On the Relationship between I-O Logic and Connectionism

Guido Boella
University of Torino
guido@di.unito.it

Silvano Colombo Tosatto
University of Torino
colombotosatto.silvano@gmail.com

Artur S. d'Avila Garcez
City University London
aag@soi.city.ac.uk

Valerio Genovese
University of Luxembourg
and University of Torino
valerio.click@gmail.com

Abstract

In this paper we present an embedding of (a fragment of) Input/Output logic into feed forward Neural Networks. We make use of the neural-symbolic methodology in order to allow neural networks to reason about normative systems. By doing so we are able to exploit normative reasoning within the neural networks setting. We aim at showing how neural networks can be used to represent a knowledge base of Input/Ouput logic rules and to reason about dilemma and contrary to duty problems.

1. Introduction

In this paper we study how to relate a symbolic representation of normative systems in Input/Output logic (I/O) with the computational model of Neural Networks (NN).

We employ the Neural-Symbolic paradigm that combines two different reasoning approaches: the symbolic and the connectionist. In particular, we rely on the methodology presented in (d'Avila Garcez, Lamb, and Gabbay 2002) which permits us to tackle some significant limitations of I/O logic.

I/O logic is a powerful tool to specify normative systems but lacks of important features such as learning capacity and scalability. These properties are pivotal for modeling complex, dynamic and distributed entities such as normative systems.

The symbolic nature of I/O logic presents certain limits regarding the learning capacity, like the knowledge acquisition bottleneck discussed in (Lavrac and Dzeroski 1994). Such limits are related to the *expert systems* which consist in the problem relative to the formalization of the knowledge base and the new rules that result from machine learning due to the symbolic knowledge representation. Moreover, the reasoning mechanism designed for I/O lacks of scalability w.r.t the number of rules that represent a normative code.

The integration of I/O logic with neural networks using a Neural-Symbolic paradigm (d'Avila Garcez, Lamb, and Gabbay 2002) can overcome the current limits of I/O logic by introducing a massive parallel computational model and an efficient mechanism to learn from instances. I/O logic presents a strong (and natural) similarity with NNs: both have a separate specification of inputs and outputs.

The standard semantics used for I/O based on classical logic closure (Makinson and van der Torre 2003) is purely theoretical and does not have a computational counterpart like, for instance, fixed point semantics for logical programs.

In order to define a mapping between semantics of I/O logic and neural computation, we have identified a fragment of I/O that permits a natural translation from an I/O knowledge base into a neural network.

We will show that the identified fragment can be reduced to *extended logic programs* (Brewka 1996). This approach will enable us to employ the algorithms that maps definite logical programs with NNs presented in (d'Avila Garcez, Lamb, and Gabbay 2002).

We will then study how problems related to normative reasoning such as the dilemma (Goble 2004) and contrary to duty (Wyner 2006) can be represented within the neural-symbolic paradigm.

In Sections 2 and 3 we present the Neural-Symbolic approach and introduce the I/O logic, respectively. Section 4 introduces the fragment of I/O that can be mapped into definite programs and present the translation from I/O rules into neural networks. Section 5 shows how to represent and reason about dilemma and contrary to duty problems with NNs.

2. Neural-Symbolic approach

In order to translate an I/O program into a NN capable to compute the same semantics, we use a *Neural-Symbolic* approach to obtain a sound translation. By *sound* we mean that given a set of input, the NN returns exactly the same output as the semantics of the definite fragment of I/O does. As explained in (d'Avila Garcez, Lamb, and Gabbay 2002), a Neural-Symbolic approach allows a *correct* and *complete* translation of the symbolic program because the resulting network is able to compute all and only the output of the I/O program given the input.

A *Neural-Symbolic* system aims to integrate two different reasoning approaches: the connectionist and the symbolic methodologies. Neural-symbolic systems have been applied to knowledge acquisition tasks in bioinformatics and fault diagnosis, outperforming state-of-the-art symbolic and purely connectionist systems as discussed in (d'Avila Garcez, Lamb, and Gabbay 2002). By combining the symbolic and connectionist approaches, neural-symbolic systems may also help integrate existing (legacy) systems with

new technology in either of these areas. More generally, they provide a principled way of combining reasoning and robust learning in a unified computer science model.

