Lecture 08: Function Approximation with Supervised Learning

André
Bodmer

Table of contents

The machine learning triad

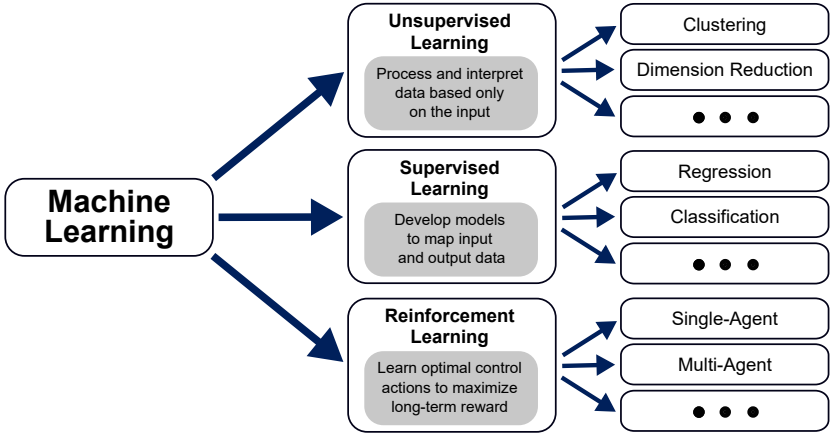


Fig. 1.1: Disciplines of machine learning

Introductory material

Machine learning (ML) and especially the field of supervised learning (SL) is extensively researched and taught.

- Courses at UPB
  - *Statistical and Machine Learning* by the Communications Engineering Dept. (NT)
  - *Machine Learning I & II* by the Data Science 4 Engineering Dept. (DS4Eng)
- Renowned online courses
  - *Coursera ML* by Stanford's Andrew Ng
  - *Practical deep learning for coders* by fast.ai
  - *Intro to ML* by Kaggle Courses
- Books classics
  - *Pattern Recognition and Machine Learning* by C. M. Bishop
  - *The Elements of Statistical Learning* by Hastie et al.
  - *Deep Learning* by I. Goodfellow, Y. Bengio, and A. Courville

Machine learning in industry

Machine learning applications are a fast growing industry itself, and enhance more and more automation in classical industry as well.
Among others, popular industries are:

- Embedded systems,
- Mobility, and
- Digital assistants

Most applications are of the supervised type.
The demand for highly skilled ML engineers is growing correspondingly.

Instances of ML applications

- Recommendation systems
  - Which ads to display on a website?
  - Which items are most likely put into cart next by the user?
- Forecasting
  - Weather, sales, geospatial Uber calls, restaurant/website traffic
  - Material attrition in engineering processes (predictive maintenance)
- Classification/Regression
  - Speech assistants (Alexa/Siri), pedestrial detection (autonomous driving), fault detection in engineering processes
  - large language models (LLM), credit scoring (fintech)
- Generative models

ML competitions with price pool

Fig. 1.2: Kaggle and DrivenData

Open ML competition platforms like kaggle or DrivenData offer a multitude of diverse competitions to participate in at no cost.

- Most competitions come with a decent price pool of 15 tsd. dollars up to 1 mil. dollars hosted by stakeholders from the industry and government.
- These competitions are almost exclusively of the supervised type, but RL challenges are increasing.

Typical supervised learning pipeline

Fig. 1.3: A typical supervised learning pipeline – sometimes more art than science

Supervised learning in reinforcement learning

SL approximates functions, RL approximates policies.

However, there are two situations where SL is auxiliary in RL:

- Function approximation of (action-)state values, if the number of possible states exceeds any reasonable memory capabability, which is often the case.
  - $v_\pi(x) \approx \hat{v}(x, \boldsymbol{w})$ with $\boldsymbol{w}$ being a trainable weight vector.
- Imitation learning. A simple-to-implement, deterministic baseline policy is often available, but an RL agent might fail to achieve that performance when learning from scratch. With SL, this baseline policy can be approximated to be the initial behavior of the agent.
  - Expert moves in board games.
  - Basic linear controllers in engineering applications with feedback-loops.

Supervised learning problem statement

Supervised learning

Given a labeled data set $\langle \boldsymbol{x}_k, \boldsymbol{y}_k \rangle \in \boldsymbol{\mathcal{D}}$ with $k \in [0, K-1]$ and K being the data set size, approximate the mapping function $f^* : \boldsymbol{x}_k \mapsto \boldsymbol{y}_k$ with a parameterizable ML model $f_{\boldsymbol{w}} : \boldsymbol{x}_k \mapsto \hat{\boldsymbol{y}}_k \approx \boldsymbol{y}_k \quad \forall k$.

- Goodness of fit can be measured by a manifold of metrics (e.g., mean squared error, classification accuracy, etc.).
- Reducing the look-up-table-like mapping f^* to a parameterized function $f_{\boldsymbol{w}}$ degrades any metric on the data set but enables interpolation to unseen data.
- The dimension ξ of model parameters $\boldsymbol{w} \in \mathbb{R}^{\xi}$ is adjustable in many model families, which trades off bias with variance (among other factors, leading to so-called under- and overfitting).
- On top of $\boldsymbol{w}$, an ML model might also have hyperparameters that can be optimized (e.g., number of layers in a neural network).

Bias and variance

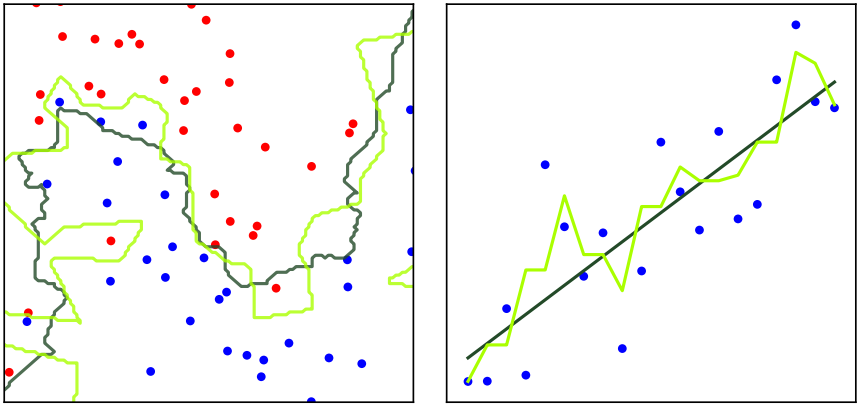


Fig. 1.4: Left: Decision boundaries in binary classification, k-nearest neighbors with one (bright) and nine (dark) neighbors. Right: Regression example, least squares (dark) and 2-nearest neighbors (bright).

Generalization error

Supervised learning performance

SL performance is measured by a model's generalization error, i.e., goodness of fit on unseen data.

A data set is often finite as opposed to RL environments generating arbitrarily many observations.

- How to generate unseen data?
  - Hold out portions of the data set for cross-validation.

k-fold cross-validation

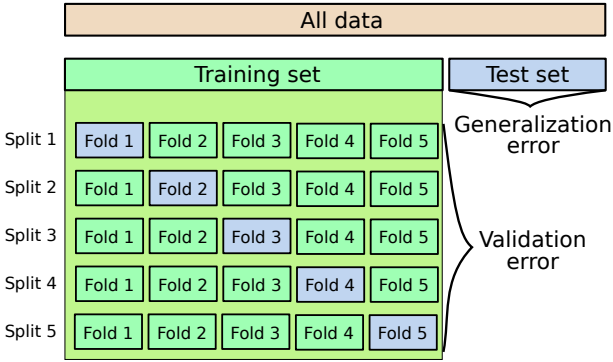


Fig. 1.5: k-fold CV with five folds

- Cross-validation (CV) can be conducted with k-fold CV.
- Training is repeated k times with k different splits of the training set.
- Each observation serves as unseen instance at least once.
- The validation error is an indicator for tuning hyperparameters.

Means to improve an SL model

SL performance can be improved by:

- Collecting more data, i.e., increasing K (more data is always better).
- Choosing a more appropriate model.
- Optimizing hyperparameters of the model.
- Averaging over several different models (ensembling).
- Most effectively: Revealing the most predictive patterns in the data to the model (feature engineering).

Table of contents

Feature engineering

Additional features might be:

- Coming from the real world via additional sensors or additional tracking mechanisms (think of a user's click behavior on a website)
- Hand-designed (*engineered*) by experts in the corresponding domain from the original feature set
- Automatically built according to properties of each feature in the original set (Auto-ML)

Caution

Adding more features is not equivalent to having more data (which is always better). Having a fixed data set size, adding arbitrarily many features, regardless of their origin, increases chances to align statistical fluctuations with the target $\boldsymbol{y}_k$ - overfitting is the result.

Feature engineering example (classification)

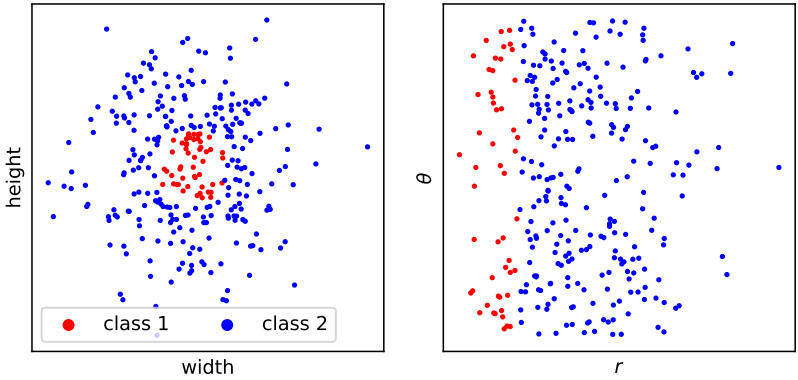


Fig. 1.6: Features $r = \sqrt{\text{width}^2 + \text{height}^2}$ and $\theta = \arctan\left(\frac{\text{height}}{\text{width}}\right)$ reveal linearly separable class distribution

Feature engineering example (regression)

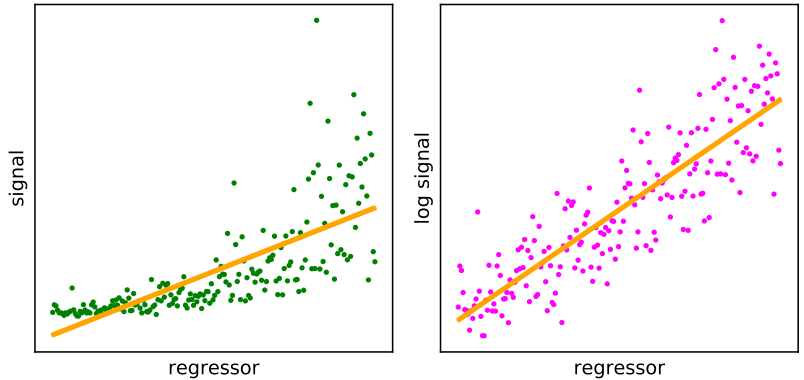


Fig. 1.7: Log-transform of the target signal exhibits linear relationship to the regressor

Normalization

Most models require data to be normalized before training (apart from tree-based models).
Typical normalizaton schemes:

- Standard scaling: $\tilde{\boldsymbol{x}} = (\boldsymbol{x} - \text{Avg}(\boldsymbol{x}))/\text{Std}(\boldsymbol{x})$
- Min-Max scaling: $\tilde{\boldsymbol{x}} = (\boldsymbol{x} - \min(\boldsymbol{x}))/(\max(\boldsymbol{x}) - \min(\boldsymbol{x}))$
- Plain scaling: $\tilde{\boldsymbol{x}} = \boldsymbol{x}/\max(|\boldsymbol{x}|)$

In an unnormalized data set, features with high variance will eclipse patterns in other features.

Data types

Several different data types can be utilized for ML:

- Binary: 1 or 0 (True or False).
- Integer: $\mathbb{N}$ (e.g., number of rooms in a building).
- Real-valued: $\mathbb{R}$ (e.g., temperature).
- Categorical: like {blue, green, red}
- Ordinal: Categoricals that can be ordered, e.g., educational experience (From elementary school to Ph.D.)

Data type specific normalization

How to normalize categorical data?

- *One-hot* encoding
  - Replace a categorical of n values with n binary features.
  - Feature space gets sparse and might get too big for memory.
- Mean target encoding
  - Replace each value of a categorical with the average (regression) or mode (classification) of the dependent variable being observed with the corresponding value.
  - This might lead to information *leaking* from the dependent variables into the independent variables, and might exhibit high performance that cannot be reproduced on unseen data.
- Entity embeddings
  - Let a neural network find a cardinality-constrained set of real-valued features for each categorical.
  - Works well in practice but is more intricate than alternatives.

Typical feature engineering schemes

Feature design is often of the following form (tricks of the trade):
Given K feature vectors $\boldsymbol{x}_k \in \mathbb{R}^P$ with, e.g., $P = 3$ (two real-valued regressors and a categorical independent variable $\boldsymbol{x}_k = (x_{k,r_1}, x_{k,r_2}, x_{k,c})$):

- $\tilde{x}_k = x_{k,r_1} + x_{k,r_2}$ (or any other combination, e.g., product, division, subtraction, also cf. Fig. 1.6),
- $\tilde{x}_k = x_{k,r} - \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_{i,r} \quad \forall r = \{r_1, r_2\}$ with $\mathcal{B} = \{i : x_{i,c} = x_{k,c}\}$,
- Clip/drop/aggregate outliers away,
- Coordinate transformations for spatial features (e.g., rotation),
- In time domain:
  - $\tilde{\boldsymbol{x}}_k = (x_{k,r_1}, x_{k-1,r_1}, x_{k-2,r_1}, x_{k,r_2}, x_{k,c})$ (lag features),
  - $\tilde{x}_k = (1 - \alpha)\tilde{x}_{k-1} + \alpha x_{k,r}$ (moving averages).
- In frequency domain:
  - Amplitude and index of frequencies from a fast fourier transform (FFT)

Table of contents

Model landscape

When trying to find an appropriate mapping between input and output data, one can choose from a variety of models:

- Linear/logistic regression (with regularization)
  - The simplest data-fitting algorithm
- Support vector machines (SVM)
  - Most popular algorithm before 2012
- (Deep) neural networks (DNN)
  - Also coined as *deep learning*, soared in popularity since 2012
  - Most prevalent in the domains of natural language processing (NLP) and image processing
- Gradient Boosting Machines (GBM)
  - Chaining of *weak* models (most of the time decision trees)
  - The best performing stand-alone model in tabular ML competitions

Model choice

Fig. 1.8: Choose models appropriate for the problem! (Source: Adapted from reddit)

Linear regression (1)

Linear models assume a linear relationship between $\boldsymbol{x}_k = (1, x_{k,1}, x_{k,2}, \ldots, x_{k,P})$ and y_k via trainable coefficients $\boldsymbol{w} \in \mathbb{R}^{P+1}$:

$$f(\boldsymbol{x}_k) = \hat{y}_k = w_0 + \sum_{p=1}^{P} x_{k,p} w_p, \tag{1.1}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{\Xi} \boldsymbol{w}, \tag{1.2}$$

where $\boldsymbol{\Xi} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_K)$. Among other methods, $\boldsymbol{w}$ can be estimated from K samples by minimizing the residual sum of squares (RSS), which is coined the least squares method:

$$\mathrm{RSS}(\boldsymbol{w}) = \sum_{k=1}^{K} (y_k - f(\boldsymbol{x}_k))^2 = (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w})^{\mathsf{T}} (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w}). \tag{1.3}$$

Linear regression (2)

Deriving (1.3) with respect to $\boldsymbol{w}$ and setting it to zero while assuming $\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi}$ is positive-definite, yields an analytically closed solution form:

$$\hat{\boldsymbol{y}} = \boldsymbol{\Xi}\hat{\boldsymbol{w}} = \boldsymbol{\Xi}(\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi})^{-1}\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{y}. \tag{1.4}$$

Multicollinearity

If two regressors exhibit strong linear correlation, their coefficients can grow indeterministically. This corresponds to high variance in $\hat{\boldsymbol{w}}$. Regularization of $\hat{\boldsymbol{w}}$ alleviates this effect - it induces bias for less variance. Most prevalent linear regularized techniques are LASSO and Ridge:

$$\text{RSS}_{\text{LASSO}}(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w})^\mathsf{T}(\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w}) + \lambda||\boldsymbol{w}||_1, \tag{1.5}$$

$$\text{RSS}_{\text{Ridge}}(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w})^\mathsf{T}(\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w}) + \lambda||\boldsymbol{w}||_2, \tag{1.6}$$

where λ controls the growth penalty.

Artificial neural networks

Artificial neural networks (ANNs) describe nonlinear approximators $\hat{\boldsymbol{y}} = f(\boldsymbol{\Xi}; \boldsymbol{w})$ that are end-to-end differentiable.

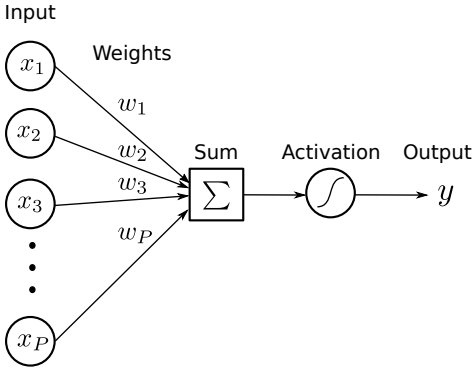


Fig. 1.9: A typical neuron as the key building block of ANNs.

- An ANN consists of nodes or neurons in one or more layers.
- Each node transforms the weighted sum of all previous nodes through an activation function.
- The weighted connections are called edges, which represent the ANN's parameters.

Multi-layer perceptron

A vanilla ANN is the so-called feed-forward ANN or multi-layer perceptron.

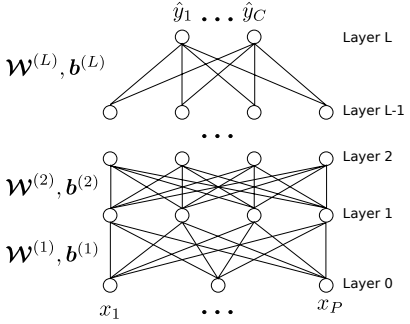


Fig. 1.10: Multi-layer perceptron.

- Only forward-flowing edges.
- The depth L and width $H^{(l)}$ are hyperparameters.

With $\varphi^{(l)}$ and $\boldsymbol{\mathcal{Z}}^{(l)}$ denoting the activation function and activation of layer l respectively, we get for the output matrix $\boldsymbol{\mathcal{H}}^{(l)}$

$$\boldsymbol{\mathcal{H}}^{(l)} = \varphi^{(l)}\big(\underbrace{\boldsymbol{\mathcal{H}}^{(l-1)}\boldsymbol{\mathcal{W}}^{(l)} + \boldsymbol{b}^{(l)}}_{\boldsymbol{\mathcal{Z}}^{(l)}}\big).$$

Weight matrix $\boldsymbol{\mathcal{W}}^{(l)} \in \mathbb{R}^{H^{(l-1)} \times H^{(l)}}$ and (broadcasted) bias matrix $\boldsymbol{b}^{(l)} \in \mathbb{R}^{K \times H^{(l)}}$ are iteratively optimized and denote the full set of parameters $\boldsymbol{w}$.

Activation functions

Within hidden layers most prevalent activation functions $\varphi(\cdot)$ are

- $h = \tanh(z)$
- $h = \frac{1}{1+e^{-z}}$ (sigmoid)
- $h = \max(0, z)$ (rectified linear unit (ReLU))

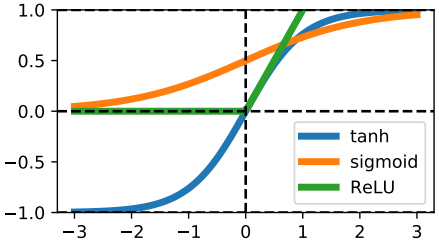


Fig. 1.11: Common activation functions

Whereas $\varphi^{(L)}(\cdot)$ is task-dependent:

- Regression: $\hat{y} = h^{(L)} = z^{(L)}$
- Binary classification: sigmoid
- Multi-class classification:

$$h_c^{(L)} = \frac{e^{z_c}}{\sum_{i=1}^{C} e^{z_i}} \quad \text{(softmax)}$$

Training neural networks (1)

ANN parameters are usually iteratively optimized via a variant of gradient descent, e.g., stochastic gradient descent (SGD).

$$\boldsymbol{\mathcal{W}}^{(l)} \leftarrow \boldsymbol{\mathcal{W}}^{(l)} - \alpha \nabla_{\boldsymbol{\mathcal{W}}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}), \tag{1.7}$$

$$\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \alpha \nabla_{\boldsymbol{b}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}), \tag{1.8}$$

with α being the step size and $\mathcal{L}(\cdot)$ denoting the loss between the ground truth vector and the estimation vector.
Typical loss functions:

- Regression: (root) mean squared error (RMSE), mean absolute error
- Classification: Cross-entropy (CE)

Several iterations over the data set $\mathcal{D}$ are called epochs.

Training neural networks (2)

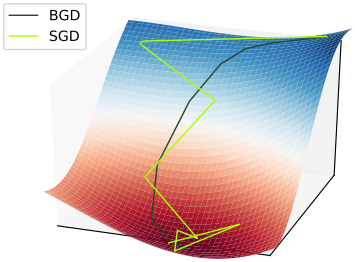


Fig. 1.12: BGD vs. SGD

Gradient descent alternatives:

- Batch gradient descent (BGD): Average gradients over all samples, then update weights.
- Stochastic gradient descent (SGD): Update weights after each sample.

SGD is more computationally efficient, but steps are more random.
Nowadays, mini-batch gradient descent (mix of SGD and BGD) and further improvements are used, e.g., momentum and second derivatives, to ensure faster convergence to better optima.

Training neural networks (3)

How to retrieve the gradients:
Recall chain rule for vector derivatives, e.g., with $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ where $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$:

$$\nabla_{\boldsymbol{x}} z = \frac{\partial z}{\partial \boldsymbol{x}} = \underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\mathsf{T}}}_{\text{Jacobian of g}} \cdot \underbrace{\frac{\partial z}{\partial \boldsymbol{y}}}_{\text{gradient}} = \sum_j \frac{\partial y_j}{\partial \boldsymbol{x}} \cdot \frac{\partial z}{\partial y_j}. \tag{1.9}$$

This can be used equivalently for matrices/tensors of any shape $\nabla_{\boldsymbol{\Xi}} y = \frac{\partial y}{\partial \boldsymbol{\Xi}}$ when we assume to enumerate each element of the tensor consecutively and loop through them.

Error Backpropagation

After a forward step through the network, make a backward step in which the gradient $\boldsymbol{\gamma}$ of the loss $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is computed w.r.t the ANN's parameters from the output layer back to the input layer.

