Interpretable Utility-based Models Applied to the FightingICE Platform

Tianyu Chen
Tokyo Institute of Technology
Tokyo, Japan
scdev.tianyu.chen@gmail.com

Florian Richoux
AIST and JFLI, CNRS
Tokyo, Japan
florian.richoux@aist.go.jp

Javier M. Torres
Brainific SL
Madrid, Spain
javier.m.torres@brainific.com

Katsumi Inoue
NII
Tokyo, Japan
inoue@nii.ac.jp

***Abstract*—One task of game designers is to give NPCs fun behaviors, where "fun" can have many different manifestations. Several classic methods exist in Game AI to model NPCs' behaviors; one of them is utility-based AI. Utility functions constitute a powerful tool to define behaviors but can be tedious and time-consuming to make and tune correctly until the desired behavior is achieved. Here, we propose a method to learn utility functions from data collected after some human-played games, to recreate a target behavior. Utility functions are modeled using Interpretable Compositional Networks, allowing us to get interpretable results, unlike regular neural networks. We show our method can handle noisy data and learn utility functions able to credibly reproduce different target behaviors, with a median accuracy from 64.5% to 83.7%, using the FightingICE platform, an environment for AI agent competitions. We believe our method can be useful to game designers to quickly prototype NPCs' behaviors, and even to define their final utility functions.**

***Index Terms*—Utility-based AI, Machine Learning, Interpretable Results, Fighting Games.**

I. Introduction

Rushing towards the player and raining blows. Dodging attacks and waiting for the right moment. Or, showing resistance and letting an opening on purpose so the player can hit and enjoy the thrill of the fight. Part of the job of game designers is to attribute adequate, challenging, yet fun behaviors to Non-Player Characters (NPCs). There are several ways to define a behavior logic. Among the most classical methods used in the game industry, one can find finite state machines, behavior trees, and utility-based AI [23]. This paper focuses on the latter.

The idea behind utility-based AI controlling an NPC is to have one function for each possible NPC action. Given the current game state (players and NPCs hit points, energies, positions, distances, etc), these functions output a score indicating how appropriate each action is, regarding the current situation. Such functions are named **utility functions**, and they must be carefully designed to get the desired NPC behavior.

One advantage of using utility-based AI is that utility functions can be reused among different NPCs if they share some facets of their behaviors. However, designing and tuning utility functions can be quickly a complex and tedious task [9], in particular if many actions are available or if game designers are targeting a very specific behavior. Indeed, an NPC behavior cannot most of the time be summed up to one or a couple of independent actions, but is expressed by complex action interlocks and relations. Moreover, utility-based AI can imply some loss of control for the game designer [9]. This paper proposes a method to lessen this first issue, but also the second one to a certain extent.

Rather than letting game designers and game developers craft utility functions by hand, this study aims at helping them by proposing a method to learn interpretable utility functions automatically from game data. This paper constitutes a first step towards this goal. It does not claim to provide a tool that can be deployed at the moment for commercial games; but this is an objective for future work.

Let's consider the following use-case: Alice is a game designer and wants to design the behavior of the final boss of the game. She grabs a game controller and plays a couple of games, but rather than controlling the player character, Alice controls the boss while Bob, a game developer, controls the regular player. During the runs, Alice makes the boss act with a target behavior, e.g. very aggressively. Game states as well as actions performed are recorded at regular intervals, for example at each frame. These recorded data will constitute our training data. Now, Alice gives the training data to a system, based on our method, that learns and produces utility functions for each available boss action, such that the combination of these utility functions satisfyingly reproduce the target behavior, and such that these utility functions are easily interpretable. The latter is important so that Alice can modify and tweak them at will according to her needs, and Bob can implement them within the game engine.

In this paper, we use a variant of artificial neural networks named **Interpretable Compositional Networks** (ICN) to represent utility functions. ICNs were originally proposed to model and learn error function in Constraint Programming [13]. However, one can use an ICN to learn any kind of functions decomposable into elementary operations. In [13], they only need to learn one ICN at a time, but learning utility functions require learning several functions with the same inputs. Indeed, unlike [13], we are not aiming to learn one function returning a target output, but a set of functions where only the ranking of their output is considered. This raises a scientific challenge: How to efficiently learn several utility functions simultaneously with ICNs, as well as the relation among their outputs?

This paper contributes to answering those questions and proposes a method to learn interpretable utility functions from game data. It is organized as follows: Section II introduces Interpretable Compositional Networks, a variant of artificial neural networks we use to model utility functions, and presents the FightingICE platform we use as a test-bed for our method. Section III suggests some related work on player modeling, and Section IV contains the main contribution of this paper, presenting our method. Experimental results are given and analysed in Section V, and the paper concludes with some discussions and perspectives.

