# Batch Reinforcement Learning

Tobias Weber

07.07.2020

Department of Statistics
Master Seminar ”Reinforcement Learning” - SS 20

# Agenda i

1. Introduction
2. Offline Reinforcement Learning
3. Challenges
4. Offline Policy Evaluation
5. Algorithms

# Agenda ii

# Introduction

# Online & Offline Policy Reinforcement Learning

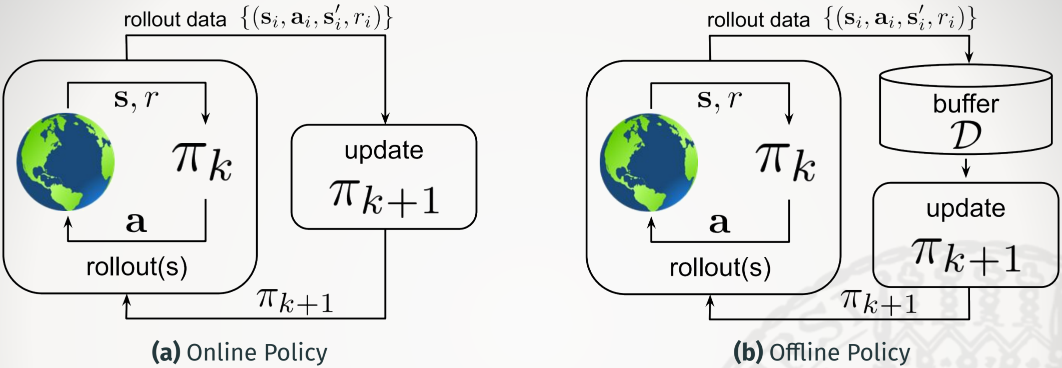



**Figure 1:** Architectures for Online & Offline Policy Reinforcement Learning.

*Credits: Levine et al. 2020*

# Problems with interactive RL (Fu et al. 2020, Levine et al. 2020)

- Interaction with environment is necessary in frequent intervals.

→ Runtime issues & computational effort

# Problems with interactive RL (Fu et al. 2020, Levine et al. 2020)

- Interaction with environment is necessary in frequent intervals.

→ Runtime issues & computational effort

- Interaction with environment is not possible or accelerable.

→ Need for simulators or slow data collection

→ Additional problem of simulator-to-world transfer

# Problems with interactive RL (Fu et al. 2020, Levine et al. 2020)

- Interaction with environment is necessary in frequent intervals.

→ Runtime issues & computational effort

- Interaction with environment is not possible or accelerable.

→ Need for simulators or slow data collection

→ Additional problem of simulator-to-world transfer

- A lot of Data is already existing but RL needs to collect samples on each training run.

→ Low sample efficiency & redundancy

# More examples...

- Decision making in healthcare & autonomous driving
  $\rightarrow$ Agent cannot interact with environment
- Recommender systems $\rightarrow$ Insane amount of data available
- Learning dialogues $\rightarrow$ Interaction with humans would be needed
- Learning multi-tasking $\rightarrow$ Blending environments and collected data

# Offline Reinforcement Learning

# Offline Reinforcement Learning

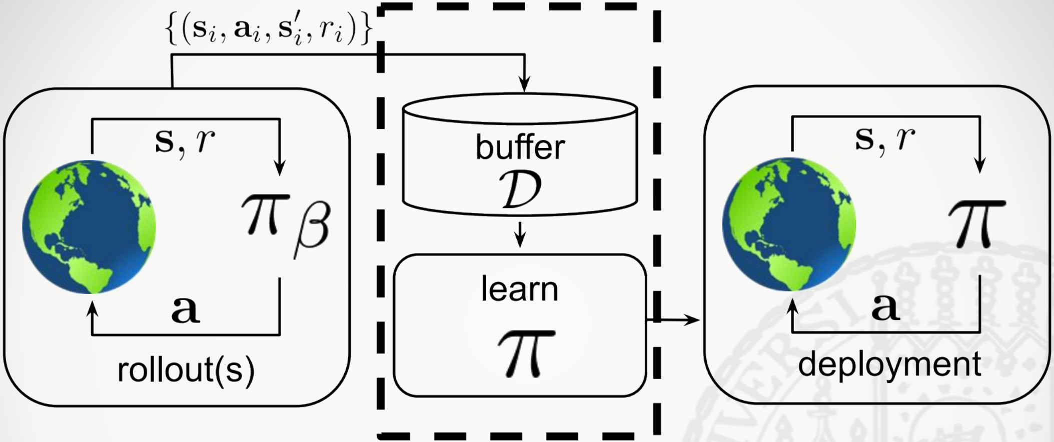



**Figure 2:** Offline Reinforcement Learning.
*Credits:* LEVINE *et al. 2020*

# Taxonomy

## Lange, Gabel, and Riedmiller 2012

- Batch Reinforcement Learning $\rightarrow$ Learning policy from $\mathcal{D}$
- Growing Batch Reinforcement Learning $\rightarrow$ Learning policy from $\mathcal{D}$ but allow environment interaction after Batches

# Taxonomy

**Lange, Gabel, and Riedmiller 2012**

- Batch Reinforcement Learning $\rightarrow$ Learning policy from $\mathcal{D}$
- Growing Batch Reinforcement Learning $\rightarrow$ Learning policy from $\mathcal{D}$ but allow environment interaction after Batches

**Current papers, e.g. Agarwal, Schuurmans, and Norouzi 2019, Levine et al. 2020, Fu et al. 2020:**

- Offline Reinforcement Learning $\rightarrow$ Learning policy from $\mathcal{D}$
- Offline Policy Reinforcement Learning $\rightarrow$ Learning policy from $\mathcal{D}$ but allow environment interaction after Batches

# Notation

| | |
|---|---|
| $\tau$ | Trajectory: $(s_1, a_1, r_1, ..., s_H, a_H, r_H)$ |
| $H$ | Length of Trajectory = Number of state, action, reward pairs |
| $G$ | Accumulated rewards / Return of a trajectory |
| $\mathcal{D}$ | Dataset with transitions $(s, a, r, s')$ |
| $\pi$ | Offline policy that is being learned |
| $\pi_\beta$ | Behavioral policy where $\mathcal{D}$ is sampled from |
| $\pi(a\|s)$ | Probability of taking action $a$ when being in state s |
| $d^\pi(s)$ | State visitation frequency of s when following $\pi$ |

# Objective & Goal

Maximize value of offline policy $\pi$:

$$V^\pi = \mathbb{E}[\sum_{t=1}^{H} \gamma^t r_t | \pi]$$

# Objective & Goal

Maximize value of offline policy $\pi$:

$$V^{\pi} = \mathbb{E}[\sum_{t=1}^{H} \gamma^t r_t | \pi]$$

or:
Maximize probability that value of $\mathcal{A}(\mathcal{D})$ is better or equal to $\pi_\beta$, where $\mathcal{A}(\mathcal{D})$ is an algorithm that produces $\pi$ under $\mathcal{D}$:

$$\max P(V^{\mathcal{A}(\mathcal{D})} \geq V^{\pi_\beta})$$

# Challenges

# Exploration Problem

- Training algorithm $\mathcal{A}$ has to rely entirely on static $\mathcal{D}$
- No means for exploration

$\rightarrow$ If $\pi_\beta$ does not contain areas of high-reward regions then they may remain undiscovered

$\rightarrow$ Structure of $\mathcal{D}$ is critical!

