Towards Efficient Methods for Training Robust Deep Neural Networks

Sanjit Bhat
MIT PRIMES
sanjit.bhat@gmail.com

Dimitris Tsipras
MIT
tsipras@mit.edu

Aleksander Mądry
MIT
madry@mit.edu

November 14, 2018

Abstract

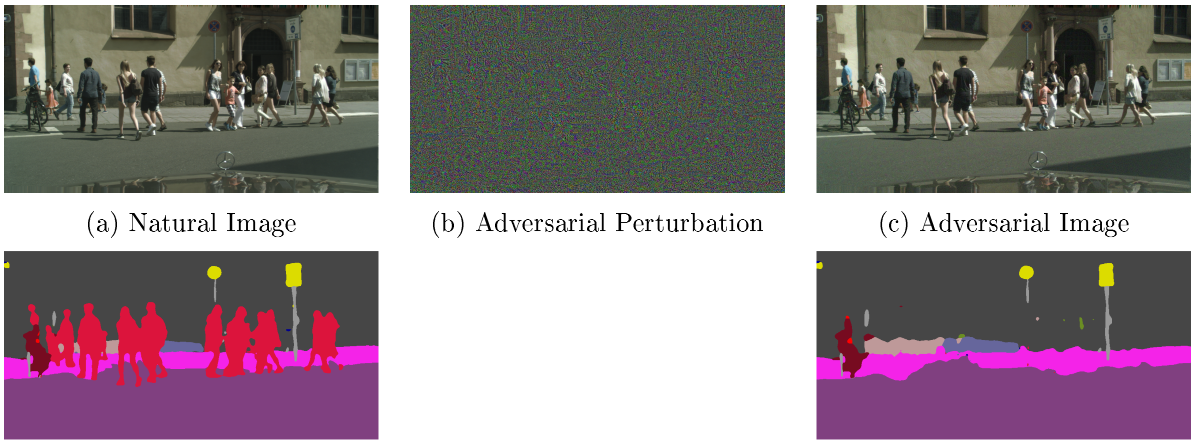
In recent years, several works have shown that neural networks are vulnerable to adversarial examples, i.e., specially crafted inputs that look visually similar to humans yet cause neural networks to misclassify. Adversarial training creates robust models by augmenting the dataset with adversarially perturbed inputs. However, since these inputs require significant computation time to create, adversarial training can be impractical in real-world applications. In this work, we explore the strength of inputs needed for successful Adversarial Training and asynchronous parallelization approaches. Taken together, these two techniques enable comparable robustness on the MNIST dataset to prior art with a $26\times$ reduction in training times from 4 hours to just 9 minutes. Overall, our work moves a step closer to the efficient training of robust deep neural networks.

1 Introduction

In recent years, Deep Learning (DL) systems have achieved surprising success in several domains such as Computer Vision and Natural Language Processing [13, 16]. In contrast to traditional rule-based or symbolic AI systems, DL systems learn input and output pairings automatically through labeled data. This provides them independence from human programming, allowing them to act in complex scenarios without being explicitly programmed to. In addition, since their accuracies are generally a function of the amount of labeled training data used, they can be improved simply by adding more data, which is not the case for traditional rule-based systems.

Combined, these two properties have allowed DL systems to achieve super-human performance in several settings such as Computer Vision and complex game playing. In Computer Vision, large-scale datasets such as ImageNet [22] spurred a race to construct better Computer Vision DL models. This led to efforts such as the widely-used VGG [24] and ResNet [11] models that achieved super-human classification performance. In game playing, Google DeepMind's AlphaGo system used DL to act and perceive in one of the most complex games in existence, Go [23]. With a combination of knowledge from prior games and self-play, AlphaGo defeated the foremost human Go player, Lee Sedol, in 2016 [4].

Owing to its success in several fields, DL is increasingly being adopted in security-critical applications. For example, Tesla and Waymo use Computer Vision systems to enable their self-driving cars to perceive their surroundings [25, 10]. Instead of classifying objects, these systems perform



(a) Natural Image (b) Adversarial Perturbation (c) Adversarial Image

(d) Natural Prediction (e) Adversarial Prediction

Figure 1: Adversarial examples for self-driving cars [7]. Figure 1a shows a natural image taken from a car camera that is later fed into an image segmentation model. The output of the model's predictions are shown in Figure 1d, with the car recognizing pedestrians crossing the street and most likely taking the necessary precautions. In the adversarial case, however, a small amount of seemingly random noise shown in Figure 1b is added to the natural image. Even though the resulting image in Figure 1c looks visually indistinguishable to humans, it causes the segmentation model to fail to recognize the humans, as shown in Figure 1e. Thus, the car would act as if no pedestrians were crossing and fail to apply the brakes, resulting in catastrophic consequences.

segmentation, distilling an image into its various classes (e.g., road signs, people, trees, and buildings) to aid the driving system in making critical decisions. Similarly, Apple's new Iphone X uses Computer Vision to power its new FaceID system [27]. The phone extracts facial features, measures similarity to its owner's face, and grants entrance to contacts, passwords, emails, and bank accounts if the two faces match. In these security-critical applications where one's life and privacy are at stake, it is paramount that we ensure the proper functioning of DL systems. As such, one natural question that arises is whether these systems are actually ready for real-world deployment.

In several recent works [8, 19, 28], researchers have discovered *adversarial examples*, inputs that look visually similar to their natural counterparts yet cause DL models to misclassify them. For example, with self-driving cars, the adversary crafts a perturbation such that to humans both the original and adversarial images look the same (see Figure 1). However, when fed into the image segmentation model, the adversarial example causes the car to not recognize the humans.

Clearly then, adversarial examples bring into question the reliability of our state-of-the-art DL systems. Apart from the security implications of hackers being able to wreck havoc on large-scale, security-critical systems, these flaws raise a couple more fundamental issues:

1. **Reliability.** In addition to DL systems malfunctioning on artificially crafted perturbations, DL models also misclassify naturally occurring phenomena that humans would otherwise be invariant to. Examples of such phenomena in self-driving cars include rain, sleet, and snow [26]. These vulnerabilities could reduce the reliability of DL classifiers in the real-world.

2. **Intelligence.** From a more fundamental aspect, the goal of artificial intelligence research has and always will be to create general-purpose systems that solve a wide variety of problems.

Oftentimes, we use the human brain as an example of such a system. However, given that most adversarial perturbations are quasi-imperceptible, a human brain would never get tricked, whereas a DL system does. Thus, the existence of adversarial examples points to key flaw in current DL systems: these systems have not learned the same meaningful abstractions that we do.

Given the implications of having large-scale models susceptible to adversaries, recently, several works have proposed methods for training *robust* models, models that produce the same classifications even for adversarial inputs. While several such "defenses" against adversarial attacks have been proposed, one method called Adversarial Training has shown to be robust even against counter-attacks [1]. While Adversarial Training provides resistance to adversarial attacks, it adds significant amounts of computational overhead compared to regular training. This makes it harder for robust DL models to be used in the real-world, especially in applications that require frequent model re-training. Consequently, in this work, we ask the following question:

Can robust DL models be trained efficiently?

