A Project on Patch Adversarial Attacks on TartanVO

Vladislav Bogun, Ayala Avital
Department of Computer Science – Technion, Haifa, Israel

I. Abstract.

This project illustrates the susceptibility of visual odometry (VO) models to universal adversarial perturbation attack. These types of attacks usually have been shown to change a classification network output while here it is demonstrated that a VO network can be affected as well, our aim is to maximize the deviation from the original path in 3D space of a drone by displaying a patch as a billboard sign since such a scenario is plausible and can be a severe security issue.

The main goal of our work is to improve the results of this paper, while taking into consideration the fact that we are provided with very little data therefore generalization properties are going to be one of the main focus points of our work.

Our target model that we aim to fool is a pre-trained TartanVO model, which is a Visual SLAM (Simultaneous Localization and Mapping) system used in autonomous robotic systems. This network uses visual data extracted from the camera view of the scene such as optic flow, translation between frames and rotation in order to calculate the next step of a trajectory.

In the project, we researched different criteria that use trajectory data and several optimizers in order to get the optimal perturbation for unseen data. It is important to state that we worked in a setting of data shortage. Our main aims were producing smooth loss as well as enforcing generalization properties.

Our key insights and results derived from working on this project are such that in the setting of Visual Odometry trajectory based systems and PGD based algorithms for producing perturbations, it is not easy to construct a smooth training loss, but still we managed to improve some of the existing results. The best performing algorithm according to our experiments seems to be the Momentum PGD. And lastly, the use of flow criteria as well as working in a stochastic setting during training appears to have less desirable effect in terms of trajectory deviation and training stability, therefore we discarded them from our training scheme.

II. Introduction.

In this section we are going to explain the problem domain of Adversarial Perturbations, especially the Universal Adversarial Perturbations and their applications for Physical Passive Patch Adversarial Attacks on Visual Odometry Systems, since in this project we aim to optimize the result of such attacks in terms of generalization and performance.
We will do so by going through some important papers that helped us understand the problem domain as well as give some inspiration and ideas.

- Universal Adversarial Perturbations:
  This paper is a good introductory material to the researched subject. It not only describes the theoretical background behind what is an Universal Adversarial Perturbation, but also provides empirical justifications that such perturbations exist with simple algorithms that compute them.
  As we stated above, perturbations are slight changes of the input image that are able to fool a deep network. The meaning of **universal** is such that those perturbations are image-agnostic and work well on the whole natural images distribution, and even more than that, those perturbations have good generalization properties across models - **cross model universality**.
  Since it's not very insightful that a change of the input differs the output, a constraint of the l_p norm constraint is imposed: $|v|_p \leq \epsilon$, for perturbation $v \in R^d$, and $\epsilon \in (0, 1)$. This is a **projection** operation and hence the name Projected Gradient Descent algorithm that in many places is used as a base for perturbation computations, our project is not an exception.
  The algorithm that the paper provides is based on the decision boundary of classifiers and for each image requires a calculation of the smallest translation vector that casts it to the decision boundary of its class with a further projection to ensure the norm constraint.
  The computation of decision boundaries seems to be computationally consuming and at the same time not provide very strong confidence of an attack, since the adversarial examples will concentrate on the boundaries. We are planning to look for more specific and strong adversarial attacks such as PGD.

- Towards Deep Learning Models Resistant to Adversarial Attacks:
  The main idea that we extracted from this paper is understanding PGD based algorithms and more specifically getting some theoretical and empirical insight why they work so well.
  PGD based algorithms is a very simple idea of harnessing the gradient descent methods and loss functions used in training the deep network, but to achieve the opposite goal by updating the adversarial inputs in the direction of the gradient (Gradient Ascent), the constraint of the norm (i.e. projection) is, as usual, applied after each step. The proposed version of an attack with good properties and computational simplicity is a FGSM (Fast Gradient Sign Method) which is a result of choosing the ∞ constraint norm in the projection step.

  The work lacks exploring variations of PGD based algorithms that potentially can

improve the adversarial effect. We are going to focus on that in our work.

The paper also provides a min-max optimizations scheme for training a robust to adversarial examples model, but it is beyond the scope of this project.

- Physical Passive Patch Adversarial Attacks on Visual Odometry Systems:
  This is probably the most important paper that contributed to our work. Firstly, the setting that we worked in was almost identical to the setting of this paper, therefore many of our code and technical ideas were taken or inspired by the code provided by this paper. The only major difference between the works is the trajectory datasets.

  The work goes deeper into specific details of producing adversarial examples for VO systems using PGD optimization. Instead of producing an adversarial example which is a fully modified input image, the paper suggests altering only a specific area of the image described as Patch that can be an equivalent of a billboard sign in the scene. Therefore, an adversarial example is produced by applying a PGD created perturbation on a patch with the help of Homography transformation and Lightning conditions of the scene. VO models take as an input a stream of frames that provide an orientation in the space knowledge for the system. Given a consecutive pair of frames, VO model estimates the translation and rotation needed for calculation of the next step of a trajectory. Given perturbed and groundtruth trajectories an estimation according to some criteria is then calculated.

  The paper presents some useful criteria such as RMS (Root Mean Square) of the deviation in the 3D physical translation between the accumulated trajectory motion, as estimated by the VO, and the ground truth, which is the target criteria that they aimed to maximize, **our task stays the same.**
  In addition to RMS, there was presented an alternative such as MPRMS (Mean Partial RMS) that takes into account all possible subtrajectories normalized by their length that is supposed to be more suitable for generalization properties. We will adopt the extensive use of RMS and PRMS in our optimization scheme.

  The work was done on synthetic data created in Blender 3D, the data that we worked on came from the same setting but with slightly different trajectory and patch characteristics. Therefore we can strongly rely on the insights of this paper.

  The paper does not address the issues related to the training loss function such as smoothness and does not try to seek an algorithmical improvement of the optimization scheme. Those issues will be addressed in our work.

- EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES:
  Until this paper came out there was speculation that deep networks were particularly susceptible to adversarial attacks due to their non-linear characteristics. In this paper the authors argue instead that it's the linear nature of modern deep networks, the very thing that makes them easy to train, that makes them susceptible. The paper also presents a fast way to generate adversarial examples, introduce an adversarial training method, and shows that this is an effective regularizer.

The paper proposes that the vulnerability of DN models to adversarial examples is considered a by-product of their linear behavior and high-dimensional feature space. In other words, small perturbations on an input can alter its classification because the change in the neural network's activation layer (as a result of the perturbation) scales with the size of the input vector.

The authors suggest a fast way of generating adversarial examples: **"fast gradient sign method"** . They found that this method reliably causes a wide variety of models to misclassify their input. Indeed we used this method in our work.

There is an interesting explanation behind generalization features of adversarial examples that states: adversarial examples occur in broad subspaces and this is why an example misclassified by one classifier has a fairly high prior probability of being misclassified by another classifier. And neural networks trained with current methodologies all resemble the linear classifier learned on the same training set.This reference classifier is able to learn approximately the same classification weights when trained on different subsets of the training set. The stability of the underlying classification weights in turn results in the cross model characteristics of adversarial examples.

Since the work states that vulnerability of DN models is due to some linearity characteristics of the models, algorithms that use first gradient information for adversarial examples computations should be sufficiently fine and our work will heavily rely on such optimization algorithms.

- Reliable Evaluation of Adversarial Robustness with an Ensemble of Diverse Parameter-free Attacks:
  The paper offers two main additions to the PGD-attack, the first is a way to update the step size, and the second is an improvement to the loss function. Since the latter is not relevant to the problem we are optimizing, we will focus only on updating the gradient step size.

  In addition, the article claims that they created a parameter-free algorithm, and thus it allows testing the robustness of the classification algorithms in a user-independent manner. But after we made adjustments to the algorithm so it will fit to our work, we noticed that some parameters are hard-coded in the code and their initial values have a significant effect on the loss value, therefore, we added an option to initialize one of them using a flag at the beginning of the program run.

