EPFL

# Impact of Optimisation Algorithm on Adversarial Training of a Neural Net for Image Recognition

Matthias Franke, Matthias Zeller, Baris Sevilmis

***Abstract*—This paper studied the impact the choice of optimisation algorithm has on the robustness of a convolutional neural network under exposure to adversarial images created by the Fast Gradient Sign Method, as well as its impact on the improvement through adversarial training. It found adversarial training to strengthen robustness not only against FGSM but also against Projected Gradient Descent attacks. The choice of algorithm made a difference in the effectiveness of adversarial training, with Adam performing best and Mini-batch worst, though the latter stayed competitive despite much cheaper training and hyper-tuning.**

## I. Introduction

Our primary goal is to study the impact of various optimisation algorithms on the robustness of neural networks against adversarial images. The "Fast Gradient Sign Method" (FGSM) on images gives an avenue to explore this feature. It uses slight perturbations that can elude human vision but can cause a naively trained model to misclassify the image with great confidence. The FGSM can additionally be used in a process called "adversarial training" to increase the robustness of the model against adversarial images. The increase in robustness is validated through attacking the robust models with a second attack called the $\mathbf{L}_\infty$-norm Projected Gradient Descent. The three chosen algorithms are the Mini-Batch Gradient Descent, the Adam algorithm and Nesterov's Accelerated Gradient Descent.

## II. Literature Review

The study of the robustness of machine learning-based classifiers to adversarial inputs has been ongoing since the last decade. One of the first and most effective methods of generation was found in 2015 [1] with the development of the "Fast Gradient Sign Method". The method is a simplified method of maximising the loss caused by misclassification [2]:

$$\boldsymbol{\delta} := \underset{\|\delta\| \leq \epsilon}{\arg\max}\ l(\mathbf{h}_\theta(\mathbf{x}+\boldsymbol{\delta}), y) \tag{1}$$

Here, $\mathbf{h}_\theta$ represents the neural network and $\mathbf{x}+\boldsymbol{\delta}$ its perturbed input.

This amounts to finding the perturbation $\boldsymbol{\delta}$ which will maximise the loss. Goodfellow, et al. demonstrated that not just linear networks, but in addition neural networks like convolutional networks were vulnerable to this. They also demonstrated that training said networks with adversarial examples could enhance the robustness of the models to FGSM attacks [1]. These so-called adversarial examples can be devastating, as demonstrated by Carlini through the usage of audio-based inputs to successfully fool speech-to-text classifiers [3]. Adversarial training is not flawless though and research has been conducted on avoiding so-called "catastrophic over-fitting" by FGSM to make it competitive with more complex, multi-step training methods [4].

## III. Optimisation Algorithms

### A. Theoretical Background

Our study includes optimization algorithms Mini-batch Gradient Descent, Nesterov and Adam, as mentioned in the introduction. Mini-batch trains the network on batches of the data-set and should thus produce an effective but simple model. Nesterov is the combination of a normal gradient step with a more aggressive step and has the theoretically fastest convergence rate. Finally, Adam is utilized, which includes both a more aggressive step and momentum.

### B. Software Implementation

We re-implemented the aforementioned optimizers, where each subclasses the `torch.optim.Optimizer` abstract class, and re-implemented the `step` function. This enables code

flexibility since our custom optimisers can be used just as PyTorch's built-in optimisers.

*1) Mini-batch Gradient Descent:* To ensure correctness of the implementation, we validated it against its typical implementation with Pytorch's SGD optimizer [5]. It works by selecting a batch of the data-set via Pytorch's data loaders and using the batch to train the parameters with a Gradient Descent step:

$$\mathbf{x}^{(t+1)} = \mathbf{x} - \lambda \nabla_{\text{batch}} f(\mathbf{x}^{(t)})$$

Additionally, the hyper-tuner is allowed to utilize a decreasing learning rate as well if so desired:

$$\mathbf{x}^{(t+1)} = \mathbf{x} - \frac{\lambda}{t+1} \nabla_{\text{batch}} f(\mathbf{x}^{(t)})$$

