A Proof of Lemma 3.1

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ denote the set of points associated to a task, and a untrained policy π that collects a trajectory of length T by iteratively drawing points in $\mathcal{D}$ uniformly without replacement. We note $(z_1, \dots, z_T)$ the sequence of function values observed during the trajectory, and $(r_1, \dots, r_T)$ the sequence of rewards obtained. We remind that $r_t = \max\limits_{1 \leq \ell \leq t} z_\ell$, and we consider that r_t (and z_t) is informative if $z_t = \max\limits_{1 \leq \ell \leq t} z_\ell$. We want to compute the probability of obtaining exactly m informative rewards by sampling a trajectory from π. As each sequence is equiprobable, we do so by counting the number of sequences leading to m informative rewards. We assume for now that the values in $\{y_i\}_{i=1}^n$ are pairwise distinct.

We first note that there are $\binom{n}{T}$ ways to choose the $T \leq n$ points composing a trajectory of length T from $\mathcal{D}_n$. We now consider that the set of T sampled value functions is fixed (without loss of generality we note this set $V = \{v_\ell\}_{1 \leq \ell \leq T}$). For $1 \leq k \leq T$, we note $C(k, T)$ the number of ways to order them such that the resulting trajectory $(z_1, \dots, z_n)$ contains exactly k informative values, and we will give a recurrent formula for $C(k, T)$.

We can see that $k = 1$ necessarily implies that $z_1 = \max\limits_{1 \leq \ell \leq T} v_\ell$. Thus there are $(T-1)!$ ways to order the remaining elements of the trajectory $\{z_\ell\}_{2 \leq \ell \leq n}$, hence $C(1, T) = (T-1)!$. On the other hand, $k = T$ informative rewards are only obtained for when the element of V are sorted in increasing order, i.e. with $(z_1 < z_2 < \dots < z_T)$, and therefore $C(T, T) = 1$.

Finally, for $2 \leq k < T$ we can establish a recurrence relation by reasoning on z_n, the last element of the sequence:

- If $z_T = \max\limits_{1 \leq \ell \leq n} v_\ell$, then z_T is informative, and there remains to count the number of ways to order the first $T-1$ elements $V \backslash \{z_T\}$ to get $k-1$ informative steps, which is by definition $C(k-1, T-1)$.
- If $z_T = v_j < \max\limits_{1 \leq \ell \leq T} v_\ell$, then z_T is not informative. We note that there are $T-1$ choices for such v_j, and for each choice of v_j there remains to order $V \backslash \{v_j\}$ such that the resulting sub-trajectory $(z_1, \dots, z_{T-1})$ has exactly k informative values. There are therefore $(T-1)C(k, T-1)$ trajectories with k informative rewards such that $z_T \neq \max\limits_{1 \leq \ell \leq T} v_\ell$

From this analysis we get that $C(k, T) = C(k-1, T-1) + (T-1)C(k, T-1)$.

This relation, along with boundary values $C(1, T) = (T-1)!$ and $C(T, T) = 1$, allow to identify $C(k, T)$ as the Stirling number of first kind $\left[{T \atop k}\right]$. This number notably corresponds to the number of permutations in S_T made of exactly k cycles.

Finally, the number of trajectories of length T that can be obtained from $\mathcal{D}$ and having k informative rewards, is given by $\frac{\binom{n}{T} \times C(k,T)}{\binom{n}{T} \times T!} = \frac{1}{T!}\left[{T \atop k}\right]$, which is equal to the probability of getting k cycles in a permutation sampled randomly in S_T. Therefore the expected number of informative rewards in a trajectory of length T is equal to the average number of cycles in a permutation sampled uniformly in S_T, which is a known result [59], thus is equal to the T^{th} harmonic number $H_T = \log T + \mathcal{O}(1)$.

B Experimental setup

In this section we give more details about the experimental setup. In particular, we add details about the experiments environments and baselines where needed. We also give some more intuition on the NAP training and data augmentation scheme used. Finally we list all important hyperparameters as well as the hardware used in our experiments.

