Incremental learning with state features in gErl

No Author Given
No Institute Given

Abstract. gErl is a new general learning system with an intrinsic ability to incrementally learn from related tasks. gErl, as an Erlang-coded general learning system, allows the users to write (or adapt) their own operators in Erlang, according to the problem, data representation and the way the information should be navigated. As changing operators affect how the search space needs to be explored, heuristics are learnt as a result of a decision process based on reinforcement learning where each action is defined as a choice of operator and rule. In this paper we show one specific feature of gErl, namely the reuse of knowledge acquired solving a specific task that may accelerate the learning of the same or related task due to the abstraction of states and actions using features.

Keywords: machine learning operators, complex data, incremental learning, transfer learning, inductive programming, reinforcement learning, Erlang.

1 Introduction

One way of reducing training time is by learning a source task before learning the target task [11]. While previous transfer work has mainly focused on reducing training time on closely related tasks by transferring from a simple to complex task in a single domain, we show a (potentially) more powerful way of simplifying this transfer task formulating it as an abstraction of states and actions.

In this paper we present and explore gErl [1], a general rule-based learning system where operators can be defined and customised for each kind of problem. While one particular problem may require generalisation operators, another problem may require operators which add recursive transformations to explore the structure of the data. A correct choice of operators can embed transformations on the data but can also determine the way in which rules are generated and transformed, so leading to (apparently) different learning systems. Making the user of the problem adapt its own operators is powerful but significantly more difficult than the use of feature transformations or specific background knowledge, since operators can be very complex things and usually embed the essence of a machine learning system.

From here, we devise a flexible architecture which works with populations of rules and programs, which evolve as in an evolutionary programming setting or a learning classifier system [4]. Operators are applied to rules generating new rules, which are combined with existing or new programs. With appropriate operators and using some optimality criteria (based on coverage and simplicity)
we will eventually find some good solutions to the learning problem. However, without heuristics, the number of required iterations gets astronomically high. This issue is addressed with a reinforcement learning (RL) approach, where the application of an operator over a rule is seen as a decision problem, for which learning also takes place, guided by the optimality criteria which feed a rewarding module. Interestingly, the system is able to reuse knowledge acquired in the current learning process in order to improve or accelerate the learning of future tasks in a totally transparent way due to the feature space used to describing states and actions.

The paper is organised as follows. Section 2 makes a short account of the related and previous work. Section 3 introduces the gErl system and the RL-based heuristics used to guide the learning process. Section 4 discusses how tasks can take advantage of previous knowledge in terms of the abstracted Q-value function. Section 5 includes an empirical study. Section 6 closes the paper.

2 Previous work

Although the system we use in this paper is related to different areas of machine learning: learning from complex data, reinforcement learning, Learning Classifiers Systems, evolutionary techniques, meta-learning, etc., in this section we will only focus on its ability to transfer previous acquired knowledge between tasks and summarise some of the previous transfer learning (TL) works in RL, which are most closely related to our proposal (for a survey, see [11]).

Several methods improve learning by introducing additional knowledge into the exploration and exploitation choices. The main idea consists of breaking down a single large target task into a series of simpler source tasks, where transfer learning can be performed by initializing the Q-values of a new episode with previously learnt Q-values [3, 6]. Other algorithms introduce previous knowledge through macro-actions or options [10] (somewhat more general actions) to learn (with less data) new action policies in Semi-Markov Decision Processes.

Another way to transfer learning can be performed by initialising the Q-values of a new episode with previously learnt Q-values. However, in almost all cases it cannot be directly applied because explicit mappings are needed in order to transfer between tasks with different actions and state representations. This inter-task mapping may be provided to the learner or may be autonomously learnt. In the first subgroup we found methods as the advice rules [7] where the user advise appropriate actions to be preferred in different sets of states or by composing solutions of elementary sequential tasks [8]. In the second subgroup we can found methods which work with graph mapping techniques that can automatically find a mapping between actions and state variables in two tasks as in qualitative dynamic Bayes networks [5].