# Contradict: Exploration Problem

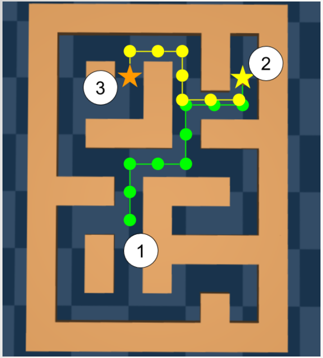



**Figure 3:** Trajectories in a maze.
*Credits: Fu et al. 2020*

- Fu et al. 2020 claims that theoretically $\pi$ can outperform $\pi_\beta$ by assembling different parts in the dataset

$\Rightarrow$ Create optimal strategy from different experiences

# Counterfactual queries

- Counterfactual queries $\approx$ "What if?"-questions
- To be better than $\pi_\beta$ different $\tau$s than in $\mathcal{D}$ need to be executed
- Learn a policy with different behavior

- ⚡ Classic supervised Machine learning: data is independent and identically distributed (i.i.d.)
- ⚡ Goal: Perform good on data of same distribution

# Distribution Shift


**(a)** Distribution of state $s$ under $\pi_\beta$ **(b)** Distribution of state $s$ under $\pi$

**Figure 4:** Distributional shift between the behavioral and evaluation policy.

# Distributional Shift Problems

- ⚡ $\pi$ may produce unexpected and erroneous actions in out-of-distribution states $s \sim d^{\pi_\beta}(s)$ (Levine et al. 2020)

$\Rightarrow$ Restrict deviation from behavioral policy, e.g. with Kullback-Leibler Divergence: $D_{KL}(\pi(a|s) \| \pi_\beta(a|s)) \leq \epsilon$

## Action Distribution Shift

- Mostly, target calculation relies on an estimate of $Q(s_{i+1}|a_{i+1})$.
- Can contain actions that aren't in $\mathcal{D} \rightarrow$ can produce high erroneous targets.

$\Rightarrow$ Greedy policy will further use actions that seem to give most reward.

$\Rightarrow$ No correctional feedback available!

# Offline Policy Evaluation

# Motivation

- General question: How to evaluate an agent or policy without contact to the environment?
- Available: Behavioral policy $\pi_\beta$

?

# Motivation

- General question: How to evaluate an agent or policy without contact to the environment?
- Available: Behavioral policy $\pi_\beta$

$\rightarrow$ $\pi_\beta$ has a different distribution than $\pi$ (Distribution shift)

$\rightarrow$ How to adapt given $\tau$ and $G$ to $\pi$?

?

# Importance Sampling (Precup 2000, Rubinstein and Kroese 2016)

- Estimate $\mathbb{E}$ of a variable $x$ under distribution $q$
- Given: samples of $x$ generated under distribution $p$

$$\mathbb{E}_{x \sim q} = \int_x x q(x) dx = \int_x x \frac{q(x)}{p(x)} p(x) dx = \mathbb{E}_{x \frac{q(x)}{p(x)} \sim p}$$

- Approximation through samples:

$$\approx \frac{1}{n} \sum_{i=1}^{n} x_i \frac{q(x)}{p(x)}$$

# Adapting Importance Sampling to RL (Precup 2000)

The probability for a trajectory $\tau$ is defined as follows:

$$\begin{aligned} p(\tau|s=s_1) =& p(a_1|s_1)p(r_1|s_1,a_1)p(s_2|s_1,a_1) \\ & \cdot \ldots \cdot p(a_{H-1}|s_{H-1})p(r_{H-1}|s_{H-1},a_{H-1})p(s_H|s_{H-1},a_{H-1}) \\ =& \prod_{t=1}^{H-1} p(a_t|s_t)p(r_t|s_t,a_t)p(s_{t+1}|s_t,a_t) \end{aligned}$$

When given a policy $\pi$:

$$p(\tau|\pi,s=s_1) = \prod_{t=1}^{H-1} \pi(a_t|s_t)p(r_t|s_t,a_t)p(s_{t+1}|s_t,a_t)$$

# Adapting Importance Sampling to RL (PRECUP 2000)

For Importance Sampling the fraction of two probabilities is needed:

$$\frac{p(\tau|\pi)}{p(\tau|\pi_\beta)} = \prod_{t=1}^{H-1} \frac{\pi(a_t|s_t)}{\pi_\beta(a_t|s_t)} \frac{p(r_t|s_t, a_t)}{p(r_t|s_t, a_t)} \frac{p(s_{t+1}|s_t, a_t)}{p(s_{t+1}|s_t, a_t)}$$

$$= \prod_{t=1}^{H-1} \frac{\pi(a_t|s_t)}{\pi_\beta(a_t|s_t)}$$

# Value of $\pi$ under Importance Sampling

$$
\begin{aligned}
V^{\pi} &\approx \frac{1}{n}\sum_{n}^{j=1}\frac{p(\tau_j|\pi)}{p(\tau_j|\pi_\beta)}G(\tau_j) \\
&= \frac{1}{n}\sum_{n}^{j=1}(\prod_{t=1}^{H-1}\frac{\pi(a_{j,t}|s_{j,t})}{\pi_\beta(a_{j,t}|s_{j,t})})G(\tau_j) \\
&= \frac{1}{n}\sum_{n}^{j=1}(\prod_{t=1}^{H-1}\frac{\pi(a_{j,t}|s_{j,t})}{\pi_\beta(a_{j,t}|s_{j,t})})(\sum_{t=0}^{H}\gamma^t r_{j,t})
\end{aligned}
$$

$\rightarrow$ Trajectories & returns with higher probability in $\pi$ will be upweighted

$\rightarrow$ Trajectories & returns with lower probability in $\pi$ will be downweighted

# Algorithms

# Linear Function Approximation

# Preliminaries: Basis Functions

Idea: Transform states and action via basis functions:

$$\phi(s,a) = \begin{pmatrix} \phi_1(s,a) \\ \phi_2(s,a) \\ \dots \\ \phi_k(s,a) \end{pmatrix}$$

For a given set of state-action pairs:

$$\Phi(s,a) = \begin{pmatrix} \phi(s_1,a_1) \\ \phi(s_2,a_2) \\ \dots \\ \phi(s_n,a_n) \end{pmatrix}$$

Linear Value Function Approximation:

$$\hat{Q} = \Phi w$$

# Bellman Residual Minimizing Approximation (Lagoudakis and Parr 2003)

$Q$ as solution for the Bellman equation:

$$Q = \mathcal{R} + \gamma P^{\pi} Q$$

Insert Linear approximation:

$$\Phi w \approx \mathcal{R} + \gamma P^{\pi} \Phi w \implies (\Phi - \gamma P^{\pi} \Phi) w \approx \mathcal{R}$$

Solving the linear system via least-squares method:

$$w = ((\Phi - \gamma P^{\pi} \Phi)^T (\Phi - \gamma P^{\pi} \Phi))^{-1} (\Phi - \gamma P^{\pi} \Phi)^T \mathcal{R}$$

# Least squares temporal difference Q-learning (Lagoudakis, Parr, and Littman 2002)

- Projected Fixed-Point iteration method
- Orthogonal least squares projection $\Pi = \Phi(\Phi^T\Phi)^{-1}\Phi^T$

$$
\begin{aligned}
Q_{i+1} &= \Pi Q_i \\
\Phi w &= \Pi(\mathcal{R} + \gamma P^\pi \Phi w) \\
\Phi w &= \Phi(\Phi^T\Phi)^{-1}\Phi^T(\mathcal{R} + \gamma P^\pi \Phi w)
\end{aligned}
$$

- Transform to linear system that can be solved

$$
\implies \Phi^T(\Phi - \gamma P^\pi \Phi)w = \Phi^T\mathcal{R} \implies Aw = b
$$

# Least Squares Policy Iteration (Lagoudakis and Parr 2003)

- LSTDQ $\rightarrow$ One step of optimization
- LSPI:
  $\rightarrow$ Execute LSTDQ in a loop and update $w$
  $\rightarrow$ Stopping criterium: Distance of $w_{old}$ to $w_{new}$ is smaller than given $\epsilon$

