Learning Agents: Introduction

Liliana Mamani Sanchez
lmamanis@tcd.ie
November 2, 2015

Learning in agent architectures

Machine Learning for Games

- Reasons to use Machine Learning for Games:
  - Play against, and beat human players (as in board games, DeepBlue etc)
  - Minimise development effort (when developing AI components); avoid the knowledge engineering bottleneck
  - Improve the user experience by adding variability, realism, a sense that artificial characters evolve, etc.

Some questions

- What is (Machine) Learning?
- What can Machine Learning really do for us?
- What kinds of techniques are there?
- How do we design machine learning systems?
- What’s different about reinforcement learning?
• Could you give us some examples?
  – YES:
  – *Draughts* (checkers)
  – *Noughts & crosses* (tic-tac-toe)

**Defining “learning”**

• ML has been studied from various perspectives (AI, control theory, statistics, information theory, ...)

• From an AI perspective, the general definition is formulated in terms of agents and tasks. E.g.:

  [An agent] is said to learn from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with $E$.

  *(Mitchell, 1997, p. 2)*

• Statistics, model-fitting, ...

**Some examples**

• Problems too difficult to program by hand

---

**Data Mining**

<table>
<thead>
<tr>
<th>Name:</th>
<th>Corners</th>
<th>Name:</th>
<th>Corners</th>
<th>Name:</th>
<th>Corners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bearing:</td>
<td>100</td>
<td>Bearing:</td>
<td>40</td>
<td>Bearing:</td>
<td>20</td>
</tr>
<tr>
<td>Velocity:</td>
<td>20</td>
<td>Velocity:</td>
<td>20</td>
<td>Velocity:</td>
<td>20</td>
</tr>
<tr>
<td>Energy:</td>
<td>30</td>
<td>Energy:</td>
<td>20</td>
<td>Energy:</td>
<td>20</td>
</tr>
<tr>
<td>Heading:</td>
<td>90</td>
<td>Heading:</td>
<td>90</td>
<td>Heading:</td>
<td>90</td>
</tr>
</tbody>
</table>
| ... | ... | ... | ... | ... | ...

---

| (t0) | (t1) | (t2) | time |
if Name = Corners & Energy < 25
then
  turn(91 - (Bearing - const))
  fire(3)

User interface agents

• Recommendation services,
• Bayes spam filtering
• JIT information retrieval

Designing a machine learning system

• Main design decisions:
  – Training experience: How will the system access and use data?
  – Target function: What exactly should be learned?
  – Hypothesis representation: How will we represent the concepts to be learnt?
  – Inductive inference: What specific algorithm should be used to learn the target concepts?

Types of machine learning

• How will the system be exposed to its training experience?
  – Direct or indirect access:
    * indirect access: record of past experiences, databases, corpora
    * direct access: situated agents → reinforcement learning
  – Source of feedback (“teacher”):
    * supervised learning
    * unsupervised learning
    * mixed: semi-supervised (“transductive”), active learning, ....
The hypothesis space

- The data used in the induction process need to be represented uniformly. E.g.: 
  - representation of the opponent’s behaviour as feature vectors
- The choice of representation constrains the space of available hypotheses (inductive bias).
- Examples of inductive bias:
  - assume that positive and negative instances can be separated by a (hyper) plane
  - assume that feature co-occurrence does not matter (conditional independence assumption by Naïve Bayes classifiers)
  - assume that the current state of the environment summarises environment history (Markov property)

Determining the target function

- The goal of the learning algorithm is to induce an approximation $\hat{f}$ of a target function $f$
- In supervised learning, the target function is assumed to be specified through annotation of training data or some form of feedback.
- Examples:
  - a collection of texts categorised by subject $f : D \times S \rightarrow \{0, 1\}$
  - a database of past games
  - user or expert feedback
- In reinforcement learning the agent will learn an action selection policy (as in action : $S \rightarrow A$)

Deduction and Induction

- Deduction: from general premises to a conclusion. E.g.: 
  - $\{A \rightarrow B, A\} \vdash B$
- Induction: from instances to generalisations
- Machine learning algorithms produce models that generalise from instances presented to the algorithm
- But all (useful) learners have some form of inductive bias:
  - In terms of representation, as mentioned above,
  - But also in terms of their preferences in generalisation procedures. E.g:
    - prefer simpler hypotheses, or
Given a function \( \hat{f} : X \to C \) trained on a set of instances \( D_c \) describing a concept \( c \), we say that the inductive bias of \( \hat{f} \) is a minimal set of assertions \( B \), such that for any set of instanced \( X \):

\[
\forall x \in X (B \land D_c \land x \vdash \hat{f}(x))
\]

This should not be confused with estimation bias, which is a quantity (rather than a set of assumptions) which quantifies the average “loss” due to misclassification. Together with variance, this quantity determines the level of generalisation error of a learner (Domingos, 2012).

