DNN for Program Behavior Prediction

Shantanu Chandorkar
NCSU
sachando@ncsu.edu

Abstract

Predicting program behaviors is important for guiding runtime optimizations. This project, investigates the use of DNNs for predicting program behaviors, and explores ways to accelerate the prediction.

Keywords DNN, Optimization

1. Motivation

Predicting branches or functions taken during an execution of a program, can help optimize memory and cache to speed up the execution or reduce memory usage of the program in context. The prediction can be done using LSTM or variants which have been shown to understand and model temporal dependencies in input sequences. This could help model a non intuitive access pattern a program may undertake.

2. Objectives

1. Collect traces for 7 programs in cpu2017 using a pin tool to record branch sequences and function call sequences. The traces were for 100 iterations of each program.
2. Model program behaviour with traces collected using LSTM variants for both function call sequences and branch sequences.
3. Optimize prediction using data transformation (1) and model optimization (2) to speed up the model.

3. Challenges

1. Dealing with large volume of trace. It was difficult to determine how much and which part of trace to keep.
2. Determining the structure of model, input data and parameters.
3. Difficulty in implementation of ideas presented such as those in (1) and (3).

4. Solutions

4.1 Pin tool

Pin is a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools (4).

Using pin, the cpu2017 programs were instrumented by inserting calls before and after routines and for direct branches. Using pin APIs (11) such as InsertPredicatedCall and IsBranch, the tool was developed.

The implemented tool loads all images that the main image loads such as libraries and can optionally include them while recording call and branch sequences. It also generates a file that maps all function IDs to their names and images. This can be used to decode the collected call sequences.

4.2 Data generation

Using the previously described pin tool, a bash script was written to run each program with some input and generate the relevant traces for 100 iterations of each program.

To add to some degree of variance, some of the program inputs depended on the iteration count such as imagick rotation or shear while some had a program with random outcomes such as makerand.pl. All the generated files were trimmed to first 50,000 lines each. Producing a trace of 5×10^6 lines for each branch and function call sequence for every program.

5. Algorithm

5.1 LSTM Encoder Decoder

Encoder-Decoder LSTM is an RNN designed to address sequence-to-sequence problems, sometimes called seq2seq. An LSTM cell has and three gates that manipulate of the flow of information inside the cell. These gates are input gate, output gate and a forget gate. The input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the

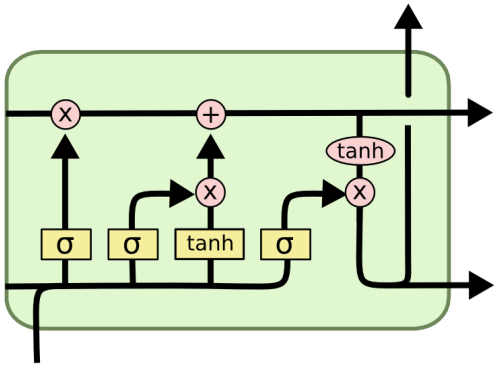

Figure 1. LSTM cell (10)

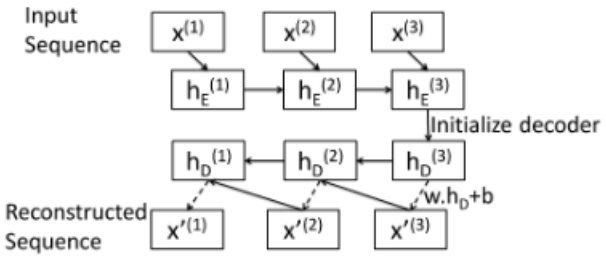

Figure 2. LSTM Encoder Decoder Approach
x^i is the time series
h_e^t is the final state of encoder
h_D^t is the initial state of decoder obtained from h_e^i and x^i

LSTM unit (6).

The prediction problem can be posed as a translation problem to use this model. This architecture is comprised of two models, one for reading the input sequence and encoding it into a fixed-length vector, or a latent space, and a second for decoding the latent space representation and outputting the predicted sequence (5).

For each program, 2 models are created, one for branch trance and one for call sequence trace. I therefore trained and evaluated a total of 14 models. The structure of models is different for branch and call sequence.

The call sequence takes in a 2 dimensional input <Function ID, i/e> whereas the branch model takes only <Branch ID> as input units. These are taken in the form of a time series. To generate the time series use keras' TimeSeriesGenerator. (7).

```
Layer (type)                 Output Shape
=================================================
lstm_1 (LSTM)                (None, 3, 16)

dropout_1 (Dropout)          (None, 3, 16)

lstm_2 (LSTM)                (None, 3, 16)

dropout_2 (Dropout)          (None, 3, 16)

flatten_1 (Flatten)          (None, 48)

dense_1 (Dense)              (None, 2)
=================================================
Total params: 3,426
Trainable params: 3,426
Non-trainable params: 0
```

Figure 3. LSTM Encoder Decoder variant used

Therefore a window of default length 3 is created to generate the input data. This acts like a time series and can be passed to the lstm encoder decoder for training. This is a parameter, the optimum value of which, along with the model hyper-parameters, could only be determined after hyper-parameter tuning which has been addressed as a remaining issue in section 7.

