

# Large batch sizes for Deep Learning

Mario René Surlemont  
University Vienna

Elnaz Javadi Farahzadi  
University Vienna

Alexander Stähle  
University Vienna

**Abstract**—A critical decision when using SGD variants is choosing the size of a batch. In the past, it has been shown that a generalization gap occurs when using large batch sizes rather than small batch sizes. This can be compensated by a larger learning rate, but the minima found usually remain sharper than with smaller batch sizes. Within this project we evaluate whether a formulation that regulates the sharpness of minimizers is suitable to compensate for the problems of large batch sizes. We do this by assessing the results of an empirical study where we train a neural network with and without this formulation comparing different batch sizes.

## I. INTRODUCTION

Deep Learning has become a popular approach to solve classification problems. The optimization problem to be solved in Deep Learning applications is typically of the form

$$\min_{\omega \in \mathbb{R}^d} f(\omega) := \frac{1}{M} \sum_{i=1}^M f_i(\omega) \quad (1)$$

where  $f_i$  is the loss function that calculates the loss for data point  $i \in \{1, 2, \dots, M\}$ ,  $M$  is the number of data points and  $\omega$  is the weight vector. For minimizing such functions one often uses Stochastic Gradient Descent (SGD) and its variants.  $f$  is minimized by iteratively taking steps of the form

$$x_{k+1} = x_k - \alpha_k \left( \frac{1}{|B_k|} \sum_{i=1}^{|B_k|} \nabla f_i(\omega_k) \right). \quad (2)$$

Here  $B_k \subset \{1, 2, \dots, M\}$  is a randomly selected batch of size  $|B_k|$  and  $\alpha_k$  is the learning rate in iteration step  $k$ . Usually one uses batches of size  $|B_k| \in \{32, 64, \dots, 512\}$ , where  $|B_k| \ll M$ .

Since the calculation of the gradient of the loss function cannot be parallelised well when using these rather small batches and can therefore increase the runtime considerably, the influence of large batch sizes on the learning process and the final result has been investigated in the past.

In general, it appears that using larger batches results in a larger generalization gap with one possible reasoning behind this being so-called "sharp minima" [1]. This can be partially compensated by an adjusted learning rate [2], but the sharp minima remain. As sharp minimiser usually generalise worse than flat minima, efforts have been made to develop a problem formulation that not only minimizes the given loss function, but simultaneously adjusts the sharpness of the minima [3]. This formulation is referred to as Sharpness-Aware Minimization (SAM).

Within this project, we will now evaluate to what extent SAM is suitable to enable training with large batches. We will train a very simple neural network for image recognition

once using SGD or Adam with small to large batch sizes and then apply SAM to the same batch sizes for comparison.

The minima will be examined for sharpness and we also compare the runtime and validation accuracies. We additionally evaluate how learning rate and batch size are related in the SGD scenario and check whether this relationship is also evident when SAM is used.

## II. DRAWBACKS OF LARGE BATCH SIZES

With SGD being one of the most prominent algorithms, it is used as a default method in numerous applications. While there are many different parameters to be tuned and perfected, choosing a batch size is one of the most important ones to optimize. It is often observed that smaller batch sizes lead to a better generalization, while larger batch sizes seem to degrade the quality of the model. In this section, we will look into some known causes and have a deeper look into sharpness of minima.

### A. Known issues

While small batch sizes, in theory, are not guaranteed to converge to better minima, they seem to outperform large batch sizes repeatedly when looking at the error on the test set [1]. Interestingly, their error on the training set seems to be similar to that of small batch sizes. This discrepancy between train and test error is also referred to as the generalization gap. Practitioners seem to run into this generalization gap, especially in large batch learning [4]. Nevertheless large batch sizes allow for parallelized computations in a manner that is way above the possibilities of parallelizing small batch training even considering efforts to overcome this shortcoming (e.g. [5]). Parallelization leads to huge speed-ups but as long as large batch sizes suffer larger generalization gaps this time-saving property does not always outweigh the advantages of smaller batches.

