Time series processing for software failure prediction in deep learning training

Lev Faivishevsky[1] **Ori Ernst**[1] **Amitai Armon**[1]

Abstract

Deep learning models rely on an iterative training process. This training is highly time consuming, encouraging the development of many software packages with extensive optimizations. These complex optimizations may introduce small numeric differences between implementations, but it is crucial to verify that they do not cause non-convergence to the desired accuracy. We propose a time series processing approach to detect a failure to converge in the beginning of the training. The approach predicts whether the current execution differs significantly from a valid training run, with the help of statistical tests. This makes it possible to detect most implementation faults without completing full training runs, thus significantly accelerating software development. This method was developed as part of an effort to optimize deep-learning frameworks.

1. Introduction

Research on how to predict software failures accurately is of great practical importance. Many software failures prediction models have been developed , see e. g. (Goel, 1985) and (Xie, 1993), among which many popular classic probability models are widely discussed (S. Ozekici, 2003) and (Reussner et al., 2003). However, many of these models make restrictive assumptions of reality to ensure tractability and solvability (Bishop & Pullen, 1988).

These assumptions have been relaxed by making no distribution assumptions on the failure process (Pfefferman & Cemuschi-Frias, 2002),and various variants of neural networks have been applied in (Cai et al., 2001) and (Ho et al., 2003). Bayesian networks were used in (Bai et al., 2005). In order to use the Markov Bayesian network model to predict software failure, the initial distribution of software defects and the distribution of software failure time distributions are essential. This type of the information is rarely available in the early stages of software implementations for modern deep learning algorithms. This limits the applicability of the Bayesian networks in this important use case, which is characterized by long training time and a severe need for fast software development time, in the current competitive environment of data science software market. In addition, the iterative nature of the training for such algorithms gives rise to another type of data, which manifests itself in time series form. This way the beginning of the time series may help to predict the behaviour of the whole time series, see (Li et al., 2016).

In this paper we suggest a new method for software failure detection by time series processing. The method relies on statistical hypothesis testing for comparison of time series generated during training stage. Our results empirically show its potential ability to significantly reduce the software development duration.

2. Deep Learning Training Software Failure Prediction

Many modern machine learning algorithms are trained in an iterative manner. This way an initial solution for the training optimization task is gradually improving through a number of iterations. The quality of the solution is assessed during the training iterations by applying the underlying machine learning algorithm on a test set. Training an iterative machine learning algorithm G yields a sequence of testing set accuracy estimates $A_1, A_2, .., A_i.., A_N$, where A_i is the testing accuracy of the algorithm after i training iterations and N is the total number of training iterations. Such sequence of accuracy estimates is dubbed *Learning curve*, see (Domhan et al., 2015). The accuracy of the completely trained algorithm is given obviously by A_N.

We denote the reference software implementation of the algorithm G as G^0, and its subsequent implementations by G^m, so that the training of G^m outputs a time series of accuracies A_N^m. Usually, the subsequent software implementations should be at least as fast as G^0, and, of course, they should achieve the same final accuracy as the reference implementation: $A_N^m = A_n^0$. Due to the complicated nature of the highly optimized software implementations a software failure (bug) may be introduced into a subsequent software implementation, such that the running time of the training

[1]Intel Advanced Analytics. Correspondence to: Lev Faivishevsky <lev.faivishevsky@intel.com>.

ICML 2017 Time Series Workshop, Sydney, Australia, 2017. Copyright 2017 by the author(s).

G^m may be higher, or alternatively, it may not reach the desired final accuracy: $A^m_N < A^0_N$. The former case is easy to reveal during the training run, as it clearly manifests itself after the first few iterations of the training. The latter case is of much higher severity, as it might take the whole training run of N iterations in order to reveal that the desired final accuracy is not achieved. Our goal is therefore to detect whether the bug exists using only a few training iterations $n \ll N$.

A straightforward approach might be to predict the final accuracy as a function of the learning curve beginning: $A_N = f(A_1, ..., A_n)$. One disadvantage of this approach is that function f is dependent on the algorithm G, so we need to rebuild the predictor for each new algorithm to be implemented. In addition, training the predictor f usually requires a high number of verified runs of G, which is too time-consuming.

Instead we base our method on comparing the time series corresponding to the beginnings of the learning curves: $A_1, ..., A_n$. We would like to define a distance D and a threshold T such that we may alert about a possible software bug in software implementation G^m if $D(\{A^0_1, ..., A^0_n\}, \{A^m_1, ..., A^m_n\}) > T$. Evaluating such a distance for the nonstationary time series $\{A^i\}$ might pose a difficult problem. Therefore we make a further assumption, that there is an additional reference learning curve $A^1_1, ..., A^1_N$, which is also valid: $A^0_N \approx A^1_N$. Then we consider time series of differences of accuracies between the current run and a reference run $A^m_i - A^0_i$.