# Function Approximation via Deep Learning

# Classic algorithms

## Neural Fitted Q Iteration (Riedmiller 2005)

- Using a Multilayer-Perceptron as Function Approximator
- Update using full gradient descent for generalization
- Cost approach: Minimize transition costs

# Classic algorithms

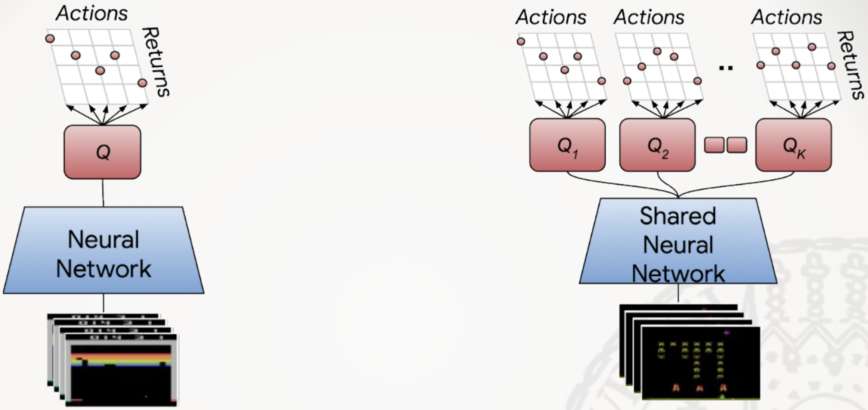
## Neural Fitted Q Iteration (Riedmiller 2005)

- Using a Multilayer-Perceptron as Function Approximator
- Update using full gradient descent for generalization
- Cost approach: Minimize transition costs

## Deep Q Network (Mnih et al. 2015)

- Algorithm that allows playing Atari games from visual inputs
- Adds experience replay buffer
- Adds policy & target network
- Lots of extensions, e.g. Rainbow (Hessel et al. 2017)

# Ensemble DQN (Anschel, Baram, and Shimkin 2016)

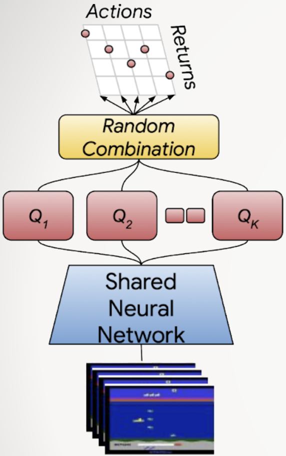



**(a)** DQN

**(b)** Ensemble DQN

**Figure 5:** Architectures for DQN & Ensemble DQN.
*Credits: Agarwal, Schuurmans, and Norouzi 2019*

# Random Ensemble Mixture (AGARWAL, SCHUURMANS, and NOROUZI 2019)




- Q value estimate: Random convex combination of Q estimates of $k$ heads

$\rightarrow$ $Q_\theta(s,a) = \sum_k \alpha_k Q_\theta^k(s,a)$

$\rightarrow$ where $\alpha \in \mathbb{R}^k$ given $\sum_k a_k = 1 \wedge \forall a_k : a_k \geq 0$

- $\alpha$ is only relevant for training!
  $\rightarrow$Using average of heads for inference

# Other approaches

- `Quantile Regression`: Learning a distribution of returns (Dabney et al. 2017)
- `QT-OP`: Scaleable approach for vision based robotic manipulation (Kalashnikov et al. 2018)
- `BEAR`: Tackling distributional shift (Kumar et al. 2019)
- `MOReL` : Model-Based Offline Reinforcement Learning (Kidambi et al. 2020)

# Practice Project

# My Project & Project goals

**Main goal:** Explore different algorithms in the domain of Offline RL

**Target:** First Atari Games, then Lunar Lander

## Subgoals

- Handle & transform visual inputs
- Efficient data collection & persistent storage
- Efficient computing on GPU
- Convenient logging capabilities

# Tools

# Tools

OpenAI

PyTorch

TensorBoard

# Check out my GitHub repository!

https://github.com/saiboxx/offline-reinforcement-learning

# The Data

# What data do we need?

$\Rightarrow$ At least so much that $\tau$ can be recreated:

$\rightarrow$ States of the game

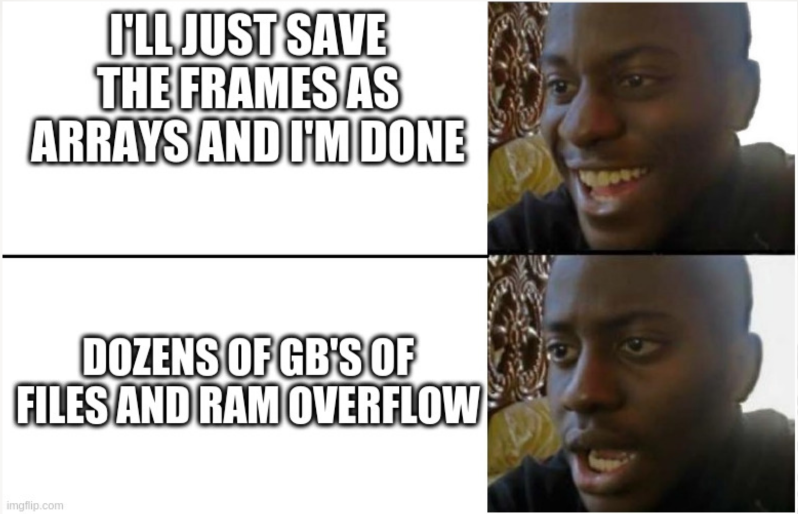
$\rightarrow$ Actions taken

$\rightarrow$ Rewards

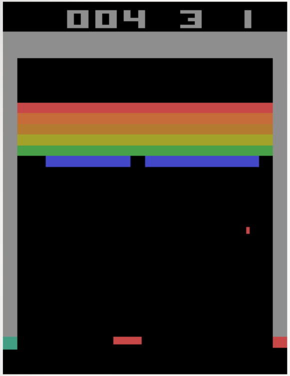
$\rightarrow$ Done Booleans

$\rightarrow$ New state $s'$ does not need to be saved explicitly!

# First findings...



# But why exactly?




- 1 pixel = value from to 255
- Smallest fitting datatype = uint8

  $\Rightarrow$ Minimum of 8 bit $\approx$ 1 byte
- 1 Frame has 210 $\times$ 160 $\times$ 3 pixels = 100.800 pixels

$\Rightarrow$ 1 Frame needs atleast 100.800 bytes = 100 KB

- This doesn't scale well...

  $\Rightarrow$10 Frames = 1 MB

  $\Rightarrow$1.000 Frames = 100 MB

  $\Rightarrow$10.000 Frames = 1 GB

  $\Rightarrow$1.000.000 Frames = 100 GB

# Image transformation & compression is necessary!

**Preprocessing**

- Cropping: Cut off unnecessary parts of the screen
- Grayscale: Reduce number of channels to one
- Black & White: Value of pixel is whether 0 or 255
- Resizing: Downsampling to $80 \times 80$ pixels

$\Rightarrow$ Result is of shape $80 \times 80 \times 1$ pixels = 6400 pixels

$\Rightarrow$ Save as `.png`-file for better compression

# Is this the perfect solution?

- 1 mio steps = 1 mio image files
- Literally killing the filesystem
- Extremely many I/O Operations

**What to do next?**

- Use a Database, e.g. Spark
- Train a Variational Autoencoder (Kingma and Welling 2013)