In a *Neural-Symbolic* approach both approaches are exploited so that the deficiencies of both can be compensated by their advantages. In this way it is possible to take advantage of the symbolic representation of the knowledge and use NNs massive parallelism that allows a better scalability along with their robustness and learning capacity. (Thrun et al. 1991) describes how the learning mechanisms of NNs can outperform symbolic learning and are more tolerant to noisy data.

As described in (Hilario 1995), there are two categories of *Neural-Symbolic* approaches, *Hybrid Systems* which are composed by interacting connectionist and symbolic systems and *Unification Systems* consisting in performing a symbolic computation with connectionst systems. Even if NNs are believed to be black box systems, (Kurfess 1997) discusses how they can be employed for the representation of symbolic knowledge; in (d'Avila Garcez, Lamb, and Gabbay 2002) is described how symbolic knowledge can be embedded within a NN and later, after having enhanced the network exploiting their learning capacity, how it is possible to extract the symbolic knowledge from the weights and thresholds inside the network.

3. Input/Output Logic

The I/O logic, is a branch of the *deontic logic* which formalizes obligation and permissions. Also I/O logic captures normative concepts and provides a formal mechanism for reasoning about obligations and permissions.

In I/O logic norms are represented as ordered pairs of formulas like (α, β), meaning: if α is present in the current situation then β should be the case.

These two formulae are also named correspondingly *input* and *output*, to make clear the fact that the input of the norm is the current situation and what is desirable for this situation is the output.

If we consider the I/O rules as norms that specify what should be done in a specific situation[1], we could then consider a normative code as built by a set of I/O rules. We can call $\mathbf{G}$ our normative code and considering a set of inputs $\mathbf{X}$ composed by a collection of atoms χ_1, χ_2, ..., χ_n as an hypothetical situation, then we could try to process the situation supplied from $\mathbf{X}$ with our normative code $\mathbf{G}$.

In this view we can define different kinds of relations between $\mathbf{X}$ and $\mathbf{G}$. For instance, *simple minded output* is the output given from the processing of $\mathbf{G}$ on *Cn*($\mathbf{X}$) (the classical closure of $\mathbf{X}$) in this case for each χ_n into $\mathbf{X}$ the normative code will check its norms (α, β) and produce an output β if it finds a correspondence between χ_n and the input α of the norm. The output obtained is then $\mathbf{G}(Cn(\mathbf{X}))$, but we can also apply to this the classical closure so we obtain what is represented as: $out_1(\mathbf{G}, \mathbf{X}) = Cn(\mathbf{G}(Cn(\mathbf{X})))$.

[1]Unlike deontic logic, which tries to apply truth value to the norms

Simple minded output

The *simple minded output* is a relevant mechanism of the I/O logic which can be extended to obtain more complex inference mechanisms.

We introduce now the three rules that characterize the *simple minded output*.

1. *Strengthening Input* ($\mathbf{SI}$): From (α, χ) to (β, χ) whenever $\alpha \in Cn(\beta)$
2. *Conjoining Output* ($\mathbf{AND}$): From (α, χ), (α, γ) to $(\alpha, \chi \wedge \gamma)$
3. *Weakening Output* ($\mathbf{WO}$): From (α, χ) to (α, γ) whenever $\gamma \in Cn(\chi)$

Besides those rules there is another important feature of the *simple minded output* that we should consider: the feature of not carrying the input into the output (unless a rule explicitly dictates so). The principle of carrying the input into the output is called *identity principle* but, in this context, it assumes the name of *throughput*.

Further outputs

Although the *simple minded output* is an interesting semantics for I/O rules, it lacks of some properties that could be useful in particular circumstances. One of this circumstances could be the following example: if $\chi \in out_1(\mathbf{G}, \alpha)$ and $\chi \in out_1(\mathbf{G}, \beta)$ then $\chi \in out_1(\mathbf{G}, \alpha \vee \beta)$. It is impossible to understand if χ is derived from α or β without using the input disjunction. The *simple minded output* that employs the input disjunction is called *basic output*.

