A Neural Network Search Engine

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Michael Andrzej Klaczynski

Spring, 2020.

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Hongning Wang, Department of Science

Abstract

We define a method by which one can search the weights of existing neural networks for weights that could be useful for a developing neural network, as well as a way in which these weights can be incorporated into a neural network during the training process.

Introduction

An artificial neural network can be thought of as a network of functions. Many small functions, put together, create a large function that solves complex problems. In past years, we have seen the rise of “transfer learning” where one retrains part of an existing neural network to solve a different problem than it was originally intended to solve. This is possible because the functions which make up a neural network are useful for more than just one problem.

What this project attempts to do is transfer learning at a more granular scale. Rather that importing an entire neural network to start with, we wish to import only the pieces that matter. By breaking up a neural network into smaller pieces, we become more capable of repeating patterns in the data, and can use that to our advantage. For example, if a similar kernel of a convolution operation appears in multiple neural networks, and in each network is preceded by the same pattern, one might be able to infer from seeing those patterns in another network that that convolution operation would be useful in that place.

Part of the inspiration for this research was analogical reasoning in humans. An analogy is an “inference that if two or more things agree with one another in some respects they will probably agree in others” (Woolf 1971). Given knowledge of one system, a human can make guesses about an unrelated yet structurally similar system (Kedar-Cabelli 1988). Using

analogical reasoning it is often possible to find a solution to a problem much more quickly, as it allows humans to skip a lot of trial and error learning.

Related Work

Cognitive and computer scientists have been fascinated by analogical reasoning for decades. The concept is simple, in theory, but in practice it is often hard to replicate due to complexity (Halford, 1992). While modern language models have significantly improved a computer's ability to recognize and parse analogies (Hoshen, 2018), analogical reasoning remains hard to grasp.

Implementation

There are two parts to this problem: finding repeating patterns, and using them. To find them, we need a lot of neural network data. This data must be well-organized and easy to traverse. Finding patterns is difficult even with organized data, and we stick mostly to standard clustering techniques.

There are several ways we could use this data, but here we define a way to test various kernels against the data to find the best ones, as well using existing weights to find relevant convolutions. (We focus mainly on convolutional neural networks here because of the availability of large networks, but the ideas could be applied to any neural network.)

Data Collection

Collecting weight data from a sufficient number of large neural networks was surprisingly difficult. Neural networks are rarely held in convenient formats. Most deep learning projects have custom-built programs to import the weights from databases of h5 or checkpoint

files. In those cases, the structure of the network can only be determined by parsing or running the code. While this is possible, most frameworks provide no good way of easily traversing their networks.

ONNX is a format designed to store neural networks for the purpose of easily converting them between frameworks. It therefore contains both the weights and the structure of the network. It also comes with a convenient python library that can be used to observe its internal structures. Because of this transparency, we parsed ONNX files to generate our library of neural networks.

Function Mapping

Knowing the structure of a neural network as a deep learning framework understands it, however, is insufficient for our purposes. The problem is that normally neural networks are represented as *layers* of functions. A convolution layer, for example, is usually composed of many convolution operations, each of which has a different kernel and outputs its own channel. Entire layers are unlikely to bear similarities across networks, as each channel is trained to detect a different feature, and the ordering of these features within the layer is not at all guaranteed to be the same between networks. Therefore, layers must be broken up into their component functions.

This is not a trivial task. Modern deep learning frameworks including ONNX don’t have a way to do this, because it’s not normally how one would use a neural network. So, each layer had to be taken apart function by function, mapping each to its inputs and outputs. The resulting expanded networks were stored in a Neo4j database, most having tens of millions of nodes.

Pattern Mining

In this paper we only apply a primitive method of pattern mining to the data, though in the future this can be traded out for more sophisticated methods.

Here we focus only on specific shapes of subgraphs: in particular, convolution trees of specific depths, where the outputs of child nodes are used as input for parent nodes (disregarding activation and normalization functions which may lie in-between). By looking at a set of child nodes, we can then attempt to predict a useful parent node based on similar subsets in our database.

It would be very slow to check against every possible subset of nodes, so instead we rely on grouping subsets using a combination of clustering and linear classification algorithms. First, we used clustering to create a set of classes for all kernels that are parents. In this case, it is a K-Means algorithm, though this can be swapped out easily if further experimentation proves another to yield superior groups. Then, we trained a linear classifier to predict the class of the parent based on the kernels of its children. This, we can send the classifier queries of convolution kernels, and the classifier can return the central point of the associated K-Means group, which the querier can use for the value of the next kernel.

Note that we use a classifier here instead of a regressor. Why not directly guess the values within the kernel? There are multiple reasons for this. It is also faster to train, for one. For the other, the clustering algorithm creates an interesting set of “standard kernels” that can be observed by data scientists. This lends its self towards the potential for more transparent machine learning algorithms if the behaviors of the common kernels can be identified.

Function Building

There is one fault in this design: it only works as intended if part of the neural network is known to be well-trained and another part is not. If the entire network is untrained, the kernels you query with could change to something more useful, making the results of the query irrelevant. If the whole network is trained, then it's also irrelevant. So, to get into a situation where we can use the knowledge of the database, we have to do something a bit contrived.

In our implementation, we "grew" a neural network from the ground up. We started with a single convolution layer, a single linear layer, and a very thick pooling layer. The weights of the convolution were initialized to those of a set in the database with no children nodes. Then, only the linear layer was trained. Once it finished, the convolution weights were swapped out for a different set. This was done dozens of times. Finally, the set that had resulted in the least loss was chosen. Thus, we found a bottom layer with relatively useful features.

Then we added a second layer of convolutions, this time querying the classifier for good classes of kernels based on the first layer. The linear layer trained on top of this for a while before it was swapped out for the next suggestion given by the classifier. Once the third convolution layer was added, the network was allowed to train naturally.

Results

Naturally, this process of constantly swapping out weights and retraining took a lot longer than letting the control, a three-layered convolutional neural network, just training its self to begin with. Of course, that doesn't say much. Deep learning frameworks have been designed to work a certain way for years. This mechanism was built in the last few months. Perhaps it could work with more optimizations.

Further Research

There are many directions in which one can go with this project. Would other clustering algorithms do better? Or a different way of classifying patterns entirely? Could one find a better way of integrating it into existing frameworks? Could analysis of the patterns found reveal anything about the inner workings of a neural network?

The implementation may not have appeared to do so well, but that could likely be solved with incremental improvement. It’s too intriguing an idea to give up on just yet.

Works Cited

Halford, G. S. (1992). Analogical reasoning and conceptual complexity in cognitive development. *Human Development*, *35*(4), 193-217.

Hoshen, Y., & Wolf, L. (2018). Identifying analogies across domains.

Kedar-Cabelli S. (1988) Analogy — From a Unified Perspective. In: Helman D.H. (eds) Analogical Reasoning. Synthese Library (Studies in Epistemology, Logic, Methodology, and Philosophy of Science), vol 197. Springer, Dordrecht

Woolf, H. B. (1971). In *The Marriam-Webster Dictionary*, *Pocket* (pp. 433–439).