II. Preliminaries

In this section, we introduce Interpretable Compositional Networks. We show why ICNs are convenient models for our goal and what scientific challenges about them required to be unlocked for this study. We also introduce the FightingICE platform and explain why we chose it to test our utility function learning method.

A. Interpretable Compositional Networks

The first scientific challenge of this study was to find how to express utility functions in such a way we can learn them from data and easily understand them. Indeed, learning utility functions through a regular artificial neural network would have been certainly possible, but we would then get functions that game designers and developers would not be able to extract from the neural network, forcing them to run the learned neural network in a feed-forward manner to compute the utility function. This may not be desirable or suitable, both from a game design and a technical point of view, because computing a function through a neural network can induce significant overheads, and because each slight modifications of the game mechanics or game properties would imply to re-train these neural networks.

In this study, we break down every utility function into small computable pieces called elementary operations. A utility function is thus a (non-linear) combination of elementary operations. Learning a utility function boils down to learning the right combination of elementary operations.

To fit our needs, Interpretable Compositional Network is the right model. ICNs are inspired from Compositional Pattern-Producing Networks (CPPN) [17] often used to generate 2D or 3D images. It takes these two principles from CPPNs: 1. each neuron contains one activation function picked among a large, diverse set of activation functions, and 2. the network can deal with inputs of any size. What distinguishes ICN from CPPNs is that their architecture is fixed, with a specific purpose for each layer, and the weights of the network are binary, taking either 0 or 1 as value. This last characteristic is actually critical to get interpretable functions. Therefore, the main idea behind ICN is the following: it takes as input the same input as the function it aims to learn. Information within the input is extracted, broken down and recomposed through many different elementary operations (the activation functions of neurons composing the ICN). The outputed value entirely depends on the input and the combination of elementary operations.

Thus, some new questions are raised:

- What elementary operations to choose to describe generic utility functions?
- What kind of elementary operations combinations do we need, *i.e.*, what ICN architecture to choose?

These questions are answered in Section IV describing our method.

B. FightingICE

FightingICE [7] is a fighting game platform developed for research purposes at Ritsumeikan University. Figure 1 depicts a typical match of FightingICE. It provides interfaces in JAVA and Python to get (almost) all pieces of game information needed to allow artificial agents playing the game.

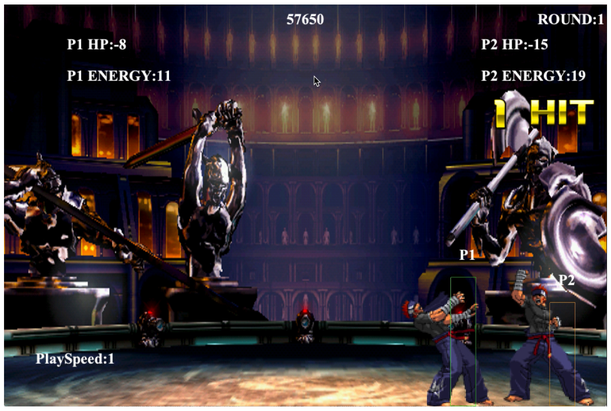

Figure 1: A screenshot of the game in FightingICE.

This platform has been designed in such a way that there is always a delay between actions performed within the game logic and the game state received by users through its programming interfaces. With the current version of FightingICE, this delay is about 250ms (or 15 frames), corresponding to the mean human reaction time to a visual stimulus. This design choice is motivated by the will of its authors to propose a fighting game platform where AIs play in similar conditions to human beings, but also to balance AI games by penalizing strategies that are overly defensive-oriented.

We choose this platform to be the test-bed of our method because it offers several advantages:

- It is simple to use.
- It is easy to control a character and record game states and actions.
- There are not too many possible actions (about 10, see below), which is perfect for a first approach.

We can consider that a FightingICE agent is able to do about 10 actions: stay idle, move forward, move backward, dash, crouch, jump, block, punch, kick and if the character has enough energy, throw a fireball. More actions are available in the FightingICE API, but many of them can be grouped. For instance, one can consider that the "air kick" action is doing a kick action while being in the air.

Note that our method can work for any kind of games with NPCs, not only fighting games. Of course, this implies choosing a correct architecture of ICNs and elementary operations to deal with different game state inputs than for FightingICE. Fighting games usually do not use utility-based AI; we simply consider FightingICE to be a convenient test-bed for this study.

