

Deep Learning Fall 2019 Project Report

Large, Random Non-trainable Layers in Neural Networks with Memory Constraints

Bram van den Heuvel & Matej Sladek

January 12, 2020

Abstract

We realize that an arbitrary number of non-trainable weights can be represented by a (single) seed. We propose a memory saving technique for non-trainable parameters that recomputes weights during training and evaluation in a way that incurs a "lower than naïvely expected" runtime costs. We then attempt to utilize large numbers of non-trainable weights in various architectures and report on the model performance. Our experiments generally showed that performance doesn't improve when adding more non-trainable weights to an architecture while training stability was improved in one classification task.

1 Introduction

Classical statistics suggests there is a paradox going on in deep learning: the models have an enormous number of parameters, but they do not suffer from the downsides of that fact the same way that virtually all other model classes do.

1.1 Not all parameters are equal

In the early 1990's, Schmidt et. al. [4] attempted to explain this phenomena by showing that many of the parameters in a deep model are of less importance. They describe how the first layers of a neural net can be viewed as a collection of nonlinear features of the input data, with a simple linear regression attached to it in the final layer. The weights in the final layer are of higher importance. They demonstrate this by randomly initializing some of the hidden layers with random weights and keeping their values fixed throughout the training procedure. We call these layers *random layers*. The training procedure still works surprisingly well.

1.2 Performance, depth and memory

The value of depth and a large number of parameters is demonstrated by the models performing best in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [1]. In the last decade, the number of layers in the top-performing model has gone up to 152 with ResNet. Also the number parameters has grown; models often use tens of millions of parameters, VGG-19 as much as 155 million. This translates to a maximum memory usage during training of approximately 200 MB for most models, a little over 800 MB for VGG-19, when trained with a batch size of one. This then increases to above 1000 MB for several top-

performing models once the batch is increased to 16, and increases approximately linearly with batch size beyond that [1]. This means that with a batch size of 64, none of the top performing models can be trained on a mobile device or a consumer-grade GPU because of memory limitations.

2 Regeneration of weights

We realize that an arbitrary number of non-trainable weights can be represented by a (single) seed. A Pseudo Random Number Generator (PRNG) can generate values that are "hard to distinguish" from independent realizations of a certain distribution. They typically work sequentially, or multiple seeds are used between threads if the values should be generated in a reproducible manner.

These methods are sufficient to save on model storage costs for large models. A more realistic problem is memory usage during training, as was made clear in section 1.2. The memory usage typically peaks at the end of the forward pass, when the model weights and the activations are stored. Recomputation of activations during the backward pass, by only saving a subset of the computed activations and recomputing them from the last layer they are known from, has allowed for sub-linear maximum memory usage in the number of layers during training [2].

While one typically recomputes a majority of the activations with the algorithm in [2], the runtime increase is shown to be only 30% for a 4x reduction in maximum memory usage when training an LSTM under a long sequence unrolling setting. It is hypothesized that this can be explained in part by the under utilization of computational resources in a GPU, where memory bandwidth is more often a limiting factor¹ (a simple solution would be to increase the batch size, but

¹We found no scientific research on this (engineering) subject, but plenty of anecdotal evidence in blog posts such as these: <https://nestlogic.com/index.php/2019/02/11/gpu-utilization-with-neural-networks/> and <https://towardsdatascience.com/measuring-actual-gpu-usage-for-deep-learning-training-e2bf3654bcfd>

this directly increases the maximum memory usage, see again section 1.2).

All of this changes when instead, we utilize a Pseudo Random Function (PRF). These can be viewed as PRNG's which don't generate values (v_1, v_2, \dots) in sequence, but instead can evaluate in constant time a function $\mathbb{N} \ni n \mapsto v_n$. They stem from cryptography and the well-known ones are made for such applications, unsuitable for training neural networks. The requirements for a random number generator are different for this application: speed and memory usage matter, while the quality of the generated numbers less. For a sketch of a first idea of a pseudo random function for this application, see the appendix. The memory requirements of such a function are low enough to evaluate it on-chip (A TitanX GPU has approximately 1KB of memory for a thread [3]), without any communication across cores. In this way, there is no memory access incurred for non-trainable weights, which we think will be reflected favorably in the training time of these models.

We describe this difference in terms of model storage size, memory requirements during training and evaluation, as well as training and evaluation time. We then experiment with random layers in different architectures and evaluate the performance.

3 Models and Methods

We study the effect of introducing new random layers, or replacing existing layers by random layers. We propose idea to store weights of random layer in memory effective way. We do not have the means to provide an implementation of our idea. We measure model performance exactly by simply storing the random layers completely.

3.1 Performance

To measure the effect of random layers on performance we identified the following tasks:

- Convolutional architecture to classify images from the MNIST fashion dataset
- Dense architecture to classify level of income on the census income dataset
- Dense architecture to do regression on the census income dataset
- Run bayesian optimization to identify best model architecture with possibility of random layers and cap on trainable parameters

For MNIST dataset we evaluate 3 dense architectures, 3 convolution architectures with one convolution, max pooling and varying number of dense layers, 3 convolution architectures with added varying number of random layers.

For census income dataset we evaluate dense models without random layer, with 1 random layer and with 2 random layers. Every of this configuration we train

with multiple sizes of layers before and after random layer. We also evaluate architecture with residual random layers where we can train one scalar for random layer or one scalar for every node of random layer.

In optimization, domain of search is size of random layer, number of random layers, size of end layer, activation function of random layer, activation of last layer and whether use batch norm.

For the training we use Adam optimizer and early stopping callback to stop learning after evaluation loss does not change much.