  The main idea of **Auto-PGD** (APGD) algorithm is to partition the epoch iterations in the initial exploration phase, where one searches the feasible set for good initial points, and an exploitation phase, during which one tries to maximize the knowledge so far accumulated. The transition between the two phases is managed by progressively reducing the step size. The reduction of the step size is not a priori scheduled, but rather governed by the trend of the optimization: if the value of the objective grows sufficiently fast, then the step size is most likely proper, otherwise, it is reasonable to reduce it.While

the update step in APGD is standard, what distinguishes our algorithm from usual PGD is the choice of the step size across iterations, which is adapted to the overall budget and to the progress of the optimization, and that, once the step size is reduced, the maximization restarts from the best point so far found
We are going to implement this optimization technique and test it in our setting.

III. Methods.

As we already stated in the review of the Physical Passive Patch Adversarial Attacks on Visual Odometry Systems paper, the original approach consists of several important methods and approaches that we briefly described above, so we will dig more into the details here:

- **Problem Domain:**
  The original paper takes into account perturbations that come from the domain of low energy images, in our case patches of specific dimensions with values limited to be in (0, 1), an epsilon of 0.05 is used to ensure exclusion of 0 and 1 from the possible values.

- **Task:**
  The original task was maximizing RMS deviation in the 3D physical translation between the accumulated trajectory motions, as estimated by the VO, and the ground truth. The task of this work stays the same.

- **Patch adversarial attack setting:**
  Given a pair of consecutive trajectory frames I_1, I_2, a model will calculate the translation and rotation in space between those:
  $VO(I_1, I_2) = (T, R)$, where $T \in R^3$ and $R \in so(3)$.

  Given an image I that represents the current state of a trajectory, or more specifically the view of the scene, a perturbation P, a homographic transformation H that describes the view of P from perspective of I and albedo images I^0, I^1 that describe lightning conditions of P in the scene. Then a perturbed image computed as:
  $I^P = H(P) * (I^1 - I^0) + I^0$.

  For a given criteria over the trajectory motions l, we then aim to maximize the following expression: $l(VO(\{I_t^P\}_{t=0}^{L}), \{\delta_t^{t+1}\}_{t=0}^{L-1})$, where δ_t^{t+1} is a groundtruth translation and rotation between frames t and t+1.
  The target criteria that we want to maximize is the RMS n between the accumulated trajectory motion, as estimated by the VO, and the ground truth

  We are not planning to change anything in this approach because the whole aim of our work is to improve results over this specific setting.

- **Optimization and evaluation criteria:**

  The paper introduces two practical criterions for optimization: RMS and MPRMS. The latter incorporates the use of all subtrajectories for the loss calculation and therefore might be more suitable for inducing generalization properties:

  $l_{RMS}l(VO(\{I_t^P\}_{t=0}^L), \{\delta^{t+1}{}_t\}_{t=0}^{L-1}) = ||T(\prod_{t=0}^{L} VO(I_t^P, I_{t+1}^P)) - T(\prod_{t=0}^{L} \delta_t^{t+1})||_2$, where T is the translation between frames.

  $l_{MPRMS}l(VO(\{I_t^P\}_{t=0}^L), \{\delta^{t+1}{}_t\}_{t=0}^{L-1}) = \sum_{l=1}^{L} \frac{1}{L-l+1} \sum_{i=0}^{L-i} ||T(\prod_{t=i}^{i+l-1} VO(I_t^P, I_{t+1}^P)) - T(\prod_{t=i}^{i+l-1} \delta_t^{t+1})||_2$

  The optimization is done through a PGD attack with l_{inf} norm limitation using one of the two stated criterions. A separate criterion for training and evaluation steps as well as separate datasets are allowed.

  First of all, we used **RMS** loss in the **evaluation step** since the whole task is to produce a perturbation patch that will enforce maximum RMS deviation.

  The training loss function on the other hand is going to consist of **MPRMS** over the cumulative translations since we have very little data and such approach will mimic a process of **data augmentation** by taking into account all subtrajectories.

  More than that we are going to implement additional criteria that will harness the use of rotation, optical flow and deviation from the target coordinate which are all accessible to us both from the trajectory frames. The Translation criterion that was defined above will be denoted as **T_CRIT.** Then we make a **weighted composite** of all the losses and try to find the best combination of the weights.
  We do that in order to enforce a more smoothed optimization function that will help the PGD based algorithms with l_{inf} norm constraint that are in fact a signed gradient technique to smoothly converge into a better local maxima.

  ROT_CRIT: incorporating the knowledge rotation matrices in order to smooth out the cumulative loss. One of the main two parts that a trajectory step consists of: (T, R).
  Quanterion Product was used as a loss function of choice for this type of data.
  Quaternions are objects of mathematical notation for representing spatial orientations and rotations of elements in three dimensional space.
  Compared to rotation matrices, quaternions are more compact, efficient, and numerically stable.

T_TARGET_CRIT: harnessing available data of deviation from the target position. Provides additional information about the state of a trajectory at each step. Might be useful as it is another indicator of a robot's position in space.
This criterion goes hand in hand with T_CRIT as both are translation deviation measures. Therefore, the choice of T_CRIT imposes the same aggregation of translation deviations for target position. The only difference is the weights that we are going to assign to them.

FLOW_CRIT: model's computations as well as the groundtruth data make it available to provide the optical flow. Optical flow is the motion of objects between consecutive frames of sequence, caused by the relative movement between the object and camera. It is described by a function of image intensity I(x,y,t) between two timestamp positions related to the consecutive frames.
We are going to use a **MSE** (Mean Squared Error) deviation of optical flow between predicted trajectory steps and groundtruths as our flow criterion. We choose MSE, because it is a smooth convex function that is a core criterion of regression problems. In our case, the intuition is that similar optical flow properties indicate small deviation of a trajectory.

T_FACTOR, **ROT_FACTOR**, **T_TARGET_FACTOR**, **FLOW_FACTOR** are the respective weights for each of the criterions. Zero weight means, the use of a specific criterion is taken out of the composite.
Implementations of the all mentioned above criteria were taken from GitHub of this paper.

- **Optimization algorithm:**
  The original approach was to use a vanilla PGD with l_{inf} norm limitation.
  We decided to implement and test different PGD based variants to see if they can boost the training process: **Scheduled PGD**, **Momentum PGD**, **RMSPROP PGD**, **APGD**, **Stochastic PGD**.
  All of the algorithms mentioned above were based on GitHub's code of vanilla PGD taken from this paper.

  Scheduled PGD:
  The idea is to set a decaying factor **beta** that will reduce the alpha, which acts as a learning step in the PGD, the reduction is done each **t_decay** epochs.
  It should help prevent the perturbation update zigzag effect at a later epoch, thus not missing the maxima and better convergence accuracy are enforced.

  Momentum PGD:
  Incorporating a classic ML momentum optimization step that uses previously calculated gradients discounted by a momentum decay factor during each epoch step. Momentum optimization usually improves training speed as well as accuracy of convergence. The reason is that the accumulation of previously calculated gradients prevents the training from serious deviations due to some random noise in the data and other factors that can result in noisy gradient at some epoch.

RMSPROP PGD:
It is an adaptive learning rate technique that uses the moving average of the squared gradients for each weight corresponding to a parameter. Therefore the learning rate can be decreased as well as increased during each epoch. Hopefully this technique is going to help in better convergence of the perturbation, but there are fears that it won't work as well since with signed updates of PGD with l_{inf}.

APGD (Auto-PGD):
The main idea behind this variation of PGD is already described in the **Introduction** section under the review of Reliable Evaluation of Adversarial Robustness with an Ensemble of Diverse Parameter-free Attacks paper. Generally an adaptive algorithm that harnesses first and second order derivative information is probably going to be useful in our setting and therefore should be tested.

Stochastic PGD:
The original PGD that we based our implementations on is non-stochastic, therefore we implemented a functionality of stochastic minibatch updates. Stochastic updates have more variance, therefore they can help the model escape from unwanted well ("potential well") points of the function but at a cost of higher fluctuations.

- **Data Manipulation:**

As was stated before, the setting allows a differentiation of data into 3 different sets, while the trajectories are divided into 5 folders by their initial position in the scene.

Training set: which consists of data used for optimization step with PGD update that is done using the Training criteria.

