Improving Universal Adversarial Attacks against TartanVO

Gabriele Serussi (915781579), Elie Nedjar (336140116)

September 2022

Abstract

Deep Neural Networks achieve state of the art results in Visual Odometry. This field is critical in the development of self-driving vehicles, in particular in an environment where GPS cannot work properly. Nonetheless, as it has been shown in [3], the performance of TartanVO, the state-of-the-art model for Visual Odometry, can be severely undermined by a patch attack. Our work tries to further harm the performance of this model. In particular, our contribution is three-fold: we make use of a **novel loss** that targets the estimation of the optical flow, we make use of the mean partial RMS for the rotation yielded by the model and we make use of momentum in the optimization of the patch. We notice that with the addition of these terms, the performance of TartanVO further decrease.

1 Introduction

Despite their remarkable performance, deep neural networks are known to be fragile. Indeed, as it has been first investigated by [5], it is possible to make a DNN misclassify an input by mildly perturbing it. In the following years, there has been an ever growing interest between researchers to find more and more powerful attacks. In the following years it has been discovered that it is possible to find a single perturbation that leads to a misclassification for several different input that works across different architectures. This kind of attack takes the name of Universal Adversarial Attack (UAA) and it presents a much more realistic threat than an attack crafted for a single input. Indeed, as DNNs take more and more critical roles in several domains such as in self-driving vehicles, exploiting an UAA does not require the access to the input that needs to be attacked, and this makes this kind of attack much more practical to exploit.

UAA are not limited to the task of image classification, but rather they can be found in several other fields. In this work we are going to assess the robustness of DNNs in the task of Visual Odometry, i.e. estimating the displacement of an agent using a sequence of images, which is an essential component for

Visual SLAM. In particular, we will evaluate the fragility of TartanVO [6] which constitutes the-state-of-the-art in this field, outperforming geometric-based models.

We started from the work of Nemcovskyy et al [3] and we have tried to look for some room for improvement. First, we have seen that TartanVO has been attacked using an end-to-end approach, without dealing with the internal working of the model. In particular, the authors do not exploit in its loss the intermediate estimate of the optical flow and the final estimate of the rotation. According to our tests, this leads to sub-optimal performance and it leaves room for improvement. For this reason, by exploiting the work of Ranjan et al[4], we have designed a special loss that manages to make up for this lack. Moreover, the authors of [3] make use of vanilla PGD. While this latter yields good performance, there are some researches ([1]) which show that by incorporating a momentum component in the optimizer, the step size of the attack becomes aware of the trend of the optimization, and it gets more responsive. Also, in order to balance all of the components of the loss, we have tried to use of Population Based Training [2], an hyper-parameter optimizer which overcomes the limits of a fixed set of hyperparameters by finding an optimal schedule of hyperparameters. Despite this, we see that in our case it leads to overfitting and its performance can be beaten by a less aggressive approach.

2 Methods

In our approach we have decided to explore the internal structure of TartanVO in order to find a weak point that could be exploited. For this reason it is of great importance to understand how TartanVO gets to its final answer. In detail, the architecture is made up by two main components: a matching network, constituted by a PWC-Net, and a pose network, made up by a slightly modified ResNet50.

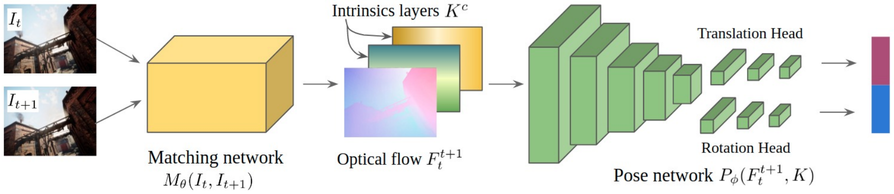


Figure 1: TartanVO internal structure. Taken from the original paper [6]

The matching network looks after the estimation of the optical flow between two consecutive images, whereas the pose network takes as input the optical flow and the intrisics of the camera and it outputs the final translation and rotation vector. Given that the assessment of the optical flow takes a crucial role in the final estimate, we wondered whether it could represent a potential point of failure. After some further research, we discovered that Ranjan et al