Although TL in RL between related problems has made significant progress in recent years, our work aims to be more transparent, without the need of human experts or the use of inter-tasks mappings. In order to do this, we use an abstract description of the knowledge transferred because of the feature-based representation of the Q-value function. In addition, our work may also follow
the idea of learning from easy missions [2], where the learning of a hard problem could be solved in an incrementally way.

3 The gErl system

As we have mentioned in Section 1, in this paper we use the gErl system [1], a general learning system which can be configured with different (possibly user-defined) operators. As a long-term goal, this can be roughly seen as a general system for designing customised systems for applications with complex data. It can be described as a flexible architecture which works with populations of rules (expressed as unconditional / conditional equations) and programs in the functional language Erlang, which evolve as in an evolutionary programming setting or a learning classifier system [4]. Operators are applied to rules and generate new rules, which are combined with existing or new programs. With appropriate operators, using some optimality criteria (based on coverage and simplicity) and using a reinforcement learning-based heuristic (where the application of an operator over a rule is seen as a decision problem fed by the optimally criteria) many complex problems can be solved. Figure 1 shows the general architecture of gErl. In the rest of this section we will resume the main characteristics of the system. We refer the reader to [1] for more details.

![gErl's system architecture](image)

3.1 Data and model representation and learning problem statement

gErl can be considered an inductive (functional) programming system, as data and rules (and hence solutions) are represented in a functional programming language, Erlang. Let us see how these elements are represented and how the learning problem is stated.

An equation is an expression of the form $l = r$ where $l$ and $r$ are terms. $l$ is called the left hand side (lhs) of the equation and $r$ is the right hand side (rhs). $R$ denotes the space of all (conditional) functional rules $\rho$ of the way $l$ [when $G \rightarrow T, r$ where $l$ and $r$ are the lhs and the rhs of $\rho$ (respectively), $G = \{g_1, g_2, \ldots, g_m \mid m \geq 0\}$] is a set of conditions or Boolean expressions called guards, and $T = b_1, \ldots, b_n$, the tail of $\rho$, is a sequence of equations. If $G = \emptyset$, then $\rho$ is said to be an unconditional rule. An expression (term, equation or
rule) is *ground* if it does not contain any variable symbol. As usual, variables are denoted by a sequence of letters and digits starting with a capital letter. An example \( e \) is a ground rule \( l \to r \) (that is, without condition nor tail), being \( r \) in normal form. Let \( \mathcal{P} = 2^\mathcal{R} \) be the space of all possible functional programs formed by sets of rules \( \rho \in \mathcal{R} \). We say that an example \( l \to r \) is covered by a program \( p \) (denoted by \( p \models \{ l \to r \} \)) if \( l \) and \( r \) have the same normal form with respect to \( p \). A functional program \( p \in \mathcal{P} \) is a solution of a learning problem defined by a set of positive examples \( E^+ \), a (possibly empty) set of negative examples \( E^- \) and a background theory \( B \) if it covers all positive examples, \( B \cup p \models E^+ \) (completeness), and does not cover any negative example, \( B \cup p \not\models E^- \) (consistency). Our system has the aim of obtaining complete solutions, but their consistency is not a mandatory property, so approximate solutions are possible.

The function \( \text{Cov}^+ : 2^\mathcal{R} \to \mathbb{N} \) calculates the positive coverage of a program \( p \in 2^\mathcal{R} \) and it is defined as \( \text{Cov}^+(p) = \text{Card}(\{ e \in E^+ : B \cup p \models e \}) \), where \( \text{Card}(S) \) denotes the cardinality of the set \( S \). The negative coverage \( \text{Cov}^- \) is defined analogously.