# Autoencoder Training Progress

# Change of project scope

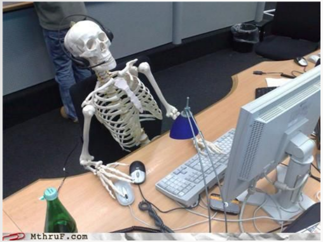
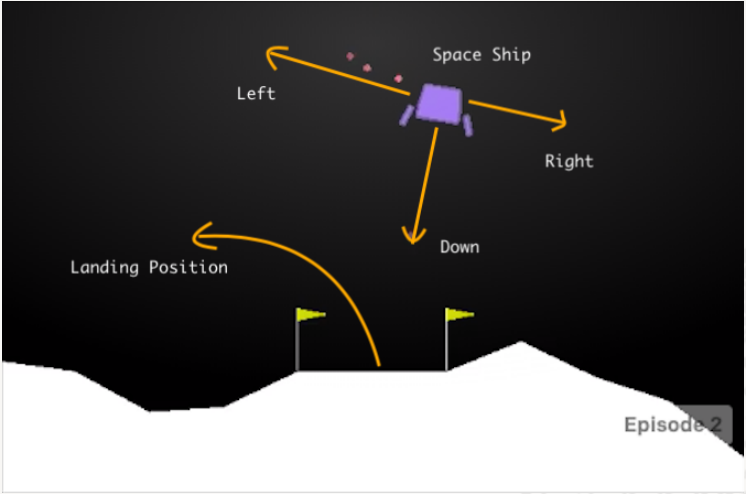



**Figure 8:** Me waiting for training to finish (2020, colorized)

- AGARWAL, SCHUURMANS, and NOROUZI 2019 trained each Atari game on 200 mio frames
- Limited time & ressource constraints

⇒ Too big of a project for a proof of concept

⇒ Change to an easier domain: Lunar Lander!




*Credits: Shiva Verma - Solving Lunar Lander*

# Offline Training Setup

# DQN Multi-Head Neural Network

```
class DQNMultiHead(nn.Module):
  def __init__(self, *args, **kwargs):
    super(DQNMultiHead, self).__init__()
    [...]
    self.elu = nn.ELU()

    self.fc1 = nn.Linear(in_features=obs_space, out_features=128)
    self.bn1 = nn.BatchNorm1d(128)
    self.fc2 = nn.Linear(in_features=128, out_features=32)
    self.bn2 = nn.BatchNorm1d(32)

    self.heads = nn.Conv1d(in_channels=32 * num_heads,
                           out_channels=action_space * num_heads,
                           kernel_size=1,
                           groups=num_heads)
```

# What's this Convolutional Layer about?

- Optimal: Processing independent heads in parallel
- Problem: How to implement parallelization in PyTorch & GPU?

$\rightarrow$ Idea: Utilize 1D-Convolutions!

☑

# Parallelizing Linear layers over Convolutions

- Transform input `(b × state_dim)` to `(b × state_dim * num_heads × 1)`

→ Length of input sequence is 1 with a lot of channels.

→ Use kernel size 1: Collapse to Linear layer.

$$out_{conv_j} = \sum_{k=1}^{C_{in}} w_k C_{in_k} + b \Rightarrow x^T w + b = out_{lin_j}$$

- Use grouped convolutions: Channels will be divided into $n$ partitions, where each partition has own weight set.

→ Output: `(b × (action_dim * num_heads) × 1)`

⇒ Reshape to preferred matrix layout

# REM Walkthrough i

- Create coefficient matrix $\alpha$ witch shape $(\texttt{num\_heads} \times \texttt{batch})$. Sum of columns is 1

```
alpha = torch.rand(self.num_heads).to(self.device)
alpha = alpha / torch.sum(alpha)
alpha = alpha.unsqueeze(-1).expand(-1, len(action))
```

- Actions shape = $(\texttt{batch}) \rightarrow (\texttt{num\_heads} \times \texttt{batch} \times 1)$

```
actions = action.unsqueeze(-1).expand(self.num_heads, -1, -1)
```

## REM Walkthrough ii

- Retrieve all Q-Values for all actions from policy and choose the taken actions via multi-index selection
- Output of network has shape $(\texttt{num\_heads} \times \texttt{batch} \times \texttt{action\_space})$
- Next, weigh the heads via $\alpha$ and sum up axis 0 $\rightarrow (\texttt{batch})$

```
state_action_values = self.policy(state).gather(2, actions).squeeze()
state_action_values = torch.sum(alpha * state_action_values, dim=0)
```

# REM Walkthrough iii

- Retrieve all Q-Values for all actions from target
- Weigh the heads via $\alpha$ and sum up axis 0 $\rightarrow$ $(\texttt{batch} \times \texttt{action\_space})$

```
all_next_states = self.target(new_state).detach()
all_next_states = torch.sum(all_next_states * alpha.unsqueeze(-1).
                            expand(-1, -1, self.action_space),
                            dim=0)
```

- Choose actions that have the highest Q-Value $\rightarrow$ $(\texttt{batch})$
- Use a mask to set all states to zero, that are the end of an episode

```
next_state_values, _ = torch.max(all_next_states, dim=1)
next_state_values[done] = 0
```

# REM Walkthrough iv

- Calculate expected Q-values
- Use difference between expected Q-Values and Q-Values from policy network as loss

```
expected_state_action_values = (next_state_values * self.gamma) +
                                    reward

loss = self.loss(state_action_values, expected_state_action_values)
```

# REM Walkthrough v

- Do one step of gradient descent with clipping

```
self.optimizer.zero_grad()
loss.backward()
for param in self.policy.parameters():
  param.grad.data.clamp_(-1, 1)
self.optimizer.step()
```

- Update target network on a fixed interval

```
if self.batches_done % self.target_update_steps == 0:
  self.target.load_state_dict(self.policy.state_dict())
```

# Additional advice

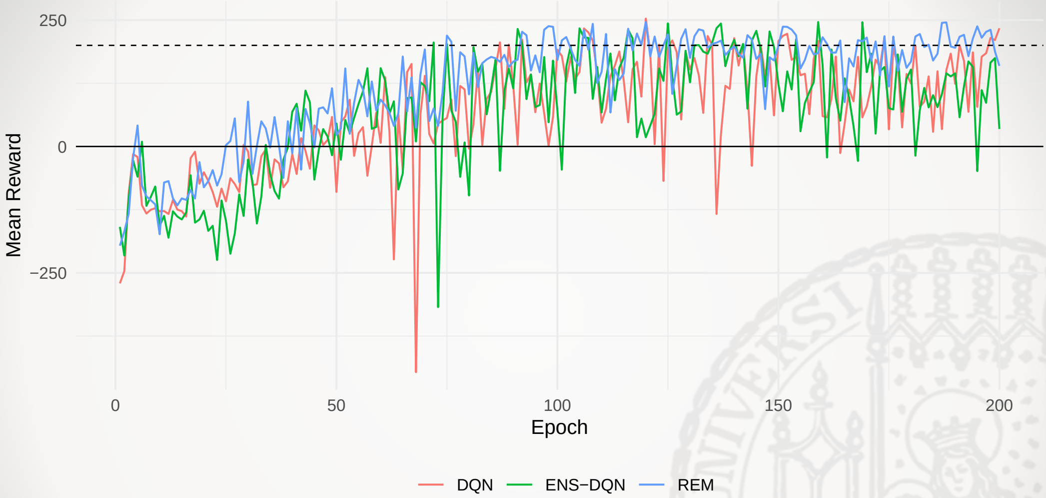
- ! Skip rendering of environment!
- ! Limit stdout and logging!
- ! Limit I/O Ops
  - $\rightarrow$ load dataset to memory
  - $\rightarrow$ if it is too big: load until memory is full!

