Pruning Neural Networks with Lottery Tickets in a MDP Approach

Andrey de Aguiar Salvi
Machine Intelligence and Robotics Research Group
School of Technology
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, RS, Brazil

Abstract

We can find many different forms of model reduction in the literature for different purposes. In this work, we performed an adaptation of the Lottery Tickets Hyphotesys (LTH), which aims to reduce the model while maintaining as much accuracy as possible. Our adaptation used an agent trained in a Markov Decision Process. We compared the original LTH result with our approach on a LeNet 300-100 trained in the MNIST dataset, and we achieve good results for a classical planner. Our work even has a margin for improvements and we believe that in future works our proposed method will be able to beat the LTH.

Introduction

Artificial neural networks are state of the art on computer vision tasks, despite the fact of creating giant models with many parameters. For example, the work of Simonyan and Zisserman creates a model with 144 million of weights. The various possible ways of combining convolutional filters with different sizes and many other details allowed to create models with fewer weights and less computation, as are the examples of the works of Szegedy et al., Ioffe and Szegedy, He et al., Szegedy et al.. Even so, none of the models we present have less than 10 million weights or performs less than 2.5 Giga floating-point operations per second.

This high computational cost makes it necessary to use GPUs for real-time computer vision and makes it impracticable to use these models on CPUs or mobile devices. With these problems in mind, many authors have worked to find ways to compress these models for some purposes such as reducing the number of operations per second, or just reducing the size of storage required, or even reducing the energy required for use of the model for use on mobile or embedded devices, depending on the need for the problem to be attacked.

Consequently, there are many different forms of model compression with different purposes in the state of the art. For example, Han, Mao, and Dally has created a method for compressing convolutional networks with the primary purpose of reducing the model's disk storage. After an iterative sequence of pruning connections, it applies a clustering algorithm to the remaining connections. The algorithm stores only the found centroids.

Han et al. has created a compression technique to reduce energy consumption to run the model on mobile devices. The developed method was a process of training the model and discarding connections with weights lower than a threshold, iteratively. Also, in training is used L2 regularization and dropout with an adjusted rate after each pruning.

One of the state-of-the-art model compression aimed at maximum compression with minimal loss in accuracy is the work of Frankle and Carbin. The method employed also utilizes an iterative process of training and pruning connections.

There are also in the literature some techniques that use automated planning for model compression. For example, Ashok et al. uses policy gradient reinforcement learning to compress a student network, a model created to mimics the behavior of a teacher network. Another example is the work of Zhang et al., which creates a reinforcement learning framework with Markov Decision Process (MDP) to remove layers from a DenseNet model.

The objective of this work is to investigate the possibility of generating a model compression algorithm based on Frankle and Carbin using an MDP. We will replace the original way of choosing weights to be removed in the Lottery Tickets Hypothesis by a policy extracted from a Q-table.

Background

Here we will describe the main concepts, the Lottery Tickets Hypothesis and the Markov Decision Process, and after the proposed method.

Lottery Tickets Hypothesis

A layer of a fully connected neural network can be given by the equation below

$$\sigma(x) \leftarrow f(W^T x) \tag{1}$$

where x is the input vector, W is the weight matrix, $f()$ is an activation function and σ is the output computed. In the lottery thickets (Frankle and Carbin 2018), the forward computation is given by

$$\sigma(x) \leftarrow f((W' \odot W)^T x) \tag{2}$$

where W' is a binary mask (initialized with ones) that holds or prunes the weights, having the same shape as W and $\odot$ is the Hadamard Product. Given a pruning rate p, in the Lottery Tickets Hypothesis (LTH), the $p\%$ remaining weights with lower magnitude are prune (this means $W'_{i,j} = 0$), until the total of remaining weights is the desired value. The full algorithm of the Lottery Tickets Hypothesis is in algorithm 1.

