Differentiable Programming: Neural Networks and Selection Cuts Working Together

Gordon Watts[1*]

[1]University of Washington, Department of Physics, BOX 351560, Seattle, Washington, 98122, USA

Abstract. Differentiable Programming could open even more doors in HEP analysis and computing to Artificial Intelligence/Machine Learning. Current common uses of AI/ML in HEP are deep learning networks – providing us with sophisticated ways of separating signal from background, classifying physics, etc. This is only one part of a full analysis – normally skims are made to reduce dataset sizes by applying selection cuts, further selection cuts are applied, perhaps new quantities calculated, and all of that is fed to a deep learning network. Only the deep learning network stage is optimized using the AI/ML gradient decent technique. Differentiable programming offers us a way to optimize the full chain, including selection cuts that occur during skimming. This contribution investigates applying selection cuts in front of a simple neural network using differentiable programming techniques to optimize the complete chain on toy data. There are several well-known problems that must be solved – e.g., selection cuts are not differentiable, and the interaction of a selection cut and a network during training is not well understood. This investigation was motived by trying to automate reduced dataset skims and sizes during analysis – HL-LHC analyses have potentially multi-TB dataset sizes and an automated way of reducing those dataset sizes and understanding the trade-offs would help the analyser make a judgement between time, resource usages, and physics accuracy. This contribution explores the various techniques to apply a selection cut that are compatible with differentiable programming and how to work around issues when it is bolted onto a neural network. Code is available.

1 Introduction

Physicists have been writing code to analyse and mine the data from their particle physics experiments since the 1970's, at least. The conceptual operations used then – is this jet greater than 1500 MeV – look at lot like SQL operations of today (SQL was developed in the 1970's). What we do today looks remarkably similar. However, analysis started to change with the LeCun's seminal paper in 1989 demonstrating the possibility of large neural networks to do signal/background discrimination [1]. Machine Learning has become a part of a scientist's toolbox and been creeping further and further into the analysis pipeline.

Selection cuts remain an important part of the analysis pipeline, and likely will remain, for several reasons. First, unlike a Machine Learning (ML) based selection cut, it is

* Corresponding author: gwatts@uw.edu

immediately understandable what is happening. Certainly, the physics impact of a selection cut is more apparent than a ML based selection. This lack of interpretability is one of the weaknesses of ML. In fact, ML has gotten so powerful it can encode Lorentz Boosts and 4-vector addition and other basic physics operations – the community now understands how to build these operations into their ML networks [2].

For this proceeding, consider Machine Learning as consisting of two components. First is the *complex function* that is to be fit to the data. During inference this function, with its many fit parameters, can look at the data and determine if it is looking at signal or background. Second is the *training process*. This is finding the best set of parameters so the function will be best at discriminating between signal and background. This is usually called *training the network* and uses a technique like *gradient-descent*. To put it in physics parlance, this is just a function fit, albeit a function with 100's of thousands of parameters (or more – GPT-4 has more than a trillion parameters [3]). Gradient-descent refers to the act of determining the gradient of the function we are fitting, and then moving towards a minimum by adjusting each of those 100's of thousands of parameters.

However, the analysis of particle physics data is challenging due to the complexity and uncertainty of the physical processes involved, as well as the limitations of the computational resources available.

Generically, *differentiable programming* is a paradigm that allows programmers to write code that can be automatically differentiated and, thus, optimized by a compiler or an interpreter or other tool. Differentiable programming is a superset of ML. One of the potential applications of differentiable programming is physics analysis. Differentiable programming can offer a novel way to address these challenges, by allowing physicists to express their models and hypotheses in a differentiable form, and then use gradient-based optimization to fit them to the data and infer the underlying parameters. This can also enable physicists to incorporate prior knowledge and domain-specific constraints into their models, as well as to explore new possibilities and scenarios that are not easily accessible by traditional methods.

Casting back to the analysis example above, the differentiable function now includes not just the ML network parts, but also the selection cuts, and any physics we might want to add. For example, performing a window selection cut on the mass of a reconstructed Z: $80\ GeV < Z_{e^+e^-} < 100\ GeV$. The 80 and the 100 can now become parameters for optimization – optimized with the final physics measurement in mind, not just the local goal of sampling a clean selection of Z's. Once the final optimization is done, one can come back and look at what numbers were chosen by the optimization process. Perhaps it was 88 and 93 or maybe 70 and 110 – both of which give one physics information about how downstream the Z is being used, how clean a sample you needed, how well you could deal with radiation, etc. To calculate the gradient of this, the gradient of the four-vector addition, mass calculation, and the low and high window cuts must be differentiable.