Our contributions. In this work, we study the above question via two orthogonal ideas: understanding attacker power and asynchronous parallelization. In the former, we investigate the effects of using weak and strong attackers and use our results to find a balance between attacker strength and computational overhead. In the latter, we analyze a phenomenon called staleness that results from multi-GPU asynchronous parallelization. Finally, we bring our findings together to improve the efficiency of Adversarial Training, reducing the state-of-the-art robust training time on the MNIST dataset 26× from 4 hours to 9 minutes.

2 Background

Regular neural network training. A standard neural network is a type of Machine Learning model inspired by the human brain. It is composed of several layers and automatically learns input and output pairings for a certain data distribution $\widehat{\mathcal{D}}$ by optimizing its weight parameters, θ. Specifically, to perform this optimization it feeds input, predicted output, and actual output into a loss function $\mathcal{L}$ that provides a metric of how well the network performs. Since lower loss refers to a better prediction, the training procedure for a regular neural network corresponds to solving the following optimization problem:

$$\min_{\theta}\left[\mathop{\mathbb{E}}_{(x,y)\sim\widehat{\mathcal{D}}}\mathcal{L}(x,y,\theta)\right]. \tag{1}$$

In practice, we use Stochastic Gradient Descent to minimize loss and the Backpropagation Algorithm [21] to compute gradients.

Adversarial examples. While the trainer aims to minimize loss, conversely, the adversary tries to maximize it to reduce classification accuracy, the proportion of inputs identified as the correct class. If unconstrained, the adversary could maximize loss simply by transforming a data point from one class into a data point from a different class. However, doing so would look drastically different to humans. As such, we impose a constraint on perturbations wherein the resulting adversarial example must look "visually similar" to its original (i.e., such that a human would identify it as

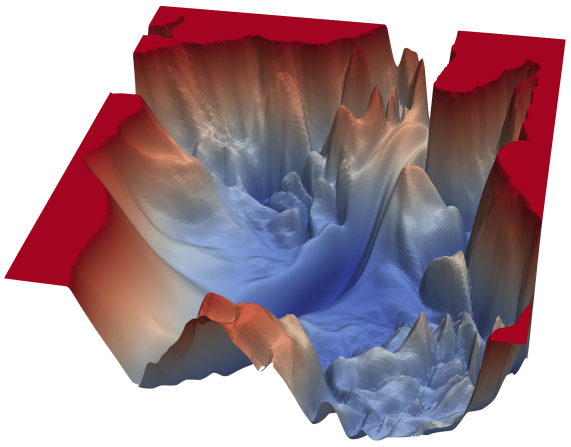


Figure 2: An actual neural network loss landscape from Li et al. [18]. Although the x and y axes represent changes in the network parameters rather than the network inputs, the visualization still leads to an important observation: neural networks have highly complex, non-concave loss landscapes. If a single-step adversary such as FGSM [8] tries to approximate a far-away local maxima with locally linear information, it might not arrive at a reasonable solution.

being part of the same class). Letting the set of possible visually perturbations be $\mathcal{S}$, the adversary tries to achieve the following:

$$\max_{\delta \in \mathcal{S}} \mathcal{L}(x + \delta, y, \theta). \tag{2}$$

Set of allowed perturbations. The set of human visual invariants is broad and hard to define mathematically, including rain and snow, rotations and translations [6], and several other distortions. Since optimization algorithms need a precise definition of similarity to form their constraint, we use the ℓ_∞ metric, consistent with prior work [8, 19]. Specifically, every perturbation δ with components $x_1, x_2, \ldots, x_n$ must adhere to the following constraint:

$$\max_i |x_i| \leq \varepsilon. \tag{3}$$

Using this constraint, we precisely define the set $\mathcal{S}$ of all allowable perturbations, which forms an ε-ball around the natural point.

Computing adversarial examples. In prior work, researchers have proposed algorithms such as the Fast Gradient Sign Method (FGSM) [8] to optimize Equation 2. FGSM uses the gradient of the loss with respect to the input to form a local linear approximation of the loss for any change in input. It then takes a step in the direction of the gradient to maximize the predicted loss. 'Fast' in FGSM refers to the fact that the algorithm only takes one step of size ε, computing gradients once and hitting the edge of the ℓ_∞ ε-ball immediately:

$$x + \varepsilon \operatorname{sgn}(\nabla_x \mathcal{L}(x, y, \theta)). \tag{4}$$

However, while FGSM's one-step greedy procedure finds loss maxima in well-behaved loss landscapes, it fails to find suitable maximizers in more complex landscapes [19]. Figure 2 shows this phenomenon graphically, wherein a one-step adversary falsely assumes that the steepest hill closest to it will yield the best ascent direction. In reality, this hill may just correspond to a low local maxima, leading the adversary askew.

While FGSM provides a fast method for finding adversarial examples, it relies too heavily on the local linearity assumption. In reality, neural networks have complex, non-linear loss landscapes (see Figure 2), which makes it hard to predict the loss of a point far out in the ε-ball [19]. As such, a natural extension to FGSM known as Projected Gradient Descent (PGD) [14, 19] uses κ smaller steps of size α. At each step, PGD re-computes gradients relative to its current solution and projects back into the ε-ball. Intuitively, by re-measuring the local linearity at several checkpoints along its optimization trajectory, PGD relies less on any one approximation. This leads to better adversarial examples than those produced by FGSM [19]:

$$x^{t+1} = \Pi_{x+\mathcal{S}}(x^t + \alpha \operatorname{sgn}(\nabla_{x^t} \mathcal{L}(x^t, y, \theta))). \tag{5}$$

In Equation 5, $\Pi_{x+\mathcal{S}}$ refers to projection back into the ε-ball surrounding the natural point. By projecting at each timestep, we ensure we always stay within the constraints. Specifically, for an adversarial example v with components $v_1, v_2, \ldots, v_n$ and natural example x with components $x_1, x_2, \ldots, x_n$, projection ensures that v_i lies between $x_i - \varepsilon$ and $x_i + \varepsilon$.

Training a robust deep neural network (DNN). When competing against an opponent, it is important to understand how that opponent plays. Similarly, training a robust DNN (i.e., a network unaffected by input changes within a certain constraint) can be viewed within this context. Throughout the training procedure, the adversary tries to exploit the DNN's flawed feature mappings. To defeat the adversary, the DNN must then understand how it's being exploited.

One natural way of achieving this is called Adversarial Training. Instead of training on natural examples, the DNN simulates the adversary, computes adversarial examples, and trains on them. It then updates its weight parameters, θ, to reflect its new knowledge of the adversary and its own prior flaws. Since the adversary continually points out new flaws to the network, the DNN needs several timesteps of Adversarial Training to achieve robustness.

