Chapter 15
Managing the World Complexity: From Linear Regression to Deep Learning

Yann Bouteiller

15.1 Objectives of the Chapter

At the end of this chapter, you will:

– understand the fundamentals of modern ML, and in particular deep learning,
– become familiar with linear regressions, MLPs, CNNs, and RNNs,
– be aware of the supervised techniques that are most relevant for robotics,
– understand the fundamentals of deep reinforcement learning,
– become familiar with Gym environments and DQN,
– be aware of the deep RL algorithms that are most relevant for robotics.

15.2 Introduction

Classical robot algorithms for perception and control are often based on simple, linear models of the world. These approaches are very effective for simple tasks where the system reasonably satisfies the corresponding assumptions in its domain of operation. However, they become inoperative in many high-level reasoning tasks where the complexity of the real world is relevant and needs to be captured. A typical example is the task of driving autonomously from camera pixels, which requires a deep, conceptual understanding of the environment. How can an autonomous car detect other agents such as vehicles and pedestrians, often partially when not entirely occluded? How to predict their individual behaviors and react accordingly? How to reliably detect traffic signalization in all possible variations of the environment, including

Y. Bouteiller (✉)
Department of Computer and Software Engineering, Polytechnique Montréal, Montreal, Canada
e-mail: yann.bouteiller@polymtl.ca

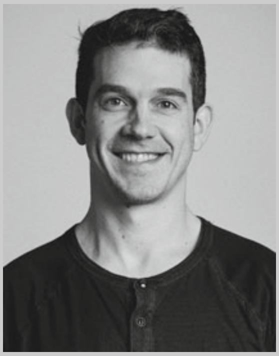
D. Herath and D. St-Onge (eds.), *Foundations of Robotics*,
https://doi.org/10.1007/978-981-19-1983-1_15

light and weather? Over the past decade, the state-of-the-art solutions to these problems have emerged from statistical approximation techniques, nowadays referred to as *machine learning* (ML). Instead of relying on engineered representations of the world, ML approaches build their own representations automatically from large amounts of data, collected either directly from the real world, or from a simulator. The process of building these representations is called *learning* (or, equivalently, *training*). In modern ML, learnt representations can be so abstract that they are often interpreted as being similar to a human-like, conceptual understanding of the world. For instance, ML algorithms are able to learn high-level concepts such as *pedestrian* and *car* by analyzing a large number of images featuring road scenes and can then be used to complete tasks in which these concepts are relevant.

An Industry Perspective

Jonathan Lussier
Director, Intellectual Property and Innovation

Kinova inc.



I graduated in mechanical engineering and started my career in the aerospace industry, mainly in system engineering over 10 years both in simulation and in the product. Meanwhile, in my free time, I was building robots on the side (in my basement) as I saw it as the wave of the future coming. So I started with online resources, 3D printing, and sheet metal work in order to start building some small and subsequently some larger robots arms. It was at this moment that I found out about Kinova (a company close to where I live) and the great work they were doing in the assistive field and decided to apply.

My first task when I started at Kinova was to build the proof of concept for what is now our Gen3 lite robot. Over a span of eight months, I read as much as I could and benefitted from the extremely high level of expertise from Kinova engineers to ramp up and design and build it. Afterward, I transitioned into a role ensuring the Gen3 robot was launched on time, which was very challenging but a great experience especially from the collaborations between different groups within the company, which is so great about robotics—the integration of mechanical and electrical hardware, quality assurance, and of course all the different software disciplines.

The field of machine learning in robotics is changing extremely quickly. One aspect I love is that, contrary to some other fields or even other aspects of robotics where research and development are heavily either industry-led or academic-led, we are seeing many practical applications based on the integration of machine learning and robotics launched commercially, often by people still in academia. These advances, built off the back of thousands of researchers doing more segmented AI (natural language processing, image recognition or classification, etc.), can be combined into the robotic system in an integrated way. As mentioned above, there are so many different disciplines that need to be combined when launching a robotic or automation product that there are opportunities for hardware, traditional software (e.g., machine vision), and AI-led approaches at all the different levels—it is very exciting!

Start small and simple! When working on robotics, getting a complete system up and running can be a challenge in itself. Limiting the number of hardware and software components is key and that applies to AI as well. Take advantage of existing libraries (e.g., Gym, Stablebaselines, etc.) which often combine different options for simulators, datasets, and algorithms wrapped in an easy to use and (most importantly) well-documented interface.

15.3 Definitions

ML is a vast field consisting of many techniques developed for various purposes. We can roughly separate these techniques under two categories: *supervised* and *unsupervised*.[1]

Supervised learning consists of using a set of *labeled* data points to train a *model*. In other words, given a dataset $\mathcal{D} = \{x_i, y_i\}$ of data points x_i (e.g., camera images ...) and corresponding labels y_i (e.g., type of the closest agent present in the image, position estimate of this agent ...), the goal is to find a model f mapping data points to labels such that $f(x) \approx y$ for all data point x and corresponding label y ... including those not present in the dataset! This last property is a central objective of ML, called *generalization*: A good model is not a model that fits the dataset, but a model that fits the real phenomenon (of which the dataset is only a comparatively tiny sample). You will often hear that a good model is one that "generalizes well". Depending on the nature of the labels, the task is called *classification* or *regression*. A *classifier* produces categorical outputs (e.g., is it a car or a cat?). This is typically done

[1] This is a very rough categorization. In particular, semi-supervised learning and reinforcement learning are other important ML categories, which borrow aspects from both supervised and unsupervised learning.

by outputting a vector of probabilities where each dimension represents a category of interest. For instance, a vector of dimension 3 could represent three classes of interest such as “pedestrian”, “car”, and “none”. The output $f(x) = [0.1, 0.8, 0.1]^\top$ could then mean that the closest agent in the x input image is most likely a car. A *regressor* instead produces a real-valued output (e.g., the three-dimensional relative position of the closest agent). In this case, the model directly outputs the value of interest. For instance, the output $f(x) = [3.5, -1.0, 0.0]^\top$ could mean that the closest agent is 3.5 m ahead and 1.0 m on the right. In classifiers and regressors alike, the model usually consists of a set of tunable parameters θ. Thus, finding a good model essentially consists of finding good values for these parameters. Among the most relevant types of parametric models, we can cite decision trees/random forests, which are simple ML algorithms with good properties in terms of interpretability, linear regressions, and neural networks. In this chapter, we will denote parametric models as f_θ, and we will focus our attention on neural networks, which are omnipresent in modern ML. Note that there also exist ML algorithms using nonparametric models, where the model is typically the dataset itself. For instance, the K-nearest neighbors (KNN) algorithm compares new data points to the whole dataset, so as to infer their corresponding labels from the closest labeled data available.

