Online representation learning in the state update function

Rich Sutton

Summer, 2009

Rep’n learning is a little different in RL

- we don’t want to be batch
  - because we have a use for improvements in rep’n as soon as we can find them
  - because we want to handle nonstationarity
- we have a natural source of multiple tasks
  - knowledge! learning to predict everything
  - some will be easy, some hard
  - effectively a sequence of tasks of graded difficulty

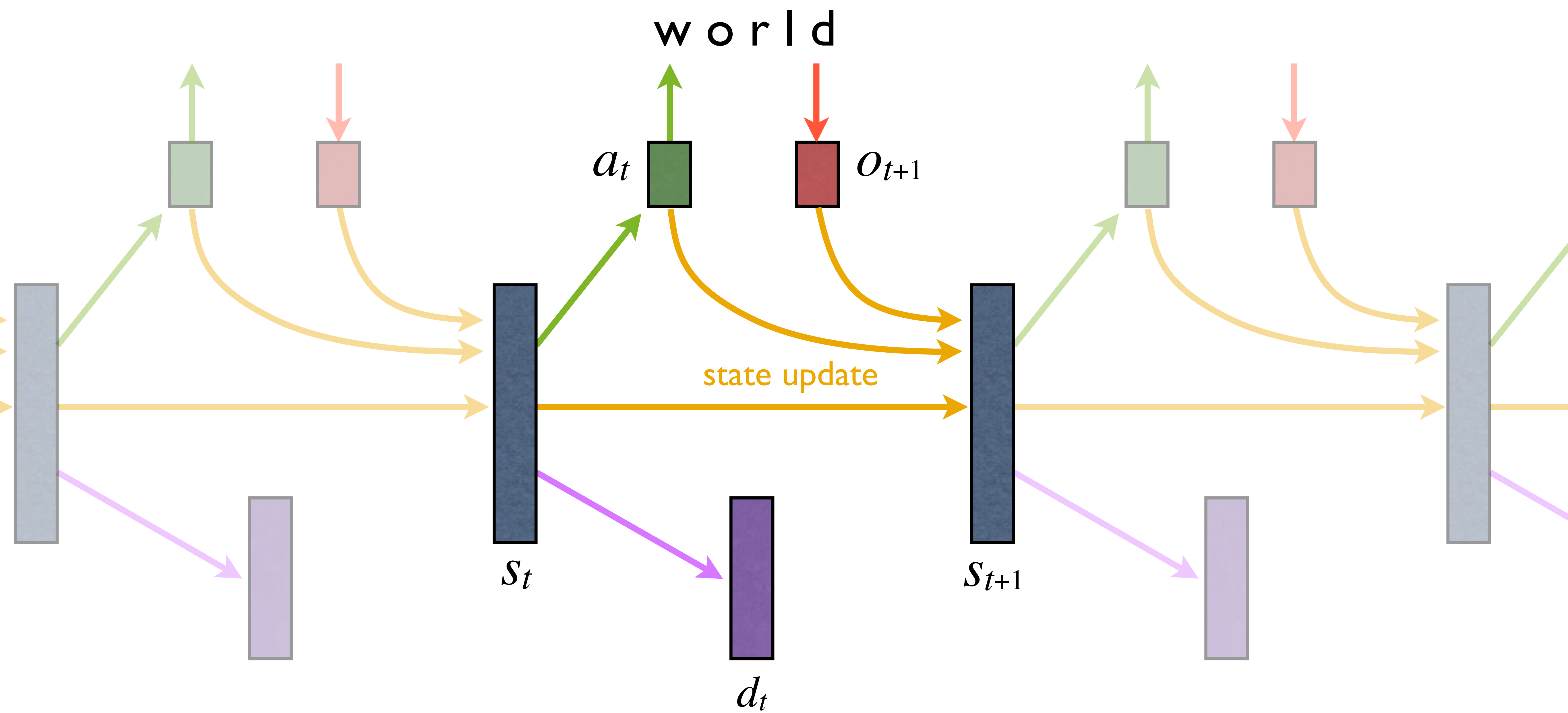
outline

1. the state-update function
2. the expand-and-add architecture
3. an insight into how to combine them nicely

