks and networks based on transformer architecture. By comparing ROC curves and the background rejection rates at a fixed hard-scatter jet detection efficiency of 95 %, we conclude that the addition of several input variables can lead to increase in pile-up jet rejection of 8.8 % for the feed-forward network, and 17.3 % for the transformer network when compared to the performance of the legacy neural network employed by ATLAS. These values are for the entire kinematic range spanning a transverse momentum of 15 to 60 GeV

Contents

1 Introduction

During a bunch crossing of the LHC beams in the ATLAS detector, multiple protons in the bunch interact. This ultimately leads to multiple events per bunch crossing, and therefore multiple similar proton-proton interactions detected by ATLAS. The jets of the hard-scatter event originates from the primary interaction vertex, which is the vertex with the highest jet transverse momentum. Hard-scatter events typically have a higher momentum transfer than pileup interactions. Pileup jets also do not originate from the primary interaction vertex but instead are coming from pileup vertices as shown in figure 1.

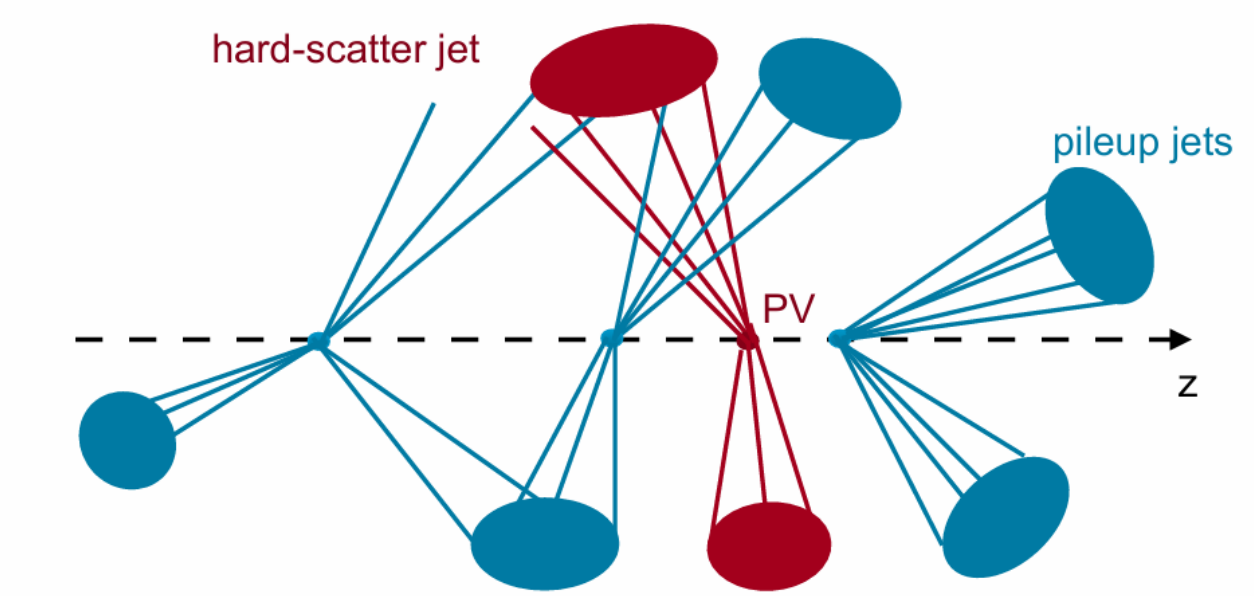


Figure 1: Schematic illustrating the difference between pileup and the hard-scatter interactions during a bunch crossing at the LHC. Here, PV stands for primary (interaction) vertex, and each line corresponds to a particle's track. Figure by F. Rubbo (2016).

