Higgs Machine Learning Challenge with Neural Network Adversarial Training

Anonymous Author(s)
Affiliation
Address
email

Abstract

The Higgs particle was theorized in 1964 and finally discovered in 2012 by the ATLAS and CMS experiments at CERN. This particle is the keystone of the Standard Model of particle physics, and provides the mechanism by which other particles acquire mass. Such discoveries are statistical in nature, and require vast amounts of information from the debris of particle collisions. Sifting and categorizing this data is a continual challenge, and in 2014 CERN and Kaggle invited the public to try their hand at such work in the ”Higgs Machine Learning Challenge”. A number of top entries used neural networks, including the winner. An extended approach using the so-called ”adversarial training” technique is presented herein.

1 Introduction

Alongside the well-known types of material and force-carrying particles, the Standard Model of particle physics has an altogether different type of entity, which was theorized to exist in order to explain why certain particles have mass. This came to be known as the Higgs particle, named after the author of the groundbreaking 1964 paper by one of the pioneers of this field [**?**]. So began a search that spanned nearly four decades, culminating with the discovery of the Higgs particle in 2012 by the ATLAS and CMS experiments, which operate at the Large Hadron Collider (LHC) at the CERN lab in Geneva [**?**]. The tell-tale signals were given by the physics properties of the Higgs particle’s decay products, as the particle itself cannot be directly observed due to decaying so rapidly that it cannot travel out of its creation region and into the detector.

First and foremost for this discovery to happen was the particle collision energy, and second was the ability of the detectors to record and analyze data. Every particle has a characteristic mass energy, and this energy must be met by a device such as the LHC in order for the particle to be created. The Higgs particle happens to be a relatively massive particle compared to a number of others, and therefore requires a relatively large amount of energy. A technological evolution over many years and many generations of device was necessary to develop the LHC, which is powerful enough to provide the Higgs creation energy.

Once created, a high mass particle such as the Higgs rapidly decays into other, lighter particles, which may themselves decay further. A variety of these decay products that are detected and categorized by such properties as momentum and electric charge, based on their trajectories through the machine. This data is then used to infer the ancestral particle at the initial creation point.

Despite the LHC providing an appropriate energy range for the creation of Higgs particles, their production is still exceedingly rare compared with the production of a number of other particles. An exceedingly large number of creation events is therefore required to compensate for the rarity, and the particle tracks simply cannot be assessed individually. The vast stream of electronic readout must be dealt with computationally and statistically, and a great deal of time and effort is spent by scientists determining how best to seek the signals of interest.

Machine learning has inevitably become invaluable for this task, which is simply one of classification, namely a question of signal versus background. In the case of the Higgs particle, its various decay channels are not unique, and are in fact shared by the decays of other heavy particles. Seeing a certain set of decay products does not then give indisputable proof of one particular ancestor particle or the other. But there are nonetheless subtle signs within the data when viewed *en masse*, which machine learning algorithms can be trained to recognize.

2 Higgs Machine Learning Challenge

In 2014, CERN scientists collaborated with the Kaggle machine learning organization, developing an open competition for the public to try and develop a machine learning classifier to distinguish the Higgs particle from other heavy particles in the tau-tau decay channel [**?**]. This was the Higgs Machine Learning Challenge (HMLC), which at the time of its release was the most popular Kaggle challenge up to that point, with a total of 35,772 uploaded entries submitted by 1,785 teams (1,942 people) comprised of participants from around the world, some working within the physics community, others simply interested in computing [**?**]. The winning entry came from a sole programmer with no advanced physics training, who developed a neural network model that was clearly superior to anything else [**?**].

The HMLC has two datasets - one fully labelled dataset for training, and another unlabelled dataset for testing. The training dataset has information about 250,000 labelled events, and the testing dataset corresponds to 550,000 unlabelled events. The labels are simply ”signal” and ”background”. In a particle physics collider, two input particles (protons in the case of the LHC for ATLAS and CMS experiments) are accelerated and collided at high speed, resulting in a high energy state that can form any particle which ”costs” that amount of energy or less. Call this created particle the *initial state*. This initial state often has a number of possible decay channels to some set of *final state* particles, and some initial states share similar final states. In the case of the Higgs particle, one decay channel is that to a pair of lighter tau particles, with these then decaying to yet lighter particles still. These events are those labelled as ”signal”. Such final states are possible from the decay of other initial states such as W and Z particles (which are the carriers of the weak force) and top particles (which are the heaviest of the quark family) [**?**]. These events are those labelled as ”background”.