Training neural networks (4)

init: $\boldsymbol{\mathcal{H}}^{(0)} \leftarrow \boldsymbol{\Xi}$
// forward propagation
for $l = 1, \ldots, L$ *layers* **do**
    $\boldsymbol{\mathcal{Z}}^{(l)} \leftarrow \boldsymbol{\mathcal{H}}^{(l-1)} \boldsymbol{\mathcal{W}}^{(l)} + \boldsymbol{b}^{(l)}$
    $\boldsymbol{\mathcal{H}}^{(l)} \leftarrow \varphi^{(l)}(\boldsymbol{\mathcal{Z}}^{(l)})$
// backward propagation
$\boldsymbol{\gamma} \leftarrow \nabla_{\boldsymbol{h}^{(L)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ // note that $\boldsymbol{h}^{(L)} = \hat{\boldsymbol{y}}$
for $l = L, \ldots, 1$ *layers* **do**
    $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} \odot \partial(\varphi^{(l)})(\boldsymbol{\mathcal{Z}}^{(l)}) = \nabla_{\boldsymbol{\mathcal{Z}}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ // $\odot$: elementwise mult.
    Append $\boldsymbol{\gamma} = \nabla_{\boldsymbol{b}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to list of bias gradients
    Append $(\boldsymbol{\mathcal{H}}^{(l-1)})^{\mathsf{T}} \cdot \boldsymbol{\gamma} = \nabla_{\boldsymbol{\mathcal{W}}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to list of weight gradients
    $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} \cdot (\boldsymbol{\mathcal{W}}^{(l)})^{\mathsf{T}} = \nabla_{\boldsymbol{\mathcal{H}}^{(l-1)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$

Algo. 1.1: Error backpropagation

Error backpropagation example (1)

Assume $\boldsymbol{x}_0 = [2, 5, 7], y_0 = 2.5$, and a two-layered ANN with the MSE cost, and sigmoid activation functions $\sigma(z) = \frac{1}{1+e^{-z}}$. The hidden layer contains two neurons with output $\boldsymbol{h}^{(1)} \in \mathbb{R}^2$, while the weight vectors are initialized with $\boldsymbol{\mathcal{W}}^{(1)} = \begin{bmatrix} 0.1 & -0.3 & 0.2 \\ 0.0 & 0.4 & -0.9 \end{bmatrix}^\mathsf{T}, \boldsymbol{b}^{(1)} = [0.05, -0.03]$, and $\boldsymbol{\mathcal{W}}^{(2)} = [0.2, -0.8]^\mathsf{T}, \boldsymbol{b}^{(2)} = [0.1]$. Applying SGD, we start with forward propagation:

$$\begin{aligned} \boldsymbol{h}^{(1)} &= \varphi^{(1)}(\boldsymbol{x}_0 \boldsymbol{\mathcal{W}}^{(1)} + \boldsymbol{b}^{(1)}) \\ &= \sigma([0.1, -4.3] + [0.05, -0.03]) = [0.53, 0.01] \\ \hat{y}_0 &= \boldsymbol{h}^{(1)} \boldsymbol{\mathcal{W}}^{(2)} + \boldsymbol{b}^{(2)} = 0.198 \end{aligned}$$

Error backpropagation example (2)

Backpropagation (with $\sigma'(z) = \partial_z \sigma(z) = \sigma(z)(1 - \sigma(z))$):

$$
\begin{aligned}
\boldsymbol{\gamma}^{(2)} = \nabla_{\hat{y}_0} \mathcal{L}(y_0, \hat{y_0}) &= \nabla_{\hat{\boldsymbol{y}}} (y_0 - \hat{y}_0)^2 = -2(y_0 - \hat{y}_0) = -4.604 \\
\nabla_{\boldsymbol{b}^{(2)}} \mathcal{L}(y_0, \hat{y_0}) &= \boldsymbol{\gamma}^{(2)} \odot \partial(\varphi^{(2)})(\boldsymbol{z}^{(2)}) = \boldsymbol{\gamma}^{(2)} \\
\nabla_{\boldsymbol{\mathcal{W}}^{(2)}} \mathcal{L}(y_0, \hat{y_0}) &= (\boldsymbol{h}^{(1)})^\mathsf{T} \cdot \boldsymbol{\gamma}^{(2)} = [-2.44, -0.046]^\mathsf{T} \\
\boldsymbol{\gamma}^{(1)} = \nabla_{\boldsymbol{h}^{(1)}} \mathcal{L}(y_0, \hat{y_0}) &= \boldsymbol{\gamma}^{(2)} \cdot (\boldsymbol{\mathcal{W}}^{(2)})^\mathsf{T} = [-0.921, 3.683] \\
\nabla_{\boldsymbol{b}^{(1)}} \mathcal{L}(y_0, \hat{y_0}) &= \boldsymbol{\gamma}^{(1)} \odot \partial(\varphi^{(1)})(\boldsymbol{z}^{(1)}) = \boldsymbol{\gamma}^{(1)} \odot \sigma'(\boldsymbol{x}_0 \boldsymbol{\mathcal{W}}^{(1)} + \boldsymbol{b}^{(1)}) \\
&= \boldsymbol{\gamma}^{(1)} \odot (\boldsymbol{h}^{(1)}(1 - \boldsymbol{h}^{(1)})) = [-0.229, 0.036] \\
\nabla_{\boldsymbol{\mathcal{W}}^{(1)}} \mathcal{L}(y_0, \hat{y_0}) &= \boldsymbol{x}_0^\mathsf{T} \cdot \boldsymbol{\gamma}^{(1)} = \begin{bmatrix} -1.84 & -4.605 & -6.447 \\ 7.366 & 18.415 & 25.781 \end{bmatrix}^\mathsf{T}
\end{aligned}
$$

Now update weights and biases according to (1.7) and (1.8).

Weight initialization

Early in deep learning research, it was found that random uniform or random normal weight initialization leads to poor training.
According to Glorot and Bengio[1], use the following layer-specific initialization schemes (with H_{in} and H_{out} denoting amount of hidden units of previous and current layer, respectively):

- uniform: $\boldsymbol{w} \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{H_{\text{in}}+H_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{H_{\text{in}}+H_{\text{out}}}}\right)$
- normal: $\boldsymbol{w} \sim \mathcal{N}\left(0, \frac{\sqrt{2}}{\sqrt{H_{\text{in}}+H_{\text{out}}}}\right)$

Please note that generally due to the random weight initialization the result of repeated error backpropagation training is always different regardless of having the same hyperparameters and the same data.
This equals to local optimization in highly non-linear parameter spaces at random starting points.

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", *Proceedings of Machine Learning Research*, 2010

Regularizing neural networks

In order to mitigate overfitting, ANNs must be regularized by

- weight decay, i.e., adding an ℓ_2 penalty term to the weights, see (1.6),
- layer normalization during training,
  - i.e all layers' activations are normalized by standard scaling separately,
- dropout, i.e., randomly disable nodes' contribution.
  - This helps especially in deep networks,
  - and effectively builds an ensemble of ANNs with shared edges.

Advanced topologies

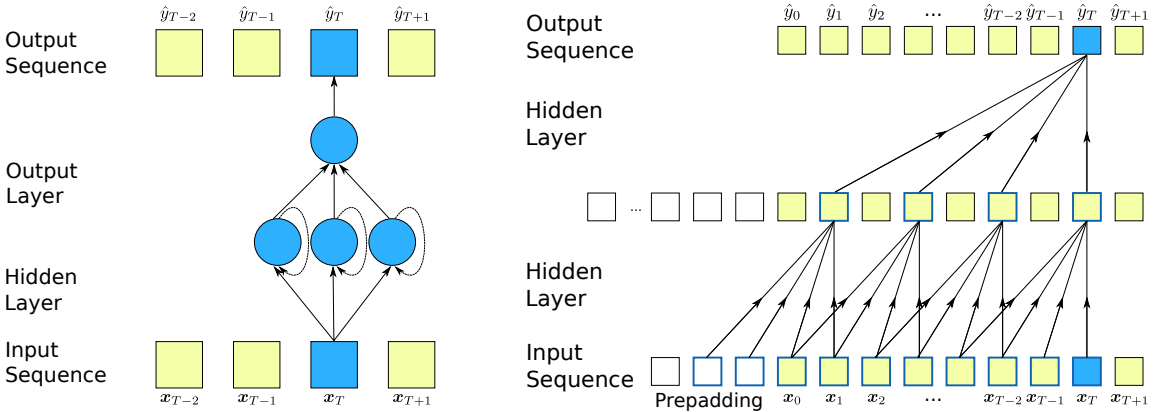


Fig. 1.13: Recurrent (left) and 1-D convolutional (right) ANNs are more appropriate in time domains, e.g., where the given data set has a dynamic system background

Hyperparameter optimization (1)

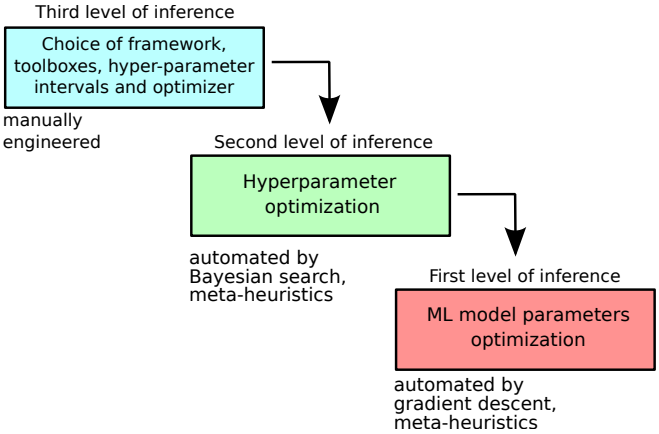


Fig. 1.14: The three levels of optimization

Hyperparameter optimization (2)

- Hyperparameter optimization is, again, a non-linear optimization problem.
- Evaluation of any point in this space can be very costly, though.
- Information gathered during a search must be fully utilized.
- Toolboxes (incomprehensive)
  - Optuna
  - Scikit-optimize
  - Pyswarm

SL toolboxes

- Deep learning
  - Tensorflow 2 (Keras)
  - PyTorch
  - Chainer
  - CNTK
- Gradient boosting machines
  - XGBoost
  - LightGBM
  - CatBoost
- Linear, tree-based, memory-based models, SVMs, among others
  - Scikit-learn

Summary: what you've learned today

- Industry has high demand for ML applications.
- Higher bias trades off variance for a better overall score.
- How to cross-validate and improve SL models.
- How features are engineered and normalized.
- Fundamentals of linear regression and neural networks.

Lecture 09: On-Policy Prediction with Function Approximation

André
Bodmer

Preface

Until further notice we assume that:

- The state space is consisting of at least one continuous quantity or an unfeasible large amount of discrete states (quasi-continuous).
  - The state is considered a vector: $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \cdots \end{bmatrix}^\mathsf{T}$.
- The action space remains discrete and feasible small.
  - The action can be represented as a scalar: $\boldsymbol{u} = u$ (cf. 2nd lecture).
- The applied approximation functions $J(\boldsymbol{w})$ are differentiable with the parameter vector $\boldsymbol{w}$.
  - Therefore, the gradient $\nabla J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{w})}{\partial w_1} & \frac{\partial J(\boldsymbol{w})}{\partial w_2} & \cdots \end{bmatrix}^\mathsf{T}$ exists.

Focus of this and the next lecture:

- Transferring previous RL methods from discrete to continuous state-space problems in the on-policy case.
- Applying off-policy approaches with function approximation is not straightforward and will be largely skipped.
  - For further insights we refer to chapter 11 in *R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018*.

Table of contents

Non-stationarity

- Standard assumption of supervised ML: static and i.i.d. data processes
- Deviating impacts in the RL framework:
  - Changing environments (e.g., by tear and wear)
  - Dynamic learning in control tasks, i.e., changing policy (next lecture)

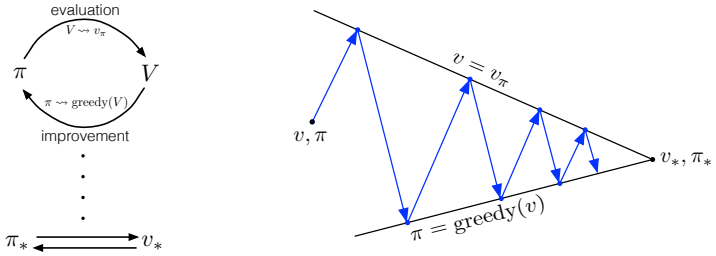


Fig. 2.1: GPI changes the underlying stochastic processes generating data inputs to be learned by function approximators (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Prediction framework with function approximation (1)

- Estimate true value function $v_\pi(\boldsymbol{x})$ using a parametrizable approximate value function

$$\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w}) \approx v_\pi(\boldsymbol{x}). \tag{2.1}$$

- The state $\boldsymbol{x}$ might be enhanced by feature engineering (i.e., additional signal inputs are derived in the feature vector $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \in \mathbb{R}^\kappa$).
- Above, $\boldsymbol{w} \in \mathbb{R}^\zeta$ is the parameter vector.
- Typically, $\zeta << |\mathcal{X}|$ applies (otherwise approximation is pointless).

Generalization

Due to the usage of function approximation one incremental learning step changes at least one element $w_i \in \boldsymbol{w}$ which

- affects the estimated value of many states compared to
- the tabular case where one update step affects only one state.

Prediction framework with function approximation (2)

- In the tabular case a specific prediction objective was not needed:
  - The learned value function could exactly match the true value.
  - The value estimate at each state was decoupled from other states.
- Due to generalization impact we need to define an accuracy metric on the entire state space (the RL prediction goal):

Definition 2.1: Mean Squared Value Error

The RL prediction objective is defined as the mean squared value error

$$\overline{\text{VE}}(\boldsymbol{w}) = \int_{\mathcal{X}} \mu(\boldsymbol{x}) \left[v_\pi(\boldsymbol{x}) - \hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})\right]^2 \tag{2.2}$$

with $\mu(\boldsymbol{x}) \in \{\mathbb{R} | \mu(\boldsymbol{x}) \geq 0\}$ being a state distribution weight with $\int_{\mathcal{X}} \mu = 1$.

- Practical note: As the true value $v_\pi(\boldsymbol{x})$ is most likely unknown in most tasks, (2.2) cannot be computed exactly but only estimated.

Simplification for on-policy prediction

- For prediction we focus entirely on the on-policy case.
- Hence, $\mu(\boldsymbol{x})$ is the on-policy distribution under π.
- For practical usage we can therefore approximate the weighted integration over the entire state space $\mathcal{X}$ in (2.2) by the sampled MSE of the visited state trajectory:

$$\overline{\mathrm{VE}}(\boldsymbol{w}) \approx J(\boldsymbol{w}) = \sum_k \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})\right]^2 . \tag{2.3}$$

- If we would perform off-policy prediction we have to transform the sampled value (estimates) from the behavior to the target policy.
- Likewise when doing this for tabular methods, this increases the prediction variance.
- In combination with generalization errors due to function approximation, the overall risk of diverging is significantly higher compared to the on-policy case.

Prediction challenges with function approximation

Summarizing the two previous slides:

- The goal is to find

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w}). \tag{2.4}$$

First challenge:

- Function approximator $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ requires certain form to fit $v_\pi(\boldsymbol{x})$.

Second challenge:

- If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is linear: convex optimization problem.
  - The nice case: the local optimum equals the global optimum and is uniquely discoverable. But requires linear feature dependence.
- If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is non-linear: non-linear optimization problem.
  - The ugly case: possible multitude of local optima with no guarantee to locate the global one.
  - Depending on optimization strategy the RL algorithm may diverge.

Table of contents

Updating the parameter vector to find (local) optimum

Transferring the idea of incremental learning steps from the tabular case

$$\hat{v}(x) \leftarrow \hat{v}(x) + \alpha \left[v_\pi(x) - \hat{v}(x) \right] \tag{2.5}$$

to function approximation using a gradient descent update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}). \tag{2.6}$$

- The search direction is the prediction objective gradient $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$.
- The learning rate α determines the step size of one update.

How to retrieve the gradient?

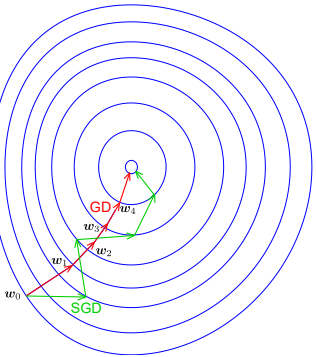


Fig. 2.2: Exemplary optimization paths for (stochastic) gradient descent (derivative work of www.wikipedia.org, CC0 1.0)

- Full calculus of $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$:
  - Batch evaluation on sampled sequence $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ might be computationally costly.
  - In RL control: since π changes over time, past data in batch is not fully representative.
- SGD: sample gradient at a given state $\boldsymbol{x}_k$ and parameter vector $\boldsymbol{w}_k$:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -\left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k).$$

- Regular gradient descent leads to same result as SGD in expectation (averaging of samples).

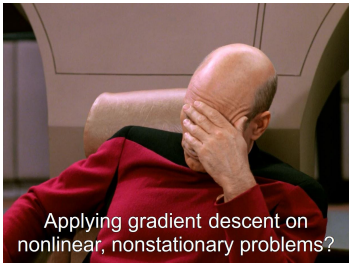
Asking an expert on convergence properties

The optimization task (2.4) could be

- non-linear,
- multidimensional and
- non-stationary.

Applying gradient descent to such a problem requires:

- Enormous luck to initialize $\boldsymbol{w}_0$ close to the global optimum.
- Cautious tuning of α to prevent diverging or chattering of $\boldsymbol{w}_k$.



SGD-based learning step

Despite the possible problems we apply SGD-based learning due to its striking simplicity (and wide distribution in the literature):

Gradient-based parameter update

To optimize (2.4) by an appropriate function approximator $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ the incremental learning update per step is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k). \tag{2.7}$$

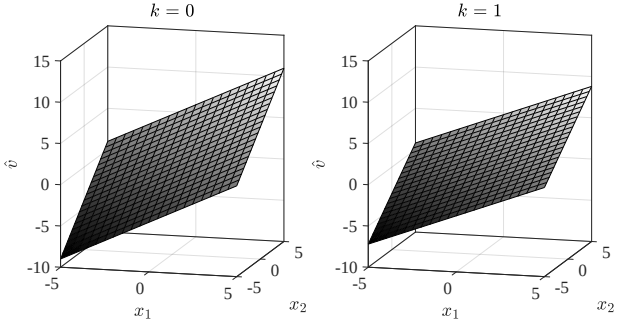
Nevertheless, the true update target $v_\pi(\boldsymbol{x}_k)$ is often unknown due to

- noise or
- the learning process itself (e.g., bootstrapping estimates).

Generalization example for parameter update

- Function approximation $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w}) = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix}^\mathsf{T}$
- Initial parameter: $\boldsymbol{w}_0^\mathsf{T} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$, $v_\pi(\boldsymbol{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^\mathsf{T}) = 1$, $\alpha = 0.1$
- New parameter set:

$$\begin{aligned} \boldsymbol{w}_1^\mathsf{T} &= \boldsymbol{w}_0^\mathsf{T} + \alpha \left[v_\pi(\boldsymbol{x}_0) - \hat{v}(\tilde{\boldsymbol{x}}_0, \boldsymbol{w}_0) \right] \left(\nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_0, \boldsymbol{w}_0) \right)^\mathsf{T} \\ &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} + 0.1 \, (1 - 3) \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.8 & 0.8 & 0.8 \end{bmatrix} \end{aligned}$$



Algorithmic implementation: gradient Monte Carlo

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $v_\pi(\boldsymbol{x}_k) \approx g_k$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
input: a differentiable function $\hat{v} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate an episode following π: $x_0, u_0, r_1, \ldots, x_T$;
  calculate every-visit return g_k;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$;

Algo. 2.1: Every-visit gradient MC (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_\pi$)

Semi-gradient methods

- If bootstrapping is applied, the true target $v_\pi(\boldsymbol{x}_k)$ is approximated by a target depending on the estimate $\hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$.
- If $\hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$ does not fit $v_\pi(\boldsymbol{x}_k)$, the update target becomes a biased estimate of $v_\pi(\boldsymbol{x}_k)$.
  - For example, in the TD(0) case applying SGD we receive:

$$
\begin{aligned}
v_\pi(\boldsymbol{x}) &\approx r + \gamma \hat{v}(\tilde{\boldsymbol{x}}', \boldsymbol{w}), \\
J(\boldsymbol{w}) &\approx \sum_k \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) \right]^2, \\
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &\approx \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) \right] \\
&\quad \nabla_{\boldsymbol{w}} \left[\gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) \right].
\end{aligned}
\tag{2.8}
$$

Semi-gradient methods

When bootstrapping is applied, the gradient does not take into account any gradient component of the bootstrapped target estimate.

- Motivation: speed up gradient calculation while assuming that the simplification error is small (e.g., due to discounting).

Algorithmic implementation: semi-gradient TD(0)

The semi-gradient of $J(\boldsymbol{w})$ for TD(0) from prev. slide is then

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -\left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k). \tag{2.9}$$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
input: a differentiable function $\hat{v} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ with $\hat{v}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$;
        exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 2.2: Semi-gradient TD(0) (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_{\pi}$)

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
input: a differentiable function $\hat{v} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ with $\hat{v}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, prediction steps $n \in \mathbb{Z}^+$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2 \ldots$ *episodes* **do**
    initialize and store $\boldsymbol{x}_0$;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action from $\pi(\boldsymbol{x}_k)$, observe and store $\boldsymbol{x}_{k+1}$ and r_{k+1};
            if $\boldsymbol{x}_{k+1}$ is terminal: $T \leftarrow k + 1$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{v}(\tilde{\boldsymbol{x}}_{\tau+n}, \boldsymbol{w})$;
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g - \hat{v}(\tilde{\boldsymbol{x}}_{\tau}, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_{\tau}, \boldsymbol{w})$;
    until $\tau = T - 1$;

Algo. 2.3: n-step semi-gradient TD (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_{\pi}$)

Table of contents

Background and motivation

- As already discussed in the tabular case: incremental learning is not data efficient (cf. example Fig. **??**).
  - During one incremental learning step we are not utilizing the given information to the maximum possible extent.
  - Also applies to SGD-based updates with function approximation.

- Alternative: batch learning methods
  - Find $\boldsymbol{w}^*$ given a fixed, consistent data set $\boldsymbol{\mathcal{D}} = \{\langle \boldsymbol{x}_0, v_\pi(\boldsymbol{x}_0)\rangle, \langle \boldsymbol{x}_1, v_\pi(\boldsymbol{x}_1)\rangle, \ldots\}$.