In this project we train neural networks for the distinction of hard-scatter and pileup jets using a set of input features available at trigger level and an expanded input features set. We compare the performance with the legacy neural network employed by ATLAS. Throughout this report, we will use the following abbreviations:

- HS - Hard-scatter
- PU - Pileup
- NN - Neural network

2 Training data and input features

The data used in this project is simulated with Monte Carlo generators, and uses detector data taking conditions from 2015, 2017 and 2018. These simulations generate di-jet events, which result in a set of reconstructed jets. From each of these jets, we calculate a set of variables that have some HS and PU seperating power. The full list of variables used in this project is given later in this section. It is important to note that in both the Monte Carlo simulated data and real data, the primary vertex is the interaction vertex for which the sum of the transverse momenta for tracks associated with that vertex is largest.

The NNs are trained on 10 million jets with the majority of p_T values ranging from 15 to 60 GeV. The test dataset consists of 5 million jets covering the same regime. Both datasets contain PU and HS jets in a roughly 1:1 ratio. The full list of input features used for our neural networks is given in the lists below. Set notation is used to refer to a collection of variables, each corresponding to a different vertex number. vx0 is the primary HS vertex, the vertex with the greatest transverse momentum. vx1 and vx2 are the PU vertices with largest and second largest sum of track p_T. Variables with $Pt500$ as part of the name only consider tracks with a p_T past 500 MeV. A similar definition applies to $Pt1000$.

Input variables associated with a particular jet

- p_T: Jet transverse momentum
- **eta**: Jet pseudorapidity
- **JVFCorr**: Corrected Jet Vertex Fraction as given in equation 1.

Input variables derived from per-vertex information of a jet

- **MaxNumTrkPt1000**: Maximum number of tracks in jet assosciated with any vertex
- **MaxSumPtTrkPt500**: Maximum value of sum of track p_T from any vertex
- **MaxRPtTrkPt500**: "MaxSumPtTrkPt500" divided by p_T
- **MaxTrackWidthPt1000**: The maximum value for the track width, defined as the p_T-weighted average of the difference of the angular distance of each track to the jet axis.
- **SumPtTrkOrderedNumTrkPt1000_vx**$\{0, 1, 2\}$: Number of tracks from the corresponding vertex
- **SumPtTrkOrderedSumPtTrkPt500_vx**$\{0, 1, 2\}$: The sum of track transverse momenta from the corresponding vertex
- **SumPtTrkOrderedRPtTrkPt500_vx**$\{0, 1, 2\}$: SumPtTrkOrderedSumPtTrkPt500_vx$\{0, 1, 2\}$ divided by p_T

- **SumPtTrkOrderedTrackWidthPt1000_vx**$\{0, 1, 2\}$: The track width from the corresponding vertex

Input variables derived as the differences between other input features

- **DNumTrkPt1000_vx**$\{1, 2\}$: Difference between SumPtTrkOrderedNumTrkPt1000_vx0 and -vx$\{1, 2\}$
- **DSumPtTrkPt500_vx**$\{1, 2\}$: The difference between SumPtTrkOrderedSumPtTrkPt500_vx0 and -vx$\{1, 2\}$
- **DTrackWidthPt1000_vx**$\{1, 2\}$: The difference between SumPtTrkOrderedTrackWidthPt1000_vx0 and -vx$\{1, 2\}$
- **DRPtTrkPt500_vx**$\{1, 2\}$: The difference between SumPtTrkOrderedSumPtTrkPt500_vx0 and -vx$\{1, 2\}$

$$\text{JVFCorr} = \frac{\sum_k p_T^{\text{trk}_k}(\text{PV}_0)}{\sum_l p_T^{\text{trk}_l}(\text{PV}_0) + \sum_{n\geq 1} p_T^{\text{trk}_l}(\text{PV}_n)} \tag{1}$$

Here, PV_0 is the primary HS vertex and PV_n with $n \geq 1$ are the PU vertices.

Since we have less jets at high values of p_T, we reweigh the data such that high p_T jets have higher weights in the loss function for the training. This should prevent the NN from losing performance in the higher p_T regimes. The p_T distribution before reweighing, and plots for the variables η, JVFCorr and Rpt can be seen in figure 2.