### B. Sharpness of Minima

Keskar et al make the interesting observation that models trained with large batch sizes tend to converge to sharp minima rather than flat minima, which is the case for small batch sizes [1]. This is a possible explanation for the above-mentioned generalization gap. The model might perform well on the training data but as the test data is generally not identical to the training data even small shifts in the optimal minima from training set to test set can lead to significant differences in the error. Small batch sizes however converge to flatter minima which is more robust to any shifts of the minimum, leading

to a better performance on the test set. It is therefore assumed to be desirable for a model to converge to flatter minima.

To encourage flatter minima Keskar et al attempt a number of techniques involving data augmentation and conservative training but while they do improve the generalization of models with large batch sizes the sharpness of the minima stays the same [1]. We therefore turn to a work of Foret et al in which they present an easy-to-use and effective way to penalize sharp minima [3]. In the following sections we will explain the idea behind their approach and evaluate if this idea remedies the problem of large batches.

### III. SAM

In [3] a problem formulation was presented in which an arbitrary loss function of the form (1) can be minimized simultaneously with its sharpness. The basic goal is to ensure that the value of the loss function does not change too much within a  $\epsilon$ -environment of the  $p$ -norm. The basic problem formulation here is

$$\min_{\omega} L_S^{SAM}(\omega) + \lambda \|\omega\|_2^2, \quad (3)$$

where

$$L_S^{SAM} := \max_{\|\epsilon\|_p \leq \rho} L_S(\omega + \epsilon). \quad (4)$$

$\rho$  is a hyperparameter and  $p$  indicates which norm is used. The authors show that usually  $p = 2$  is the optimal choice. Therefore, we also use this parameter in the further course. In order to minimize this problem, the authors propose an approximation of the gradient

$$\nabla_{\omega} L_S^{SAM}(\omega) + \lambda \|\omega\|_2^2 \approx \nabla_{\omega} L_S(\omega)|_{\omega+\hat{\epsilon}(\omega)}, \quad (5)$$

where

$$\hat{\epsilon}(\omega) := \arg \max_{\|\epsilon\|_p \leq \rho} L_S(\omega + \epsilon) = \rho \frac{\nabla_{\omega} L_S(\omega)}{\|\nabla_{\omega} L_S(\omega)\|_2}, \quad (6)$$

which can be computed by standard libraries like TensorFlow via autodifferentiation. This results in a problem formulation that can be minimized using SGD or Adam. The resulting SAM algorithm can be found in the appendix.

### IV. MODELS AND METHODS

This paragraph briefly describes our model and methods.

#### A. Model

The first layer of our convolutional neural network consists of a Conv2d layer, consisting of 32 filters with a kernel size of  $3 \times 3$  and a stride of 1. The padding is chosen in a way such that the output again has the same dimensions as the input. We use ReLu as the activation function. This is followed by a  $2 \times 2$  max pooling. Following this layer, we use a dropout layer with dropout rate  $p = 0.5$ . Then comes another Conv2D layer, this time with 64 filters, but otherwise the same properties as the first. This is followed by another  $2 \times 2$  max pooling and a dropout layer with  $p = 0.5$ . Finally, the tensor is flattened and a fully connected layer including biases follows. We use Softmax as the activation function, i.e., to

determine the probabilities of belonging to the various classes. This model is implemented in TensorFlow. We have taken the implementation of the SAM algorithm as a TensorFlow model from the Github repository [6].

#### B. Dataset

We train and evaluate our network using the FASHION-MNIST dataset [7].

#### C. Experiment setup

In order to minimise our loss function we use either SGD or Adam [8] with batch sizes  $|\mathcal{B}| \in \{32, 64, 128, 256, 512, 1024\}$ . We use SGD in all tests with heavy ball momentum and parameter "momentum" = 0.9. In the case of Adam, we use the default configuration of Keras. We do not use learning rate decay.

Our tests include two scenarios with respect to the learning rate. First, we use a fixed learning rate  $\alpha = 0.001$  across all batch sizes and in the other scenario we double the learning rate for a doubled batch size. We also start with a learning rate of  $\alpha_{32} = 0.001$  and double it for each doubled batch size, so that  $\alpha_{64} = 0.002$ ,  $\alpha_{128} = 0.004$ ,  $\alpha_{256} = 0.008$ ,  $\alpha_{512} = 0.016$ ,  $\alpha_{1024} = 0.032$ .

