Bag of Tricks for Faster & Stable Image Classification

Aman Bansal
Stanford University
aman0456@stanford.edu

Shubham Anand Jain
Stanford University
shubhamj@stanford.edu

Bharat Khandelwal
Stanford University
bharatkh@stanford.edu

Abstract

In recent years, breakthroughs have brought us closer to our goal of Artificial General Intelligence. GPT-3 [2], one such recent success, took 3640 PF-days to train, and cost over 5M dollars. In an era where we use progressively larger pre-trained models and fine-tune on downstream tasks, an important question that remains to be answered is efficient (re)training of such models; such techniques can not only save compute time and help research progress at a faster rate, but also help reduce our carbon footprint. Accordingly, many papers in top conferences focus on techniques for faster and stable (i.e, better generalization ability) training of architectures. In this paper, we specifically focus on various techniques from recent papers (mainly NeurIPS 2020 & 2021) such as Bandit-based Importance Sampling, Convolutional Normalization, approximate tensor operations and Gradient-based weight initialization, and apply these techniques on an image classification problem.

Our results show that using these methods in combination can lead to a speedup of the order of ***1.5-2.5***$\times$ *along with higher validation accuracy with models such as VGG-16 and ResNet-18, and on even small datasets such as CIFAR-100. We provide intuition as to why and when these methods work, and what regimes of hyper-parameters work best for different model architectures.* ***We combine our 4 different proposed methods in a easy to use plug-and-play module (link can be found here), which allows any PyTorch nn.Module model to be trained with any combination of these methods on a range of datasets.***

1. Introduction

In this project, we explore faster and more stable (in terms of lower generalization gap) methods to train models for image classification tasks. To this end, we consider recent papers from top conferences; there has been a blowup in the number of papers at these conferences in the past years, and it was interesting to us to further accelerate the pace of ML with techniques from reading and implementing/working on multiple papers.

We tackle the problem of faster and stable image classification. The image classification problem statement can be formalized as follows: Given a dataset $\mathcal{D}$ consisting of images $x_{i=1}^{n}$ each belonging to one of k classes having labels $y_{i=1}^{n}, y_i \in \{1, .., k\}$, we want to create a model that outputs predictions $m(x_i) = \hat{y}_i, \hat{y}_i \in \{1, .., k\}$.

By "faster and stable" image classification, we mean that the training algorithm should be able to optimize the loss to a similar amount but in lesser time and with lesser generalization error (that is, higher accuracy or lower loss on held-out datasets). The metrics we plan to use to compare various techniques (and combinations) are

1. Accuracy: $\frac{1}{n}\mathbb{1}\left(y_i = \hat{y}_i\right)$
2. Loss values
3. Time taken to train the model

With the growing number of larger models being trained, it is increasingly useful to have a number of tricks that make training and optimizing our models easier, along with a high level intuition of how these work; in this paper, we try to focus on this aspect.

Another motivation of our project is to highlight how the small advances each paper makes can be combined to improve the training of our ML models significantly; with the increasing number of papers, it can be hard to keep track of the improvements offered by each paper, and providing all of these advantages in an easy-to-use module is one of our main goals with this project. Our module can be found here; you can add your own model architecture in models.py, and simply run the script in main.ipynb similar to the given examples to check whether these methods might help training your model. More detailed instructions are given in the README on Github.

2. Related Work

In this paper we work on the image classification task (however, the methods we discuss are suited for any general

purpose task), and try various (recently published) ideas that help learn networks faster and might be relevant today, and merge them together. At a high level, some ideas that may be considered can be classified as follows:

1. **Importance sampling:** This is a method to assign different "importance" weights to samples in order to sample training data with more informative gradients relatively frequently. While this is generally done using loss values due to computational constraints (though using the gradient norm works better), [9] proposed a per-sample gradient upper bound that works well in practice. [11] propose a somewhat different importance sampling method based on bandit sampling termed as AdamBS, and show that using Adam along with their method leads to asymptotically faster convergence.
2. **Sparsification of networks:** Neural networks today are significantly over-parameterized, with most neurons not contributing to the output in a significant way. We can leverage this fact to speed up training by pruning certain channels; [16] suggest a complete sparse training method by sampling from a Bernoulli mask and not propagating gradients through pruned weights.
3. **Adaptive Dropout:** Dropout is a classic method used to regularize neural networks, and to force neurons to learn useful representations. However, we often use dropout with the same drop parameter for all neurons in the network during training, which intuitively does not seem to match the test-time distribution because the contributions of neurons are not equal in the final output. The paper [15] proposes a dropout network that learns parameters along with the original neural network; they obtain the surprising observation that a good dropout network tends to regularize parameters according to their magnitude.
4. **Network Initialization:** As we go toward deeper neural networks, initialization becomes a crucial factor; choosing the correct initialization avoids issues like exploding/vanishing gradients and accelerates training. To this end, in the past decade we have seen proposals such as Xavier initialization [5] and Kaiming initialization [6]. Recently, [17] proposed a process called "GradInit" to initialize weight by performing small batches of initialization training and tuning to allow for large but stable gradient updates in the first epoch.
5. **Approximate tensor operations:** The paper [1] uses approximate tensor operations to speed up training. The main idea is to use CRS (column-row sampling), which is a sample based approximation to tensor multiplication. They claim a speed-up of about 1.37x.
6. **Better convolution & normalization:** Batch normalization [8] is a technique that is used to reduce internal co-variate shift in networks by resetting the mean and variance per layer. The paper [12] proposes a layer called "ConvNorm" that exploits the dual multiplicative structure (in the fourier domain) of the convolutional layer in CNNs, and performs a normalization in this space. This is claimed to smoothen the optimization landscape. They observe fast, stable, and robust training with ConvNorm as compared to other normalization techniques.

