Daniel's Notebook

Ideas → Text

Training a POMDP (with Python)

with 11 comments

Working on my Bachelor Thesis[5], I noticed that several authors have trained a Partially Observable Markov Decision Process (POMDP) using a variant of the Baum-Welch Procedure (for example McCallum [4][3]) but no one actually gave a detailed description *how* to do it. In this post I will highlight some of the difficulties and present a possible solution based on an idea proposed by Devijver [2]. In the first part I will briefly present the Baum-Welch Algorithm and define POMDPs in general. Subsequently, a version of the alpha-beta algorithm tailored to POMDPs will be presented from which we can derive the update rule. Finally I will present a sample implementation in Python. If you are not interested in the theory, just skip over to the last part.

The Baum-Welch Procedure

In its original formulation, the Baum-Welch procedure[1][6] is a special case of the EM-Algorithm that can be used to optimise the parameters of a Hidden Markov Model (HMM) against a data set. The data consists of a sequence of observed inputs to the decision process and a corresponding sequence of outputs. The goal is to maximise the likelihood of the observed sequences under the POMDP. The procedure is based on Baum's forward-backward algorithm (a.k.a. alpha-beta algorithm) for HMM parameter estimation.

The application that I had in mind required two modifications:

- The original forward-backward algorithm suffers from numerical instabilities. Devijver proposed a equivalent reformulation which works around these instabilities [2].
- Both the Baum-Welch procedure and Devijver's version of the forward-backward algorithm are designed for HMMs, not for POMDPs.

POMDPs

In this section I will introduce some notation.

Let the POMDP be defined by $(\mathbb{A}, C, n_a, n_o, p)$.

- $\mathbb{A}$ is the state transition matrix. It defines the probability of transferring to state $s_{\tau+1}$ given that the current state is s_τ .
- C is the output matrix. It defines the probability of observing output x_τ given that the current state is s_τ .
- n_a is the number of states (or, equivalently: the dimensionality of the latent variable).
- n_a is the number of outputs (the dimensionality of the observed variable)
- p is the initial state distribution.

This may not be the standard way to define POMDPs. However, this vectorized notation has several advantages:

1. It allows us to express state transitions very neatly. If s_τ is a vector expressing our belief in which state we are in time

step τ, then (without observing any output) $s_{\tau+1} = As_\tau$ should express our beliefs in time step $\tau+1$
2. We can use it in a similar way to deal with output probabilities.
3. It is (in my opinion) much nicer to implement an actual algorithm using the toolbox of vector and matrix operations.

Applying Devijver's Technique to POMDPs

This section will demonstrate, how Devijver's forward-backward algorithm can be restated for the case of POMDPs

The aim of the POMDP forward-backward procedure is to find an estimator for $P\left(S(\tau)=s \mid [Y]_1^T=[y]_1^T, [X]_1^T=[x]_1^T\right)$. Using Bayes' Rule

$$
\begin{aligned}
&P\left(S(\tau)=s \mid [Y]_1^T=[y]_1^T, [X]_1^T=[x]_1^T\right)\\
=&\frac{P\left(S(\tau)=s, [Y]_1^T=[y]_1^T \mid [X]_1^T=[x]_1^T\right)}{P\left([Y]_1^T=[y]_1^T \mid [X]_1^T=[x]_1^T\right)}\\
=&\frac{P\left(S(\tau)=s, [Y]_1^\tau=[y]_1^\tau, [Y]_{\tau+1}^T=[y]_{\tau+1}^T \mid [X]_1^T=[x]_1^T\right)}{P\left([Y]_1^\tau=[y]_1^\tau, [Y]_{\tau+1}^T=[y]_{\tau+1}^T \mid [X]_1^T=[x]_1^T\right)}\\
=&\frac{P\left(S(\tau)=s, [Y]_1^\tau=[y]_1^\tau \mid [X]_1^T=[x]_1^T\right)}{P\left([Y]_1^\tau=[y]_1^\tau \mid [X]_1^T=[x]_1^T\right)}\frac{P\left([Y]_{\tau+1}^T=[y]_{\tau+1}^T \mid S(\tau)=s, [X]_1^T=[x]_1^T\right)}{P\left([Y]_{\tau+1}^T=[y]_{\tau+1}^T \mid [Y]_1^\tau=[y]_1^\tau, [X]_1^T=[x]_1^T\right)}\\
=&\alpha(\tau,s)\cdot\beta(\tau,s)
\end{aligned}
$$