- What batch learning options do we have?
  - Experience replay (cf. planning and learning lecture, e.g., Fig. **??**)
  - If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is linear: closed-form least-squares solution

SGD with experience replay

Based on the data set

$$\boldsymbol{\mathcal{D}} = \{\langle \boldsymbol{x}_0, v_\pi(\boldsymbol{x}_0)\rangle, \langle \boldsymbol{x}_1, v_\pi(\boldsymbol{x}_1)\rangle, \ldots\}$$

repeat:

1. Sample uniformly $i = 1, \ldots, b$ state-value pairs from experience (so-called mini batch)

$$\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i)\rangle \sim \boldsymbol{\mathcal{D}}.$$

2. Apply (semi) SGD update step:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \frac{\alpha}{b} \sum_{i=1}^{b} \left[v_\pi(\boldsymbol{x}_i) - \hat{v}(\tilde{\boldsymbol{x}}_i, \boldsymbol{w}_i)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_i, \boldsymbol{w}_i).$$

- Universally applicable: $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ can be any differentiable function.
- The usual technical tuning requirements regarding α apply.
- True target $v_\pi(\boldsymbol{x})$ is usually approximated by MC or TD targets.

(Ordinary) least squares

Assuming the following applies:

- $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is a linear estimator and
- $\mathcal{D}$ a fixed, representative data set following the on-policy distribution.

Then, minimizing the quadratic cost function (2.3) becomes

- an ordinary least squares (OLS) / linear regression problem.

We focus on the combination of OLS and TD(0) (so-called LSTD), but the following can be equally extended to n-step learning or MC.

- Rewriting $J(\boldsymbol{w})$ from (2.3) using linear approximation TD(0) target:

$$v_\pi(\boldsymbol{x}_k) \approx r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}) = r_{k+1} + \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \boldsymbol{w} \tag{2.10}$$

$$J(\boldsymbol{w}) = \sum_k \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}) \right]^2 = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right) \boldsymbol{w} \right]^2 .$$

Ordinary LSTD

The quadratic cost function

$$J(\boldsymbol{w}) = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right) \boldsymbol{w} \right]^2$$

obtains the least squares

- target / dependent variable r_{k+1} and
- regressor / independent variable $\left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right)$.

With b samples we can form a target vector $\boldsymbol{y}$ and regressor matrix $\boldsymbol{\Xi}$:

$$\boldsymbol{y} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_b \end{bmatrix}, \quad \boldsymbol{\Xi} = \begin{bmatrix} \left(\tilde{\boldsymbol{x}}_0^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_1^\mathsf{T} \right) \\ \left(\tilde{\boldsymbol{x}}_1^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_2^\mathsf{T} \right) \\ \vdots \\ \left(\tilde{\boldsymbol{x}}_{b-1}^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_b^\mathsf{T} \right) \end{bmatrix}. \tag{2.11}$$

Ordinary LSTD and regularization

Applying the linear regression solution (1.4) from previous lecture:

LSTD solution

Having arranged $i = 1, \ldots, b$ samples $\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i) \rangle \sim \mathcal{D}$ using TD(0) and linear function approximation as in (2.11), the LSTD solution is

$$\boldsymbol{w}^* = (\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi})^{-1}\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{y}. \tag{2.12}$$

- The parameter $\boldsymbol{w}^*$ is also called the TD fixed point.
- The state-value prediction is simply $\hat{v}(\tilde{\boldsymbol{x}}_k) = \tilde{\boldsymbol{x}}_k^\mathsf{T}\boldsymbol{w}^*$.

Depending on the policy π the rows in $\boldsymbol{\Xi}$ might be linearly correlated.

- Bad matrix condition of $\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi}$ can lead to unfeasible values in $\boldsymbol{w}^*$.
- Counter measure: Add Tikhonov regularization (Ridge regression with penalty term ϵ, cf. (1.6)):

$$\boldsymbol{w}^*_{\text{Ridge}} = (\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi} + \epsilon\boldsymbol{I})^{-1}\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{y}. \tag{2.13}$$

Recursive least squares

- OLS computational complexity is in the range of $\mathcal{O}(\kappa^{2.3}) \dots \mathcal{O}(\kappa^3)$.
  - κ being the number of features.
- Computational costly if new data points $\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i) \rangle$ are added to $\boldsymbol{\mathcal{D}}$.
- Consider supplement / extension: recursive least square (RLS).
  - Each RLS update complexity is $\mathcal{O}(\kappa^2)$.
- In the following, we briefly represent the recipe-style RLS equations.
  - Detailed derivation can be found e.g. R. Isermann and M. Münchhof, *Identification of Dynamic Systems*, Springer-Verlag Berlin Heidelberg, 2011 (also as electronic copy on Panda).

RLS-TD

After every step we receive

- a new regressor vector $\boldsymbol{\xi}_{k+1}^{\mathsf{T}} = \left(\tilde{\boldsymbol{x}}_k^{\mathsf{T}} - \gamma \tilde{\boldsymbol{x}}_{k+1}^{\mathsf{T}}\right)$ and
- a new update target $y_{k+1} = r_{k+1}$.

The RLS update rule is then

$$\begin{aligned} \boldsymbol{c}_k &= \frac{\boldsymbol{P}_k \boldsymbol{\xi}_{k+1}}{\lambda_{k+1} + \boldsymbol{\xi}_{k+1}^{\mathsf{T}} \boldsymbol{P}_k \boldsymbol{\xi}_{k+1}}, \\ \boldsymbol{w}_{k+1} &= \boldsymbol{w}_k + \boldsymbol{c}_k \left(y_{k+1} - \boldsymbol{\xi}_{k+1}^{\mathsf{T}} \boldsymbol{w}_k\right), \\ \boldsymbol{P}_{k+1} &= \left(\boldsymbol{I} - \boldsymbol{c}_k \boldsymbol{\xi}_{k+1}^{\mathsf{T}}\right) \frac{\boldsymbol{P}_k}{\lambda_{k+1}}, \end{aligned} \tag{2.14}$$

with

- $\lambda_k \in \{\mathbb{R} | 0 < \lambda \leq 1\}$ is an optional forgetting factor,
- $\boldsymbol{P}_k$ is the covariance matrix and
- $\boldsymbol{c}_k$ is an adaptive correction to reduce the error $\left(y_{k+1} - \boldsymbol{\xi}_k^{\mathsf{T}} \boldsymbol{w}_{k+1}\right)$.

Algorithmic implementation: RLS-TD

input: a policy π to be evaluated
input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{v}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$)
parameter: forgetting factor $\lambda \in \{\mathbb{R} | 0 < \lambda \leq 1\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily, covariance $\boldsymbol{P} > 0$ (e.g. $\boldsymbol{P} = \beta \boldsymbol{I}$)
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$, observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        $y \leftarrow r_{k+1}$;
        $\boldsymbol{\xi}^\mathsf{T} \leftarrow \tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}$;
        $\boldsymbol{c} \leftarrow (\boldsymbol{P}\boldsymbol{\xi}) \,/\, \left(\lambda + \boldsymbol{\xi}^\mathsf{T} \boldsymbol{P} \boldsymbol{\xi}\right)$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{c}\left(y - \boldsymbol{\xi}^\mathsf{T} \boldsymbol{w}\right)$;
        $\boldsymbol{P} \leftarrow \left(\boldsymbol{I} - \boldsymbol{c}\boldsymbol{\xi}^\mathsf{T}\right) \boldsymbol{P}/\lambda$;
        exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 2.4: RLS-TD (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_\pi$)

Some remarks on RLS usage in RL prediction

- Covariance matrix $\boldsymbol{P}$ can be inspected for certainty analysis.
  - Small-valued elements in $\boldsymbol{P}$ suggest an accurate estimate.
- For $\lambda = 1$ the RLS converges to a static solution.
  - Never forgets something (i.e., problematic for non-stationary problem).
  - Given the same data set $\boldsymbol{\mathcal{D}}$ the RLS converges to OLS solution.
- However, if RLS-TD should be used online $\lambda \in [0.95, 0.99]$ is typical.
  - Application-dependent λ_k might be adapted online after each step.
  - As seen in (2.14), $\lambda < 1$ increases the covariance which potentially could lead to numerical instabilities depending on the given data set.
  - In this case, regularization is required.[1]
- General RLS approach (2.14) is also applicable to MC or n-step TD.
  - Derivation follows presented scheme based on the altered update rules.

[1] Recommended reading: S. Gunnarson, *Combining Tracking and Regularization in Recursive Least Squares Identification*, Proceedings of 35th IEEE Conference on Decision and Control, 1996

Summary: what you've learned today

- To cover unfeasible large or continuous state spaces function approximation is required.
  - Feature engineering supports the learning process.
- On-policy prediction seems rather straightforward with function approximation:
  - Just transfer the incremental learning from tabular case to gradient descent on parameter vector $\boldsymbol{w}$.
  - Stochastic gradient descent allows step-by-step based updates.
- Gradient-based prediction is not risk free (especially non-linear case):
  - no convergence guarantees,
  - local optima vs. global optimum.
- If bootstrapping is applied, the update target depends on $\boldsymbol{w}$.
  - True gradient becomes computationally more complex.
  - Semi-gradient methods reduce computational burden at accuracy costs.
- Batch learning squeezes out all available prediction information from a given data set.
  - If linear function approximation is applied, closed-form solutions exist.

Lecture 10: Value-Based Control with Function Approximation

André
Bodmer

Preface

Problem space: it is further assumed that

- the states $\boldsymbol{x}$ are (quasi-)continuous and
- the actions u are discrete.

Today's focus:

- valued-based control tasks, i.e., transferring the established tabular methods to work with function approximation.
- Hence, we need to extend the previous prediction methods to action values

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q_\pi(\boldsymbol{x}, u). \tag{3.1}$$

- And apply the well-known generalized policy iteration scheme (GPI) to find optimal actions:

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q^*(\boldsymbol{x}, u). \tag{3.2}$$

Types of action-value function approximation

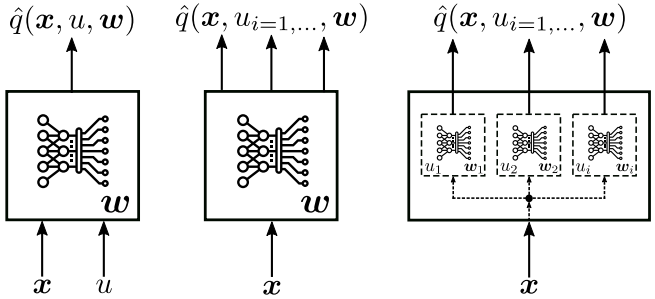


Fig. 3.1: Possible function approximation settings for discrete actions

- Left: one function with both states and actions as input
- Middle: one function with $i = 1, 2, \ldots$ outputs covering the action space (e.g., ANN with appropriate output layer)
- Right: multiple (sub-)functions one for each possible action u_i (e.g., multitude of linear approximators in small action spaces)

Feature engineering

- Also for action-value estimation a proper feature engineering (FE) is of vital importance.
- Compared to the state-value prediction, the action becomes part of the FE processing:

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) = \hat{q}\left(\boldsymbol{f}\left(\boldsymbol{x}, u\right), \boldsymbol{w}\right). \tag{3.3}$$

- Above, $\boldsymbol{f}(\boldsymbol{x}, u) \in \mathbb{R}^{\kappa}$ is the FE function.
- For sake of notation simplicity we write $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ and understand that FE has already been considered (i.e., is a part of $\hat{q}$).

Table of contents

Gradient-based action-value learning

- Transferring the objective (2.3) from on-policy prediction to control yields:

$$J(\boldsymbol{w}) = \sum_k \left[q_\pi(\boldsymbol{x}_k, u_k) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right]^2. \tag{3.4}$$

- Analogous, the (semi-)gradient-based parameter update from (2.7) is also applied to action values:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[q_\pi(\boldsymbol{x}_k, u_k) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k). \tag{3.5}$$

- Depending on the control approach, the true target $q_\pi(\boldsymbol{x}_k, u_k)$ is approximated by:
  - Monte Carlo: full episodic return $q_\pi(\boldsymbol{x}_k, u_k) \approx g$,
  - SARSA: one-step bootstrapped estimate $q_\pi(\boldsymbol{x}_k, u_k) \approx r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u_{k+1}, \boldsymbol{w}_k)$,
  - n-step SARSA: $q_\pi(\boldsymbol{x}_k, u_k) \approx r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}(\boldsymbol{x}_{k+n}, u_{k+n}, \boldsymbol{w}_{k+n-1})$.

Houston: we have a problem

- Recall tabular policy improvement theorem (Theo. **??**): guarantee to find a globally better or equally good policy in each update step.
- With parameter updates (3.5) generalization applies.
- Hence, when reacting to one specific state-action transition other parts of the state-action space within $\hat{q}$ are affected too.

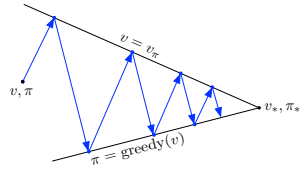


Fig. 3.2: GPI

Loss of policy improvement theorem

- Is not applicable with function approximation!
- We may improve and impair the policy at the same time!

Algorithmic implementation: gradient MC control

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $q_\pi(\boldsymbol{x}_k, u_k) \approx g_k$
- If operating ε-greedy on $\hat{q}$: baseline policy (given by $\boldsymbol{w}_0$) must (successfully) terminate the episode!

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate episode following π or ε-greedy on $\hat{q}$: $x_0, u_0, r_1, \ldots, x_T$;
  calculate every-visit return g_k;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;

Algo. 3.1: Every-visit gradient MC-based action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

Algorithmic implementation: semi-gradient SARSA

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$ or ε-greedy on $\hat{q}(\boldsymbol{x}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        if $\boldsymbol{x}_{k+1}$ *is terminal* **then**
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;
            go to next episode;
        choose u' from $\pi(\boldsymbol{x}_{k+1})$ or ε-greedy on $\hat{q}(\boldsymbol{x}_{k+1}, \cdot, \boldsymbol{w})$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u', \boldsymbol{w}) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;

Algo. 3.2: Semi-gradient SARSA action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

SARSA application example: mountain car (1)

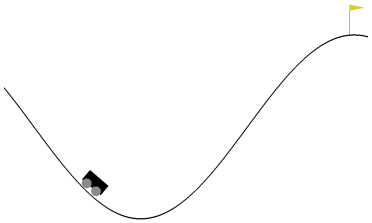


Fig. 3.3: Classic RL control example: mountain car (derivative work based on https://github.com/openai/gym, MIT license)

- Two cont. states: position, velocity
- One discrete action: acceleration given by $\{\text{left}, \text{none}, \text{right}\}$
- $r_k = -1$, i.e., goal is to terminate episode as quick as possible
- Episode terminates when car reaches the flag (or max steps)
- Simplified longitudinal car physics with state constraints
- Position initialized randomly within valley, zero initial velocity
- Car is underpowered and requires swing-up

SARSA application example: mountain car (2)

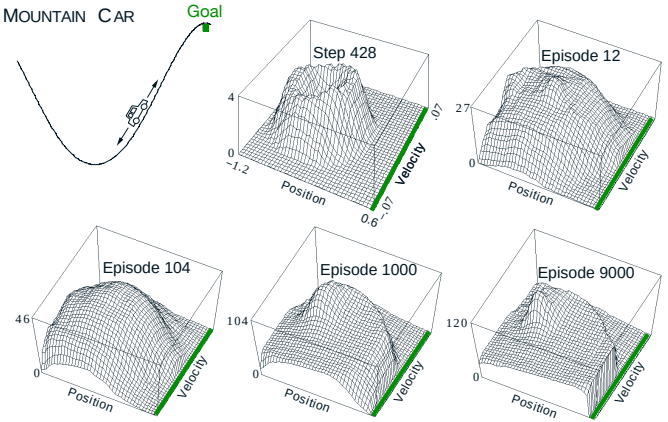


Fig. 3.4: Cost-to-go function $-\max_u \hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ for mountain car task using linear approximation with SARSA and tile coding (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Tile coding

- Problem space is grouped into (overlapping) partitions / tiles.
- Performs a discretization of the problem space.
- Function approximation serves as interpolation between tiles.
- Find an example here: https://github.com/MeepMoop/tilecoding .

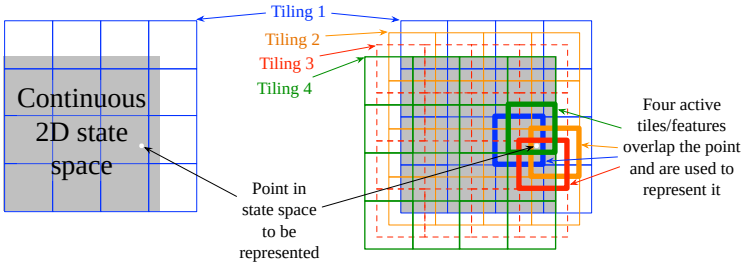


Fig. 3.5: Tile coding example in 2D (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$
input: a policy π (only if estimating q_{π})
parameter: $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, $n \in \mathbb{Z}^{+}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2 \ldots$ *episodes* **do**
    initialize and store $\boldsymbol{x}_0$;
    select and store $u_0 \sim \pi(\boldsymbol{x}_0)$ or ε-greedy w.r.t. $\hat{q}(\boldsymbol{x}_0, \cdot, \boldsymbol{w})$;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action u_k observe and store $\boldsymbol{x}_{k+1}$ and r_{k+1};
            if $\boldsymbol{x}_{k+1}$ *is terminal* **then** $T \leftarrow k + 1$;
            else select & store $u_{k+1} \sim \pi(\boldsymbol{x}_{k+1})$ or ε-greedy w.r.t. $\hat{q}(\boldsymbol{x}_{k+1}, \cdot, \boldsymbol{w})$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(\boldsymbol{x}_{\tau+n}, u_{\tau+n}, \boldsymbol{w})$;
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g - \hat{q}(\boldsymbol{x}_{\tau}, u_{\tau}, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_{\tau}, u_{\tau}, , \boldsymbol{w})$;
    until $\tau = T - 1$;

Algo. 3.3: n-step semi-gradient SARSA (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_{\pi}$ or $\hat{q}^{*}$)

Table of contents

Transferring LSTD-style batch learning to action values

- In the previous lecture we developed a closed-form batch learning tool: LSTD.
  - Linear function approximation.
  - Fixed, representative data set $\mathcal{D}$.
- Same idea can be transferred to action values when bootstrapping with one-step Sarsa, called LS-SARSA (or sometimes LSTDQ):

$$\begin{aligned} q_\pi(\boldsymbol{x}_k, u_k) &\approx r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u_{k+1}, \boldsymbol{w}_k), \\ \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k) &= \hat{q}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) = \tilde{\boldsymbol{x}}_k^\mathsf{T} \boldsymbol{w}_k. \end{aligned} \tag{3.6}$$

- The cost function for action-value prediction is then:

$$J(\boldsymbol{w}) = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right) \boldsymbol{w} \right]^2. \tag{3.7}$$

- Hence, the closed-form least squares solution for the action values is the same as for the state value case but the feature vector depends also on the actions:

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{f}(\boldsymbol{x}_k, u_k).$$

On and off-policy LS-SARSA

With b samples we can form a target vector $\boldsymbol{y}$ and regressor matrix $\boldsymbol{\Xi}$:

$$\boldsymbol{y} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_b \end{bmatrix}, \quad \boldsymbol{\Xi} = \begin{bmatrix} \left(\tilde{\boldsymbol{x}}_0^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_1^\mathsf{T}\right) \\ \left(\tilde{\boldsymbol{x}}_1^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_2^\mathsf{T}\right) \\ \vdots \\ \left(\tilde{\boldsymbol{x}}_{b-1}^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_b^\mathsf{T}\right) \end{bmatrix}. \tag{3.8}$$

Regarding the data input to $\boldsymbol{\Xi}$ we can distinguish two cases: The actions u_k and u_{k+1} in the feature pair $\left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}\right)$ per row in $\boldsymbol{\Xi}$ either descends from the

- same policy π (on-policy learning) or
- the action u_{k+1} in $\tilde{\boldsymbol{x}}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_{k+1}, u_{k+1})$ is chosen based on an arbitrary policy π' (off-policy learning).

If we apply off-policy LS-SARSA then

- we retrieve the flexibility to collect training samples arbitrarily
- at the cost of an estimation bias based on the sampling distribution.

LS-SARSA

LS-SARSA solution

Having arranged $i = 1, \ldots, b$ samples $\langle \boldsymbol{x}_i, u_i, r_{i+1}, \boldsymbol{x}_{i+1}, u_{i+1} \rangle \sim \boldsymbol{\mathcal{D}}$ using one-step bootstrapping (3.6) and linear function approximation as in (3.8), the LS-SARSA solution is

$$\boldsymbol{w}^* = (\boldsymbol{\Xi}^\mathsf{T} \boldsymbol{\Xi})^{-1} \boldsymbol{\Xi}^\mathsf{T} \boldsymbol{y}. \tag{3.9}$$

Again, basic usage distinction:

- If $\{u_i, u_{i+1}\} \sim \pi$: on-policy prediction (as in LSTD)
- If $u_i \sim \pi$ and $u_{i+1} \sim \pi'$: off-policy prediction (useful for control)

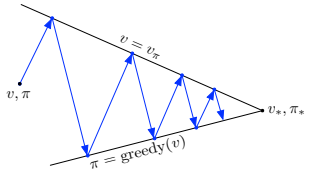
Possible modifications:

- To prevent numeric instability regularization is possible, cf. (2.13)
- Recursive implementation for online usage straightforward, cf. (2.14)

Least squares policy iteration (LSPI)

General idea:

- Apply general policy improvement (GPI) based on data set $\mathcal{D}$,
- Policy evaluation by off-policy LS-SARSA,
- Policy improvement by greedy choices on predicted action values.



Some remarks:

- LSPI is an offline and off-policy control approach.
- Exploration is required by feeding suitable sampling distributions in $\mathcal{D}$:
  - Such as ε-greedy choices based on $\hat{q}$.
  - But also complete random samples are conceivable.

