

Classification Adversarial Attacks

Intro

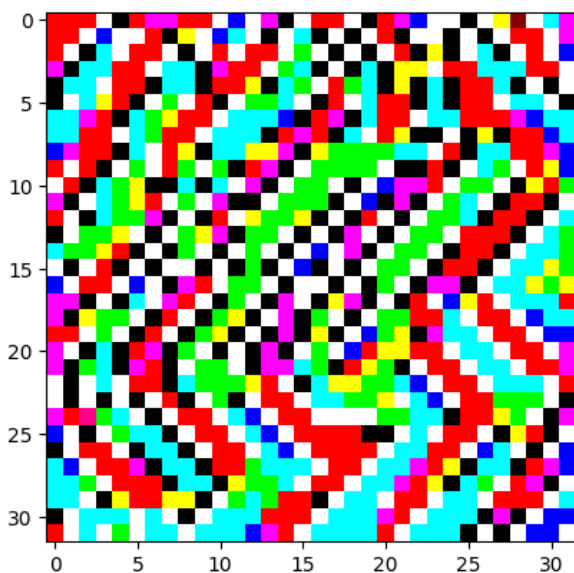
Adversarial Perturbations are small perturbations that adhere to strict norm limitations, that once combined with an input, can alter a deep neural network's output. In this assignment we explore the creation and usage of universal perturbations against two classification networks. Unlike non-universal perturbations, which are crafted individually for each sample, a single universal perturbation is made to be used against the entire dataset. Presenting a more realistic use case. Therefore, our goal is to find two perturbations δ such that

$$\delta = \underset{\delta \mid \forall i: x_i + \delta \in X, \|\delta\|_\infty \leq \epsilon_\infty}{\operatorname{argmax}} \sum_{i=1}^n \ell_{01}(M(\operatorname{Clip}(x_i + \delta, 0, 1)), y_i) \text{ under the bound } \epsilon_\infty = \frac{8}{255} \text{ for each corresponding model } M.$$

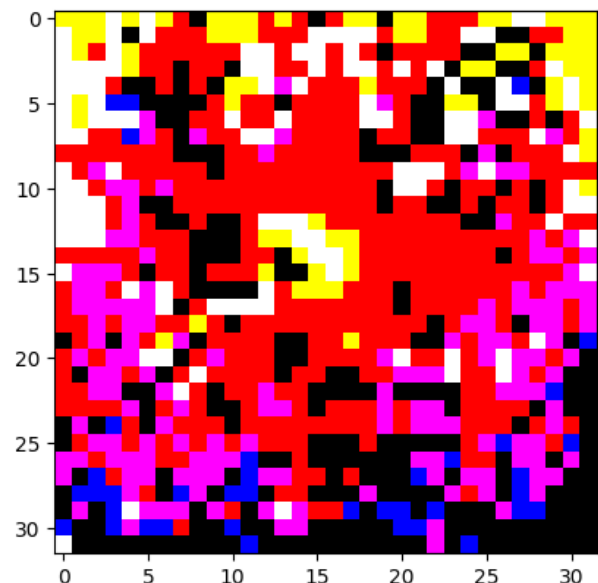
The classification models provided, which we will attack using our implemented methods, are ResNet18 trained on the Cifar10 dataset as our standard model, and the Wong2020Fast model that was similarly trained on Cifar10 but was trained using adversarial training and thus robust to adversarial attacks.

For the task, we have implemented a universal PGD algorithm with a step-scheduler. An algorithm like PGD, that instead uses a single perturbation which is calculated by aggregating the gradients throughout all the batches. We have also chosen cross-entropy loss as a substitute for ℓ_{01} .

We generated two universal adversarial perturbations. The perturbation that targets the standard model has successfully reduced the model's accuracy by 71.27% (75.65% attack success ratio) and the perturbation that targets the robust model has reduced its accuracy by 2.01% (2.41% attack success ratio). Thus, overall proving effective and highlighting the dangers of universal perturbations.



Standard model perturbation: normalized to (0, 255) for better visualization.



Robust model perturbation: normalized to (0, 255) for better visualization.

Implementation and Methods

Projected gradient descent (PGD) is widely regarded as a proven and effective method for generating adversarial perturbations by iteratively applying gradient ascent on an input image to maximize the loss function, while ensuring the perturbation remains within a predefined norm constraint, in our case L_∞ , through projection. It is used on each sample to craft a corresponding perturbation that can alter the output of the model for corresponding input. However, for this assignment we aim to generate universal perturbations that are effective for as many samples as possible for each corresponding models. Meaning that we can't use PGD directly and thus require a modified approach.