Previous studies at ATLAS have resulted in a neural network for the distinction between HS and PU jets. This network is currently still used, and will be referred to as the Legacy NN in this project. The legacy NN is a simple dense linear neural network. The first network architecture that we will be considering in this project is therefore also a simple dense linear network. The architecture of this neural network is given in section A.1. Furthermore, due to the large amount of data available, we will also consider a transformer neural network. The architecture of this network is given in A.2. The legacy network was trained on variables p_T, η, JVFCorr and SumPtTrkOrderedRPtTrkPt500_vx0.. Previous studies at atlas resulted in a set of variables that have the most discriminating power for identifying PU and HS jets. these variables are p_T, η, SumPtTrkOrderedSumPtTrkPt500_vx0, SumPtTrkOrderedRPtTrkPt500_vx0, SumPtTrkOrderedTrackWidthPt1000_vx0, DNumTrkPt1000_vx1, DTrackWidthPt1000_vx1 and DRPtTrkPt500_vx1. we will train a neural network with these variables to see if this conclusion also holds at reconstruction level. We will also train a neural network using all of the inputs described previously.

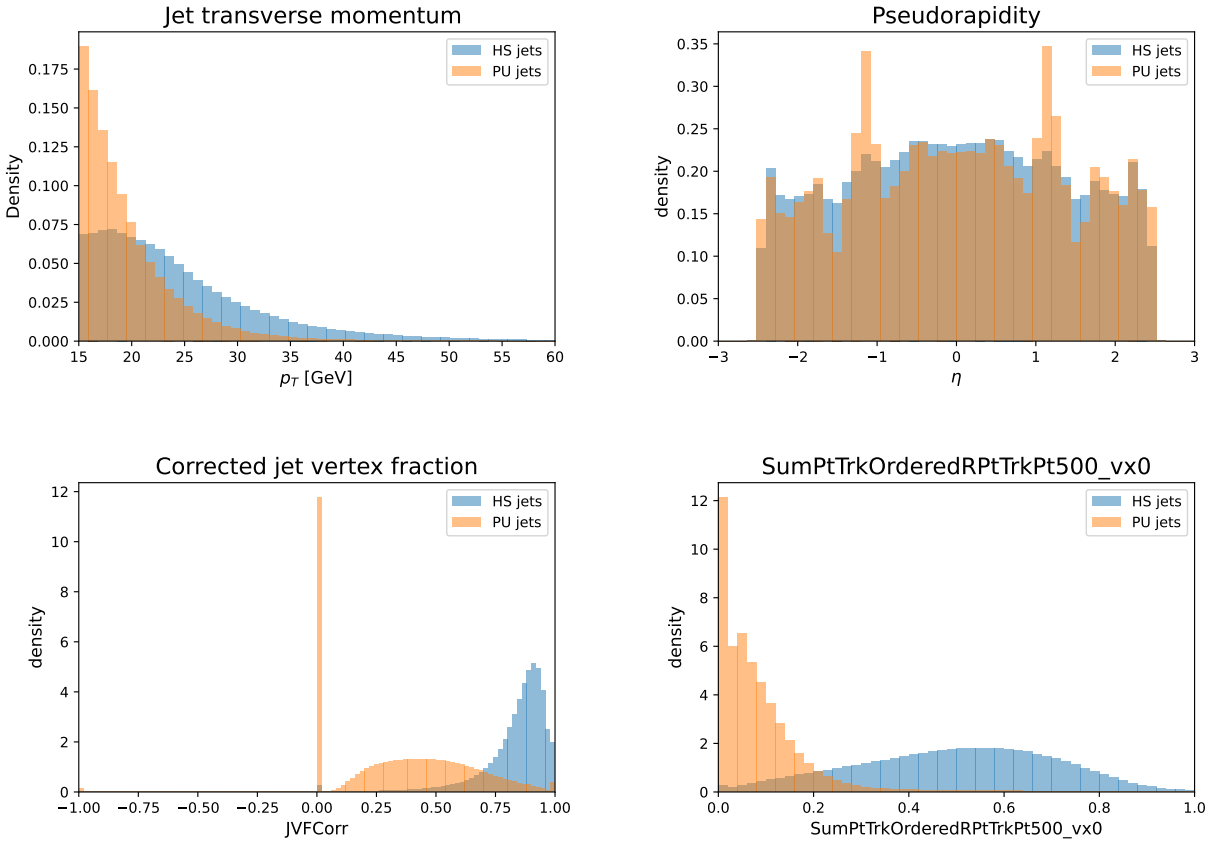


Figure 2: Density distributions of 4 sample variables in training data. If JVFCorr has a value of -1, this represents a jet that has no associated track.

3 Neural network training and architecture

The architecture of the feed forward neural network employed in this project is given in appendix $A.1$, and will be referred to as the DeepNN from now. The network consists of 5 hidden layers, with $128, 64, 32, 16, 1$ neurons respectively. The network uses ReLU activation functions, and the final activation function is a sigmoid. This ensures that the output of the NN is a number in $(0, 1)$, which can be interpreted as a probability of a jet originating from a HS vertex.

The transformer NN's architecture is given in appendix A.2. This network first projects each input feature into a latent space of dimension 128 using a linear layer, and then applies 4 transformer layers. This output is then used in a feed-forward neural network with ReLU activation functions and dropout layers. Similar to the DeepNN, the final activation function is a sigmoid to once again get a probability of a jet originating from a HS vertex.

