Centre for Mathematical Sciences
Mathematics, Faculty of Science

Deep reinforcement learning for real-time power grid topology optimization

by

Jacob Henning Rothschild

Abstract In our pursuit of carbon neutrality, drastic changes to generation and consumption of electricity will cause new and complex demands on the power grid and its operators. A cheap, promising, and under-exploited mitigation is real-time power grid topology optimization (RTTO). However, beyond the simplest action of line switching, the combinatorial and non-linear nature of RTTO has made all computational approaches infeasible for grids of interesting scale. At the same time, checking many of its boxes, RTTO may be a task for deep reinforcement learning (DRL).

This thesis starts by providing some further background as to why we care about RTTO. It then covers the deep learning and reinforcement learning concepts that underpin the Deep Q-Network (DQN). Building on this, it explains the DQN, Double DQN, Dueling DQN, and prioritized experience replay, in some depth. After that, it briefly covers the theory behind line losses, line overloads, and cascading line failures, as well as how bus switching and line switching can help. This is followed by a case study on the winning DRL submission of Geirina to the RTTO competition L2RPN 2019. Finally, the case for DRL for RTTO is made.

Acknowledgements

I would like to express my sincerest gratitude to Dr. Magnus Wiktorsson of Lund University and Dr. Alexander Gleim of Cognite for their excellent supervision.

Contents

List of Figures

1

Introduction

In light of rapid climate change, the sustainable development goals, the Paris climate agreement, and dire outlooks by IPCC, recent years have seen The European Union, Japan, South Korea, and more than 110 other countries pledge carbon neutrality by 2050 [United Nations, 2020]. Living up to such pledges will require a paradigm shift across economies, societies and communities [IRENA, 2017].

A key pillar of this shift is the transition to renewable energy sources, with the largest investments expected in wind and solar based generation [IRENA, 2017]. Unlike traditional generators whose production can be ramped up or down, wind and solar based generation fluctuate unpredictably with the weather, and are thus referred to as variable renewable energy (VRE). In a power grid without storage, we must always maintain balance between total generation and total load:

$$D(t) = \sum generation - \left(\sum load + \sum losses\right) \approx 0 \tag{1.1}$$

(The following paragraph cites Kroposki [2017].) Thus, as the proportion of VRE generation increases, we must find mitigations to hinder VRE fluctuations from causing $D(t)$ fluctuations. One such mitigation is making the other generators more flexible, both in how quickly they can ramp and in how large of a range they can ramp, as this allows them to more easily compensate for VRE fluctuations. Moreover, in the case where $D(t) > 0$, we can *curtail* VRE generation by simply wasting the surplus electricity. Another mitigation is distributed generation (DG), where we disperse the VRE geographically, thus averaging out the fluctuations and uncertainty caused by local weather. We can also break with tradition by making the load follow the generation, for example by only heating water tanks when VRE generation is high. Finally, although expensive, we can install storage capacity in the grid. This allows $D(t)$ to deviate from zero, as electricity is stored during overgeneration and released during undergeneration.

However, we also care about *load flow*: how current runs through the network. Although we may have $D(t) = 0$, the grid network can still suffer from overloads, compromised security, or large line losses. Load flow depends on the topology of the network and on all the individual generators, loads and storage capacities. By tweaking the variables under his or her control, an operator can thus improve the load flow. Individual VRE generators complicate this task by bringing fluctuations and uncertainty to the load flow and thus to what actions the operator should take. Moreover, load control adds more tweakable variables, increasing the dimensionality and complexity of the optimization task, while DG increases the complexity of load flow and thus also of its optimization. In total, as we expect an increasing penetration of VRE, optimizing load flow will become a more complex task that has to take into consideration faster fluctuations and more uncertainty.

At the same time, emission reductions is not only needed in electricity production. In 2019, transportation was responsible for 24% of direct CO2 emissions from fuel combustion, with road vehicles accounting for nearly three-quarters of transport CO2 emissions [Teter et al., 2020]. In 2017, heating and cooling in buildings and industry globally accounted for approximately 40% of all final energy consumption, with nearly 65% of its energy generated by fossil fuel sources [IEA, 2017]. To achieve climate targets, part of the transportation sector has to switch to electrical mobility, while the building and industry sectors must further electrify their heating and cooling processes [IRENA, 2017]. Both these transformations represent enormous sources of power demand and potential load peaks [Colle, 2020; Gray, 2017]. Thus, the electricity grid will need to transport substantially more power, which, ceteris paribus, means that the average line current will increase, and thus that load flows must be ever more optimized.

This combination of challenges would traditionally be addressed by a *fit-and-forget* approach [IRENA, 2017]. In such an approach, the grid would be physically expanded so as to increase its *robustness* to the point where load flow optimization would again become non-essential. However, rising DG penetration is making this costly [IRENA, 2017], and Transmission System Operators (TSOs) are facing difficulties in getting permits to construct overhead lines [Cole et al., 2011]. Generation *behind the meter*, for example with solar PV roofs, further complicates the economics of this, as it cuts into the revenue streams of utilities without reducing the infrastructure and transmission capacity they must provide [IRENA, 2017]. Operators are thus forced to increasingly optimize the load flow, even as this becomes more difficult.

Theoretically, load shedding should be able to optimize load flow, by simply disconnecting so much load and generation that all line currents are unproblematic. However, this would negatively impact the grid reliability and be highly disruptive to society. Similarly, re-dispatching can improve load flows, but its economic cost can be high [Kelly et al., 2020]. The operator must thus not just optimize load flows, but also do so at minimal cost and disruption. An under-exploited and under-researched mitigation is real-time power grid topology optimization (RTTO). In RTTO, the switchgear built into the grid is used as topological flexibilities that can be activated cheaply and quickly. Neither these grid flexibilities nor the knowledge that RTTO can improve load flows are new [Bacher and Glavitsch, 1986]. However, the computational complexity of RTTO has made it infeasible at scale, and thus beyond the state of the art [Marot et al., 2020]. Deep reinforcement learning (DRL) may be about to change this, making RTTO with DRL the topic of this thesis.

DRL is a merging of reinforcement learning (RL) and deep learning (DL). RL is about learning intelligent strategies for sequential decision making, only through trial-and-error interactions with an environment. DL is about learning to make sense of complex, high-dimensional input. Thus, DRL allows learning intelligent strategies for complex, high-dimensional environments. DRL is a promising approach to RTTO, as DRL is flexible enough to handle many of the complicating factors of RTTO [Zhang et al., 2020], DRL can be combined with traditional methods [Zhang et al., 2020], DL has shown promise in "understanding" load flow [Donnot et al., 2018], and DRL can cheaply gain lots of experience with an AC simulator [Lerousseau, 2018].

The goal of this thesis is to explain some central DRL concepts and to see them in the context of RTTO. Part I (Theory) will therefore focus on DRL and the concepts it builds on, only explaining power grids to the extent necessary to understand what RTTO is and how it can help. Part II (DRL for RTTO) focuses on the actual combination of DRL and RTTO, by presenting a case study of a winning DRL submission to an RTTO competition and motivating why DRL should be suitable for RTTO. More specifically:

- Chapter 2 will briefly introduce DL as a way to approximate an unknown function through fitting a deep neural network (DNN) to possibly noisy samples of that function.
- Chapter 3 will explain the finite Markov decision process (MDP) and its extensions, synchronous value iteration (SVI), asynchronous value iteration (AVI), real-time value iteration (RTVI), Q-learning, and Q-learning with action-value function approximation (AVFA), and how these all build on each other.
- Chapter 4 will explain the Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Dueling Deep Q-network (Dueling DQN), and prioritized experience replay (PER), in some depth, from the perspectives of the latter three patching the DQN.
- Chapter 5 will provide the necessary understanding of the power grid, present the overarching goals of power grid operations, and explain how RTTO helps address these.
- Chapter 6 will present a case study of the winning DRL-based submission of Geirina to the 2019 competition of the L2RPN series.
- Chapter 7 will make the case for why DRL is a promising method for RTTO.

1.1 Conventions and notation

- $\mathbb{N} = \{0, 1, 2, ...\}$.
- $\mathbb{1}\{\cdot\}$ is the indicator function on $\{\cdot\}$.
- Δw is a change in w, not to be confused with the Laplacian.
- $\mathbb{E}(A)$ and $\mathrm{Var}(A)$ are, respectively, the expectation and variance of the random variable (r.v.) A, while $\mathbb{P}(A = a)$ is the probability that $A = a$.
- Sets are denoted by calligraphic, uppercase letters, such as $\mathcal{A}$.
- If we have some function $f : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$, then $f(a, b) \in \mathcal{C}$, while $f(a, \cdot) : \mathcal{B} \to \mathcal{C}$, and $f(\cdot, \cdot) : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$.
- Using the character x as an example, tensors of various orders are denoted as follows: x is a scalar (zeroth-order tensor), $\boldsymbol{x}$ is a vector (first-order tensor), $\boldsymbol{X}$ is a matrix (second-order tensor), and $\boldsymbol{\mathcal{X}}$ is a third-order tensor.

- A variable with a subscript or superscript belongs to a variable whose tensor order is one larger. Using the example above, we may for example have:

$$\boldsymbol{X} = \begin{bmatrix} \text{—}\boldsymbol{x}^{(1)}\text{—} \\ \text{—}\boldsymbol{x}^{(2)}\text{—} \\ \vdots \\ \text{—}\boldsymbol{x}^{(n)}\text{—} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_m^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} \end{bmatrix} \tag{1.2}$$

- Again using the character x as an example, the meaning of the various scripts are as follows:
  - x_k represents the k-th coordinate of the vector $\boldsymbol{x}$.
  - $x_{(k)}$ represents the k-th version of a variable x, thus the k-th element in a sequence. Whenever we want to be extra clear about which iteration of x is being described, the notation $x_{(k)}$ will be used.
  - $x^{(k)}$ represents element k in some collection of similar elements. Whenever we want to be extra clear about which x in this collection is described, the notation $x^{(k)}$ will be used.
  - $x^{<k>}$ represents the k-th layer of a neural network.
- An r.v. is uppercase, while its realization is lowercase. For example, A is an r.v., while a is a realization of A.
- A function is given its notation based on what output it gives for deterministic input. For example, $f(x)$ is a scalar, $F(x)$ is an r.v., and $\boldsymbol{f}(x)$ is a vector.
- Exceptions to the above are $\mathcal{P}_{ss'}^a$, $\mathcal{R}_s^a$, and $\mathcal{Z}_{s'o}^a$, which are scalars; Q, V, and G, which are function approximators of, respectively, the functions q, v, and g; and $\mathcal{L}$, which is the total loss function of an estimator.

Part I

Theory

2

Deep learning

This chapter is inspired by Ng. The reader already familiar with deep learning can skim or skip this chapter, as no concepts will be covered in detail.

2.1 Linear regression

Imagine we have some unknown, affine function $f(\boldsymbol{x})$, defined on some unknown domain $\mathcal{D}$, and we can only make noisy measurements

$$f(\boldsymbol{x}) + \varepsilon,\ \mathbb{E}(\varepsilon) = 0,\ \operatorname{Var}(\varepsilon) < \infty. \tag{2.1}$$

We uniformly randomly sample

$$\boldsymbol{X} = \begin{bmatrix} —\boldsymbol{x}^{(1)}— \\ —\boldsymbol{x}^{(2)}— \\ \vdots \\ —\boldsymbol{x}^{(n)}— \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_m^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} \end{bmatrix},\ \boldsymbol{x}^{(i)} \text{ i.i.d.} \tag{2.2}$$

together with noisy measurements

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{x}^{(1)}) + \varepsilon^{(1)} \\ f(\boldsymbol{x}^{(2)}) + \varepsilon^{(2)} \\ \vdots \\ f(\boldsymbol{x}^{(n)}) + \varepsilon^{(n)} \end{bmatrix},\ \varepsilon^{(i)} \text{ i.i.d.} \tag{2.3}$$

and merge these into

$$\boldsymbol{S} = \begin{bmatrix} —\boldsymbol{x}^{(1)}— & y^{(1)} \\ —\boldsymbol{x}^{(2)}— & y^{(2)} \\ \vdots & \vdots \\ —\boldsymbol{x}^{(n)}— & y^{(n)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_m^{(1)} & y^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_m^{(2)} & y^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} & y^{(n)} \end{bmatrix} \tag{2.4}$$

Linear regression is about using $\boldsymbol{S}$ to find the weights $\boldsymbol{w}$ of an affine function

$$h(\boldsymbol{x}; \boldsymbol{w}) = w_0 + w_1 x_1 + w_1 x_2 + \ldots + w_m x_m \tag{2.5}$$

such that $h(\boldsymbol{x}; \boldsymbol{w})$ approximates $f(\boldsymbol{x})$ as closely as possible over $\mathcal{D}$. This can be viewed as the simplest form of supervised learning (SL), where SL is the task of using $\boldsymbol{S}$ to find a "nice" function h that approximates some function f over the full domain of f.

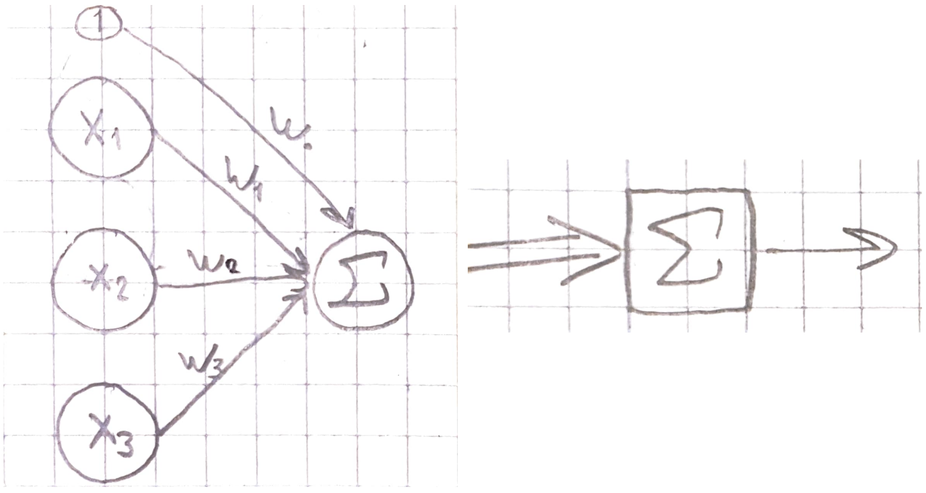

Figure 2.1: Two ways to visualize the affine function h.