Universal Projected Gradient Descent (UPGD) extends the method into crafting a single perturbation applicable across multiple inputs. The gradients of the loss w.r.t the perturbation are accumulated across all the batches in the training set to compute a universal perturbation by applying gradient ascent w.r.t the entire set. This makes the perturbation broadly effective as opposed to tailored for only specific samples and thus suitable as a universal adversarial example. The perturbation is updated using the sign of the accumulated gradients and scaled by the step size variable α and then projected to ensure it remains within the specific L_∞ bound ϵ , similarly to PGD.

Algorithm 1 Non-universal PGD adversarial attack

Input M : Classification model
Input (x, y) : attack data sample
Input ℓ : differentiable criterion
Input ϵ : attack L_∞ norm bound
Input K : number PGD iterations
Input α : step size for the attack

```
1:  $\delta \leftarrow \text{Uniform}(0, 1)$ 
2:  $\delta_{\text{best}} \leftarrow \delta$ 
3:  $\text{Loss}_{\text{best}} \leftarrow \ell_{01}(M(x), y)$ 
4: for  $k = 1$  to  $K$  do
5:   optimization step:
6:    $g \leftarrow \nabla_{\delta} \ell(M(x + \delta), y)$ 
7:    $\delta \leftarrow \delta + \alpha \cdot \text{sign}(g)$ 
8:    $\delta \leftarrow \text{clip}(\delta, -\epsilon, \epsilon)$ 
9:   evaluate perturbation:
10:   $\text{Loss} \leftarrow \ell_{01}(M(x + \delta), y)$ 
11:  if  $\text{Loss} > \text{Loss}_{\text{best}}$  then
12:     $\delta_{\text{best}} \leftarrow \delta$ 
13:     $\text{Loss}_{\text{best}} \leftarrow \text{Loss}$ 
14:  end if
15: end for
16: return  $\delta_{\text{best}}$ 
```


Non-Universal PGD, taken from the assignment's instructions.

Algorithm 2 Universal PGD adversarial attack

Input M : target model
Input $(X_{\text{train}}, Y_{\text{train}})$: target dataset
input ℓ : differentiable criterion
input ϵ : attack L_∞ norm bound
input K : number of UPGD iterations
input α : step size for the attack

```
1:  $P \leftarrow \text{Uniform}(0, 1)$ 
2:  $P_{\text{best}} \leftarrow P$ 
3:  $\text{Loss}_{\text{best}} \leftarrow 0$ 
4: for  $k = 1$  to  $K$  do:
5:   optimization step:
6:    $g \leftarrow 0$ 
7:   for  $(x, y)$  in  $(X_{\text{train}}, Y_{\text{train}})$  do:
8:      $g \leftarrow g + \nabla_P \ell(M(\text{clip}(x + P, 0, 1)), y)$ 
9:   end for
10:   $P \leftarrow P + \alpha * \text{sign}(g)$ 
11:   $P \leftarrow \text{clip}(P, -\epsilon, \epsilon)$ 
12:  evaluation step:
13:   $\text{Loss} \leftarrow 0$ 
14:  for  $(x, y)$  in  $(X_{\text{train}}, Y_{\text{train}})$  do:
15:     $\text{Loss} \leftarrow \text{Loss} + \ell(M(\text{clip}(x + P, 0, 1)), y)$ 
16:  end for
17:  if  $\text{Loss} > \text{Loss}_{\text{best}}$  then:
18:     $P_{\text{best}} \leftarrow P$ 
19:     $\text{Loss}_{\text{best}} \leftarrow \text{Loss}$ 
20:  end if
21: end for
22: return  $P_{\text{best}}$ 
```

Our Universal PGD implementation.



We adapted the UPGD algorithm presented in lecture 7 to our own use case, by replacing the VO model with our classification models, the input with CIFAR10 images and labels, and patch-perturbation with an image-sized perturbation.