Disjoining Input ($\mathbf{OR}$): From (α, χ), (β, χ) to $(\alpha \vee \beta, \chi)$

Another plausible property highlighted in (Makinson and van der Torre 2003), is the *reusability*, that allows the output to be carried in the input pool and be reused. The rule that permits the *reusability* takes the name of *Cumulative Transitivity* and the mechanism that implements it is called *reusable output*.

Cumulative Transitivity ($\mathbf{CT}$): From (α, χ), $(\alpha \wedge \chi, \gamma)$ to (α, γ)

Then, if we merge the two rules introduced above, we can obtain the *reusable basic output* containing both input disjunction and reusability. We can formally define it as $out_4(\mathbf{G}, \mathbf{A}) = \cap \{Cn(\mathbf{G}(\mathbf{V})): \mathbf{A} \subseteq \mathbf{V} \supseteq \mathbf{G}(\mathbf{V}), \mathbf{V}$ *complete*$\}$.

Each of the four mechanisms introduced, can be further enhanced by adding the *throughput*, which can be explained considering the *simple minded output*, with $\mathbf{G}$ consisting of a single rule (α, β) and the input set $\mathbf{X}$ composed by only α. In the case we do not allow the throughput, we will have as output only β; however, if we allow the throughput, then the output will consist of the union of the generated output and the already existing input, in this case by both α and β.

The *throughput* is usually called *Identity* in other systems and must be said that this property runs against the I/O logic principle of keeping the input and the output separated.

4. From I/O logic to Neural Networks

Before taking into consideration the translation algorithm for I/O logic, we need to discuss the assumptions made to make the translation possible. In the previous section, we defined $out_1(\mathbf{G}, \mathbf{X})$ and other I/O mechanisms, where the I/O logic makes use of the classical closure. This raises a problem, because NNs do not compute the classical closure. To solve this, we simplified the input/output rules to *logic programming*, avoiding the classical closure and because the *logic programming semantics* is better suited to be computed by a NN.

In order to simplify the I/O logic rules to logic programs, we will make a few assumptions so that we can make use of the translation algorithm developed by d'Avila Garcez, Broda and Gabbay in (d'Avila Garcez, Lamb, and Gabbay 2002).

First, we have to define what is a *Extended Logic Program*:

Definition 1 *An* Extended Logic Program *is a finite set of clauses of the form* $L_0 \leftarrow L_1, ..., L_n$*, where* $L_i (0 \leq i \leq n)$ *is a literal (an atom or a classical negation of an atom, denoted by* $\neg$*).*

In order to use the algorithms to translate I/O logic rules into NNs, we have to make them comply with *Extended Logic Programs* clauses. Considering an ordered pair (α, β), where α and β are both conjunctions of literals, we can see it in a clause shape like this: $\beta \leftarrow \alpha$. What we want to obtain are clauses like the following: $L_0 \leftarrow L_1 \wedge ... \wedge L_n$, where the head is composed by a single literal and the body by a conjunction of literals.

Considering the input as the clause body is automatic because both are composed by conjunctions of literals so we do not have to make any other assumption. However when it is time to map the output into the head of the rule, we have to constrain th output to be composed by a single literal.

As reported in (d'Avila Garcez, Lamb, and Gabbay 2002), the literal in the head of a clause must be positive in order to be translated into a neural network using the *Connectionist Inductive Learning and Logic Programming System* algorithm. To overcome this problem, we have decided to always consider the literals in the output of the rule (the head for the clause) as positive. Considering the example $(\alpha \wedge \gamma, \neg\beta)$: this cannot be translated. Instead, if we start to consider $\neg\beta$ not as the negation of β but as a new positive atom that we can call β', with the same mean as $\neg\beta$, then the new I/O rule resulting from this translation $(\alpha \wedge \gamma, \beta')$ is compliant with the algorithm requests and can be translated.

In this way we are not using the *negation by failure*, where an atom is considered false if it is not explicitly declared as true. Then because we are not considering an unknown fact as a negated one, we have to use the explicit negation for the cases where a negative output is useful, like for a norm that in a certain situation forbids to do something.

An obvious question that can be raised after making the I/O rules viable for translation is the following: what is going to happen if two separate I/O rules give opposite outputs like β and $\neg\beta$?
With this I/O logic simplification, we have the advantage to introduce the explicit negation needed for the translation and useful for normative reasoning. In this way we also introduce *dilemmas*(outputs containing contradictions like β and $\neg\beta$) that must be handled in order to maintain the system consistent as we are going to discuss in section 5.

