Reinforcement Learning

Lecture 3 Deep Reinforcement Learning

Nan Ye

School of Mathematics and Physics
The University of Queensland

Roadmap

- Introduction and overview

  motivation, bandits, big picture
- Classical ideas

  temporal difference methods, policy gradient, . . .
- Deep Reinforcement learning

  neural networks, DQN, DDPG, . . .
- Advanced techniques

  representation learning, stabilization, few-shot learning
- Applications

  AlphaGo, AlphaTensor, . . .

Function Approximation

Recall: function approximation is needed to scale to large S and A

e.g., use parametric Q_θ instead of tabular Q in Q-learning

deep RL: use deep neural networks (DNNs) as function approximators

achieved impressive performance on many hard problems

conceptually: plug DNNs in as function approximators in algorithms

in practice: many tricks needed

Artificial Neural Nets (ANNs)

neural networks are highly expressive parametric functions

a neural network (linear function):

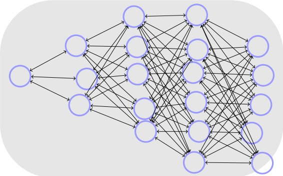
$$f(\mathbf{x};\mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_d x_d$$

another one (logistic model):

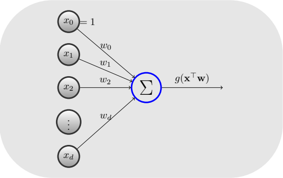
$$f(\mathbf{x};\mathbf{w}) = \sigma(w_0 + w_1 x_1 + \ldots + w_d x_d)$$

and RBF networks:

$$f(\mathbf{x};\mathbf{w}) = \sum_{i=1}^{N} a_i \rho(\mathbf{x} - \mathbf{c}_i), \quad \text{where } \mathbf{w} = \{a_{1:N}, \mathbf{c}_{1:N}\}$$



ANN



artificial neuron

- ANNs
  - interconnected simple computational units (neurons)
  - universal approximators
  - often trained to minimize loss
- Neurons
  - input from incoming edges, output along outgoing edges
  - computes nonlinearly transformed weighted input sum $g(\mathbf{w}^\top \mathbf{x})$
  - nonlinearity g known as activation/transfer function

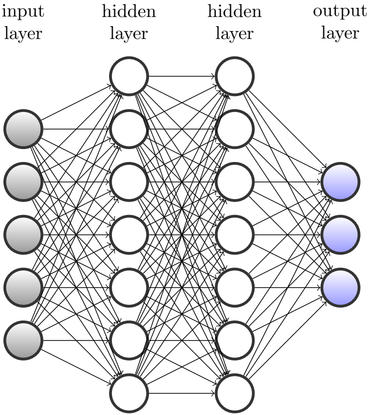
architecture	**activation**	**optimizer**	**software**
MLP	threshold	SGD	PyTorch
CNN	sigmoid	AdaGrad	TensorFlow
RNN	ReLU	RMSprop	Google JAX
ResNet	ELU	AdaDelta	Keras
transformer	GELU	Adam	MXNet
...	...	...	...

$\uparrow$

often first-order methods

gradients computed using automatic differentiation

Multilayer Perceptron (MLP)
aka multilayer feedforward neural network

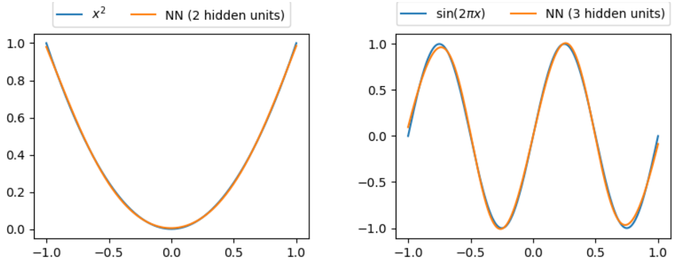


- neurons organized in layers
- forward edges only (from input neurons to output neurons)
- single-hidden layer sigmoid MLPs are universal approximators