III. Related Work

Player modelling, described in depth by Yannakakis et al. [22], obtains computational models of players in games. Some of these can be more detailed than others, and they may have different scopes across video game development and execution. They can be built following a top-down approach using an existing theoretical description, or a bottom up approach using data to tune a generic structure.

We focus on models that act as agents within the game itself, rather than models used on support functions like churn or matchmaking, or models of the player used to predict their actions [5], or their interests [15]. These "models-as-agents" combine the game input and some internal state to decide which action to take. In the terms described by Smith et al. [16], they are *induced from human reactions*, have a *generative purpose*, and are *restricted to an individual*.

The first game to create and use such models in a commercial setting was Forza Motorsport in 2005, to the extent of our knowledge [19]. The Drivatar concept involved training Bayesian Neural Networks locally in the user's system, and later on using a cloud to obtain these models with more computing power. Note that the use of trainable agents using neural networks was already present in games at least as back as 1997 in the game Creatures [4]. Other techniques to synthesize driving controllers with specific styles include fuzzy logic: Wang and Liaw [21] aim at imitating a player's style, although they fine-tune the existing controller described by Onieva et al. [11] (already based on fuzzy logic) instead of building a controller from scratch. Traditional Q-Learning was used by Trusler and Child [20], while Bayesian methods have been implemented by several authors (see [2], [18], [20]).

Other types of games have also been addressed in other work. Agent controllers that mimic the style of a specific player have been described for the Super Mario Bros platform game [12]. In the case of our specific genre of fighting games, Saini's Ph.D. Thesis [14] explores mimicking human player strategies with k-nearest neighbor classification, data-driven finite state machines and hierarchical clustering to identify patterns and strategies upon data collected from games where the two players are human. Lueangrueangroj and Kotrajaras [8] propose real-time imitating learning via dynamic scripting to mimic play style of human players upon the commercial game Street Fighter Zero 3 Upper. Konečný's Master thesis [6] introduces a 7-variable model to characterise both Fighting game human and artificial players. This model would have been convenient for our research but it was not trivial to get most of the 7 metrics composing the model from the FightingICE platform. This would certainly need a lot of engineering, requiring us to modify the platform. Finally, we can cite the work of Martínez-Arellano et al. [10], using genetic algorithm to generate enjoyable strategies to play against in the MUGEN engine. Like this work, we also use a genetic algorithm to learn models of our ICNs (although this is not the main point of our paper). One interesting aspect of Martínez-Arellano et al.'s result is that they do not need any prior knowledge of how to code strategies of their agents, and they do not require to directly interact with the code of the game, meaning their method could a priori work as-is on other fighting games, with minor engineering.

For a utility-based AI learning, we can cite the work of Bradley and Hayes [1], focusing on group utility functions (*i.e.*, utility functions describing the behavior of several agents, not only for one agent), to learn cooperative NPC behaviors via reinforcement learning. The main drawback with this approach is that it is a black-box system where utility functions are masked or unclear for human beings, and need to be retrained entirely if the game has been modified, which is an undesirable property during game development.

IV. Main contribution

We start to answer questions listed in the introduction and in Section II in this section. Since everything in our method depends on the way we model utility functions, we start showing how we describe game states and decompose utility functions.

A. Game states

In this study, a game state is a vector of data from the current frame of the game. These data can be of different nature: integers, real values, Boolean values, ...

To describe FightingICE game states, we only used integer values (one of them describing in fact a Boolean value). These values are:

- Player's hit points,

- Opponent's hit points,
- Player's x-coordinate,
- Opponent's x-coordinate,
- Player's x-speed,
- Opponent's x-speed,
- Boolean value indicating if the opponent is attacking.

More data could have been taken into account, such as y-coordinates and speeds, energies, the type of action performed by the opponent, the player status whether available to do an action (when the player is not currently in a recovery phase, for instance), etc. For the moment, we consider a minimalist description of game states to learn utility functions of simple behaviors.

B. Utility function decomposition

All utility functions u in this study can be represented by Equation 1

$$u(\vec{x}) = coef \times combine_{i=1}^{k}\Big(transform_i(\vec{x})\Big) \qquad (1)$$

where $\vec{x}$ is a game state, $coef$ is a real value, $combine_{i=1}^{k}$ is a combination of k elements (for instance, the sum $\sum_{i=1}^{k}$) and $transform_i$ is a transformation operation extracting and transforming specific data from a given game state.

