

# Business 4720 - Class 22

## Reinforcement Learning – Function Approximation

Joerg Evermann

Faculty of Business Administration  
Memorial University of Newfoundland  
`jevermann@mun.ca`



Unless otherwise indicated, the copyright in this material is owned by Joerg Evermann. This material is licensed to you under the [Creative Commons by-attribution non-commercial license \(CC BY-NC 4.0\)](https://creativecommons.org/licenses/by-nc/4.0/)

## What You Will Learn:

- ▶ Reinforcement Learning
  - ▶ Action-value approximation
  - ▶ Policy approximation

Richard S. Sutton and Andrew G. Barto (2018) *Reinforcement Learning – An Introduction*. 2nd edition, The MIT Press, Cambridge, MA. (SB)

<http://incompleteideas.net/book/the-book.html>

Chapters 9–13

Sudharsan Ravichandiran (2020) *Deep Reinforcement Learning with Python*. 2nd edition. Packt Publishing, Birmingham, UK.

Chapters 9–11

Implementations are available on the following GitHub repo:

`https://github.com/jevermann/busi4720-rl`

The project can be cloned from this URL:

`https://github.com/jevermann/busi4720-rl.git`

# Function Approximation

## Previously

- ▶ Tabular methods only suitable for small state space and discrete actions

## Now

- ▶ Approximate the state-value function  $v$  by parameterized function  $\hat{v}$ :

$$\hat{v}(s) = \hat{v}(s, \theta) \approx v_{\pi}(s)$$

- ▶ Approximate the action-value function  $q$  by a parameterized function  $\hat{q}$ :

$$\hat{q}(s, a) = \hat{q}(s, a, \theta) \approx q_{\pi}(s, a)$$

- ▶ Approximate policy  $\pi$  by a parameterized function  $\hat{\pi}$ :

$$\hat{\pi}(a|s) = \hat{\pi}(a|s, \theta) \approx \pi(a|s)$$

## Advantages

- ▶ Continuous states and/or actions
- ▶ Tractable problems despite large state space
- ▶ Flexible functions (linear, trees, neural networks)
- ▶ Generalization to related states
  - ▶ Changing  $\theta$  changes the values of multiple states
- ▶ Applicable to partially observable problems
  - ▶ State function may not depend on complete state information

# Stochastic Gradient Methods

Assume a MSE value error:

$$\bar{V}E = \sum_{s \in \mathcal{S}} \mu(s) [q_{\pi}(s, a) - \hat{q}(s, a, \theta)]^2$$

Follow the steepest slope ("gradient"; vector of partial derivatives) of the function to update parameters:

$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{1}{2} \alpha \nabla [q_{\pi}(S_t, A_t) - \hat{q}(S_t, A_t, \theta_t)]^2 \\ &= \theta_t + \alpha [q_{\pi}(S_t, A_t) - \hat{q}(S_t, A_t, \theta_t)] \nabla \hat{q}(S_t, A_t, \theta_t)\end{aligned}$$

True value  $q_{\pi}(S_t, A_t)$  generally unknown; use an unbiased estimate  $U_t = R_t + \gamma \hat{q}(S_{t+1}, A_{t+1}, \theta)$  instead:

$$\theta_{t+1} = \theta_t + \alpha [U_t - \hat{q}(S_t, A_t, \theta_t)] \nabla \hat{q}(S_t, A_t, \theta_t)$$

**Replace update to  $Q$  with update to  $\theta$**

# Example – Semi-gradient SARSA

Initialize  $\theta \in \mathbb{R}^d$  arbitrarily

Loop for each episode:

Initialize  $S_0$

Choose  $A$  as a function of  $\hat{q}(S_0, \cdot, \theta)$  e.g.,  $\epsilon$ -greedy

Loop for each step of episode:

Take action  $A$ , observe  $R, S'$

Choose  $A'$  as a function of  $\hat{q}(S', \cdot, \theta)$  e.g.,  $\epsilon$ -greedy

$\theta \leftarrow \theta + \alpha [R + \gamma \hat{q}(S', A', \theta) - \hat{q}(S, A, \theta)] \nabla \hat{q}(S, A, \theta)$

