The Lottery Ticket Hypothesis in an Adversarial Setting

by

James Gilles

B.S. Computer Science and Engineering, Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 14, 2020

Certified by .....
Michael Carbin
Assistant Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

The Lottery Ticket Hypothesis in an Adversarial Setting

by

James Gilles

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Deep neural networks are vulnerable to adversarial examples, inputs which appear natural to humans but are misclassified by deep models with a high degree of confidence. The best known defenses against adversarial examples are network capacity and adversarial training. These defenses are very expensive, greatly increasing storage, computation, and energy costs. The Lottery Ticket Hypothesis ("LTH") may help ameliorate this problem. LTH proposes that deep neural networks contain "matching subnetworks", sparse subnetworks to which the network can be pruned early in training, without losing accuracy. In this thesis, we study whether LTH applies in the setting of adversarial training for image classification networks. We find that sparse matching subnetworks **indeed exist**, and can reduce model sizes as much as 96% early in training. We also find that the size of an architecture's smallest matching subnetworks is always roughly the same, whether or not adversarial training is used.

Thesis Supervisor: Michael Carbin
Title: Assistant Professor

Acknowledgments

In my training as a software engineer I did not anticipate working in a discipline as novel and mysterious as deep learning. I am grateful for the opportunity, and hope my work can help reduce the social and environmental costs of "artificial intelligence".

I am indebted to Jonathan Frankle and Michael Carbin for their patient guidance over the past year and a half. Their cautious approach to scientific work in uncharted territories has been an invaluable lesson. The members of the Programming Systems Group as a whole have been excellent friends. I'm glad I got to get silly questions answered by experts in a whole bunch of subfields I'd never heard of before.

In the past few years, the unwavering support of my friends and family has helped me overcome seemingly impassable obstacles. In no particular order, I'd like to thank: Max for pointing out the bugs; Em and Aaron for the fanfic recs; A for the reading; the Abu Jabers for the laughs; Beth and Rynn for the conversation; Floyd and Barbara for the peace of mind; E. Shavit for the referral; J. Cullen for the homework; J. Bartok and L. Wallace for the practice; Richard and Rayellen, for their endless love and support; and all those who helped meet my material needs while I performed this work.

Contents

List of Figures

List of Tables

Chapter 1

Intro

The study of deep learning has advanced by leaps and bounds in the past decade. Deep learning, a subfield of machine learning, studies *deep neural networks* (DNNs), an extremely expressive class of machine learning model. DNNs have proven very effective at recognizing images [27], text [22], and speech [21], and at other tasks like playing games [43]. Because of their effectiveness, they have begun to be deployed in high-risk situations, such as in loan decision algorithms [44] and the control systems of self-driving cars [2]. However, DNNs are difficult to understand. They have been termed "black boxes" [3], ingesting input and producing output, with no way to tell what is going on inside. We are therefore limited in our ability to tell if they are learning incorrect algorithms [14]. There is growing public concern about whether DNNs are safe to deploy in contexts with the potential to impact people's lives [39][17].

In this atmosphere of growing concern about the trustworthiness of DNNs, *adversarial attacks* pose a particularly thorny problem [15]. Adversarial attacks make small, nearly human-imperceptible changes to the inputs of DNNs, creating *adversarial examples*. These examples are misclassified by DNNs with high confidence [46]. (For example, see Figure 1-1.)

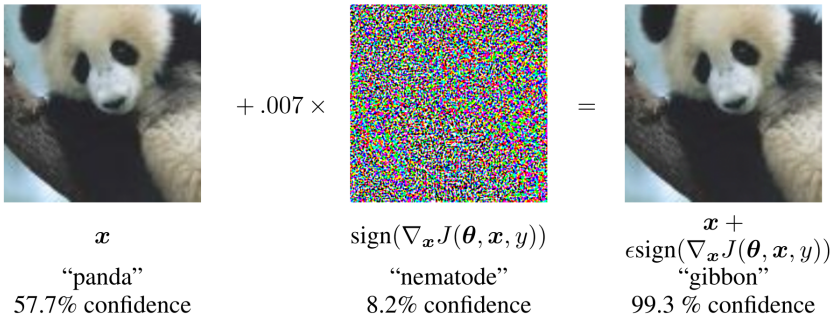

Figure 1-1: An adversarial example targeting the GoogLeNet [45] architecture, causing it to misclassify a panda as a gibbon. Image credit [15].

Adversarial attacks pose a significant security threat to real world systems. They can be used to attack online systems, even if the underlying DNN architecture is not known [36]. They can also be used to attack systems that perceive the physical world through sensors [28][7][5]. (For example, see Figure 1-2.) This style of attack poses a practical security risk for many DNN applications.

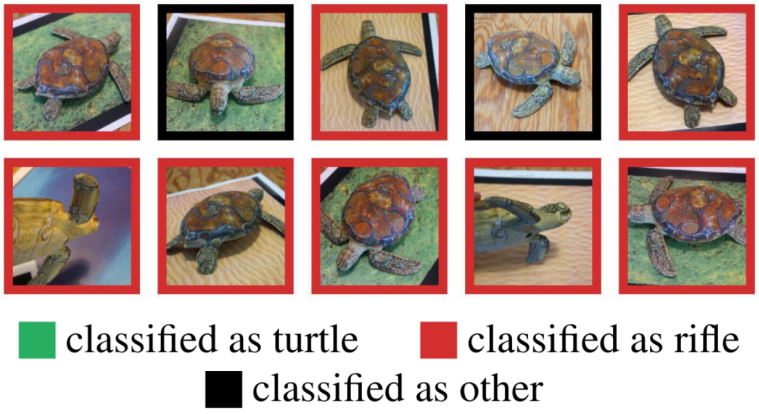

Figure 1-2: A physical adversarial example. The model used to 3d print this toy turtle was attacked using adversarial techniques. Image classification systems consistently mis-identify the 3d-printed toy as a rifle. Image credit [5].

It is desirable to defend DNNs from this form of attack. Two robust defenses that have been found are *increasing capacity* and *adversarial training* [55]. Increasing capacity requires adding more connections to a DNN, for instance by multiplying

its layer sizes by a factor of 10. Adversarial training is standard DNN training [1] where every input is modified by a synthetic adversary before being fed to the DNN. These defenses allow DNNs to retain some accuracy in the face of an adversary. Unfortunately, they are also very expensive, requiring significantly more storage and computation power. This makes them difficult to use in practice.

Pruning is a technique that severs connections within a DNN in order to reduce its storage and computation costs. An exciting recent development in pruning is the Lottery Ticket Hypothesis (LTH) [12][13]. LTH work hypothesizes the existence of *matching subnetworks*. These are small, sparse sub-networks that can be found within full DNNs early in training. If a DNN is pruned to a matching subnetwork, it can be trained to the same accuracy as a full DNN, nearly entirely in a pruned state. In practice, techniques such as *pruning with rewinding* (defined later) can shrink DNNs to 1-20% of their original size without any loss of accuracy. This reduces the storage needed for training and deploying DNNs, and also has the potential to reduce the and energy needed for DNN training.

In this thesis, I study whether the *pruning with rewinding procedure* finds *matching subnetworks* in the setting of adversarial training. I find that matching subnetworks **do exist** in the context of adversarial training, and validate the subnetworks I find by comparison with a number of baselines.

[1] That is, the *stochastic gradient descent* algorithm, typically using simple data augmentations and a learning rate schedule. I give a more rigorous definition in Chapter 3.

Chapter 2

Related Work

2.1 Adversarial Attacks and Defenses

Adversarial attacks in the context of deep learning were introduced by Szegedy et al. [46]. They coined the term *adversarial example*. In the broadest possible terms, an adversarial example is any DNN input where the DNN output is different from what a human viewing the input would expect [48]. Typically, adversarial examples are created by taking an input to a DNN, and modifying it by adding a small *perturbation*. For example, for an image classification DNN, the input would be an image, and the perturbation would be a set of small changes to the pixel values of the image. These pixel changes would be invisible to a human, but would be able to trick the model with high probability. An *adversarial attack*, then, is simply an algorithm to generate adversarial examples. Szegedy et al. introduced an attack that used numerical optimization to discover adversarial examples in the context of digit recognition[1]. They were able to identify very small changes, invisible to humans, but which fooled their target DNN.

Goodfellow et al. [15] constructed a simpler attack, the *Fast Gradient Sign Method* (FGSM). FGSM is very simple, involving only a single optimization step. It is therefore very efficient to compute. Goodfellow et al. also introduced one of the first

[1]Specifically, they used the box-constrained L-BGFS algorithm, and attacked a small neural network trained on the MNIST dataset [30].

adversarial defenses. An adversarial defense aims to prevent a DNN from being tricked by adversarial examples. The defense introduced was *adversarial training*, which is simply standard DNN training where every input has been adversarially attacked. Over the course of training, the DNN learns not to be fooled by the attack, and becomes *robust*. Adversarial training became feasible because FGSM was much less expensive to compute, relative to earlier attacks. Adversarial training was found to result in DNNs that are much more robust to adversarial attack, becoming much harder to fool.

Many other attacks and defenses have been proposed since. Xu et al. [55] provide an up-to-date survey of these attacks and defenses in a variety of settings. Broadly speaking, the attacking side is winning. Adversarial attacks are very effective, often reaching 100% effectiveness on the DNN for which they are generated. They also *transfer*: attacks generated for one DNN can fool other DNNs, even DNNs based on different architectures. It is also possible to create adversarial examples that work despite many kinds of distortion and filtering of the input. Kurakin et al. [28] create examples that work through cell phone cameras. That is, they print adversarial examples on standard printer paper, take photographs of them using a cell phone, and show that DNNs examining the photographs are still fooled. The attack they used is called the *Basic Iterative Method*; I focus on an extension of that attack in this thesis, *Projected Gradient Descent* (PGD). I will discuss PGD shortly.

