Università degli Studi di Pavia

ARIADNE

Smart inventory management:

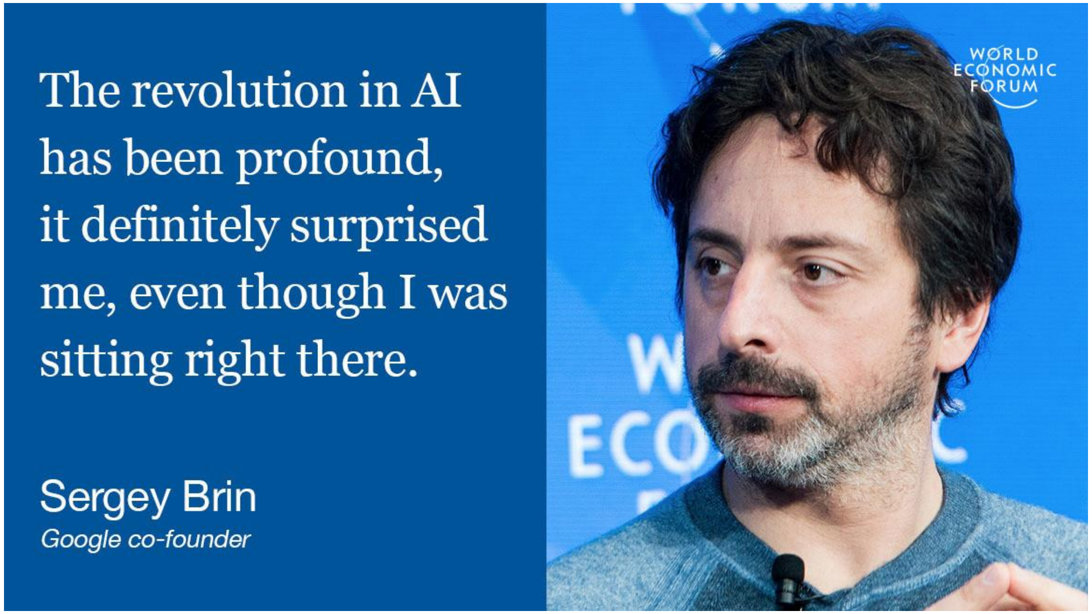
Will Deep Reinforcement Learning help us win the game?

Marco Piastra

25-01-2018

This presentation is available at:
http://vision.unipv.it/AI/AIRG.html

What happened with Artificial Intelligence?



- Sergey Brin [Google Co-Founder, January 2017]

  "I didn't pay attention to it [i.e. Artificial Intelligence] at all, to be perfectly honest."

  "Having been trained as a computer scientist in the 90s,
  everybody knew that AI didn't work.
  People tried it, they tried neural nets and none of it worked."

[Quote and image from https://www.weforum.org/agenda/2017/01/google-sergey-brin-i-didn-t-see-ai-coming/]

Reinforcement Learning: we knew that already…

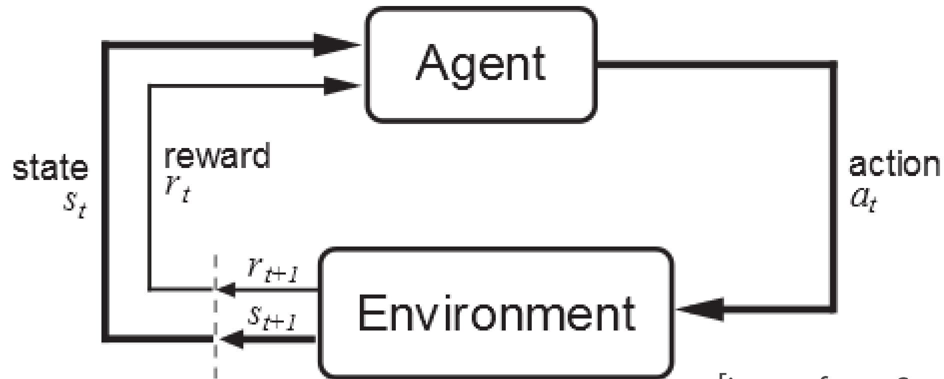
Agent/Environment Interactions

- General setting with **Reinforcement Learning**

  An *agent*, that performs *actions* on an *environment*

  The actions of the agent change the *state* of the environment

  The agent gets a *reward* (either positive or negative) in consequence of its action



[image from: Sutton, Barto, *Reinforcement Learning.* 1998]

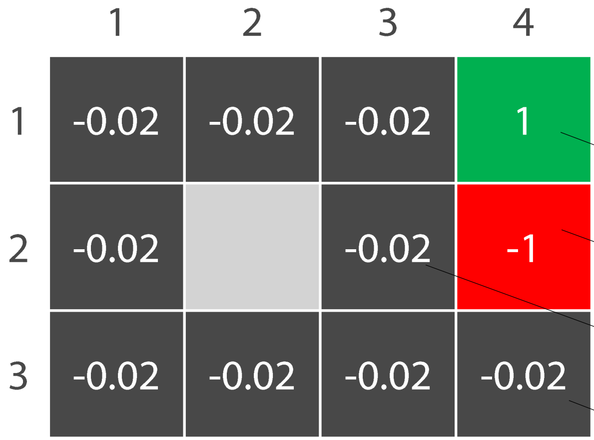
  Examples:

  - a_t could be a *move in a game*, whereby the agent changes the state of the game
  - a_t could be a *movement*, whereby the agent changes its position in the environment

  The agent seeks an *optimal strategy* towards a given goal…

Markov Decision Process (MDP)

The Gridworld example



this is the *goal* state, to be reached

this is a <u>pitfall</u>, to be avoided

in this example, *reward* values depend on *states* only

each box is a *state* the *agent* may be in

Markov Decision Process: $< \mathcal{S}, \mathcal{A}, r, P, \gamma >$

Set of *<u>states</u>* : $\mathcal{S} = \{s_1, s_2, \dots\}$

Set of *<u>actions</u>* : $\mathcal{A} = \{a_1, a_2, \dots\}$

<u>reward function</u> : $r : \mathcal{S} \rightarrow \mathbb{R}$

the outcome of *agent's actions* is uncertain

<u>transition probability distribution</u> : $P(S_{t+1} \mid S_t, A_t)$ (also called a *<u>model</u>*)

Markov property: the transition probability depends only the previous state and action

$$P(S_{t+1} \mid S_t, A_t) = P(S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, S_{t-2}, A_{t-2}, \dots)$$

<u>discount factor</u> : $0 \leq \gamma < 1$

Markov Decision Process (MDP): policies and values

The agent is supposed to adopt a *deterministic* policy : $\pi : \mathcal{S} \to \mathcal{A}$

In other words, the agent always chooses its *action* depending on the *state* alone

Given a policy π , the ***state value function*** is defined, for each state s as:

$$V^{\pi}(s) := \mathbb{E}[r(S_t) + \gamma r(S_{t+1}) + \gamma^2 r(S_{t+2}) + \ldots \mid \pi, S_t = s]$$

Note the role of the *discount factor*: a value $\gamma < 1$ means that that future rewards could be weighted less (by the agent) than immediate ones

Note also that all states S_t must be described by *random variables* :
i.e. the *policy* is deterministic but *state transitions* are not

Note also that when the reward is *bounded,* i.e. $r(S) \leq r_{\max}$

$$\sum_{t=0}^{\infty} \gamma^t \, r(S_t) \;\leq\; r_{\max} \sum_{t=0}^{\infty} \gamma^t \;=\; r_{\max} \, \frac{1}{1-\gamma}$$

for $\gamma < 1$ this is the *geometric series*

Bellman equations

By working on the definition of value function:

$$
\begin{aligned}
V^{\pi}(s) &:= \mathbb{E}[r(S_t) + \gamma r(S_{t+1}) + \gamma^2 r(S_{t+2}) + \ldots \mid \pi, S_t = s] \\
&= \mathbb{E}[r(S_t) + \gamma(r(S_{t+1}) + \gamma r(S_{t+2}) + \ldots) \mid \pi, S_t = s] \\
&= r(s) + \gamma \mathbb{E}[r(S_{t+1}) + \gamma r(S_{t+2}) + \ldots \mid \pi, S_t = s] \\
&= r(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) \cdot \mathbb{E}[r(S_{t+1}) + \gamma r(S_{t+2}) + \ldots \mid \pi, S_{t+1} = s'] \\
&= r(s) + \gamma \sum_{S_{t+1}} P(S_{t+1} \mid s, \pi(s)) \cdot V^{\pi}(S_{t+1})
\end{aligned}
$$

This means that in a Markov Decision Process:

$$V^{\pi}(s) = r(s) + \gamma \sum_{S_{t+1}} P(S_{t+1} \mid s, \pi(s)) \cdot V^{\pi}(S_{t+1})$$

This is true for any *state*, so there is one such equation for each of those

If the set of states is finite, there are exactly $|S|$ (linear) Bellman equations for $|S|$ variables:
in general, for any deterministic policy, V^{π} can be computed analytically

Optimal policy – Optimal value function

- Basic definitions

$$\pi^*(s) := \mathrm{argmax}_\pi V^\pi(s),\ \forall s \in S$$

$$V^*(s) := \max_\pi V^\pi(s),\ \forall s \in S$$

Property: for every MDP, there exists such an optimal deterministic policy (*possibly non-unique*)