After making this assumptions, we can consider a set of I/O logic rules as an *Extended Logic Program*. Considering G a *normative code*, a set of I/O rules like (α_1, β_1), ..., (α_n, β_n), where α_i = $\{\ l_{i_1} \wedge ... \wedge l_{i_m}\ \}$, a conjunction of literals and $\beta_i = l_{i_0}$, a single literal. So to resume (α, β) = $(l_{i_1} \wedge ... \wedge l_{i_m}, l_{i_0})$.

We are going to define what is the input of a *normative code* as $I(G)$ = $\{\ a_{1_1}; ...; a_{i_j}; ...; a_{n_m}\}$ where $(1 \leq i \leq n)$ and $(1 \leq j \leq m)$ as the conjunction of the atoms belonging to the literals in the Inputs of the rules.

The output of a *normative code* is defined as $O(G)$ = $\{l_{1_0}; ...; l_{i_0}; ...; l_{n_0}\}$ where $(1 \leq i \leq n)$ as the set of literals in the output of the rules.

Before redefining the properties of I/O logic we need to explain that in: $(l_{I_1} \wedge ... \wedge l_{I_m}, l_{O_1} \wedge ... \wedge l_{O_n})$ we mean that by activating the the input layer neurons of the network like the literals in the input part of the rule, what we obtain as a result of the computation is what is shown in the Output part of the rule. Otherwise in the case we have disjunctions in the Input parts $(l_{I_1} \vee ... \vee l_{I_m}, l_{O_i})$ means that by activating at least on neuron in the input layer of the network corresponding to one of literals shown in the Input part, the result that is computed is what is shown in the Output part.

We have redefined the I/O logic properties to be translated into neural networks.

- *Strengthening Input*: From (α, χ) to (β, χ) whenever $\alpha \in$ *Cn*(β): $\frac{(\alpha,\chi)}{(\beta_1 \wedge ... \wedge \alpha \wedge ... \wedge \beta_n, \chi)}$ with $\beta_1, ..., \beta_n \subseteq I(G)$
- *Conjoining Output*: From (α, χ), (α, γ) to $(\alpha, \chi \wedge \gamma)$: $\frac{(\alpha,\chi)(\alpha,\gamma)}{(\alpha,\chi\wedge\gamma)}$
- *Weakening Output*: From (α, χ) to (α, γ) whenever $\gamma \in$ *Cn*(χ): $\frac{(\alpha,\gamma_1 \wedge ... \wedge \chi \wedge ... \wedge \gamma_n)}{(\alpha,\chi)}$ with $\gamma_1, ..., \gamma_n \subseteq O(G)$
- *Disjoining Input*: From (α, χ), (β, χ) to $(\alpha \vee \beta, \chi)$: $\frac{(\alpha,\chi)(\beta,\chi)}{(\alpha\vee\beta,\chi)}$
- *Cumulative Transitivity*: From (α, χ), $(\alpha \wedge \chi, \gamma)$ to (α, γ): $\frac{(\alpha,\chi)(\alpha\wedge\chi,\gamma)}{(\alpha,\gamma)}$

Translation algorithms

Having shown the assumptions made on the I/O rules and how we redefined I/O properties, we can now introduce an outline of the translation algorithms that we are going to use to build NNs from the I/O rules, the *Connectionist Inductive Learning and Logic Programming System* and the *Metalevel Priorities* algorithms described in (d'Avila Garcez, Lamb, and Gabbay 2002).

Connectionist Inductive Learning and Logic Programming System algorithm

For each rule R_i = $(\alpha_1 \wedge ... \wedge \alpha_n, \beta)$ in a normative code $\mathbf{G}$ do:

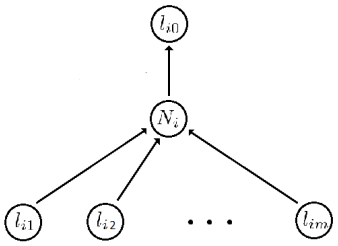

Figure 1: Translation example of a generic rule $(l_{i1} \wedge ... \wedge l_{im}, l_{i0})$.