RLAI architecture has two parts

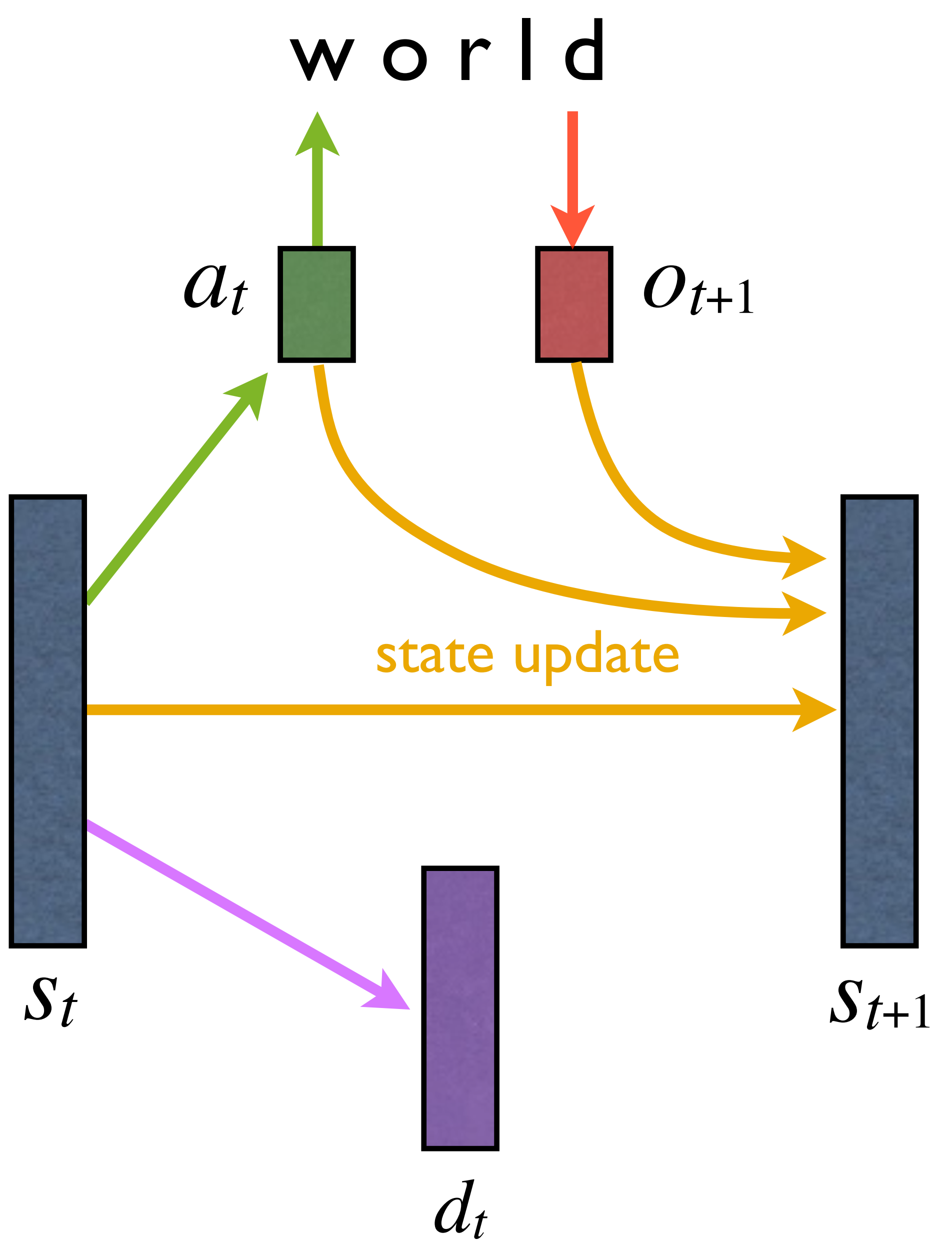
- the *reactive part* contains all the things that must run at the fastest rate of agent-environment interaction
  - e.g., in the critterbot, this is 100 times a second
  - the agent's state rep'n must be updated this fast
- the *deliberative part* is slower, accumulative
  - responsible for planning, cognition

RLAI architecture (reactive part)



- everything updates and learns 100 times a second
- *the pulse of the mind*

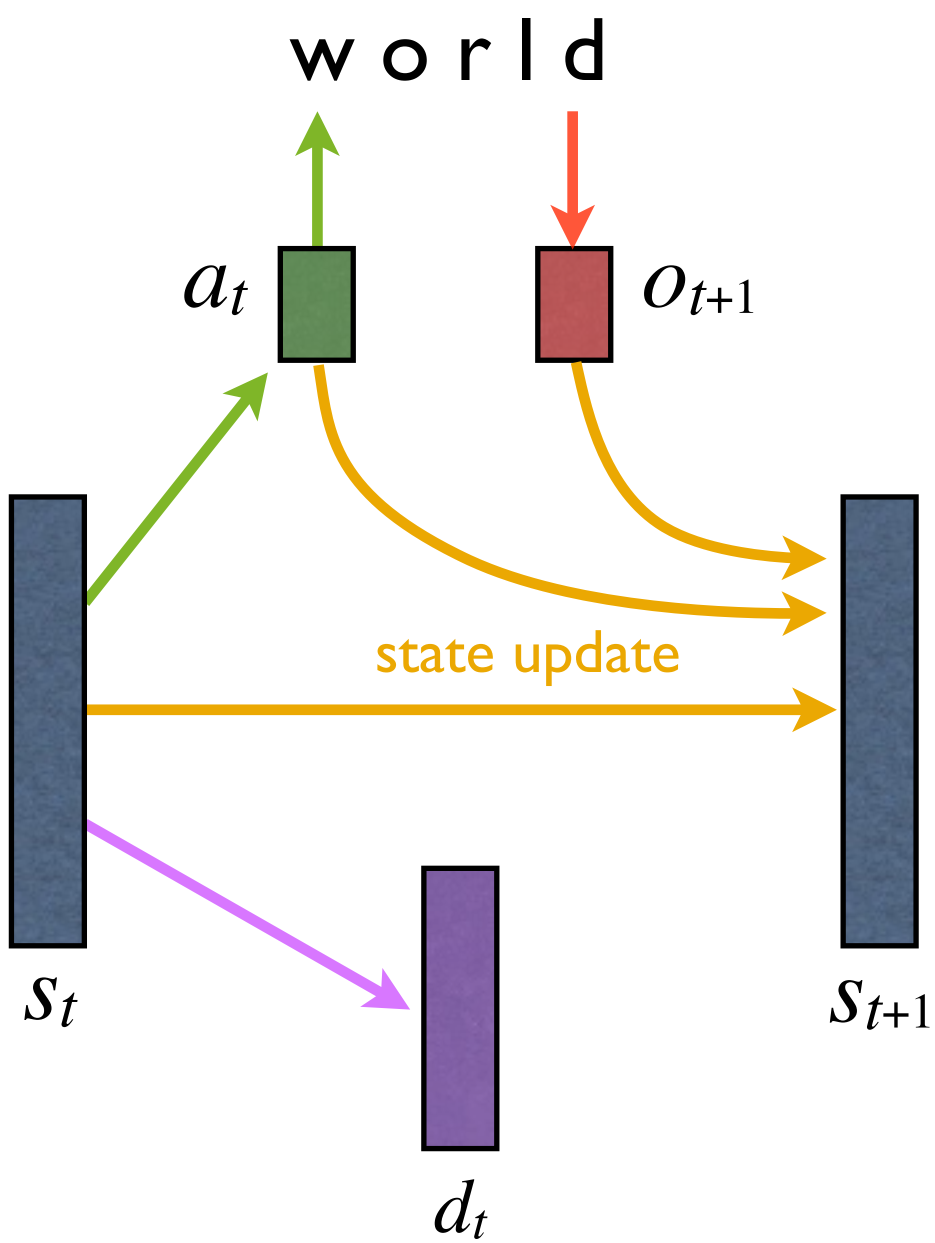
The (agent-)state update function



$$s_{t+1} = u(s_t, a_t, o_{t+1})$$

- the state-update function u creates/defines the state
- state update = *perception*
- changes in u = *representation learning*
- state is used by the demons to make predictions, learn policies
- state is also used for *planning* (not covered here)