$S \leftarrow S'; A \leftarrow A'$

until  $S$  is terminal



# "The Deadly Triad"

**Instability** and **Divergence** arise when combining all three elements:

- ▶ **Function approximation:** Generalizing from a state space using linear functions or neural networks
- ▶ **Bootstrapping:** Targets include existing estimates (e.g. SARSA) rather than actual rewards only (e.g. MC methods)
- ▶ **Off-policy training:** Training on a distribution of transitions other than that produced by the target policy

## Experience Replay

- ▶ Store sequences  $S, A, R, S', A'$  in *replay buffer*
- ▶ FIFO queue of limited size
- ▶ Sample from replay buffer for each batch
- ▶ Removes/limits correlation of states within batches
- ▶ Smooths data distribution changes

## Target Network

- ▶ Maintain stable targets during updates
- ▶ Periodic update from "main" network

# DQN – Algorithm

Init replay buffer  $D \leftarrow \emptyset$

Init main action-value function approximation  $\hat{q}_M$  with random parameters  $\theta_M$

Init target action-value function approximation  $\hat{q}_T$  with parameters  $\theta_T = \theta_M$

Loop for each episode:

    Initialize  $S$

    For each step of the episode:

        Select action  $A$  using an  $\epsilon$ -greedy policy based on  $\hat{q}_M$

        Take action  $A$  and observe  $R, S_{t+1}$

        Store transition  $(S_t, A_t, R_t, S_{t+1})$  in  $D$

        Sample minibatch  $(S_j, A_j, R_j, S_{j+1})$  from  $D$

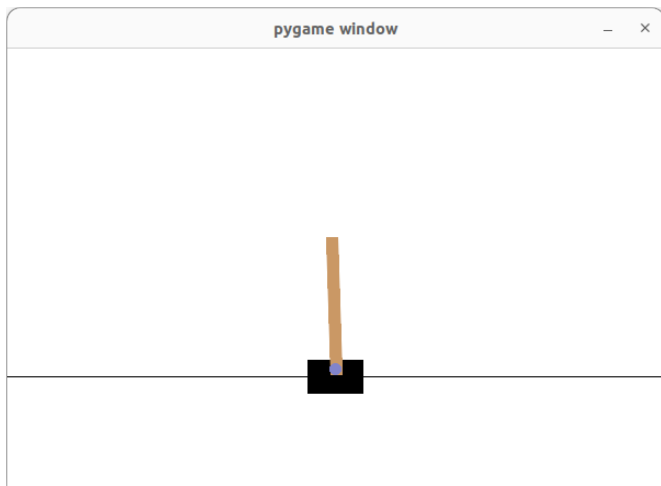
        Target  $y_j \leftarrow \begin{cases} r_j & \text{if } S_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{A'} \hat{q}_T(S_{j+1}, A'; \theta_T) & \text{otherwise} \end{cases}$

$\theta_M \leftarrow \theta_M + \alpha [y_j - \hat{q}_M(S_j, A_j, \theta_M)] \nabla \hat{q}_M(S_j, A_j, \theta_M)$

    Every  $C$  steps, update  $\hat{q}_T \leftarrow \hat{q}_M$  by setting  $\theta_T \leftarrow \theta_M$

- ▶ In practice,  $S$  is a function  $\phi(X)$  of inputs  $X$  through feature-extraction and pre-processing
- ▶ In practice, the update  $y_j - Q(S_j, a_j, \theta)$  is clipped to  $[-1, 1]$

# DQN Example – Cart Pole



# DQN Example – Cart Pole

## Action Space:

0	Push cart to the left
1	Push cart to the right

## State/feature space:

<i>Num</i>	<i>Observation</i>	<i>Min</i>	<i>Max</i>
0	Cart position	-4.8	4.8
1	Cart velocity	-Inf	Inf
2	Pole angle	-24 deg	24 deg
3	Pole angular velocity	-Inf	Inf

**Rewards** are +1 for every step taken

**Termination** occurs either:

- ▶ Pole angle is greater than  $\pm 12$  deg
- ▶ Cart position is greater than  $\pm 2.4$
- ▶ Episode length is greater than 200