**Choosing an algorithm**

- Induction task as search for a hypothesis (or model) that fits the data and sample of the target function available to the learner, in a large space of hypotheses.
- The choice of learning algorithm is conditioned to the choice of representation.
- Since the target function is not completely accessible to the learner, the algorithm needs to operate under the inductive learning assumption that:
  
an approximation that performs well over a sufficiently large set of instances will perform well on unseen data.
- *Computational Learning Theory* addresses this question.

**Two Games: examples of learning**

- **Supervised** learning: draughts/checkers
  
  [Mitchell, 1997]

  \[
  \begin{array}{|c|c|c|}
  \hline
  X & O & O \\
  \hline
  O & X & X \\
  \hline
  \end{array}
  \]

- **Reinforcement** learning: noughts and crosses
  
  [Sutton and Barto, 1998]

  \[
  \begin{array}{|c|c|c|}
  \hline
  X & O & O \\
  \hline
  O & X & X \\
  \hline
  \end{array}
  \]

- Task? (target function, data representation) Training experience? Performance measure?
A target for a draughts learner

- Learn.... \( f : \text{Board} \rightarrow \text{Action} \) or \( f : \text{Board} \rightarrow \mathbb{R} \)
- But how do we label (evaluate) the training experience?
- Ask an expert?
- Derive values from a rational strategy:
  - if \( b \) is a final board state that is won, then \( f(b) = 100 \)
  - if \( b \) is a final board state that is lost, then \( f(b) = -100 \)
  - if \( b \) is a final board state that is drawn, then \( f(b) = 0 \)
  - if \( b \) is a not a final state in the game, then \( f(b) = f(b') \), where \( b' \) is the best final board state that can be achieved starting from \( b \) and playing optimally until the end of the game.
- How feasible would it be to implement these strategies?
  - Hmmmm... Not feasible...

Hypotheses and Representation

- The choice of representation (e.g. logical formulae, decision tree, neural net architecture) constrains the hypothesis search space.
- A representation scheme: linear combination of board features:
  \[
  \hat{f}(b) = w_0 + w_1 \cdot bp(b) + w_2 \cdot rp(b) + w_3 \cdot bk(b) + w_4 \cdot rk(b) + w_5 \cdot bt(b) + w_6 \cdot rt(b)
  \]
  where:
  - \( bp(b) \): number of black pieces on board \( b \)
  - \( rp(b) \): number of red pieces on \( b \)
  - \( bk(b) \): number of black kings on \( b \)
  - \( rk(b) \): number of red kings on \( b \)
  - \( bt(b) \): number of red pieces threatened by black
  - \( rt(b) \): number of black pieces threatened by red

Training Experience

- Some notation and distinctions to keep in mind:
  - \( f(b) \): the true target function
  - \( \hat{f}(b) \): the learnt function
  - \( f_{\text{train}}(b) \): the training value (obtained, for instance, from a training set containing instances and its corresponding training values)
- Problem: How do we obtain training values?
• A simple rule for obtaining (estimating) training values:

\[ f_{\text{train}}(b) \leftarrow \hat{f}(\text{Successor}(b)) \]

Remember that what the agent really needs is an action policy, \( \pi : B \rightarrow A \), to enable it to choose an action \( a \) given a board configuration \( b \). We are assuming that the agent implements a policy \( \pi(s) \) which selects the action that maximises the value of the successor state of \( b \), \( f(b') \).

How do we learn the weights?

Algorithm 1: Least Means Square

```
1 LMS(\( c \): learning rate)
2   for each training instance <\( b \), \( f_{\text{train}}(b) \)>
3       do
4           compute error\( (b) \) for current approximation
5               (i.e. using current weights):
6                 error\( (b) = f_{\text{train}}(b) - \hat{f}(b) \)
7                 for each board feature \( t_i \in \{1p(b), rp(b), \ldots\} \),
8                     do
9                       update weight \( w_i \):
10                         \( w_i \leftarrow w_i + c \times t_i \times \text{error}(b) \)
11                     done
12       done
```

LMS minimises the squared error between training data and current approx.: \( E \equiv \sum_{(b,f_{\text{train}}(b)) \in D} (f_{\text{train}}(b) - \hat{f}(b))^2 \) Notice that if \( \text{error}(b) = 0 \) (i.e. target and approximation match) no weights change. Similarly, if or \( t_i = 0 \) (i.e. feature \( t_i \) doesn’t occur) the corresponding weight doesn’t get updated. This weight update rule can be shown to perform a gradient descent search for the minimal squared error (i.e. weight updates are proportional to \( -\nabla E \) where \( \nabla E = [\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots] \)).

