Embedded Continual Learning for High-Energy Physics

Marco Barbone[1,*], *Christopher* Brown[1], *Georgi* Gaydadjiev[2], *Thomas* Maguire[2], *Mikael* Mieskolainen[1], *Benjamin* Radburn-Smith[1], *Wayne* Luk[1], and *Alexander* Tapper[1]

[1]Imperial College London, South Kensington, London, United Kingdom
[2]Bernoulli Institute, University of Groningen, The Netherlands

Abstract. Neural Networks (NN) are often trained offline on large datasets and deployed on specialised hardware for inference, with a strict separation between training and inference. However, in many realistic applications the training environment differs from the real world, or data arrives in a streaming fashion and is continuously changing. In these scenarios, the ability to continuously train and update NN models is desirable. Continual learning (CL) algorithms allow training of models on a stream of data. CL algorithms are often designed to work in constrained settings, such as limited memory and computational power, or limitations on the ability to store past data (e.g, due to privacy concerns or memory requirements). High-energy physics experiments are developing intelligent detectors, with algorithms running on computer systems located close to the detector to meet the challenges of increased data rates and occupancies. The use of NN algorithms in this context is limited by changing detector conditions, such as degradation over time or failure of an input signal which might cause the NNs to lose accuracy leading, in the worst case to the loss of interesting events. CL has the potential to solve this issue, using large amounts of continuously streaming data to allow the network to recognise changes, and to learn and adapt to detector conditions. It has the potential to outperform traditional NN training techniques as not all possible scenarios can be predicted and modelled in static training data samples. However, NN training is computationally expensive and when combined with the strict timing requirements of embedded processors deployed close to the detector, current state-of-the-art offline approaches cannot be directly applied to the real-time systems. Alternatives to typical backpropagation-based training that can be deployed on FPGAs for real-time data processing are presented, and their computational and accuracy characteristics are discussed in the context of High-Luminosity LHC.

1 Introduction

Contemporary Machine Learning (ML) excels at training powerful models from fixed datasets and stationary environments; often exceeding human-level ability. However, a crucial challenge arises due to the difference between training and inference environments, which can significantly erode the accuracy of machine learning models [1]. Various factors contribute to this disparity, including:

*e-mail: m.barbone19@imperial.ac.uk

- The computational expense of generating a dataset that encompasses all potential scenarios;
- The excessive storage requirements for a dataset covering all possible scenarios;
- The inherent unpredictability of certain scenarios;
- The possibility of certain scenarios being inadvertently omitted by developers.

In particular, reality's intricate nature presents a formidable hurdle, as achieving an exact simulation to construct a comprehensive database might be infeasible within reasonable time-frames or entirely unattainable. The multitude of possible variables leads to an exponential increase in potential scenarios, resulting in dataset sizes that could become unmanageable for storage. This also introduces the challenge of training machine learning models on such a massive dataset.

To maintain high accuracy between training environments and inference environments it is possible to use concepts such as Online Learning (OL) and Continual Learning (CL), both of which allow the model to learn at inference time from online data. The key distinction between OL and CL lies in the fact that OL is susceptible to **catastrophic forgetting**. Catastrophic forgetting is the tendency for knowledge of the previously learned task(s) (e.g. task A) to be abruptly lost as information relevant to the current task (e.g. task B) is incorporated [2]. Alternatively, CL offers a range of techniques to alleviate or entirely address the challenges posed by catastrophic forgetting.

The upcoming high luminosity upgrade to the LHC, the High-Luminosity LHC (HL-LHC) necessitates upgrades to the CMS experiment. These upgrades will see a substantial array of ML algorithms being used in CMS's hardware Level-1 (L1) trigger system [3]. These ML algorithms are designed to aid in the selection of potentially significant events for offline analysis and storage. In this context, CL might offer higher accuracy than traditional ML as the current strategies involves training these ML models on extensive, meticulously refined Monte Carlo simulation datasets and subsequently integrating them into the detector environment. However, the possibility of changing detector conditions over time could lead to alterations in the model's output, possibly resulting in a deterioration of trigger performance.

Using ML within the hardware trigger system presents several challenges; the demands of strict latency and resource constraints impose limitations on the dimensions and complexity of these models, primarily due to their integration within a high-speed trigger setup and their eventual deployment on FPGA hardware. Deploying CL within the hardware trigger is significantly more challenging as it requires labelled data at inference time and training at inference time to learn new knowledge.