With Bellman Equations:

$$\max_\pi V^\pi(s) = r(s) + \gamma \max_\pi \left(\sum_{S_{t+1}} P(S_{t+1} \mid s, \pi(s)) \cdot V^\pi(S_{t+1}) \right)$$

$$V^*(s) = r(s) + \gamma \max_\pi \left(\sum_{S_{t+1}} P(S_{t+1} \mid s, \pi(s)) \cdot V^*(S_{t+1}) \right)$$

$$= r(s) + \gamma \max_a \left(\sum_{S_{t+1}} P(S_{t+1} \mid s, a) \cdot V^*(S_{t+1}) \right)$$

Therefore:

$$\pi^*(s) = \mathrm{argmax}_a \left(\sum_{S_{t+1}} P(S_{t+1} \mid s, a) V^*(S_{t+1}) \right)$$

Computing V^* *directly from these equations is unfeasible, however*

There are in fact $|A|^{|S|}$ *possible strategies*

However, once V^* *has been determined,* π^* *can be determined as well*

Optimal policy – Optimal value function

- Value iteration algorithm

Initialize: $V(s) := r(s),\ \forall s \in S$
Repeat:

Note that there is no policy: all actions must be explored

1) For every state, update: $V(s) := r(s) + \gamma \max_a \sum_{s'} P(s' \mid s, a) V(s')$

Theorem: for every fair way (*i.e. giving an equal chance*) of visiting the states in S, this algorithm converges to V^*

Computing the optimal policy

	1	2	3	4
1	-0.02	-0.02	-0.02	1
2	-0.02		-0.02	-1
3	-0.02	-0.02	-0.02	-0.02

Initialize states
(e.g. using rewards as initial values)

Iterate and compute

V^*

	1	2	3	4
1	0.86	0.90	0.93	1
2	0.82		0.69	-1
3	0.78	0.75	0.71	0.49

V^*

Define the optimal policy as:

$$\pi^*(s) := \text{argmax}_a \left(\sum_{S_{t+1}} P(S_{t+1} \mid s, a) \cdot V^*(S_{t+1}) \right)$$

	1	2	3	4
1	→	→	→	1
2	↑		↑	-1
3	↑	←	←	←

π^*

Computing the optimal policy

Nice, but not very realistic …

- *Main limitations of the* ***value function*** *approach*

  Everything must be known in advance:

  - the *environment* (i.e. the map, in the *gridworld* example)
  - the *model*, i.e. the transition probability

  These elements allow a direct (albeit expensive) computation of π^*

- *In reality*

  - the *environment* is in general unknown to agent
    which has to ***explore*** in order to gain knowledge of it
  - the *model*, i.e. the transition probability that determines the outcome of actions
    is also unknown to the agent
    (which implies that even more ***exploration*** is required)

Moral: we need to learn by doing…

Action value function

An analogous of the value function V^{π}

Given a policy π , the ***action value function*** is defined, for each pair (s, a) as:

$$Q^{\pi}(s,a) := \sum_{S_{t+1}} P(S_{t+1} \mid s,a) \cdot V^{\pi}(S_{t+1})$$

$$= \sum_{S_{t+1}} P(S_{t+1} \mid s,a) \cdot \mathbb{E}[r(S_{t+1}) + \gamma r(S_{t+2}) + \ldots \mid \pi, S_{t+1}]$$

$$= \sum_{S_{t+1}} P(S_{t+1} \mid s,a) \cdot [r(S_{t+1}) + \mathbb{E}[\gamma r(S_{t+2}) + \ldots \mid \pi, S_{t+1}]]$$

$$= \sum_{S_{t+1}} P(S_{t+1} \mid s,a) \cdot [r(S_{t+1}) + \gamma Q^{\pi}(S_{t+1}, \pi(S_{t+1}))]$$

In other words, $Q^{\pi}(s,a)$ is the expected value of the reward in S_{t+1}
by taking action a in state s and then following policy π from that point on

Following a similar line of reasoning, the ***optimal*** *action value function* is

$$Q^{*}(s,a) = \sum_{S_{t+1}} P(S_{t+1} \mid s,a) \cdot [r(S_{t+1}) + \gamma \max_{a'} Q^{*}(S_{t+1}, a')]$$

This is an expected value:
it can be approximated by an empirical average…

Q-Learning

- Q-learning algorithm (ε-*greedy version*)

Initialize $\hat{Q}(s,a)$ at random, put the agent in a random state s
Repeat:

An estimator of the 'real' Q function

1) Select the action $\mathrm{argmax}_a \hat{Q}(s,a)$ with probability $(1-\varepsilon)$ otherwise, select a at random
2) The agent is now in state s' and has received the reward r
3) Update $\hat{Q}(s,a)$ by

$$\Delta \hat{Q}(s,a) = \alpha[r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)]$$

Exponential Moving Average

Note in step 1) the dualism between ***exploration*** *and* ***exploitation****:*

- *with probability* $(1-\varepsilon)$ *the agent will* ***exploit*** *its knowledge* $\hat{Q}(s,a)$
- *with probability* ε *the agent will* ***explore*** *new actions*

Q-Learning

- Q-learning algorithm (ε-*greedy version*)

  Initialize $\hat{Q}(s,a)$ at random, put the agent in a random state s

  Repeat:

  1) Select the action $\text{argmax}_a \hat{Q}(s,a)$ with probability $(1-\varepsilon)$ otherwise, select a at random
  2) The agent is now in state s' and has received the reward r
  3) Update $\hat{Q}(s,a)$ by

$$\Delta \hat{Q}(s,a) = \alpha[r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)]$$

A very nice mathematical model, however:

- *the argmax in step 1) is expensive, in particular when* $\mathcal{A}$ *is* *<u>continuous</u>*....
- *learning* $\hat{Q}(s,a)$ *requires in general a* *<u>huge</u>* *amount of trials*....
- *and the latter problem becomes even worse when* $\mathcal{S}$ *is* *<u>continuous</u>*

Deep Learning:
this is new

- **Increasing network depth**

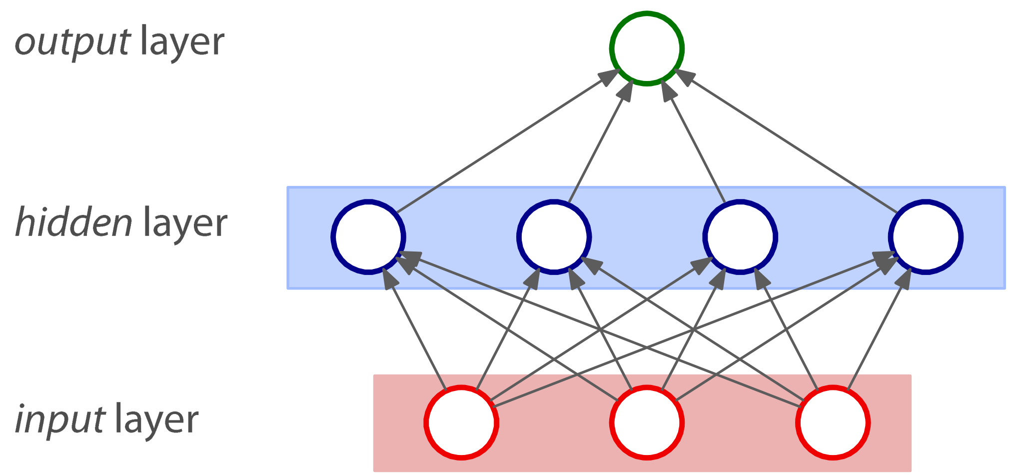
A feed-forward neural network with one hidden layer

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{c}^{(1)}) + c$$

output layer

hidden layer

input layer



Shallow vs. Deep Feed-Forward Neural Networks

- **Increasing network depth**

  A feed-forward neural network with one hidden layer

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{c}^{(1)}) + c$$

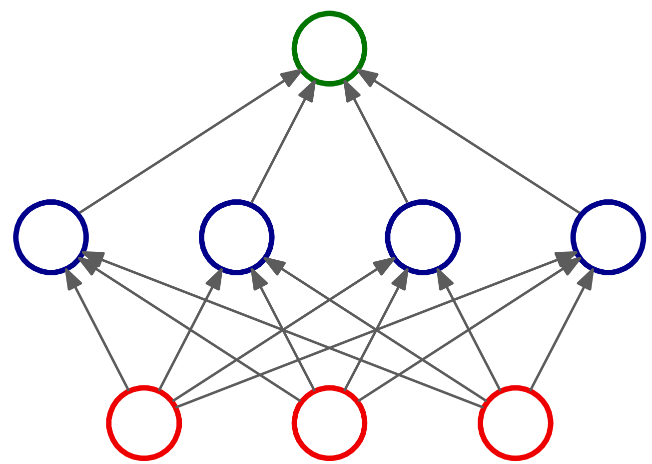
Universal approximation theorem (Cybenko, 1989, Hornik, 1991)

When g is a non-linear function of a certain class
any continuous target function

$$y = f^*(x), \quad x \in \mathbb{R}$$

can be approximated arbitrarily well by $\tilde{y}$
(in the sense that there exists parameters
$\boldsymbol{w}, \boldsymbol{W}^{(1)}, \boldsymbol{c}^{(1)}, c$ such that the above holds)

Want a better approximation?
Increase the number of units in the <u>hidden</u> layer …