We use a maximum number of epochs  $E := 500$  for each test scenario. We also use an early stopping mechanism in which the learning process is terminated after 10 epochs with no increase in training accuracy.

We run all our experiments once with SAM and once without.

#### D. Quality measures

We assess the quality of the minimizers by the validation accuracy. To quantify the generalization gap, we use the ratio of our test accuracy to the validation accuracy. To evaluate the sharpness of a minimiser, we use a metric from [1], where the function value of the loss function is maximised within a constraint set around the minimiser. The exact metric is referred to as **Metric 2.1** in [1] and can be found there. Like the authors there, we use the L-BFGS-B algorithm with a maximum of 10 iterations to approximate the value of this metric. In contrast to the authors, we always use the parameter  $\epsilon = 10^{-2}$ . This measure is the most important metric for our tests.

### V. RESULTS

Using SGD with an increasing learning rate and SAM did in fact achieve a reduced sharpness for some batches. Also the generalisation gap could be reduced in these cases. This reduction was strongest for batch size 32, where the sharpness was reduced by a factor of nine. For batch size 64, the sharpness was also reduced by half. However, we could no longer see any improvement for batch size 128 and the sharpness even got worse for the larger batch sizes. The largest batch size (1024) is an outlier. Here the sharpness is lowest, but the validation accuracy is also reduced (Figure 1 and 2). Accordingly, this cannot generally be called a success.

However the outlier in the sharpness and the validation accuracy for batch size 1024 is interesting. We cannot provide an exact explanation for this. It is possible that the minimiser is not only in a local minimum of the function value, but also in a local minimum of the sharpness, from which it cannot break out. It would be interesting to observe in the future whether an increase in the hyperparameter rho can bring about a change here. If so, this would contribute to the hypothesis that there are also local sharpness minima that need to be taken into account.

Surprisingly, and in contradiction to previous publications, in our experiments, in the cases where we do not use SAM, we can find no difference in the sharpness of the minimisers at different batch sizes when the learning rate is adjusted (see Figure 2).

When using large batches, the runtime is reduced quite a bit. This is because the calculation of the gradients can be better parallelized with larger batches. This reduces the training time per epoch. The runtime when using SAM basically doubles, regardless of the size of the batches used. This is primarily because the gradient must be calculated twice for each iteration step when using SAM.

Our tests also showed that in no scenario could an improvement be achieved by SAM when using Adam. Neither the sharpness nor the validation accuracy could be improved no matter if we use a fixed learning rate or an adjusted (see Figure 4 and 5). SAM could not provide a benefit here.

In summary, we can say that (in our scenario) SAM, while maintaining validation accuracy, is only able to reduce sharpness at small batch sizes, i.e.  $|\mathcal{B}| \in \{32, 64\}$  and only when using SGD. Therefore, it cannot be deduced from our experiments that SAM brings an advantage with large batches. On the contrary, the runtime doubles and the final validation accuracy decreases.

Using Adam instead of SGD worsened the results even more as can be seen in Figure 4 and Figure 5 for Adam.

## VI. SUMMARY

Since large batch sizes often suffer from a larger generalization gap we tried to investigate the causes of this phenomena. As a result we focussed on achieving flatter minima as this seems to be a deviance between small and large batch sizes. We apply a solution in form of Sharpness-Aware-Minimization (SAM) to evaluate if SAM can be used to circumvent this issue. We found that although there are some cases where the sharpness is reduced, we could not achieve flatter minima in large batch training. The sharpness score rather increased than decreased. This result suggests that at least for our architecture and setting SAM does not provide an advantage since its runtime is roughly twice as large as without SAM.