1. For each atom $\alpha_j (1 \leq j \leq n)$ in the input of the rule: If there is no input neuron labeled α_j in the input level, then add a neuron labeled α_j in the input level.
2. Add a neuron labeled N_i in the hidden level.
3. If there is no input neuron labeled β in the output level, then add a neuron labeled β in the output level.
4. For each atom $\alpha_j (1 \leq j \leq n)$ in the input of the rule: Connect the respective input neuron with the neuron labeled N_i in the hidden level with a positive (negative) weighted arc if α_j is positive (negative).
5. Connect the neuron labeled N_i with the neuron in the output level labeled β with a positive weighted arc[2].

In *Figure* 1 is shown the neural network translation of a generic rule $(l_{i1} \wedge ... \wedge l_{im}, l_{i0})$.

Metalevel Priorities Algorithm This algorithm is used to embed priority-based ordering within NNs. We will use this algorithm to translate two types of ordering, one for the *Dilemma* where the outputs of the rules are in contradiction, for instance (α, χ) and $(\beta, \neg\chi)$. The other case is relative to *Contrary to Duty* problem where the output of one rule is in contradiction with one of the inputs of the other, such as (α, χ) and $(\neg\chi, \gamma)$. Both cases are treated equally by the algorithm so we are going to consider some general rules as $R_g = (\alpha_g, \gamma)$ and $R_h = (\beta_h, \mu)$[3].

For each ordering rule $R_g \succ R_h$ in a normative code **G** do:

1. Connect the hidden neuron labeled H_g with the output neuron labeled μ using a negative weighted arc.

The *Metalevel Priorities* algorithm function is to add those negative weighted arcs between the hidden level and the output level of the network, which act by inhibiting the output neurons of the lower priority rule if the higher one is activated.

Using the *Connectionist Inductive Learning and Logic Programming System* and the *Metalevel Priorities* algorithms, we are able to build NNs that behave like the normative codes from which they are translated by interpreting

[2]Note that due to our previous assumptions β is always considered a positive atom so the arc between a hidden neuron representing the rule and the output neuron representing its output will be always positively weighted.

[3]In a *Dilemma* case we have $\gamma = \neg\mu$. Instead in a *Contrary to Duty* case we have $\gamma = \neg\beta$ where $\beta \in \beta_h$

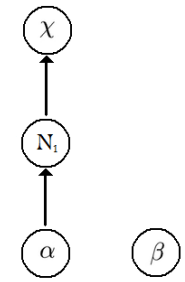

Figure 2: Strengthening Input translated into a neural network.

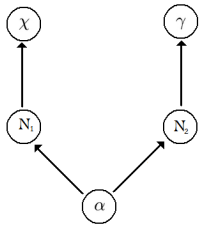

Figure 3: Conjoining Output translated into a neural network.

a rule like $(l_{i1} \wedge ... \wedge l_{im}, l_{i0})$ as: if the input layer neurons corresponding to the Input part are activated, then the neural network will compute the output corresponding to the output shown in the output of the rule. This is proven in (d'Avila Garcez, Lamb, and Gabbay 2002).

Neural Networks for the Input/Output rules

In this section we will show the NNs that compute the six I/O rules introduced earlier.

For the sake of simplicity we will discuss only the architecture of the network, omitting the weights and activation thresholds.

Strengthening Input

Definition 2 $\frac{(\alpha,\chi)}{(\beta_1 \wedge ... \wedge \alpha \wedge ... \wedge \beta_n, \chi)}$ *with* $\beta_1, ..., \beta_n \subseteq I(G)$

We translated the antecedent with the algorithm displayed in the previous section and verified if the consequent is computed in the constructed network.

In *Figure* 2 we can see how the consequent is verified by the network because if we consider another input neuron β as any of the other possible inputs of the network $\beta_1, ..., \beta_n \subseteq I(G)$, that does not interfere with the norm in the antecedent, then it is obvious that $\alpha \wedge \beta$ is equivalent to have only α.

Conjoining Output

Definition 3 $\frac{(\alpha,\chi)(\alpha,\gamma)}{(\alpha,\chi\wedge\gamma)}$

Figure 3 shows how α, contained in both rules as the input, can alone generate the output $\chi \wedge \gamma$. We obtain $(\alpha, \chi \wedge \gamma)$ by the natural composition of the network.

