Logfile Failure Prediction using Recurrent and Quasi Recurrent Neural Networks

Austin Jiao
NVIDIA Corporation
Santa Clara, CA
ajiao@nvidia.com

Isuru Daulagala
NVIDIA Corporation
Santa Clara, CA
idaulagala@nvidia.com

Abstract

Inumerable man hours are spent waiting for running programs to finish and then debugging them when they crash by analyzing the logfile. Often times logfiles will contain a plethora of text extraneous to debugging, providing for a tedious undertaking. Therefore, we present an application of state-of-the-art natural language processing with neural networks to classify a run's final outcome prior to it occurring. Variations of pointer sentinel mixed models and quasi-recurrent neural networks were utilized for this task. The implemented models are first benchmarked on WikiText-2 language modeling then trained on NVIDIA logfile data. To demonstrate the model's ability to classify before the run ends, F1 scores are plotted against proportion of logfile consumed and are compared to a TF-IDF baseline.

1 Introduction

The Physical Design phase of Very Large Scale Integration (VLSI) or Integrated Digital Circuit design is composed of multiple optimization algorithms converting an Register-Transfer Level (RTL) netlist into a physical design containing physical information about all components including wires, transitors and capacitors. These algorithms are also known as Electronic Design Automation (EDA) algorithms and are often propietary and provided by 3rd party vendors as part of a suite of EDA tools. The engineers who work on designing and optimizing a chip (Physical Designers), execute multiple algorithms in a pipeline to meet design targets. Each of these executions are known as a "run". During each run, the EDA tool dumps a logfile which summarizes certain warinings, errors as well as other statements such as results. At NVIDIA, runtime for individual stages is kept to a minimum but if something goes wrong, a run can span multiple days. Since the EDA tool and the algorithms it uses are propietary, the only indication that a run might not meet the optimization objectives or "fails" before it ends is through the logfile.

A Physical Design Engineer might be able to deduce the reason why a specific run failed by analyzing the logfile. Yet, this task is not trivial and can take a considerable effort and time. If a failed run could be detected early in the runs life, a significant amount of man hours could be saved from the early termination of these runs. Moreover, the key lines that warranted the failure classification could provide insight into the error's cause. One method used to detect run failure is to maintain a blacklist of regular expressions matching fail strings acquired through experience. However, these lists grow fast and become unwieldy to maintain. Another method, the most popular traditional method, is to use TF-IDF to classify whether a running program will crash fatally. Both of these approaches are limited in their ability to deal with new messages that indicate failure. In order to surpass TF-IDF, we applied recent advancements in natural language processing using deep learning with the hope that these models can infer something from logfiles that goes beyond critical string searching.

We will apply language models to directly predict the logfile label. The prediction is based on two recent language modelling architectures: Pointer Sentinel Mixed Models (PSMM) [4] and Quasi-Recurrent Neural Networks (QRNN) [3]. These models were first validated against the published results of each model using the wikitext2 dataset [4].

2 Background

In last year's session of CS224D, peers from our NVIDIA team attempted logfile anomaly detection using RNNs [7]. The aim of their project was to train a language model on logfiles from successful runs and detect a failing run when the model's prediction accuracy drops. They were able to achieve an F1 score of 0.501 using their best model, a simple character based RNN. This result was in spite of testing using word vectors as well as more complex LSTM and GRU models.

Unlike the previous attempt, which suffered from the fact that they could only collect successful logfiles, our models directly predict the logfile label, fail or success. In our attempt to accomplish this, we replicated and applied variations of two state-of-the-art neural network architectures: Pointer Sentinel Mixed Model (PSMM) and Quasi-Recurrent Neural Networks (QRNN).

QRNNs have shown to have performance comparable to LSTMs with considerable speedup [3]. This is achieved by applying masked convolutions with multiple filter matrices across multiple time steps of input to simulate the gates of an LSTM.

$$\begin{aligned} Z &= tanh(W_z * X) \\ F &= \sigma(W_f * X) \\ O &= \sigma(W_o * X) \end{aligned} \tag{1}$$

Then, by taking and combining slices of the resulting "gate matrices", the states and outputs at multiple time steps are generated.

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + (1 - f_t) \odot z_t \\ h_t &= o_t \odot c_t \end{aligned} \tag{2}$$

Because this model lacks backpropagation through time from not having trainable variables in the recurrent cell, the flow graph execution can take place primarily on GPU.