The locution *unsupervised learning* refers to all ML techniques that instead use an unlabeled dataset $\mathcal{D} = \{x_i\}$. Famous examples of unsupervised methods are generative adversarial networks (GANs), intensively used in image generation/transformation, and trained with unlabeled pictures. While GANs are definitely useful for robotics, they are used in very advanced situations that we will only briefly cite in this chapter.

An alternative to the aforementioned categories, called *reinforcement learning* (RL), will be covered with greater attention in the second part of this chapter. RL algorithms learn a controller from their own experience, in a near-unsupervised fashion. However, this is done by leveraging an external reward signal that remotely resembles a label, and thus RL stands somewhere in between supervised and unsupervised approaches.

In robotics, supervised methods are typically useful for perception. In particular, we use neural networks for image analysis, spatial perception, speech recognition, signal processing. … Supervised learning is also possible for control, in particular through behavioral cloning, which consists of imitating the *policy*[2] of an expert. However, behavioral cloning is inherently limited by the expert level. Thus, policy optimization strategies based on trial-and-error, such as RL and genetic algorithms, are often preferred for learning a controller.

[2] A policy is a set of relations that maps observations to actions.

15.4 From Linear Regression to Deep Learning

15.4.1 Loss Optimization

The goal of supervised learning is to find a model f such that, for all input x and desired output y, $f(x) \approx y$. In ML, the quality of the model f is typically evaluated in terms of a *loss function*. A loss function takes a model and a dataset as input and outputs a real value that represents how bad the model is performing on the dataset. In this chapter, we will denote loss functions as $L(f, \mathcal{D})$ in the general case and $L(\theta, \mathcal{D})$ for parametric models. The smaller the loss is, the better the model is considered. In other words, the goal of ML is almost always an optimization problem consisting of minimizing a loss function. Many different loss functions exist, each with their own properties in terms of what they consider being a good model and how easy they are to optimize. The most common losses are the *mean squared error* (MSE) loss and the *cross-entropy* (CE) loss.

Mean Squared Error Loss
The MSE loss is typically used for regression. It is defined as follows:

$$L_{\text{MSE}}(f, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}(|f(x_i) - y_i|^2)$$

where n is the size of the dataset $\mathcal{D}$ (i.e., the number of (x_i, y_i) pairs in $\mathcal{D}$).

An important property of the MSE loss is that it strongly penalizes models that have a large prediction error $|f(x_i) - y_i|$ for some data point x_i. In other words, the MSE loss prefers models that do not ignore any data point. Although this is desirable in general, this also has the drawback of being strongly impacted by outliers.[3]

Cross-Entropy Loss
The CE loss is typically used for classification. It is defined as follows:

$$L_{\text{CE}}(f, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} -\ln(f_{y_i}(x_i))$$

where n is the size of the dataset and where $f_{y_i}(x_i)$ is the probability that $f(x_i)$ outputs for the class y_i.

A reason why the CE loss is widely used is that its gradient[4] is easy to compute (we will see why this is important later in this chapter). Since $f_{y_i}(x_i)$ is a probability,[5] its

[3] In ML, outliers are data points whose labels are far from what is expected.

[4] A gradient is the Jacobian of a single multivariate function, i.e., with one row. Note that, in deep learning, we often transpose the gradient to work with column vectors only. For instance, $\nabla_{[a,b]^\top}(a + b^2) = [1, 2b]^\top$.

[5] In practice, this is not really a probability, but the output of a softmax function, which also sums to 1.

value lies between 0 and 1 (0 being excluded in practice). The closer this probability is to 1, the smaller the loss is, while a probability close to 0 is strongly penalized. Indeed we want $f_{y_i}(x_i)$ to be 1, since the label of x_i is y_i in our dataset.

15.4.2 Linear Regression

One of the oldest and most fundamental supervised ML techniques is the *linear regression*. Linear regression was introduced by Legendre and Gauss who used it to predict astronomical trajectories in the early 1800s (Stigler, 1981), way before the term "machine learning" was introduced and popularized. Performing a linear regression consists of fitting a parametric linear model to a labeled dataset $\mathcal{D} = \{\mathbf{x_i}, y_i\}$ where the labels $y_i \in \mathbb{R}$ are single real values. As seen in Chap. 6, a linear model is of the form $f_\theta(\mathbf{x_i}) = \mathbf{w}^\top \mathbf{x_i} + b$, where $\mathbf{w}$ is a vector of *weights* and b is a single *bias*. The set of tunable parameters is $\theta = \{\mathbf{w}, b\}$.

Interestingly, it is possible to find the optimal solution to the linear regression problem using matrix calculus. For this matter, a useful trick is to write θ as the concatenation of $\mathbf{w}$ and b, and to append a 1 to the $\mathbf{x_i}$ vector:

$$\boldsymbol{\theta} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \\ b \end{bmatrix} \quad \text{and} \quad \overline{\mathbf{x}}_\mathbf{i} = \begin{bmatrix} x_{i,1} \\ \vdots \\ x_{i,m} \\ 1 \end{bmatrix}$$

This allows us to write the linear model as a simple vector multiplication:

$$f_\theta(\mathbf{x_i}) = \boldsymbol{\theta}^\top \overline{\mathbf{x}}_\mathbf{i}$$

Now we can minimize the MSE loss of our parametric f_θ model. For our dataset $\mathcal{D}$ of n $(\mathbf{x_i}, y_i)$ pairs, we define $\overline{\mathbf{X}} \in \mathbb{R}^{n\times(m+1)}$ as the matrix formed by the n "augmented" data points and $\mathbf{Y} \in \mathbb{R}^n$ as the vector formed by the n labels:

$$\overline{\mathbf{X}} = \begin{bmatrix} \overline{\mathbf{x}}_\mathbf{1}^\top \\ \vdots \\ \overline{\mathbf{x}}_\mathbf{n}^\top \end{bmatrix} \quad \text{and} \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

This enables us to write the MSE loss in matrix form:

$$L_{\text{MSE}}(\theta, \mathcal{D}) = \frac{1}{n}(\overline{\mathbf{X}}\boldsymbol{\theta} - \mathbf{Y})^\top(\overline{\mathbf{X}}\boldsymbol{\theta} - \mathbf{Y})$$

To minimize this loss, we take its gradient with respect to our vector of tunable parameters $\boldsymbol{\theta}$, which can be shown to be (the proof is out of this chapter scope):

$$\nabla_\theta L_{\text{MSE}}(\theta, \mathcal{D}) = \frac{2}{n}(\overline{\mathbf{X}}^\top \overline{\mathbf{X}}\boldsymbol{\theta} - \overline{\mathbf{X}}^\top \mathbf{Y})$$

We then set this derivative to 0 to find the minimum (the convexity of the loss with respect to $\boldsymbol{\theta}$ is easy to prove), which yields our optimal vector of parameters[6]:

$$\boldsymbol{\theta}^* = (\overline{\mathbf{X}}^\top \overline{\mathbf{X}})^{-1} \overline{\mathbf{X}}^\top \mathbf{Y}$$

on the condition that $\overline{\mathbf{X}}^\top \overline{\mathbf{X}}$ is invertible. Note that this is the left MPGI of $\overline{\mathbf{X}}$ multiplied by $\mathbf{Y}$, as we have essentially minimized the Euclidean norm of $\overline{\mathbf{X}}\boldsymbol{\theta} - \mathbf{Y}$ (c.f. Chap. 6).