This decomposition is flexible enough to express different utility functions such as:

$$u_1(\vec{x}) = 0.5\Big(exp\_diff\_HP(\vec{x}) + log\_diff\_speed(\vec{x})\Big)$$
$$u_2(\vec{x}) = 2.Mean\Big(can\_hit(\vec{x}), logistic\_distance(\vec{x})\Big)$$

The expressive power of such utility functions depends mainly on transformation operations. We introduce them along with our ICN architecture.

C. ICN architecture

Since each utility function is modeled by an ICN, we need a generic ICN architecture that can be use to learn any kind of utility function for our target platform FightingICE. Designing a dedicated ICN architecture for each action would be too time consuming and tedious for users, for uncertain benefits.

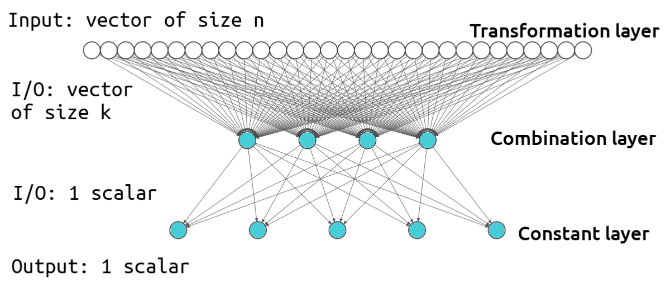

Figure 2: Our ICN model to represent utility functions in FightingICE.

The unique ICN architecture to learn our different utility functions is illustrated by Figure 2. The network is composed of three layers, each of them with their own purpose. The **transformation layer** contains 30 neurons, each with a unique elementary operation as an activation function. The goal of this layer is to extract relevant data from the game state given as input (like the player and opponent hit points for example) and to transform these data by applying a function. With our architecture, this function can be either linear, exponential, logarithmic or logistic. Due to page limit, we cannot give here the detailed list of the 30 elementary operations in this layer[1].

If we consider game states to be vectors of n values (potentially mixing integer, real and Boolean values), the transformation layer will output a vector of k values, where k is the number of its neurons that have been selected to contribute to the computation of the utility function. Let's call $\vec{y} = y_1 y_2 \dots y_k$ this k-vector. Then $\vec{y}$ is given as input to the **combination layer**, which has the mission to aggregate these k values into a unique real value. The combination layer is composed of 4 neurons with the following elementary operations: the sum of k elements $\sum_{i=1}^{k} y_i$, the mean $Mean(y_1, y_2, \dots, y_k)$, the minimum and maximum value $Min(y_1, y_2, \dots, y_k)$ and $Max(y_1, y_2, \dots, y_k)$. This outputted real value is then given to the last layer, the **constant layer**, that simply multiplies its input with a real coefficient: 0.25, 0.5, 1, 2 or 4. The final output is the utility value of the action represented by the ICN.

Each neuron of a layer is connected to each neuron of the next layer. Like written in Section II-A, weights between neurons are binary, taking the value either 0 or 1. Neurons for which the weight on the edge connecting them is equal to one are *selected*. During the learning process, we make sure that ill-formed configurations, such as selected neurons with no input values, are impossible. Blue neurons in Figure 2, corresponding to neurons from the combination and the constant layers, are mutually exclusive, meaning that exactly one neuron in each of these layers must be selected.

Binary weights, as well as a structured architecture with a specific purpose for each layer, is the key to get interpretable utility functions. Indeed, with real-valued weights as for regular neural networks, we would end up with utility functions expressed by a combination of all (non-mutually exclusive) elementary operations with different real coefficients, making unreadable functions that would be very difficult to understand.

Notice that, like artificial neural networks, an ICN can have a very different architecture than the one proposed in this paper, with different layers and different elementary operations. Notice also that the scalability property of ICN, making them able to learn functions of any arity, is not used here: we consider that game states are vectors composed of a fixed number n of values describing the

[1]However, their exact computation can be found in this C++ file.

current state of the game. There is thus no need for scalability in this context.

D. Supervised learning

We learn utility functions in a supervised learning fashion. As written in the introduction, we collect game data by playing a character and recording the game state together with the action performed. We make one game dataset for each target behavior we want to learn. Therefore, we have labeled data: for each game state, we know the action performed by the person controlling the character, allowing us to learn utility functions through supervised learning.

Observe that elementary operations within the ICN may be not differentiable (it is the case with discrete elementary operations). Therefore one cannot find their derivative and then the classic back-propagation algorithm to learn the weights of the network is not applicable. For this reason, we use a genetic algorithm to learn models of our ICNs. We reuse the same algorithm as the one learning error functions in Constraint Programming in the original paper introducing ICNs [13].