The stream of labelled data can be generated either from full offline analysis or systems that run at full collision rate collecting unbiased data [4]. The amount of labelled data required is significantly lower than the amount of data generated by the detector itself. As a result, slower offline analysis algorithms can provide the necessary labels. Training at inference time usually requires executing computationally expensive training algorithms such as backpropagation at inference time. This is particularly challenging in embedded systems or in the context of the hardware trigger systems as GPU accelerators are not available. Hence, it is necessary to find lower complexity alternatives to backpropagation that could be deployed in constrained environments.

2 Online Alternating minimisation

Auxiliary variable methods are approaches in which the objective functions employed to update the weights of a model are divided into local subproblems; utilising auxiliary variables linked to the activation vectors of hidden layers. Although previous methods have proposed auxiliary variable approaches, they have been limited to offline settings and require the complete training dataset at each iteration. **Online Alternating minimisation** (Altmin) [5] is the first auxiliary variable method that operates in an online setting, making it potentially useful in continual learning scenarios. Choromanska et al. [5] propose two variants of Altmin, AM-Mem and AM-Adam; in this study we focus our attention to AM-Adam as AM-Mem is slower and requires the storage of large memory matrices and thus is not suitable for computing in resource constrained environments.

The algorithm's high-level overview involves storing auxiliary variables (codes) during the forward pass for each linear layer. The auxiliary variables are equal to a linear transformation of the previous-layer activations, equivalent to the output of the respective layer. The error is then propagated backwards using these codes. The codes variables are updated in reverse order, starting with the last hidden layer. The objective function for updating the codes in the last hidden layer employs the model's loss function to calculate the error rate. For hidden layers preceding the last one, a local function utilising Mean Squared Error loss computes the error rate between the updated codes at the next hidden layer and the predicted codes at that layer. This error is propagated backward to update the codes, rather than propagating the error from the output layer all the way back significantly reducing the length of gradient chains. After the code updates, the weights are updated using the updated codes. This process lends the name "alternating minimisation" - during the first stage, weights and biases remain fixed while codes are updated, and then codes are fixed while weights and biases are updated. Each layer's weights possess their own objective function, adjusted to optimise the layer's prediction of updated codes at the next layer. Because the objective functions are local and independent, weight updates can occur in parallel.

$$\begin{aligned}
c^L &= \arg\min_c \mathcal{L}(y, \sigma_L(c), W^{L+1}) + \mu \|c - W^L \sigma_{L-1}(c^{L-1})\|_2^2 && (1)\\
c^l &= \arg\min_c \mu \|c^{l+1} - W^{l+1}\sigma_l(c)\|_2^2 + \mu \|c - W^l \sigma_{l-1}(c^{l-1})\|_2^2 && (2)\\
&\quad \text{for } l = L-1, \dots, 1 \\
W^{L+1} &= \arg\min_W \mathcal{L}(y, \sigma_L(c), W) && (3)\\
W^l &= \arg\min_W \mu \|c^l - W\sigma_{l-1}(c^{l-1})\|_2^2 && (4)\\
&\quad \text{for } l = L-1, \dots, 1
\end{aligned}$$

The equations above represent a neural network with L hidden layers, indexed from 1 to L. Conventionally, $c^0 = x$, where x is the input data. c^1 to c^L refer to the codes at the corresponding hidden layers. σ^l refers to the non linear transformation between layers l and l+1. μ is a hyperparameter used to adjust the learning rate. Equation 1 outlines the objective function for updating codes in the last hidden layer, and Equation 2 corresponds to updating codes in other layers. Equations 3 and 4 correspond to updating weights in the last layer and other layers, respectively. Weight and bias gradients are calculated with respect to the loss function, and the Adam algorithm [6] is employed to update weights and biases based on the gradients.

The use of local objective functions reduces the risk of vanishing and exploding gradients [7], as error propagation occurs over only a small section of the network; limiting the length of derivatives calculated using the chain rule. Compared to backpropagation, which necessitates full chain rule expansion. Online alternating minimisation operates at a complexity one order lower, potentially resulting in lower training time than backpropagation.

However, the storing of code variables is a small additional memory cost of Altmin. The codes are fixed size so the memory can be reserved initially but they only need to be stored until the weights are updated and can then be replaced by the codes for the next batch of data by overwriting the pre-allocated code matrices.

3 Dataset

In order to demonstrate the use of alternative CL methods a small example scenario was used. This scenario aims to reproduce the environment of an upgraded High Energy Physics detector, namely the High-Luminosity LHC upgrade of CMS. Due to the challenging conditions of the high-luminosity LHC, CMS is upgrading its hardware trigger system and will be incorporating more ML than ever before [3].