## PyTorch specific

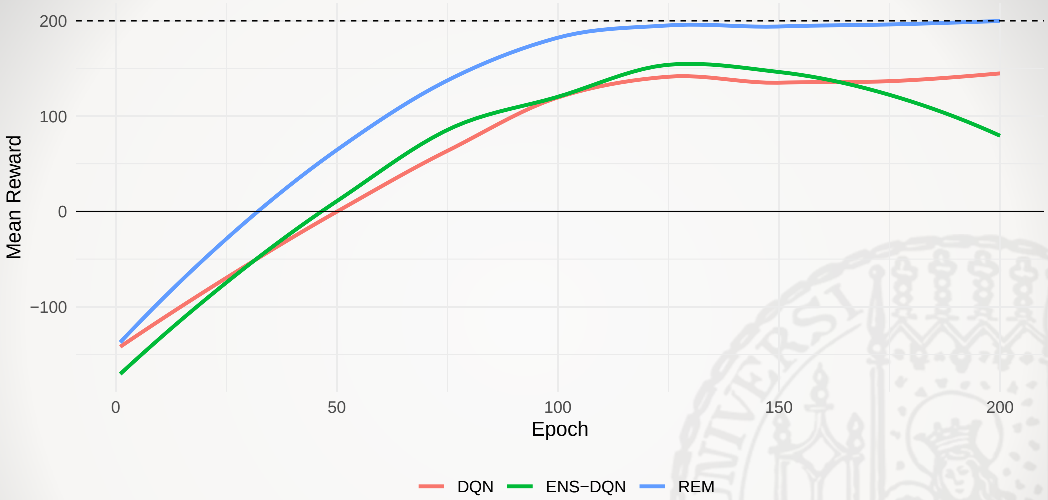
- Utilize custom Dataset and Dataloader
- Set `num_workers` $\geq$ number of cpu cores
- Set `pin_memory` to True
  - $\rightarrow$ workers will prepare data for faster loading into VRAM

# Experiments

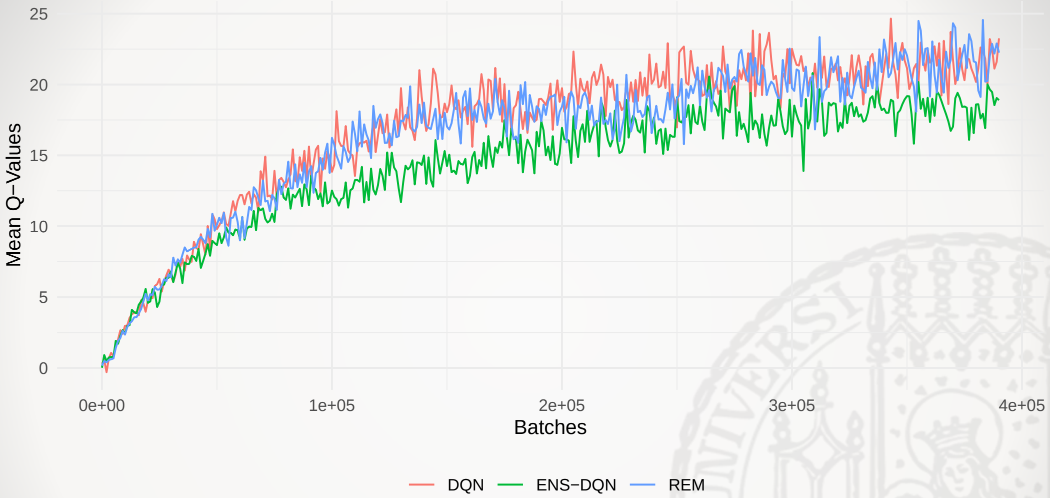
# Mean Reward - 512 Batch - 200 Epochs - DQN Data



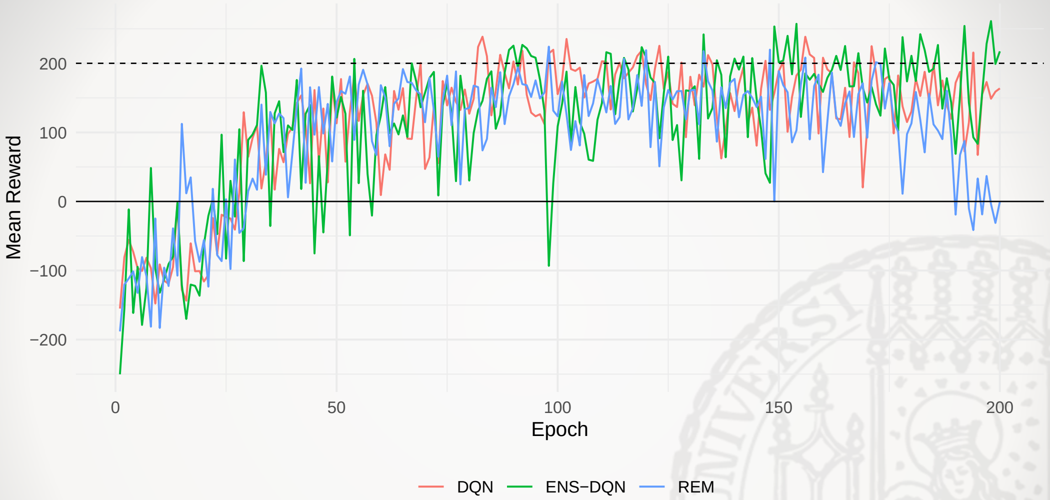
# Mean Reward Smooth - 512 Batch - 200 Epochs - DQN Data



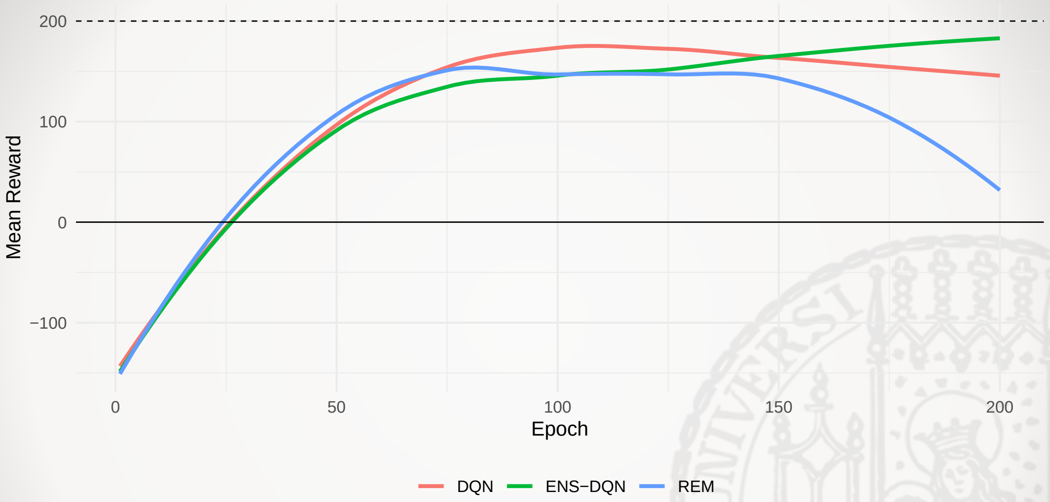
# Q-Values - 512 Batch - 200 Epochs - DQN Data