As we can see in Figure 1, our system works with two sets: a set of rules \( \mathcal{R} \subseteq \mathcal{R} \) and a set of programs \( \mathcal{P} \subseteq \mathcal{P} \), where each program \( p \in \mathcal{P} \) is composed of rules belonging to \( \mathcal{R} \). Initially, the set of rules \( \mathcal{R} \) is populated with the positive evidence \( E^+ \) and the set of programs \( \mathcal{P} \) is populated with as many unitary programs as there are rules in \( \mathcal{R} \).

### 3.2 Operators and learning process

The definition of customised operators is one of the key concepts of our proposal. The main idea is that, when the user wants to deal with a new problem, s/he can define her/his own set of operators \( O \in \mathcal{O} \) to suit the data structures of the problem. This feature allows our system to adapt to the problem at hand.

An operator can be seen as a piece of code (as complex as the user may want) which performs modifications over the \textit{lhs} or \textit{rhs} of a rule and which is written in the same functional language of the system (Erlang) to take advantage of its high-order and reflection capabilities. Our system also has a special kind of operators \( C \), called \textit{combiners}, that only apply to programs. The \textit{Program Generator} module (Figure 1) applies a combiner to the last rule \( \rho' \) generated by the \textit{Rule Generator} module and the population of programs \( \mathcal{P} \).

As the process progresses, new rules and programs will be generated. First, the \textit{Rule Generator} process (Figure 1) gets the operator \( o \) and the rule \( \rho \) returned as an action \( a = \langle o, \rho \rangle \) by the \textit{Reinforcement Learning Module} (policy) we will explain in the next section. This process applies the operator over the rule obtaining a new rule \( \rho' \) (if the operator is not suitable for the rule selected, the process returns the same rule) which is added to \( \mathcal{R} \). Then, the \textit{Program Generator} module takes the new rule generated \( \rho' \) (if appropriate) as input, the set of programs \( \mathcal{P} \) and the set of combiners \( C \) and generates a new program \( p' \) (which is added to \( \mathcal{P} \)) applying the combiners over the previous inputs.
3.3 The Reinforcement Learning process

In order to give the users the freedom to define their own operators, the search space heuristics in gErl has been conceived as decisions about the operator that must be used at each particular state of the learning process. For this, a model-based reinforcement learning approach has been developed guided by the optimality criteria which feed a rewarding module.

**RL problem statement** We model the decision process as a typical reinforcement learning task. Formally, our decision problem is a four-tuple \( (S, A, \tau, \omega) \) where: \( S \) is the state space, where a state at each iteration \( t \) of the system is represented as a tuple \( \sigma_t = (R, P) \) being \( R \) and \( P \) the population of rules and programs in \( t \), respectively; \( A \) is a finite actions space \( \mathcal{A} = \mathcal{O} \times \mathcal{R} \); \( \tau : S \times A \rightarrow S \) is a transition function between states and \( \omega : S \times A \rightarrow \mathbb{R} \) is the reward function. These components are defined below:

- **States.** As we want to find a good solution to the learning problem, we describe each state \( \sigma_t \) by a tuple of features \( s_t = (\phi_1, \phi_2, \phi_3) \) from which to extract relevant information in \( t \):

  **Global optimality** \( (\phi_1) \): This feature shows the average optimality of all programs in \( P_t \). In turn, the optimality of each program \( p \) is a weighted combination of three simpler heuristics (according to its importance):

  \[
  \text{Opt}(p) = \beta_1 \cdot \text{CRD}(p) - \beta_2 \cdot \text{OU}(p) + \beta_3 \cdot \text{RP}(p)
  \]

  where \( \text{CRD} \) (coverage rate difference) is a normalised measure of coverage difference between positive examples and negative examples covered, \( \text{OU} \) (operator usage) is a normalised measure of how many operators have been used to derive the rule and \( \text{RP} \) is a measure of the expressiveness of the programs in terms of number of variables, functions and constants. As for the current implementation of gErl, the weights are \( \beta_1 = 1 \), \( \beta_2 = 0.2 \) and \( \beta_3 = 0.1 \). Finally, the **Global optimality** factor is then calculated as the average of the optimality of all programs in the system:

  \[
  \text{OptGlobal}(P_t) = \frac{1}{\text{Card}(P_t)} \sum_{p \in P_t} \text{Opt}(p)
  \]