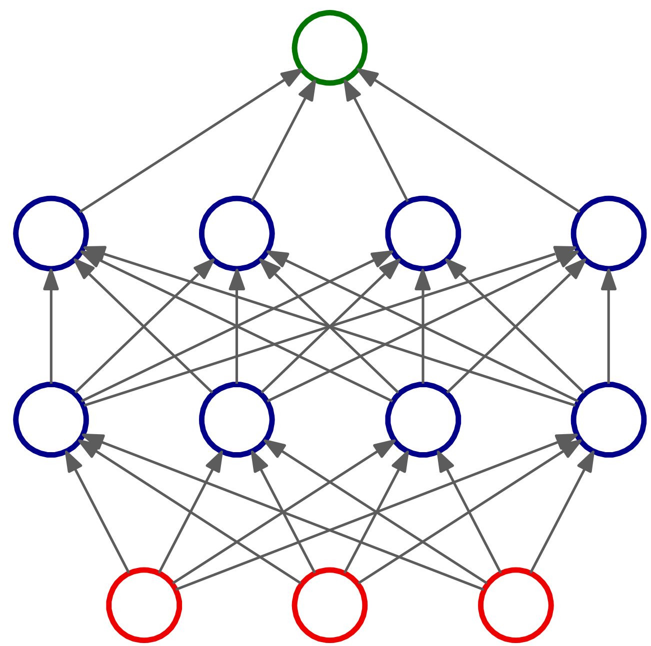


Shallow vs. Deep Feed-Forward Neural Networks

- **Increasing network depth**

  A feed-forward neural network with two hidden layers

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)} g(\boldsymbol{W}^{(2)} \boldsymbol{x} + \boldsymbol{c}^{(2)}) + \boldsymbol{c}^{(1)}) + c$$

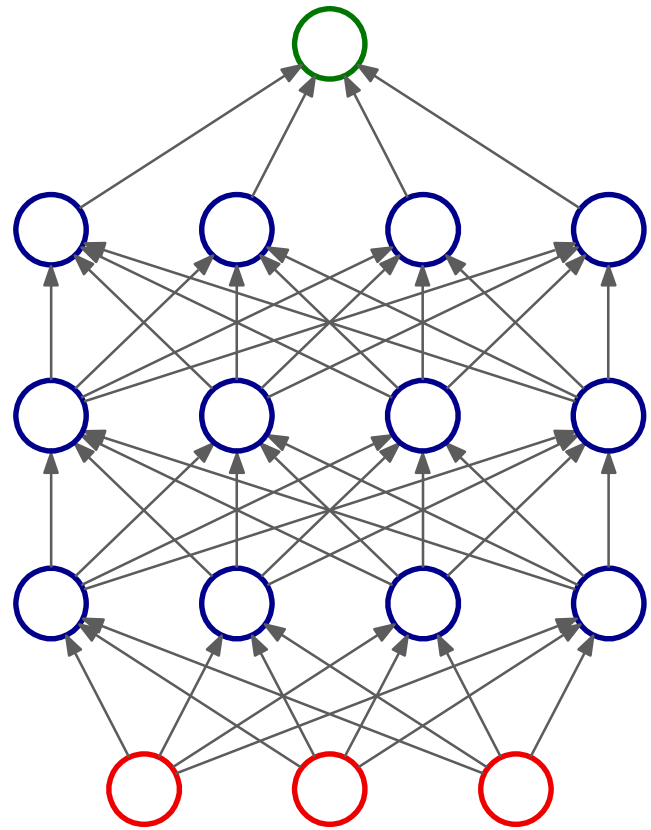


Shallow vs. Deep Feed-Forward Neural Networks

- **Increasing network depth**

  A feed-forward neural network with three hidden layers

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)} g(\boldsymbol{W}^{(2)} g(\boldsymbol{W}^{(3)} \boldsymbol{x} + \boldsymbol{c}^{(3)}) + \boldsymbol{c}^{(2)}) + \boldsymbol{c}^{(1)}) + c$$



Shallow vs. Deep Feed-Forward Neural Networks

- **Increasing network depth**

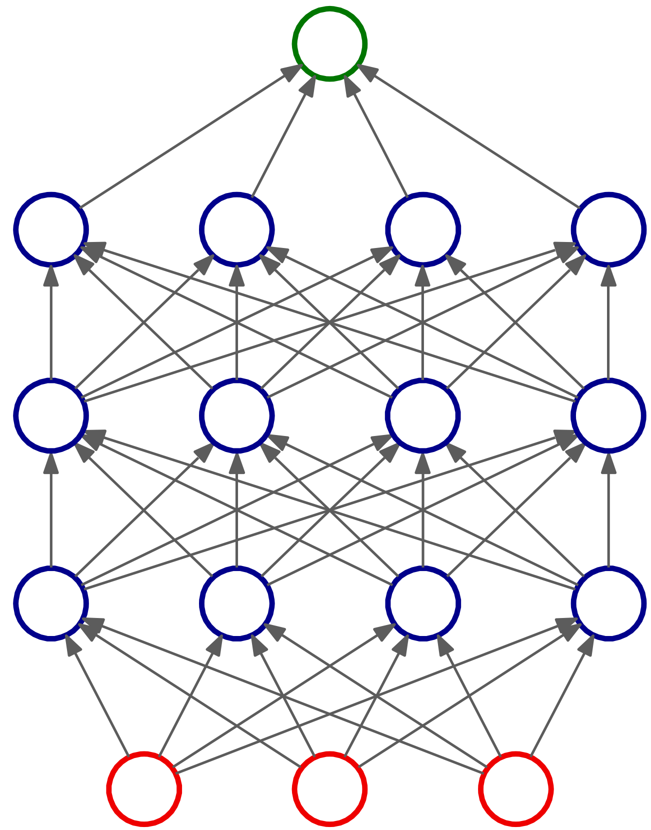
  A feed-forward neural network with three hidden layers

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)} g(\boldsymbol{W}^{(2)} g(\boldsymbol{W}^{(3)} \boldsymbol{x} + \boldsymbol{c}^{(3)}) + \boldsymbol{c}^{(2)}) + \boldsymbol{c}^{(1)}) + c$$

OK, but what is there to gain from such increase in depth?

There are formal results (plus empirical evidence) that depth promotes greater effectiveness of the *hidden* units (in blue)

In other words, using depth you can do more with less blue units



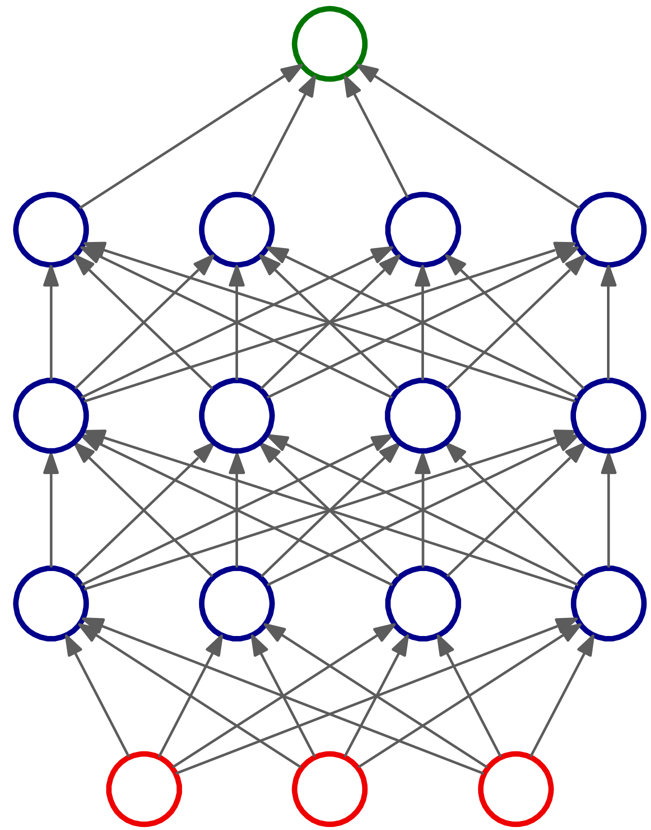
- **Increasing network depth**

  A feed-forward neural network with three hidden layers

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)} g(\boldsymbol{W}^{(2)} g(\boldsymbol{W}^{(3)} \boldsymbol{x} + \boldsymbol{c}^{(3)}) + \boldsymbol{c}^{(2)}) + \boldsymbol{c}^{(1)}) + c$$

  Problem: deeper networks are harder to train

  from examples $D = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(1)})\}_{i=1}^{N}$

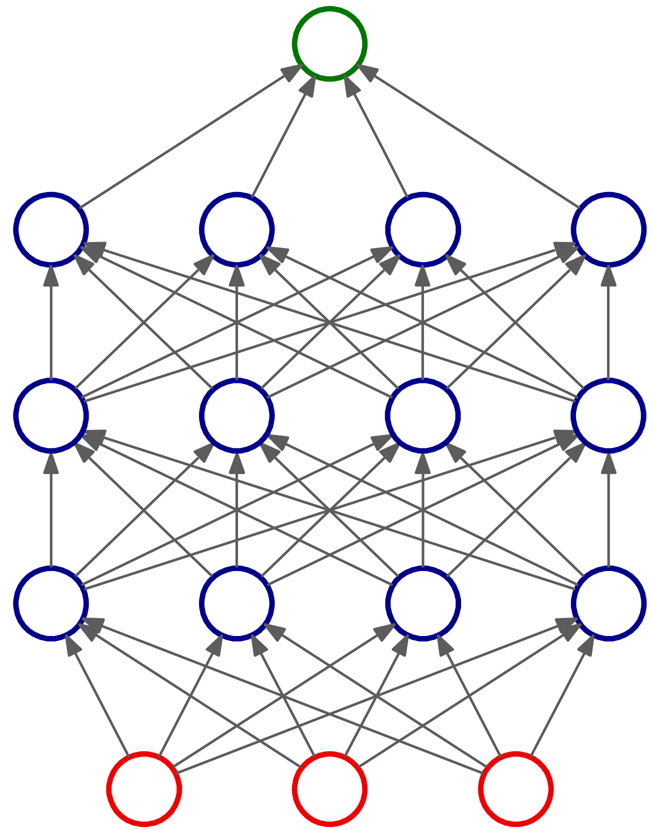


Shallow vs. Deep Feed-Forward Neural Networks

- **Increasing network depth**

A feed-forward neural network with three hidden layers

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{(1)} g(\boldsymbol{W}^{(2)} g(\boldsymbol{W}^{(3)} \boldsymbol{x} + \boldsymbol{c}^{(3)}) + \boldsymbol{c}^{(2)}) + \boldsymbol{c}^{(1)}) + c$$



Problem: deeper networks are harder to train

from *examples* $D = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(1)})\}_{i=1}^{N}$

This is new!