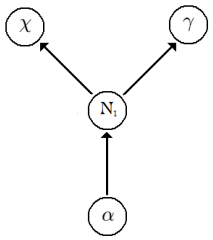

Figure 4: Weakening Output translated into a neural network.

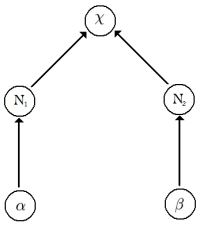

Figure 5: Disjoining Input translated into a neural network.

Weakening Output

Definition 4 $\frac{(\alpha,\gamma_1\wedge...\wedge\chi\wedge...\wedge\gamma_n)}{(\alpha,\chi)}$ *with* $\gamma_1,...,\gamma_n \subseteq O(G)$

In *Figure* 4 we have the resulting neural network of the translation. The network is similar to the one built from the conjoining output. The difference with the network in figure 3 lies in the hidden level of the two networks, in the one referenced to the weakening output there is only one neuron associated to the rule $(\alpha, \chi \wedge \gamma)$, instead in the network relative to the *Conjoining Output* there are two neurons in the hidden level, because of the two rules that compose the NN.

Disjoining Input

Definition 5 $\frac{(\alpha,\chi)(\beta,\chi)}{(\alpha\vee\beta,\chi)}$

In *Figure* 5 is shown the network resulting from the translation of the two rules (α, χ) and (β, χ), it is obvious that this network can compute successfully the rule ($\alpha \vee \beta$, χ), because only one of the two possible input is necessary to produce χ in the output level. However the true task of the rule is to help us to understand which of the two inputs is generating the output. This is obtainable by considering the two initial rules in two separated steps, in this way we can say which one is the cause of the presence of the positive output.

For each of the four NNs introduced before, the computation is clamped after the first run and not allowing the net to stabilize. This because the networks must behave like the I/O mechanisms, that process the input only once and producing the output.

Transitivity

Definition 6 $\frac{(\alpha,\chi)(\alpha\wedge\chi,\gamma)}{(\alpha,\gamma)}$

Figure 6 shows how we translated the *Cumulative Transitivity* into a NN. What we obtain from the translation is the

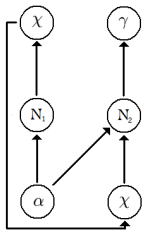

Figure 6: Cumulative Transitivity translated into a neural network.

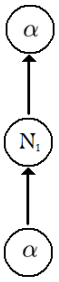

Figure 7: Throughput rule translated into a neural network.

Transitivity because the combination of the *cumulative transitivity* and the *Strengthening Input* results in the transitivity that is stronger and implies the *Cumulative Transitivity*. This happens because in the *simple minded output* and the other mechanism, the *Strengthening Input* rule is present.

For this translation a simple application of the algorithm to the I/O rules is not enough because, in this case, we have to bring back the output to be reused as input for the network. *Figure 5* highlights the connection that brings back the output to the input because, in this case, we will not clamp the computation and let the network to stabilize.

Throughput Besides the rules described, we should also consider the *throughput* and how to translate it into a NN. This property, usually called *identity*, can be translated into a network by adding to the normative code a rule for each input atom present as an input in the rules, with that atom both as input and output. In this way, after the translation, every input will be carried on as an output.

Figure 7 shows how a rule (α, α) built to allow the *Throughput* is translated.

5. Deontic logic problems

Because of his origins in the deontic logic, the I/O logic is affected by the same problems. Here we are going to discuss some of this problems and, in particular, the *dilemma* and the *contrary to duty*.

Dilemma problem

We have already mentioned the dilemma problem earlier, while discussing the assumptions adopted to allow the I/O rules to be considered logic programs. A *dilemma* occurs

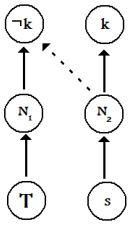

Figure 8: Soldier's dilemma rules translated into a neural network.