15.4.3 Training Generalizable Models

Overfitting

As seen in the previous section, the linear regression problem has an optimal solution that can be written analytically. This solution is not really straightforward, though, since computing the inverse of $\overline{\mathbf{X}}^\top \overline{\mathbf{X}}$ can be challenging, especially when the $\mathbf{x_i}$ are high dimensional. Moreover, linear regressions are a very simple and special case. In advanced ML approaches, such analytical solution is virtually never available. Indeed, to find the optimal vector of parameters $\boldsymbol{\theta}^*$, we have computed the gradient of the loss function with respect to $\boldsymbol{\theta}$ and found where this gradient was equal to zero. The reason why this worked is that, in a linear regression, the MSE loss is convex with respect to $\boldsymbol{\theta}$. This property is generally not satisfied in complex models such as neural networks, and thus it is not possible to apply the same strategy. But more importantly, this is not even a suitable thing to do!

Remember that we are *not* looking for the model that best fits our dataset (which is exactly what we have computed in the previous section), but the model that best fits the real world. In fact, the “optimal” set of parameters that we have computed is the worst possible example of *overfitting* that one can commit with a linear regression: We have selected our set of parameters θ^* not because it is best at describing the real world, but because it is best at describing the dataset.

This is usually not a big deal when using linear regressions: If the phenomenon of interest is indeed linear, any linear approximation using a reasonable number of data points is likely to be a good approximation. However, practical problems are scarcely ever linear. In fact, high-level reasoning tasks—such as driving from pixels—are highly nonlinear and require nonlinear models like neural networks (try to imagine what would happen if you performed a linear regression on a dataset of

[6] The scikit-learn Python library can be used to compute this set automatically: scikit-learn.org.

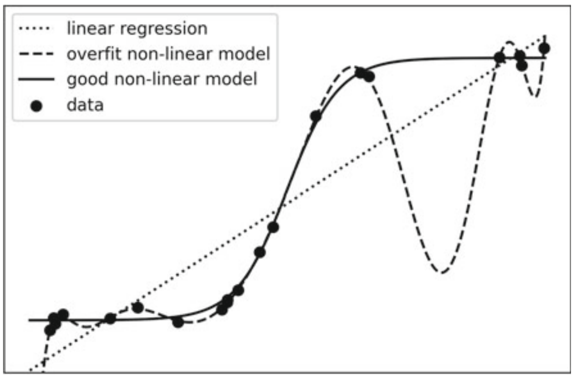


Fig. 15.1 Overfitting. Given data points sampled from a nonlinear phenomenon, a model that perfectly describes all the data points is likely to generalize poorly

camera images x_i to compute outputs $f_\theta(x_i)$ that represent, say, the distance to the nearest car …).

Typically, nonlinear models do not have strong *inductive biases*[7] like linearity and have a much bigger *capacity*.[8] They are able to represent crazily complex shapes, which can fit the dataset exactly and yet generalize horribly. For instance, in Fig. 15.1, we are trying to model a nonlinear phenomenon from which a dataset has been sampled (each black circle represents a data point and its label: the x_i are the values on the x axis and the y_i are the values on the y axis). Linear regression (dotted) performs poorly on this simple nonlinear problem. Using a complex nonlinear model instead and minimizing the MSE loss all the way down to zero produce a strongly overfit model (dashed). The model represented with a full line has a slightly bigger MSE loss when evaluated on our dataset, but it is likely to generalize much better to unseen data.

Minimizing a loss in ML is not a typical optimization problem where one seeks to actually find the minimum of the loss. Instead, the loss minimization procedure is merely a tool to find a good set of parameters for our model. But how exactly do we find this set of parameters, and how do we know that it is a good one?

Stochastic Gradient Descent

Although the loss function for nonlinear models is typically not strictly convex, it can usually be considered approximately pseudoconvex[9] in practice. Complex nonlinear models such as neural networks have many tunable parameters (i.e., a very high-dimensional $\boldsymbol{\theta}$), and we are unlikely to find a $\boldsymbol{\theta}$ vector that cannot be improved in any of its dimensions. Thus, despite being unable to find the analytical solution to the loss minimization problem, we can select a random parameter vector $\boldsymbol{\theta_0}$ and iteratively optimize our loss from there by following the negative gradient. For a given value of $\boldsymbol{\theta_t}$, we compute the local negative gradient of the loss:

[7] An inductive bias is an assumption about the structure of the world that we force into our model.

[8] The capacity of a model indicates the degree of complexity that it is able to represent.

[9] A pseudoconvex function increases forever in the direction of any of its local gradients.

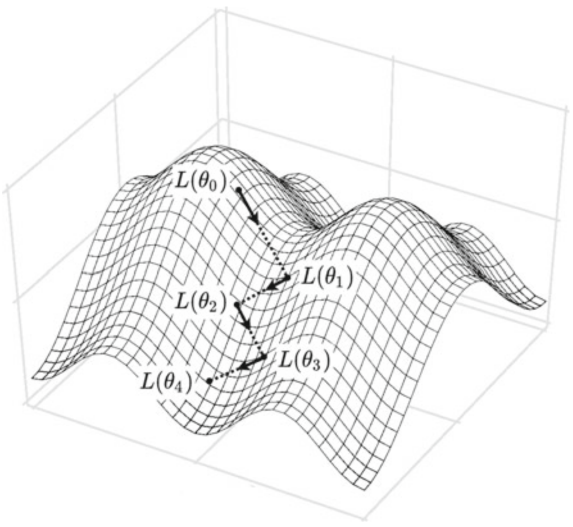


Fig. 15.2 Gradient descent. The GD algorithm optimizes the loss by iteratively descending its slope in the directions of its local gradient with respect to $\boldsymbol{\theta}$ (arrows)

$$\boldsymbol{\nabla}_t = -\nabla_\theta L(\boldsymbol{\theta}_t, \mathcal{D}),$$

and we update our parameter vector in the direction of this local negative gradient:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \boldsymbol{\nabla}_t,$$

where the *learning rate* α is an hyperparameter.[10] This procedure is illustrated in Fig. 15.2.

Computing the true local gradient of the loss at each gradient descent iteration is very computationally intensive. The gradient needs to be averaged over the entire dataset at each iteration, which is not suitable. A better way of performing gradient descent is *stochastic gradient descent* (SGD), an important key of modern ML success. In its “pure” (vanilla) version, SGD is the same algorithm as gradient descent, except instead of averaging the local gradient over the whole dataset, the gradient is taken with respect to a single (x_i, y_i) pair sampled from the dataset. This technique produces a very rough estimate of the gradient, at a much smaller computational cost. But despite this estimate being rough, a small optimization step can still be taken in its direction. This operation can be performed rapidly and repeated over many times. Moreover, the stochastic nature of the gradient estimate enables SGD to escape from *local extrema* and *saddle points*[11] easily where vanilla gradient descent would fail. These properties make SGD much more efficient than gradient descent in practice.