Algorithmic implementation: LSPI

input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{q}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$)
input: a data set $\langle \boldsymbol{x}_i, u_i, r_{i+1}, \boldsymbol{x}_{i+1} \rangle \sim \boldsymbol{\mathcal{D}}$ with $i = 1, \ldots, b$ samples
parameter: an accuracy threshold $\Delta \in \{\mathbb{R} | 0 < \Delta\}$
init: linear approximation function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
$\pi \leftarrow \arg\max_u \hat{q}(\cdot, u, \boldsymbol{w})$ (greedy choices based on $\hat{q}(\boldsymbol{w})$);
repeat
    $\boldsymbol{w}' \leftarrow \boldsymbol{w}$;
    $\boldsymbol{w} \leftarrow$ LS-SARSA$(\boldsymbol{\mathcal{D}}, u_{i+1} \sim \pi)$;
    $\pi \leftarrow \arg\max_u \hat{q}(\cdot, u, \boldsymbol{w})$;
until $||\boldsymbol{w}' - \boldsymbol{w}|| < \Delta$;

Algo. 3.4: Least squares policy iteration (output: $\boldsymbol{w}$ for $\hat{q}^*$)

- In a (small) discrete action space the $\arg\max_u$ operation is straightforward.
- After one full LSPI evaluation the data set $\boldsymbol{\mathcal{D}}$ might be altered to include new data obtained based on the updated $\boldsymbol{w}$ vector.
- Source: M. Lagoudakis and R. Parr, *Least-Squares Policy Iteration*, Journal of Machine Learning Research 4, pp. 1107-1149, 2003

LSPI application example: inverted pendulum (1)

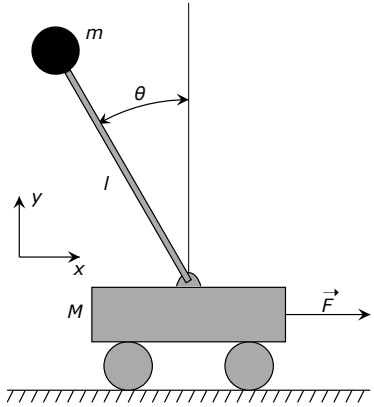


Fig. 3.6: Classic RL control example: inverted pendulum (source: www.wikipedia.org, CC0 1.0)

- Two continuous states: angular position θ and velocity $\dot{\theta}$
- One discrete action: acceleration force (i.e., torque at shaft)
- Action noise as disturbance
- Non-linear system dynamics
- State initialization randomly close to upper equilibrium
- $r_k = 0$ if pendulum is above horizontal line
- $r_k = -1$ if below horizontal line and episode terminates
- $\gamma = 0.95$

LSPI application example: inverted pendulum (2)

- Initial training samples for $\mathcal{D}$ following a policy selecting actions at uniform probability
- Additional samples have been manually added during the training
- Radial basis function as feature engineering

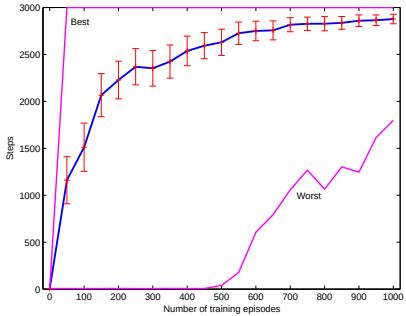


Fig. 3.7: Balancing steps before episode termination with a clipping of maximum 3000 steps (source: M. Lagoudakis and R. Parr, *Least-Squares Policy Iteration*, Journal of Machine Learning Research 4, pp. 1107-1149, 2003)

Algorithmic implementation: online LSPI

input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{q}(\tilde{\boldsymbol{x}}_T, \cdot, \cdot) = 0$)
parameter: forgetting factor $\lambda \in \{\mathbb{R} | 0 < \lambda \leq 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily, policy π being ε-greedy w.r.t. $\hat{q}(\boldsymbol{w})$, covariance $\boldsymbol{P} > 0$
for $j = 1, 2, \ldots$ *episodes* **do**
  initialize $\boldsymbol{x}_0$ and set $u_0 \sim \pi(\boldsymbol{x}_0)$;
  for $k = 0, 1, 2 \ldots$ *time steps* **do**
    apply action u_k, observe $\boldsymbol{x}_{k+1}$ and r_{k+1}, set $u_{k+1} \sim \pi(\boldsymbol{x}_{k+1})$;
    $y \leftarrow r_{k+1}$;
    $\boldsymbol{\xi}^\mathsf{T} \leftarrow \tilde{\boldsymbol{x}}_k^\mathsf{T}(\boldsymbol{x}_k, u_k) - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}(\boldsymbol{x}_{k+1}, u_{k+1})$;
    $\boldsymbol{c} \leftarrow (\boldsymbol{P}\boldsymbol{\xi}) \,/\, \left(\lambda + \boldsymbol{\xi}^\mathsf{T}\boldsymbol{P}\boldsymbol{\xi}\right)$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{c}\left(y - \boldsymbol{\xi}^\mathsf{T}\boldsymbol{w}\right)$;
    $\boldsymbol{P} \leftarrow \left(\boldsymbol{I} - \boldsymbol{c}\boldsymbol{\xi}^\mathsf{T}\right)\boldsymbol{P}/\lambda$;
    if $k \mod k_w = 0$ **then**
      $\pi \leftarrow \varepsilon$-greedy w.r.t. $\hat{q} = \tilde{\boldsymbol{x}}^\mathsf{T}(\boldsymbol{x}, u)\boldsymbol{w}$;
    exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 3.5: Online LSPI with RLS-SARSA (output: $\boldsymbol{w}$ for $\hat{q}^*$)

Remarks on online LSPI

- k_w depicts the number of steps between policy improvement cycles.
- Forgetting factor λ and k_w require mutual tuning:
  - After each policy improvement the policy evaluation requires sample updates to accurately predict the altered policy.
  - Numerically instability may occur for $\lambda < 1$ and requires regularization.
- Hence, the algorithm is online-capable but its policy is normally not updated in a step-by-step fashion.
- Alternative online LSPI with OLS-SARSA can be found in L. Buşoniu et al., *Online least-squares policy iteration for reinforcement learning control*, American Control Conference, 2010.

Table of contents

General background on DQN

- Recall incremental learning step from tabular Q-learning:

$$\hat{q}(x,u) \leftarrow \hat{q}(x,u) + \alpha \left[r + \gamma \max_u \hat{q}(x',u) - \hat{q}(x,u) \right].$$

- Deep Q-networks (DQN) transfer this to an approximate solution:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[r + \gamma \max_u \hat{q}(\boldsymbol{x}',u,\boldsymbol{w}) - \hat{q}(\boldsymbol{x},u,\boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x},u,\boldsymbol{w}). \tag{3.10}$$

However, instead of using above semi-gradient step-by-step updates, DQN is characterized by

- an experience replay buffer for batch learning (cf. prev. lectures),
- a separate set of weights $\boldsymbol{w}^-$ for the bootstrapped Q-target.

Motivation behind:

- Efficiently use available data (experience replay).
- Stabilize learning by trying to make targets and feature inputs more like i.i.d. data from a stationary process (prevent windup of values).

Summary of DQN working principle (1)

- Take actions u based on $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ (e.g., ε-greedy).
- Store observed tuples $\langle \boldsymbol{x}, u, r, \boldsymbol{x}' \rangle$ in memory buffer $\boldsymbol{\mathcal{D}}$.
- Sample mini-batches $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$.
- Calculate bootstrapped Q-target with a delayed parameter vector $\boldsymbol{w}^-$ (so-called target network):

$$q_\pi(\boldsymbol{x}, u) \approx r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}^-).$$

- Optimize MSE loss between above targets and the regular approximation $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ using $\boldsymbol{\mathcal{D}}_b$

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}^-) \right) - \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \right]^2_{\boldsymbol{\mathcal{D}}_b}. \tag{3.11}$$

- Update $\boldsymbol{w}^-$ based on $\boldsymbol{w}$ from time to time.

Summary of DQN working principle (2)

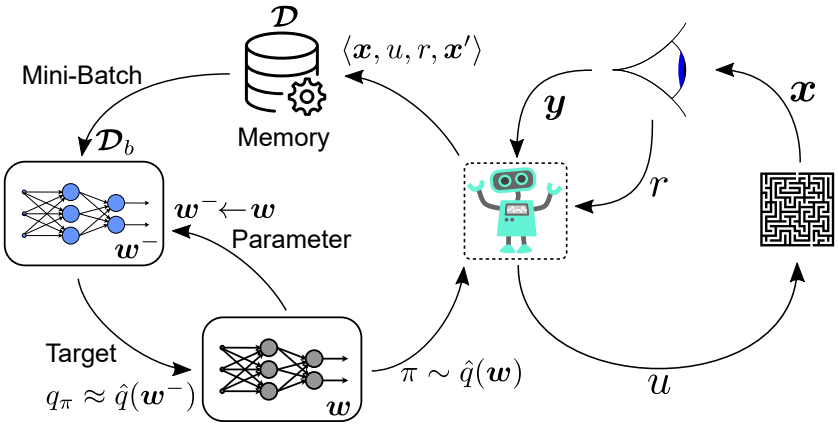


Fig. 3.8: DQN structure from a bird's-eye perspective (derivative work of Fig. **??** and wikipedia.org, CC0 1.0)

Algorithmic implementation: DQN

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ (including feature eng.)
parameter: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} = \boldsymbol{w}^- \in \mathbb{R}^{\zeta}$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$ with certain capacity
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action ε-greedy w.r.t $\hat{q}(\boldsymbol{x}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        store tuple $\langle \boldsymbol{x}_k, u_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
        sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
        for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
            if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
            else $y_i = r_{i+1} + \gamma \max_u \hat{q}(\boldsymbol{x}_{i+1}, u, \boldsymbol{w}^-)$;
        fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y_i - \hat{q}(\boldsymbol{x}_i, u_i, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$;
        if $k \mod k_w = 0$ **then** $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ (update target weights);

Algo. 3.6: DQN (output: parameter vector $\boldsymbol{w}$ for $\hat{q}^*$)

Remarks on DQN implementation

- General framework is based on V. Mnih et al., *Human-level control through deep reinforcement learning*, Nature, pp. 529-533, 2015.
- Often 'deep' artificial neural networks are used as function approximation for DQN.
  - Nevertheless, other model topologies are fully conceivable.
- The fit of $\boldsymbol{w}$ on loss $\mathcal{L}$ is an intermediate supervised learning step.
  - Comes with degrees of freedom regarding solver choice.
  - Has own optimization parameters which are not depicted here in details (many tuning options).
- Mini-batch sampling from $\mathcal{D}$ is often randomly distributed.
  - Nevertheless, guided sampling with useful distributions for a specific control task can be beneficial (cf. Dyna discussion in 7th lecture).
- Likewise, the simple ε-greedy approach can be extended.
  - Often a scheduled/annealed trajectory ε_k is used.
  - Again referring to the Dyna framework, many more exploration strategies are possible.

DQN application example: Atari games (1)

- End-to-end learning of $\hat{q}(\boldsymbol{x}, u)$ from monitor pixels $\boldsymbol{x}$
- Feature engineering obtains stacking of raw pixes from last 4 frames
- Actions u are 18 possible joystick/button combinations
- Reward is the change of highscore per step
- Interesting lecture from V. Minh with more details: YouTube

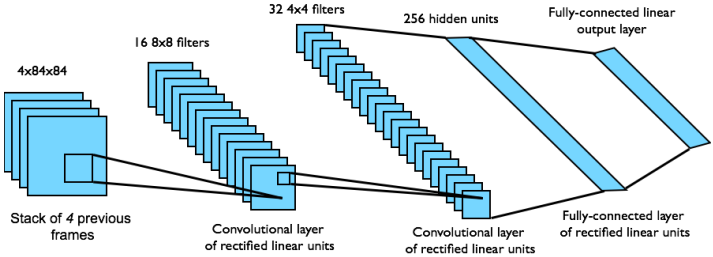


Fig. 3.9: Network architecture overview used for DQN in Atari games (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

DQN application example: Atari games (2)

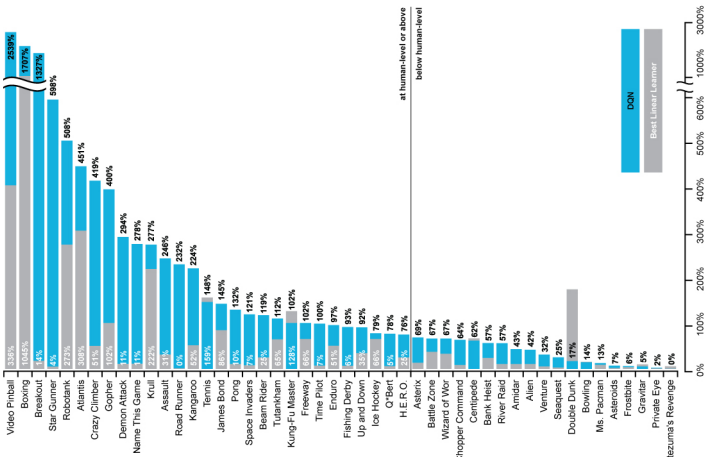


Fig. 3.10: DQN performance results in Atari games against human performance (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Summary: what you've learned today

- From a simplified perspective, the procedures from the approximate prediction can simply be transferred to value-based control.
- On the contrary, the policy improvement theorem no longer applies in the approximate RL case (generalization impact).
  - Control algorithms may diverge completely.
  - Or a performance trade-off between different parts of the problem space could emerge.
- Off-policy batch learning approaches allow for efficient data usage.
  - LSPI uses LS-SARSA on linear function approximation.
  - DQN extends Q-learning on non-linear approximation with additional tweaks (experience replay, target networks,...).
  - However, a prediction bias results (off-policy sampling distribution).

Lecture 11: Stochastic Policy Gradient Methods

André
Bodmer

Preface (1)

Shift from (indirect) value-based approaches

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q(\boldsymbol{x}, u) \tag{4.1}$$

to (direct) policy-based solutions:

$$\boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}) = \mathbb{P}\left[\boldsymbol{U} = \boldsymbol{u}|\boldsymbol{X} = \boldsymbol{x}\right] \approx \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\,. \tag{4.2}$$

- Above, $\boldsymbol{\theta} \in \mathbb{R}^d$ is the policy parameter vector.
- Note, that $\boldsymbol{u}$ is now vectorial and might contain multiple continuous quantities.

Goal of today's lecture

- Introduce an algorithm class based on a parameterizable policy $\pi(\boldsymbol{\theta})$.
- Extend the action space to continuous actions.
- Combine the policy-based and value-based approach.

Preface (2)

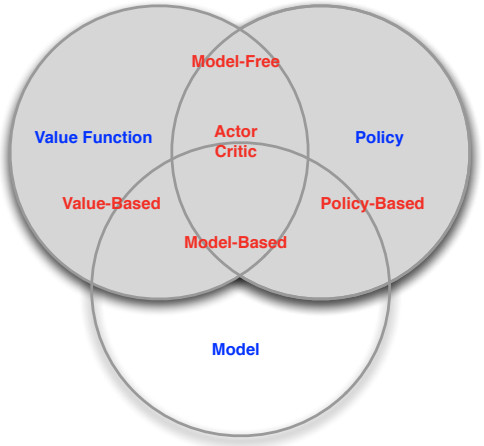


Fig. 4.1: Main categories of reinforcement learning algorithms
(source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Table of contents

Motivating example: strategic gaming

Task: Two-player game of extended rock-paper-scissors

- A deterministic policy (i.e., value-based with given feature representation) can be easily exploited by the opponent.
- Conversely, a uniform random policy would be unpredictable.

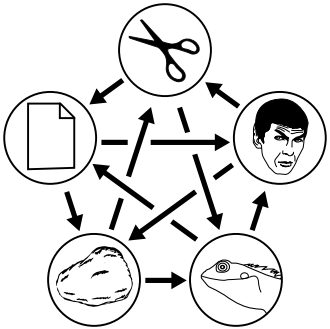


Fig. 4.2: Rock paper scissors lizard Spock game mechanics
(source: www.wikipedia.org, by Diriector Doc CC BY-SA 4.0)

Example policy function: discrete action space

Assumption:

- Action space is discrete and compact.

A typical policy function is:

- Soft-max in action preferences

$$\pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \frac{e^{h(\boldsymbol{x},u,\boldsymbol{\theta})}}{\sum_i e^{h(\boldsymbol{x},i,\boldsymbol{\theta})}} \tag{4.3}$$

with $h(\boldsymbol{x},u,\boldsymbol{\theta}) : \mathcal{X}\times\mathcal{U}\times\mathbb{R}^d \to \mathbb{R}$ being the numerical preference per state-action pair.

- Denominator of (4.3) sums up action probabilities to one per state.
- Is designed as a stochastic policy but can approach deterministic behavior in the limit.
- The preference is parametrized via a function approximator, e.g., linear in features

$$h(\boldsymbol{x},u,\boldsymbol{\theta}) = \boldsymbol{\theta}^{\mathsf{T}}\tilde{\boldsymbol{x}}(\boldsymbol{x},u). \tag{4.4}$$

Example policy function: continuous action space (1)

Assumption:

- Action space is continuous and there is only one scalar action $u \in \mathbb{R}$.

A typical policy function is:

- Gaussian probability density

$$\pi(u|\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{\sigma(\boldsymbol{x}, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(u - \mu(\boldsymbol{x}, \boldsymbol{\theta}))^2}{2\sigma(\boldsymbol{x}, \boldsymbol{\theta})^2}\right) \tag{4.5}$$

  with mean $\mu(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}$ and standard deviation $\sigma(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}$ given by parametric function approximation.

- Variants regarding function μ and σ:
  1. Both share a mutual parameter set $\boldsymbol{\theta}$ (e.g., artificial neural network with multiple outputs).
  2. Both are parametrized independently $\boldsymbol{\theta} = \begin{bmatrix}\boldsymbol{\theta}_\mu & \boldsymbol{\theta}_\sigma\end{bmatrix}^\mathsf{T}$ (e.g., by two linear regression functions).
  3. Only $\mu(\boldsymbol{x}, \boldsymbol{\theta})$ is parametrized while σ is scheduled externally.

Example policy function: continuous action space (2)

- Output of the functions $\mu_k = (\boldsymbol{x}_k, \boldsymbol{\theta}_k)$ and $\sigma_k = (\boldsymbol{x}_k, \boldsymbol{\theta}_k)$ can change in every time step.
- Depending on σ exploration is an inherent part of the (stochastic) policy.

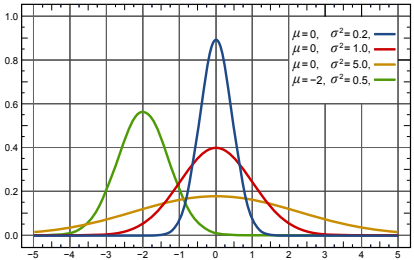


Fig. 4.3: Exemplary univariate Gaussian probability density functions (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Example policy function: continuous action space (3)

Assumption:

- Action space is continuous and there are multiple actions $\boldsymbol{u} \in \mathbb{R}^m$.

A typical policy function is:

- Multivariate Gaussian probability density

$$\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{\sqrt{(2\pi)^m \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{u}-\boldsymbol{\mu})^\mathsf{T}\boldsymbol{\Sigma}^{-1}(\boldsymbol{u}-\boldsymbol{\mu})\right) \tag{4.6}$$

  with mean $\boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}^m$ and covariance matrix $\boldsymbol{\Sigma}(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}^{m \times m}$ given by parametric function approximation.
- Same parametrization variants apply to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as in the scalar action case.
- In addition, $\boldsymbol{\Sigma}$ can be considered a diagonal matrix and clipped to reduce complexity as well as ensure nonsingularity.

Example policy function: continuous action space (4)

- Below we find an example for

$$\boldsymbol{\mu} = \begin{bmatrix} -0.4 & 0.3 \end{bmatrix}^\mathsf{T} \quad \text{and} \quad \boldsymbol{\Sigma} = \begin{bmatrix} 0.04 & 0 \\ 0 & 0.02 \end{bmatrix}.$$

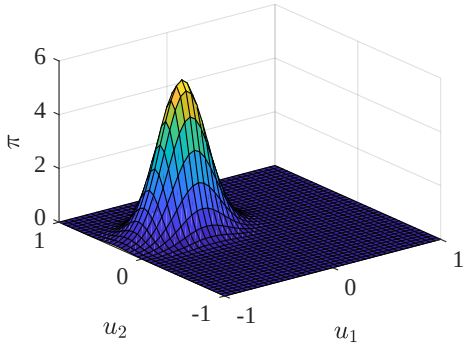


Fig. 4.4: Exemplary bivariate Gaussian probability density function

Policy objective function

- Goal: find optimal $\boldsymbol{\theta}^*$ given the policy $\boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$.
- Problem: which measure of optimality should we use?

Possible optimality metrics:

- Start state value (in episodic tasks):

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{x}_0) = \mathbb{E}\left[v|\boldsymbol{X} = \boldsymbol{x}_0, \boldsymbol{\theta}\right] \tag{4.7}$$

- Average reward (in continuing tasks):

$$J(\boldsymbol{\theta}) = \overline{r}_{\pi_{\boldsymbol{\theta}}} = \int_{\mathcal{X}} \mu_\pi(\boldsymbol{x}) \int_{\mathcal{U}} \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \int_{\mathcal{X},\mathcal{R}} p(\boldsymbol{x}', r|\boldsymbol{x}, \boldsymbol{u}) r \tag{4.8}$$

  - Above, $\mu_\pi(\boldsymbol{x})$ is again the steady-state distribution $\mu_\pi(\boldsymbol{x}) = \lim_{k\to\infty} \mathbb{P}\left[\boldsymbol{X}_k = \boldsymbol{x}|\boldsymbol{U}_{0:k-1} \sim \pi\right]$.

Policy optimization

- In essence, policy-based RL is an optimization problem.
- Depending on the policy function and task, finding $\boldsymbol{\theta}^*$ might be a
  - non-linear,
  - multidimensional and
  - non-stationary problem.
- Hence, we might consider global optimization techniques[1] like
  - Simple heuristics: random search, grid search,...
  - Meta-heuristics: evolutionary algorithms, particle swarm,....
  - Surrogate-model-based optimization: Bayes opt.,...
  - Gradient-based techniques with multi-start initialization.

[1] Recommended reading: J. Stork et al., *A new Taxonomy of Continuous Global Optimization Algorithms*, https://arxiv.org/abs/1808.08818, 2020

Policy gradient

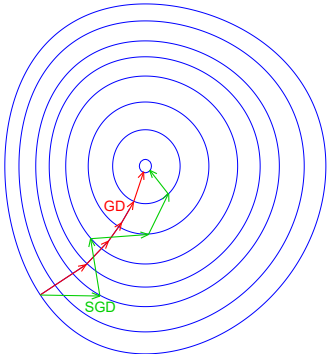


Fig. 4.5: Exemplary optimization paths for (stochastic) gradient ascent (derivative work of www.wikipedia.org, CC0 1.0)

- We will focus on gradient-based methods (policy gradient).
- Hence, we will assume that the gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \left[\frac{\partial J}{\partial \theta_1} \quad \cdots \quad \frac{\partial J}{\partial \theta_d} \right]^{\mathsf{T}}$$

  required for gradient ascent optimization always exists:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

- True gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is usually approximated, e.g., by stochastic gradient descent (SGD) or derived variants.