Pointer Sentinel Mixture Models (PSMM) [4] use a mixture gate to decide between a Pointer Network [6] and a traditional LSTM. This mixture gate is referred to as a sentinel and is passed onto the Pointer Network as an extra element. If the Pointer Network is not confident enough in predicting the output, the sentinel prompts the network to use the tradtional LSTM.

$$P(y|x) = gP_{rnn}(y|x) + (1 - g)P_{ptr}(y|x) \tag{3}$$

Because PSMMs are able to hone onto words in the input context, they perform much better at predicting words that occur rarely than regular LSTMs. In terms of predicting logfile failure, if a run is going to fail, it is reasonable to assume that the logfile's words would change considerably after the point that failure is inevitable. Therefore, it makes sense to look at the last few words to predict if a failure is about to occur.

3 Approach

3.1 Word vector generation

Like in the previous attempt [7], our approach to this problem was to tokenize word inputs. Given that the previous team didn't opt to use these word vectors due to poor performance, we had to make modifications to their methodology. Physical design domain knowledge was used, increased from before, replacing file locations, block names, and user names with generic token words. Any special characters or punctuation was removed. Next, rather than simply tokenizing the words, we opted to pre-train the word embeddings. Using a sample of 5000 logfiles, a vocabulary of 100K pre-trained embeddings were generated by training a GloVe model [5]. GloVe was used because logfile data is not exclusively natural language and therefore does not have a consistent grammar that would justify using skip-gram. Furthermore, GloVe computation is fast.

3.2 Models

3.2.1 Quasi-Recurrent Neural Network

In our implementation of the QRNN, we followed the model as outlined by [3], with dropout. Feeding in the data also required extra consideration. Every logfile is assigned to a batch index and data is continually fed into the batches with the final hidden state provided as the initial state on the next batch iteration. Losses are calculated on the final output state of the sequence rather than all states. When one logfile is finished training, its batch's hidden state is then reset to zero. In addition, since there are multiple time steps of outputs, one extension that can be made to the standard QRNN model is to add a pointer network. The pointer network calculates attention across all previous time steps in the sequence length window and outputs the hidden state with the highest attention instead of the final hidden state. We evaluated three QRNN architectures on logfile classification, 1 layer, 3 layers, and 3 layers with pointer sentinel. An unfavorable outcome of adding layers is that it increases training time significantly because each output state must be sequentially calculated and then fed into the inputs for the next layer. One way to alleviate this is to use a large batch size and distribute the training across multiple GPUs. Similar to the described method in the tensorflow CIFAR10 tutorial [1], all variables and weights were created on the CPU while the entire flow graph is executed on each GPU to produce multiple ”towers” of gradients. The gradients are then collected, clipped, and applied to the variables by the CPU, utilizing its lower data latency. When training on 8 Tesla P100 GPUs with each GPU taking a batch size of 64, both 3 layer QRNN models completed 20 epochs in 3 hours when trained on the WikiText-2 dataset. The same training took 2 days on a single GPU.

3.2.2 Pointer Sentinel Mixture Models

The Pointer Sentinel Mixture Model used to predict logfile classfication closely followed the implementation of [4]. Similar to the paper, we used a word window length (L) of 100 for the Pointer Network while also using both 1 and 2-layer LSTMs to calculate the Softmax vocabulary probabilities.

The training on 1 Titan XP took 6 days to complete for the logfiles for 50 epochs while taking 9 hours on the WikiText-2 dataset.

3.3 Gradient Clipping and Learning Rate

For all models described above except for the 1-layer QRNN, we observed exploding gradients. Therefore all of the models except the 1-layer QRNN was implemented with gradient clipping. Gradients were clipped at 5 for all of these models. For any network with large number of hidden layers the gradients will tend to explod or vanish due to gradient instability.

Likewise for the model to not get stuck in a local minima, we used a high learning rate at the beginning and decreased it using learning rate decay.

3.4 TF-IDF Baseline

To compare these results against a non-deep learning approach, we used TF-IDF. For each class (fails and sucesses), a TF-IDF vector was trained. Then a Naive-Bayes Classifier based on the TF-IDF vectors was used to predict if each logfile in the test set was success or fail at different points in the logfile.