As our method relies directly on gradient-based optimization, the loss $\ell_{01}(M(x), y) = \begin{cases} 0, & M(x) = y \\ 1, & M(x) \neq y \end{cases}$, which is non-differentiable and discrete, is not applicable. Instead, we use Cross Entropy $\ell_{CE}(y, \hat{y}) = \sum_{i=1}^C y_i \log(\hat{y}_i)$ as a differentiable approximation, which is an effective loss function in classification tasks, as it quantifies the difference between two probability distributions, making it suitable for measuring the differences between the predictions and labels in our adversarial setting. Thus, allowing us to maximize the model's prediction error on perturbed inputs so that we can navigate to more efficient adversarial examples.

Additionally, we also use a step scheduler that decreases α by a factor of 10^{-1} every preset number of iterations. By decreasing the scale of α we allow steps to become finer and finer, allowing the algorithm to converge to slightly better results as opposed to a single fixed step size which might get stuck outside a maximum with a distance smaller than the step. Furthermore, several restarts per experiment are possible and suggested as the perturbation's efficiency relies on a random uniform initialization at the beginning of each experiment.

We have implemented UPGD with the step-scheduler in the provided code file `upgd.py` as the function `generate_uap_upga` alongside a loss evaluation function `evaluate_pert`.

The function takes the model, a torch DataLoader, number of iterations, norm bounding epsilon, step size alpha, a provided criterion, number of restarts and the number of iterations per step decrease, and returns an adversarial perturbation.

We have also implemented a final evaluation function `check_results` that is based on a tester made by Yonathan Kohli that was kindly shared with the rest of the students.

The code is showcased in `upgd.py` with usage examples against both models demonstrated in the main function, creating a perturbation for each model and then printing the evaluation of its effectiveness.

We ran multiple experiments with various hyperparameters (batch size, step size, number of iterations, number of iterations per step size decay) and various initialization strategies, including restarts of the same experiments. Saving the perturbations that provided the best results according to the final evaluation function.

To run our code file `upgd.py`, the file `resnet.py` and the weights of the ResNet18 model must be located in the path `"/models/cifar10/"` relatively to the code file, as was provided to us in the assignment.

Results

Our results are as following:

Model	Initial Accuracy	Robust Accuracy	Attack Success Ratio
ResNet18	94.22%	22.94%	75.65%
Wong2020Fast	83.34%	81.33%	2.41%

Where initial accuracy refers to the accuracy of the model on the dataset, robust accuracy refers to the accuracy of the model on the perturbed dataset, and attack success ratio calculates the reduction in accuracy w.r.t the initial accuracy.

These perturbations, bounded by $\varepsilon = \frac{8}{255}$, are almost invisible to us but are effective against the models.

