SPECnet: Predicting SPEC Scores using Deep Learning

Extended Abstract†

Dibyendu Das
AMD
dibyendu.das@amd.com

Prakash Raghavendra
AMD
prakash.raghavendra@amd.com

Arun Ramachandran
AMD
aruncoimbatore.ramachandran@amd.com

ABSTRACT

In this work we show how to build a deep neural network (DNN) to predict SPEC® scores – called the **SPECnet**. More than ten years have passed since the introduction of the SPEC CPU2006 suite (retired in January 2018) and thousands of submissions are available for CPU2006 integer and floating point benchmarks. We build a DNN which inputs hardware and software features from these submissions and is subsequently trained on the corresponding reported SPEC scores. We then use the trained DNN to predict scores for upcoming machine configurations. We achieve 5%-7% training and dev/test errors pointing to pretty high accuracy rates (93%-95%) for prediction. Such a prediction rate is very comparable to expected human-level accuracy of 97%-98% achieved via careful performance modelling of the core and un-core system components. In addition to the CPU2006 suite, we also apply SPECnet to SPEComp2012 and SPECjbb2015. Though the reported submissions for these benchmark suites number in hundreds only, we show that such a DNN is able to predict for these benchmarks reasonably well (~85% accuracy) too. Our SPECnet implementation uses state-of-the-art Tensorflow infrastructure and is extremely flexible and extensible.

ACM Reference format:
Dibyendu Das, Prakesh Raghavendra, and Arun Ramachandran. 2018. SPECnet: Predicting SPEC Scores using Deep Learning. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion, April 9 – 13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. DOI: https://doi.org/10.1145/3185768.3186301

INTRODUCTION

SPEC® [8] benchmarks are as the supporting memory, interconnect, I/O etc). widely used to measure CPU and system performance. But predicting SPEC scores for upcoming system configurations remains a challenging task (Here a system will refer to the SoC containing the core as well). This task is important in view of the criticality of such scores for positioning these systems in the market - especially in the server business. Long before such systems are actually built and shipped, micro-architects and performance experts try to predict the scores that can be achieved by the new systems. This involves careful modeling (analytical or otherwise) of the core as well as the other components like memory and interconnect. With the advent of hyper-threaded multi-cores having a large number of cores, presence of NUMA as well as NUCA, and complex interconnects between the dies hosting the cores, such modeling have becoming increasingly complicated. And if you are trying to predict SPEC scores ahead of time, such modeling may need to depend on simulators which are slow and time-consuming. All these factors usually lead to high prediction error even after considerable effort is spent.

We take a different approach. Our idea is to reach a prediction estimate using coarse-grained system features rather than fine-grained modeling. This saves time and effort in a big way as long as the prediction error is within acceptable limits. In view of this, we extract these features from the SPEC submissions. These features comprise of both hardware and software components as reported in each of the SPEC submissions. Once the features are extracted, we apply a Deep Neural Network (DNN) that uses these features as inputs and learns from the corresponding SPEC scores using an iterative process called Gradient Descent that minimizes the prediction error. This phase is called the training phase. Once the prediction error-rate in training reaches acceptable levels and saturates we stop the training process. The DNN is now ready for predicting scores for new systems that have not been seen before. Since some of these systems are forward-looking it may be difficult to estimate the full efficacy of the method till those systems are built. However, we see low prediction error-rates when such a DNN is fed with system configurations from the SPEC-submitted results not encountered by the DNN before.

We call our DNN the **SPECnet**. SPECnet can predict scores for SPECint_rate2006, SPECfp_rate2006, SPEComp2012 and SPECjbb2015 [8]. The architecture of SPECnet for each of these benchmark suites are very similar though they differ slightly in certain parameters. Theoretically, we could have built a single DNN for predicting all these scores, but at this point we prefer to keep them separate in order to learn more about the characteristics of each of these suites and how they behave for different kinds of submitted system configurations. SPECnet achieves good prediction rates for the dev/test sets that are kept aside from the submitted results and are not presented to the DNN for training. In the following sections we give a brief overview of Deep Neural Networks and Tensorflow. We show

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186301

the architecture of SPECnet and present results. We conclude with background and future work.