Table 1: Reported and Replicated Perplexity Values for the wikitext2 dataset

Model	Parameters	Published Results		Replicated	
		Valid	Test	Valid	Test
PSMM	21M	84.8	80.8	86.8	82.0
1-Layer QRNN	37M	Unreported		105.4	102.3
2-Layer QRNN	40M	Unreported		89.9	86.0
3-Layer QRNN	62M	Unreported		76.2	73.3

4 Experiment Results

4.1 Dataset

To provide a dataset, 90K logfiles were collected, 30K from successful runs and 60K from fatal ones. These logfiles were accumulated over the last 3 months and are taken from a variety of VLSI design stages. Unfortunately the data distribution is skewed towards failed cases because usually such logfiles saved for debug purposes. However, from experience, we know that successful runs account for 0.75 of cases; thus, to simulate this environment we trained and validated by generating datasets that combined the successful run logifles with sampled fatal logfiles in a ratio of 3:1. The data was further split into training, validation, and test sets containing 80%, 10%, and 10% of the logfiles, respectively.

4.2 Wikitext2

To validate the network architectures as language models, we benchmarked them using the Wikitext2 dataset outlined and provided by [4]. For all models a hidden size of 650 was used while the window L for the PSMM was 100.

4.3 Computational complexity

Despite the speedup possible when utilizing multiple GPUs, we found the largest contributors to long runtimes were the CPU tasks of tokenization and batch creation. When the logfile size ranges from 15KB to 100MB it isn't feasible to load the entire tokenized set of 50K logfiles into memory. Instead, we opted to batch and compute asynchronously. The CPU limitation became especially noticeable when increasing batch sizes, the model spends a large proportion of total execution time waiting for the next batch after GPU computation completed. This combined with the size and number of logfiles meant that we weren't able to train as many epochs as we would have liked. The total number of epochs was 5 for both 1 layer QRNN and 1 layer PSMM, both of which reached fairly stable loss convergence. On the other hand, the more complex models took longer to train and as a result had to be stopped before convergence was reached. For 2 layer PSMM, only 1 epoch of training could be completed in time. Similarily, for both standard and pointer sentinel versions of 3 layer QRNN, only 2 epochs of training were completed. Thus, the results for the evaluation of these models are not concrete representations of their final efficacy to act as logfile predictions. Instead, given this limitation, if any model is able to show promising then there is an impetus for further exploration.

4.4 F1 scores by logfile percentage

The principle goal of this project is to predict a run failure early on in the logfile. To evaluate this, the plot we look at is the F1 score by percent of the logfile already seen by the network which is shown in Fig. 1 Through this plot, it is possible to ascertain the accuracy of model as it steadily gains more information. The F1 score used is the weighted average of F1 scores for each label. The networks are compared in conjunction with TF-IDF as a baseline.

All the models performed better than the baseline at all points of the logfile. This alone justifies the use of natural language processing and deep learning to predict run outcomes based on logfiles. Now, experiments on real-time prediction and implementation on a production environment can be undertaken. Among all the models tested, the PSMM model performs the best, likely due to

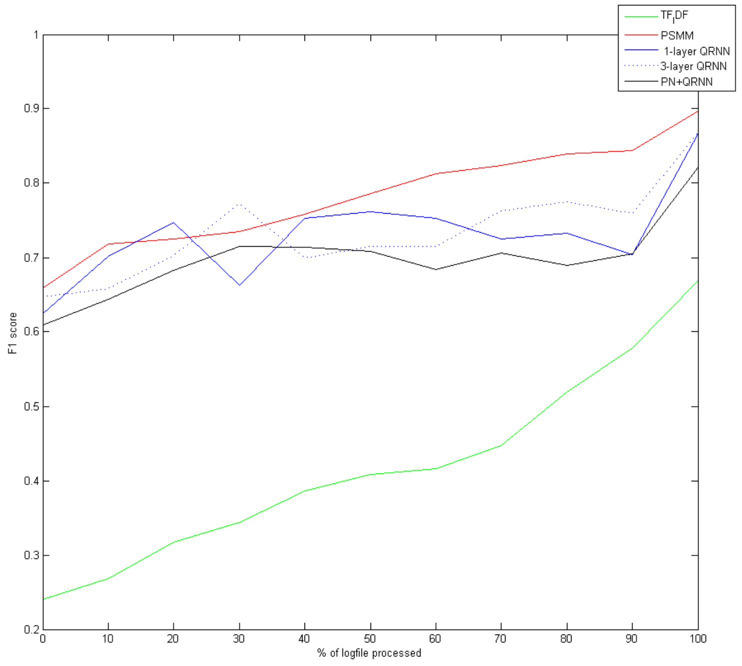

Figure 1: Variation of F1 scores by percentage of logfile processed.