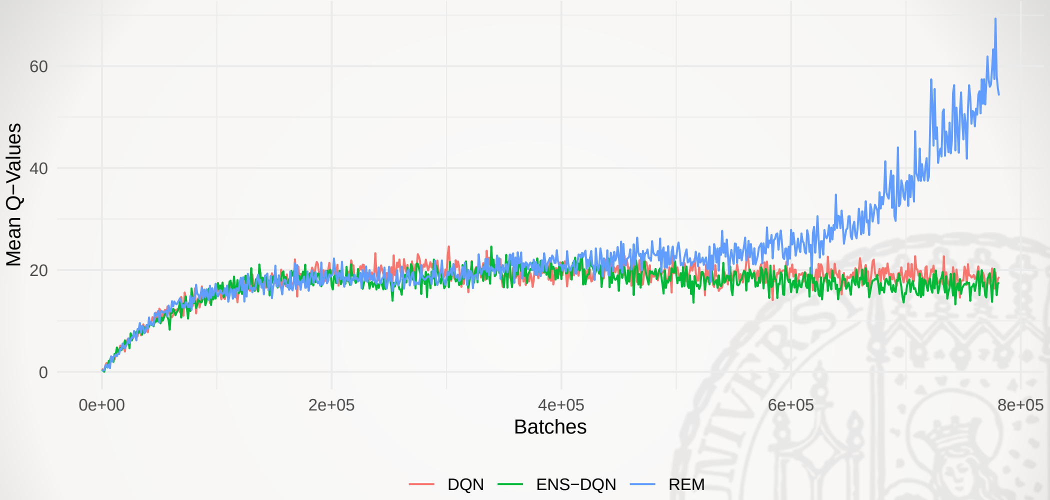
# Mean Reward - 256 Batch - 200 Epochs - DQN Data



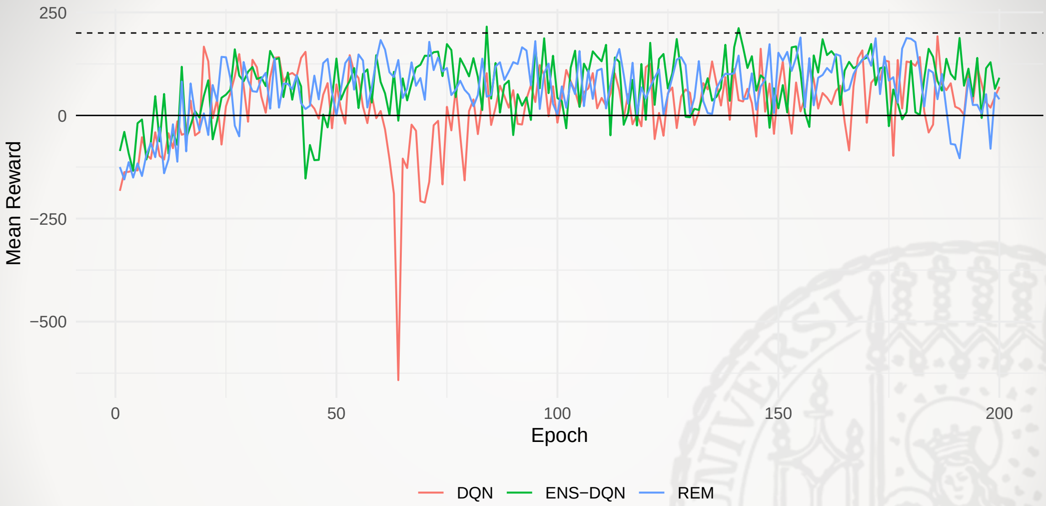
# Mean Reward Smooth - 256 Batch - 200 Epochs - DQN Data



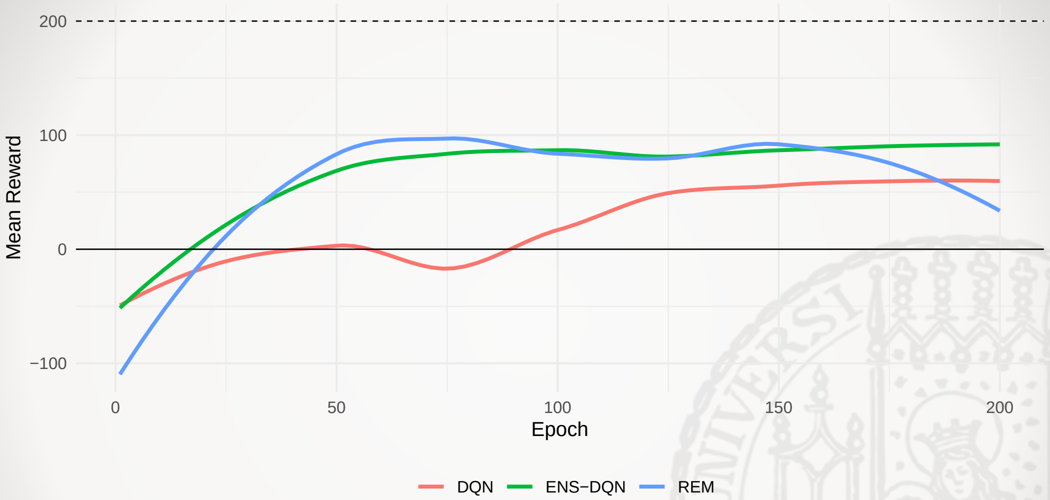
# Q-Values - 256 Batch - 200 Epochs - DQN Data



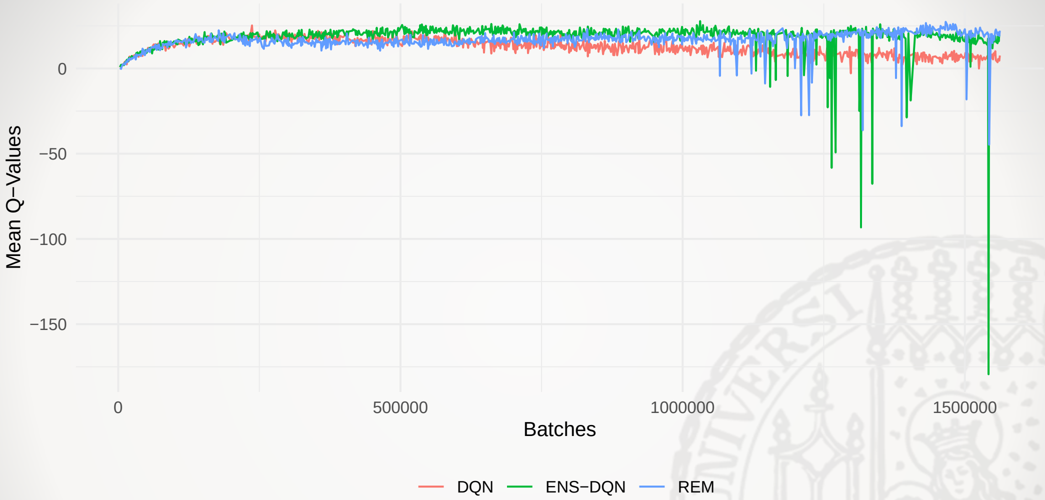
# Mean Reward - 128 Batch - 200 Epochs - DQN Data



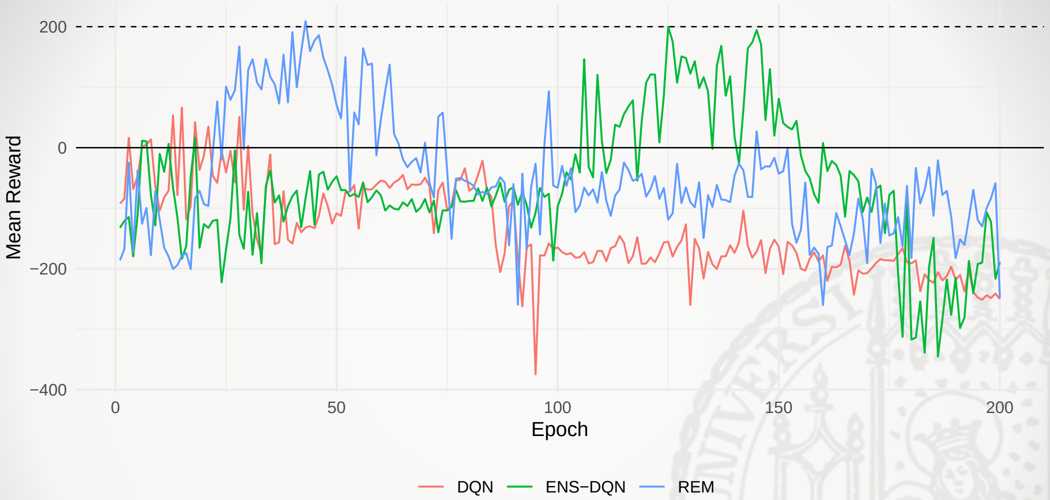
# Mean Reward Smooth - 128 Batch - 200 pochs - DQN Data