However, this version of SGD is still computationally inefficient. In fact, computing an average gradient over several samples is a parallelizable task, and thus it

[10] Hyperparameters are parameters not learnt by the optimization algorithm (often just set manually).
[11] Point where the gradient is close to zero on all dimensions ($\nabla_\theta L(\theta_t) \approx \mathbf{0}$), but that is not a local extremum.

is not really a good idea to use something as extreme as one single (x_i, y_i) pair for our local gradient estimate. Modern GPUs enable using several of them at no additional cost in terms of computation. This is why, in practice, we never use one single sample from the dataset, but a certain number of them. The number of samples per gradient estimate is an hyperparameter, called the *batch size*. Batch sizes between 16 and 4096 are common choices. As a rule of thumb, small batches yield rough gradient estimates and work best with smaller learning rates, whereas larger batches yield better[12] estimates and can afford larger learning rates (He et al., 2019). The resulting algorithm, called *minibatch gradient descent* (or also SGD), is the basis of most state-of-the-art loss optimizers, such as *Adam* (Kingma & Ba, 2014) and *RMSProp*.

Algorithm 1 Minibatch gradient descent (SGD)

```
Require: $\mathcal{D}, f_\theta, L, \alpha, n$                ▷ dataset, model, loss function, learning rate, batch size
Ensure: $\boldsymbol{\theta} \approx \boldsymbol{\theta}^*$                ▷ near-optimal parameters for the model
  $\boldsymbol{\theta} \leftarrow$ random values                ▷ initialize parameters
  repeat
    batch $\leftarrow$ n $(x, y)$ pairs sampled from $\mathcal{D}$                ▷ sample minibatch from dataset
    $\nabla \leftarrow -\nabla_\theta L(\boldsymbol{\theta}, \text{batch})$                ▷ estimate gradient on minibatch
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\nabla$                ▷ update parameters by descending gradient
  until convergence of $L(\boldsymbol{\theta}, \mathcal{D})$                ▷ once in a while, evaluate actual loss on dataset
```

Training, Validating, and Testing

As long as the local gradient can be computed for any value $\boldsymbol{\theta}$ of the parameter vector, SGD enables minimizing the loss of approximately pseudoconvex nonlinear models. However, when optimized by SGD, the loss on our dataset will still eventually converge too close to its true minimum. In other words, if not stopped early enough, SGD will overfit!

Fortunately, there is a way of stopping convergence right before this happens. This technique, called *early stopping*, also enables evaluating the true performance of the model on unseen data. The main idea is that we do not train our algorithm on the entire available dataset. Instead, we shuffle the dataset $\mathcal{D}$ and split the result into three disjoint subsets:

- a *training set* $\mathcal{D}_{\text{train}}$,
- a *validation set* $\mathcal{D}_{\text{validation}}$,
- a *test set* $\mathcal{D}_{\text{test}}$.

We then perform SGD by estimating gradients only on the training set, with a small change: instead of stopping the algorithm when we think the loss has converged on the training set, we stop the algorithm when the loss stops improving on the validation set.

[12] In the sense of being closer to the true gradient and thus less stochastic, which is only partly suitable!

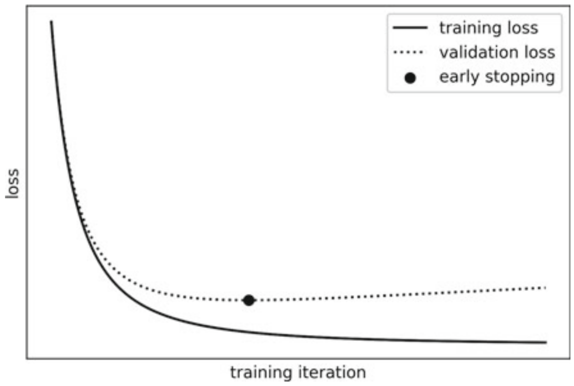


Fig. 15.3 Early stopping. The validation set enables finding when to stop training before overfitting starts harming the generalization properties of the model

This is because, when the model starts overfitting, its performance on the validation set (on which it is not trained) starts decreasing (i.e., the validation loss starts increasing). Figure 15.3 displays a typical example of this phenomenon over training.

Note that the loss function is not necessarily what people use to determine when early stopping should happen. It is possible to use metrics we are more directly interested in. For instance, in classification tasks, we often use the accuracy or the $F1$-score.

Finally, you may wonder why we have split our dataset into three parts rather than two. Indeed, we have not used $\mathcal{D}_{\text{test}}$ at all. And there is a good reason for this: You should never use the test set before your system is final and ready for production! There are many subtle ways in which it is possible to overfit on our dataset in an ML project, and early stopping is one of them. Because we have selected our best model based on its performance on the validation set, we have slightly overfit on this subset. To really evaluate our performance, the only unbiased way is to do it on the unseen test set once the model is final. In supervised learning, this is important as a last sanity check. Of course, this can be replaced by testing the model directly in the real world when possible, in which case the real world becomes the test set. Typically, the performance on the test set is slightly worse than the performance on the validation set, which is noticeably worse than the performance on the training set.

Regularization

On top of early stopping, many existing techniques, called *regularizers*, help improve the generalization performance of a model. Some of these techniques, such as *L1* and *L2*, add a term to the loss in order to penalize large parameter values and promote models that use as few parameters as possible (this avoids crazy models similar to Fig. 15.1). Others, such as *dropout*, introduce noisy modifications to the model during training in order to promote robustness.

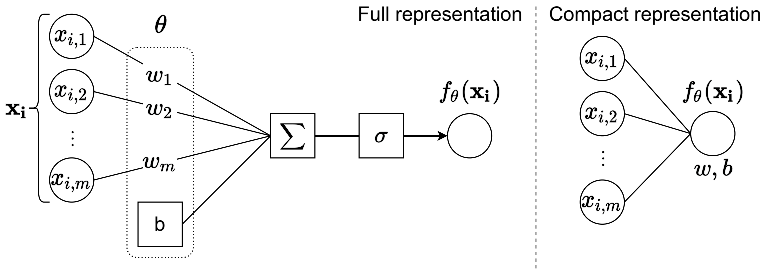


Fig. 15.4 Simple neuron

15.4.4 *Deep Neural Networks*

ML has attracted a lot of attention over the past few years. The main reason for this surge of interest is that modern GPUs (and, more recently, TPUs/IPUs) have provided enough computational power to train a class of complex nonlinear models invented in the 1940s–1960s (Fitch, 1944; Ivakhnenko & Lapa, 1965), whose potential had remained unknown for several decades (Krizhevsky et al., 2012). These models, called *deep neural networks* (DNNs), are an algorithmic attempt to mimic the brain. They project their input into successive, more and more abstract representations, that eventually map to the desired output. DNNs are today at the core of most ML successes. In fact, they have become so prominent that modern ML is often simply called *deep learning*.