It is the subtle differences in the ensemble physics variables that we seek to distinguish through machine learning. Both training and testing datasets are defined by a feature set comprised of 30 different physics variables, representing quantities such as the mass, energy and momentum of decay products that entered the detector or were otherwise inferred [**?**, **?**]. These entities are either (1) electrons, (2) heavier variants of electrons (muons and taus), (3) jets of particles comprised of quarks, and (4) so-called missing energy, representing very light and weakly interacting particles called neutrinos [**?**].

Physics details aside, this format of problem is ubiquitous in machine learning, and requires no knowledge or understanding of what the features actually are. The HMLC dataset is actually a vastly simplified version of Monte Carlo simulated data used by CERN physicists. It was carefully tailored to be simply enough for public use, but still interesting enough to promote useful machine learning techniques [**?**]. The signal to background ratio in the HMLC dataset is much larger than that in real data, which requires many millions of events in order to provide statistically significant findings, so being impractical for the HMLC.

3 HMLC Winner and Initial Impulse

The HMLC winning entry was based on a neural network with 3 layers of 600 nodes each, finished with 2 softmax output units (one for signal, one for background) [**?**]. Some feature preprocessing was performed on the input data, normalizing each to have zero mean and unit standard deviation, with some also being log-transformed. Further to the given features, four more were derived from some of the given angular features, and five more were derived from some of the given mass features, so giving nine extra hybrid quantities [**?**]. Cross-entropy minimisation was used for training, with a correctness probability then assigned to each classification, and with only those above a certain threshold retaining their determined class labels at any one iteration [**?**]. Further to this, a range of model variants were used simultaneously via *bagging*.

All these elements of the winning entry were included for good reasons. The effects of each were carefully examined over a four month development period, along with many others that were found to be of no benefit and discarded. The speed with which computer science is evolving means that even now, little more than a year after the close of the original Kaggle competition, there are new things to try that were not part of the winning formula. Soon after the start of this current project, the Kaggle winner was contacted for advice based on their past experience and their understanding of the machine learning field in general. They proposed that a generative approach could be applicable to this task, and so initial research was then undertaken in this direction, and became the seed of this project's central concept.

4 Adversarial Training

The generative approach originally suggested by the Kaggle winner regarded Generative Adversarial Networks (GANs), where the idea is to train not only one machine learning system, but two [**?**]. One system trys to discriminate the classes of a given set of training data, and the other system tries to generate new data examples that can fool the former system into passing them as real. The discriminator *D* is reinforced for every example it correctly classifies, with the generator *G* being reinforced for every example it makes which passes off as real. In practice this can be done by setting up *D* and *G* as multilayer perceptrons using backpropagation and dropout, with the reinforcement criteria set up as a competitive "minimax" game between the two systems, hence the "adversarial" in the GAN acronym. Although the original GAN work was intended to create an adept generator, the fact is that both *G* and *D* become strong together, and it is the discriminator that is of interest for the HMLC.

However, the short timeframe of this project required rapid evaluation of feasibility, and the GAN approach, although interesting and promoted by the Kaggle winner, seemed to present a challenge in itself to come anywhere near putting into practice as needed. Thankfully, the authors of the original GAN paper responded swiftly to a request for further advice, and a much simplified adversarial approach was suggested.

The full GAN concept described above is still useful to understand the adversarial process in general, namely having one part of a system increase the challenge for another part that we wish to strengthen. Neither opponent in the adversarial system is aware of the other, as such, but rather each component simply affects the other's error minimisation, effectively supplying an additional regularization term. To simplify matters, we can instead begin with some discriminator and modify its error function to emulate the regularization effect of a sophisticated generator.

In practice this is still done by use of training examples that are not in the supplied training data, just not in the same way as the GAN approach. The simpler method generates new examples by slightly perturbing actual training data, ideally on the order of the data noise level, if known. The perturbed data should ideally be classified in exactly the same way as the unperturbed data. This is an economical way of generating high quality adversarial examples, of the type that an actual generator aims to produce from scratch, namely within the data noise limit and, therefore, hopefully indistinguishable from the real thing.