The second reference run enables computing a time series of differences between partial accuracies of the two valid runs of the training. These difference are caused by the computational effects of different random seeds and possibly rounding order noise (in the multi-node case). We therefore expect that these deviations $A^1_i - A^0_i$ follow the normal distribution with zero mean. If the current run is not valid $A^m_N < A^0_N$ then at some iteration we expect a significant deviation between partial accuracies and $A^m_i - A^0_i$ will not be so close to zero as $A^m_i - A^0_i$. For a faulty run the partial accuracies may be either lower or higher in the beginning of the learning curve, which motivates us to measure a magnitude of the deviation as the distance. More specifically, we propose to compare the variance of the differences between partial accuracies as a distance between the current run and the valid runs. If the current run m is valid as well then we would have $Var(A^m_i - A^0_i) \approx Var(A^1_i - A^0_i)$. We apply the Bartlett test (Snedecor & Cochran, 1989) to compute the probability for a difference of variances between two sequences of samples of a normal random variable. We use it for a required confidence level α, so that we may mistakenly alert about a bug in the valid implementation G^m with a probability less or equal to α.

Algorithm 1 Software Bug prediction by Bartlett test (SBPBT)

```
**Inputs:** Current Software implementation G$^m$, two valid learning curves $\{A^0_i\}$, $\{A^1_i\}$, confidence level $\alpha$, fraction of the learning curve to be computed $\gamma$
**Run** $G^m$ **training for** $n = \gamma N$ iterations
$G^m \rightarrow \{A^m_1, ..., A^m_n\}$
**Compute** $p$**-value of Bartlett test**
$p = Prob(Var(\{A^m_i - A^0_i\}^n_{i=1}); Var(\{A^1_i - A^0_i\}^n_{i=1}))$
**Alert about a bug if:**
$p \leq \alpha$
```

Table 1. Testing normality of difference between first 20% of learning curves of valid runs of deep learning training by Shapiro-Wilk test. P-values shown for 6 pairs of single node AlexNet runs and 3 pairs of single node GoogleNet runs

pair index	p-value of AlexNet	p-value of GoogleNet
1	0.64	0.15
2	0.57	0.96
3	0.78	0.87
4	0.60	-
5	0.84	-
6	0.52	-

$$Prob(Var(\{A^m_i - A^0_i\}); Var(\{A^1_i - A^0_i\})) \leq \alpha \quad (1)$$

This approach may be applied in the early stages of software optimization, as it requires only two valid runs. It does not require training a sophisticated predictor specific for the implemented algorithm G. The method gives a probabilistic prediction of the software bug with a controlled false alarm rate α using only a $\gamma = \frac{n}{N}$ fraction of the training run. We summarize the algorithm which we dub *Software Bug Prediction by Bartlett Test* (SBPBT) in Algorithm 1

3. Experiments

We performed a number of experiments to justify the SBPBT numerically. First we checked the assumptions of normality of differences of corresponding partial accuracies between valid runs. Then we compared predictive performance of Bartlett test with other statistical tests. Finally we applied the SBPBT to real-world single and multinode deep learning training runs performed by software optimization teams for testing their implementation.

3.1. Normality of differences between beginning of learning curves

Using the Bartlett test assumes that under null hypothesis the differences between beginnings of (valid) learning curves are normally distributed: $A_i^0 - A_i^1 \sim N(\mu, \sigma)$. We used first 20% of training runs for AlexNet (Krizhevsky et al., 2012) and GoogleNet (Szegedy et al., 2015) in the single node implementations. As we had 12 valid training runs for AlexNet and 6 valid runs for GoogleNet we randomly divided them into 6 and 3 pairs correspondingly and checked the normality of the difference between runs in each pair by the Shapiro-Wilk test (Shapiro & Wilk, 1965). In our experiments all differences were found to be normal, see p-values in Table 1. This demonstrates empirically the validity of Bartlett test usage for measuring similarity between differences of learning curves.

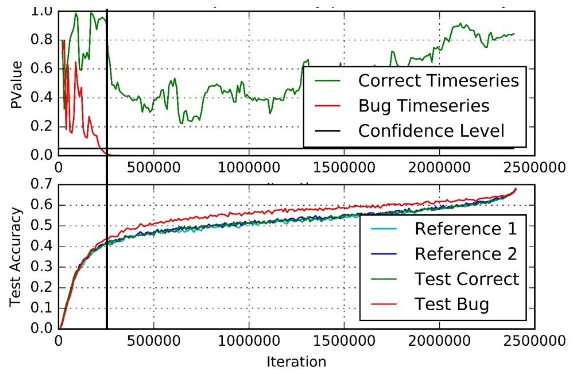

Figure 1. Distinguishing a correct learning curve from a faulty learning curve. Bottom part: learning curves of 2 valid reference runs of GoogleNet single node training, a correct run and a faulty run. Top part: p-value of the Bartlett test comparing the differences between the beginnings of the learning curves

