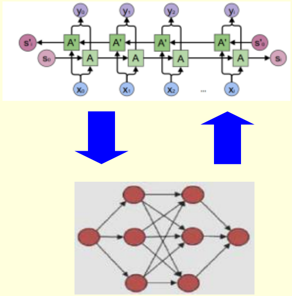
Deep Learning by Example on Biowulf

Class #5. Deep Reinforcement Learning Networks and their application to *de novo* drug molecule design

Gennady Denisov, PhD

G



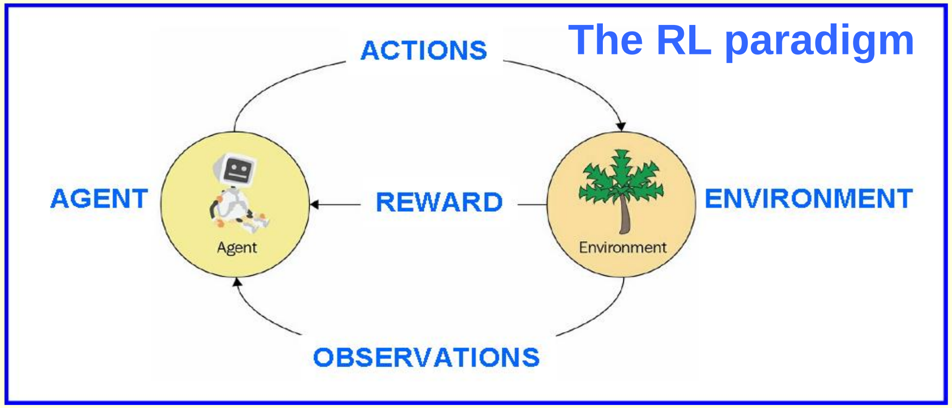
P

Intro and goals

agent, environment, observations, actions, rewards

What is Reinforcement Learning (RL)?

- a framework for <u>decision making</u> ("observe and act")
- two entities: **AGENT** and **ENVIRONMENT**
- the **AGENT** receives **OBSERVATIONS,** based on which it executes **ACTIONS,** and, in response to them, receives **REWARDS**
- the **ENVIRONMENT** receives **ACTIONS** and emits **OBSERVATION** and **REWARDS**



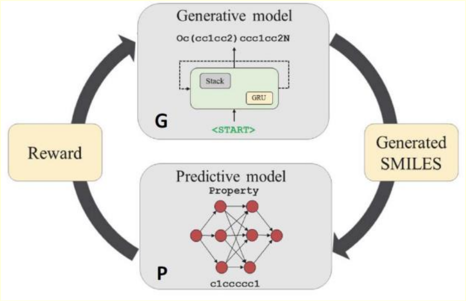
Goal:

- determine the **ACTION(S)** ("<u>decision</u>") that will maximize an expected <u>**cumulative REWARD**</u>

Examples:

Playing ATARI game with Deep RL

ReLeaSE: Deep RL for *de novo* drug design



Generator network ≈ AGENT
Predictor network ≈ ENVIRONMENT

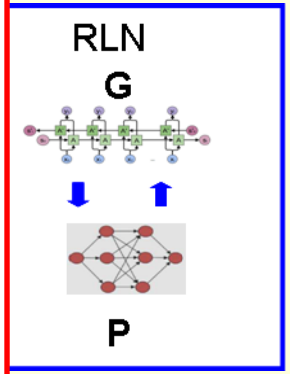
SMILES (Simplified Molecular-Input Line-Entry Specification) string:

N1CCN(CC1)C(C(F)=C2)=CC(=C2C4=O)N(C3CC3)C=C4C(=O)O

Examples overview

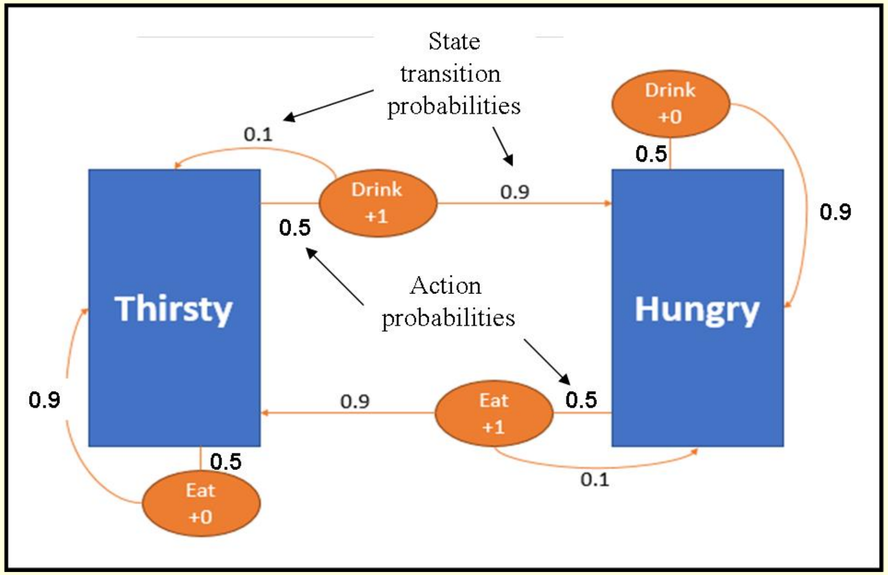
#	1	2	3	4	5
Biological Appliation	Bioimage segmentation/ fly brain connectome project	Genomics/ predicting the function of non-coding DNA	Genomics/ classification of cancer types based on gene expression	Bioimage synthesis / developmental biology	Drug molecule design
Network type	Convolutional Neural Network	Recurrent Neural Network	Auto-encoder	Generative Adversarial Network	Reinforcement Learning Network
ML type	Supervised	Supervised	Unsupervised	Unsupervised	Reinforcement

- RL = 3rd camp of methods:
  SL: requires a ground truth with static/fixed labels/targets;
  UL: no ground truth and predefined/supervised labels
  RL: labels/targets are adjusted dynamically, based on rewards
- DRL is a challenging topic; marries RL to DL
  RL => learning objective, DL => mechanism
  will illustrate with 3 simple/prototype examples



Value-based RL: the simplest (tabular) Q-learning example

state, policy, return, discount rate, episode, state-action value (Q-)function, learning rate



Input: an agent that emulates a **newborn child**

Actions:
{Eat, Drink }

Rewards →

Rewards		Actions: Eat	Actions: Drink
States	Hungry	1	0
States	Thirsty	0	1

States (≈Observations):
{Hungry, Thursty}

Task: determine the best **deterministic policy**

π: $a = \pi(s)$

that will **maximize** an **expected** future **return**

return G_t =cumulative **discounted** ($0 < \delta \leq 1$) future reward over the duration of an **episode**

State-action value function:
(= the learning objective)

$$Q(s_t, a_t) = \mathrm{E}\,[\overbrace{r_t + \delta \cdot r_{t+1} + \delta^2 \cdot r_{t+2} + \ldots}\,] \rightarrow \max_{\pi}$$

Bellman equation:
(employs dynamic programming)

$$newQ(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot [\, r_t + \delta \cdot \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t)]$$

α: learning rate ($0 \leq \alpha \leq 1$)
r_t: reward
δ: discount rate ($0 < \delta \leq 1$)

Q-table		Actions: Eat	Actions: Drink
States	Hungry	$Q(s_H, a_E)$	$Q(s_H, a_D)$
States	Thirsty	$Q(s_T, a_E)$	$Q(s_T, a_D)$

A code for the tabular Q-learning example

denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro

Initial Q-table		Actions	
		Eat	Drink
States	Hungry	0	0
	Thirsty	0	0

Header

Set params

Define a model

Run the model

(length of each episode = 1)

```
#!/usr/bin/env python
import random
import numpy as np

alpha, delta, num_episodes = 0.001, 0.995, 50

Agent = {"Actions"  : ['E', 'D'], \
         "Q_table"  : {'H' : {'E' : 0, 'D' : 0}, 'T' : {'E' : 0, 'D' : 0}},
         "Policy"   : {}}
Env   = {"States"   : ['H', 'T'],
         "Rewards"  : {'H' : {'E' : 1, 'D' : 0}, 'T' : {'E' : 0, 'D' : 1}},\
         "Probs"    : {'H' : {'E' : [('H', 0.1), ('T', 0.9)], \
                              'D' : [('H', 0.9), ('T', 0.1)]},\
                       'T' : {'E' : [('H', 0.1), ('T', 0.9)], \
                              'D' : [('H', 0.9), ('T', 0.1)]}}}
for e in range(num_episodes):
    state = random.choice(Env["States"])
    action_to_take  = random.choice(Agent["Actions"])    # sample actions
    all_next_states = [t[0] for t in Env["Probs"][state][action_to_take]]
    all_next_probs  = [t[1] for t in Env["Probs"][state][action_to_take]]
    next_state      = np.random.choice(all_next_states, 1, p=all_next_probs)[0]
    Agent["Q_table"][state][action_to_take] = \
          Agent["Q_table"][state][action_to_take] \
        + alpha * ( Env["Rewards"][state][action_to_take] \
                  + delta * max(Agent["Q_table"][next_state].values()) \
                        - Agent["Q_table"][state][action_to_take])
    print("e=%d/%d state=%s Q=[%.7f, %.7f]" % \
          (e+1, num_episodes, state, Agent["Q_table"][state]['D'], \
                        Agent["Q_table"][state]['E']))
    for s in Env["States"]:
        Agent["Policy"][s] = \
            Agent["Actions"][np.argmax(list(Agent["Q_table"][s].values()))]
    print("        Policy=", Agent["Policy"])
                                                              33,54
```

Each episode is of length = 1

Updating the Q-table

Updating the best policy

Deep Q-learning: a prototype sequence optimization example

Original study: MolDQN, Zhou et al. Nature Sci. Reports (2019)

Input:
state: a sequence of 0's and 1's
action: random substitution of a character at a random position
a target **motif** sequence, e.g. 0011
reward: the diff. between the LA scores after and before an action.

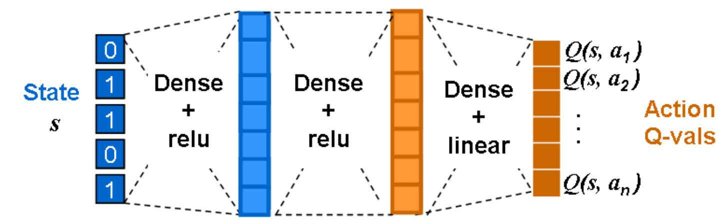
Task:
infer the best **deterministic policy** π: $a = \pi(s)$ that will **produce an optimal sequence** (= containing the motif) from any sequence

Q-table		Actions	
		Eat	Drink
States	Hungry	$Q(s_H, a_E)$	$Q(s_H, a_D)$
	Thirsty	$Q(s_T, a_E)$	$Q(s_T, a_D)$

One layer transformation:
Y = A(W·**X** + **b**)

A

Deep Q-learning network (DQN) for sequence optimization

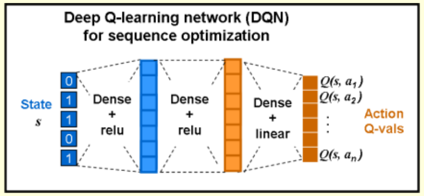


denisovga@biowulf:/usr/local/apps/DLBio/class5/bin

```
#!/usr/bin/env python
import numpy as np, random, copy, re
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from Bio import Align

num_episodes,slen,alpha,delta,lr,motif = 20000,7,0.1,0.995,1.e-5,"0011"
States  = ['0'*(slen+2-len(bin(s)))+bin(s)[2:] for s in range(int('1'*slen,2)+1)]
Actions = [('0',i) for i in range(slen)]+[('1',j) for j in range(slen)]
Policy_enum = {}

model = Sequential()
model.add(Dense(len(States), activation='relu', input_shape=(slen,)))
model.add(Dense(len(States), activation='relu'))
model.add(Dense(len(Actions), activation='linear'))
print(model.summary())
model.compile(loss='mse', optimizer=Adam(lr=lr))

A = Align.PairwiseAligner()
A.alphabet,A.match,A.mismatch,A.open_gap_score,A.extend_gap_score,A.mode=('01',1,-4,-2,-1,'local')
def get_reward(state, next_state, motif, A):
    return  max([a1.score for a1 in A.align(next_state, motif)]) \
          - max([a1.score for a1 in A.align(     state, motif)])
def preprocess(state):
    return np.reshape([2.*(float(k)-1.)+1. for k in state], [1,len(state)])
```

2^{slen} states
2*slen actions

Local alignment (LA):
0101110 – sequence
|-|||
0-011 – motif



A prototype sequence optimization example (cont.)

denisovga@biowulf:/usr/local/apps/DLBio/class5/bin

```
popul, tot = 0, pow(2,slen)*2*slen
for i in range(num_episodes):
    state = States[np.random.randint(0, len(States))]
    action = (c,pos) = Actions[np.random.randint(0, len(Actions))]
    if not state in Policy_enum.keys():
        Policy_enum[state] = [action]
    elif not action in Policy_enum[state]:
        Policy_enum[state].append(action)
    next_state = list(copy.deepcopy(state))
    next_state[pos] = c
    next_state = "".join(next_state)
    reward = get_reward(state, next_state, motif, A)
    # Training
    Qtarget = model.predict(preprocess(state))
    pos = action[1] if action[0]=='0' else action[1]+len(state)
    Qtarget[0][pos]=Qtarget[0][pos] + alpha*(reward + \
        delta*np.amax(model.predict(preprocess(next_state)))-Qtarget[0][pos])
    model.train_on_batch(preprocess(state), Qtarget)
    ###
    popul = sum([len(v) for v in Policy_enum.values()])
    if i > 0 and i % 100 == 0:
        print("i=%d/%d, policy_slots_unpopulated=%d/%d" % \
               (i,num_episodes,tot-popul,tot))
count = 0
for state in States:
    Qtarget = model.predict(preprocess(state))[0]
    c, pos = "0", np.random.choice(np.where(Qtarget == np.max(Qtarget))[0])
    if pos >= slen:
        c, pos = "1", pos - slen
    next_state = list(copy.deepcopy(state))
    next_state[pos] = c
    reward = get_reward(state,"".join(next_state),motif,A)
    ok,count = ("ok",count+1) if reward>=0 else ("",count)
    print("Best_policy[",state,"]=(",c,",",pos,")"," motif=", motif,
          " next_state=", "".join(next_state), " reward=", reward, ok)
print("success count=%d/%d policy_slots_unpopulated=%d/%d" % \
   (count,len(States),tot-popul,tot))
                                                              64,67
```

Bellman equation:

$$newQ(s, a_i) = Q(s, a_i) + \alpha \cdot [r_t + \delta \cdot max_a\, Q(s_{next}, a) - Q(s, a_i)]$$

Training algorithm:
(1) **predict** Q-targets using current network weights
(2) **update** the Q-targets with Bellman equation
(3) **re-train** the network against the updated Q-targets

Each episode is of length = 1

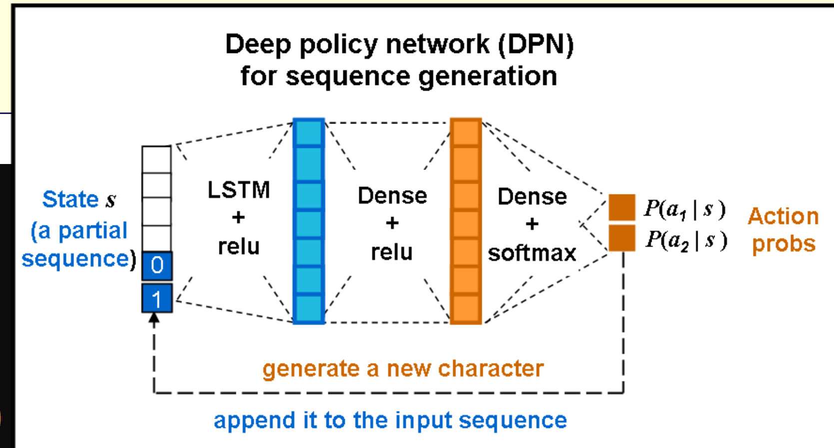
Limitations of Q-learning:

- **instability:** small variations in Q-vals may dramatically change the best policy
- **low performance** for large #states

Policy-based deep RL: a prototype *de novo* sequence generation example

Input:
state: a "partial" sequence of 0's and 1's
action: appending a random character (0 or 1) at the end of the sequence
a target **motif** sequence, e.g. 0011
reward: the difference between the LA scores after and before an action; $\boldsymbol{r \geq 0}$

Task:
infer the best **probabilistic policy** $\boldsymbol{\pi} = P\,(a \mid s)$ that will allow **generation of an optimal sequence** (i.e. containing a predefined motif) **from scratch**



denisovga@biowulf:/usr/local/apps/DLBio/class5/bin

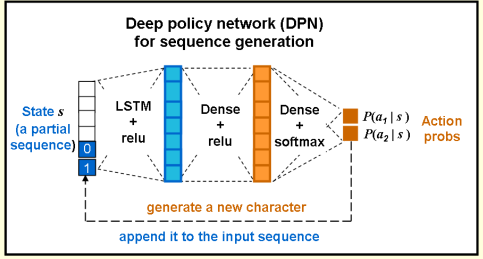
```python
#!/usr/bin/env python
import numpy as np, collections, copy, math, re
from keras.models import Sequential
from keras.layers import LSTM, Dense, Softmax
from keras.optimizers import Adam
from Bio import Align
A = Align.PairwiseAligner()
A.alphabet,A.match,A.mismatch,A.open_gap_score,\
    A.extend_gap_score,A.mode= ('01',1,-4,-2,-1,'local')

num_episodes, slen, alpha, motif, baseline = 1500, 5, 0.001, "0011", []
np.random.seed(7)

model = Sequential()
model.add(LSTM(slen, input_shape=(slen,1),activation='relu'))
model.add(Dense(slen, activation='relu'))
model.add(Dense(2, activation='softmax'))
print(model.summary())
model.compile(loss='mse', optimizer=Adam(lr=1.e-3))

def tostr(state):
    tochar = lambda s: '0' if s < 0 else '1'
    return "".join([tochar(s) for s in state]) if len(str(state)) > 1 \
                                                else tochar(s)
def get_reward(state, next_state, motif, A):
    return  max([al.score for al in A.align(tostr(next_state), motif)]) \
          - max([al.score for al in A.align(tostr(     state), motif)])

                                                   28,0-1        Top
```

Addressing the limitations of Q-learning:

(a) **stability:** small variations in $P(a|s)$ will not affect the (random) actions dramatically

(b) **efficiency:** a policy gradient algorithm does not require exploring all possible states

Local alignment (LA):
```
0101110   – sequence
|-|||
0-011         – motif
```



A code for the *de novo* sequence generation example (cont.)

baselined reward, REINFORCE algorithm

REINFORCE: R.J.Williams, Machine Learning (1992)

REINFORCE: **RE**ward **I**ncrement = **N**onnegative **F**actor x **O**ffset **R**einforcement x **C**haracteristic **E**ligibility

```
def update_state(state, model, pos, slen):
    prob = np.array(model.predict(state.reshape([1,slen,
                                           1]))).flatten())
    action = np.random.choice([0,1],1,p=prob)[0]
    state[pos] = -1. if action == 0 else 1.
    return (state, prob, action)

for e in range(num_episodes+1):
    states,probs,pseudo_gradients,rewards = [],[],[],[]
    state, pos, reward = np.zeros([slen,]), 0, 0
    while pos < slen:
        state0 = copy.deepcopy(state)
        states.append(state0.reshape([slen,1]))
        state, prob, action = \
            update_state(state,model,pos,slen)
        probs.append(prob)
        y = np.zeros([len(prob)])
        y[action] = 1
        reward = get_reward(state,motif,A)
        baseline.append(reward)
        pseudo_gradients.append((np.array(y).astype('float32')\
            - prob)*(reward-np.mean(baseline)))
        pos += 1
    out.append(copy.deepcopy(state))
    X = np.vstack([states])
    Y = probs + alpha * np.squeeze(np.vstack([pseudo_gradients]))
    err = model.train_on_batch(X, Y)
    ok = "ok" if re.search(motif, tostr(state)) else ""
    if e > 0 and e % 10 == 0:
        print("e=%d/%d err=%.6g motif=%s seq=%s %s" % \
              (e,num_episodes,err,motif,tostr(state),ok))
    states, probs, pseudo_gradients, rewards = [],[],[],[]
                                                61,61          Bot
```

Training algorithm:
(1) **predict** P-targets using current network weights
(2) **update** the P-targets with REINFORCE
(3) **re-train** network against the updated P-targets

The REINFORCE algorithm (by example):

action probs $P = \begin{bmatrix} P_0 \\ P_1 \end{bmatrix}$ action $A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$$A - P = \begin{bmatrix} 1 - P_0 \\ -P_1 \end{bmatrix} \begin{matrix} > 0 \\ < 0 \end{matrix} \qquad \bar{r} = r - baseline(r) \begin{cases} > 0 & \text{if } r > 0 \\ < 0 & \text{if } r = 0 \end{cases}$$

$$\boxed{P^{t+1} \leftarrow P^{t} + \alpha \cdot (A^{t} - P^{t}) \cdot \bar{r}^{t}}$$

Conclusion:
Action probability will be **increased / decreased** if the action resulted in a **positive / negative** baselined reward.

Each episode is of length = slen

How to run the prototype examples on Biowulf

denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro

```
$ sinteractive --gres=gpu:p100:1 --mem=4g
$ module load DLBio/class5
$ ls $DLBIO_BIN | sort -r
tabular_q-learning.py
dqn_seqopt.py
dpn_seqgen.py
```

Tabular Q-learning

```
$ tabular_q-learning.py
...
e=50/50 state=H Q=[8.7489345, 16.2050961]
        Policy= {'T': 'D', 'H': 'E'}
```

Deep Q-network for sequence optimization

```
$ dqn_seqopt.py
...
e=0, policy_slots_unpopulated=277/320
...
i=19900/20000, policy_slots_unpopulated=0/320

Best_policy[ 00000 ]=( 1 , 4 )  next_state= 00001  reward= 1.0 ok
Best_policy[ 00001 ]=( 1 , 3 )  next_state= 00011  reward= 1.0 ok
...
```

Deep policy network for sequence *de novo* generation

```
$ dpn_seqgen.py
...
e=10/1500 err=1.15963e-07 mofif=0011 seq=00101
...
e=1490/1500 err=5.83785e-10 mofif=0011 seq=00111 ok
e=1500/1500 err=5.56437e-10 mofif=0011 seq=00111 ok
                                                  28,57          All
```

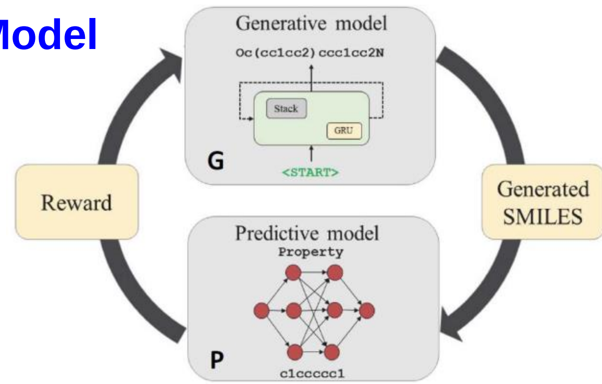
Biological example #5.
ReLeaSE: Reinforcement Learning for Structural Evolution

M.Popova et al., Sci. Adv. (2018)
https://github.com/isayev/ReLeaSE
https://hpc.nih.gov/apps/ReLeaSE.html

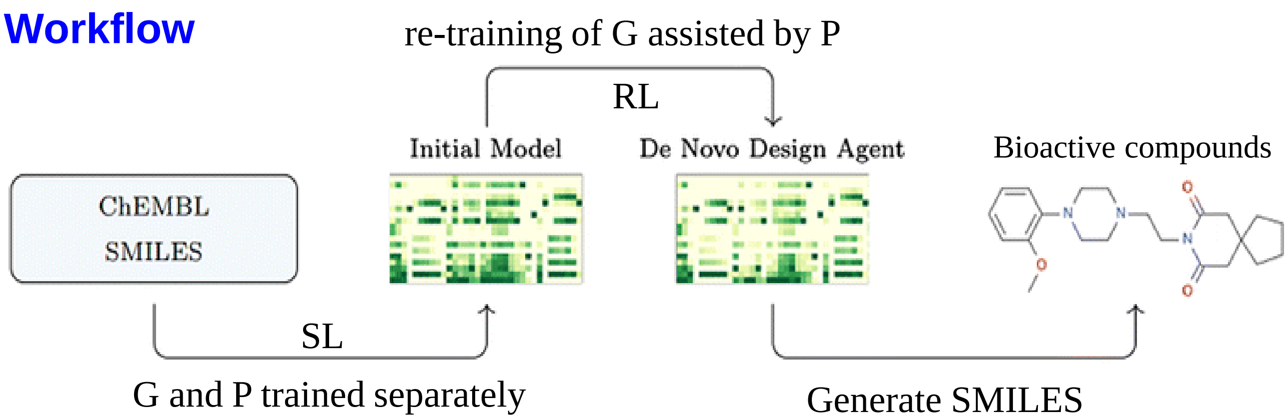
Summary

- novel method for ***de novo* generation** of chemical compounds
- with **desired physico-chemical** (e.g. logP) **and/or bioactivity properties** (e.g. JAK2)
- **based on DRL**, with **2 network models** and **2 stages of training**:
  1) both **G**enerator and **P**redictor are trained separately with SL
  2) both models are trained jointly with RL

Model



Workflow



Source code (reimplemented in Keras from PyTorch)

release_train.py — data.py models.py options.py

release_predict.py — smiles.py utils.py stackAugmentedRNN.py

release_visualize.py

Overview of the ReLeaSE training code
(only the main function has been shown)

Header
- imports,
- parsing command line options

Get data
- SMILES strings
- preprocessing, incl. tokenization

Define models
- Generator and Predictor
- Embedding layer
- StackAugmentedRNN layer
- GRU layer

Run the models
- Reinforcement
- Delayed rewards
- Rollout
- Adam optimizer

denisovga@biowulf:/usr/local/apps/release/20200516/bin

```python
if __name__ == '__main__':
    opt = parse_training_arguments()
    opt = process_options("train", opt)


    # Get data
    gdata, pdata = get_data(opt)


    # Define models
    opt = get_model_parameters(opt)
    generator, predictor, reinforce = \
        define_models(opt, gdata, pdata)


    # Run the models
    if opt.training_mode == "generator":
        generator.train()
    if opt.training_mode == "predictor":
        predictor.train()
    if opt.training_mode == "reinforce":
        reinforce.train()
                                178,45          Bot
```

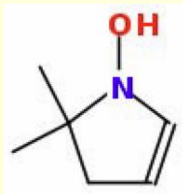
ReLeaSE data: SMILES strings and target property values

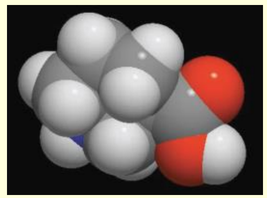
https://www.youtube.com/watch?v=zqUaxbSAYHQ
https://www.molinspiration.com/cgi-bin/properties

SMILES string:

(**S**implified **M**olecular **I**nput **L**ine **E**ntry **S**pecification)

Example:
OC(=O)C1CCCNC1





Generator data: <u>**SMILES string**</u> + ChEMBL id
Total size: ~ 1.6M

```
denisovga@biowulf:/data/denisovga/1_DL_Course/5_RLNs
CCO      CHEMBL545
C        CHEMBL17564
CO       CHEMBL14688
NCCS     CHEMBL602
NCCN     CHEMBL816
CN       CHEMBL43280
...
CC1=CN(C2CC(O)C(COP(O)(=O)OC3CC(OC3COP(O)(=O)OC3C(COP(O)(=O)OC4C(COP(
O)(=O)OC5C(COP(O)(=O)OC6C(COP(O)(=O)OC7C(COP(O)(=O)OC8C(COP(O)(=O)OC9
C(COP(O)(=O)OC%10C(COP(O)(=O)OC%11C(COP(O)(=O)OC%12C(COP(O)(=O)OC%13C
(COP(O)(=O)OC%14C(COP(O)(=O)OC%15C(COP(O)(=O)OC%16C(COP(O)(=O)OC%17C(
COP(O)(=O)OC%18C(COP(O)(=O)OC%19C(COP(O)(=O)OC%20C(COP(O)(=O)OC%21C(C
O)OC(C%21O)N%21C=CC(=O)NC%21=O)OC(C%20O)N%20C=CC(N)=NC%20=O)OC(C%19O)
N%19C=CC(=O)NC%19=O)OC(C%18O)N%18C=CC(N)=NC%18=O)OC(C%17O)N%17C=CC(=O
)NC%17=O)OC(C%16O)N%16C=CC(N)=NC%16=O)OC(C%15O)n%15cnc%16c%15NC(N)=NC
%16=O)OC(C%14O)N%14C=CC(N)=NC%14=O)OC(C%13O)N%13C=CC(N)=NC%13=O)OC(C%
12O)n%12cnc%13c(N)ncnc%12%13)OC(C%11O)n%11cnc%12c%11NC(N)=NC%12=O)OC(
C%10O)N%10C=CC(=O)NC%10=O)OC(C9O)N9C=CC(N)=NC9=O)OC(C8O)N8C=CC(N)=NC8
=O)OC(C7O)n7cnc8c(N)ncnc78)OC(C6O)N6C=CC(N)=NC6=O)OC(C5O)n5cnc6c(N)nc
nc56)OC(C4O)N4C=CC(N)=NC4=O)OC(C3O)N3C=CC(N)=NC3=O)N3C=C(C)C(=O)NC3=O
)O2)C(=O)NC1=O CHEMBL1253224
...
                                                  8,1           All
```

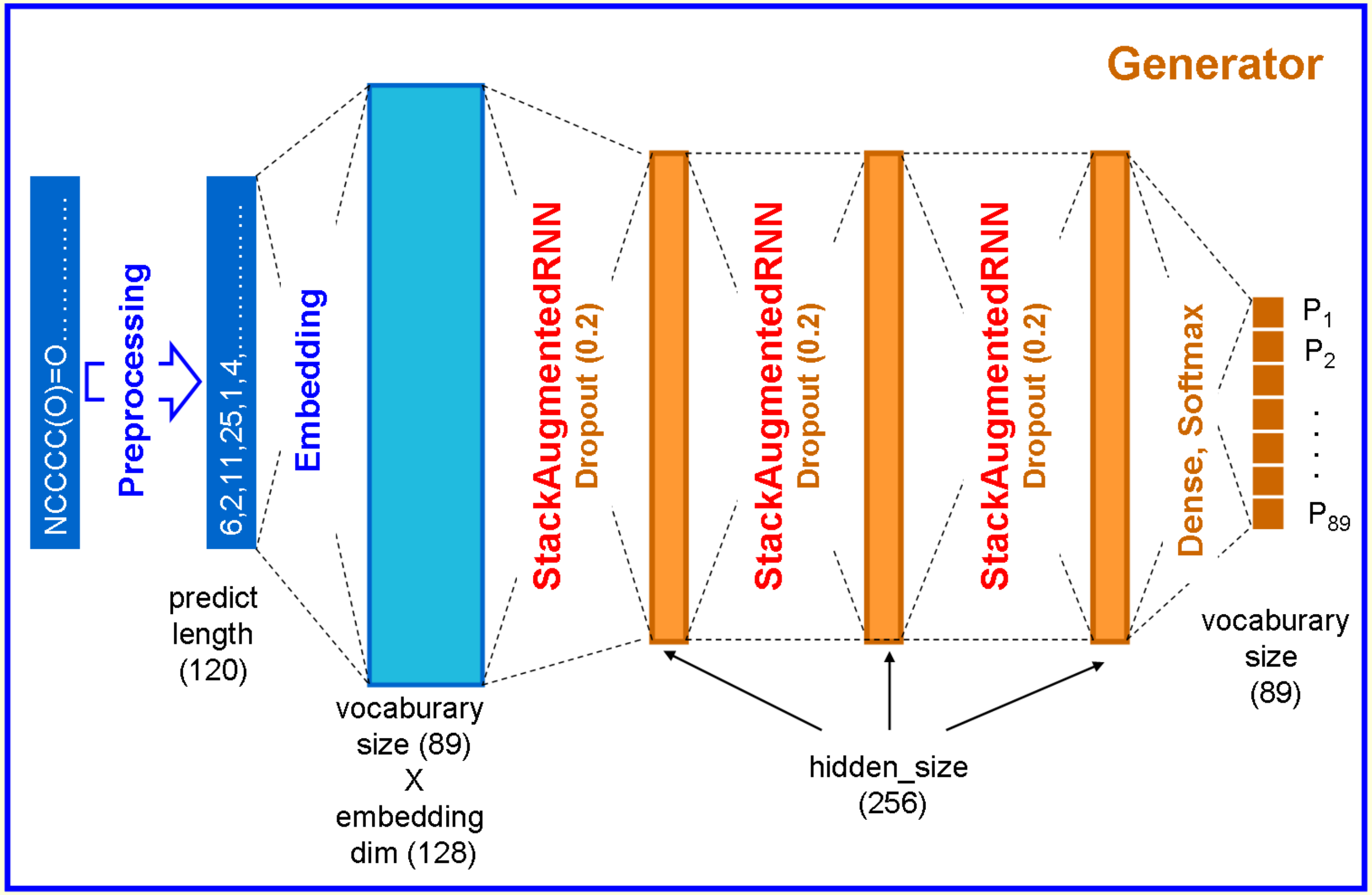
Target property values:

- **physical properties** considered important for drug molecules: e.g. the n-octanol-water partition coefficient, <u>**logP**</u> (= a measure of **lipophilicity**)
- **biological activity**: e.g., the Janus protein kinase 2 **inhibition coefficient**, **JAK2**

Predictor data: <u>**SMILES string**</u> + <u>**property value**</u>
Total size: ~14K for **logP**, and
~2K for **JAK2**

```
denisovga@biowulf:/data/denisovga/1_DL_Course/5_RLNs
O=S(=O)(Nc1cccc(-c2cnc3ccccc3n2)c1)c1cccs1,4.26
NC(=O)c1ccc2c(c1)nc(C1CCC(O)CC1)n2CCCO,4.53
NCCCn1c(C2CCNCC2)nc2cc(C(N)=O)ccc21,4.56
CNC(=S)Nc1cccc(-c2cnc3ccccc3n2)c1,4.59
O=C(Nc1cccc(-c2cnc3ccccc3n2)c1)C1CC1,4.6
O=C(Nc1cccc(-c2cnc3ccccc3n2)c1)c1ccco1,4.61
N#CCC(=O)Nc1cccc(-c2cnc3ccccc3n2)c1,4.74
...
CS(=O)(=O)N1CCCC(Nc2nc(N3CC(C#N)C3)ncc2-c2cnc3[nH]ccc3n2)C1,10.31
COC(=O)N1CCC(Nc2ncccc2-c2cnc3[nH]ccc3n2)CC1,10.33
Cc1ccc(-c2cnc3[nH]ccc3n2)c(NC2CCCN(S(C)(=O)=O)C2)n1,10.33
CCC(CO)Nc1ncc(-c2cnc3[nH]ccc3n2)c(NC2CCCN(S(C)(=O)=O)C2)n1,10.36
O=S(=O)(CC1CC1)N1CCCC(Nc2ncccc2-c2cnc3[nH]ccc3n2)C1,10.38
CC(=O)N1CCC(Nc2ncccc2-c2cnc3[nH]ccc3n2)C1,10.4
CS(=O)(=O)N1CCCC(Nc2ncccc2-c2cnc3[nH]cc(Cl)c3n2)C1,10.4
CS(=O)(=O)N1CCCC(Nc2ncccc2-c2cnc3[nH]cc(C4CC4)c3n2)C1,10.42
CN(c1[nH]cnc2nccc1-2)C1CCC(CS(=O)(=O)N2CCC(C)(O)C2)CC1,10.44
...
                                                  18,13         All
```

The Generator network overview



- takes a **(preprocessed) partial SMILES string** as an input
- outputs the **next token probability** values
- makes use of **Embedding**, **StackAugmenteRNN** and **Softmax** layers
- the **StackAugmenteRNN** layer has been implemented in Keras for the first time; it surpasses LSTM in the accuracy of the next token prediction

Data preprocessing and embedding by Generator

SMILES tokenization, preprocessing, embedding

SMILES pair encoding: https://github.com/XinhaoLi74/SmilesPE

Sample SMILES string:

```
NC(=O)CCCl        CHEMBL171266
```

Tokenization

```
'N',  'C',  '(',  '=',  'O',
 ')', 'C',  'C',  'Cl'

# Two control tokens
# (not a part of SMILES):
 '<'  (start token)
 '>'  (end token)
```

X data / Y data

```
X data          Y data
           <    N
          <N    C
         <NC    (
        <NC(    =
       <NC(=    O
      <NC(=O    )
     <NC(=O)    C
    <NC(=O)C    C
   <NC(=O)CC    Cl
 <NC(=O)CCCl    >
```

Replace tokens with their order #'s

```
      30    39
   30,39    35
30,39,35    16
    ...     ...
```

Pad X data with 0's

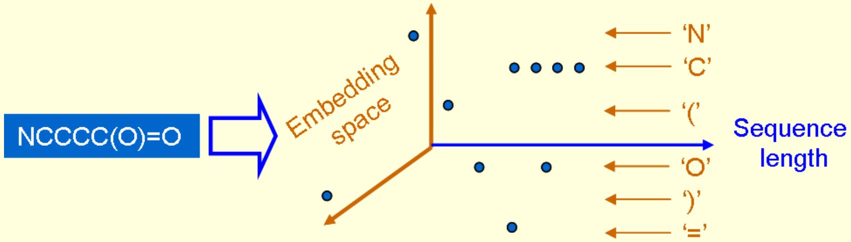
```
0, 0, 0,30    39
0, 0,30,39    35
0,30,39,35    16
    ...       ...
```

Other available SMILES tokens (total = 87):

```
'#', '%10', ...,  '/', '1', '2', ...,

'B', 'Br', 'F', 'I', 'N', 'P', 'S', ...

'[B-]', '[Br]', '[CH-]', '[CH2]', ...

'[NH+]', '[NH-]', '[NH2+]', ...

'[cH-]', '[n+]', '[n-]', '[nH+]',

'[nH]', '[o+]', '[s+]', '\\',

'c', 'n', 'o', 'p', 's'
```

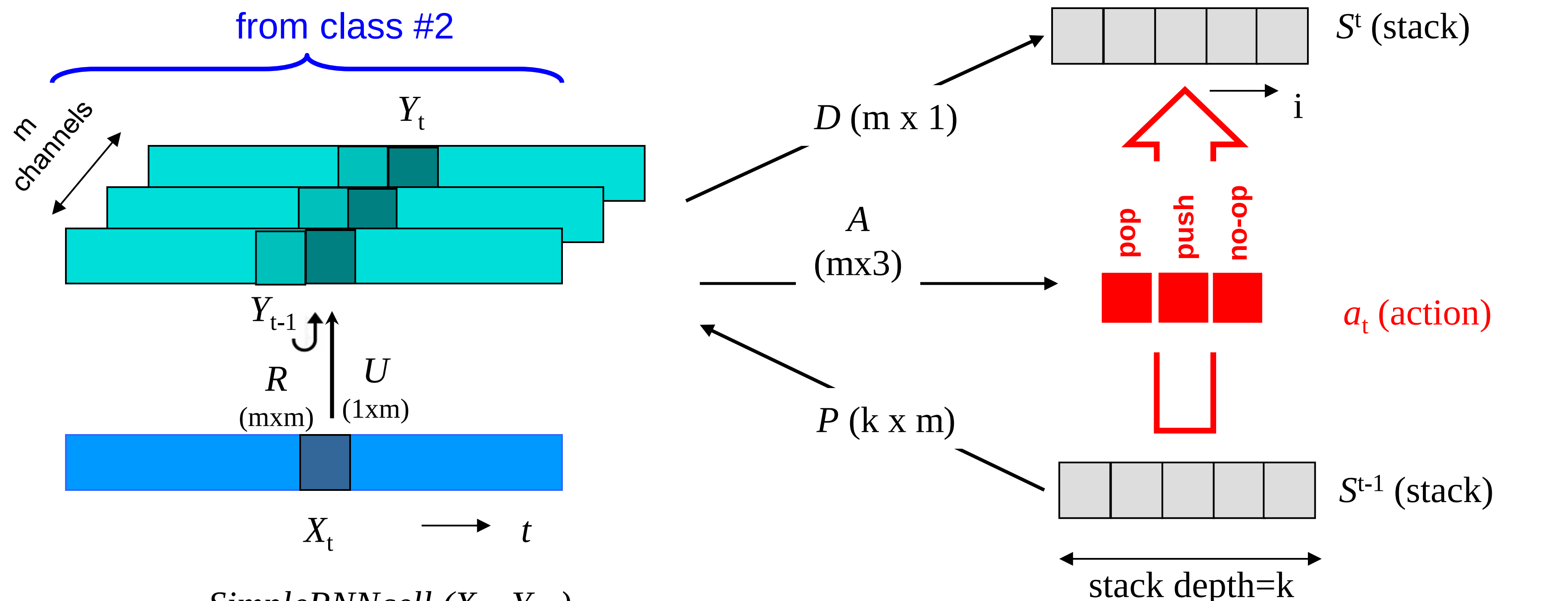
Embedding layer:

- transforms order #'s to float vectors in the Embedding space
- **purpose:** make all the tokens ≈ equi-distant
- takes two positional arguments:
  input dim – size of the input vocabulary (89)
  output_dim – dimension of the embedding space (128)



The Stack-Augmented RNN layer

A.Joulin and T.Mikolov . arXiv:1503.01007v4 (2015)



$$Y_t = \begin{cases} \overbrace{A\,(U\cdot X_t + R\cdot Y_{t-1} + b)}^{SimpleRNNcell\,(X_t\,,\,Y_{t-1})} + P\cdot S^{t-1} \\ LSTMcell\,(X_t\,,\,Y_{t-1}) + P\cdot S^{t-1} \\ GRUcell\,(X_t\,,\,Y_{t-1}) + P\cdot S^{t-1} \end{cases}$$

$$\boxed{Y_t = StackAugmentedRNNcell\,(X_t\,,\,Y_{t-1},\,S^{t-1})}$$

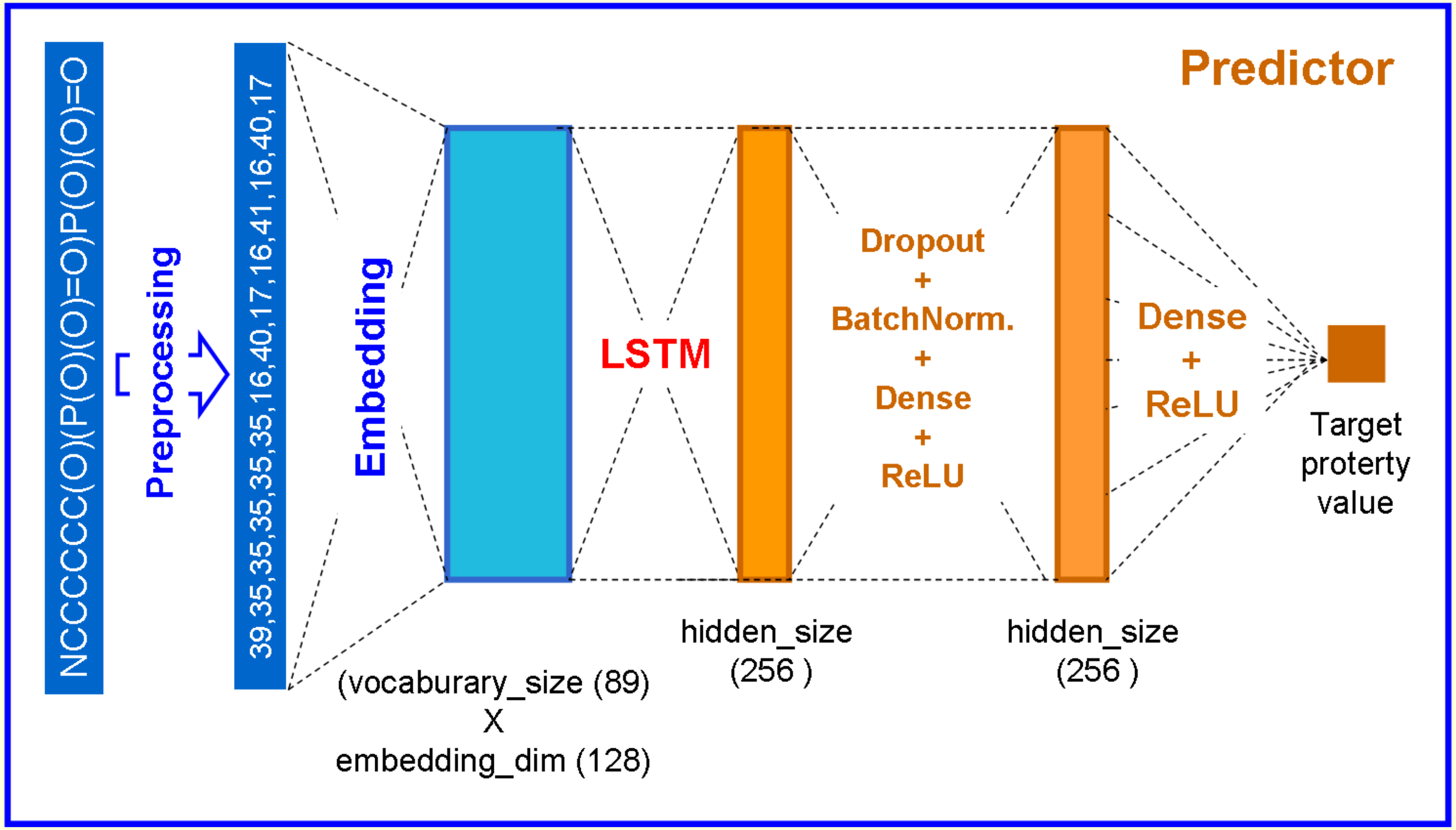
$$\boxed{S^t = F(Y_t,\,S^{t-1})}$$

$$a_t = A\cdot Y_t$$

$$S^t[i] = a_t[push]\cdot S^{t-1}[i-1] + a_t[no\text{-}op]\cdot S^{t-1}[i] + a_t[pop]\cdot S^{t-1}[i+1]$$

$$S^t[0] = a_t[push]\cdot D\cdot Y_t + a_t[no\text{-}op]\cdot S^{t-1}[0] + a_t[pop]\cdot S^{t-1}[1]$$

The Predictor network overview

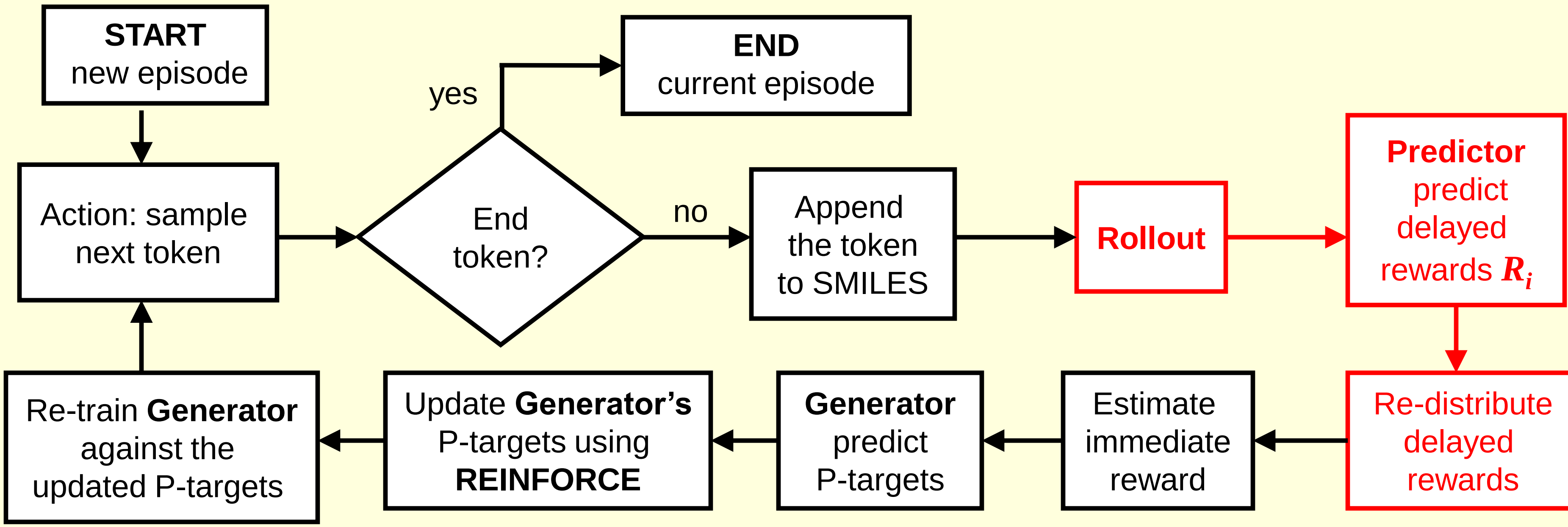


- takes a **complete SMILES string** as an input
- performs a single pass through the network
- outputs a **target property value**
- makes use of **LSTM** as a (single) recurrent layer
- there is still a room for improvement, see e.g.
  Message Passing NNs, *Jo et al, Methods 179 (2020) 65-72*

The Reinforcement framework: a flowchart

re-distributing delayed rewards, rollout

RUDDER: J.A.Arjona-Medina et al. arXiv:1806.07857 (2019)



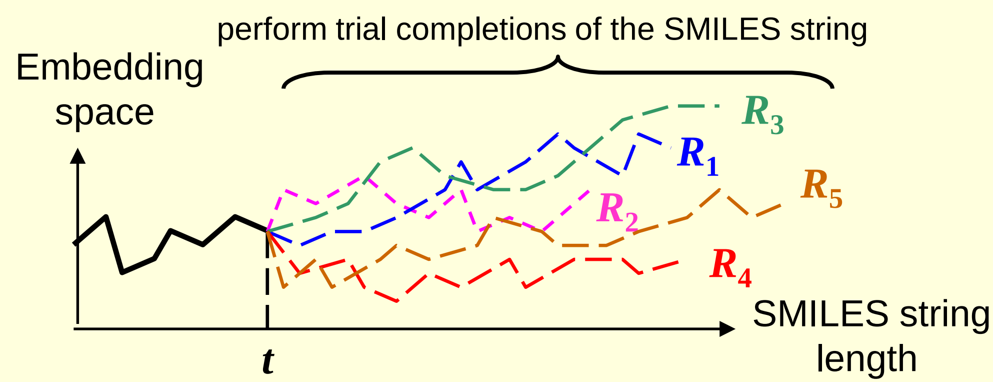
Re-distributing the delayed rewards:

- make a guess about immediate rewards based on delayed rewards

$$\boxed{r_t \approx \delta \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)}$$

(derived from the definition of a Q-function):

Rollout: $Q(s_t, a_t) \approx (R_1 + R_2 + \ldots R_N) / N$



How to run the ReLeaSE application on Biowulf

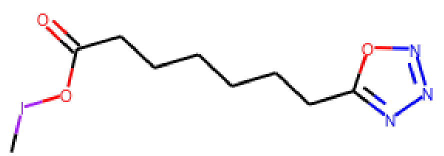
https://hpc.nih.gov/apps/ReLeaSE.html

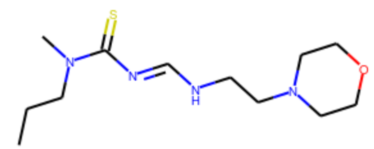
Using a single GPU:

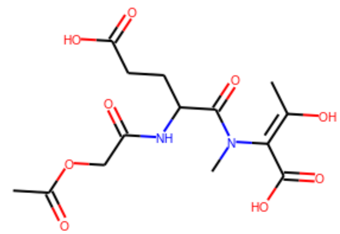
```
denisovga@biowulf:/data/denisovga/1_DL_Course/5_RLNs
sinteractive --mem=160g --gres=gpu:k80:1,lscratch:20 -c 14

module load release

ls $RELEASE_SRC
data.py      release_visualize.py  utils.py
models.py    release_predict.py    smiles.py
options.py   release_train.py      stackAugmentedRNN.py

release_train.py -m generator -r SA_GRU -g 4 -b 1000 --lr 3.e-4

release_train.py -d jak2 -m predictor -g 1 -b 128 --lr 0.0001 -e 500

release_train.py -d logp -m predictor -g 1 -b 128 --lr 0.0001 -e 500

release_train.py -m reinforce -d logp [ other options ]


release_predict.py -i checkpoints/generator.weights.SA_GRU.1.h5
...
generated SMILES string = C(NCCN2CCOCC2)=NC(=S)N(CCC)C

release_predict.py -i checkpoints/generator.weights.SA_GRU.2.h5 --stack_width 2
...
generated SMILES string = CIOC(=O)CCCCCCc1nnno1

release_predict.py -r LSTM -i checkpoints/generator.weights.LSTM.h5
...
generated SMILES string = CC(O)=C(C(O)=O)N(C)C(=O)C(CCC(O)=O)NC(=O)COC(C)=O


release_visualize.py -s "C(NCCN2CCOCC2)=NC(=S)N(CCC)C"

release_visualize.py -s "CIOC(=O)CCCCCCc1nnno1"

release_visualize.py -s "CC(O)=C(C(O)=O)N(C)C(=O)C(CCC(O)=O)NC(=O)COC(C)=O"
                                                                  41,77          Top
```

Using 4 GPUs:

```
denisovga@biowulf:/data/denisovga/1_DL_Course/5_RLNs
sinteractive --mem=120g --gres=gpu:p100:4,lscratch:10 -c 16

module load release

release_train.py -m generator -g 4 [ other options ]

release_train.py -m predictor -g 4 -d jak2 [ other options ]

release_train.py -m reinforce -g 4 -d logp [ other options ]
                                                       1,1          Top
```







Conclusions

1) Introduction to RL and DRL
- Agent and Environment
- Actions, States/Observations, Rewards and Policy
- Value-based RL: (tabular) Q-learning and Deep Q-network (DQN)
- Policy-based RL: Deep Policy Network (DPN) and the REINFORCE algorithm

2) The ReLeaSE application:
- a composite network: Generator + Predictor
- Tokenization and Preprocessing
- Embedding layer
- Stack-Augmented RNN layer
- GRU layer
- Distributing the Delayed Rewards and Rollout

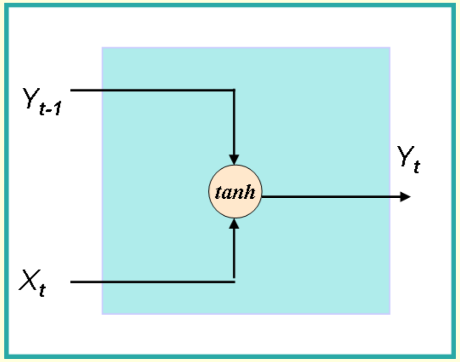
3) Other topics:
- Adam optimizer

BACKUP SLIDES

The Gated Recurrent Unit (GRU) cell

K.Cho et al, arXiv:1409.1259v2 (2014)

SimpleRNN cell: one neuron



$$Y_t = \tanh(b + w_{XY} \cdot X_t + w_{YY} \cdot Y_{t-1})$$

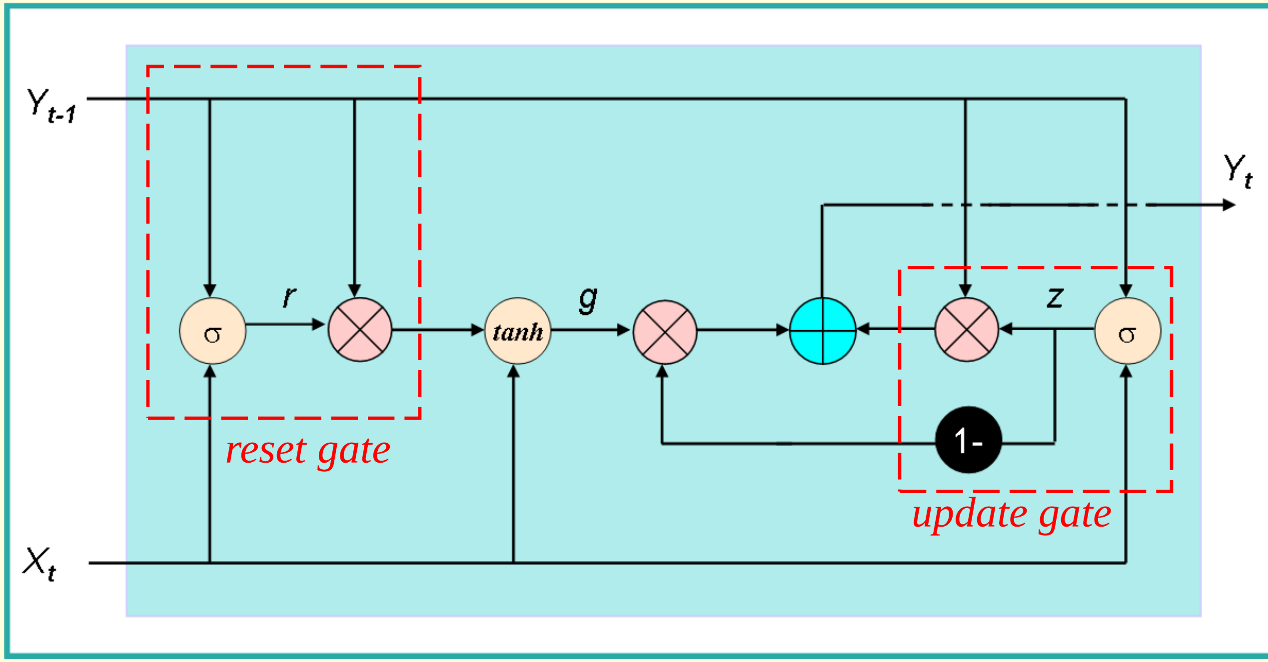
$\Updownarrow$

$$\boxed{X_t\,,\, Y_{t-1} \rightarrow Y_t}$$

Short-term memory:

$$\ldots \rightarrow Y_{t-2} \rightarrow Y_{t-1} \rightarrow Y_t \rightarrow \ldots$$

GRU cell: 3 neurons / 2 gates



$$r_t(X_t, Y_{t-1}) = \sigma(b_r + w_{Xr} \cdot X_t + w_{Yr} \cdot Y_{t-1})$$

$$z_t(X_t, Y_{t-1}) = \sigma(b_z + w_{Xz} \cdot X_t + w_{Yz} \cdot Y_{t-1})$$

$$g_t(X_t, Y_{t-1}) = \tanh(b_g + w_{Xg} \cdot X_t + w_{rYg} \cdot (r_t \otimes Y_{t-1}))$$

$$\mathbf{Y_t = z_t(X_t, Y_{t-1}) \otimes Y_{t-1} + [1 - z_t(X_t, Y_{t-1})] \otimes g_t(X_t, Y_{t-1})}$$

$\Updownarrow$

$$\boxed{X_t\,,\, Y_{t-1} \rightarrow Y_t}$$

LSTM (Long Short-Tetm Memory) cell: 4 neurons / 3 gates

$$\boxed{\begin{array}{l} 1)\ X_t,\ Y_{t-1},\ S_{t-1} \rightarrow S_t \\ 2)\ X_t,\ Y_{t-1},\ S_t \ \ \rightarrow Y_t \end{array}}$$

S_t = state tensor

The Adaptive Moment Estimation (Adam) optimizer

D.P.Kingma and J.L.Ba, Int. Conf. on Learning Representations, 2015.

Basic gradient descent formula for updating weights

$$w_{t+1} = w_t - \gamma \cdot \nabla_w J(w_t)$$

or

$$\Delta w_t = - \gamma \cdot \nabla_w J(w_t)$$

w = vector of weights
t = update #

γ = learning rate (a hyperparameter)
$\nabla_w J$ = gradient of the loss with respect to weights

Momentum (class #3)

$$\Delta w_t = \mu \cdot \Delta w_{t-1} - \gamma \cdot \nabla_w J(w_t)$$

RMSprop optimizer (class #4)

$E[\ldots]$ = running average
ε = small parameter

$$w_{t+1} = w_t - \frac{\gamma}{\sqrt{E[\nabla_w J_w(w_t)^2] + \varepsilon}} \cdot \nabla_w J(w_t)$$

$$E[\nabla_w Jw(w)^2]_t = \beta \cdot E[\nabla_w Jw(w)^2]_{t-1} + (1 - \beta) \cdot \nabla_w J(w_t)^2; \quad \beta \sim 0.9$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_w J(w_t)$$

- a momentum-like update

$$E[\nabla_w J(w)^2]_t = \beta_2 \cdot E[\nabla_w J(w_t)^2]_{t-1} + (1 - \beta_2) \cdot [\nabla_w J(w_t)]^2$$

- a RMSprop-like update

Adam gradient descent formula

$$w_{t+1} = w_t - \frac{\gamma}{\sqrt{\hat{v}_t} + \varepsilon} \cdot \hat{m}_t$$

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$

$$\hat{v}_t = E[\nabla_w J(w)^2]_t / (1 - \beta_2^t)$$

Homework assignments for Class #5

Initial Q-table		Actions	
		Eat	Drink
States	Hungry	0	0
	Thirsty	0	0

1. 1. Consider the agent discussed in the 1st simple example of the lecture. All the cells in the Q-table are initialized to zero values (Figure). Assume that $\alpha = 0.1$, $\delta = 0.9$ and (1) initially, an AGENT is in the state “Hungry” and executes the action “Eat”; (2) the next state of the AGENT will still be “Hungry” and it will execute the action “Drink”; and (3) after that the AGENT will stay “Thirsty” and will execute the action “Drink”. What will be the resulting values in the Q-table, if they are updated using to the Bellman equation?

2. Explore the limitations of deep Q-learning by modifying the dqn_seqopt.py script and looking at how this affects the result. In particular, (1) replace the motif sequence with ‘000111’ or other sequence of your choice, (2) increase the sequence length to 10 and (3) increase the number of iterations as needed. How many iterations will be needed in order to populate all the policy_enumerator slots?

3. Executable release_train.py supports the command line option --lin_length that specifies the lower limit on the length of the input SMILES strings that would be used by the training procedure. Using this option, compare the results of of training Generator on long SMILES strings, performed with LSTM and StackAugmentedGRU as the recurrent layers. Consider using for training only the SMILES strings > 120 tokens long, train for one epoch only and use the duration of the training and the reduction of the loss as the criteria for the comparison. NOTE: to speed up the training procedure, you may use up to 4 GPUs.

Solutions to the homework assignment #1

Initial Q-table		Actions	
		Eat	Drink
States	Hungry	0	0
	Thirsty	0	0

1. Consider the agent discussed in the 1st simple example of the lecture. All the cells in the Q-table are initialized to zero values (Figure). Assume that $\alpha = 0.1$, $\delta = 0.9$ and (1) initially, an AGENT is in the state "Hungry" and executes the action "Eat"; (2) the next state of the AGENT will be "Hungry" and it will execute the action "Drink"; and (3) after that the AGENT will stay "Thirsty" and will execute the action "Drink". What will be the resulting values in the Q-table, if they are updated using to the Bellman equation?

$$newQ(s_t , a_t) = Q(s_t , a_t) + \alpha \cdot [r_t + \delta \cdot \max Q(s_{t+1}, a) – Q(s_t , a_t)]$$

- Q11 = $\alpha \cdot 1 = 0.1$

(2) Q12 = $\alpha \cdot \delta \cdot 0.1 = 0.1 * 0.9 * 0.1 = 0.009$

(3) Q22 = 0.1

Initial Q-table		Actions	
		Eat	Drink
States	Hungry	0.1	0.009
	Thirsty	0	0.1

Solutions to the homework assignment #2

2. Explore the limitations of deep Q-learning by modifying the dqn_seqopt.py script and looking at how this affects the result. In particular, (1) replace the motif sequence with '000111' or other sequence of your choice, (2) increase the sequence length to 10 and (3) increase the number of iterations as needed. How many iterations will be needed in order to populate all the policy_enumerator slots?

Modify dqn_seqopt.py:

Near line #28:

```
popul, tot, i = 0, pow(2,slen)*2*slen, 0
while i < num_episodes or popul < tot:
```

Near line #48:

```
if i > 0:
```

Near line #51:

```
i += 1
```

Run the modified code:

```
./dqn_seqopt.py
...
i=200100/10000, policy_slots_unpopulated=1/20480
i=200101/10000, policy_slots_unpopulated=0/20480
```

Solutions to the homework assignment #3

3. Executable release_train.py supports the command line option --lin_length that specifies the lower limit on the length of the input SMILES strings that would be used by the training procedure. Using this option, compare the results of of training Generator on long SMILES strings, performed with LSTM and StackAugmentedGRU as the recurrent layers. Consider using for training only the SMILES strings > 120 tokens long, train for one epoch only and use the duration of the training and the reduction of the loss as the criteria for the comparison. NOTE: to speed up the training procedure, you may use up to 4 GPUs.

```
1) time release_train.py -m generator -r SA_GRU -b 1000 -e 1 --min_length 120
...
len(dataX)= 6,012,000  ...
...
Epoch 00001: loss improved from inf to 0.62374 ...
real    201m27.625s
user    171m28.651s
sys     46m46.296s
```

```
2) time release_train.py -m generator -r LSTM -b 1000 -e 1 --min_ length 120
...
Epoch 00001: loss improved from inf to 0.77492 ...
   real    165m42.253s
user    132m5.213s
sys     46m56.972s
```