A simple neural network based on a multilayer perceptron works well for this procedure. In this case the error function is present as an update to the network weights with every training example, with the examples assessed in turn for a pre-defined number of epochs. The weights are randomly initialized via a normal distribution with zero mean. Feeding a training example through the network leads to a set of errors at the output layer, which are backpropagated to obtain adjustment derivatives for each layer of weights.

Unlike the usual case, for simplified adversarial training the weight update does not occur at this stage in the algorithm. The error is instead backpropagated once more, giving adjustment derivatives for the input layer itself. The *signs* of these input layer derivatives are taken and multiplied by some constant, with the result then added to the input to produce the perturbed, adversarial, counterpart input. The constant involved is ideally on the order of the data noise level, or at least small enough to not perturb the real training example too much. This procedure is referred to as the ”fast gradient sign method”.

The adversarial example is then fed forward through the same network (i.e. with the same weights) as used for the real training example. Once again the output errors are calculated and backpropagated, giving a second set of adjustment derivatives for each layer of weights. We then have one set of derivatives for each layer of weights from the real training example, and another set from the adversarial training example. Each set describes how much the network must be corrected to cater for the example in question. If the network treats both the training example and its adversarial counterpart the same, which is the intention, then the two sets of derivatives should be the same. If this is not the case then the two sets of derivatives will differ. By updating the weights by the average of the two sets of derivatives, the corrected network should not only become more adept in general, but also become more likely to treat further examples and their adversarial partners more equally.

The above can be mathematically summarized via the following modified cost function, in which J is the original cost function, $\mathbf{\Theta}$ is the model parameters, $\mathbf{x}$ is the input vector, y is the target label, α is the mixing parameter (set to 0.5 for equal treatment of real and adversarial training examples), and ϵ is the perturbation constant (ideally on the order of the data noise, as mentioned):

$$\tilde{J}(\mathbf{\Theta}, \mathbf{x}, y) = \alpha J(\mathbf{\Theta}, \mathbf{x}, y) + (1-\alpha) J(\mathbf{\Theta}, \mathbf{x} + \epsilon \text{sign}(\nabla_{\mathbf{x}} J(\mathbf{\Theta}, \mathbf{x}, y)), y)$$

5 Baseline Algorithms

In order to observe the effects of the chosen adversarial training approach, and to observe the evolution of the implementation, a number of baselines were created. First was the adaptation of two existing classifier codes, one being driven by a Support Vector Machine (SVM) algorithm and the other by a Kernelized Perceptron (KP) algorithm. The former operates as part of the Python SciKit-Learn framework, and the latter was in fact based on the MatLab code developed as part of SFU CMPT419/726 assignment 2, but translated into Python in order to be run outside of the MatLab environment. This was soon found to be necessary for the sake of running on significant fractions of the given training data, which is large enough to give memory management issues when running on personal machines, as so with the license-restricted MatLab suite. Given the relative simplicity of the HMLC dataset, there was nothing to suggest that these relatively simple SVM and KP approaches would be invalid, and indeed they performed as desired.

5.1 SVM

As SciKit-Learn has a mature SVM implementation in sklearn.svm.SVC, it was a clear choice to use with the HMLC dataset. The SVC class supports several kernel functions by default, each with its own set of parameters. To find optimal parameters, a 5-fold cross-validation was performed using sklearn.grid_search. To facilitate this parameter tuning, grid search was run on a 50,000 example subset of the entire dataset. The following table lists the kernels and parameters tried during the grid search. Note that γ and C are ranges. For C the range is exponents, while the gamma values are spaced evenly on a log scale, starting and ending with the values in the table.

kernel	C	γ	coef0	degree
radial-basis function	$10 - 100,000$	0.00001 $-$ 100,000	N/A	N/A
sigmoid	$10 - 100,000$	0.001 $- 10,000$	$0, 1$	N/A
polynomial	$10 - 100,000$	0.001 $- 10,000$	0,1	$1, 2, 3$

Having chosen an optimal kernel and parameters, the entire training set of 250,000 examples was randomly split into 225,000 training samples, and 25,000 validation samples. The SVM was trained using a radial-basis function (RBF) kernel, with parameters $C = 1,000$ and $\gamma =$ 0.00024883895413685354. The overall score obtained was 73.744%. For the signal class, precision was 83 percent, and recall was 30 percent. For the background class, precision was 72%, while recall was 97%.