A full bag of formal results and *empirical tricks* have made such training of deep neural networks *feasible*

TensorFlow

Tools like TensorFlow (by Google Inc.) contain lots of such provisions already implemented

Putting things together:
Deep Reinforcement Learning

Deep Reinforcement Learning

- *Q-Learning Algorithm*

Initialize $\hat{Q}(s,a)$ at random, put the agent in a random state s
Repeat:

1) Select the action $\text{argmax}_a \hat{Q}(s,a)$ with probability $(1-\varepsilon)$
otherwise, select a at random
2) The agent is now in state s' and has received the reward r
3) Update $\hat{Q}(s,a)$ by

$$\Delta\hat{Q}(s,a) = \alpha[r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)]$$

***Fundamental Idea**:*
*use a deep neural network to learn the approximator $\hat{Q}(s,a)$ from the examples collected while **exploring** - **exploiting***

Normalized Advantage Function (NAF) algorithm

S. Gu, T. P. Lillicrap, I. Sutskever, S. Levine.
Continuous deep Q-learning
***with model-based acceleration**, 2016*

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s, a|\theta^Q_{PRED})$ $\quad\theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\text{TAR}} \leftarrow \theta^Q_{\text{PRED}}$
Initialize replay buffer $R \leftarrow 0$
for each episode do:
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T do:
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\text{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma\tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial\theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\text{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\text{PRED}} \leftarrow \theta^Q_{\text{PRED}} - \eta\left(\overline{\frac{\partial}{\partial\theta^Q}L(\theta^Q)}\right)$
      $\theta^Q_{\text{TAR}} \leftarrow \tau\theta^Q_{\text{PRED}} + (1+\tau)\theta^Q_{\text{TAR}}$
    end for
  end for
end for

Normalized Advantage Function (NAF) algorithm

Algorithm Highlights

- *a deep neural network for* $\hat{Q}(s,a)$

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s,a|\theta^Q_{PRED})$ $\theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\text{TAR}} \leftarrow \theta^Q_{\text{PRED}}$
Initialize replay buffer $R \leftarrow 0$
for each episode do:
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T do:
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\text{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma\tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial\theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\text{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\text{PRED}} \leftarrow \theta^Q_{\text{PRED}} - \eta\left(\overline{\frac{\partial}{\partial\theta^Q}L(\theta^Q)}\right)$
      $\theta^Q_{\text{TAR}} \leftarrow \tau\theta^Q_{\text{PRED}} + (1+\tau)\theta^Q_{\text{TAR}}$
    end for
  end for
end for

Normalized Advantage Function (NAF) algorithm

Algorithm Highlights

- *a deep neural network for* $\hat{Q}(s,a)$
- *two deep networks:*
  one TARget, which is the objective and one PREDictor for transient approximations

---

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s,a|\theta^Q_{PRED})$ $\qquad \theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\text{TAR}} \leftarrow \theta^Q_{\text{PRED}}$
~~Initialize replay buffer $R \leftarrow 0$~~
for each episode **do:**
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T **do:**
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\text{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma \tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial \theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\text{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\text{PRED}} \leftarrow \theta^Q_{\text{PRED}} - \eta\left(\overline{\frac{\partial}{\partial \theta^Q} L(\theta^Q)}\right)$
      $\theta^Q_{\text{TAR}} \leftarrow \tau\theta^Q_{\text{PRED}} + (1+\tau)\theta^Q_{\text{TAR}}$
    end for
  end for
end for

Normalized Advantage Function (NAF) algorithm

Algorithm Highlights

- *a deep neural network for* $\hat{Q}(s,a)$
- *two deep networks:*
  one TARget, which is the objective and one PREDictor for transient approximations
- *careful* <u>*tensorial*</u> *formulation to avoid the argmax step*

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s,a|\theta^Q_{PRED})$ $\qquad \theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\text{TAR}} \leftarrow \theta^Q_{\text{PRED}}$
Initialize replay buffer $R \leftarrow 0$
for each episode **do**:
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T **do**:
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\text{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma \tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial \theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\text{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\text{PRED}} \leftarrow \theta^Q_{\text{PRED}} - \eta\left(\overline{\frac{\partial}{\partial \theta^Q} L(\theta^Q)}\right)$
      $\theta^Q_{\text{TAR}} \leftarrow \tau\theta^Q_{\text{PRED}} + (1+\tau)\theta^Q_{\text{TAR}}$
    end for
  end for
end for

Normalized Advantage Function (NAF) algorithm

Algorithm Highlights

- *a deep neural network for* $\hat{Q}(s,a)$
- *two deep networks:*
  one TARget, which is the objective and one PREDictor for transient approximations
- *careful <u>tensorial</u> formulation to avoid the argmax step*
- *noise based on a stochastic process (i.e. a random walk, see later) forcing* ***exploration***

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s,a|\theta^Q_{PRED})$ $\qquad \theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\mathrm{TAR}} \leftarrow \theta^Q_{\mathrm{PRED}}$
Initialize replay buffer $R \leftarrow 0$
for each episode do:
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T do:
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\mathrm{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma \tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial \theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\mathrm{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\mathrm{PRED}} \leftarrow \theta^Q_{\mathrm{PRED}} - \eta\left(\overline{\frac{\partial}{\partial \theta^Q} L(\theta^Q)}\right)$
      $\theta^Q_{\mathrm{TAR}} \leftarrow \tau\theta^Q_{\mathrm{PRED}} + (1+\tau)\theta^Q_{\mathrm{TAR}}$
    end for
  end for
end for

Normalized Advantage Function (NAF) algorithm

Algorithm Highlights

- *a deep neural network for* $\hat{Q}(s,a)$
- *two deep networks:*
  one TARget, which is the objective
  and one PREDictor for transient
  approximations
- *careful* *<u>tensorial</u>* *formulation*
  to avoid the argmax step
- *noise based on a stochastic process*
  (i.e. a random walk, see later)
  forcing ***exploration***
- ***replay buffer*** *with random extraction*
  of ***mini-batches*** *to avoid temporal*
  correlation arising from sequential
  exploration

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s,a|\theta^Q_{PRED})$ $\quad\theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\text{TAR}} \leftarrow \theta^Q_{\text{PRED}}$
Initialize replay buffer $R \leftarrow 0$
for each episode **do:**
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T **do:**
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\text{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma \tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial \theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\text{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\text{PRED}} \leftarrow \theta^Q_{\text{PRED}} - \eta\left(\overline{\frac{\partial}{\partial \theta^Q} L(\theta^Q)}\right)$
      $\theta^Q_{\text{TAR}} \leftarrow \tau\theta^Q_{\text{PRED}} + (1+\tau)\theta^Q_{\text{TAR}}$
    end for
  end for
end for

Normalized Advantage Function (NAF) algorithm

Algorithm Highlights

- *a deep neural network for* $\hat{Q}(s, a)$
- *two deep networks:*
  one TARget, which is the objective and one PREDictor for transient approximations
- *careful tensorial formulation to avoid the argmax step*
- *noise based on a stochastic process (i.e. a random walk, see later) forcing* ***exploration***
- ***replay buffer*** *with random extraction of* ***mini-batches*** *to avoid temporal correlation arising from sequential exploration*
- ***no need to discretize*** $\mathcal{A}$ ***and*** $\mathcal{S}$

