Neural Network Compression
Ternary Networks

Project Handout (Revision 1)

1 Introduction

As Deep Neural Networks are permeating the Automatic Driving domain, accuracy and robustness are soaring. However, these gains bear a large computational cost price tag. Even multi-GPU systems with more than 10GB of memory may struggle to provide real-time performance for contemporary state-of-the-art solutions. But on a vehicle, even when setting financial considerations aside, the limited available power rules out such architectures from the start.

Luckily, it turns out state-of-the-art can be achieved with much smaller and faster neural network architectures. Successful strategies being explored in the literature include pruning kernel weights, entire feature maps, or restricting weights and/or activations to only take on a small set of discrete values.

Your assignment is to investigate a particular instance of the latter strategy – weight ternarization. The idea is to restrict weights to only three possible values. A neural network conforming to this restriction is called a *ternary neural network*. The most common variation is to restrict these values to be in the set $\{-\alpha, 0, \alpha\}$, where α is a learnt kernel-specific parameter. See figure 1 for an example.

The challenge is to come up with a training algorithm that can produce an accurate ternary neural network. You will be focusing on three methods from the literature, namely Ternary Weight Networks (TWN)[4], Trained Ternary Quantization (TTQ)[5] and Alternating Direction Method of Multipliers(ADMM)[1] [3].

Your tasks are as follows:

- Literature survey: Read the relevant papers from the literature and understand their contributions, as well as their limitations (section 2.1).
- Performance Indicators: Decide what metrics you want to use for comparing the methods (section 2.2).
- Baseline: Implement your baseline method and run a baseline experiment, focusing on both accuracy and efficiency (section 2.3). The baseline is

[1]ADMM is a general numerical optimization method for constrained problems. However, in this project, we shall use ADMM to refer to the application of this general method to training ternary neural networks

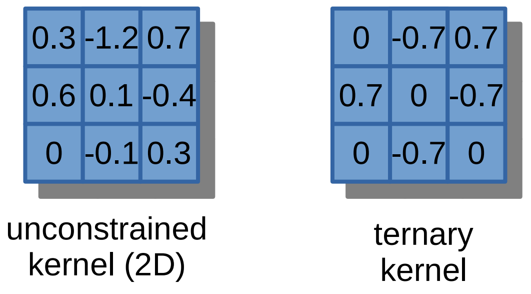

Figure 1: Example of an unconstrained (left) vs. ternary (right) kernel. Note that the ternary kernel only contains three possible weight values, one which is zero (*i.e.* it is sparse).

important, as it gives you a reference point with respect to which you can evaluate the methods.

- Method implementation: Write your own implementation of TWN, TTQ and ADMM in PyTorch (section 2.4).
- Experiments: Compare your methods against the baseline and against each other, under both accuracy and efficiency metrics. You most probably will need to tune hyperparameters (such as the learning rate) to achieve good convergence and to improve the results (section 2.5).
- Documentation: Write a report summarizing your findings (section 2.6).

2 Steps

This section describes the steps you need to take to finish the project. You will most likely need to iterate between some of these steps (for example, run experiments and document results). Also, since this is a team project, some steps need to be done in parallel (*e.g.* each team member implements a method). Be careful with handling dependencies between steps when working together!

2.1 Step 1: Literature Survey

Before reading the TWN[4] and TTQ[5] papers, it is highly recommended to read the BinaryConnect article[2], as TWN and TTQ build heavily upon this work. Similarly, before reading [3], you should get acquainted with the Alternating Direction Method of Multipliers. Chapters 1, 2 and 3 of [1] provide good introductory material.

Note that the ADMM paper is more general, allowing for more than three quantization levels. In this project, you are only concerned with ternary networks.

It is good to keep a critical mindset when reading literature. For example:

- Why would ternary kernels speed up computation? How many multiplications are needed for forward propagation through a ternary convolutional layer compared to the unconstrained (classical) case?
- Are the networks trained using TTQ, TWN and ADMM really ternary after deployment on hardware? After all, Batch Normalization layers are used in practice to achieve convergence. Is there a way to change the network **after** training to remove these layers and still keep the network ternary?
- Couldn't the authors simply restrict weights to be in the set $\{-1, 0, 1\}$, and get rid o multiplications altogether? Had they done so and used their algorithm as presented, would the network really become ternary? *Hint:* See the bullet above about Batch Normalization.
- Why do you think ADMM outperforms TWN and TTQ[2] in
- the experiments on large datasets?