# DQN in Python

Use the "CartPole" environment from  
<https://gymnasium.farama.org/>:

```
import math
import random
import keras
from keras import layers
import gymnasium as gym
import tensorflow as tf
import numpy as np
import pygame

env = gym.make("CartPole-v1", render_mode="human")

Actions = range(0, env.action_space.n)
Ssize = env.observation_space.shape[0]
```

## Neural network and RL hyperparameters:

```
# Neural net parameters
batch_size = 8
dropout = 0.25
activation = 'relu'

epsilon = 0.8 # initial epsilon
gamma = 0.9 # discount factor
neps = 1000 # decreasing epsilon factor
C = 50*batch_size # When to update weights

# Replay buffer D
D = collections.deque(maxlen=5000)
```



## Define the neural networks:

```
# Main network, used to select actions
Q = keras.Sequential([
    layers.InputLayer(input_shape=(Ssize+1),
                       batch_size=batch_size,
                       dtype=tf.float32),
    layers.Dense(Ssize*4, activation=activation),
    layers.Dropout(rate=dropout),
    layers.Dense(Ssize*2, activation=activation),
    layers.Dropout(rate=dropout),
    layers.Dense(1, activation='linear')
])
Q.compile(loss='huber', optimizer='adam')

# Target network, used to compute targets
Qhat = keras.models.clone_model(Q)
Qhat.compile(loss='huber', optimizer='adam')
Qhat.set_weights(Q.get_weights())
```

Getting a  $Q(s, a)$  value involves predicting from the neural net:

```
def getQ(Q, s, a):  
    return Q.predict(np.expand_dims(np.array( \  
        s.tolist()+[a]), axis=0), verbose=0)[0][0]
```

Max/Argmax operator for  $Q(s, a)$  using prediction from main or target network:

```
def maxQ(Q, s, arg):  
    maxq = -np.inf  
    maxa = None  
    for a in Actions:  
        q = getQ(Q, s, a)  
        if q > maxq:  
            maxq = q  
            maxa = a  
    return maxa if arg else maxq
```

$\epsilon$ -greedy policy  $p_i(s)$

```
def pi(s, epsilon):  
    if random.random() < epsilon:  
        return random.choice(Actions)  
    else:  
        return maxQ(Q, s, True)
```

Update target for DQN:

```
def target_DQN(Q, Qhat, a, r, sprime):  
    return r + gamma * maxQ(Qhat, sprime, False)
```

Update target for DDQN:

```
def target_DDQN(Q, Qhat, a, r, sprime):  
    return r + gamma * getQ(Qhat, sprime, \  
                             maxQ(Q, sprime, False))
```

Creating x and y data for training:

```
def training_xy(batch, ddqn=False):
    x = np.zeros((batch_size, Ssize+1))
    y = np.zeros(batch_size)
    for i, (s, a, r, t, sprime) in enumerate(batch):
        x[i] = list(s) + [a]
        if t == 1:
            y[i] = r
        else:
            if ddqn:
                y[i]=target_DDQN(Q, Qhat, a, r, sprime)
            else:
                y[i]=target_DQN(Q, Qhat, a, r, sprime)
    return x, y
```

## DQN/DDQN code:

```
for t in range(max_steps):
    s = env.reset()[0]
    a = pi(s, epsilon*math.exp(-t/neps))
    sprime, r, terminal, _, _ = env.step(a)
    G += r
    D.append((s, a, r, int(terminal), sprime))
    s = sprime
    if t >= batch_size:
        batch = random.sample(D, batch_size)
        x, y = training_xy(batch, ddqn=True)
        loss = Q.train_on_batch(x=x, y=y)

    if t % C == 0:
        Qhat.set_weights(Q.get_weights())
```

Complete code at

[https://evermann.ca/busi4720/DDQN\\_tuples.py](https://evermann.ca/busi4720/DDQN_tuples.py)

## Prioritized Experience Replay (PEX)

- ▶ Important actions are sampled with higher probability
- ▶ Use absolute TD error as priorities
- ▶ Faster learning

