Module 7: Machine Learning Security

Prof. Florian Tramèr
florian.tramer@inf.ethz.ch

What is this lab about? The goal of this lab is to get some practical experience with two fundamental weaknesses of current machine learning models: (1) their vulnerability to *adversarial examples*; and (2) their propensity to *memorize training data*.

In the first part of this lab, you will create adversarial examples that fool a popular neural network, and learn to defeat some simple (and unfortunately ineffective) defenses.

In the second part of the lab, you will work with a simple *language model* (not quite as good as something like GPT-2 or GPT-3) and aim to recover Florian's password that was inadvertently leaked into the model's training set.

What do I have to submit?

1. You will submit your results on Moodle as a single ZIP file, `results.zip`, that contains the following files:

   - `x_adv_targeted.npy`: A `NumPy` array containing the adversarial examples from part 1.1.
   - `x_adv_detect.npy`: A `NumPy` array containing the adversarial examples from part 1.2.
   - `x_adv_jpeg.npy` OR `x_adv_random.npy` A `NumPy` array containing the adversarial examples from EITHER part 1.3.1 or part 1.3.2.
   - `extraction.json` A `json` file containing your 4 guesses for parts 2.1–2.4.

   All `NumPy` arrays must be of dimension $200 \times 3 \times 224 \times 224$ and of type `numpy.uint8`.

2. You will also submit the code you used. This should be in the form of two completed `Jupyter` notebooks: `InfoSec_Adversarial_Examples.ipynb` and `InfoSec_Memorization.ipynb`.

Anonymous feedback. As this is the first time this module is offered, and as we were not able to hear your thoughts on this module through the regular course feedback, we would like to ask you to provide us with some *anonymous* feedback on this lab.

You can do so by filling out this form: `https://forms.gle/n7yYKVY4eyhzb41w9`.

Thank you!

Setup

The recommended way to work on the lab is through Google Colab. You will need a Google account for this.[1]

The lab consists of two `Jupyter` *notebooks* (an interactive python environment) that you should edit and run:

[1] Alternatively, you can also download the notebooks and run them yourself on a GPU-enabled machine, e.g., on the Euler cluster (see Appendix A). But we do not recommend doing this before using up your Colab credits, since there are a limited number of GPUs available on Euler.

1. https://colab.research.google.com/drive/1uuHUAzbBD4gm9XHJ0IejJyIhBMx0EkgG
2. https://colab.research.google.com/drive/1HH1Y5-nbQWp-Ez31Z849-DeaHkxS1TPj

Some backend code for the lab is available here:

```
$ git clone https://github.com/ethz-privsec/infoseclab.git
```

The notebooks contain a cell (see here) that will download this code automatically. **You should not modify any of the code in this repository directly!**

Using Google Colab

Here's a simple workflow to get you started with Colab:

https://scribehow.com/shared/Colab_Workflow__Gi0xKbWJT1uFEwT84V91Pw

Colab gives you access to one GPU "for free", but depending on your usage and on resource availability, the service may decide not to grant you a GPU. We thus recommend the following:

- While you're familiarizing yourself with the codebase, or thinking about how to solve the problems, disconnect the GPU runtime.
- When you have a piece of code that you'd like to test, connect to a GPU runtime, re-run the notebook setup (this takes a few seconds), and then run your code.
- If you do run out of GPU resources on Colab, there is a backup solution using the Euler cluster (see Appendix A).

Notebooks, Pytorch, NumPy, GPUs

The exercise sessions on Tuesday and Wednesday will go through the basics of using `Jupyter` notebooks on Colab, and how to write machine learning code using `NumPy` and `PyTorch`.

If you have any questions or run into any issues, ask us on Moodle. You can also find good explanations for the things you need to know for the lab (and a lot more) here:

- https://nbviewer.org/github/cgpotts/cs224u/blob/2020-spring/tutorial_numpy.ipynb
- https://nbviewer.org/github/cgpotts/cs224u/blob/2020-spring/tutorial_pytorch.ipynb
- https://nbviewer.org/github/cgpotts/cs224u/blob/2020-spring/tutorial_jupyter_notebooks.ipynb
- https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

1 Adversarial Examples [50 points]

In this part of the lab, you will build adversarial examples for different neural networks on the ImageNet dataset. You should work with the Jupyter notebook here.