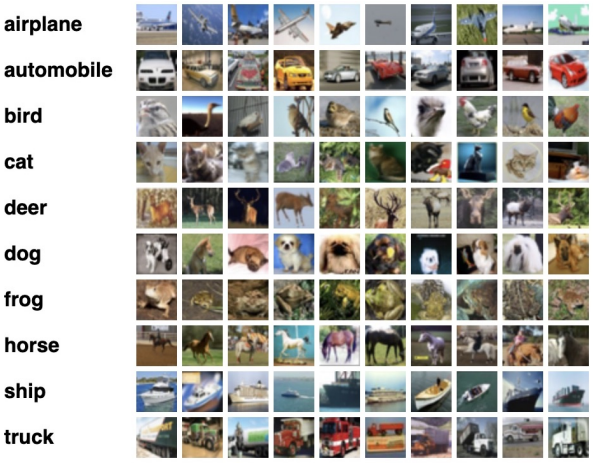

Figure 1. Examples from CIFAR-100 dataset

In this project, we compare results of Importance Sampling, Network Initialization, Approximate tensor operations and ConvNorm, and combine these to train models on a single image classification task to quantify the advantage we can gain.

3. Data

For all our experiments in this report, we use the CIFAR-100 dataset [10]. Samples from this dataset can be seen in Figure 1. Dataset details are as follows:

- Number of classes: 100
- Training images: 50000; 5000 per class
- Test images: 10000 test images total
- Train-validation split: 80-20
- Preprocessing: We normalize the dataset with the mean and standard deviation of the trainset
- Resolution: $32 \times 32 \times 3$ images

Our repository also contains modular implementations of other datasets such as ImageNet, MNIST/FashionMNIST, and Places365, but we do not present results on them. Note that when using models with 3-channel inputs for datasets with a different number of color channels like MNIST, our code automatically adds an extra convolutional layer with stride 1 and kernel size 1 before the 3-channel model to fix dimensions, allowing

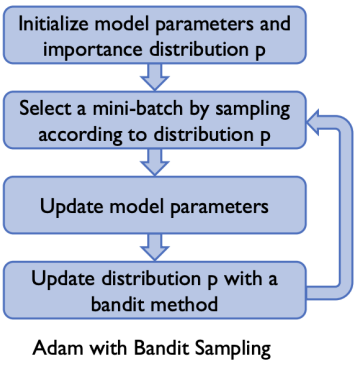

Figure 2. Importance Sampling (with Bandit Update)

us to experiment with famous 3-channel architectures on datasets with different number of channels per image.

4. Methods

Our baseline models for this project are SimpleCNN (diagram in Appendix B), VGG-16 [14] and ResNet-18 [7]. We use these models in combination with Importance Sampling, GradInit, CRS-sampling and ConvNorm, and test their performance on image classification datasets such as CIFAR-10 and CIFAR-100.

In the coming subsections, we explain the various strategies that we choose to explore for our project.

4.1. Importance Sampling

The idea of importance sampling is that taking data points uniformly at random for creating a mini-batch is not the best strategy since all points need not contribute equally to the model's learning. Some points might be more informative for the model to train on, and the idea of importance sampling is to sample the mini-batch by sampling from a weighted distribution which represents the "importance" of the points. With p denoting the distribution over data points, the algorithm for importance sampling is given in Figure 2.

4.1.1 Reproducing Bandit Sampling Update

We first explored the exact importance sampling method described in [11] and illustrated in Figure 2. The Bandit update step is given in Algorithm 1. The terminology is as follows:

- p^t: The importance sampling distribution at time t
- I^t: The batch sampled at time t
- $\{g_{I_k^t}\}_{k=1}^K$: The gradients at time t for all the points in the batch

Algorithm 1 The distribution update rule for p^t.

Function: $update(p^{t-1}, I^t, \{g_{I_k^t}\}_{k=1}^K)$
  Compute loss $l_{t,j} = -\frac{\|g_j^t\|^2}{(p_j^t)^2} + \frac{L^2}{p_{min}^2}$ if $j \in I^t$; otherwise, $l_{t,j} = 0$;
  Compute an unbiased gradient estimate $\hat{h}_{t,j} = \frac{l_{t,j} \sum_{k=1}^K \mathbb{1}(j=I_k^t)}{K p_j^t}, \forall 1 \leq j \leq n$;
  $w_j^t = p_j^{t-1} \exp(-\alpha_p \hat{h}_{t,j}), \forall 1 \leq j \leq n$;
  $p^t = \arg\min_{q \in \mathcal{P}} D_{kl}(q \| w^t)$;
  Return: p^t

Algorithm 2 (Novel) The distribution update rule for p^t.