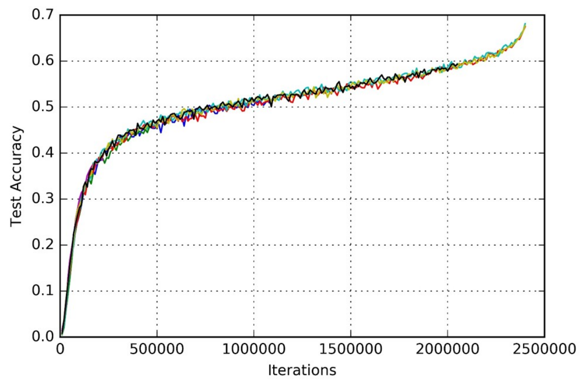

Figure 2. Valid learning curves of GoogleNet single node training

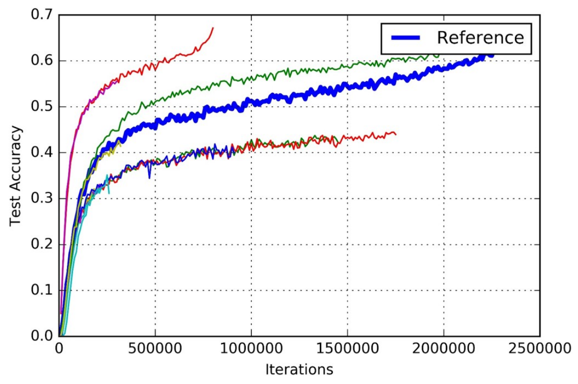

Figure 3. Faulty learning curves of GoogleNet single node training. A valid 'Reference' run is shown to visualize deviations of the faulty learning curves from a valid run

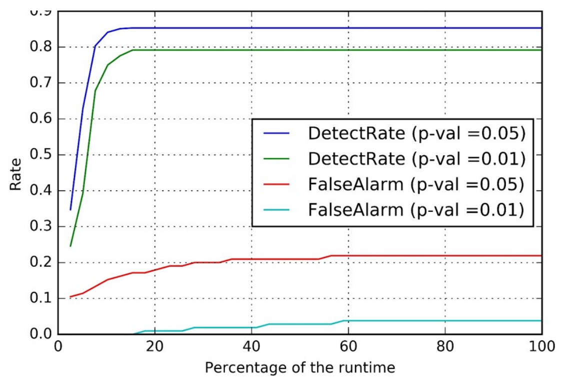

Figure 4. Effect of changing γ and α of SBPBT on the detection rate and false alarm rate. GoogleNet single node implementations

3.2. Motivating example

We begin with a motivating example of differentiating between a correct and faulty training runs for GoogleNet single node implementation, see Figure 1. In the bottom part of the figure we provide learning curves for two a priori valid runs, called 'Reference 1' and 'Reference 2' and two runs to be predicted, the correct and the faulty, which we dub 'Test Correct' and 'Test Bug' correspondingly. First we compared difference between 'Reference 1' and 'Reference 2' from one hand and the difference between 'Reference 1' and 'Test Correct' runs from the other. We computed a whole time series of Bartlett test p-values ${p_i}_{i=1}^N$, where p_i is computed by considering the first i iterations of the training. We dub this time series as 'Correct Timeseries', see the top part of the Figure 1. In the same way we computed the 'Bug Timeseries' comparing differences between 'Reference 1' and 'Reference 2' from one hand and difference between 'Reference 1' and 'Test Bug' runs from

Table 2. SBPBT prediction performance. Using $\gamma = 20\%$ of the training run at confidence level $\alpha = 0.01$

Dataset	Valid runs	Faulty runs	Number of samples N	Detection rate	False Alarm rate
Alexnet SingleNode	12	2	45	100%	1%
Googlenet SingleNode	7	12	240	79%	1%
Googlenet MultiNode I	2	104	48	89%	-
Googlenet MultiNode II	2	45	48	88%	-

the other. It is easy to see that starting approximately from 10% of the run the Bartlett test correctly predicts whether a current run is going to fail at confidence level of 5%, i.e. $\gamma = 0.1$ and $\alpha = 0.05$. It is worth to notice that at first 10% of the run the differences between learning curves are small, see the left part of the bottom graph in Figure 1, and Bartlett test is able to make this difficult prediction by accumulating the differences for the whole beginning of the learning curves.

3.3. Application to AlexNet and GoogleNet multiple runs

To assess SBPBT performance in early detection of software failures we have examined logs of deep learning runs using four model implementations: single-node AlexNet and single-node GoogleNet and two multi-node GoogleNet with two different mini-batch sizes. These logs are of training runs performed by software optimization teams.