its ability to identify words that contribute to failures based on the current inputs. Thus, one may posit that the QRNN with a Pointer Network attached should have performed better than the vanilla QRNNs; however, this was probably because it was only trained for 2 epochs, so instead it performs slightly worse overall. The 3-layer QRNN without pointer network trained for the same amount of time but performed better than the same model with a pointer network. As is demonstrated in [4], pointer networks are slow to train. So, it is entirely possible that, at low epochs, the bare QRNN will out perform the pointer network version. The 1-layer QRNN F1 scores lags the 3-layer QRNN's a majority of the time, which is expected since the 3-layer QRNN has more parameters. Another characteristic of the 1-layer QRNN is that it quickly reaches 0.7 F1 and hovers there until it reaches the end. In contrast, the deeper models exhibit more linear relationship with percentage of logfile processed. The increased parameters presumably maintain the historical context of the logfile much longer and models with Pointer Networks exhibit this characterestic more, even the QRNN although it performs worse overall. All the deep learning models show a sharp spike when going from 90% to 100% of logfile processed. This is probably because many lines are outputted at the end of the logfile that directly indicate if a run has failed or succeeded. This spike in F1 scores also validates the efficacy of the models. What is particularly surprising is that TF-IDF didn't experience as significant spike in accuracy meaning the label of a logfile must have a contextual component that depends on more than the last few lines of a logfile.

5 Conclusion

The first conclusion gathered from our experiments is that using QRNNs rather than LSTMs offers significant speedup. Running on a single GPU, training was approximately 3 to 4 times faster running the QRNN for the same tasks. Given that QRNNs benefit appreciably from distributed GPU execution due to less sequential operations, the actual speedup with more computing resources can be even more prominent.

Comparing F1 scores from all models versus TF-IDF shows that the neural networks are able to better predict the logfile label across at any point in the logfile. This by itself is a significant achievement because it validates the hypothesis that word based natural language processing with deep learning can be used to classify run outcomes. Beyond this, all models are able see a boost in prediction accuracy by the end of the logfile. This is an expected result as log messages that literally state the run is going to fail often show up at the end of the logfile; thus, acting as a sanity check that basic inferences were able to be gathered through training. Moreover, the deeper networks, 3 layer QRNN with and without pointer sentinel and the PSMM, exhibit a greater linear relationship between F1 score and proportion of logfile seen by the network. One possible interpretation of this is that these deeper models are able to maintain signals from farther back in the file, making a classification by taking into account the history of words it has previously seen. Hopefully, this effect becomes more prenounced given a longer training time.

6 Future Work

Due to the limitations of the time taken to takenize each logfile, training times for models are limited by the CPU. Therefore the multilayer QRNN models ran only 2-epochs. Without training time limitation, it would be possible to train the models proposed in this project for longer and with a larger dataset.

Apart from classifying a logfile is a fail or a success, we also intend to classify fails into categories such as variable errors, syntax errors and dependency errors. These categories are generated after the run is completed using the stackrace of the run. If the category of a failure can be predicted before a run is completed using the logfile, the run can be restarted fixing the errors.

In the future, we intend to bring the logfile failure prediction to production by predicting the failure or succcess of runs in real time.

Acknowledgments

We would like to thank our project mentor Dr. Richard Socher for providing valuable insight and guidance throughout the project. We would also like to thank Stephen Merity for providing guidance regarding the PSMM and QRNN models.

We would also like to acknowledge the use of the structure used to generate TensorFlow models using the PA3 starter code from the class

References

[1] Training a model using multiple gpu cards. `https://www.tensorflow.org/tutorials/deep_cnn#training_a_model_using_multiple_gpu_cards`. Accessed: 2017-03-12.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] J. Bradbury, S. Merity, C. Xiong, and R. Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

[4] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[5] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[6] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[7] T. Yang and V. Agrawal. Log file anomaly detection. *CS224d Fall 2016*, 2016.