B.1 Tasks

HPO-B As mentioned in Section 4 for this experiment we choose a subset of tasks from the HPO-B benchmark set. HPO-B is a collection of HPO datasets first grouped by search space. Each search space corresponds to the hyperparameters of a particular model, e.g. SVM, XGBoost, . . . Each such search space then has multiple associated datasets split into a set for training, validating and for testing. The multi-task RL setting from Section 3.1 states that we limit ourselves to MDPs sharing state and action spaces across tasks hence we don't train NAP on multiple search spaces at the same time. We train one model per search space, being careful to choose a search space for each type of underlying model. When multiple search spaces related to the same underlying model we choose the search space with the least amount of total data in order to focus on the low-data regime as much as possible. We pick the following search spaces: 5860 (glmnet), 4796 (rpart.preproc), 5906 (xgboost), 5889 (ranger), 5859 (rpart), 5527 (svm). Refer to Table 3 in Pineda-Arango et al. [44] for more details.

Electronic Design Automation Following the description in Section 4 we search for the sequence of operators to optimise an objective combining two metrics associated with circuit performance, the area and the delay. The area is the number of gates in the mapped netlist, while the delay corresponds to the length of the longest directed path in the mapped netlist. As these two metrics are not directly commensurable, we normalise them by the area and delay obtained when running twice the reference sequence `resyn2` [51] made of 10 operators, as follows:

$$f_{\text{EDA}}(\texttt{seq}) = \frac{\text{Area}(\texttt{seq})}{\text{Area}(2 \times \texttt{resyn2})} + \frac{\text{Delay}(\texttt{seq})}{\text{Delay}(2 \times \texttt{resyn2})}$$

where `seq` is a sequence of operators from `ABC`.

B.2 Baselines

OptFormer The results reported by Chen et al. [41] are for the continuous HPO-B benchmark where XGBoost models approximate the black-box functions from the discrete points. To evaluate every approach in a fairer and more robust manner, we instead focused on the original discrete setup of HPO-B which uses only true values from the black-boxes. We relied on the shared OptFormer checkpoint trained and validated on HPO-B. We adapted the inference code for the discrete setting and noticed that the default parameters were set to NaPolicy.DROP which drop the missing values in the HPO-B benchmark and removes the additionnal "na" columns. We switched it to NaPolicy.CONTINUOUS to keep every column leading to better performances. We also had to increase the maximum number of tokens possible in a trajectory from 1024 to 2048.

B.3 NAP training

We conduct our experiments in a consistent manner. We define a training , validation and test sets that are kept the same for each method. We also ensure reproducibility by ensuring random seeds are similar across experiments and initial points too. Finally we run the tests on 10 different random seeds in all experiments and 5 for HPO-B as there are only 5 seeds available for other baselines.

Data augmentation When training on tasks with low number of points in each dataset, we perform data augmentation using Gaussian processes. On each dataset we fit an exact GP and during training we sample a new dataset directly from the posterior of that GP. This has an interpolating effect such that between original data points, the GP can make predictions that are roughly realistic. Training on these augmented datasets sampled from GP posteriors is handy in practice because it helps to create datasets of the desired size (we can sample as many data points as we like) and datasets that resemble the original one, provided we don't sample from the GP posterior too far from the original inputs. In practice we sample inputs points from the original dataset, add to them a random uniform perturbation and sample from the GP posterior at those new points.

Value function The value function takes as input t/T and the best y value observed in $\mathcal{H}_t$.

B.4 Hyperparameters

We share in Table 2 a comprehensive list of the hyperparameters used during training and inference. More details can be found in the associated code repository. We want to underline that none of these presented hyperparameters were tuned. This is only fair as we did also not optimise any

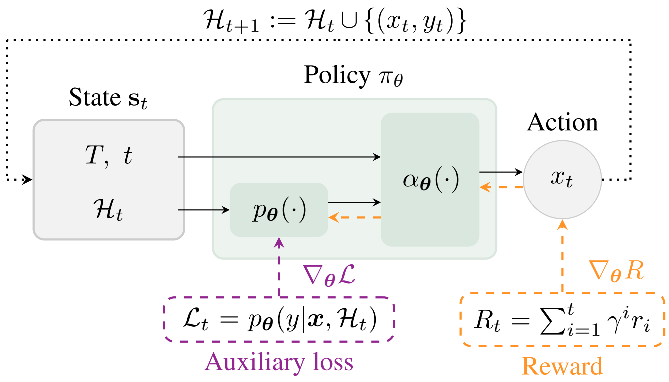
$\mathcal{H}_{t+1} := \mathcal{H}_t \cup \{(x_t, y_t)\}$
State $\mathbf{s}_t$
Policy π_θ
Action
$T,\ t$
$\mathcal{H}_t$
$p_{\boldsymbol{\theta}}(\cdot)$
$\alpha_{\boldsymbol{\theta}}(\cdot)$
x_t
$\nabla_{\boldsymbol{\theta}}\mathcal{L}$
$\nabla_{\boldsymbol{\theta}} R$
$\mathcal{L}_t = p_{\boldsymbol{\theta}}(y|\boldsymbol{x}, \mathcal{H}_t)$
$R_t = \sum_{i=1}^{t} \gamma^i r_i$
Auxiliary loss
Reward