One of the key tasks for triggers in the high-luminosity LHC is to identify soft overlapping proton-proton interactions versus the high momentum hard scatter in an event. These soft interactions serve as a background to the trigger system so reducing their impact is vital for physics selectivity. To emulate this high-luminosity CMS scenario a Delphes [8] simulation of the upgraded CMS detector was run on a Pythia [9] top-quark pair production event overlaid with 200 soft additional interactions. Tracks were generated and kinematically constrained in order to match those available in the CMS hardware trigger with track transverse momentum > 2 GeV, absolute pseudorapidity < 2.4, and absolute transverse impact parameter < 15 cm.

To emulate a scenario where a model is trained on a perfect MC dataset and a changing detector environment this Delphes dataset was split in two. One dataset was preserved as the initial "perfect" training sample and the other dataset was further split into ten further sub-datasets each with a gradually worsening Gaussian smear on the detector parameters. These smears were implemented by creating a Gaussian of standard deviation from 1 up to 10 % (incrementing in each of the sub-datasets) of the detector parameters and re-sampling the parameters from these Gaussians. The mean was kept unchanged so no systematic shift is introduced to the parameters; only a random noise. These small datasets were used as the CL experiences to demonstrate how an ML model might change with a gradually worsening detector. While this emulation is not perfect and does not necessarily represent the way in which CMS will degrade in the high-luminosity LHC it does provide a useful example to experiment with CL algorithms.

4 Network

As the scenario is based on a hardware trigger example, the model was also designed to be similar to those in the hardware trigger. This motivated the use of a very lightweight model; a three-layer neural network with ReLU activation functions and total of 61 neurons. The very simple model has the added advantage of being faster to train and simpler to implement, making the CL algorithm the main focus of the experimentation. The model was trained on track helix parameters from the Delphes simulation using track momentum, pseudorapidity, azimuthal angle, and distance along the beam direction (z_0). An additional variable used was the distance between the track and the vertex location in the event. This vertex is the point along z_0 where the hard scatter in the event is estimated to be, determined using a simple histogram-based vertex finding algorithm. The model was trained, using a binary cross entropy loss function, to identify whether or not a track originated from the Pythia top-quark pair production or from the additional soft interactions; a target variable determined from truth-level matching in Delphes.

5 Experimental results

This section discusses the accuracy and speed of Altmin compared to Stochastic Gradient Descent (SGD). We refer as Altmin to the algorithm in general, as Reference to the original python implementation provided in [10] and as fastAltmin [11] to the C++ implementation that we developed. The main difference between the reference implementation and fastAltmin is that the latter leverages vectorization and additional parallelism, so that layers can be updated in parallel. Results obtained using fastAltmin apply to Altmin in general as there is no difference or loss of generality compared to the original algorithm. As this study targets resource constrained computing environments, all the experiments are executed on CPU as GPUs are rarely available on such systems. The CL network presented here is implemented using the state-of-the-art CL library Avalanche [12] and Altmin is integrated directly into the Avalanche training loop.

5.1 Convergence speed

Before analysing Altmin's performance it is necessary to ensure the correctness of the method when compared to SGD, as well as to ascertain its convergence speed. Assessing the convergence speed against SGD is crucial, as while Altmin might exhibit accelerated performance, its potential benefits could be offset if it requires more epochs to converge. This comparison is essential to ensure a comprehensive understanding of the method's overall efficacy. Figure 1 shows the loss at each epoch, Reference and fastAltmin and exhibit slightly faster but more oscillatory convergence.

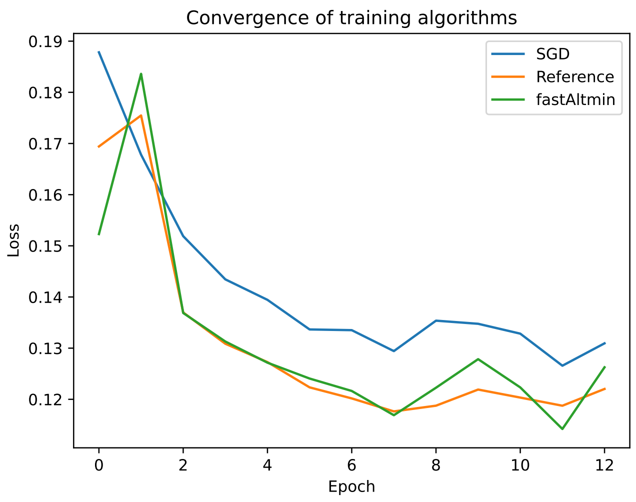

Figure 1: Reference (python), fastAltmin (C++) and SGD convergence.