Both the Transformer and the deep neural network are set up with a patience of 15, meaning that after 15 epochs of no improvement in performance the networks stop training. The deep NN uses an exponentially decaying learning rate, such that every 10 epochs the learning rate reduces by a factor of 2. The starting value of the learning rate is 10^{-3}. The Transformer NN uses a warm-up phase such that the learning rate linearly increases batch by batch to a value of 10^{-3}, which lasts for 10 epochs. After this we use the "ReduceOnPlateau" scheduler to reduce the learning rate by a factor of 2 once it detects no improvement in performance for 7 epochs.

4 Results and Discussion

The neural networks will be compared using several metrics. These are the receiver operater characteristic curves (ROC-curves) and the PU rejection rate defined as

$$\frac{1}{\epsilon_{bkg}} \tag{2}$$

where ϵ_{bkg} corresponds to the false positive rate (fpr) at a fixed HS jet efficiency. In this project we assign the value of 0.95 to this fixed tpr. This means we compare the PU rejection rate of the networks when the retainment threshold of the output scores of the NN is chosen such that 95 percent of the HS events are kept by the neural network.

4.1 Impact of training data features and effects of weights

Previous studies done at ATLAS have shown that the variable JVFCorr does not add a significant boost in performance when adding it to the training dataset. However, a NN trained with JVFCorr in the input features seems to perform better than a NN without JVFCorr supplied, as shown in figure 3. The NN's otherwise have the same architecture and inputs. In each sub regime of p_T as well as the entire p_T range, the NN with JVFCorr outperforms the NN without JVFCorr in terms of background rejection at a fixed HS jet efficiency of 0.95.

Furthermore, we find that the addition of weights to the loss function does not improve performance. In fact, an unweighed loss function seems to result in a better NN, as depicted in figure 4. The change in performance in each sub regime of p_T is minimal, but the unweighed net performs better across the entire p_T spectrum. We would expect that the weights would improve performance in the high p_T regions, but this does not seem to be the case.

4.2 Performance of dense linear networks VS legacyNN

We now make a comparison between 3 different neural networks: a NN trained on a set of variables used at HLT trigger level, and a NN trained on the full set of input features. Each of these NNs is a dense linear network with architecture given in A.1. The final network is the legacyNN employed by ATLAS.