The atomic component of a DNN is a very simple, usually nonlinear model, called *neuron*. A neuron is made of a linear model,[13] directly followed by an easily differentiable, usually nonlinear function, called *activation*. Using the same notation as for linear regressions, the operation performed by a neuron is

$$f_\theta(\mathbf{x_i}) = \sigma(\boldsymbol{\theta}^\top \overline{\mathbf{x}}_\mathbf{i})$$

where σ is the activation function. We often represent a neuron as a graph, which helps visualize the flow of operations. In particular, we will use the compact representation to understand more complex DNNs (Fig. 15.4).

The only structural difference with a linear regression model is the activation σ, which plays a central role in deep learning. Many activation functions exist in the literature, the most common being the *sigmoid* and the *rectified linear unit* (ReLU).

The sigmoid is defined as follows:

$$\text{sigmoid}(a) = \frac{1}{1 + \mathrm{e}^{-a}}$$

[13] The same model as used by linear regression.

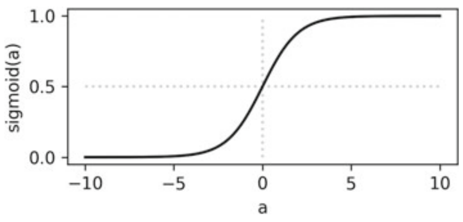


Fig. 15.5 Sigmoid activation

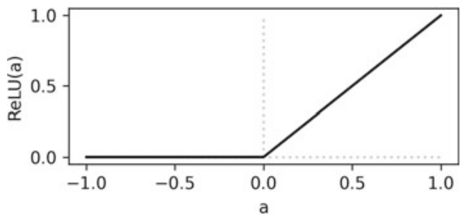


Fig. 15.6 ReLU activation

The sigmoid is generally used when one needs to squash an output between 0 and 1. However, its derivative is near zero everywhere except around the origin, which is harmful to the convergence of SGD. Plus, compared to ReLU, the sigmoid is relatively costly to compute (Fig. 15.5).

The ReLU is a simple clipping operation:

$$\text{ReLU}(a) = \max(x, 0)$$

Computing a ReLU is blazing fast and so is computing its derivative (0 for negative numbers, and 1 for strictly positive numbers, the derivative at the origin being arbitrary) (Fig. 15.6).

The point of using such a simple nonlinearity may seem unclear at first: A neuron with a ReLU activation is just a crippled linear model unable to output anything negative! But contrary to linear models, a neuron is never used alone: Its representational power comes from being coupled with other neurons to form a DNN. In its simplest form, called *multilayer perceptron* (MLP), a DNN is a stack of *layers*, each made of several parallel neurons (Fig. 15.7).

Remember that each individual neuron has a vector of tunable weights and a single tunable bias as parameters. Since a layer has several parallel neurons, this translates to each layer having a matrix of tunable weights and a vector of tunable biases. The set of tunable parameters of an MLP is thus $\theta = \{\mathbf{W_i}, \mathbf{b_i}\}_{i=1\ldots k+1}$. The operation performed by an MLP is as follows:

$$\begin{aligned}
h_1(\mathbf{x_i}) &= \sigma_1(\mathbf{W_1 x_i} + \mathbf{b_1}) \\
h_2(\mathbf{h_1}) &= \sigma_2(\mathbf{W_2 h_1} + \mathbf{b_2}) \\
&\dots \\
f_\theta(\mathbf{x_i}) = f(\mathbf{h_k}) &= \sigma_{k+1}(\mathbf{W_{k+1} h_k} + \mathbf{b_{k+1}})
\end{aligned}$$

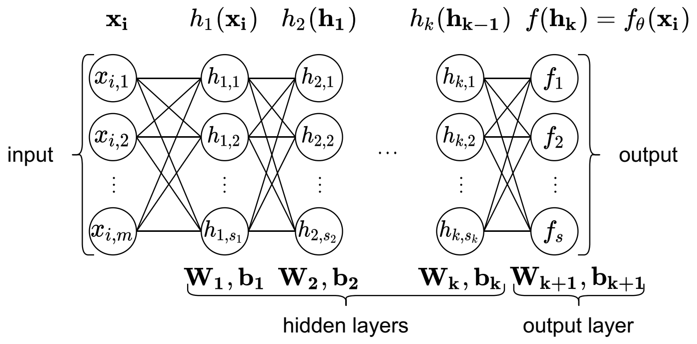


Fig. 15.7 Multilayer perceptron

Despite their fairly simple structure, DNNs perform extremely complex nonlinear projections and are typically treated as black boxes. Thus, we say that layers other than the last one are *hidden*. On the other hand, the last layer has a special role and is typically a simple linear layer with no activation (i.e., σ_{k+1} is the identity function). This layer projects the output of the last hidden layer into the output space of the DNN (for instance, into a 3D vector if we are predicting a position …).

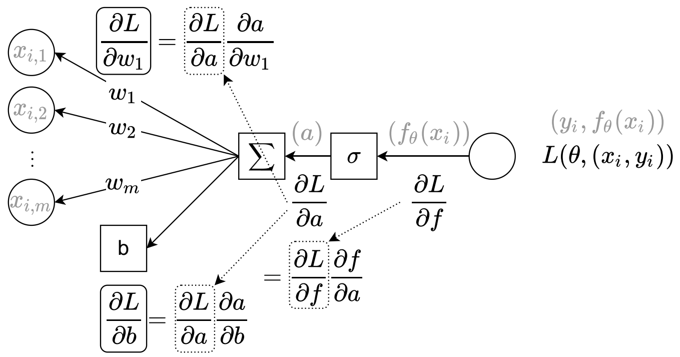
Notice that, without nonlinear activation functions, this structure would only be a crazy way of building a linear model. This is because combining linear combinations yields other linear combinations. Yet, simple nonlinearities such as ReLUs make DNNs much more powerful. In fact, a famous result called the *universal approximation theorem* (Hornik, 1991) shows that, even with a single (large enough) hidden layer, a neural network can approximate any continuous function arbitrarily well.[14] This includes mappings from raw camera images to conceptual information about their content, or even directly to optimal control commands for our robot!

15.4.5 Gradient Back-Propagation in Deep Neural Networks

We know that DNNs can approximate virtually any complicated nonlinear mapping of interest, such as mappings from camera images to conceptual descriptions of their content. Moreover, we know a way of searching for this mapping: SGD with early stopping. The only ingredient we are missing for applying this strategy is an estimate of the gradient of the loss with respect to all tunable weights and biases of our DNN.