The problem reduces thus to finding α and β which shall be called the forward estimate and the backward estimate respectively.

$$
\mathcal{N}(\tau)=\frac{1}{P\left(Y(\tau)=y_\tau \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)}
$$

Repeated application of Bayes' Rule and the definition of the POMDP leads to the following recursive formulation of α:

$$
\begin{aligned}
&\alpha(\tau,s)=\frac{P\left(S(\tau)=s, [Y]_1^\tau=[y]_1^\tau \mid [X]_1^T=[x]_1^T\right)}{P\left([Y]_1^\tau=[y]_1^\tau \mid [X]_1^T=[x]_1^T\right)}\\
=&P\left(S(\tau)=s \mid Y(\tau)=y_\tau, [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)\\
=&\frac{P\left(S(\tau)=s, Y(\tau)=y_\tau \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)}{P\left(Y(\tau)=y_\tau \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)}\\
=&\mathcal{N}(\tau)P\left(S(\tau)=s, Y(\tau)=y_\tau \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)\\
=&\mathcal{N}(\tau)\sum_\sigma P\left(S(\tau)=s \mid S(\tau-1)=\sigma, [X]_1^T=[x]_1^T\right)P\left(Y(\tau)=y_\tau \mid S(\tau)=s, [X]_1^T=[x]_1^T\right)P\left(S(\tau-1)=\sigma \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)\\
=&\mathcal{N}(\tau)\sum_\sigma P\left(S(\tau)=s \mid S(\tau-1)=\sigma, X(\tau)=x_\tau\right)P\left(Y(\tau)=y_\tau \mid S(\tau)=s\right)P\left(S(\tau-1)=\sigma \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)\\
=&\mathcal{N}(\tau)\sum_\sigma \alpha(\tau-1,\sigma)A_{(s,\sigma)}^{(x_{\tau-1})}C_{(y_\tau,s)}
\end{aligned}
$$

This yields a recursive policy for calculating α. The missing piece, $\mathcal{N}(\tau)$, can be calculated using the preceding recursion step for α:

$$
\begin{aligned}
&\frac{1}{\mathcal{N}(\tau)}=P\left(Y(\tau)=y_\tau \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)\\
=&\sum_s P\left(S(\tau)=s, Y(\tau)=y_\tau \mid [Y]_1^{\tau-1}=[y]_1^{\tau-1}, [X]_1^T=[x]_1^T\right)\\
=&\sum_s\sum_\sigma \alpha(\tau-1,\sigma)A_{(s,\sigma)}^{(x_{\tau-1})}C_{(y_\tau,s)}
\end{aligned}
$$

The common term of the α-recursion and the $\mathcal{N}$-recursion can be extracted to make the process computationally more efficient:

$$
\begin{aligned}
\alpha(\tau,s)&=\mathcal{N}(\tau)\gamma(\tau,s)\\
\mathcal{N}(\tau)&=\frac{1}{\sum_\sigma \gamma(\tau,\sigma)}\\
\gamma(\tau,s)&=C_{(y_\tau,s)}\sum_\sigma \alpha(\tau-1,\sigma)A_{(x,\sigma)}^{(x_{\tau-1})}
\end{aligned}
$$

The result differs from the original formulation [2] merely by the fact that the appropriate transition matrix is chosen in every recursion step. It is still necessary to calculate β, which can be reduced to a similar recursion:

$$\begin{aligned}\beta(\tau, s) &= \frac{P\left([Y]_{\tau+1}^{T} = [y]_{\tau+1}^{T} \mid S(\tau) = s, [X]_1^T = [x]_1^T\right)}{P\left([Y]_{\tau+1}^{T} = [y]_{\tau+1}^{T} \mid [Y]_1^{\tau} = [y]_1^{\tau}, [X]_1^T = [x]_1^T\right)} \\ &= \mathcal{N}(\tau+1) \sum_{\sigma} A_{(\sigma,s)}^{\langle x_\tau \rangle} C_{(y_{\tau+1},\sigma)} \beta(\tau+1, \sigma)\end{aligned}$$