5.2 Training time

After assessing the convergence speed of Altmin, the subsequent phase involves the evaluation of its performance. For a fair comparison we provide fastAltmin, a C++ implementation of Altmin as SGD is implemented in C inside the torch framework [13]. The performance of the reference python implementation (Reference) is also included for completeness. Figure 2 illustrates the speedup of fastAltmin in comparison to SGD. Notably, for smaller batch sizes, fastAltmin demonstrates a twofold acceleration over SGD. As batch sizes grow larger, the speedup goes from 2x to 1.12x. However, for a batch size of 1000, SGD exhibits a 10% advantage in terms of speed. While an overall downward trend in speedup is apparent with larger batch sizes, fastAltmin always outperform SGD. The original python implementation (Reference) is slightly slower than SGD for a batch size of 500, however this is not a fair comparison as SGD is finely optimised inside the torch framework [13] while Reference is not.

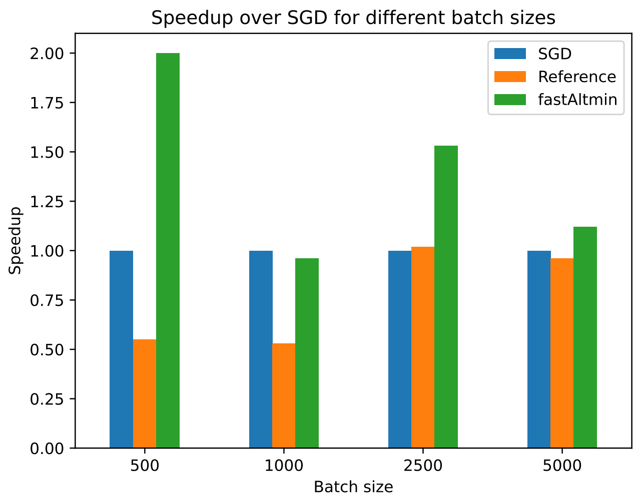

Figure 2: Speedup of Reference (python), fastAltmin (C++) compared to SGD.

5.3 Continual Learning

Using a small batch size is very common across the CL methods [14]. As demonstrated in Section 5.2, fastAltmin showcases the potential to be twice as fast as SGD with smaller batch sizes, rendering it a promising contender for CL applications. However, prior to delving into performance discussions, it is necessary to measure the accuracy of fastAltmin also in a CL scenario. This step is vital to ensure that fastAltmin maintains its performance without any degradation when compared to SGD.

Figure 3 illustrates the accuracy concerning increasing degrees of detector degradation. Initially trained under optimal detector conditions, the model was subsequently evaluated on progressively deteriorating levels, as detailed in Sections 3 and 4. We employed CL across all evaluated methods: while the network was initially trained without degradation, during inference, if the CL algorithm detects decreased accuracy, it incorporates this newfound knowledge into the network. Consequently, we anticipate the CL network's accuracy to remain constant across all degradation levels.

In our evaluation, we progressively assessed the network at higher degradation levels. For instance, when evaluating the network at degradation level 4, it incorporates knowledge up to

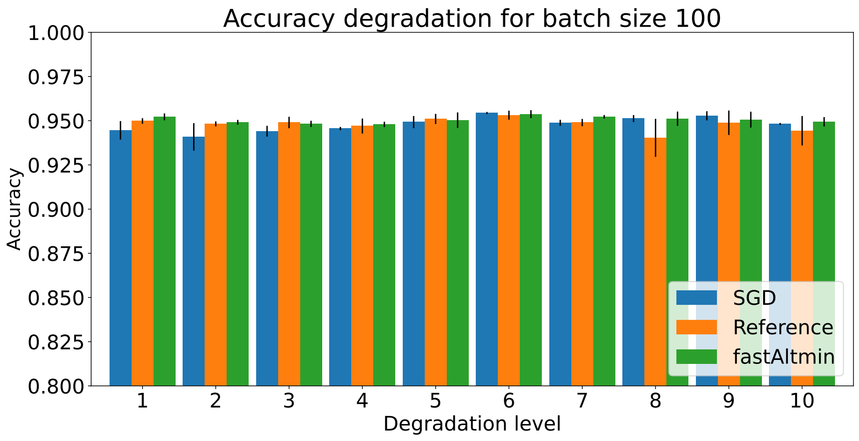

Figure 3: Accuracy of Reference (python), fastAltmin (C++) compared to SGD.

degradation level 3 due to the CL network's capacity to learn from previous experiences. The results showcase remarkable similarity across the three algorithms, with the most substantial discrepancy hovering around 1%. This marginal distinction might potentially be improved through more precise hyperparameter tuning specifically tailored for the CL network.

