Effects of Input Addition in Learning for Adaptive Games: Towards Learning with Structural Changes

Iago Bonnici, Abdelkader Gouaich, Fabien Michel

Effects of Input Addition in Learning for Adaptive Games: Towards Learning with Structural Changes

Iago Bonnici[1], Abdelkader Gouaïch[1], and Fabien Michel[1]

LIRMM, Université de Montpellier, CNRS, Montpellier, France
{iago.bonnici,gouaich,fmichel}@lirmm.fr
http://www.lirmm.fr/

Abstract. Adaptive Games (AG) involve a controller agent that continuously feeds from player actions and game state to tweak a set of game parameters in order to maintain or achieve an objective function such as the flow measure defined by Csíkszentmihályi. This can be considered a Reinforcement Learning (RL) situation, so that classical Machine Learning (ML) approaches can be used. On the other hand, many games naturally exhibit an incremental gameplay where new actions and elements are introduced or removed progressively to enhance player's learning curve or to introduce variety within the game. This makes the RL situation unusual because the controller agent input/output signature can change over the course of learning. In this paper, we get interested in this unusual "protean" learning situation (PL). In particular, we assess how the learner can rely on its past shapes and experience to keep improving among signature changes without needing to restart the learning from scratch on each change. We first develop a rigorous formalization of the PL problem. Then, we address the first elementary signature change: "input addition", with Recurrent Neural Networks (RNNs) in an idealized PL situation. As a first result, we find that it is possible to benefit from prior learning in RNNs even if the past controller agent signature has less inputs. The use of PL in AG thus remains encouraged. Investigating output addition, input/output removal and translating these results to generic PL will be part of future works.

Keywords: adaptive games · reinforcement learning · transfer learning · recurrent networks.

1 Introduction

Adaptive Games Video games are part of our digital culture and economy with many applications in ranging from entertainment to training and sport. A video game has a specific feature: It is a software that interacts with players to create a subjective experience that mixes rationality, emotion and aesthetics. The degree to which the player values this experience determines how much the video game is accepted and successfully used as an entertainment or training

means. The subjective nature of the player's experience reinforces the need for adaptation to better take into account individual characteristics during the play.

This defines an *adaptive game* (AG) as a game that exposes some parameters to be tweaked by an external controller in order to maintain some metrics – often related to player's experience – in an acceptable range of values. AG therefore requires a particular component, called a *control agent*, that continuously feeds from player actions and game state and tweaks a set of game parameters so that player's experience metrics remain optimal. This can be considered a Reinforcement Learning situation (RL) [1,2], so classical machine learning approaches (ML) can be used. With the development of neural networks, in particular Recurrent Neural Networks (RNNs) that are well suited for sequential processing [3,4], new opportunities are offered to develop a game controller using machine learning techniques.

The protean situation Games that aim to create elaborated and original experiences need to provide a rich set of actions and rules. Due to this intrinsic complexity, these games cannot be designed, neither exposed to players, as a monolithic block. In fact, an incremental approach is often required to gradually introduce the gameplay, follow player's learning curve and ease understanding of the game mechanics. This incremental approach is also followed by many training strategies where trainees are trained with only a subset of the whole system and additional concepts are introduced only when this subset is mastered. For instance, Chess can be introduced with sequential subgames involving only pawns and kings, then bishops and rooks, *etc.* Also, the richness of the set of actions and rules is sometimes explored back and forth, with the succession of various game levels where the player's abilities vary from one situation to the other (*e.g.* can fly *vs.* cannot, senses enemies *vs.* does not). In a sense, the game structure changes, leading to changes in the control agent's set of inputs, outputs and feedbacks. We refer to this set as the agent's "signature".

Suppose now that a control agent has been trained using ML techniques and tools, such as RNN, to optimize the player's metrics under the first game signature Δ_0. The question asked is: How can this agent, trained with Δ_0, be used later under other signatures like Δ_1, Δ_2, *etc.*? Is it more interesting to restart the training from scratch on each signature change or is it possible to benefit from previous experience even though the signature is not the same?

In this paper, we trial this question by exploring and evaluating a learning solution which capitalizes on past experiences, even when the signature partially changes. We use the adjective *protean* to characterize this variable nature of the control agent signature, and refer to the situation as a generalization of RL to Protean Learning (PL). After a succinct overview of related works (next section), we shall offer a formalization of PL (section 2.2), and a description of the experimental setting (section 2.3). Section 3 shall expose and discuss our first results before we finally conclude.