*2) Nesterov:* The implementation is based on [6]. We maintain a parameter vector $\mathbf{y}^{(t)}$ alongside the model parameter vector $\mathbf{x}^{(t)}$. Start with random $\mathbf{x}^{(0)}$ and set $\mathbf{y}^{(0)} := \mathbf{x}^{(0)}$. Next iterates are implemented as

$$\begin{aligned}\mathbf{y}^{(t+1)} &:= \mathbf{y}^{(t)} - \gamma \, \nabla f(\mathbf{x}^{(t)}) \\ \mathbf{x}^{(t+1)} &:= \mathbf{y}^{(t+1)} + \frac{t}{t+3} \left( \mathbf{y}^{(t+1)} - \mathbf{y}^{(t)} \right)\end{aligned}$$

*3) Adam:* Adam implementation is based on [7], where additional set of parameters have to be kept, namely moment 1 and 2 ($\mathbf{m}$ and $\mathbf{v}$ respectively). For sake of clarity, Adam is validated against the Pytorch's Adam optimizer as well. Adam is initialized with additional moment vectors of $\mathbf{m}^{(0)} = \mathbf{v}^{(0)} = \mathbf{0}$. Then, the gradients of the parameters for step $t$ are computed and used to calculate biased moments. $\beta_1$ and $\beta_2$ are hyper-parameters that are fine-tuned and utilized for the computation of moments.

$$\begin{aligned}\mathbf{g}^{(t)} &:= \nabla_{\mathbf{x}^{(t)}} f(\mathbf{x}^{(t)}) \\ \mathbf{m}^{(t)} &:= \beta_1 * \mathbf{m}^{(t-1)} + (1 - \beta_1) * \mathbf{g}^{(t)} \\ \mathbf{v}^{(t)} &:= \beta_2 * \mathbf{v}^{(t-1)} + (1 - \beta_2) * (\mathbf{g}^{(t)})^2\end{aligned}$$

Iterates are implemented in the following part by enabling $\mathbf{m}$ and $\mathbf{v}$ to be bias-corrected and later used in parameter update as explained in [7] in a detailed manner. Bias correction and its utilization in parameter update reduce the oscillations of gradient descent steps and aim steps more directly towards the global minimum.

$$\begin{aligned}\mathbf{m}_{corr}^{(t)} &:= \mathbf{m}^{(t)} / (1 - \beta_1^{(t)}) \\ \mathbf{v}_{corr}^{(t)} &:= \mathbf{v}^{(t)} / (1 - \beta_2^{(t)}) \\ \mathbf{x}^{(t)} &:= \mathbf{x}^{(t-1)} - \alpha * \mathbf{m}_{corr}^{(t)} / (\sqrt{\mathbf{v}_{corr}^{t}} + \gamma)\end{aligned}$$

Lastly, if provided weight decay parameter is positive, it is multiplied with the learning rate and weight parameters to be decreased from the weight parameters.

$$\mathbf{x}^{(t)} := \mathbf{x}^{(t)} - \alpha \cdot weight\_decay \cdot \mathbf{x}^{(t)}$$

## IV. Experiment Setup and Methodology

A classical convolutional neural network [8] is used for the MNIST data-set. Initially, the hyper-parameters of all three algorithms were tuned via grid search, starting with coarse grids and a simple train/validation split for performance evaluation, then iteratively refined the hyper-parameter grids and eventually used 5-fold cross-validation to take the variance of validation accuracy into account. The performances of the naive models are captured in Table I and robust models are captured in Table II and are notably comparable across all three algorithms, validating our competitive hyper-tuning.

Our pipeline is as follows. Naive models were trained with each optimizer and corresponding hyper-parameters, and attack it over a grid of $\epsilon$ values. This is repeated three times. Then, this whole procedure is repeated with robust models, created by adversarial training with FGSM. Finally, our stress models are experimented with a Projected Gradient Descent attack to ensure our adversarial training improved the robustness of the models.