Table 1 below gives the values of the background rejection rate as defined in equation 2 for the entire regime of p_T and for several sub-ranges. We can see that the network trained on the full set of input features clearly outperforms the NN trained trained with only the subset of features used for the trigger-level PU classifier. Although the LegacyNN outperforms the NN trained on the full set of features in each seperate bin of p_T, the overall performance when considering all regions simultaneously is slightly worse. The performance increase of the NN trained on the full set of input features is 8.8% when considering the entire p_T spectrum.
We investigate further the differences between the NN trained on the full set of input features and the LegacyNN of ATLAS. Figure 5 shows the ROC curves split in several bins of p_T, as well as the entire p_T spectrum.

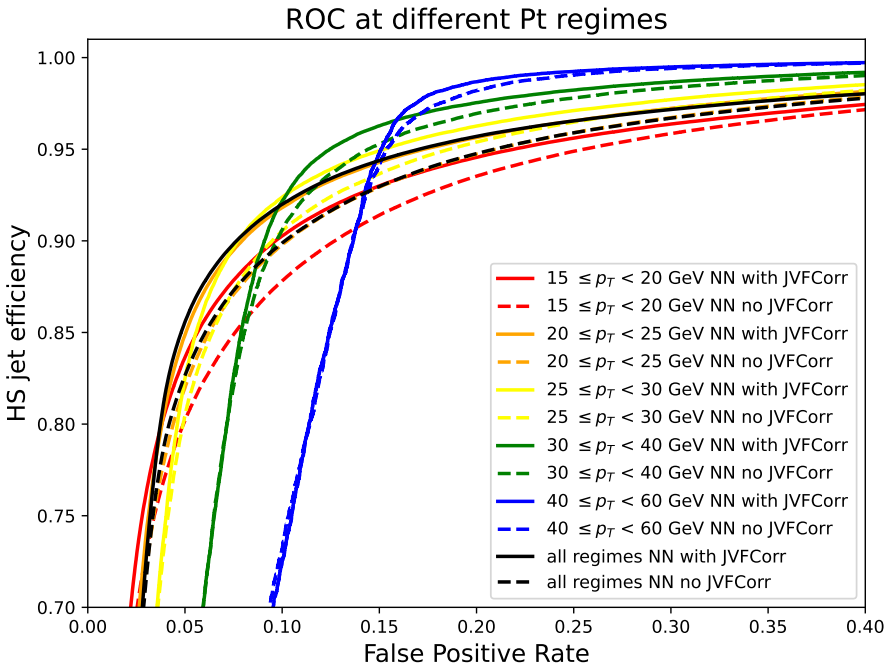


Figure 3: ROC curves of a NN trained with and without JVFCorr in the input features, split by regimes of p_T.

Regime	NN HLT	NN full	LegacyNN
All p_T regimes	4.96	6.05	5.56
$15 < p_T < 20$ GeV	4.01	4.64	4.71
$20 < p_T < 25$ GeV	4.98	5.94	6.25
$25 < p_T < 30$ GeV	5.70	6.86	7.21
$30 < p_T < 40$ GeV	7.36	8.31	8.60
$40 < p_T < 60$ GeV	6.57	6.68	6.67

Table 1: Comparison of the background rejection rate 2 for several neural networks in different regimes of p_T. Here, NN HLT is the neural network trained on the features used for the trigger level classifier, NN full is the network trained on the full input features set, and LegacyNN is the legacy network employed by ATLAS. These values are derived from a true positive rate of 0.95.

4.3 Performance of transformer networks VS legacyNN

Given that the full set of input features outperforms the set used at trigger level, we will train our first transformer NN on the full set of input features in table 2. In particular, we use the architecture given in A.2, where the embedding dimensiion is set to a value of 128, the number of attention heads is 8, and the maximum learning rate is 10^{-3}. Furthermore, at the start of training the learning rate increases linearly on a batch by batch basis for 10 epochs until it reaches 10^{-3}.