There are also practical uses. The idea for this project was stumbled upon when looking at how Analysis Facilities (AF) and the very large HL-LHC datasets are expected to interact. **Fig 1** shows a possible AF design, with the AF located near a large storage element capable of holding a complete HL-LHC dataset. The two components are connected via a high-bandwidth network. However, by performing local thinning and skimming at the storage element, a dramatic reduction on networking requirements can be achieved. This will be particularly important if the storage element or facility is not collocated with the AF. These so-called preselection cuts that can be applied in the *Simple Query Processor* (see **Fig 1**) must be arrived at – with differentiable programming we have the option of determining them automatically.

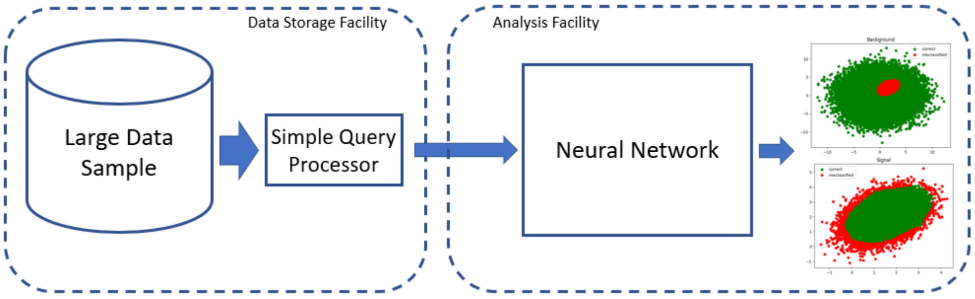

Fig 1. A possible design of an Analysis Facility (AF) and its interactions with the very large datasets from the HL-LHC experiments. The user will want to preselect some of the very large datasets that are then used in their downstream analysis (represented by a NN (Neural Network) here). Minimizing the network bandwidth between the storage elements and the AF will improve overall efficiency.

This short proceedings is structured in two sections. First a section outlining the tools and code used to explore this, along with some of the issues. The next section describes some of the problems faced implementing this and workaround and solutions.

2 The Toy Analysis Model

A very simple toy model was set up to explore the problem space. The toy data are in a simple 2D space (call the axes x and y). The background is a 2D gaussian centred at (0,0) with $\left(\sigma_x, \sigma_y\right) = (9,9)$ and no correlation between the two gaussians. The signal is also a 2D gaussian distribution, centred at $(1.5, 2.0)$ with widths $(0.5, 0.5)$ and a 20% correlation between the two axes.

The logical flow of the analysis consists of an individual cut on x and y, followed by a dense Neural Network (NN). The dense NN is implemented in the Haiku library and has 5 layers (`hk.nets.MLP(output_sizes= [2, 15,30, 15, 1])`). The last single output layer is, of course, the weight and is meant to tell the difference between signal and background. It was verified that this network could discriminate between signal and background on its own – cutting a very nice hole in the middle of the background distribution to allow signal through.

Training can proceed stochastically or via weights. As described below the weight method was chosen. As shown in **Fig 2**, the weights representing each cut on x and y are multiplied by the weight from the NN. In theory if the weights of the selection cuts are zero, gradient descent should not try to alter the weights in the NN during the training process.

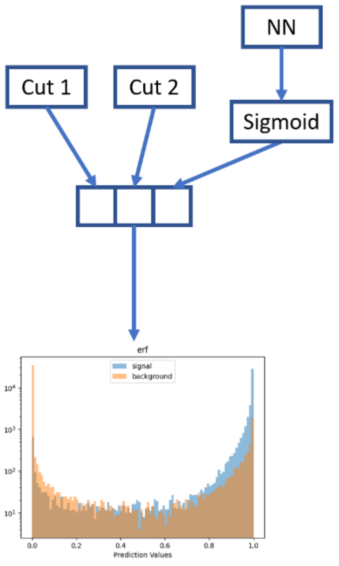

Fig 2. To apply gradient-decent optimization the cuts and dense NN in the toy analysis must be combined into a function. This work uses weights to represent the selection cuts, and the NN produces a weight by design. The three weights are multiplied together, and the result of that is the output of the overall function.