## Double DQN

- ▶ Based on Double Q-Learning
- ▶ Uses target network  $\hat{Q}$  as second  $Q$  function
- ▶ Removes upwards bias from using  $\max()$  functions as estimator

$$A(s, a) = Q(s, a) - V(s)$$

**Advantage of action  $a$  in state  $s$  over the average action in state  $s$**

Rewrite as:

$$Q(s, a) = V(s) + A(s, a)$$

# Dueling DQN

- ▶ Neural network from features  $x$  for value function  $V(s)$  ("value stream")
- ▶ Neural network from features  $x$  for advantage function  $A(s, a)$  ("advantage stream")
- ▶ Typically, value stream and advantage stream follow one or more common layers
- ▶ Aggregate to compute  $Q(s, a)$

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + \left( A(s, a, \theta, \alpha) - \frac{a}{|\mathcal{A}|} A(s, a', \theta, \alpha) \right)$$

Where  $\theta$  are shared neural-network parameters,  $\beta$  are parameters only for the "value-stream" neural network, and  $\alpha$  are parameters only for the "advantage-stream" neural network



## Idea

Learn a parameterized policy :

$$\pi(s, a) = \pi(s, a, \theta) = \Pr(A_t = a | S_t = s, \theta_t = \theta)$$

Optimize:

$$J(\theta) = v_{\pi_\theta}(s_0)$$

## Advantages

- ▶ Simpler to approximate than action-value function
- ▶ Selection of actions with arbitrary probabilities
- ▶ Can better approach deterministic policy than  $\epsilon$ -greedy action selection over action values
- ▶ Suitable for large and continuous action spaces

REINFORCE update:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

- Update proportional to return  $G_t$
- Update inversely proportional to action probability  $\pi$

## REINFORCE: Monte-Carlo Control (episodic)

Input: A differentiable policy  $\pi(a|s, \theta)$ ; step size  $\alpha > 0$

Initialize policy parameters  $\theta \in \mathbb{R}^d$  arbitrarily

Loop forever (for each episode):

    Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ ,

    Loop for each step of the episode  $t = 0, 1, \dots, T - 1$  :

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$$

$$\theta \leftarrow \theta + \alpha \mathbf{G} \nabla \ln \pi(A_t | S_t, \theta)$$

# REINFORCE in Python

## Policy function/network for discrete actions:

```
# Batch size of 1 as we update after every action
p_net = keras.Sequential([
    layers.InputLayer(batch_input_shape=(1, Ssize),
        dtype=tf.float32),
    layers.Dense(Ssize*4, activation='relu'),
    layers.Dropout(rate=0.25),
    layers.Dense(Ssize*2, activation='relu'),
    layers.Dropout(rate=0.25),
    layers.Dense(len(Actions), activation='softmax')
])
```

## Action selection using policy network probabilities:

```
def select_action(state):
    probs = p_net(np.expand_dims(state, axis=0))[0].numpy()
    action = np.random.choice(Actions, size=1, p=probs)[0]
    return action
```

# REINFORCE in Python [cont'd]

## Generate an episode:

```
# Do this for each episode
for e in range(neps):
    # Initialize variables and lists
    T = 0
    rewards, states, actions = [], [], []
    # Reset environment
    s = env.reset()[0]
    terminal = False
    # Generate an episode and keep track
    # of states, actions, rewards
    while (T < max_steps) and not terminal:
        a = select_action(s)
        sprime, r, terminal, _, _ = env.step(a)
        states.append(s)
        actions.append(a)
        rewards.append(r)
        s = sprime
        T += 1

    print(f'Episode {e:5} goes to step {T:3}')
    # Compute discounted returns
    returns = discounted_returns(rewards, gamma)
```

# REINFORCE in Python [cont'd]

## Learning from episode:

```
# Learn for each step of the episode
for t in range(len(returns)):
    with tf.GradientTape() as tape:
        # Action probabilities
        pi = p_net(np.expand_dims(states[t], axis=0))
        # Action index
        action_idx = np.array(actions[t], dtype=np.int32)
        # Return
        G = np.array(returns[t])
        # Loss
        loss = - G * gamma**t *
            tf.math.log(tf.reduce_sum(tf.math.multiply(pi,
                tf.one_hot(action_idx, env.action_space.n)),
                axis=1))

    # Calculate gradients and update parameters
    grads = tape.gradient(loss, p_net.trainable_variables)
    optimizer.apply_gradients(zip(grads,
        p_net.trainable_variables))
```