RLAI architecture (demons)



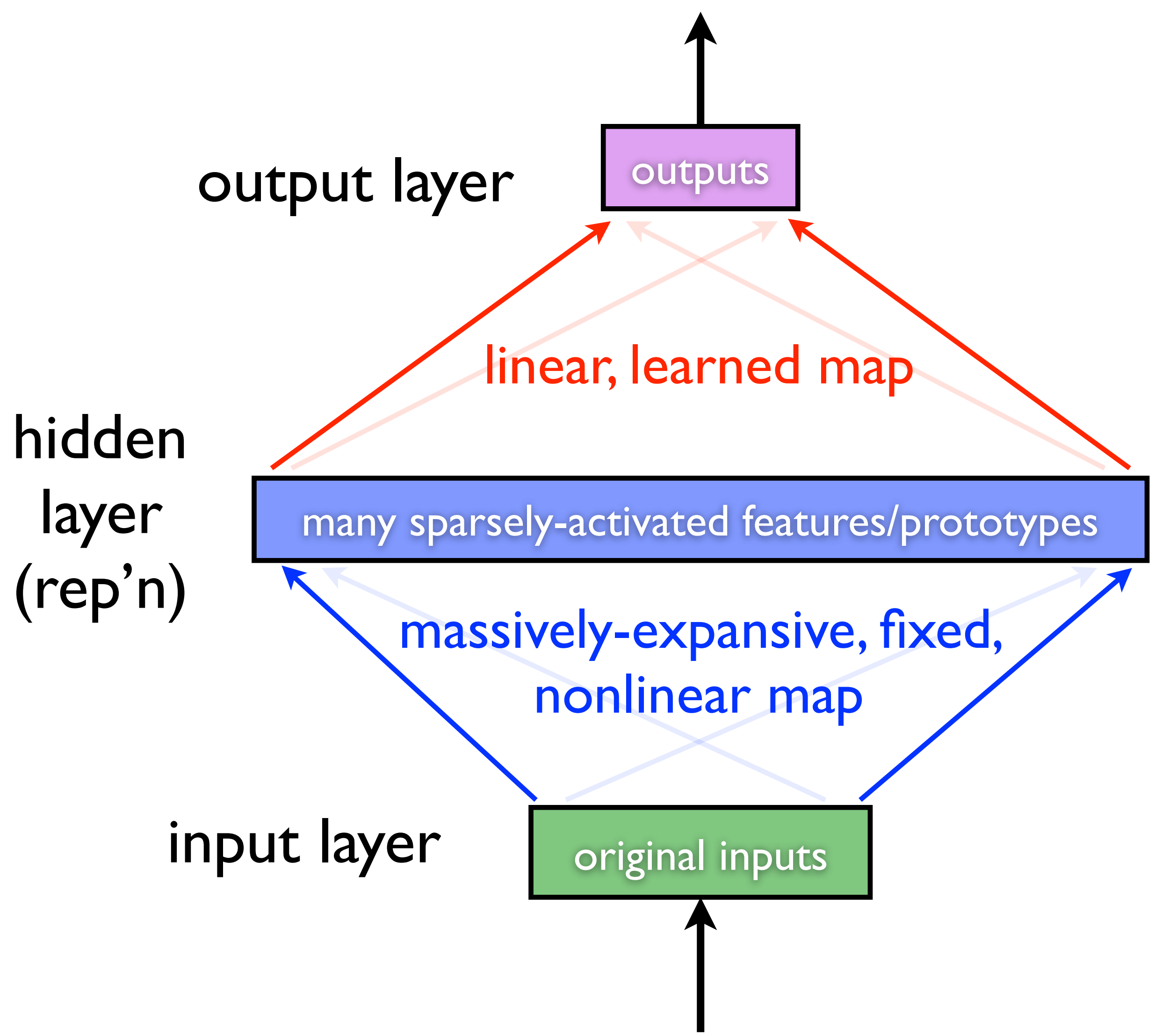
- the demons don't directly affect the state-update function
- but they can provide a reason for changing state update
- they provide a large set of tasks (general value functions)
- a feature good for one demon might also be good for another
- the demons are the *tester* in the generate-and-test search for good state features

outline

1. the state-update function
2. the expand-and-add architecture
3. an insight into how to combine them nicely