Function: $update(p^{t-1}, I^t, model, \gamma, \beta)$
  Compute loss over I^t: $l^t = model(I^t)$
  Add the momentum component $p_{t,j} = \gamma * p_{t-1,j} + (1-\gamma) * l^t$, if $j \in I^t$
  $p_j^t = \frac{p_{t,j}}{1-\beta^t}$, if $j \in I^t$; otherwise, $p_j^t = p_j^{t-1}$;
  Return: p^t

4.1.2 Novel Ideas

1. The Bandit Sampling Algorithm as presented in [11] has a major deficit - it is not robust to outliers. Giving weights to individual points based on their informative power for the model could lead to outliers getting a lot of weight, leading to mini-batches that are full of outliers. Therefore, we devised a scheme to mitigate this issue: giving weights to each output class instead of individual points. Giving weight to classes ensures that the model is not neglecting any class that doesn't have outliers, as well as not focusing on the outliers any more than it would have without importance sampling.
2. We found that the bandit sampling algorithm is slow because it tries to maintain weights per datapoint and calculates gradients per datapoint. [9] proposes a strategy to weigh points by their loss values instead of gradients. We further built upon this to take loss momentum into account along with a bias factor, similar to the idea in the Adam optimizer. The final algorithm is given in 2.

4.2. GradInit

GradInit is an automated way to initialize neural network weights to speed up training. The intuitive idea behind GradInit is to set learning rates in a way that minimizes loss after an epoch of training, while ensuring that we do not run into the exploding or vanishing gradients landscape, and try to minimize gradient variance. The idea behind GradInit is as follows:

Sample one batch of samples, S. Let $\tilde{\theta} = \theta - \eta \nabla \ell$ be

the updated model parameters that minimize the loss on S; Update the weights to minimize the gradients on a new batch of samples $\tilde{S}$ with parameters $\tilde{\theta}$, while ensuring gradient clipping to avoid exploding gradients. Intuitively, the idea seems very similar to meta-learning. Indeed, the main comparison of GradInit is performed with MetaInit [3], which is a method that tries to smoothen second-order gradients around a point using meta-learning, effectively making it a third order computation.

There are a few hyperparameters in GradInit; the important ones are γ (the gradient clipping parameter $= 1$), a lower bound on the ratio of new to old weights $\alpha = 0.01$, the gradinit optimizer and learning rate (different from the model's optimizer and learning rate), and the intersection ratio of batches used to compute loss and gradient $\frac{|S \cap \tilde{S}|}{|S|}$, which is set to 0.5 which seems to perform best in practice (the idea is that having completely disjoint sets of batches leads to a bad estimate when we have noisy gradients).

4.3. Approximate Tensor operations

We implement the method from the paper [1], which uses an approximation to tensor operations that is faster in practice, and also seems to help speed up neural network training. The idea is to use CRS (column-row-sampling) [4], which approximates matrix multiplication by sampling random columns from A^T and random rows from B with normalization to get an unbiased estimate of $A^T \cdot B$, while minimizing the Frobenius norm; the key thing to note is that columns/rows are sampled with a weighted probability proportional to the magnitude of the column/row.

Note that approximations are only done at training time, not at validation/test time. We came up with the additional hypothesis (separate from the paper) that it is more acceptable to approximate final layers instead of initial layers, and we can get away with a higher rate of approximation; we also test this hypothesis in our experiments. The idea is that there is a ripple effect; approximating initial layers can lead to higher error that adds across layers, and we avoid this by only approximating final layers; moreover, since many architectures are parameter-heavy at the end, this is an additional benefit of this method.

4.4. ConvNorm

Normalization techniques like Batch Normalization and Layer Normalization are an important part of efficiently training deep convolutional neural networks. A disadvantage of the above mentioned normalization technique examples are that they do not preserve the convolutional-properties like translation invariance. Convolutional Normalization (ConvNorm) [12] is a technique that preserves translation invariance of deep CNNs, and therefore is very useful in dealing with image data.

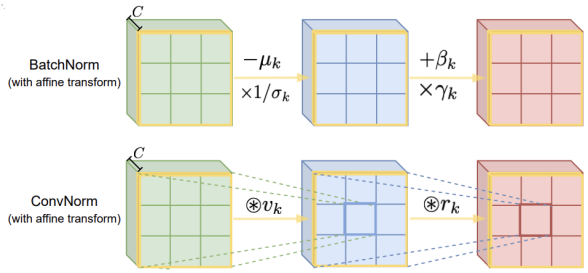

Figure 3. ConvNorm [12]

The general idea of ConvNorm is depicted in Figure 3. We now give an intuition of how ConvNorm works and we refer the reader to [12] for further details. Let's assume we're dealing with a 1D convolution. This idea can easily be extended to 2D conv-nets.

