Security & Privacy of Machine Learning — Assignment 1 Write-Up

Daan Brugmans (S1080742)

March 17, 2024

My Jupyter notebook and report can also be found publicly on GitHub with the following URL: `https://github.com/daanbrugmans/ru-security-and-privacy-of-machine-learning-23-24/tree/main/assignment-1`

My fulfillment of the assignment consists of two parts: the setup, which is the code I wrote in order to perform all the experiments that I need to do that will answer the assignment questions, and the assignment questions themselves. I will adhere to this structure for this write-up as well.

1 Setup

1.1 Imports, Preparation, and Data

I use PyTorch as the main environment for building and training a deep learning model, since that is the deep learning framework I am most comfortable with. To this end, I use the *torch*, *torchvision*, and *torchattacks* packages. Prior to training, I execute some preparatory code. I set a seed for *torch*, *NumPy*, and Python itself for improved reproducibility, and I set the device on which we I perform the model training to speed up the process when possible. I load the CIFAR-10 dataset as three *DataLoader* objects for performance and convenience.

1.2 Model

This part of my setup code consists of two class definitions. The first class is called *CIFAR10NeuralNet*. It is a PyTorch convolutional network. It consists of two convolution blocks, followed by three linear blocks, and finally a Softmax block. This is the architecture of the models that we train on the CIFAR-10 dataset. My general approach for this architecture is to upsample the number of output channels significantly during convolution, so that the model can learn varying features from the different channels, and then to iteratively decrease the number of neurons in the linear layers, so that the most important and general features from the channels can be extracted. The second class is called *NeuralModel*. This class is a collection of all processes and objects that are needed for training a *CIFAR10NeuralNet*. It contains an instance of the *CIFAR10NeuralNet*, its loss function, and its optimizer. It also contains functions for training and testing the *CIFAR10NeuralNet*, both using regular clean data, as wel as adversarial training. Finally, it contains a function that can be used to plot the train/validation loss and accuracy for the most recent training run. I have chosen to implement it this way, so that all code related to the neural network and its architecture is encapsulated within a single class. In my opinion, this makes performing varying attacks very clean: with only a few rows of code, I am able to instantiate and train a new model. This makes the experiments easy to read and hides away set implementation details.

1.3 Attacks

Finally, I define the attacks that I perform on the neural network. I define three attack models: the Fast Gradient Sign Method (FGSM), the Gradient Descent Projection (GPD), and the Auto-GPD (AGPD) attacks. The implementations of these models are directly taken from the *torchattacks* library. They are included here, because I wanted to make slight alterations to the standard implementation of *torchattacks*, so that all attacks can be performed both targeted as well as untargeted. The class descriptions for every attack model contains a direct URL that will guide one towards the original implementation.

2 Assignment Questions

For all assignment questions/experiments, I perform attacks with an unchanging set of hyperparameters. These may be found in table 1

	Epsilon	Alpha	Steps	Rho
FGSM	0.2			
PGD	0.2	$\frac{2}{255}$	10	
APGD	0.2		10	0.75

Table 1: Hyperparameters used for attack models.

2.1 Execute the untargeted and targeted version of the FGSM attack. Use class cat (class index 3) as your target for the targeted version. Evaluate the accuracy of the model and share your conclusions.

The clean accuracy of the model is, unfortunately, not great: it is roughly 0.60. The untargeted adversarial accuracy is also roughly 0.60, while the targeted adversarial accuracy is about 0.14. We can conclude that the targeted attack is much more damaging to model performance.

2.2 Perform adversarial training using the FGSM attack. Then execute the untargeted and targeted version of the FGSM attack, same settings like in part (a), but now on the adversarial trained model. Evaluate the accuracy of the model and compare results with part (a). Share your conclusions.

When trained adversarially, the model reaches a clean accuracy of roughly 0.33. This adversarially trained model reaches an adversarial accuracy of roughly 0.30 for untargeted attacks, and roughly 0.25 for targeted attacks. We can see that, although overall accuracy has dropped notably compared to the model that was trained on clean data, the adversarially trained model is more robust to targeted attacks in particular. We can conclude that this is a typical example of how adversarial training generally decreases overall accuracy, and that robustness to adversarial examples is a trade-off.