Combining Equation 1 for natural DNN training with Equation 2 for the adversary's objective, we arrive at the following min-max saddle-point formulation for Adversarial Training:

$$\min_{\theta} \rho(\theta), \quad \text{where} \quad \rho(\theta) = \mathop{\mathbb{E}}_{(x,y)\sim\widehat{\mathcal{D}}} \left[\max_{\delta \in \mathcal{S}} \mathcal{L}(x+\delta, y; \theta) \right]. \tag{6}$$

According to Danskin's Theorem [3, 2], the gradients of the loss, $\nabla_\theta \mathcal{L}(x+\delta; y; \theta)$, are equivalent to the gradients of the max loss, $\nabla_\theta \max_{\delta \in \mathcal{S}} \mathcal{L}$ if δ is a global maximum. However, as the loss function is non-concave (see Figure 2), the solutions yielded by PGD are actually local maxima, not global maximum. Nonetheless, using a sufficiently large number of steps usually leads to strong-enough local maxima to solve the min-max problem.

Thus, we use the descent direction provided by $\nabla_\theta \mathcal{L}(x+\delta; y; \theta)$ to minimize the max loss and solve the saddle point problem. For a strong-enough inner adversary, this procedure yields universally robust DNNs (i.e., DNNs robust to *any* first-order adversary constrained by the ℓ_∞ norm) [19, 1].

3 Towards Efficient Robust Training

As the universal robustness described in Section 2 is highly desirable, we improve upon Adversarial Training in the following sections by addressing its efficiency. While PGD improves upon FGSM with a multi-step optimization procedure, it also introduces significant computation overhead. For each added step, the algorithm must compute one additional set of gradients using the Backpropagation algorithm, a slow process. For example, with the current implementation of PGD in the Mądry Lab MNIST Challenge, the added benefit of increased strength maximizers comes at the cost of a roughly 40× longer training procedure [15].

Clearly, the number of PGD steps used for training robust DNNs plays an important role in its final robustness. In Section 4, we study the broad impact of varying the attacker power and use our findings to drastically reduce the number of steps needed for Adversarial Training.

Another downside of Adversarial Training is its synchronous nature. With each iteration of the min-max problem in Equation 6, the outer minimization process waits for the inner maximization to finish, and vice-versa. This increases the overall time to execute Adversarial Training. In addition, we cannot use multiple GPUs to speed up computation of a single adversarial batch since PGD is an inherently sequential process.

Instead, in Section 5 we develop an asynchronous implementation of PGD wherein the adversary and trainer work independently. This allows for multiple GPUs to drastically speed up adversarial example computation. However, it also comes at the cost of *staleness*, or the adversarial examples pointing out flaws from a previous network timestep. We study staleness in-depth through detailed experiments, and we arrive at a final asynchronous implementation that gains a roughly-linear speedup as more GPUs become available.

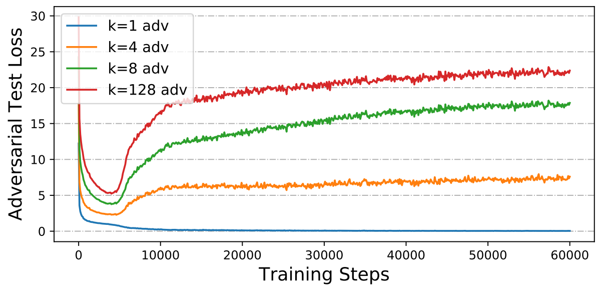
Finally, combining our studies of attacker power and asynchrony, we present our results for efficient Adversarial Training on the MNIST dataset [17]. Using both techniques, we achieve a 26× reduction in training time from 4 hours to 9 minutes.

4 Understanding Attacker Power

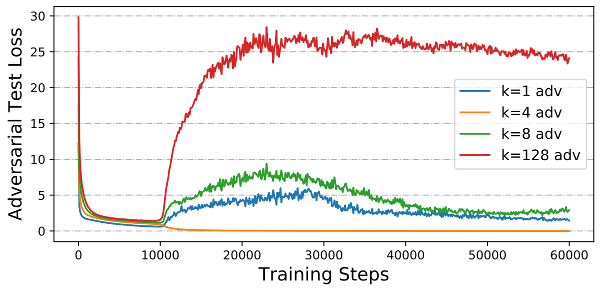
To solve the min-max optimization problem in Equation 6 and train a robust model, Danskin's Theorem tells us that we should use global maximum to calculate gradients for descent directions. However, finding the global maximum in a non-concave, highly complex loss landscape is nearly impossible and would take far too long to compute. As such, researchers have used first-order methods such FGSM and PGD to compute local maxima, sacrificing quality for speed of computation [8, 19]. In this section, we investigate the trade-off between quality and speed for PGD-like adversaries.

The quality of adversarial examples (e.g., their ability to maximize loss) is a function of the number of steps used to create them. Given a constant epsilon, ε, we split the overall trajectory into κ equally sized steps, recalculating gradients and a descent direction at each step. Since PGD uses first-order gradients to form a local linear approximation at each step, a larger number of steps results in a smaller step size and less reliance on the approximation. Given how actual neural network landscapes are complex and non-linear, using more steps should therefore yield better quality adversarial examples, helping both the inner maximization, outer minimization, and overall robustness.

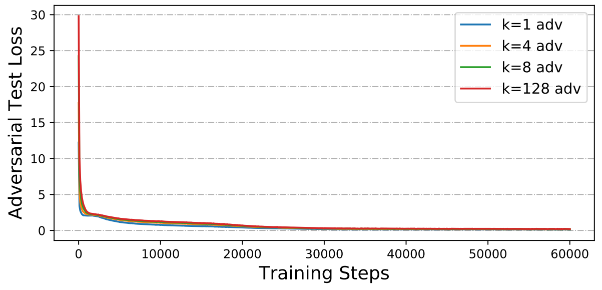
While using more steps results in better maxima, it also comes with significant computational costs. With normal training, each iteration of the outer minimization problem would only take one pass of the relatively slow Backpropagation algorithm. Adversarial Training, however, requires additional passes for each additional step, a linear increase in training time. Moreover, PGD's iterative optimization process cannot be parallelized since each step builds off of the adversarial example from the previous step. Thus, the minimization process must wait for the lengthy maximization