Many types of defenses have been created – and then broken. For instance, *gradient masking* defenses attempt to hide gradient information from an adversary. In theory, this would prevent an adversary from applying most adversarial attacks, which use gradients. However, Carlini & Wagner [8] were able to overcome defenses of this type using attacks leveraging sparse gradient information. In general, gradient masking gives a false sense of security [4] – it prevents simple optimization-based adversaries from discovering adversarial examples, but that doesn't mean they can't be found in general! This sequence of events has repeated itself since, with new defenses frequently being broken by attacks adapted specifically to them [50]. For this reason, it is a best practice to validate new defenses using *adaptive attacks* [9]

custom-designed to foil them.

Madry et al. [37] give a useful formal characterization of adversarial attacks and training, describing previous algorithms as instances of a more general pattern. We summarize their formalization in Chapter 3. They introduced the *Projected Gradient Descent* (PGD) attack, a generalization of previous attacks. They suggest that PGD is *universal*, in the sense that it uses only first-order gradient information, and so should do about as well as any other algorithm with access to the same information. They also show empirically that DNNs trained against this attack become robust to future attacks of the same type. (However, the DNNs do not gain much robustness to other kinds of attack.)

Madry et al. also show that adding capacity to the DNN (for instance, multiplying layer sizes by ten) can increase robustness. A ResNet-20 [19] trained on the CIFAR10 dataset [26] can reach a standard accuracy of 92.7%. However, standard training results in DNNs that are extremely vulnerable to adversarial attack. A ResNet-20 achieves only 0.3% *adversarial accuracy* – i.e. accuracy on a test dataset where every input has been modified by an adversary. An adversarially trained ResNet-20 has 51.7% adversarial accuracy. Increasing DNN capacity improves robustness further. A WideResNet-20-10 [57] is a normal ResNet with layer sizes multiplied by 10. WideResNet-20-10 has accuracies of 95.2% standard : 3.5% attacked : 56.1% adversarially trained. This change (from 51.7% to 56.1%) is small, but it is one of the best known defenses beyond adversarial training.

This is a promising result. However, this style of adversarial training is extremely expensive. The DNN size is increased, requiring more storage and computation. Additionally, a PGD adversary that takes n steps for each input makes each training step take n times as long. DNN training is already extremely time-, storage-, and energy-intensive [18]. So the additional costs imposed by adversarial training can make it infeasible for large DNNs and datasets, which already push the limits of existing hardware. In the next subsection, I discuss a set of techniques that might help reduce these costs.

2.2 Network Compression and The Lottery Ticket Hypothesis

Network compression is the problem of reducing the size of DNNs, in storage or at inference time. There are many approaches to network compression, including pruning, quantization, and lossless algorithms; all of which have been applied successfully to DNNs [16]. Here I focus on pruning.

Pruning has a long history in the neural network literature, originally introduced in 1980s for MNIST networks [31], and later extended to large DNNs [34]. It can be divided into *structured* and *unstructured pruning*. Structured pruning removes entire neurons. Unstructured pruning removes only individual connections, leaving most neurons with at least a few inputs. In this thesis, I focus on unstructured pruning.

A typical pruning algorithm first performs a full training run on an unpruned DNN. After training, the DNN is pruned. This reduces accuracy, so the pruned DNN is then *fine-tuned* for some amount of time – that is, re-trained in a pruned state, to recover some accuracy.

Frankle et al. [12] introduce the Lottery Ticket Hypothesis, which conjectures that DNNs can be pruned **at the start of training** without harming accuracy. They show that some DNNs contain *matching subnetworks*[2] at initialization. DNNs can be pruned down to these subnetworks, and then trained entirely in a pruned state. The resulting DNNs are sparser and more accurate than DNNs that were fully trained, pruned, and then fine-tuned.

Early lottery ticket work [12] conjectured that these matching subnetworks exist within all DNNs at initialization, and showed that this is the case for small DNNs, like ResNet20 on CIFAR10. More recent work [13] has found that in larger DNNs, matching subnetworks do not necessarily exist at initialization. Instead, they appear early in training – between 2% and 10% of the way through the training regimen (when studying a variety of DNNs trained on the large ImageNet dataset).

[2]Earlier work termed these *lottery tickets*, but more recent work has adopted *matching subnetworks*. I use the latter terminology in this thesis.

Currently, the only technique known for discovering matching subnetworks is *pruning with rewinding*. Pruning with rewinding works by performing a full training run and then applying *global magnitude pruning* – that is, dropping the lowest-magnitude weights, until the DNN reaches the desired density. The dropped weights are extracted into a *pruning mask*, which selects which weights to keep and which to zero. The DNN is then *rewound* to a point early in training and pruned by applying the mask. The pruned subnetwork is then trained, as if for the first time. If the subnetwork reaches the same or higher accuracy, it is declared *matching*.

Note that there is currently no known way to discover matching subnetworks, aside from a full training run. However, the training run does not affect the weights of the pruned DNN in any way. It is used only as an oracle to discover which weights to prune. Finding ways to select weights to prune without a full training run is an active area of research [32][53][47].

Frankle et al. [13] also introduce the concept of the *trivial regime*. They observe that, in lightly pruned DNNs, many different techniques for pruning appear to work equally well. When 90% of connections are left intact, even completely random pruning will discover matching subnetworks. However, as pruning rates grow more aggressive, simple pruning techniques fail, and only pruning with rewinding is able to discover matching subnetworks. In this work, I compare pruning with rewinding to random early pruning (applied at the same epoch), to ensure that my results are not within the trivial regime.

2.3 Combining Compression and Adversarial Training

There have been several other attempts to combine network compression and adversarial training. Ye et al. [56] introduce a technique for concurrent adversarial training and weight pruning, repeatedly applying small amounts of pruning over the course of a training run. They also attempt to find matching subnetworks at initialization,

following the original LTH paper [12]. However, fail to find them a– because they only look for matching subnetworks at initialization, and not early in training.

Lin et al. [35] introduce "Defensive Quantization", which combines standard weight quantization with the "Parseval Networks" defense [10]. Parseval networks constrain the Lipschitz constants of individual DNN layers – the amount any layer, treated as a matrix, can increase the magnitude of its input vector. Constraining the Lipschitz constant prevents small changes in the input from "blowing up" into large changes in the output. Their technique is orthogonal to the application of adversarial training and pruning. Future work could investigate combining defensive quantization with adversarial LTH.

Li et al. [33] take a different approach to find matching subnetworks in the adversarial setting. Instead of using pruning with rewinding, they use hyperparameter search over learning rate schedules. They find schedules that give good accuracy when DNNs are trained, pruned, and retrained from initialization. They term the DNNs found through this procedure "Boosting Tickets", in a nod to the original Lottery Ticket paper. They achieve good accuracies and pruning rates, reaching 45.7% accuracy at an 80% pruning ratio on $\text{PGD}_{\infty,\frac{2}{255},10}$. They also show that with carefully chosen learning rate schedules, pruned subnetworks can converge to higher accuracies than the full network, after only a small number of epochs. However, their technique requires expensive hyperparameter searches to discover learning rate schedules for any choice of DNN architecture and adversary.

In this thesis, I focus on the *pruning with rewinding* technique, which has only one hyperparameter to tune, and has not previously been applied to adversarially trained DNNs.

Chapter 3

Methods

I aim to answer the question: How small are the smallest matching subnetworks found by pruning with rewinding combined with adversarial training? To answer this, I implement a framework combining adversarial training and pruning with rewinding, and run experiments to see if I can find matching subnetworks in a variety of experimental settings. Here I will provide a sketch of the mathematics of the relevant training algorithms, adversarial attacks, and pruning techniques, and an overview of my implementation.

Madry et al. [37] provide a useful formal characterization of adversarial attacks and defenses. I summarize their formalization here.

3.1 Standard Training

Consider a data distribution $\mathcal{D}$ consisting of a set of pairs $(x, y) \sim \mathcal{D}$, where $x \in \mathbb{R}^n$ is an input vector and $y \in \{0, 1\}^k$ is a human-supplied label vector. We can construct a deep neural network $f(x; \theta)$, which takes an input x and a vector of parameters $\theta \in \mathbb{R}^p$, and produces a label. I also define a loss $L(f(x; \theta), y)$, which measures how "wrong" a DNN is on some particular input-output pair (for instance, cross-entropy loss). Standard DNN training, then, attempts to find parameters that minimize the

loss over the training set:

$$\min_{\theta} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} [L(f(x;\theta), y)]$$

Where $\mathbb{E}$ is expectation, a weighted sum over all inputs. In practice, this minimization is performed using *stochastic gradient descent* (SGD) on the network weights. In pseudocode, SGD works as follows:

```
# given starting weights, an architecture, data, and a learning rate
def train(θ_0, f, 𝒟, λ):
  # repeat:
  for i in range(0, len(𝒟)):
    # sample input and label
    x, y ∼ 𝒟
    # compute DNN output
    ŷ ← f(x; θ_i)
    # compute how much the DNN output differs from the correct label
    l ← L(ŷ, y)
    # compute gradient of loss with respect to parameters --
    # that is, the amount the loss will increase or decrease,
    # with changes in each parameter
    g ← ∇_{θ_i} l
    # update weights to reduce loss
    # λ is the "learning rate", a hyperparameter
    θ_{i+1} ← θ_i − λg
  return θ_{len(𝒟)}
```

It's important to note that there is no guarantee that stochastic gradient descent will actually find the global minimum described above! It may not even find a local minimum – it can diverge completely if the learning rate is set too high. Nevertheless, in practice, stochastic gradient descent works extremely well for tuned DNNs.

In practice, SGD training is usually divided into *epochs* using a *learning rate schedule*. During each epoch of training, every input is seen once, in a random order. The *learning rate* hyperparameter controls how responsive the algorithm is to the loss gradient. If it is set too high, the algorithm will diverge, and if it is set too low, the algorithm will converge very slowly. A learning rate schedule allows the learning rate to be changed over the course of training. Typically, a high learning rate is used during early epochs, and a lower learning rate is used later.

