Maynooth University
National University of Ireland Maynooth

Taighde Éireann
Research Ireland

A Framework for Property-driven Machine Learning in PyTorch

Thomas Flinkow

Department of Computer Science
Maynooth University

29th October 2025

About

Principles of Programming Research Group @ Maynooth University

https://www.cs.nuim.ie/research/pop/

Principles of Programming Research Group

Principal Researchers: Rosemary Monahan, Hao Wu, Barak A. Pearlmutter, Kevin Casey

3 PostDocs: Medet Inkarbekov, Ali Bukhari, Arshad Beg

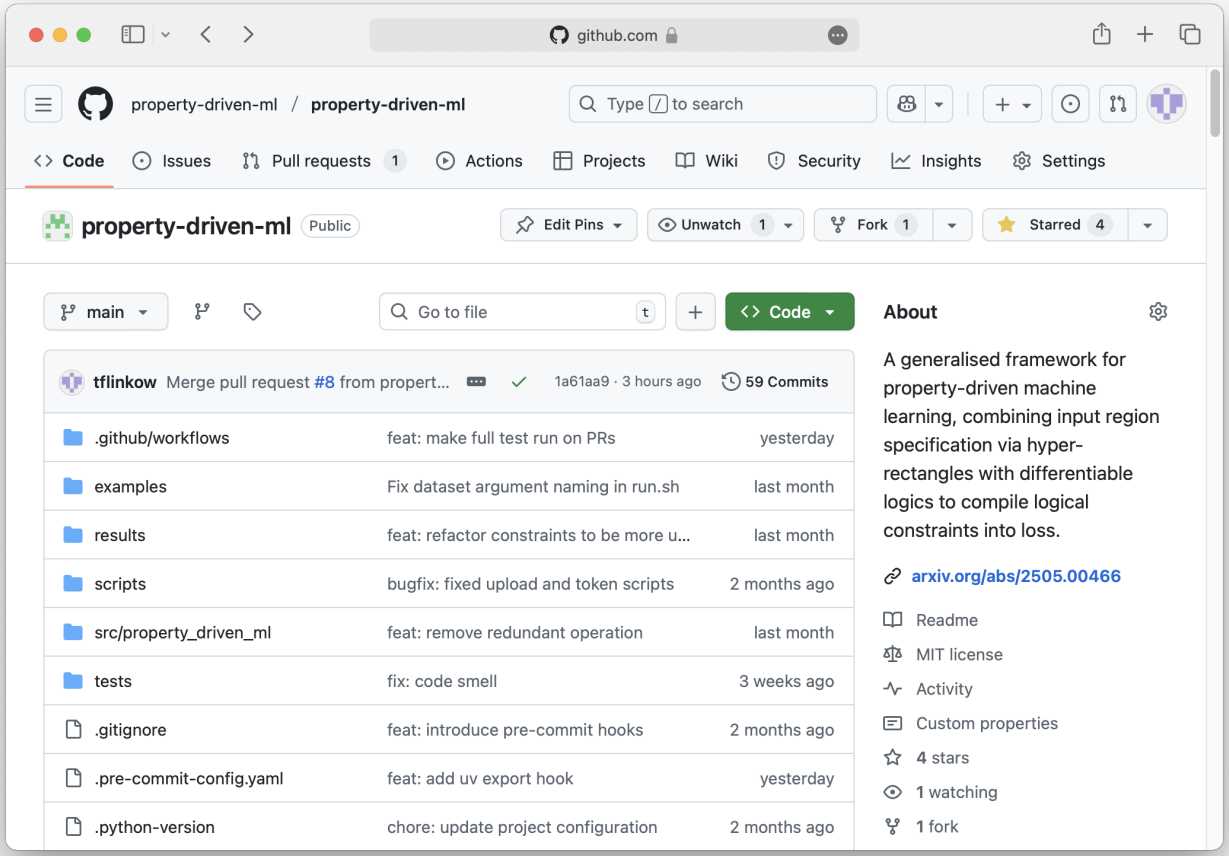
7 PhD Researchers: Oisín Sheridan, Dara MacConville, Thomas Flinkow, Ankit Jha, Dan Farcas, Karol Skowronski, Huan Zhang

Projects:

- MAIVV: Modular AI Verification and Visualisation (2021 – 2025, SFI)
- VerifAI: Writing Formal Requirements using AI (2024 – 2026, SFI ADAPT)
- VALU3S: Verification and Validation of Automated Systems' Safety and Security (2020 – 2023, EI and H2020 ESCEL)
- A Model Checker for Python (2022 – 2026, SFI CRT Foundations of Data Science)
- A Constructive Framework for Software Specification and Refinement (2012 – 2023, 2 × IRC)

Full Code on GitHub[1]

```
# Install the latest version from PyPI
pip install property-driven-ml
```



[1] https://github.com/property-driven-ml/property-driven-ml

Integration into Standard PyTorch Training Pipeline

```
logic = GoedelFuzzyLogic()
constraint = StandardRobustnessConstraint(epsilon=0.001, delta=0.1)

oracle = training.PGD(logic, steps=10, restarts=5, step_size=0.01)

optimiser = optim.Adam(model.parameters(), lr=0.001)
grad_norm = training.GradNorm(model, optimiser, lr=0.001, alpha=1.5)

for epoch in range(n):
  model.train()

  for _, (x, y) in enumerate(dataloader):
    # standard forward pass
    pred_loss = nn.CrossEntropyLoss()(model(x), y)

    # generate adversarial examples for logical constraint
    x_adv = oracle.attack(model, x, y, constraint)

    constraint_loss, constraint_sat = constraint.eval(
      model, x, x_adv, None, logic, reduction="mean"
    )

    # use GradNorm to balance losses
    grad_norm.balance(pred_loss, constraint_loss)
    optimizer.zero_grad()
```

Differentiable Logics[3]

The `property_driven_ml.logics.Logic` base class provides common logic operations:

- comparison: `LEQ(x, y)`, `EQ(x, y)` and `NEQ(x, y)`, `LT(x, y)` and `GT(x, y)`,
- n-ary[2] conjunction and disjunction `AND(*xs)` and `OR(*xs)`,
- implication `IMPL(x, y)` and equivalence `EQUIV(x, y)`.

Provide your own implementation or use one of many inbuilt logics:

```
import property_driven_ml.logics as logics

logic_boolean = logics.BooleanLogic()
logic_godel = logics.GoedelFuzzyLogic()
logic_lukasiewicz = logics.LukasiewiczFuzzyLogic()
logic_reichenbach = logics.ReichenbachFuzzyLogic()
logic_yager = logics.YagerFuzzyLogic()
logic_stl = logics.STL()
logic_dl2 = logics.DL2()
```

[2] via repeated application of associative & commutative binary operators `AND2(x, y)` and `OR2(x, y)` (DL2, fuzzy logics), or true n-ary operators (STL).

[3] https://github.com/property-driven-ml/property-driven-ml/tree/main/src/property_driven_ml/logics

Constraints[4]

The `property_driven_ml.constraints.Constraint` base class describes the common constraint API.

```python
class Constraint(ABC):

  @abstractmethod
  def __init__(self, device: torch.device):
    self.precondition: Precondition
    self.postcondition: Postcondition

  # usage:
  #   loss, sat = eval(N, x, x_adv, y, logic)
  def eval(
    self,
    N: torch.nn.Module,
    x: torch.Tensor,
    x_adv: torch.Tensor | None,
    y_target: torch.Tensor | None,
    logic: Logic,
  ) -> tuple[torch.Tensor, torch.Tensor]:
        ...
```

[4] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/constraints.py

Pre-[5] & Postconditions[6]

The `property_driven_ml.constraints.Precondition` base class describes the common precondition API.

```
class Precondition(ABC):

    @abstractmethod
    def get_bounds(self, *args, **kwargs)
      -> Tuple[torch.Tensor, torch.Tensor]:
        pass
```

The `property_driven_ml.constraints.Postcondition` base class describes the common postcondition API.

```
class Postcondition(ABC):

    @abstractmethod
    def get_postcondition(self, *args, **kwargs)
      -> Callable[[Logic], torch.Tensor]:
        pass
```

[5] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/preconditions.py

[6] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/postconditions.py

Example Preconditions: ϵ-ball[7]

We provide an implementation of ϵ-balls.

```python
class EpsilonBall(Precondition):

  def get_bounds(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    x = x.to(self.device)
    epsilon = self.epsilon * torch.ones_like(x, device=self.device)

    if self.std is not None:
      epsilon = epsilon / self.std

    lo = x - epsilon
    hi = x + epsilon

    return lo, hi
```

For arbitrary per-dimension lower and upper bounds, use the class `property_driven_ml.constraints.preconditions.GlobalBounds`.