We choose the mean squared error between h and f over $\mathcal{D}$ as our true closeness metric:

$$\mathrm{MSE}^{(\mathcal{D})}(\boldsymbol{w}) = \mathbb{E}_{\boldsymbol{x}\sim U(\mathcal{D})}\left([f(\boldsymbol{x}) - h(\boldsymbol{x};\boldsymbol{w})]^2\right) = \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}}\left(f(\boldsymbol{x}) - h(\boldsymbol{x};\boldsymbol{w})\right)^2 \tag{2.6}$$

As we do not know $\mathcal{D}$, we cannot compute (2.6). Thus needing a proxy, we instead minimize the mean squared error between h and our noisy measurements y:

$$\mathrm{MSE}(\boldsymbol{w}) = \frac{1}{n}\sum_{a=1}^{n}\left(y^{(a)} - h(\boldsymbol{x}^{(a)};\boldsymbol{w})\right)^2 \tag{2.7}$$

This choice is made since then

$$\mathrm{MSE}(\boldsymbol{w}) = 0 \implies \mathbb{E}\left(\mathrm{MSE}^{(\mathcal{D})}(\boldsymbol{w})\right) = 0 \tag{2.8}$$

2.1.1 Multivariate linear regression

We can generalize this to *multivariate linear regression*, simply by replacing all scalar targets and functions with their multivariate equivalent:

(2.1) is replaced by

$$\boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{\varepsilon},\ \mathbb{E}(\boldsymbol{\varepsilon}) = \boldsymbol{0},\ \forall j\left(\mathrm{Var}(\varepsilon_j) < \infty\right), \tag{2.9}$$

(2.4) is replaced by

$$\boldsymbol{S} = \begin{bmatrix} —\boldsymbol{x}^{(1)}— & —\boldsymbol{y}^{(1)}— \\ —\boldsymbol{x}^{(2)}— & —\boldsymbol{y}^{(2)}— \\ \vdots & \vdots \\ —\boldsymbol{x}^{(n)}— & —\boldsymbol{y}^{(n)}— \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_m^{(1)} & y_1^{(1)} & y_2^{(1)} & \cdots & y_K^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_m^{(2)} & y_1^{(2)} & y_2^{(2)} & \cdots & y_K^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} & y_1^{(n)} & y_2^{(n)} & \cdots & y_K^{(n)} \end{bmatrix}, \tag{2.10}$$

(2.5) is replaced by

$$\boldsymbol{h}(\boldsymbol{x};\boldsymbol{W}) = \boldsymbol{w}_0 + \boldsymbol{w}_1 x_1 + \boldsymbol{w}_1 x_2 + \ldots + \boldsymbol{w}_m x_m, \tag{2.11}$$

and (2.7) is replaced by

$$\mathrm{MSE}(\boldsymbol{W}) = \frac{1}{n}\sum_{a=1}^{n}\left(\boldsymbol{y}^{(a)} - \boldsymbol{h}(\boldsymbol{x}^{(a)};\boldsymbol{w})\right)^2 \tag{2.12}$$

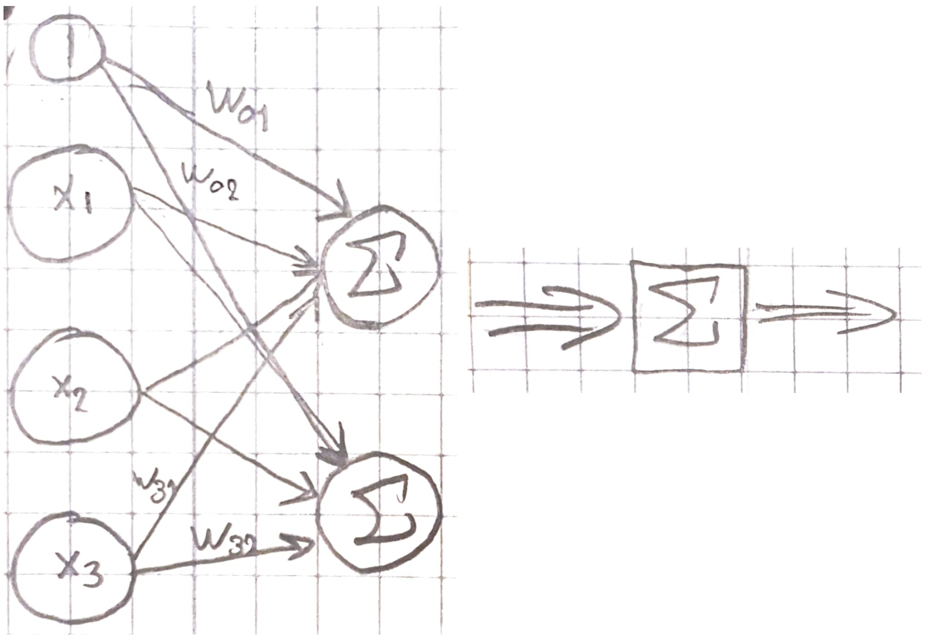

Figure 2.2: Two ways to visualize the affine function $\boldsymbol{h}$.

2.2 Artificial neural networks

For many interesting problems, the true function $\boldsymbol{f}$ is far from affine. An artificial neural network (ANN) is a type of function $\boldsymbol{h}(\boldsymbol{x};\boldsymbol{\mathcal{W}})$ that can mimic a wide range of affine and non-affine functions, solely through tweaking its weights $\boldsymbol{\mathcal{W}}$.

An ANN consists of multiple fully-connected (FC) layers feeding into each other. Each layer $\boldsymbol{h}^{<l>}$ consists of a multivariate affine function

$$\boldsymbol{g}^{<l>}(\boldsymbol{u};\boldsymbol{W}^{<l>}) = \boldsymbol{w}_0^{<l>} + \boldsymbol{w}_1^{<l>}u_1 + \boldsymbol{w}_2^{<l>}u_2 + \ldots + \boldsymbol{w}_{m^{<l>}}^{<l>}u_{m^{<l>}}$$

followed by an activation function $\boldsymbol{\psi}^{<l>}$. The activation function is some multivariate non-affine function where the i-th output only depends on the i-th input:

$$\boldsymbol{\psi}^{<l>}(\boldsymbol{v}) = (\psi^{<l>}(v_1), \psi^{<l>}(v_2), \ldots, \psi^{<l>}(v_{m^{<l>}}))$$

These non-linear activation functions are what give the ANN its flexibility, as only feeding affine functions into each other would just give another affine function:

$$w_0^{<1>} + w_1^{<1>} \cdot \left(w_0^{<0>} + w_1^{<0>}x\right) = \left(w_0^{<1>} + w_1^{<1>}w_0^{<0>}\right) + \left(w_1^{<1>}w_1^{<0>}\right)x \tag{2.13}$$

Put together, we get

$$\boldsymbol{h}^{<l>}(\boldsymbol{u}; \boldsymbol{W}^{<l>}) = \boldsymbol{\psi}^{<l>}(\boldsymbol{g}^{<l>}(\boldsymbol{u}; \boldsymbol{W}^{<l>}))$$

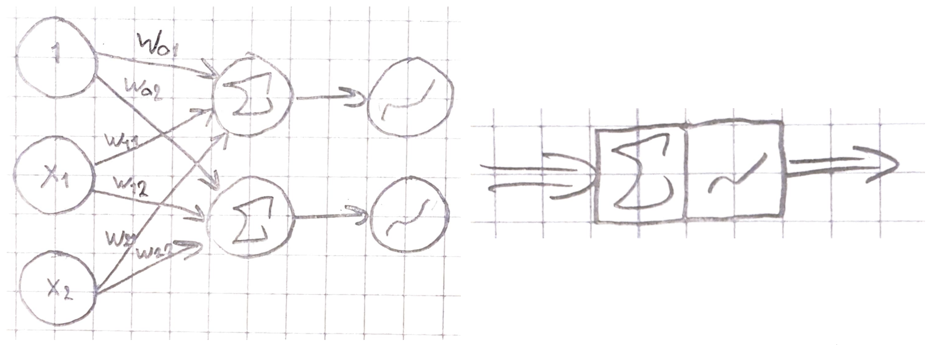

Figure 2.3: Two ways to visualize the nonlinear function $\boldsymbol{h}^{<1>}$.

In an ANN, the first layer is called the input layer, taking $\boldsymbol{x}$ as its input; the last layer is called the output layer, giving $\boldsymbol{y}$ as its output; and all layers in-between are called hidden layers. Feeding into each other, all of these layers together make up the ANN function $\boldsymbol{h}$:

$$\boldsymbol{h}(\boldsymbol{x}; \boldsymbol{\mathcal{W}}) = \boldsymbol{h}^{<k>}(\boldsymbol{h}^{<k-1>}(\ldots(\boldsymbol{h}^{<1>}(\boldsymbol{x}; \boldsymbol{W}^{<1>})\ldots); \boldsymbol{W}^{<k-1>}); \boldsymbol{W}^{<k>})$$

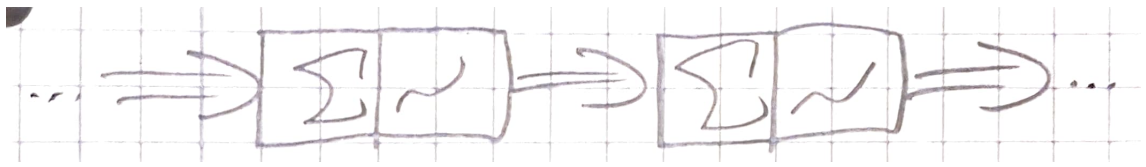

Figure 2.4: A way to visualize parts of the ANN function $\boldsymbol{h}$.

For the input layer and all the hidden layers, a common choice of activation function is the Rectified Linear Unit (ReLU):

Definition 1 (ReLU)**.** *The Rectified Linear Unit is the identity function for positive input and the zero function for non-positive input:*

$$\psi(v) = v \cdot \mathbb{1}\{v \geq 0\}$$

For the output layer, the activation function is chosen based on what format we want the final output to be in. In this paper, we want $y \in \mathbb{R}$, and will thus constrain ourselves to $\psi^{<k>}(v) = v$.

Like in the case of linear regression, the goal is finding the $\boldsymbol{h}$ minimizing

$$\text{MSE}^{(\mathcal{D})}(\boldsymbol{\mathcal{W}}) = \mathbb{E}_{\boldsymbol{x} \sim U(\mathcal{D})}\left([\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{h}(\boldsymbol{x}; \boldsymbol{\mathcal{W}})]^2\right), \tag{2.14}$$

but everything except the weights $\boldsymbol{\mathcal{W}} = (\boldsymbol{W}^{<1>}, \boldsymbol{W}^{<2>}, \ldots, \boldsymbol{W}^{<k>})$ is locked in place, and $\mathcal{D}$ is unknown. Thus, the relevant problem becomes finding the $\boldsymbol{\mathcal{W}}$ minimizing

$$\text{MSE}(\boldsymbol{\mathcal{W}}) = \frac{1}{n}\sum_{a=1}^{n}\left(\boldsymbol{y}^{(a)} - \boldsymbol{h}(\boldsymbol{x}^{(a)}; \boldsymbol{\mathcal{W}})\right)^2 \tag{2.15}$$

This process of finding $\boldsymbol{\mathcal{W}}$ is called machine learning. In the case of three or more hidden layers, we describe our ANN as a deep neural network (DNN) and the learning process as deep learning (DL).

2.3 Gradient descent

From multivariable calculus, we know that for any differentiable function L, its gradient ∇L gives the direction of steepest ascent. Since a differentiable function is locally linear, the opposite direction of ∇L must then give the direction of steepest descent.

In batch gradient descent (BGD) we have L take the weights $\mathcal{W}$ as its input, and then repeat the following update many times:

$$\Delta \mathcal{W} = -\frac{1}{2}\alpha_{(i)} \nabla L(\cdot) \tag{2.16}$$

$\alpha_{(i)}$ is the *learning rate*, a *hyperparameter*. All hyperparameters are chosen by us before BGD *training* begins. The process of finding good hyperparameters is called *hyperparameter tuning*.

Assuming a sufficiently small learning rate, (2.16) will make $L(\mathcal{W}_{(i)})$ monotonously decreasing in i. If we also assume that L is convex with a minimum and that ∇L is Lipschitz continuous, gradient descent will make $L(\mathcal{W}_{(i)})$ converge to its global minimum as $i \to \infty$.

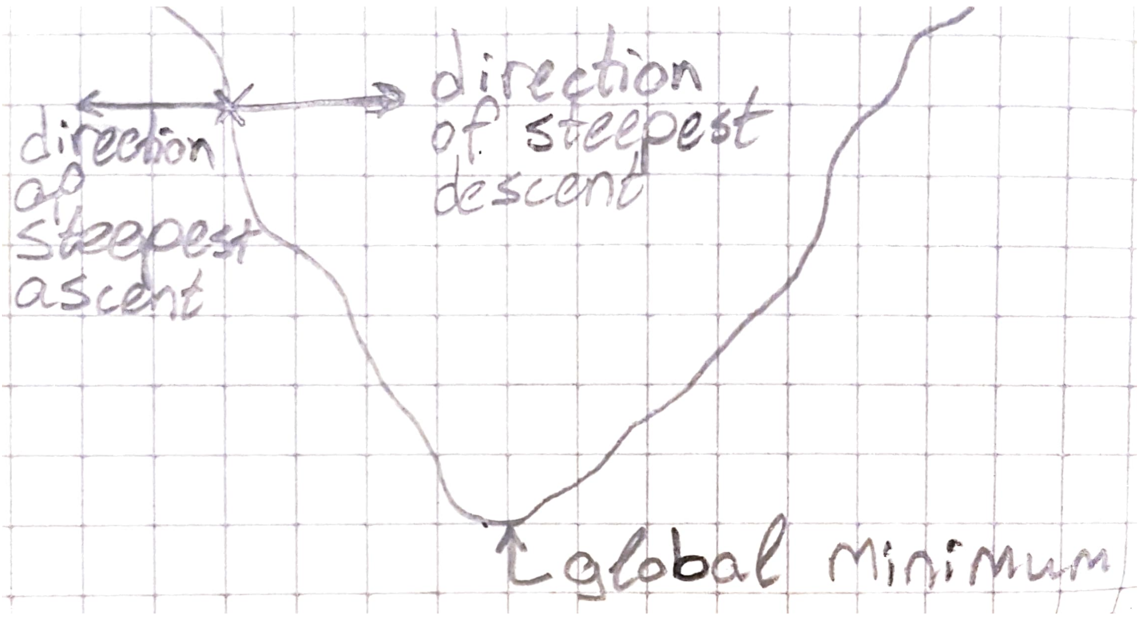

Figure 2.5: An (almost) convex, univariate function of x.

Thus, assuming that MSE($\mathcal{W}$) satisfies these constraints, which it does in Section 2.1, batch gradient descent will asymptotically make MSE($\mathcal{W}_{(i)}$) converge to its global minimum, through updates

$$\Delta \mathcal{W} = -\frac{1}{2}\alpha_{(i)} \nabla \text{MSE} \tag{2.17}$$

However, when $\boldsymbol{h}$ is an ANN, MSE is generally non-convex, causing BGD to lose its global convergence guarantees. Moreover, with ReLU, ∇MSE is only Lipschitz almost everywhere. However, in practice, optimizing an ANN through BGD usually makes its MSE converge to a good local minimum [Ng] and ReLU is usually a good choice of activation function [Szandała, 2021].

2.3.1 Stochastic sampling

In batch gradient descent, the full

$$\nabla \text{MSE} = \frac{1}{n} \sum_{a=1}^{n} \nabla \left([\boldsymbol{y}^{(a)} - \boldsymbol{h}(\boldsymbol{x}^{(a)}; \cdot)]^2 \right) \tag{2.18}$$

must be computed in every update. As such the computational cost of a single update increases linearly with n, the number of data points.