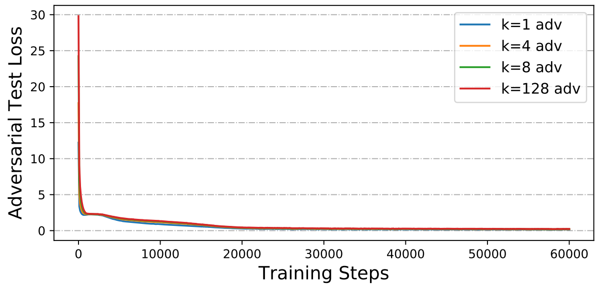
(a) Trained with FGSM



(b) Trained with $\kappa = 4$ PGD



(c) Trained with $\kappa = 8$ PGD



(d) Trained with $\kappa = 128$ PGD

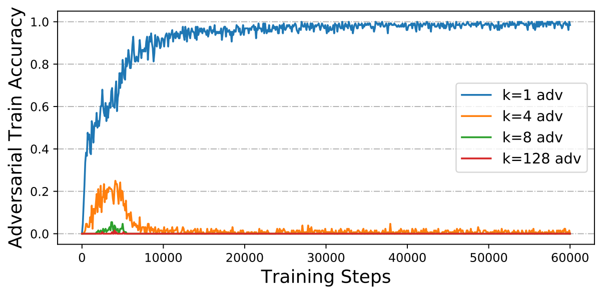
Figure 3: Figures 3a, 3b, 3c, and 3d show adversarial loss on the testing set for networks trained with a κ = 1, 4, 8, and 128 PGD adversary, respectively. The four lines within each figure display the loss graph for a κ = 1, 4, 8, and 128 PGD adversary attacking that respective network. Compared to the networks trained with $\kappa = 1$ and 4 PGD adversaries, the $\kappa = 8$ and 128 networks exhibit robustness against a wide range of attackers.

to finish before continuing its next iteration. With the trade-off between strength of adversarial examples and training time, the following question arises:

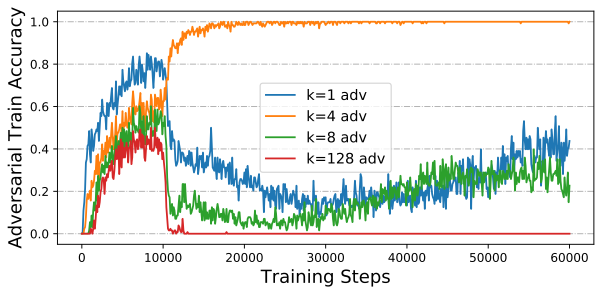
Is there a balance between attacker strength and training time?

Adversarial Training with a weak attack. Upon first glance, low-power and high-power PGD adversaries all perform the same for a network trained with a powerful adversary such as PGD with $\kappa = 8$ or 128 (see Figures 3c and 3d). However, their similar loss values result from the network's strong robustness: every first-order attack will have low loss because the network has become robust to all of them. Conversely, Figures 3a and 3b show the intriguing difference between adversarial power when weak attacks are used to train the model. Not only do higher power adversaries yield higher loss, but also the adversary used to train the network has a loss near 0 for a significant portion of training. Is this because the network has simply gained robustness against the weak adversaries and not the stronger ones, or is something more interesting going on?

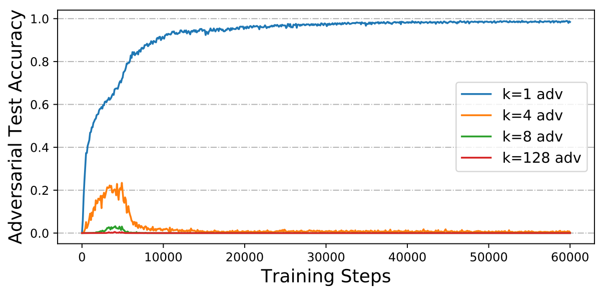
Our evidence points to the latter. First, observe that for the networks trained with $\kappa = 1$ and 4 PGD, adversarial accuracy on the training set for that specific attacker goes to nearly 100% after a small number of training steps (see Figures 4a and 4b). Since natural accuracy for these MNIST models hovers around 98% and the adversarial setting is harder than its natural counterpart, these networks likely memorized the distribution of possible weak adversarial examples in the training set to achieve 100% adversarial accuracy. Indeed, recent work has shown that even state-of-the-art



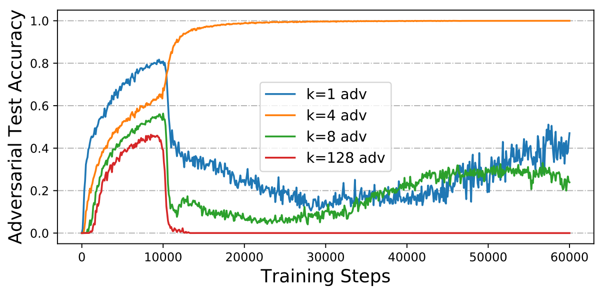
(a) Train Accuracy, FGSM Training



(b) Train Accuracy, $\kappa = 4$ PGD Training



(c) Test Accuracy, FGSM Training



(d) Test Accuracy, $\kappa = 4$ PGD Training

Figure 4: Figures 4a and 4b show adversarial accuracy on the training set for networks trained with a $\kappa = 1$ and 4 PGD adversary, respectively. Figures 4c and 4d show adversarial accuracy on the testing set for networks trained with a $\kappa = 1$ and 4 PGD adversary, respectively. The four lines within each figure display the accuracies for a $\kappa = 1$, 4, 8, and 128 PGD adversary attacking that respective network. The 100% train accuracy against the $\kappa = 1$ adversary in Figure 4a and the $\kappa = 4$ adversary in Figure 4b point towards these networks memorizing their attacker training distribution. The corresponding near-100% test accuracy in Figures 4c and 4d point to a much more intriguing phenomenon: gradient shattering.

CNN's are capable of memorizing completely random input data [29]. Since the adversarial examples are not fully random and instead rely on the underlying natural training images, the network could certainly have memorized them.

However, more intriguing is the near 100% adversarial accuracy on the *testing* set (see Figures 4c and 4d). Firstly, adversarial accuracy for a higher κ, stronger adversary should never be higher than that of a lower κ, but it is in Figure 4d. In addition, the network has never before seen the natural testing images, yet it still manages near-perfect accuracy on their adversarial counterparts for a same-strength attack. Taken together, both observations point to a phenomena called *gradient shattering*, introduced by Papernot et al. [20] and Athalye et al. [1]. When one trains a network with a weak adversary, the model learns to augment its loss landscape such that any adversary with the same strength cannot generate meaningful gradients. Thus, even though the testing set introduces new natural images, the adversary cannot generate strong adversarial examples due to the network's broken loss landscape.

The gradient shattering phenomena can be further evidenced in Table 1. All of the above experiments have been in the white-box setting, where the adversary has access to the model's parameters and directly uses them to create examples. In this black-box setting, the adversary is not provided

Table 1: Adversarial accuracy on the testing set for various attacks. Each value represents a testing set created with a certain κ PGD adversary on a certain κ source network and evaluated on a certain κ target network. The diagonal represents white-box attacks whereas the off-diagonal represents black-box attacks. The highlighted values show evidence of gradient shattering: weak white-box attacks on a weak network are not effective, but the same strength attack transferred over from a different network is effective.