3.2 Adversarial Attacks

Adversarial attacks attempt to find *adversarial examples* which "fool" a DNN. In the broadest sense, this means finding any input x^* where the DNN output $f(x^*;\theta)$ disagrees with a human oracle $\mathcal{O}(x^*)$: $f(x^*;\theta) \neq \mathcal{O}(x^*)$. Unfortunately, this definition is difficult to use in practice [48]. Human perception is hard to pin down mathematically. The best one can do is rely on human labeling of inputs, which is labor-intensive.

For this reason, we focus on a relaxation of this problem based on "small" perturbations. Given an input x with a human-provided label $y = \mathcal{O}(x)$, we would like to find an input x^* that is a "small distance" from x that is mislabeled: $f(x^*;\theta) \neq y$. We assume that, since the distance between x^* and x is "small", the oracle label will usually not change: $\mathcal{O}(x^*) \approx \mathcal{O}(x)$[1]. So we will usually find a true adversarial example.

The precise meaning of "small" depends on the distance metric chosen. Most work in image classification has focused on standard ℓ_p norms, which treat the input and modification as vectors in high-dimensional space[2]. So given x, y and f, the problem is then to select an x^*, such that $f(x^*;\theta) \neq y$ and $\ell_p(x^* - x) \leq \epsilon$, for some maximum distance ϵ. This can be viewed as selecting a perturbation d from a set of allowable

[1] Of course this is not always the case. See Tram{\'e}r et al. [48] for more discussion of this topic.

[2] Other optimization spaces are possible, though. [54] propose a distance metric which allows, roughly, small smudges to be applied to an image. The adversarial examples they generate have high ℓ_p distances from the original inputs, but they are nearly indistinguishable to human vision.

perturbations S:

$$\text{find } d \in S \text{ such that } f(x+d;\theta) \neq y$$

Where $S = \{d \mid d \in \mathbb{R}^n, \ell_p(d) \leq \epsilon\}$ is the set of perturbations of magnitude less than ϵ. This problem has a sharp boundary. We can smooth it by using a loss function L to measure the distance between $f(x+d;\theta)$ and y[3]. We then have a maximization problem:

$$\max_{d \in S} L(f(x+d;\theta), y)$$

To solve this maximization problem, Madry et al. [37] define *Projected Gradient Descent* attack. Projected gradient descent is works by selecting a random starting perturbation within S. Then, n steps of gradient descent are taken to maximize the loss. Each step i is constrained to be small, by projecting it into a subspace S_i around the previous perturbation. This can be thought of as taking a number of small steps of length a, which are allowed to sum to a maximum length of ϵ. This forces the adversary to explore the **interior** of the subspace S[4]

```
# given an architecture, weights, an input and label,
# an subspace, and a number of steps
def pgd(f, θ, x, y, S, n):
  # initialize a random perturbation within the allowed subspace
  p_0 ~ S
  # repeat:
  for i in range(0, n):
    # compute output for attacked input
    ŷ ← f(x + p_i; θ)
    # compute loss
    l ← L(ŷ, y)
```

[3] L does not have to be the same loss used during training, although in this thesis it always is – training and attacks both use cross-entropy loss.

[4] Without being forced to take small steps, projected gradient descent will only be able to explore the **boundary** of S. This greatly reduces the space available for the algorithm to explore, and so should be avoided.

```
        # compute the gradient of the loss wrt the perturbation:
        g ← ∇_p l
        # update the perturbation to *maximize* the loss:
        p_{i+1} ← p_i + g
        # project the perturbation back into
        # the subspace for this step (and the overall subspace)
        p_{i+1} ← project(p_{i+1}, S_i ∩ S)
    return x + p
```

Again, this algorithm is in no way guaranteed to find the best possible perturbation in the given subspace. However, in practice, it is known to be very good at fooling DNNs [37].

Besides the network and image to attack, this algorithm has three hyperparameters: the subspace S, the step subspace S_i, and the number of steps to take n. Following previous work [37], I focus on ℓ_p ball subspaces, where perturbations are allowed to have a maximum magnitude of ϵ according to the ℓ_p metric. I describe PGD adversaries as $\text{PGD}_{p,\epsilon,n}$, where p is the order of the metric, ϵ is the radius of the l_p ball, and n is the number of PGD steps taken within it. For S_i, I choose a ball of size a around the previous PGD step. I choose $a = \epsilon/n$; that is, the adversary takes n steps of size a, which can add up to a maximum of ϵ.

For example: $\text{PGD}_{\infty,0.1,8}$ refers to an adversary that takes 8 steps in an ℓ_∞ ball of size 0.1 (given an input vector with components normalized to $[0, 1]$).

3.3 Adversarial Training

Adversarial training attempts to find a network that is difficult to attack. We would like to limit the amount an adversary can possibly confuse the network. This can be written as a min-max problem.

$$\min_{\theta} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \left[\max_{d\in S} L(f(x+d;\theta), y)\right]$$

That is, find the parameters θ which *minimize* the *maximum* amount an adversary can possibly confuse the network, around all inputs in the training data.

In practice, this problem can be solved by applying PGD to every to every input seen during training. This trains the network to ignore the adversary, and can result in DNNs that are significantly more robust than those trained in the standard way. The algorithm is as follows:

```
def adv_train(f, 𝒟, λ, S, n):
    θ_0 ∼ RAND_INIT()
    for i in range(0, len(𝒟)):
        x, y ∼ 𝒟
        # compute prediction on clean data
        ŷ ← f(x_adv; θ_i)
        # attack with pgd:
        # attempt to change the network's prediction
        x_adv ← pgd(f, θ_i, x, ŷ, S, n)
        # train on the attacked input
        ŷ_adv ← f(x_adv; θ_i)
        l ← L(ŷ_adv, y)
        g ← ∇_{θ_i} l
        θ_{i+1} ← θ_i − λg
```

Note that, during training, the label argument passed to PGD is based on the model's prediction $\hat{y}$, rather than the data's true label y. This is necessary to avoid the phenomenon of *label leaking* [29], where using the true label during training can inadvertently leak information to the model for weak attacks.

This algorithm is very expensive. PGD with n steps is roughly $n + 2$ times as expensive as standard training, in terms of computation: it requires a full inference to compute the model's prediction for the input data, one run of backpropagation for each PGD step, and then another backpropagation step to update the model weights.

3.4 Pruning

Now I add pruning to the algorithm. Given a DNN $f(x;\theta)$ I define a *subnetwork* $f(x; m \odot \theta)$, where $\odot$ denotes elementwise multiplication, and $m \in \{0,1\}^p$ is a mask which selects weights to force to zero – that is, which connections to sever. There are many different algorithms to discover subnetworks. In this thesis I use *global magnitude pruning*. Global magnitude pruning (GMP) simply severs the lowest-magnitude connections in a network. If a network with 25% of connections remaining is desired, GMP would sort all the connections in a network by weight, and drop the bottom 75% of them.

Typically, global magnitude pruning is applied at the end of training, and the network is then trained further to recover lost performance [12]. This procedure is called "fine-tuning":

```
# given an architecture, data, an epoch count, a pruning rate, and a learning rate schedule
def fine_tune(f, 𝒟, N, w, λ):
    # randomly initialize weights
    θ_0 ← RAND_INIT()
    # initial training run; N epochs
    for i in range(0, N):
      θ_{i+1} ← train(f, θ_i, 𝒟, λ_i)
    # discover mask
    m ← global_magnitude_pruning(θ_N, w)
    # apply mask
    θ*_0 ← θ_N · m
    # fine-tuning training run; N epochs
    for i in range(0, N):
      θ*_{i+1} ← train(f, θ*_i, 𝒟, λ_i)
    return θ*_N
```

However, it is in fact possible to prune very early in training. The *pruning with rewinding* methodology [13] gives us a technique for this. First, a full training run is performed, and a mask is discovered through global magnitude pruning. However, before applying the mask, the network is *rewound* to early in training:

```
# new parameter: r, rewind epoch
def prune_with_rewinding(f, 𝒟, N, w, λ, r):
    θ_0 ← RAND_INIT()
    for i in range(0, N):
      θ_{i+1} ← train(f, θ_i, 𝒟, λ_i)
    m ← global_magnitude_pruning(θ_N, w)
    # apply mask to rewound network: θ_r instead of θ_N
    θ*_0 ← θ_r · m
    for i in range(0, N):
       θ*_{i+1} ← train(f, θ*_i, 𝒟, λ_i)
    return θ*_N
```

A *matching subnetwork* is a subnetwork where the final pruned network beats the accuracy of the full network. The fact that these matching subnetworks exist (for standard training) show that the network **could** have been pruned early in training and trained in a pruned state. However, currently there is no known algorithm for discovering the relevant mask, aside from a full training run.

3.5 Final Algorithm

Now I aim to answer whether this is possible for adversarial training. The final algorithm is a simple combination of pruning with rewinding and adversarial training, where `adv_train` is used instead of `train`.

```
def adv_prune_with_rewinding(f, 𝒟, N, w, λ, r, S, n):
    θ_0 ← RAND_INIT()
```

```
    for i in range(0, N):
      θ_{i+1} ← adv_train(f, θ_i, D, λ_i, S, n)
    m ← global_magnitude_pruning(θ_N, w)
    θ*_0 ← θ_r · m
    for i in range(0, N):
      θ*_{i+1} ← adv_train(f, θ*_i, D, λ_i, S, n)
    return θ*_N
```

The algorithm is straightforward, but has many hyperparameters, including:

- The dataset ($\mathcal{D}$)
- The network architecture (f)
- The training schedule (N, λ)
- The perturbation subspace (S), step subspace (S_i) and number of steps taken by the adversary (n)
- The fraction of weights to leave intact (w)
- The rewind epoch (r)

Following previous work, I focus on adversarial attacks on image classification DNNs. Adversarial training is very expensive, and I aim to run a large number of experiments, so I focus on the moderately-sized CIFAR10 dataset [26] rather than a larger dataset like ImageNet [41]. For network architecture, I select ResNets [19] and WideResNets [57], which are widely-studied architectures that allow easy scaling. [5]. Note that WideResNets can be viewed simply as ResNets with scaled layer sizes: WideResNet-20-1 is precisely the same as ResNet-20, and WideResNet-20-8 is the same as a ResNet-20 with its activation vector sizes multiplied by 8. I use the standard ResNet learning rate schedule.