- The convolution that we have is $\mathbf{z}_{out} = \mathbf{a} * \mathbf{z}_{in}$, where $\mathbf{a}$ is the kernel. This can be written as a product with a circulant matrix (matrix with each row shifted by 1 from the previous row): $\mathbf{z}_{out} = C_{\mathbf{a}} \mathbf{z}_{in}$.
- The paper [13] showed that normalizing $\mathbf{z}_{out}$ via *preconditioning* improves the learning of the weight matrices by making some poor local optimizers unreachable. They do so by multiplying with a preconditioning matrix which can be approximated for our use case as: $\mathbf{P} = \left(\mathbf{C_a C_a^\top}\right)^{-1/2}$. This gives: $\widetilde{\mathbf{z}}_{out} = \mathbf{P}\mathbf{z}_{out} = \underbrace{\left(\mathbf{C_a C_a^\top}\right)^{-1/2} \mathbf{C_a}}_{\mathbf{Q(a)}} \cdot \mathbf{z}_{in}$.
- Since $\mathbf{P}\mathbf{P}^\top = \mathbf{I}$, we can write $\mathbf{P}$ as $\mathbf{C_v}$. Thus we can write the original equation as $\widetilde{\mathbf{z}}_{out} = \mathbf{C_v} \cdot \mathbf{C_a} \mathbf{z}_{in}$ where $\mathbf{v}$ refers to $\mathbf{v}_k$ in Figure 3.
- The paper [12] showed that instead of running full matrix operations, $\widetilde{\mathbf{z}}_{out}$ can be computed much more efficiently using a FFT.

In this paper, we further build upon the work of [12] and try to analyse how useful these ConvNorm layers are in conjunction with our other methods. Moreover, we think of ways to boost the performance of these layers.

The trade-off offered by this layer is that it is much more computationally expensive than other normalized convolution techniques but it provides more stability by reducing the possibility of our model getting stuck in a poor local optima. Our intuition suggests that replacing regular convolutional and normalization layers with ConvNorm will have diminishing returns as we replace more and more layers while the computational complexity will keep increasing.

Hence, we experimented with replacing a small number of layers with ConvNorm instead of all the layers, and locating the optimal place in the network to do so.

5. Experiments

We first look at each method independently, performing a search for optimal hyper-parameters, and then combine the methods that work well.

In all our experiments apart from importance sampling (for which hyperparameters are given separately), our optimizer is Adam, and the other common hyperparameters can be found in Table 1. The learning rates were chosen by parameter sweeps on the base model, and the batch size was chosen since it is the largest that can fit in a GPU with capacity 16GB memory, which is a common configuration. We look at each of the proposed methods independently and see if they help boost the performance of our models. All the results we report are on a separate validation set that isn't used to tune hyperparameters, so that we can have a more realistic estimate of test accuracy.

Model	LR	Num Epochs	Batch Size
ResNet-18	10^{-4}	15	1024
VGG-16	10^{-5}	60	1024

Table 1. Model hyperparameters common across experiments

5.1. Importance Sampling

We use the following settings for the experiment:

- Dataset: CIFAR-10
- Model: SimpleCNN (see Figure 15 and Table 2 in Appendix B)
- Sampling strategies: a) Uniform, b) Basic Importance Sampling (weight points based on the magnitude of their gradients), c) Bandit Sampling (AdamBS) [11], d) Bandit Sampling (our implementation of AdamBS after fixing the bugs in [11])
- Hyperparameters: {Batch Size: 128, Learning rate: 0.003, Learning rate reductions: 10000}

We present the training and validation plots in Figure 4 and 5 (for CIFAR-10 dataset with SimpleCNN model) respectively.

We can infer the following from the plots:

- Basic Importance Sampling performs better than uniform sampling. The loss falls slightly faster (though not statistically significant) but the validation accuracy achieved is significantly higher.
- We were able to reproduce the results of Bandit Sampler [11]. However, the results of that paper are erratic. As seen in Figures 4 and 5, for the line labelled by *Bandit (AdamBS)* the training loss does go down suddenly at some particular step but accuracy also stops increasing from thereon. It seems that the huge sudden drop at an arbitrary step leads to the better performance on training for AdamBS, but also causes the model to do much worse on validation. We went through the author's Tensorflow code and found a deviation in their code from their paper.
- We fixed the author's code to match their paper and experimented with it resulting in the *Bandit (ours)* line in Figures 4 and 5. The plot is now smoother and more consistent but the validation accuracy increases slower than both uniform and basic importance sampling. We therefore conclude that this is not the best sampling strategy.
- We then changed the author's code to implement our proposed bandit updates at class level. We also implemented our own idea of weighing points by their batch's loss value and momentum from scratch in PyTorch. Both these strategies did not perform any better than the above strategies so we do not include them in our plots; however, our GitHub repository does contain an option to perform this version of importance sampling.

5.2. GradInit

For GradInit, we perform a sweep over the learning rates for the GradInit optimizer. We take the GradInit optimizer to be Adam, keep the gradient clipping $\gamma = 1$, and choose GradInit iterations = 200. We provide plots for both ResNet-18 and VGG-16 for GradInit in Figures 6, 7.

We note that GradInit seems to help in both ResNet-18 and VGG-16. The optimal learning rate for the GradInit Adam optimizer itself in ResNet-18 is found to be 10^{-4}, and for VGG-16 it is found to be 10^{-5}, which is very close to the model learning rate; thus, we recommend keeping the GradInit LRs to be in the same range as model learning rates. We use this learning rate for GradInit in further experiments. Note that in the Figures, we do not take into consideration the time taken to initialize the weights; this is because this cost is amortized over multiple hyperparameter searches and thus is negligible compared to the actual training time (about 1% of the total training time for large datasets).