Figure 3: Summary of our proposed Neural Acquisition Process (NAP) architecture. At iteration $t = 1, \dots, T$ the state consists of $\mathbf{s}_t = \{\mathcal{H}_t, t, T\}$, respectively the history of collected points, the current iteration index and the total budget. The action is sampled from the policy $x_t \sim \pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s}_t)$. For a set of locations $\boldsymbol{x} \subseteq \mathcal{A}$, the gradients flow back to parameters $\boldsymbol{\theta}$ from both the cumulative regret returns R_t and the auxiliary likelihood loss $\mathcal{L}_t$.

hyperparameters from the other baseline methods. Hyperparameters that are shared between our method and previous baselines are simply taken as is from their respective codebases as we assume the authors have already tuned them. We use the same for all experiments. Note that the ablation study in C.1 can be seen as a simple tuning of the parameter λ introduced to weight both losses in NAP.

Table 2: List of used hyperparameters in NAP.

PPO	
Learning rate for gradient descent	$3 \cdot 10^{-5}$
Learning rate decay	Linear decay to 0 over 2000 iterations
Number of training PPO iterations	2000
Horizon of episodes used in training	24
Trajectories collected per iteration	60
Total numbers of transformer updates	90,000
Minibatch size	32
Weight of auxiliary loss in total loss (λ)	1.0
Weight of the value function loss in total loss	1.0
Generalised Advantage Estimator-λ	0.98
Discount factor γ	0.98
Clip of importance sampling ratio ϵ	0.15
L2 gradient clipping	0.5
BO environment	
Range of Uniform perturbation for data augmentation	$[-0.05, 0.05]$
Number of random initial points	0 during training, 5 during validation
Architecture	
Number of buckets in the output histogram	1000
Point-wise feed-forward dimension of Transformer	1024
Embedding dimension of Transformer	512
Number of self-attention layers of Transformer	6
Number of self-attention heads of Transformer	4
Dropout rate of Transformer	0.0
Softmax temperature to compute π from α	0.1 for training, argmax otherwise
Value function network	Linear(2, 512), TanH, Linear(512, 1)

B.5 Hardware

We train our model on a machine with 4 GPUs Tesla V100-SXM2-16GB and an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz with 88 threads with an average training time of approximately 10 hours per experiment.

C Additional Results

C.1 Ablation

In this section we perform an ablation study to answer some of the questions that arise naturally from the proposed framework. Notably, is end-to-end training useful? Does the auxiliary loss help? Is it useful to learn the acquisition function and not simply a surrogate model? We run an additional experiment on a dataset from the HPOBench benchmark [60]. For reference, we detail the methods of this ablation in Table 3 for easier comparison.

Table 3: Variations of NAP and their components.

	$p_{\boldsymbol{\theta}}$	$\alpha_{\boldsymbol{\theta}}$	RL	Supervision	End-to-end
NAP (ours)	✔	✔	✔	✔	✔
Pre-NAP	✔	✔	✔	✔	✘
NAP-RL	✔	✔	✔	✘	✔
NP-EI	✔	✘	✘	✔	✘

This study uses datasets of HPOBench [60] for XGBoost hyperparameters, similarly to Volpp et al. [13]. It consists of hyperparameter configurations and their associated accuracy of the XGBoost model on a classification task.

We analyse the effects of training end-to-end with the introduced auxiliary loss to better understand the method's strengths and limitations. Six hyperparameters (learning rate, regularisation, etc.) and 48 classification tasks exist. We have 1000 hyperparameter configurations evaluated for each task and the corresponding XGBoost model accuracy, creating 48 datasets of different black-box functions. We meta-train on 20 datasets, validate on 13 and test on 15.