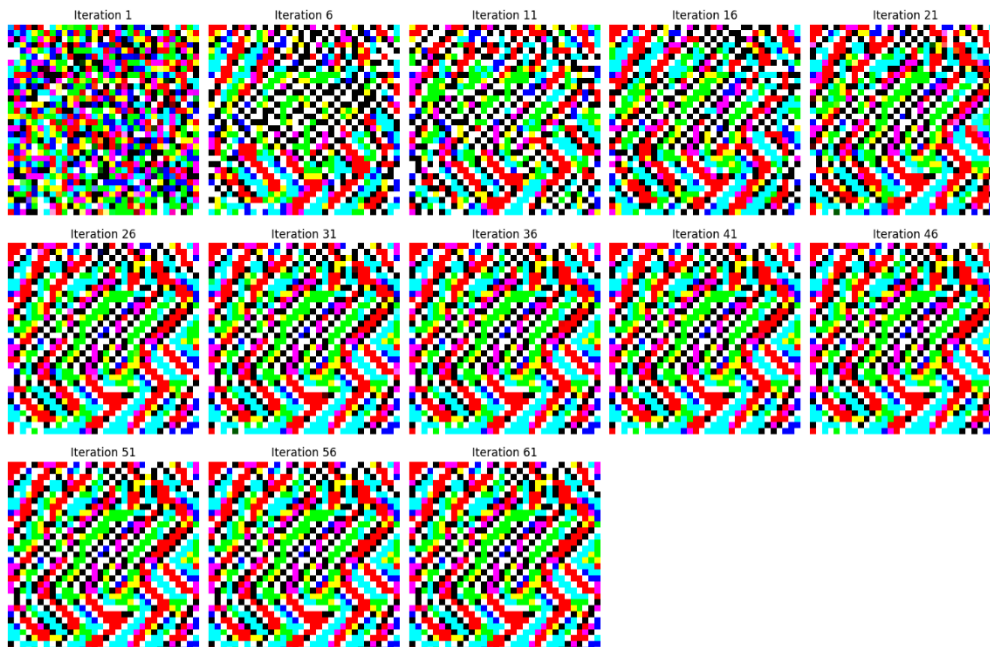


CAT (True Label) according to ResNet18

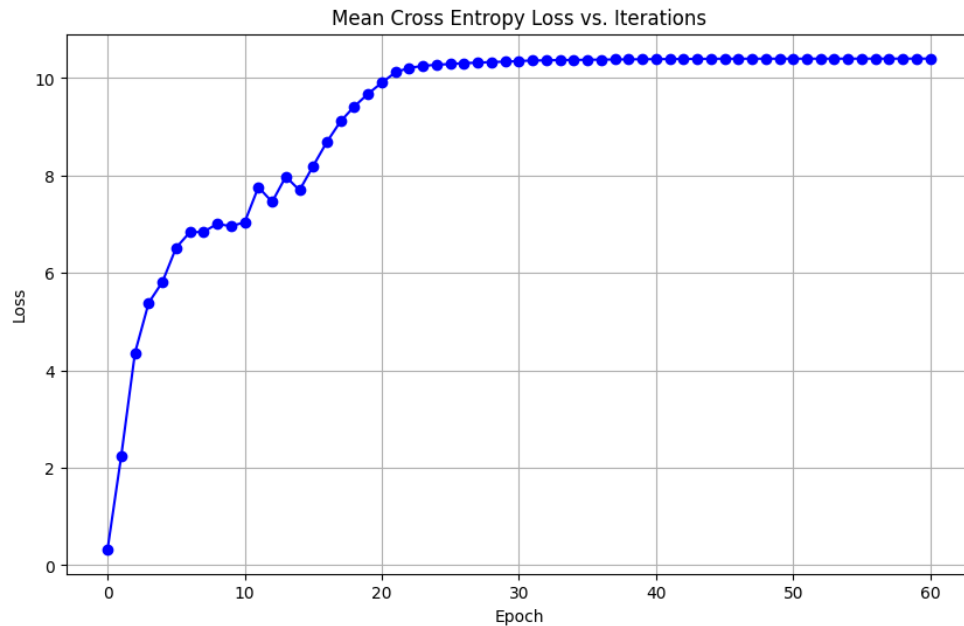


DOG (False Label) According to ResNet18

The adversarial example for the standard model has been created by running the algorithm for 60 iterations, with initial step size 0.1 and a decay every 15 iterations over batches of size 500. With the perturbation that generates the highest loss being chosen.



Evolution of the standard model perturbation over 60 iterations, normalized over (0, 255) for better visualization.



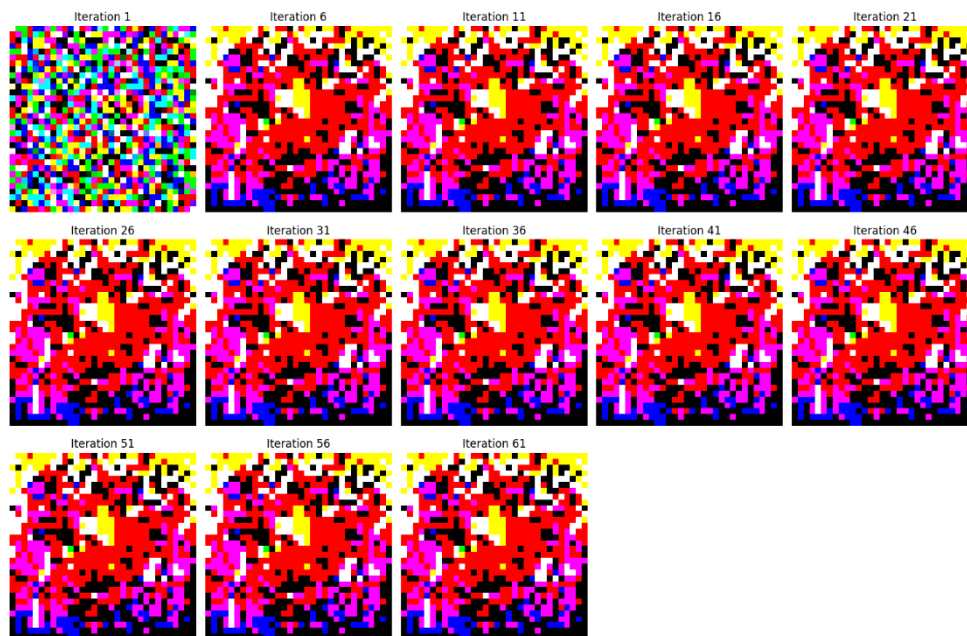
The mean cross entropy loss produced by ResNet18 on the perturbed input, per iteration.

