Deep Learning Project: Using Bayesian Optimization to Find Good Augmentation Policies from Data

NAMES: Samuel Frommenwiler, Gian König
NETHZ: fsamuel, koenigg
EMAIL: {fsamuel, koenigg}@student.ethz.ch
ID: 08-738-601, 09-913-245

Abstract

This document presents the results for the deep learning course at ETH as required in [2]. In the first section data augmentation as the problem of interest is described, as well as Reinforcement Learning as an approach to improve data augmentation. In the second section, the problem is described in more detail and Bayesian Optimization is introduced as an alternative approach to tune hyper parameters for automated data-augmentation policies. This section also includes the final results. Finally, a conclusion is presented.

1. AutoAugment with Reinforcement Learning and Other Approaches

Machine learning algorithms usually achieve better results with more data, however acquiring this additional data can be expensive and time-consuming. A common trick to increase the amount of training data is to add copies of existing data with small perturbations to the training set. For a dataset of natural images dataset augmentation methods include random cropping, image mirroring, rotation, color shifting and color whitening. Picking a good combination of these methods is usually done by hand and requires expert knowledge and time.

Therefore, an automated approach was introduced in order to find the best policies [7]. Cubuk et. al. present a process of finding an efficient data augmentation policy, in which each policy contains possible augmentation operations. Each operation contains an image processing function (e. g. translation, rotation or color normalization) combined with a probability that this function is applied with a corresponding magnitude. To find the best choices of these functions and suitable scaling factors, Cubuk et. al. use a Reinforcement Learning search algorithm such that a wide resnet (WRN-40-2), trained on these hyper parameters, yields the best validation accuracy.

Geng et al. [9] extended on this idea using augmented random search to efficiently find augmentation policies in a continuous hyper parameter search space. Tran et al. [12] follow a slightly different approach where they use GANs to find valid augmented data. While Fawzi et al. [6] pursued a worst-case augmentation approach where they generate augmented data which gives the biggest loss for the current classifier.

2. AutoAugment with Bayesian Optimization

We use Bayesian Optimization, a technique already used for tuning hyperparameters, to automatically select a good combination of augmentation policies. And to compare this to the approach used in [7]. Specifically, we investigate the Bayesian Optimization [8], [10] approach with a Tree Parzen Esimator [1] by using the Hyperopt library [4] with the help of [5]. For this specific task, [11] gives us a guideline on how to integrate Bayesian Optimization into the data augmentation task.

We focus on optimizing the parameters for a single network architecture WRN-28-10(Wide Residual Network) [13], using the same hyper parameters as the authors of the original AutoAugment paper [7], changing the search space from discrete to continuous and utilizing Bayesian Optimization to select the optimal parameters for the augmentation policies (and to some extent also combinations as setting the probability of some sub-policy means this sub-policy is ignored) and using a less complex WRN-40-2 than is used for the final classification task.

Training the simpler WRN-40-2 once still required about 2.5 minutes on the Google Cloud which meant trying to get close to the 15'000 evaluations the authors of AutoAugment did would've required about 26 days of running the code. We therefore decided to reduce the search-space to the policy-combinations already found by Ekin et al. [7] and check if hyperopt would be a good solution to optimize these further for CIFAR-10 in a continous search space.

2.1. Available Policies

The available policies [7] used in their program are ShearX(Y) (shear the image along the horizontal (vertical) axis with rate magnitude), TranslateX(Y) (translate the image in the horizontal (vertical) direction by magnitude number of pixels), Rotate (rotate the image magnitude degrees), AutoContrast (maximize the the image contrast, by making the darkest pixel black and lightest pixel white), Invert (invert the pixels of the image), Equalize (equalize the image histogram), Solarize (invert all pixels above a threshold value of magnitude), Posterize (reduce the number of bits for each pixel to magnitude bits), Contrast (control the