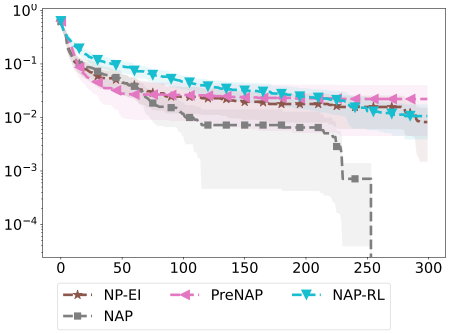

Figure 4: Average regret vs iterations on HPOBench dataset for XGBoost. Error bars are confidence intervals across ten runs.

Is it worthwhile to learn a meta-acquisition function? A valid question to ask is whether or not we need to meta-learn an acquisition function. There exist popular methods for meta-learning models, as discussed in the main paper, so one could use such a surrogate model and apply it directly in BO, using its posterior distribution to compute an acquisition function. Such an approach matches our baseline NP-EI, where we train a PFN [11] on the same training data and apply it directly in BO. Comparing NAP and NP-EI, Figure 4 reveals the benefits of learning the acquisition function as part of the end-to-end architecture instead of using a pre-defined expected improvement acquisition function.

Does training end-to-end using the auxiliary loss help? Training end-to-end, we check that using supervised information through the auxiliary loss helps compared to end-to-end training with only the reinforcement learning reward. We name the latter NAP-RL and show the results in Figure 4 which suggests that indeed, the inductive bias introduced with the auxiliary loss is beneficial for downstream performance.

Is end-to-end training beneficial? To investigate this question, we first pre-train the probabilistic model part of a NAP with the supervised auxiliary loss and then use PPO to update the meta-acquisition function while keeping the rest of the weights frozen. We denote this method as PreNAP as the architecture is partly pre-trained. Figure 4 shows that PreNAP . Thus, training jointly end-to-end NAP with both objectives translate into improved regret at test time, validating our hypothesis in Section 3.

C.2 HPO-B per search space results

Additionally to the aggregated regret plot in Figure 2 of Section 4 we show ranks and regrets per search space.

Figure 5 shows the regret for each method per search space, aggregated on all the test datasets of that search space and across 5 random seeds. Figure 6 shows the relative rank of each method (lower is better) per search space, aggregated across all the test datasets of that search space and 5 random seeds.

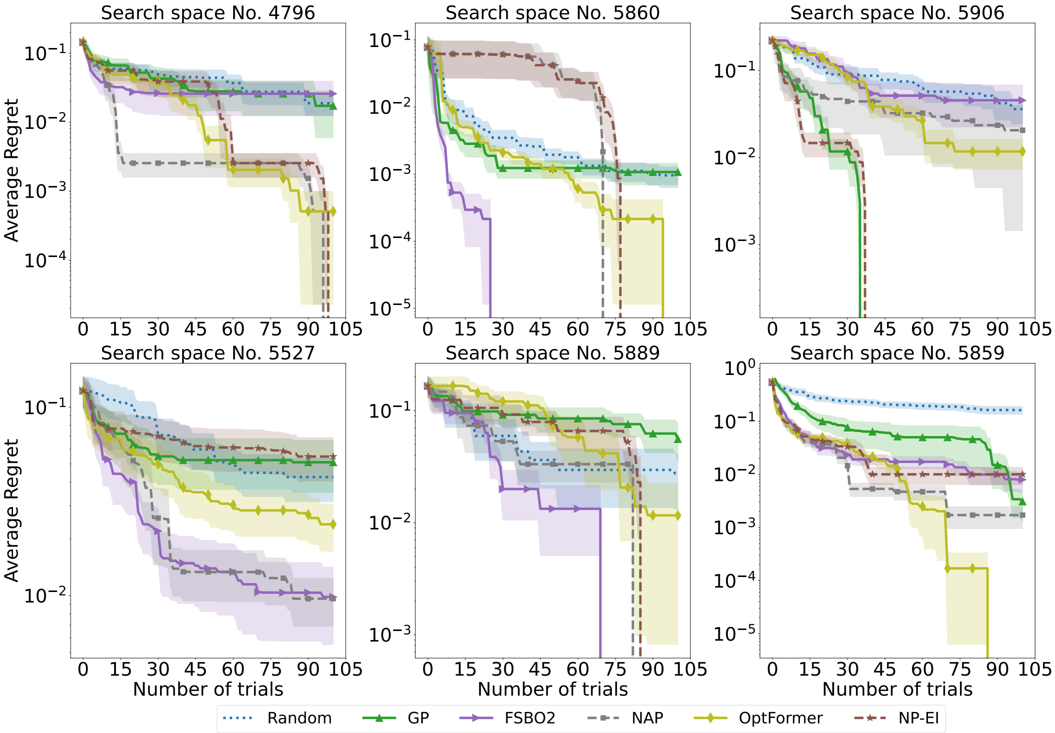

Figure 5: Average regret vs. BO iterations on each search space with 5 initial points. For each method, error bars show confidence intervals computed across 5 runs.