Policy gradient theorem

Theorem 4.1: Policy Gradient

Given a metric $J(\boldsymbol{\theta})$ for the undiscounted episodic (4.7) or continuing tasks (4.8) and a parameterizable policy $\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$ the policy gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) \frac{\nabla_{\boldsymbol{\theta}} \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})}{\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})}\right]. \tag{4.9}$$

- Having samples $\langle \boldsymbol{x}_i, \boldsymbol{u}_i \rangle$, an estimate of q_{π} and the policy function $\pi(\boldsymbol{\theta})$ available we receive an analytical solution for the policy gradient!
- Using identity $\nabla \ln a = \frac{\nabla a}{a}$ we can re-write to

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\right] \tag{4.10}$$

  with $\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$ also called the score function.
- Derivation available in chapter 13.2 / 13.6 in the lecture book of Barto and Sutton.

Intuitive interpretation of policy parameter update

- Inserting the policy gradient theorem into gradient ascent approach:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbb{E}_{\pi} \left[q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) \frac{\nabla_{\boldsymbol{\theta}} \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})}{\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})} \right].$$

- Move in the direction that favor actions that yield an increased value.
- Scale the update step size inversely to the action probability to compensate that some actions are selected more frequently.

Also note:

- The policy gradient is not depending on the state distribution!
- Hence, we do not need any knowledge of the environment and receive a model-free RL approach!

Simple score function examples

Soft-max policy with linear function approximation:

$$\pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u)}}{\sum_i e^{\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},i)}}$$

$$\Leftrightarrow \quad \nabla_{\boldsymbol{\theta}} \ln \pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u) - \ln \left(\sum_i e^{\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},i)} \right) \right)$$

$$= \tilde{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}) - \mathbb{E}_\pi \left[\tilde{\boldsymbol{x}}(\boldsymbol{x},\cdot) \right]$$

Univariate Gaussian policy with linear function approximation and given σ:

$$\pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(u - \boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u))^2}{2\sigma^2} \right)$$

$$\Leftrightarrow \quad \nabla_{\boldsymbol{\theta}} \ln \pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\ln \left(\frac{1}{\sigma\sqrt{2\pi}} \right) - \frac{(u - \boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u))^2}{2\sigma^2} \right)$$

$$= \frac{\left(u - \boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u)\right)\tilde{\boldsymbol{x}}(\boldsymbol{x},u)}{\sigma^2}$$

Pro and cons: policy vs. value-based approaches

Pro value-based solutions (e.g., Q-learning):

- Estimated value is an intuitive performance metric.
- Considered sample-efficient (cf. replay buffer or bootstrapping).

Pro policy-based solutions (e.g., using policy gradient):

- Seamless integration of stochastic and dynamic policies.
- Straightforward applicable to large/continuous action spaces. In contrast, value-based approaches would require explicit optimization

$$\boldsymbol{u}^* = \arg\max_{\boldsymbol{u}} q(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}).$$

Mutual hassle:

- Gradient-based optimization with (non-linear) function approximation is likely to deliver only suboptimal and local policy optima.

Table of contents

Basic concept

Initial situation:

- Score function $\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$ can be calculated analytically using suitable policy and chain rule (e.g., by algorithmic differentiation).
- Open question: how to retrieve $q_\pi(\boldsymbol{x}, \boldsymbol{u})$ in

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[q_\pi(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right] \ ?$$

Monte Carlo policy gradient:

- Use sampled episodic return g_k to approximate $q_\pi(\boldsymbol{x}, \boldsymbol{u})$:

$$q_\pi(\boldsymbol{x}, \boldsymbol{u}) \approx g_k$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \gamma^k g_k \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta}_k).$$

- The discounting of the policy gradient is introduced as an extension to Theo. 4.1 (which assumed an undiscounted episodic task).
- Also known as *REINFORCE* approach.

Algorithmic implementation: Monte Carlo policy gradient (REINFORCE)

- Usual technical convergence requirements regarding α apply.
- Use sampled return as unbiased estimate of q.
- Recall previous MC-based methods: high variance, slow learning.

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$
parameter: step size $\alpha \in \{\mathbb{R}|0<\alpha<1\}$
init: parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate an episode following $\pi(\cdot|\cdot,\boldsymbol{\theta})$: $\boldsymbol{x}_0, \boldsymbol{u}_0, r_1, \ldots, \boldsymbol{x}_T$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $g \leftarrow \sum_{i=k+1}^{T} \gamma^{i-k-1} r_i$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^k g \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k,\boldsymbol{\theta})$;

Algo. 4.1: Monte Carlo policy gradient (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}^*)$)

REINFORCE example: short-corridor problem (1)

- Gridworld style problem with two actions: left (l), right (r)
- Second-left state's action execution is reversed
- Feature representation: $\tilde{x}(x, u = \mathsf{r}) = \begin{bmatrix} 1 & 0 \end{bmatrix}^\mathsf{T}$, $\tilde{x}(x, u = \mathsf{l}) = \begin{bmatrix} 0 & 1 \end{bmatrix}^\mathsf{T}$
- A policy-based approach searches for the optimal probability split

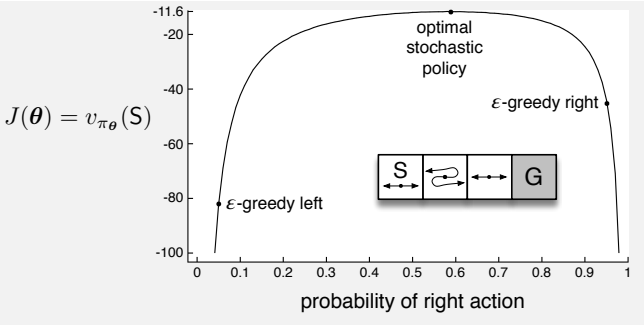


Fig. 4.6: Short-corridor problem with $\varepsilon = 0.1$ (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

REINFORCE example: short-corridor problem (2)

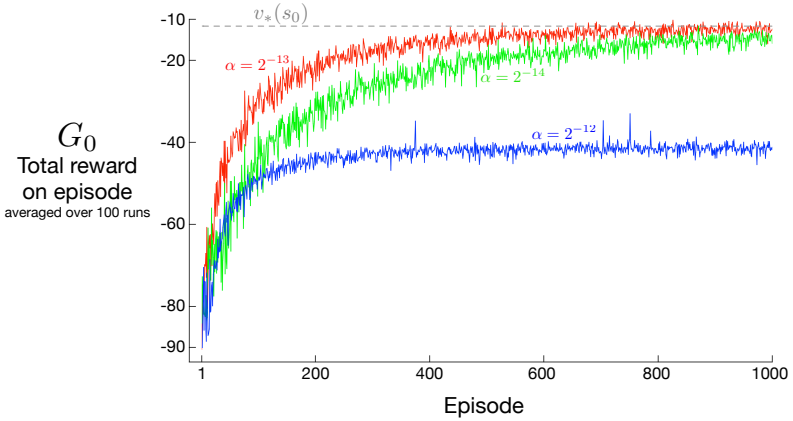


Fig. 4.7: Comparison of Monte Carlo policy gradient approach on short-corridor problem from Fig. 4.6 for different learning rates (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Baseline

- Motivation: add a comparison term to the policy gradient to reduce variance while not affecting its expectation.
- Introduce the baseline $b(\boldsymbol{x})$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi} \left[\left(q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) - b(\boldsymbol{x}) \right) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right]. \tag{4.11}$$

- Since $b(\boldsymbol{x})$ is only depending on the state but not on the actions/policy we did not change the policy gradient in expectation:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi} \left[q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right] - \underbrace{\mathbb{E}_{\pi} \left[b(\boldsymbol{x}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right]}_{=0}.$$

- Consequently, the Monte Carlo policy parameter update yields:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \gamma^k \left(g_k - b(\boldsymbol{x}_k) \right) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta}_k).$$

Advantage function

- Intuitive choice of the baseline is the state value $b(\boldsymbol{x}) = v_\pi(\boldsymbol{x})$.
- The resulting policy gradient becomes

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\left(q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x}) \right) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right]. \tag{4.12}$$

- Here, the difference between action and state value is the advantage function

$$a_\pi(\boldsymbol{x}, \boldsymbol{u}) = q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x}). \tag{4.13}$$

- Interpretation: value difference taking (arbitrary) action $\boldsymbol{u}$ and thereafter following policy π compared to the state value following same policy (i.e., choosing $\boldsymbol{u} \sim \pi$) given the state.
- Hence, we might rewrite to:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[a_\pi(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right]. \tag{4.14}$$

Algo. implementation: MC policy gradient with baseline

- Implementation requires approximation $b(\boldsymbol{x}) \approx \hat{v}(\boldsymbol{x}, \boldsymbol{w})$.
- Hence, we are learning two parameter sets $\boldsymbol{\theta}$ and $\boldsymbol{w}$.
- Keep using sampled return as action-value estimate: $q_\pi(\boldsymbol{x}, \boldsymbol{u}) \approx g_k$.

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$ and state-value function $\hat{v}(\boldsymbol{x}, \boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate an episode following $\pi(\cdot|\cdot, \boldsymbol{\theta})$: $\boldsymbol{x}_0, \boldsymbol{u}_0, r_1, \ldots, \boldsymbol{x}_T$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $g \leftarrow \sum_{i=k+1}^{T} \gamma^{i-k-1} r_i$;
    $\delta \leftarrow g - \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \gamma^k \delta \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta})$;

Algo. 4.2: Monte Carlo policy gradient with baseline (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x}, \boldsymbol{w}^*)$)

REINFORCE comparison w/o baseline

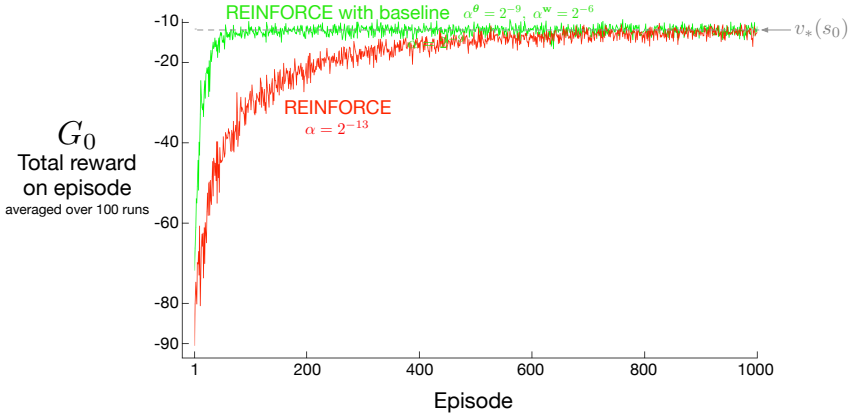


Fig. 4.8: Comparison of Monte Carlo policy gradient on short-corridor problem from Fig. 4.6 where both algorithms' learning rates have been tuned (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Table of contents

General actor-critic idea

Conclusion of Monte Carlo policy gradient with baseline:

- Will learn an unbiased policy gradient.
- As the other MC-based methods: learns slowly due to high variance.
- Updates only available after full episodes.

Alternative: use an additional function approximator, the so-called critic, to estimate q_π (i.e., approximate policy gradient):

$$
\begin{aligned}
v(\boldsymbol{x}) &\approx \hat{v}(\boldsymbol{x}, \boldsymbol{w}_v), \\
q(\boldsymbol{x}, \boldsymbol{u}) &\approx \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_q), \\
a(\boldsymbol{x}, \boldsymbol{u}) &\approx \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_q) - \hat{v}(\boldsymbol{x}, \boldsymbol{w}_v).
\end{aligned}
$$

- Realization: any prediction tool discussed so far (TD(0), LSTD,...).
- Potential: we can use online step-by-step updates to estimate $\hat{q}$.
- Disadvantage: we would train two value estimates by $\boldsymbol{w}_v$ and $\boldsymbol{w}_q$.

Integrating the advantage function

- The TD error is

$$\delta_\pi = r + \gamma v_\pi(\boldsymbol{x}') - v_\pi(\boldsymbol{x}). \tag{4.15}$$

- In expectation the TD error is equivalent to the advantage function

$$\begin{aligned} \mathbb{E}_\pi \left[\delta_\pi | \boldsymbol{x}, \boldsymbol{u}\right] &= \mathbb{E}_\pi \left[r + \gamma v_\pi(\boldsymbol{x}') | \boldsymbol{x}, \boldsymbol{u}\right] - v_\pi(\boldsymbol{x}) \\ &= q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x}) \\ &= a_\pi(\boldsymbol{x}, \boldsymbol{u}). \end{aligned} \tag{4.16}$$

- Hence, the TD error can be used to calculate the policy gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\delta_\pi \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\right]. \tag{4.17}$$

- This results in requiring only one function parameter set:

$$\delta_\pi \approx r + \gamma \hat{v}_\pi(\boldsymbol{x}', \boldsymbol{w}) - \hat{v}_\pi(\boldsymbol{x}, \boldsymbol{w}). \tag{4.18}$$

Actor-critic structure

- Critic (policy evaluation) and actor (policy improvement) can be considered another form of generalized policy iteration (GPI).
- Online and on-policy algorithm for discrete and continuous action spaces with built-in exploration by stochastic policy functions.

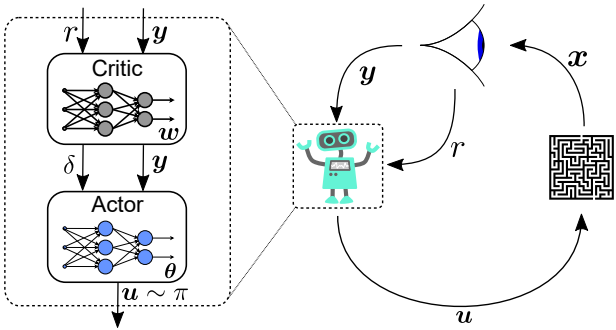


Fig. 4.9: Simplified flow diagram of actor-critic-based RL

Algo. implementation: actor-critic with TD(0) targets

- Analog to MC-based policy gradient optional discounting on the gradient updates is introduced.

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$ and state-value function $\hat{v}(\boldsymbol{x},\boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    apply $\boldsymbol{u}_k \sim \pi(\cdot|\boldsymbol{x}_k,\boldsymbol{\theta})$ and observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    $\delta \leftarrow r_{k+1} + \gamma\hat{v}(\boldsymbol{x}_{k+1},\boldsymbol{w}) - \hat{v}(\boldsymbol{x}_k,\boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w\delta\nabla_{\boldsymbol{w}}\hat{v}(\boldsymbol{x}_k,\boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta\gamma^k\delta\nabla_{\boldsymbol{\theta}}\ln\pi(\boldsymbol{u}_k|\boldsymbol{x}_k,\boldsymbol{\theta})$;

Algo. 4.3: Actor-critic for episodic tasks using TD(0) targets (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x},\boldsymbol{w}^*)$)

Actor-critic generalization

- Using the TD(0) error as the target to train the critic is convenient.
- However, the usual alternatives can be applied to train $\hat{v}(\boldsymbol{x}, \boldsymbol{w})$.
- n-step bootstrapping:

$$v(\boldsymbol{x}_k) \approx r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{v}_{k+n-1}(\boldsymbol{x}_{k+n}, \boldsymbol{w}).$$

- λ-return (forward view):

$$v(\boldsymbol{x}_k) \approx (1-\lambda) \sum_{n=1}^{T-k-1} \lambda^{(n-1)} g_{k:k+n} + \lambda^{T-k-1} g_k.$$

- TD(λ) using eligibility traces (backward view):

$$\begin{aligned} \boldsymbol{z}_k &= \gamma\lambda \boldsymbol{z}_{k-1} + \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{x}_k, \boldsymbol{w}_k), \\ \delta_k &= r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\boldsymbol{x}_k, \boldsymbol{w}_k). \end{aligned}$$

Algo. implementation: actor-critic with TD(λ) targets

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$
input: a differentiable state-value function $\hat{v}(\boldsymbol{x}, \boldsymbol{w})$
parameter: $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$, $\{\lambda_w, \lambda_\theta\} \in \{\mathbb{R}|0 \leq \lambda \leq 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$, $\boldsymbol{z}_w = 0$, $\boldsymbol{z}_\theta = 0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    apply $\boldsymbol{u}_k \sim \pi(\cdot|\boldsymbol{x}_k, \boldsymbol{\theta})$ and observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    $\delta \leftarrow r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}, \boldsymbol{w}) - \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{z}_w \leftarrow \gamma \lambda_w \boldsymbol{z}_w + \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{z}_\theta \leftarrow \gamma \lambda_d \boldsymbol{z}_\theta + \gamma^k \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \boldsymbol{z}_w$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \delta \boldsymbol{z}_\theta$;

Algo. 4.4: Actor-critic for episodic tasks using TD(λ) targets (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x}, \boldsymbol{w}^*)$)

Summary: what you've learned today

- Policy-based methods are a new class within the RL toolbox.
  - Instead of learning a policy indirectly from a value the policy is directly parametrized.
  - The policy function allows discrete and continuous actions with inherent stochastic exploration.
- Solving the underlying optimization task is complex. However, the policy gradient theorem provides a suitable theoretical baseline for gradient-based optimization.
- Anyhow, to calculate policy gradients we require a value estimate.
  - Monte Carlo prediction is straightforward, but comes with high variance and slow learning.
  - Adding a state-dependent baseline comparison does not change the policy gradient in expectation but enables decreasing the variance.
- Extending this idea naturally leads to integrating a critic network, i.e., an additional function approximation to estimate the value.
- The critic can be fed by the usual targets (TD(0), TD(λ),...).

Lecture 12: Deterministic Policy Gradient Methods

André
Bodmer

Table of contents

Background and motivation

Recap on policy gradient so far:

- The previously discussed policy functions and the policy gradient theorem were assuming stochastic polices.
- The resulting on-policy algorithms may not provide top-class learning performance:
  - Non-guided exploration with step-by-step updates and
  - Greedy actions only in the limit (i.e., infeasible long learning).

The alternative:

- Apply a deterministic policy with separate exploration.
- Enable off-policy learning (with experience replay as a possible extension).
- Hence, we will focus on a deterministic policy function

$$\pi(\boldsymbol{x}, \boldsymbol{\theta}) = \mu(\boldsymbol{x}, \boldsymbol{\theta}). \tag{5.1}$$

Deterministic policy gradient (DPG) theorem

Theorem 5.1: Deterministic Policy Gradient

Given a metric $J(\boldsymbol{\theta})$ for the undiscounted episodic (4.7) or continuing tasks (4.8) and a parameterizable policy $\mu(\boldsymbol{x}, \boldsymbol{\theta})$ the deterministic policy gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mu} \left[\nabla_{\boldsymbol{\theta}} \mu(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} q(\boldsymbol{x}, \boldsymbol{u})|_{\boldsymbol{u}=\mu(\boldsymbol{x})} \right]. \tag{5.2}$$

- Again, q needs to be approximated using samples, e.g., implementing a critic via TD learning.
- It turns out that (5.2) is also (approximately) valid in the off-policy case, i.e., if the sample distribution is obtained from a behavior policy.
- Proof can be found in D. Silver et al., *Deterministic Policy Gradient Algorithms*, International Conference on Machine Learning, 2014

Exploration with a deterministic policy

- If the DPG approach is applied on-policy there is no inherent exploration.
- How to learn something?
  - The environment itself is sufficiently noisy (random impacts, measurement noise).
  - Or we have to add noise to the actions, i.e., making the approach off-policy.
  - Hence, utilizing a behavior policy is also possible.
- That additional action noise could be:
  - Simple Gaussian noise or
  - a shaped noise process like a discrete-time Ornstein-Uhlenbeck (OU) process

$$\nu_{k+1} = \lambda \nu_k + \sigma \epsilon_k$$

where ν_k is the OU noise output, $0 < \lambda < 1$ is a smoothing factor and σ is the variance scaling a standard Gaussian sequence (no mean) ϵ_k.

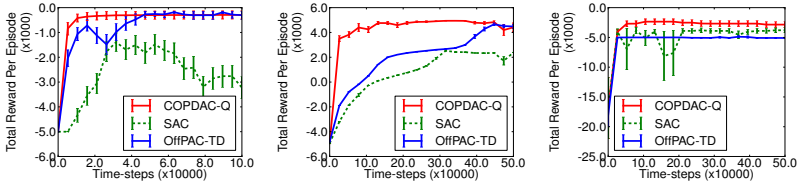
Algo. implementation: deterministic actor-critic

input: a differentiable deterministic policy function $\mu(\boldsymbol{x}, \boldsymbol{\theta})$
input: a differentiable action-value function $\hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{u}_k \leftarrow$ apply from $\mu(\boldsymbol{x}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
    observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    choose $\boldsymbol{u}'$ from $\mu(\boldsymbol{x}_{k+1}, \boldsymbol{\theta})$;
    $\delta \leftarrow r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, \boldsymbol{u}', \boldsymbol{w}) - \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \gamma^k \nabla_{\boldsymbol{\theta}} \mu(\boldsymbol{x}_k, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w})|_{\boldsymbol{u}=\mu(\boldsymbol{x})}$;

Algo. 5.1: Deterministic actor-critic for episodic tasks using SARSA(0) targets applicable on- and off-policy (output: parameter vector $\boldsymbol{\theta}^*$ for $\mu^*(\boldsymbol{x}, \boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{q}^*(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}^*)$)

Exemplary comparison to stochastic policy gradient

- DPG-based approach uses compatible function approximation, i.e., suitable linear $\hat{q}$ estimation. A fixed Gaussian behavior policy is applied for exploration.
- SAC uses a Gaussian policy with linear function approximation.



(a) Mountain Car (b) Pendulum (c) 2D Puddle World

Fig. 5.1: Comparison of stochastic on-policy actor-critic (SAC), stochastic off-policy actor-critic (OffPAC), and deterministic off-policy actor-critic (COPDAC) on continuous-action reinforcement learning (source: D. Silver et al., *Deterministic Policy Gradient Algorithms*, International Conference on Machine Learning, 2014)

Table of contents

Motivation / general idea

- The upcoming deep deterministic policy gradient (DDPG) algorithm was very much inspired by the successes of DQNs (cf. Algo. 3.6 and landmark paper by Mnih et al.) on discrete action spaces.
- However, DQNs are not directly applicable to (quasi-)continuous action spaces.
- Recall the incremental Q-learning equation using function approximation

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r + \gamma \max_{u} \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}) - \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}).$$

- For every policy inference and updating step we need to find $\max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w})$.
- If $u \in \mathcal{U} \subset \mathbb{Z}$ (i.e., using integer-encoded actions) is a sufficiently small discrete set, that is straightforward by an exhaustive search.
- In contrast, if $\boldsymbol{u} \in \mathcal{U} \subset \mathbb{R}^m$ is a (quasi-)continuous variable solving $\max_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}', \boldsymbol{u}, \boldsymbol{w})$ requires an own optimization routine which is computationally expensive if we use nonlinear function approximation.