Algorithm 1 Lottery Tickets Hypothesis
Source: adapted from (Frankle et al. 2019)

Require: weight matrix W, mask W', pruning rate p
1: $W_0 \leftarrow W$
2: $train(W, X, Y, n\_epochs)$
3: **while** $remaining\_weights > total\_weights \times p$ **do**
4: $\quad indexes \leftarrow find\_smallest\_values(|W|, p)$
5: $\quad W'[indexes] \leftarrow 0$
6: $\quad W \leftarrow W_0$
7: $\quad train(W, W', X, Y, n\_epochs)$
8: **end while**

The function $train()$ will train the model with weight matrix W in the dataset composed by X and Y, $remaining\_weights$ is the number of remaining weights in W (in the same indexes, $W' = 1$), $total\_weights$ is the total number of weights. The function $find\_smallest\_values()$ with $|W|$ and p returns the indexes of $p\%$ absolute smallest values in matrix W.

Markov Decision Process

Markov Decision Process (MDP) is a non-deterministic state-transition system where the probability distribution of the next state depends only on the current state (Ghallab, Nau, and Traverso 2016). Given a set of states S, a set of actions A, the state transition function $\gamma : S \times A$, we have $P_a(s, s')$ as the probability that action a in state s will lead to state s' and $r_a(s, s')$ as the immediate reward received after transitioning from state s s s to state s' with the action a.

In the cases where the probabilities or rewards are unknown, we have a Reinforcement Learning problem, and Q-Learning must be used to solve the MDP. In Q-Learning, we have a table called Q where states index table rows and actions index table columns. . Q table is initialized with an arbitrary value accordingly to the problem and is incrementally updated on $Q(s, a)$ by some equation. Examples of equations to update the Q table can be the Q-Learning

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \Big[r_a(s,s') + \max_{a'} \{Q(s',a')\} - Q(s,a) \Big] \tag{3}$$

or this variant of Q-Learning defined by these sequence of equations

$$Old\_Weighted\_Value \leftarrow (1-\alpha) \cdot Q(s,a) \tag{4}$$

$$Learned\_Value \leftarrow r_a(s,s') + \gamma \cdot \max_a Q(s',a) \tag{5}$$

$$Q(s,a) \leftarrow Old\_Weighted\_Value + \alpha \cdot Learned\_Value \tag{6}$$

where the rewards $r_a(s, s')$ and the probabilities $P(s'|s, a)$ are unknown and the value of Q in s' is observed in the step of the trial (Ghallab, Nau, and Traverso 2016). α is the learning rate and determines how the newly acquired information ($Learned\_Value$) will override the old information ($Old\_Weighted\_Value$). The γ is the discount factor and determines the importance of future rewards. Low values of the discount factor make the agent consider more the current rewards, while high values make the agent strive for a long-term high reward. Both α and γ have the values between [0, 1].

The whole process is in the algorithm 2

Algorithm 2 Q-Learning algorithm
Source: adapted from (Ghallab, Nau, and Traverso 2016)

Require: set of actions A and set of states S
1: **for** i **to** MAX_ITERATIONS **do**
2: $\quad agent \leftarrow create\_new\_agent()$
3: $\quad$ **while** goal not achieved **do**
4: $\quad\quad a \leftarrow agent.choose\_action(Q(s))$
5: $\quad\quad s' \leftarrow agent.apply\_action(s, a)$
6: $\quad\quad r_a(s, s') \leftarrow compute\_reward(s, a)$
7: $\quad\quad Q(s, a) \leftarrow compute\_quality(Q, \alpha, \gamma, s, a, s')$
8: $\quad$ **end while**
9: **end for**

where $choose\_action()$ returns a possible action in state s, $apply\_action()$ performs the action creating the new state s' and $compute\_reward$ returns the reward of perform action a with state s. We can choose the $compute\_reward()$ function to be the equation 3 or 6.

The Proposed Method

We propose in this work to modify the algorithm 1 changing the function $find\_smallest\_indexes()$. Instead of choosing the lowest weights in absolute value on W to prune ((line 4 from algorithm 1)), we choose an action by a policy derived from Q-table. After the execution of algorithm 2 is create the policy, where the state s represents a mask W' and the action a means remove (set as 0) a row from the mask W' creating a new mask s'. When an agent finds the goal, we reward W' to the initial state.