[5] Specifically, I focus on ResNet/WideResNet v1 rather than ResNet v2 [20], with Conv-BN-ReLU residual units

For data augmentation, I apply random left-right flips, and padding of 4 pixels + random cropping back to 32 pixels. I normalize input pixels to the range $[0, 1]$, apply adversarial attacks constrained to $[0, 1]$, and then **inside the model** perform dataset normalization. This simulates real-world attacks, where an adversary cannot create pixel values out of the $[0, 1]$ range expressible in an RGB image. I focus on the PGD adversarial attack under the ℓ_∞ norm, although other attacks could easily be swapped into the algorithm. [6] I fix $\epsilon = 0.1$, $a = \epsilon/n$.

This leaves a number of hyperparameters available to explore:

- PGD parameters:
  - n: the number of steps taken by projected gradient descent.
- WideResNet parameters:
  - The number of layers. I fix this to 20.
  - k: width, the multiplier applied to all layer sizes. I study $k = 1$ (Resnet-20) and $k = 8$ (WideResNet-20-8)[7].
- Pruning parameters:
  - w: the density of the pruned network (i.e. the fraction of weights **remaining** after pruning).
  - r: the rewind epoch I apply pruning at. I select $r = 8$.

I perform several grid searches across these hyperparameters. **I find that matching subnetworks exist early in training for a variety of adversary strengths, pruning rates, and WideResNet sizes.**

My code is implemented using Tensorflow 2[1], using the CleverHans library for adversarial attacks[40] – specifically, the `cleverhans.future` extension for Tensorflow

[6] The ℓ_∞ norm constrains the maximum amount any individual pixel can vary, so an ϵ of 0.1 would allow all pixels to be modified by a maximum of 0.1.

[7] The original WideResNets paper [57] adds 2 to the layer count in names, for some reason denoting this architecture WideResNet-22-1. I ignore this and denote it WideResNet-20-1, for the sake of consistency with the original ResNets paper[19].

2. I parallelize experiments using the Dask library for cluster management[11]. I collate and plot data using the xarray[23], Holoviews[6], and Matplotlib[24] libraries.

Chapter 4

Experimental Results

I aim to answer the question: How small are the smallest matching subnetworks found by pruning with rewinding combined with adversarial training? I break this into three sub-questions:

Sub-question 1. Does pruning with rewinding discover **any** matching subnetworks? (That is, does it ever find subnetworks that **meet or exceed** the accuracy of the full DNN?)

Sub-question 2. If found, are any of these subnetworks **non-trivial**? (That is, are they found at densities where random pruning cannot match the accuracy of the full DNN?)

Sub-question 3. If found, how do the minimum sizes of these subnetworks compare to the minimum sizes found for natural training?

4.1 Methodology

Experimental Framework I use the experimental framework described in Methods, applying pruning with rewinding combined with adversarial training. The main independent variable I vary is the density of the pruned subnetwork, w. The depen-

dent variable is adversarial test accuracy: classification accuracy on a set of images the DNN has never seen before, attacked by the same adversary used during training.

I study multiple architectures and adversaries to ensure my results hold in a variety of settings. I plot network density against adversarial test accuracy for 4 adversaries of different strengths: $\text{PGD}_{\infty,0.1,1}$, $\text{PGD}_{\infty,0.1,2}$, $\text{PGD}_{\infty,0.1,4}$, and $\text{PGD}_{\infty,0.1,8}$. I also include standard training, i.e. no adversary, for the sake of comparison. I compare across two DNN architectures, ResNet-20 and WideResNet-20-8. I use a rewind epoch (r) of 8, out of 160 training epochs total. The random pruning baseline is applied at the same training epoch, and prunes randomly across the network, with all weights given an even chance of being cut. A full table of all hyperparameters used in this experiment is available in Table A.1.

4.2 Results

Plots. Figure 4-1 shows the adversarial accuracy of ResNet-20 when pruned to various densities, and Figure 4-2 shows the same for WideResNet-20-8.

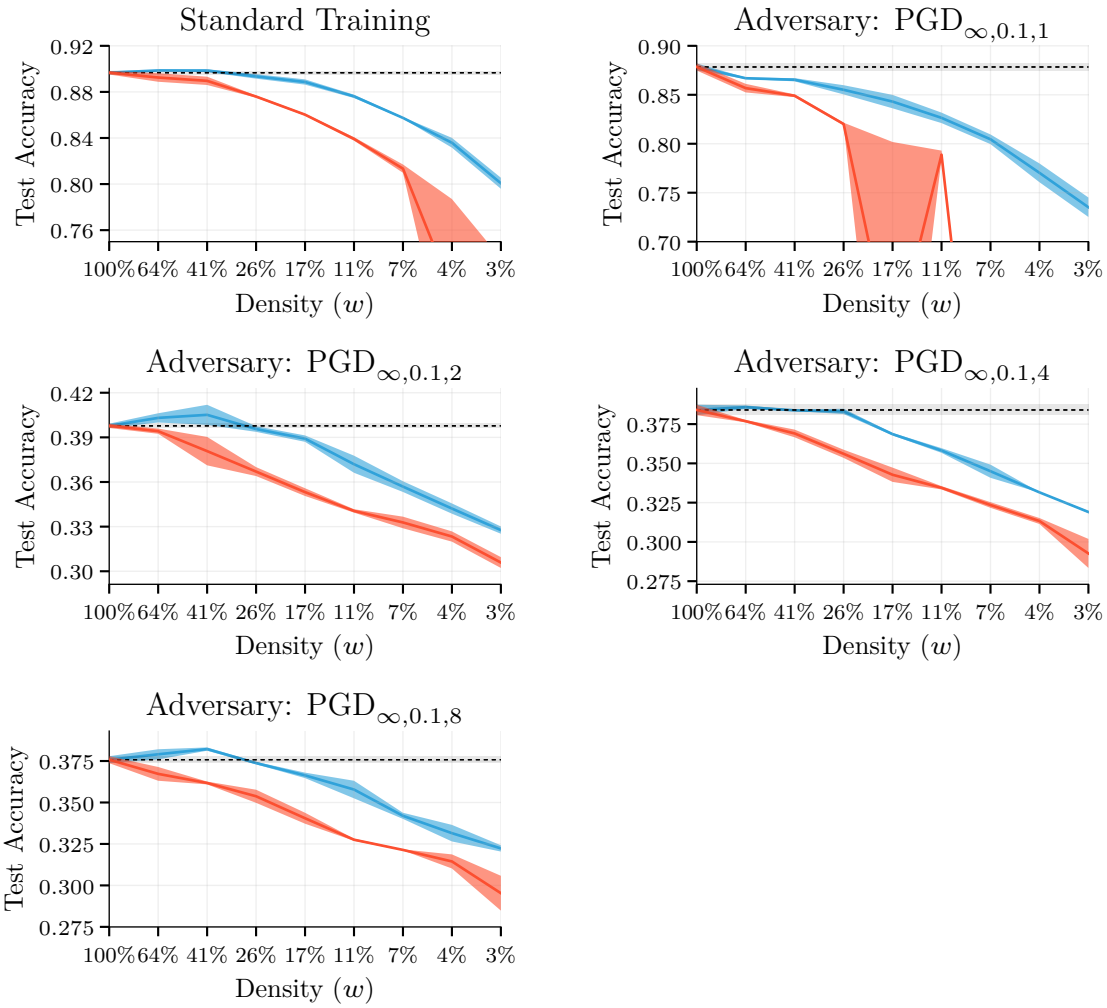

Figure 4-1: Pruning-accuracy curves for **ResNet-20** on **CIFAR10**. Pruning with rewinding in blue, random pruning baseline in red; dashed line shows full network accuracy. Spreads show standard deviation across **2** replicates. All DNNs were tested on the adversary they trained on. **Different y axes**. Full hyperparameters available in Table A.1.