The deterministic policy trick

- When using a greedy, deterministic policy $\boldsymbol{\pi}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta})$ we can utilize it to approximate

$$\max_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}', \boldsymbol{u}, \boldsymbol{w}) \approx \hat{q}(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}), \boldsymbol{w}). \tag{5.3}$$

- Hence, we can obtain explicit Q-learning targets for continuous actions when using a deterministic policy.
- For improving the policy we reuse the deterministic policy gradient theorem in an off-policy fashion

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_b \left[\nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}(\boldsymbol{X}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} q(\boldsymbol{X}, \boldsymbol{U}) \, | \boldsymbol{U} = \boldsymbol{\mu}(\boldsymbol{X}, \boldsymbol{\theta}) \right] \tag{5.4}$$

given a behavior policy $b(\boldsymbol{u}|\boldsymbol{x})$.

DDPG $\approx$ DQN + DPG

- Hence, we can consider the DDPG approach as a combination of DQN + DPG rendering it an actor-critic off-policy approach for continuous state and action spaces.
- Similarly to DQN we will introduce several 'tweaks' to stabilize and improve the DDPG learning process.

Tweak #1: experience replay buffer

- We store $\langle \boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}' \rangle$ in $\boldsymbol{\mathcal{D}}$ after each transition step.
- The replay buffer $\boldsymbol{\mathcal{D}}$ is of limited capacity, i.e., it discards the oldest data sample when updating once it is full (ring memory).
- This allows us to improve the Q-learning critic minimizing the mean-squared Bellman error (MSBE):

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma q(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}), \boldsymbol{w}) \right) - q(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}) \right]^2_{\boldsymbol{\mathcal{D}}}. \tag{5.5}$$

Additional DDPG tweaks (1)

Tweak #2: target networks

- Similar to DQN we introduce a (delayed) target network to estimate the Q-learning target

$$r + \gamma q(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}), \boldsymbol{w})$$

  since it depends on the same parameters $\boldsymbol{w}$ which we want to update.
- Hence, the target network's purpose it to mimic the generation of i.i.d. data as the ground truth to minimize (5.5).
- Since the policy parameters $\boldsymbol{\theta}$ are also part of the target calculation it turns out that an additional policy target network is also beneficial to stabilize the Q-learning.
- In contrast to the classical DQN implementation, the original DDPG algorithm does not perform periodically hard target network updates but continuous ones using a low-pass filter characteristic

$$\boldsymbol{w}^- \leftarrow (1-\tau)\boldsymbol{w}^- + \tau \boldsymbol{w}, \quad \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta} \tag{5.6}$$

  with τ representing the equivalent filter constant (hyperparameter).

Additional DDPG tweaks (2)

Tweak #3: mini-batch sampling

- Given a sufficiently filled memory $\boldsymbol{\mathcal{D}}$ and the target networks parametrized by $\boldsymbol{w}^-$ and $\boldsymbol{\theta}^-$ we draw uniformly distributed mini-batch samples $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$.
- The actual Q-learning is then based on the loss

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma q(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}^-), \boldsymbol{w}^-)\right) - q(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})\right]^2_{\boldsymbol{\mathcal{D}}_b} . \tag{5.7}$$

Tweak #4: batch normalization

- Minimizing (5.7) is a supervised learning step within the DDPG.
- The original DDPG paper by Lillicrap et al. back in 2015/16 suggested to use batch normalization, i.e., re-centering and re-scaling the inputs of each layer in an ANN.
- This idea of batch normalization was presented at that time shortly before by Ioffe and Szegedy (cf. original paper).
- Today's perspective: stick to the current state-of-the-art supervised ML algorithms for top-class Q-learning stability and speed (which are normally well-covered in popular supervised ML toolboxes).

Additional DDPG tweaks (3)

Tweak #5: exploration

- Since our policy is deterministic we require an exploratory behavior policy.
- Similar to DPG the standard approach is to add noise to the greedy actions, e.g., again from an Ornstein-Uhlenbeck (OU) process

$$\boldsymbol{u}_k \sim \boldsymbol{b}(\boldsymbol{u}|\boldsymbol{x}_k) = \boldsymbol{\mu}(\boldsymbol{x}_k, \boldsymbol{\theta}_k) + \boldsymbol{\nu}_k, \quad \boldsymbol{\nu}_k = \lambda \boldsymbol{\nu}_{k-1} + \sigma \boldsymbol{\epsilon}_{k-1}.$$

- One might also add a schedule for λ and σ along the training procedure, e.g., starting with significant noise levels (increased exploration) while reducing it over time (focusing exploitation)[1].
- However, many other behavior policies are possible, e.g., using model or expert-based guidance.

[1] Please note that this 'lambda' is not related to TD(λ), SARSA(λ), etc. Here, it is representing the stiffness of the OU noise process.

Visual summary of DDPG working principle

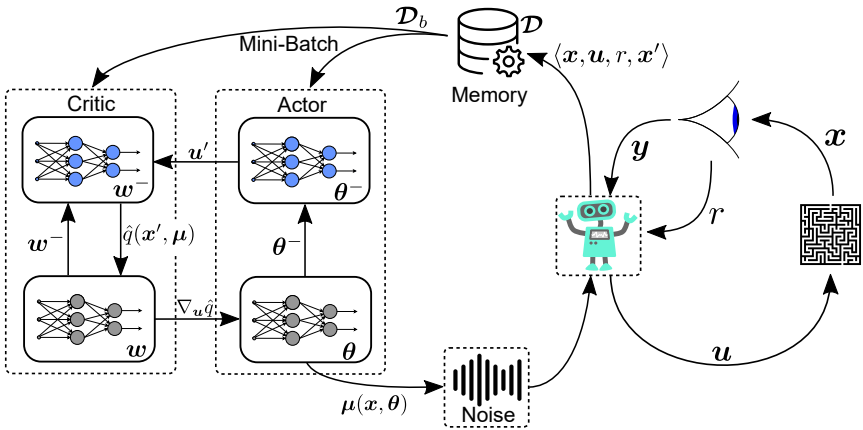


Fig. 5.2: DDPG structure from a bird's-eye perspective (derivative work of Fig. **??** and wikipedia.org, CC0 1.0)

Algo. implementation: DDPG

input: diff. deterministic policy function $\boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta})$ and action-value function $\hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})$
parameter: step sizes and filter constant $\{\alpha_w, \alpha_\theta, \tau\} \in \{\mathbb{R}|0 < \alpha, \tau < 1\}$
init: weights $\boldsymbol{w} = \boldsymbol{w}^- \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} = \boldsymbol{\theta}^- \in \mathbb{R}^d$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{u}_k \leftarrow$ apply from $\boldsymbol{\mu}(\boldsymbol{x}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
    observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    store tuple $\langle \boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
    sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
    for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
      if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
      else $y_i = r_{i+1} + \gamma \hat{q}(\boldsymbol{x}_{i+1}, \boldsymbol{\mu}(\boldsymbol{x}_{i+1}, \boldsymbol{\theta}^-), \boldsymbol{w}^-)$;
    fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y - \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$ with step size α_w;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta [\nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})|_{\boldsymbol{u}=\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x})}]_{\boldsymbol{\mathcal{D}}_b}$;
    Update target net. $\boldsymbol{w}^- \leftarrow (1-\tau)\boldsymbol{w}^- + \tau \boldsymbol{w}$, $\boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}$;

Algo. 5.2: Deep deterministic policy gradient (output: parameter vectors $\boldsymbol{\theta}^*$ for $\boldsymbol{\mu}^*(\boldsymbol{x}, \boldsymbol{\theta}^*)$)

Table of contents

Overestimation bias

- For Q-learning in the tabular case we have already discussed the maximization bias (cf. Fig. **??**) issue.
- Recap: Due to the greedy policy targets, $\hat{q}$ was overestimated when calculated using sampled values of stochastic MDPs.
- Additional problem when applying function approximation: the estimator itself introduces additional variance during the learning process which represents another source of the maximization bias problem.

This issue is already known in the DQN context (cf. Algo. 3.6). Similar to the tabular case, double DQN introduces a second Q-network counteracting the overestimation issue (cf. paper by van Hasselt et al.).

However, we did not address this possible problem in an actor-critic context using function approximation (e.g., DDPG).

Overestimation bias in actor-critic approaches (1)

- It turns out that the overestimation bias is also an issue for actor-critic methods[1].
- Consider an actor-critic policy with the current policy parameters $\boldsymbol{\theta}$.
- Let $\tilde{\boldsymbol{\theta}}$ define the parameters from the actor update induced by the maximization of the approximate critic $\hat{q}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{u})$.
- Let $\boldsymbol{\theta}^*$ be the parameters from the hypothetical actor update w.r.t. the true underlying value function $q^{\boldsymbol{\pi}}(\boldsymbol{x}, \boldsymbol{u})$.
- Then, we perform the policy update

$$\begin{aligned} \tilde{\boldsymbol{\theta}} &= \boldsymbol{\theta} + \frac{\alpha}{Z_1} \mathbb{E}_{\pi} \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\boldsymbol{X}) \nabla_{\boldsymbol{u}} \hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \boldsymbol{U}) \, | \boldsymbol{U} = \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{X}) \right], \\ \boldsymbol{\theta}^* &= \boldsymbol{\theta} + \frac{\alpha}{Z_2} \mathbb{E}_{\pi} \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\boldsymbol{X}) \nabla_{\boldsymbol{u}} q^{\boldsymbol{\pi}}(\boldsymbol{X}, \boldsymbol{U}) \, | \boldsymbol{U} = \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{X}) \right], \end{aligned} \tag{5.8}$$

where Z_1 and Z_2 normalize the gradient such that $Z^{-1} || \mathbb{E} [\cdot] || = 1$.

[1] Source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, https://arxiv.org/abs/1802.09477, 2018

Overestimation bias in actor-critic approaches (2)

- Lets denote $\tilde{\boldsymbol{\pi}}$ and $\boldsymbol{\pi}^*$ as the policies with updated parameters $\tilde{\boldsymbol{\theta}}$ and $\boldsymbol{\theta}^*$ respectively.
- As the gradient direction is a local maximizer, there exists ϵ_1 sufficiently small such that if $\alpha \leq \epsilon_1$ then the *approximate* value of $\tilde{\boldsymbol{\pi}}$ will be bounded below by the *approximate* value of $\boldsymbol{\pi}^*$:

$$\mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right] \geq \mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right]. \tag{5.9}$$

- Conversely, there exists ϵ_2 sufficiently small such that if $\alpha \leq \epsilon_2$ then the *true* value of $\tilde{\boldsymbol{\pi}}$ will be bounded above by the *true* value of $\boldsymbol{\pi}^*$:

$$\mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right] \geq \mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right]. \tag{5.10}$$

- In other words: if the approximate and true critics differ from each other, the according policy gradient updates cannot lead to better policy updates of the respective other framework.

Overestimation bias in actor-critic approaches (3)

- If the expected, estimated action value will be at least as large as the *true* action value w.r.t. $\boldsymbol{\theta}^*$

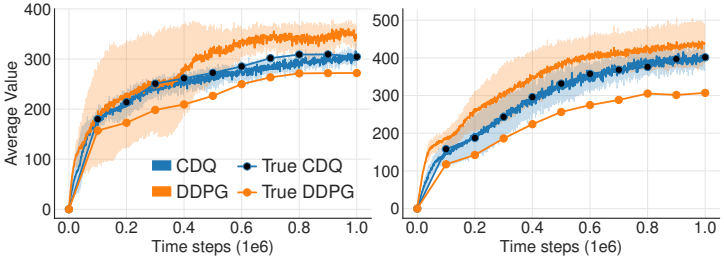
$$\mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right] \geq \mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right], \tag{5.11}$$

  then (5.9) and (5.10) imply

$$\mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right] \geq \mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right] \tag{5.12}$$

  with a sufficiently small $\alpha < \min\{\epsilon_1, \epsilon_2\}$.
- Hence, the maximization bias is also present in actor-critic updates.
- It can add up over several estimation updates and, therefore, may lead to suboptimal policy updates.
- A proof for unnormalized gradients can be also found in S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018.

Overestimation example for DDPG



(a) Hopper-v1 (b) Walker2d-v1

Fig. 5.3: Comparison of true and estimated values averaged over 10000 states in two robotic examples from OpenAI Gym. Estimated values originate from the approximate DDPG critic while the true values are based on the average discounted return over 1000 episodes following the current policy, starting from states sampled from the replay buffer (source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018.

Increased variance due to accumulating TD errors

- Using function approximation, the Bellman equation is never exactly satisfied leaving room for some amount of residual TD-error $\tilde{\delta}(\boldsymbol{x}, \boldsymbol{u})$:

$$\hat{q}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{u}) = r + \gamma \mathbb{E}_{\pi} \left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}', \boldsymbol{U}') | \boldsymbol{X}' = \boldsymbol{x}', \boldsymbol{U}' = \boldsymbol{u}' \right] - \tilde{\delta}(\boldsymbol{x}, \boldsymbol{u}). \tag{5.13}$$

- Although this error might be considered small per update step, it may accumulate over future steps if biased:

$$\hat{q}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{u}) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k \left(R_k - \tilde{\delta}_k(\boldsymbol{X}, \boldsymbol{U}) \right) \middle| \boldsymbol{X} = \boldsymbol{x}, \boldsymbol{U} = \boldsymbol{u} \right]. \tag{5.14}$$

- Observation: the variance of $\hat{q}$ will be proportional to the variance of future reward and residual TD-errors.
- If γ is large, the estimation variance might increase significantly.
- Mini-batch sampling will contribute to this variance issue.

TD3 extensions and modifications (1)

In order to reduce both the maximization bias and the learning variance, TD3 introduces mainly three measures on top of the DDPG algorithm. Hence, TD3 is a direct successor of DDPG.

Measure #1: clipped double Q-learning for actor-critic

- Following double Q-learning, a pair of critics $\{\hat{q}_{\boldsymbol{w}_1}, \hat{q}_{\boldsymbol{w}_2}\}$ is introduced.
- In contrast, the clipped target (with target networks $\{\boldsymbol{w}_1^-, \boldsymbol{w}_2^-\}$)

$$y = r + \gamma \min_{i=1,2} \hat{q}_{\boldsymbol{w}_i^-}(\boldsymbol{x}', \boldsymbol{u}') \tag{5.15}$$

  provides an upper-bound on the estimated action value.
- May introduce some underestimation, which is considered less critical than overestimation, since the value of underestimated actions will not be explicitly propagated through the policy update.
- The $\min$ operator will also (indirectly) favor actions leading to values with estimation errors of lower variance.

TD3 extensions and modifications (2)

Measure #2: target policy smoothing regularization

- Background: deterministic policies $\boldsymbol{\mu}$ tend to overfit to narrow peaks in the action-value estimate.
- Counteraction: fit the action value of a small area around the target action (i.e., smoothing $\hat{q}$ in the action space):

$$y = r + \gamma \hat{q}_{\boldsymbol{w}^-}(\boldsymbol{x}', \boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{x}') + \boldsymbol{\epsilon}). \tag{5.16}$$

- Here, $\boldsymbol{\epsilon} \sim \text{clip}\left(\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}), -\boldsymbol{c}, \boldsymbol{c}\right)$ is a mean-free, Gaussian noise with covariance $\boldsymbol{\Sigma}$, which is clipped at $\pm\boldsymbol{c}$ while $\boldsymbol{\theta}^-$ are the policy target network parameters.
- To satisfy possible action constraints (denoted by upper and lower box constraints $\{\underline{\boldsymbol{u}}, \overline{\boldsymbol{u}}\}$), we add an additional clipping:

$$\boldsymbol{u}' = \text{clip}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{x}') + \boldsymbol{\epsilon}, \underline{\boldsymbol{u}}, \overline{\boldsymbol{u}}\right). \tag{5.17}$$

- This modified action is then used for the target calculation (5.15).

TD3 extensions and modifications (3)

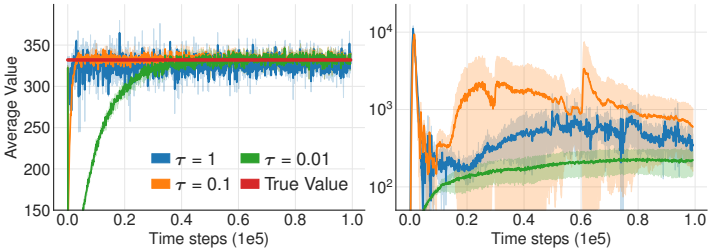
Measure #3: delayed policy updates

- Similar to DDPG, TD3 uses policy target networks $\boldsymbol{\theta}^-$ and (two) critic target networks $\{\boldsymbol{w}_1^-, \boldsymbol{w}_2^-\}$ in order to provide (rather) fixed Q-learning targets trying to stabilize the learning of $\hat{q}$.
- The target networks are also continuously updated using

$$\boldsymbol{w}_i^- \leftarrow (1-\tau)\boldsymbol{w}_i^- + \tau \boldsymbol{w}_i, \quad \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}.$$

- However, each policy update will inherently change the (true) Q-learning target directly adding variance to the learning process (cf. Fig. 5.4 on next slide).
- Therefore, it is argued that a policy update should not follow after each Q-learning update such that the critic can adapt properly to the previous policy update.
- The original TD3 implementation suggests a policy update every second Q-learning update, however, we can consider this update rate a hyperparameter.

TD3 extensions and modifications (4)



(a) Fixed Policy (b) Learned Policy

Fig. 5.4: Average estimated action value of a randomly selected state on Hopper-v1 environment from OpenAI Gym (source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018.

input: diff. deterministic policy function $\boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta})$ and action-value function $\hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})$
parameter: step sizes and filter constant $\{\alpha_w, \alpha_\theta, \tau\} \in \{\mathbb{R}|0 < \alpha, \tau < 1\}$, policy update rate $k_w \in \{\mathbb{N}|1 \leq k_w\}$, target noise $\boldsymbol{\Sigma} \in \mathbb{R}^{m \times m}$ and $\boldsymbol{c} \in \mathbb{R}^m$
init: weights $\{\boldsymbol{w}_1 = \boldsymbol{w}_1^-, \boldsymbol{w}_2 = \boldsymbol{w}_2^-\} \in \mathbb{R}^\zeta$, $\boldsymbol{\theta} = \boldsymbol{\theta}^- \in \mathbb{R}^d$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{u}_k \leftarrow$ apply from $\boldsymbol{\mu}(\boldsymbol{x}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
    observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    store tuple $\langle \boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
    sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
    for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
      if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
      else
        $\boldsymbol{u}' = \text{clip}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{x}_{i+1}) + \text{clip}\left(\mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma}), -\boldsymbol{c}, \boldsymbol{c}\right), \underline{\boldsymbol{u}}, \overline{\boldsymbol{u}}\right)$;
        $y_i = r_{i+1} + \gamma \min_{l=1,2} \hat{q}(\boldsymbol{x}_{i+1}, \boldsymbol{u}', \boldsymbol{w}_l^-)$;
    fit $\boldsymbol{w}_l$ on loss $\mathcal{L}(\boldsymbol{w}_l) = [y - \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_l)]^2_{\boldsymbol{\mathcal{D}}_b}$ with step size $\alpha_w \; \forall \, l$;
    if $k \mod k_w = 0$ **then**
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta [\nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_1)|_{\boldsymbol{u}=\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x})}]_{\boldsymbol{\mathcal{D}}_b}$;
      $\boldsymbol{w}_l^- \leftarrow (1-\tau)\boldsymbol{w}_l^- + \tau \boldsymbol{w}_l, \; \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}$;

Algo. 5.3: Twin delayed deep deterministic policy gradient (TD3)

Summary: what you've learned today

- The deep deterministic policy gradient (DDPG) approach 'transfers' many deep Q-network (DQN) ideas to continuous action spaces.
- It mainly combines DQN + deterministic policy gradients + policy and value target networks (plus additional minor tweaks).
- However, the DDPG actor-critic suffers from value overestimation and high variance during learning. Hence, sampled policy gradients might not be optimal (pointing towards overrated action values).
- Twin delayed DDPG (TD3) adds clipped double Q-learning, delayed policy updates and target policy smoothing to counteract these issues.

Lecture 13: Further Contemporary RL Algorithms

André
Bodmer

Table of contents

Reinterpreting the stochastic policy gradient (1)

- In the following we will focus on stochastic policies $\pi(\boldsymbol{u}|\boldsymbol{x})$.
- First, we rewrite the performance metric (4.7) to obtain

$$J_\pi = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_k \right]. \tag{6.1}$$

- Using the advantage $a_\pi(\boldsymbol{x}, \boldsymbol{u}) = q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x})$ we can calculate the performance of an updated policy $\boldsymbol{\pi} \rightarrow \tilde{\boldsymbol{\pi}}$[1]:

$$J_{\tilde{\pi}} = J_\pi + \int_{\mathcal{X}} p^{\tilde{\pi}}(\boldsymbol{x}) \int_{\mathcal{U}} \tilde{\pi}(\boldsymbol{u}|\boldsymbol{x}) a_\pi(\boldsymbol{x}, \boldsymbol{u}). \tag{6.2}$$

- While for finite MDPs, the policy improvement theorem guaranteed $J_{\tilde{\pi}} \geq J_\pi$ for each policy update, there might be some states where $\int_{\mathcal{U}} \tilde{\pi}(\boldsymbol{u}|\boldsymbol{x}) a_\pi < 0$ for continuous MDPs using function approximation.