For each update, rather than using the whole $\boldsymbol{S}$, stochastic gradient descent (SGD) randomly samples a *mini-batch* $\boldsymbol{B}_{(i)} \subset \boldsymbol{S}$ and estimates ∇MSE from this:

$$\nabla \text{MSE} = \mathbb{E}(G_{(i)}), \; G_{(i)} := \frac{1}{|\boldsymbol{B}_{(i)}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \boldsymbol{B}_{(i)}} \nabla \left([\boldsymbol{y} - \boldsymbol{h}(\boldsymbol{x}; \cdot)]^2 \right) \tag{2.19}$$

Thus, the computational cost of an update becomes independent of n. Moreover, the variance of $G_{(i)}$ typically causes SGD to converge to better results than BGD, as noise helps SGD escape the poor, sharp minima of an ANN's cost function [Keskar et al., 2017].

2.3.2 Other optimizers

RMSprop [Hinton et al., 2012], *Adam* [Kingma and Ba, 2015], and *Nadam* [Dozat, 2016] are popular examples of more complicated optimizers that improve upon SGD. Given sufficient hyperparameter tuning, these three seemingly always perform better than or equal to SGD [Choi et al., 2020].

3

Reinforcement learning

This chapter is a paraphrasing of parts of Silver [2015] and Sutton and Barto [2018]

In SL, the goal was to make one optimal decision, for example on whether a photo is of a dog or a cat. In reinforcement learning (RL), the goal is rather to make an optimal sequence of decisions. The distinction is that for SL, the next time we make a decision, it will be independent of all previous decisions we have made, while for RL, consecutive decisions can be highly dependent. Thus, a RL agent must not only take into account the immediate rewards of various actions, but also all future rewards that its actions may make more or less obtainable.

The goal of RL is to find the optimal policy π, such that following π means maximizing the expectation of the total reward, no matter the situation. What this means will become clearer over the next section, while how this is achieved is the content of the rest of this chapter.

3.1 Formalization

In RL, we are given an agent in an environment, and our goal is to learn which action to take in each state, through episodes of interactions between agent and environment.

An episode is a sequence $(s_{(t)}, a_{(t)}, r_{(t+1)})_{t=1}^{\infty}$, where every time-step t is broken down as follows:

1. Agent is in state $s_{(t)}$
2. Agent takes action $a_{(t)}$
3. Environment gives immediate reward $r_{(t+1)}$
4. Environment moves agent to state $s_{(t+1)}$

We assume that the environment can be perfectly modeled by a finite Markov Decision Process (MDP):

Definition 2 (Markov decision process)**.** *A Markov Decision Process is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$*, constant over time, where:*

- $\mathcal{S}$ *is the set of states that* $s_{(t)}$ *can occupy.*
- $\mathcal{A}$ *is the set of actions that* $a_{(t)}$ *can occupy.*

- *$\mathcal{P}$ determines the distribution of $S_{(t+1)}$, conditioned on only $S_{(t)}$ and $A_{(t)}$: $\mathcal{P}^a_{ss'} = \mathbb{P}(S_{(t+1)} = s' \mid S_{(t)} = s, A_{(t)} = a)$. In an MDP, we assume that all states are Markov: no other states and actions than $S_{(t)}$ and $A_{(t)}$ directly impact the distribution of $S_{(t+1)}$.*
- *$\mathcal{R}$ determines the expected immediate reward, also under the assumption that only $S_{(t)}$ and $A_{(t)}$ directly impact $R_{(t+1)}$: $\mathcal{R}^a_s = \mathbb{E}(R_{(t+1)} \mid S_{(t)} = s, A_{(t)} = a)$.*
- *$\gamma \in [0,1]$ is the discount factor.*

We refer to an MDP as *finite* when $|\mathcal{S}| < \infty$ and $|\mathcal{A}| < \infty$.

The goodness metric of an agent interacting with an MDP is defined to be the expected total return $\mathbb{E}(T)$, where

$$T := R_{(2)} + \gamma R_{(3)} + \gamma^2 R_{(4)} + \dots \tag{3.1}$$

Thus, γ determines how much an optimal agent should sacrifice immediate reward for delayed reward. The usual choice is $\gamma < 1$, primarily for mathematical convenience. This paper will everywhere assume that $\gamma < 1$, for that same mathematical convenience.

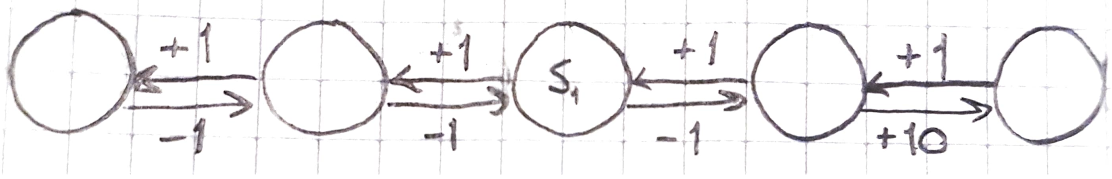

Figure 3.1: An example of a deterministic MDP, where $\mathcal{S}$ consist of the five circles, we start at $s_{(1)}$, $\mathcal{A} = \{$move left, move right$\}$, and T is maximized by sacrificing immediate reward in the beginning, given $\gamma > \frac{1}{4}(\sqrt{137} - 11) \approx 0.18$.

An agent is guided by some policy π:

Definition 3 (Policy)**.** *The policy π determines the probability mass function of $A_{(t)}$ conditioned on $S_{(t)}$:*

$$\pi(s,a) = \mathbb{P}(A_{(t)} = a \mid S_{(t)} = s)$$

Together, an MDP and a policy generate a stochastic sequence $(S_{(t)}, A_{(t)}, R_{(t+1)})_{t=1}^{\infty}$, of which an episode $(s_{(t)}, a_{(t)}, r_{(t+1)})_{t=1}^{\infty}$ is a realization.

Being guided by a policy, our agent can be assigned a value function and an action-value function:

Definition 4 (Value function)**.** *The value function $v^{(\pi)}(s)$ is the expected total return when starting in state s and following policy π:*

$$v^{(\pi)}(s) = \mathbb{E}(T \mid S_{(1)} = s, \textit{all actions chosen according to } \pi)$$

Definition 5 (Action-value function)**.** *The action-value function $q^{(\pi)}(s,a)$ is the expected total return when starting in state s, taking action a, and then following policy π:*

$$q^{(\pi)}(s,a) = \mathbb{E}(T \mid S_{(1)} = s, A_{(1)} = a, \textit{all other actions chosen according to } \pi)$$

Thus, the goal of RL can more precisely be formulated as finding a *globally optimal* policy π, such that $q^{(\pi)}(s,a) \geq q_\mu(s,a)$ for all s, a and policies μ. We denote such an optimal policy $\pi^{(*)}$ and its corresponding value and action-value functions $v^{(*)}$ and $q^{(*)}$, and get

$$q^{(*)}(s,a) = \max_\pi q^{(\pi)}(s,a) \tag{3.2}$$

$$v^{(*)}(s) = \max_\pi v^{(\pi)}(s) \tag{3.3}$$

where (3.3) follows from the relationship between $q^{(\pi)}$ and $v^{(\pi)}$:

Lemma 1.

$$q^{(\pi)}(s,a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \cdot v^{(\pi)}(s')$$

$$v^{(\pi)}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \cdot q^{(\pi)}(s,a)$$

Proof. This follows directly from Definition 2, Definition 4, and Definition 5. □

A natural question is whether at least one $\pi^{(*)}$ always exists. To answer this, we must make some observations: First, randomly choosing an action is never better than choosing an optimal action, so there is for any stochastic policy μ a deterministic policy π that is everywhere at least as good. Second, if different π are locally optimal on different states s, we can construct a $\pi^{(*)}$ that for any s follows whichever π is locally optimal on s; such a $\pi^{(*)}$ is globally optimal. Third, as $\mathcal{S}$ and $\mathcal{A}$ are finite, so is the set $\mathcal{D}$ of deterministic policies, as $|\mathcal{D}| = |\mathcal{S}|^{|\mathcal{A}|}$; thus, $\mathcal{D}$ is complete. In total, there must for any state s exist at least one deterministic, locally optimal policy π, and there must therefore exist at least one deterministic, globally optimal policy $\pi^{(*)}$.

One such $\pi^{(*)}$ can be trivially deduced if we already know $q^{(*)}$:

Definition 6 (Optimal greedy policy)**.** *Given some state s, the optimal greedy policy picks the action a that maximizes $q^{(*)}(s,a)$:*

$$\pi^{(*)}(s,a) = \mathbb{1}\{a = \arg\max_{a' \in \mathcal{A}} q^{(*)}(s,a')\} \tag{3.4}$$

Theorem 1. *The optimal greedy policy is globally optimal.*

This allows us to do *value-based RL*, in which we only focus on estimating $q^{(*)}$, implicitly having (3.4) as our policy. Although other forms of RL exist, value-based RL is the only type of RL this paper will cover.

An environment may follow some nice MDP, but hide $s_{(t)}$ from the agent, only letting it make partial observations $o_{(t)}$, where $o_{(t)}$ is based on $s_{(t)}$ and $a_{(t-1)}$. Episodes with such an environment will look like $(a_{(t)}, o_{(t+1)}, r_{(t+1)})_{t=1}^{\infty}$, and the environment can be modeled as a POMDP:

Definition 7 (Partially observable Markov decision process)**.** *A Partially observable Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma)$, constant over time, where:*

- *$\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma$ are defined as in Definition 2.*

- *$\mathcal{O}$ is the set of observations that $o_{(t)}$ can occupy.*
- *$\mathcal{Z}$ determines the distribution of $O_{(t+1)}$, conditioned on only $s_{(t+1)}$ and $a_{(t)}$: $\mathcal{Z}^a_{s'o} = \mathbb{P}(O_{(t+1)} = o \mid S_{(t+1)} = s', A_{(t)} = a)$.*

In other environments, the state or action space may be infinite, or even continuous. By allowing partial observability and infinite $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$, we can essentially model all realistic environments. For brevity and simplicity, the rest of Part I will assume fully observable and finite MDPs. However, the techniques of Section 3.4 and Chapter 4 also work with partial observability and $|\mathcal{S}| = \infty$. Nonetheless, they do require that $|\mathcal{A}| < \infty$, as (3.4) is otherwise incomputable. To support $|\mathcal{A}| = \infty$, other types of RL — such as *policy-based RL* or *actor-critic* — are needed.

3.2 Model-based planning

Let us assume that we know the MDP of our environment, and that this MDP is simple enough that we can use it for computations. This section shows how we can then iteratively approach $q^{(*)}$, and thus $\pi^{(*)}$. Better Big-Oh complexity can be achieved by instead approaching $v^{(*)}$. However, picking actions based on $v^{(*)}$ requires us to know $\mathcal{P}$, while we can pick actions based on $q^{(*)}$ without knowing $\mathcal{P}$. As the MDP, and thus $\mathcal{P}$ is unknown in model-free RL, approaching $q^{(*)}$ is what we will focus on, as model-free RL is what we actually care about, model-based planning just being a stepping stone for getting there.

3.2.1 Synchronous value iteration

Definition 8 (Bellman operator). *$B^{(*)} : \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ is an operator where*

$$(B^{(*)}f)(s,a) = \mathbb{E}(R_{(2)} + \gamma \max_{a'\in\mathcal{A}} f(S_{(2)}, a') \mid S_{(1)} = s, A_{(1)} = a),\ \ f \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$$

As $\gamma < 1$, the Bellman operator is a contraction mapping, which by Banach's fixed point theorem means it has a unique fixed point $h \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ that $B^{(*)}B^{(*)}B^{(*)}\ldots f$ will converge to for all $f \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$.

From Section 3.1, we know that

$$v^{(*)}(s) = q^{(*)}(s, \arg\max_{a\in\mathcal{A}} q^{(*)}(s,a)) = \max_{a\in\mathcal{A}} q^{(*)}(s,a) \tag{3.5}$$

$$q^{(*)}(s,a) = \mathcal{R}^a_s + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} \cdot v^{(*)}(s') \tag{3.6}$$

This allows us to trivially deduce the Bellman equation

Theorem 2 (Bellman equation). *$q^{(*)}$ is the unique fixed point of the Bellman operator:*

$$q^{(*)}(s,a) = \mathcal{R}^a_s + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} \cdot \max_{a'\in\mathcal{A}} q^{(*)}(s',a')$$

Iteratively applying $B^{(*)}$ to any $f \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ will thus cause asymptotic convergence to $q^{(*)}$, directly motivating the synchronous value iteration (SVI) algorithm:

Definition 9 (Synchronous value iteration)**.** *In synchronous value iteration, we start by estimating all action-values to be 0, and then apply the Bellman operator iteratively to our estimate function, asymptotically converging to $q^{(*)}$:*

$$\begin{aligned} q_{(1)} &= \mathbf{0}, \\ q_{(n+1)} &= B^{(*)} q_{(n)}, \ n \in \mathbb{Z}_+ \end{aligned}$$

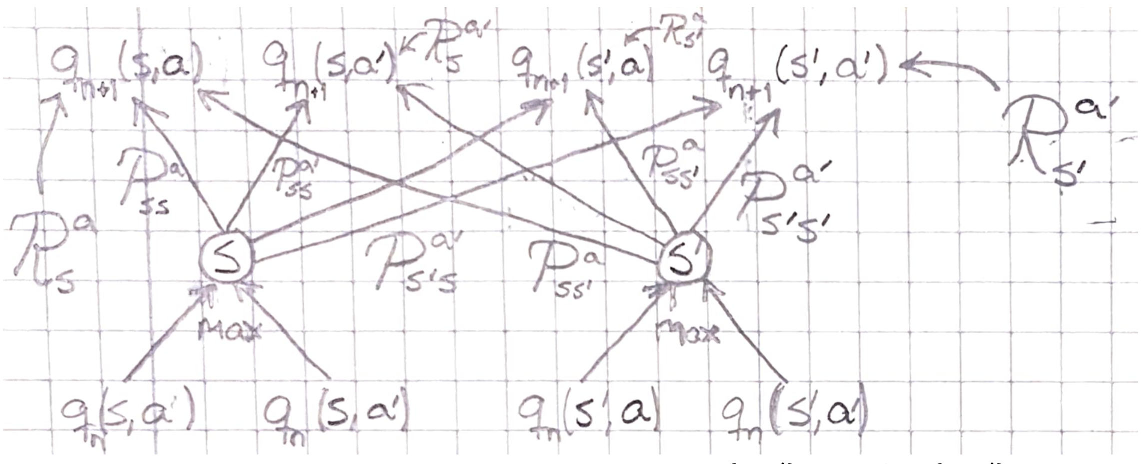

Figure 3.2: A single update of SVI, where $\mathcal{S} = \{s, s'\}$ and $\mathcal{A} = \{a, a'\}$.

3.2.2 Asynchronous value iteration

As $B^{(*)}$ acts on a whole function, a single step of SVI requires computing

$$\mathbb{E}(R_{(2)} + \gamma \max_{a' \in \mathcal{A}} q_{(n)}(S_{(2)}, a') \mid S_{(1)} = s, A_{(1)} = a)$$

for every $(s, a) \in \mathcal{S} \times \mathcal{A}$. In asynchronous value iteration (AVI), we again start with $q = \mathbf{0}$, but rather than an update looking like $q \leftarrow B^{(*)} q$, an update looks like

1. Sample an $(s, a) \in \mathcal{S} \times \mathcal{A}$
2. Only change q for that (s, a):

$$q(s, a) \leftarrow \mathbb{E}(R_{(2)} + \gamma \max_{a' \in \mathcal{A}} q(S_{(2)}, a') \mid S_{(1)} = s, A_{(1)} = a) \tag{3.7}$$

Thus, the complexity of an update shrinks from $\mathcal{O}(|\mathcal{A}|^2 |\mathcal{S}|^2)$ to $\mathcal{O}(|\mathcal{A}||\mathcal{S}|)$, allowing progress to be made with less computation. This updated $q(s, a)$ is used when updating other $q(s', a')$, a phenomenon known as *bootstrapping*. Due to this bootstrapping, improvements in estimating $q(s, a)$ can result in faster improvements when estimating other $q(s', a')$. Hence, AVI can significantly improve computational efficiency.