However, we must adapt both the loss function and the gene representation of individuals composing the genetic algorithm's population. Indeed, in [13], they only need to learn one ICN to model an error function. Here, we need to learn simultaneously several ICNs modeling one utility function each, and the ICNs outputs are not directly meaningful to guide the learning: here, the output of the ICN representing the desired action in the current situation must be higher than the output of other ICNs.

Hence, our loss function only considers the rank of the expected output rather than its value. Given a game state labeled with the action A, if the utility for A is the highest one, the loss equals 0, if it is the second highest one, the loss equals 1, and so on.

Observe that if our problem looks like a classification problem (choosing the right action), it is actually not: we are not aiming to learn a classifier function, but a set of utility functions. Therefore, loss functions usually employed for classification problems do not fit our needs.

More formally, our loss function corresponds to the following equation:

$$loss = Rank_A\Big(o_1, \ldots, o_m\Big) + \frac{\#\text{selected neurons}}{\#\text{neurons}} \qquad (2)$$

where o_i is the output of the i-th ICN and $Rank_A$ is the rank of the output of the ICN modeling the expected action A. The fractional term in Equation 2 is the number of selected neurons in all ICNs divided by the total number of neurons composing them. This is a regularization term to favor ICNs with fewer selected neurons, *i.e.*, to favor shorter representations of utility functions.

Since we need to learn utility functions simultaneously, their model are all encoded into an individual's gene of our genetic algorithm. In [13], a gene was a Boolean vector encoding the weight of the ICN. Here, we simply concatenate the Boolean vector of each ICN weights into a unique gene. An individual is then the weights of all our ICNs.

The total loss for a game dataset is the sum of losses over all training samples from the dataset. Since each dataset is only composed of some thousands of samples collected after 20 games in about 30 minutes (we won't require game designers to play their game for hours to collect data), we split them into training and test sets following a 5-fold cross validation procedure.

Finally, an important aspect to compute utility functions with ICNs is normalization. Since elementary operations composing a utility function are very different one from another, it is critical to have outputted values within the same range. For this, we normalized each elementary operations from the transformation layer to get values in the range [-1,1], according to the data they are dealing with. For instance, elementary operations dealing with hit points divide the given hit points by the maximal hit point for a character, and elementary operations dealing with x-speed divide them by the maximal speed (speed can be negative regarding the direction). Then, we design transformation functions such that they output values in [-1,1].

V. Empirical results

To empirically test our method, we make three distinct datasets, each of them with a characteristic behavior: aggressive, defensive, and a more complex behavior mixing the two. To build each dataset, we played 20 FightingICE games, controlling a character to make it act with the target behavior. Our method's source code, as well as experimental setups and results, are accessible on GitHub[2].

We run FightingICE with an option allowing it to write in a JSON file the game state at each frame, with the action IDs performed by both players. We then parse this JSON file to extract data that interest us: each time our character is doing an action we monitored, we extract it from the JSON file and save the vector of data constituting our game states (see Section IV-A) into a text file.

We defined our three target behaviors as follows:

- **Aggressive**: punch the opponent if it is in reach, otherwise move forward the opponent.
- **Defensive**: block if the opponent is attacking us, otherwise try to move backward to flee.
- **Hybrid**: apply the aggressive behavior first. If our hit points is at least 50 points below our opponent's hit points, then switch to the defensive behavior until the end of the game.

We have monitored the move_forward and punch actions for the aggressive behavior, move_backward and block actions for the defensive behavior, and finally these

[2]github.com/Scdevcty/LearningUtilityFunctions/releases/tag/1.1

four actions for the hybrid behavior. The total size for each dataset and the number of entries for each action composing them can be found in Table I.

Table I: Number of action entries of our different datasets

	Aggressive	Defensive	Hybrid
move forward	3,456	–	2,302
punch	4,907	–	3,256
move backward	–	4,586	755
block	–	5,053	2,725
Total	8,363	9,639	9,011

Table II: Statistics on the accuracy of our 100 models for each dataset

	Aggressive	Defensive	Hybrid
Best model	79.7%	85.2%	69.3%
Worst model	76.7%	82.4%	57.1%
Median	78.6%	83.7%	64.5%
Mean	78.5%	83.7%	65.1%
Standard deviation	0.004	0.005	0.027

Table III: Aggressive dataset: Test sets loss values of the most frequently learned model

	Loss = 0	Loss = 1
move forward	455.6 (65.9%)	235.6 (34.1%)
punch	857.6 (87.4%)	123.8 (12.6%)
Total	1,313.2 (78.5%)	359.4 (21.5%)

Table IV: Defensive dataset: Test sets loss values of the most frequently learned model

	Loss = 0	Loss = 1
move backward	820.4 (89.4%)	96.8 (10.6%)
block	786.2 (77.8%)	224.4 (22.2%)
Total	1,606.6 (83.3%)	321.2 (16.7%)

Notice that we have **unbalanced datasets**: for instance in the dataset for the hybrid behavior, we have 3,256 punches but only 755 move backward actions. This is not surprising, since there are no reasons to have a uniform number of performed actions among each type of actions in our behavior.