The decay in the step size has been effective in helping the perturbation converge better. As we can see between the 10th and 15th iteration the step continues to jump around the maximum, raising and then lowering the loss. The smaller step then allows the algorithm to continue improving the perturbation, converging closer to the maximum.

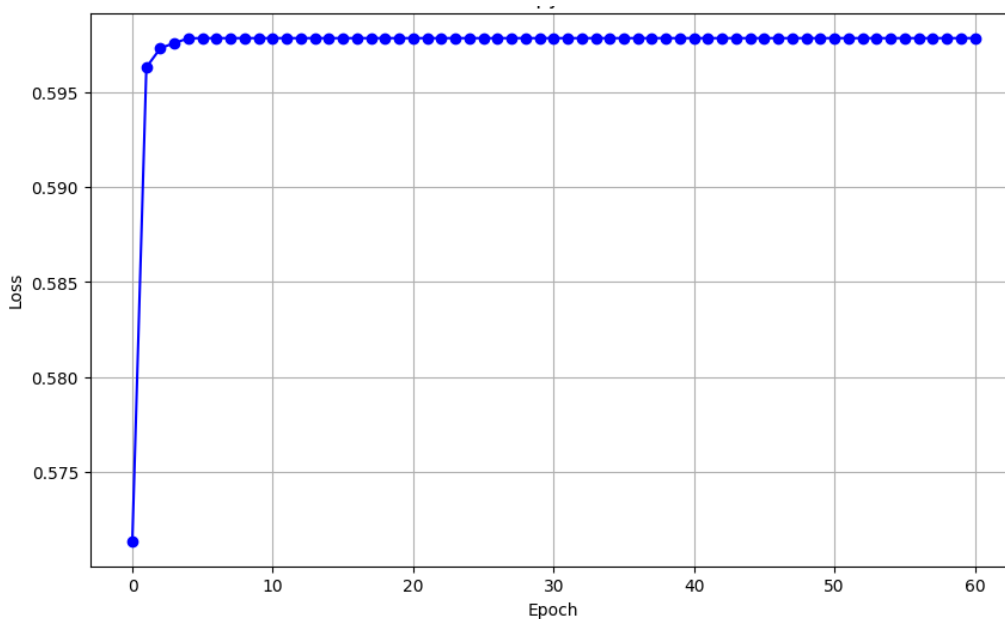
Overall, the perturbations seem to converge very quickly and with notable results, but there are very large differences between the effects of each perturbation on its corresponding model. 2020WongFast, a model trained using adversarial training, shows some robustness to universal perturbations. As various hyperparameters have been experimented with, it seems that the perturbation converges incredibly quickly and to the same 2%~ range in accuracy reduction. The results, given that the step sizes are suitable, are affected mostly by the initialization of the perturbation. Our best perturbation has been generated using 20 iterations with an initial step size of 0.1, a decay every 5 iterations and batch of size 500.

For comparison, we will present a robust model experiment with hyperparameters like those of the standard model run. That visualize the speed of the convergences and the difficulty of learning better perturbations.

The difference in patterns should be noted. Whereas the standard perturbation shows many finer patterns, the robust perturbation seems to converge into large shapes with homogeneous colors and some noise. Perhaps due to the nature of the robust model to withstand the universal patterns we see in the standard perturbation. The lack of complexity in the perturbations that are effective against the robust model, focusing instead on vulgar values to make any type of effect against a small number of samples, can explain the quick convergence.



Evolution of the robust model perturbation over 60 iterations, normalized over (0, 255) for better visualization.



The mean cross entropy loss produced by 2020WongFast on the perturbed input, per iteration.

To summarize, UAPs demonstrate a significant capacity to alter the outputs of deep neural networks yet underscoring a stark contrast in their effectiveness against standard versus robust models – with the latter showing an overall insignificant accuracy drop in comparison to the standard model. The stark difference highlights the important of incorporating adversarial training into the development of various models to enhances their robustness against similar sophisticated attacks.