[4] managed to attack PWC-Net successfully by using a patch attack, similarly to the attack that we were trying to improve.
In particular, in their work they formulated the problem as the minimization of the **cosine similarity**. More in detail, by minimizing this function (or, as we have done, by maximizing the opposite of this function) we are looking for adversarial attacks whose direction is as opposite as possible compared to the direction of the optical flow estimated by the network (in terms of angle).
At first, we have tried to apply this loss to the estimate of the optical flow and to the estimate of the rotation. Nonetheless, later we have found empirically that by using the mean partial RMS for the rotation component, our results further improved. Moreover, in order to provide an additional improvement to the loss, we have mapped the cosine similarity with the log, thus computing the log of the minus cosine similarity. As we will show later, this yields an improvement in the gradients, which will make the optimization process more efficient. Consequently, our next idea was to use a hyper parameter optimizer since a good tuning can make a big difference in terms of generalization. We decided to tune the three coefficients that control the weight of each component ($(t_{flow}, t_{rot}, t_{tr})$) and the step size using an API called Ray Tune. Therefore, we needed to bring some modifications to the code to match the API needs. To our surprise, PBT made the patch overfit the validation dataset, and thus we moved to a less aggressive approach.
Finally, as advised in the guidelines, we have tried to implement the APGD algorithm to optimize our patch. The motivation behind this, is that APGD makes use of an adaptative learning rate that is more aware of the trend of the optimization.

3 Implementation and experiments

In order to ensure good comprehension of the report we'll define the notations that we used along this part: We have $\mathcal{J} = (0,1)^{h\times w\times 3}$ the input space We denote I^P as a perturbed image, $I^P = A(I,P)$ where $A : \mathcal{J} \times \mathcal{J} \to \mathcal{J}$ is the perturbation function and $P \in \mathcal{J}$ is a patch. For a set of images $\{I_t\}$ we define their perturbed set as $\{I_t^P\} = \{A(I_t,P)\}$ We define the output of the TartanVO model being $VO : \mathcal{J} \times \mathcal{J} \to \mathbb{R}^3 \times so(3)$ i.e for a pair of consecutive images $\{I_t, I_{t+1}\}$ it assesses the relative camera motion $\delta_t^{t+1} = (T_t, R_t)$ where $T_t \in \mathbb{R}^3$ is the 3D translation and $R_t \in so(3)$ is the 3d rotation. We define a trajectory as a set of consecutive images $\{I_t\}_{t=0}^{L}$ for some length L

3.1 Defining the training loss

As said in the previous sections our first modification was to make use of the cosine similarity as a criterion to optimize the patch that will exploit the optical flow

$$\hat{p} = \underset{p}{argmax}\ \mathbb{E}_{(I_t, I_{t+1})} \log(1 - \frac{(u,v)\cdot(\tilde{u},\tilde{v})}{||(u,v)|| \cdot ||(\tilde{u},\tilde{v})||})$$

Where

$$(u, v) = VO(I_t, I_{t+1})$$
$$(\tilde{u}, \tilde{v}) = VO(I_t^p, t_{t+1}^p)$$

The above equation measures the similarity between the optical flow estimated by the network for the clean images and the optical flow of the perturbed images. Our goal is to make the estimation of the optical flow of the perturbed image as different as possible from the estimation of the optical flow of the clean image. Given than TartanVO relies upon the estimation of the optical flow in order to assess the translation vector, by attacking the optical flow estimate, our attack manages to get more effective in creating a good performing universal adversarial attack.

Also, to improve the numerical stability of the optimization process we have decided to apply the logarithm which, being an homomorphic map, does not change the result of the problem, but it aids to make the optimization process more efficient and faster, requiring less iterations to converge. Indeed, as we can see on the figure, for the worst performing samples (close to zero) the gradient of the loss with the logarithm is very high, whereas in the same area the gradient of the loss without the log is small. In the best performing samples (from $\frac{\pi}{2}$ until 2) the gradient of the loss which incorporates the log is mildly below the original one. But in this case, since we are talking about well-performing samples, the need of big gradients is less prominent.

