## **Introduction**

Through the observation of exotic particles via high-energy colliders, physicists are able to learn valuable insights about the nature of the material world. Exotic particles are created in particle accelerators, which collide protons and/or antiprotons to create new particles that only occur at extremely high energy density. Most particle collisions do not produce exotic particles and detecting them is challenging, which makes the data available limited and expensive. Adequate detection methods must be robust to detecting a particle of interest (signal) from a collision versus other particles that may be spawned (background). In the past “shallow” classifiers, such as feed-forward neural networks with a single hidden layer, or boosted-decision trees were primarily used to detect exotic particles. The data produced from the collider experiments is fairly high dimensional, but researchers would often derive new high-level features from the raw features in order to better train their models. These new features were often non-linear in nature and relied on subject matter expertise and labor-intensive implementation. Ideally, more robust modeling approaches could automatically find discriminating relationships of the features in the raw data, which would obviate the need for additional costly human intervention [1].

In this case study, we build a replica of the network used in *Searching for Exotic Particles in High-Energy Physics with Deep Learning* using the same architecture and hyper-parameters of the original authors. For training, we used the HIGGS boson simulation data provided by the authors to fit our own network weights. Using a similar architecture and training data, we show that we are able to reproduce results with an AUC of 0.84 compared the max of 0.88 achieved in the original paper. Additionally, in the future work section, we discuss the original network and contrast it to other standard practices common to these tasks now, 5 years after the original publication.

## **Background**

The authors of *Searching for Exotic Particles in High-Energy Physics with Deep Learning* sought new methods in deep learning, which at the time of the publication had become more accessible based on advancements in the algorithm and technology. These advancements included optimizing stochastic gradient descent to run on graphics processors, the ability to train on large data sets, and new learning algorithms which included dropout and pre-training with autoencoders. Through deep learning, the authors demonstrated that simpler techniques, even when informed with manually created features, fail to capture all of the available information. The attempt to prove that not only will more robust deep learning methods perform better than the previous methods, but they will not require manual feature engineering to be effective. [1]

The one of the largest differentiators between deep learning models and more traditional shallow methods is their ability to automate the feature engineering portion of the problem. It is able to achieve this through the stacking of layers that are able to extract representations of data that are often times more complex and more useful to the task at hand. These representations are learned during the model training process by adjusting the layer weights or parameters to minimize a loss function. This process is referred to as gradient based optimization. While a thorough explanation of this process is outside the scope of the assignment, a high-level understanding is important to understand how deep learning develops these data representations. Using gradient based optimization, we can efficiently adjust our layer weights using the gradient of the loss function with respect to the network coefficients. The weights are adjusted in batches until a minimum loss on the training data is achieved.

The data we are using for this assignment is the data published with the original study. The data was produced using Monte Carlo simulations representing the kinematic properties measured by the particle detectors in the accelerator. There are 21 raw features (low-level) in the data as well as seven features which are functions of the first 21 features (high-level features). The 7 features were derived by subject matter experts for use in shallow machine learning applications. The target variable in this study is a binary class variable that indicates whether the observation was a signal from a collision versus other background particles that may be spawned.

## **Method**

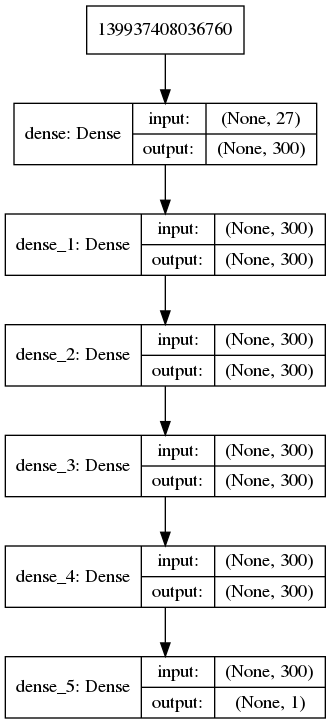
## In order to build a replica of the best performing neural network described in the paper, we utilize TensorFlow and keras. These packages allow us to quickly frame the network topography in a high-level API. The original authors performed a hyper-parameter search across hyper-parameters shown in Table 1. They used 2.6 million training examples and 100,000 validation examples. The authors admit that this is not an exhaustive search, because they were limited by compute resources. They settle on their final model to consist of 5 layers with 300 hidden units in each layer, a learning rate of 0.05, and a decay coefficient of 1 x .