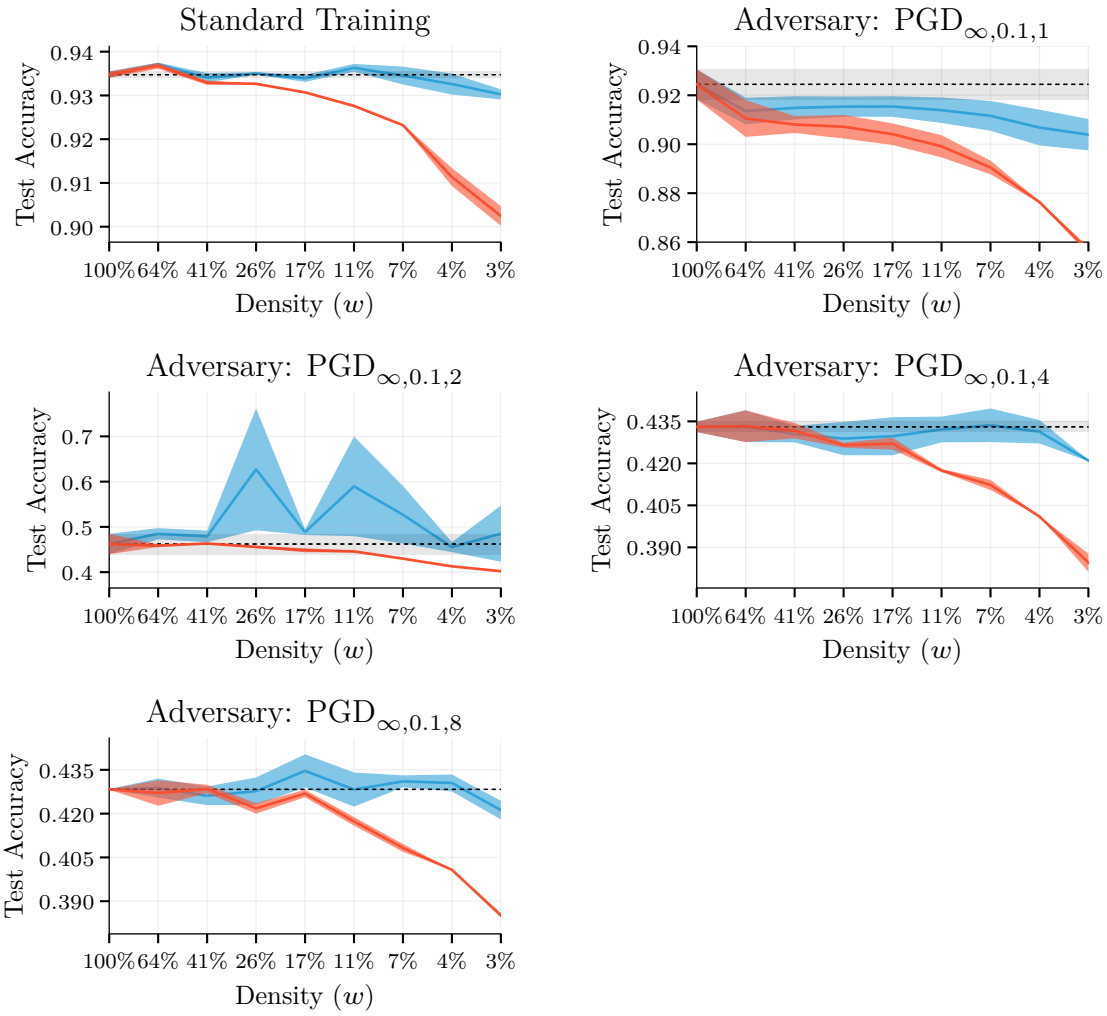

Figure 4-2: Pruning-accuracy curves for **WideResNet-20-8** on **CIFAR10**. Pruning with rewinding in blue, random pruning baseline in red; dashed line shows full network accuracy. Spreads show standard deviation across **2** replicates. All DNNs tested on the adversary they trained on. **Different y axes**. Full hyperparameters available in Table A.1.

Sub-question 1. I find that **matching subnetworks exist** for most adversaries. Matching subnetworks can be found where the pruning-with-rewinding accuracy meets or exceeds the full-network accuracy. This occurs for all adversaries except $\text{PGD}_{\infty,0.1,1}$.

Sub-question 2. These matching subnetworks occur at **nontrivial sparsities**. For very dense subnetworks, both pruning strategies are equivalent; this can be seen at the left side of each plot, where both random pruning and pruning with rewinding match the full network accuracy. However, as density decreases, random pruning quickly stops matching, making the remaining matching subnetworks found by pruning with rewinding **nontrivial**.

Sub-question 3. I find that the smallest matching subnetworks found under adversarial training are **the same size or smaller** as those found under standard training. The smallest matching subnetworks exist at **26%** density for ResNet-20, and at **4%** density for WideResNet-20-8, for standard training and all adversaries except $\text{PGD}_{\infty,0.1,1}$. $\text{PGD}_{\infty,0.1,2}$ on WideResNet-20-8 has a matching subnetwork at a lower density than that of natural training, 3%.

Chapter 5

Discussion

Findings. I find that **nontrivial matching subnetworks exist** for most adversaries, and that they are **no larger for adversarial training than for standard training**. That is to say, we can prune very early in training, train only the pruned subnetwork, and have resulting DNN reach the same accuracy as a full-sized network. This result cannot currently reduce the cost of training, since a full training run is required to discover the early pruning mask. However, discovering early pruning masks is an active area of research[32][53][47]. These results suggest that, if a technique for discovering matching subnetworks early in training is found, it can be fruitfully applied in the context of adversarial training.

Possible Implications. More speculatively, these results suggest that there may be a limitation on the number of parameters a DNN can effectively use. Previous research has implied that adversarial training may be a qualitatively different task from natural training [51][25]. However, I found that standard and adversarially trained networks can be pruned the same amount. On CIFAR10, after 8 epochs of training, **75%** of weights in a ResNet-20 and **96%** of weights in a WideResNet-20-8 seem to become useless. Why are these weights not being used? From these results I cannot say, but it is possible that this is a fundamental limitation of these architectures. Investigating whether these pruning rates hold on other tasks could be an interesting direction for future work.

5.1 Limitations

Limited hyperparameters. While I sample a large number of pruning rates, I am only able to study a small number of other hyperparameters. I study only two DNN architectures – ResNet-20, and its wider variant WideResNet-20-8. I also study only a small number of adversary settings, using only PGD under the ℓ_∞ metric. Future work could study other architectures and adversaries – such as PGD under the ℓ_2 or ℓ_1 metrics, or more exotic attacks [54]. I also run only 2 replicates for each set of hyperparameter settings. A few of our plots have high variances, suggesting that more replicates might be useful.

Nonstandard ϵ. I use a nonstandard value of ϵ for CIFAR10 – 0.1, for pixels in the range $[0, 1]$. In the literature, many different ϵ values are used, but smaller values such as $\frac{8}{256}$ are used for CIFAR10 in some seminal works[37]. Overlarge values of ϵ have been known to sometimes cause problems[48]. Unfortunately, I realized this discrepancy after running most of my experiments, and due to time limitations I was unable to re-run them using a typical value. Appendix B reports additional results for $\epsilon = .3$ which are consistent with my observations. Future work could try the standard value of ϵ, and indeed study other possible values as well.

No adaptive validation. It is a best practice to validate new adversarial defenses using *adaptive attacks* custom-designed to defeat them [9]. I do not perform adaptive validation in this thesis, because I have not introduced any new defenses. I am instead investigating the interaction between an existing defense (adversarial training) and pruning. Moreover, adversarial training is known to be quite strong. In competitions to discover adaptive attacks for WideResNets trained on CIFAR10 [52], no adaptive attacks have been able to reduce the accuracy of PGD-trained networks by more than a few percent[1]. So it is not clear what adaptive attack I could have used, even if I did wish to use one.

[1] …when optimizing within the same subspace S. However, such networks are not robust within subspaces induced by other distance metrics[37]. Defending networks against multiple perturbation types is an active area of research [49][38].

5.2 Other Observations

Relative Pruning Rates. WideResNet-20-8 can be pruned much farther than ResNet-20, to 4% density rather than 24% density. This is somewhat unsurprising, since WideResNet-20-8 is 64 times larger than ResNet-20[2]. The parameter count of the pruned WideResNet is $64 * 0.04 = 2.56$ times the parameter count of the original ResNet. Increasing network size is a common adversarial defense [37]; this result shows that pruning with rewinding can significantly reduce the cost of this defense, at least in terms of parameter count. An future experiment could push pruning further for WideResnet, until parameter counts are directly comparable with ResNet, and then compare their relative accuracies.

Groupings of adversarial accuracies. For both architectures under study, standard training and training on $\text{PGD}_{\infty,0.1,1}$ result in accuracies within 10 points of 90%. By comparison, $\text{PGD}_{\infty,0.1,2}$, $\text{PGD}_{\infty,0.1,4}$, $\text{PGD}_{\infty,0.1,8}$ result in accuracies within 10 points of 40%. This is more apparent in the full-y-axis versions of the figures available in the appendices, Figures C-1 and C-2). So, rather than accuracy decreasing steadily as n increases, it stays constant and then drops abruptly. The exception is $\text{PGD}_{\infty,0.1,2}$ with WideResNet-20-8. Networks trained with these settings are mostly within 10 points of 40% accuracy, except for a few outliers with 76%, 69%, and 59% respectively. These large jumps in adversarial accuracy occur nowhere else in the dataset. To me, this behavior resembles a phase transition [42], with high accuracy on one side and low accuracy on the other. WideResNet-20-8 with $\text{PGD}_{\infty,0.1,2}$ resides on a metastable position on the border, able to fall into one phase or the other by chance. Future work could study whether scanning across other hyperparameters such ϵ leads to similar phase-like behavior. See Appendix C.3 for more information.

[2]This may be somewhat confusing, given the name. So, note that WideResNet-20-8 has all *activation vectors* widened by a factor of 8. This increases the overall parameter count of the network by a factor of 8^2, or 64. This is because the parameter count of a layer scales linearly with the sizes of its input **and** output vector. So if both are increased, the size of the network increases quadratically.

5.3 Conclusion

I find that, once pruning masks are discovered, they can be applied very early in adversarial training, and the resulting sparse networks train to comparable accuracy to full networks. This allows significant size reductions of trained networks. In addition, it has the potential for energy savings and training time reductions. Unfortunately, there are currently no known algorithms for discovering early pruning masks aside from pruning with rewinding [13]. Our results suggest that, if such a technique is discovered, it can be applied fruitfully in the adversarial setting.

Appendix A

Hyperparameter Settings

Variable parameters are in **bold**.