At each experiment, we tagged the runs as valid or faulty, according to their final accuracy, number of iterations required to converge, and behavior during the run. The best results with similar behavior were labeled as valid, while the rest as faulty, see examples of valid runs for single-node implementation of GoogleNet in Figure 2 and faulty runs in Figure 3. In each case we assessed the prediction performance of SBPBT in terms of detection rate. In cases where we had more than two valid runs for the implementation we assessed the false alarm rate of SBPBT as well.

In each case we chose a pair of valid runs to be the reference runs, and then we applied SBPBT to the first 20% of learning curves of remaining runs at a confidence level of 1%. Detection rate then was measured as a fraction of correctly classified faulty runs and the false alarm was measured as a fraction of mistakenly classified remaining valid runs. In order to gain statistics we repeat the experiments for all possible pairs of valid runs to be the reference in each case. The results of the experiments showed that we are able to detect at least 79% of faulty runs. The false alarm rate was as expected 1% in the relevant cases.

To examine the possible effect of changing the fraction γ of the beginning of the learning curve and the impact of confidence level α, we varied both quantities for the example of single node GoogleNet implementations, see the resulting detection rate and false alarm curves in Figure 4.

Table 3. Comparison of methods to measure differences between learning curves deviations. AlexNet and GoogleNet single node implementations

	AlexNet		GoogleNet	
	Detection	FA	Detection	FA
Bartlett	100%	1%	79%	1%
Mann-Whitney	70%	1%	95%	33%
T-test	86%	3%	91%	38%

3.4. Alternative methods to measure differences between learning curves deviations.

We compared other methods to quantify similarity between differences of learning curves, instead of comparison of variances. We compared the Bartlett test with the t-test (Welch, 1947) that measures the similarity of sample means assuming normality and the Mann-Whitney test (Mann & Whitney, 1947), checking the similarity of sample medians without assuming normality. We applied all three tests to runs of single node implementations of AlexNet and GoogleNet, see Table 3. The Bartlett test showed the best results in terms of detection rate and false alarm rate, that supporting our choice for the SBPBT method.

4. Conclusion

We introduced a novel simple method for predicting faults in software implementations of deep learning training algorithms. We based it on comparison of variances of differences between partial accuracies time series of valid runs and the current run. The resulting algorithm requires neither a sophisticated training stage nor a significant amount of training data. It is easily tuned for a desired false alarm rate using the p-value of the Bartlett test.

Our assumptions about validity of Bartlett test for this case were supported empirically. We demonstrated that using only 20% of the beginning of training run one could detect most of the of faults in the datasets we had, while maintaining a low false alarm rate of 1%.

References

Bai, C.G., Hu, Q.P., Xie, M., and Ng, S.H. Software failure prediction based on a markov bayesian network model. *Journal of Systems and Software*, 2005.

Bishop, P. and Pullen, F. D. Probabilistic modeling of software failure characteristics. In *Proceedings of the IFAC Workshop SAFECOMP*, 1988.

Cai, K.Y., Cai, L., and Wang, W.D. On the neural network approach in software reliability modeling. *Journal of Systems and Software*, 2001.

Domhan, T., Springenberg, J. T., and Hutter, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, 2015.

Goel, A.L. Software reliability models: assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, 1985.

Ho, S.L., Xie, M., and Goh, T.N. A study of the connectionist models for software reliability prediction. *Computers and Mathematics with Application*, 2003.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

Li, Lisha, Jamieson, Kevin G., DeSalvo, Giulia, Rostamizadeh, Afshin, and Talwalkar, Ameet. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016. URL http://arxiv.org/abs/1603.06560.

Mann, H. B. and Whitney, D. R. ”on a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 1947.

Pfefferman, J.D. and Cemuschi-Frias, B. A non-parametric non-stationary procedure for failure prediction. *IEEE Transactions on Reliability*, 2002.

Reussner, R.H., Schmidt, H.W., and Poernomo, I.H. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 2003.

S. Ozekici, R. Soyer. Reliability of software with an operational profile. *European Journal of Operational Research*, 2003.

Shapiro, Samuel Sanford and Wilk, Martin B. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

Snedecor, George W. and Cochran, William G. *Statistical Methods*. Iowa State University Press, 8 edition, 1989.

Szegedy, C., L., Wei, Y., Jia, P., Sermanet, S., Reed, D., Anguelov, D., Erhan, V., Vanhoucke, and A., Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

Welch, B. L. The generalization of student’s problem when several different population variances are involved. *Biometrika*, 1947.

Xie, M. Software reliability models: a selected bibliography. *Journal of Software Testing, Verification and Reliability*, 1993.