5.2 Model Training and Instrumentation

For training, a total of 2.5×10^6 lines of trace for both function call and branch sequence is used. This is first converted into a time series as descibed previously.

For training all the models, lin05 was used and the trained models weights, along with the class pickle files were generated in the results section.

5.3 Evaluation

Evaluation was done using a sample of the trace collected for each model. If the models are trained, they are loaded and evaluated on a fraction (40%) of collected data. This data is also converted into a time series of length 3 and dimentions 1 for branch trace and 2 for functionc all sequence.

5.4 Optimiztions

For initial attempts to optimize, I tried changing the length of timeseriesgenerator, using sgd or rmsprop in place of adam or mse. After running a combination of diffrent layer number and unit sizes on small training set, I was not able to reduce the loss significantly. I also tried replacing LSTM cells with GRU cells for speed up. Increasing or decreasing size of the model or number of epochs with higher patience on callbacks (12) did not reduce the loss significantly.

After a lot of trial and error and going through (3) it became evident the the problem could be stated with data as a categorical in nature. And using categorical cross entropy (13) could result in better performance. This is discussed in more details in Section 5 and 7.

When I tried in add the vector as size of LSTM and dense layer, the execution got halted during model generation probably due to memory consumption. This could mean that reducing the model complexity was necessary to implement this part.

For speeding up training and prediction, using data transformations such as (1) could help.

The following lists all the parameters that could be changed.

1. Window Length.
2. Decoder output length.
3. Size of LSTMs.
4. Number of LSTM layers.
5. Size of Dense layer.
6. Activation function of each layer.
7. Loss Function.
8. Optimizer.
9. Type of Cells, LSTM/GRU.

6. Lessons and Experiences

The project provided valueable lessons and a steep learning curve in deep learning and binrary instrumentation.

The were a lot of lessons apart from the hands on experience in the field. Some of them are described below.

6.1 Data is categorical in nature

Initially implemented the task as regression which resulted in high loss. After reading (3), I realized that treating the data categorically with categorical cross entropy as loss and a softmax activation for dense layer would have resulted in better performance.

```
49984   <4682768, i, #4>
49985   <5304336, i, #52>
49986   <5303904, i, #1>
49987   <5303904, e, #21>
49988   <4682768, e, #14>
49989   <4822128, i, #4>
49990   <4822128, e, #20>
49991   <4799248, i, #4>
49992   <4758576, i, #6>
49993   <4758576, e, #41>
49994   <4799248, e, #29>
49995   <4823616, i, #4>
49996   <4823616, e, #24>
49997   <4851792, i, #4>
49998   <4851792, e, #20>
49999   <4821376, i, #4>
50000   <4821376, e, #12>
```

Figure 4. Sippet of Function trace for 600.perlbenc
There are 100 files with limit of 50,000
These are generated for every program

Due to absence of deep learning background, this was an important lesson.

6.2 Model Complexity is not proportional to performance

Initially I used an encoder decoder model with 2 more LSTM layers and higher number of units. While this resulted in a longer training duration, it did not reduce the loss. I also tried replacing LSTM cells with GRU cells or adding a few more dense layers but it did not improve the loss significantly.

6.3 Variance in data is critical

The traces I initially collected where those of a single program being run again and again, for instance running test.pl with perlbench. All the traces collected when trimmed to a certain run duration looked similar. Though I did not quantitatively evaluate the variance in data, it seemed that using different input files would have made the data more robust.

```
49989   <5293176, #8>
49990   <5293218, #1>
49991   <4332546, #54>
49992   <4334835, #3>
49993   <4332760, #16>
49994   <4332834, #11>
49995   <4334885, #3>
49996   <4381852, #23>
49997   <5282317, #41>
49998   <5275985, #21>
49999   <5268593, #18>
50000   <5268619, #3>
```

Figure 5. Sippet of Branch trace for 600.perlbenc
There are 100 files with limit of 50,000
These are generated for every program

7. Results

To generate the results, 40% of the trace collected (5×10^6) was used, for each model. This data was taken sequentially broken using sklearn's train_test_split (8). The saved models are loaded and each model's accuracy and loss as mean squared error is calculated and stored in results.

To evaluate time series data, we use keras's evaluate_generator (9). The same parameters for generator are used as were for generating training data. The results obtained are tabulated in Table 1.

8. Remaining Issues

1. Phase 3 Optimization. Implementation of ideas expressed in (1) and (3).
2. Parameter tuning after Phase 2. Could not tune parameters due to high loss. I tried changing a number of different parameters to see if loss over epochs reduced or accuracy over epochs improved. But no parameter change seemed to affect it significantly.
3. Through model evaluation. Currently only scores model on sampled test data on accuracy and loss. I was not able to understand the generated accuracy as the function models were assigned an accuracy of 1.0 and others, 0.0.