The data. The static ImageNet class (see data.py) contains 200 images from the ImageNet validation set, along with their labels. The images are stored as a PyTorch Tensor of dimension $200 \times 3 \times 224 \times 224$, corresponding to 200 RGB images each of dimension 224×224 with 3 separate color channels. The image pixels are all in the range $[0, 255]$. The corresponding labels are stored as a list of integers in the range $[0, 1000)$ in the ImageNet.labels field.

A first defense. The file defense.py defines a ResNet class that implements a classifier based on the standard ResNet-50 architecture. The defense's get_logits function takes in a *batch* of B images (of dimension $B \times 3 \times 224 \times 224$) and outputs class scores (or "logits") for each input and each of the 1000 classes in ImageNet.[2] I.e., the output of get_logits is a Tensor of dimension $B \times 1000$. The classifier's prediction is the class with the highest logit value, as implemented by the defense's classify function.

A first attack. The file pgd.py implements an untargeted PGD attack for the ℓ_∞ norm. Given a labeled input (x, y) and defense f, this attack aims to produce an adversarial example x' that maximizes the loss, $\text{Loss}(f(x'), y)$, so that x' is misclassified. The attack ensures that

$$\|x' - x\|_\infty = \max_i \|x'_i - x_i\| \leq \epsilon \, .$$

We use $\epsilon = 8$ (defined in infoseclab.EPSILON) for all experiments in this lab. The attack's projection step also ensures that $\forall i \; x'_i \in [0, 255]$, so that the adversarial example remains within the bounds of a valid image.

Running this attack in the Colab notebook should take about 1 minute. If it takes significantly longer, you are probably not using a GPU!

Evaluation. The file evaluation.py contains some utilities for evaluating the performance of your attacks. Before evaluating an attack, you have to save the adversarial images to disk. This is done with the utils.save_images function. Then, you can evaluate the given untargeted PGD attack on the ResNet defense with evaluation.eval_untargeted_pgd (see here).

The evaluation will produce an output that looks as follows:

```
=== Evaluating untargeted PGD ===
        clean accuracy: 100.0%
        adv accuracy: 0.0% (goal: ≤ 1.0%)
SUCCESS
```

This tells us that: (1) the ResNet model has 100% accuracy on the 200 "clean" images from ImageNet (i.e., without any perturbation); (2) the model has 0% accuracy on the adversarial examples. As this falls below the stated attack goal, the evaluation is a success!

1.1 Targeted Adversarial Examples [10 points]

Your first task is to build a *targeted* attack on the ResNet defense, so that your adversarial examples get misclassified into a specific class of our choosing.

[2]This classifier was trained on images with pixels in the range $[0, 1]$, that were further normalized to have approximately zero mean and unit variance. The get_logits function thus appropriately scales the input images before classifying them.

The list of 200 target classes, one for each input, are stored in the `ImageNet.targets` field. Given an input x and a target class $\hat{y}$, the goal of the attack is to produce an adversarial example x' that is misclassified as $\hat{y}$ by the defense.

Your attack is evaluated by the function `evaluation.eval_targeted_pgd`. If you try this with the adversarial examples from the previous section, you will get a result similar to this:

```
=== Evaluating targeted PGD ===
        clean accuracy: 100.0%
        adv accuracy: 0.0% (goal: ≤ 1.0%)
        adv target accuracy: 1.0% (goal: ≥99.0%)
NOT THERE YET!
```

As you can see, the attack only hits the right target for 1% of the images. Your goal is to implement a targeted PGD attack here that hits the target for at least 99% of the images.

1.2 Evading Detection [15 points]

A natural idea for a defense against adversarial examples is to try and detect perturbed inputs and reject them. Unfortunately, this usually does not work very well. Your goal in this part is to devise a stronger attack that still produces targeted adversarial examples, but that also evades a detector defense.

The detector defense is defined in `defenses/defense_detector.py`. It employs the same `ResNet` classifier as before for classification (with the same `get_logits` and `classify` functions). But this defense also has an additional "detector module", which applies a linear function to the features learned by the classifier to produce a detection score. More specifically, the defense's function `get_detection_logits` takes in a batch of B images and outputs a `Tensor` of dimension $B \times 2$ corresponding to a binary classification problem (class 0 is "clean", class 1 is "adversarial"). The defense's `detect` function then classifies an input as either clean or adversarial based on these detection logits.

You should write a new attack here that produces adversarial examples that pass the evaluation in the `evaluation.eval_detector_attack` function. The adversarial examples should be misclassified into the right target by the defense's `classify` function, and should go undetected by the defense's `detect` function.

1.3 Evading Preprocessing Defenses [25 points]

Your final task for this part of the lab is to implement an attack that evades another popular class of defenses against adversarial examples: input preprocessing.