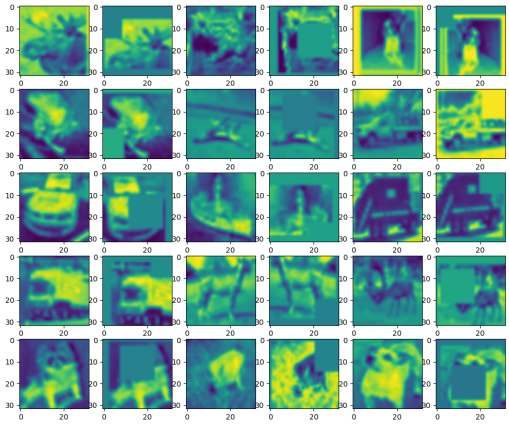

Figure 1: Example of augmentation policies applied to CIFAR-10

contrast of the image. A magnitude=0 gives a gray image, whereas magnitude=1 gives the original image), Color (adjust the color balance of the image, in a manner similar to the controls on a colour TV set. A magnitude=0 gives a black and white image, whereas magnitude=1 gives the original image), Brightness (adjust the brightness of the image. A magnitude=0 gives a black image, whereas magnitude=1 gives the original image), Sharpness (adjust the sharpness of the image. A magnitude=0 gives a blurred image, whereas magnitude=1 gives the original image), Cutout (set a random square patch of side-length magnitude pixels to gray), Sample Pairing (linearly add the image with another image (selected at random from the same mini-batch) with weight magnitude, without changing the label). The code which includes the Reinforcement Learning to generate the optimal policies is not included in their repository[1]. But the results from the optimization is included in the file `policies.py`, or in table 8 in [7]. Figure 1 shows samples of the policies applied to some images from the CIFAR-10.

2.2. Generate Optimal Policies using Bayesian Optimization

Because our computation power was limited, we decided to use only the 25 sub-policies listed in table 2 found by [7] and optimize over the probability and magnitude of each sub-policy.

The following procedure briefly describes the steps of the process to find optimal probability and magnitude for each sub-policy defined in 2 with reference to the files in our repository[2]:

1. A WRN-40-2 is trained on a reduced CIFAR-10 data set for each sub-policy defined in the search space for the number of trials specified beforehand. The optimal policies are safed.
2. Train CIFAR-10 on WRN-28-10 with optimal policies found in the step above.

2.3. Results

We performed three trainings where we optimized the probabilities and magnitude of the good policies listed in table 2. A first run with 10 trials for each policy and 10 epochs on the WRN-40-2, a second one with 20 additional trials for each policy and again 10 epochs. For the last run the search space was slightly adjusted because some found policies from the second round seemed unreasonable (translations outside of the image were in the search space). In an effort to increase accuracy the number of trials was increased to 20 per policy and the epoch number was increased to 12. The results are listed in the table 1 below. [3]

	Train Acc:	Test Acc:
Reinforcement Learning [7]	None	0.9727
Bayesian Optimization$_{S1,10,10}$	0.9353	0.9728
Bayesian Optimization$_{S1,10,30}$	0.9354	0.9684
Bayesian Optimization$_{S2,12,20}$	0.9364	0.9702

Table 1: Simulation results

We performed the training on the Google Cloud [3] using one Tesla P100 GPU. The training of the WRN-28-10 with the policies given from the Reinforcement Learning search required 9 hours 15 minutes. The time required to optimize all probabilities and magnitudes of the policies in table 2 with 20 trials for each subpolicy required 12 hours 30 minutes.

2.4. Discussion

The tests done show that it is possible to achieve similar results as the original authors of AutoAugment achieved (on a restricted search space) using Bayesian Optimization. As the code which they used to find their policies was not publicly available and they do not state the effort taken to find these policies, unfortunately we were unable to compare the efficiency of Bayesian Optimization to their Reinforcement Learning approach.