5.3. Approximate Tensor Operations

For CRS sampling, we first perform the vanilla sweep over approximation ratio in Figure 8. We note that all approximation ratios perform worse than simply doing the whole matrix multiplication with respect to time. While we hypothesize that this is due to our dataset being smaller

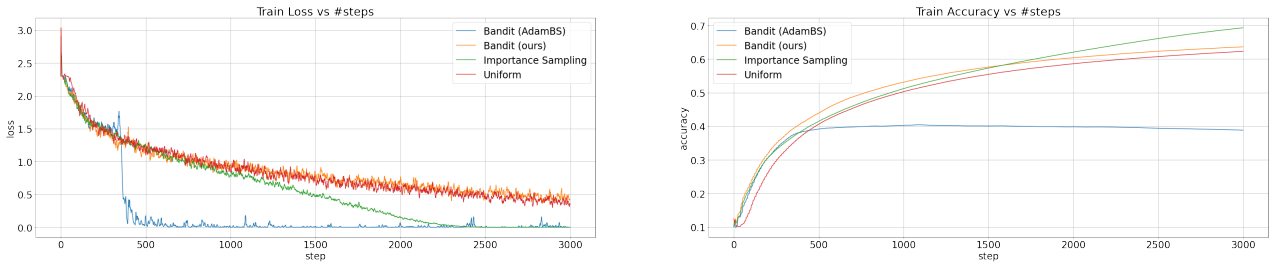

Figure 4. SimpleCNN Importance Sampling Training

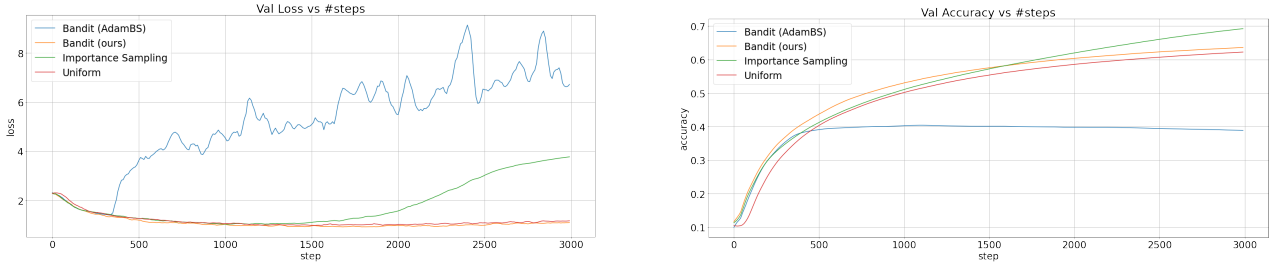

Figure 5. SimpleCNN Importance Sampling Validation

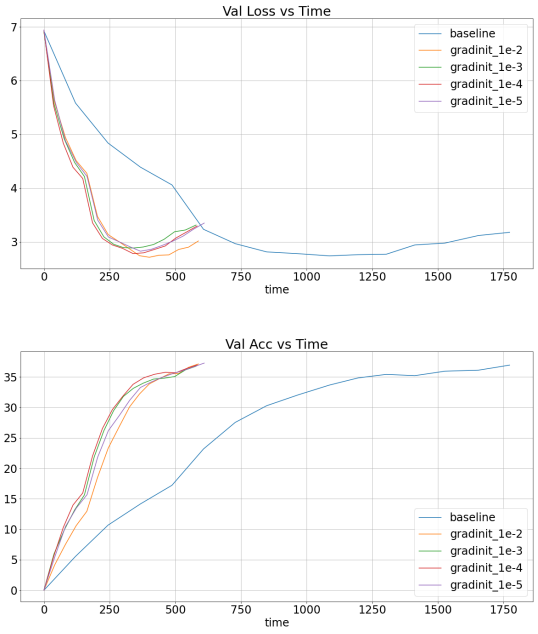

Figure 6. Resnet18 Gradinit validation vs time

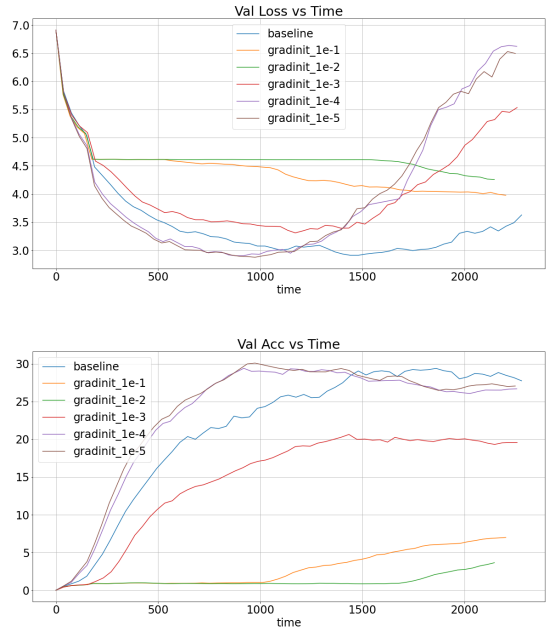

Figure 7. VGG16 validation statistics vs time

compared to the original paper (the approximation will cause a larger saving in time when the dataset is harder to learn; the time saved will accumulate over epochs), we try out a new idea; since the first few layers are usually more important in determining the output as they develop the basic features that get used by all downstream layers, we approximate only the last few layers, the hypothesis being that this should not lead to a loss in accuracy but will reduce training time significantly, especially due to the costly fully connected classification layers at the end.