**Table 1:** Hyper-parameter Choices for Deep Network

## 

## We constructed a similar deep neural network trained using the first 10.5M data points from and evaluated using the last 500,000 observations. These data points replicate what the authors used for training and testing. We also implemented the same stochastic gradient descent optimizer with identical hyper-parameters which we described above. The input to our model are the 27 features (both low-level & high-level features) which feed into 5 layers with 300 hidden units with tanh activation. The final layer condenses our hidden layers down to the our binary target prediction using sigmoid activation. The final architecture of our model is shown in Figure 1 below.

**Figure 1:** Trained Model Layers

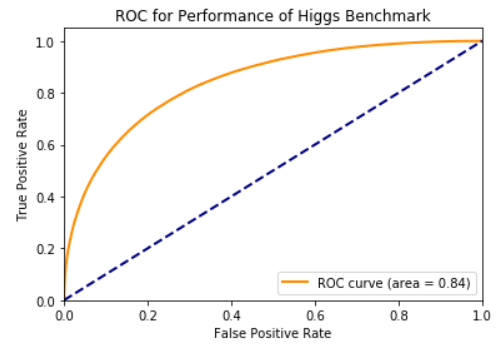


## **Results**

We replicated the original paper’s model in architecture: 5 layers with 300 hidden units in each layer, a learning rate of 0.05, and a decay coefficient of 1 x . To get as close to the original as possible, we used the tanh activation function, and we ran 100 epochs with a batch size of 5000. While this is fewer epochs that indicated in the paper (200-1000 were mentioned), and larger batch sizes (100 was used in the original), we were limited by the time constraints of the computing power to exactly replicate. These results were only reasonably achieved due to GPU computing capabilities available to our team.

In order to evaluate our results, we plotted the ROC curve, and output the calculation of the area under the curve (AUC), as seen in Figure 2. The best results that were found by the original authors were in the 0.800 to 0.885 range, depending on the level of features included in the model. Our model fell right between these values, and that difference may be attributed to the difference in the number and size of the training epochs used.

**Figure 2:** Results of Initial Duplicated Model



## **Conclusions**

Using the low level & high-level features described above, we were able to produce comparable results to those from the original paper. Our network was able to achieve a max AUC score of 0.84 on our test set using a 5 layer network without dropout. The original authors best performing model on the same features with dropout on the final layer achieved an AUC of 0.88. In order to quantify our ability to reproduce the original results, we would need to establish some sort of confidence interval for our results. One method for computing the confidence in our results would be to calculate a bootstrap CI of our results against the test set. This is achieved by taking several samples with replacements from the test set and computing our AUC on those samples using our model. Then using the resulting scores we take the 2.5th and 97.5th percentile to be our 95% confidence interval of our AUC score. Unfortunately, we have limited compute resources and unable to perform this test. However, if we were able to bootstrap our confidence interval if the papers AUC of 0.88 fell within our computed range we would consider the results replicated. One limitation of the process is that is does not take into account the correlated nature of our comparisons as both results come from the same test set. One method to overcome this would be through the use of DeLong’s method for comparing 2 or more ROC curves [5]. This is a non-parametric approach to estimating the AUC using Mann Whitney’s U-statistic. This method has become a common approach for comparing ROC curves for results on the same observations.