The base cases of these recursions follow directly from their probabilistic interpretation:

$$\begin{aligned}\gamma(1, s) &= p_s \cdot C_{(y_1,s)} \\ \beta(T, s) &= 1\end{aligned}$$

Using the definitions of α and β it is now possible to derive an unbiased estimator for $P\left(S(\tau) = s \mid [Y]_1^T = [y]_1^T, [X]_1^T = [x]_1^T\right)$. It's definition bears a striking resemblance to the estimator derived by Devijver [2]. Analogue to the steps Baum and Welch took to derive their update rule, the need for two more probabilities arises: $P\left(S(\tau) = s, S(\tau+1) = \sigma \mid [Y]_1^T = [y]_1^T, [X]_1^T = [x]_1^T\right)$ and $P\left(S(\tau) = s, Y(\tau) = y \mid [Y]_1^T = [y]_1^T, [X]_1^T = [x]_1^T\right)$. Luckily, α, β and $\mathcal{N}$ again can be used to compute these probabilities.

$$\begin{aligned}&P\left(S(\tau) = s, S(\tau+1) = \sigma \mid [Y]_1^T = [y]_1^T, [X]_1^T = [x]_1^T\right) \\ &= \frac{P\left(S(\tau) = s, [Y]_1^{\tau} = [y]_1^{\tau} \mid [X]_1^T = [x]_1^T\right)}{P\left([Y]_1^{\tau} = [y]_1^{\tau} \mid [X]_1^T = [x]_1^T\right)} \frac{P\left(S(\tau+1) = \sigma \mid S(\tau) = s, X(\tau) = x_\tau\right)}{P\left(Y(\tau+1) = y_{\tau+1} \mid [Y]_1^{\tau} = [y]_1^{\tau}, [X]_1^T = [x]_1^T\right)} \\ &\cdot P(Y(\tau+1) = y_{\tau+1} \mid S(\tau+1) = \sigma) \frac{P\left([Y]_{\tau+2}^{T} = [y]_{\tau+2}^{T} \mid S(\tau+1) = \sigma, [X]_1^T = [x]_1^T\right)}{P\left([Y]_{\tau+2}^{T} = [y]_{\tau+2}^{T} \mid [Y]_1^{\tau+1} = [y]_1^{\tau+1}, [X]_1^T = [x]_1^T\right)} \\ &= \alpha(\tau, s) \beta(\tau+1, \sigma) A_{(\sigma,s)}^{\langle x_\tau \rangle} C_{(y_{\tau+1},\sigma)} \mathcal{N}(\tau+1)\end{aligned}$$

It turns out, that the values for $\mathcal{N}$ calculated above can also be used to calculate the likelihood of the observed data under the current model parameters.

$$P\left([Y]_1^T = [y]_1^T \mid [X]_1^T = [x]_1^T\right) = \prod_{\tau=1}^{T} \frac{1}{\mathcal{N}(\tau)}$$

In practice the log-likelihood will be of more interest, as it's calculation is more efficient and numerically more stable:

$$\log\left(\prod_{\tau=1}^{T} \frac{1}{\mathcal{N}(\tau)}\right) = -\sum_{\tau=1}^{T} \log(\mathcal{N}(\tau))$$

EM-Update Rule

The methods derived up to this point are useful to calculate state probabilities, transition probabilities and output probabilities given a model and a sequence of inputs and outputs. As Baum and Welch did in the case of HMMs, these very probabilities will now be used to derive estimators for the model's parameters. Let $\mathbb{T}_x = \{\tau \mid x_\tau = x\}$. Then

$$\begin{aligned}&\sum_{\tau \in \mathbb{T}_x} \frac{P\left(S(\tau) = s \mid [X]_1^T = [x]_1^T, [Y]_1^T = [y]_1^T\right)}{|\mathbb{T}_x|} \\ &\approx P(S(\tau) = s \mid X(\tau) = x)\end{aligned}$$