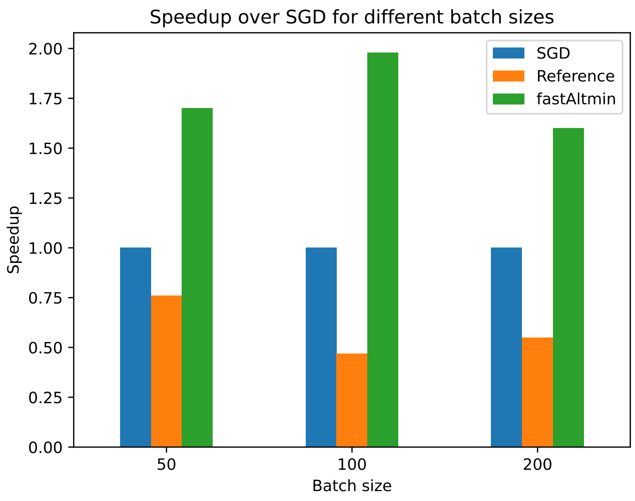

Figure 4: Speedup of Reference (python), fastAltmin (C++) compared to SGD in a CL scenario.

The ultimate assessment gauges the speed-based performance of fastAltmin relative to SGD within a Continual Learning (CL) context. Illustrated in Figure 4, the speedup for three distinct batch sizes—50, 100, and 200—is showcased. Notably, fastAltmin attains a speedup ranging from 1.5x to 2x in comparison to SGD. These findings underscore fastAltmin's aptness for CL scenarios due to its enhanced speed performance.

6 Conclusions

Altmin emerges as a highly promising training algorithm for CL within resource-constrained environments. Its C++ implementation demonstrates exceptional performance, particularly with small batch sizes, achieving a remarkable speedup of up to 2x compared to SGD, while maintaining comparable levels of accuracy and convergence speed. This becomes particularly significant in CL, given that most CL models utilise very small batch sizes, allowing Altmin to retain its performance advantages.

This indicates the potential for reduced latency in CL networks compared to those leveraging SGD. Furthermore, in FPGA implementations, Altmin presents two main benefits: the capability to update all layers in parallel simplifies the dataflow design, potentially enhancing achievable clock speed. Additionally, an optimised implementation might require fewer resources (smaller unroll factors) to achieve performance levels similar to those of SGD. This holds significant relevance in systems like CMS hardware trigger systems, where strict latency and resource constraints dictate the dimensions and complexity of ML models. Altmin could pave the way for deploying compact CL networks that sustain accuracy over time.

References

[1] R. Hadsell, D. Rao, A.A. Rusu, R. Pascanu, Trends in Cognitive Sciences **24**, 1028 (2020)

[2] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A.A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska et al., Proceedings of the National Academy of Sciences of the United States of America **114**, 3521 (2017)

[3] The CMS Collaboration (CMS), Tech. rep., CERN (2020), `https://cds.cern.ch/record/2714892`

[4] T.O. James, Tech. rep. (2023), `https://cds.cern.ch/record/2852916`

[5] A. Choromanska, B. Cowen, S. Kumaravel, R. Luss, M. Rigotti, I. Rish, P. Diachille, V. Gurev, B. Kingsbury, R. Tejwani et al., *Beyond Backprop: Online Alternating Minimization with Auxiliary Variables* (2019)

[6] D.P. Kingma, J. Ba, *Adam: A Method for Stochastic Optimization* (2017)

[7] S. Hochreiter, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems **06**, 107 (1998)

[8] J. de Favereau, C. Delaere, P. Demin, A. Giammanco, V. Lemaître, A. Mertens, M. Selvaggi, T.D.. collaboration, Journal of High Energy Physics **2014**, 57 (2014)

[9] T. Sjöstrand, Computer Physics Communications **246**, 106910 (2020)

[10] *IBM/online-alt-min*, `https://github.com/IBM/online-alt-min`

[11] *DiamonDinoia/altmin*, `https://github.com/DiamonDinoia/altmin`

[12] V. Lomonaco, L. Pellegrini, A. Cossu, A. Carta, G. Graffieti, T.L. Hayes, M. De Lange, M. Masana, J. Pomponi, G.M. van de Ven et al., *Avalanche: An End-to-End Library for Continual Learning*, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (2021), pp. 3600–3610

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury Google, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., Advances in Neural Information Processing Systems **32** (2019)

[14] S.I. Mirzadeh, M. Farajtabar, R. Pascanu, H. Ghasemzadeh, *Understanding the Role of Training Regimes in Continual Learning*, in *Advances in Neural Information Processing Systems*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, H. Lin (Curran Associates, Inc., 2020), Vol. 33, pp. 7308–7320