Thus, for CRS sampling, we perform a sweep over which

layers we should approximate, keeping the approximation ratio constant at 0.8. Note that we only replace Convolutional and Linear layers with their approximate versions. We try out 4 different variations of layers to remove, and provide all of this functionality in our code:

- Replace first x layers
- Replace last x layers
- Replace first $x\%$ fraction of layers
- Replace last $x\%$ fraction of layers

After looking at the plot from Figure 9, we realize that approximating the last layer is indeed significantly better than approximating the first few layers, and thus we decide to approximate the last 1 layer for ResNet-18.

For VGG-16, we do not observe a benefit on using CRS-sampling. We hypothesize this to be because VGG-16 does not contain any kind of normalization, which makes an approximation unstable.

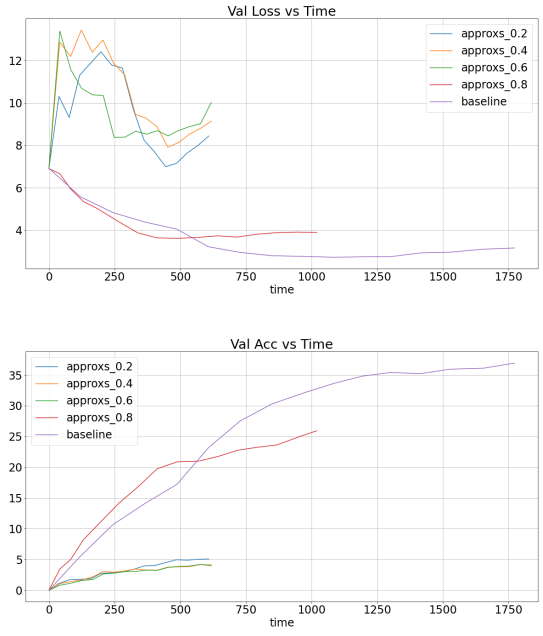

Figure 8. Resnet18 CRS ratio sweep over all layers

5.4. Convolutional Normalization

For ConvNorm, we perform a sweep by replacing different Conv2d layers with ConvNorm. We experiment with 2 major variations:

1. *Replacing first $x\%$ layers (or replace first x layers)*: The intuition behind this strategy is that the first few layers are very crucial for deciding which features will be chosen for classification and reducing the bad local minimums for those layers could become crucial for the whole training. One disadvantage, however, is that these layers usually learn to generate features that don't depend a lot on the downstream classification task. Hence changing these layers might adversely affect fundamental image recognition features.
2. *Replacing last $x\%$ of layers (or replace last x layers)*: These layers are the ones that are actually doing the specific classification task leveraging the features generated by the initial layers. Replacing them will positively impact gradient descent the most. Moreover, these are not the feature generating layers so they don't have the disadvantage that replacing first few layers have.

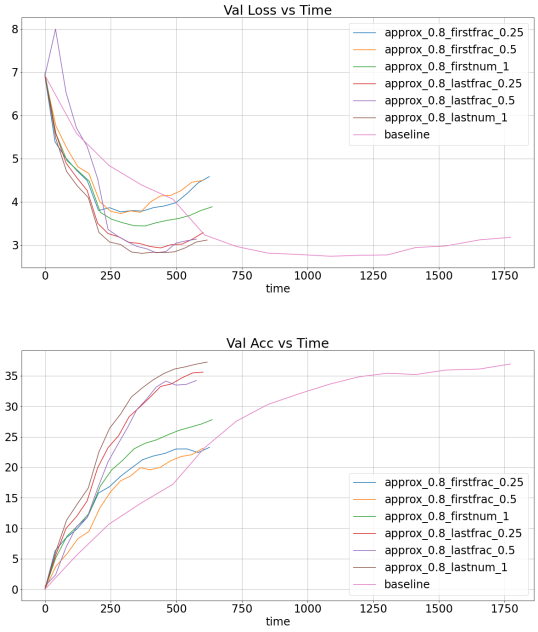

Figure 9. Resnet18 CRS validation statistics v/s time

Metric: The trade-off that we face with ConvNorm is of getting better stability as measured by validation accuracy at the cost of speed as measured by time taken to run an epoch. We, therefore, decided to compare the different variations by plotting the validation accuracy vs. time and validation accuracy vs. epoch plots.

The plots generated for the above experiment with settings given in Table 1 is shown in Figure 10. The plots show that the strategy of replacing first 50% layers (in green line) performs the best and performs significantly better than the baseline (in blue line). Note that we get higher accuracy without much jump in time taken. We choose to replace the first 50% variation for our combined model.