2.1 Tools Used

All code is available on GitHub [4]. All work was performed with Jupyter notebooks as the primary user interface [5]. The following software packages were used:

1. JAX – This is a python package for differentiable programming and building deep NN's developed and maintained by Google's DeepMind AI (Artificial Intelligence) research group. It combines Autograd and XLA to enable high-performance machine learning research. Autograd is a library that can automatically differentiate native Python and NumPy functions, even through complex control flow and higher-order derivatives. XLA is a compiler that can optimize and run NumPy programs on GPUs and TPUs. JAX provides a simple and flexible API based on NumPy, as well as composable function transformations such as `grad`, `jit`, `vmap`, and `pmap`. These transformations allow one to express sophisticated algorithms and get maximal performance without leaving Python. Unlike PyTorch and TensorFlow, which use static or dynamic graphs to represent computations, JAX uses a functional approach that enables automatic differentiation and parallelization of any Python function. [5]
2. **Haiku** – This library is based on the programming model and APIs of Sonnet, a neural network library for TensorFlow. Haiku enables users to use familiar object-oriented programming models while allowing full access to JAX's pure function transformations. Haiku provides a module abstraction, `hk.Module`, and a function transformation, `hk.transform`, to manage model parameters and state. Haiku, released by researchers at DeepMind, has been used at scale to tackle challenges in various domains such as image and language processing, generative models, and reinforcement learning. Particularly important for this work is its composability – it was very easy to build new `hk.Modules` for the selection cuts (and the multiplication of the weights) [6].
3. **Optax** - A Python module for optimization and gradient transformation in JAX. It provides a collection of composable gradient transformations that can be combined to implement various optimization algorithms. Optax also supports custom gradient transformations and higher-order differentiation. Optax is designed to be simple, flexible, and easy to use. [7]

Other well-known packages are used as well (e.g., `numpy` [8], `matplotlib` [9], etc.).

3 Implementation Discussion

Several problems were encountered in implementing this: 1) Cuts are not differentiable, 2) NN design and training, and 3) how the loss function is calculated. Each sub-section discusses these issues.

3.1 Cuts Are Not Differentiable

Cuts, like $p_T > 30$ GeV, are discontinuous by their nature, which means their first derivative is infinite right at the cut value. Gradient-descent needs the function to be continuous. If the cut is represented as a weight, it must smoothly transition between zero and one as it crosses the boundary. The error function and sigmoid were both explored, as shown in **Fig 3**. The slope of the function can be easily controlled by a parameter.

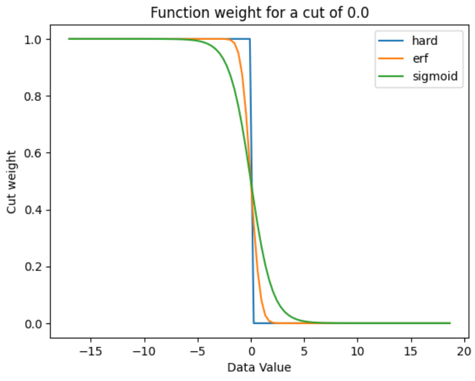

Fig 3. The weight of a selection represented as a cut for the hard cut (a step function), the Error Function, and the Sigmoid Function.

Note that for all functions the derivate will be zero far away from the cut value. This can cause a problem, for example, if your training starts with a cut value that is far away from the bulk of your distribution. As an example, **Fig 4** is the derivative of the binary cross-entropy for the error function, along one of the two axes of data. Note that the gradient is zero far away from the bulk of the distributions (around -15 or above 15).

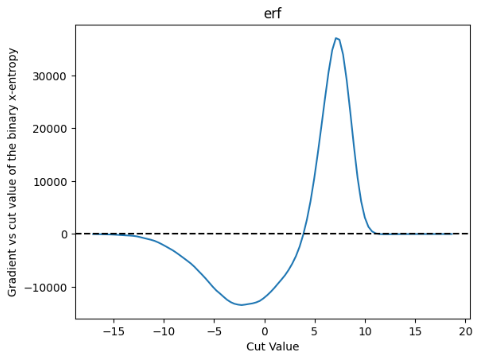

Fig 4. The derivative of the binary cross-entropy for one axis of the signal and background data samples for the error function as a function of the cut value.