Expand and Add

the world's most popular function-approximation architecture

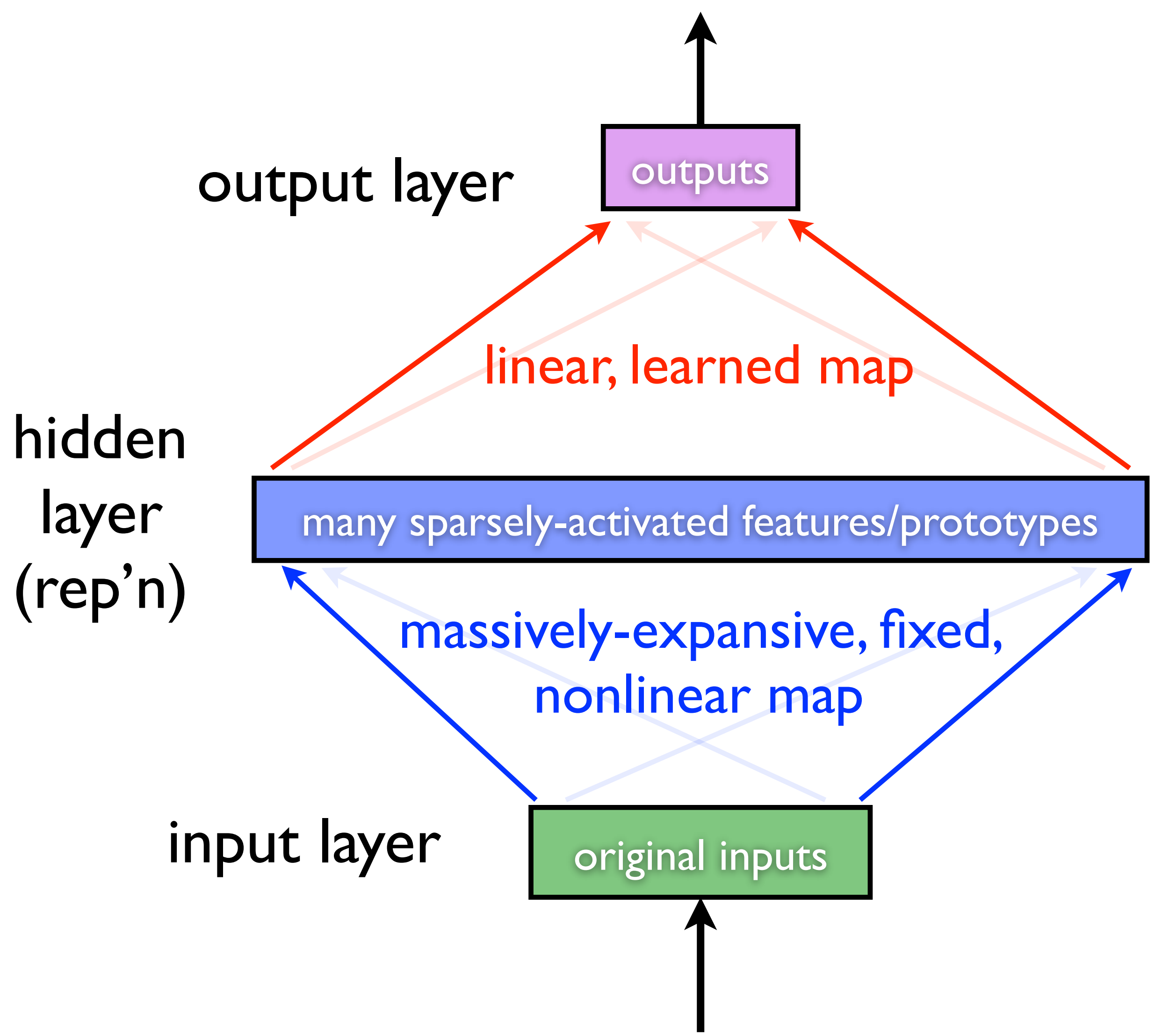


for example:

- tile coding
- radial basis functions
- support vector machines
- the original perceptron
- coarse coding
- kanerva coding

