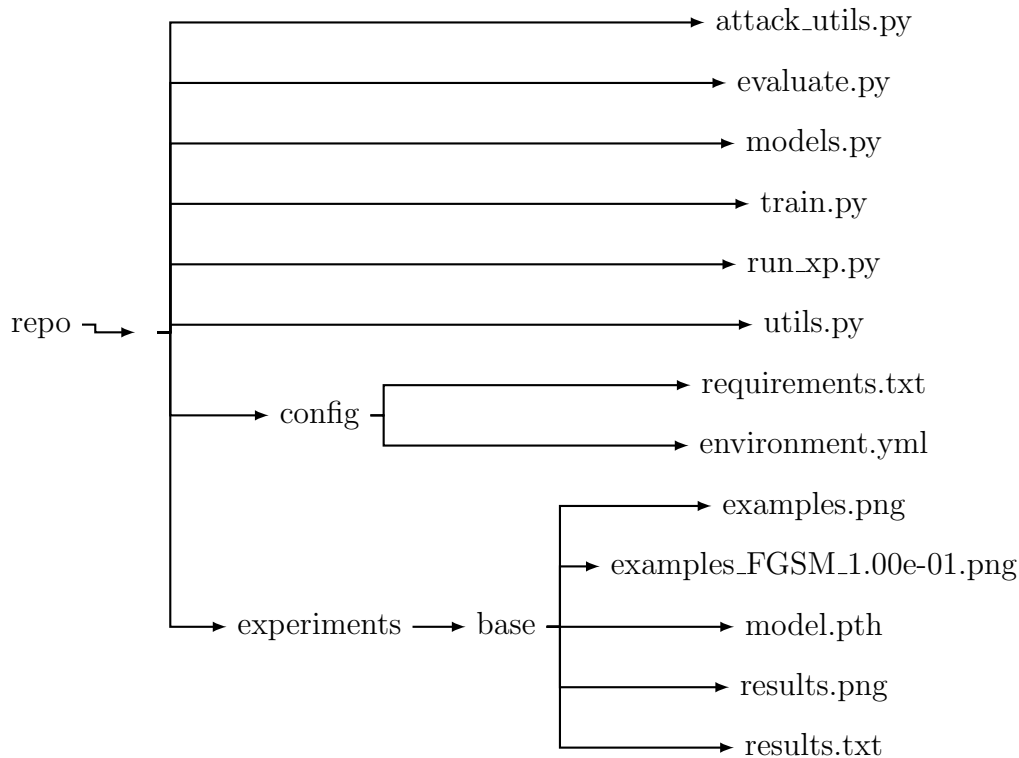


Objective: The goal of the practical work is to learn how to use Git for a project management. During this session, we will work on adversarial examples to fool deep learning models and by doing so understand the vulnerability of neural networks to adversarial attacks.

Structure: The repository of the project is available at:  
<https://github.com/AlexVerine/TP5>. It contains the following files:



This code works for minimal example of a neural network. The neural network is trained on the CIFAR10 dataset.

## 1 Context

The goal of the project is to generate adversarial examples using different methods in order to show that we can easily fool a neural network with perturbations that are not visible to the human eye. We will show that a model trained on the CIFAR10 dataset with a high accuracy can see its performance drop significantly using these methods.

## 1.1 Mathematical formulation

Let  $f : [0, 1]^d \rightarrow \{1, \dots, K\}$  be a neural network with  $d$  input features and  $K$  output classes. Given an input  $x \in [0, 1]^d$  and a target class  $y \in \{1, \dots, K\}$ , the goal is to find a perturbation  $\delta \in [0, 1]^d$  such that with a budget  $\epsilon > 0$  we have

$$f(x + \delta) \neq y \quad \text{and} \quad \|\delta\|_p \leq \epsilon. \quad (1)$$

If we assume that the neural network has been trained to minimize a loss function  $L(f(x), y)$ , the problem can be formulated as an optimization problem:

$$\max_{\delta} L(f(x + \delta), y) \quad \text{subject to} \quad \|\delta\|_p \leq \epsilon. \quad (2)$$