Related work Situations similar to PL are known in the domain of Transfer Learning (TL). In TL, the agent has already learned tasks called "source" tasks, and the challenge is to benefit from this previous "knowledge" while tackling a new "target" task. In other words, the TL agent is expected to generalize not only *within* tasks, but also *across* tasks [5,6]. This domain is transversal to ML as it applies both to Supervised Learning (SL) and RL. TL may be invoked in various situations: **(1)** The source training process has been successful but costly: one wishes to benefit from transfer to tackle a new target task more efficiently [7,5]. **(2)** The target task is challenging: one wishes to split it up into several easier source tasks, expecting that the overall TL process will be more efficient than direct tackling of the target [5,8,9]. **(3)** Several tasks must be learned at once: one wishes that TL occurs from the ones to the others, and speeds up the parallel process [10]. **(4)** It is known that the task at hand will undergo future changes: one wishes to design an agent able to adapt these changes and benefit from transfer from one task to the next. AG controllers are in the latter situation. PL is an instance of this situation.

PL is also related to Concept Drift (CD), a situation where the environment is assumed to undergo changes while the agent keeps learning [11,12]. It is also related to Continual Learning (CL) or "lifelong learning", an AI design where the agent keeps learning although its environment is changing and it regularly faces new challenges [13,14]. To our knowledge, even though the environment function is expected to change in TL, CD, CL, the signature of the agent is often assumed to be fixed. Here, we focus on the various changes in signature that may occur during the learning process (*e.g.* new input, change in output domain, input loss), and how transfer can be achieved in each case with the classical RNNs tools.

2 Material and Methods

2.1 Approach

Studying PL in AGs must first be conducted with small AGs that optimize basic player's metrics. For instance, the metric optimized may be Csíkszentmihályi's flow, a psychological state universally perceived as a positive experience [15,16]. The video game flOw already does this adaptation [17,18], and would easily be adapted so that its signature changes from one level to the other. This will serve as an experimental setup in subsequent works.

To reach this objective, we first need to formalize the system that we call PL. This is done in in section 2.2. Before testing PL in AGs, the basic relevance of PL has to be asserted. In section 2.3, we conduct an experiment involving RNNs in a idealized PL situation to assess the first operation among 6 elementary signature changes: input addition, input removal, output or feedback addition or removal. This experiment verifies that adding an input to the agent during the learning process does not dramatically alter learning, and that a protean learner performs better in this situation than a learner resuming from scratch

on a signature change. Therefore, it encourages subsequent works towards the PL AG objective.

2.2 Formalization of PL

In this section, we sum up a formalization of PL situation, based on detailed report [19]. It is offered as a generalization of RL to non-fixed signature learning situations.

A RL agent is continuously fed with inputs, so its inputs can be seen as *signals*: values that change over time. It also feeds from feedback signals, and produces output signals. Therefore RL can be viewed as a case of *signal processing*, where the learner continuously transforms the inputs signals into output signals, with the objective that feedback signals are kept high. On the other hand, the environment continuously transforms the output signals into input signals and feedbacks, by strict application of the universe rules.

The agent *signature* is the number and the type of signals it produces and feeds from. In other words, it is the collection of domains the various signals take their value from. The idea with this formalization is to make the signature a signal itself. As such, it may also change in time, which makes the agent *protean* and extends RL to PL.

We formalize signals with plain functions of continuous time that we call *flows*: $h : \mathbb{R}^+ \to D$. They take their value in arbitrary domains D. Flows can be discretized in time with arbitrary precision $\epsilon \in \mathbb{R}^{+*}$ by sequences ${}^{\epsilon}h : \mathbb{N} \to D$ such that:

$$\forall t \in \mathbb{N}, \quad {}^{\epsilon}h(t) = h(t\epsilon) \tag{1}$$

Flows transform into each other. Viewed another way, a flow can be determined by another flow. We call a *determination function* f a function able to determine an outgoing flow k from an incoming flow h no matter the precision ϵ considered:

$$\forall \epsilon \in \mathbb{R}^{+*},\ \forall t \in \mathbb{N}^*, \quad {}^{\epsilon}k(t) = f_\epsilon({}^{\epsilon}h(t),\ {}^{\epsilon}h(t-1),\ \ldots,\ {}^{\epsilon}h(0)) \tag{2}$$