Evaluation set: after each optimization step (epoch), an evaluation criteria is then applied on the evaluation set in order to choose the best perturbation computed up to this epoch. In other words, the perturbation that had the highest evaluation loss on the evaluation set will be returned as a result of the whole training process.

Test set: as was said at the beginning of this paper, our main goal is to create a Universal Adversarial Perturbation for VO system that will result in the highest RMS of the trajectory deviation. This set is used exactly to test this indicator.

The reasons behind choosing training and evaluation criteria are already discussed in the **Optimization and evaluation criteria** section.

What is left to do is decide how to manipulate the data:
For hyperparameter tuning of the weights of the training loss as well as for the optimizers we will use the whole data for each of the 3 sets.

For optimization scheme out of sample evaluation and comparison we will use 3/1/1

disjoint splits in a 5-fold Cross Validation manner. Such an approach will allow us to estimate how well a chosen training scheme's perturbation deals with unseen data.

For producing the final perturbation that we are going to submit we will choose 1 folder for evaluation, the rest 4 folders will comprise the training set, while all 5 folders will act as a test set to see how the final perturbation performs on all available data. The reason behind choosing a disjoint folder for the evaluation set is to enforce good generalization properties.

We decided to divide sets folder wise because trajectories in each folder are very close to each other since they begin at the same starting point, therefore trying to use part of one folder as test and the other part as evaluation or test may result in data leakage.

IV. Implementation and Experiments:

Our implementation is largely based on the code from the paper cited above Physical Passive Patch Adversarial Attacks on Visual Odometry Systems**.** All the optimization schemes that are presented in this work are modified versions of the original Vanilla PGD optimizer from the paper. The relevant details are going to be discussed in this section.

First we will describe the general pipeline of our experiments scheme and then dive more into the details. The pipeline consists of several stages that gradually build the optimal training scheme, the results of one stage will be used in the experiments of the next stage and so on.

Setting:
If not stated otherwise, the experimental setting will be done with a **vanilla PGD** optimizer with **alpha=0.05** while training, evaluating and testing on **the whole available data** in a **non-stochastic** manner. **T_FACTOR** is set to **1** and **T_CRIT** is **MPRMS**, while **evaluation** criterion is **RMS**. Number of **Epochs** is 20.

Evaluation Metric:
The evaluation metric is **RMS ratio**: $\frac{advRMS}{cleanRMS}$, where advRMS is RMS of a perturbed trajectory and cleanRMS is RMS of an original trajectory.
Since the calculations are presented per frame, we are going to aggregate as mean only the last frames values that represent the cumulative trajectory deviations.

1. Discovering an optimal training criteria:
As we stated in part III of this paper, we will try to smooth out our training criteria with the help of a weighted sum of several loss functions:T_CRIT, T_TARGET_CRIT, ROT_CRIT, FLOW_CRIT (for definitions and details go to part III).

The reason we **start with smoothing out the optimization loss function** is because if it's not smooth enough, then there will be a high instability while training with gradient based algorithms such as PGD and all experiments done with higher probability are going to be

compromised.

The setting of this experiment: since we care about the proportions of the weights in means that if there is an optimal weights solution, then multiplying it by a non-zero scalar leaves the solution to be optimal. We are going to fix **T_FACTOR=1** for the rest of the experiment, in fact it will be so for all of them. In addition to maximizing the RMS ratio we are going to focus on the smoothness of a training loss function.

- **Experiments:**

  - Experiment: optimal T_TARGET_FACTOR:
  Run for all values and determine the best T_TARGET_FACTOR.
  ROT_FACTOR=0
  FLOW_FACTOR=0
  T_TARGET_FACTOR in [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]
  - **Experiment: optimal ROT_FACTOR:**
  Run for all values and determine the best ROT_FACTOR.
  T_TARGET_FACTOR=0
  FLOW_FACTOR=0
  ROT_FACTOR in [0,100,200,300,400,500,600,700,1000,2000,5000,10000,20000]

  - Experiment: optimal FLOW_FACTOR:
  Run for all values and determine the best FLOW_FACTOR.
  ROT_FACTOR=0
  FLOW_FACTOR=0
  T_TARGET_FACTOR in [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.5]

  - Experiment: optimal training criteria:
  Given best values of flow, rot and t_target weights, we are going to search around a 3D cube whose center is those factors and infer the **final optimal training loss function**.

- **Implementations:**
  All criteria implementations are taken from the code of the paper cited above Physical Passive Patch Adversarial Attacks on Visual Odometry Systems

2. Determining a suitable stochastic/deterministic setting:
Upon finding out the most optimal training criteria, we are going to determine if a stochastic setting can improve our results.

Stochastic experiment **was chosen to be before optimizers hyperparameters tuning** since sometimes some problems can perform very badly in a stochastic setting because subgradients can have high variance and increase instability of gradient based optimizers. On the other hand, for some other problems stochastic approach can be a major improvement since it can prevent the optimizer from falling into undesirable “potential well”.

- **Experiments:**

  - Stochastic Experiment
  We are going to execute and compare the results of 4 separate training runs:
  - **Non-stochastic.**
  - **Minibatch size** in [1, 2, 5]

- **Implementations:**
  In order to incorporate stochastic and minibatch optimizations into our scheme flags: **–attack_sgd** and **–minibatch_size** were added to the CLI parser. **gradient_ascent_step** function of the attack.py module was modified to distinguish between stochastic/deterministic modes and accumulate gradients according to the minibatch_size or to use all the training data.

3. Hyperparameters tuning of the optimizers:
At this point we have at our disposal an optimal training loss function as well as knowledge of which stochastic setting to use and so we have a high confidence that our training process from now on will have better stability, thus we can start the tuning of our optimizers.

- Vanilla PGD:
- **Experiment:** No tuning is done and we use the default set of parameters.

- **Implementation:** the original implementation.

- Scheduled PGD:
- **Experiments: Optimal beta** and **t_decay values:**
  For beta in [0.5,0.6,0.7,0.8, 0.9]
  For t_decay in [3, 5,10]
  run and compare according to the **evaluation metric** defined above.

- **Implementation:**
  The original PGD was updated to have such hyperparameters as: **beta** and **t_decay**. CLI parser was modified accordingly. The **perturb** function of module **pgd.py** was altered to decay learning rate alpha by beta factor each t_decay epochs. This alpha update is done only if the t_decay argument is set through the CLI.

- Momentum PGD:
- **Experiment: Optimal Momentum value:**
  For momentum in [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9] run and compare according to the **evaluation metric** defined above.

- **Implementation:**

Is done by subclassing the PGD class of the pgd.py module into a MomentumPGD class and adding **momentum** hyperparameter field to the constructor as well as redefining **gradient_ascent_perturb_step** to execute momentum update step accordingly.

- RMSPROP PGD:

- **Experiment: Optimal rmsprop_decay value:**
  For rmsprop_decay in [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9] run and compare according to the **evaluation metric** defined above.

- **Implementation:**
  Is done by subclassing the PGD class of the pgd.py module into a RMSPropPGD class and adding **rmsprop_decay** hyperparameter field to the constructor as well as redefining **gradient_ascent_perturb_step** to execute rmsprop update step accordingly. Another field was added to ensure numerical stability and prevent division by "zero": **rmsprop_eps**, which is set to 10^{-8}.

- APGD:

- **Experiment:** No experiment is done since the APGD has no tunable hyperparameters.

- **Implementation:**
  In our implementation of the APGD attack, we extended the subclass Attack from the attack.py module so that it will use the main concepts implemented here. The original algorithm is intended for a classification problem hence we did some adjustments for the parts that were relevant to us (the gradient step with an emphasis on determining the step size) and redefined the methods **gradient_ascent_perturb_step** and **perturb** from the parent class to perform the attack according to the algorithm.

4. Determining the best optimizer.

At our hands a set of tuned optimizers on smoothed out training loss function in a suitable stochastic/deterministic manner. Let us try to infer which optimizer is the best for our task with the help of the following two experiments.

- **Experiments:**
  - Best optimizer in-sample evaluation experiment:
  For tuned optimizer in [PGD, MomentumPGD, RMSpropPGD, ScheduledPGD, APGD]: run and and compare results according to the **evaluation metric** defined above.