Target κ	Source $\kappa = 1$	Source $\kappa = 4$	Source $\kappa = 8$	Source $\kappa = 128$	Adv κ
1	**98.55%**	48.63%	27.16%	28.36%	1
4	57.34%	**99.98%**	54.18%	60.31%	4
8	96.40%	97.27%	93.88%	94.26%	8
128	96.28%	96.69%	93.92%	91.70%	128

access to the model, needing to use a surrogate ‘source’ network to transfer adversarial examples to a ‘target’ network. For a black-box attack on the networks trained with $\kappa = 1$ and 4, notice how an adversary using the exact same power PGD manages to break the networks even though that same adversary cannot break the networks in a white-box setting. This indicates that networks trained with weak adversaries shatter their gradients to achieve robustness. Unfortunately, this method does not provide robustness to higher strength adversaries or even same-strength adversaries transferred from a surrogate network.

Towards a balance point. In our experiments, we sweep from $\kappa = 1$ to $\kappa = 128$ in powers of 2 to understand whether certain strength attackers yield favorable balance points between attacker strength and computational overhead. In the white-box setting, observe that networks trained with a $\kappa = 8$ and a $\kappa = 128$ adversary both yield strong robustness against the full spectrum of first-order attackers (see Figures 3c and 3d). This is unexpected since the $\kappa = 8$ adversary never saw such a strong adversary during its training procedure. One likely explanation is that above $\kappa = 4$, networks gain full, universal robustness on the MNIST dataset. Thus, even attackers with greater numbers of steps cannot produce strong adversarial examples.

Furthermore, moving from $\kappa = 4$ to $\kappa = 8$ and $\kappa = 128$, we see no evidence of gradient shattering in either of the latter two networks (see Figure 1). For example, for the target network trained with $\kappa = 8$ PGD, computing a $\kappa = 8$ example on the $\kappa = 128$ source network yields marginal differences in adversarial accuracy.

While networks trained with $\kappa \leq 4$ lead to faster training times, they lack universal robustness and exhibit gradient shattering. Conversely, networks trained with a $\kappa \geq 8$ offer these desirable traits while remaining otherwise indistinguishable (e.g., all such networks achieved around 98% natural accuracy and 89–91% adversarial accuracy on the test set). As such, we train our final network with $\kappa = 8$ compared to the $\kappa = 40$ in the Mądry Lab MNIST Challenge code to balance robustness with training time. This technique provides a roughly 5$\times$ improvement in training time.

5 Asynchronous Parallelization

In Section 4, we studied trade-offs with attacker strength to arrive at a 5$\times$ reduction in training time. Here, we look at an orthogonal idea involving asynchronous parallelization to arrive at even greater speedups.

Traditionally, researchers solved Equation 6 in a synchronous manner, simulating the adversary in the inner loop, feeding the adversarial examples to the model, and using the model to create adversarial examples. While this makes the adversarial examples used by the network current— they are computed for the most recent timestep—it also introduces significant delays; the minimization process must wait for the much slower maximization process before continuing. One solution is to trivially make them asynchronous, with a separate trainer GPU for minimizing, a separate worker GPU for maximizing, and a queue inbetween to hold finished adversarial examples. However, since the maximization takes nearly 8$\times$ as long as the minimization with $\kappa = 8$, the trainer GPU would still spend most of its time waiting. This raises the following question:

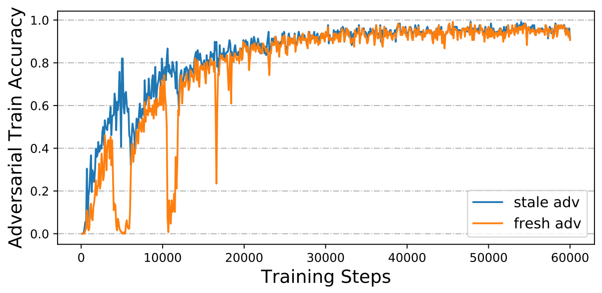
Can we parallelize the inner maximization to make it faster?

Unfortunately, due to the inherently serial nature of PGD wherein each step builds off of the previous step (see Equation 5), we cannot achieve parallelism by applying multiple GPUs to a single, small batch. Instead, we achieve parallelism by using multiple worker GPUs to simultaneously create adversarial batches and store them in a queue once completed. Later, the trainer GPU picks up batches from the queue, calculates gradients from them, and uses the gradients to minimize loss.

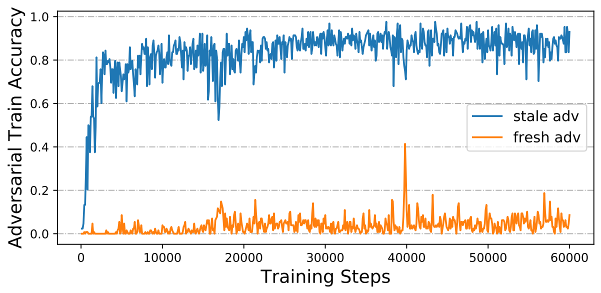
To demonstrate asynchronous parallelization, consider a scenario with one trainer GPU and two worker GPUs. The workers start computing an adversarial batch using parameters θ_t from the network at time t. Since both workers perform the same optimization, they finish at around the same time and add their batches to the queue. Assuming the trainer GPU was previously unoccupied, it takes the first batch and uses it to calculate new parameters, θ_{t+1}, that minimize the adversarial loss. Now, when the trainer takes the second sample from the queue, there exists a discrepancy between timesteps. The adversarial batch was computed for θ_t while the network now has parameters θ_{t+1}. This causes the adversarial batch to point out slightly older network flaws, reducing its effectiveness. We call this phenomenon *staleness* and study its effects in-depth to arrive at a final asynchronous parallelization implementation.

Simulating staleness. Real-world multi-GPU parallelization has several tunable parameters, making it hard to evaluate staleness. Instead, we use a single GPU with a queue of size s to simulate an exact staleness s that persists throughout network training. Specifically, at the start of training, we fill the queue to its maximum capacity and iterate through the normal, synchronous training procedure. Since each example is processed in one timestep, if the example at the back of the queue enters at timestep t, it will exit at timestep $t + s$. The difference between entrance and exit timesteps, s, matches the difference in the network parameters used to create the adversarial example. Since every example has this exact timestep difference s, we have a constant staleness s for every adversarial batch.

The downsides of high staleness. Just like how models trained with weak attacks in Section 4 overfit to their specific attack, high staleness networks overfit to stale data. Within the context of staleness, we use the terms “fresh” and “stale” to refer to accuracy on adversarial examples generated at the current timestep and at a previous timestep. Consider Figures 5a and 5b, which show fresh



(a) Trained with $s = 32$



(b) Trained with $s = 64$

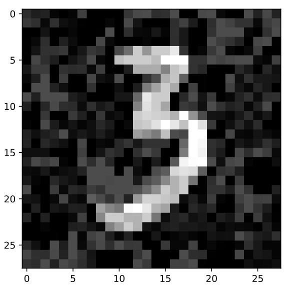
Figure 5: Figures 5a and 5b show adversarial accuracy on the training set for networks trained with $s = 32$ and 64 staleness, respectively. The two lines within each figure represent the accuracy on fresh and stale adversarial examples. Even though each network was trained on stale examples, we evaluated fresh accuracy by generating fresh examples at the current network timestep. The difference between fresh and stale training accuracy in Figure 5b indicates that $s = 64$ networks memorize the stale data distribution, lacking robustness to fresh adversarial examples. In contrast, the $s = 32$ network achieves high accuracy on both fresh and stale examples.