A similar scheme can be used to derive an estimator for the transition probabilities:

$$\begin{aligned}&\sum_{\tau \in \mathbb{T}_x} \frac{P\left(S(\tau) = s, S(\tau+1) = \sigma \mid [X]_1^T = [x]_1^T, [Y]_1^T = [y]_1^T\right)}{|\mathbb{T}_x|} \\ &\approx P(S(\tau) = s, S(\tau+1) = \sigma \mid X(\tau) = x) \\ &= P(S(\tau+1) = \sigma \mid S(\tau) = s, X(\tau) = x) \, P(S(\tau) = s \mid X(\tau) = x)\end{aligned}$$

where $P(S(\tau) = s \mid X(\tau) = x)$ can be approximated using the above estimator (this might mean that the new estimator is biased, not quite sure about that). The result is an estimator for $P(S(\tau+1) = \sigma \mid S(\tau) = s, X(\tau) = x)$. An estimator for the output probabilities can be derived accordingly, making use of the Markov property:

$$\sum_{\tau \in \mathbb{T}_x} \frac{P\left(S(\tau)=s, Y(\tau)=y \mid [X]_1^T=[x]_1^T, [Y]_1^T=[y]_1^T\right)}{|\mathbb{T}_x|}$$
$$\approx P(S(\tau)=s, Y(\tau)=y \mid X(\tau)=x)$$
$$= P(Y(\tau)=y \mid S(\tau)=s, X(\tau)=x)\, P(S(\tau)=s \mid X(\tau)=x)$$
$$= P(Y(\tau)=y \mid S(\tau)=s) \cdot P(S(\tau)=s \mid X(\tau)=x)$$

Thus,

$$P(Y(\tau)=y \mid S(\tau)=s)$$
$$\approx \sum_{\tau \in \mathbb{T}_x} \frac{P\left(S(\tau)=s, Y(\tau)=y \mid [X]_1^T=[x]_1^T, [Y]_1^T=[y]_1^T\right)}{|\mathbb{T}_x|}$$

This equation holds for every value of x. Therefore a better estimator can be derived by averaging the values over all inputs:

$$P(Y(\tau)=y \mid S(\tau)=s)$$
$$\approx \sum_{\tau=1}^{T} \frac{P\left(S(\tau)=s, Y(\tau)=y \mid [X]_1^T=[x]_1^T, [Y]_1^T=[y]_1^T\right)}{|\mathbb{T}_{x_\tau}| \cdot n_a \cdot P(S(\tau)=s \mid X(\tau)=x_\tau)}$$

Implementation

To get a bit more concrete, I will add the Python code I wrote to execute the steps described above.

The definition of a POMDP. For simplicity, inputs and outputs are supposed to be natural numbers. The transition matrices corresponding to each of the input characters are stored in `alist` (where `alist[i]` is the transition matrix that corresponds to input symbol `i`). As before, the matrix that maps state- to observation probabilities is given by `c`. The initial state distribution is stored in `init`.

```
class MarkovModel(object):
    def __init__(self,alist,c,init):
        self.alist = alist
        self.c = c
        self.ns = c.shape[1]
        self.os = c.shape[0]
        self.inps = len(alist)
        self.init = init
```

In the above sections, the procedure was stated in a recursive form. It is, however, not advisable to actually implement the algorithm as a recursion as this will lead to a bad performance. A better approach is to use a dynamic programming approach: The α and β values are stored in a matrix where

- each column corresponds to one point in time τ
- each row corresponds to one state

This is where actually most of the magic happens.

This function will generate

- a tableau for $\alpha(\tau)$
- a tableau for $\beta(\tau)$ and
- a tableau for $\mathcal{N}(\tau)$

To make the computation of α and $\mathcal{N}$ more efficient, I also calculate the common factor γ as derived above:

```
def make_tableaus(xs,ys,m):
    alpha = np.zeros((len(ys),m.ns))
    beta  = np.zeros((len(ys),m.ns))
    gamma = np.zeros((len(ys),m.ns))
    N     = np.zeros((len(ys),1))

    # Initialize:
    gamma[0:1,:] = m.init.T*m.c[ys[0]:ys[0]+1,:]
    N[0,0] = 1 / np.sum(gamma[0:1,:])
    alpha[0:1,:] = N[0,0]*gamma[0:1,:]
    beta[len(ys)-1:len(ys),:] = np.ones((1,m.ns))

    for i in range(1,len(ys)):
        gamma[i:i+1,:] = m.c[ys[i]:ys[i]+1,:] * np.sum((m.alist[xs[i-1]].T*alpha[i-1:i,:].T),axis=0)
        N[i,0] = 1 / np.sum(gamma[i:i+1,:])
        alpha[i:i+1,:] = N[i,0]*gamma[i:i+1,:]
    for i in range(len(ys)-1,0,-1):
        beta[i-1:i] = N[i]*np.sum(m.alist[xs[i-1]]*(m.c[ys[i]:ys[i]+1,:]*beta[i:i+1,:]).T,axis=0)
    return alpha,beta,N
```

These tableaus can be used to solve many inference problems in POMDPs. For instance, given a sequence of inputs xs and a sequence of observations ys that correspond to xs, estimate the probability of being in a certain state during each state. Put differently: The function `state_estimates` will calculate the posterior distribution over all latent variables.