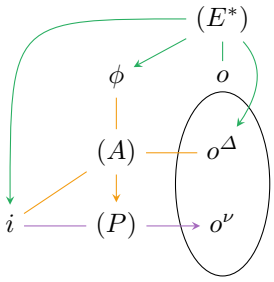
Flow determination has a *memory* in that current value of k may depend on past values of h, so it is not Markovian in general. However, it is always *causal* in that future values of k cannot be determined given only the current and past values of h. We shall use the following graphical alias for the determination relation (2):

$$h \text{ — } (f) \rightarrow k \tag{3}$$

The symbol in parentheses represents the determination function, the symbol pointed by the arrow head is the consequence flow, and the symbol pointed by the line with no head is the cause flow. For instance, i — (P) → o means that the inner agent process P feeds from input signal i to produce the output signal o in a causal, yet potentially non-Markovian way. And o — (E) → i means that the environment E works the other way round.

Multiple flows h carry both a flow of domains h^Δ (or "signature") and a flow of actual values h^ν. *Domains* are a tuple of sets $(S_1,\ S_2,\ \dots)$ and *values* are a tuple of elements of these sets $(v_1 \in S_1,\ v_2 \in S_2,\ \dots)$. For instance, at $t = 0.9$, an agent that is sensitive to both "wind direction and player speed" in the game simulation may receive $h^\Delta(0.9) = \big([0,\ 2\pi[,\ \mathbb{R}^+\big)$ and $h^\nu(0.9) = (0.2,\ 50)$. The particularity of PL is that the flow of domains is not constant. We call *transformation* of the agent any variation of h^Δ resulting in that the agent may later receive values with different domain signatures, *e.g.* "player speed and rain intensity" $h^\Delta(1.1) = \big(\mathbb{R}^+,\ [\![1,\ 5]\!]\big)$ and $h^\nu(1.1) = (31,\ 4)$.

At the highest level, a PL learner agent can be represented by 3 multiple flows $(i,\ o,\ \phi)$ and 1 flow of determining functions P with the following determination scheme:



(4)

The latter scheme can be considered a set of formal equations according to (3) and (2). Each color corresponds to one determination triplet, where:

- E represents the environment in which the agent is immersed. The * denotes that initial values for $i,\ \phi,\ o^\Delta$ are determined by E. For a control agent in AG, E represents both the player and the game engine.
- i represents the agent's *inputs* or *sensors*. In AG, i informs the agent of player's actions and game state. Their nature may change in time as the signature i^Δ evolves (*e.g.* among game levels).
- o represents the agent's *outputs* or *actuators*. In AG, o controls the game parameters that need to be adjusted to maintain good player's metrics. Their nature may also change in time as o^Δ evolves. Note that output values o^ν are determined by the agent, but the output signature o^Δ is determined by the environment.
- ϕ represents the agent's *feedback*, *rewards* or *objectives*, a continuously fed evaluation of the actions it undertakes. In AG, ϕ may represent the player's experience metrics like Csíkszentmihályi's flow. They also may change in nature as ϕ^Δ evolves.

The environment determines i, ϕ and o^Δ, so the agent cannot directly decide its input or feedback data, nor its output signature.

- P represents the agent current behavior. It is an inner computational procedure that determines current output values based on current input/feedback values and all their history.

– A is the abstract strategy of the agent, which is continuously adapting its behavior P based on environmental information.

Only two objects are not depending on time here: the environment E and the inner agent strategy A.

The classical RL problem of "maximizing reward" [1] can be reformulated in PL as: Given environment E and a corresponding flow of rewards ϕ with all domains being numerical, find an agent procedure A such that all values taken by signal ϕ^ν are maximized. A is considered an interesting learner if it can maximize the rewards for a whole family of environments, and no matter the changes in signatures i^Δ, o^Δ or ϕ^Δ.

In other words, an AG control agent is interesting if it can maintain good experience metrics no matter the player skills, or the game evolution.