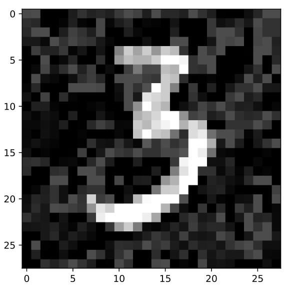
and stale adversarial accuracy for networks trained with $s = 32$ and $s = 64$. While fresh accuracy tracks stale accuracy for $s = 32$, it is much lower than stale accuracy for $s = 64$. This indicates that high staleness networks memorize the stale data distribution, failing to become robust to fresh adversarial examples. In the following discussion, we hypothesize on what happens in the network training procedure to cause these issues.

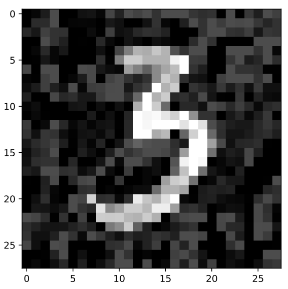
The cyclic behavior caused by high staleness. Consider taking a snapshot of the stale adversarial examples created throughout the training procedure, the same examples as those used for network training. For regular robust training with an ℓ_∞ constraint, the adversary does not care how many individual pixels it perturbs since the constraint restricts the maximum perturbation (see Equation 3). Figures 6a through 6d show the adversary's choice to perturb what looks like a large number of random pixels. In contrast, we notice that with $s = 64$ in Figures 6e through 6l, the adversarial examples look a lot less random. Specifically, the set of stale examples shadows a line segment that moves over the page and back again.

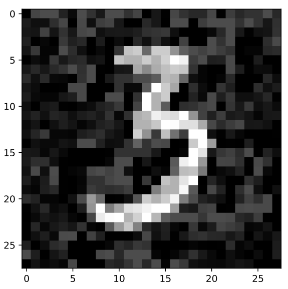
A game-playing analogy. Why do the adversarial examples cycle for high staleness? To understand further, consider a thought experiment with a game of rock, paper, and scissors. For each move made by a player, there is exactly one other move that wins against it, losses against it, and ties against it. Thus, if the adversary played randomly, the optimal move is to randomly choose an action, yielding a 1/3 chance of winning, a 1/3 chance of losing, and a 1/3 chance of tying. Now consider a large lag between the adversary's actions and when they are played out (i.e., add staleness). If by random chance the player has several rocks that win, he might put more weight into that action, playing it more often. The adversary would certainly punish this decision, but his paper moves would not appear for several more turns.

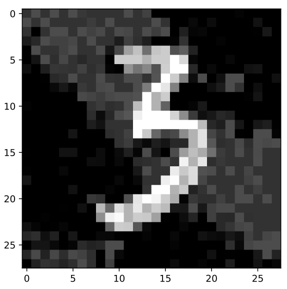
Finally, once the model gets flooded by paper, it learns to switch its weights, constantly playing scissors instead. In response, the adversary changes to rock, but its actions still show paper due

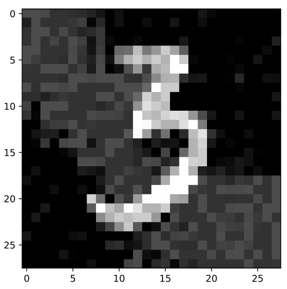

(a) $s = 0$

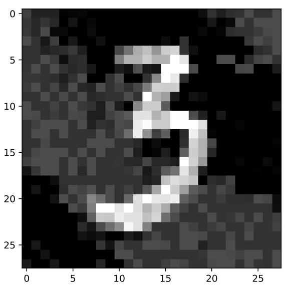

(b) $s = 0$

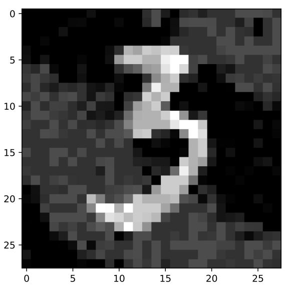

(c) $s = 0$

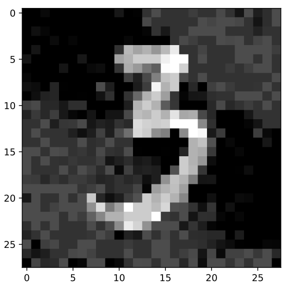

(d) $s = 0$

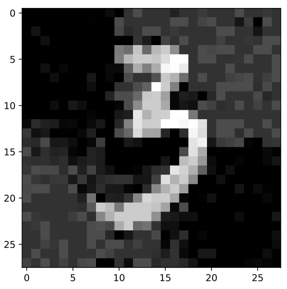

(e) $s = 64$

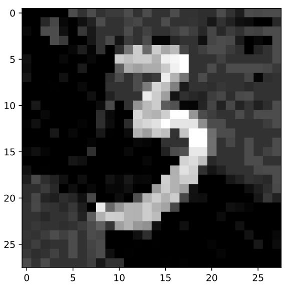

(f) $s = 64$


(g) $s = 64$


(h) $s = 64$


(i) $s = 64$


(j) $s = 64$


(k) $s = 64$

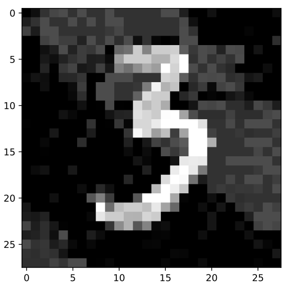

(l) $s = 64$

Figure 6: Figures 6a through 6d show adversarial examples at sequential timesteps for a network trained with no staleness. Figures 6e through 6l show adversarial examples at sequential timesteps for a network trained with $s = 64$. Note how the former looks nearly random while the latter shows a line segment cycling through the image. The cyclic behavior is one the drawbacks of high staleness: the network and adversary become entranced in a cat-and-mouse game trying to catch up with each other.

to the lag. This cat and mouse game would continue with the model switching to paper and the adversary switching to scissors until the model finally ends back where it started - rock.

This cyclic behavior is caused by the staleness between player and adversary. Both parties play the same move for a considerable amount of time due to the lag, causing a seemingly never-ending cycle. Drawing parallels between the rock, paper, scissors thought experiment and Adversarial Training, we believe high staleness also causes cyclic behavior, as evidenced by the line segments in Figure 6.