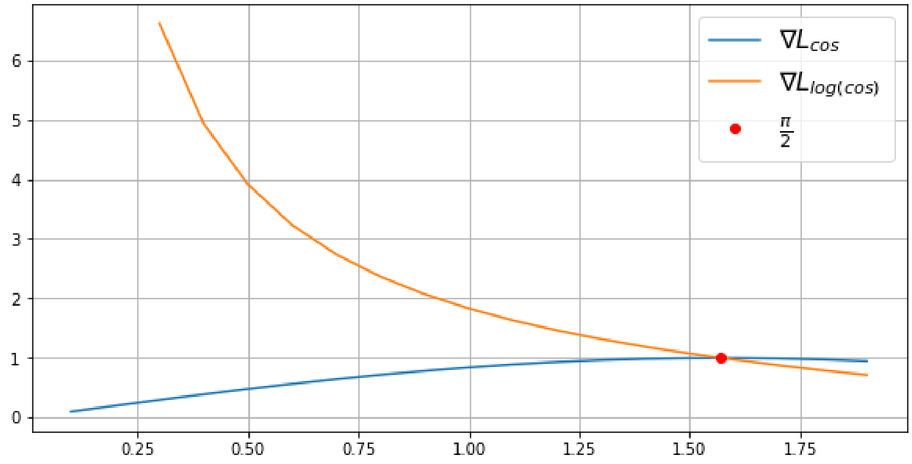


Figure 2: In orange the gradient of the loss with the log, and in blue the gradient of the original loss

In order to evaluate the performance of the patch on the validation set, we considered only the translation error, for which we used the mean partial root mean squared loss (MPRMS) on the last pair of frames. For each trajectory we used the last pair of frames since they accumulate the error of all of the previous frames.

Then, by denoting the rotation loss, the optical loss and the translation loss with respectively L^{rot}, L^{flow} and L^{traj}, we get that the loss that we used to optimize the patch is the following:

$$t^{rot}L^{rot} + t^{flow}L^{flow} + t^{traj}L^{traj}$$

Where $t^{rot}, t^{flow}, t^{traj}$ denote the weight given to each loss component in the final criterion, which lead us to the next experiment that we have made: tuning.

3.2 Hyperparameter tuning

A loss with some parameters is not easy to balance manually and in this situation the solution is to do hyperparameter tuning, but a traditional tuning algorithm did not seem to be sufficient. In order to go beyond the limits of a traditional scheduler and make the patch as efficient as possible, we tried to use an hyperparameter tuner based on Population Based Training. PBT doesn't find the best fixed parameters, but rather it tries to learn the best hyperparameter configuration for each epoch, i.e instead of learning just some hyperparameters, it will learn an hyperparameter policy. Consequently, with PBT we are training the patch with a loss that changes over time (since the weight of each component vary over the epochs).
Population based training trains in parallel a population of models and then selects the best hyperparameters, striking a good balance between exploitation and exploration:
-Exploiting is taking the set of hyperparameters that leads our model to the best results, by making use of the knowledge that it has accumulated over the previous epochs. During exploitation, PBT ranks all of the models in the population. When the current agent is in the bottom 25% it samples uniformly from the top 25% configurations and it copies its parameters.
-Exploring is trying new parameters on a model in order to get better results. During the exploration, PBT first perturbs the parameter with a factor of 0.8 or 1.2. It also resamples the parameters from the original prior distribution defined with some probability.
In order to keep proportions between the three losses we let PBT sample uniformly new values for the weights between 0 and 1. The loss that we used with PBT is the average of the MPRMS of the last frame of every trajectory. The reason of this choice is that MPRMS tends to generalize better. Consequently, during the optimization, Population Based Training tries to *craft* a loss that will lead us to the lowest evaluation loss. Despite all of these good premises, PBT did not manage to give the best results, given that it lead to overfitting and it was outperformed by our next attempt.