Our learning method also needs to deal with **noisy datasets**: human players can make mistakes while making datasets, and they can play different actions under the same situation (or very similar ones), which gives us non-deterministic datasets since the same game state can be labeled with different actions. Thus, a method for learning utility functions from such human-made data must be robust to noise to be usable in practice.

It is time to describe our experimental setup in detail. We already wrote how we collect the three different datasets and what they contain. Since we learn the models of our ICN, *i.e.*, their weights, via a genetic algorithm, running twice the learning may not lead to the same model. Therefore, we run 100 times the 5-fold cross-validation learning process for each dataset, and the results below compile statistics such as the best model, the worst one, the average accuracy over the 100 models, their median and their sample standard deviation, compiled in Table II, as well as the detailed results on test sets of the most frequently learned model for each dataset, in Tables III, IV and V. Learning one model takes about 7 minutes on a regular MacBook with a 2.3GHz Intel Core i5, independently of the number of utility functions composing the model (learning a group of 2 or 4 utility functions is done in 7 minutes for both). By parallelizing the learning of 100 models over 4 threads, running our experiments required about 3 hours for each dataset. We use the same parameters of the genetic algorithm as the ones of the original ICN paper [13].

Results from Table II are good. Notice that, unlike most of Machine Learning results, we do not focus on the best model among our 100 runs but on their average and median. Indeed, if some game designers want to use our method to learn utility function, we think the average accuracy they can expect is a more useful metric than the best one after many training instances. Another interesting metric is the median, indicating that 50 learned models over 100 show an accuracy equals to or greater than 78.5%, 83.7% and 64.5% for the aggressive, defensive and hybrid behaviors, respectively. This means that, over unbalanced and noisy datasets, 50% of learned utility function sets can find the correct action at least about 2 times over 3 for the more complex target behavior and about 4 times over 5 for the simpler ones. We also give the precision, recall and F1-score of the most frequently learned model over test sets of each dataset in Table VI.

We can see that for simple behaviors, the variance between the worst and the best models is very low. However, when the behavior becomes more complex like with the hybrid one, the variance can be significantly higher, with an accuracy going from 57.1% to 69.3%. Several factors can explain these results: of course, the behavior being more complex and with more utility functions to learn, it is not surprising that efficient models are harder to learn properly. However, looking at the median, half of the learned models reach an accuracy of at least 64.5% (where a random guess would have a 25% accuracy). Without begin impressive, this is a sufficient accuracy to obtain credible complex behavior, and it shows that our method is able to learn more complex behaviors than the basic ones. Therefore, this gap between the worst and the best model is more likely to be explained by the unbalanced and noisy nature of the data. In particular, detailed results from Table V show some evidence that unbalanced data could be problematic, as discussed below.

Tables III, IV, and V show some more detailed results on the loss values (without the regularization term which

Table V: Hybrid dataset: Test sets loss values of the most frequently learned model

	Loss = 0	Loss = 1	Loss = 2	Loss = 3
move forward	279.0 (60.6%)	161.8 (35.1%)	15.2 (3.3%)	4.4 (1.0%)
punch	550.6 (84.6%)	97.4 (15.0%)	3.2 (0.5%)	0.0 (0.0%)
move backward	0.0 (0.0%)	0.4 (0.3%)	30.6 (20.3%)	120.0 (79.5%)
block	415.2 (76.9%)	18.2 (3.4%)	73.0 (13.5%)	33.2 (6.2%)
Total	1,244.8 (69.1%)	277.8 (15.4%)	122.0 (6.8%)	157.6 (8.7%)

$$\left(x_1 - \frac{1}{1+e^{-(x_1-0.5)}} + x_2 + \frac{e^{x_2}}{2} - x_3 + \frac{e^{x_3}}{2} - \frac{1}{1+e^{-(x_3-0.5)}} + \frac{1-2*x_5}{2}\right)$$