2 DEEP NEURAL NETWORK: OVERVIEW

Neural networks are inspired from a neuron's computation that involves a weighted sum of the input values. Furthermore, the neuron does not output that weighted sum. Instead there is a functional operation which is a non-linear function that causes a neuron to generate an output only if the inputs cross some threshold. Thus by analogy, neural networks apply a non-linear function to the weighted sum of the input values [10].

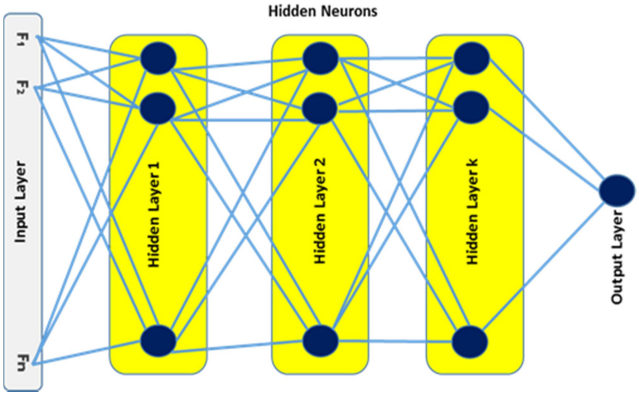


Figure 1: Neural Network with hidden neurons and layers

Fig 1 shows a picture of a computational neural network with several neurons in the input layer and only one neuron in the output layer. It also has a number of intermediate layers called the 'hidden layers'. Each neuron in a layer is connected to all the neurons in the next layer – this architecture is known as a 'Fully Connected' layer architecture [1,9]. The neurons in the input layer receive some values (features in this case) and propagate them to the neurons in the middle layers of the network. The propagation is nothing else but matrix multiplications (usually) followed by the application of a non-linear function (ex: sigmoid, tanh, ReLu [1,9] etc.). The activation function is not shown in the figure for simplicity. The outputs from the hidden layers are propagated to the output layer, which presents the final outputs of the network. The computation at each layer can be stated as:

$$Y = f(W * X + B)$$

Where W is the weight matrix representing the edge connectivity between two successive layers. X and Y are vectors representing input and output activations, respectively. f is a non-linear function like *ReLU (Rectified Linear Unit)*. The bias vector is represented as B. For the first hidden layer $X = F$ where F is the feature vector. For latter layers, the output Y from an earlier layer becomes the input X of the next layer.

Neural networks having more than three layers, i.e., more than one hidden layer is usually termed a Deep Neural Network (DNN). DNNs are capable of learning high-level features with more complexity and abstraction than shallower neural networks. An example that demonstrates this point is using DNNs to process visual data [1]. This deep feature hierarchy enables DNNs to achieve superior performance in many tasks. DNNs can be used either for classification (as in what kind of image is this) and regression (function fitting). SPECnet is an example of the latter type whereby the DNN learns a function from the input feature set to a SPEC score. In DNNs, learning involves determining the value of the weights (and bias) in the network, and is referred to as training the network. Once trained, the program can perform its task by computing the output of the network using the weights determined during the training process referred to as inference or prediction. The main goal for training a DNN is to determine the weights that minimizes the loss. For SPECnet the loss is the deviation of the predicted score from the actual score submitted. When training a network, the weight matrices are usually updated using a hill-climbing optimization process called gradient descent. A multiple of the gradient of the loss relative to each weight, which is the partial derivative of the loss with respect to the weight, is used to update the weight. Note that this gradient indicates how the weights should change in order to reduce the loss. The process is repeated iteratively to reduce the overall loss. An efficient way to compute the partial derivatives of the gradient is through a process called backpropagation [1,9].