REINFORCE with a baseline  $b(S_t)$ :

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t)) \frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

Choose  $b(S_t) = \hat{v}(S_t)$  the state-value function:

$$\theta_{t+1} = \theta_t + \alpha(G_t - \hat{v}(S_t)) \frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

- ▶ Baseline leaves the expected value unchanged (unbiased)
- ▶ Can significantly reduce the variance
- ▶ Can improve speed of learning



## REINFORCE with Baseline (episodic)

Input: A policy  $\pi(a|s, \theta)$ ; step size  $\alpha_\theta > 0$

Input: A state-value function  $\hat{v}(s, w)$ ; step size  $\alpha_w > 0$

Initialize parameters  $\theta \in \mathbb{R}^d$ ,  $w \in \mathbb{R}^d$  arbitrarily

Loop forever (for each episode):

    Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ ,

    Loop for each step of the episode  $t = 0, 1, \dots, T - 1$  :

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$$

$$\delta \leftarrow G - \hat{v}(S_t, w)$$

$$w \leftarrow w + \alpha_w \delta \nabla \hat{v}(S_t, w)$$

$$\theta \leftarrow \theta + \alpha_\theta \gamma^t G \nabla \ln \pi(A_t | S_t, \theta)$$

# REINFORCE with Baseline in Python [cont'd]

## Learning from episode:

```
for t in range(len(returns)):
    with tf.GradientTape() as p_tape:
        with tf.GradientTape() as v_tape:
            pi = policy_network(np.expand_dims(states[t], axis=0))
            action_idx = np.array(actions[t], dtype=np.int32)
            # Value function
            v = value_network(np.expand_dims(states[t], axis=0))
            # Return
            G = np.array(returns[t])
            delta = G - v
            # Losses for policy and value networks
            p_loss = -delta * gamma**t * tf.math.log(tf.reduce_sum(pi))
            v_loss = -delta * v

    # Calculate gradients and update parameters for policy network
    p_grads = p_tape.gradient(p_loss, p_net.trainable_variables)
    p_optimizer.apply_gradients(zip(p_grads,
                                    p_net.trainable_variables))

    # Calculate gradients and update parameters for value network
    v_grads = v_tape.gradient(v_loss, v_net.trainable_variables)
    v_optimizer.apply_gradients(zip(v_grads,
                                    v_network.trainable_variables))
```

## One-Step Actor-Critic:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha(G_t - \hat{v}(S_t)) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta_t)} \\ &= \theta_t + \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta_t)} \\ &= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta_t)}\end{aligned}$$

- ▶ Analogous to TD, SARSA and Q-Learning for tabular methods
- ▶ Improve on slow learning of MC methods
- ▶ Useful for non-episodic, continuous problems

# Policy Gradient Methods

## One-Step Actor-Critic

Input: A policy  $\pi(a|s, \theta)$ ; step size  $\alpha_\theta > 0$

Input: A state-value function  $\hat{v}(s, w)$ ; step size  $\alpha_w > 0$

Initialize parameters  $\theta \in \mathbb{R}^d$ ,  $w \in \mathbb{R}^d$  arbitrarily

Loop forever (for each episode):

    Initialize  $S$  (first state of episode);  $l \leftarrow 1$

    Loop while  $S$  not terminal (for each time step):

        Sample  $A$  from  $\pi(\cdot|S, \theta)$

        Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

$w \leftarrow w + \alpha_w \delta \nabla \hat{v}(S_t, w)$

$\theta \leftarrow \theta + \alpha_\theta G \nabla \ln \pi(A_t|S_t, \theta)$

$S \leftarrow S'; l \leftarrow \gamma l$

# Reference Implementations

## Stable Baselines

- ▶ Reference Python implementation of RL algorithms
- ▶ Pre-trained agents ("Baselines Zoo")
- ▶ Originally developed at OpenAI
- ▶ Pointers to additional learning materials

<https://stable-baselines.readthedocs.io/en/master/>

## Gymnasium

- ▶ A standard programming interface (API) for RL environments
- ▶ Collection of reference environments
- ▶ Originally developed at OpenAI

<https://gymnasium.farama.org/index.html>



<https://www.alphagomovie.com>

Google's DeepMind division became famous in 2017 when it trained a computer to beat the human world champion at the game of Go. An award-winning full-length documentary has been made about this achievement.