Figure 3: Most frequently learned utility function for the move forward action in the aggressive behavior

$$4*\left(\frac{e^{x_1}}{2} + \ln(x_1+1) - \frac{1}{1+e^{-(x_1-0.5)}} - x_2 + \frac{e^{x_2}}{2} - \ln(x_2+1) + \frac{1}{1+e^{-(x_2-0.5)}} - \frac{e^{x_3}}{2} + \ln(x_3+1) + \frac{1-2\cdot x_4}{2}\right)$$

Figure 4: Most frequently learned utility function for the move backward action in the defensive behavior

$$4*\left(\frac{e^{x_1}}{2} - \ln(x_2+1) + \frac{1}{1+e^{-(x_2-0.5)}} - \frac{e^{x_3}}{2} + \frac{1-2*x_5}{2} + \frac{2*x_6-1}{2}\right)$$

Figure 5: Most frequently learned utility function for the block action in the hybrid behavior

Table VI: Precision, recall and F1-score of the most frequently learned models

	Precision	Recall	F1-score
Aggressive	0.78	0.78	0.79
Defensive	0.84	0.83	0.84
Hybrid	0.64	0.69	0.67

Table VII: Extracted and partially transformed inputs

Input	*Meaning*
x_1	Hit points diff. between the player and the opponent
x_2	Distance between the player and the opponent
x_3	Speed difference between the player and the opponent
x_4	Is the opponent attacking?
x_5	Is the opponent in range of attack?
x_6	Is the player at a border of the stage?

is not meaningful here) of the most frequently learned model over test sets of the aggressive, defensive and hybrid dataset, respectively.

Table V is certainly more interesting to analyse. We can see that the model is correctly guessing the expected action most of the time for every action but one: in this model, the utility function corresponding to the "move backward" action is unable to output a value sufficiently high to be rank first or even second when moving backward is expected, explaining why the precision, recall and F1-score in Table VI are significantly lower here than for other datasets. Actually, its output values are almost always the smallest ones. Now, moving backward is also the action with the smallest number of entries by far in the dataset, when the punch action, with the highest accuracy among the hybrid behavior's actions, also has the highest number of entries in the dataset (see Table I). Looking at Table IV, we can see that our method is perfectly able to learn good utility functions for the "move backward" action. We formulate the hypothesis that our method is currently too sensitive to unbalanced training data. Currently, it may require game designers to deactivate some rarely used actions in datasets before learning a behavior, which is of course not the desired usage of our method. Future improvements should focus of this current limitation.

Finally, we give below some examples of most frequently learned utility functions, to illustrate their interpretable nature; see equations of Figures 3, 4, and 5. At first glance, these functions may look complicated, but they are in fact simple compositions of very basic operations. To make them shorter to write for this paper, instead of writing the plain name of elementary operations of the transformation layer, we choose to display the simple final transformation operations applied to already extracted and partially transformed input data. These inputs are explained in Table VII.

VI. Conclusion

In this paper, we propose a method to learn utility functions upon data coming from game played by human beings, typically game designers and game developers looking for utility functions to reproduce a target behavior.

The originality of our method is to produce interpretable utility functions that can be easily understood, modified at will and implemented within the game logic. For this, we propose a decomposition of utility functions into elementary operations and use Interpretable Compositional Networks, or ICN, to model utility functions. Our method unlocks several scientific questions, mainly showing how to decompose utility functions and how to learn simultaneously n functions with n ICNs where the loss function implies a dependency between ICNs' outputs.

We made three distinct datasets with their own target behavior, and experimentally tested our method to learn utility functions on each of these datasets, via supervised learning with a 5-fold cross-validation. Instead of focusing on the best results, like it is often the case with Machine Learning results, we show some statistics of 100 learned models for each dataset, with the best, worst, median and mean accuracy, *i.e.*, the rate of correctly predicted action given a game state, regarding the label of this game state. Our method exhibits satisfying results with a median accuracy of 78.6%, 83.7% and 64.5% for the aggressive, defensive and hybrid behaviors, respectively. Those data are noisy since the same game situation, or very similar ones, can be labeled with different actions. These results confirm that our method is a promising direction to learn utility functions from game data.

However, we think our method may be sensitive to unbalanced data,and further work is required to get around this limitation. A first direction could be to do data augmentation. A second one could be to both consider a larger scope of data composing game states, but also to penalize models extracting too many data from these game states: forcing learned utility function to only consider the most significant values from game states to compute the utility of their action could lead to utility functions more focused to essential data, and then more efficient even with rare labels.

Although linear functions, logistic functions and alike are commonly used in utility functions for games [3], having more intuitive and game-dependent functions would certainly be welcomed by game designers and game developers. In addition, improvements can certainly be done regarding the ICN architecture: for instance, splitting the first transformation layer into two layers for data extraction and then transforming them would be welcome to have a clearer architecture and more independent layers.