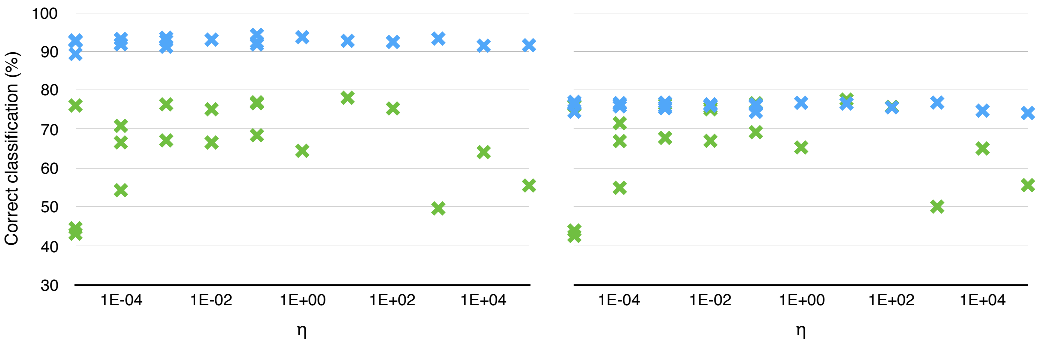

Figure 1: *Kernelized perceptron performance*
Left: Training data; Right: Testing data
Blue: Un-normalized; Green: Normalized

5.2 Kernelized Perceptron

The kernelized perceptron algorithm has the ability to evaluate a range of kernel parameters through cross-validation in order to find the optimum for a given kernel function. However, in order to have a direct comparison with the SVM, it was chosen to use the RBF kernel with the γ-parameter value found to be optimum for this kernel with the SVM (the C-parameter mentioned above is specific to the SciKit-Learn implementation of the RBF, and so not relevant for the MatLab kernelized perceptron). A scan of the gradient descent step size η was performed, first with un-normalized data and then with normalized data. Three such runs were done for each value in the range $10^{-5} \leq \eta \leq 10^{-1}$, but due to time restrictions only one such run was done for each value in the range $10^{0} \leq \eta \leq 10^{5}$, and the results are shown in Figure 1. Only 10% of the full 250,000 example HMLC training data was used here, with 20,000 examples used for training and 5,000 examples used for testing. All runs were done on a single node of the SFU High Energy Physics (HEP) group computing cluster.

It can be seen that there is similar performance for both the training and testing data when normalized, and although performance is generally improved in both cases for un-normalized data, the improvement is more significant for the training set than the testing set. These observations indicate that normalization is not only a hindrance for this algorithm, but that it is somehow affecting the expected performance, which should typically be better for training data compared to testing data. Furthermore, there seems to be a greater spread in the performance values of normalized data, showing lower stability with respect to the stochastic procedure. Although not conclusive due to relatively low statistics, it could be that there is a peaked trend in the performance for normalized data, where the average seems to tend to a maximum around $\eta = 1$. This may also be true of the un-normalized data, but the effect is much more subtle and would require more work to assess properly. The best performance on un-normalized testing data was 76.98% for $\eta = 10^{-5}$, and the best performance on normalized testing data was a slightly larger 77.54% for $\eta = 1$, but, for the reasons discussed, this latter value is not as impressive in light of the general negative aspects of normalization seen here.

6 Main Algorithms

With the intention of developing a neural network with adversarial training in mind, a logical first step was to modify an existing neural network already in hand. Once again the SFU CMPT419/726 assignments came in useful here, with the third and final assignment involving the completion of a simple MatLab neural network for handwritten digit recognition. This code was adapted for use with the HMLC dataset, and given some further refinements such as optional training data normalization. Satisfied that the code worked as intended in this state, the adversarial components were then introduced. These additions sit entirely within the inner calculation loop, following the existing error backpropagation code lines but before the weight update step, as described in §4. By simply setting the perturbation size ϵ to zero, the adversarial calculations are rendered null and void, and the code performs the same as prior to their inclusion, as should be so.