3 SPECNET

In this section we look at the architecture of the SPECnet. Before that we need to focus on the features that need to be extracted from the SPEC submitted results that will be fed to the DNN for training and inference.

3.1 Input Features

Each SPEC submission (CPU2006, OMP2012, jbb2015) comprises of a number of hardware and software features that can be downloaded from the SPEC website [8] in .csv format. These features include the processor type, clock speed, memory type, compiler used, vendor name, year of publication etc. In addition, the text/pdf files contain details about the compiler flags and associated details. We have whittled down the set of features to be input to SPECnet to 13. They are the following:

< Number of cores, Number of chips, Base Frequency, Boost Frequency, Processor Type, Base Copies, Year of Publication, L3 per core, L2 per core, L1 per core, Memory Speed, Memory Size, Compiler Used >

Note that some of these features are 'categorical' in nature which means that there is no direct numerical association of this feature. The processor type and the compiler used are two such cases. Popular methods of encoding categorical variables include 'one hot encoding' [9] as well as hashing. For SPECnet we have used a slightly different approach. We have manually associated numerical values with the processor type and compiler used. To simplify things, we have encoded the processor type with a number for each processor manufacturing company, with a higher number signifying a more powerful processor. A similar policy has been adopted for compiler encoding.

3.2 SPECnet Architecture

The DNN for SPECnet consists of an input layer which accepts the 13 features described above, an output layer of one neuron that emits the predicted SPEC score and three hidden layers. Hence it's a 5-level DNN. All the layers are fully connected. Each hidden neuron has an activation function that is a ReLU. As mentioned earlier, each layer applies a matrix multiplication followed by a ReLU operation on each of the outputs of the neurons. Fig 2. Shows the architecture of SPECnet:

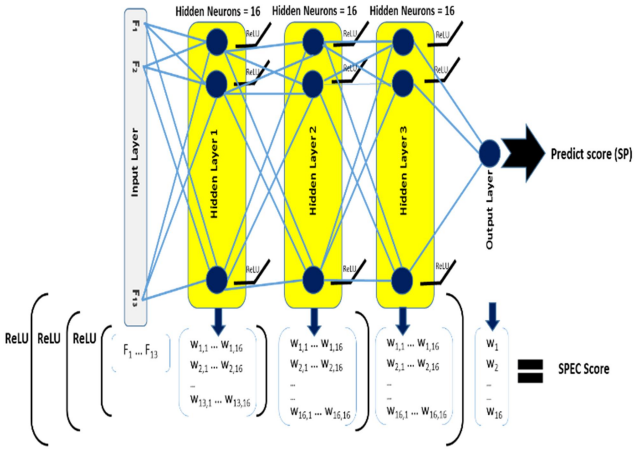


Figure 2: SPECnet architecture

The input features to SPECnet are listed as $F_1 \ldots F_{13}$ as we have 13 input features. At each successive layer, the weight matrix is multiplied by the output of the previous layer and ReLU applied on the resultant vector. We arrived at the architecture above via several experiments and hyper-parameter tuning. We observed that reducing the number of hidden layers reduced the quality of prediction (error rate increases). Also, adding layers beyond 3 did not help in improving the error rates further.

3.3 SPECnet Implementation using Tensorflow

We use Tensorflow [3] to implement SPECnet. Tensorflow is an interface for expressing machine-learning algorithms from Google, based on Python. The algorithms can be expressed as high-level computation graphs in a data-flow style with matrix operations and activation functions available as basic operations. The SPECnet is specified as a computation graph and a gradient descent optimizer is used that minimizes the loss between the predicted SPEC score and the reported SPEC score. We use MAPE (Mean Absolute Percentage Error) [2] as the loss function. If the predicted SPEC score is PSP and the actual score is SP then the loss function is given by:

$$\frac{100 * |PSP - SP|}{|SP|}$$

The gradient descent optimizer [1] works iteratively using a forward-propagation pass and a backward propagation pass. Each such combined pass is called an epoch. Tensorflow provides in-built gradient descent optimization functions which can minimize the specified loss function over a number of epochs. The number of epochs is a tunable hyper-parameter and the iterative process is continued till the loss reaches acceptable limits or there is divergence between the training and dev/test error rates.

