Mid-Term Hydro-Scheduling Problem: The Battle Between Stochastic Dynamic Programming and Reinforcement Learning

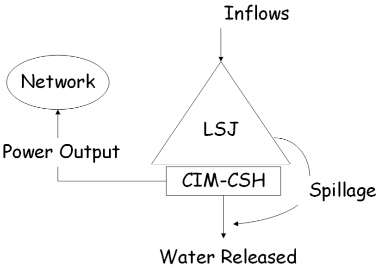
Pascal Côté [1], Richard Arsenault [2], Quentin Desreumaux [3]

[1]Power Operation, Rio Tinto Aluminum Saguenay, Québec, Canada. pascal.cote@riotinto.com
[2]École de technologie supérieure, Montréal Canada. richard.arsenault@etsmtl.ca.
[3]Université de Sherbrooke, Sherbrooke Canada. qd@terminal.io

September 12, 2018

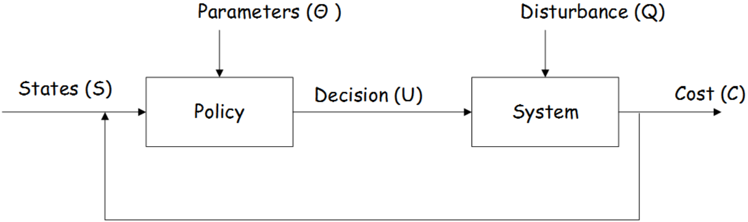
Optimization Problem I

- Minimize the expected cost
  - Energy imports
  - Reservoir level constraints
- Subject to
  - Mass balance equation
  - Energy demands
  - Recreational reservoir limits



Optimization Problem II

- We are looking for a feedback policy



(**S**) Reservoir storage, hydrological state of the watershed, ...

(**U**) Water released, spillage, imported energy, ...

(**Q**) Natural inflows to the resevoir, energy demand, ...

(**C**) Composite objective function

(Θ) Parameters of the policy

Optimization Problem III

Policy should be found such as the following expected cost function is minimized:

$$\min J(\Theta) = \mathrm{E}\left[\sum_{t=0}^{T} \alpha_t C(u_t, s_t, q_t)\right]$$

$$s_{t+1} = G(u_t, s_t, q_t)$$

$$u_t = \pi_\Theta(s_t)$$

Solution Methods:

- Dynamic programming methods (explicit)
  - Stochastic Dynamic Programming [1] [2] [3]
  - Approximate SDP for larger systems [4] [5]
- Direct policy search methods (implicit)
  - Black-box optimization [6], [7]
- Other types of methods (among others)
  - Extended Linear Quadratic Gaussian algorithm [8]
  - Optimal Trajectory Approach [9]

Stochastic Dynamic Programming

The SDP feedback policy is given by:

$$\pi_{\theta_t}(s_t) = \underset{u_t}{\mathrm{argmin}} \left\{ \underset{q_t}{\mathrm{E}} \left[\alpha_t C\left(u_t, s_t, q_t\right) + \alpha_{t+1} F_{t+1}^{(\theta_{t+1})}\left(s_{t+1}\right) \right] \right\}$$

where

$$\Theta = [\theta_0, \theta_1, \ldots, \theta_T]$$

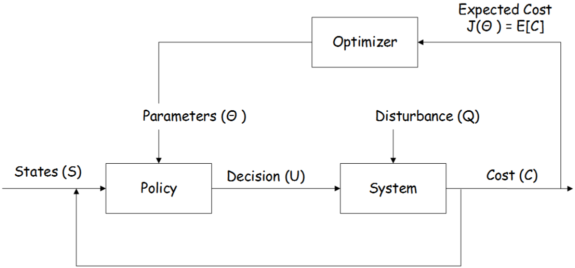
$$F_t^{(\theta_t)} \approx F_t\left(s_t\right) = \min_{u_t} \left\{ \underset{q_t}{\mathrm{E}} \left[C\left(u_t, s_t, q_t\right) + \alpha_{t+1} F_{t+1}^{(\theta_{t+1})}\left(s_{t+1}\right) \right] \right\}$$

$F_t^{(\theta_t)}$ can be estimated by

- Lookup table
- Piecewise linear approximation [10]
- Multivariate spline [11], [12]
- Neural network [13],[14]

Direct Policy Search Methods

- Parameters of the policy are directly optimized by "black box" optimization where the objective function is estimated by simulating the policy.
- Universal approximator is used to model the policy (neural network)



Direct Policy Search Methods

Many algorithms can be used to solve this black-box optimization problem (to name only a few)

- Differential Evolution [15]
- Covariance Matrix Adaptation Evolution Strategy [16]
- Particle Swarm [17]
- **Mesh Adaptive Direct Search** [18]
  - Efficient and easy to use NOMAD software [19]
  - Proof of convergence, parallel computation, handles constraints

Reinforce Algorithm I

- If the policy is continuously differentiable w.r.t its parameters Θ, the Reinforce algorithm[20] can be used to estimate the gradient of the objective function.
- The objective function can be rewritten using the expected cumulative cost $\mathbb{C}$ over all possible trajectories:

$$J(\Theta) = \mathrm{E}\left[\sum_{t=0}^{T} \alpha_t C(u_t, s_t, q_t)\right]$$

$$J(\Theta) = \int_{\mathbb{T}} p_{\Theta}(\tau)\, \mathbb{C}(\tau)\, d\tau$$

$$\nabla_{\Theta} J(\Theta) = \int_{\mathbb{T}} \nabla_{\Theta} p_{\Theta}(\tau)\, \mathbb{C}(\tau)\, d\tau$$

$$\nabla_{\Theta} J(\Theta) = \int_{\mathbb{T}} p_{\Theta}(\tau)\, \nabla_{\Theta} \log p_{\Theta}(\tau)\, \mathbb{C}(\tau)\, d\tau$$

Reinforce Algorithm II

Rio Tinto

- By using a stochastic policy $u_t \sim \pi_\Theta(u_t \mid s_t)$, we have that:

$$p_\Theta(\tau) = p(s0) \prod_{t=0}^{T} p(s_{t+1} \mid s_t, u_t)\, \pi_\Theta(u_t \mid s_t)$$

As only the policy depends on Θ we have :

$$\nabla_\Theta \log p_\Theta(\tau) = \log \left(p(s0) \prod_{t=0}^{T} p(s_{t+1} \mid s_t, u_t)\, \pi_\Theta(u_t \mid s_t) \right)$$

$$\nabla_\Theta \log p_\Theta(\tau) = \sum_{t=0}^{T} \nabla_\Theta \log \pi_\Theta(u_t \mid s_t)$$

- In the case of random gaussian stochastic policy, i.e. $\pi_\Theta(u_t \mid s_t) \sim \mathcal{N}\left(\mu = \Phi_\Theta(s_t), \sigma^2\right)$, where $\Phi_\Theta(s_t)$ is a policy approximator, with the derivative chain rule we have:

$$\nabla_\Theta \log \pi_\Theta(u_t \mid s_t) = \nabla_\Theta \log \mathcal{N}(\;) = \nabla_\Theta \Phi_\Theta(s_t) \left(\frac{\pi_\Theta(u_t \mid s_t) - \mu}{(\sigma^2)^2} \right)$$

Reinforce Algorithm III

- We finally have the following results:

$$\nabla_\Theta J(\Theta) = \int_{\mathbb{T}} p_\Theta(\tau) \nabla_\Theta \log p_\Theta(\tau) \mathbb{C}(\tau) \, d\tau$$
$$\nabla_\Theta J(\Theta) = \mathrm{E}\left[\nabla_\Theta \log p_\Theta(\tau) \mathbb{C}(\tau)\right]$$

- The expected gradient can be estimated be simulating the following equation with many sequences, where a constant baseline is inserted to reduce the variance of the approximator [21]:

$$\nabla_\Theta J(\Theta) = \sum_{t=0}^{T} \nabla_\Theta \log \pi_\Theta(u_t \mid s_t) (\mathbb{C}(\tau) - b)$$

- The cost of the deterministic policy can be used as a baseline [22]:

$$\mathbb{C}_\pi(\tau) = \textstyle\sum_{t=0}^{T} \alpha_t C(u_t, s_t, q_t), \; u_t \sim \pi_\Theta(u_t \mid s_t)$$
$$\mathbb{C}_\Phi(\tau) = \textstyle\sum_{t=0}^{T} \alpha_t C(u_t, s_t, q_t), \; u_t = \Phi_\Theta(s_t)$$
$$\nabla_\Theta J(\Theta) = \textstyle\sum_{t=0}^{T} \nabla_\Theta \log \pi_\Theta(u_t \mid s_t) (\mathbb{C}_\pi(\tau) - \mathbb{C}_\Phi(\tau))$$

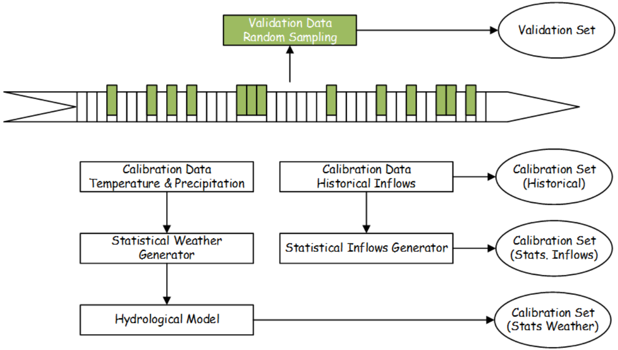
Reinforce Algorithm IV

N : Number of sequences to compute the expected gradient
V_Ψ : Approximator for water value function (critic)
$\mathcal{A}$: Heuristic for step descent (we suggest Adam method [23])

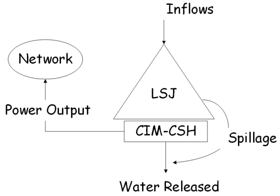
while *Until termination condition* **do**
  for *i=1,2, . . ., N* **do**
    Randomly select initial state $s_{i,0}$
    Stochastic Pass:
    $R_i^\pi = \sum_{t=0}^T \alpha_t C\left(u_t, s_t, q_t\right) + \alpha_{T+1} V_\Psi\left(s_{T+1}\right)\right)$ with $u_t \sim \pi_\Theta(s_t)$
    $g_i = \sum_{t=0}^T \nabla_\Theta \log \pi_\Theta(u_t \mid s_t)$
    Deterministic Pass:
    $R_i^d = \sum_{t=0}^T \alpha_t C\left(u_t, s_t, q_t\right) + \alpha_{T+1} V_\Psi\left(s_{T+1}\right)\right)$ with $u_t = \Phi_\Theta(s_t)$
  end
  Actor Update:
  $\nabla_\Theta J\left(\Theta\right) = \frac{1}{N} \sum_{i=1}^N g_i \left(R_i^s - R_i^d\right)$
  $\Theta \leftarrow \mathcal{A}\left(\Theta, \nabla_\Theta J\left(\Theta\right)\right)$
  Critic Update:
  $L\left(\Psi\right) = \frac{1}{N} \sum_{i=1}^N R_i^d - V_\Psi\left(s_{i,0}\right)$
  $\Psi \leftarrow \mathcal{A}\left(\Psi, \nabla_\Psi L\left(\Psi\right)\right)$
end

Synthetic Inflow Generation

- Many inflow sequences must be used to find a robust policy [6], [22]
- Synthetic inflow generation
  - Stochastic inflow generator with SAMS method [24]
  - Stochastic weather generator with KNNCad v4 [25] to drive the GR4J hydrological model [26]



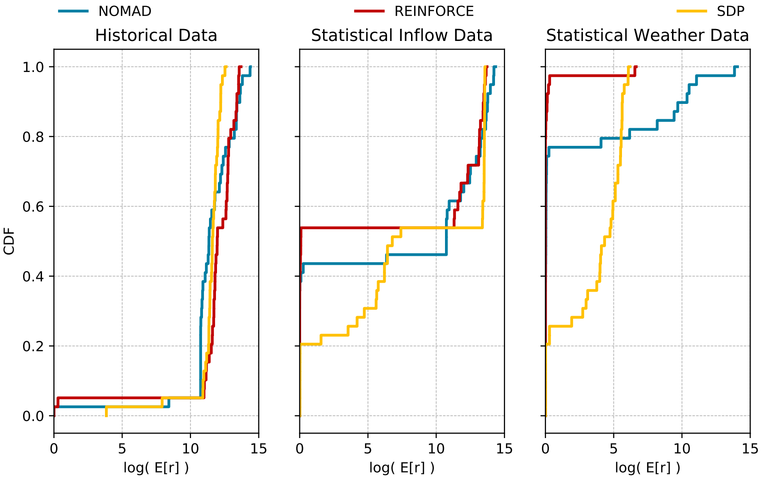
Numerical Tests I



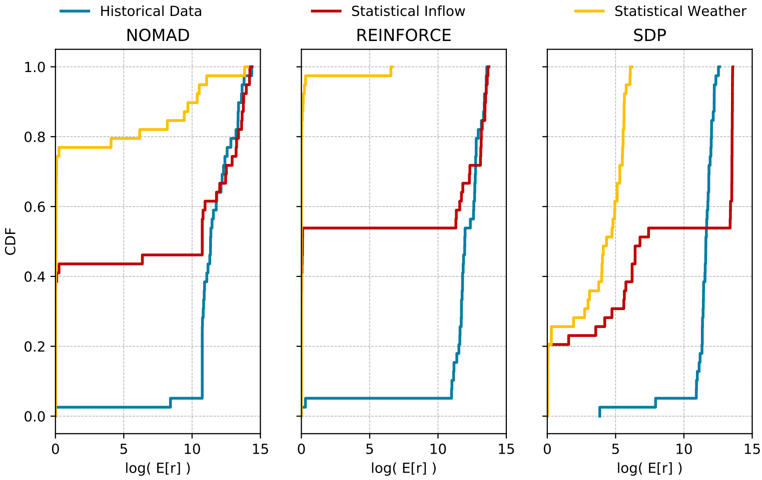
- South part of the Saguenay Lac St-Jean Hydropower system
  - 1 reservoir of 5000 hm^3 live storage
  - 2 power houses
  - Installed capacity of 2000 MW
  - 860 m^3/sec avg. annual inflow
- Historical Data
  - 60 years of natural infows
  - 25 years for validation
  - 35 years for calibration
  - Randomly create 40 sets of 5000 sequences
- SDP method
  - 100 discretization points for storage
  - 2 state variables (storage and hydrological variable)
- DPS
  - 1000 iterations
  - Neural neural with 1 hidden layer of 20 nodes

Optimization Method Results

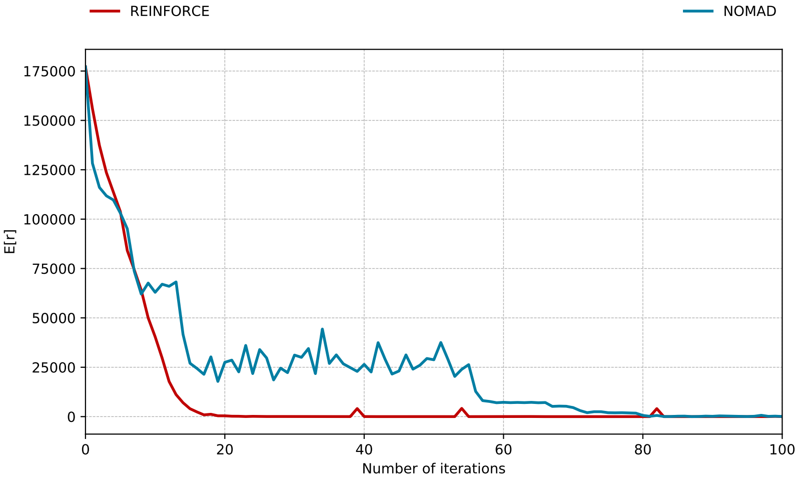
Rio Tinto



Inflow Data Generation Results I

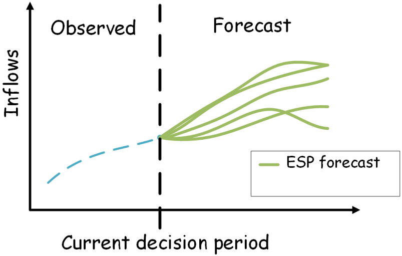


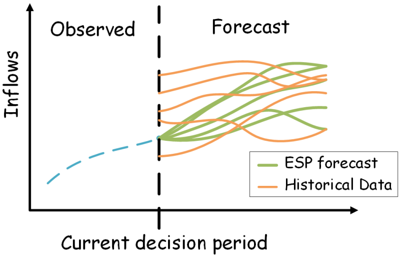
Convergence Rate



Discussion

- RL and DPS outperformed SDP for the SLSJ hydropower system in this study.
- Requires a lot of sequences to perform the training.
- RL is a more complex and "*black box*" method for operation engineers.
- How to implement RL in day-to-day operations with updated forecasts?





References I

[1] P. Côté and R. Leconte, "Comparison of Stochastic Optimization Algorithms for Hydropower Reservoir Operation with Ensemble Streamflow Prediction," *Journal of Water Resources Planning and Management*, vol. 142, no. 2, 2016.

[2] C. Davidsen, S. J. Pereira-cardenal, D Ph, S. Liu, X. Mo, D. Rosbjerg, and P. Bauer-gottwein, "Using Stochastic Dynamic Programming to Support Water Resources Management in the Ziya River Basin , China," *Journal of Water Resources Planning and Management*, vol. 141, no. 7, 2014. DOI: 10.1061/(ASCE)WR.1943-5452.0000482..

[3] A. Turgeon, "Solving a stochastic reservoir management problem with multilag autocorrelated inflows," *Water Resources Research*, vol. 41, no. 12, 2005. DOI: 10.1029/2004WR003846.

[4] M. N. Hjelmeland, J Zou, A Helseth, and S Ahmed, "Nonconvex Medium-Term Hydropower Scheduling by Stochastic Dual Dynamic Integer Programming," *IEEE Transactions on Sustainable Energy*, 2018. DOI: 10.1109/TSTE.2018.2805164.

[5] J. Pina, A. Tilmant, and P. Côté, "Optimizing Multireservoir System Operating Policies Using Exogenous Hydrologic Variables," *Water Resources Research*, vol. 53, no. 11, 2017. DOI: 10.1002/2017WR021701.

[6] Q. Desreumaux, P. Cote, and L. Robert, "Comparing Model-Based and Model-Free Streamflow Simulation Approaches to Improve Hydropower Reservoir Operations," *Journal of Water Resources Planning and Management*, vol. 144, 2018. DOI: 10.1061/(ASCE)WR.1943-5452.0000860.

[7] A. Castelletti, F. Pianosi, X. Quach, and R. Soncini-Sessa, "Assessing water reservoirs management and development in northern Vietnam," *Hydrology and Earth System Sciences*, vol. 16, no. 1, 2012. DOI: 10.5194/hess-16-189-2012.

[8] A. P. Georgakakos, "Extended linear quadratic Gaussian control: Further extensions," *Water Resources Research*, vol. 25, no. 2, 2018. DOI: 10.1029/WR025i002p00191.

[9] A. Turgeon, "Stochastic optimization of multireservoir operation: The optimal reservoir trajectory approach," *Water Resources Research*, vol. 43, no. 5, 2007. DOI: 10.1029/2005WR004619.

[10] Q. Goor, R. Kelman, and A. Tilmant, "Optimal Multipurpose-Multireservoir Operation Model with Variable Productivity of Hydropower Plants," *Journal of Water Resources Planning and Management*, vol. 137, no. 3, 2011. DOI: 10.1061/(ASCE)WR.1943-5452.0000117.

[11] S. A. Johnson, J. R. Stedinger, C. A. Shoemaker, Y. Li, and J. A. Tejada-Guibert, "Numerical Solution of Continuous-State Dynamic Programs Using Linear and Spline Interpolation," *Operations Research*, vol. 41, no. 3, 1993. DOI: 10.1287/opre.41.3.484.

[12] V. C. P. Chen, D. Ruppert, and C. A. Shoemaker, "Applying Experimental Design and Regression Splines to High-Dimensional Continuous-State Stochastic Dynamic Programming," *Operations Research*, vol. 47, no. 1, 1999. DOI: 10.1287/opre.47.1.38.

[13] A. Castelletti, D. de Rigo, A. E. Rizzoli, R. Soncini-Sessa, and E. Weber, "Neuro-dynamic programming for designing water reservoir network management policies," *Control Engineering Practice*, vol. 15, no. 8, 2007. DOI: 10.1016/j.conengprac.2006.02.011.

[14] C. Cervellera, A. Wen, and V. C. Chen, "Neural network and regression spline value function approximations for stochastic dynamic programming," *Computers and Operations Research*, vol. 34, no. 1, 2007. DOI: 10.1016/j.cor.2005.02.043.

[15] R. Storn and K. Price, "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, vol. 11, no. 4, 1997. DOI: 10.1023/A:1008202821328.

References II

[16] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evolutionary Computation*, vol. 9, no. 2, Jun. 2001.

[17] J Kennedy and R Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995.

[18] C Audet and J Dennis, "Mesh Adaptive Direct Search Algorithms for Constrained Optimization," *SIAM Journal on Optimization*, vol. 17, no. 1, 2006. DOI: `10.1137/040603371`.

[19] S. Le Digabel, "Algorithm 909: Nomad: Nonlinear optimization with the mads algorithm," *ACM Trans. Math. Softw.*, vol. 37, no. 4, 2011. DOI: `10.1145/1916461.1916468`.

[20] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, 1992.

[21] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, no. 4, 2008. DOI: `10.1016/j.neunet.2008.02.003`.

[22] Q. Desreumaux, "Amélioration de la représentation des processus stochastiques pour l'optimisation appliquée à la gestion des systèmes hydriques," PhD thesis, Université de Sherbrooke, 2016.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[24] O. G. Sveinsson, J. D. Salas, W. L. Lane, and D. K. Frevert, "Stochastic analysis, modeling, and simulation (sams) version 2007, user's manual," *Computing Hydrology Laboratory, Department of Civil and Environmental Engineering, Colorado State University, Fort Collins, Colorado*, 2007.

[25] L. M. King, A. I. McLeod, and S. P. Simonovic, "Improved weather generator algorithm for multisite simulation of precipitation and temperature," *JAWRA Journal of the American Water Resources Association*, vol. 51, no. 5, pp. 1305–1320, 2015.

[26] L Coron, G. Thirel, O Delaigue, C. Perrin, and V. Andréassian, "The suite of lumped gr hydrological models in an r package," *Environmental Modelling & Software*, vol. 94, pp. 166–171, 2017.

Python code (use at your own risk !) I

```
import pickle
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras import backend as k
import tensorflow as tf
from scipy.stats import norm, uniform

# Reward function
def reward(storage, inflow, release, t):
    # Update state
    end_storage = storage + delta * (inflow - release)
    # Check if release is feasible
    z = 0.0
    if end_storage > smax:
        z += 0.0005*(end_storage - smax)**2.0
        release += (end_storage - smax) / delta
        end_storage = storage + delta * (inflow - release)
    if end_storage < 0.0:
        z += 0.0005*(end_storage)**2.0
        release += (end_storage-1.e-3) / delta
        end_storage = storage + delta * (inflow - release)
    if release > rmax:  # Spillage
        mw = mwmax - 0.02 * (release - rmax)**1.3
    else:               # No Spillage
        mw = (release)*0.35 * (0.1*(storage)/sref)**0.5
    return (-(0.99)**float(t) * mw / 1000.0 + z, end_storage)
```

Python code II

```
# Simulate a sequence
def simulate(ini_storage, inflows, var, stochastic, actor):
    # To cumulate the reward
    R = 0.0
    # To cumulate the gradient
    G = [np.zeros_like(a) for a in actor.get_weights()]
    # Start simulation
    storage = ini_storage
    # Number of period
    T = inflows.size
    nsin = np.sin(np.array(range(T)) / float(T-1) * 2 * np.pi)
    ncos = np.cos(np.array(range(T)) / float(T-1) * 2 * np.pi)
    for t in range(T):
        # Get input for critic
        x = np.array([[inflows[t]/Qmax[t], storage/smax,
                       ncos[t], nsin[t]]])
        # Get the output of the network
        yd = actor.predict(x)[0,0]
        # Add noise
        if stochastic:
            ys = norm.rvs(loc=yd, scale=var)
        else:
            ys = yd
        # De-Normalization
        release = max(0.0, ys*rmax)
        (r, storage) = reward(storage, inflows[t], release, t)
        # Cumulate information
        R += r
        if stochastic:
            g = sess.run(gradients,feed_dict={actor.input:x})
                        G = [G[i] + ((ys-yd)/var**2 * g[i]) for i in range(len(g))]
        else:
            G = None
    return (R, G, storage)
```

Python code III

```
if __name__ == "__main__":
    # Load inflow sequences and compute stats for normalization
    (Qcalib, Qvalid) = pickle.load(open('Inflows.data','rb'))
    Qmax = Qcalib.max(axis=0)

    # Parameters for reward function
    (smax, rmax, mwmax, sref, delta) = (5000., 2500.0, 1800.0, 3000.0, 0.6048)

    # Create network for Actor
    (anInput, anHidden) = (4, 20)
    actor = Sequential()
    actor.add(Dense(anHidden, input_dim=anInput,
                    activation='tanh', init='uniform'))
    actor.add(Dense(1, activation='sigmoid', init='uniform'))
    actor.compile(loss='mean_squared_error',
                  optimizer='adam', metrics=['accuracy'])

    # To compute the gradient w.r.t. output
    outputTensor = actor.output
    listOfVariableTensors = actor.trainable_weights
    gradients = k.gradients(outputTensor, listOfVariableTensors)
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())

    # Create network for Critic
    cnInput = 1
    cnHidden = 20
    critic = Sequential()
    critic.add(Dense(cnHidden, input_dim=cnInput,
                     activation='relu', init='uniform'))
    critic.add(Dense(1, activation='linear', init='uniform'))
    critic.compile(loss='mean_squared_error',
                   optimizer='adam', metrics=['accuracy'])
```

Python code IV

```
# Parameters for Adam update formula
beta1 = 0.4
beta2 = 0.7
eps = 1.e-8
alpha = 0.1
disc_end = (0.99)**float(Qvalid.shape[1])

# Initial storage for simulation
s0Valid = 0.85*smax

# Number of sequence per iteration
N = 100
# Number of iteration
MaxIter = 100
# Initial variance
#  NOTE : This paramter can be optimized in the same way
#         as the weights of the network are optimized.
var = 0.1
# to store the weights
Wold = actor.get_weights()
Wnew = [np.zeros_like(w) for w in Wold]
Mg = [np.zeros_like(w) for w in Wold]
Vg = [np.zeros_like(w) for w in Wold]
grad = [np.zeros_like(w) for w in Wold]
step = [np.zeros_like(w) for w in Wold]
# Rewards (stochastic and deterministic pass)
Rs = np.zeros((N, 1))
Rd = np.zeros((N, 1))
# To update the critic
s0 = np.zeros((N,1))
```

Python code V

```
#
# Loop on each outer iteration
for j in range(MaxIter):

    #======================================================================
    # Do printing
    if j % 10.0 == 0.0:
        print "Iter        E[ R() ]         || g(J) ||      alpha" +\
              "           || step ||     Valid: E[ R() ]"
    # Perform a simulation over validation set
    (Rvalid, sini) = (0.0, s0Valid)
    for k in range(Qvalid.shape[0]):
        Q = Qvalid[k,:]
        (R, _, sini) = simulate(sini, Q, var, False, actor)
        Rvalid += R
    Rvalid /= float(Qvalid.shape[0])
    fmt = "%4d %15.7f %15.7f %15.7f %15.7f %15.7f"
    print fmt % (j, Rs.mean(), sum([np.linalg.norm(g) for g in grad]), alpha,
                                 sum([np.linalg.norm(s) for s in step]), Rvalid)
    #======================================================================
    # To store gradient
    g = [None] * N
    #
    # Simulate each sequences
    iseq = np.random.permutation(Qcalib.shape[0]-1)
    for k in range(N):
        s0[k,0] = uniform.rvs() * smax
        i = iseq[k]
        Q = Qcalib[i, :]
        # Stochastic pass
        (Rs[k, 0], g[k], sT) = simulate(s0[k,0], Q, var, True, actor)
        # Add critic
        Rs[k, 0] += disc_end*critic.predict(np.array([[sT/smax]]))[0,0]
        # Deterministic pass
        (Rd[k, 0], _, sT) = simulate(s0[k,0], Q, var, False, actor)
        # Add critic
        Rd[k, 0] += disc_end*critic.predict(np.array([[sT/smax]]))[0,0]
```

Python code VI

RioTinto

```
# Update the critic
critic.fit(s0/smax, Rd, batch_size=10, epochs=100, verbose=0)
#
# Expected gradient
grad = [np.zeros_like(w) for w in Wold]
for k in range(N):
    for i in range(len(grad)):
        grad[i] += g[k][i] * (Rs[k,0] - Rd[k,0])
grad = [g / float(N) for g in grad]
#
# Update weights with  Adam
Mg = [beta1*m + (1.0-beta1)*g for (m,g) in zip(Mg, grad)]
Vg =[beta2*v + (1.0-beta2)*g**2 for (v,g) in zip(Vg, grad)]
alpha = alpha*np.sqrt(1.0 - beta2**(j+1.0))
alpha /= (1.0 - beta1**(j+1.0))
step = [m/(np.sqrt(v) + eps) for (m,v) in zip(Mg,Vg)]
Wnew = [w - alpha*s for (w, s) in zip(Wold, step)]
actor.set_weights(Wnew)
Wold = [w for w in Wnew]
```

Python code output

```
Iter      E[ R() ]        || g(J) ||       alpha          || step ||     Valid: E[ R() ]
   1   13235.6422075    6620.7401123    0.0912871     20.6913638     8274.0557306
   2   10886.2509498   13714.1228027    0.0776096     20.1195977     7394.6909440
   3    7114.4585637    5315.5012207    0.0672082     12.7249644     6470.1754653
   4    7327.4838608   10749.0970459    0.0601261     15.3354113     4981.3557104
   5    5740.6599404    6342.7301025    0.0554085     13.5942354     4463.4299631
   6    6549.4258533    1967.5065536    0.0522612      8.7417132     4361.7644330
   7    6199.3866973   13925.8970947    0.0501452     14.6356510     4555.0299688
   8    5194.8458706    5908.9888306    0.0487103      5.4383829     4442.8724605
   9    5669.7952595    3181.6757812    0.0477299      4.1133877     4271.2831183
Iter      E[ R() ]        || g(J) ||       alpha          || step ||     Valid: E[ R() ]
  10    5436.6479017    2435.9600983    0.0470558      4.0149955     4243.0616360
  11    6628.2210682    2331.7276611    0.0465902      5.0736826     4141.0821997
  12    6310.2819751    3661.7816162    0.0462675      7.9634422     4037.0038614
  13    6614.8746142    4420.9333496    0.0460431      8.6875266     3846.5559875
[...]
Iter      E[ R() ]        || g(J) ||       alpha          || step ||     Valid: E[ R() ]
  50    7769.0894629    1403.5239983    0.0455247      7.3211062     4000.6439511
  51    9281.7242786   14915.5718079    0.0455247     12.9929575     2966.6926542
  52    8305.2400483    6395.8134804    0.0455247     10.2150637     2521.9583363
  53    7239.6641027    4375.2785721    0.0455247      9.7960255     2628.4929596
  54    8108.4698861   12411.9685974    0.0455247     11.5797659     2677.8480926
[...]
  95    8336.9839646    7207.6943054    0.0455247     12.0878678     2455.6306705
  96    8274.7471423    5084.1004944    0.0455247     12.0743954     2621.1963913
  97    8813.3507141   13633.3601074    0.0455247     18.8322048     2872.9041490
  98    8443.7042156    2772.4992523    0.0455247      8.1704949     3024.1396766
  99    7582.1613727    3571.3728905    0.0455247      9.4253207     2992.7194769
```