To learn the Q-table, the agent must make the most of what it already knows to ensure the benefit of its actions for the task at hand and also need to find the best actions. However, to find the best actions, it needs to explore options that the agent does not know enough about it. This is the trade-off called as exploration vs. exploitation (Ghallab, Nau, and Traverso 2016). We propose in this work to use a decaying ϵ-greedy search, which means that in the algorithm 2, each agent is initialized with a value ϵ that regulates the probability of the agent making a random decision. In the first iteration, ϵ = 1, the agent is performing exploitation, and it randomly chooses all the actions. The next agent is created

with a smaller ϵ, meaning that the agent will choose a little bit of the actions through the higher values of Q-table. We repeat this process until $i == MAX\_ITERATIONS$ (line 1 from algorithm 2), and as the value of ϵ decreases along with the iterations, the agent performs less exploitation and more exploration.

To perform the algorithm 2 and learn the Q-table, we first need to create a neural network and train by some epochs. This network with weight matrices W and their respective masks W' will be the initial state s_0. After the action a we compute the reward with the accuracy of the model on the validation dataset (to ensure that the resulting mask does not affect the model result in data not seen in training) by the equation bellow

$$r_a(s, s') \leftarrow \frac{-(1 - accuracy) * remaining\_weights}{total\_weights} \tag{7}$$

where $accuracy$ will value $\in [0, 1]$, $remaining\_weights$ is the number of remaining weights after the pruning and $total\_weights$ is the total number of the weights in the state s_0. With this reward function, we ensure that between two actions with the same accuracy, the agent will choose one that removes a larger number of connections. The rewards are negative to force the agent to find the final state. If the rewards were always positive, we would risk allowing the agent always to choose to remove rows W' from the final network layers, which have fewer connections and are likely to have less impact on accuracy drop. In this way, we would be creating an agent refusing to remove connections as much as possible (the opposite of we need). When the agent reaches the final state, we reward it with more 100 points.

After learning the Q-table, we create another neural network from scratch and finally perform our adapted version of Lottery Tickets. Our proposed method can be seen in algorithm 3

Algorithm 3 Our proposed method

Require: weight matrix W, mask W', pruning rate p

1: $W_0 \leftarrow W$
2: $train(W, X, Y, n\_epochs)$
3: **while** $remaining\_weights > total\_weights \times p$ **do**
4: $\quad s \leftarrow W'$
5: $\quad indexes \leftarrow \pi(s)$
6: $\quad W'[indexes] \leftarrow 0$
7: $\quad W \leftarrow W_0$
8: $\quad train(W, W', X, Y, n\_epochs)$
9: **end while**

where the state s is a representation of the weight matrix W' and the policy $\pi()$ has generated from the Q-table learned with the algorithm 2.

Experiments

We perform validation of our proposed model on a LeNet 300-100 trained on MNIST dataset, just as in the work of (Frankle and Carbin 2018). First, we train the neural network by 30 epochs on the training dataset, splitting 20% to validation and using the Adam optimizer with learning rate equal to 0.001, and the other hyperparameters we set to default values of Pytorch implementation (Pytorch 2019). After we perform the algorithm 2 choosing $MAX\_ITERATIONS$ equal to 15, because it was a value found in preliminary tests that ensured that Q-table learning converged. The goal of each agent is pruning this network until the rate of the remaining weight be equal to 20%

Table 1: Train and validation accuracy's

Q-update	α	γ	Train_Acc	Valid_Acc
EQ_3	0.5	-	89.68%	89.31%
EQ_3	0.6	-	90.70%	90.40%
EQ_3	0.7	-	91.09%	90.86%
EQ_3	0.8	-	8.85%	0.09%
EQ_3	0.9	-	90.59%	89.82%
EQ_3	**1.0**	**-**	**91.21%**	**90.86%**
EQ_6	0.9	0.4	9.10%	0.09%
EQ_6	0.9	0.5	88.98%	88.69%
EQ_6	**0.9**	**0.6**	**90.60%**	**89.94%**
EQ_6	0.9	0.7	89.83%	89.22%
EQ_6	0.9	0.8	90.09%	89.39%
EQ_6	0.9	0.9	90.16%	89.79%
EQ_6	0.9	1.0	9.10%	0.09%
LTH	**-**	**-**	**96.86%**	**95.92%**

We execute the Q-Learning with different combinations of hyperparameters, generating different Q-tables, varying the Q-update with the equations 3 and 6. For the case of equation 6, we vary discount factor γ. ϵ was initialized with 0 and decreased by each step by $0.1/MAX\_ITERATIONS$, reaching the value 0.1 in the final step. As equation 6 has 2 hyperparameters, whereas the equation 3 has only one, we choose to vary just the discount factor and keep the learning rate with 0.9.