## V. Adversarial Image Generation

In order for our results to be of use, recommendations set out by Carlini, et al. [9] were followed, such as using multiple attack approaches, attacking properly hyper-tuned models and clarifying the specific attack scenarios studied. The FGSM attack used is based on the original Goodfellow paper [1] and resulting implementations of it [10], [11]. Specifically, adversarial images are utilized that are clamped to be within the bounds of normal images [2]. Into Equation (1), we insert the following perturbation:

$$\boldsymbol{\delta} := \epsilon \cdot \text{sign}(\mathbf{g})$$

The tested epsilons were in the range of 0 to 0.5 so that the maximum perturbation stays within the input bounds. Resulting from the average-based normalisation of the input data, we relaxed the bounds on the adversarial images when using FGSM.

### A. Adversarial Training

To increase the robustness of our models, the loss due to misclassification needs to be decreased and which is

done by hyper-tuning followed by training a convolutional network with adversarial inputs generated by the FGSM method. An epsilon of 0.25 is utilized, which is the midpoint of the epsilon range and should thus provide a suitable training set. The resulting three models, one for each studied optimization algorithm, should thus perform better against the FGSM than their naive counterparts.

### *B. Projected Gradient Descent*

To ensure the robustness increase due to adversarial training is not due to the model over-fitting to possible FGSM attacks, we then further stress-test the robust models by using a second attack. $\mathbf{L}_\infty$-norm Projected Gradient Descent was chosen for the second attack. This thus projects onto the same norm-ball as FGSM, while having relaxed bounds as the FGSM had before. Thus, the perturbation here is $\boldsymbol{\delta} := \mathcal{P}(\boldsymbol{\delta}+\alpha\nabla_{\boldsymbol{\delta}} l(\mathbf{h}_\theta(\mathbf{x}+\boldsymbol{\delta}), y))$, where $\alpha$ denotes the step size at each iteration and $\mathcal{P}$ indicates a projection onto the $\mathbf{L}_\infty$-norm ball [2]. A step size of 0.01 is utilized to be sure that it is smaller than epsilon, but big enough to extract the solver from local minima. Furthermore, 40 iterations were used in the PGD attack, as per Kolter [2]. This amounts to iteratively making many "mini-FGSM" steps, resulting in more difficult adversarial images. It is a more nuanced attack and thus suitable to test the impact of adversarial training.

## VI. Results and Discussion

Figure 1 showcases that Adam, the strongest naive model, also performed best under adversarial training and with very little variance. Nesterov with its aggressive steps performs similarly until around an epsilon of 0.35 but does have much higher variance. Mini-batch, the simplest of the three algorithms, does gain the least from adversarial training but does so with very little variance and the fastest. Until an epsilon of 0.35, it stays competitive with the more aggressive Nesterov. A pattern common to all three algorithm is the sigmoid curve shape of the improved robustness due to adversarial training, highlighting diminishing returns at higher epsilons. When validating this with the PGD results, also found in Figure 1, we can see that Nesterov and Mini-batch perform as before, but Adam is no longer the clear winner. After an epsilon of 0.35, the two other algorithms saturated while Adam continues with the ultimately highest increase in robustness. This makes Adam the best performer of the three, though notably, Mini-batch was able to stay close against even the harsher PGD while being much cheaper to train.