References

[1] J. Bradley and G. Hayes, "Adapting reinforcement learning for computer games: Using group utility functions," in Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2005), 2005, pp. 1–8.

[2] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Learning drivers for TORCS through imitation using supervised methods," in Proceeding of the 2009 IEEE Symposium on Computational Intelligence and Games (CIG 2009), 2009, pp. 148–155.

[3] D. Graham, "An introduction to utility theory," in Game AI Pro. A. K. Peters, Ltd., 2013, ch. 9, pp. 113–126.

[4] S. Grand, D. Cliff, and A. Malhotra, "Creatures: Artificial life autonomous software agents for home entertainment," in Proceedings of the first international conference on Autonomous agents, 1997, pp. 22–29.

[5] M.-J. Kim and K.-J. Kim, "Opponent modeling based on action table for MCTS-based fighting game AI," in Proceedings of the 2017 IEEE Conference on Computational Intelligence and Games (CIG 2017), 2017, pp. 22–25.

[6] R. Konecny, "Modeling of fighting game players," Master's thesis, Utrecht University, 2016.

[7] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, , and R. Thawonmas, "Fighting game artificial intelligence competition platform," in Proceedings of the 2nd IEEE Global Conference on Consumer Electronics (GCCE 2013), 2013, pp. 320–323.

[8] S. Lueangrueangroj and V. Kotrajaras, "Real-time imitation based learning for commercial fighting games," in Proceedings of the Annual International Conferences on Computer Games, Multimedia and Allied Technology 2009, 2009.

[9] D. Mark, "AI architectures: A culinary guide," http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/, 2012.

[10] G. Martínez-Arellano, R. Cant, and D. Woods, "Creating AI characters for fighting games using genetic programming," IEEE Transactions on Computational Intelligence and AI in Games, vol. 9, no. 4, pp. 423–434, 2017.

[11] E. Onieva, D. A. Pelta, V. Milanés, and J. Pérez, "A fuzzy-rule-based driving architecture for non-player characters in a car racing game," Soft Computing, vol. 15, no. 8, pp. 1617–1629, 2011.

[12] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis, "Imitating human playing styles in Super Mario Bros," Entertainment Computing, vol. 4, no. 2, pp. 93–104, 2013.

[13] F. Richoux and J.-F. Baffier, "Error function learning with interpretable compositional networks for constraint-based local search," in Proceedings of the 2021 Genetic and Evolutionary Computation Conference (GECCO 2021), 2021 (to appear).

[14] S. S. Saini, "Mimicking human player strategies in fighting games using game artificial intelligence techniques," Ph.D. dissertation, Loughborough University, 2014.

[15] M. Sharma, M. Mehta, S. Ontanón, and A. Ram, "Player modeling evaluation for interactive fiction," in Proceedings of the AIIDE 2007 Workshop on Optimizing Player Satisfaction, 2007, pp. 19–24.

[16] A. Smith, C. Lewis, K. Hullett, G. Smith, and A. Sullivan, "An inclusive view of player modeling," in Proceedings of the 6th International Conference on Foundations of Digital Games (FDG 2011), 2011.

[17] K. O. Stanley, "Compositional Pattern Producing Networks: A Novel Abstraction of Development," Genetic Programming and Evolvable Machines, vol. 8, no. 2, pp. 131–162, 2007.

[18] C. Thurau, T. Paczian, G. Sagerer, and C. Bauckhage, "Bayesian imitation learning in game characters," in Proceedings of the 2005 International Workshop on Automatic Learning and Real-Time (ALaRT 2005), 2005, pp. 143–151.

[19] J. Togelius, R. D. Nardi, and S. Lucas, "Making racing fun through player modeling and track evolution," in Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games, 2006.

[20] B. P. Trusler and C. Child, "Implementing racing AI using q-learning and steering behaviours," in Proceeding of the 15th annual European Conference on Simulation and AI in Computer Games (GAMEON 2014), 2014.

[21] T. Wang and K. Liaw, "Driving style imitation in simulated car racing using style evaluators and multi-objective evolution of a fuzzy logic controller," in 2014 IEEE Conference on Norbert Wiener in the 21st Century (21CW), 2014, pp. 1–7.

[22] G. N. Yannakakis, P. Spronck, D. Loiacono, and E. André, "Player Modeling," in Artificial and Computational Intelligence in Games, 2013, vol. 6, pp. 45–59.

[23] G. N. Yannakakis and J. Togelius, Artificial Intelligence and Games, 1st ed. Springer Publishing Company, Incorporated, 2018.