I included an optional `tableaus` parameter. This ensures that, if we want to solve several inference problems, we only need to calculate the tableaus once.

```
def state_estimates(xs,ys,m,tableaus=None):
    if tableaus is None: tableaus = make_tableaus(xs,ys,m)
    alpha,beta,N = tableaus
    return alpha*beta
```

With the formulas that we derived above and using the tableaus, this becomes very simple. Note that the standard meaning of the *-operator in numpy is not matrix multiplication but element-wise multiplication.

A bit more sophisticated is the following inference problem: Given a sequence of inputs xs and a sequence of observations ys that correspond to xs, estimate the probability of transferring between two states at each time step.

```
def transition_estimates(xs,ys,m,tableaus=None):
    if tableaus is None: tableaus = make_tableaus(xs,ys,m)
    alpha,beta,N = tableaus
    result = np.zeros((m.ns,m.ns,len(ys)))
    for t in range(len(ys)-1):
        a = m.alist[xs[t]]
        result[:,:,t] = a*alpha[t:t+1,:]*m.c[ys[t+1]:ys[t+1]+1,:].T*beta[t+1:t+2,:].T*N[t+1,0]
    a=m.alist[xs[len(ys)-1]]
    result[:,:,len(ys)-1] = a*alpha[-1:,:]
    return result
```

The next function again takes an input sequence and an output sequence and for each time step computes the posterior probability of being in a state **and** observing a certain output.

Note that as the sequence of outputs is already given, this function does not add much: If in time step τ an output of 1 was observed, obviously the probability of observing 2 in time step τ will be zero.

```
def stateoutput_estimates(xs,ys,m,sestimate=None):
    if sestimate is None: sestimate = state_estimates(xs,ys,m)
    result = np.zeros((m.os,m.ns,len(ys)))
    for t in range(len(ys)):
        result[ys[t]:ys[t]+1,:,t] = sestimate[t:t+1,:]
    return result
```

Having defined these functions, we can implement the Baum-Welch style EM update procedure for POMDPs. It simply calculates

- The posterior state estimates for each τ
- The posterior transition estimates for each τ
- The posterior joint state/output estimates for each τ

This method can be repeated until the model converges (for some definition of convergence). The return value of this function is a new list of transition probabilities and a new matrix of output probabilities. These two define an updated POMDP model which should explain the data better than the old model.

```
def improve_params(xs,ys,m,tableaus=None):
    if tableaus is None: tableaus = make_tableaus(xs,ys,m)
    estimates = state_estimates(xs,ys,m,tableaus=tableaus)
    trans_estimates = transition_estimates(xs,ys,m,tableaus=tableaus)
    sout_estimates = stateoutput_estimates(xs,ys,m,sestimate=estimates)

    # Calculate the numbers of each input in the input sequence.
    nlist = [0]*m.inps
    for x in xs: nlist[x] += 1

    sstates = [np.zeros((m.ns,1)) for i in range(m.inps)]
    for t in xrange(len(ys)): sstates[xs[t]] += estimates[t:t+1,:].T/nlist[xs[t]]

    # Estimator for transition probabilities
    alist = [np.zeros_like(a) for a in m.alist]
    for t in xrange(len(ys)):
        alist[xs[t]] += trans_estimates[:,:,t]/nlist[xs[t]]
    for i in xrange(m.inps):
        alist[i] = alist[i]/(sstates[i].T)
        np.putmask(alist[i],(np.tile(sstates[i].T==0,(m.ns,1))),m.alist[i])

    c = np.zeros_like(m.c)
    for t in xrange(len(ys)):
        x = xs[t]
        c += sout_estimates[:,:,t]/(nlist[x]*m.inps*sstates[x].T)
    # Set the output probabilities to the original model if
    # we have no state observation at all.
    sstatem = np.hstack(sstates).T
    mask = np.any(sstatem == 0,axis=0)
    np.putmask(c,(np.tile(mask,(m.os,1))),m.c)

    return alist,c
```