Table A.1: PGD Strength Comparison Hyperparameters

Hyperparameter	Value
Replications	2
Dataset	CIFAR10
Pixel range	$[0, 1]$ + CIFAR10 standardization in model
Data augmentation	Horizontal flips, random crop with up to 4 0 pixels per side
Data shuffling	Yes
Batch size	128
Training epochs	160
Training loss	Cross-entropy
Training Optimizer	SGD
SGD LR (epoch 0)	0.1
SGD LR (epoch 80)	0.01
SGD LR (epoch 120)	0.001
SGD momentum	0.9
SGD Nesterov	No
Density	80% ^ [2,4,6,8,10,12,14,16]
Pruning strategy	[global magnitude pruning with rewinding, random]
Rewind point	epoch 8
Training attack	PGD
PGD norm	l_∞
PGD steps (k)	[0,1,2,4,8]
PGD epsilon (ϵ)	0.1
PGD step epsilon (a)	ϵ/n
PGD random initialization	Yes
PGD label-leaking mitigation	Yes
PGD loss	Cross-entropy
Model	WideResnet v1 (Conv-BN-ReLU)
Model width multiplier	[1, 8]
Resnet Blocks	20

Table A.2: Rewind Epoch Search Hyperparameters

Hyperparameter	Value
Replications	4
Dataset	CIFAR10
Pixel range	$[0, 1]$ + CIFAR10 standardization in model
Data augmentation	Horizontal flips, random crop with up to 4 0 pixels per side
Data shuffling	Yes
Batch size	128
Training epochs	160
Training loss	Cross-entropy
Training Optimizer	SGD
SGD LR (epoch 0)	0.1
SGD LR (epoch 80)	0.01
SGD LR (epoch 120)	0.001
SGD momentum	0.9
SGD Nesterov	No
Weights remaining (w)	[30%, 50%, 70%]
Test adversary	Same as train adversary
Pruning strategy	[global magnitude pruning with rewinding, random]
Rewind point (r)	epochs [0, 1, 2, 4, 8, 16, 32, 64, 128]
Training attack	PGD
PGD norm	l_∞
PGD epsilon (ϵ)	0.3
PGD steps (n	1
PGD step epsilon (a)	ϵ/n
PGD random initialization	Yes
PGD label-leaking mitigation	Yes
PGD loss	Same as training (Cross-entropy)
Model	ResNet v1 (Conv-BN-ReLU)
Model width multiplier	1
ResNet Blocks	20

Appendix B

Rewind Epoch Comparison

Before the primary experiment listed in the paper, I ran an experiment to determine the correct rewind epoch r to use for pruning with rewinding. I fixed 3 pruning rates and compared power-of-two rewind epochs. Unfortunately, this experiment used $\text{PGD}_{\infty,0.3,1}$ – an ϵ of 0.3, which is extremely large for CIFAR10. The fact that these networks converged at all is somewhat surprising, and is likely due to the limited ability of PGD with $n = 1$ to optimize within the available space; it could also be an indication of learned gradient masking [4]. I realized this after the fact, and lowered ϵ for my later experiments. Full hyperparameters for this experiment available in Table A.2.

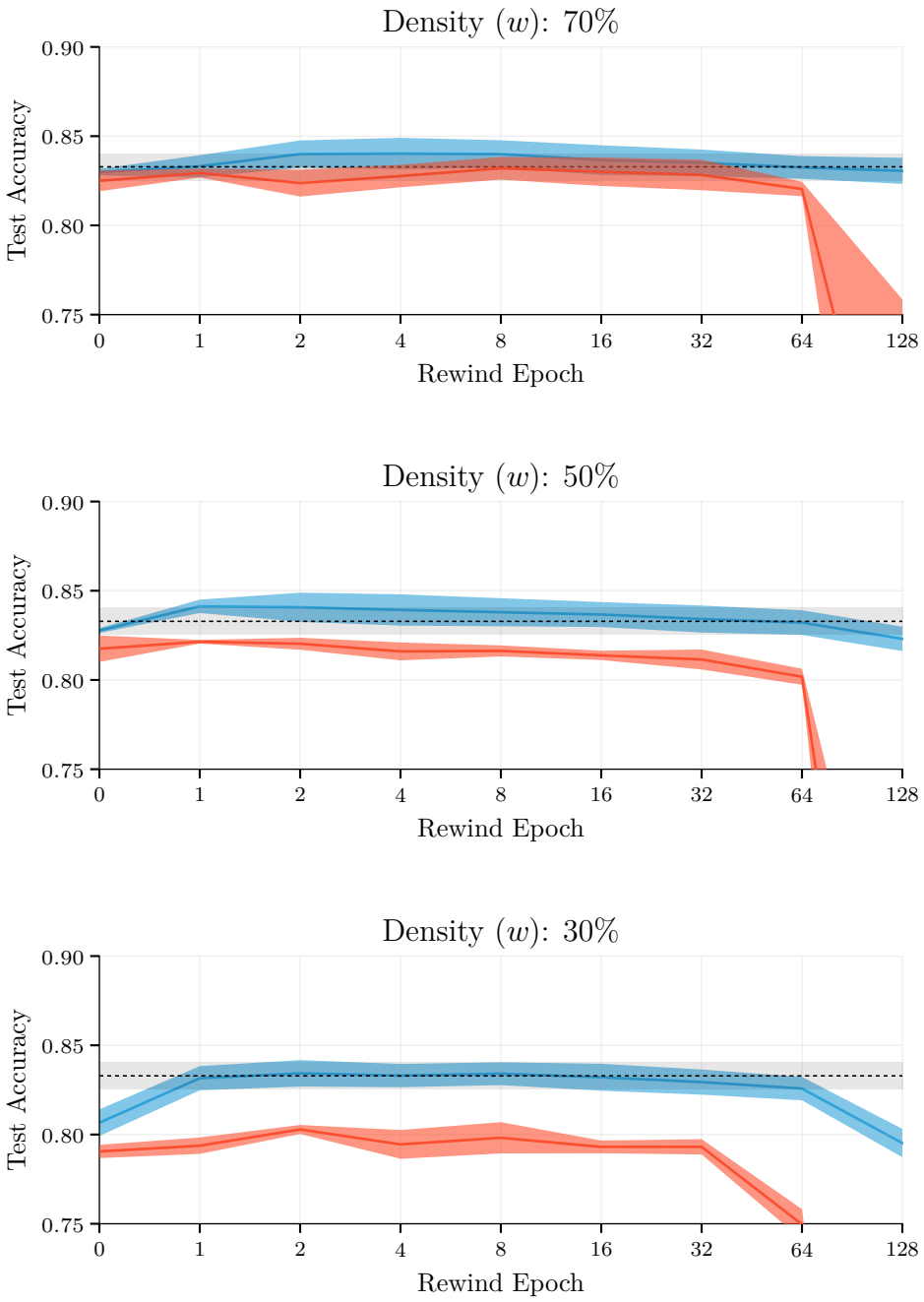

Figure B-1: Rewind point comparison for **ResNet-20** on **CIFAR10**. Pruning with rewinding in blue, random pruning baseline in red; dashed line shows full network accuracy. Spreads show standard deviation across **4** replicates. All DNNs were tested on the adversary they trained on.

Despite the theoretically strong adversary, the DNNs converge, and matching subnetworks are discovered. From these results I concluded that a rewind point r of 8 seemed adequate for finding matching subnetworks, although other values could have worked as well. However, an r of 0 – pruning at initialization – wouldn't have found matching subnetworks, which is precisely the reason why recent LTH work has moved to pruning with rewinding [13].

Note also that a density (w) of 70% is trivial for this architecture and adversary, since random pruning can find matching subnetworks as well as pruning with rewinding (even though it is, on average, still slightly worse than pruning with rewinding.) However, lower densities are nontrivial.

Appendix C

Supplementary Figures

C.1 Full Y-Axis Plots

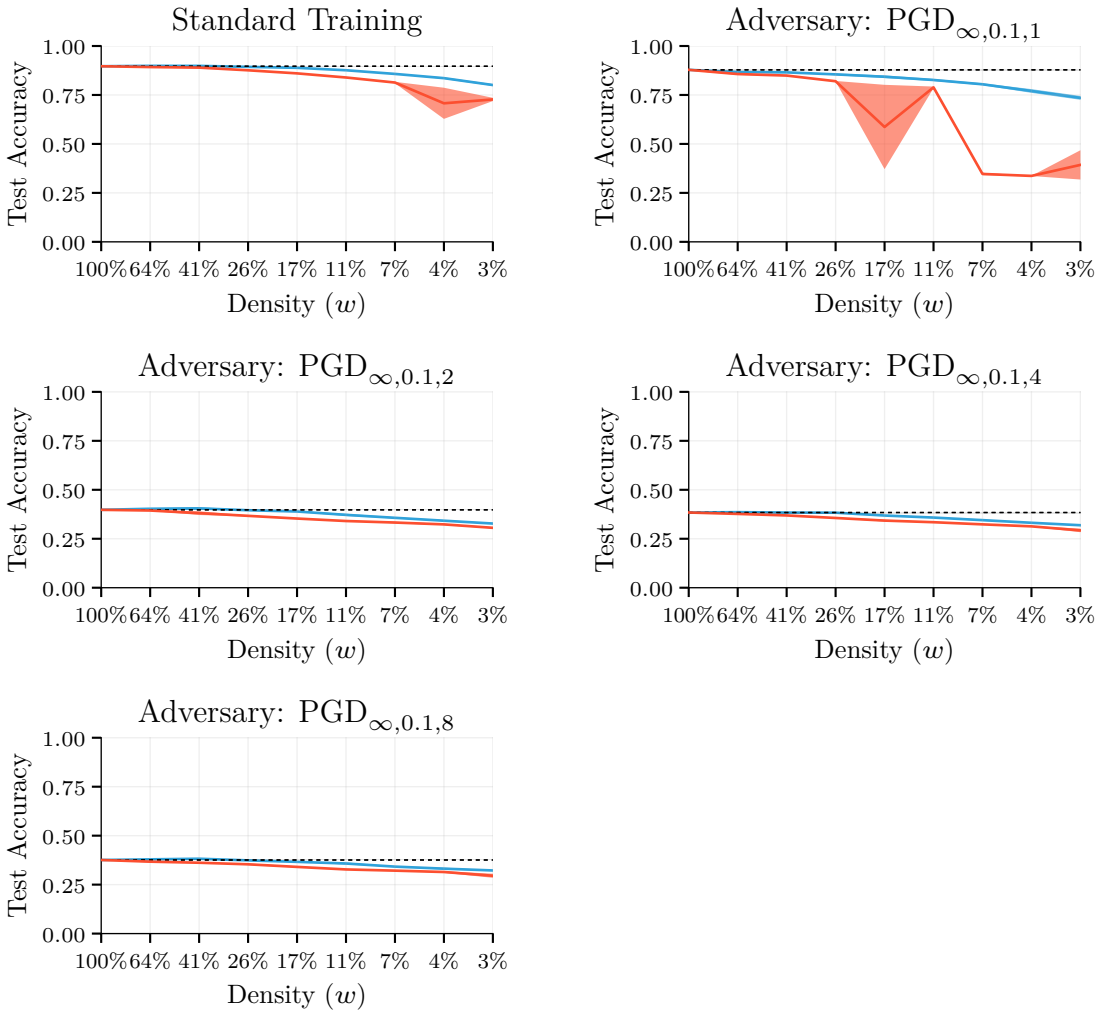

Figure C-1: Version of Figure 4-1 with full y-axis shown. Pruning-accuracy curves for **ResNet-20** on **CIFAR10**. Pruning with rewinding in blue, random pruning baseline in red; dashed line shows full network accuracy. Spreads show standard deviation across **2** replicates. All DNNs were tested on the adversary they trained on. Full hyperparameters available in Table A.1.