  - Best optimizer out-of-sample evaluation experiment:
  For 2-3 best performing optimizers based on in-sample evaluation, run 5-folds Cross Validation of the test folder and compare overall results according to the **evaluation metric.** The comparison will be between graphs that show dependency of RMS deviation by a specific split.

- **Implementations:**
  The **in-sample evaluation** is simply done through training/evaluating and testing on the whole available dataset, so no modifications are needed.
  The **out-of-sample evaluation** is performed by iterating over test folder possibilities, for each one a disjoint evaluation folder chosen and the rest of the 3 folders go to the training set. There will be 5 splits of 3/1/1 and for each one training cycle will be executed. The code is taken from the GitHub of the original paper.

5. Improving generalization properties.

We already discussed some points about enforcing generalization properties such as using MPRMS criterion of the Translation vector for optimization loss which has an effect of **data-augmentation**. We explained the theoretical background of why it helps but also the original paper empirical finding suggests the same. Therefore we are not going to conduct an experiment for that. The findings of the original paper also encourage use of RMS criterion for evaluation step.

Another way to impose generalization is by training the final perturbation in a splitted manner to avoid data leakage. We are suggesting a 4-1-5 split: one folder for evaluation, the rest 4 go to the training set, while the test set comprises all the data. The following experiment will help us to choose the optimal split and to make sure the data doesn't have any features that would harm the training in a splitted setting.

- **Split experiment:**
  For evaluation folder in [0, 1, 2, 3, 4] train in a 4-1-5 split using the best optimizer. Examine the results according to the **evaluation matrix** and choose the optimal split.

- **Implementation:** a slight modification to the **utils.py** model is needed to allow the CLI to accept the desired evaluation folder, calculate the rest of the split and set up Datasets and Dataloaders needed for the training.

6. FInal optimization scheme.

After conducting all the experiments defined above we are going to have an **optimization scheme** that consists of smoothed out **training loss function** that maximizes the task criterion and provides generalization, optimal **stochastic/deterministic** solution update approach, best out of the proposed here tuned **optimizers** that enforces generalization, 4-1-5 **data split** for even more generalization.

A perturbation that is going to be computed using this scheme is then going to be submitted.

7. More on implementations.

A higher level optimization scheme that suits our needs is implemented in **run_attacks.py** as **run_attacks_train_split** and manages the train-eval-test split as well as reporting results for each dataset of the split.

A folder **experiments** consists of modules like **res_parser.py** and **plot.py** which we implemented as an auxiliary tools for conducting this research.

- **res_parser.py**:
  This module is responsible for parsing the path tree of a results folder in order to retrieve all of the results. **Hyperparameters** of a single run are parsed from the path, **result values** are mean-aggregated from the csv files at rows corresponding to the last frames of the trajectory. The returned object is a list of dictionaries, each dict is composed of hyperparameters and result values of a single run.
  An option of filtering as well as sorting is implemented as well.

- **plot.py**:
  For the experiment's visualizations and graphs a plotting module was implemented. This module gets a list of results generated by **res_parser**, x_key values that are going to be aggregated into a single x “coordinate” and y_label - name of the dict key that should be extracted from the result's dict and displayed on the y-axis.

V. Results:

We are going to present the results of the experiments defined in the previous part of the work. <u>All graphs are sorted in a descending order.</u>

1. **Discovering an optimal training criteria**:
   Experiment: optimal T_TARGET_FACTOR:

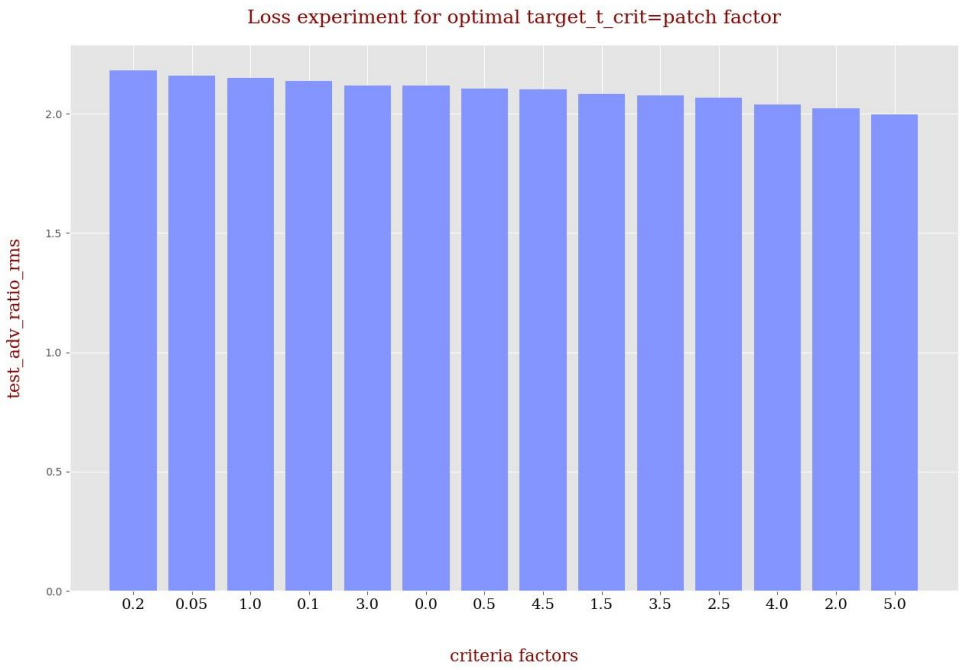

Fig 1. Results of the effect of the target_t criteria factor on test_adv_ratio_rms.

According to the evaluation metrics plotted on the graph, we will consider t_target_factor for the final training criterion to be around 0.2.

- **Experiment**: optimal ROT_FACTOR:

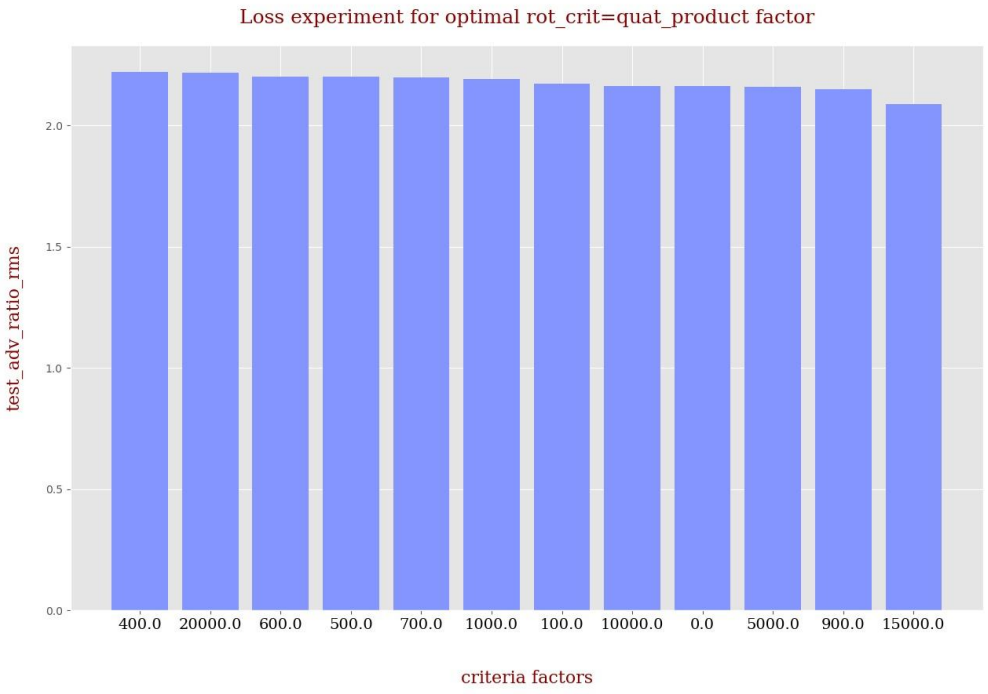

Fig 1. Results of the effect of the rotation criteria factor on test_adv_ratio_rms.
optimal FLOW_FACTOR:

According to the evaluation metrics plotted on the graph, we will consider t_target_factor for the final training criterion to be around 400.