How to test that? Well, we've seen how to calculate the log-likelihood of the data under a model:

```
def likelihood(tableaus):
    alpha,beta,N = tableaus
    return np.product(1/N)

def log_likelihood(tableaus):
    alpha,beta,N = tableaus
    return -np.sum(np.log(N))
```

Summary and Caveats

This post showed how to learn a POMDP model with python. The technique seems to be reasonably numerically stable (while I experienced major problems with a version based on the original alpha-beta method). Still there are some degeneracies that can occur, e.g. if a certain input was never observed the result of the division by `nlist[xs[t]]` may not be defined. That is why I used that strange `mask` construction to work around the problem. I do not claim that the implementation that I used is extraordinarily fast or optimised and I'd be glad about suggestions how to improve it.

I also experimented with a version of the function that creates a weighted average of the old and the new transition probabilities. This technique seemed to be more appropriate for some applications.

From a theoretical point of view the derivation I presented is no black magic at all: It turns out that learning a POMDP model is almost the same as learning a HMM, except that we have one transition matrix for each input to the system. Yet, it is still nice to see that it does work!

Finally, there is the unfortunate caveat of every EM-based technique: Even though the algorithm is guaranteed to converge, there is no guarantee that it finds the global optimum.

References

[1]	LE Baum, T Petrie, and G Soules. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 1970. [http]
[2]	Pierre a Devijver. Baum's forward-backward algorithm revisited. *Pattern Recognition Letters*, 3(6):369-373, December 1985. [DOI | http] In this note, we examine the forward-backward algorithm from the computational viewpoint of the underflow problem inherent in Baum's (1972) oritinal formulation. We demonstrate that the conversion of Baum's computation of joint likelihoods into the computation of posterior probabilities results in essentially the same algorithm, except for the presence of a scaling factor suggested by Levinson et al. (1983) on rather heuristic grounds. The resulting algorithm is immune to the underflow problem, and Levinson's scaling method is given a theoretical justification. We also investigate the relationship between Baum's algorithm and the recent algorithms of Askar and Derin (1981) and Devijver (1984). Keywords: estimation,forward-backward,hidden markov chains,maximum likelihood estimation
[3]	Andrew McCallum. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 190-196. Citeseer, 1993. [http] This paper presents a method by which a reinforcement learning agent can solve the incomplete perception problem using memory. The agent uses a hidden Markov model (HMM) to represent its internal state space and creates memory capacity by splitting states of the HMM. The key idea is a test to determine when and how a state should be split: the agent only splits a state when doing so will help the agent predict utility. Thus the agent can create only as much memory as needed to perform the task at hand – not as much as would be required to model all the perceivable world. Icall the technique UDM for Utile Distinction Memory.
[4]	Andrew McCallum. *Reinforcement learning with selective perception and hidden state*. Phd thesis, University of Rochester, 1996. [.ps.gz]
[5]	Daniel Mescheder, Karl Tuyls, and Michael Kaisers. Opponent Modeling with POMDPs. In *Proc. of 23nd Belgium-Netherlands Conf. on Artificial Intelligence (BNAIC 2011)*, pages 152-159. KAHO Sint-Lieven, Gent, 2011. [.pdf]

	Reinforcement Learning techniques such as Q-learning are commonly studied in the context of two-player repeated games. However, Q-learning fails to converge to best response behavior even against simple strategies such as Tit-for-two-Tat. Opponent Modeling (OM) can be used to overcome this problem. This article shows thatOMbased on Partially Observable Markov Decision Processes (POMDPs) can represent a large class of opponent strategies. A variation of McCallum's Utile Distinction Memory algorithm is presented as a means to compute such aPOMDPopponent model. This technique is based on Baum-Welch maximum likelihood estimation and uses a t-test to adjust the number of model states. Experimental results demonstrate that this algorithm can identify the structure of strategies against which pure Q-learning is insufficient. This provides a basis for best response behavior against a larger class of strategies.
[6]	Lloyd R Welch. Baum-Welch Algorithm. *IEEE Information Theory Society Newsletter*, 53(4), 2003.