By default, when a training is initialized in most training frameworks, parameters are given an initial random value. If the value is outside the range $[-15, 15]$, then the derivative will be zero – and the training algorithm will not be motivated to change the cut value.

There are several possible approaches to address this issue. The simplest of which, adopted here, is to analyse the initial distribution and place the cut somewhere in the bulk of the distribution. Other options include artificially including adding a small, but nonzero, positive, or negative first derivative at the ends that push the gradient decent algorithm back towards the area that has data.

As described in the introduction, a cut in a real analysis might be implemented in a simple query processor near a storage element. In this case the cuts will not generate a weight, but rather skim the dataset before it is sent downstream to the AF. The cut will be hard at exactly the value of the cut. Thus, the slope for the error and sigmoid functions shown in **Fig 3** represents an inaccuracy. However, the larger the slope, the easier time the training algorithm will have. One possibility is annealing – as the training gets closer and closer to the answer the slope can be increased until it gets close to being vertical.

3.2 NN Design and Training

It took several iterations to arrive at the weight-multiplication design of the network shown in **Fig 2**. There are two options to approach this. First is a stochastic training method – which uses estimates based on an expectation value. The attractive part of this method is that it uses the actual hard cuts. Each iteration through the training the algorithm estimates how often each branch of the cut is taken and calculates an expectation value based on that. However, for something as simple as selection cuts, this just looks like the weight. As a result, the decision was made to go with a fully weight-based approach.

This has at least one other consequence during the training: the full training dataset is required for training. If one goal is to use cuts to limit the data from the storage element, this training method means this is not possible – during training. However, even in the stochastic approach the cuts will be varied and likely the full phase space will need to be explored during the training – and as a result the full training-dataset will likely also be required here as well. The conclusion is that during training it is unlikely one can avoid looking at the full training dataset. Further that during training it does not make sense to put in machinery to limit the size of the training dataset coming from a query processor (if one is used). In HL-LHC physics analyses, however, the largest single source of data for an analysis is usually the detector data itself, which is frequently not used for training. As a result, this technique should remain a big help during the inference phase of the analysis.

3.3 The Loss Function

There are a large variety of loss functions available to the particle physicist. Perhaps some of the most attractive are $S/\sqrt{B}$ and binary cross-entropy. The latter was used during these studies. At some point the full analysis pipeline might be made differentiable, in which case the loss function becomes the sensitivity and perhaps the entire chain is driven by a statistical tool like `pyhf` [10]

A typical practice when training NN is to apply the softmax function to the output of the NN, just before the loss function is calculated. The softmax function is a way of transforming a vector of real numbers into a probability distribution over a finite number of classes. It normalized the output of the NN to a probability distribution. The softmax function applies the exponential function to each element of the input vector and then divides by the sum of all exponentials, ensuring that the output values are between 0 and 1 and sum to 1. The formula for softmax is:

$$softmax(x_i) = \frac{e^{x_i}}{\sum e^{x_i}}$$

For this discussion it is important to note that the value of an element $softmax(x_i)$ depends on all the other x_i's! During training this can cause results to drift in several ways, when applied to the final weight calculated in **Fig 2**. The weights from the selection cuts are no longer playing the role of a selection as a result. During training the NN can be pushed very hard to overcompensate for the selection cuts driving the training into unsensible corners of phase space. Instead, the $softmax$ function should be applied only to the output of the NN, and that multiplied by the selection cuts to produce the final value that is used in the loss function.

As a side note – JAX and Haiku's open and transparent building blocks for NN's made a huge difference in debugging this problem. It was not very difficult to train the NN along with the selection cuts and the softmax function, extract the trained NN and look at its output values and be able to see very quickly something had gone off the rails. And then one step at a time to get back to the result.

4 Conclusions

Fig 5 shows the results of one of the trainings. The figure caption contains the final value of the selection cuts. Note the selection cuts are not fully outside of the signal – so besides cutting out a great deal of the background, they are also cutting out some of the signal. It is interesting to note that the training has some artifacts. The author did not have time to explore the cause or solution to those artifacts. However, the basic approach appears to be solid.