Expand and Add

the world's most popular function-approximation architecture

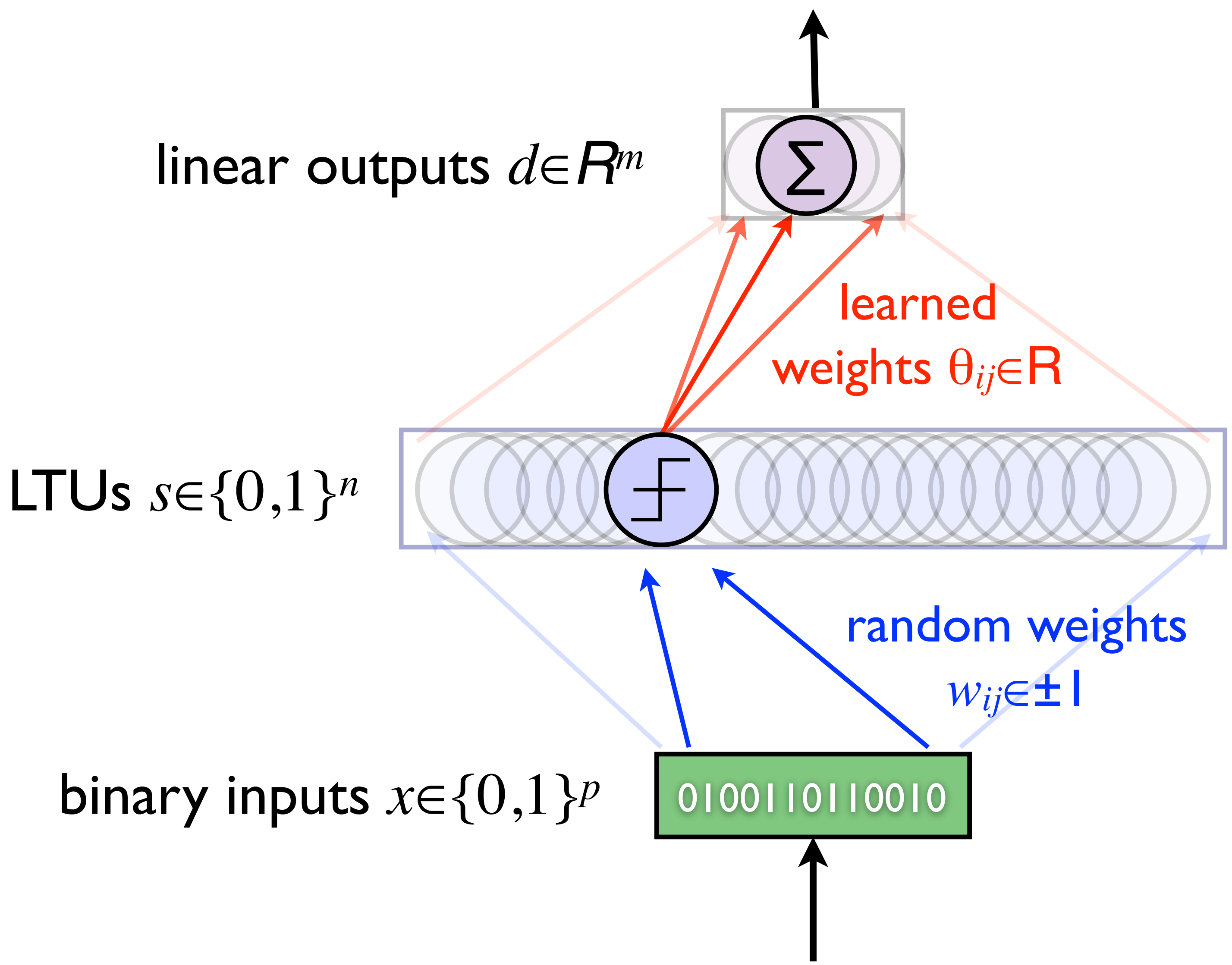


strengths:

- fast learning
- learns well online or batch
- powerful (expressive)
- well suited to representation learning

LTU-based Expand and Add

using Linear Threshold Units (LTUs) to form the feature rep'n



$$d_i = \sum_j \theta_{ij} s_j$$

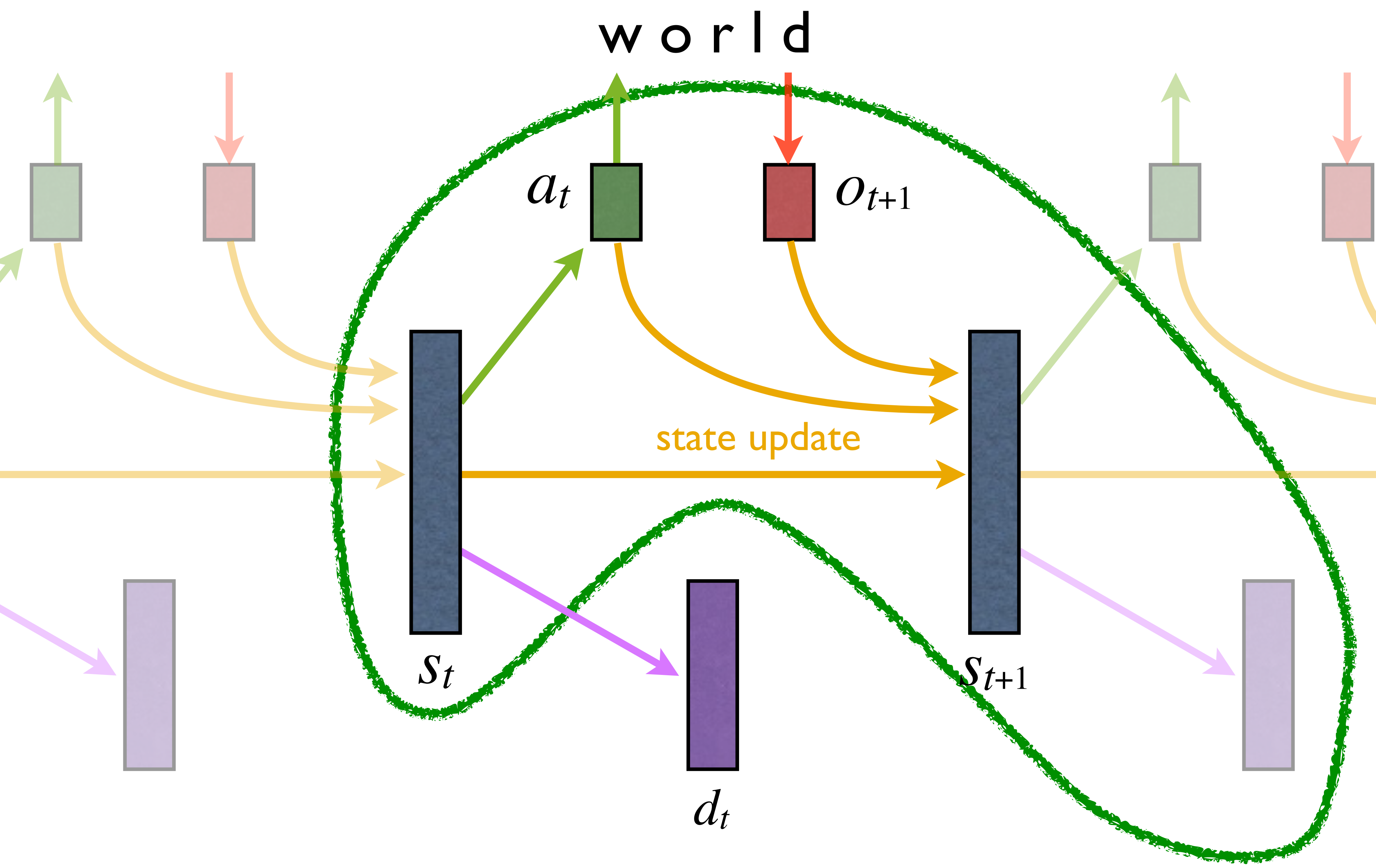
$$s_i = \sigma\left(\sum_j w_{ij} x_j\right)$$

$\sigma(x) = 1$ if $x >$ *threshold* else 0

Can we map this onto the state-update function?

outline

1. the state-update function
2. the expand-and-add architecture
3. an insight into how to combine them nicely