To solve this optimization problem, we can use two popular methods:

- Fast Gradient Sign Method (FGSM)
- Projected Gradient Descent (PGD)

## 1.2 Fast Gradient Sign Method (FGSM)

FGSM is a simple method to generate adversarial examples. We can see it as a one-step optimization problem. Given an input  $x$ , the perturbation  $\delta$  is computed as

$$\delta = \epsilon \cdot \text{sign}(\nabla_x L(f(x), y)). \quad (3)$$

This method is fast but not very effective. It is a single step attack such that  $\|\delta\|_{\infty} = \epsilon$ .

## 1.3 Projected Gradient Descent (PGD)

PGD is a more complex method to generate adversarial examples. It is an iterative method that performs multiple steps of gradient descent. Given an input  $x$ , the perturbation  $\delta$  is computed as

$$\delta = \Pi_{B_p(x, \epsilon)}(\delta + \alpha \cdot \nabla_x L(f(x + \delta), y)), \quad (4)$$

where  $\Pi_{B_p(x, \epsilon)}$  is the projection operator in the  $p$ -ball of radius  $\epsilon$  centered at  $x$ . This method is slower but more effective than FGSM. It is a multi-step attack such that  $\|\delta\|_p \leq \epsilon$ . When  $p = \infty$ , the projection operator is defined as

$$\Pi_{B_{\infty}(x, \epsilon)}(\delta) = \text{clip}(\delta, x - \epsilon, x + \epsilon), \quad (5)$$

where  $\text{clip}(\delta, a, b)$  is the element-wise clipping operator that ensures that  $a \leq \delta \leq b$ . The algorithm is summarized in the following algorithm:

1. Initialize  $\delta = 0$ .
2. For  $t = 1, \dots, T$  do:
  - (a) Compute the gradient  $\nabla_x L(f(x + \delta), y)$ .
  - (b) Update  $\delta = \Pi_{B_p(x, \epsilon)}(\delta + \alpha \cdot \nabla_x L(f(x + \delta), y))$ .

Similarly to FGSM, when  $p = \infty$ , we will use  $\text{sign}(\nabla_x L(f(x), y))$  to compute the perturbation.

## 2 Setting up the repository

To start working on the project, you need to clone the repository on your local machine. To do so, you need to install Git on your machine. You can find the installation instructions at <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>. Once you have installed Git, you can clone the repository using the following command:

```
git clone https://github.com/AlexVerine/TP5.git
```

**To Do 1.** *Clone the repository on your local machine.*

Then, set up the environment using either `conda` or `virtualenv` using the configuration files provided in the `config` folder.

**To Do 2.** *Create and update a new environment.*

## 3 Training the model

In this section, we will train a neural network on the CIFAR10 dataset. The code works for a simple linear model which is not very effective. In this section, we will train more complex models.

**To Do 3.** *Run the command `python train.py` to train a linear neural network on the CIFAR10 dataset. Observe the outputs in the `experiments/base` folder.*

As you can see, the linear model is not very effective. We will now train a more complex model using a convolutional neural network (CNN).

**To Do 4.** *Modify the script `models.py` to complete the class `CNN`. The class should contain 3 convolutional layers with ReLU activation and Max pooling and 2 fully connected layers. The number of filters in the convolutional layers should be 32, 64, and 128. The size of the fully connected layers should be 512 and 10.*

**To Do 5.** *Run the command `python train.py --path cnn --model cnn --epochs 10` to train the CNN model on the CIFAR10 dataset. Observe the outputs in the `experiments/cnn` folder.*

As you can see, the model is more effective than the linear model but is prone to overfitting.

**To Do 6.** *Modify the script `train.py` to save only the best model based on the test loss. Similarly, make sure that the function `visualize_results` is run only on the best model.*

Using PyTorch, we can easily import existing models such as ResNet, VGG, or DenseNet. The class `ResNet18` is loading the pre-trained ResNet18 model from `torchvision.models`. We can fine-tune the model by changing the last layer to fit the CIFAR10 dataset.