- **Experiment:** optimal FLOW_FACTOR.

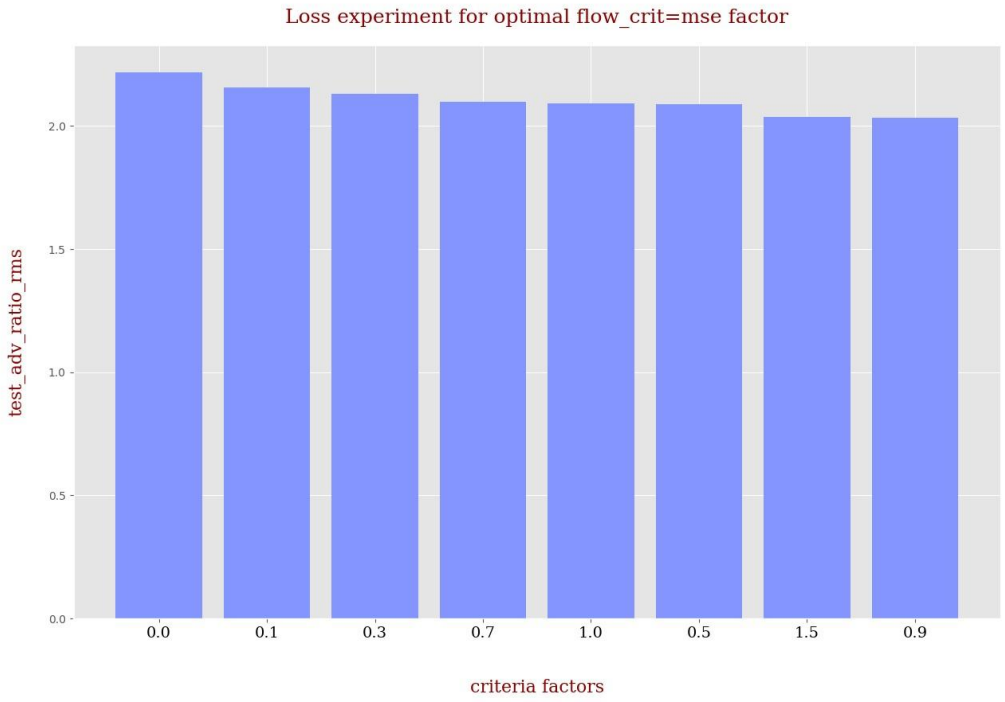

Fig 3. Results of the effect of the flow criteria factor on test_adv_ratio_rms.

As we can see from Fig. 3, Flow criterion as we defined it in this work does not help to produce a better training criteria with respect to our evaluation metrics.

- **Experiment:** optimal training criteria:

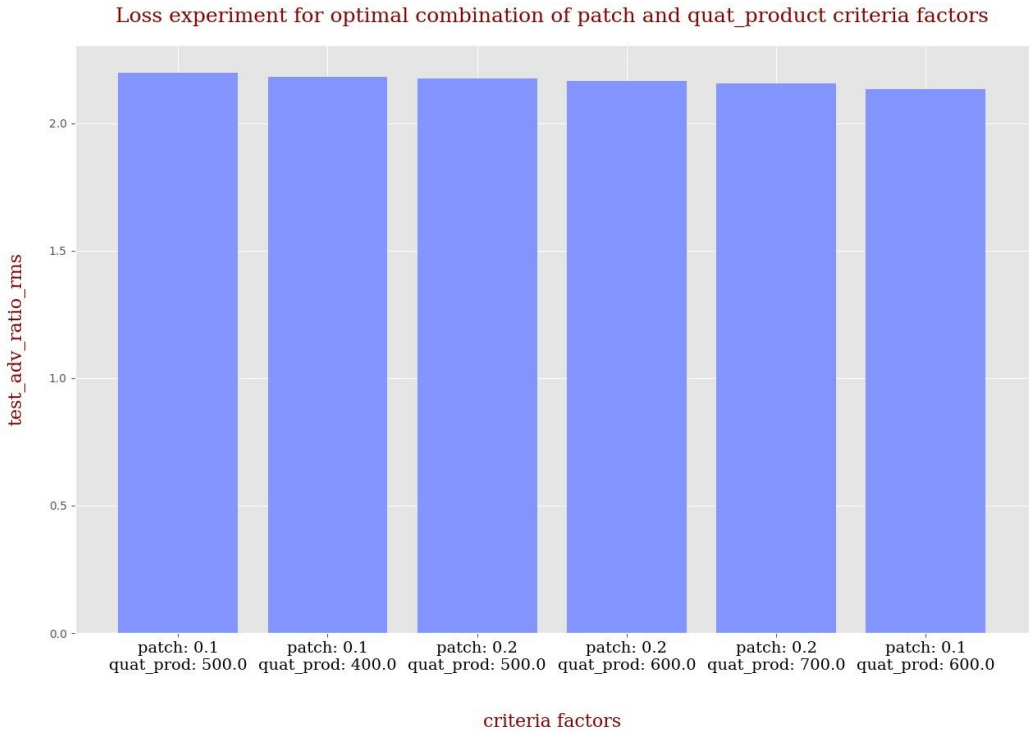

Fig 4. The best factor combinations received in Fig1-3.

As a result of series of loss experiments, we are going to test the final optimization criteria as a combination of ROT_CRIT=quat_product and T_TARGET_CRIT=patch, with weights of 500 and 0.1 respectively.

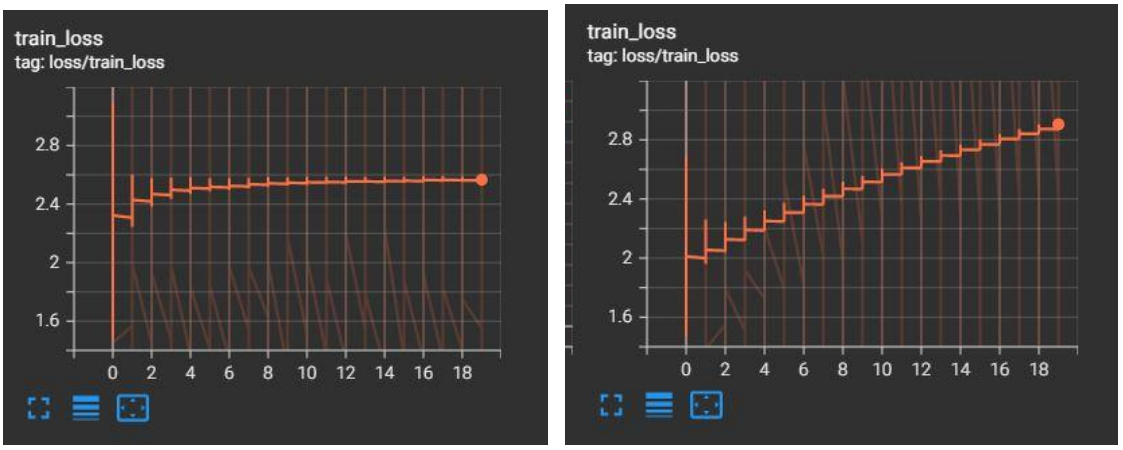

Fig 5. On the left side, you can see the loss before the adjustment made in the experiment, and on the right side after. Although the loss function is not smooth at least it increases in a stable manner, which indicates that learning improved.