Universal approximation property of single hidden layer neural net

$$\sum_{i=1}^{m} \alpha_i \sigma(w_i x + b_i) + \beta,$$

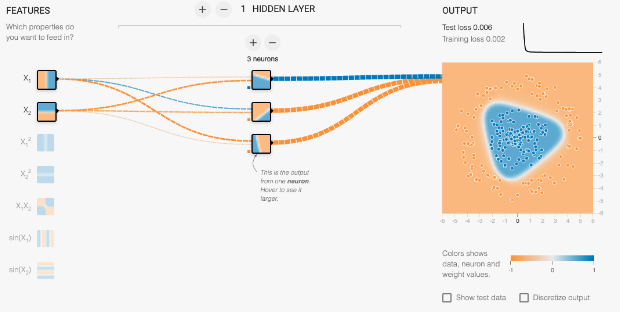
where $\sigma(u) = 1/(1 + e^{-x})$ is the sigmoid function.



$$x^2 \approx 2.2\sigma(-3.15x - 3) + 2.2\sigma(3.15x - 3) - 0.205, \; x \in [-1, 1].$$

$$\sin(x) \approx 10.9\sigma(-6.35x - 3.05) - 10.9\sigma(6.35x - 3.05) - 36.6\sigma(-1.3x) + 18.23, \; x \in [-1, 1].$$

Feature Learning



https://playground.tensorflow.org/

a sigmoid unit approximately learns the concept of a circular area in 2D plane

- In deep neural networks (> 1 hidden layer), deeper layers are capable of learning higher-level features.
- This allows learning accurate models from raw features without handcrafting high-level features.

PyTorch

```
# define an MLP with 10 ReLU hidden units
net = nn.Sequential(nn.Linear(2, 10),
                    nn.ReLU(),
                    nn.Linear(10, 1))
# specify the optimization algorithm
optimizer = optim.SGD(net.parameters(), lr=0.001)
# define the loss function
mse = MSELoss()
# use a dataloader for sampling mini-batches
dataloader = DataLoader(DatasetWrapper(X, y), batch_size=10, shuffle=True)
# train it
for epoch in range(nepochs):
  for i, (X_batch, y_batch) in enumerate(dataloader):
      optimizer.zero_grad()
      loss = mse(net(X_batch), y_batch)
      loss.backward()
      optimizer.step()
```

MLP regression in PyTorch

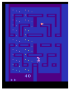
try me: https://tinyurl.com/2oxmal74

Deep Q-Networks (DQN) for Atari Games

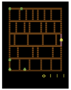
recall...

Adventure

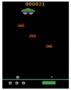
Air Raid



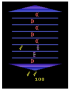
Alien



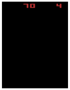
Amidar



Assault



Asterix



Asteroids

Atlantis

Atari

https://gymnasium.farama.org/environments/atari/

recall...

Algorithm Q-learning with function approximation

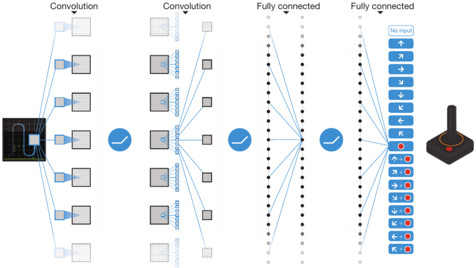
Initialise the state-action value function Q_θ
while termination condition not met **do**
    Execute an appropriate behavior policy to obtain a new experience (s, a, s', r)
    Perform TD update

$$\theta \leftarrow \theta - \alpha(Q_\theta(s,a) - r - \gamma \max_{a'} Q_\theta(s', a'))\, \nabla\, Q_\theta(s,a).$$

Key ideas in DQN

- A deep CNN approximation $Q_\theta(s, a)$
  - state consists of the last 4 frames, and Q_θ is a CNN that takes in a preprocessed representation $\phi(s)$, and outputs the action probabilities.
- Experience replay
  - instead of using current observed transition to update model, use a randomly sampled minibatch from the experience memory
- Separate target Q-network
  - A separate Q-network Q_{θ^-} is used to compute the TD target, and Q_{θ^-} is updated to Q_θ after a given number of steps