Theorem 3. *If every (s, a) is visited infinitely many times, asynchronous value iteration will converge to $q^{(*)}$.*

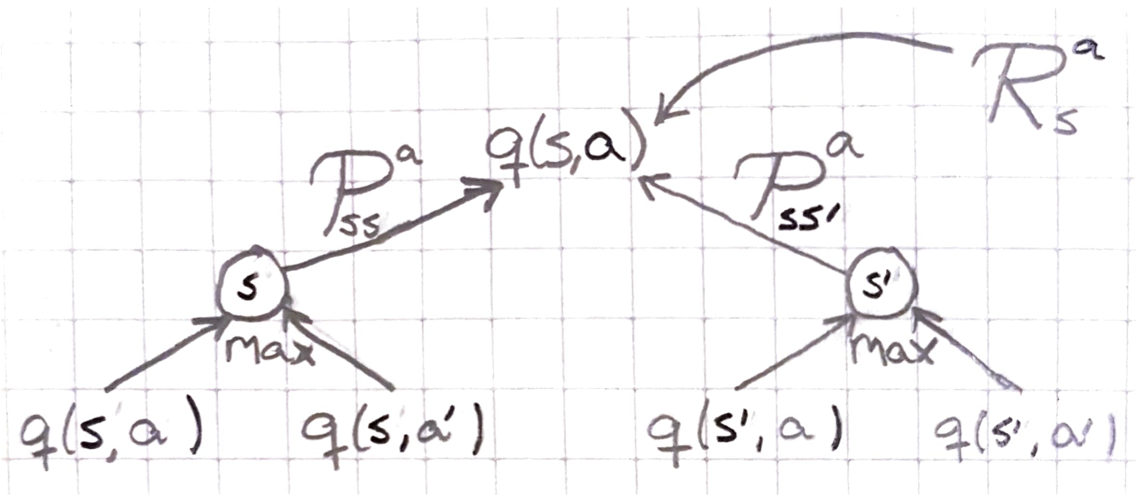

Figure 3.3: A single update of AVI, where $\mathcal{S} = \{s, s'\}$ and $\mathcal{A} = \{a, a'\}$.

3.2.3 Real-time value iteration

Sampling (s, a) uniformly at random ensures that the condition of Theorem 3 is satisfied with probability 1, thus that

$$q_{(k)} \xrightarrow{p} q^{(*)}$$

However, if $q(s, a) > q(s, a')$, then $q(s, a)$ will be used in bootstrapping, while $q(s, a')$ will not; thus, estimating $q(s, a)$ well will speed up improvements for other (s'', a''), while estimating $q(s, a')$ well will not. By sampling with priority, rather than uniformly, we can thus improve our computational efficiency considerably.

Real-time value iteration (RTVI) is one such form of prioritized AVI. In RTVI we make our agent follow an *agent policy* $\pi_{(k)}$ that is generated from $q_{(k)}$ and that thus changes on every q-update. The agent will wander through the environment, following this ever-changing $\pi_{(k)}$. At every t, the agent will be in some state $s_{(t)}$, pick some action $a_{(t)}$, and we will update $q(s_{(t)}, a_{(t)})$. Thus, we sample (s, a) by

$$(S, A) = (S_{(t)}, A_{(t)}),\ A_{(t)} \sim \pi_{(t)}(S_{(t)}, \cdot), \tag{3.8}$$

simplifying our updates to:

$$q(s_{(t)}, a_{(t)}) \leftarrow \mathbb{E}(R_{(2)} + \gamma \max_{a' \in \mathcal{A}} q(S_{(2)}, a') \mid S_{(1)} = s_{(t)}, A_{(1)} = a_{(t)}). \tag{3.9}$$

If we set

$$\pi_{(t)}(s, a) = \mathbb{1}\{a = \arg\max_{a' \in \mathcal{A}} q_{(t)}(s, a')\}, \tag{3.10}$$

this will *exploit* the information in q by focusing only on the (s, a) we believe to be most relevant to estimating $q^{(*)}$. However, this will generally mean that some (s, a) are only visited a finite number of times, and thus that

$$q_{(t)} \overset{p}{\not\to} q^{(*)}$$

Concretely, the problem is that an (s, a) which would actually be visited by $\pi^{(*)}$ — and thus matters a lot to $q^{(*)}$ — may be discarded by (3.10): if (s, a) gives lots of delayed reward, but its initial reward is small or has high variance, the early estimates of $q(s, a)$ may be small, thus causing (s, a) to never again be visited and $q(s, a)$ to never again be updated.

We can solve this by mixing in a bit of *exploration*: visiting less visited (s, a), even though $q(s, a)$ tells us otherwise. This means we may discover that an (s, a) is better than we thought, and thus avoid overlooking ideal (s, a), but it also means we will spend more resources computing irrelevant $q(s, a)$.

This exemplifies the *exploration-exploitation trade-off*: if you explore too much, you will converge too slowly; if you exploit too much, you will probably get stuck in a local optimum.

There are many techniques for dealing with this trade-off. The simplest and most popular is ε-greedy, as it is easy to implement, ensures $q_{(t)} \xrightarrow{p} q^{(*)}$, and requires no dedicated computations:

Definition 10 (ε-greedy)**.** *An agent following the ε-greedy policy has probability ε (hyperparameter) to choose its next action uniformly at random and probability $1 - \varepsilon$ to choose its next action greedily on q:*

$$\pi_{(t)}(s, a) = \begin{cases} \frac{\varepsilon}{|\mathcal{A}|} + 1 - \varepsilon, & \textit{if } a = \arg\max_{a' \in \mathcal{A}} q_{(t)}(s, a') \\ \frac{\varepsilon}{|\mathcal{A}|}, & \textit{otherwise} \end{cases} \tag{3.11}$$

3.3 Model-free reinforcement learning

In Section 3.2 we assumed that the MDP of the environment was known and computationally feasible. However, as this seldom holds in reality, RL does away with that assumption, thus making the updates of SVI and AVI incomputable or computationally infeasible. There are two main approaches for solving this: *model-based RL* and *model-free RL*. In model-based RL, we use episodes to approximate the MDP, and then use the techniques of Section 3.2 to find $q^{(*)}$ on this MDP approximation. In model-free RL, we use episodes to directly approximate the q-updates of Section 3.2. This thesis will only focus on model-free RL.

If the agent is in state $s_{(t)}$ and performs action $a_{(t)}$, the environment will return $r_{(t+1)}$ and $s_{(t+1)}$. $r_{(t+1)}$ and $s_{(t+1)}$ will then be realizations of, respectively,

$$(R_{(2)} \mid S_{(1)} = s_{(t)}, A_{(1)} = a_{(t)}) \text{ and } (S_{(2)} \mid S_{(1)} = s_{(t)}, A_{(1)} = a_{(t)}),$$

and thus estimate, respectively,

$$\mathbb{E}(R_{(2)} \mid S_{(1)} = s_{(t)}, A_{(1)} = a_{(t)}) \text{ and } \mathbb{E}(S_{(2)} \mid S_{(1)} = s_{(t)}, A_{(1)} = a_{(t)}).$$

Thus, we can estimate (3.9) with

$$q(s_{(t)}, a_{(t)}) \leftarrow r_{(t+1)} + \gamma \max_{a' \in \mathcal{A}} q(s_{(t+1)}, a') \tag{3.12}$$

while knowing nothing about the underlying MDP of the environment.

Employing a more cautious version of this update estimation, we get the famous *Q-learning* algorithm:

Definition 11 (Q-learning)**.** *Have an agent follow $\pi_{(t)}$, with $\pi_{(t)}$ relying on $q_{(t)}$. Set $q_{(1)} = \mathbf{0}$, and for every transition $(s_{(t)}, a_{(t)}, r_{(t+1)}, s_{(t+1)})$ experienced by the agent, make the following update:*

$$q(s_{(t)}, a_{(t)}) \leftarrow \big(1 - \alpha_{(t+1)}\big) \cdot q(s_{(t)}, a_{(t)}) + \alpha_{(t+1)} \underbrace{\left(r_{(t+1)} + \gamma \max_{a' \in \mathcal{A}} q(s_{(t+1)}, a')\right)}_{\textit{TD-target}} . \quad (3.13)$$

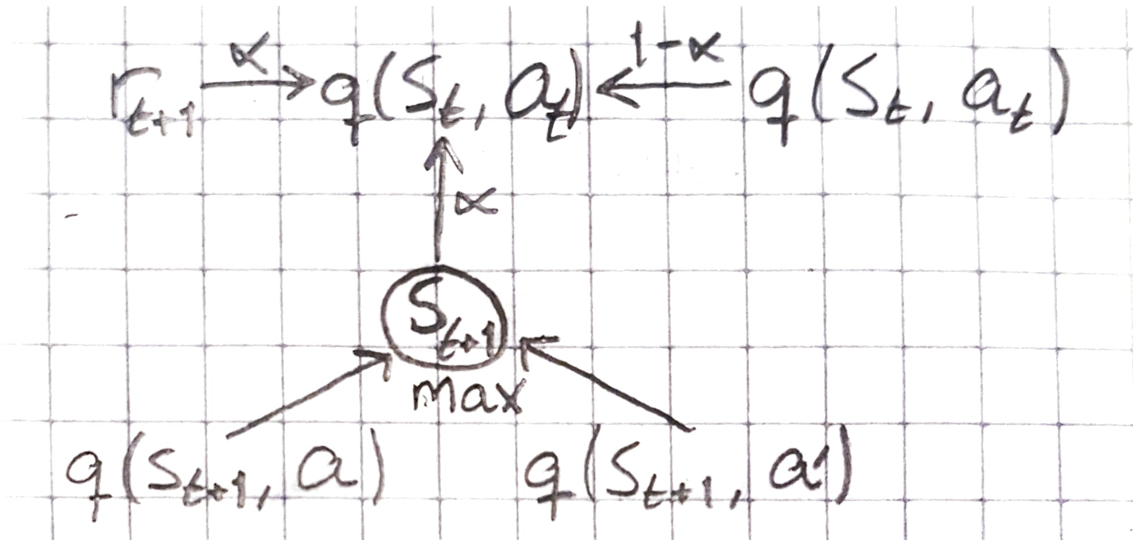

Figure 3.4: A single update of Q-learning, where $\mathcal{A} = \{a, a'\}$.

With such a weighted average, (3.13) makes $q(s_{(t)}, a_{(t)})$ an exponentially decaying average of all estimates of $q^{(*)}(s_{(t)}, a_{(t)})$ made so far. This ensures that the fluctuations in q caused by variance in R and S are kept small, while newer estimates relying on better bootstraps are given more weight than past estimates.

$(\alpha_{(t+1)})_{t=1}^{\infty}$ is a hyperparameter called *learning rate*. In Section 3.4, we will see that this learning rate is the same as that of Section 2.3.

Moreover, (3.13) can be rewritten as computing the TD-error and adding a multiple of it to $q(s_{(t)}, a_{(t)})$:

$$q(s_{(t)}, a_{(t)}) \leftarrow q(s_{(t)}, a_{(t)}) + \alpha_{(t+1)} \underbrace{\left(r_{(t+1)} + \gamma \max_{a' \in \mathcal{A}} q(s_{(t+1)}, a') - q(s_{(t)}, a_{(t)})\right)}_{\text{TD-error}} . \quad (3.14)$$

The value of the TD-error — and thus of this rewriting — will become apparent in Section 4.4.

Q-learning has similar conditions for guaranteed convergence as does AVI, but also places some requirements on $(\alpha_{(t+1)})_{t=1}^{\infty}$:

Theorem 4. *If every (s, a) is visited infinitely many times and*

$$\sum_{t=1}^{\infty} \alpha_{(t+1)} = \infty \qquad \textit{and} \qquad \sum_{t=1}^{\infty} \alpha_{(t+1)}^2 < \infty \quad (3.15)$$

then Q-learning will converge to $q^{(*)}$.

3.4 Action-value function approximation

In many applications, the state-action space is enormous. This makes the table-based Q-learning approach of Section 3.3 infeasible, as an action-value is only updated when its state-action pair is visited by the agent, which may be extremely rare. Moreover, storing all action-values in an enormous, dense matrix can lead to memory issues. To overcome both of these issues, we can resort to action-value function approximation (AVFA).

Function approximation is the task of finding a "nice" function h that approximates a predetermined function f. In our case, we wish to approximate $q(s, a)$ with the composite function

$$Q(s, a; \mathcal{W}) = g_a(\boldsymbol{x}(s); \mathcal{W}) \tag{3.16}$$

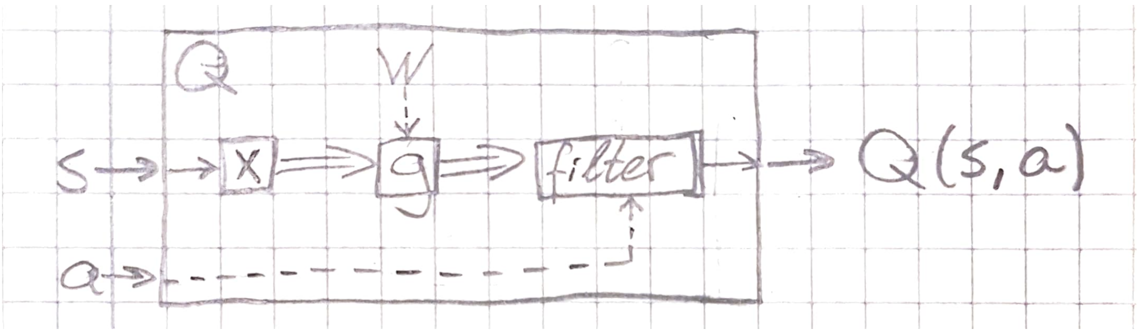

Figure 3.5: How Q generates an output from $s, a, \mathcal{W}$.

$\boldsymbol{x}(s)$ is a predetermined function that takes a state s as input and gives the feature vector of s as its output. A feature vector is a multi-dimensional, quantitative representation of a state. In practice, our environments usually omit s, rather giving us $\boldsymbol{x}(s)$ directly.

$\boldsymbol{g}(\boldsymbol{x}; \mathcal{W})$ is a predetermined function which outputs an $|\mathcal{A}|$-dimensional vector of estimated action-values, whose treatment of $\boldsymbol{x}$ depends on the values in $\mathcal{W}$, and for which $\nabla \boldsymbol{g}(\boldsymbol{x}; \cdot)$ is Lipschitz almost everywhere. $\boldsymbol{g}(\boldsymbol{x}; \mathcal{W})$ can for example be one of the affine functions or ANNs of Chapter 2. Note that we have here constrained $\boldsymbol{g}$ to functions of interest to this thesis. Many learnable functions that do not fulfil our constraints, such as decision trees and nearest neighbor, may still be viable choices for $\boldsymbol{g}$.