2.3 Experiments

As a preliminary to the construction of generic PL agents, we design an experiment to assess viability of the first type of signature change during learning: "input addition". To this end, we restrict ourselves to an ideal situation where we, as experimenters, know the optimal behavior $\hat{P}$. The agent A will be able to access a set of example optimal realizations $T = \{(i_k,\ \hat{o}_k)\}_{k \in [\![1,\ 1000]\!]}$ with every $i_k - (\hat{P}) \twoheadrightarrow \hat{o}_k$ (see section 2.2). This will constitute a *training set* so the agent search for an approximation of $\hat{P}$ can be solved by a supervised learning approach (SL). To simulate the signature change, we stop the SL procedure, change the dimension of each i_k along with the signature of $\hat{P}$, and resume SL. Comparison is made with an agent that directly learns with the second signature. The less efficient the second agent compared to the first one, the more encouraging it is in favor the PL approach.

Relative efficiency of these agents is measured in various experimental settings to address results robustness. In the next section, we shall describe protocol for the generation of T, the SL approximation, the signature change simulation and the comparison of the agents, along with the variable experimental settings.

Inputs Generation Idealized input i_k, are 2-dimensional signals generated in two different ways, depending on the first experimental parameter:

– In the noisy setting, each value is drawn from a uniform distribution so that inputs have no structure: $i_k(t) \hookrightarrow \mathcal{U}\big([-1,\ 1]^2\big)$.
– In the smooth setting, each dimension of the signal is generated independently as a combination of sine waves, so that inputs are autocorrelated. The generation procedure is described as follows (example Fig. 1 top panel):

  for each dimension:
    Draw the number of sine waves from a Poisson distribution $\mathcal{P}(500)$

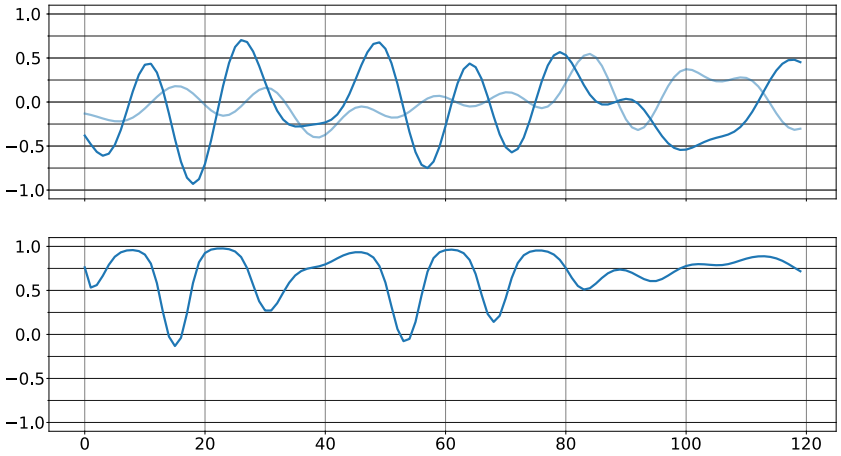


Fig. 1. Example of smooth processed signals. Top: 2D input signal i_k. Bottom: 1D output signal $\hat{o}_k$. In this mem example, $p = 0.8$ (high influence of first dimension of the signal (solid one)) and $c = 1$ (highly skewed output result).

for each sine wave:
draw the pulsation ω from uniform $\mathcal{U}([0,\ 30])$
draw the phase ϕ from uniform $\mathcal{U}([0,\ 2\,\pi])$
evaluate $\sin(\omega\, t + \phi)$ for 120 values of t evenly spaced between 0 and 2
end for
sum all sine waves into one combined signal sequence
draw the final amplitude a from uniform $\mathcal{U}([0,\ 1])$
rescale the sequence so it has desired amplitude: sequence $\leftarrow a \times \frac{\text{sequence}}{\max(|\text{sequence}|)}$
end for
join both sequences into one input signal $i_k : [\![1,\ 120]\!] \to [-1,\ 1]^2$

Optimal Outputs Generation Training outputs examples $\hat{o}_k$ are generated using idealized behavior $\hat{P}$. Depending on the experimental setting, two different formulae are used for $\hat{P}$ to address the effect of task complexity: one is instantaneous while the other exhibits a temporal memory.

– In the nomem setting, $\hat{P}$ has no memory and performs a plain combination of the 2 input signals dimensions i_k^1 and i_k^2 that is affine in the atanh transformed space:

$$\hat{o}_k(t) = \tanh\big(p\,\text{atanh}\big(i_k^1(t)\big) + (1-p)\text{atanh}\big(i_k^2(t)\big) + c\big) \tag{5}$$

With $p \in [0,\ 1]$. The higher p, the more information contained in the first input signal dimension i_k^1, and not in the second i_k^2. The further c is from zero, the more skewed is the resulting output signal $\hat{o}_k$ (see Fig. 1 bottom panel). The settings $c = 0$ and $c = 1$ are tested, in interaction with p being given the values 0, 0.2, 0.5 and 0.8.