Mnih et al., Human-level control through deep reinforcement learning, 2015



A schematic illustration of a CNN for Q_θ

Architecture for Q_θ in (Mnih et al., 2015)

- Input: $84 \times 84 \times 4$ stack of last 4 frames (after an RGB-to-gray conversion and scaling)
- 1st hidden layer: Conv(8x8, 32, S=4) + ReLU
- 2nd hidden layer: Conv(4x4, 32, S=2) + ReLU
- 3rd hidden layer: Conv(3x3, 64, S=1) + ReLU
- 4th hidden layer: FC-512 + ReLU
- output layer: softmax over actions

Algorithm DQN (Mnih et al., 2015)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q with weights $\theta^- = \theta$
for episode=1 to M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$.
  for $t = 1$ to T **do**
    Select a_t randomly w.p. ϵ and as $\operatorname{argmax}_a Q_\theta(\phi(s_t), a)$ w.p. $1 - \epsilon$
    Execute a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $\{(\phi_j, a_j, r_j, \phi_{j+1}) : j \in J\}$ from D
    For $j \in J$, set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j \\ r_j + \gamma \max_{a'} Q_{\theta^-}(\phi_{j+1}, a'), & \text{otherwise.} \end{cases}$
    Perform a gradient descent step on $\frac{1}{|J|} \sum_{j \in J} (y_j - Q_\theta(\phi_j, a_j))^2$ wrt θ.
    Set $\theta^- = \theta$ if t is a multiple of C

experience collection, incremental update

Rainbow DQN

Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel DeepMind · **Joseph Modayil** DeepMind · **Hado van Hasselt** DeepMind · **Tom Schaul** DeepMind · **Georg Ostrovski** DeepMind

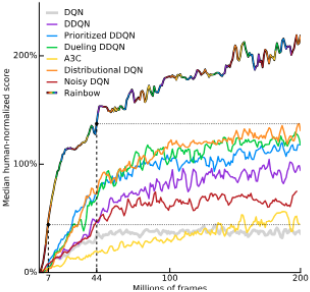
Will Dabney DeepMind · **Dan Horgan** DeepMind · **Bilal Piot** DeepMind · **Mohammad Azar** DeepMind · **David Silver** DeepMind

Abstract

The deep reinforcement learning community has made several independent improvements to the DQN algorithm. However, it is unclear which of these extensions are complementary and can be fruitfully combined. This paper examines six extensions to the DQN algorithm and empirically studies their combination. Our experiments show that the combination provides state-of-the-art performance on the Atari 2600 benchmark, both in terms of data efficiency and final performance. We also provide results from a detailed ablation study that shows the contribution of each component to overall performance.

Introduction

The many recent successes in scaling reinforcement learning (RL) to complex sequential decision-making problems were kick-started by the Deep Q-Networks algorithm (DQN; Mnih et al. 2013, 2015). Its combination of Q-learning with convolutional neural networks and experience replay enabled it to learn, from raw pixels, how to play many Atari



a combination of six tricks (Hessel et al., 2018)

Hessel et al., Rainbow: Combining improvements in deep reinforcement learning, 2018

Deep Policy Optimization

many DNN-based policy optimization algorithms

- DDPG (Deep Deterministic Policy Gradient)
- TRPO (Trust Region Policy Optimization)
- PPO (Proximal Policy Optimization)
- . . .

more than just plugging in DNNs into existing algorithms

DDPG

Deterministic policy gradient (DPG) theorem (Silver et al., 2014)

For a deterministic policy $\pi_\theta : S \to A$ where A is continuous, it holds that

$$\nabla V(\theta) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \rho_{\pi_\theta}} \left[\underbrace{[\nabla_\theta \pi_\theta(s)]^\top}_{d_\theta \times d_A} \underbrace{\nabla_a Q_{\pi_\theta}(s,a)|_{a=\mu_\theta(s)}}_{d_A \times 1} \right],$$

where $\rho_{\pi_\theta}(s)$ is the discounted state distribution for π_θ.