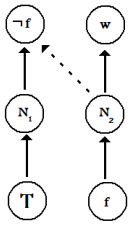

Figure 9: Neural network of the contrary to duty example.

when a system produces a contradictory output, for example β and $\neg\beta$. We must assume that this contradiction is not caused by a faulty system or because of some wrong rules as in this case we must not consider the contradiction a *dilemma* but a system error to be solved. A classic dilemma example is Sartre's soldier, that has the duty to kill and the moral obligation not to kill. In this case, we can see the two obligations as rules that produce both k and $\neg k$, the soldier dilemma that has to choose if to kill or not to kill.

R_1**:** $(T, \neg k)$

R_2**:** (s, k)

The rules R_1 and R_2 represent the soldier's dilemma. A way to solve this dilemma is to prioritize one rule over another using a *priority-based ordering*, we can say that $R_2 \succ R_1$ because *we have decided* to favor the most specific of the two. Other dilemmas can be solved by using a different way to choose the preferred rule, it is a user choice.

In *Figure* 8 is represented how the algorithm handles the ordering between the rules. The dotted arrow created from the preference ordering $R_2 \succ R_1$, has a negative weight and blocks the activation of the output $\neg k$ if R_2 is activated.

Contrary to duty problem

We can describe a *contrary to duty* as the circumstance in which we have to deal with an exceptional situation but that could be still considered legal. What follows is an example of a contrary to duty problem: we introduce an example similar to the one made by Marek Sergot from (Prakken and Sergot 1996) and used by D. Mackinson and L. van der Torre, in which he considers whether a cottage should have a fence or not. This can be described by means of two simple i/o rules:

1. $(T, \neg f)$, in this rule we have a tautology in input, so it always gives the output $\neg f$ and states: A cottage should not have a fence.
2. (f, w), this rule states: If a cottage has a fence, then it must be white.

Those rules are enough to build a contrary to duty problem. The first rule is the one that should be applied in an optimal situation. Rule number 2 is the one which effectively tells what to do in a suboptimal situation.

A *contrary to duty problem* turns out when a i/o rule states that α must be true (or false) whilst another rule considers what to do in the case α is false (or true). The two following rules are what is necessary to form a contrary to duty problem.

1. (T, α)
2. $(\neg\alpha, \beta)$

After having explained what a *contrary to duty problem* is, we will now discuss how to treat it. As already described in (van der Torre and Tan 1999), we can solve the contrary to duty problem using a preference ordering of the rules. Considering the cottage example, in a situation where we have a fence, we want to consider the situation acceptable if the fence is white, even if the first rule violated. In order to do so and solve the *contrary to duty problem*, if the second rule (f, w) is complied, then we should consider the situation acceptable even if the first rule is violated. Formally we added a priority-based ordering in the example: $(f, w) \succ (T, \neg f)$.

Therefore, in order to allow this bypassing of rules, we can introduce a preference ordering where the contrary to duty rule inhibits the rule that should be applied in the optimal situation, so that the system can be kept safe from any contradiction. Note that as in the *Dilemma* problem, priority-based orderings are supplied to the system by the user.

Contrary to duty problem translation into a Neural Network We will now show this example translated into a NN. These are the example's rules:

R_1**:** $(T, \neg f)$

R_2**:** (f, w)

$R_2 \succ R_1$

Figure 9 shows the resulting network for the contrary to duty example after the translation. In the picture we omitted weights and thresholds because they are not too relevant for this example; we have instead highlighted the connection showed as a dotted line because it was built from the priority rule $R_2 \succ R_1$ allowing the network to handle the contrary to duty. So in the case that the neuron N_2 is activated, the negative weight on the red connection, inhibits the activation of the output of the neuron $\neg f$, the one that should be activated by N_1 that is built from the rule $(T, \neg f)$, exactly what we want from $R_2 \succ R_1$.

Adding violation nodes In the way just described we can keep the network consistent in presence of contrary to duty rules but we lose any data about which rules are violated. In

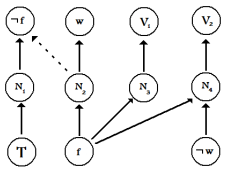

Figure 10: Neural network of the contrary to duty example extended with violation rules.

order to maintain this information we can provide the normative code with some factitious rules that can keep track of occurred violations. These *violation rules* are built in this way:

1. For each rule $r_1, ..., r_n$ in the Normative Code $\mathbf{G}$ where $r_i = (\alpha_i, \beta_i)$ and $1 \leq i \leq n$:

   - Build a new rule $r_{n+i} = (\alpha_i \wedge \neg\beta_i, V_i)$

After the new rules to the normative code have been added, we can translate it with the algorithms presented.