[1] proof from: S. Kakade and J. Langford, *Approximately optimal approximate reinforcement learning*, ICML, vol. 2, pp 267-274, 2002

Reinterpreting the stochastic policy gradient (2)

- For easier calculation, we introduce a local approximation to (6.2)

$$\mathcal{L}_\pi(\tilde{\pi}) = J_\pi + \int_{\mathcal{X}} p^\pi(\boldsymbol{x}) \int_{\mathcal{U}} \tilde{\pi}(\boldsymbol{u}|\boldsymbol{x}) a_\pi(\boldsymbol{x}, \boldsymbol{u}) \tag{6.3}$$

  where $p^\pi(\boldsymbol{x})$ is used instead of $p^{\tilde{\pi}}(\boldsymbol{x})$, i.e., neglecting the state distribution change due to a policy update.
- For any parametrized and differentiable policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})$, it can be shown that

$$\begin{aligned} \mathcal{L}(\pi_{\boldsymbol{\theta}_0}) &= J(\pi_{\boldsymbol{\theta}_0}), \\ \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\pi_{\boldsymbol{\theta}_0}}(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} &= \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \end{aligned} \tag{6.4}$$

  for any initial parameter set $\boldsymbol{\theta}_0$.
- For a sufficiently small step size, improving $\mathcal{L}_{\pi_{\boldsymbol{\theta}_0}}$ will also improve J.

However, we do not know how much the actual stochastic policy will change while moving through the parameter space. Hence, we do not have a good decision basis to choose the policy gradient step size.

Adding a trust region constraint (1)

- From the previous discussion it can be concluded that we want a metric describing how much a policy is changed in the action space when updating the policy in the parameter space.
- Against this background, we make use of the Kullback-Leibler divergence (also called relative entropy)

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) \mathrm{d}x \tag{6.5}$$

  defined for continuous distributions P and Q with their probability densities p and q.
- Example: for two multivariate Gaussian distributions of equal dimensions d, with means $\boldsymbol{\mu}_0, \boldsymbol{\mu}_1$ and with (non-singular) covariance matrix $\boldsymbol{\Sigma}_0, \boldsymbol{\Sigma}_1$ we receive

$$D_{\mathrm{KL}}\left(\mathcal{N}_0 \parallel \mathcal{N}_1\right) = \frac{1}{2} \Big(\mathrm{tr}\left(\boldsymbol{\Sigma}_1^{-1} \boldsymbol{\Sigma}_0\right) + \left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0\right)^{\mathsf{T}} \boldsymbol{\Sigma}_1^{-1} \left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0\right) - d + \ln \left(\frac{\det \boldsymbol{\Sigma}_1}{\det \boldsymbol{\Sigma}_0} \right) \Big).$$

Adding a trust region constraint (2)

- The trust region policy optimization (TRPO) updates the policy parameters while constraining the KL divergence between the new and the old policy distribution:

$$\begin{aligned} &\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}), \\ \text{s.t.} \quad &\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) \leq \kappa \end{aligned} \tag{6.6}$$

with

$$\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) = \overline{D}_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}, \pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X})) \right].$$

- Hence, we want to limit the average KL divergence w.r.t. the states visited by the old policy.
- The constraint κ is a TRPO hyperparameter (typically $\kappa << 1$).
- Although (6.6) does not provide any formal convergence guarantee, we at least have a link between changes in the parameter and policy distribution space. Therefore, we can use this tool to prevent erratic policy changes.

Smooth policy updates via TRPO

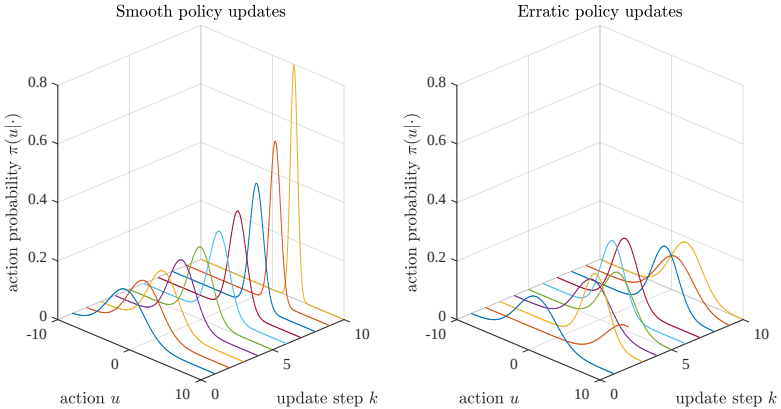


Fig. 6.1: Simplified representation of the policy evolution for a scalar action given some fixed state. Left: TRPO-style updates finding the optimal action with increasing probability. Right: Unmonitored policy distributions not converging towards an optimal policy ('policy chattering').

Sample-based objective and constraint estimation (1)

- To actually solve (6.6) we will make use of samplings from Monte Carlo rollouts.
- Expanding the objective yields

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} J_{\pi_k} + \int_{\mathcal{X}} p^{\pi_k}(\boldsymbol{x}) \int_{\mathcal{U}} \pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}). \tag{6.7}$$

- The first term J_{π_k} can be dropped, since it is irrelevant for the optimization result (constant).
- Using samples we can approximate $\int_{\mathcal{X}} p^{\pi_k}(\boldsymbol{x}) \approx \frac{1}{1-\gamma} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}[\boldsymbol{X}]$.
- Moreover, $\int_{\mathcal{U}} \pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}) \approx \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U}) \right]$ is also approximated applying importance sampling based on data from the old policy.
- Hence, the sampled objective is

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U}) \right]. \tag{6.8}$$

Sample-based objective and constraint estimation (2)

- Applying the previous sample-based estimation we obtain

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U}) \right], \\ \text{s.t.} \quad & \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X})) \right] \leq \kappa. \end{aligned} \tag{6.9}$$

- Hence, we have a three-step procedure for each TRPO update:

1. Use Monte Carlo simulations based on the old policy to obtain data.
2. Use the data to construct (6.9).
3. Solve the constrained optimization problem to update the policy parameter vector.

Solving (6.9) is generally a nonlinear optimization problem. The original TRPO implementation uses a local objective and constraint approximation together with conjugate gradient and line search algorithms. However, many other constrained-nonlinear solvers are also applicable.

Generalized advantage estimation

- Having data $\langle \boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}' \rangle$ in $\mathcal{D}$ from a Monte Carlo rollout available, an imporant problem is to estimate $a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$ in (6.9).
- A particular suggestion in the TRPO context is to use a generalized advantage estimator (GAE) [1] defined as

$$\hat{a}_k^{(\gamma,\lambda)} = \sum_{i=0}^{\infty} (\gamma\lambda)^i \delta_{k+i}. \tag{6.10}$$

- Here, $\delta_k = r_k + \gamma v(\boldsymbol{x}_{k+1}) - v(\boldsymbol{x}_k)$ is a single advantage sample.
- Hence, the GAE is the exponentially-weighted average of the discounted advantage samples with an additional weighting λ.
- Similar formulation compared to TD(λ) but the estimator's target is the advantage.
- The choice of $(\gamma\lambda)$ trade-offs the bias and variance of the estimator.

[1] cf. J. Schulmann et al., *High Dimensional Continuous Control Using Generalized Advantage Estimation*, https://arxiv.org/abs/1506.02438, 2015

TRPO summary

The TRPO's key facts are:

- The TRPO constrains policy distribution changes when updating the policy parameters (for stochastic policies and on-policy learning).
- The objective is to enable a monotonically improving learning process.
- Using trust regions, erratic policy updates should be prevented.

The TRPO's main hurdles are:

- Constructing the objective function and constraint requires Monte Carlo rollouts (time consuming, data inefficient).
- When the sampled optimization problem is set up, a nonlinear and constrained optimization step is required (no simple policy gradient, computational costly).

We will not provide any specific TRPO implementation suggestion at this point, since this is rather cumbersome. Instead we will move forward to a similar algorithm which is pursuing the same goal (prevent erratic policy changes) with a much simpler implementation.

Table of contents

Background and motivation

- The upcoming proximal policy optimization (PPO) algorithm tries to mimic the constrained TRPO problem

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U}) \right], \\ \text{s.t.} \quad & \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X})) \right] \leq \kappa. \end{aligned}$$

based on related unconstrained problems.

- Hence, the objective will be reformulated to incorporate mechanisms preventing excessively large variations of the policy distribution during a parameter update (leading to an updated policy with sufficient proximity to the old one).
- Moreover, PPO incorporates two variants which we will discuss:
  1. Clipping the surrogate objective,
  2. Adaptive tuning of a KL-associated penalty coefficient.

Clipped surrogate objective

- The first approach is based on the following clipped objective:

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}\left[\min\left\{\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})}a_{\pi_k}(\boldsymbol{X},\boldsymbol{U}), \text{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})}, 1-\epsilon, 1+\epsilon\right)a_{\pi_k}(\boldsymbol{X},\boldsymbol{U})\right\}\right]. \tag{6.11}$$

- Above, $\epsilon < 1$ is a PPO hyperparameter serving as a regularizer.
- The first element of $\min\{\cdot\}$ is the previous TPRO objective.
- The second element of $\min\{\cdot\}$ modifies the surrogate objective by clipping the importance sampling ratio $\pi_{\boldsymbol{\theta}}/\pi_{\boldsymbol{\theta}_k}$.
- The latter should remove the incentive for moving the importance sampling ratio outside of the interval $[1-\epsilon, 1+\epsilon]$.
- The modified objective is therefore a lower bound of the unclipped TRPO objective.

Clipped surrogate objective: positive advantage

- Consider a single sample $(\boldsymbol{x}, \boldsymbol{u})$ with a positive advantage $a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$:

$$\max_{\boldsymbol{\theta}} \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}), \operatorname{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1-\epsilon, 1+\epsilon \right) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}) \right\}.$$

- Because the advantage is positive, the objective will increase if the action becomes more likely, i.e., if $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})$ increases.
- If $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) > (1+\epsilon)\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})$ the clipping becomes active.
- Hence, the objective reduces to

$$\max_{\boldsymbol{\theta}} \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1+\epsilon \right\} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}).$$

- Due to the $\min\{\cdot\}$ operator, the entire objective is therefore limited to $(1+\epsilon)a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$.
- Interpretation: the new policy does not benefit from going very away from the old policy distribution.

Clipped surrogate objective: negative advantage

- Consider a single sample $(\boldsymbol{x}, \boldsymbol{u})$ with a negative advantage $a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$:

$$\max_{\boldsymbol{\theta}} \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}), \operatorname{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1-\epsilon, 1+\epsilon \right) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}) \right\}.$$

- Because the advantage is negative, the objective will increase if the action becomes less likely, i.e., if $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})$ decreases.
- If $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) < (1-\epsilon)\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})$ the clipping becomes active.
- Hence, the objective reduces to

$$\max_{\boldsymbol{\theta}} \max \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1-\epsilon \right\} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}).$$

- Due to the $\max\{\cdot\}$ operator, the entire objective is limited to $(1-\epsilon)a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$.

Adaptive KL penalty

- The second PPO variant makes use of the following KL-penalized objective

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X},\boldsymbol{U}) - \beta D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X}))\right]. \tag{6.12}$$

- Transfers the KL-based constraint into a penalty for large policy distribution changes.
- The parameter β weights the penalty against the policy improvement.
- The original PPO implementation suggests an adaptive tuning of β w.r.t. the sampled average KL divergence $\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta})$ estimated from previous experience

$$\begin{aligned} \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) < \overline{D}^*_{\mathrm{KL}}: &\quad \beta \leftarrow \beta/2, \\ \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) > \overline{D}^*_{\mathrm{KL}}: &\quad \beta \leftarrow \beta \cdot 2. \end{aligned} \tag{6.13}$$

with some target value of the KL divergence $\overline{D}^*_{\mathrm{KL}}$ (additional hyperparameter).

Algo. implementation: PPO

input: diff. stochastic policy fct. $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$ and value fct. $\hat{v}(\boldsymbol{x},\boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$
for $j = 1, 2, \ldots,$ *(sub-)episodes* **do**
  initialize $\boldsymbol{x}_0$ (if new episode);
  collect a set of tuples $\langle \boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ by a rollout using $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}_j)$;
  store them in $\boldsymbol{\mathcal{D}}$;
  estimate the advantage $\hat{a}_{\pi_j}(\boldsymbol{x},\boldsymbol{u})$ based on $\hat{v}(\boldsymbol{x},\boldsymbol{w}_j)$ and $\boldsymbol{\mathcal{D}}$ (e.g., GAE);
  $\boldsymbol{\theta}_{j+1} \leftarrow$ policy gradient update based on the PPO variant (6.11) or (6.12);
  $\boldsymbol{w}_{j+1} \leftarrow$ minimizing the mean-squared TD errors using $\boldsymbol{\mathcal{D}}$ (critic);
  delete entries in $\boldsymbol{\mathcal{D}}$ (due to on-policy learning);

Algo. 6.1: Proximal policy optimization (output: parameter vectors $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x},\boldsymbol{w}^*)$)

Some PPO remarks

- Clipping the surrogate objective (6.11) was reported to achieve higher performances than the KL penalty (6.12).[1]
- Like TRPO, PPO is an on-policy algorithm. Hence, the memory $\mathcal{D}$ is not a rolling replay buffer (cf. off-policy algorithms like DQN, DDPG or TD3) but a rollout buffer using one fixed policy.
- These rollouts are likely to result in an increased sample demand either using a simulator or a real experiment.

Although PPO is derived from a TRPO background pursuing monotonically increasing policy performance, its realization is based on multiple heuristics and approximations. Hence, there is no guarantee on achieving this goal and the specific performance of the PPO algorithm must be evaluated empirically given a certain application.

[1] cf. original PPO paper results by J. Schulman et al., *Proximal Policy Optimization Algorithms*, https://arxiv.org/abs/1707.06347, 2017

Exemplary performance comparison

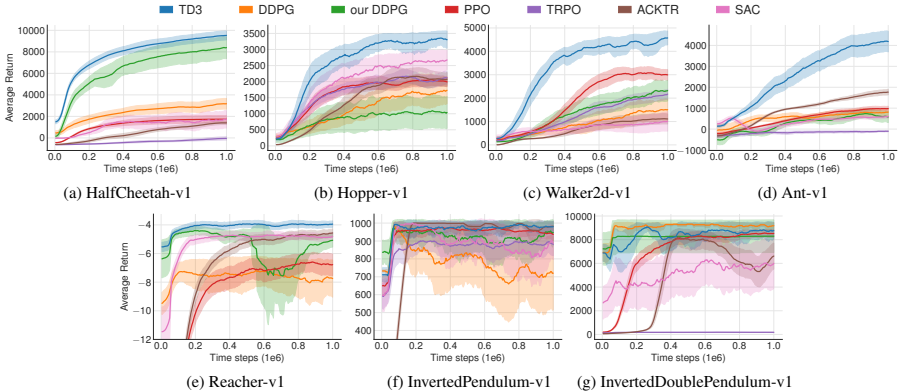


Fig. 6.2: Learning curves for OpenAI Gym continuous control tasks. The shaded region represents half a standard deviation of the average evaluation over ten trials (source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018).

Algorithmic outlook: other contemporary model-free algorithms (1)

The selection of algorithms appears endless:

- DQN variants such as
  - (Prioritized) dueling DQN
  - Noisy DQN
  - Distributional DQN
- Rainbow (combining multiple DQN extensions)
- Soft actor-critic (SAC)
- Actor critic using Kronecker-factored trust region (ACKTR)
- Asynchronous advantage actor-critic (A3C)
- ....

Remarks:

- You have already learned the basic building blocks in order to make yourself familiar with any value-/policy-based model-free RL approach.
- Use this knowledge!
- Focus on primary scientific literature for self-studying and not on unreliable sources!

Algorithmic outlook: other contemporary model-free algorithms (2)

Algorithm collections with tutorial-style documentation:

- Intel Reinforcement Learning Coach
- OpenAI Spinning Up

Algorithm collections with decent application-oriented documentation:

- RLlib (Ray)
- Stable Baselines3
- Acme
- Garage
- Google Dopamine
- Tensorforce
- TF-Agents
- ...

Summary: what you've learned today

- Trust region policy optimization (TRPO) pursues monotonically increasing policy performance by limiting policy distribution changes.
- This results in a nonlinear constrained optimization problem adding computational complexity (no simple policy gradients).
- Proximal policy optimization (PPO) converts the TRPO idea into an unconstrained optimization problem by a modified objective. Likewise, the PPO's objective is to prevent erratic policy distribution changes.

Lecture 14: Outlook and Research Insights

André
Bodmer

Table of contents

Recap: optimal control and constraints

Real-world systems are always subject to certain state constraints $\mathcal{X}$ and input limitations $\mathcal{U}$. Violating those can lead to safety issues.

$$v_k^* = \max_{\boldsymbol{u}_k} \sum_{i=0}^{N_\mathrm{p}} \gamma^i r_{k+i+1}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}) \,, \tag{7.1}$$

$$\text{s.t.} \qquad \boldsymbol{x}_{k+i+1} = \boldsymbol{f}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}), \quad \boldsymbol{x}_{k+i} \in \mathcal{X}, \quad \boldsymbol{u}_{k+i} \in \mathcal{U} \,.$$

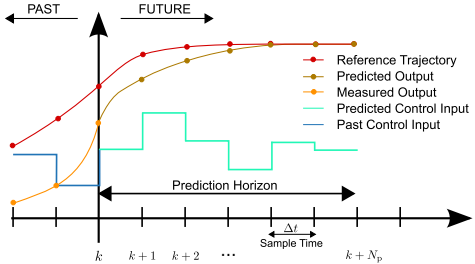


Fig. 7.1: MPC scheme (source: www.wikipedia.org, by Martin Behrendt CC BY-SA 3.0)

Application examples with safety-relevant constraints

Collaborative robot control (source: www.wikipedia.org, CC BY-SA 4.0)

Autonomous car driving (source: www.wikipedia.org, CC BY-SA 4.0)

Energy system control

Medication control (source: www.wikipedia.org, CC BY-SA 4.0)

Safety levels

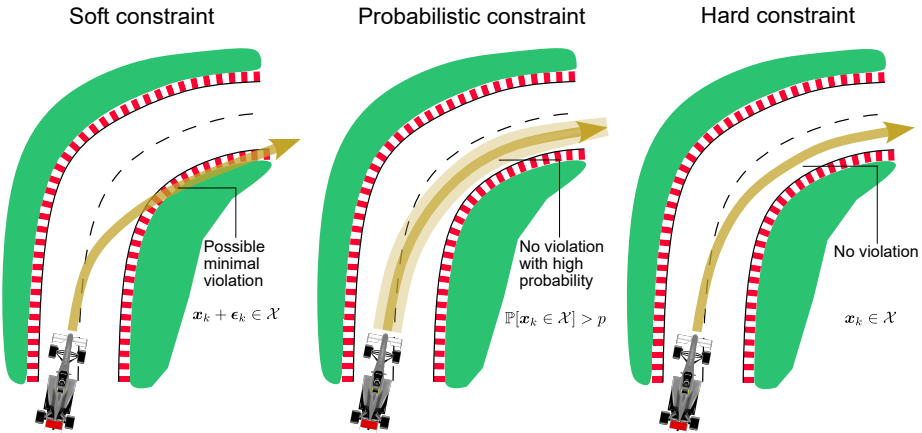
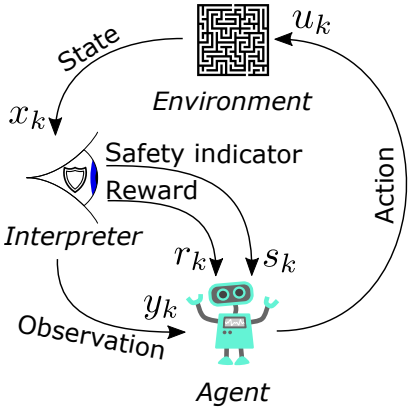
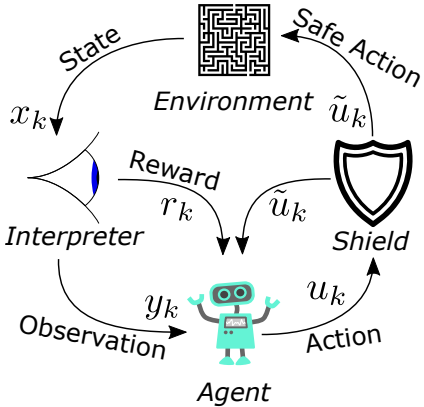


Fig. 7.2: Different levels of safety (derived from L. Brunke et al., *Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning*, Annual Review of Control, Robotics, and Autonomous Systems, 2022)

Bird's eye view on RL concepts integrating safety



(a) Safety critic: add a critic which indicates to which extent the current data sample fits to a safe situation



(b) Safety shield: use a priori or learned model knowledge of the environment to make predictions identifying actions leading to unsafe situations

Achievable safety levels and model knowledge

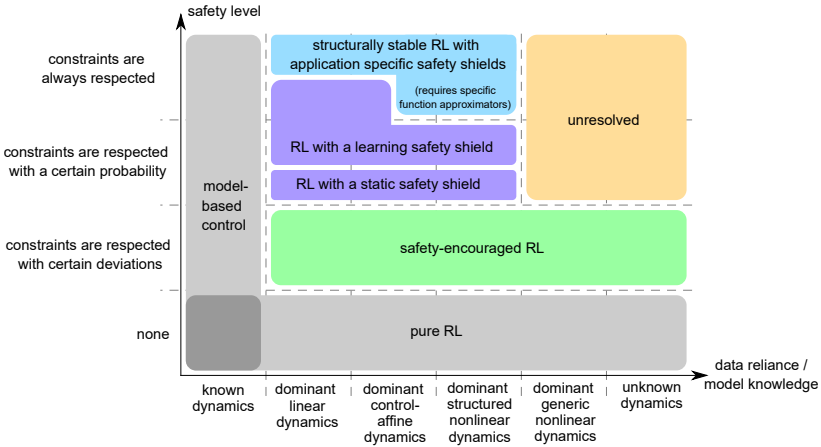
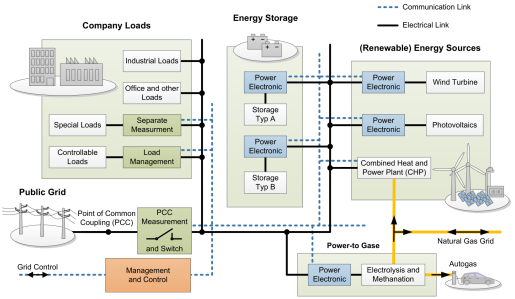
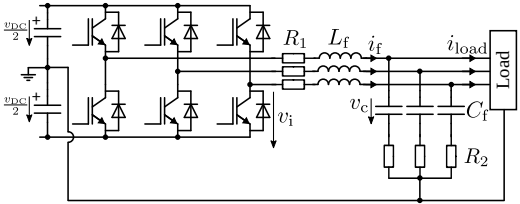


Fig. 7.3: Safety and model knowledge map (derived from L. Brunke et al., *Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning*, Annual Review of Control, Robotics, and Autonomous Systems, 2022)

Energy system control application



(a) Example microgrid that can be emulated in the LEA Microgrid Laboratory.