NB The Jacobian $\nabla_\theta \pi_\theta(s)$ has shape $d_A \times d_\theta$, where d_A and d_θ are the dimensions of A and θ.

computing deterministic policy gradient is more sample efficient that the stochastic one

- *people thought this is not doable before DPG*

this requires computing Q_{π_θ}. how? Q-learning!

Silver et al., Deterministic policy gradient algorithms, 2014

Algorithm DDPG (Lillicrap et al., 2015)

Randomly initialize critic $Q_\phi(s, a)$ and actor $\pi_\theta(s)$ with weights ϕ and θ
Initialize target networks $Q_{\phi'}$ and $\pi_{\theta'}$ with weights $\phi' \leftarrow \phi$ and $\theta' \leftarrow \theta$
Initialize replay buffer R
for episode = 1 to M **do**
    Initialize a random process N for action exploration
    Receive initial observation state s_0
    for $t = 0$ to $T - 1$ **do**
        Select action $a_t = \pi_\theta(s_t) + \epsilon_t$, where $\epsilon_t \sim N$
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_i = r_i + \gamma Q_{\phi'}(s_{i+1}, \pi_{\theta'}(s_{i+1}))$
        Update critic Q_ϕ by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q_\phi(s_i, a_i))^2$.
        Update the actor π_θ using the sampled policy gradient:

$$\frac{1}{N}\sum_i [\nabla_\theta \, \pi_\theta(s_i)]^\top \, \nabla_a \, Q_\phi(s_i, a)|_{a=\pi_\theta(s_i)}$$

        Update the target networks: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$, $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$

sample-efficient, but brittle – very sensitive to the hyperparameters

Lillicrap et al., Continuous control with deep reinforcement learning, 2015

Software

CleanRL (Clean Implementation of RL Algorithms)

license MIT | tests passing | docs success | discord 46 online | Views 12k | code style black | imports isort | Models Huggingface | Open in Colab

CleanRL is a Deep Reinforcement Learning library that provides high-quality single-file implementation with research-friendly features. The implementation is clean and simple, yet we can scale it to run thousands of experiments using AWS Batch. The highlight features of CleanRL are:

- Single-file implementation
  - *Every detail about an algorithm variant is put into a single standalone file.*
  - For example, our `ppo_atari.py` only has 340 lines of code but contains all implementation details on how PPO works with Atari games, **so it is a great reference implementation to read for folks who do not wish to read an entire modular library**.
- Benchmarked Implementation (7+ algorithms and 34+ games at https://benchmark.cleanrl.dev)
- Tensorboard Logging
- Local Reproducibility via Seeding
- Videos of Gameplay Capturing
- Experiment Management with Weights and Biases
- Cloud Integration with docker and AWS

https://github.com/vwxyzjn/cleanrl

single-file implementation!

- Tianshou: https://github.com/thu-ml/tianshou
- RLlib: https://github.com/ray-project/ray/tree/master/rllib/
- Stable Baselines3: https://github.com/DLR-RM/stable-baselines3
- Spinning Up (educational): https://github.com/openai/spinningup

Roadmap

- Introduction and overview

  motivation, bandits, big picture
- Classical ideas

  temporal difference methods, policy gradient, . . .
- Deep Reinforcement learning

  neural networks, DQN, DDPG, . . .
- Advanced techniques

  representation learning, stabilization, few-shot learning
- Applications

  AlphaGo, AlphaTensor, . . .

References I

Hessel, M. et al. (2018). Rainbow: Combining improvements in deep reinforcement learning. In: *Thirty-second AAAI conference on artificial intelligence*.

Lillicrap, T. P. et al. (2015). Continuous control with deep reinforcement learning. In: *arXiv preprint arXiv:1509.02971*.

Mnih, V. et al. (2015). Human-level control through deep reinforcement learning. In: *Nature* 518.7540, pp. 529–533.

Schulman, J. et al. (2015). Trust region policy optimization. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1889–1897.

Schulman, J. et al. (2017). Proximal Policy Optimization Algorithms. In: *arXiv preprint arXiv:1707.06347*.

Silver, D. et al. (2014). Deterministic policy gradient algorithms. In: *ICML*.