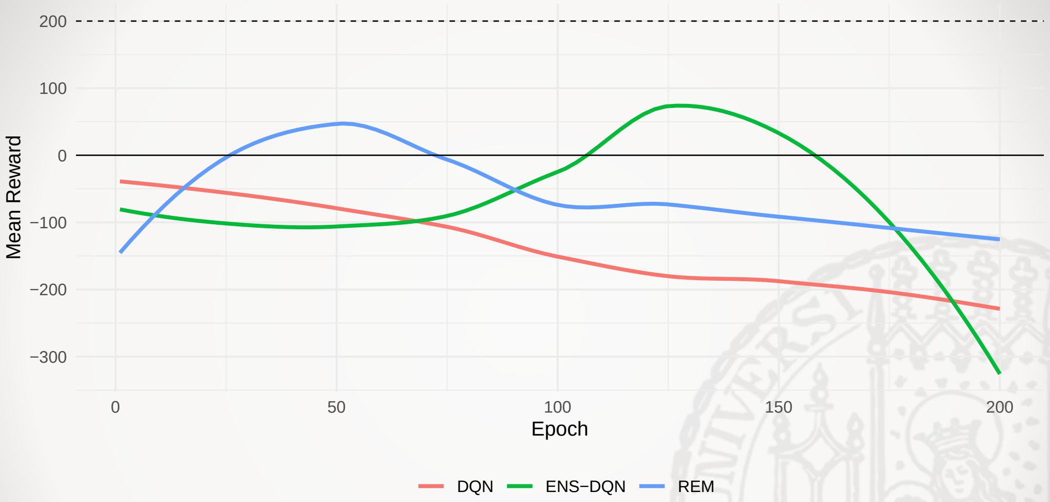
# Q-Values - 128 Batch - 200 Epochs - DQN Data



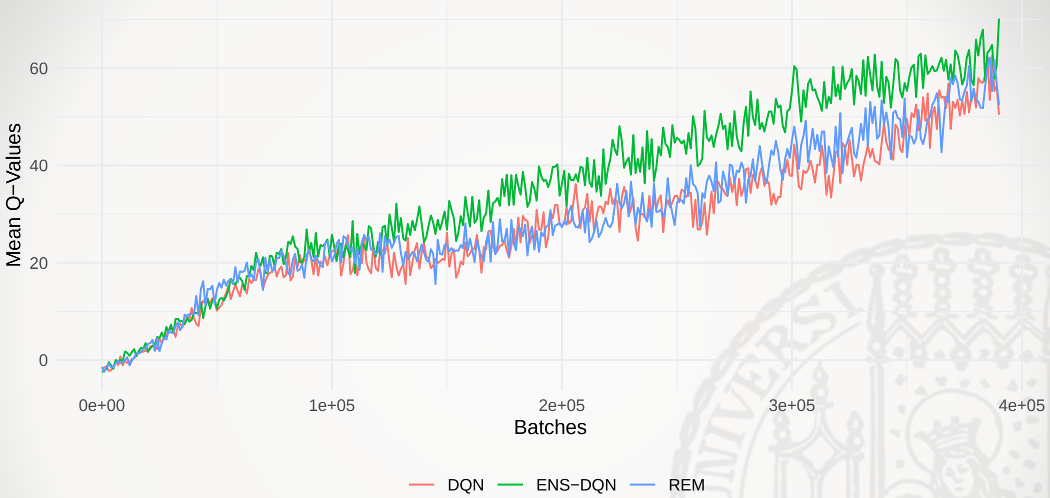
# Mean Reward - 512 Batch - 200 Epochs - Random Data



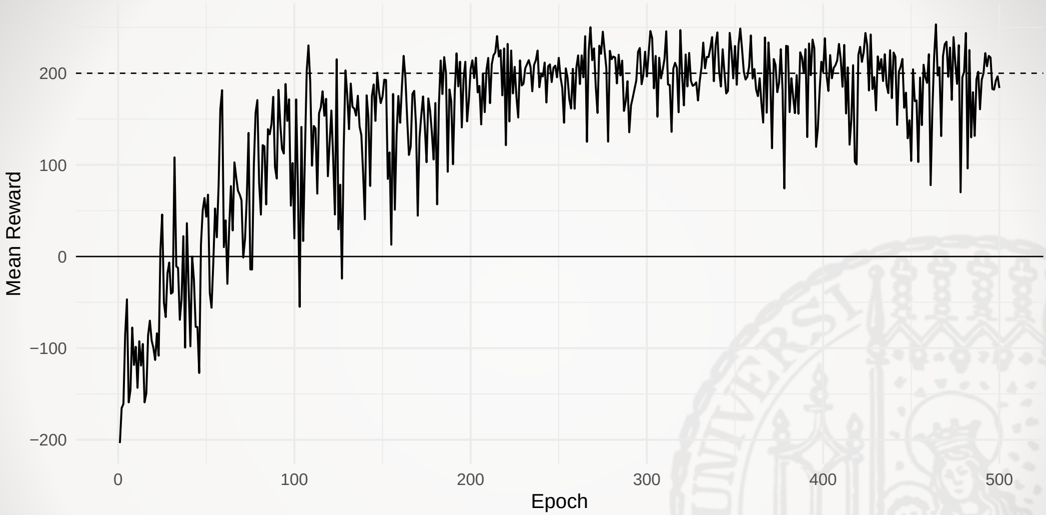
# Mean Reward Smooth - 512 Batch - 200 Epochs - Random Data



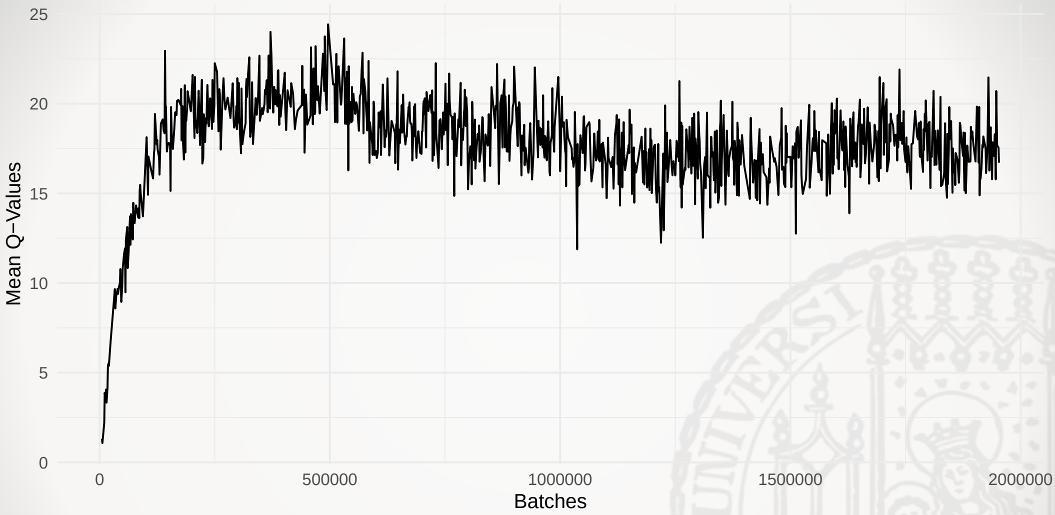
# Q-Values - 512 Batch - 200 Epochs - Random Data