2. Determining a suitable stochastic/deterministic setting:

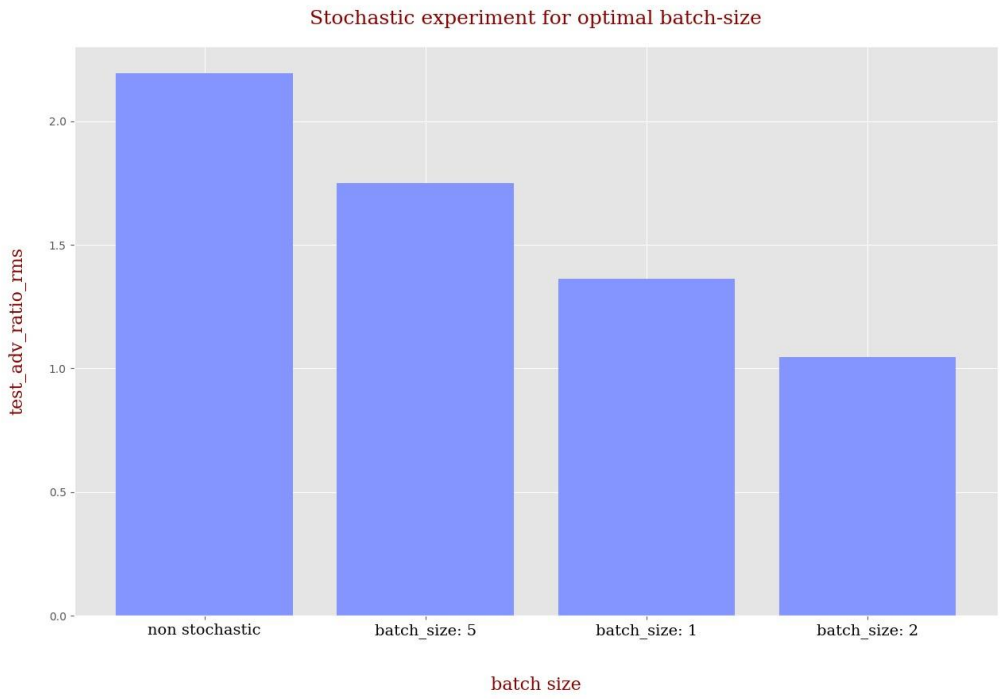

Fig 6. Results of stochastic method on test_adv_ratio_rms.

From the results, we have concluded that there is no advantage to the stochastic method. The probable explanation would be the fact that our optimization loss is still not the smoothest, while stochastic subgradients are known to have higher variance and as a result we have a very unstable learning process.

3. Hyperparameters tuning of the optimizers:

At our hands we have a smoothed optimization loss and the empirical results showed that the deterministic gradient update approach provides more stability to the learning process.
We will use both as base settings for the hyperparameter's tuning of the optimizers we are considering.

- **Experiment:** Scheduled PGD:

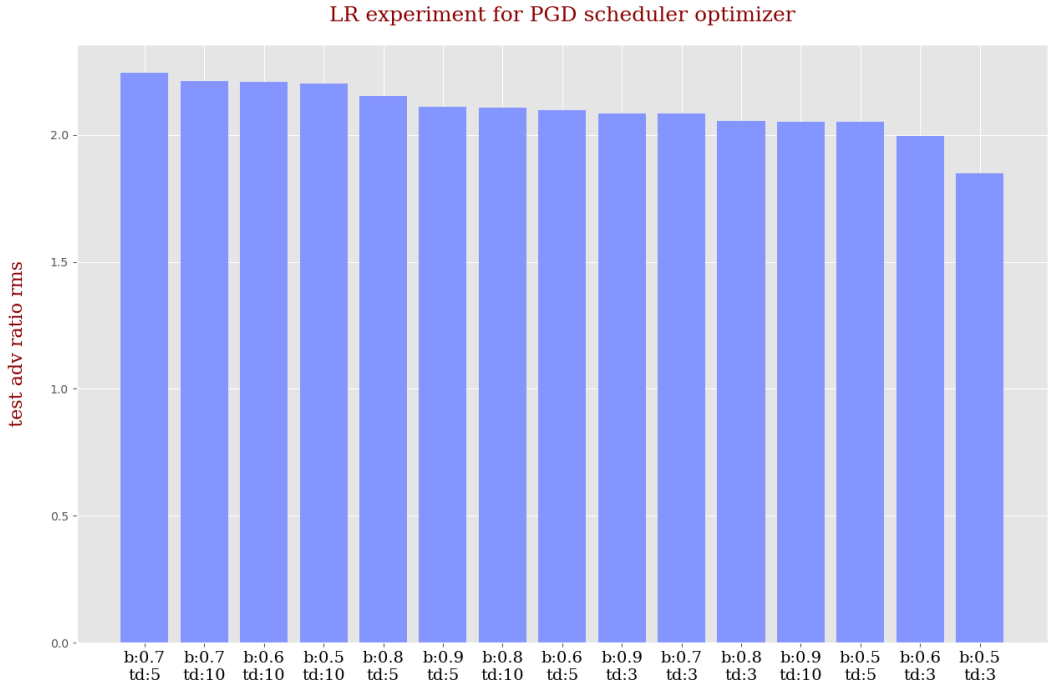

Fig 7. Results of the effect of the LR parameters on test_adv_ratio_rms.
The best hyperparameters are: beta=0.7 and t_decay=5.

- **Experiment:** Momentum PGD:

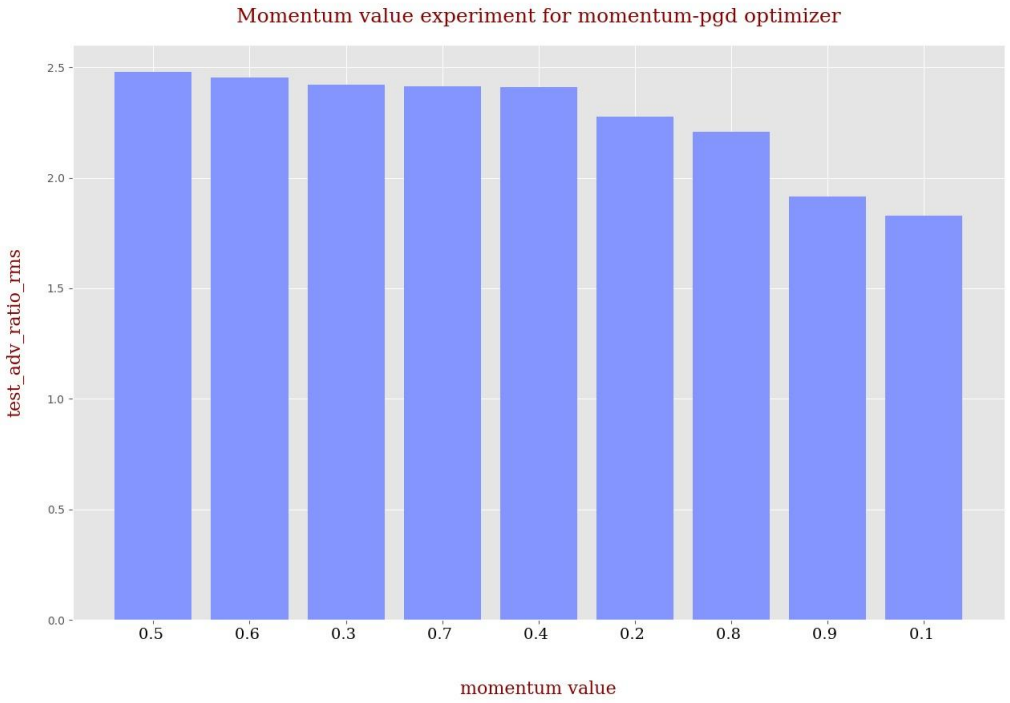

Fig 8. Results of the effect of the momentum parameter on test_adv_ratio_rms. Momentum=0.5 seems to be the most optimal.