## REFERENCES

- [1] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, “On large-batch training for deep learning: Generalization gap and sharp minima,” 2017.
- [2] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch SGD: Training ImageNet in 1 hour,” 2018.
- [3] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur, “Sharpness-aware minimization for efficiently improving generalization,” 2021.
- [4] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient Back-Prop,” in *Neural Networks: Tricks of the Trade: Second Edition*, ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer, 2012, pp. 9–48.
- [5] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, “Distributed Deep Learning Using Synchronous Stochastic Gradient Descent,” *arXiv:1602.06709 [cs]*, Feb. 2016.
- [6] S. Paul, “Sharpness-aware-minimization-tensorflow,” 2021. [Online]. Available: <https://github.com/sayakpaul/Sharpness-Aware-Minimization-TensorFlow>
- [7] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.
- [8] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

## APPENDIX

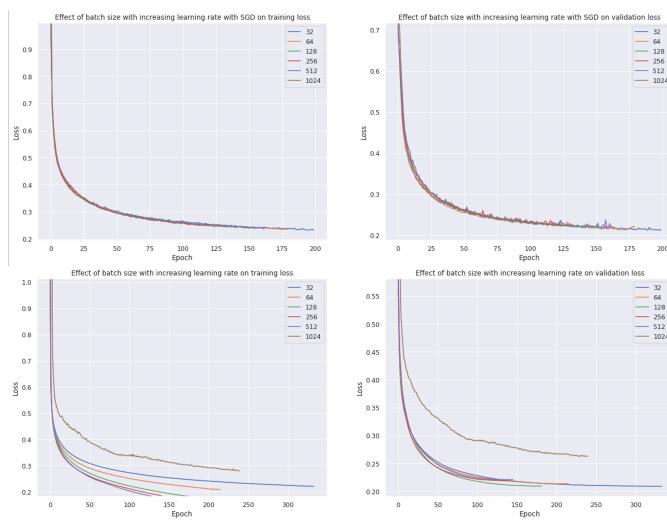


Fig. 1. Training and test loss for SGD with an increasing learning rate, once without (top) and once with (bottom) SAM.

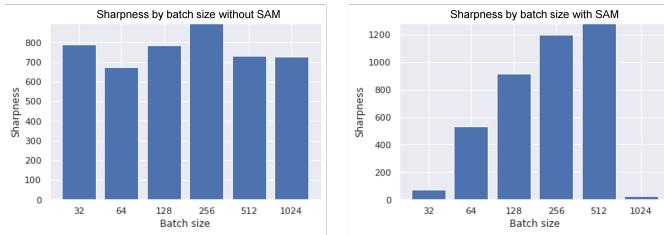


Fig. 2. Sharpness scores for SGD with an increasing learning rate, once without (left) and once with (right) SAM. Note the y-axis is not shared. One can see the decrease in sharpness for batch sizes 32 and 64 as well as the outlier in 1024.

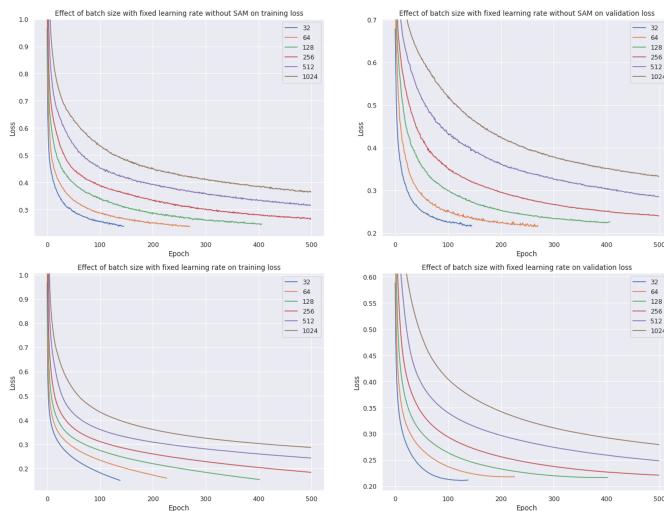


Fig. 6. Training and test loss for SGD without (top) and with (bottom) SAM with a fixed learning rate