(b) Application under investigation: Three-phase grid-forming inverter disturbed by stochastic load

Reference tracking with disturbance rejection

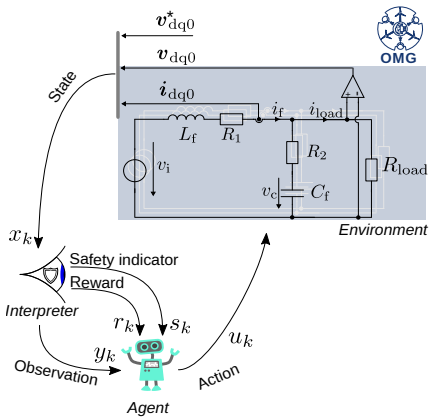


Fig. 7.4: Simulation setting with environment modeled using OpenModelica Microgrid Gym

- Cont. state- and actionspace
- Deep deterministic policy gradient agent
- Gird-forming inverter
- Stochastic load acts as disturbance
- State per phase: $\boldsymbol{x_k} = [i_{\mathrm{f}}, v_{\mathrm{C}}]$, $v_{\mathrm{i}} = v_{\mathrm{DC}} \cdot u_k$
- $r_k = f(v_{\mathrm{C}}, v^*, i_{\mathrm{f}}) \in [1, -0.75]$
- $s_k = -1$, if limit (i_{f} or v_{C}) is exceeded

Reward design for grid-forming inverter

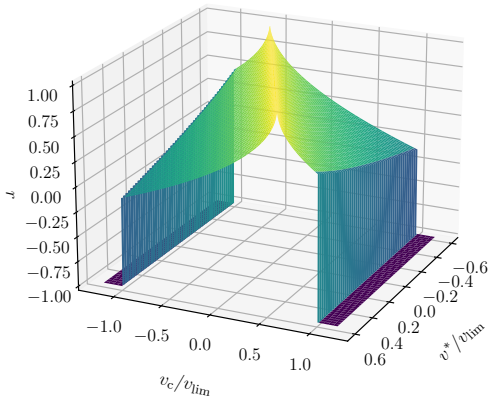


Fig. 7.5: Reward function 7.2 for different reference and measured voltages and currents below nominal current

- Three cases, depending on operation point

$$r = \begin{cases} \mathrm{MRE}(v_\mathrm{C}, v^*), & \text{Ⓐ} \\ \mathrm{MRE}(v_\mathrm{C}, v^*) + f(i_\mathrm{f}), & \text{Ⓑ} \\ -1, & \text{Ⓒ} \end{cases} \tag{7.2}$$

- Ⓐ $v_\mathrm{C} \leq v_\mathrm{lim} \wedge i_\mathrm{f} \leq i_\mathrm{nom}$
- Ⓑ $v_\mathrm{C} \leq v_\mathrm{lim} \wedge i_\mathrm{nom} \leq i_\mathrm{f} \leq i_\mathrm{lim}$
- Ⓒ otherwise
- Linear punishment term $f(i_\mathrm{f})$

Reference tracking with disturbance rejection using saftey shield

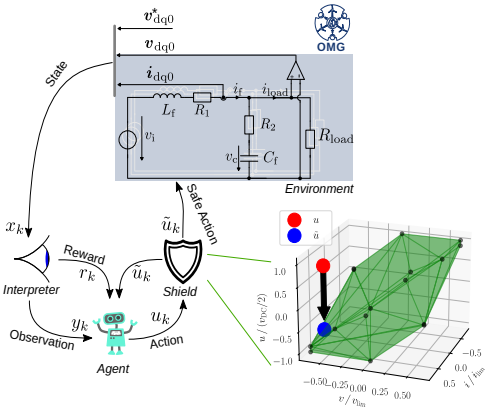


Fig. 7.6: Safety shield based on feasible set

- Safety shield: Ensure that action does not cause state limit violation in future system trajectories
- Such a state action pair is called feasible
- Calculation of feasible set requires a model
- Training data can be utilized to identify model
- Here, recursive least squares (RLS) is applied

Saftey shield based on feasible set - proof of concept (1)

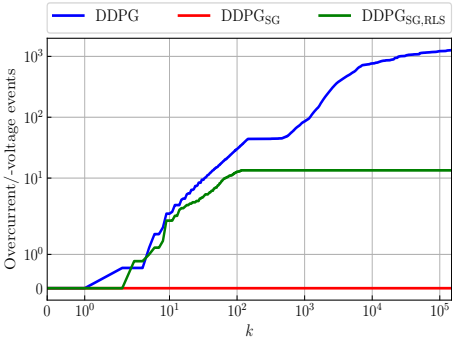


Fig. 7.7: Accumulated unsafe events (overcurrent/-voltage) per trainingstep k

- Three different approaches
- DDPG: Agent without safety shield
- DDPG$_{\mathrm{SG}}$: Agent with safety shield using perfect a priori knowledge
- DDPG$_{\mathrm{SG,RLS}}$: Agent with safety shield without a priori knowledge, identifying model using RLS
- Five agents trained per approach
- Results in D. Weber et al., *Safe Reinforcement Learning-Based Control in Power Electronic Systems*, 2023

Saftey shield based on feasible set - proof of concept (2)

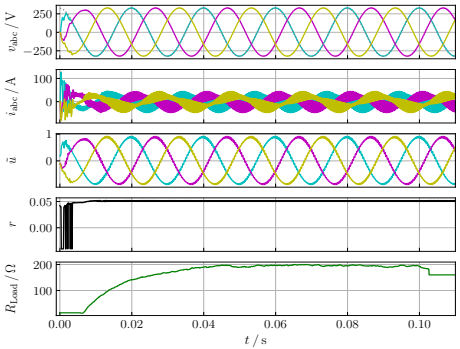


Fig. 7.8: Blackstart after training using $\mathrm{DDPG}_{\mathrm{SG,RLS}}$

- $\mathrm{DDPG}_{\mathrm{SG,RLS}}$ agent trained for 150000 steps
- R_{Load} changes every step based on random process
- Additional events – load steps and drifts – trigged randomly

Table of contents

Real-time implementation aspects (1)

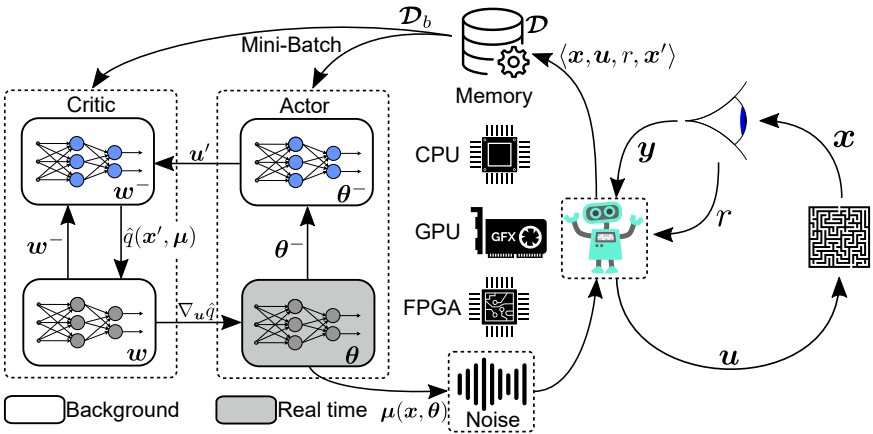
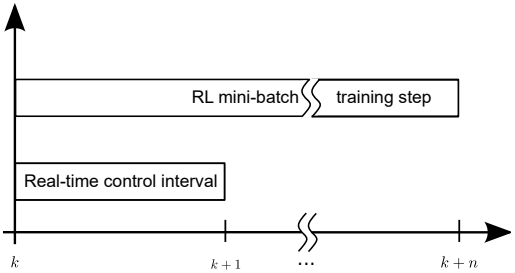
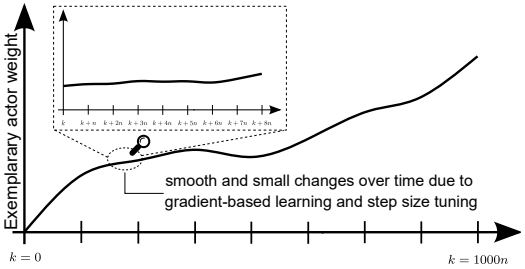


Fig. 7.9: DDPG implementation example (derivative work of Fig. **??** and wikipedia.org, CC0 1.0)

Real-time implementation aspects (2)



(a) Real-time control requirement vs. learning time



(b) Typical evolution of RL parameter weights during learning

Application example: deep Q direct torque control

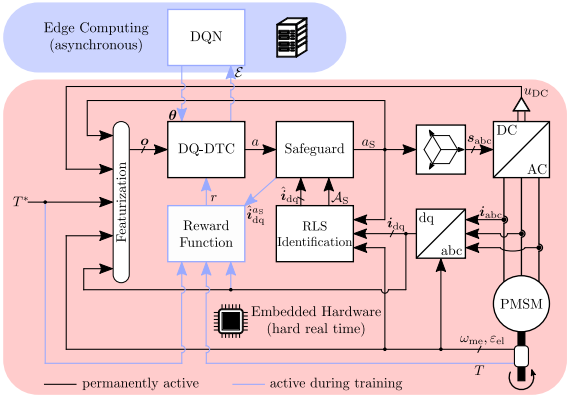


Fig. 7.10: Deep Q direct torque control schematic

- The DQ-DTC is basically a DQN
- Sampling time of the plant system is $T_{\mathrm{s}} = 50\,\mu\mathrm{s}$
- DQN inference, safeguarding and system identification must fit into T_{s}
- Source: M. Schenke et al., *Finite-Set Direct Torque Control via Edge Computing-Assisted Safe Reinforcement Learning for a Permanent Magnet Synchronous Motor*, 2023

Fast neural network inference

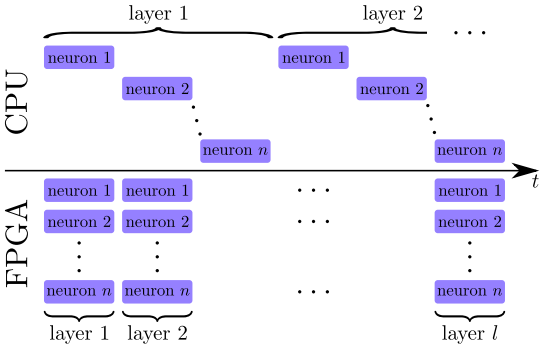


Fig. 7.11: Conceptual comparison of CPU and FPGA evaluation of a neural network

- Each neuron has the same job $y_{n,l+1} = f(\boldsymbol{y}_l^\top \boldsymbol{w}_{n,l} + b_{n,l})$
- CPU must evaluate each neuron sequentially
- FPGA can evaluate each neuron at the same time
- Maximum number of parallel computations is limited

Edge reinforcement learning

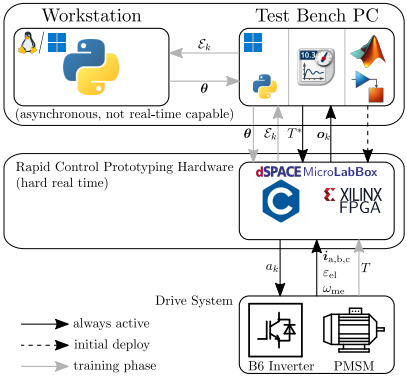


Fig. 7.12: Our edge reinforcement learning pipeline

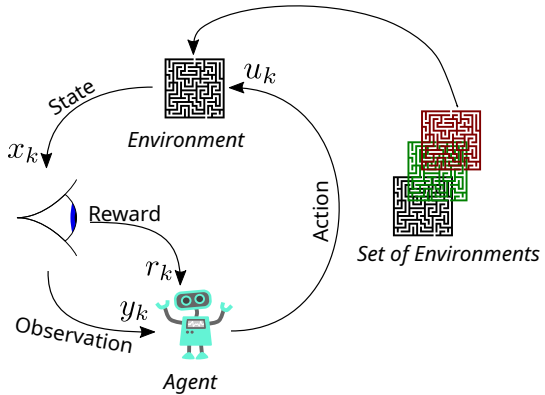
- Backward pass / learning steps are outsource to workstation
- Communication between test bench and workstation is based on TCP/IP
- Backward pass is generic and has no time constraints $\rightarrow$ low application effort

Demonstration video

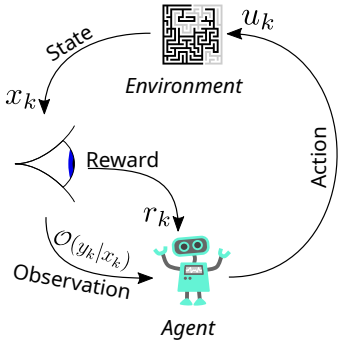
Youtube link: Coffee machine vs. deep Q direct torque control

Table of contents

Meta reinforcement learning - the setting (1)



(a) General problem class is similar, environments only differ in some characteristics, the agent could transfer learned behavior



(b) Solution approach: treat the environment as partially observable, distinguishing details are not directly available

Meta reinforcement learning - the setting (2)

- The agent must have some mechanism that allows adaptation to the specific environment
- This means, the distinguishing details must be extracted in some way
- Usually, they can be retrieved from a larger set of observations

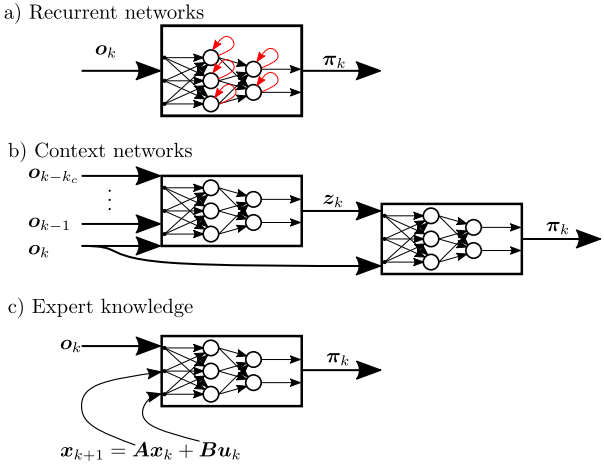


Fig. 7.13: Different concepts of meta learning

Usage in electric drive control: classical agent

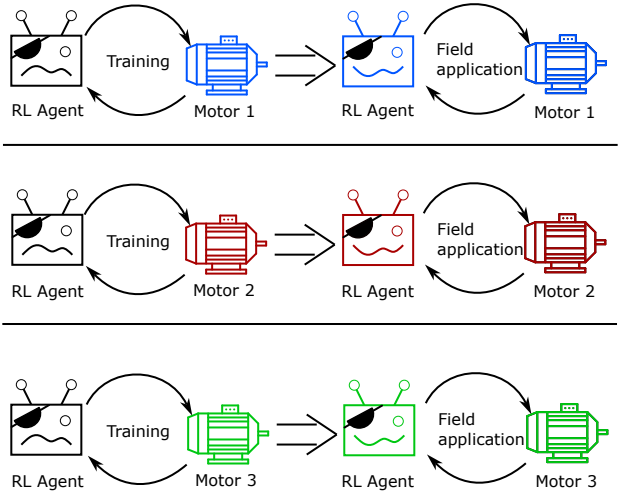


Fig. 7.14: Each agent must be trained individually → huge effort

Usage in electric drive control: meta agent

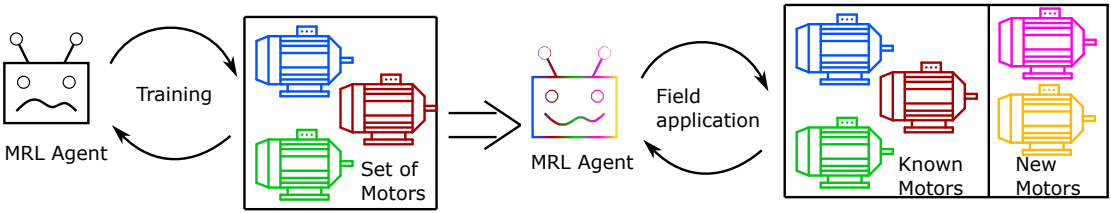


Fig. 7.15: One agent to control them all → effort is limited and independent of the number of controlled environments

Our setup

- Make use of context network
- Generate context $\boldsymbol{z}$ with a fix set of observations $\rightarrow \boldsymbol{z} = \text{const.}$
- Source: D. Jakobeit et al., *Meta-Reinforcement Learning-Based Current Control of Permanent Magnet Synchronous Motor Drives for a Wide Range of Power Classes*, IEEE TPEL, 2023

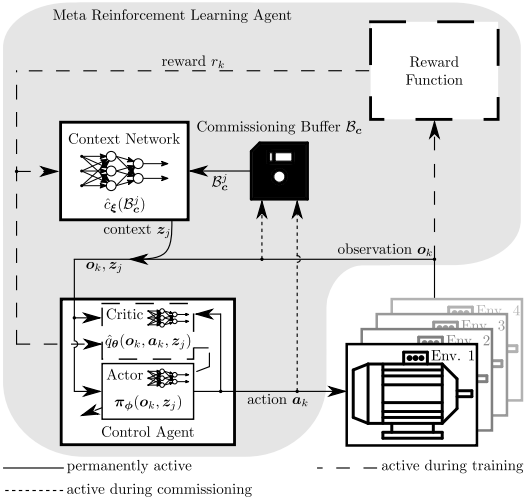
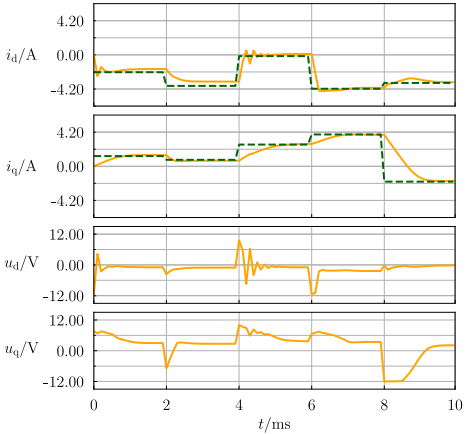
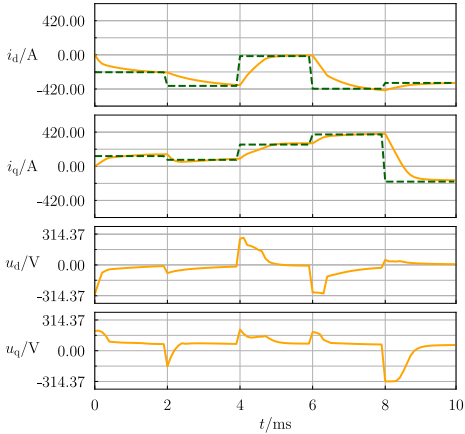


Fig. 7.16: A meta learning concept that we implemented successfully

Evaluation on (very) different motors



(a) Current control on a PMSM with low rated power



(b) Current control on a PMSM with high rated power

Summary

- Application of RL on technical systems comes with many challenges, e.g.,
  - Safety limits,
  - Real-time / computational constraints,
  - Varying and/or partially unknown environments.
- Real-world implementations often require more than bare RL algorithms, e.g.,
  - Integration of available a priori expert knowledge,
  - Combination with model-based control engineering tools.
- Ideal integration of data-driven RL solutions together with expert-based control engineering parts is subject to many open research question.

Summary of Part II: Reinforcement Learning Using Function Approximation

André
Bodmer

What was covered in the course

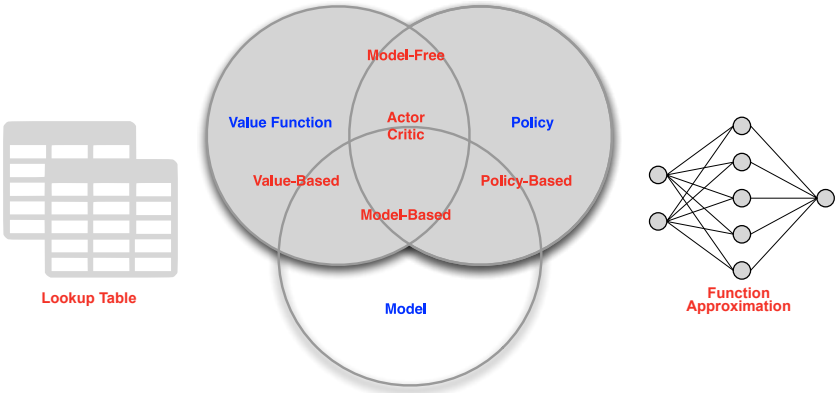


Fig. S-II.1: Main categories of reinforcement learning algorithms
(derived work based on D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)

Additional topics not covered in this lecture (1)

- Structured exploration: can we find a systematic way for fast and robust exploration?
  - R. Houthooft et al., "Vime: Variational information maximizing exploration", Advances in Neural Information Processing Systems, 2016
  - S. Levine, CS285 Deep Reinforcement Learning (lecture notes UC Berkeley), 2019
  - D. Silver, Reinforcement Learning (lecture notes UC London), 2015

- Imitation learning: how can we mimic the behavior of a certain baseline agent / controller / human expert?
  - A. Hussein et al., "Imitation learning: A survey of learning methods", ACM Computing Surveys (CSUR) 50.2, pp. 1-35, 2017
  - A. Attia and S. Dayan, "Global overview of imitation learning", arXiv:1801.06503, 2018

Additional topics not covered in this lecture (2)

- Multi-agent algorithms: finding solutions to distributed problems (e.g., for distributed energy systems).
  - L. Busoniu, R. Babuska and B. De Schutter. "A comprehensive survey of multiagent reinforcement learning." IEEE Transactions on Systems, Man, and Cybernetics, Part C 38.2, pp. 156-172, 2008
  - P. Hernandez-Leal, B. Kartal and M. Taylor. "Is multiagent deep reinforcement learning the answer or the question? A brief survey", Researchgate preprint, 2018
- Federated learning: finding solutions to distributed problems via multiple independent sessions, each using its own local information (addressing critical issues such as data privacy, data security, data access rights).
  - H. Zhuo et al. "Federated Deep Reinforcement Learning", arXiv:1901.08277, 2019
  - J. Qi et al. "Federated Reinforcement Learning: Techniques, Applications, and Open Challenges", arXiv:2108.11887, 2021

Learning summary

What you should have learned

- How to model decision processes using a Markov framework.
- Finding exact solutions using iterative tabular methods for discrete problem spaces.
- Finding approximate solutions for large discrete or continuous problem spaces based on function approximation.
- Application of just these techniques on a practical programming level.

Concluding remarks

- This is an introductory course to RL. We have only scratched the surface.
- Some aspects, especially within the exercises, had a control focus. In other application, specific RL solutions can look quite different.
- If you are interested in more practical RL insights in the field of electrical power systems, do not hesitate to contact us.