That the LMS weight update rule implements gradient descent can be seen by differentiating \( \nabla E \):

\[
\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \sum_{(b,f_{\text{train}}(b)) \in D} [f(b) - \hat{f}(b)]^2 \\
= \sum_{(b,f_{\text{train}}(b)) \in D} \frac{\partial}{\partial w_i} [f(b) - \hat{f}(b)]^2 \\
= \sum_{(b,f_{\text{train}}(b)) \in D} 2 \times [f(b) - \hat{f}(b)] \times \frac{\partial}{\partial w_i} [f(b) - \hat{f}(b)] \\
= \sum_{(b,f_{\text{train}}(b)) \in D} 2 \times [f(b) - \hat{f}(b)] \times \frac{\partial}{\partial w_i} [f(b) - \sum_j w_j t_j] \\
= -\sum_{(b,f_{\text{train}}(b)) \in D} 2 \times \text{error}(b) \times t_j
\]

Design choices: summary
Determine Target Function
Determine Representation of Learned Function
Determine Type of Training Experience
Determine Learning Algorithm

Games against self
Games against experts
Table of correct moves
Linear function of six features
Artificial neural network
Polynomial
Gradient descent
Board value
Board move

Completed Design

(from Mitchell, 1997) These are some of the decisions involved in ML design. A number of other practical factors, such as evaluation, avoidance of “overfitting”, feature engineering, etc. See (Domingos, 2012) for a useful introduction, and some machine learning “folk wisdom”.

The Architecture instantiated

Reinforcement Learning
• What is different about reinforcement learning:
  – Training experience (data) obtained through direct interaction with the environment;
  – Influencing the environment;
  – Goal-driven learning;
  – Learning of an action policy (as a first-class concept)
  – Trial and error approach to search:
    * Exploration and Exploitation
Basic concepts of Reinforcement Learning

• The policy: defines the learning agent’s way of behaving at a given time:
  \[ \pi : S \rightarrow A \]

• The (immediate) reward function: defines the goal in a reinforcement learning problem:
  \[ r : S \rightarrow \mathbb{R} \]
  often identified with timesteps: \( r_0, \ldots, r_n \in \mathbb{R} \)

• The (long term) value function: the total amount of reward an agent can expect to accumulate over the future:
  \[ V : S \rightarrow \mathbb{R} \]

• A model of the environment

Theoretical background

• Engineering: “optimal control” (dating back to the 50’s)
  – Markov Decision Processes (MDPs)
  – Dynamic programming

• Psychology: learning by trial and error, animal learning. Law of effect:
  – learning is selectional (genetic methods, for instance, are selectional, but not associative) and
  – associative (supervised learning is associative, but not selectional)

• AI: TD learning, Q-learning

  Law of effect: “Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.” (Thorndike, 1911, p. 244, quoted by (Sutton and Barto, 1998))

  The selectional aspect means that the learner will select from a large pool of complete policies, overlooking individual states (e.g. the contribution of a particular move to winning the game).

  The associative aspect means that the learner associates action (move) to a value but does not select (or compare) policies as a whole.
Example: Noughts and crosses

Possible solutions: minimax (assume a perfect opponent), supervised learning (directly search the space of policies, as in the previous example), reinforcement learning (our next example).

A Reinforcement Learning strategy

- Assign values to each possible game state (e.g. the probability of winning from that state):

<table>
<thead>
<tr>
<th>state</th>
<th>( V(s) )</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 = \begin{array}{c} \text{x} \ \text{x} \ \text{o} \end{array} )</td>
<td>0.5</td>
<td>??</td>
</tr>
<tr>
<td>( s_1 = \begin{array}{c} \text{x} \ \text{x} \ \text{x} \end{array} )</td>
<td>0.5</td>
<td>??</td>
</tr>
<tr>
<td>( \vdots )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_i = \begin{array}{c} \text{x} \ \text{o} \ \text{o} \end{array} )</td>
<td>0</td>
<td>loss</td>
</tr>
<tr>
<td>( \vdots )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_n = \begin{array}{c} \text{x} \ \text{x} \ \text{x} \end{array} )</td>
<td>1</td>
<td>win</td>
</tr>
</tbody>
</table>

Algorithm 2: TD Learning

While learning

select move by looking ahead 1 state
choose next state \( s \):

\[
\begin{array}{l}
\text{if } \text{exploring} \\
\text{pick } s \text{ at random} \\
\text{else} \\
\quad s = \text{arg max}_s V(s)
\end{array}
\]

N.B.: exploring could mean, for instance, pick a random next state 10% of the time.

How to update state values
Some nice properties of this RL algorithm

- For a fixed opponent, if the parameter that controls learning rate ($\alpha$) is reduced properly over time, converges to the true probabilities of winning from each state (yielding an optimal policy)

- If $\alpha$ isn’t allowed to reach zero, the system will play well against opponents that alter their game (slowly)

- Takes into account what happens during the game (unlike supervised approaches)

What was not illustrated

- RL also applies to situations where there isn’t a clearly defined adversary (“games against nature”)

- RL also applies to non-episodic problems (i.e. rewards can be received at any time not only at the end of an episode such as a finished game)

- RL scales up well to games where the search space is (unlike our example) truly vast.
  - See (Tesauro, 1994), for instance.

- Prior knowledge can also be incorporated

- Look-ahead isn’t always required
References