The key to the success of DNNs is an algorithm introduced in 1970 (Linnainmaa, 1970) and made practical by the use of modern GPUs/TPUs/IPUs, called *gradient back-propagation* (or *backprop* for short). Backprop is a dynamic programming algorithm that efficiently computes the gradient of the loss. To perform a backprop,

[14] For an animated illustration of this result: http://neuralnetworksanddeeplearning.com/chap4.html (Nielsen, 2015).



$\boxed{\frac{\partial L}{\partial w_1}} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w_1}$
$x_{i,1}$
$x_{i,2}$
$x_{i,m}$
w_1
w_2
w_m
$\sum$
(a)
σ
$(f_\theta(x_i))$
$(y_i, f_\theta(x_i))$
$L(\theta, (x_i, y_i))$
$\frac{\partial L}{\partial a}$
$\frac{\partial L}{\partial f}$
b
$= \frac{\partial L}{\partial f}\frac{\partial f}{\partial a}$
$\boxed{\frac{\partial L}{\partial b}} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial b}$

Fig. 15.8 Gradient back-propagation

one first needs to perform a *forward propagation* in the DNN, i.e., compute the $f_\theta(x_i)$ output. Then, backprop uses the *chain rule* of partial derivatives to propagate gradients backward in the graph. For simplicity, let us visualize this process on a single neuron.

In Fig. 15.8, we want to compute the gradient of the loss L with respect to the weights w_i and the bias b. Once the output f has been computed by a forward propagation, it is straightforward to compute the derivative of the loss with respect to this output, $\frac{\partial L}{\partial f}$. We can then use the result to compute $\frac{\partial L}{\partial a}$,[15] which, according to the chain rule, is equal to $\frac{\partial L}{\partial f}\frac{\partial f}{\partial a}$. Indeed, $\frac{\partial f}{\partial a}$ is just the derivative of the activation σ. The result can then be propagated further back to compute the partial derivatives we are interested in: $\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w_j}$ and $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial b}$. Indeed, $\frac{\partial a}{\partial w_j}$ is $x_{i,j}$ and $\frac{\partial a}{\partial b}$ is 1. To generalize this procedure to DNNs, we also use $\frac{\partial L}{\partial a}$ to compute $\frac{\partial L}{\partial x_{i,j}} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial x_{i,j}}$, where $\frac{\partial a}{\partial x_{i,j}}$ is w_j, and we repeat this process in previous neurons. Note that intermediate results such as $\frac{\partial L}{\partial a}$ are computed only once and reused many times. This makes this dynamic programming procedure very efficient in DNNs.

We now master the basics of deep learning! In practice, we will not implement SGD and backprop manually, because highly optimized libraries have done all the work for us. Nowadays, the most popular such libraries are PyTorch and TensorFlow.[16]

15.4.6 Convolutional Neural Networks

We have seen how DNNs can learn extremely complex nonlinear tasks such as mapping camera pixels to relevant high-level information … in theory. In reality, using an MLP to process camera images is bound to fail.

[15] Here, a denotes the intermediate forward value after the sum and before the activation.

[16] pytorch.org and tensorflow.org.

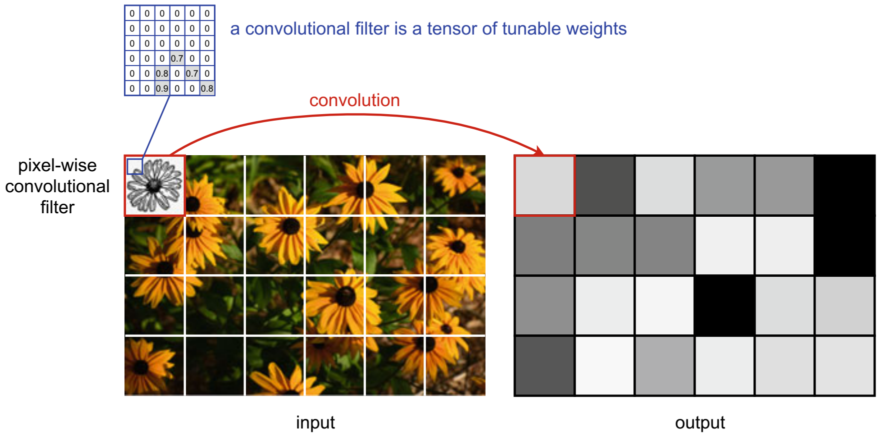

a convolutional filter is a tensor of tunable weights
convolution
pixel-wise convolutional filter
input
output

Fig. 15.9 Neural convolution

To understand why, let us consider a small 100 × 100 RGB image input. When flattened with all three color channels concatenated, this becomes a 30,000-dimensional vector. For an MLP to be able to extract meaning of this vector, it should have at least about as many neurons in its first layer; otherwise, a near-linear regression on pixels would happen. The number of weights in an MLP layer being the number of inputs multiplied by the number of parallel neurons, this would require around 9×10^8 tunable weights in the first layer alone. Computing their individual partial derivatives at each SGD iteration would be painfully slow.

Furthermore, deep learning is never guaranteed to converge to anything interesting. The convergence and generalization properties of SGD rely on the pseudo-convexity assumption and depend on many hyperparameters (structure of the neural network, learning rate, initial set of parameters …). Thus, it is often a good idea to help our models learn meaningful mappings by enforcing inductive biases when possible, similar to how linearity makes linear models efficient for linear problems. In particular, camera images have a strong spatial structure that we can use to our advantage. Convolutional neural networks (CNNs) are an effective way of doing so. Instead of connecting each color channel of each pixel to each neuron of the first hidden layer, CNNs borrow a much lighter technique from traditional computer vision: image convolution.[17] This technique, illustrated in Fig. 15.9, uses *filters* (also called *kernels*) to “scan” images for specific patterns and perform local projections.

A filter is a small array of weights, plus a single optional bias,[18] all tunable. We split the image into pieces of the same size as the filter. Typically, these pieces are overlapping (e.g., shifted by only one pixel), although for the sake of clarity they do not overlap in Fig. 15.9 (they form the white “grid”). We apply the filter to

[17] The mathematical operation is actually a cross-correlation, but ML practitioners call it “convolution”.

[18] Another (less common) version exists in which there is one bias per output pixel.

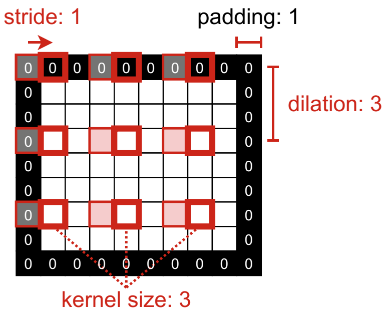


Fig. 15.10 Convolution options

each piece individually. This is done by multiplying each pixel of the piece with the corresponding weight of the filter, then summing the results into a single value, and adding the bias (NB: This operation is a linear combination).