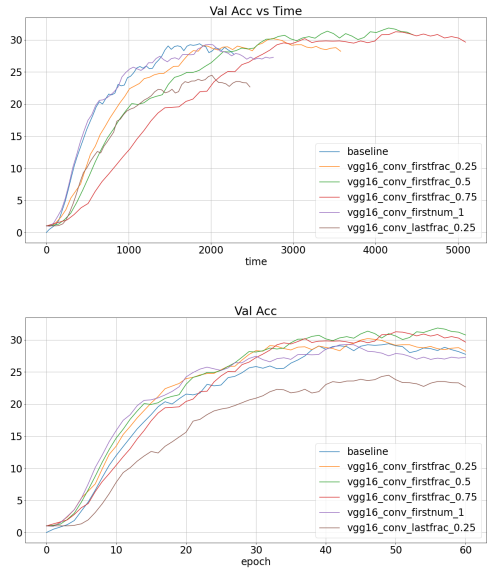

Figure 10. VGG16 Conv-Norm Layer sweep

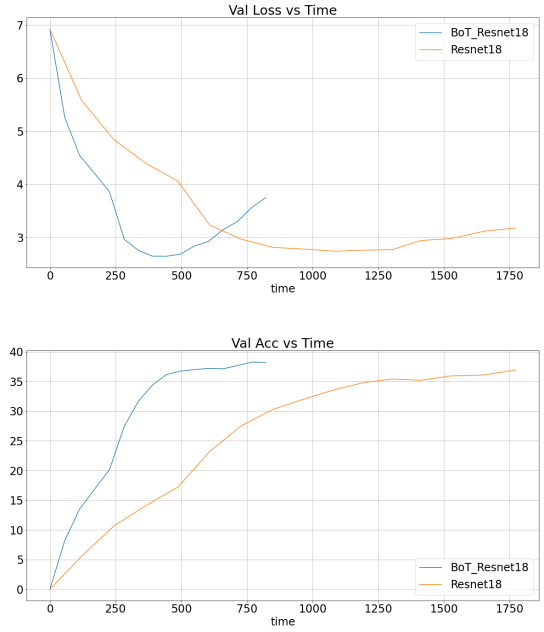

Figure 11. Resnet18 BoT Validation statistics v/s Time

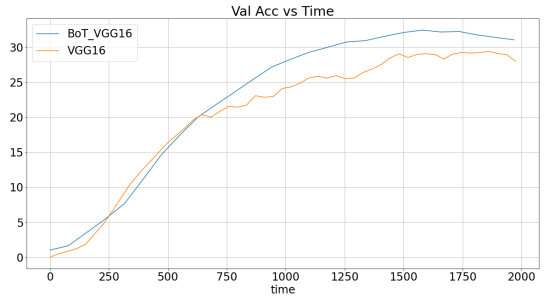

Figure 12. VGG16 BoT Validation accuracy v/s Time

5.5. Combined Models

Considering all our observations above, we create combined models for ResNet-18 and VGG-16 to measure our jump in performance.

5.5.1 ResNet-18

For the combined ResNet-18 model, which we term as ResNet-BoT (Bag of Tricks), we decide to use GradInit, CRS-sampling and ConvNorm. The results can be seen in Figure 11, and additional images can be seen in Appendix A, Figures 13, 14. We note that the base ResNet-18 observes the minimum validation loss at around 1100 seconds, whereas ResNet-18 BoT observes a smaller validation loss at around 450 seconds, which is a speed difference of $2.4\times$! Moreover, ResNet-BoT has a much higher best validation accuracy compared to base ResNet. Even if we look at the plots of Loss or accuracy v/s Loss, we note that ResNet BoT does better; this is likely the effects of GradInit and ConvNorm, which lead to stable training. Overall, our method seems to be faster and better at every epoch!

5.5.2 VGG-16

For the combined VGG-16 model, we decide to use GradInit and ConvNorm. The results can be seen in Figure 12; we note that VGG-16 BoT is about $1.5\times$ faster than the base VGG-16 in reaching an accuracy of 28%, and achieves a much higher peak validation accuracy than the base VGG-16.

6. Conclusion

In this paper, we analyze and quantify the advantages of multiple recent advances in machine learning. We implement Importance Sampling, GradInit, CRS Sampling for approximate tensor operations and Convolutional Normalization, and show that combining these methods can lead to a significant improvement in training neural networks for image classification. We also provide code that can run a combination of these methods on any module. Given more time and resources, an interesting future work is to check how these methods do on larger models such as ResNet110/ResNet1202 and datasets such as ImageNet. If these significantly help training with these larger datasets, we believe this is a useful contribution that can be folded into the training pipeline for DNNs today.

7. Contributions

Equal Contribution.

Bharat and Aman worked on setting up importance sampling code in Tensorflow and ran all experiments. Aman also rewrote the importance sampling pipeline in PyTorch to integrate it with the other methods. Shubham and Bharat worked on setting up the pipeline, and Shubham did a major chunk of the literature review for the project. Bharat also implemented all the log parsing and plotting utilities. All written components were done by each member equally.

While we significantly modified code (such as in Importance Sampling) from each repository (in some places to try our own methods, and in others to fix compatibility of all packages across methods to the same version), we borrowed some code for each method from the following repositories:

1. Importance Sampling (with & without Bandit algorithms) - AdamBS Code, Importance Sampling
2. GradInit - GradInit Repository
3. Approximate Tensor Operations - Faster NN training with approximate tensor operations
4. ConvNorm - ConvNorm Repository

References

[1] Menachem Adelman, Kfir Levy, Ido Hakimi, and Mark Silberstein. Faster neural network training with approximate tensor operations. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 27877–27889. Curran Associates, Inc., 2021. 2, 4

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. 1

[3] Yann N Dauphin and Samuel Schoenholz. Metainit: Initializing learning by learning to initialize. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. 4

[4] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast monte carlo algorithms for matrices i: Approximating matrix multiplication. *SIAM Journal on Computing*, 36(1):132–157, 2006. 4

[5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. 2

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015. 2

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. 3

[8] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. 2

[9] Angelos Katharopoulos and Francois Fleuret. Not all samples are created equal: Deep learning with importance sampling. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2525–2534. PMLR, 10–15 Jul 2018. 2, 3