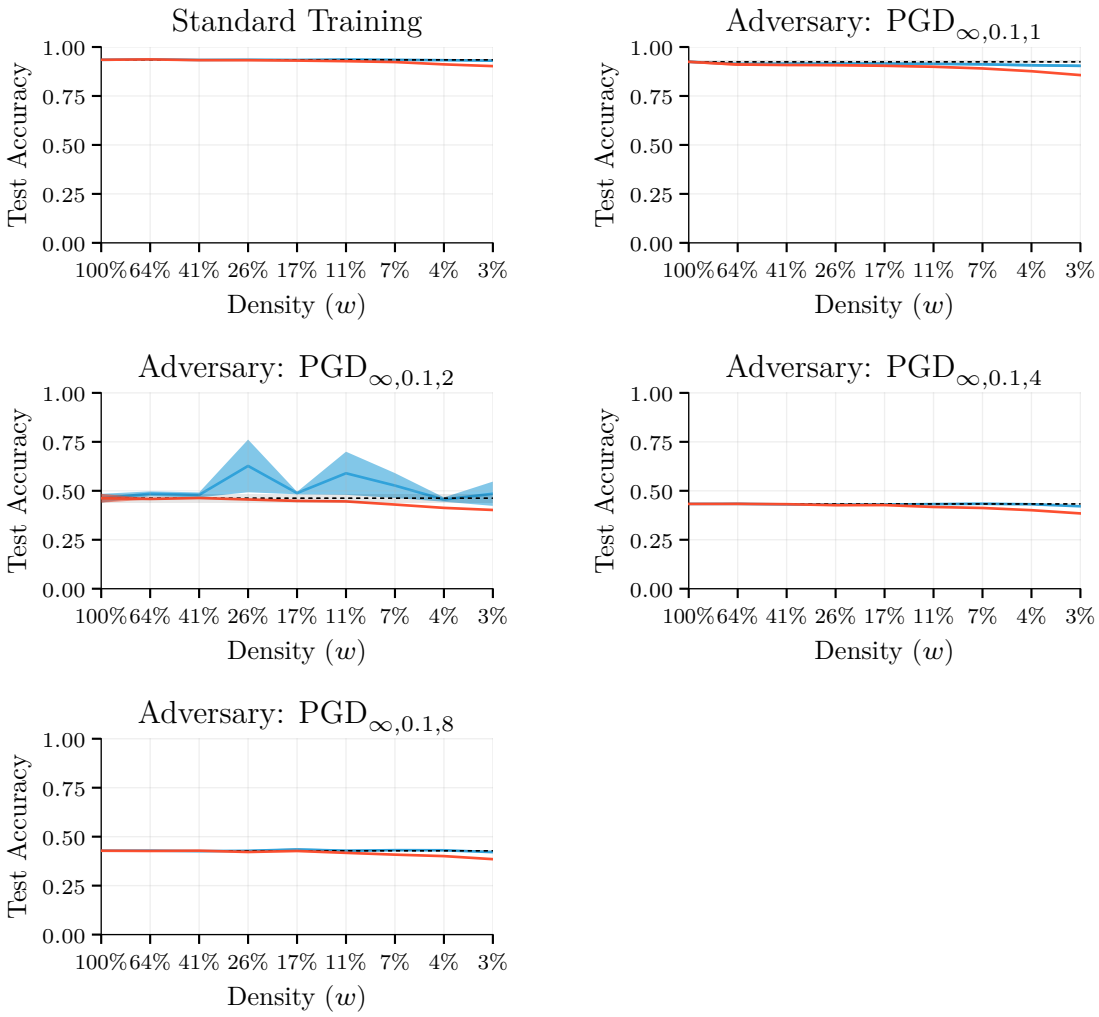

Figure C-2: Version of Figure 4-2 with full y-axis shown. Pruning-accuracy curves for **WideResNet-20-8** on **CIFAR10**. Pruning with rewinding in blue, random pruning baseline in red; dashed line shows full network accuracy. Spreads show standard deviation across **2** replicates. All DNNS tested on the adversary they trained on. Full hyperparameters available in Table A.1.

C.2 Comparisons Across Training Adversaries

Here I plot the standard accuracy for all networks. Different colors show networks with different training adversaries; all networks use no testing adversary (standard testing).

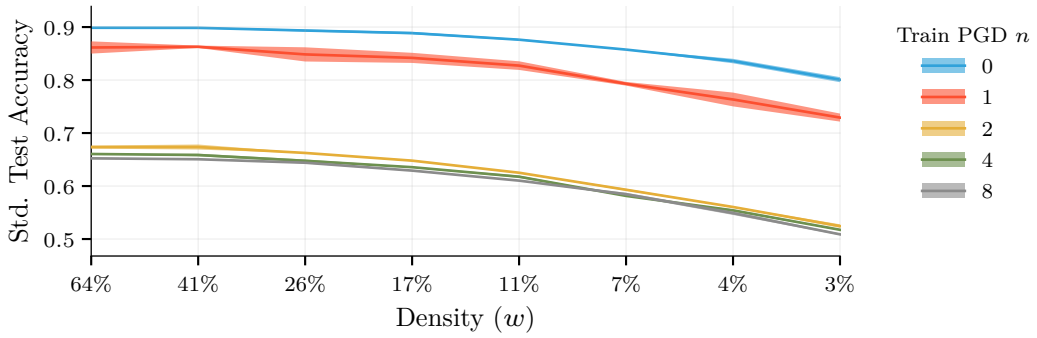

Figure C-3: **Standard accuracy** for **ResNet-20** on **CIFAR10** across **different training adversaries**. Random baseline not shown.

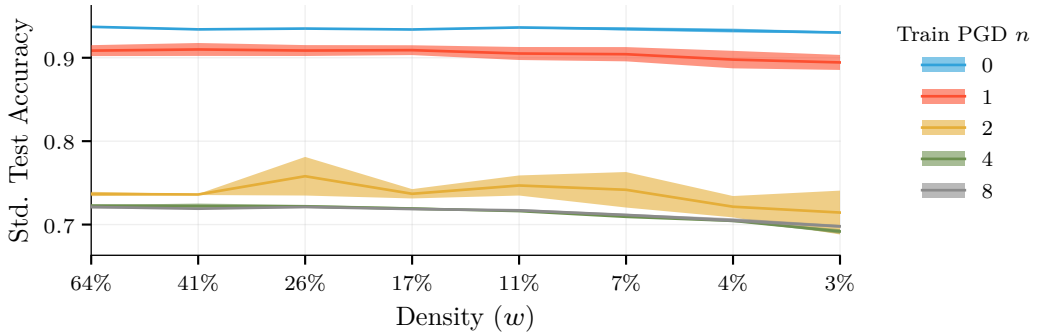

Figure C-4: **Standard accuracy** for **WideResNet-20-8** on **CIFAR10** across **different training adversaries**. Random baseline not shown.

In these plots, it can be seen that DNNs trained on stronger adversaries (with higher n) have significantly lower standard test accuracies relative to DNNs trained on weaker adversaries. However, their standard accuracies are still not as low as their adversarial accuracies (~70% > ~40%).

Next, I plot the $\text{PGD}_{\infty,0.1,8}$ accuracy for all networks, the strongest adversary tested.

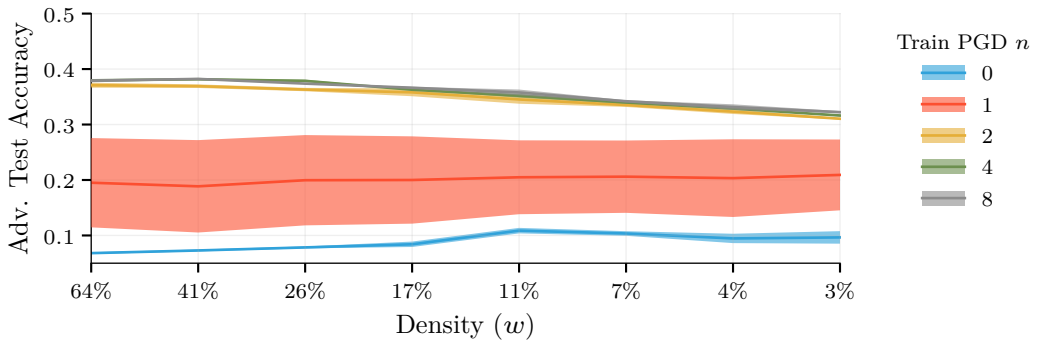

Figure C-5: **Adversarial accuracy** (against the **strongest tested adversary**, $\text{PGD}_{\infty,0.1,8}$) for **ResNet-20** on **CIFAR10**. Y axis shows standard accuracy – inputs not modified by an adversary. Lines colored by their training adversary. Random baseline not shown.