[7] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/preconditions.py#L25C1-L72C22

Example Postconditions: Groups

(a) unique signs

(b) danger signs

(c) derestriction signs

(d) speed limit signs

(e) other prohibitory signs

(f) mandatory signs

Definition

'the sum of probabilities of a group should be very high or very low'

Example Postconditions: Groups[8]

Definition
'the sum of probabilities of a group should be very high or very low'

```python
class GroupPostcondition(Postcondition):

  def get_postcondition(
    self, N: torch.nn.Module, x_adv: torch.Tensor
  ) -> Callable[[Logic], torch.Tensor]:

    y_adv = F.softmax(N(x_adv), dim=1)
    sums = [torch.sum(y_adv[:, i], dim=1) for i in self.indices]

    return lambda logic: logic.AND(
      *[
        logic.OR(logic.LEQ(s, self.delta), logic.GEQ(s, 1.0 - self.delta))
        for s in sums
      ]
    )
```

[8] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/postconditions.py#L217C1-L265C10

Example Postconditions: Standard Robustness[9]

Definition

$$\|\mathcal{N}(x') - \mathcal{N}(x)\|_\infty \leq \delta$$

```python
class StandardRobustnessPostcondition(Postcondition):

  def get_postcondition(
    self, N: torch.nn.Module, x: torch.Tensor, x_adv: torch.Tensor
  ) -> Callable[[Logic], torch.Tensor]:

    y, y_adv = N(x), N(x_adv)

    diff = F.softmax(y_adv, dim=1) - F.softmax(y, dim=1)

    return lambda logic: logic.LEQ(
      LA.vector_norm(diff, ord=float("inf"), dim=1), self.delta
    )
```

[9] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/postconditions.py#L41C1-L82C10

Example Postconditions: Lipschitz Robustness[10]

Definition

$$\|\mathcal{N}(\boldsymbol{x}') - \mathcal{N}(\boldsymbol{x})\|_2 \leq L\|\boldsymbol{x}' - \boldsymbol{x}\|_2$$

```
class LipschitzRobustnessPostcondition(Postcondition):

  def get_postcondition(
    self, N: torch.nn.Module, x: torch.Tensor, x_adv: torch.Tensor
  ) -> Callable[[Logic], torch.Tensor]:

    y, y_adv = N(x), N(x_adv)

    diff_x = LA.vector_norm(x_adv - x, ord=2, dim=1)
    diff_y = LA.vector_norm(y_adv - y, ord=2, dim=1)

    return lambda logic: logic.LEQ(diff_y, self.L * diff_x)
```

[10] https://github.com/property-driven-ml/property-driven-ml/blob/main/src/property_driven_ml/constraints/postconditions.py#L85C1-L120C64

Addressing Practical Problems: GradNorm[11]

Problem 1: Loss Balancing

$$\theta^* = \arg\min_{\theta} \mathop{\mathbb{E}}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}} \left[\alpha \underbrace{\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}; f_\theta)}_{\text{prediction loss}} + \beta \underbrace{\max_{\boldsymbol{x}' \in (\!| \mathcal{P}(\boldsymbol{x}) |\!)} [\![\mathcal{Q}(f(\boldsymbol{x}))]\!](\boldsymbol{x}', \boldsymbol{y}; f_\theta)}_{\text{worst-case constraint loss}} \right]$$

It is *crucial* to find close to optimal values for α and β!

Solution: Adaptive Loss Balancing with GradNorm

- Key idea: $\alpha(t)$ and $\beta(t)$.

```
from property_driven_ml.training import GradNorm

grad_norm = GradNorm(N, device, optimiser, lr, alpha=1.5)

# instead of
#   loss = alpha * pred_loss + beta * constr_loss
grad_norm.balance(pred_loss, constr_loss)
```

[11] Z. Chen et al. (2018). 'GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks'.

Addressing Practical Problems: AutoPGD[12]

Problem 2: Attack Failures

Approximately solve $\max_{x' \in (\!| \mathcal{P}(x) |\!)} [\![\mathcal{Q}(f(x))]\!](x', y; f)$ via Projected Gradient Descent (PGD), i.e.

$$x^0 \sim (\!| \mathcal{P}(x) |\!),$$
$$x^{t+1} = \Pi_{(\!| \mathcal{P}(x) |\!)} \left[x^t + \eta \operatorname{sign}(\nabla_x [\![\mathcal{Q}(f(x))]\!](x^t, y; f)) \right]$$

Attack success has been shown to strongly depend on η.