The result is then fed to an activation function, which produces a new pixel value. Together, new pixels form an output image called *feature map*. More precisely, a 2D convolutional filter is in fact a 3D tensor[19] whose depth is the number of input channels (e.g., 3 when the input is a RGB image). It combines all input channels into a single output feature map. CNNs commonly use hundred of filters in parallel, each producing a different feature map depending on the weights and bias of the filter. These feature maps then become the input channels of the next convolutional layer. Using CNN filters greatly reduces the number of trainable parameters when compared to MLPs: Only the weights and bias of each filter are trainable, and convolutional filters are often of size 3×3 in practice. Typically, we find that filters become edge detectors or pattern detectors during training. For instance, in Fig. 15.9, the trained filter has naturally taken the shape of a flower so as to detect flower patterns.

Additionally, we commonly use the following operations in convolutional layers:

– *Zero padding*: We append zeroes to the border of the input image.
– *Stride*: We shift pixels between convoluted pieces of the input image.
– *Dilation*: We shift pixels between elements of the convolutional filter.

These options are illustrated in Fig. 15.10 (integer values describe both dimensions).

Finally, kernel-based operations other than image convolution are often used in CNNs. The most common is *max pooling*, which reduces the size of a feature map by selecting the pixel with the maximum value in the area of the kernel, as illustrated in Fig. 15.11.

CNNs are typically made of alternating convolutional and max pooling layers and are often very deep (i.e., they have many layers). They are by far the current state of the art in a wide range of computer vision tasks, some of which we will highlight later in this chapter. CNNs are a building block of many GANs used for image

[19] A tensor is a multidimensional array, for instance a matrix is a 2D tensor.

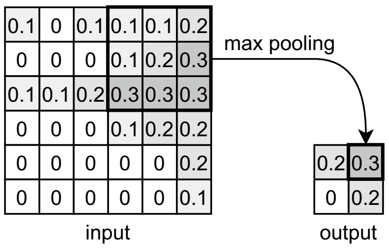


Fig. 15.11 Max pooling

manipulation and generation, and they are not limited to 2D image processing. For instance, 1D convolutions can be used for signal processing, and 3D convolutions can be used for video processing.

15.4.7 Recurrent Neural Networks

Time series are often central in robotics: We need them to analyze the past and plan in the future. Thus far, we have seen how DNNs can analyze the present, but can they keep track of the past? Is it possible to predict and plan in the future? Can we generate coherent sequences such as paths, or even sentences? Can we process time series such as video streams, or even sound and voice?

In deep learning, the past can be analyzed by feeding the whole history of relevant observations to a DNN. For instance, a self-driving car would be unable to output a relevant command from one camera image only, as this would contain little information about the dynamics of the world (i.e., only contextual information ...). On the other hand, a history of the last few camera images equally spaced in time is enough to infer simple dynamics. This concern is more generally known as the *Markov property*: The history fed as input to the model must be long enough so that any earlier observation is irrelevant to the task.

Planning in the future can be done by recursively feeding a DNN with its own last few outputs. For instance, let us imagine a model that takes a target position and a path as input. The model appends a waypoint to the path so that it gets closer to the target. The updated path can then be fed back to the model. This procedure repeated several times yields a path planning algorithm. A similar procedure can be used to generate speech or music. ...

Although feeding a history of observations directly to a vanilla DNN is possible, this quickly gets inefficient and impractical. Naively processing the whole history of relevant observations at each forward propagation is computationally intensive and may perform poorly due to the lack of inductive biases. Fortunately, a better alternative exists: Recurrent neural networks (RNNs) are able to automatically detect and keep track of only the relevant information from past observations. Instead of being fed the whole history at each forward propagation, they take a single observation as input and store the relevant information directly in their hidden layers, within a persistent *hidden state*.

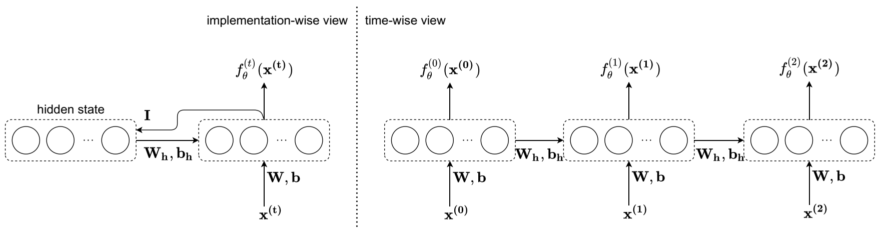

implementation-wise view
time-wise view
hidden state
$f_\theta^{(t)}(\mathbf{x^{(t)}})$
$\mathbf{I}$
$\mathbf{W_h}, \mathbf{b_h}$
$\mathbf{W}, \mathbf{b}$
$\mathbf{x^{(t)}}$
$f_\theta^{(0)}(\mathbf{x^{(0)}})$
$f_\theta^{(1)}(\mathbf{x^{(1)}})$
$f_\theta^{(2)}(\mathbf{x^{(2)}})$
$\mathbf{x^{(0)}}$
$\mathbf{x^{(1)}}$
$\mathbf{x^{(2)}}$

Fig. 15.12 Recurrent layer

Figure 15.12 describes a simple RNN layer. For the sake of clarity, arrows represent matrix multiplications, i.e., connections between layers, instead of individual connections between neurons ($\mathbf{I}$ is the identity matrix). Time is discretized into timesteps. The output $\mathbf{f}_\theta^{\mathbf{(t)}}$ of the layer at timestep t is computed from both the input $\mathbf{x^{(t)}}$ and the output from the previous timestep $\mathbf{f}_\theta^{\mathbf{(t-1)}}$. An additional set of parameters $\mathbf{W_h}$ and $\mathbf{b_h}$ handles how memorized information is combined with new information. Mathematically, the output at timestep t is as follows:

$$\mathbf{f}_\theta^{\mathbf{(t)}} = \sigma(\mathbf{W}\mathbf{x^{(t)}} + \mathbf{b} + \mathbf{W_h}\mathbf{f}_\theta^{\mathbf{(t-1)}} + \mathbf{b_h})$$

To train an RNN, all observations in the relevant portion of the history are fed to the model one by one. Then, the gradient of the loss can be back-propagated through time. This operation is similar to how back-propagation is performed in MLPs, except the gradient also flows back through the horizontal arrows in the time-wise view of Fig. 15.12.

RNNs not only make the forward propagation computationally efficient (since only one observation is fed to the model at each timestep), but also constitute an inductive bias that promotes memorization of high-level concepts rather than raw inputs. Indeed, the persistent information consists of the values projected by hidden layers. In deep learning, these projections are typically seen as extracted concepts.

15.4.8 Deep Learning for Practical Applications

The field of deep learning is very competitive and evolving rapidly. This yields many high-performance models that practitioners can use directly in robot applications. Due to Python being particularly popular in the deep learning community, most readily available implementations are found in Python. Nevertheless, it is always possible, although a bit cumbersome, to extract readily trained weights and biases from Python in order to implement the model in more efficient languages for production. In fact, PyTorch and TensorFlow both provide ways of facilitating the transfer of Python models to C++. We provide a non-exhaustive list of supervised and unsupervised approaches that are relevant for robotics.