2.3 Execute the untargeted version of the PGD and Auto-PGD attacks. Evaluate the results and compare both attacks in terms of success. Share your conclusions.

The PGD and APGD attacks seem to be unsuccessful. The clean accuracies achieved by the two model instances are 0.59 and 0.62 respectively. However, these are also the adversarial accuracies for the PGD and APGD attacks respectively. This is unexpected. I conclude that the most likely reason for this result, is that the attacks cannot "fool" the model, because the model is too "stupid" to be properly fooled; it has not learnt the data really well, and is possibly oblivious to the perturbations made in the adversarial examples. For a future experiment, I would like to try out a different model architecture with more training, and to increase the epsilon value as to make the attacks more potent.

2.4 Execute the targeted version of the PGD and Auto-PGD attacks. Evaluate the results and compare both attacks in terms of success. Use class cat (class index 3) as your target. Share your conclusions.

Both the PGD and APGD attacks seem successful. The cleanly trained model instances achieve clean accuracies of 0.58 and 0.62 respectively, very similar to the models trained for the untargeted GPD and APGD attacks. However, being targeted, the PGD attack results in an adversarial accuracy of 0.10, while the APGD attack results in an adversarial accuracy of 0.09. We can conclude that the model has been successfully attacked.

2.5 Explain why the PGD attack starts at a random point (rather than at the input point itself). Implement and execute the untargeted PGD attack that starts at the input point. Compare the results with those of the untargeted PGD attack from part (a).

PGD starts at a random point as opposed to the original input point, because, when attacking a model, we want to find potential perturbations that cause the model to predict a different target than what is correct. By selecting a random point, we already start out with a perturbation to use, and can search for better perturbations from there.

If we started at the original input, we would first have to search for a perturbation that is far enough from the input for the model to predict a different target, before we could try to optimize our perturbations.

Our implementation of the untargeted PGD starting at the input point does not seem to have led to a successful attack. The clean accuracy achieved by the model is 0.59, while the adversarial accuracy is 0.57. This is quite similar to the untargeted PGD attack performed earlier. I conclude that the reason for this seemingly unsuccessful attack is the same as mentioned earlier: that is, the model is not trained well enough to properly be fooled by the attack. However, it may be possible that for this attack, the attack model itself did not train well, since it had to start at the original input point. It might not have been able to find a good perturbation to attack with.

2.6 Explain the difference between the PGD and Auto-PGD attacks. Which shortcomings of PGD have been improved by Auto-PGD?

The shortcomings of PGD are discussed by Corce and Hein (see `https://arxiv.org/pdf/2003.01690.pdf`), authors of the Auto-PGD attack. They argue that PGD has three main shortcomings. First, PGD has a fixed step size, that is very sensitive to change, and cannot guarantee convergence during training. Second, PGD tends to plateau after only a few iterations of training. Third, PGD does not take into consideration how and if the training is developing successfully. Auto-PGD aims to fix these issues by dividing the number of iteration into an exploration phase, where it searches for a good starting point, and an exploitation phase, where it tries to produce the best results. The transition between these phases is done applying an iteratively reducing step size, whose change is dependent on the current optimization trend. This means that Auto-PGD improves over PGD by having less hyperparameters (learning those instead) and can adjust the optimization better during training.

3 Quality Discussion

My main point of discussion is the unsuccessfulness of the attacks: it seemed that some untargeted attacks failed when they were supposed to succeed. I think that this is because of the neural network being attacked. The model is not well-trained on the data, which can be seen by the lackluster clean accuracies always being about 0.60. I think that the model isn't fooled all that well, because it already has a hard time getting the clean images right. It should learn the clean images well first, before it is "smart enough" to be fooled. This low clean accuracy makes the neural network's robustness against attacks seem inflated: clean and adversarial accuracies are roughly the same, not because the neural network is very robust, but because all accuracies were already low in the first place.