This file was generated by bibtex2html 1.95.

Written by Daniel Mescheder

December 5, 2011 at 12:57 am

Posted in Tech

Tagged with MachineLearning, Programming, Python

11 Responses

Subscribe to comments with RSS.

This is really interesting stuff.
Thanks so much for sharing your hard work.

Can you upload a test data set to play with as well

Mike

February 10, 2012 at 5:32 pm

Reply

Cheers Mike,
I'll try to upload some data when I find the time.
However, it is quite easy to generate some dummy data just to test how well the algorithm works yourself.
What I did is to simply created a few POMDPs and used them to sample data. Then I compared the learned model with the POMDP that I sampled the data from.
I don't know whether there is any standard benchmark for that problem.

Daniel Mescheder

February 10, 2012 at 10:36 pm

Reply

Hi Daniel,

I was using your code to train a POMDP .. and as i realized that I get stuck to the same probabilities if I initialize them flat (same prob to all) … but works fine when i initialize them randomly. Should this happen with this code or am i committing some mistake.

Thanks a lot,
Rahul

Rahul Gupta

February 28, 2012 at 4:57 am

Reply

Hi Rahul,

Yes, that's normal.
I suspect that this is somehow due to the "local search" nature of the algorithm.
However, I did not investigate into this phenomenon so I don't know whether there might be a clever way around it – I indeed also ended up using random initialisation.

I hope that helps,
Daniel

Daniel Mescheder

February 28, 2012 at 12:20 pm

Reply

and to update the initial state probabilities am I correct in updating them with the first row of apha*beta that you calculated in state_estimates function ?

Rahul Gupta

February 28, 2012 at 6:33 pm

Hi Daniel,

I am curious about the derivation of the whole formula. Why did you omit the influence on the transition probability by dialogue action variable? What if considering the dialogue acts space? Thanks.

-Yang

Xuesong Yang

October 28, 2013 at 1:57 am

Reply

Hi Yang,
I'm afraid I don't quite understand what your question is aiming at. The derivation above is for a generic EM-like update algorithm for a specific kind of probabilistic model (namely a POMDP).

From your comment I suspect you want to apply this model to some kind of speech recognition/NLP problem?

Cheers,
Daniel

Daniel Mescheder

October 29, 2013 at 10:12 pm

Reply

Hi Daniel,

Thank you for your post and I found it is very helpful. I am struggling with POMDP training problem recently.

In terms of your post, I have some questions.

1. What is X and Y? Are they actions and observations?

2. When I tried to run your code, I was unable to run through. I built a POMDP with 2 states, 2 actions and 9 observations. Therefore, the state transition matrix alist was a 9*2*2 matrix, the observation matrix was a 9*2 matrix and initial state distribution was a 1*2 matrix. When the code ran to gamma[0:1,:] = m.init.T*m.c[ys[0]:ys[0]+1,:], I found that

m.init.T is a 2*1 matrix and m.c[ys[0]:ys[0]+1,:] is a 1:2 matrix, thereby generating a 2*2 matrix, while gamma[0:1,:] is a 1*2 matrix. Did I build a POMDP with wrong state transition matrix, observation matrix and initial state distribution?

Thank you very much.

Regards,

Ben

ben

December 9, 2013 at 11:51 am

Reply
Hi Daniel,

Thank you for your post. It is very helpful.

I do not quite understand how to derive the last step in EM-Update Rule to estimate P(Y|S), Why we have to divide P(St = s, Yt = y | Y(1->T), X(1->T)) by |Tx(t)| and na (or no? I cannot see it clearly). I am wondering if you can give some clue on deriving it?

Thank you very much.

Regards,

Ben

benedict1986

December 20, 2013 at 4:54 am

Reply
Hi all,
we integrated some POMDP code (clean controller, file-IO for models and policies and planning algorithms) with Daniel's learning stuff at bitbucket (see. https://bitbucket.org/bami/pypomdp/). Feel free to join us and develop the code base.

All the best

Bastian

Bastian Migge

February 14, 2014 at 10:43 am

Reply
[…] [A] Training a POMDP (with Python) https://danielmescheder.wordpress.com/2011/12/05/training-a-pomdp-with-python/ […]

Tutorial: EM Algorithm Derivation for POMDP | Ben's Footprint

April 21, 2014 at 6:19 am

Reply