As everything in $Q(\cdot, \cdot; \mathcal{W})$ except $\mathcal{W}$ is predetermined, our approximation task boils down to finding the optimal $\mathcal{W}$. To approach this optimal $\mathcal{W}$, we combine Q-learning with one of the optimizers of Section 2.3. In Section 3.3, Q-learning would combine an encountered transition (s, a, r, s') with the s'-bootstraps $q(s', \cdot)$ to update $q(s, a)$ towards the target

$$r + \max_{a' \in \mathcal{A}} q(s', a'; \mathcal{W}). \tag{3.17}$$

Instead of doing that, we now make the optimizer change $\mathcal{W}$ in a way that moves $Q(s, a; \mathcal{W})$ towards

$$r + \max_{a' \in \mathcal{A}} Q(s', a'; \mathcal{W}). \tag{3.18}$$

4

Deep reinforcement learning

In practice, all algorithms introduced in this chapter sample mini-batches of transitions, rather than single transitions. However, where this does not meaningfully change the logic, we will for convenience pretend like they sample single transitions.

Just as in SL, the linear function approximators of Section 3.4 require directly applicable, hand-crafted features for their input. However, in scenarios approaching real-world complexity, such feature engineering becomes infeasible, forcing the function approximators to themselves infer useful features from complex, high-dimensional sensory input [Mnih et al., 2015]. DL has proven highly competent at discovering intricate structures in high-dimensional data [LeCun et al., 2015]. This motivates the desire to use a DNN as our action-value function approximator, thus combining DL and RL into DRL. However, simply replacing the linear function approximators of Section 3.4 with DNNs often leads to catastrophic divergence [Silver, 2015]. This chapter describes the tweaks that made DRL possible and practical.

4.1 Deep Q-Network

Except where otherwise stated, this section is based entirely on Mnih et al. [2015].

In practice, catastrophic divergence will often be the result when implementing AVFA as in Section 3.4 with a neural network as the approximator [Silver, 2015].

This instability has several causes:

- The dependence in the sequence of observed transitions, leading to increased variance in the updates. An explanation may be that the dependence between transitions $e_{(t)}$ and $e_{(t+1)}$ often leads to a positive correlation between their network updates $\Delta_{e_{(t)}}\boldsymbol{\mathcal{W}}$ and $\Delta_{e_{(t+1)}}\boldsymbol{\mathcal{W}}$, which leads to increased variance in mini-batch updates containing both $e_{(t)}$ and $e_{(t+1)}$.

- That small changes to $Q(\cdot, \cdot; \boldsymbol{\mathcal{W}})$ can significantly change the agent policy π, thus data distribution, possibly leading the parameters to oscillate, diverge, or get stuck in poor local minima. Part of the reason may be that π narrows in on an ever-smaller subset $\mathcal{B}$ of $\mathcal{S} \times \mathcal{A}$ with high Q-estimates, leading to Q overfitting pairs in $\mathcal{B}$, while forgetting about, thus under- or overestimating pairs in $\mathcal{B}^C$; if most pairs in $\mathcal{B}^C$ are underestimated, the policy may become prone to stick to $\mathcal{B}$, thus get stuck in a local minimum; if most pairs in $\mathcal{B}^C$ are overestimated, the parameters may become prone to oscillate, as π eventually switches to narrow in on the overestimated state-action pairs, overfit these, overestimate other state-action pairs, and so on.

- That estimates and targets are correlated, possibly leading the policy to oscillate or diverge. A likely explanation is that this correlation gives a feedback loop, where — over multiple updates — an increase in $Q(s, a)$ causes an increase in its target y, which later causes an increase in $Q(s, a)$, and so on; thus, the variance of $Q(s, a)$ increases, making oscillations or catastrophic divergence of $Q(s, a)$ — thus π — more likely.

The deep Q-network (DQN) was first introduced in Mnih et al. [2013] and then later refined in Mnih et al. [2015]. DQN addresses the first two problems through *experience replay* and the last problem through *parameter freezing*. Although not the first stable method for DRL, DQN was the first to be fast enough for successful use with large neural networks.

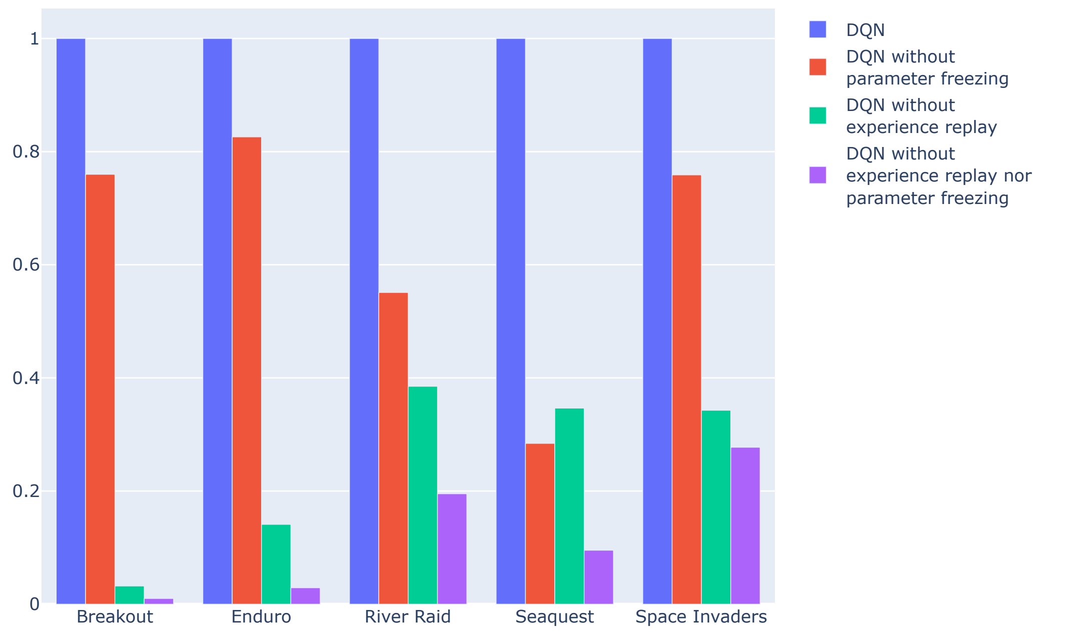

Figure 4.1: For five different Atari games: effect of experience replay and parameter freezing on quality of best agent trained. Raw scores were taken from Mnih et al. [2015] and divided by the DQN score (blue).

4.1.1 Experience replay

In Section 3.4, the next training sample $z_{(t)}$ is the next transition encountered by the agent. This next agent transition $e_{(t)} := \big(s_{(t)}, a_{(t)}, r_{(t+1)}, s_{(t+1)}\big)$ is highly dependent on $e_{(t-1)} := \big(s_{(t-1)}, a_{(t-1)}, r_{(t)}, s_{(t)}\big)$ and the current agent policy $\pi_{(t)}$. As such, this leads to the first two of the three aforementioned instability causes. Moreover, this reduces sample efficiency, as each agent transition is used only once for training.

Experience replay instead stores $e_{(t)}$ in the replay memory ($\mathcal{T}$), where it will be stored for the next η time-steps:

Definition 12 (Replay memory)**.** *Besides an initial warm-up period, the replay memory $\mathcal{T}$ will at any time-step t contain the last η agent transitions, possibly extending over multiple episodes*

$$\mathcal{T}_{(t)} = \{e_{(t-\eta)}, e_{(t-\eta+1)}, \dots, e_{(t-2)}, e_{(t-1)}\}$$

Whenever we want to train our estimator, we sample its next input $z_{(k)}$ uniformly at random from $\mathcal{T}$. This separates sampling and experiencing, hence the need to replace t with k in our subscripts, as there no longer needs to be any connection between our current time-step and how many times we have updated $\boldsymbol{\mathcal{W}}$. This also means that we are free to make many updates between each time-step of the agent, thus reusing most transitions many times. Hence, sample efficiency can be increased, reducing the waste of rare or costly data [Lin, 1992].

An important assumption is, however, that the underlying MDP is mostly unchanged during the duration of η, as the older transitions in $\mathcal{T}$ could otherwise become irrelevant or misleading [Lin, 1993], potentially harming performance [Lin, 1992].

Although experience replay makes $e_{(t)}$ no less dependent on $e_{(t-1)}$ and π, it makes $z_{(k)}$ drastically less dependent on $z_{(k-1)}$ and $\pi_{(k)}$. It thus reduces the variance of the updates and avoids oscillations or divergence in the parameters.

4.1.2 Parameter freezing

In RL with AVFA, TD-targets, and $z_{(k)} = (s, a, r, s')$, the target of $Q(s, a; \boldsymbol{\mathcal{W}}_{(k+1)})$ is

$$y_{(k+1)} = u(r, s'; \boldsymbol{\mathcal{W}}_{(k)}) = r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\mathcal{W}}_{(k)}) \tag{4.1}$$

In practice, an update increasing $Q(s, a)$ often also increases $Q(s', a')$ for all $a' \in \mathcal{A}$, and thus also $u(r, s')$ [Mnih et al., 2015]. Hence, this leads to the last of the three aforementioned causes of instability.

With parameter freezing, a second set of weights $\boldsymbol{\mathcal{W}}^{(-)}_{(k)}$ is used for the targets and is only updated on every κ-th update of $\boldsymbol{\mathcal{W}}$, when $\boldsymbol{\mathcal{W}}^{(-)}_{(k)}$ is made identical to $\boldsymbol{\mathcal{W}}_{(k)}$. The estimates still rely on the estimation network $Q(\cdot, \cdot; \boldsymbol{\mathcal{W}}_{(k)})$, but the targets now rely on the target network $Q(\cdot, \cdot; \boldsymbol{\mathcal{W}}^{(-)}_{(k)})$:

$$y_{(k+1)} := u(r, s'; \boldsymbol{\mathcal{W}}^{(-)}_{(k)}) = r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\mathcal{W}}^{(-)}_{(k)}) \tag{4.2}$$

Thus, an update to the estimates has no impact on y until the next time $\boldsymbol{\mathcal{W}}^{(-)}$ is copied from $\boldsymbol{\mathcal{W}}$. This reduces the correlation between targets and estimates, making oscillations or divergence of the policy much more unlikely. An explanation may be that states more time-steps apart often have more dissimilar feature representations, thus more dissimilar changes to their action-value estimates when network weights change.

4.2 Double Deep Q-Network

This section is a paraphrasing of parts of van Hasselt et al. [2016].

In a nutshell, Q-learning consists of repeating the following procedure many times:

1. Change action-value function, based on existing policy and action-value function.

2. Change policy to greedily choose the highest-valued state-action for each state.

Thus, the more overestimated an action-value $Q(s, a)$ is, the more likely (s, a) is to be picked by our policy π and included in trajectories from other states, thus propagating to their action-values through bootstrapping. However, by similar logic, the more underestimated $Q(s, a)$ is, the less likely (s, a) is to be included in trajectories, thus the less likely its underestimation is to propagate to other state-action estimates. Hence, when estimation errors are possible, Q-learning will suffer from *overestimation bias*, a bias towards overestimating action-values, which can significantly hurt performance.

Thus, asymptotic estimation errors, whether induced by stochastic transitions or rewards, an insufficiently flexible function approximator, or something else, will induce asymptotic overestimation of action-values. Moreover, Q-learning relies on bootstraps, which are usually initialized to an incorrect value and often become incorrect when the policy changes. Thus, Q-learning will usually be biased towards overestimating action-values, even in the best-case scenario of a deterministic MDP and an AVFA that is flexible enough to fit the true action-value function perfectly.

If all action-values are equally overestimated, the policy — which is what we care about — will remain the same as if no action-values are overestimated, since their differences will be unchanged. Alternatively, if the overestimation is concentrated in unexplored state-action pairs, a form of *optimism in the face of uncertainty*, it can be a boon for exploration. However, the Q-learning overestimations only occur after a state-action pair has been visited, and thus give no overestimation in the case of total uncertainty. Moreover, in practice, overestimation errors will differ for different states and actions. As such, in practice, overestimation bias harms Q-learning and the policies it learns.

To address this, Hasselt [2010] introduced *Double Q-learning*, in which estimation of action-values and choice of policy are decoupled.

In Q-learning, all agent transitions are used to improve q, with $\pi(s, \cdot)$ then created from $q(s, \cdot)$ and evaluated on $q(s, \cdot)$:

$$\begin{aligned} q(s,a) &\leftarrow (1-\alpha)\, q(s,a) + \alpha \left[r + \gamma \cdot \max_{a' \in \mathcal{A}} q(s', a') \right] \\ &= (1-\alpha)\, q(s,a) + \alpha \left[r + \gamma \cdot q(s', \arg\max_{a' \in \mathcal{A}} q(s', a')) \right] \end{aligned} \tag{4.3}$$

In Double Q-learning, we replace the single action-value function $q(\cdot, \cdot)$ with the two action-value functions $q^{(1)}$ and $q^{(2)}$. Each of these generate their own policy, but rather than evaluating their own policies, the evaluate each other's:

$$q^{(1)}(s,a) = (1-\alpha)\, q^{(1)}(s,a) + \alpha \left[r + \gamma \cdot q^{(2)}(s', \arg\max_{a' \in \mathcal{A}} q^{(1)}(s',a')) \right] \tag{4.4}$$

$$q^{(2)}(s,a) = (1-\alpha)\, q^{(2)}(s,a) + \alpha \left[r + \gamma \cdot q^{(1)}(s', \arg\max_{a' \in \mathcal{A}} q^{(2)}(s',a')) \right] \tag{4.5}$$

Every transition encountered by the agent is randomly assigned to either (4.4) or (4.5), and we arbitrarily choose the policy of $q^{(1)}$ as our output policy.

This separation effectively makes Double Q-learning unbiased. A likely explanation is that the cross-covariance between $q^{(1)}(s,\cdot)$ and $q^{(2)}(s,\cdot)$ is $\forall s$ nearly zero, ensuring that overestimations and underestimations on average cancel.

We would like to combine Double Q-learning and AVFA. We train separate weights for the two resulting function approximators, but make them share the same network architecture:

- $Q(\cdot,\cdot;\boldsymbol{\mathcal{W}})$ approximates $q^{(1)}$.
- $Q(\cdot,\cdot;\boldsymbol{\mathcal{U}})$ approximates $q^{(2)}$.

Then, when combined with a function-approximator, the only difference between standard Q-learning and Double Q-learning is that the targets

$$y^{(Q)} = r + \gamma \cdot \max_{a' \in \mathcal{A}} Q(s',a';\boldsymbol{\mathcal{W}}) = r + \gamma \cdot Q(s', \arg\max_{a' \in \mathcal{A}} Q(s',a';\boldsymbol{\mathcal{W}}); \boldsymbol{\mathcal{W}}) \tag{4.6}$$

are replaced by

$$y^{(DQ)} := r + \gamma \cdot Q(s', \arg\max_{a' \in \mathcal{A}} Q(s',a';\boldsymbol{\mathcal{U}}); \boldsymbol{\mathcal{W}}) \tag{4.7}$$

However, although the approximators $Q(\cdot,\cdot;\boldsymbol{\mathcal{W}})$ and $Q(\cdot,\cdot;\boldsymbol{\mathcal{U}})$ share the same network architecture, this approach requires their weights $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{U}}$ to be trained separately. As such, when function approximating, replacing Q-learning with Double Q-learning results in twice as many weights having to be trained.