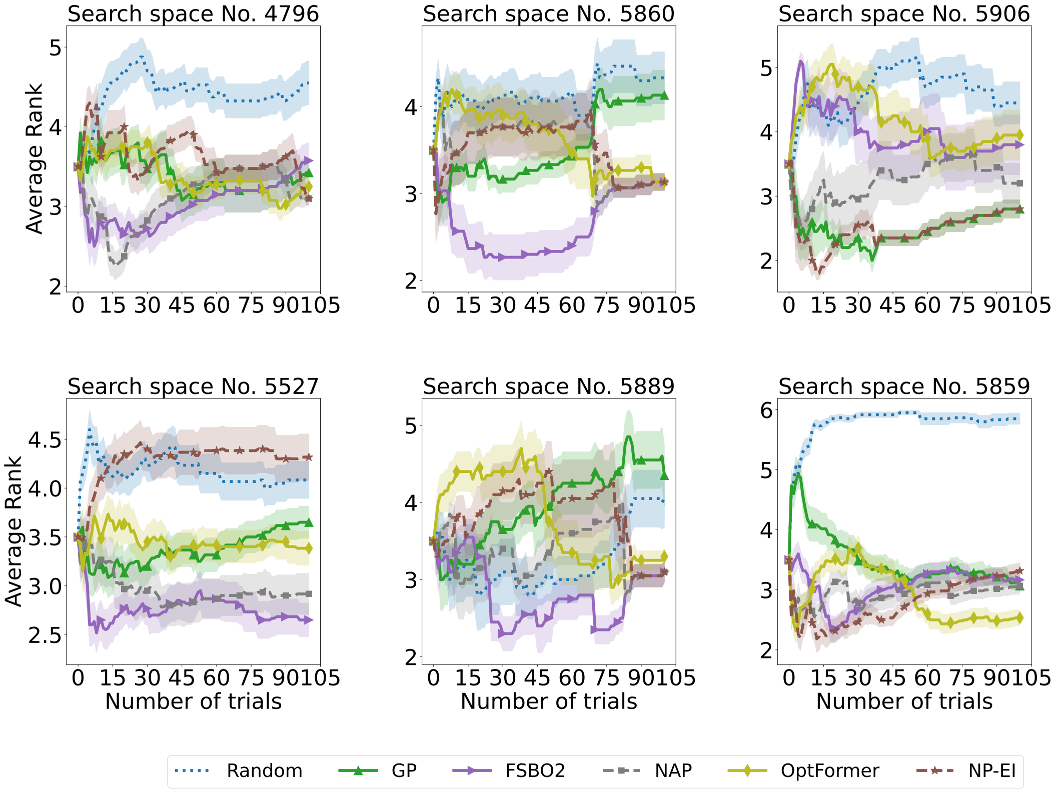

Figure 6: Average rank (lower is better) vs. BO iterations on each search space with 5 initial points. For each method, error bars show confidence intervals computed across 5 runs.

C.3 Time Comparison

While in our BO setting we assume that querying the black-box objective is the main bottleneck (hence our focus on sample efficiency), it is also interesting to analyse the time efficiency of algorithms to gain another perspective on various methods. In this section we summarise some average test running time results to provide a different point of view. In short, we see that even though some methods require offline pre-training (e.g. MetaBO, FSBO, NAP), the time required to evaluate points in the objective function out-weights this cost. Hence when measuring the total running time, it makes little difference if some methods require this pre-training.

For example, in the antibody experiment, evaluating the objective is costly. This is true both in terms of monetary and time costs as evaluating the objective could mean manufacturing the molecule and testing it in a wet-lab experiment. In our experiments we use a simulator as a proxy. The HPO-B black-box function is also very expensive to evaluate as it relies on training and testing several models. We use result files posted on the authors' repository which only contain black-box values for some baselines.