9. Possible Solutions

1. Change currently implemented model to (3) with reduction in model complexity.
2. Transform input data as described in (1) to improve performance.

Model	MSE Loss	Accuracy
600.perlbench branch	1.58×10^{12}	0.0
600.perlbench function	11.6×10^{12}	1.0
602.gcc branch	8.41×10^{12}	0.0
602.gcc function	8.37×10^{12}	1.0
605.mcf branch	17.6×10^{12}	0.0
605.mcf function	0.56×10^{12}	1.0
619.lbm branch	0.85×10^{12}	0.0
619.lbm function	0.51×10^{12}	1.0
638.imagick branch	27.3×10^{12}	0.0
638.imagick function	8.87×10^{12}	1.0
641.leela branch	1.44×10^{12}	0.0
641.leela function	0.49×10^{12}	1.0
644.nab branch	0.91×10^{12}	0.0
644.nab function	8.88×10^{12}	1.0

Table 1. Scores generated by (9)

3. Try parameter tunining with a different metric and smaller training dataset.

10. Related Work

RNNs have been used to model program behaviours and optimize performance. Some of the papers in the field cover optimizations such as cache optimization or prefetching.

(14) and (15) propose two different vartieties of LSTMs for prefetching inside a processor.

(16) uses an LSTM-NN as a popularity predictor and then manages the cache as a priority queue where contents with the smallest predicted popularity are evicted on cache misses.

(17) and (18) use reinforcement learning or RL to modify the caching policy dynamically. A reinforcement learning algorithm tries optimize policy based on rewards that action results in by a trial-and-error approach mostly augmented with some search heuristics.

(19) and (20) apply machine learning to program phase change detection and phase prediction.

(3) learns objects access patterns for prefetching program counter's memory address in order to avoid on-chip misses. The problem is treated as sequence classification solved using LSTMs.

Perceptron branch predictor (21) uses a linear classifier to predict whether a branch is taken or not taken. It is an online model that learns incrementally by adjusting weights based on outcome of jumps.

11. Conclusion

The idea of the project has a lot of potential. If implemented correctly, it could provide significant performance gains.

The ideas covered in literature survey and the hands on experience was valuable. Going through all the ideas and possible opimizations has been very interesting and exciting.

References

[1] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. Efficient docu-ment analytics on compressed data: Method, challenges, algorithms, insights. InProceedingsof the 44th International Conference on Very Large Data Bases (VLDB), 2018

[2] Lin Ning, Hui Guan, Xipeng Shen. Adaptive Deep Reuse: Accelerating CNN Training on the Fly.

[3] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis,and P. Ranganathan, Learning Memory Access Patterns, in InternationalConference on Machine Learning (ICML), 2018.

[4] `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

[5] `https://keras.io/examples/lstm_seq2seq/`

[6] Wikipedia contributors, "Long short-term memory," Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=895314352` (accessed May 6, 2019).

[7] `https://keras.io/preprocessing/sequence/`

[8] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`

[9] `https://keras.io/models/sequential/#evaluate_generator`

[10] `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

[11] `https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/group__INS__BASIC__API__GEN__IA32.html`

[12] `https://keras.io/callbacks/#EarlyStopping`

[13] `https://www.dlology.com/blog/how-to-choose-last-layer-activation-and-loss-function/`

[14] Y. Zeng and X. Guo, Long short term memory based hardware prefetcher: A case study, inProceedings of the International Symposium on Memory Systems, MEMSYS 17, (New York, NY, USA),pp. 305311, ACM, 2017

[15] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. E.Kozyrakis, and P. Ranganathan, Learning memory access patterns,in Proc. of the International Conference on Machine Learning (ICML),2018.

[16] H. Pang, J. Liu, X. Fan, and L. Sun, Toward smart and cooperativ eedge caching for 5g networks: A deep learning based approach, in Proc. of IEEE/ACM International Symposium on Quality of Service(IWQoS), 2018.

[17] C. Zhong, M. C. Gursoy, and S. Velipasalar, A deep reinforcement learning-based framework for content caching, in Proc. of the 52nd Annual Conference on Information Sciences and Systems (CISS), 2018.

[18] E. Rezaei, H. E. Manoochehri, and B. H. Khalaj, Multi-agent learning for cooperative large-scale caching networks,ArXiv e-prints, 2018.arXiv:1807.00207

[19] M. Chiu and E. Moss. Run-time program-specific phase prediction for Python programs. In Proceedings of the 15th International Conference on Managed Languages Runtimes

[20] M.-C. Chiu, B. M. Marlin, and E. Moss. Real-time program-specific phase change detection for Javaprograms. InPPPJ, 2016.

[21] Jimenez, Daniel A. and Lin, Calvin. Dynamic branch prediction with perceptrons. InIEEE International Symposiumon High-Performance Computer Architecture, pp. 197206, 2001.