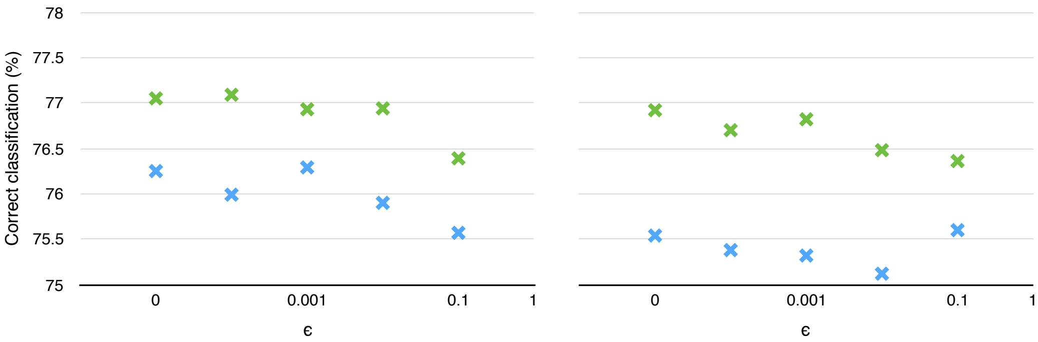

Figure 2: *MatLab neural network performance*
Left: Training data; Right: Testing data
Blue: Un-normalized; Green: Normalized

Due to time restrictions, this MatLab code was unfortunately not translated into a more general language such as Python, which, as mentioned in §5, would have allowed ease of use of a greater fraction of training data. However, in the interests of time even the kernelized perceptron was run on only 10% of the full HMLC training data, as mentioned in §**??**, despite being able to handle larger datasets than its MatLab version. The MatLab neural network could comfortably be run in the same way, so with 20,000 examples for training and 5,000 examples for testing. The weight update step size for these tests was maintained at a constant 10^{-5}, with the perturbation size ϵ scanned in the range $0 \leq \epsilon \leq 10^{-1}$. The level of adversarial training is directly proportional to ϵ, so with $\epsilon = 0$ essentially being the same as having no adversarial component in the code at all.

Figure 2 shows the results of this testing. Like the kernelized perceptron, the MatLab neural network displays similar values for both training and testing data when normalized, but with the former noticably greater than the latter, which was not so with the kernelized perceptron results. The values for the normalized case here are similar to those towards the upper performance range of the normalized KP, but the best normalized KP values are greater than those found here. However, there is only a relatively slight variation with ϵ, to be discussed shortly, whereas the normalized KP values have much greater spread overall. So regardless of the better performance coming from the normalized KP rather than the normalized neural network, the latter code gives the impression of being more stable and consistent.

The most stark contrast between the KP and MatLab neural network is in going from normalized to un-normalized data. Normalization was deemed to be a hindrance for the KP, but it seems to be beneficial for the MatLab neural network. Not only is performace better in the normalized case for the training data, but the relative gains for the testing data seem greater still i.e. there is a greater jump from un-normalized to normalized for the testing data than the training data. Exactly why this is the case is unknown, so without going into speculation this will have to remain a point for further investigation.

Perhaps the most crucial aspect of these tests is the effect of the adversarial perturbation size ϵ, this being the key to the technique under scrutiny. Note that the x-axes of Figure 2 are on a log-scale, and so do not naturally show the output for $\epsilon = 0$. These data points are nonetheless present on the far left of each plot by means of artificially setting $\epsilon = 0$ to $\epsilon = 10^{-6}$. In all cases, for both training and testing data, un-normalized and normalized, the performance seems to decrease with ϵ. This trend is significantly deviated from only for the last point of the un-normalized testing data, which may just be a statistical effect. The important point is that there does seem to be a clear trend here, but, again, further investigation would be required to effectively exploit it.