– In the mem setting, the processor is more complex because it exhibits a 1-step memory, so it is non-Markovian. It performs the same combination with the exception that $5 \times (i_k^d(t) - i_k^d(t-1))$ is used instead of $\operatorname{atanh}\big(i_k^d(t)\big)$ in (5). In other words, the derivative of the input signals are used instead of their actual values.

Agent Structure The agent approximates $\hat{P}$ with its produced actual P. P takes the form of a Recurrent Neural Network (RNN) [3,4]. We use a standard Gated Recurrent Unit (GRU) [20]. P produces the agent actual outputs according to $i_k - (P) \rightarrow o_k$. Two different structures are used for P to address their effect on PL.

– In the flat setting, 1 GRU cell is used with 1 internal state, used as network output. So P is Markovian.
– In the deep setting, 2 GRU cells are used as different network layers with 6 internal states, the last one being used as the network output. So P may be non-Markovian.

Loss function The loss function to optimize is a classical Mean-Squared-Error (MSE) between the agent predictions o_k and the expected results $\hat{o}_k$.

The learning procedure A processes training examples by batches of 10, and updates the weights parameters of P with a stochastic gradient descent respecting Adam update rule [21] (learning rate = 0.01).

Convergence is achieved using pytorch [22] for 1000 iterations.

Realization of Signature Change Three convergences are achieved on each run (see Fig. 2), according to the following protocol:

– first, one protean "first-form" agent A_{f_1} is constructed with a 1D-1D signature. Its parameters are randomly initialized, then it is trained against the training set T but it only feeds from the first dimension i_k^1 of input signals, being blind to i_k^2.
– second, one protean "second-form" agent A_{f_2} is constructed with a 2D-1D signature. Its parameters are initialized by copying the final value of all homologous parameters from A_{f_1}, and setting additional parameters to zero. Then, it is trained against the whole training set T, not ignoring the second dimension i_k^2 anymore.
– third, one classical "direct" agent A_d is constructed with a 2D-1D signature. Its parameters are randomly initialized, then it is directly trained against the whole training set T.

The $(A_{f_1},\ A_{f_2})$ agent is the experimental model of a PL agent in the idealized learning situation. It experiences the signature change "input addition".

1000 such replicates are run for each combination of experimental settings.

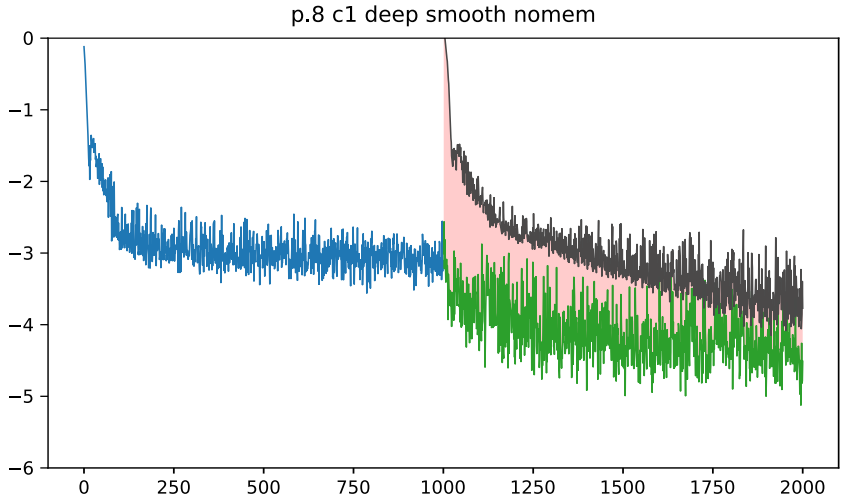


Fig. 2. example of experimental learning curves l: evolution of the $\log_{10}$(MSE): the "first form" learner A_{f_1} ($l_{A_{f_1}}$ in blue) is trained to predict a combination of a 2D signal, but it can only access the first dimension as an input so it has incomplete information. The "direct" learner A_d (l_{A_d} in black) is trained on the same task, but it can access the whole information so it performs better. The "second-form" learner A_{f_2} ($l_{A_{f_2}}$ in green) can also access the whole information so it has the same signature as A_d. But A_{f_2} parameters are grown from previous A_{f_1} instead of being randomly initialized. The gain score (in red ≈ 5.242) measures the benefit of PL (based on previous experience) compared to starting the learning from scratch on a signature change. The signature change "input addition" occurs at $t = 1000$.