From each Q-table, we generate a policy that returns the indexes to prune in our proposed model. Except for line 4, all the rest of the algorithm 1 remains the same. We chose the number of n_epochs to equal 9. In table 1 we can see the performance of the LeNet 300-100 pruned by each policy. The Q-update on the table refers to the use of equation 3 or 6 to update the Q-table.

As some models presented hugely discrepant accuracy between some classes, in table 1, we compute the harmonic average of the accuracy of each class. The LTH value is the original method from (Frankle and Carbin 2018). The original LTH achieves the best results, although the difference was not too big for the second and third-placed, with a 5.06% difference in the validation dataset.

We perform an analysis of variance, which revealed that the change in hyperparameters was not significant in the result of the validation accuracy obtained. Therefore, we decided to choose the settings that obtained the best validation using the equations 3, 6, and LTH. We highlight the settings that we choose in bold in the 1 table.

The table 2 shows the accuracy on the test dataset, where

Table 2: Test accuracy of the best models

Class	Model 1	Model 2	Model 3	Instances
0	97.44%	97.55%	98.57%	980
1	97.26%	97.18%	98.23%	1135
2	91.08%	88.56%	95.54%	1032
3	88.81%	87.62%	96.33%	1010
4	92.46%	94.80%	95.92%	982
5	83.85%	81.83%	94.05%	892
6	94.67%	94.05%	96.97%	958
7	91.34%	90.66%	96.39%	1028
8	89.73%	86.34%	94.76%	974
9	88%	86.91%	94.84%	1009
Overall	91.6%	90.7%	96.2%	10000

$Model1$ is the model with configuration $\alpha = 1$ and Q-update with equation 3, $Model2$ is the model with configuration $\alpha = 0.9$, $\gamma = 0.6$ and Q-update with equation 6 and the $Model3$ refers to the original LTH. We can see o the table 2 that the $Model3$ achieves the high accuracy in all of the classes. Our proposed models achieves 4.6% less accuracy from $Model1$ and 5.4% less on $Model2$.

Conclusions

In this work we propose an adaptation for the LTH. The results, although inferior to the original algorithm, were very promising for a classic planner. We modeled the agent action to remove a row from the mask W', so that the search space would be reduced due to the time available. If the actions were to remove only a single connection from the weight matrix and the learning was directed by the descending gradient, then perhaps the proposed model could surpass the original LTH. Another thing that should be worked on in future works is the study of our proposed model in convolutional neural networks, which were not performed here due to the long time required to do this processing.

References

Ashok, A.; Rhinehart, N.; Beainy, F.; and Kitani, K. M. 2017. N2n learning: Network to network compression via policy gradient reinforcement learning.

Frankle, J., and Carbin, M. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks.

Frankle, J.; Dziugaite, G. K.; Roy, D. M.; and Carbin, M. 2019. The lottery ticket hypothesis at scale. *CoRR* abs/1903.01611.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge University Press.

Han, S.; Pool, J.; Tran, J.; and Dally, W. 2015. Learning both weights and connections for efficient neural network. In Cortes, C.; Lawrence, N. D.; Lee, D. D.; Sugiyama, M.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc. 1135–1143.

Han, S.; Mao, H.; and Dally, W. J. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Pytorch. 2019. Source code for torch.optim.adam.

Simonyan, K., and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Szegedy, C.; Ioffe, S.; Vanhoucke, V.; and Alemi, A. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning.

Zhang, X.; liu, H.; Zhu, Z.; and Xu, Z. 2019. Learning to search efficient densenet with layer-wise pruning.