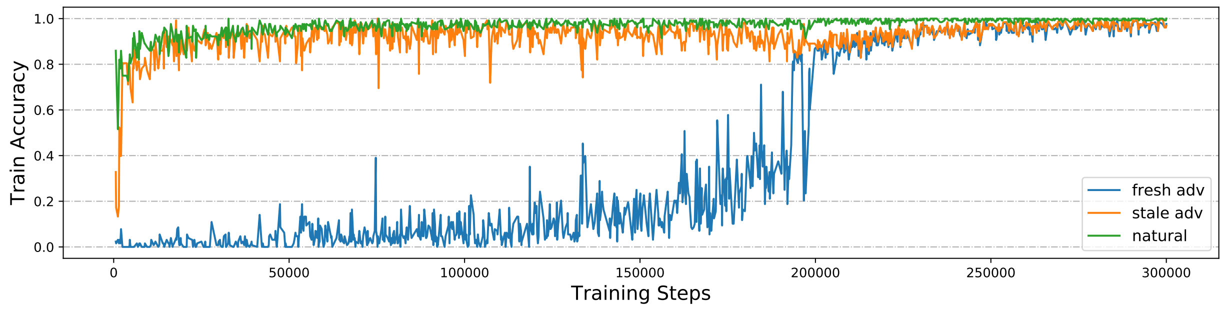
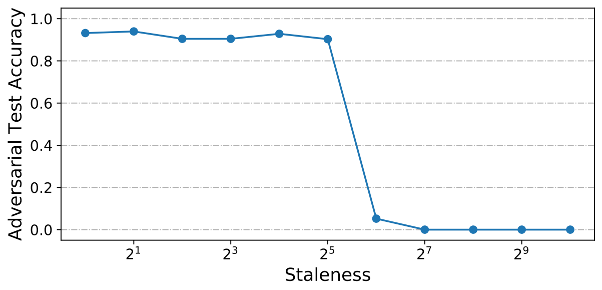
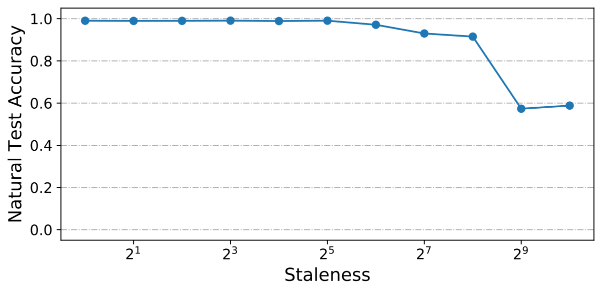


Figure 7: The broken $s = 64$ network run for 300,000 training steps instead of the usual 60,000.



(a) Final Adversarial Accuracy



(b) Final Natural Accuracy

Figure 8: Figures 8a and 8b show final adversary and natural accuracies, respectively, on the testing set for varying values of staleness. We increment staleness on the x-axis in powers of 2.

Breaking the cycles. Interestingly enough, we found that in some cases the cycles can be stopped by training the network with more timesteps. We took a network trained with $s = 64$ that previously exhibited memorization issues and let it run for 300,000 training steps instead of 60,000 training steps.

Figure 7 shows the results. About halfway through training, fresh adversarial accuracy suddenly climbs up to match stale adversarial accuracy and closely tracks it for the rest of training. While its unclear what exactly happened here, one theory is that the cyclic staleness behavior has a certain entropy over time. For higher staleness, it takes a longer build-up of this entropy to break the network's paralysis, but eventually it could happen.

Unfortunately, in the real-world it is impractical to train networks 300,000 timesteps due to the increased computation time. Thus, we pursue a balance between high staleness, which allows greater parallelization, and its drawbacks, which include memorization, lack of robustness, and cyclic behavior.

Towards a suitable staleness value. In our experiments, we sweep the staleness value from 1 to 1024 in powers of 2 to determine if there is a clear breaking point where the effects of staleness hindered robustness. Based on Figure 5a and 6, we know that a different exists between network trained with $s = 32$ and 64, at least based upon train accuracy. In Figure 8a, we plot the final adversarial accuracy on the *test* set as a function of the staleness used.

We note a similar phenomena as before regarding the difference between $s = 32$ and 64. For

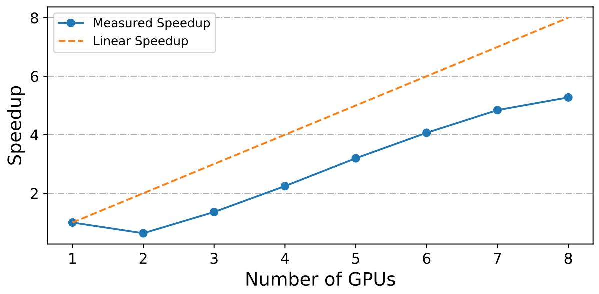


Figure 9: The speedup we achieve from using more GPUs in our asynchronous implementation of Adversarial Training. The baseline here (i.e., number of GPUs = 1) is regular PGD Adversarial Training with $\kappa = 8$. All subsequent runs used that same number of steps. Using asynchronous parallelization, we achieve a roughly-linear speedup as more GPUs are added.

$s \leq 32$, the final adversarial accuracy on the testing set remains about the same at 90%. However, after $s = 32$, adversarial accuracy drops down from 90% to around 5%, a major decrease that indicates a lack of robustness. Thus, for MNIST it appears that a network trained with $s = 32$ manages to achieve sufficient robustness, all while keeping a natural accuracy of around 98% (see Figure 8b).

Implementing Asynchronous Parallelization Finally, we bring staleness into the real world by implementing asynchronous parallelization. To achieve parallelism, we use between 0–7 worker GPUs and 1 trainer GPU. Between the workers and trainer sits a shared memory queue to store completed adversarial batches, and we restrict the queue's maximum size to mitigate staleness. Finally, we curb staleness by also having the trainer GPU re-save the model's parameters every 100 training steps, a fairly rapid pace.

In Figure 9, we plot the speedup as a function of the number of GPUs. Speedup is the time taken to train the baseline divided by the time taken to run the actual trial. Compared to the ideal linear-speedup shown in orange, we achieve slightly worse speedup for 2 GPUs since the trainer spends most of its time waiting for adversarial batches on the queue. We note that this can be fixed if the trainer also computes adversarial examples in its spare time waiting.

However, after 2 GPUs, we see an almost-linear speedup as the number of GPUs increases. By the time we get to 7 GPUs, we reach a nearly 5× speedup and start to plateau. By this time the worker GPUs combined begin to make adversarial batches at a faster rate than the network can consume them. This causes the queue to fill up, blocking the workers from being utilized until the queue has more space. It makes sense that this saturation point should occur when with 8 GPUs. We used $\kappa = 8$ for these experiments, so it takes approximately 8× as long to make an adversarial batch as it does to train on it.