Measure of the Advantage The advantage of PL compared to direct learning is estimated on learning curves l by a score (see Fig. 2). Score is calculated on each run as a *gain* according to:

$$\text{score} = \text{mean}\left(\log_2\left(\frac{l_{A_d}}{l_{A_{f_2}}}\right)\right) \tag{6}$$

A score = 1 means that the second-form agent error is twice as low as the direct agent on average. A score = 0 means that the second-form agent and the direct agent perform similarly.

3 Results

The scores obtained on each run are summarized and illustrated in Fig. 3.

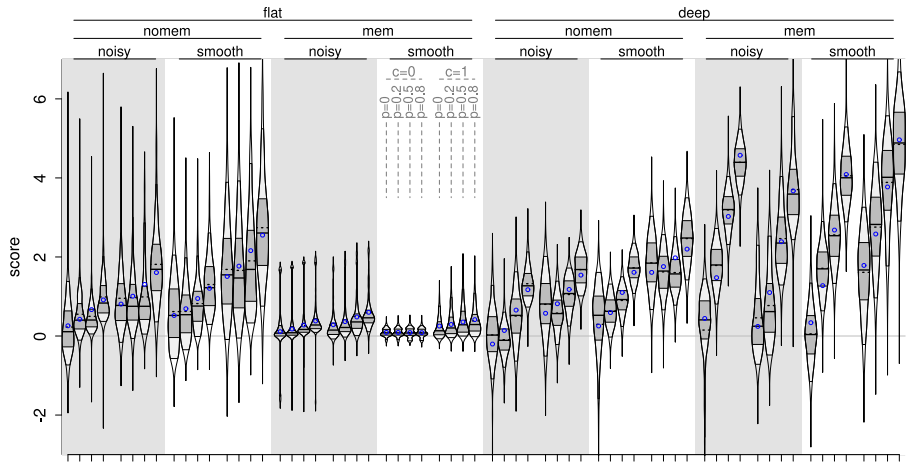


Fig. 3. Violin plot: Comparison of scores in various PL situations differing by the RNN structure of P (flat or deep), the task complexity $\hat{P}$ (nomem or mem), the input data i_k properties (smooth or noisy) and the relative informativeness of partial information (c, p). The benefit of PL differs depending on the situation, but is overall positive. Within each violin, the dashed line represents mean value for the group, the solid line represents median value, and the two gray areas represent 50% and 90% percentiles. Blue circles represent predictions of the linear model fitted on the data.

To address relevance of the observed variations, we fitted a linear model on the data, testing all interactions between experimental settings considered as factors and one nested slope for parameter p considered numerical (degrees of freedom: 63698, residual stde: 0.7689). Convergence and analysis of the model have been achieved with R-Cran software [23]. All effects were different from zero with high significance $p\text{-}value \leq .001$ with the only exception of the nested slope

flat:mem:smooth:c=0:p ($p\text{-}value = .8118$). We therefore consider that all trends among groups observable on Fig. 3 are relevant to discuss, except the latter slope that must be considered null.

However, convergence of the network and good approximation of P within the setting flat:mem is impossible because P (non-Markovian then) uses a memory information that neither A_{f_1}, A_{f_2} nor A_d (Markovian then) can access. As a consequence, they all poorly approximate P so these score values must be considered carefully. Correctly interpreting these values requires further investigations under conditions where networks fail to converge, which is out of the scope of this paper.

As expected, no matter the experimental setting, the measured score is mostly positive on average. This reflects the advantage of the second-form agent A_{f_2} compared to the direct agent A_d after the signature change. When the event "add input" occurs on 1D-1D A_{f_1}, it is better to transform it into a 2D-1D adapted A_{f_2} — and keep its parameters that have converged so far, than to replace it with a new, naive 2D-1D A_d — and forget everything that has been learned so far. This advantage needs to be qualified depending on the learning situation:

- *the lower p, the lower the advantage*: The lower p, the less informative the first dimension i^1_k that A_{f_1} is sensitive to, because it has low impact on the desired output. As a consequence, there is less learning gain to harvest for the protean agent during the first phase of PL. This confirms a naive intuition that PL is only interesting if past experience of the learner is somehow relevant.
- *the higher c, the higher the advantage*: The higher c, the more skewed the target output $\hat{o}_k$. Regardless of the informativeness of i^1_k, A_{f_1} can always benefit from the first PL phase to learn this skew. With high c, and during the second phase, A_d has to learn both the skew and the formula $\hat{P}$, so it is disadvantaged against A_{f_2} that already approximates the skew correctly. Interestingly, this advantage remains *even if p=0*. We must therefore consider that the "relevance of past learner experience" does not only lie in the informativeness of the inputs it was sensitive to, but also in the skews and patterns of the objective behavior $\hat{P}$ itself.
- *PL advantage is higher with smooth input signals*: Smooth signals are more structured than the noisy ones. Regardless of the informativeness of i^1_k, A_{f_1} can always benefit from the first PL phase to learn this structure. During the second smooth phase, A_d will have to learn both the structure and the formula $\hat{P}$, so it will be disadvantaged against A_{f_2} that already relies on the structure correctly. Interestingly, this advantage remains *even if p=0 and c=0*. Therefore, the "relevance of past learner experience" also lies in the very structure of past processed data.
- *PL advantage is lower with deep RNNs* (at least in nomem setting for we do not discuss flat:mem): Deeper RNNs are more flexible in the sense that there exists more combinations of their parameters that approximate the

objective function $\hat{P}$ correctly. As a consequence, it is easier for A_d to find a path down the loss function and converge during the second PL phase, which makes A_{f_2} advantage less decisive at this stage.

As a summary, we observe that PL seems mostly beneficial after an "input addition", not only if past inputs were relevant ($p > 0$), but also if the learning data is somehow structured (c=1, smooth). Our interpretation is that prior A_{f_1} learner can learn and transmit partial information about this structure to current A_{f_2} agent *via* RNN parameters. One possibility could be that A_{f_1} sometimes converges towards a "dead-end" direction during the first phase of the experiment, so that it has to "unlearn" during the second phase and be disadvantaged against A_d — a phenomenon known as *negative transfer* [5] — but this is not the average behavior we observe here.

There exists several potential advantageous effects of TL for the second-form agent A_{f_2} that benefits from transfer [5] (see Fig. 2). First, A_{f_2} initial loss may be lower than A_d, because A_{f_1} has already started convergence; this is known as *jumpstart* benefit. Second, A_{f_2} loss may decrease faster than A_d, so A_{f_2} is said to learn *faster*. Lastly, A_{f_2} final loss may be lower than A_d, so A_{f_2} is said to learn *better*. In our experiment, the *gain* measured according to (6) is an aggregated estimation of these 3 potential advantages. Therefore, we cannot distinguish them from each other. However, considering that the only difference between A_d and A_{f_2} is their initial RNN parameter values, we strongly conjecture that the jumpstart effect is the major benefit of PL in this experiment.

This experiment is idealized because the agent can access a whole training set of optimal outputs $\hat{o}_k$, and it is directly guided by the loss gradient relative to its inner P RNN parameters. However, it is an instance of the generic PL procedure formalized in section 2.2. Benefiting from these preliminary results in AG is therefore a matter of relaxing the ideal hypotheses and rely more on RL than SL approaches. In addition, PL can be used in other game-related problems like generation of believable behaviors [24,25,26].

In subsequent works, consistently with our general approach (section 2.1), PL will be used again to address other types of signature changes like input removal, and output/feedback addition/removal. Then, actual PL agents will be constructed and trained against an abstract PL benchmarking task. When stabilized, they will be tested as control agents in simple adaptive games like experimental extensions of flOw.

4 Conclusion

AG demand that a controller agent continuously adapts to the player. This is a case of RL. However, complex games make this learning special because the agent has to face changes in its signature. We generalize the idea of RL to a broader PL theoretical situation explicitly taking these changes into account. This enables a rigorous assessment of how this kind of learners could be developed. With a controlled idealized PL experiment, we have shown that considering input addition

for protean learners can be addressed efficiently at least with RNNs. Moreover, the benefits of PL in this situation are both easy to use (simply pad RNN with zeroes) and robust to changes in the learning context (informativeness of partial input, task complexity, data structure). These first results are encouraging, suggesting that the protean approach can be both simple and robust, and that controlling complex, changing AGs with PL is promising. The assessment of PL is still incomplete, since other basic operations must be tested: input removal and output/feedback addition/removal. Moreover, exporting these results from idealized PL to generic PL tasks still needs to be done.