Here's a snippet of the Tensorflow code that shows the computation graph with 'x' being the input layer and the 'out_layer' being the predicted SPEC score. 'weights' and 'biases' are the matrices corresponding to the weights and biases that the gradient descent algorithm tries to infer for minimizing the loss.

```
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'h3': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_3])),
    'out': tf.Variable(tf.random_normal([n_hidden_3, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'b3': tf.Variable(tf.random_normal([n_hidden_3])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

def computation_graph(x, weights, biases):

    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    #
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    #
    layer_3 = tf.add(tf.matmul(layer_2, weights['h3']), biases['b3'])
    layer_3 = tf.nn.relu(layer_3)
    #
    out_layer = tf.matmul(layer_3, weights['out']) + biases['out']
    return out_layer
```

```
Y_hat = computation_graph(x, weights, biases)
cost = (100*abs((Y_hat - y)/y))
# Gradient descent
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)

init = tf.global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
   sess.run(init)

   for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(num_examples/batch_size)

        for i in range(total_batch):
          batch_x = data[i*batch_size:(i+1)*batch_size]
          Y = target[i*batch_size:(i+1)*batch_size]

          # Run optimization op (backprop) and cost op (to get loss value)
          _, c = sess.run([optimizer, cost], feed_dict={x: batch_x, y: Y})
          # Compute average loss
          avg_cost += c / total_batch
```

4 EXPERIMENTS

In this section we outline our experiments using SPECnet for CPU2006 INT and FP suites, OMP2012 and jbb2015.

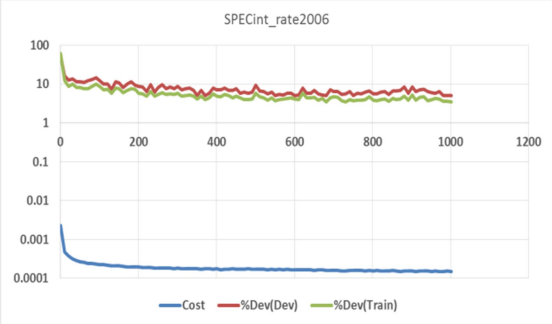
4.1 Methodology

For our experiments we have downloaded the entire history of submitted results for the SPEC suites mentioned above. The SPECint_rate2006 and SPECfp_rate2006 suites contain more than 14000 and 13000 results respectively. We remove those results which are invalid (SPEC score of 0) as well as the ones submitted with auto-parallelization turned ON. We now divide these results into separate training and dev/test sets in the rough ratio of 80%-20%. So for INT we use about 11000 results for training and about 2500 for dev/test. The dev/test set is not presented to the DNN for training. The SPEComp2012 and SPECjbb2015 on

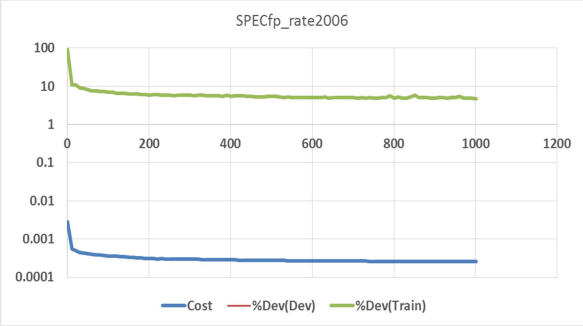
the other hand contain only about 100-plus submitted results. This poses a problem as DNNs require thousands of training data to achieve good prediction rate. However, we use 'transfer learning' [1] that provide good prediction rates using a much smaller data set.

4.2 SPECint_rate2006/fp_rate2006

For INT we use a DNN with 3 hidden layers of 16 neurons each. Fig 3 show the how the cost, the training error (%Dev(Train)) and the dev/test error (%Dev(Dev)) change with the epochs. The training and dev errors start diverging around epoch number 725. Hence we stop the training after this (known in DNN literature as 'early stopping' [9]. At this point we observe that the training error is about 5% and the dev/test error is about 7%.