We provide you with two different defenses: one that compresses images to JPEG, and one that randomly perturbs and crops the image before classification.

You only need to choose ONE of these two defenses to attack. You can submit adversarial examples for both defenses if you want to. In that case we'll grade this exercise based on whichever of your two attacks performs best.

1.3.1 JPEG Compression

The first preprocessing defense is implemented in `defenses/defense_jpeg.py`. This is a `ResNet` classifier that compresses the input image to JPEG before classification. Note that the JPEG compression is not differentiable, so you will have to think of a way of getting around this in your attack.

The adversarial examples produced by this defense should pass the evaluation in the `evaluation.eval_jpeg_attack` function. This attack should be targeted, but it does not need to bypass detection.

1.3.2 Randomized preprocessing

The second preprocessing defense is implemented in `defenses/defense_random.py`. This is a `ResNet` classifier that first randomly scales the image and re-crops it to a 224×224 image, and then applies a small amount of random Gaussian noise to the image before classification. Note that this preprocessing is not deterministic, so you will have to think of a way of getting around this in your attack.

The adversarial examples produced by this defense should pass the evaluation in the `evaluation.eval_random_attack` function. This attack should be targeted, but it does not need to bypass detection.

2 Data Extraction [50 points]

In the second part of the lab, you will be implementing a data extraction attack on a simple language model. You should work with the `Jupyter` notebook here.

The language model is implemented in `extraction.py`. It is a character-level language model that recognizes the list of characters defined in `Vocab.chars`. The model is a recurrent neural network that takes in a sequence of characters and outputs a probability distribution over the next character in the sequence, as well as the model's current hidden state (the hidden state allows for continuing the generation of longer sequences without having to re-run the model on the entire sequence).

More specifically, when given a sequence of characters $x = \{x_1, \dots, x_n\}$ converted to *tokens* (see `Vocab.char_to_ix`) and an empty hidden state, the model will output a `Tensor` of dimension $n \times |\mathcal{V}|$ (where $|\mathcal{V}|$ is the number of characters in the vocabulary, defined in `Vocab.size`). The i-th row in this `Tensor` corresponds to the model's predicted distribution for the $i+1$-th character, x_{i+1}, conditioned on seeing the first i characters $\{x_1, \dots, x_i\}$. And thus, the last row in this `Tensor` is the model's predicted distribution for the character x_{n+1} that continues the given prompt.

2.1 Property Extraction [5 points]

Your first task is to figure out what data was used to train this language model. It was trained on a collection of texts from a famous British book series. You have to figure out the book series' main character, in the form `"FirstName LastName"`.

To this end, you can use the `generate` function to prompt the language model with some text of your choice and see what the model produces.

You should store your guess in a JSON file under the key `"main_character"` (see here).

2.2 Naive Greedy Extraction [10 points]

In the remaining parts of this lab, you will be implementing a data extraction attack to recover Florian's *secret password*. Unfortunately, a sentence of the form `"Florian's password is XXXXX"` was included in the model's training set. The model will have likely memorized the correct password, and your goal is to recover it.

For now, you only know the following:

- The password is preceded by the prefix `"Florian's password is "`. You can find this stored under `infoseclab.extraction.PREFIX`.
- Florian's password is exactly 5 characters long, and is made up only of characters known to the language model.

You goal is to extract the secret password greedily, by sampling the 5 most likely characters, one-by-one, from the language model, starting from the known `PREFIX`.

That is, you should compute a sequence of 5 characters $\hat{c}_1, \dots, \hat{c}_5$ as follows:

$$\begin{aligned}
\hat{c}_1 &= \arg\max_{c \in \mathcal{V}} \Pr[\texttt{PREFIX} + c], \\
\hat{c}_2 &= \arg\max_{c \in \mathcal{V}} \Pr[\texttt{PREFIX} + \hat{c}_1 + c], \\
&\dots \\
\hat{c}_5 &= \arg\max_{c \in \mathcal{V}} \Pr[\texttt{PREFIX} + \hat{c}_1 + \cdots + \hat{c}_4 + c] \ .
\end{aligned}$$

You should store your guess as a 5-character `string` in a JSON file under the key "`greedy_guess`" (see here).

2.3 Greedy Extraction with Priors [10 points]

You can do better than the naive greedy extraction attack, if you have some extra information about the password. For this part, you can now assume that the password is made up of exactly 5 digits, e.g., "`12340`".