Algorithm 1.2 NAF algorithm for continuous Q-learning

Randomly initialize $\tilde{Q}(s, a|\theta^Q_{PRED})$ $\qquad \theta^Q := (\theta^\mu, \theta^P, \theta^V)$
Initialize the target network with $\theta^Q_{\mathrm{TAR}} \leftarrow \theta^Q_{\mathrm{PRED}}$
Initialize replay buffer $R \leftarrow 0$
for each episode **do**:
  Initialize random process $\mathcal{N}$ for action exploration
  $s_0 \leftarrow Environment(reset)$
  for $t = 0$ to T **do**:
    $a_t \leftarrow \mu(s_t|\theta^\mu_{\mathrm{PRED}}) + \mathcal{N}_t$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \leftarrow Environment(s_t, a_t)$
    $RB \leftarrow RB \cup \{(s_t, a_t, r_t, s_{t+1})\}$store transition in the replay buffer
    Sample at random and normalize the mini batch MB
    for each sample $i = (s_i, a_i, r_i, s_{i+1})$ in m
      $y_i = r_i + \gamma \tilde{V}(s_{i+1}|\theta^V_{TAR})$
      Compute gradients
        $\frac{\partial}{\partial \theta^Q}\left(y_i - Q\left(s_i, a_i|\theta^Q_{\mathrm{PRED}}\right)\right)^2$ (Loss function $L(\theta^Q)$)
      $\theta^Q_{\mathrm{PRED}} \leftarrow \theta^Q_{\mathrm{PRED}} - \eta\left(\overline{\frac{\partial}{\partial \theta^Q} L(\theta^Q)}\right)$
      $\theta^Q_{\mathrm{TAR}} \leftarrow \tau\theta^Q_{\mathrm{PRED}} + (1 + \tau)\theta^Q_{\mathrm{TAR}}$
    end for
  end for
end for

Smart Inventory Management (thanks to ProfumeriaWeb)

ProfumeriaWeb

Inventory Management: the environment

- ***Environment description*** (Simplified for this experiment)
  - the *agent* is the e-commerce company, as a whole
  - the agent has an *inventory*, where *products* are stored
  - keeping products in the inventory has a *cost*
    (yearly estimate: 18% of overall product cost)
  - the website can only sell products that are in the inventory — *This is NOT true in reality*
  - sales occur on a daily basis
  - products can be obtained by the agent via requests to *suppliers*
  - there exist different suppliers,
    some are more expensive others are cheaper, also delivery times may differ
  - suppliers have their own inventories and they serve multiple buyers

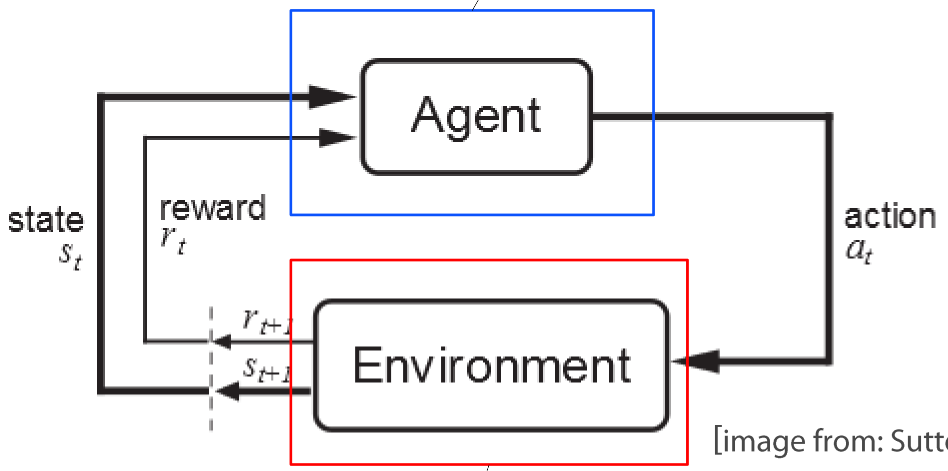
Goal

Manage the inventory to maximize *marginality* (i.e. revenues – total costs)

Inventory Management: experiments

- ***Implementation***

NAF algorithm
implemented with TensorFlow



[image from: Sutton, Barto, *Reinforcement Learning.* 1998]

(Simplified) Market Simulator
built in python and Numpy
using stochastic processes

An aside: *stochastic processes for market simulations*

- ***Random walk: Ornstein-Uhlebneck (OU) stochastic process***

quantity that
varies over time (e.g. a *price*)

volatility

$$\Delta q^{(t)} = a(\bar{q} - q^{(t-1)})\Delta t + \sigma \Delta W^{(t)}$$

mean reverting rate

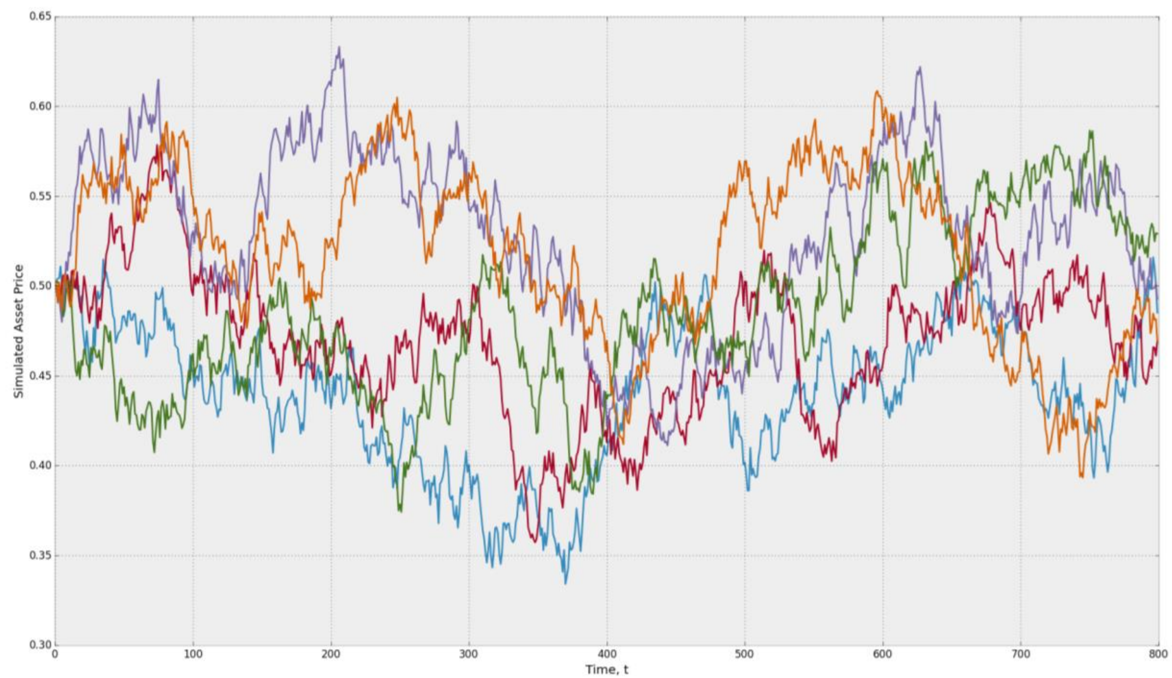
mean value

time interval

brownian motion (i.e. Wiener process)

A popular choice for market simulations:

- it is a *random walk*
- it is *mean reverting*
- its is *fully controllable* (via its parameters)



[image from: http://www.turingfinance.com/random-walks-down-wall-street-stochastic-processes-in-python/]

Inventory Management: basic assumptions

- **Suppliers**

  Product *average market cost* (AMC): mean reverting random walk (OU)

  Product *cost* (*per supplier, per product*): $\mathrm{AMC} + \mathcal{N}(\delta, 0.1)$

  gaussian, supplier-specific cost delta
  (delta negative => the supplier is cheaper)

  Product *availability* (*per supplier*): mean reverting random walk (OU)

- **Requests** (per product, per supplier)

  Limited to product availability (*per supplier*)

  Competing model (*with other buyers*): requests will accepted with binomial probability

  Delivery times: Poisson stochastic process
  with supplier-specific *lambda* parameter (e.g. different geographic distance)

- **Sales**

  Product average market price $\mathrm{AMP} := 1.65\ \mathrm{AMP}$

  Selling price (P, *per product*): $\mathrm{P} := 1.65\ \mathrm{AIC}$ — *(daily) average inventory cost*

  Sales potential (SP, *per product*): maximum theoretical amount of sales (fixed a priori)

  Actual sales potential (*per product*): $\mathrm{ASP} := \sigma(\mathrm{AMP} - \mathrm{P})\ \mathrm{SP}$

Inventory Management: simulation scenario


```
"Products": {
  "Product1": {
    "initial_cost": 30,
    "sales_potential": 12
  },
  "Product2": {
    "initial_cost": 50,
    "sales_potential": 8
  }
},
"Suppliers": {
  "Supplier1": {
    "delivery_time": 5,
    "products": {
      "Product1": {
        "initial_availability": 20,
        "initial_cost": 29,
        "average_cost_delta": -1.0
      },
      "Product2": {
        "initial_availability": 15,
        "initial_cost": 48,
        "average_cost_delta": -5.0
      }
    }
  },
  "Supplier2": {
    "delivery_time": 2,
    "products": {
      "Product1": {
        "initial_availability": 100,
        "initial_cost": 32,
        "average_cost_delta": 2.0
      },
      "Product2": {
        "initial_availability": 100,
        "initial_cost": 54,
        "average_cost_delta": 4.0
      }
    }
  }
```

Inventory Management: daily routine

1. **Determine sales** (environment)

   Determine agent sales (previous day)
   Update agent inventory
   Compute agent daily marginality

2. **Requests** (agent)

   Based on current *state* (see after)
   determine agent product requests to each supplier

3. **Prepare orders** (environment)

   Each supplier receives agent product requests and resolve competition (i.e. binomial)

   Orders are enqueued for later delivery

   Update product availability (per product, per supplier)

4. **Order delivery** (environment)