For FP we use a DNN of 3 hidden layers of 12 neurons each. In this case we observe that the training and dev error almost have similar error rates. Hence we continue running the iterative process for 1000 epochs. The error rate is achieved is of the order of 6%. The dev/train trends are very similar and overlap in the figure below.



4.3 SPEComp2012 and SPECjbb2015

We use a DNN similar to CPU2006 for SPEComp2012 and SPECjbb2015. For OMP we observe that a 3-hidden layer DNN having 18,10,12 neurons work well. The accuracy level of this DNN is lower than that of CPU2006. This is expected as the training data set is really small. However we still achieve a prediction rate of about 90%. SPECjbb2015 uses a 3-hidden layer DNN comprising of 14,10,12 hidden neurons. The accuracy achieved is about 85%.

5 Related Work

Using regression models for predicting performance is not new. This is known as empirical performance modeling [7]. However, most of these works depend on either linear regression or much simpler neural networks [4,5,6]. Also, these target mainly micro-architectural performance but not a full system. The work that is closest to our current work [4] was done in 2008 using the SPEC CPU2000 suite which had about 3000 data sets. The authors applied both a linear regression model and a simple neural network. They concluded that the prediction accuracy of linear regression was better than the neural net. This could be due to various reasons including the fact that DNN was not prevalent ten years back. Also tools like Tensorflow were not available.

6 Conclusions and Future Work

In summary, we have performed an empirical modeling of current computer systems as reported in the SPEC website vis-à-vis several performance benchmarks. We use a modern DNN implemented using Tensorflow. We have shown that our prediction error rate is low despite such coarse-grain modeling. This augurs well for understanding the approximate performance envelope of future systems which are still being designed and built. Our technique achieves better accuracy when compared to Linear Regression or Decision-Tree-based techniques.

We would like to build DNNs with lower error rates. Whether that would require additional features that are unavailable in SPEC submissions, has to be investigated. Encoding categorical data like the processor type and compiler is a challenge. We would like to compare SPECnet with other modern machine learning algorithms like Boosted Decision Trees to get a fair comparison. Lastly, we would like to investigate alternate architectures of SPECnet (number of layers, number of neurons).

REFERENCES

[1] Ian Goodfellow, Yushio Bengio and Aaron Courville. 2016. Deep Learning. *MIT Press*, http://www.deeplearningbook.org/ .

[2] Arnaud De Myttenaere, Boris Golden, Benedicte Le Grand and Fabrice Rossi. 2015. Using the Mean Absolute Percentage Error for Regression Models. https://hal.archives-ouvertes.fr/hal-01162980

[3] Matin Abadi et al. 2015. Tensorflow: Large Scale Machine Learning on Heterogeneous Distributed Systems. Preliminary White Paper.

[4] Berkin Ozisikyyilmaz, Gokhan Memik and Alok Choudhary. 2008. Machine Learning Models to Predict Performance of Computer System Design Alternatives. ICPP '08.

[5] Engin Ipek, Sally A. McKee, Bronis R. de Supinski and Rich Caruana. 2006. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. ASPLOS '06.

[6] Benjamin C. Lee. 2011. An Architectural Assessment of SPEC CPU Benchmark Relevance. Harvard Computer Science Group Technical Report TR-02-06.

[7] Sam Van den Steen, Sander De Pestel, Moncef Mechri, Stijn Eyerman, Trevor Carlson, David Black-Schaffer, Erik Hagersten and Liven Eeckhout. 2015. Micro-Architecture Independent Analytical Processor Performance and Power Modeling. ISPASS 2015.

[8] SPEC®: Standard Performance Evaluation Corporation. http://www.spec.org

[9] Andrew Ng. https://www.coursera.org/learn/machine-learning

[10] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey.