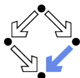
SOME EXPERIMENTS ON REINFORCEMENT LEARNING

The Case of the “Shortest Path Problem”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria

JKU JOHANNES KEPLER UNIVERSITY LINZ



Rationale

My primary interest is in the power of ML to guide search processes in abstract spaces; thus I've started to investigate its power for search in concrete spaces.

- Previous talk: experiments on Supervised Learning (SL).
  - Training on data sets to predict their labels.
- This talk: experiments on Reinforcement Learning (SL).
  - Training on agent executions to maximize rewards.
- RL has achieved spectacular successes:

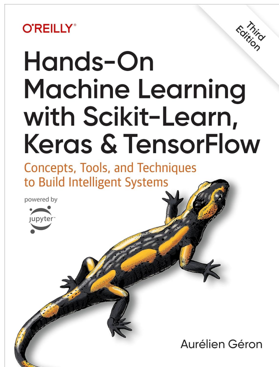
  > *RL . . . has been applied successfully to various problems, including energy storage, robot control, photovoltaic generators, backgammon, checkers, Go (AlphaGo), and autonomous driving systems.*
  >
  > https://en.wikipedia.org/wiki/Reinforcement_learning

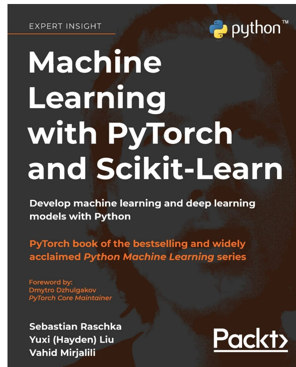
- But it is also notoriusly difficult:

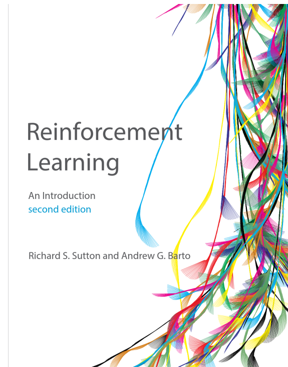
  > *SL wants to work. . . . RL must be forced to work.*
  >
  > Andrej Karpathy, https://news.ycombinator.com/item?id=13519044

So let's see how far we can get. . .

Textbooks

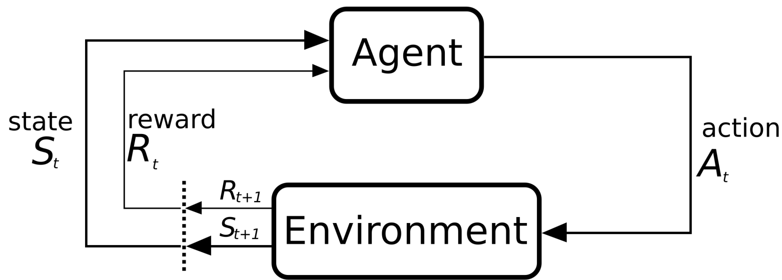






Mostly relied on [Géron, 2022]; see [Sutton and Barto, 2018] for all of the theory.

Reinforcement Learning



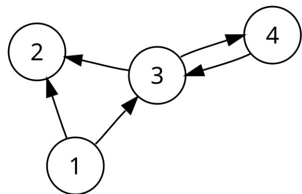
https://commons.wikimedia.org/wiki/File:Markov_diagram_v2.svg

Agent learns to perform "good" actions from "rewards" given by the environment.

The Problem

- Given: a directed graph G with n nodes and two nodes i, j.
  - $G = (V, E)$, $n \in \mathbb{N}$, $V = \mathbb{N}_n$, $E : \mathbb{N}_n \times \mathbb{N}_n \to$ Bool; $i, j \in \mathbb{N}_n$.
- Find: a *next node* $k \in \mathbb{N}_n$ on a shortest path from i to j in G.
  - A path with minimal *length*, i.e., the minimal number of edges.
  - $k = -1$, if there is no path from i to j in G.

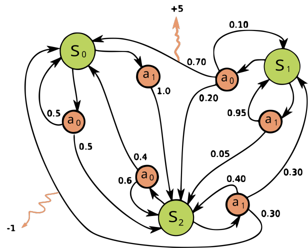
Reinforcement learning: the action is the choice of the next node k along the path; a reward is given if the next node is the target j.



https://commons.wikimedia.org/wiki/File:Directed_graph_no_background.svg

$next(1, 4) = 3.$

Markov Decision Processes



https://commons.wikimedia.org /wiki/File:Markov_Decision_Process.svg

- Markov Decision Process (S, A, T, R)
  - State set S, action set A (both finite).
  - Transition probability $T(s, a, s') \in [0, 1]$, reward $R(s, a, s') \in \mathbb{R}$.
- State/action sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$
  - Rewards $r_0 := R(s_0, a_0, s_1), r_1 := R(s_1, a_1, s_2), \ldots$
  - Return $(r_0 + \gamma \cdot r_1 + \gamma^2 \cdot r_2 + \ldots)$ for discount rate γ with $0 \leq \gamma < 1$.
  - The goal is to choose actions that maximize the expected return.
- Optimal Q-value $Q(s, a)$ ("quality value")
  - Expected return if process chooses in state s action a and from then on chooses optimally.
  - Bellman optimality equation: $Q(s, a) = \sum_{s'} T(s, a, s') \cdot [R(s, a, s') + \gamma \cdot \max_{a'} Q(s', a')]$.
- Optimal policy $\pi(s)$
  - The action that the process should choose in state s to maximize the expected return.
  - $\pi(s) = \operatorname{argmax}_a Q(s, a)$.

The optimal Q-values can be iteratively approximated: $Q_{k+1}(s, a) := \ldots Q_k(s', a')$.

The Q-Learning Algorithm

If the transition probabilities are unknown, we can statistically approximate Bellman's optimality equation $Q(s,a) = \sum_{s'} T(s,a,s') \cdot [R(s,a,s') + \gamma \cdot \max_{a'} Q(s',a')]$ as follows:

- Initialize $Q(s,a)$ as 0 for every s, a.
- Set initial "exploration probability" ε, "learning rate" α, and state s.
- Loop:
  - Choose action a that is possible in state s according to the "ε-greeedy" policy:
    - Either randomly (with probability ε, an "exploratory choice").
    - Or as $\operatorname{argmax}_a Q(s,a)$ (with probability $1-\varepsilon$, a "policy choice").
  - Determine from s and a the reward r and successor state s'.
  - Update $Q(s,a) := (1-\alpha) \cdot Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} \; Q(s',a')]$
  - Reduce ε, reduce α, and set $s := s'$.

This algorithm provably converges to the optimal Q-values (and the optimal policy).

Shortest Path Search in a Fixed Graph

```
NODE_NUMBER = 5

EPISODE_NUMBER = 10_000_000

ALPHA = 0.01  # initial learning rate
DECAY = 0.005 # learning rate decay
GAMMA = 0.95  # discount factor

# Q_matrix[node1, node2, node0] for "state" (node1, node2) and "action" node0
Q_matrix = np.full((NODE_NUMBER, NODE_NUMBER, NODE_NUMBER), 0.0)

# the problem is to find in the graph the shortest path from node1 to node2
# where 'dist' is the distance matrix associated to the graph
graph, node1, node2, dist = choose_problem()
```

The chosen action is the next node along the path; if the chosen node is the target, the reward is 1 (and 0, otherwise).

Searching for the Next Node in a Fixed Graph

```
for episode in range(EPISODE_NUMBER+1):
    alpha = ALPHA/(1+episode*DECAY)
    epsilon = max(0.01, 1-episode/(0.9*EPISODE_NUMBER))
    for step in range(NODE_NUMBER-1):
        node0 = choose_next_node(graph, node1, node2, epsilon)
        if node0 == -1:
            break
        reward = 1 if node0 == node2 else 0;
        Q_next = Q_matrix[node0, node2].max()
        Q_matrix[node1, node2, node0] *= 1-alpha
        Q_matrix[node1, node2, node0] += alpha*(reward+GAMMA*Q_next)
        if reward == 1:
            break
        node1 = node0
    node1, node2 = choose_nodes(graph, dist)
    if node1 == -1:
        break
```

In every episode, we choose a pair of nodes and follow the predicted path.

Searching for the Next Node in a Fixed Graph

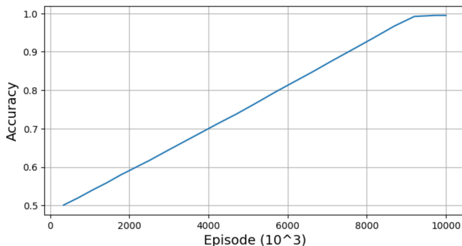
```
def choose_next_node(graph, node1, node2, epsilon):
    candidates = [ node0 for node0 in range(NODE_NUMBER)
                   if node0 != node1 and graph[node1, node0] == 1 ]
    if len(candidates) == 0:
        return -1
    if np.random.rand() < epsilon:
        return candidates[random.randint(len(candidates))]
    Q_values = [ Q_matrix[node1, node2, node0] for node0 in candidates ]
    index = max(enumerate(Q_values), key=lambda x: x[1])[0]
    return candidates[index]
```

An exploratory choice or a policy choice.

Evaluation

We consider the ratio of choices that decrease the distance to the target node.

```
graph: [[1 0 1 0 0][0 1 0 0 0][0 1 1 1 1][1 0 1 1 1][1 1 1 1 1]]
dist:  [[0 2 1 2 2][5 0 5 5 5][2 1 0 1 1][1 2 1 0 1][1 1 1 1 0]]
time: 5s, episodes: 340091, choices: 762252, learning rate: 0.000005877, accuracy: 0.50
...
time: 112s, episodes: 10000000, choices: 17038967, learning rate: 0.000000200, accuracy: 0.99
```



A large number of episodes needed to achieve a high ratio of good choices.

Deep Q-Learning

For large state spaces, a matrix representation of the Q-values is not feasible.

- Deep Q-Learning: use a deep neural network to estimate Q-values.
  - In principle, also other machine learning models (regressors) could be used.
- Q-value estimation: `Q_values(`s`) = Q_model.predict(np.array([`s`]))[0]`
  - The Q-values for all actions in state s are estimated in a single step.
- Q-value training: $y(s, a) = r + \gamma \cdot \max(\texttt{Q_values}(s'))$
  - From the Q-values for s', the target $y(s, a)$ for the pair (s, a) is determined.
  - A training step has to be performed that minimizes the error between $y(s, a)$ and the currently estimated Q-value for (s, a).

Simple basic idea but a lot of technical details.

Shortest Path Search in a Fixed Graph

```
NODE_NUMBER = 5
MODEL_WIDTH = 2*NODE_NUMBER
MODEL_DEPTH = 1

EPISODE_NUMBER = 10000
STARTUP_NUMBER = 50

BATCH_SIZE = 32
REPLAY_SIZE = max(BATCH_SIZE, 1000)
DISCOUNT_FACTOR = 0.95

Q_model = deep_net(MODEL_WIDTH, MODEL_DEPTH)
optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
lossfun = keras.losses.MeanSquaredError()
```

We use a multilayer perceptron to estimate the Q-values.

Shortest Path Search in a Fixed Graph

```
def deep_net(width,depth):
    input1 = keras.layers.Input(shape=(1,))
    input2 = keras.layers.Input(shape=(1,))
    encoded1 = keras.layers.CategoryEncoding(num_tokens=NODE_NUMBER, output_mode="one_hot")(input1)
    encoded2 = keras.layers.CategoryEncoding(num_tokens=NODE_NUMBER, output_mode="one_hot")(input2)
    inputs = keras.layers.concatenate([encoded1,encoded2])
    layer = inputs
    for _ in range(depth):
        layer = keras.layers.Dense(
            width,
            activation="selu",
            kernel_initializer="lecun_normal"
        )(layer)
    output = keras.layers.Dense(NODE_NUMBER)(layer)
    return tf.keras.Model(inputs=[input1,input2], outputs=[output])
```

We apply “one-hot” encoding to the node numbers.

Shortest Path Search in a Fixed Graph

```
replay_buffer = deque(maxlen=REPLAY_SIZE)
graph, node1, node2, dist = choose_problem()
for episode in range(EPISODE_NUMBER+1):
    epsilon = max(0.01, 1-episode/(0.9*EPISODE_NUMBER))
    for step in range(NODE_NUMBER-1):
        node0 = choose_next_node(graph, node1, node2, epsilon)
        if node0 == -1:
            break
        reward = 1 if node0 == node2 else 0;
        replay_buffer.append((node1, node2, node0, reward))
        if reward == 1:
            break
        node1 = node0
    if episode > STARTUP_NUMBER:
        train(replay_buffer)
    node1, node2 = choose_nodes(graph, dist)
    if node1 == -1:
        break
```

In each episode, the encountered situations are recorded for later training.

Shortest Path Search in a Fixed Graph

```
def choose_next_node(graph, node1, node2, epsilon):
    candidates = [ node0 for node0 in range(NODE_NUMBER)
                   if node0 != node1 and graph[node1, node0] == 1 ]
    if len(candidates) == 0:
        return -1
    if np.random.rand() < epsilon:
        return candidates[random.randint(len(candidates))]
    nodes1 = np.array([node1])
    nodes2 = np.array([node2])
    Q_values = Q_model.predict((nodes1,nodes2), verbose=0)[0]
    Q_values = [ Q_values[node0] for node0 in candidates ]
    index = max(enumerate(Q_values), key=lambda x: x[1])[0]
    return candidates[index]
```

Application of the network to estimate the Q-values.

Shortest Path Search in a Fixed Graph

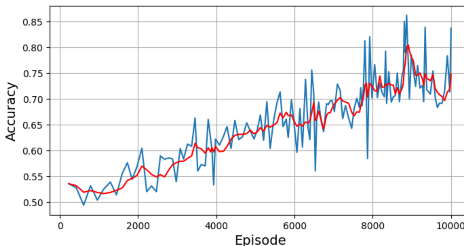
```
def train(replay_buffer):
    indices = np.random.randint(len(replay_buffer), size = BATCH_SIZE)
    batch = [ replay_buffer[index] for index in indices ]
    nodes1  = np.array([ node1  for node1, node2, node0, reward in batch ])
    nodes2  = np.array([ node2  for node1, node2, node0, reward in batch ])
    nodes0  = np.array([ node0  for node1, node2, node0, reward in batch ])
    rewards = np.array([ reward for node1, node2, node0, reward in batch ])
    Q_next = Q_model.predict((nodes0, nodes2), verbose=0)
    Q_target = rewards + DISCOUNT_FACTOR*Q_next.max(axis=1)
    Q_target = Q_target.reshape(-1,1)
    mask = tf.one_hot(nodes0, NODE_NUMBER)
    with tf.GradientTape() as tape:
        Q_all = Q_model((nodes1, nodes2))
        Q_now = tf.reduce_sum(Q_all*mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(lossfun(Q_target, Q_now))
    grads = tape.gradient(loss, Q_model.trainable_variables)
    optimizer.apply_gradients(zip(grads, Q_model.trainable_variables))
```

A gradient descent step on a randomly selected batch from the replay buffer.

Evaluation

We consider the ratio of choices that decrease the distance to the target node.

```
graph: [[1 0 1 0 0][0 1 0 0 0][0 1 1 1 1][1 0 1 1 1][1 1 1 1 1]]
dist:  [[0 2 1 2 2][5 0 5 5 5][2 1 0 1 1][1 2 1 0 1][1 1 1 1 0]]
time: 10s, episodes: 228, total choices: 491, new choices: 491, accuracy: 0.54
time: 20s, episodes: 421, total choices: 904, new choices: 413, accuracy: 0.53
...
time: 1326s, episodes: 10000, total choices: 20757, new choices: 49, accuracy: 0.84
```



Much fewer episodes needed, but slower execution and unstable behavior.

Fixed Q-Value Targets

Deeq Q-Learning tends to instability.

- Core reason: model both makes predictions and sets its own targets.
  - "Dog chases its own tail".
- Solution: use two separate models.
  - Online model: is trained and used to select actions.
  - Target model: sets the targets in the training.
  - Regularly (but not too frequently) copy the online model to the target model.

Since the target model is less frequently updated than the online model, the feedback loop is "dampened" and the Q-value estimations become more stable.

Shortest Path Search in a Fixed Graph

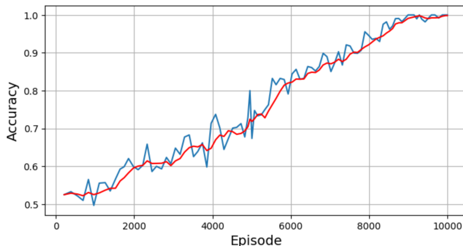
```
Q_model = deep_net(MODEL_WIDTH, MODEL_DEPTH)
Q_target_model = keras.models.clone_model(Q_model)
Q_target_model.set_weights(Q_model.get_weights())

def choose_next_node(graph, node1, node2, epsilon):
    ...
    Q_values = Q_model.predict((nodes1,nodes2), verbose=0)[0]
    ....
def train(replay_buffer):
    ...
    Q_next = Q_target_model.predict((nodes0, nodes2), verbose=0)
    ....

MODEL_UPDATE = 25
for episode in range(EPISODE_NUMBER+1):
    ...
    if episode % MODEL_UPDATE == 0:
        Q_target_model.set_weights(Q_model.get_weights())
    ...
```

Only minor code changes needed.

Evaluation

We consider the ratio of choices that decrease the distance to the target node.

```
graph: [[1 0 1 0 0][0 1 0 0 0][0 1 1 1 1][1 0 1 1 1][1 1 1 1 1]]
dist:  [[0 2 1 2 2][5 0 5 5 5][2 1 0 1 1][1 2 1 0 1][1 1 1 1 0]]
time: 10s, episodes: 212, total choices: 478, new choices: 478, accuracy: 0.53
time: 20s, episodes: 384, total choices: 857, new choices: 379, accuracy: 0.53
...
time: 981s, episodes: 10000, total choices: 17289, new choices: 51, accuracy: 1.00
```



Much more stable behavior, much higher accuracy.

Shortest Path Search in Arbitrary Graphs

```
def deep_net(width,depth):
    input1 = keras.layers.Input(shape=(1,))
    input2 = keras.layers.Input(shape=(1,))
    input3 = keras.layers.Input(shape=(NODE_NUMBER*NODE_NUMBER,))
    encoded1 = keras.layers.CategoryEncoding(num_tokens=NODE_NUMBER, output_mode="one_hot")(input1)
    encoded2 = keras.layers.CategoryEncoding(num_tokens=NODE_NUMBER, output_mode="one_hot")(input2)
    inputs = keras.layers.concatenate([encoded1,encoded2,input3])
    layer = inputs
    for _ in range(depth):
        layer = keras.layers.Dense(width, ...)(layer)
    output = keras.layers.Dense(NODE_NUMBER)(layer)
    return tf.keras.Model(inputs=[input1,input2,input3], outputs=[output])

MODEL_WIDTH = 2*NODE_NUMBER + 1*NODE_NUMBER*NODE_NUMBER
MODEL_DEPTH = 1
Q_model = deep_net(MODEL_WIDTH, MODEL_DEPTH)
Q_target_model = ...
```

Also the graph becomes input to the model.

Shortest Path Search in Arbitrary Graphs

```
for episode in range(EPISODE_NUMBER+1):
    ...
    graph, node1, node2, dist = choose_problem()
    ...
    for step in range(NODE_NUMBER-1):
        node0 = choose_next_node(graph, node1, node2, epsilon)
        if node0 == -1:
            break
        reward = 1 if node0 == node2 else 0;
        replay_buffer.append((graph, node1, node2, node0, reward))
        if reward == 1:
            break
    ...
```

A new graph is chosen in every episode.

Shortest Path Search in Arbitrary Graphs

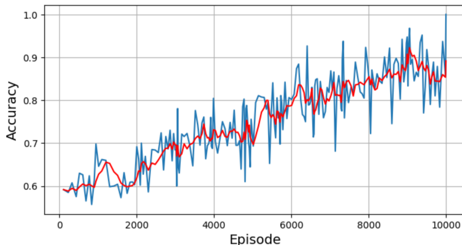
```
def choose_next_node(graph, node1, node2, epsilon):
    ...
    nodes1 = np.array([node1])
    nodes2 = np.array([node2])
    graphs = np.array([graph.flatten()])
    Q_values = Q_model.predict((nodes1,nodes2,graphs), verbose=0)[0]
    ...

def training(replay_buffer):
    ...
    graphs  = np.array([ graph.flatten() for graph, node1, node2, node0, reward in batch ])
    ...
    Q_next = Q_target_model.predict((nodes0, nodes2, graphs), verbose=0)
    ...
        Q_all = Q_model((nodes1, nodes2, graphs))
    ...
```

The graph is added as input to every model query.

Evaluation

We consider the ratio of choices that decrease the distance to the target node.

```
time: 5s, episodes: 106, total choices: 262, new choices: 262, accuracy: 0.59
time: 15s, episodes: 228, total choices: 580, new choices: 318, accuracy: 0.58
...
time: 2025s, episodes: 9915, total choices: 19810, new choices: 111, accuracy: 0.94
time: 2035s, episodes: 9987, total choices: 19930, new choices: 120, accuracy: 0.86
time: 2037s, episodes: 10000, total choices: 19948, new choices: 18, accuracy: 1.00
```

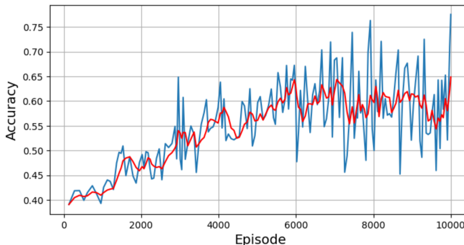


More unstable, but still a reasonably high accuracy.

Evaluation

We now try graphs with 10 nodes and use `MODEL_UPDATE=50`.

```
time: 5s, episodes: 130, total choices: 773, new choices: 773, accuracy: 0.39
time: 15s, episodes: 274, total choices: 1576, new choices: 803, accuracy: 0.42
...
time: 1613s, episodes: 9913, total choices: 34868, new choices: 142, accuracy: 0.52
time: 1623s, episodes: 9972, total choices: 34996, new choices: 128, accuracy: 0.71
time: 1627s, episodes: 10000, total choices: 35045, new choices: 49, accuracy: 0.78
```

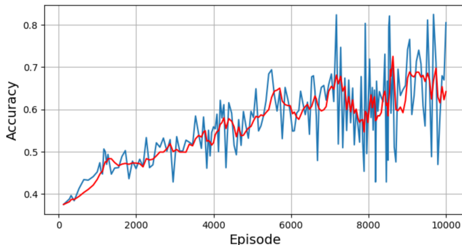


Unstable, low accuracy.

Evaluation

Try a larger replay buffer: `REPLAY_SIZE=5000`.

```
time: 5s, episodes: 128, total choices: 751, new choices: 751, accuracy: 0.38
time: 15s, episodes: 280, total choices: 1639, new choices: 888, accuracy: 0.39
...
time: 1699s, episodes: 9900, total choices: 34088, new choices: 131, accuracy: 0.68
time: 1709s, episodes: 9950, total choices: 34212, new choices: 124, accuracy: 0.67
time: 1718s, episodes: 10000, total choices: 34299, new choices: 87, accuracy: 0.80
```



A moderate improvement.

The SARSA Algorithm

Similar to Q-Learning but different update rule that does not use $\max_{a'} Q(s', a')$ (the optimal policy) but simply $Q(s', a')$ for action a' chosen according to the current policy.

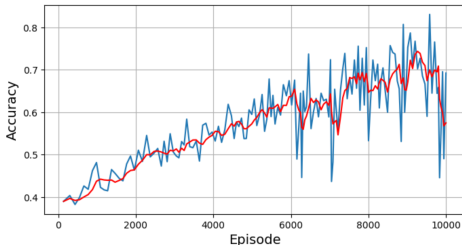
- Initialize $Q(s, a)$ as 0 for every s, a.
- Set initial “exploration probability” ε, “learning rate” α, and state s.
- Choose action a that is possible in state s according to the “ε-greeedy” policy:
  - Either randomly (with probability ε, an “exploratory choice”).
  - Or as $\operatorname{argmax}_a Q(s, a)$ (with probability $1 - \varepsilon$, a “policy choice”).
- Loop:
  - Determine from s and a the reward r and successor state s'.
  - Choose action a' that is possible in state s' according to the “ε-greeedy” policy.
  - Update $Q(s, a) := (1 - \alpha) \cdot Q(s, a) + \alpha \cdot [r + \gamma \cdot Q(s', a')]$
  - Reduce ε, reduce α, and set $s := s'$, $a := a'$.

“Although Q-learning actually learns the values of the optimal policy, its *online* performance is worse than that of SARSA ...” [Sutton and Barto, 2018]

Evaluation

Also `REPLAY_SIZE=5000` but just a *single* model.

```
time: 5s, episodes: 137, total choices: 772, new choices: 772, accuracy: 0.39
time: 15s, episodes: 298, total choices: 1711, new choices: 939, accuracy: 0.40
...
time: 1450s, episodes: 9913, total choices: 34041, new choices: 92, accuracy: 0.70
time: 1460s, episodes: 9946, total choices: 34145, new choices: 104, accuracy: 0.49
time: 1468s, episodes: 10000, total choices: 34249, new choices: 104, accuracy: 0.69
```

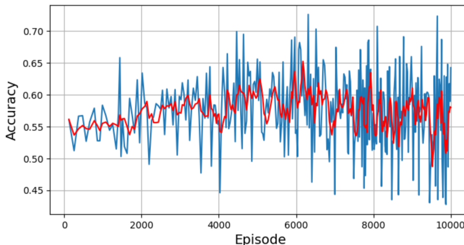


A bit higher accuracy and substantial more stability.

Evaluation

Finally, graphs with 20 nodes and 40 random edges.

```
time: 5s, episodes: 124, total choices: 936, new choices: 936, accuracy: 0.56
time: 15s, episodes: 258, total choices: 1770, new choices: 834, accuracy: 0.51
...
time: 3195s, episodes: 9962, total choices: 69097, new choices: 128, accuracy: 0.62
time: 3205s, episodes: 9985, total choices: 69248, new choices: 151, accuracy: 0.59
time: 3212s, episodes: 10000, total choices: 69357, new choices: 109, accuracy: 0.64
```



Only a slight improvement over random guessing.

Conclusions

So this is the best I could achieve (with limited efforts)...

> *But the truth is, RL is hard: training is often unstable, and you may need to try many hyperparameter values and random seeds before you find a combination that works well. [Géron, 2022]*

- *Number of episodes*, *size of replay buffer*, ε*-decay strategies*.
- Network: model, *size*, *hyperparameters*.
- Optimizer: algorithm, hyperparameters.
- Deep Q-Learning Variants:
  - *Fixed Q-value target*, *double DQN*, prioritized experience replay, *dueling DQN*, Rainbow (combination of techniques), ...
- Other Reinforcement Learning Algorithms:
  - *SARSA*, Monte Carlo, AlphaGo, AlphaZero, Actor-critic algorithms (AC, A3C, A2C, SAC), proximal policy optimization, curiosity-based exploitation, open-ended learning, ...

(*italics* what I actually but unsystematically experimented with)

Better spend efforts on supervised learning (if applicable) first...