An assumption taken for our work was that the policies can be evaluated independently of each other. Given that they are drawn with equal probability that seemed like a reasonable assumption, which should be validated though in future work. Ekin et al. implicitly also showed that it is possible to find augmentation policies for a more complex network using a simpler network. An interesting question here is how simple can the evaluating network be in order to provide useful feedback for the final training. Simpler networks obviously have the advantage that they are faster to train, therefore allow for more Bayesian Optimization evaluations in the same amount of time. The third

[1] https://github.com/tensorflow/models/tree/master/research/autoaugment
[2] https://github.com/giankoenig/DL18
[3] While writing this report we noticed that it would've been super helpful to also have a baseline which shows the results you get using the good policy combination but with randomly drawn probabilities and magnitudes

question that was raised by our work is how many evaluations of Bayesian Optimization are needed in this search space to find good values for the data augmentation hyperparameters.

3. Summary

Bayesian Optimization with a WRN-40-2 was tested as a means to find good augmentation policies for a WRN-28-10 and CIFAR-10. The tests done showed that it was indeed possible to find viable policies for the reduced search space within a reasonable computational effort. Our work raised several interesting questions related to the application of Bayesian Optimization in the area of data augmentation but unfortunately we were not able to give a definitive answer on the efficiency of our approach compared to the Reinforcement Learning approach chosen by Ekin et al.

4. Appendix

4.1. Other Infrastructure

A first attempt to run the code on the CPU with MacBook Pro (Retina, 15-inch, Early 2013) with 2.4 GHz Intel Core i7, 16 GB 1600 MHz DDR3, Intel HD Graphics 4000 1536 MB, macOS Sierra 10.12.5 resulted in a Training Time: of `INFO:tensorflow:Epoch time(min): 332.056`

We tried to execute the code on the Leonhard Cluster[4]. For this the following commands were necessary. 1) `ssh [NETHZ]@login.leonhard.ethz.ch` 2) Create folders and git clone or upload the data from PC `rsync -Pav /PATH [NETHZ]@login.leonhard.ethz.ch:/[PATH])` and 3) Load the right moduel with `module load python_gpu/2.7.13` (module list: Currently Loaded Modules: 1) StdEnv 2) gcc/4.8.5 3) openblas/0.2.19 4) cuda/8.0.61 5) cudnn/6.0 6) jpeg/9b 7) libpng/1.6.27 8) python_gpu/2.7.13) 4) Execute the code: `bsub -n 20 -R "rusage[mem=4500,ngpus_excl_p=1]" python train_cifar.py --model_name=wrn --checkpoint_dir=tmp/training/ --data_path=tmp/data/ --dataset='cifar10' --use_cpu=0.`

It turns out on the Leonhard Cluster we can only use 1 GPU and also each job has a time limit of 4h. For this run, we could analyze the output using `grep -A3 "Creating TensorFlow device" lsf.o1058716` which shows that in 4 hours only 47 epochs were executed. From the data below we can see that one epoch uses approximately 5 minutes. Another problem was that when we wanted to start a job, the queue took a long time (up to 24h).

[koenigg@lo-login-02 model]$ ls -l total 570652 -rw-rw—- 1 koenigg koenigg-group 127 Jan 8 13:49 checkpoint -rw-rw—- 1 koenigg koenigg-group 289836892 Jan 8 **13:44** model.ckpt-40.data-00000-of-00001 -rw-rw—- 1 koenigg koenigg-group 9868 Jan 8 13:44 model.ckpt-40.index -rw-rw—- 1 koenigg koenigg-group 1163579 Jan 8 13:44 model.ckpt-40.meta -rw-rw—- 1 koenigg koenigg-group 289836892 Jan 8 **13:49** model.ckpt-41.data-00000-of-00001 -rw-rw—- 1 koenigg koenigg-group 9868 Jan 8 13:49 model.ckpt-41.index -rw-rw—- 1 koenigg koenigg-group 1163579 Jan 8 13:49 model.ckpt-41.meta

[koenigg@lo-login-01 model]$ date Thu **Jan 10 09:45:45** CET 2019 [koenigg@lo-login-01 model]$ bjobs JOBID USER STAT QUEUE FROM_HOST EXEC_HOST JOB_NAME SUBMIT_TIME 1060305 koenigg PEND gpu.4h lo-login-01 *use_cpu=0 **Jan 9 22:00**

Therefore using Leonhard was not a practical solution to train the models.