   Dequeue orders that have been delivered to the agent

   Update agent inventory

Inventory Management: deep reinforcement learning

- **State**

  Daily sales (*per product*): quantity, price

  Agent inventory (*per product*): quantity, average inventory cost

  Supplier (*per supplier, per product*): availability, cost

- **Action**

  Product request (*per supplier, per product*): quantity

- **Reward**

  Daily marginality (*per product, due to sales*): $\mathrm{DM} := \mathit{quantity}\,(\mathit{price} - \mathrm{AIC})$
  Total daily marginality (TDM): sum of daily marginality per product

  Daily inventory cost (*per product*): $(0.18/365)\ \mathrm{AIC}$
  Total daily inventory cost (TAIC): sum of average inventory cost of each product

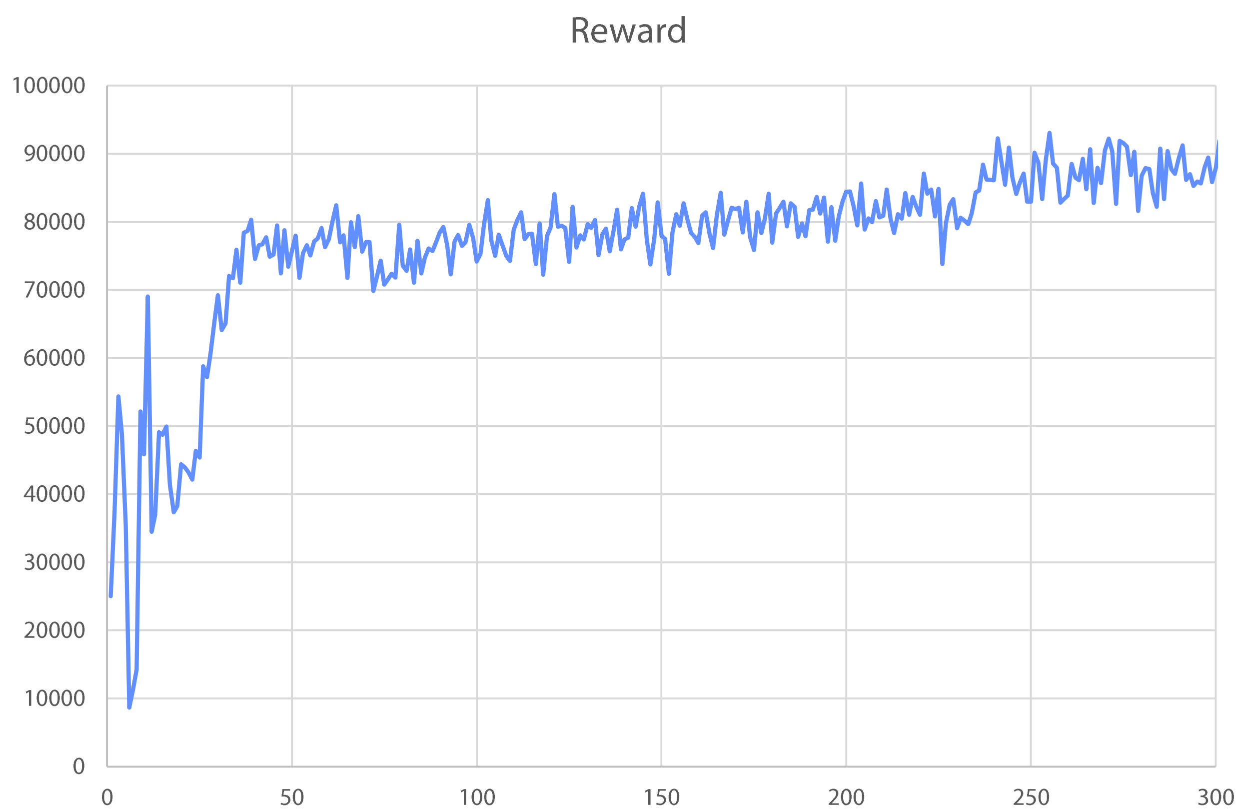
  Action size (AS): norm of request quantities, seen as a vector

$$\mathit{reward} := \mathrm{TDM} - \mathrm{TDIC} - c\ \mathrm{AS}$$

a constant, regularization factor

Inventory Management: experiments

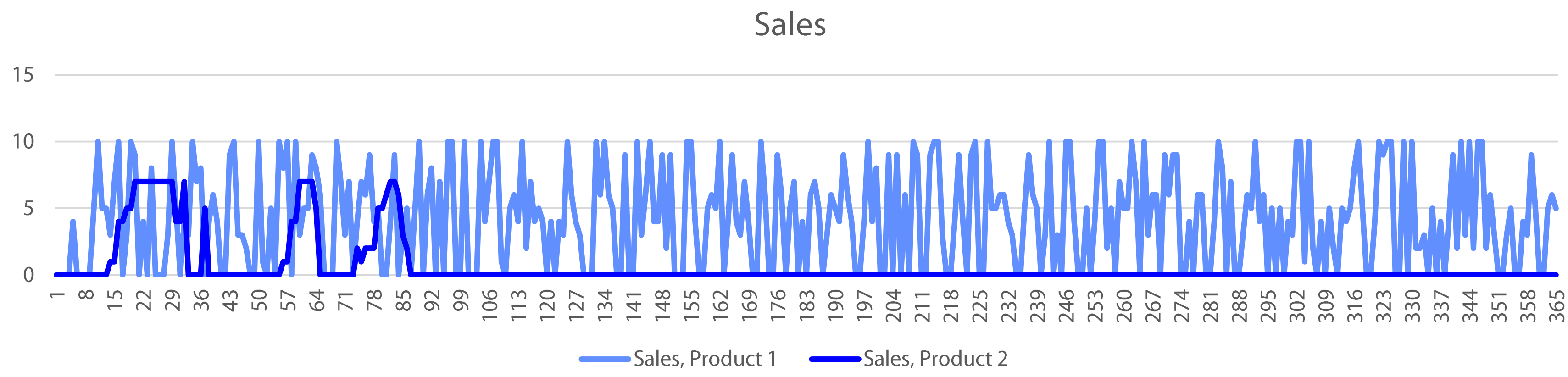
(Very preliminary results)