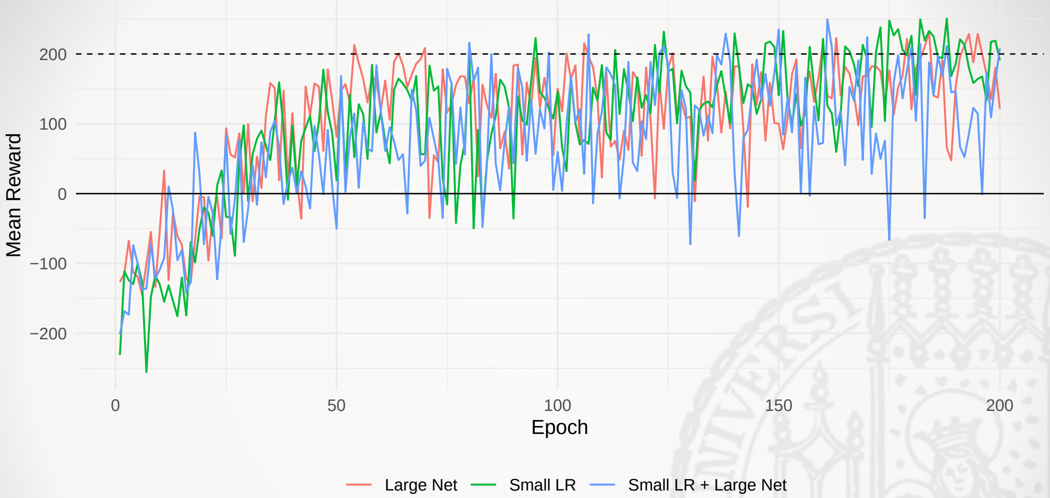
# REM - Mean Reward - 256 Batch - 500 Epochs - DQN Data



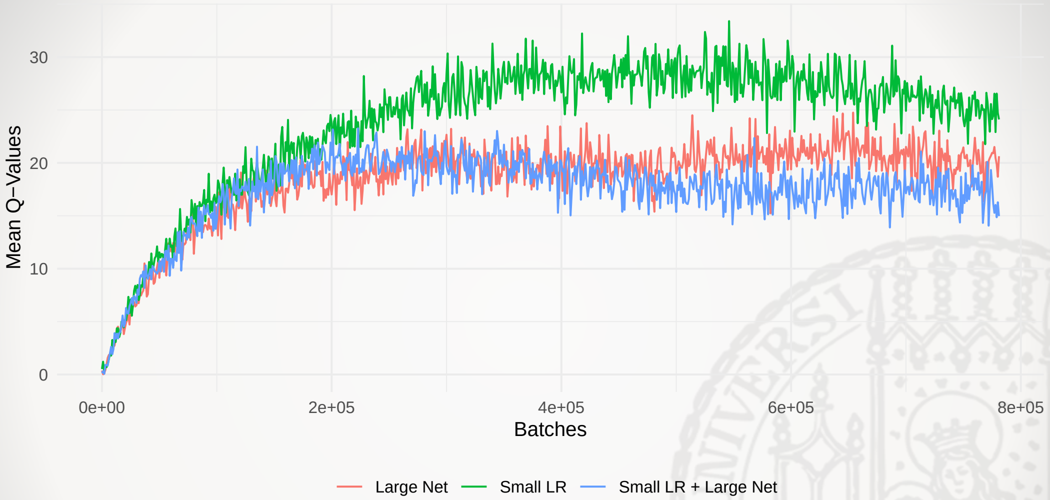
# REM - Q-Values - 256 Batch - 500 Epochs - DQN Data



# REM Variants - Mean Reward - 256 Batch - 200 Epochs - DQN Data



# REM Variants - Q-Values - 256 Batch - 200 Epochs - DQN Data



# Try it on your own: Public Datasets

**DQN Replay Dataset (Agarwal, Schuurmans, and Norouzi 2019)**

- 60 Atari Games
- 5 Datasets per Game with 200 mio. frames

→ 300 Datasets with 500GB each

**D4RL: Datasets for Deep Data-Driven Reinforcement (Fu et al. 2020)**

- Goal: Objective Evaluation of offline RL algorithms
- Datasets with different challenges like multi-objectives or mix of high/low quality data

# Conclusion

# Conclusion

- Extremely active research area
  $\Rightarrow$ Lots of new papers and ideas
- First practical success, e.g. in robotics (KALASHNIKOV et al. 2018)
- No "ultimate" or supreme offline-RL solution
- Different approach styles:
  - Theoretical finesse for understanding and solving occurring issues, e.g. distribution shift, offline Policy Evaluation
  - Deep Learning as high-capacity function approximation to cope with challenges of offline-RL (Large Scale Computing)

# References

# References i


- AGARWAL, R., D. SCHUURMANS, and M. NOROUZI (2019). *An Optimistic Perspective on Offline Reinforcement Learning*. arXiv: 1907.04543 [cs.LG] (cit. on pp. 12, 13, 39, 40, 54, 83).
- ANSCHEL, O., N. BARAM, and N. SHIMKIN (2016). *Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning*. arXiv: 1611.01929 [cs.AI] (cit. on p. 39).
- DABNEY, W. et al. (2017). *Distributional Reinforcement Learning with Quantile Regression*. arXiv: 1710.10044 [cs.AI] (cit. on p. 41).
- FU, J. et al. (2020). *D4RL: Datasets for Deep Data-Driven Reinforcement Learning*. arXiv: 2004.07219 [cs.LG] (cit. on pp. 6–8, 12, 13, 19, 83).

# References ii


Hessel, M. et al. (2017). *Rainbow: Combining Improvements in Deep Reinforcement Learning*. arXiv: 1710.02298 [cs.AI] (cit. on pp. 37, 38).

Kalashnikov, D. et al. (2018). *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. arXiv: 1806.10293 [cs.LG] (cit. on pp. 41, 85).

Kidambi, R. et al. (2020). *MOReL : Model-Based Offline Reinforcement Learning*. arXiv: 2005.05951 [cs.LG] (cit. on p. 41).

Kingma, D. P. and M. Welling (2013). *Auto-Encoding Variational Bayes*. arXiv: 1312.6114 [stat.ML] (cit. on p. 52).

Kumar, A. et al. (2019). *Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction*. arXiv: 1906.00949 [cs.LG] (cit. on p. 41).

# References iii


- Lagoudakis, M. G. and R. Parr (2003). “Least-squares policy iteration”. In: *Journal of machine learning research* 4.Dec, pp. 1107–1149 (cit. on pp. 33, 35).
- Lagoudakis, M. G., R. Parr, and M. L. Littman (2002). “Least-squares methods in reinforcement learning for control”. In: *Hellenic conference on artificial intelligence*. Springer, pp. 249–260 (cit. on p. 34).
- Lange, S., T. Gabel, and M. Riedmiller (2012). “Batch reinforcement learning”. In: *Reinforcement learning*. Springer, pp. 45–73 (cit. on pp. 12, 13).
- Levine, S. et al. (2020). *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems*. arXiv: 2005.01643 [cs.LG] (cit. on pp. 5–8, 11–13, 22).

Mnih, V. et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533 (cit. on pp. 37, 38).

Precup, D. (2000). "Eligibility traces for off-policy policy evaluation". In: *Computer Science Department Faculty Publication Series*, p. 80 (cit. on pp. 26–28).

Riedmiller, M. (2005). "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method". In: *European Conference on Machine Learning*. Springer, pp. 317–328 (cit. on pp. 37, 38).

Rubinstein, R. Y. and D. P. Kroese (2016). *Simulation and the Monte Carlo method*. Vol. 10. John Wiley & Sons (cit. on p. 26).