To avoid this problem while still eliminating the overestimation bias of DQN, van Hasselt et al. [2016] introduced the Double Deep Q-Network (DDQN). DDQN builds on DQN, with the only difference being that the targets

$$y^{(DQN)} = r + \gamma \cdot Q(s', \arg\max_{a' \in \mathcal{A}} Q(s',a';\boldsymbol{\mathcal{W}}^{(-)}); \boldsymbol{\mathcal{W}}^{(-)}) \tag{4.8}$$

are replaced by

$$y^{(DDQN)} := r + \gamma \cdot Q(s', \arg\max_{a' \in \mathcal{A}} Q(s',a';\boldsymbol{\mathcal{W}}); \boldsymbol{\mathcal{W}}^{(-)}) \tag{4.9}$$

As such, no extra weights require training, as $\boldsymbol{\mathcal{W}}^{(-)}$ is just copied from $\boldsymbol{\mathcal{W}}$ at regular intervals.

Empirical results show DDQN drastically improving on the performance of DQN. An explanation may be that the correlation between $Q(s,a;\boldsymbol{\mathcal{W}})$ and $Q(s,a;\boldsymbol{\mathcal{W}}^{(-)})$ is typically small enough for the overestimation bias to be drastically reduced; and that increases and decreases in $Q(s,a;\boldsymbol{\mathcal{W}})$ generally have similar effects on $y^{\{DDQN\}}$, causing (4.9) to retain the decorrelating effects of parameter freezing.

4.3 Dueling Deep Q-Network

This section is a paraphrasing of parts of Wang et al. [2016].

For many states, all action-values $q(s, \cdot)$ will be concentrated in some small interval I around the state's value $v(s)$. $v(s)$ can be orders of magnitude larger than the span of I. An example is when all actions yield good trajectories, with lots of expected cumulative reward, but some actions yield even better trajectories than others. A more extreme example is states where it does not matter which action is chosen, causing all action-values of that state to be equal: $\forall a \in \mathcal{A} : q(s, a) = v(s)$.

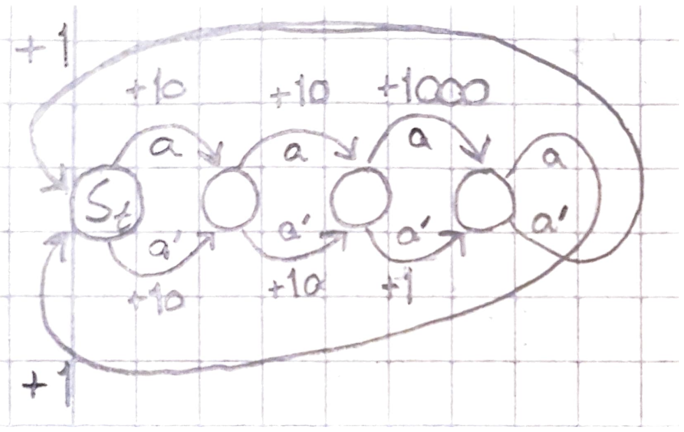

Figure 4.2: With circles as states and a, a' as actions, it does not matter which action we take in state $s_{(t)}$. Thus, $q^{(\pi)}(s_{(t)}, a) = v^{(\pi)}(s_{(t)}) = q^{(\pi)}(s_{(t)}, a')$.

For such a state, DQN and DDQN naively learn all its $|\mathcal{A}|$ action-values from zero, rather than taking advantage of the shared value-function. This causes two problems:

- Learning the action-values takes more time and data than necessary. Q-learning, thus DQN and DDQN, relies on bootstraps, propagating estimated action-values between states. As such, even if the action taken in s' does not matter, estimating $q(s', \cdot)$ correctly is important when choosing an action in s, where the action may very well matter. Thus, slow learning of action-values slows down policy learning.

- As the difference between action-values is so small compared to their magnitude, these difference can easily be overshadowed by noise in the updates, causing random and abrupt changes to the policy.

Wang et al. [2016] solved both of these problems with the Dueling Deep Q-network (Dueling DQN). Dueling DQN is like DDQN, but the last N fully-connected layers, culminating in the $|\mathcal{A}|$ action-values of the state, are replaced by two separate and parallel stacks, each consisting of N fully-connected layers.

With s as input to our neural network, one of these stacks culminates in a scalar state value $V(s)$, while the other stack culminates in an $|\mathcal{A}|$-dimensional vector representing $G(s, \cdot)$. $V(s)$ and $G(s, \cdot)$ are function approximations of, respectively, $v(s)$ and $g(s, \cdot)$:

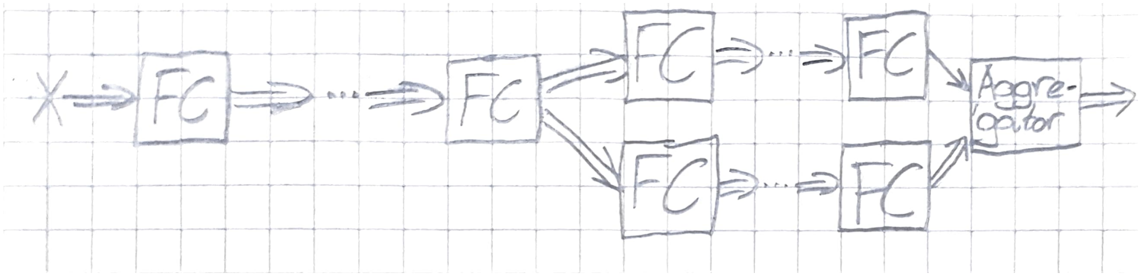

Figure 4.3: The Dueling DQN architecture.

Definition 13 (Advantage function). *The advantage function of a state-action pair tells us how much more cumulative reward we from state s can expect when choosing action a, rather than following policy π:*

$$g^{(\pi)}(s,a) = q^{(\pi)}(s,a) - v^{(\pi)}(s) \tag{4.10}$$

This value estimate and these advantage estimates are then combined in a final aggregating layer:

$$\begin{aligned} Q^{(Dueling)}(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(2)},\boldsymbol{\mathcal{W}}^{(3)}) &:= V(s;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(2)}) \\ &+ \underbrace{\left(G(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)}) - \frac{1}{|\mathcal{A}|}\sum_{a'\in\mathcal{A}} G(s,a';\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})\right)}_{=:\psi(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})} \end{aligned} \tag{4.11}$$

It may be tempting to rather suggest the following as our aggregating layer:

$$V(s;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(2)}) + G(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)}) \tag{4.12}$$

However, if we use (4.12), we become unable to determine V and G, given $Q^{(Dueling)}$, as $V + G = (V - \phi) + (G + \phi)$ for any ϕ constant in a. This lack of identifiability is mirrored by (4.12) performing poorly in practice. An explanation may be that, for optimizing loss, (V, G) has no reason to favor convergence to (v, g) over convergence to $(0, q)$, or any other $(v - \phi, g + \phi)$, thus possibly leaving the aforementioned benefits of separately estimating v and g untapped.

The following aggregating layer solves that:

$$V(s;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(2)}) + \underbrace{\left(G(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)}) - \max_{a'\in\mathcal{A}} G(s,a';\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})\right)}_{=:\varphi(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})} \tag{4.13}$$

Here, $-\max_{a'\in\mathcal{A}} G(s,a';\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})$ cancels any constant added to $G(s,\cdot)$, thus ensuring that $\varphi(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})$ is zero for the action chosen by the greedy policy under evaluation. By making it impossible to add constants to $\varphi(s,\cdot)$, this makes (V,φ) uniquely favor convergence to (v, g).

(4.11) offers a simpler and more stable alternative to (4.13). By replacing max with mean, advantages $G(s,\cdot)$ only need to compensate for their mean, rather than keeping up with changes to a singular $G(s,a)$. Hence, stability is generally increased.

In (4.11), $-\frac{1}{|\mathcal{A}|}\sum_{a'\in\mathcal{A}} G(s,a';\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})$ fails to ensure that $\psi(s,a;\boldsymbol{\mathcal{W}}^{(1)},\boldsymbol{\mathcal{W}}^{(3)})$ is zero at the action of the greedy policy we are evaluating, thus violating Definition 13. However, it still cancels any extra offset added to $G(s,\cdot)$, thus keeping $\psi(s,\cdot)$ concentrated around zero. Hence, although it violates Definition 13, (4.11) solves the problems that we in this section looked to Definition 13 to solve.

Therefore, although it makes (V,ψ) favor convergence to some $(v-\phi, g+\phi)$, (4.11) still becomes the aggregating layer of choice.

4.4 Prioritized experience replay

This section is a paraphrasing of parts of Schaul et al. [2016].

Standard, uniform experience replay, as used in DQN, treats all transitions the same. However, an RL agent can learn more effectively from some transitions than others, as a transition may be more or less surprising, redundant, task-relevant, or up-to-date. As such, a replay mechanism that takes these differences into account, while correctly dealing with resulting bias and other side-effects, can lead to better overall performance. In practice, prioritized experience replay (PER) significantly improved the training speed and top performance of both DQN, DDQN, and Dueling DQN [Schaul et al., 2016; Wang et al., 2016].

Imagine we had an oracle that for each of our stored, thus last η, transitions could tell us how much sampling that transition next, and performing an NN update from it, would reduce $\mathcal{L}$, the total estimation loss over all state-action pairs. Then, we could always sample the transition leading to the highest loss reduction, rather than uniformly at random, in some cases obtaining exponential speed-ups. However, such an oracle is unrealistic, forcing us to find computation-friendly, approximate substitutes.

4.4.1 Prioritizing with TD-error

For the DRL networks we have looked at in this chapter, every update can be decomposed into computing the TD-error of a sample and making the weights update a multiple of this:

$$\Delta\boldsymbol{\mathcal{W}} = \alpha \underbrace{\left[y - Q(s,a;\boldsymbol{\mathcal{W}})\right]}_{\text{TD-error}} \nabla Q(s,a;\cdot) \tag{4.14}$$

As such, for every transition e already sampled, we can access its last TD-error $\delta^{(e)}$ without any extra computation. Although imperfectly, $\delta^{(e)}$ can serve as somewhat of a proxy for the oracle, as it quantifies how surprising e was, thus loosely capturing how much we can learn from it. $\delta^{(e)}$ will not tell us how much updating the approximator with transition e will reduce $\mathcal{L}$. Moreover, every time the estimation network and target network change, all stored δ become ever-more outdated. However, as we get these TD-errors for free and their magnitude has some connection with how much they can teach us, prioritizing based on them seems reasonable.

Specifically, when an agent encounters a transition e, this is stored in the replay memory, together with an infinite TD-error ($\delta^{(e)} \leftarrow \infty$). Then, when sampling from the replay memory, the transition with highest absolute TD-error ($\arg\max_{e\in\mathcal{T}} |\delta^{(e)}|$) is chosen, participating in an NN update and having its newly calculated TD-error

($\delta^{(e,new)}$) replace its old TD-error ($\delta^{(e)}$) in the replay memory. Although struggling against the oracle, this prioritization scheme can significantly outperform uniform sampling.

4.4.2 Stochastic prioritization

Although better than uniform sampling, the greediness of this approach causes some problems. First, $\delta^{(e)}$ is only updated when e is sampled. Thus, once e receives a low $|\delta^{(e,new)}|$, it will effectively never be retried, even if its true TD-error later increases. This causes an increased risk of overfitting, as sampling will mostly be constrained to a small subset $\mathcal{B} \subset \mathcal{T}$, where δ of transitions in $\mathcal{B}$ take many iterations to decrease, while δ of transitions in $\mathcal{T} \backslash \mathcal{B}$ are never updated.

Second, our sampling becomes sensitive to noise spikes: a transition with unusually high TD-error will be prioritized, even if noise was the cause. Bootstrapping further worsens this, as estimation errors on other states appears as another source of noise.

To address both of these issues, Schaul et al. [2016] goes on to introduce *Stochastic prioritization*. Rather than greedily sampling the transition with the highest $|\delta^{(e)}|$, stochastic prioritization monotonously increases sampling probability as a function of $|\delta^{(e)}|$, but gives every transition a non-zero sampling probability. Concretely, every time stochastic prioritization is to sample a transition from $\mathcal{T}$, transition e is sampled with probability

$$P^{(e)} := \frac{(p^{(e)})^{\chi}}{\sum_{j \in \mathcal{T}} (p^{(j)})^{\chi}}, \; p^{(j)} > 0, \; \chi \geq 0. \tag{4.15}$$

χ is a hyperparameter determining the strength of prioritization: $\chi = 0$ gives uniform sampling, $\chi \to \infty$ gives greedy sampling by TD-error, and $\chi \in \mathbb{R}_+$ gives a bit of both. $p^{(e)}$ is a positive, monotonically increasing function of $|\delta^{(e)}|$. Examples of such functions are:

$$p^{(e)} = |\delta^{(e)}| + \varepsilon, \; 0 < \varepsilon << 1, \tag{4.16}$$

$$p^{(e)} = \frac{1}{\text{rank}(e)}, \tag{4.17}$$

where rank(e) is 1 for the transition with largest $|\delta^{(e)}|$, 2 for the transition with second largest $|\delta^{(e)}|$, and so on.

4.4.3 Debiasing

By no longer sampling uniformly, the SGD updates become biased estimates of BGD updates, causing us to lose the guarantee of converge in expectation to a local minimum. We would thus like to reduce this bias, while sticking to our non-uniform sampling. We do this through weighted importance sampling. Then, for each $e \in \mathcal{T}$, we generate an importance sampling weight

$$u^{(e)} = \left(\frac{1}{\eta} \cdot \frac{1}{P^{(e)}}\right)^{\beta}, \; \beta \in [0, 1], \tag{4.18}$$

where β determines how strongly we should be debiasing: $\beta = 0$ leaves bias unchanged, $\beta = 1$ entirely cancels the bias, and $\beta \in (0, 1)$ partially cancels the bias.

Debiasing is typically most important near convergence, thus at the end of training. As such, a schedule is used for β, where β starts at the hyperparameter $\beta_0 \in [0, 1]$ and grows linearly until it reaches 1.

The SGD update for transition e is then multiplied by

$$\frac{u^{(e)}}{\max_{j \in \mathcal{T}} u^{(j)}}, \tag{4.19}$$

where the numerator helps stability by ensuring our importance sampling never upscales updates.

Thus, this SGD update becomes

$$\Delta \boldsymbol{\mathcal{W}} = \frac{u^{(e)}}{\max_{j \in \mathcal{T}} u^{(j)}} \cdot \alpha \Big[y - Q(s, a; \boldsymbol{\mathcal{W}}) \Big] \nabla Q(s, a; \cdot) \tag{4.20}$$

In non-linear function approximators, the MSE loss function is non-linear. Thus, a gradient descent update changes $\boldsymbol{\mathcal{W}}$ in the direction minimizing a linear approximation of a non-linear function. As such an approximation can only be assumed to be "good" locally, large BGD or SGD updates can be problematic, and should generally be avoided. By downscaling updates relative to their magnitude, (4.20) reduces the magnitude of large — possibly disruptive — updates, while mostly retaining the magnitude of updates that are already small enough. Therefore, (4.20) works especially well with SGD for non-linear function approximators, such as the neural networks used in DRL.