3.3 Optimization with APGD

We wanted to make the step size of the optimization process responsive to the trend of the search. Moreover, we wanted to strike a good balance between

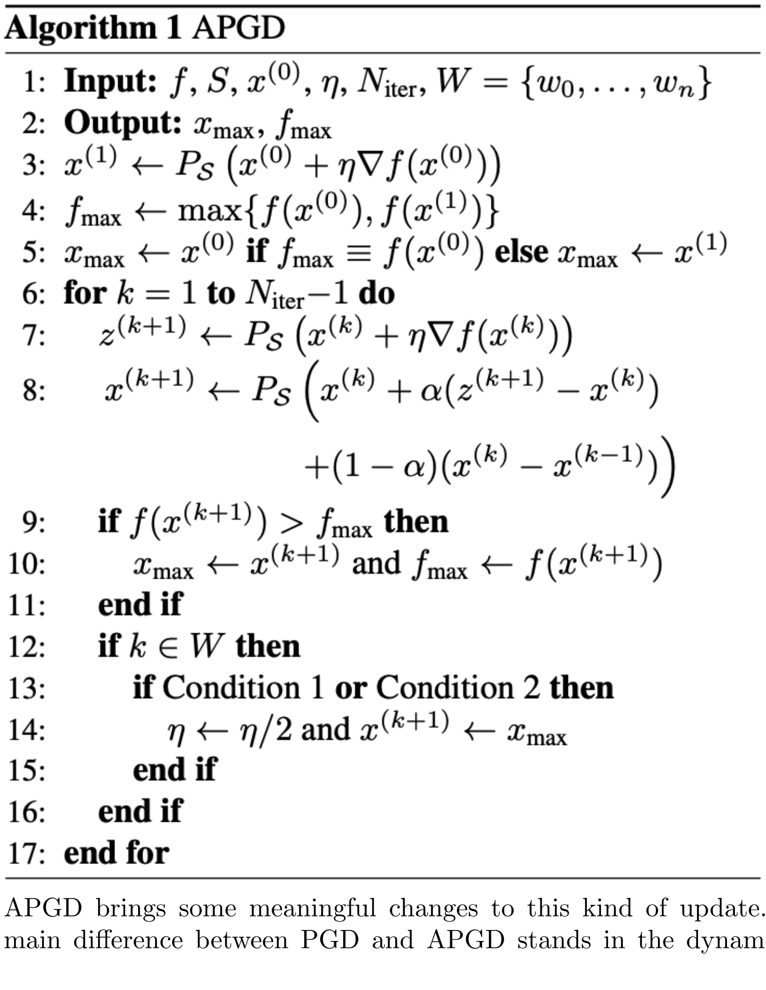
exploring new patches, and exploiting the knowledge that we have accumulated so far. These necessities were fulfilled by an algorithm called Auto-PGD [1].
In particular, the value that we want to optimize during training can be formalised in the following way:

$$\max_{p} f(g(z), y)$$

Where f is the loss criterion, g is the model's output, $z = \{I_t^P\}_{t=0}^T$ which are the patched input and $y = \{\delta_t\}_{t=0}^T$, i.e. the ground truth trajectories. In general, this problem can be solved with PGD, where each update of the current point in the iteration k is defined like the following:

$$x^{(k+1)} = P_{\mathcal{S}}(x^{(k)} + \eta^{(k)} \nabla f(x^{(k)}))$$

where $\eta^{(k)}$ is the step size at the k iteration, $\mathcal{S}$ is the space of the admissible input and $P_{\mathcal{S}}$ is the projection onto it.