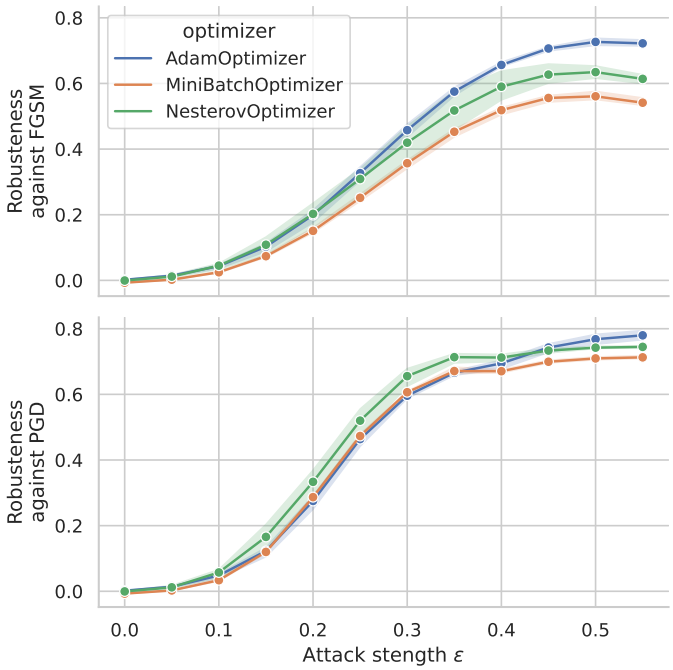


Fig. 1: Improved robustness of the model against FGSM (top) and PGD attack (bottom), computed as the difference in test accuracy between the attack on robust vs naive model. Dots represent mean over three runs, shaded area correspond to $\pm$ standard deviation.

## VII. Conclusion

All three algorithms were successfully implemented and trained by competitive convolutional networks on the MNIST data-set to have comparable performances. Further, the increase in robustness was investigated that such training brings, which found Adam to be the strongest optimizer for the naive models, to have the best performance, followed closely by Nesterov. Mini-batch performed the worst but remained competitive for smaller perturbations with a much cheaper tuning and training cost. We further validated this via PGD attacks. Therefore, the choice of the algorithm when studying adversarial training does have a non-negligible impact and this should be accounted for during the benchmarking of attacks and defences. Additionally, Mini-batch has shown itself to be a usable algorithm, which may aid researchers in prototyping more quickly due to its low cost of training.

## APPENDIX

| optimizer | mean | std |
|---|---|---|
| Adam | 0.989450 | 0.000634 |
| Mini-batch | 0.990374 | 0.000702 |
| Nesterov | 0.986399 | 0.001836 |

TABLE I: Naive model performance over three runs.

| optimizer | mean | std |
|---|---|---|
| Adam | 0.991297 | 0.000491 |
| Mini-batch | 0.983313 | 0.000261 |
| Nesterov | 0.985957 | 0.001050 |

TABLE II: Robust model performance over three runs.

## REFERENCES


[1] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–11, 2015.

[2] Z. Kolter and A. Madry, "Chapter 3 - Adversarial examples, solving the inner maximization," 2018. [Online]. Available: https://adversarial-ml-tutorial.org/adversarial_examples/

[3] N. Carlini and D. Wagner, "Audio adversarial examples: Targeted attacks on speech-to-text," *Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018*, pp. 1–7, 2018.

[4] M. Andriushchenko and N. Flammarion, "Understanding and Improving Fast Adversarial Training," *arXiv*, no. c, 2020.

[5] I. Sobh, "PyTorch: Gradient Descent, Stochastic Gradient Descent and Mini Batch Gradient Descent," 2019. [Online]. Available: https://www.linkedin.com/pulse/pytorch-gradient-descent-stochastic-mini-batch-code-sobh-phd

[6] W. Su, S. Boyd, and E. J. Candes, "A Differential Equation for Modeling Nesterov's Accelerated Gradient Method: Theory and Insights," *arXiv:1503.01243 [math, stat]*, Oct. 2015, arXiv: 1503.01243. [Online]. Available: http://arxiv.org/abs/1503.01243

[7] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[8] FloydHub, "MNIST ConvNet Model class," 2019. [Online]. Available: https://github.com/floydhub/mnist/blob/master/ConvNet.py

[9] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin, "On Evaluating Adversarial Robustness," *arXiv*, pp. 1–24, 2019.

[10] M. Jaggi, "CS-433 Lab 10 - Adversarial Robustness," 2020. [Online]. Available: https://github.com/epfml/ML_course/blob/master/labs/ex10/solution/ex10.ipynb

[11] N. Inkawhich, "Adversarial Example Generationtle," 2021. [Online]. Available: https://pytorch.org/tutorials/beginner/fgsm_tutorial.html