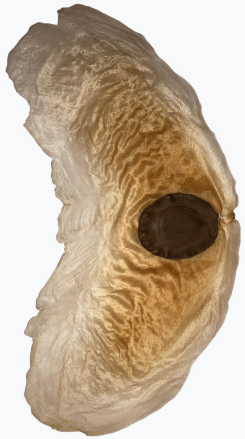
Solution: Adaptive Attacks with AutoPGD

- Key idea: $\eta(t)$.

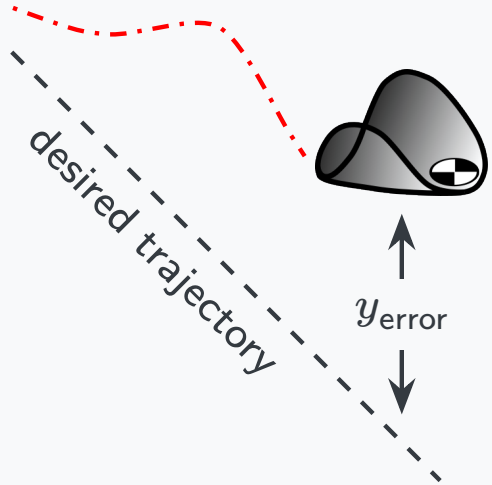
```
from property_driven_ml.training.attacks import PGD, APGD

vanilla_pgd = training.PGD(logic, device, steps, restarts, step_size)
auto_pgd = training.APGD(logic, device, steps, restarts)
```

[12] F. Croce et al. (2020). 'Reliable Evaluation of Adversarial Robustness with an Ensemble of Diverse Parameter-Free Attacks'.

Experimental Results: Alsomitra



(a) An *Alsomitra macrocarpa* seed, capable of stable flight over long distances.



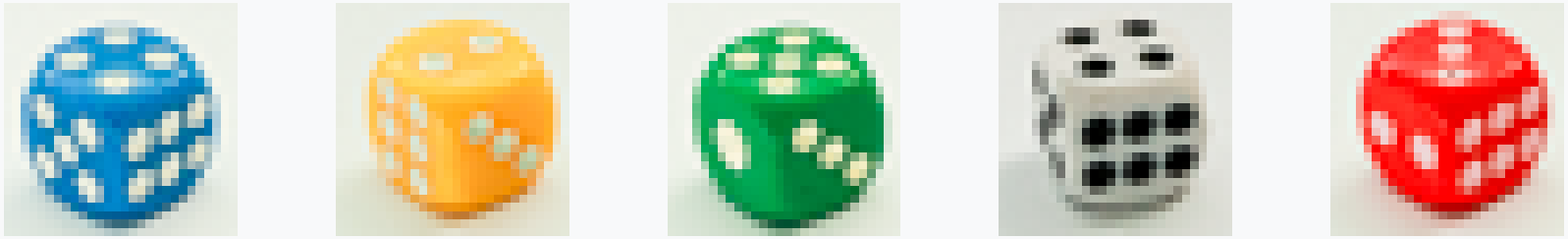
(b) The desired linear trajectory of the *Alsomitra*-inspired drone.

ϕ: *If the drone is above and close to the line, pitching down quickly and moving fast, the network will always make the drone pitch up.*

Logic	RMSE	CAcc (%)	CSec (%)
Baseline	3.6111×10^{-4}	0.00	0.00
DL2	1.2287×10^{-3}	100.00	95.31
Fuzzy logic	1.1632×10^{-3}	100.00	92.19

Experimental Results: Dice

Example: Multi-label classification of dice faces.



Constraint: *Predictions must physically be possible, i.e., network must not predict two faces at the same time that are on opposite sides of the die.*

Logic	**PAcc**	**CAcc**	**CSec**	**VSat** ($\epsilon = {}^{4}/_{255}$)
Baseline	84.6 %	98.5 %	4.4 %	15.9 % $({}^{10}/_{63})$
DL2	87.5 %	98.5 %	27.9 %	43.1 % $({}^{22}/_{51})$
Our Logic	81.6 %	100 %	100 %	100 % $({}^{68}/_{68})$

Thank you!
Any questions?