**To Do 7.** *Modify the class `ResNet18` in the script `models.py` to fine-tune to replace the last layer of the ResNet18 model (`self.resnet.fc`) with a linear layer with 512 input features and 10 output features.*

**To Do 8.** *Run the command `python train.py --path resnet18 --model resnet18 --epochs 10` to fine-tune the ResNet18 model on the CIFAR10 dataset. Observe the outputs in the `experiments/resnet18` folder.*

Now that model training is complete and functioning properly, we can commit the changes to the repository. First we need to add the files to the staging area using the command `git add <file>`. Then we can commit the changes using the command `git commit -m "message"`. Finally, we can push the changes to the remote repository using the command `git push origin master`.

**To Do 9.** *Check the status of the repository using the command `git status`.*

**To Do 10.** *Add the modified files to the staging area and commit the changes with an appropriate message.*

**To Do 11.** *Push the changes to the remote repository.*

## 4 Implementing FGSM

In this section, we will implement the FGSM and PGD methods to generate adversarial examples. The code is already implemented in the `attack_utils.py` script. The function `compute` takes the input image, the target class and returns the perturbation  $\delta$ .

**To Do 12.** *Implement the `compute` method in the class `FastGradientSignMethod` in the script `attack_utils.py`.*

**To Do 13.** Test your code by running the command `python evaluate.py --path cnn --model cnn --attack fgsm --epsilon 0.05`. Observe the outputs in the `experiments/cnn` folder.

As specified, the budget  $\epsilon$  is the maximum perturbation allowed, it defines how perceivable the adversarial example is. The higher the  $\epsilon$ , the more perceivable the adversarial example is.

To run multiple experiments, we can either run a `bash` script but this solution would not work on Windows. To simplify the process, we can use the `run_xp.py` script that will run multiple experiments with different hyperparameters.

**To Do 14.** Try different values ( $10^{-1}$ ,  $10^{-2}$  and  $10^{-3}$ ) of  $\epsilon$  for the FGSM attack using the script `run_xp.py`. Observe the outputs in the `experiments/cnn` folder both visually and in terms of accuracy.

**To Do 15.** If you observe that you have a low accuracy with almost no perceivable perturbation, you can push the changes to the remote repository with the message "Implemented FGSM attack".

## 5 Implementing PGD

In this section, we will implement the PGD method to generate adversarial examples. The code is partially implemented in the `attack_utils.py` script.

**To Do 16.** Implement the `compute` method in the class `ProjectedGradientDescent` in the script `attack_utils.py`.

**To Do 17.** Test your code by running the command `python evaluate.py --path cnn --model cnn --attack pgd --epsilon 0.1`. Observe the outputs.

**To Do 18.** Take a budget  $\epsilon = 0.05$  and try different values of the number of steps  $T$  and the step size  $\alpha$ . Make sure that PGD is more effective than FGSM.

A good practice is to use a small step size  $\alpha$  in the order of  $\epsilon/T$ . This ensures that the perturbation is not too large at each step but still reaches the maximum budget  $\epsilon$  after  $T$  steps. One way to automatically implement this value is the set `None` to the step size  $\alpha$  in the parser and change the value in the `attack_utils.py` script.

**To Do 19.** *Modify the `--init--` method in the class `ProjectedGradientDescent` in the script `attack_utils.py` to set the step size  $\alpha$  to  $\epsilon/T$  if it is set to `None`. (Round the value for better printing.)*

**To Do 20.** *After you have found the best hyperparameters for the PGD attack, push the changes to the remote repository with the message "Implemented PGD attack".*

## 6 Comparison of the attacks

The attacks are now implemented and we can compare the FGSM and PGD attacks on the CNN model. We can observe some behaviors such as the robustness of the model to the attacks.

**To Do 21.** *Compare the results of the PGD (with same hyperparameters) on the CNN model and the FC models. What do you observe?*

**To Do 22.** *Compare the results of the PGD (with same hyperparameters) on `resnet18` and a `resnet18` that has been overfitted. What do you observe?*