- **Experiment:** RMSPROP PGD:

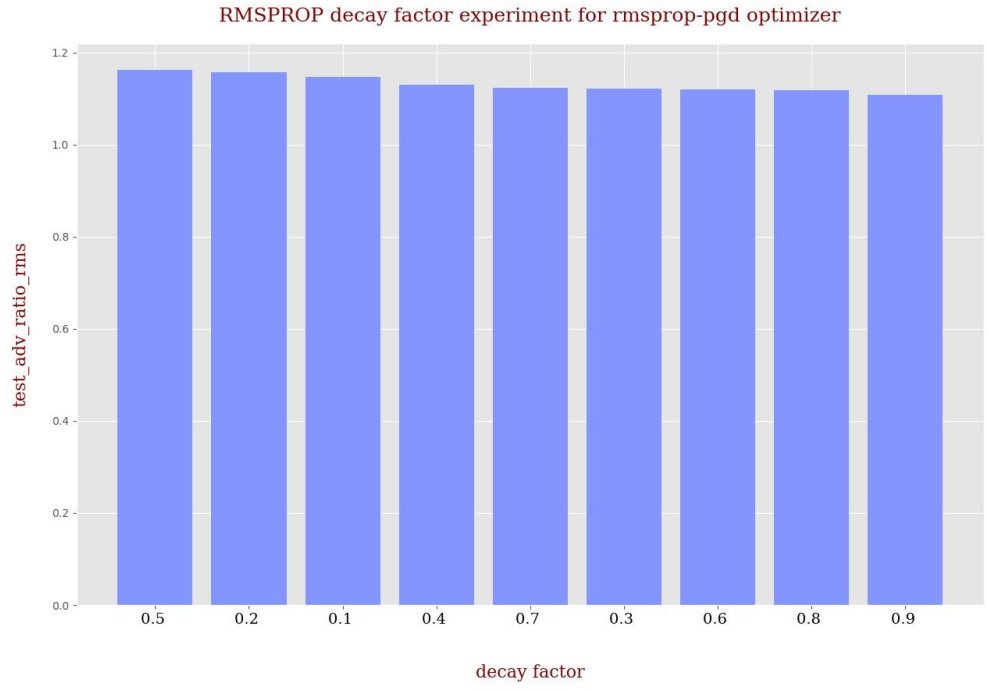

Fig 9. Results of the effect of the decay factor parameter on test_adv_ratio_rms.

Decay factor of 0.5 seems to be the most optimal, at the same all values seem to produce relatively the same results.

4. Determining the best optimizer:

- **Experiment:** Best optimizer **in-sample** evaluation experiment:

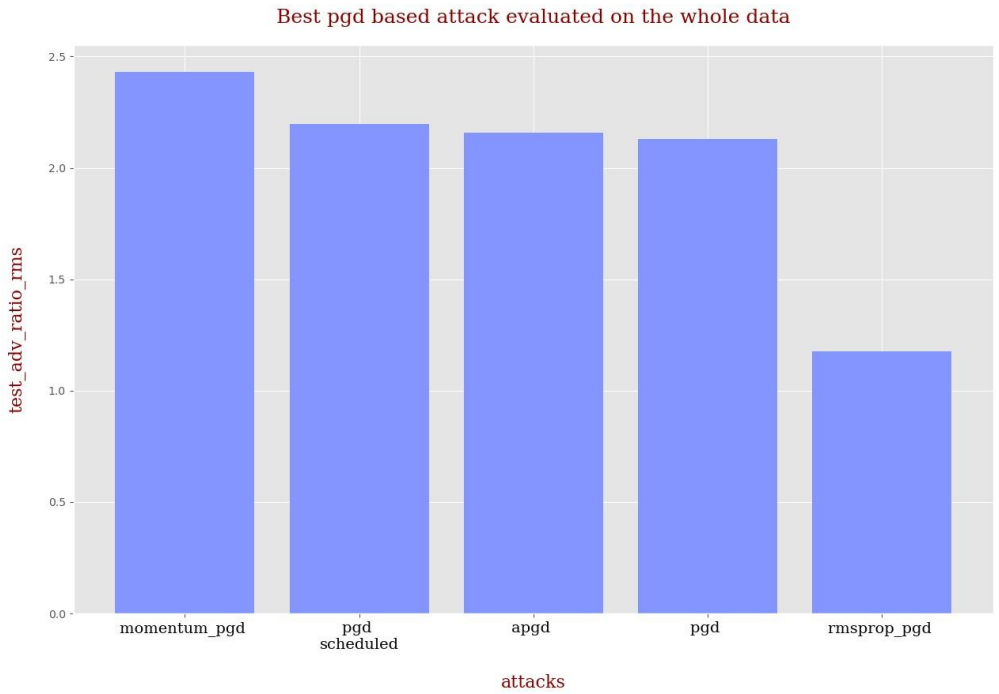

Fig 10. Comparison of the tuned optimizers.

It can be safely concluded that the Momentum PGD outperforms the rest of the algorithm at least during in-sample evaluation. We are going to test 3 best performing optimizers of the in sample experiment on the out sample setting.

- **Experiment:** Best optimizer **out-of-sample** evaluation experiment:

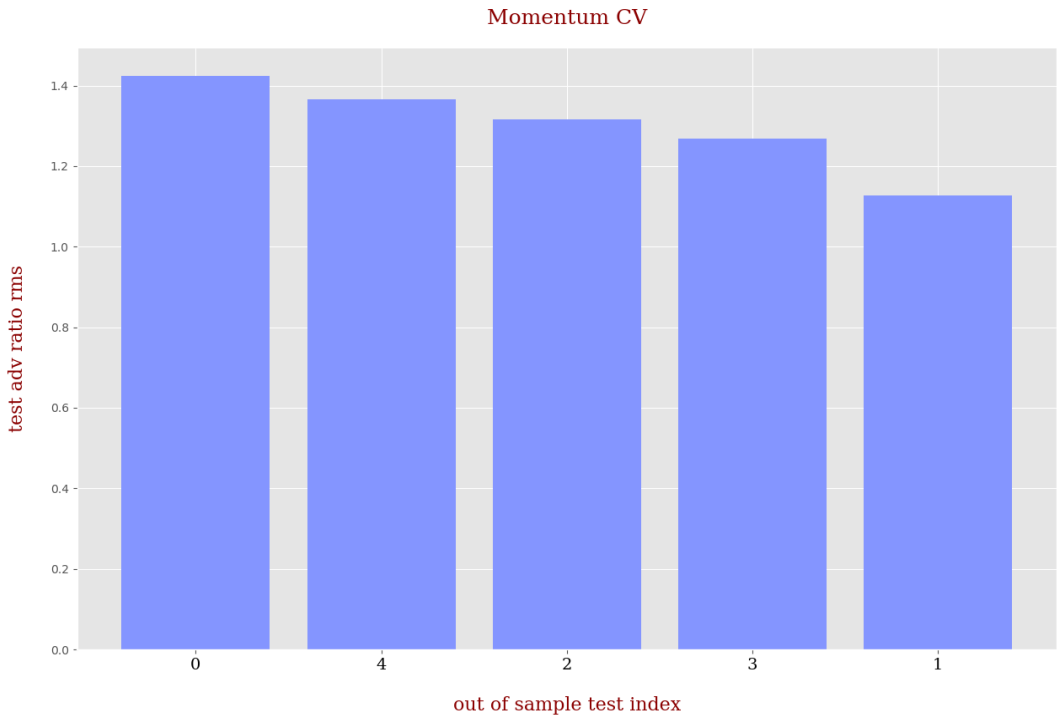

Fig11. Testing the generalization of Momentum PGD on different data folders.