5

Real-time topology optimization

This chapter is a quick summary of some relevant parts of von Meier [2006], and assumes basic familiarity with the physics of electromagnetism and electric circuits. The reader lacking this familiarity or seeking a more comprehensive review is referred to that book.

The overarching goal of the electricity grid is to provide electricity reliably — the availability of high-quality electricity is maximized, safely — the risk of fires and electrocutions is minimized, and efficiently — power is produced and delivered at minimal total costs. These overarching goals entail many sub-goals for a networked transmission grid, ranging from good-to-haves to absolute necessities. Of all these sub-goals, the ones we hope to address with RTTO are: minimizing total line losses, respecting thermal limits, ensuring reasonable security, keeping loads connected, and avoiding costly actions.

A transmission grid transfers AC power at high voltages from *generators* to *loads* and is typically arranged in a network structure. A generator is an electricity producer, such as a hydropower plant, while a load is an electricity consumer, such as a city. Generation and load must always be balanced: the sum of electricity produced by all generators must equal the sum of electricity consumed by all loads and dissipated during transmission. In the network structure, every line is redundant, as all pairs of points are connected in multiple closed loops.

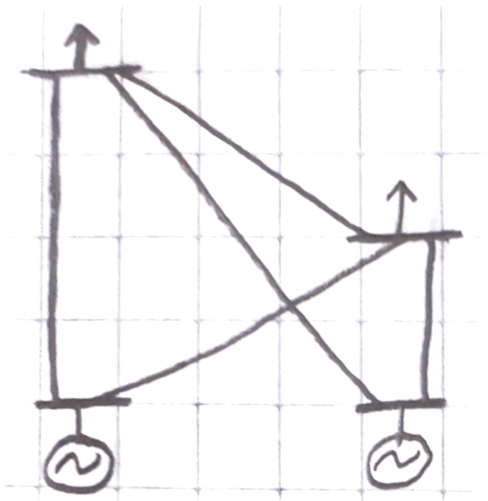

Figure 5.1: A single-line diagram of a simple, networked AC transmission grid with two loads, two generators, and four substations.

Transmission lines can be connected by a busbar: a conductor to which multiple lines can be connected, which thus transmits electricity between them. Geographically, lines meet at a substation, where a substation can contain one or more such busbars. Thus, depending on which busbar each line is connected to, lines connected to the same substation need not be connected to each other at that substation.

Being a normal conductor, a transmission line has some resistance R. Thus, sending a current I through a transmission line causes the line loss $P = I^2R$, through resistive heating of the line. Beyond wasting energy, this can also violate the *thermal limit* of a line: the maximum temperature at which it can safely operate. A line is *overloaded* if it transmits a current beyond its *line rating*. For an overloaded line, resistive heating — proportional to its line loss — may push its temperature beyond its thermal limit. Such an overheated line will stretch and start sagging. If it sags too far, the distortion becomes irreversible, and if the line sags too close to other objects, an electric arc and/or short circuit can occur [Kelly et al., 2020]. In the case of extreme heating, the line can melt. It is worth noticing that these problems are directly caused by line temperature, not line current. Thus, there is a difference between how much current a line can sustain for a short time, and how much it can sustain for an indefinite time. Typically, this is reflected by having a *normal rating* and an *emergency rating*. Moreover, as weather determines the ability of the typical overhead line to dissipate heat, the temperature induced by a certain current depends on the weather. Traditionally, lines have been rated for their worst case scenario: hot, sunny weather with no wind. However, in the pursuit of improved asset utilization, it is nowadays common to employ variable line ratings. One such rating is *dynamic line ratings*, which rely on current weather information.

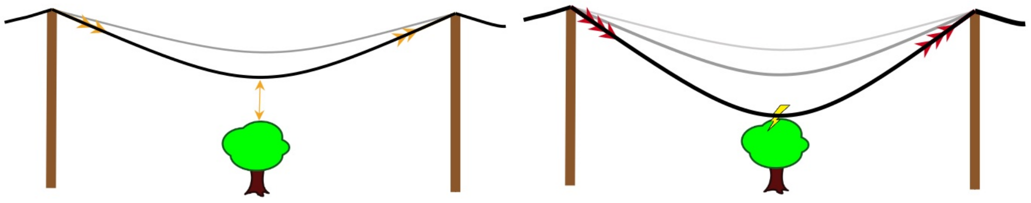
Figure 5.2: The same power line, heated to varying degrees.

If a line short circuits, a circuit breaker will automatically open to disable the line and avoid hazards and equipment damage. When a line is disabled, the same amount of current is still supposed to flow between generators and loads, only with fewer alternative paths. This typically means that other lines have to transport more current, and can thus cause these to eventually overheat, sag, short-circuit, and be disabled. That could in turn cause the story to repeat for other power lines, with one or multiple lines being taken down at a time. This is called a *cascading failure*, and can in the worst case lead to large-scale blackouts, with a large number of loads receiving no electricity. An example of this was the Northeast blackout of 2003.

Contingencies, such as falling trees, extreme weather, and terrorism, can also cause lines to be abruptly and unexpectedly disabled. To minimize the risk of cascading failures, it is thus insufficient to just avoid overloads. Rather, we must also optimize *security*, the measure of how much that can go wrong before service is compromised or equipment is damaged. The security of a grid is determined through *steady-state contingency analysis*, where we simulate what load the various lines of the grid would converge to if some combination of contingencies were to occur. A standard criterion is $N-1$, which states that the system must be secure against any one contingency. To determine whether a system fulfils this criterion, operators typically have a list of "credible contingencies", compiled based on experience, running steady-state contingency analysis for one such contingency at a time. An even stronger criterion is $N-2$, which states that the system must be secure against any combination of two contingencies. However, due to the computational cost of AC load-flow simulation, $N-2$ contingency analysis is not performed in day-to-day operations on for example the French national grid [Donnot et al., 2018].

If necessary, the operator can *shed* load by disconnecting it from the grid. In principle, by reducing the total current transmitted from generators to loads, load shedding can reduce line losses, resolve overloads, and improve security. However, load shedding has a directly negative impact on reliability, and is thus something we would like to avoid. As load and generation fluctuate throughout the day, the question becomes how we can address these sub-goals in real-time, without resorting to load shedding. This time perspective does not allow making any geographic changes to the grid. However, it does allow *re-dispatching* generation and utilizing *switchgear*.

Re-dispatching refers to changing how the total generation is distributed among the generators. As this redistributes how much current runs through each generator, it changes how much current is run through each line. However, this is typically expensive. Switchgear allows us to cheaply and near-instantaneously connect or disconnect things from each other. A line is connected to a busbar with switchgear, and busbars in a substation are connected to each other with switchgear. Switchgear then gives us full control of which lines to keep active and how the lines of a substation are connected together. Thus, we can utilize the switchgear to cheaply and near-instantaneously change which paths current can take, and thus how it ends up flowing. This is what we mean by *real-time power grid topology optimization*. Both re-dispatching and RTTO can cause reductions of line losses, and hence also resolve overloads, if any of the following are achieved:

- Currents are made to flow in opposite directions along a line, and thus cancel each other out. As the net current on that line will be low, so will the line loss be.

- Currents are made to take shorter paths. The currents will then experience less voltage drops — thus line losses — along their paths.

- Currents are more evenly distributed between identical lines. This will reduce the sum of squared line currents, and thus the total line losses.

Moreover, if they achieve that for simulated post-contingency situations, they can improve security. However, while re-dispatching has been heavily researched, the same does not hold for RTTO [Donnot, 2020], which is currently under-exploited [Marot et al., 2020]. Part of the reason may be that the non-linear and combinatorial nature of RTTO makes it difficult at scale for traditional methods [Marot et al., 2020].

Part II

DRL for RTTO

6

Case study

This chapter will offer a case study into the submission of Geirina for the simulated RTTO competition L2RPN 2019. The competition lasted for six weeks and contained 102 contestants, of which the submission of Geirina placed first. For brevity, L2RPN 2019 will be referred to as *L2RPN*.

6.1 Challenge design

Except where otherwise stated, this section paraphrases parts of Marot et al. [2020] and the references therein.

6.1.1 Simulator

L2RPN was simulation-based. More specifically, all three phases — training, evaluating, testing — were performed in the AC grid simulator *pypownet*. The simulated environment was based on the *IEEE 14-bus system*, with some modifications and additions. The resulting grid consisted of 14 substations with two busbars each, 20 lines, 11 loads, and 5 generators: nuclear, themal, wind, large solar, and small solar. Each line was either disconnected entirely or connected to exactly one busbar on each end.

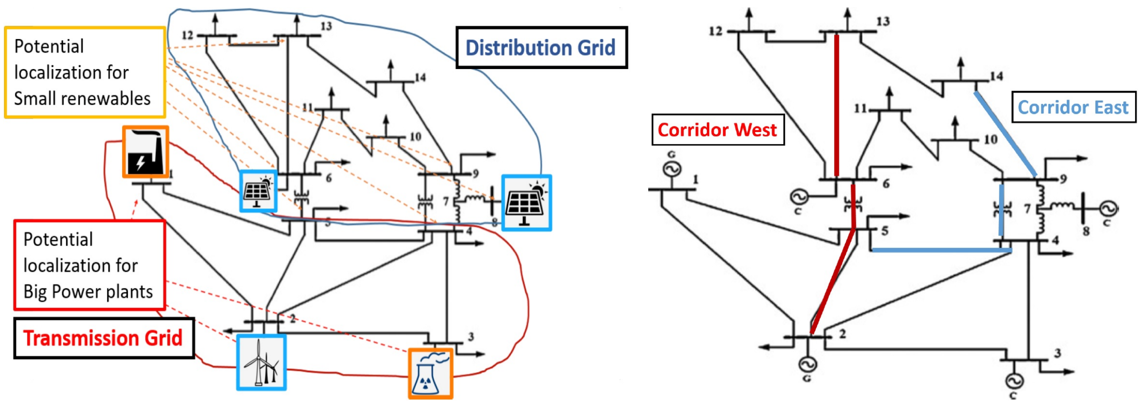
Potential localization for Small renewables
Distribution Grid
Potential localization for Big Power plants
Transmission Grid
Corridor West
Corridor East

Figure 6.1: Two ways to view the power grid used in the L2RPN 2019 competition.

The simulator only considered steady-state flows. At every time-step t, the simulator would give the agent an observation $o_{(t)}$, let it choose an action $a_{(t)}$, recompute load flows given the new topology, compute an immediate reward $r_{(t+1)}$

based on the new load flow [Lerousseau, 2018], transition to the grid state 5 minutes in the future ($s_{(t+1)}$), and let the observation $o_{(t+1)}$ of that state be the observation for the next time-step. Thus, the granularity of each time-step was five minutes, and the task could be formulated as a POMDP, with

- $\mathcal{A}$ consisting of the following actions:
  - One do-nothing action.
  - For each line: one switching action that would entirely disconnect or re-connect that line.
  - For each substation: as many reconfiguration actions as that substation had configurations, in terms of which busbar each line was connected to.
  - Composite actions, each consisting of one one substation reconfiguration action and one line switching action.
- $\mathcal{S}$ consisting of tuples $(\psi, \boldsymbol{\zeta}, \boldsymbol{\rho}, \boldsymbol{\xi})$, where:
  - ψ was the hidden part of the environment state.
  - $\boldsymbol{\zeta}$ was the current values for all generators and loads. $\boldsymbol{\zeta}$ was fully observable by the agent, and entirely independent of past and present actions of the agent.
  - $\boldsymbol{\rho}$ was the topology of the grid, which is what the agent directly controlled, and which was also fully observable by the agent.
  - $\boldsymbol{\xi}$ was the line flows, computed by AC load flow equations. $\boldsymbol{\xi}$ was fully observable by the agent and entirely determined by $\boldsymbol{\zeta}$ and $\boldsymbol{\rho}$.
- $\mathcal{O}$ consisting of tuples $(\boldsymbol{\zeta}, \hat{\boldsymbol{\zeta}}, \boldsymbol{\rho}, \boldsymbol{\xi})$, where $\hat{\boldsymbol{\zeta}}$ was a Gaussian random variable whose expectation was $\boldsymbol{\zeta}$ at the next time-step and whose variance was 5% of this expectation.

The environment was subject to some constraints. Based on thermal limits, each line had some constant, known, line rating. A *hard overload* occurred if a line was loaded at, or beyond, 150% of its rating, and would cause the line to disconnect immediately. A *soft overload* occurred if a line was loaded in the range $(100, 150)\%$ of its rating, and would cause the line to disconnect if left unresolved for two time-steps (10 minutes).

Moreover, there were *soft constraints* on the actions. Taking an action that violated a soft constraint would result in that action being replaced by a do-nothing action, and the grid topology thus remaining unchanged. The soft constraints were:

- A line that was disconnected due to an overload must remain disconnected for 10 time-steps (50 minutes).
- If an action is taken on a line or substation, no other action must be taken on that same line or substation for 3 time-steps (15 minutes).
- An action cannot cause the AC load flow computation to diverge.

Finally, there were some *hard constraints* on the topology of the grid. If a hard constraint was violated at any time during a scenario, the scenario would immediately terminate. The hard constraints were:

- All loads must be connected to the grid.
- At most one generator can be disconnected from the grid.
- There shall be no power islands: current must be able to travel between any two parts of the grid.

An *easy mode* and a *soft mode* were provided. In easy mode, overloads would not disconnect lines and soft constraints could be violated. In soft mode, the violation of a hard constraint would cause the grid topology to be reset to its initial state, but the scenario to continue. The intention behind both of these modes was to make it easier for a learning algorithm to make initial progress.

The agent could at any point request a simulation of the immediate consequences of an action, without having to take that action. However, this simulation was DC, thus providing an imperfect estimate of the true AC consequences. Moreover, this had a cost in terms of computational budget, and the computational budget was limited during evaluation and testing.

6.1.2 Scenarios

The scenarios were modelled after a typical, French January, both in terms of load, and in terms of the correlation between solar and wind generation. Nuclear was made to be slowly varying, and thermal was made to always ensure load-generation balance.

To analyse the difficulty of various scenarios, the organizers devised three baseline agents:

- Do-nothing (DN) agent: Starts out in the reference topology $\tau^{(ref)}$ and always picks the do-nothing action. $\tau^{(ref)}$ is the fully-connected topology, where all lines are enabled, and all lines entering a substation are connected together by being on the same busbar.
- Single topology ($\mathrm{DN}^{(\tau)}$) agent: Starts out in some $\tau \neq \tau^{(ref)}$ and always picks the do-nothing action.
- Greedy (GR) agent: Starts out in $\tau^{(ref)}$. At every time-step t, simulates the immediate reward of every action $a_{(t)}$ and acts greedily on these simulations.

On average, over all scenarios, the DN agent had at least one line overloaded 3% of the time. However, an ensemble of expertly selected $\mathrm{DN}^{(\tau)}$ agents could resolve 85% of these overloads. Moreover, the GR agent did not perform well on all scenarios. Thus, the scenarios were non-trivial and could not all be treated as a contextual bandit.

There was no difference between participants on which dataset they were given for training and which datasets they were evaluated and tested on. The training dataset consisted of approximately 1000 training scenarios, covering $3-4$ weeks of January.