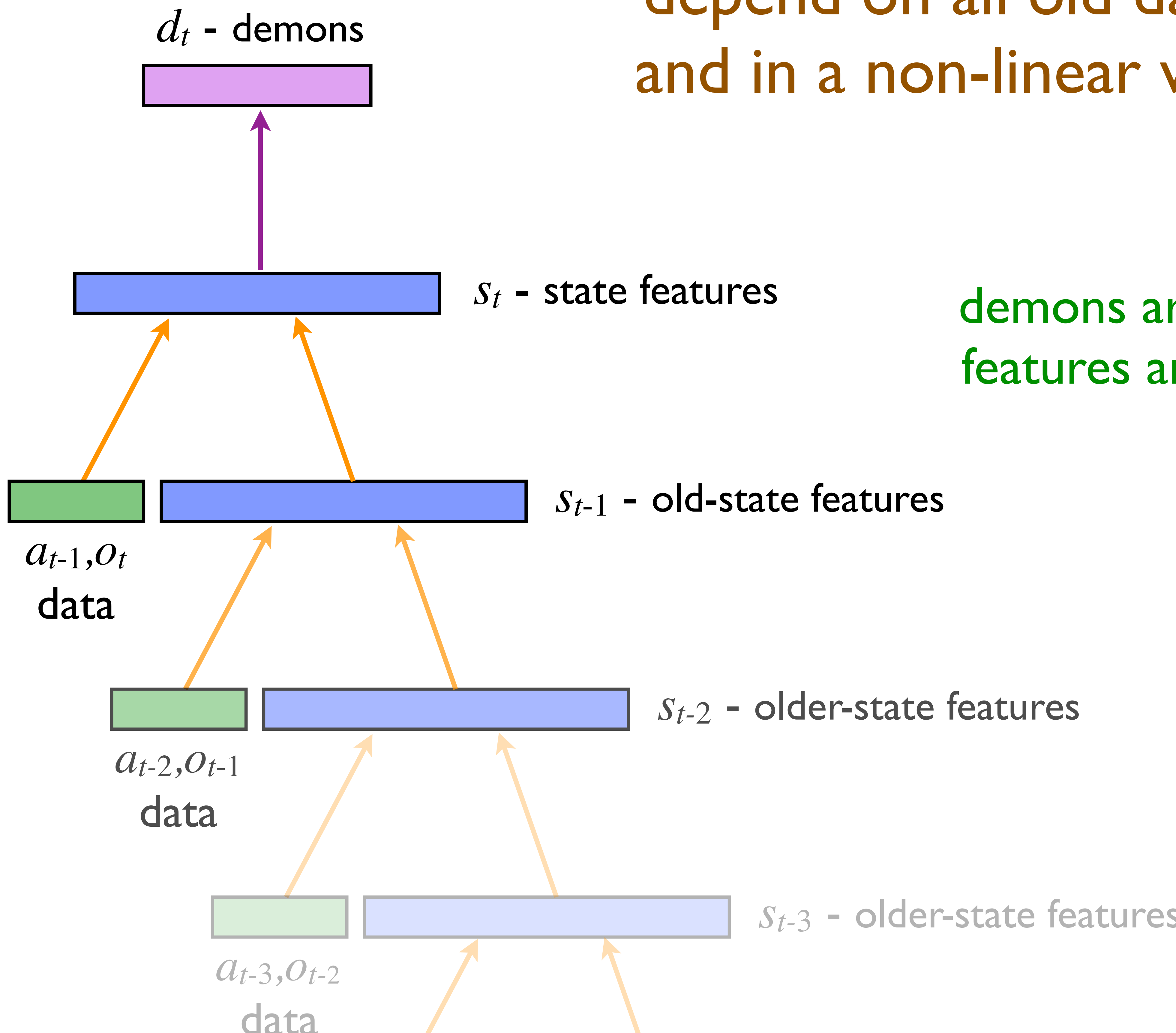


world
a_t
o_{t+1}
state update
s_t
s_{t+1}
d_t

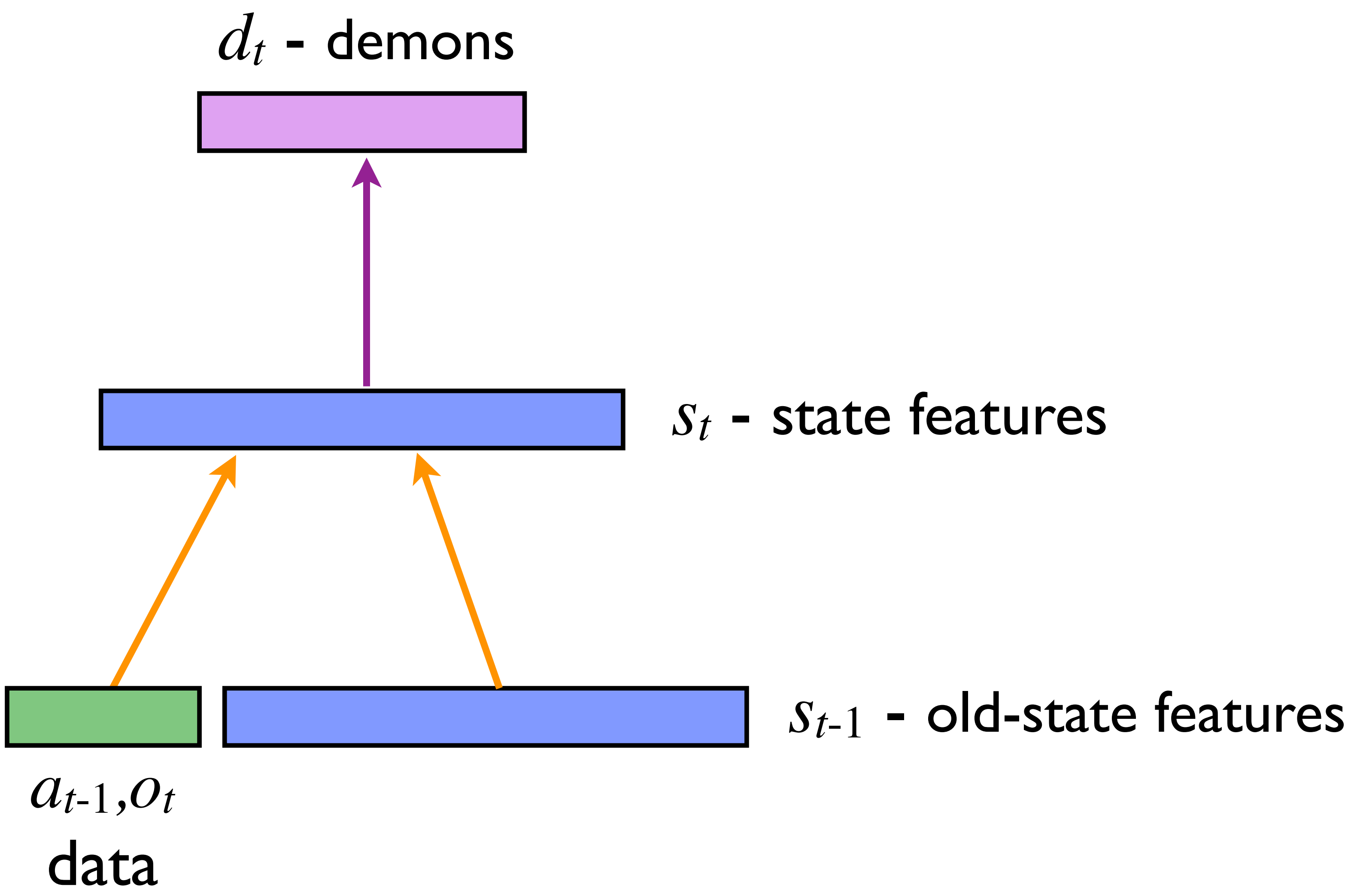
salience (step-size)

- let each feature i have a step-size, $\alpha_i \in \mathcal{R}^+$, also called its *salience*
- this determines how much the demons will generalize according to that feature