[10] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. 2

[11] Rui Liu, Tianyi Wu, and Barzan Mozafari. Adam with bandit sampling for deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 5393–5404. Curran Associates, Inc., 2020. 2, 3, 5

[12] Sheng Liu, Xiao Li, Yuexiang Zhai, Chong You, Zhihui Zhu, Carlos Fernandez-Granda, and Qing Qu. Convolutional normalization: Improving deep convolutional network robustness and training. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 28919–28928. Curran Associates, Inc., 2021. 2, 4

[13] Qing Qu, Xiao Li, and Zhihui Zhu. A nonconvex approach for exact and efficient multichannel sparse blind deconvolution. In *Advances in Neural Information Processing Systems*, pages 4017–4028, 2019. 4

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. 3

[15] Sida Wang and Christopher Manning. Fast dropout training. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 118–126, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. 2

[16] Xiao Zhou, Weizhong Zhang, Zonghao Chen, SHIZHE DIAO, and Tong Zhang. Efficient neural network training via forward and backward propagation sparsification. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 15216–15229. Curran Associates, Inc., 2021. 2

[17] Chen Zhu, Renkun Ni, Zheng Xu, Kezhi Kong, W. Ronny Huang, and Tom Goldstein. Gradinit: Learning to initialize neural networks for stable and efficient training. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 16410–16422. Curran Associates, Inc., 2021. 2

A. Additional Experiments

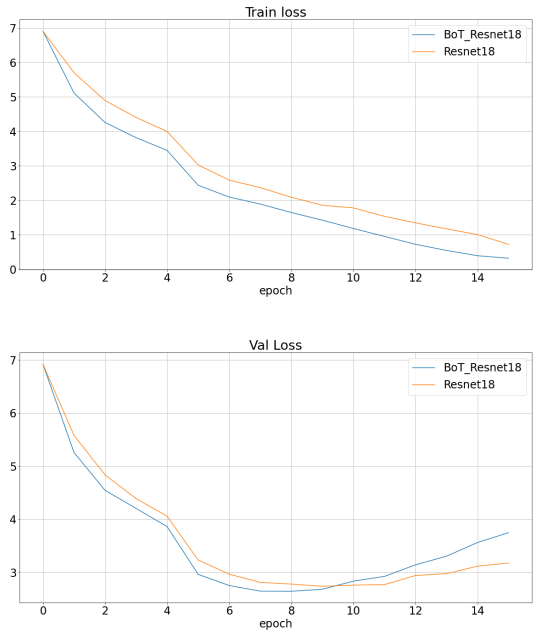

Figure 13. Resnet18 BoT Loss v/s Epochs

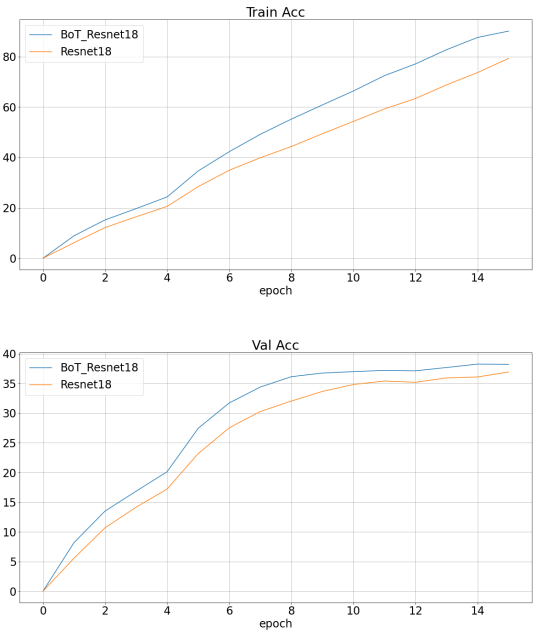

Figure 14. Resnet18 BoT Accuracy v/s Epochs

B. The SimpleCNN Architecture

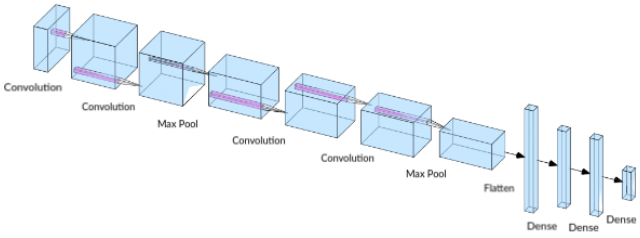

Figure 15. SimpleCNN architecture

Layer Type	Input Dimensions	Output Dimensions
Convolution + ReLU	(N, 3, 32, 32)	(N, 32, 32, 32)
Convolution + ReLU	(N, 32, 32, 32)	(N, 32, 32, 32)
MaxPool	(N, 32, 32, 32)	(N, 32, 16, 16)
Convolution + ReLU	(N, 32, 16, 16)	(N, 64, 16, 16)
Convolution + ReLU	(N, 64, 16, 16)	(N, 64, 16, 16)
MaxPool	(N, 64, 16, 16)	(N, 64, 8, 8)
Flatten	(N, 64, 8, 8)	(N, 4096)
Dense	(N, 4096)	(N, 512)
Dense	(N, 512)	(N, 512)
Dense + Softmax	(N, 512)	(N, 10)

Table 2. Architecture for SimpleCNN