The MatLab neural network was intended to be no more than an advanced baseline scenario, and to provide a development platform for the adversarial training components that could then be incorporated into something more sophisticated overall. To this end, following an in-depth survey of available tools, the Python-based Theano framework was chosen as a basis for the final stage of this work. Theano is a specialized package for performing tensor computations efficiently by treating them symbollicaly. As such, it is very useful in machine learning applications. It is a competent package which is well supported and, importantly, well documented, and was a natural choice following the progression of work up to this point. Further, as the modification to the cost function J required access to several gradients of the cost function, Theano's symbolic integration allowed for quick experimentation with many different cost functions. An example Theano neural network code was taken as a basis and adapted for the HMLC dataset. Specifically, a Multilayer Perceptron was written using Theano to classify the HMLC dataset. The network consisted of 30 node input layer, normalized, a hidden layer with 600 nodes and tanh as an activation function, and an output layer with two nodes, using softmax to read the output. The network was trained on a 2013 Macbook Pro with 2 cores and 8GB of ram, for 100 epochs. As with the baseline algorithms, the data was split into 225,000 training samples and 25,000 validation samples.Two cost functions were used in testing. The results are summarized below:

cost	non-adversarial	adversarial
Negative log likelihood	73.368	74.552
Cross entropy	65.5	65.5

7 Conclusion

The purpose of this project was to evaluate the affect of adding adversarial training to neural networks modeled for the Higgs Machine Learning Challenge. First, we implemented an SVM and a Kernel Perceptron. These gave us a baseline for how well simple models could perform on the challenge. Next, we added adversarial training to the neural network adapted from Assignment 3. Finally, we implemented a more robust neural network in Python using Theano. The following table shows each model implementation and its corresponding correct classification rate:

model implementation	correct validation classification percentage
SVM	73.744
Kernel Perceptron	76.98
Matlab NN (non-adversarial)	
Matlab NN (adversarial)	
Python/Theano NN with Negative log likelihood (non-adversarial)	73.368
Python/Theano NN with Negative log likelihood (adversarial)	74.552
Python/Theano NN with Cross entropy (non-adversarial)	65.5
Python/Theano NN with Cross entropy (non-adversarial)	65.5

The well-tuned SVM classified validation samples correctly at a rate of 73.744 percent. Similarly, the kernel perceptron classified 76.98 percent of validation samples correctly. The simple neural networks performed on par with these baseline algorithms when using a negative log likelihood cost function. Without adversarial training, the Python/Theano neural network classified 73.368 percent of validation samples correctly. This was a nearly identical success rate to the much more well-tuned SVM. Although the neural network performed slightly worse than the Kernel Perceptron, such a simple neural network shows more potential for improvement. These observations, along with the fact that the winner of the competition used neural networks, validate feet forward neural networks as a competent model for this classification problem.

Adding adversarial training to our neural networks had somewhat mixed results. In both cases where negative log likelihood was used as the cost function, adversarial training slightly improved the classification rate. On the other hand, when cross-entropy was used as the cost function, no change was observed in the classification rate between the same network with and without adversarial training on the same training and validation set. Overall, we view these results as strengthening the case for adversarial training. There were no cases where the addition of adversarial training hindered classification performance. Additionally, two of the three neural networks improved due to adversarial training.

To address the question of whether adding adversarial training is worth it or not, we examine the cost of implementation and the increased training time. In terms of implementing, the ”fast gradient sign method” lives up to its name. The calculations needed to produce each adversarial sample are performed while training against the corresponding given training sample in neural networks.

So, producing the adversarial samples is almost free in terms of time spent. The only significant time increase is due to training against each adversarial sample. Since every given training example produces a corresponding adversarial example, there are twice as many samples to train against when using adversarial training. Therefore, adversarial training takes twice as long as non-adversarial training in neural networks. For our simple neural networks, this increase in training time was not a problem. In general, doubling training time could be significant when cross-validating for other network parameters. However, once the other parameters have been chosen, trading a 50 percent reduction in training speed to improve a neural network classifier is generally beneficial.

The next step for our team would be to add adversarial training to the winning neural network model from the competition and test to see if it outperforms its predecessor. The improvement of our simple neural network shows promise for adversarial training to improve the winning model. Furthermore, due to the relative ease of implementation and minimal increase in training time, validating this method of adversarial training is advisable for any neural network classifier.

Contributions

- All: Survey of available tools; General research; Poster editing; Paper editing
- Darren Temple: MatLab/Python kernelized perceptron adaptation; MatLab neural network adaptation; Initial adversarial training implementation; Communication with external contacts
- Oren Shklarsky: Python SVM adaptation; Python/Theano neural network adaptation; Advanced adversarial training implementation
- Morgan Jenkins: Interpretation of research material; Development of adversarial training algorithm; Assistance with Python/Theano neural network
- Lola the dog: Running; Jumping; Licking heads; General distractions

References