- **Actions.** An action is a tuple \( (o, \rho) \) with \( o \in \mathcal{O} \) and \( \rho \in \mathcal{R} \), that represents the operator \( o \) to be applied to the rule \( \rho \). Each rule is described by a tuple of features \( \rho = (\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7, \varphi_8) \), that represents some relevant information of rule \( \rho \): the size \( (\varphi_1) \), the positive coverage rate \( (\varphi_2) \), the negative coverage rate \( (\varphi_3) \), the number of variables of \( \rho \) \( (\varphi_4) \), the number of constants (functors with arity 0) of \( \rho \) \( (\varphi_5) \), the number of functors with arity greater than 0 of \( \rho \) \( (\varphi_6) \), the number of structures (lists, graphs, ...) of \( \rho \) \( (\varphi_7) \) and, finally, \( \varphi_8 \) indicates if the rule \( \rho \) is recursive or not.

We use a natural index as the only feature for operators. As an action consists of a choice of operator and rule, an action is finally a tuple of **nine** features.
– **Transitions.** Transitions are deterministic. A transition $\tau$ evolves the current sets of rules and programs by applying the operators selected (together with the rule) and the combiners.

– **Rewards.** The optimality criteria seen above is used to feed the rewards. In particular, we use the result returned by equation (1) as reward.

At each point in time, the reinforcement learning policy can be in one of the states $s_t \in S$ and may select an action $a_t = \pi(s_t) \in A$ to execute. Executing such action $a_t$ in $s_t$ will change the state into $s_{t+1} = \tau(s_t, a_t)$, and the policy receives a reward $w_t = \omega(s_t, a_t)$. But as the policy does not know $\tau$ and $\omega$, they need to be learned. This is the typical formulation of RL [9] but using features to represent the states and the actions. Therefore, the aim of our decision process is to find a policy $\pi : S \rightarrow A$ that maximises:

$$V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i w_{t+i}$$

for all $s_t$, where $\gamma \in [0, 1]$ is the discount parameter which determines the importance of the future rewards ($\gamma = 0$ only considers current rewards, while $\gamma = 1$ strives for a high long-term reward).

**Modelling the state-value function by using a regression model** For the RL module in gErl, we use a hybrid between model-free value-function methods (which update a state-value matrix) and model-based methods (which learn models for $\tau$ and $\omega$) [9]. In particular, our approach uses the state-value function ($Q(s, a)$, which returns quality ($q$) values, $q \in \mathbb{R}$, as in Q-learning [12]) which is generalised by a regression model $M : S \times A \rightarrow \mathbb{R}$ that calculates the optimality or $q$ value for each state and action. By using $a_t = \arg\max_{a \in A} \{M(s_t, a_t)\}$ we get the best action for state $s_t$.

In order to train the model we use a `matrix’ $Q$ (which is actually a table), whose rows are in $S \times A \times \mathbb{R}$ where $S$ is a tuple of state features, $A$ is the tuple of actions, and $\mathbb{R}$ is a real value for $q$. So, each row in $Q$ records the state and action for each time step in the system and its q-value. Table 1 shows an example of $Q$. Abusing notation in order to work with $Q$ as a function (like the original $Q$-matrix in many RL method), we will denote by $Q[s, a]$ the value of $q$ in the row of $Q$ that corresponds to state $s$ and action $a$.

<table>
<thead>
<tr>
<th>state (s)</th>
<th>action (a)</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi_1$</td>
<td>$\phi_2$</td>
<td>$\phi_3$</td>
</tr>
<tr>
<td>1.223</td>
<td>1.473</td>
<td>0.431</td>
</tr>
<tr>
<td>1.301</td>
<td>1.51</td>
<td>0.255</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Example of a $Q$ matrix (represented as a table).