Note: Useful commands: `bjobs` to show the jobs (with option -l for more details), `module avail` to show available modules, `module list` to show loaded modules, `ls -l --block-size=M` to show detailed files with size in MB, Sync simulation results to local computer: rsync -chavzP –stats koenigg@login.leonhard.ethz.ch:/.../model.ckpt-199.meta ∼/.../autoaugment/tmp/training/model/.

	Operation 1	Operation 2
Sub-policy 0	Invert	Contrast
Sub-policy 1	Rotate	TranslateX
Sub-policy 2	Sharpness	Sharpness
Sub-policy 3	ShearY	TranslateY
Sub-policy 4	AutoContrast	Equalize
Sub-policy 5	ShearY	Posterize
Sub-policy 6	Color	Brightness
Sub-policy 7	Sharpness	Brightness
Sub-policy 8	Equalize	Equalize
Sub-policy 9	Contrast	Sharpness
Sub-policy 10	Color	TranslateX
Sub-policy 11	Equalize	AutoContrast
Sub-policy 12	TranslateY	Sharpness
Sub-policy 13	Brightness	Color
Sub-policy 14	Solarize	Invert
Sub-policy 15	Equalize	AutoContrast
Sub-policy 16	Equalize	(Equalize
Sub-policy 17	Color	Equalize
Sub-policy 18	AutoContrast	Solarize
Sub-policy 19	Brightness	Color
Sub-policy 20	Solarize	AutoContrast
Sub-policy 21	TranslateY	TranslateY
Sub-policy 22	AutoContrast	Solarize
Sub-policy 23	Equalize	Invert
Sub-policy 24	TranslateY	AutoContrast

Table 2: Search Space: These policies are used and the probability (P) and magnitude (M) attached to each operation is optimized.

[4]https://scicomp.ethz.ch/wiki/Leonhard

References

[1] Automated model tuning. https://www.kaggle.com/willkoehrsen/automated-model-tuning. Accessed: 2018-12-08.

[2] Deep Learning, data analytics lab. http://www.da.inf.ethz.ch/teaching/2018/DeepLearning/. Accessed: 2018-12-08.

[3] Google cloud platform. https://cloud.google.com/. Accessed: 2018-12-08.

[4] Hyperopt, distributed asynchronous hyperparameter optimization in python. http://hyperopt.github.io/hyperopt/. Accessed: 2018-12-08.

[5] An introductory example of baysien optimization in python with hyperopt. https://bit.ly/2KvgSuq. Accessed: 2018-12-08.

[6] D. T. P. F. Alhussein Fawzi, Horst Samulowitz. Adaptive data augmentation for image classification.

[7] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501, 2018.

[8] P. I. Frazier. A Tutorial on Bayesian Optimization. *ArXiv e-prints*, page arXiv:1807.02811, July 2018.

[9] M. Geng, K. Xu, B. Ding, H. Wang, and L. Zhang. Learning data augmentation policies using augmented random search. *CoRR*, abs/1811.04768, 2018.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] T. Tran, T. Pham, G. Carneiro, L. Palmer, and I. Reid. A Bayesian Data Augmentation Approach for Learning Deep Models. *arXiv e-prints*, page arXiv:1710.10564, Oct. 2017.

[12] T. Tran, T. Pham, G. Carneiro, L. J. Palmer, and I. D. Reid. A bayesian data augmentation approach for learning deep models. *CoRR*, abs/1710.10564, 2017.

[13] S. Zagoruyko and N. Komodakis. Wide residual networks. *CoRR*, abs/1605.07146, 2016.