Algorithm 1 APGD

```
**Input:** $f, S, x^{(0)}, \eta, N_{\text{iter}}, W = \{w_0, \ldots, w_n\}$
**Output:** $x_{\max}, f_{\max}$
$x^{(1)} \leftarrow P_{\mathcal{S}}\left(x^{(0)} + \eta \nabla f(x^{(0)})\right)$
$f_{\max} \leftarrow \max\{f(x^{(0)}), f(x^{(1)})\}$
$x_{\max} \leftarrow x^{(0)}$ if $f_{\max} \equiv f(x^{(0)})$ else $x_{\max} \leftarrow x^{(1)}$
for $k = 1$ to $N_{\text{iter}} - 1$ do
    $z^{(k+1)} \leftarrow P_{\mathcal{S}}\left(x^{(k)} + \eta \nabla f(x^{(k)})\right)$
    $x^{(k+1)} \leftarrow P_{\mathcal{S}}\Big(x^{(k)} + \alpha(z^{(k+1)} - x^{(k)}) + (1 - \alpha)(x^{(k)} - x^{(k-1)})\Big)$
    if $f(x^{(k+1)}) > f_{\max}$ then
        $x_{\max} \leftarrow x^{(k+1)}$ and $f_{\max} \leftarrow f(x^{(k+1)})$
    end if
    if $k \in W$ then
        if Condition 1 or Condition 2 then
            $\eta \leftarrow \eta/2$ and $x^{(k+1)} \leftarrow x_{\max}$
        end if
    end if
end for
```

APGD brings some meaningful changes to this kind of update. One of the main difference between PGD and APGD stands in the dynamical choice of the step size. As we can see in line 12 of the algorithm, we have a predefined set W of checkpoints iteration whose elements are given by $w_j = \lceil p_j N_{iter} \rceil \leq N_{iter}$ with $p_j \in [0,1]$ defined as $p_0 = 0, p_1 = 0.22$ and $p_{j+1} = p_j + \max\{p_j - p_{j-1} - 0.03, 0.06\}$. We can notice that the minimum period is 0.06. The step size is halved if one of the two conditions of line 13 are respected.

Condition 1: it counts in how many cases since the last checkpoint w_{j-1} the update step has been successful in increasing the loss function. If this happened for at least a fraction ρ of the total update steps, then the step size is kept, as the optimization is proceeding properly (we use $\rho = 0.75$).

Condition 2: it holds true if the step size was not reduced at the last checkpoint and there has been no improvement in the best found objective value since the last checkpoint. This prevents the optimization process from getting stuck in potential cycles.

Another important feature of APGD is that when the step size is halved during a given iteration k, we do **not** restart from $x^{(k)}$ but rather from x_{max} which is the point with the highest value of f found so far. This allows the algorithm to reach a more localized maximum. Indeed, we want our optimization algorithm to start by exploring the whole feasible set, and to gradually bring the search of the loss to a local maximum.

3.4 Training on new samples

In the original paper that tried to attack TartanVO [3], there were more samples. For this reason, our idea was to increase the size of the dataset and to increase the length of the trajectory, in order to train and evaluate our model with more samples. Unfortunately, due to memory limitations on the server's GPUs, training with longer than the default trajectories length was unfeasible as it made the job crash. This has led to a drop in performance in some cases. For instance, using PBT on a validation set with a longer trajectory would have allowed us to find a better hyper parameter policy, which in turn would have yielded better results.

4 Results

At first, in order to provide a good balance between each factor of the loss that was formulated above, we tried to use some traditional algorithms to perform hyperparameter tuning, such as Grid Search. Nonetheless, we immediately realized that this solution would not scale to be able to thoroughly search for the optimal values. It is for this reason that we made use of PBT (by using the API of Ray Tune) which should have been able to yield much better results. Indeed, its strength stood on its dynamicity, where it is able to quickly terminate bad-performing trials and it is able to effectively perturb the well-performing ones. Moreover, as it has been said above, it is able to find an optimal schedule of hyperparameters, rather than a fixed set of hyperparameters which usually