However we do have at our disposal the true black-boxes for the MIP and EDA experiments. By design of the experiment, evaluating one set of hyperparameters on the MIP experiment takes 2 hours. Compared to that, the time to train a GP model or doing a forward pass in NAP at test time is negligible. On EDA, the black-box time depends on the circuit so we approximate an average running time of 1 minute per circuit on open-source circuits, but this can take several hours on industrial circuits.

Table 4 and 5 compare the average test time of one seed across all methods. In the first column, we can see that methods which have to fit a GP during the BO loop (FSBO, MetaBO and GP-EI) are considerably slowed down compared to methods like NAP that only do forward passes through their network. This is because fitting the GP surrogate at each BO step is time consuming, and increasingly so, as its dominant computational cost is cubic in the number of observed points. Note also that

FSBO not only fits a GP at each step but also fine tunes the MLP of its deep kernel, hence the extra time. The second column, with the black-box time taken into account, further underlines that even though NAP is faster at test time than e.g. FSBO or GP-EI, this time gain it is negligible compared to the black-box evaluations. The third column takes into account the pre-training time for methods that require it. Note that for different test functions within the same search space, we can reuse the same model for NAP, NP-EI, MetaBO and FSBO without having to redo the pre-training, so we divided the pre-training time by the number of seeds and test functions. Hence, it does not add much time to the total.

It should be underlined that this way of presenting BO results is less readable than presenting regret vs BO steps as the more seeds and test tasks we have, the more negligible the pre-training time becomes compared to the black-box evaluation time.

Table 4: Average test time of 1 seed on the MIP experiment.

Method	**without bbox**	**with bbox**	**with bbox & pretrain**
GP-EI	585sec	25d 0hr 9min 45sec	25d 0hr 9min 45sec
FSBO	330sec	25d 0hr 5min 30sec	25d 0hr 10min
MetaBO	30sec	25d 0hr 0min 30sec	25d 0hr 12min
NP-EI	2sec	25d 0hr 0min 2sec	25d 0hr 36min
NAP	3sec	25d 0hr 0min 3sec	25d 1hr

Table 5: Average test time of 1 seed on the EDA experiment.

Method	**without bbox**	**with bbox**	**with bbox & pretrain**
GP-EI	17sec	1hr 5min 17sec	1hr 5min 17sec
FSBO	516sec	1hr 13min 36sec	1hr 14min 6sec
MetaBO	35sec	1hr 5min 35sec	1hr 12min 5sec
NP-EI	8sec	1hr 5min 8sec	1hr 7min 34sec
NAP	9sec	1hr 5min 9sec	1hr 7min 42sec

D Discussions

We give a more detailed explanation of Table 1 below.

OptFormer OptFormer encodes a history as follows: a short meta-data sequence describing the variables taken as input and a sequence of trials (a trial corresponds to an $\langle \boldsymbol{x}, y \rangle$ pair). Each trial is composed of the values present in $\boldsymbol{x}$, the value of y and a separator to mark the end of a trial $(D+2)$. They need to use positional encoding to keep the sequence consistent (for instance the order of the dimensions is important to identify which dimension of x it is). Because of that, their architecture is not history-order invariant.

The query independence of their model is debatable. We can achieve it by splitting the queries across batches, but it is not doable in a single batch without substantial modifications of their code, their masks and their positional encoding. Note that splitting queries across batches results in a very slow inference. To evaluate OptFormer on HPO-B in a fair manner we had to do it, and it took us more than 2 weeks to obtain the results with 16 GPUs.

Transformer NP Nguyen and Grover [12] propose several transformer-based neural process architecture. Omitting the fact that they predict a Gaussian distribution over function values and not acquisitions, their TNP-D transformer architecture is the closest to ours. It does not rely on positional encoding, hence it is history-order invariant.

However, instead of summing the embeddings of $\boldsymbol{x}$ and y as we do, they concatenate them in a fixed representation, forcing them to set $y = 0$ in the queries. Hence, the only way to achieve query independence is to train on the same queries during testing and training. As we cannot assume we know what the queries will be during testing, Property 3.3 is not respected for **any** queries.

Prior Fitted Transformer PFN satisfies the two properties 3.2 and 3.3 since they do not rely on positional encoding and sum the embeddings of $\boldsymbol{x}$ and y.

The main differences with them are that our input is different and that we predict AF values with reinforcement learning.