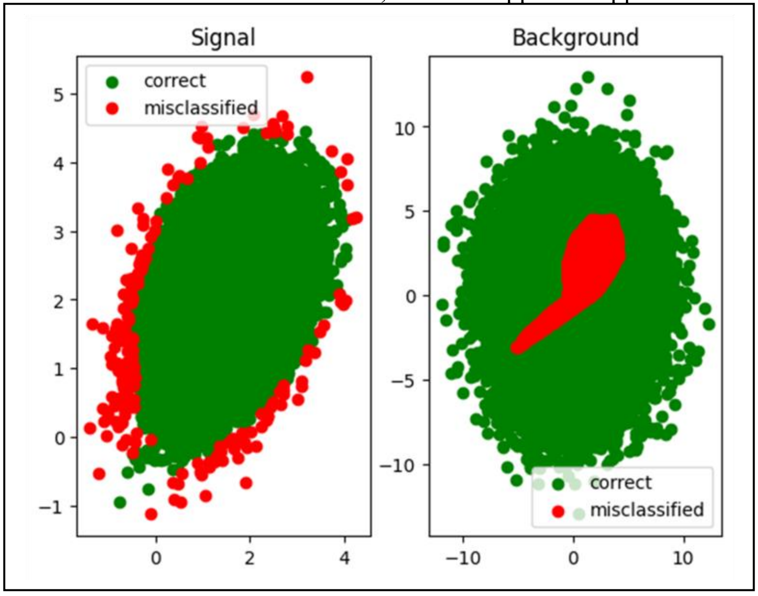

Fig 5. The result of the training is shown with correctly (green) and incorrectly (red) classified parts of the signal (left) and background (right) distributions. The sigmoid function was used to represent the cuts and at the end of training had values $x_{cut} = -1.85, y_{cut} = -1.81$.

The next steps are to build this into a fuller example. This toy data and toy analysis chain is much too simplistic. As noted in the text several shortcuts have been taken and for this to be built into a robust tool these will need to be addressed. Finally there is the issue of the user-interface. This is currently built upon JAX and Haiku. While the machinery under the hood might eventually need to look like that, it would be much nicer to be able to write `pT>30` and have the code automatically replace this with a sigmoid or error function. As improvements have been made in the stochastic approaches, a more serious comparison needs to be done there [11].

This work was supported by the National Science Foundation under Cooperative Agreements OAC-1836650 and PHY-2323298.

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, and W. E. Hubbard, “Handwritten digit recognition with a back-propagation network,” in *Advances in Neural Information Processing Systems*, 1999, pp. 396–404.

[2] A. Bogatskiy, B. Anderson, J. T. Offermann, M. Roussi, D. W. Miller, and R. Kondor, “Lorentz Group Equivariant Neural Network for Particle Physics,” *37th International Conference on Machine Learning, ICML 2020*, vol. PartF168147-2, pp. 969–979, Jun. 2020, Accessed: Sep. 26, 2023. [Online]. Available: https://arxiv.org/abs/2006.04780v1

[3] OpenAI, “GPT-4 Technical Report,” Mar. 2023, Accessed: Sep. 26, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774v3

[4] “gordonwatts/diff-prog-intro: Me, attempting to understand the basics of differentiable programming.” Accessed: Sep. 26, 2023. [Online]. Available: https://github.com/gordonwatts/diff-prog-intro

[5] T. Kluyver *et al.*, “Jupyter Notebooks – a publishing format for reproducible computational workflows,” *Positioning and Power in Academic Publishing: Players, Agents and Agendas - Proceedings of the 20th International Conference on Electronic Publishing, ELPUB 2016*, pp. 87–90, 2016, doi: 10.3233/978-1-61499-649-1-87.

[6] “google-deepmind/dm-haiku: JAX-based neural network library.” Accessed: Sep. 26, 2023. [Online]. Available: https://github.com/google-deepmind/dm-haiku

[7] “google-deepmind/optax: Optax is a gradient processing and optimization library for JAX.” Accessed: Sep. 26, 2023. [Online]. Available: https://github.com/google-deepmind/optax

[8] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature 2020 585:7825*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.

[9] J. D. Hunter, “Matplotlib: a 2D graphics environment,” *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/mcse.2007.55.

[10] L. Heinrich, M. Feickert, and G. Stark, “scikit-hep/pyhf: v0.7.4,” Sep. 2023, doi: 10.5281/ZENODO.8323306.

[11] M. Kagan and L. Heinrich, “Branches of a Tree: Taking Derivatives of Programs with Discrete and Branching Randomness in High Energy Physics,” Aug. 2023, Accessed: Sep. 26, 2023. [Online]. Available: https://arxiv.org/abs/2308.16680v1