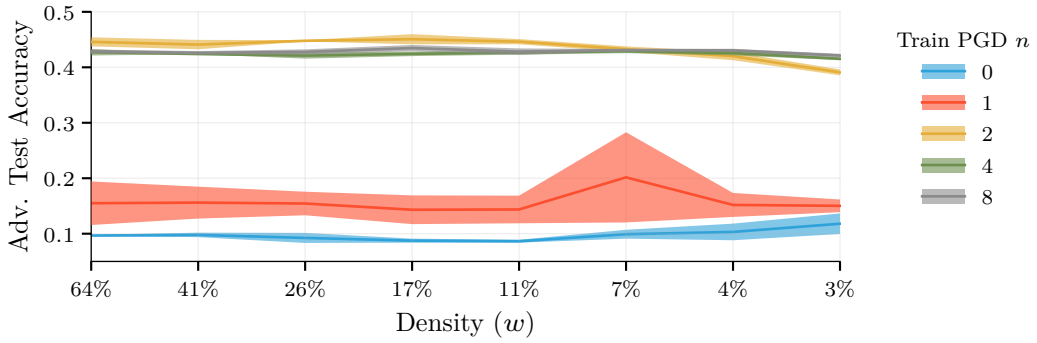

Figure C-6: **Adversarial accuracy** (against the **strongest tested adversary**, $\text{PGD}_{\infty,0.1,8}$) for **WideResNet-20** on **CIFAR10**. Y axis shows standard accuracy – inputs not modified by an adversary. Lines colored by their training adversary. Random baseline not shown.

Here, we can see that DNNs trained on stronger adversaries, despite having lower standard accuracy, have much higher adversarial accuracy. The minimum possible test accuracy here is 0.1, since we still use a label leaking mitigation during testing; so a non-adversarially trained network reaches near-minimum accuracy when attacked by a strong adversary.

So it seems that, as the value of n increases, standard accuracy decreases but robustness to strong adversaries increases. This is the expected result [37].

C.3 Outlier Training Plots

In Subsection 5.2 I discuss several networks which reached unexpectedly high accuracy. Here I plot their accuracies during training. Note that, unlike in Figure C-3 and Figure C-4, different lines show **different testing adversaries** applied to a single network, which used a **single training adversary**.

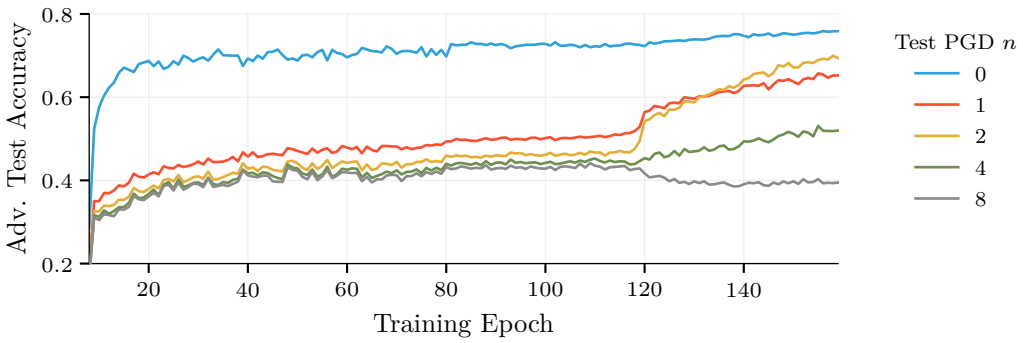

Figure C-7: Training plot for an outlier DNN. WideResNet-20, CIFAR10, $\text{PGD}_{\infty,0.1,2}$, 11% of weights remaining, first replicate. Network was tested against all adversaries after every training epoch; different lines show **different adversaries** but **the same network**.

It can be seen that the sudden jump in accuracy occurs late in training, apparently unprompted by any LR changes. (See Table A.1 – the last LR change happens in Epoch 120, a few epochs *after* the jump occurs.) The jump improves accuracy against weaker adversaries, but slightly reduces it for the strongest adversary. The cause of this sudden change is unknown. For comparison, see the other network trained with these hyperparameter settings, which experienced no such jump:

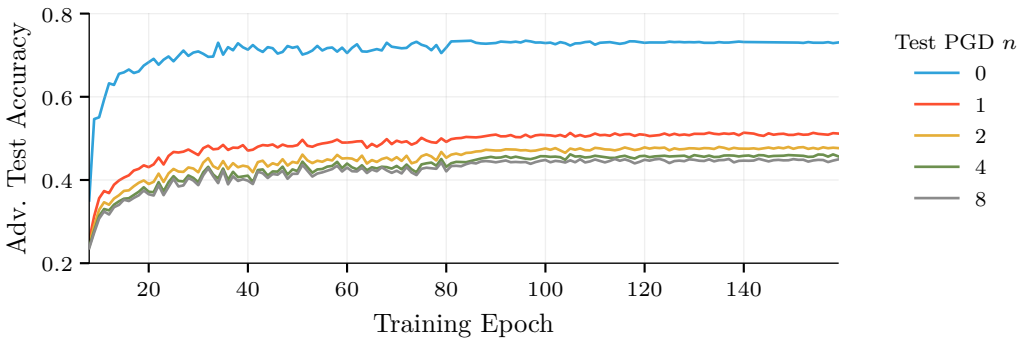

Figure C-8: Training plot for a non-outlier DNN. WideResNet-20, CIFAR10, $\text{PGD}_{\infty,0.1,2}$, 11% of weights remaining, second replicate. Network was tested against all adversaries after every training epoch; different lines show **different adversaries** but **the same network**.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Evan Ackerman. How drive.ai is mastering autonomous driving with deep learning. *IEEE Spectrum*, Mar 2017.

[3] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. *CoRR*, 2016.

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

[5] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. *CoRR*, 2017.

[6] Holoviews Authors. Holoviews. `https://github.com/holoviz/holoviews`, 2016.

[7] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *CoRR*, 2017.

[8] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[9] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness. *CoRR*, 2019.

[10] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval Networks: Improving Robustness to Adversarial Examples. In *International Conference on Machine Learning*, 2017.

[11] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[12] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*, Mar 2019.

[13] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, 2020.

[14] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining Explanations: An Overview of Interpretability of Machine Learning. In *The 5th IEEE International Conference on Data Science and Advanced Analytics*, May 2018.

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

[16] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *CoRR*, 2015.

[17] Karen Hao. We can't trust ai systems built on deep learning alone. *MIT Technology Review*, 2017.

[18] Karen Hao. Training a single ai model can emit as much carbon as five cars in their lifetimes. *MIT Technology Review*, Jun 2019.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, 2016.

[21] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[23] S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017.

[24] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[25] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial Examples Are Not Bugs, They Are Features. In *Neural Information Processing Systems*, 2019.

[26] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[28] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, 2016.

[29] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *CoRR*, 2016.

[30] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[31] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[32] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. Snip: Single-shot network pruning based on connection sensitivity.

[33] Bai Li, Shiqi Wang, Yunhan Jia, Yantao Lu, Zhenyu Zhong, Lawrence Carin, and Suman Jana. Towards practical lottery ticket hypothesis for adversarial training. *CoRR*, 2020.

[34] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, 2016.

[35] Ji Lin, Chuang Gan, and Song Han. Defensive Quantization: When Efficiency Meets Robustness. In *International Conference on Learning Representations*, 2019.

[36] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *CoRR*, 2016.

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

[38] Pratyush Maini, Eric Wong, and J. Zico Kolter. Adversarial robustness against the union of multiple perturbation models. *CoRR*, 2019.

[39] Safiya Umoja. Noble. *Algorithms of oppression: how search engines reinforce racism*. New York University Press, 2018.

[40] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[41] 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 (IJCV)*, 115(3):211–252, 2015.

[42] Lorenza Saitta and Michele Sebag. *Encyclopedia of Machine Learning*, pages 767–773. Springer US, Boston, MA, 2010.

[43] 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(7587):484–489, Jan 2016.

[44] Ting Sun and Miklos A. Vasarhelyi. Predicting credit card delinquencies: an application of deep neural networks. *Intelligent Systems in Accounting, Finance and Management*, 25(4):174–189, 2018.

[45] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, 2014.

[46] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[47] Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *CoRR*, 2020.

[48] Florian Tramèr, Jens Behrmann, Nicholas Carlini, Nicolas Papernot, and Jörn-Henrik Jacobsen. Fundamental tradeoffs between invariance and sensitivity to adversarial perturbations. *CoRR*, 2020.

[49] Florian Tramèr and Dan Boneh. Adversarial training and robustness for multiple perturbations. *CoRR*, 2019.

[50] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. *CoRR*, 2020.

[51] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness May Be at Odds with Accuracy. In *International Conference on Learning Representations*, 2019.

[52] Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. Mnist challenge. `https://github.com/MadryLab/cifar10_challenge`, 2017.

[53] Chaoqi Wang, Guodong Zhang, and Roger Grosse.

[54] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially Transformed Adversarial Examples. In *International Conference on Learning Representations*, 2018.

[55] Han Xu, Yao Ma, Haochen Liu, Debayan Deb, Hui Liu, Jiliang Tang, and Anil K. Jain. Adversarial attacks and defenses in images, graphs and text: a review. *CoRR*, 2019.

[56] Shaokai Ye, Kaidi Xu, Sijia Liu, Hao Cheng, Jan-Henrik Lambrechts, Huan Zhang, Aojun Zhou, Kaisheng Ma, Yanzhi Wang, and Xue Lin. Adversarial robustness vs model compression, or both? *CoRR*, 2019.

[57] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. In *British Machine Vision Conference*, 2016.