- important features should have high salience, irrelevant ones low salience
- salience is an important part of the rep'n
- IDBD and similar algorithms can be used to learn salience

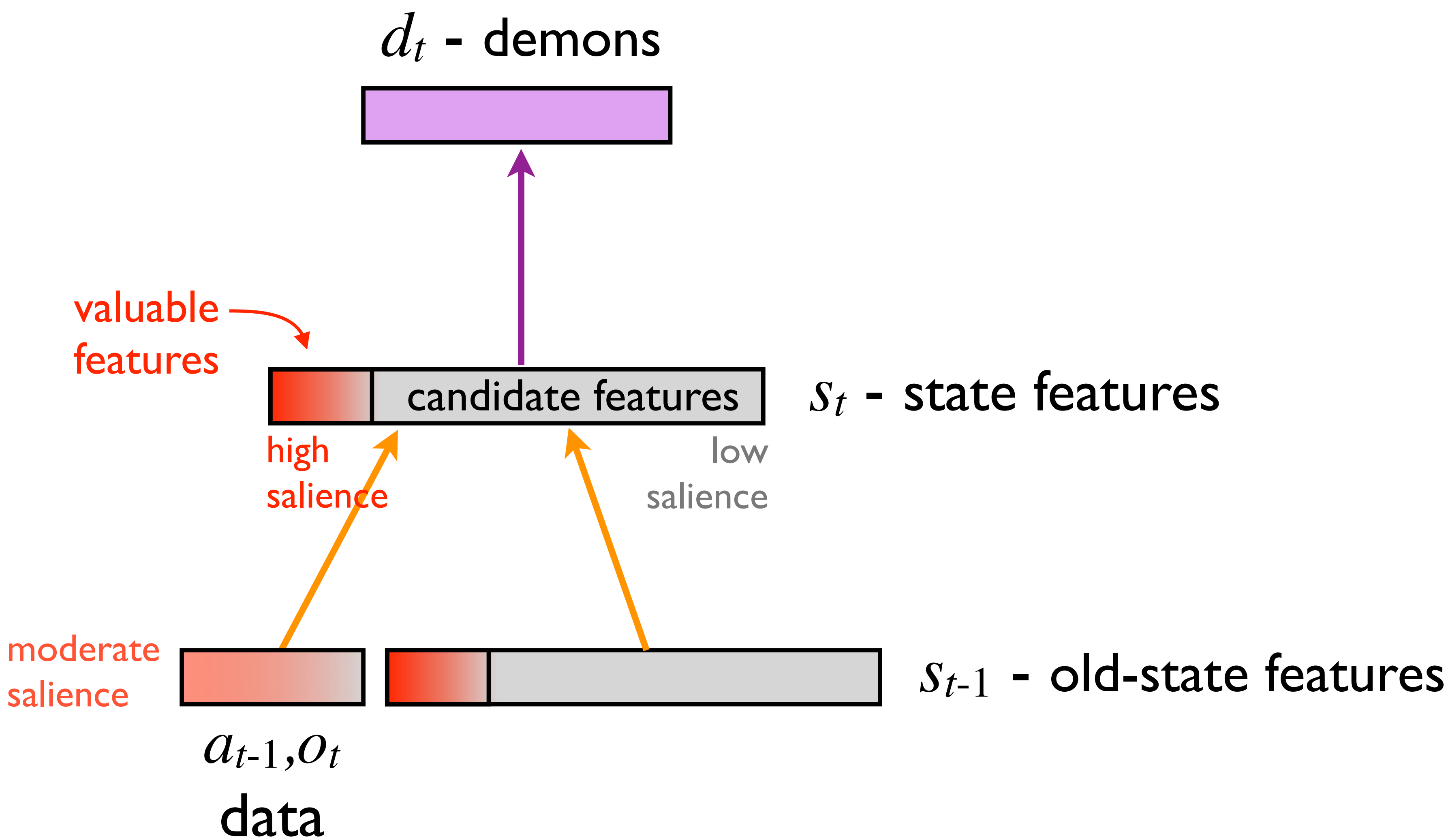
demon functions may depend on all old data, and in a non-linear way