2.2 Step 2: Performance Indicators

It is good to decide early on what metrics are relevant. What you care about are two conflicting performance indicators:

- *Accuracy*: Clearly, no matter how much you compress and speed up your model, you do not want to affect its accuracy.
- *Cost*: While you could measure speed and power consumption, this would require you to have access to a specialized hardware implementation or an optimized inference library. For this project, you should settle for the *compressed network size*, that is, the *number of nonzero weights.* Note that this quantity is proportional to the number of additions and subtractions you need to do at inference.

Clearly, you want a tradeoff between performance and cost. Therefore, any solution should be viewed as a 2D-point in a plot where the X axis is the accuracy and the Y axis is the cost (network size). The ideal model is close to the point with coordinates $(1, 0)$ in this space, *i.e.* it has 100% accuracy and negligible computational cost. The closer you get to this point, the better your solution is.

[2]TTQ is also sometimes cited as TTN in the literature.

2.3 Step 3: Baseline

You will need a baseline before you start experimenting. You should decide on:

- *The task*: For this project, you should stick to classification, as it is the standard benchmark for comparing training methods in the literature. This will also simplify comparison with results reported in the papers.
- *The dataset*: You should choose a public datasets that are small enough to experiment with. The recommendation would be to use MNIST and CIFAR-10, but you could also add the larger ImageNet if you have the computational resources and time.
- *The architecture*: You should choose a neural network architecture that is suitable for the amount of data you have. The VGG-7 architecture used in[4] could provide a good starting point. If you try running experiments on ImageNet, you should use a higher capacity model, like ResNet-18.

Once you have decided on the three elements above, you should train a network with floating point accuracy. Make sure your network does not over-fit (use the validation set), and record the networks accuracy and efficiency, as discussed in section 2.2.

2.4 Step 4: Implementation

Now that you have your baseline, extend your codebase to support TWN and ADMM, as described in the papers. Try to follow good software design practices and generalize the baseline – do not duplicate code, as this will be hard to maintain and error-prone. In the end, you should be able to configure your trainer to either train the baseline, TWN or ADMM. The user should also be able to specify the dataset or any other hyperparameters. You can rely on publicly available code for inspiration, but note that you have to hand in your own Pytorch implementation.

2.5 Step 5: Experiments

Run training experiments with both TWN and ADMM, on the same dataset and network architecture. You will want to experiment with changing hyper-parameters, to get the best results, as defined by the performance indicators defined in section 2.2.

In addition, you should investigate the contribution of having a learnable scaling parameter in ADMM to convergence and accuracy. You should experiment with:

- Constraining $\alpha_i = 1$ during the optimization process but keeping the batch normalization layers.
- Removing the batch normalization layers but keeping α unconstrained.

- Constraining $\alpha_i = 1$ during the optimization process and removing the batch normalization layers.

The first two experiments are investigating whether any of the two scaling mechanisms (batch norm and weight scale) suffices to achieve convergence and keep accuracy. The last experiment is interesting, because it would produce a ternary network that is totally multiplication free, but may be too constrained to work well in practice or may suffer from lack of convergence.

Hint: Keep a detailed record of your experiments, as you will need to centralize and plot your results.

2.6 Step 6: Technical Report

You should now write a technical report about your findings. Your report should cover the following sections:

- *Problem*: What are you trying to accomplish?
- *Algorithms*: What methods are you experimenting with? Describe the methods briefly and try to relate them to each other. You can write a very short pseudocode for each (see for example Algorithm 1 in [2]). Make sure you cite the papers properly (this is very important when you do a conference submission!)
- *Experiments*: Describe your experimental procedure, including the baseline, experimental settings, hyperparameters and code.
- *Results*: Plot the performance/efficiency of your methods as described in section 2.2 and summarize the relevant results in a table. You might want to add other interesting plots. For example, how does the performance/efficiency vary during the training process (section 2.6).
- *Discussion*: Discuss your results. Be sure to comment on the merits/drawbacks of the methods, or any surprising results you might get.

Finally, using the material in the technical report, prepare a set of slides for your presentation.

References

[1] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. *Distributed Optimization and StatisticalLearning via the Alternating DirectionMethod of Multipliers.* 2010.

[2] Matthieu Courbariaux and Yoshua Bengio. Training Deep Neural Networks with binary weights during propagation. In *Neural Information Processing Systems*, 2015.

[3] Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely Low Bit Neural Network: Squeeze the Last BitOut with ADMM. In *The AAAI Conference on Artificial Intelligence*, 2018.

[4] Fengfu Li, Bo Zhang, and Ben Liu. Ternary weight networks. In *NIPS Workshop on Efficient Methods for Deep Neural Networks*, 2016.

[5] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *International Conference on Learning Representations*, 2017.