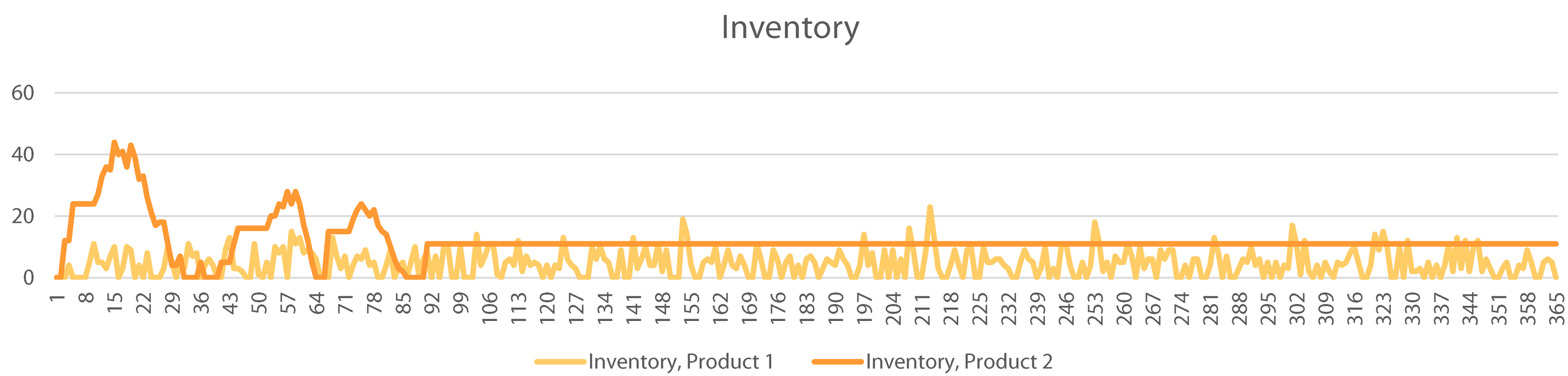


Each training episode: 1 year, i.e. 365 daily iterations

Inventory Management: experiments

Sales and inventory - after 10 episodes

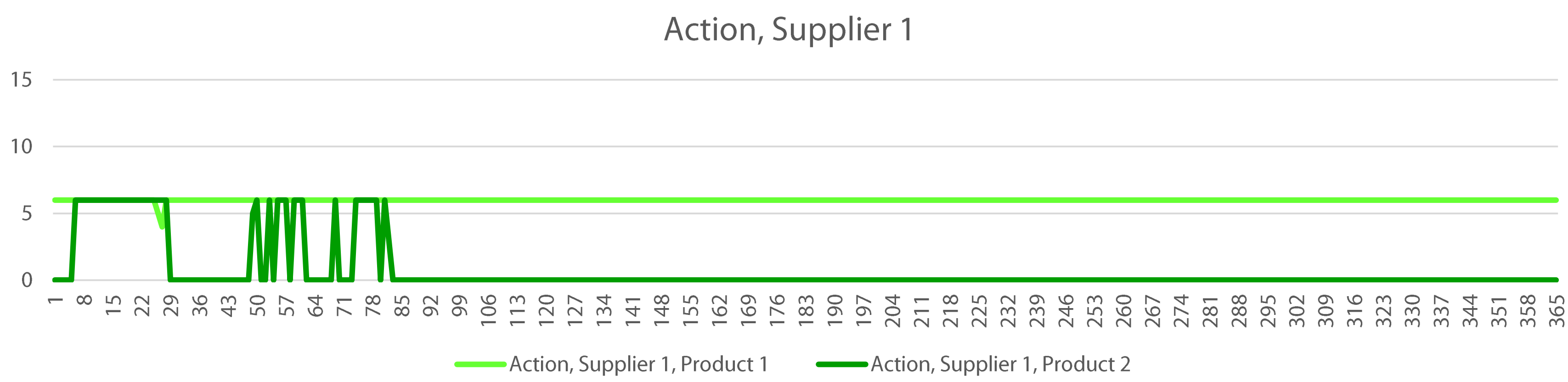


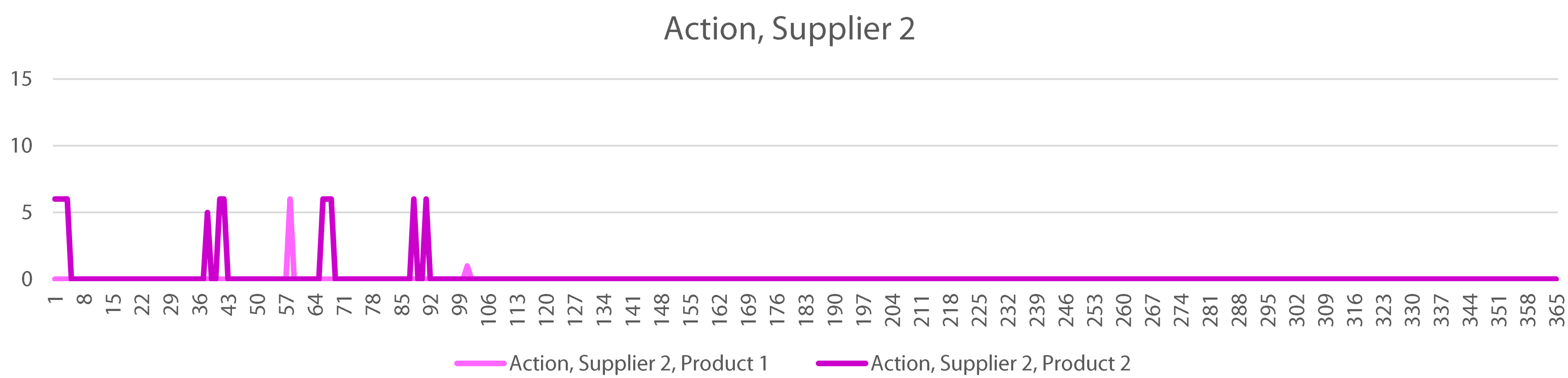


One training episode: 1 year, i.e. 365 daily iterations

Inventory Management: experiments

Agent actions - after 10 episodes

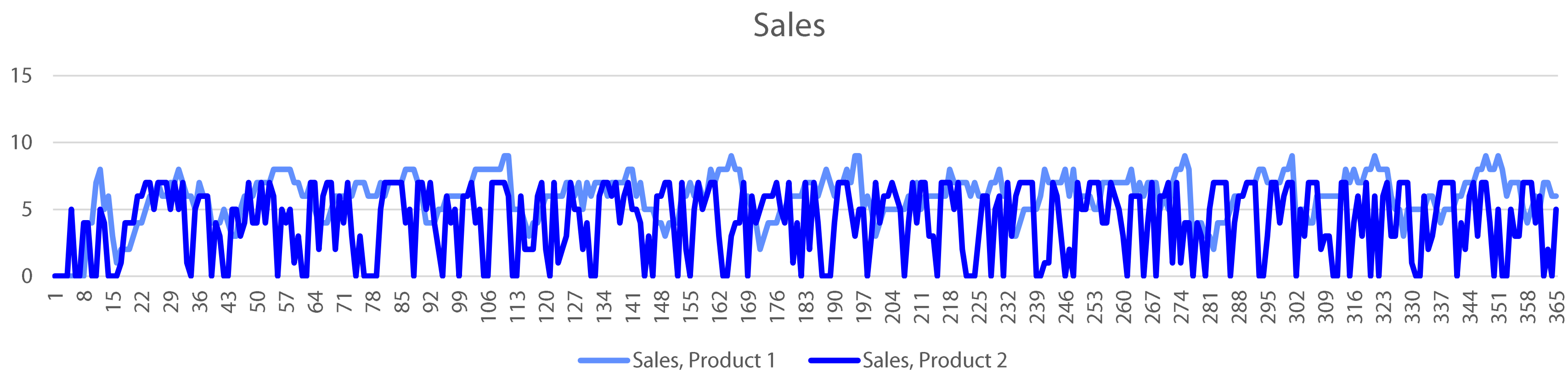


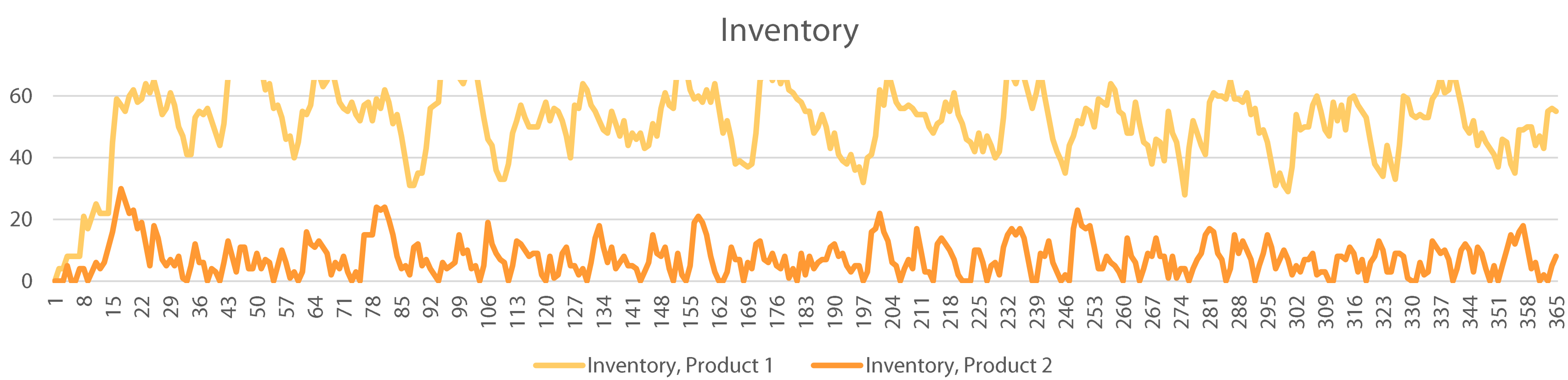


One training episode: 1 year, i.e. 365 daily iterations

Inventory Management: experiments

Sales and inventory - after 70 episodes

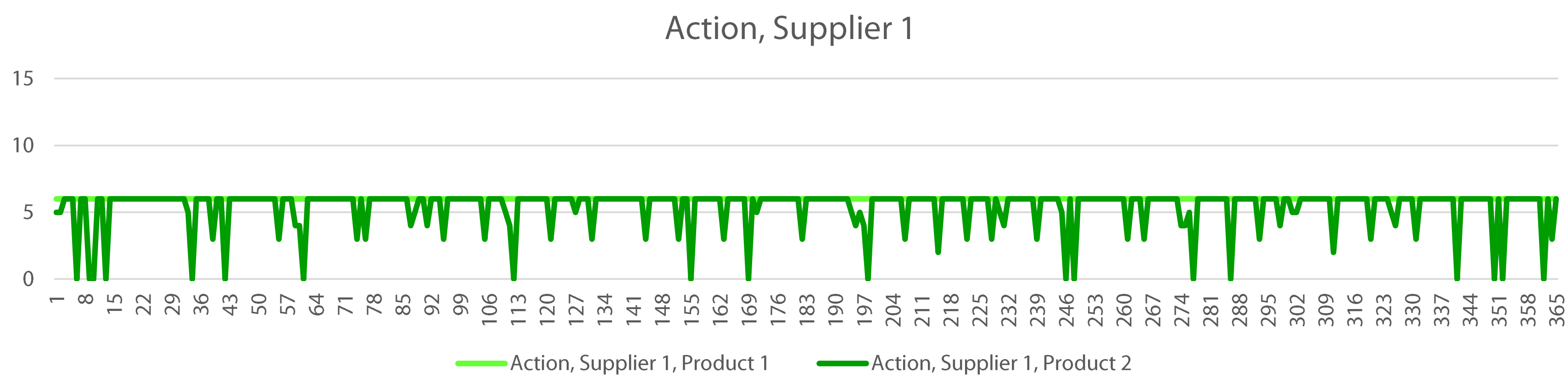