A supervised model $M$ (using $q$ as the output) is retrained periodically from table $Q$ (gErl uses linear regression by default). Once the system has started, at each step, $Q$ is updated using the following formula, as in Q-learning:

$$Q[s_t, a_t] \leftarrow \alpha [w_{t+1} + \gamma \max_{a_{t+1}} M(s_{t+1}, a_{t+1})] + (1 - \alpha)Q[s_t, a_t]$$

(4)
where the maximum future value is obtained by the model, \( \gamma \in [0,1] \) is the
discount parameter and \( \alpha \in [0,1] \) is the \textit{learning rate} which determines to what
extent the newly acquired information will override the old information (\( \alpha = 0 \)
makes the agent not to propagate anything, while \( \alpha = 1 \) makes the agent consider
only the most recent information). By default, \( \alpha = 0.5 \) and \( \gamma = 0.5 \).

4 Reloading the \( Q \)-value function

In this section, we describe how previous experience can be applied from previous
executions. We can use the \( Q \) “matrix” learnt from previous data because of the
abstract representation of states and actions (\( \phi \) and \( \varphi \) parameters) which allows
the system to reuse the previous information, namely, actions (application of
operators to rules) which have been successfully applied at certain states (from
the previous task) when it reaches a similar new state (in terms of features).

In our system, when it reaches the solution of a given problem (or it executes
a maximum number of steps), \( Q \) and the model \( M \) can be viewed as knowledge
acquired that can be transferred to the new situation. This acquired knowledge,
in the form of \( Q \), is passed to the new problem to be solved. When \( \text{gErl} \) starts
learning the new task, \( Q \) is used to train a new model which is used from the
beginning (exploiting it) and it is updated with the new information acquired
for each time step (in order to retrain the new model \( M \)).

Furthermore, our system also allows consider breaking down a task into a
series of smaller task in order to be easier for the system to learn in isolation
than in the context of the full task. For instance, many problems can consist
of a large evidence, which cannot fit in memory or need lot of computational
resources. This approach can also be considered a type of transfer in that a
single large target task can be incremental treated as a a series of simpler source
tasks, where, if the hypothesis founded in a simple source task is stable (covers
part or all positive evidence), the time required to solve the large task will be
substantially reduced if the system reuse the knowledge previously acquired.

5 Empirical study

In this section, we describe the experiment performed to demonstrate the useful-
ness of the feature-based abstraction used in the \( \text{gErl} \) system and how a problem
may be broken down intp a series of smaller tasks in order to accelerate the learn-
ing, as in incremental learning. We also show more details about our system and
how it solves these problems\(^1\).

Figure 2 shows part of an extremely simplified version of the English language
grammar which is used in our study. The problem consists of 40 positive examples
\( E^+ \) and 63 negative examples \( E^- \). It is necessary to use a large background
knowledge \( B \) in order to deal with every kind of word. The target model \( H \) is
composed by 5 general target rules (see Figure 2, known as ‘hypotheses’) which

\(^1\) \( \text{gErl} \) is available at \texttt{http://goo.gl/VrmPw}. 
Positive Examples E+:
- e1: s([an, unknown, alien, hits, the, house]).
- e2: s([a, small, boy, walks, a, dog]).
- e3: s([a, dog, walks, into, the, house]).

Negative Examples E−:
- e4: ¬s([dog, hits, a, boy]).
- e5: ¬s([the, house, a, boy]).
- e6: ¬s([dog, walks, the, house]).

Hypothesis H:
- h1: s([an, unknown, alien, hits, the, house]).
- h2: s([a, small, boy, walks, a, dog]).

Background Knowledge B:
- b1: det(S1), noun(S2), noun(S3).
- b2: adj(S4).
- b3: verbPhrase(S1).
- b4: verbPhrase(S2).