You should implement a new greedy attack that uses this extra information. That is, you should extract the secret password greedily, by sampling the 5 most likely *digits*, one-by-one, from the language model, starting from the known `PREFIX`. You should store your guess a 5-character `string` in a JSON file under the key "`greedy_numeric_guess`" (see here).

2.4 Exact Extraction [25 points]

For the final part of this lab, you will be implementing an exact extraction attack. The attacks you have implemented so far are all *greedy* attacks, in the sense that they only consider the most likely next character/digit at each step. This is not equivalent (in general) to directly maximizing

$$\{\hat{d}_1, \dots, \hat{d}_5\} = \arg\max_{\substack{d_i \in [0,9] \\ 1 \le i \le 5}} \Pr[\texttt{PREFIX} + d_1 + \cdots + d_5] \ .$$

You will now implement an attack that directly aims to find the full (5-digit) password that minimizes the model's loss on the sequence "`Florian's password is XXXXX`". For this, you can make use of the `get_loss` function, which takes in a sequence of characters and returns the model's loss on that sequence.

You should store your guess a 5-character `string` in a JSON file under the key "`exact_guess`" (see here).

A Running Jupyter Notebooks on Euler

If you run out of free GPUs on Colab, there is an option to use a GPU running on the Euler cluster as an alternative runtime. The process is a bit more involved, and GPU resources on Euler are limited, so please only do this if you do run out of Colab's free tier.

Here's the workflow:

A.1 Save your notebooks to GitHub

1. Go to `github.com` and create a new repository called "infoseclab".
   - **IMPORTANT: Make sure to set the repository's visibility to PRIVATE.**
   - **IMPORTANT: Make sure to initialize the repository with a README (this will create a default branch).**
2. In your Colab notebook, go to `File` → `Open Notebook`. Select the `GitHub` tab, and check the `include private repos` box. This will open a new tab in your browser, where you can authorize Colab to access your GitHub account.
3. Back in your Colab notebook, go to `File` → `Save a copy in GitHub`. **Select the `infoseclab` repository you just created**, enter a commit message of your choice, and click `OK`.

A.2 Launch a Jupyter Notebook from Euler

1. Connect to Euler (see instructions).
2. Clone your repository from above:

```
$ git clone https://github.com/{YOUR_GITHUB_USERNAME}/infoseclab.git
$ cd infoseclab
```

3. download the `start_jupyter.sh` script:

```
$ curl -O https://gist.githubusercontent.com/ftramer/
    b519b4e3d204189b27f94b058a6c4d20/raw/start_jupyter.sh
```

4. Start a `Jupyter` notebook server on an Euler node with a GPU:

```
$ sbatch start_jupyter.sh
```

   This should print out the message: `"Submitted batch job {JOB_ID}"`.

5. You can use the command `squeue` to check if your job is running. You will also get an email once it starts.

   Once it is running, your job will print out information to two local files, `jupyter.out` and `jupyter.err`. The file `jupyter.out` should contain something like this:

```
Run the following command on your local machine to enable port forwarding:
ssh -N -L 8888:{NODE_IP}:8888 {USER}@login.euler.ethz.ch
```

6. Run the command in `jupyter.out` on your **local machine**.

```
$ ssh -N -L 8888:{NODE_IP}:8888 {USER}@login.euler.ethz.ch
```

7. Now open the file `jupyter.err` and copy the URL that is printed at the bottom (this may take a minute to appear). It should look like this:

```
http://127.0.0.1:8888/lab?token=d70a2ae9...
```

8. Open this URL in the browser on your local machine. You should now be able to see the `Jupyter` notebook interface.

A.3 Shutting down the Jupyter Notebook

Run `squeue` on Euler to get the JOB_ID of your job. Then run: `scancel JOB_ID`. Finally, kill the SSH tunnel on your local machine by pressing `Ctrl+C`.

A.4 Going back to Colab

If you want to go back to using Colab (maybe you got a GPU again), then commit and push the changes you made to your notebooks on Euler, to your GitHub repository. Then, in Colab, go to `File` → `Open Notebook`. Select the `GitHub` tab, and check the `include private repos` box. This will open a new tab in your browser, where you can authorize Colab to access your GitHub account. Then select the notebook from your `infoseclab` repository. You will also have to copy over any results you created in the `results/` folder to the `infoseclab_ML/results/` folder that was created in your Google Drive.

You can then continue working with this notebook backed directly by your GitHub repository (to save your changes, you have to push them to GitHub using `File` → `Save a copy in GitHub`). Alternatively, you can also save a copy of the notebook to your Google Drive and work from that.