<https://www.alphagomovie.com>  
<https://www.youtube.com/watch?v=WXuK6gekU1Y>

The introductory paper by David Silver and others in Nature should be easy to understand: "Mastering the game of Go without human knowledge". Nature. 550 (7676): 354–359

<https://www.nature.com/articles/nature24270>

## David Silver, UCL and Google DeepMind

Dr. Silver (<https://www.davidsilver.uk/>) of University College London has an excellent introductory course on reinforcement learning with class materials (from 2015) and lectures in a YouTube playlist. Updated courses (2018, 2021) are available on the DeepMind YouTube channel. The 2021 course include topics on deep reinforcement learning.

<https://www.davidsilver.uk/teaching/>

<https://www.youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzF0bQ>.

[https://www.youtube.com/@Google\\_DeepMind/playlists](https://www.youtube.com/@Google_DeepMind/playlists).

# Additional Materials II

## UC Berkeley

UC Berkeley hosted a Deep RL Bootcamp in 2017 with slides and lecture videos available online. Additionally, UC Berkeley's course on Deep RL is available online, with lecture slides and videos of past years.

<https://sites.google.com/view/deep-rl-bootcamp/lectures>

<https://rail.eecs.berkeley.edu/deeprlcourse/>

## Denny Britz

Formerly at the Google AI team, Denny Britz applied RL algorithms to financial markets and trading. He has a interesting blog, and a GitHub repository with resources and algorithm implementations of popular RL algorithms.

<https://dennybritz.com/>

<https://github.com/dennybritz/reinforcement-learning>



# Additional Materials III

## Massimiliano Patacchiola, Cambridge University

Dr. Patacchiola is a postdoc at Cambridge University. He has written a series of excellent blog posts on reinforcement based on the book "Artificial Intelligence – A Modern Approach" by Russell and Norvig. There are lots of illustrations and pointers to implementation and code in multiple languages.

[https://github.com/mpatacchiola/  
dissecting-reinforcement-learning](https://github.com/mpatacchiola/dissecting-reinforcement-learning)

## Pascal Poupart, University of Waterloo

Dr. Poupart has made available videos and all course materials for all lectures for a course on reinforcement learning at UWaterloo.

[https://www.youtube.com/playlist?list=  
PLdAoL1zKcqTXFJniO3Tqqn6xMBBL07EDc](https://www.youtube.com/playlist?list=PLdAoL1zKcqTXFJniO3Tqqn6xMBBL07EDc)

[https://cs.uwaterloo.ca/~ppoupart/teaching/  
cs885-spring18/schedule.html](https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-spring18/schedule.html)

# Additional Materials IV

## Andrew Ng, Stanford University

Dr. Ng (<https://www.andrewng.org/>) has taught an introductory class on reinforcement learning, as part of a broader course on machine learning.

<https://www.youtube.com/watch?v=RtxI449ZjSc>

<https://www.youtube.com/playlist?list=PLA89DCFA6ADACE599>

## Andrei Karpathy, OpenAI, formerly Tesla

Andrei Karpathy (<https://karpathy.ai/>) was a founding member of OpenAI (makers of ChatGPT and Dall-E) and later became the Tesla lead for their Autopilot program. An early blog post by Andrei Karpathy on RL is at the introductory level.

<https://karpathy.github.io/2016/05/31/rl/>

## Lilian Weng, OpenAI

Dr. Weng (<https://lilianweng.github.io/>) is a lead researchers at OpenAI (makers of ChatGPT and Dall-E). She has written an early blog post on RL and another one on policy gradient algorithms.

`https://lilianweng.github.io/posts/2018-02-19-rl-overview/`

`https://lilianweng.github.io/posts/2018-04-08-policy-gradient/`

## OpenAI

OpenAI (<https://openai.com>; makers of ChatGPT and Dall-E) post regularly on their blog, on all things deep learning and also reinforcement learning. The blog posts are easy introduction to a variety of analytics topics.

<https://openai.com/blog/openai-baselines-ppo/>

<https://openai.com/blog/evolved-policy-gradients/>

<https://openai.com/blog/evolution-strategies/>