Fig. 2: Grammar Learning Example

Cover all the positive evidence (each $h_i$ ($1 \leq i \leq 5$) covers the same number of positive examples).

Since the examples are represented as lists of words, we need to define appropriate operators to navigate this structure and apply local or global changes to it. The first two operators, nounPhrase2 and nounPhrase3, take the input list and generalise its 2 or 3 first components (respectively) by replacing them by variable symbols. Also the noun phrase syntax structure constructed with these generalised components is added to the rhs of the rule. The definition of these operators is written in Erlang, but it can be informally defined using an example as follows: nounPhrase2(s([a, small, boy, walks, a, dog])) \Rightarrow s([S1, S2, S3, walks, a, dog]) \Rightarrow s([S1, S2, S3, S4, a, dog]) \Rightarrow nSp(S1, S2, S3, S4, a, dog))

The third and fourth operators, verbPhrase1 and verbPhrase2, work in a similar way to the first two operators but, in this case, are used to construct verb phrase syntax structures. In order to see how they work, the operators can be informally defined using an example as follows: verbPhrase1(s([S1, S2, S3, walks, a, dog]) \Rightarrow nSp(S1, S2, S3, a, dog)) \Rightarrow s([S1, S2, S3, S4, a, dog]) \Rightarrow nSp(S1, S2, S3, S4, a, dog))

As before, only one operator generates a useful rule for the input string given, in this case, the third operator.

With these four operators the gErl system is able to solve the previous problem and to find the correct order of application of the operators to obtain each one of the five target rules (hypotheses) in Figure 2. For instance, the sequence for the instance $s([an, unknown, alien, hits, the, house])$ is:
This latter equation $s([S_1, S_2, S_3, S_4, S_5, S_6] \rightarrow np(S_1, S_2, S_3), vp(S_4), np(S_5, S_6)$ (which corresponds with the hypothesis $h_5$) is the solution for this example.

In order to analyse the ability of the system to improve the learning when reusing past policies, we performed two experiments. The first experiment consists in taking a random sample of 20 examples from the original problem but keeping the same hypothesis (target rule) distribution of the original problem (stratified), and learning from scratch. After the learning process is learnt we use the $Q$ “matrix” fort the rest of the 20 positive instances not previously selected. This configuration is denoted by $20 + M_{past}$. We compare this incremental situation to two baseline situations (just learning from 20 instances and learning from 40 instance, respectively denoted by 20 and 40). We repeat each configuration five times and average the results. Figure 3 (top) shows the results. A clear improvement is observed when we reuse knowledge (“$20 + M_{past}$” sample) in contrast to when we do not (“20” sample). The results also show how the learning process has been accelerated by the use a smaller sample (“20” sample) compared to the original problem (“40” sample). The second experiment is similar to the first one but we do not take into account the distribution of examples for each hypothesis (not stratified), somewhat simulating a ‘concept drift’. As in the first experiment, we use the $Q$ “matrix” which has been previously learnt to bias the learning of the rest of 20 positive instances not previously selected. We also performed five repetitions. Figure 3 (bottom) shows the results which show that, as in the previous experiment, there exists a clear improvement when we use previous acquired knowledge to solve future related tasks.
6 Conclusions and future work

One of the problems for reusing the knowledge between different or the same problems is how to represent this knowledge in such a way that it is sufficiently general to work with even when there is a concept drift or some operators may change. In this paper we have presented the main features of a first prototype of the gErl system, especially the way previous learning sessions can be used across different problems or in an incremental way for the same problem (as has been shown here). This is based on a representation of heuristic knowledge as a tuple of abstract features. We have used our system to solve a simple problem incrementally, showing that the total learning time is reduced when the problem is split into smaller bits. This is just a promising result which encourage these ideas, although we need to perform more experiments on other domains and keep on refining our system. In the end, gErl aims at working with structured prediction, configurable operators, recursion and RL-based heuristics, a combination which is as powerful as challenging.

References