The validation dataset and the test dataset were unknown to the participants, and each consisted of 10 different scenarios. Although these datasets shared no scenarios, their distributions of scenarios were kept as similar as possible. All of these scenarios were also set in January, but with a duration of $1-3$ days. They were chosen with a diversity in difficulty, task, weekday, and duration. More specifically for each of these datasets:

- Two scenarios were completed by the DN agent. In one of these, no overload appeared.
- Six scenarios were failed by the DN agent, but completed by the DN$^{(\tau)}$ ensemble.
- Two scenarios were failed by both the DN agent and the DN$^{(\tau)}$ ensemble.

6.1.3 Scoring

For evaluation and testing, an agent was scored according to the following formulas, here displayed in the case of testing:

$$StepScore_{(t)}^{(s)} = \begin{cases} 0, & \text{if soft constraint is violated} \\ \sum_{l=1}^{LineCount} \max(0, 1 - \left(\frac{LineCurrent_{(t)}^{(sl)}}{LineCurrentLimit^{(l)}}\right)^2), & \text{otherwise} \end{cases} \tag{6.1}$$

$$ScenarioScore^{(s)} = \begin{cases} 0, & \text{if hard constraint was violated} \\ \sum_{t=1}^{ScenarioLength^{(s)}} StepScore_{(t)}^{(s)}, & \text{otherwise} \end{cases} \tag{6.2}$$

$$TotalScore = \sum_{s=1}^{TestScenarioCount} ScenarioScore^{(s)} \tag{6.3}$$

$LineCurrent_{(t)}^{(sl)}$ was obtained from $\boldsymbol{\xi}$ and was the current running through line l, at time-step t, in test scenario s. $LineCurrentLimit^{(l)}$ was the line-specific rating of line l, constant over all time-steps and all scenarios. It was determined by the organizers beforehand, and known by all participants. $LineCount = 20$ was the number of lines in the grid. $ScenarioLength^{(s)} \in [288, 864]$ was the number of five-minute intervals in test scenario s, where the length of scenarios varied from one to three days. $TestScenarioCount = 10$ was the number of scenarios the agent was tested on.

6.2 Submission design

Except where otherwise stated, this section paraphrases Lan et al. [2020].

The agent of Geirina was a Dueling DQN with ReLU activation layers, trained with PER.

To further balance the data distribution in memory, transitions directly leading to *game over* were oversampled: any encountered game-over transition was given multiple copies in replay memory.

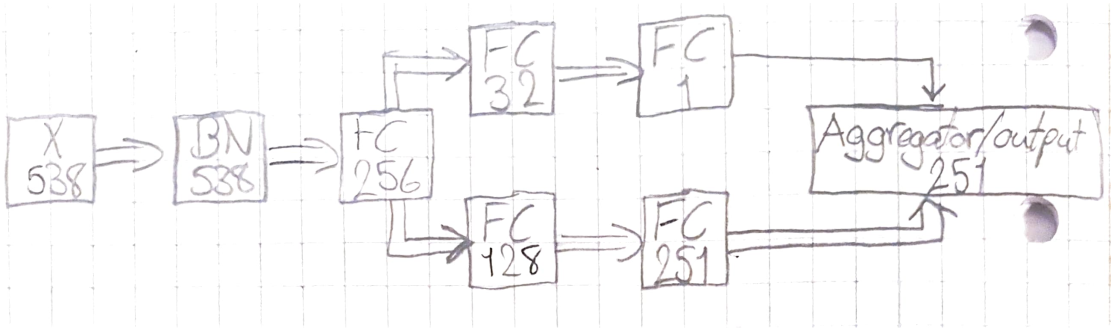

Figure 6.2: The Dueling DQN architecture of Geirina. BN is here short for batch normalization [Ioffe and Szegedy, 2015].

For their immediate reward function, Geirina used a modified version of (6.1), found in Shi [2019]:

$$R_{t+1} = \begin{cases} -1, & \text{if } A_t \text{ caused a line overload or violated a constraint} \\ \frac{1}{20}\sum_{l=1}^{20} \max(0, 1 - \left(\frac{LineCurrent_{(t)}^{(sl)}}{LineCurrentLimit^{(l)}}\right)^2), & \text{otherwise} \end{cases} \tag{6.4}$$

Moreover, *imitation learning*, a form of supervised learning, was used on simulated data to generate an initial policy. This greatly reduced the training time and made the agent less vulnerable to poor local minima.

However, owing to the large action space, this would still give unsatisfactory performance. To address that, Geirina first used domain knowledge and simulation to tease out the most relevant actions and reduce the action space to just 251 actions. This reduced action space consisted of 155 substation reconfiguration actions, 19 line switching actions, 76 composite actions, and one do-nothing action.

However, this simplified POMDP was still unsuitable for ε-greedy exploration. The action space was too large and the agent would easily fall into local minima. To address this, Geirina devised a guided exploration method that compared with ε-greedy exploration achieved comparable and more stable performance with an order of magnitude less experience. This guided exploration was an interplay between the action-value estimator and the DC simulator. Specifically, the agent policy chose its next action in the following way:

1. The action-value estimator estimates the action-value of all actions.
2. The immediate rewards of the 10 actions with the highest action-value estimates are simulated.
3. Of these 10 actions, the action whose simulated immediate reward was highest is chosen.

The idea is that this explores actions that the action-value estimator may not have believed to be the absolute best, while limiting us to promising actions. Thus, the knowledge available from the immediate reward simulator is utilized to spend less experience on exploring obviously bad actions.

The resulting action-value estimator was still imperfect, sometimes providing bad actions in complex system states. As bad actions could have serious adverse effects, reducing the occurrence of such actions was of high priority. The solution of Geirina was to add another line of defence, by always picking the next action through an early warning system, similar to how grid operators typically operate [Marot et al., 2020]:

1. Simulate do-nothing action
2. If there is some line l where $LineCurrent^{(l)} > \lambda \cdot LineCurrentLimit^{(l)}$, use the aforementioned guided exploration method to pick the next action. Otherwise, pick the do-nothing action.

Thus, the role of reinforcement learning shrunk to short-listing 10 actions for a GR agent to pick between. More importantly, this meant that obviously bad actions would effectively be weeded out by the DC simulator. The result was improved performance. $\lambda = 0.885$ was a tuned hyperparameter.

6.3 Result

This paragraph is a paraphrasing of Marot et al. [2020] and the references therein. The implementation of Geirina placed first among 102 participants, as one of two submissions to complete all scenarios. When it was validated against a larger batch of month-long training scenarios, it did however experience five game overs. On this same batch of scenarios, the submission that otherwise placed second experienced 11 game overs.

Due to its early warning system, the agent of Geirina only tried to resolve and avoid overflows, ignoring the task of continuously minimizing power loss. Thus, there may be a significant optimization gap between this agent and a perfect agent. Moreover, as L2RPN contained neither maintenance nor contingencies, security was not taken into consideration. Additionally, a real grid offers more constraints, more dependencies, more uncertainty, more diversity, larger scale, less perfection, and all-around more complexity, than what does pypownet. It remains to be seen whether DRL agents can bridge the simulation-reality gap and perform well on real grids.

7

Discussion

Although DRL has shown impressive results in various domains, that does not necessitate that it should be suitable for RTTO. However, it does mean there is a possibility. Moreover, some attributes of DRL makes it especially promising:

First, we have to look at the state and action space. For RTTO with interesting actions at interesting scale, traditional approaches are too slow [Marot et al., 2020]. This points to DL, as it has shown an impressive ability to efficiently make sense of complex, high-dimensional data. Donnot et al. [2018] further backs up this idea, by successfully predicting load flow with a DNN. This gives reason to believe that a DNN can build internal representations of the power flow, and thus the impact of various actions. If so, this would mean that an agent relying on a DNN could understand the immediate consequences of its actions. The remaining challenge of the optimization task would then be to learn which sequences of consequences are preferable; this is something RL can do.

Second, we have to look at the data. SL would require an extensive, labeled dataset, mapping state to operator action. We do not have such a dataset [Lerousseau, 2018]. Moreover, as SL would just learn to imitate human operators, it would limit us to human-level performance [Lerousseau, 2018], and we would need the dataset to evolve as load, generation, and grid undergo the rapid changes forecast. By making some assumptions about a grid, we can model it in a physics simulator [Lerousseau, 2018]. This allows quickly and cheaply generation unlimited amounts of training data for DRL. Thus, data-wise, this problem is ideal for DRL. That however assumes that the simulation is close enough to reality for DRL to generalize across the simulation-reality gap. Whether this is the case remains to be seen.

Finally, we have to look at the overall nature of the task. RTTO requires sequential decision-making over a complex, high-dimensional, partially observable, somewhat unpredictable system. Owing to its great flexibility, DRL can tolerate all of these complicating factors. In other branches of operational control, this has already manifested itself through DRL handling some unpredictable emergencies that most traditional methods cannot [Zhang et al., 2020]. This same flexibility also allows for combining DRL with many classical control methods, thus achieving even better results [Zhang et al., 2020].

8

Conclusion

RTTO can play a meaningful role in making decarbonization of society feasible, as it is cheap, fast, under-exploited, and can resolve line overcurrents. DRL is a promising approach for RTTO, as it is flexible, can predict load flow, can be combined with traditional methods, and power grids can be simulated.

DL is the process of fitting a deep ANN: a stack of alternating affine functions and non-affine activation functions, which can be fit to some true function by BGD, SGD, or more complex SGD-based optimizers. Q-learning is the result of combining RTVI with more cautious, stochastic updates, where RTVI is a form of AVI, and AVI is an optimization on SVI.

DQN successfully merges DL and RL, by using an ANN as action-value function approximator, and updating this ANN through a mixture of Q-learning and gradient descent. What makes DQN succeed where past approaches have failed is its utilization of parameter freezing and uniform experience replay. These increase its sample efficiency, while reducing the variance of its estimates.

Moreover, some of the weaknesses of DQN are addressed by DDQN, Dueling DQN, and prioritized experience replay. DDQN removes its overestimation bias, Dueling DQN handles shared action-value offsets better, while prioritized experience replay ensures more relevant experiences are chosen for replay.

By pre-training a Dueling DQN with imitation learning, then training it with prioritized experience replay, and finally employing it with some typical grid operation heuristics, good performance can be achieved in an RTTO simulator. Whether this performance can bridge the simulation-reality gap remains to be seen.

Bibliography

Bacher, R. and Glavitsch, H. (1986). Network topology optimization with security constraints. *IEEE Transactions on Power Systems*, 1(4):103–111.

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., and Dahl, G. E. (2020). On empirical comparisons of optimizers for deep learning. arXiv:1910.05446 [cs.LG].

Cole, S., Vrana, T., Curis, J.-B., Liu, C.-C., Karoui, K., Fosso, O., and Denis, A.-M. (2011). A european supergrid: Present state and future challenges. In *PSCC Stockholm*.

Colle, S. (2020). How net-zero emissions present the power sector with an opportunity. `https://www.ey.com/en_gl/power-utilities/how-net-zero-emissions-present-the-power-sector-with-an-opportunity`. Retrieved January 2021.

Donnot, B. (2020). Power network in a sustainable world. part 2: Rte competition tutorial. In *IEEE BDA Tutorial Series*. IEEE Power & Energy Society.

Donnot, B., Guyon, I., Schoenauer, M., Marot, A., and Panciatici, P. (2018). Fast power system security analysis with guided dropout. In *26th European Symposium on Artificial Neural Networks*, The European Symposium on Artificial Neural Networks, Bruges, Belgium.

Dozat, T. (2016). Incorporating nesterov momentum into adam. In *Workshop Track - ICLR 2016*, International Conference on Learning Representations.

Gray, R. (2017). The biggest energy challenges facing humanity. `https://www.bbc.com/future/article/20170313-the-biggest-energy-challenges-facing-humanity`. Retrieved January 2021.

Hasselt, H. (2010). Double q-learning. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., and Culotta, A., editors, *Advances in Neural Information Processing Systems*, volume 23, pages 2613–2621. Curran Associates, Inc.

Hinton, G., Srivastava, N., and Swersky, K. (2012). rmsprop: Divide the gradient by a running average of its recent magnitude. In *Neural Networks for Machine Learning*. Coursera.

IEA (2017). Energy technology perspectives 2017 – analysis. `https://www.iea.org/reports/energy-technology-perspectives-2017`. Retrieved January 2021.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

IRENA (2017). Adapting market design to high shares of variable renewable energy. https://www.irena.org/publications/2017/May/Adapting-Market-Design-to-High-Shares-of-Variable-Renewable-Energy.

Kelly, A., O'Sullivan, A., de Mars, P., and Marot, A. (2020). Reinforcement learning for electricity network operation. arXiv:2003.07339 [eess.SP].

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, International Conference on Learning Representations.

Kroposki, B. (2017). Integrating high levels of variable renewable energy into electric power systems. *Journal of Modern Power Systems and Clean Energy*, 5(6):831–837.

Lan, T., Duan, J., Zhang, B., Shi, D., Wang, Z., Diao, R., and Zhang, X. (2020). Ai-based autonomous line flow control via topology adjustment for maximizing time-series atcs. In *2020 IEEE Power & Energy Society General Meeting (PESGM)*, pages 1–5.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lerousseau, M. (2018). Design and implementation of an environment for learning to run a power network (l2rpn). https://github.com/MarvinLer/pypownet/blob/master/doc/project_introduction.pdf.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321.

Lin, L.-J. (1993). *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University.

Marot, A., Donnot, B., Romero, C., Donon, B., Lerousseau, M., Veyrin-Forrer, L., and Guyon, I. (2020). Learning to run a power network challenge for training topology controllers. *Electric Power Systems Research*, 189:106635.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop 2013*, Neural Information Processing Systems.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Ng, A. (Year unknown). Machine learning. https://www.coursera.org/learn/machine-learning. Retrieved December 2020.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, International Conference on Learning Representations.

Shi, D. (2019). L2rpn challenge-learning to run a power network through ai. https://l2rpn.chalearn.org/competitions.

Silver, D. (2015). Introduction to reinforcement learning with david silver. https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. Adaptive Computation and Machine Learning. MIT press.

Szandała, T. (2021). Review and comparison of commonly used activation functions for deep neural networks. In Bhoi, A. K., Mallick, P. K., Liu, C.-M., and Balas, V. E., editors, *Bio-inspired Neurocomputing*, volume 903 of *Studies in Computational Intelligence*, pages 203–224. Springer Singapore, Singapore.

Teter, J., Tattini, J., and Petropoulos, A. (2020). Tracking transport 2020 – analysis. https://www.iea.org/reports/tracking-transport-2020. Retrieved January 2021.

United Nations (2020). The race to zero emissions, and why the world depends on it. https://news.un.org/en/story/2020/12/1078612. Retrieved January 2021.

van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence and the Twenty-Eighth Innovative Applications of Artificial Intelligence Conference*, AAAI Conference on Artificial Intelligence, page 2094–2100. AAAI Press.

von Meier, A. (2006). *Electric Power Systems: A Conceptual Introduction*. John Wiley & Sons.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA. PMLR.

Zhang, Z., Zhang, D., and Qiu, R. C. (2020). Deep reinforcement learning for power system applications: An overview. *CSEE Journal of Power and Energy Systems*, 6(1):213–225.