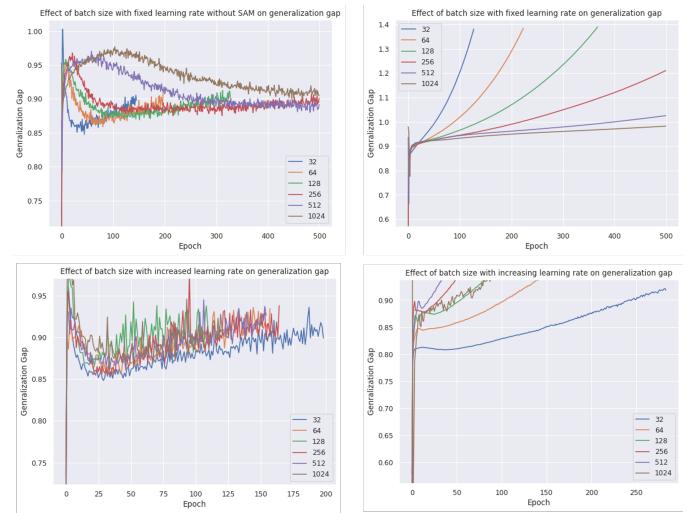


Fig. 3. Ratio of train error to test error for fixed (top) and increasing (bottom) learning rates once without (left) and once with (right) SAM.

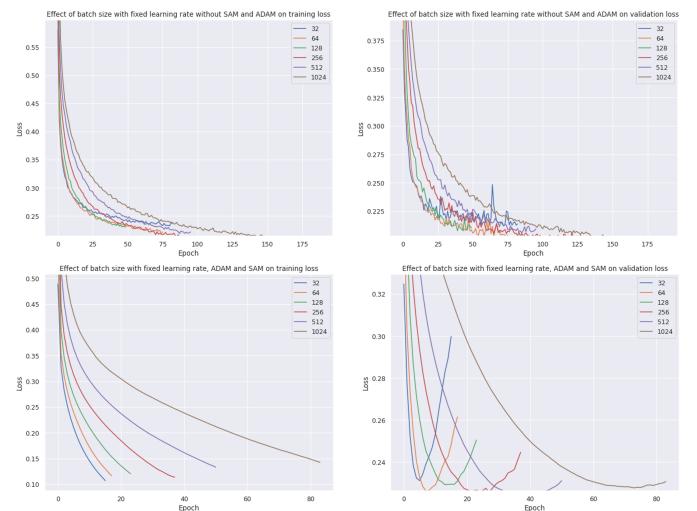


Fig. 4. Training and test loss for Adam without (top) and with (bottom) SAM. Clearly both approaches do not yield satisfactory results.

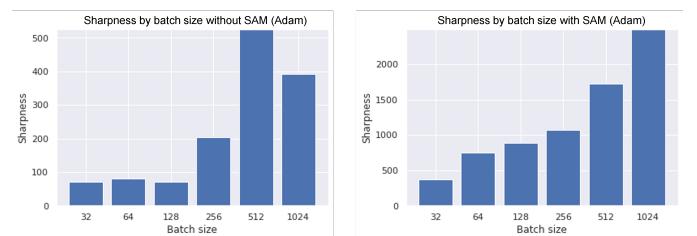


Fig. 5. Sharpness scores for Adam without (left) and with (right) SAM. Note the y-axis is not shared.



Fig. 7. Sharpness scores for SGD without (left) and with (right) SAM for a fixed learning rate. Note the y-axis is not shared.

---

**Algorithm 1** SAM-Algorithm

---

**Require:**  $X := (\cup_{i=1}^n \{x_i, y_i\})$ , Loss function  $l$ , Batch size  $b$ ,  
Optimizer  $o$ , Neighbourhood size  $\rho$

**Ensure:** Model  $M_S$  trained with SAM

Initialize weights  $\omega_0, t = 0$ ;

**while** *not converged* **do**

$\mathcal{B} \leftarrow \{(x_1, y_1), \dots, (x_b, y_b)\}$ ; ▷ a randomly chosen  
batch of size  $b$

Compute gradient  $L_\omega L_{\mathcal{B}}(\omega)$  of the batch's training loss;  
▷ Step 1

Compute  $\hat{\epsilon}(\omega)$  via (6); ▷ Step 2

Compute  $g := \nabla_\omega L_{\mathcal{B}}(\omega)|_{\omega+\hat{\epsilon}(\omega)}$ ; ▷ Step 3

Update weights  $\omega_{t+1}$  via optimizer  $o$  using  $g$ ;

$t \leftarrow t + 1$

**end while**

---