Comparison with the baseline. With our final 8 GPU configuration, we reach over a 5× speedup compared to the $\kappa = 8$ baseline with one GPU. This configuration took a total 9 minutes to train while achieving an adversarial accuracy of around 90% and a natural accuracy of 98%. Compared to the 4 hours for the Mądry Lab MNIST challenge code with $k = 40$ and synchronous training, we achieve a 26× reduction in training time for the same level of robustness and same natural accuracy.

6 Experimental Setup

We perform all our experiments on the MNIST Computer Vision dataset [17], which consists of 65,000 labeled, grayscale, 28×28 images of handwritten digits. After feeding an MNIST image into a classifier, it must output the correct label, an integer from 0–9 that the image refers to. We use MNIST here since it has been widely studied not only in regular Machine Learning community, but also in the Adversarial Machine Learning community [8, 6, 1, 28, 19].

For all our tests, we use the same Convolutional Neural Network (CNN) used by Mądry et al. [19] and the TensorFlow MNIST tutorial [5]. This network consists of two convolutional and max pooling layers with 32 and 64 filters, respectively, and a receptive field of size 5. After being sent through the convolutional layers, data is passed through a fully-connected layer with 1024 neurons before finally reaching a 10 neuron fully-connected output layer. We use the ReLU activation function for all layers except the final layer, which uses the softmax function to create a discrete probability distribution. The CNN is trained with the Adam optimizer [12], a variant of Stochastic Gradient Descent with fast convergence. We use categorical crossentropy for our loss function, a common choice for multinomial classification problems. The same random seed is used throughout all experiments, and we randomly shuffle the entire MNIST dataset and feeding it in batches of 128 samples to the model.

To evaluate model performance, we use accuracy and loss across the train and test sets. The train set consists of 55,000 examples, while the test set contains 10,000 different examples. Since they are mutually exclusive, accuracy on the test set measures the generalization capability of the model to unseen data. In addition, we measure both accuracy and loss in the adversarial and natural setting. In the former, we replace the natural examples with adversarial examples to measure the model's robustness.

Unless otherwise noted, we use 60,000 training steps for each experiment and fix $\varepsilon = 0.3$, one third of the 0–1 grayscale range for each pixel. For a given number of steps, the step size is calculated as follows:

$$\alpha = \frac{\varepsilon \times 2.5}{\kappa}.$$

This ensures that the full optimization trajectory is divided into κ equally-sized steps. In addition, we scale step size by 2.5 since all of our experiments use random restarts (i.e., before starting PGD, we move to a random starting point within the ε-ball). Random restarts allow PGD to explore new areas of the ε-ball. However, just as it takes 2 radii to travel from one end of a diameter to another, it takes at least 2ε to travel from one end of the ball to another. The 0.5 adds a little breathing room for PGD to explore far-out adversarial examples.

Our baseline experiment is the randomly initialized model from Mądry et al.'s MNIST challenge, which achieves 89%–91% adversarial accuracy and 98% natural accuracy on the test set and takes roughly 4 hours to train.

7 Discussion

Other parallelization approaches. In this work, we approached parallelization through multiple GPUs computing independent batches of adversarial examples. Another possible approach is to use a large batch size, split the batch into several small chunks that fit into an individual GPU's memory, and give the large batch to the trainer GPU. In recent work using this parallelization technique, researchers trained a Computer Vision model on the large ImageNet dataset [9] in just one hour.

Instead of viewing this technique as competing, we view it as an orthogonal idea. To achieve real-world robust training, it is likely that ideas such as asynchronous parallelization must join with other approaches such large batch size training to bring down training times even further and scale to even larger datasets.

8 Conclusion

In this work, we present two novel techniques for reducing the training time of Adversarial Training: understanding attacker power and asynchronous parallelization. We present in-depth studies of both techniques, push their limits, and search for balance points wherein we do not sacrifice crucial traits such as adversarial robustness. Combining these two techniques together, we achieve a 26× reduction in robust training time on the MNIST dataset, going from 4 hours to 9 minutes. Overall, our work provides an important step towards efficient robustness.

References

[1] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *Proceedings of the International Conference on Machine Learning*, 2018.

[2] Dimitri P. Bertsekas. *Nonlinear Programming.* Athena Scientific, 1999.

[3] John M. Danskin. *The Theory of Max-Min and its Application to Weapons Allocation Problems.* Springer-Verlag Berlin Heidelberg, 1967.

[4] Google DeepMind. The Google DeepMind Challenge Match, March 2016. `https://deepmind.com/research/alphago/alphago-korea/`, 2016.

[5] TensorFlow Developers. Build a Convolutional Neural Network using Estimators. `https://www.tensorflow.org/tutorials/estimators/cnn`, 2018.

[6] Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Mądry. A Rotation and a Translation Suffice: Fooling CNNs with Simple Transformations. *arXiv preprint arXiv:1712.02779*, 2017.

[7] Volker Fischer, Kumar Mummadi Chaithanya, Jan Hendrik Metzen, and Thomas Brox. Adversarial Examples for Semantic Image Segmentation. In *Proceedings of the International Conference on Learning Representations – Workshop Track*, 2017.

[8] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *Proceedings of the International Conference on Learning Representations*, 2015.

[9] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[10] Andrew J. Hawkins. Inside Waymo's Strategy to Grow the Best Brains for Self-Driving Cars. `https://www.theverge.com/2018/5/9/17307156/google-waymo-driverless-cars-deep-learning-neural-net-interview`, 2018.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the Conference on Neural Information Processing Systems*, pages 1097–1105, 2012.

[14] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial Examples in the Physical World. In *Proceedings of the International Conference on Learning Representations – Workshop Track*, 2017.

[15] Mądry Lab. Mądry Lab MNIST Challenge. `https://github.com/MadryLab/mnist_challenge`, 2017.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep Learning. *Nature*, 521:436–444, 2015.

[17] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets. *arXiv preprint arXiv:1712.09913*, 2017.

[19] Aleksander Mądry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of the International Conference on Learning Representations*, 2018.

[20] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2017.

[21] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-Propagating Errors. *Nature*, 323:533–536, 1986.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[23] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[24] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[25] Jo Stichbury. Tesla Doubles Down on the Deep Learning Behind Autopilot. `https://dzone.com/articles/tesla-doubles-down-on-the-deep-learning-behind-aut`, 2018.

[26] Kyle Stock. Self-Driving Cars Can Handle Neither Rain nor Sleet nor Snow. `https://www.bloomberg.com/news/articles/2018-09-17/self-driving-cars-still-can-t-handle-bad-weather`, 2017.

[27] Apple Computer Vision Machine Learning Team. An On-device Deep Neural Network for Face Detection. `https://machinelearning.apple.com/2017/11/16/face-detection.html`, 2017.

[28] Eric Wong and J. Zico Kolter. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the International Conference on Machine Learning*, 2018.

[29] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding Deep Learning Requires Rethinking Generalization. In *Proceedings of the International Conference on Learning Representations*, 2017.