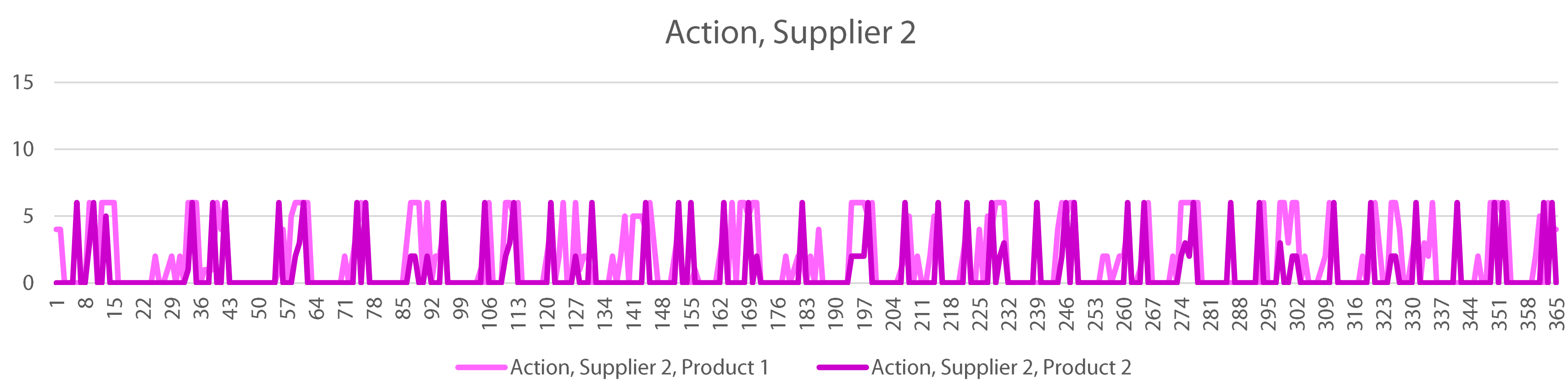


One training episode: 1 year, i.e. 365 daily iterations

Inventory Management: experiments

Agent actions - after 70 episodes

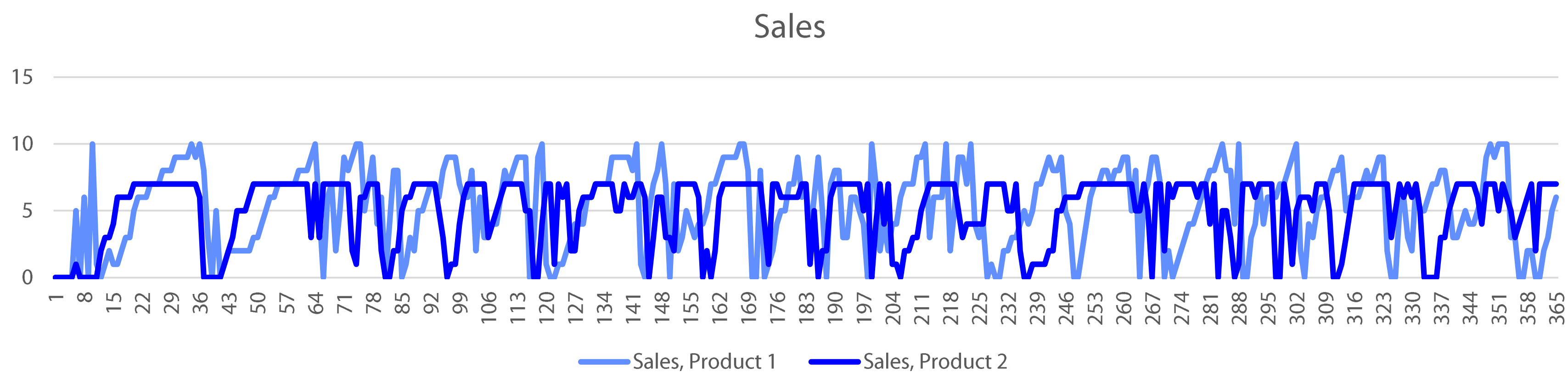


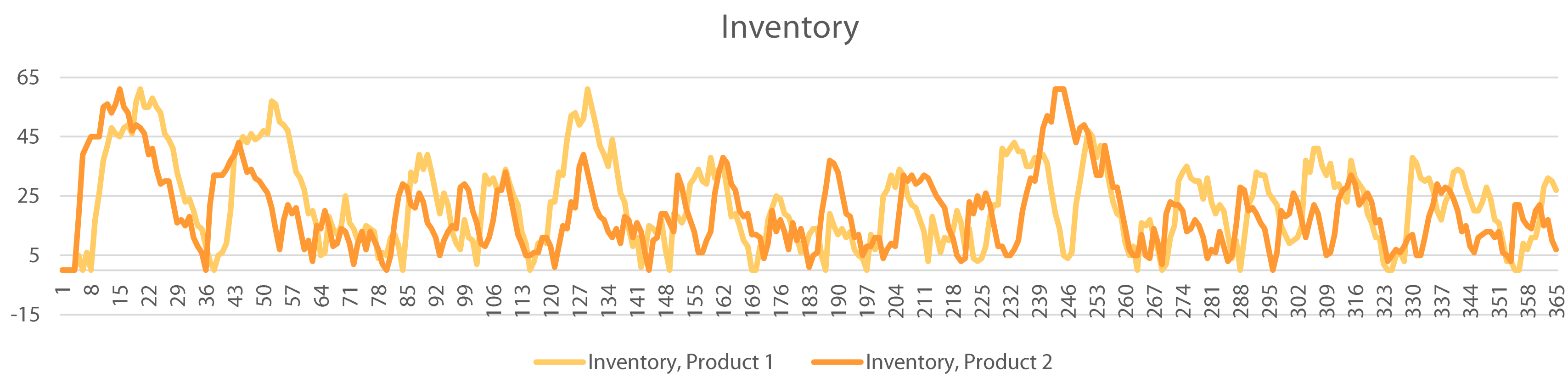


One training episode: 1 year, i.e. 365 daily iterations

Inventory Management: experiments

Sales and inventory - after 300 episodes

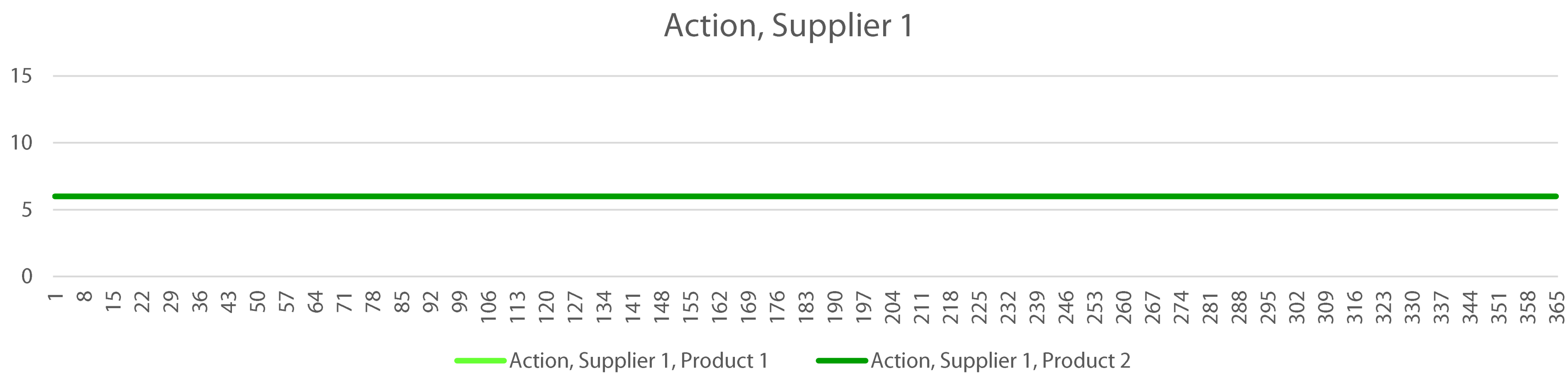


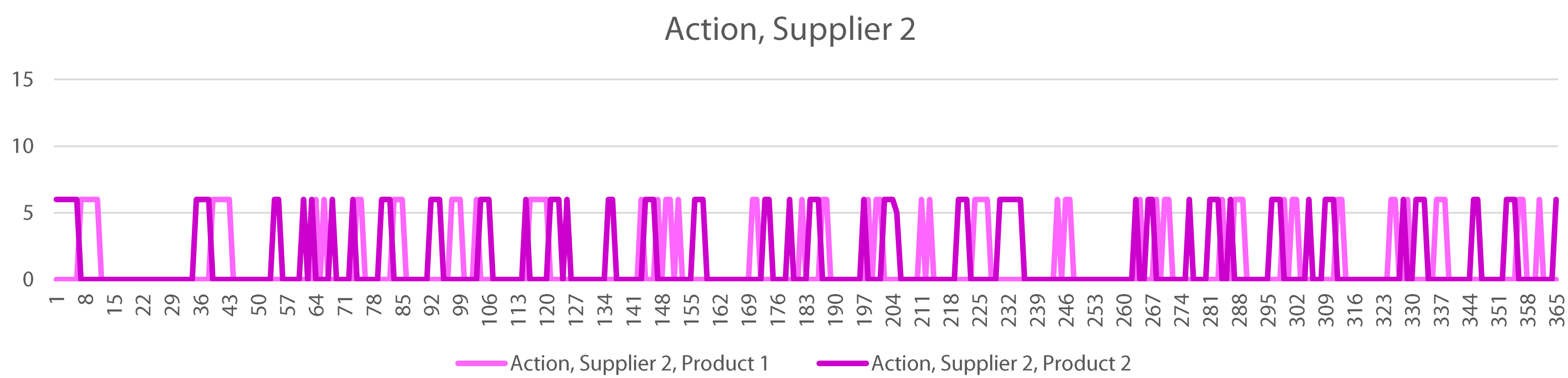


One training episode: 1 year, i.e. 365 daily iterations

Inventory Management: experiments

Agent actions - after 300 episodes





One training episode: 1 year, i.e. 365 daily iterations