## **Future Work**

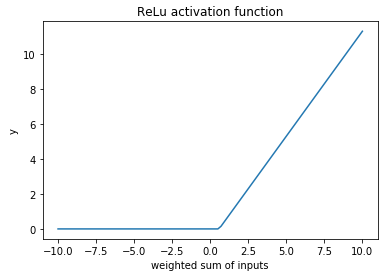
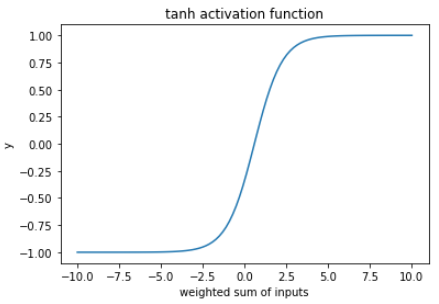
## The first area of interest for future work would be to perform a more exhaustive search across the model hyper-parameters proposed by the authors as well as other hyper-parameters not considered - including the selection of an activation function at each layer. The authors cite computational barriers on the hardware they ran their analysis, which consisted of 16 Intel Xeon cores, an NVIDIA Tesla C2070 graphics processor, and 64 GB of memory. Larger compute capacity at cheaper rates than available at the time of the paper in 2014 is available and easily accessible from cloud computing providers.

While we were able to produce a comparable network to the original authors, we would ideally like to reproduce this work using methods more commonly used now, beginning with the use an updated optimizer. Based on discussions, it is common practice to utilize an RMSProp or Adam optimizer for determining network weights. Adam is a stochastic gradient-based optimization algorithm that improves the efficiency of our learning process when adjusting the weights of our kernels throughout the network. Adam is ideal for high-dimensional learning problems like ours and has been shown to require very little memory. Adam is shown to perform most closely to RMSProp it was observed to have a lower training cost on a variety of tasks [3].

Additionally, the authors use dropout at the final layer to help normalize the results and reduce overfitting. However, a more commonly used approach for normalization is batch normalization. Batch normalization is able to better handle the shifts in variance between training and testing sets that dropout has known to suffer from while also speeding up the training time [4]. This normalization process would allow us to make a deeper network without overfitting the data.

The authors state that they use the tanh activation function with their neural network, however ReLu activation has taken favor for most neural networks recently. The tanh activation function (shown on the left in Figure 3) is similar to the logistic (sigmoid) activation function, except that it is zero-centered. However, it shares a similar characteristic called saturation, which leads to the problem known as the vanishing gradient. A neuron is said to be saturated if it reaches its peak value - either maximum or minimum. For example, if the resulting activation of a given node reaches 1, then its derivative is 0, so a pass via backwards propagation would not update that node’s weight. The vanishing gradient problem causes nodes in a network to refuse to learn or to learn at a very small rate. Furthermore, logistic (sigmoid) and tanh activation functions both have terms in their formulae thus making convergence slower versus simpler functions. More recent deep learning networks are trained using the ReLu (Rectified Linear Units) activation function largely because of its simplicity. The ReLu activation (shown on the right in Figure 3) returns 0 for any negative number, otherwise if the input is positive, it returns itself. This function does not saturate for the positive value of the weighted sum of inputs and thus is robust to the vanishing gradient problem. It also does not include any quadratic terms and therefore is simpler to compute via forward and backward propagation, which leads to efficiencies in training time [2].

**Figure 3:** Tanh (Hyperbolic Tangent) Activation Function Compared to a ReLu Activation



## Finally, we might also compare the differences in implementations and optimization compute times between the libraries used by the authors, including Theano and Pylearn2, versus more modern deep learning libraries such as TensorFlow and keras or PyTorch.

## **References**

1.<https://arxiv.org/pdf/1402.4735.pdf>

2.<https://towardsdatascience.com/analyzing-different-types-of-activation-functions-in-neural-networks-which-one-to-prefer-e11649256209>

3. Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

4. Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

5. Delong, E R et al. “Comparing the Areas Under Two or More Correlated Receiver Operating Characteristic Curves: a Nonparametric Approach.” Biometrics 44.3 (1988): 837–845. Web.

## **Appendix - Code**