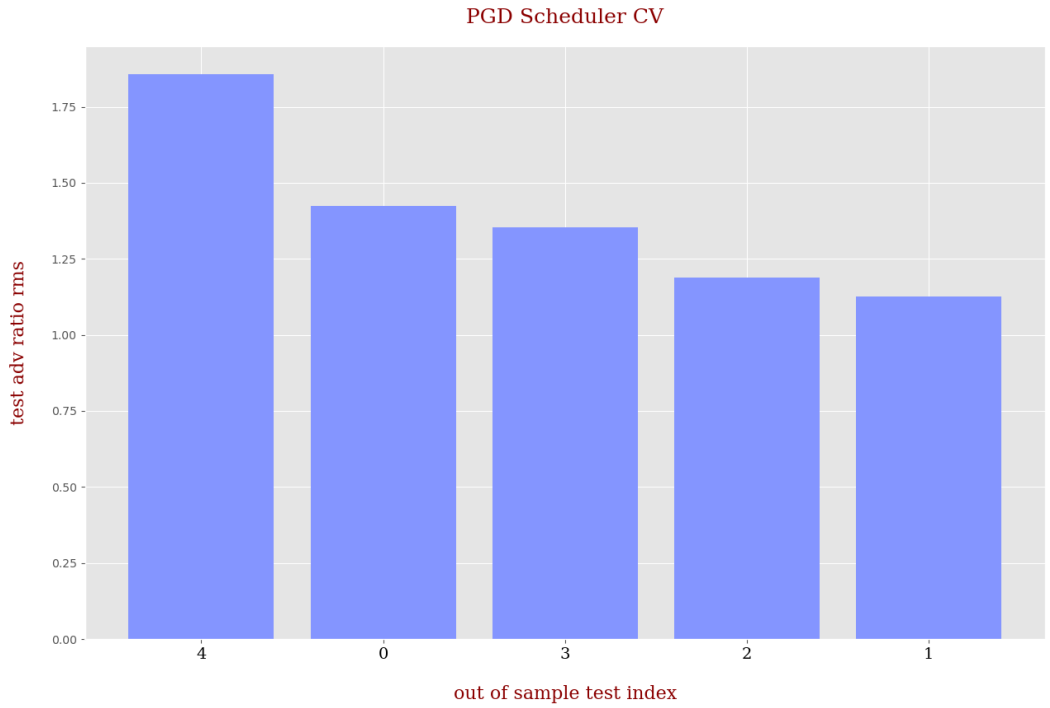

Fig 12. Testing the generalization of Scheduled PGD on different data folders.

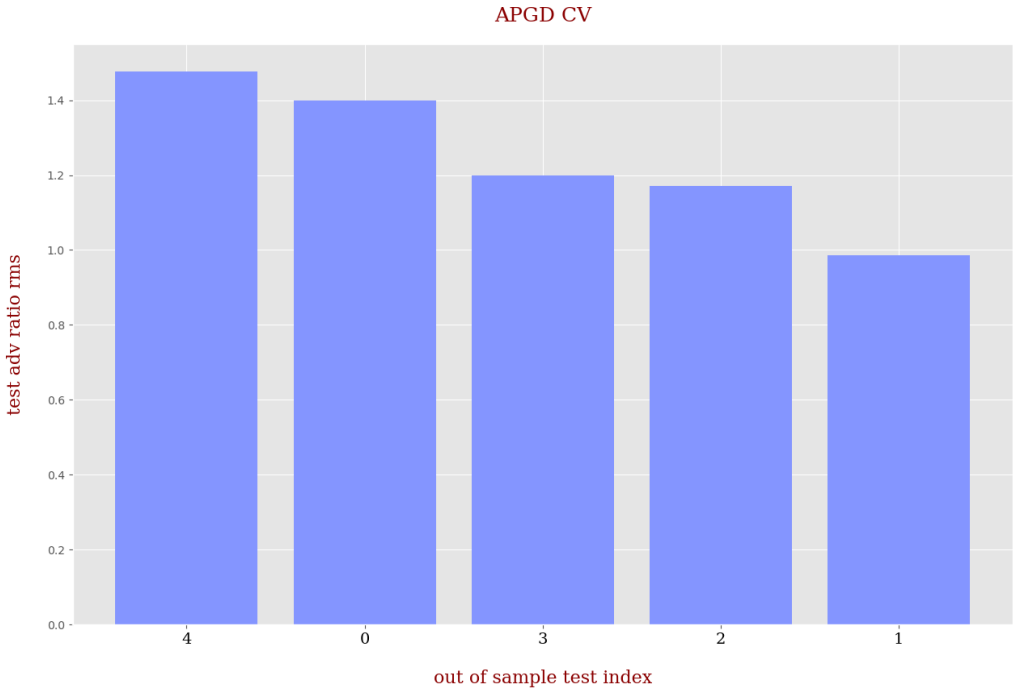

Fig 13. Testing the generalization of APGD on different data folders.

From Fig10-13 we can see that Momentum PGD has the most stable performance across different unseen test set data. While the highest value of the evaluation metric was provided by Scheduled PGD on test_folder=4. But there is a significant drop in other data split situations. We conclude that the Momentum PGD optimizer has better generalization properties.

5 . **Experiment:** Improving generalization properties

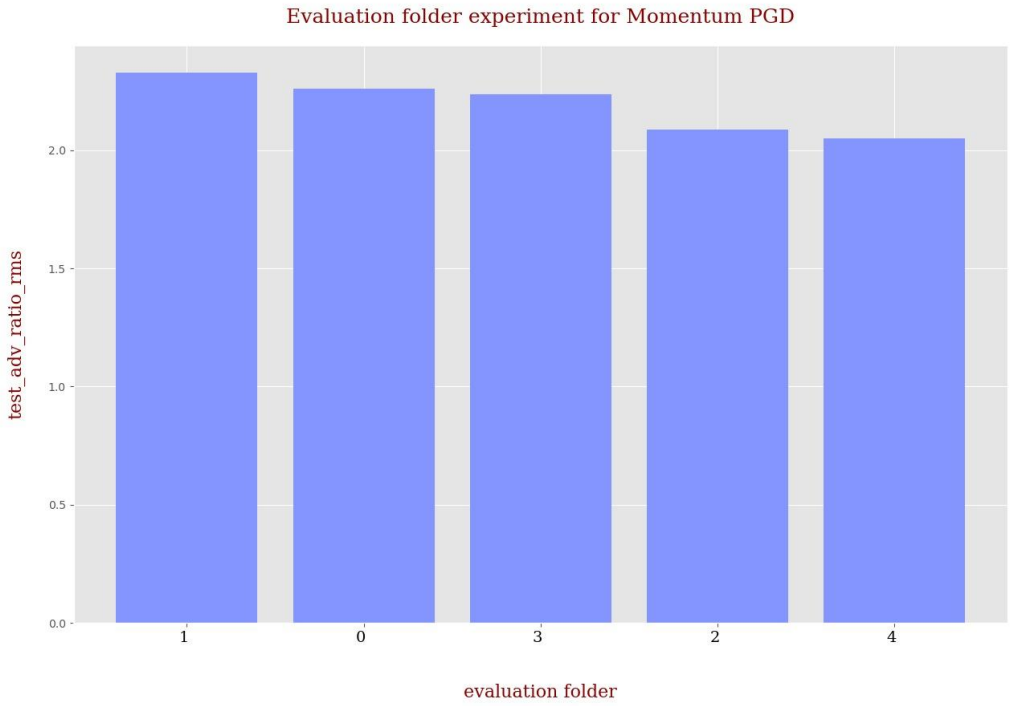

Fig 14. Testing evaluation data for Momentum PGD

As we explained in section IV, the final adversarial example is going to be trained using 4-1-5 data split. More specifically, one distinct folder will be chosen as evaluation data and the rest as train data.
This experiment helps us to make sure we are not going to harm the learning process, in case there are some unexpected data behaviors that can have unwanted effects during training.
As we can see all evaluation folders produce equally sufficient results with no significant differences between folders. We are going to train on the 4-1-5 split in which evaluation folder is scored highest result - 1.

6. Optimizers comparison table.

As we already defined above, the aggregation of the RMS values from the CSV result files are done with mean aggregation of the last frames's result of a trajectory.

Name/ Criterion	clean RMS	adv RMS	RMS ratio
Vanilla PGD	0.46	0.82	1.7826
Scheduled PGD	0.46	0.84	1.8260
Momentum PGD	0.46	0.93	2.0217
RMSProp PGD	0.46	0.5	1.0869
APGD	0.46	0.83	1.8043
Best Scheme	0.46	1.07	2.3260

7. Conclusions.

As a result of conducting our research and series of experiments we can state the optimal optimization scheme for creating Adversarial Examples that maximize the RMS of trajectory deviation of a VO system such as TartanVO.

First of all, the smoothest and **optimal training loss** in addition to MPRSM of deviations was comprised of T_TARGET criterion that we defined as 'patch' that has a weight of 0.1, and of ROT criterion that was defined as 'quat_product' that has a weight of 500.

Using a **stochastic optimization scheme** resulted in a less stable learning process and therefore was not used in the final scheme.

The best performing optimizer in sense of in sample and out of sample evaluation was Momentum PGD.

It looks like we managed to improve some of the results of this paper in terms that we came up with a training process that yielded better results than the proposed optimization scheme and implementations from the work.