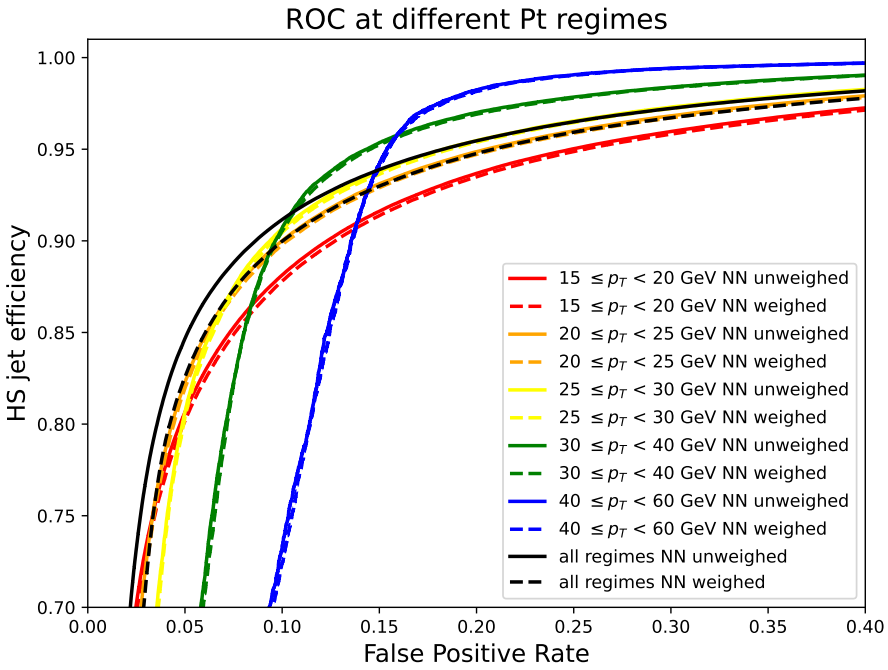


Figure 4: ROC curves of a NN trained with and without reweighed datapoints, split by regimes of p_T.

The values for the background rejection rate are reported in table 2. In particular, we note that the performance over the entire p_T regime has increased when compared to the DeepNN trained on the full set of input features, yet the rejection rate in each bin of p_T has reduced compared to the values in table 1. The corresponding ROC curves can be seen in figure 6. Ultimately, the performance across the entire p_T spectrum is increased by 16.5 %.

Regime	T-NN, n_heads = 8	T-NN, n_heads = 16	LegacyNN	DeepNN
All p_T regimes	6.48	6.52	5.56	6.05
$15 < p_T < 20$ GeV	4.51	4.55	4.71	4.64
$20 < p_T < 25$ GeV	5.72	5.75	6.25	5.94
$25 < p_T < 30$ GeV	6.59	6.60	7.21	6.86
$30 < p_T < 40$ GeV	7.97	7.99	8.60	8.31
$40 < p_T < 60$ GeV	6.63	6.67	6.67	6.68

Table 2: Comparison of the background rejection efficiency 2 for the transformer NN (abbreviated as T-NN) at different values of the n_heads parameter and the legacyNN. The values are evaluated at a true positive rate of 0.95.

A key feature of transformer networks is the attention mechanism. The results shown in table 2 are for a network with 8 attention heads. We have also trained a network with the n_heads parameter set to 16. these results are also shown in table 2, and are marginally

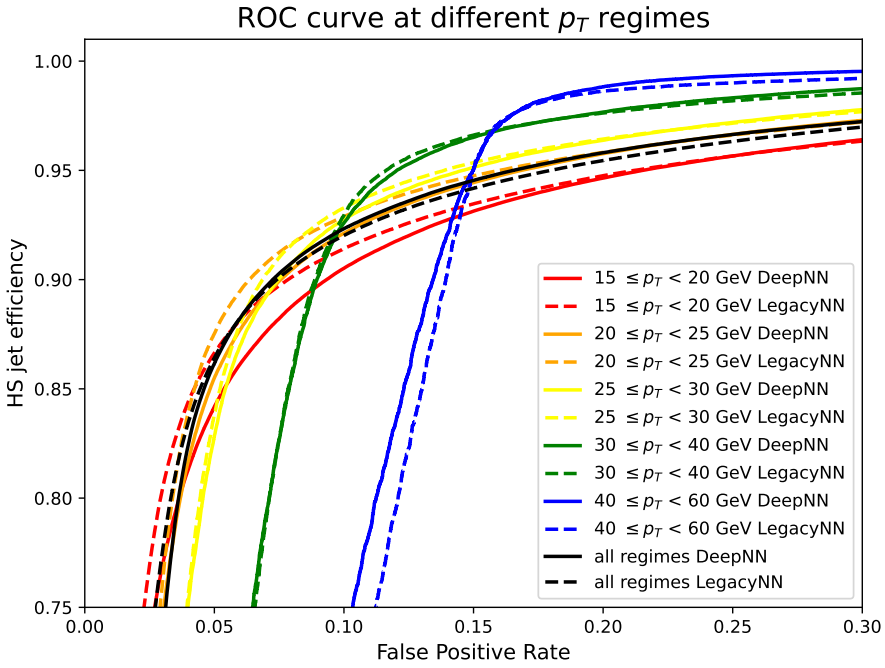


Figure 5: ROC curve of the legacyNN and the new DeepNN split in subregimes of p_T as well as the full range of $15 \leq p_T < 60$ GeV.

better than that of the transformer with 8 attention heads. The performance across the entire p_T spectrum is now improved by 17.3 %.

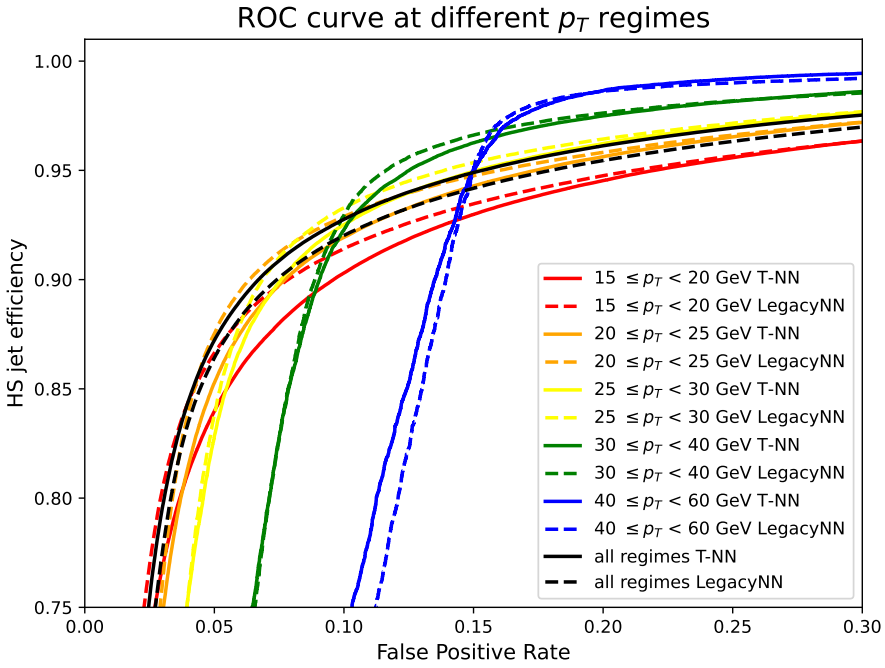


Figure 6: ROC curve of the legacyNN and the Transformer NN with 16 attention heads split in subregimes of p_T as well as the full regime of $15 \leq p_T < 60$ GeV.

5 Conclusion

As shown by the values in table 1, it is possible to construct a NN that outperforms the legacy model employed by ATLAS when considering the full range of p_T. However, within individual p_T subranges, the performance of our NN does not yet consistently surpass that of the legacyNN. Overall, a model trained on a broader set of input variables, such as those listed in table 2, yields an improvement of 8.8%, demonstrating that additional input features can improve the performance of the legacy NN.

When using a transformer neural network and the full set of available input features boosts the background rejection rate by 17.3 % compared to the LegacyNN when considering the entire p_T spectrum, despite the performance in each subregime still being lower.
However, the reason why the newly trained neural network still performs worse than the legacyNN when considering only subsections of the transverse momentum regime remains unknown. This is surprising as the new network uses the same input features in addition to new input features not present in the legacyNN. We would expect to have a performance at least equal to that of the legacyNN. Future studies could investigate this phenomenon further. Ideally, future iterations of this project will produce a NN that outperforms the legacyNN not only the overall spectrum but also across all p_T subregimes. Achieving this will likely require a combination of targeted training strategies, dedicated hyperparametertuning and changes in the networks architecture.

References

[1] F. Rubbo *Pileup and Jets at ATLAS*, https://indico.cern.ch/event/566346/contributions/2291217/attachments/1329741/1997774/jetMeetingAtBNL_20160831.pdf (2016).

A Appendix

A.1 DeepNN architecture

```
def __init__(self, lr=1e-3):
    super().__init__()
    self.save_hyperparameters()

    # Accumulators for epoch metrics
    self.train_correct_weight_sum = 0.0
    self.val_correct_weight_sum = 0.0

    # Model layers
    self.hidden1 = nn.Linear(len(var_train), 128)
    self.hidden2 = nn.Linear(128, 64)
    self.hidden3 = nn.Linear(64, 32)
    self.hidden4 = nn.Linear(32, 16)
    self.hidden5 = nn.Linear(16, 1)

    self.ReLU = nn.ReLU()
    self.sigmoid = nn.Sigmoid()

    self.lr = lr

#propagate through NN
def forward(self, x):
    x = self.hidden1(x)
    x = self.ReLU(x)

    x = self.hidden2(x)
    x = self.ReLU(x)

    x = self.hidden3(x)
    x = self.ReLU(x)

    x = self.hidden4(x)
    x = self.ReLU(x)

    x = self.hidden5(x)
    x = self.sigmoid(x)
    return x
```

A.2 TransformerNN architecture

```
def __init__(self,
             embedding_dim=128,
             n_heads=8,
             max_lr = 1e-3,
             warmup_steps = 5):
    super().__init__()
    self.save_hyperparameters() # saves init parameters to self.hparams

    # Project scalar features to embedding vectors
    self.input_projection = nn.Linear(1, embedding_dim)

    # Transformer Encoder Layer
    encoder_layer = nn.TransformerEncoderLayer(
        d_model=embedding_dim,
        nhead=n_heads,
        dim_feedforward=128,
        dropout=0.1,
        activation='relu',
        batch_first=True  # makes input shape (batch, seq, embed)
        )
    self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=4)

    # Pool transformer output (mean over sequence)
    self.pool = nn.AdaptiveAvgPool1d(1)

    # MLP stack after transformer
    self.linear_relu_stack = nn.Sequential(
        nn.Linear(embedding_dim, 128),
        nn.ReLU(),
        nn.Dropout(0.2),
        nn.Linear(128, 1024),
        nn.ReLU(),
        nn.Dropout(0.2),
        nn.Linear(1024, 128),
        nn.ReLU(),
        nn.Dropout(0.2),
        nn.Linear(128, 1),
        nn.Sigmoid()
        )

def forward(self, x):
    # x: shape (batch_size, n_features)
    x = x.unsqueeze(-1) # shape: (batch_size, n_features, 1)
```

```
        x = self.input_projection(x) # (batch_size, seq_len, embed_dim)
        x = self.transformer(x) # (batch_size, seq_len, embed_dim)

        # Mean pooling over the sequence dimension
        x = x.mean(dim=1) #(batch_size, embed_dim)

        logits = self.linear_relu_stack(x)
        return logits
```