4 Results

On MNIST dataset we data to training and test data set in ratio 8:1. We train every architecture one time. The loss function is sparse categorical crossentropy (higher is better). Best dense configuration had performance 0.89%, best convolution configuration had 0.91% and with added random layers best performance was 89%. In 1 can be seen plot with data point for every configuration. On plot we see means of every type of architecture. After adding random layers the performance dropped.

On Census income we use K-fold evaluation and we split data to 10 parts and every architecture configuration train for 10 times. In regression task loss is mean squared error(lower is better). In 2 we can see that mean of results is similar even if we add 1 or 2 random layer. We also go deeper with random layers with using residual random layers. We evaluate architectures with 10, 20, 30 and 40 residual random layers. Results are in 3 and there are not much difference when we add more random layers.

On Census income on classification task we take same models as in regression. We use sparse categorical crossentropy loss. Results are in 4. Best models have similar results even after adding random layer, but we can see mean is smaller for architectures without random layer and that is because few models has low performance, so we can see that stability of training went up after adding random layer.

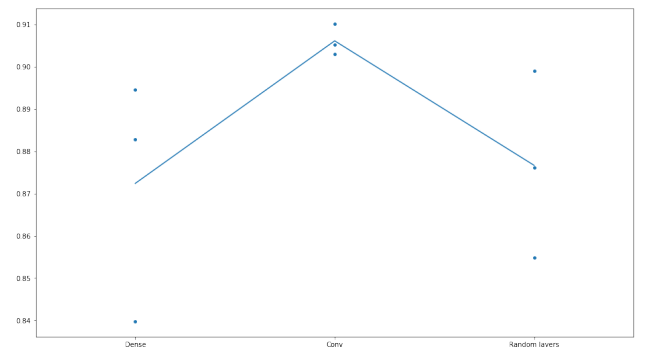


Figure 1: MNIST

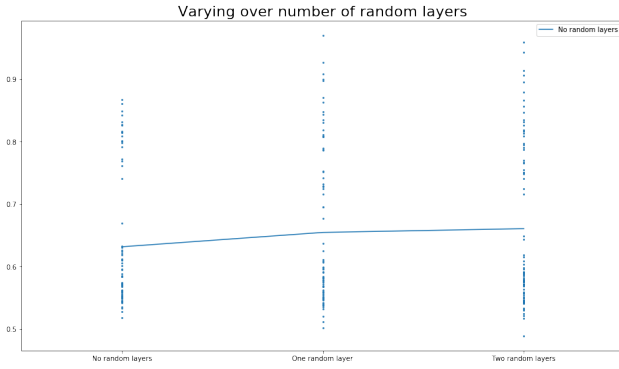


Figure 2: Census income regression

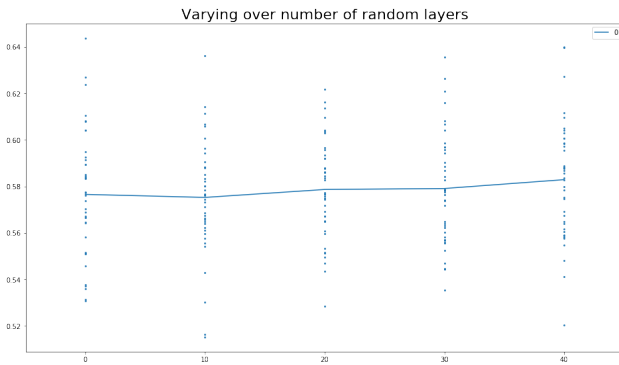


Figure 3: Census income regression residual layers

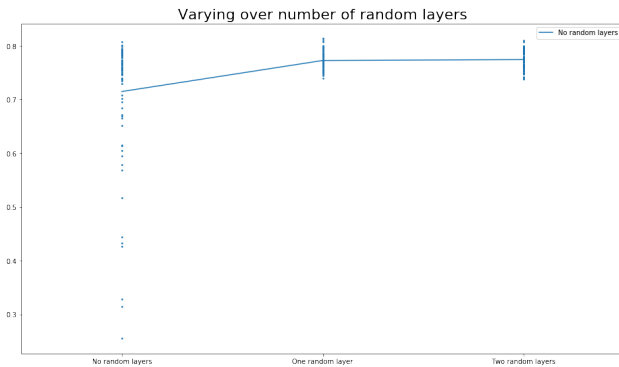


Figure 4: Census income classification

5 Discussion

From results we can say that adding few random layers does not improve performance. Mean performance of models went a little bit down after adding these layers. Same happen when we add lots of residual random layer. In classification task of census income models without any random layer happened have lower performance in some cases, but after adding random layer

we have not see it. From that we conclude that in some cases random layers can improve stability of training.

6 Summary

In conclusion we proposed idea to memory limited implementation of random layers. We evaluated models with or without random layers on regression and classification tasks. We conclude that adding random layers does not change performance a much. In evaluation of classification task of census income dataset, we have seen that training stability was improved.

7 Appendix

7.1 A possible pseudo random function

We choose a simple hash function h , to be evaluated on a tuple of

1. a unique identifier for layer and
2. the coordinates of the parameter.

The resulting bits are then interpreted as a float in $[0, 1]$ to be used as input of the inverse of a cumulative distribution function.

It goes without saying that we are no experts on such functions; the above is merely a first idea from non-experts.

References

- [1] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.
- [2] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.
- [3] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *ICML*, pages 2024–2033, 2016.
- [4] W.F. Schmidt, M.A. Kraaijveld, and R.P.W. Duin. Feedforward neural networks with random weights. In *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*. IEEE Comput. Soc. Press.