Higgs Project

Nicholas Choma, Yiyun Hu, Yulin Shen

May 11, 2018

1 High-Level Problem

Exotic particles such as dark matter, particles with negative or complex mass, or Higgs bosons are sometimes produced in high-energy particle collisions. Typically these particles decay immediately, which only leaves us with the ability to detect secondary or later particles. Inferring which exotic particle was produced has historically been carried out using methods such as probabilistic inference with inspiration from the underlying physics. However, machine learning is gaining attention in this area and is sometimes able to outperform more traditional methods. Because interesting events in particle collisions are rare - only about one in one billion in the case of the Higgs boson - improving rates of detection can offer insights into the makeup and past and future of our universe.

In our project, we will use machine learning to perform a binary classification task, distinguishing between a signal process which produces Higgs bosons and a background process which does not.

2 Dataset

2.1 Specifics

The dataset we chose is the UCI repository 'HIGGS Data set'. There are 11 million samples with 28 features, and a class balance of 53% positive. Input features were standardized over the entire train/test set with mean zero and standard deviation one. Features with values strictly greater than zero were scaled so that their mean value was one [1].

The first 21 features are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes [1]. The decay products include leptons, jets and missing energy during the collisions. For both leptons and jets, there are three measurements to determine their momentum. The b-tag is used to distinguish between jets to see if the jet is consistent with b-quarks. Missing energy contains one measurement for momentum and one measurement for mass.

2.2 Generators

As nature does not provide the label for incoming particles, this data is generated by Monte Carlo simulation using first principles from physics. Moreover, although one of the most efficient colliders produces approximately 10^{11} collisions per hour, only approximately 300 of these collisions result in a Higgs boson [2]. Thus, experimental collaborations often rely on Monte-Carlo event generators.

Monte Carlo techniques are used to select all relevant variables according to the desired probability distributions, and thereby ensure (quasi-)randomness in the final events [3]. Ideally, the output of the simulation above has the same format as the real data recorded by the detector and can therefore be put through the same event reconstruction and physics analysis chain [4]. Since a vast number of events are generated, the data should match the real world if our subset contains enough instances.

3 Model Evaluation

Our primary method for evaluation was to measure the area under the receiver operator characteristics curve (ROC AUC). ROC AUC is common method for evaluating model performance on particle physics classification tasks [5]. A ROC curve plots the rate of false positives (FPR) versus the rate of true positives (TPR) for a binary classification task as the class decision threshold is varied. Thus a model which predicts classes completely randomly will have a ROC AUC score of 0.5, while a perfect model will have a score of 1.0.

4 Features Correlation and Engineering

Because the dataset is produced by Monte-Carlo simulation, it requires no preprocessing since it is clean. It is also standardized such that each feature is mean zero and variance one. Since we have a small number of features, we first plotted the most linearly correlated features to visualize our dataset and get a better intuition about how e.g. regularization might be affected.

We used the pandas library in python to calculate each feature pair's numerical linear correlation value and plot them in the figures. We found that almost all 21 low level features have low correlations. The high level features show a bit more correlation than the low level features, presumably since they are derivatives of the low level features. Of course should two features have a nonlinear relationship, this linear feature correlation will not help us to understand them. Figure 1 and 2 are example scatter plots of correlations.

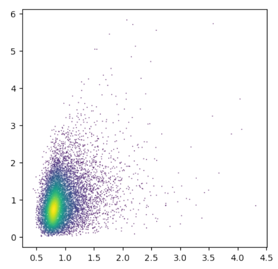

Figure 1: Correlation figure of m_wbb and m_wwbb, its numerical value is high (r=0.89).

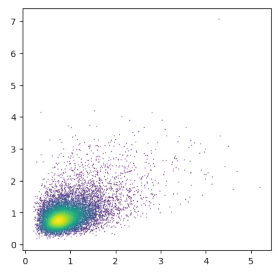

Figure 2: Correlation figure of jet2_pt and m_bb, its numerical value is lower (r=0.36).

Table 1: Baseline scores

Model	# training samples	Train ROC AUC	Test ROC AUC
Logistic regression	10 million	.685	.685
Random forest	10 million	.831	.819
KNN	1 million	.757	.623
BDT	1 million	.807	.805

5 Baseline Models

In the project, we implemented two levels of baseline algorithms. The first level serves just as a check that all our other models are working. The second level is to show that our more sophisticated models perform better than a simple baseline.

We have chosen to use logistic regression as the first level baseline algorithm. It is a basic linear model for solving binary classification tasks. We implemented it with the Scikit-Learn library, and since this model has good time complexity characteristics, we used 10 million training samples to quickly obtain our baseline. Additionally, we tried many different methods in this library such as random forests, boosted decision trees (BDTs) and KNN classification as a higher level of baseline models. Almost all of them perform better than the logistic regression model, KNN being the one exception. Note that the BDT and KNN models were trained on a smaller subset (1 million samples) of the data since they are not easily parallelized. A summary is presented in table 1 and details about each baseline model are presented in appendices A.1, A.2, A.3, and A.4.

6 Attempts at Improving Performance

6.1 Deep Neural Network

One of the first steps we took in building more sophisticated models was to replicate the existing best results. In [1], the best performance of 0.885 ROC AUC was obtained using a deep neural network with five layers and 300 hidden units per layer. This model was trained on 2.6 million examples and validated on 500,000 examples, and no hyperparameter tuning was performed in obtaining these results.

We followed what the paper described to build an identical model. Interestingly, we were unable to replicate the performance of the baseline using their exact architecture, though admittedly we did not incorporate the use of momentum directly in our optimizer, instead training with Adam. The first step for improving performance then was basic hyperparameter tuning. Our best performing model was able to achieve a ROC AUC score of 0.874, close to the baseline but still not quite as good. See appendix A.5 for further architecture details. While we are confident additional hyperparameter tuning would improve results, we chose several other paths for exploration as more in line with the goals of this project.

6.2 Graph Neural Networks

Motivation One of the interesting aspects of our dataset is the repetition of four 'jet' features, *jet * pt, jet * eta, jet * phi, jet * b-tag*, where $* \in \{1, 2, 3, 4\}$. These four jets are permutation invariant in the dataset (swapping jet 1 with jet 3 in a given sample should have no effect on our model's ability to make predictions). Thus using a deep neural network where no weights are shared between these features means using a much larger class of models than restricting our model to use the same weights for each of the four jets. If we could perform weight sharing, we would hope to reduce estimation error while keeping approximation error fixed.

Because these features live in a space which is non-Euclidean, we decided to consider each jet as a four-dimensional node in a computational graph. As such, we explored weight sharing between the nodes by using a graph neural network (GNN). GNNs have been gaining attention in recent years for their success on non-Euclidean datasets, using underlying geometry to perform convolutions on graphs in a manner similar to convolution neural networks on Euclidean domains (e.g. image and video datasets) [6].

Our network consisted of three broad stages, which are as follows:

Input & compute graph edges The input to the network is the four jets, each represented as a four-dimensional node. Required by the GNN is an adjacency matrix A which represents the pairwise edge weights between each of the four jets. For this we used a Gaussian kernel to compute pairwise distances d_{ij} between each x_i, x_j node, then applied a softmax function to each row of A to ensure equal proportion information spread to each node:

$$\begin{aligned} d_{ij} &= \exp(-\frac{1}{2}||x_i - x_j||^2/\sigma^2) \\ A_{ij} &= \frac{\exp(d_{ij})}{\sum_k \exp(d_{ik})} \end{aligned}$$

GNN layers Next the graph convolution is performed, where the input to each layer is a set of features $X^{(t)} \in \mathbf{R}^{nxd^{(t)}}$, and the output is a new set of features $X^{(t+1)} \in \mathbf{R}^{nxd^{(t+1)}}$:

$$\begin{aligned} X_{nlin} &= \texttt{ReLU}(\texttt{GConv}(X^{(t)})) \\ X_{lin} &= \texttt{GConv}(X^{(t)}) \\ X^{(t+1)} &= X_{nlin}||X_{nlin} \end{aligned}$$

Table 2: GNN vs DNN, trained on 4 jets

	# hidden	# layer	# parameters	AUC
GNN	64	6	50240	0.798
DNN	300	5	365100	0.801

where || is the concatenate operation. Graph convolution is performed in the following way:

$$\begin{aligned}\texttt{Spread}(X^{(t)}) &= AX^{(t)}||IX^{(t)} \\ \texttt{GConv}(X^{(t)}) &= \texttt{Spread}(X^{(t)})\theta_w^{(t)} + \theta_b^{(t)}\end{aligned}$$

where I is the identity matrix and θ_w, θ_b are the model parameters.

Readout Finally, the features are pooled using a summing operation over the four nodes and logistic regression is performed to output a final prediction:

$$\begin{aligned}X_k^{(end)} &= \sum_j X_{jk}^{(end-1)} \\ y_{pred} &= \texttt{Sigmoid}(X^{(end)T}\theta_w^{(end)} + \theta_b^{(end)})\end{aligned}$$

Results Our hope was to improve over performance over a deep neural network with input restricted to the four jets. If we sew an improvement, we would have reason to believe the graph neural network learned useful features. In this case, we could combine the GNN features from just before the final readout layer with the original input features as a form of feature engineering. Then using this new set of features with e.g. a deep neural network, we should expect to see improved performance (since again, the GNN may have found useful features not perceived by the deep neural network).

Unfortunately, as seen in table 2, the GNN learns to roughly the level of performance as the deep network. It is encouraging that the deep network is able to achieve the same performance with about 15% as many parameters, but nevertheless, we did not see improved performance and so did not pursue the GNN approach further.

6.3 Bagging

Motivation At inference time, each sample is given a prediction score $\in [0, 1]$ which we hope will be close to the true prediction $\in \{0, 1\}$. However, with respect to our metric of ROC AUC, these predictions are not independent of each other.

In fact, what matters for ROC AUC is not the absolute prediction on a given sample, but its rank compared with the other predictions. As a simple thought experiment, consider the following: If all samples with true class 1 are predicted as 0.501, and all samples with true class 0 are predicted as 0.499, then the ROC

Table 3: Bagged tree performance

Model	Change in sorted order (%)	ROC AUC
Decision tree	15	.760
Random forest	7	.778

Table 4: Bagged deep neural network performance

Model	Change in sorted order (%)	ROC AUC
Deep network	13	.875
Bagged deep network	5	.884

AUC score is a perfect 1. If however these predictions are reversed so class 1 samples are predicted as 0.499 and class 0 samples as 0.501, then the ROC AUC is 0 while the model loss is roughly the same as in the former case.

Thus our hypothesis was if we could improve stability in the order in which sample predictions are ranked (or sorted), we may see an improvement in the ROC AUC score.

Method We chose to explore bagging because of the relation of random forests to decision trees, where random forests are able to reduce the variance in predictions compared to decision trees, which are high-variance estimators. We believed two trained random forest models would offer more stable sorted predictions than two trained decision trees, and in fact this is what we observed. Table 3 shows how ROC AUC improves when using random forests compared with decision trees, and this correlates with the order stability of each model's predictions. See Appendix A.2 for details on model architecture.

We then extended this thought process to deep neural networks, comparing the predictions' sorted order of two regular DNNs to the predictions' sorted order of bagged DNNs, where a bagged DNN simply averages over the predictions of k trained DNNs (in our testing we used $k = 5$). We again observed that the predictions' sorted order is more stable with bagging and that ROC AUC improves, as in table 4. See Appendix A.5 for details on model architecture.

Further thoughts In our testing, we used deep neural networks with 300 hidden units per layer. One natural question to ask is what benefit bagging k neural networks together would provide that a wider single network with an equivalent number of hidden units does not. We believe the difference to be that for a single network on a given sample, only one path within that network must be active for good results in training. Thus if that path is broken during test time, the entire network is unable to make a good prediction. However, because we train our k bagged models independently, at least k paths must be found during training, and chances are better that at least one will remain during test time. This is not a proof of course, but we believe this is similar to the reason dropout performs well in practice.

7 Challenges and Insights

7.1 Parallelization

One of the significant challenges for us was to consider how we could take advantage of parallelization, given our dataset size of 11 million samples. In order to efficiently train, we both considered working only with models which are amenable to parallelization - with the exception of the boosted decision tree, used as a baseline against one of our key references [1] - and to consider the exact method for parallelization we would use.

As such, we gained exposure to high performance computing via the SLURM environment, both training on multi-node CPU branches for the baseline models, and on GPU for the neural network architecture.

7.2 Parameter tuning

A significant challenge for us was to quickly and efficiently search through hyperparameter space in our model architectures. We started by using an informed random walk procedure, noting which hyperparameters gave the best results, and randomly deviating from them by small steps. However if we were to retrace our steps, we believe choosing randomly from a pre-defined grid of hyperparameters could lead us to explore spaces we wouldn't otherwise reach, thus potentially giving us better results.

8 Future Directions

8.1 Further Bagging Ideas

There are several extensions to our bagging methods we might try, which are as follows:

1. Weak learners. We currently are training large deep neural network models for performing bagging, but this may be excessive. An experiment to try would decrease the size of each bagged model, and perhaps increase the number of bagged models.

2. Currently we perform bagging using models of the same type, averaged together. A possible extension would be to bag several models of different types, perhaps using a more sophisticated weighting scheme than averaging.

3. While we don't expect dropout to be of great help since our models do not overfit, we feel it would be necessary to compare our bagging approach to dropout, which we believe operates in a similar fashion.

4. Finally, we are currently using the deep network's parameter initialization as our source of randomness in bagging. There are many other sources to

choose from which may give better performance, including feature selection, bootstrapping training sets, etc.

8.2 GNN

While the GNN model did not show enormous potential in our testing, it is still possible we could improve performance through this method. An easy test would be to compare the deep neural network to the graph neural network such that both architectures use the same number of parameters (say 50240 to match the GNN). If the deep neural network is unable to match the GNN's performance, then perhaps using the GNN as a preprocessing stage to extract features, we could greatly reduce the complexity of the deep neural network which is then trained on all features plus those from the GNN.

8.3 Deep neural network optimization

As nonlinear optimization theory seems to lag behind practice, we generally chose to direct our efforts toward more principled improvements. However, there could still be lots of potential for improving performance through the use of different optimization strategies. Two that we would immediately explore are the incorporation of a momentum term into an SGD optimizer, and a move to smaller minibatch size. We would try momentum with SGD since our main reference paper [1] used it with good results. We would try different minibatch sizes since ours was obtained based upon GPU utilization (i.e., we chose the smallest minibatch size that would make full use of the GPU). This led to quicker training, but perhaps a smaller minibatch size would lead to noisier gradient estimates, which could potentially take us into better areas of the optimization energy landscape.

References

[1] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

[2] Johan Alwall, Michel Herquet, Fabio Maltoni, Olivier Mattelaer, and Tim Stelzer. Madgraph 5: going beyond. *Journal of High Energy Physics*, 2011(6):128, 2011.

[3] Torbjörn Sjöstrand, Stephen Mrenna, and Peter Skands. Pythia 6.4 physics and manual. *Journal of High Energy Physics*, 2006(05):026, 2006.

[4] Séverine Ovyn, Xavier Rouby, and Vincent Lemaitre. Delphes, a framework for fast simulation of a generic collider experiment. *arXiv preprint arXiv:0903.2225*, 2009.

[5] Gilles Louppe, Kyunghyun Cho, Cyril Becot, and Kyle Cranmer. Qcd-aware recursive neural networks for jet physics. *arXiv preprint arXiv:1702.00748*, 2017.

[6] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

Appendix A Model Hyperparameters

A.1 Logistic Regression

For Logistic Regression Classification, we used LogisticRegression in Scikit-Learn library, tested for parameter settings. For this model, we set a tolerance for stopping criteria of 1e-5, a inverse of regularization strength of 1000, balanced class weights and stochastic average Ggradient descent solver.

A.2 Random Forest

For Random Forest Classification, we used RandomForestClassifier in Scikit-Learn library. For this model, we set the number of trees to be 100, the Gini impurity test, max tree depth, the minimum number of samples required to split an internal node to be 300, the minimum number of samples required to be at a leaf node to be 150, the minimum weighted fraction of the sum total of weights to be 0.00003 and balanced class weights.

A.3 K Nearest Neighbours

For KNN Classification, we used KNeighborsClassifier in Scikit-Learn library, tested for parameter settings. For this model, we set 20 CPU simultaneously to train the model, but it is still very slow for this large dataset. We set K as 5.

A.4 Boosted Decision Tree

For Boosted Decision Trees (BDTs), we used GradientBoostingClassifier in Scikit-Learn library. For this model, we used exponential loss function, a learning rate of 0.01, the number of boosting stages to perform to be 1000, the minimum number of samples required to split an internal node to be 30, the minimum number of samples required to be at a leaf node to be 10, the minimum weighted fraction of the sum total of weights to be 0.01, and the max depth to be 5. The reason it has a worse performance than Random Forest Classification and KNN is that we only used a 1 million sample subset of our data for training the model. This is because it will take more than two weeks to run the BDTs on the whole training dataset.

A.5 Deep Neural Network

We used Tanh activation function, three hidden layers with 150 hidden Tanh units each, a learning rate of 0.0005, a minibatch of 2000, a learning rate decay factor of 0.995, and the number of epoch is 1500. Training was stopped early based upon performance from a validation set.

A.6 Graph Neural Network

Our graph neural network was trained with 64 hidden feature units, 6 graph convolution layers, a learning rate of 0.005, learning rate decay of 0.95, ReLU activation units, and for 2000 epochs with a minibatch size of 2000. Training was stopped early based upon performance from a validation set.