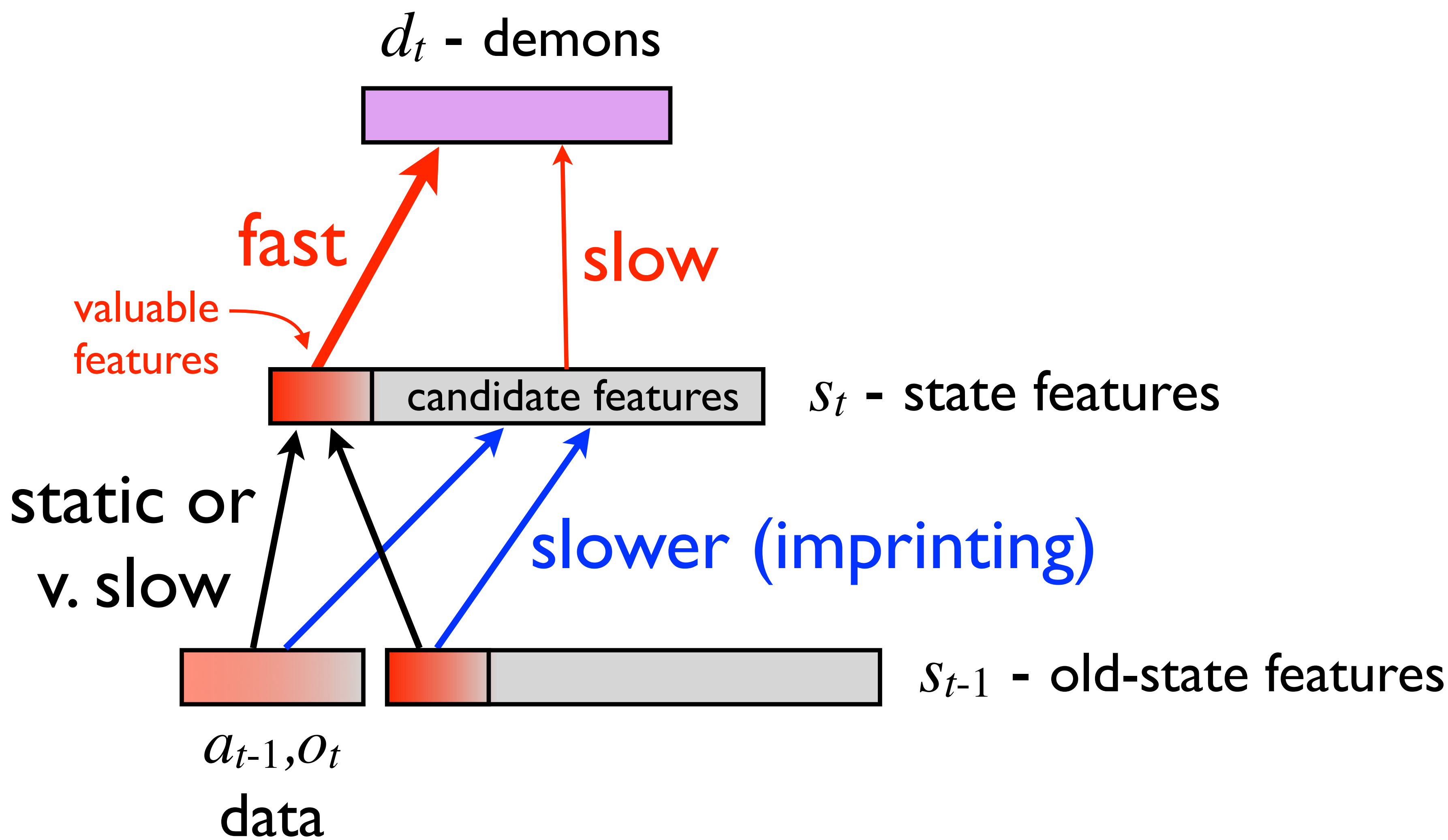
demons are linear
features are LTUs



d_t - demons
s_t - state features
$s_{t\text{-}1}$ - old-state features
$a_{t\text{-}1}, o_t$
data



d_t - demons
valuable
features
candidate features
s_t - state features
high
salience
low
salience
moderate
salience
$s_{t\text{-}1}$ - old-state features
$a_{t\text{-}1}, o_t$
data



d_t - demons
fast
slow
valuable features
candidate features
s_t - state features
static or v. slow
slower (imprinting)
$s_{t\text{-}1}$ - old-state features
$a_{t\text{-}1}, o_t$
data

imprinting

- imprint candidate features on time steps of high demon error
  - if error is low on a time step, then do nothing
  - if error is high, then try to make a feature that responds preferentially and distinctly to that time step
    - such a feature will help you reduce demon error in the future

support and valuableness

- state components (features) may be valuable because they are salient, or because they are used to construct features that are salient
- thus valuableness can be propagated from component to component
- we say that salient states are “valuable”, and valueableness propagates by supporting relationships
  - perhaps with a little friction, so that mutually supporting but non-salient components die off

conclusion

- state-update and expand-and-add combine nicely
- the state vector is *both* the small input vector *and* the massively expanded feature vector
  - via salience
- recursion, and thus higher-order features, is immediate
- we should be able to get fast, online learning and representation learning—generate and test through random feature space