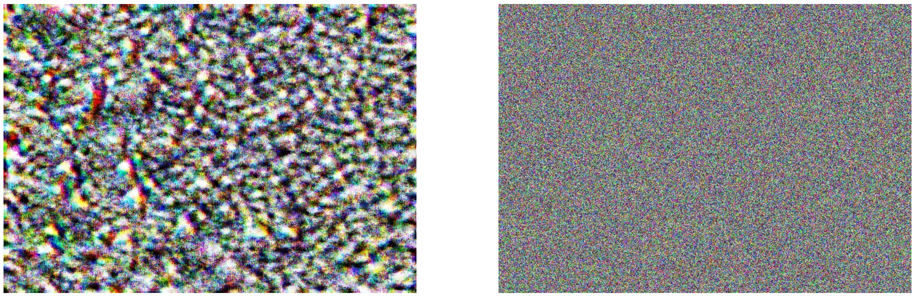
are less performant.
Nonetheless, despite delivering very good results on the evaluation set, we have noticed that the performance dropped significantly in the testing set. Also, the final patches do not present any kind of recognizable pattern, and they look as if they were randomly sampled. We suspect that such a gap between the score on the evaluation set and the score on the training set could be explained by the phenomenon of overfitting. In the first experiment we have split the dataset in the following proportions: 60% to the training set, 20% to the evaluation set and 20% to the testing set. We tried to mitigate overfitting by changing the split in the following wat: 40% to training, 40% to evaluating and 20% to test. Despite this, we did not manage to effectively counteract overfitting and we were not able to find any other sounded method to fight it.
As a consequence, we moved to a different experiment.
The next experiment was using APGD with fixed hyper parameters. Thanks to the previous experiments with ray we had an approximate idea of the value of $t_{tr}, t_{rot}, t_{flow}$ for the training criteria which were

	t_{tr}	t_{rot}	t_{flow}
value	0.6	0.3	0.1

We also used the MRPRS loss for the rotation and kept the log of the cosine similarity for the optical flow. After training with APGD algorithm we obtained a more *good-looking* patch (now there is a pattern that is clearly recognizable) which gave us a final loss of 0.764 on the validation set and a final loss of 0.728 on the testing set.

So if we compare it to our previous results we can observe that the patch is way more overfitted when using PBT comparing to the one obtained with APGD.

	L_{valid}	L_{test}	MPRMS adversarial to clean ratio
PBT	0.812	0.690	1.9818
APGD	0.764	0.728	2.141


(a) Patch obtained after APGD

(b) Patch obtained after PBT

Figure 3: Comparison of the two patches

We can deduce from these results and from the shape of the patch that Auto-PGD leads us to optimal results which are higher than the ones obtained in the original paper [3].

To sum up, in order to attack TartanVO we first revisited the loss that we used by parameterising it with multiple factors (i.e., the factor for the rotation loss, the MPRMS, and the factor of the optical flow, the log of the negative cosine similarity). The idea that immediately followed was to tune the hyperparameters in an optimal way. Unfortunately this latter did not give us the expected results. Finally, we changed the optimization algorithm and we used APGD which gave us our final result after training it on a bigger dataset.

References

[1] Francesco Croce and Matthias Hein. *Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks*. 2020. arXiv: 2003.01690 [cs.LG].

[2] Max Jaderberg et al. *Population Based Training of Neural Networks*. 2017. DOI: 10.48550/ARXIV.1711.09846. URL: https://arxiv.org/abs/1711.09846.

[3] Yaniv Nemcovsky et al. *Physical Passive Patch Adversarial Attacks on Visual Odometry Systems*. 2022. DOI: 10.48550/ARXIV.2207.05729. URL: https://arxiv.org/abs/2207.05729.

[4] Anurag Ranjan et al. *Attacking Optical Flow*. 2019. DOI: 10.48550/ARXIV.1910.10053. URL: https://arxiv.org/abs/1910.10053.

[5] Christian Szegedy et al. *Intriguing properties of neural networks*. 2013. DOI: 10.48550/ARXIV.1312.6199. URL: https://arxiv.org/abs/1312.6199.

[6] Wenshan Wang, Yaoyu Hu, and Sebastian Scherer. "TartanVO: A Generalizable Learning-based VO". In: (2020). DOI: 10.48550/ARXIV.2011.00359. URL: https://arxiv.org/abs/2011.00359.