Like any ML approach, PL is generic and can be applied in any other domain where a signature change in RL is identified. For instance, modular robotics could benefit from PL controllers. Long-term learners that cannot discard their past experience on a signature change also need to capitalize on it. And more generally, any bio-inspired agent that need to transform, split or merge, while keeping on learning, can benefit from a PL approach.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. Adaptive computation and machine learning series, MIT Press, Cambridge, Mass, 2nd edn. (2018)
2. Hanna, C.J., Hickey, R.J., Charles, D.K., Black, M.M.: Modular Reinforcement Learning architectures for artificially intelligent agents in complex game environments. In: Computational Intelligence and Games. pp. 380–387. IEEE, Copenhagen, Denmark (Aug 2010)
3. Elman, J.: Finding structure in time. Cognitive Science 14(2), 179–211 (Jun 1990)
4. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks 61, 85–117 (Jan 2015)
5. Taylor, M.E., Stone, P.: Transfer Learning for Reinforcement Learning Domains: A Survey. Journal of Machine Learning Research 10(7), 1633–1685 (2009)
6. Lazaric, A.: Transfer in Reinforcement Learning: A Framework and a Survey. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, vol. 12, pp. 143–173. Springer (2012)
7. Tanaka, F., Yamamura, M.: Multitask reinforcement learning on the distribution of MDPs. In: International Symposium on Computational Intelligence in Robotics and Automation. Computational Intelligence in Robotics and Automation for the New Millennium. vol. 3, pp. 1108–1113. IEEE, Kobe, Japan (2003)
8. Devin, C., Gupta, A., Darrell, T., Abbeel, P., Levine, S.: Learning Modular Neural Network Policies for Multi-Task and Multi-Robot Transfer. CoRR abs/1609.07088 (2016)
9. Frans, K., Ho, J., Chen, X., Abbeel, P., Schulman, J.: Meta Learning Shared Hierarchies. CoRR abs/1710.09767 (2017)
10. Teh, Y.W., Bapst, V., Czarnecki, W.M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., Pascanu, R.: Distral: Robust Multitask Reinforcement Learning. CoRR abs/1707.04175 (2017)
11. Tsymbal, A.: The Problem of Concept Drift: Definitions and Related Work (2004)
12. Heng Wang, Abraham, Z.: Concept drift detection for streaming data. In: International Joint Conference on Neural Networks. pp. 1–9. IEEE, Killarney, Ireland (Jul 2015)

13. Ring, M.B.: Continual Learning in Reinforcement Environments. Ph.D. thesis, University of Texas at Austin, Austin, TX, USA (1994)
14. Xu, J., Zhu, Z.: Reinforced Continual Learning. CoRR abs/1805.12369 (2018)
15. Sweetser, P., Wyeth, P.: GameFlow: A model for evaluating player enjoyment in games. Computers in Entertainment 3(3), 3 (Jul 2005)
16. Holt, R., Mitterer, J.: Examining video game immersion as a flow state. 108th Annual Psychological Association, Washington, DC (2000)
17. Chen, J.: flOw. http://jenovachen.info/flow/ (Jan 2019)
18. Chen, J.: Flow in games (and everything else). ACM Communications 50(4), 31 (Apr 2007)
19. Bonnici, I., Gouaïch, A.: Formalisation of metamorph Reinforcement Learning. Tech. rep., LIRMM, Montpellier, France (Nov 2018)
20. Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., Bengio, Y., Bahdanau, D.: Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. CoRR abs/1406.1078 (2014)
21. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. CoRR abs/1412.6980 (2014)
22. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch (2017)
23. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2018)
24. Le Hy, R., Arrigoni, A., Bessiere, P., Lebeltel, O.: Teaching Bayesian Behaviours to Video Game Characters. Robotics and Autonomous Systems 47, 177–185 (2004)
25. Tencé, F., Buche, C.: Automatable Evaluation Method Oriented toward Behaviour Believability for Video Games. CoRR abs/1009.0501 (2010)
26. Polceanu, M., Mora, A., Jimenez, J., Buche, C., Fernandez-Leiva, A.: The Believability Gene in Virtual Bots. In: 29th International Flairs. p. 4. AAAI Press, Key Largo, Florida (2016)