In *Figure* 10, we added *violation rules* to the contrary to duty example and translated the resulting set of rules:

R_1**:** $(T, \neg f)$

R_2**:** (f, w)

R_3**:** (f, V_1)

R_4**:** $(f \wedge \neg w, V_2)$

$R_2 \succ R_1$

In the network an activation of an output neuron V_i means a violation of the rule i.

6. Conclusions and Future Work

We described in this paper how a fragment of I/O logic that can be reduced to *Extended Logic Programs*, can be translated into *Neural Networks* with the algorithms described. We have also discussed how to handle the problems related to the *Normative reasoning*: the *Dilemma* and the *Contrary to duty* problem, by means of a *preference-based ordering* and describing the effects that our solutions have on the translated networks.

For the *Contrary to duty* problem translation, we added the *violation nodes* in order to recognise the violations that occur in a system, despite the legal situation due to contrary to duty rules.

This is a first attempt to relate the two fields of I/O logic and NNs to the best of our knowledge. This is important because for now we only got initial results, a lot remains to do in the area. However, the prospects on this regards are very promising, due to the natural relationships between the two approaches as input-output systems, the robust learning and effective parallel computation capabilities of NNs, and the importance of deontic logic in a range of application domains such as law, multi-agent systems and others.

As future work we plan to study how the learning capability of NNs can be exploited in learning new normative rules. For instance, in (d'Avila Garcez, Lamb, and Gabbay 2002) it has been shown how the standard backpropagation algorithm can be used to improve the knowledge that resides in the network. Moreover, when in normative systems a situation that violates the norms occurs but it is judged legal, the precedent created actually changes the normative system. In this case, the NNs capacity to learn from instances can be exploited to evolve normative systems.

Further work can be done by translating more complex normative systems and verifying to what extent a *Neural-Symbolic approach* can improve the efficiency of compliance and norm violation checking.

Acknowledgments Valerio Genovese is supported by the National Research Fund, Luxembourg.

References

Brewka, G. 1996. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research* 4.

d'Avila Garcez, A. S.; Lamb, L. C.; and Gabbay, D. M. 2002. *Neural-Symbolic Learning Systems*. Perspectives in Neural Computing Series. Springer, LLC.

Goble, L. 2004. A proposal for dealing with deontic dilemmas. In *Deontic Logic*. Springer Berlin / Heidelberg.

Hilario, M. 1995. An overview of strategies for neurosymbolic integration. In *Workshop on Connectionist-Symbolic Integration: From Unified to Hybrid Approaches*. IJCAI 95.

Kurfess, F. J. 1997. *Neural Networks and Structured Knowledge. In C*. Logos-Verlag, Berlin.

Lavrac, N., and Dzeroski, S. 1994. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York.

Makinson, D., and van der Torre, L. 2003. What is input/output logic? *Foundations of the Formal Sciences II: Application of Mathematical Logic in Philosophy and Linguistics*.

Prakken, H., and Sergot, M. J. 1996. Contrary-to-duty obligations. *Studia Logica* 57(1):91–115.

Thrun, S. B.; Bala, J.; Bloedorn, E.; Bratko, I.; Cestnik, B.; Cheng, J.; Keller, S.; Kononenko, I.; Kreuziger, J.; Michalski, R. S.; Mitchell, T.; Pachowicz, P.; Reich, Y.; Vafaie, H.; de Welde, K. V.; Wenzel, W.; Wnek, J.; and Zhang, J. 1991. The monk's problems: A performance comparison of different learning algorithms. cmu-cs-91-197. Technical report, Carnegie Mellon University.

van der Torre, L., and Tan, Y.-H. 1999. Contrary-to-duty reasoning with preference-based dyadig obligations. *Annals of Mathematics and Artificial Intelligence 0*.

Wyner, A. 2006. Sequences, obligations, and the contrary-to-duty paradox. 255–271.