CNNs and GANs

CNNs and GANs have attracted a large portion of the ML research focus over the past few years. They are particularly often used in modern computer vision.

- ImageNet: ImageNet (Deng et al., 2009) is a benchmark on which many high-performance CNNs are compared for pure image classification. At the moment of writing this book, the best-performing such models are the EfficientNet family (Pham et al., 2021; Tan & Le, 2019).
- YOLO: YOLO (You Only Look Once) (Bochkovskiy et al., 2020; Long et al., 2020; Redmon et al., 2016) is a very popular family of CNNs combining image classification and bounding boxes. YOLO finds all instances of known categories in an image and draws a bounding box around each instance.
- Mask-R CNN: Mask-R CNN (He et al., 2017) is similar to YOLO, but even more evolved. On top of detecting all class instances with their bounding boxes in an image, Mask-R CNN draws the actual segmentation of each instance.
- PoseNet: PoseNet (Kendall et al., 2015; Moon et al., 2018) is a CNN able to extract human poses from camera images, e.g., for non-verbal communication with the robot.
- Super-resolution: Super-resolution models (Wang et al., 2020) are able to improve the resolution of input images, e.g., for low-quality cameras. They are often based on GANs.
- Image inpainting: Image inpainting models (Elharrouss et al., 2020) are able to fill gaps in images. For example, they can be used to fill gaps in depth maps generated by LIDARs, or to reconstruct partially occluded subjects. They are also often based on GANs.
- Domain adaptation: Domain adaptation models (Wang & Deng, 2018) enable transforming data from one domain (e.g., data from a simulator) into data from another domain (e.g., real-world data!). CycleGAN (Zhu et al., 2017) is a popular example.

Sequential Modeling

The RNN structure that we have described in the previous section is often informally called "vanilla RNN". In practice, much more efficient types of RNNs are available.

- LSTM: A long short-term memory (LSTM) (Hochreiter & Schmidhuber, 1997) is a special type of RNN with a more complicated, *gated* structure. In particular, it is able to selectively forget pieces of information and keep what it thinks is relevant for many timesteps.
- GRU: A gated recurrent unit (GRU) (Chung et al., 2014) is similar to an LSTM, but computationally lighter.
- Autoregressive models: Autoregressive models are not really RNNs, but evolved forms of the naive recursive procedure described in the previous section for generating sequences with vanilla DNNs. These models use their own previous outputs directly as an input sequence and implement inductive biases to process this

sequence efficiently. They can be used to generate human voice for example, as done by WaveNet (Oord et al., 2016).
- Transformers: Transformers (Vaswani et al., 2017) are not really RNNs either. They also take whole sequences as input. Nevertheless, they are the current state of the art in many sequence processing tasks. Transformers use an *attention* mechanism to efficiently process sequences by focusing only on relevant parts of the input. For instance, GPT-3 (Brown et al., 2020) is an autoregressive transformer able to generate human-like text. Another example is BERT (Devlin et al., 2018), a transformer used for language understanding. Both BERT and GPT-3 are immense models readily pretrained that one needs to fine-tune for their application.

Note that training LSTMs and GRUs is not fully parallelizable and is thus slow, but using them is fast once trained. On the other hand, training transformers is parallelizable and is thus fast, but they are slow to use once trained. This is an important concern in practice for robotic applications where the model needs to run as fast as possible once deployed.

Behavioral Cloning
Behavioral cloning is a supervised technique that can be used with any DNN to train a robot controller. First, an expert remotely controls the robot to perform a task many times. Everything is recorded into a dataset $\mathcal{D} = \{x_i, y_i\}$, where the x_i are sensor readings and the y_i are expert commands. We then use this dataset to train a DNN as seen previously. The resulting DNN maps sensor observations to expert commands. We call this DNN a *policy network* and denote its output π_θ instead of f_θ, by convention.

15.5 Policy Search for Robotic Control

15.5.1 Limitations of Supervised Learning for Control

While behavioral cloning enables learning a policy, it is inherently limited by its supervised nature. Behavioral cloning only tries to "imitate" the expert policy from a dataset of demonstrations, and thus it is unable to really match (let alone outperform) the expert level. Moreover, expert demonstrations are likely to be concentrated around a small number of interesting trajectories from which they never deviate. Robots not being perfect, they do deviate from these known trajectories and get lost in unexplored situations.

This is where the paradigm of *policy search* comes into play. Unlike supervised methods, policy search algorithms learn from their own experience. They are not limited by the expert level, and they learn in a way that is arguably closer to how natural intelligence arises. For example, evolutionary algorithms are inspired from natural genetics. They learn their own policy by continuously applying random mutations to their model, evaluating the new performance on the task after each mutation, and choosing to keep or discard the new model based on this evaluation. These algorithms

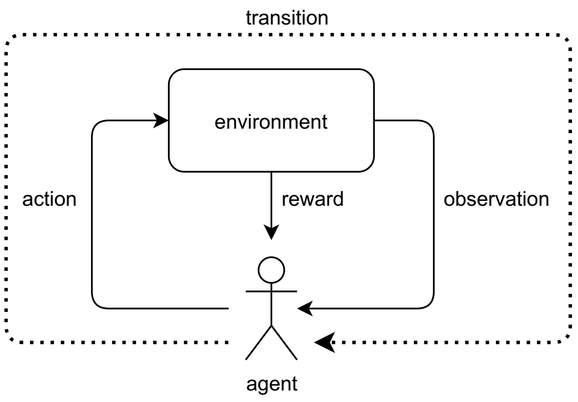


Fig. 15.13 RL transition

are able to easily find reasonable solutions to theoretically very complex scenarios, such as multi-robot tasks. However, because their mutations are random, they are often less efficient than the informed mutations performed by SGD. Deep reinforcement learning is a class of policy search algorithms able to find high-performance policies by leveraging SGD instead.

15.5.2 Deep Reinforcement Learning

Deep reinforcement learning is the modern conjunction of deep learning and RL (Sutton & Barto, 2018), a subfield of ML inspired from behavioral psychology, and more particularly from the concept of *reinforcement*. Reinforcement partially explains living organisms' behavior as the result of a history of positive and negative stimuli. Simply put, when facing a given situation, living organisms would try different actions and experience different outcomes. When later facing similar situations, they would become more likely to retry the actions that yielded the best outcomes and less likely to retry the actions that yielded the worst. RL emulates these outcomes by mean of a *reward* signal, which is a measure of how well the robot performs.

Interaction with the Environment

In RL, the world is framed as a special type of state machine called *Markov decision process* (MDP), or, less formally, *environment*[20] (Fig. 15.13).

Robotic environments usually discretize time into timesteps. At each timestep, the robot, called *agent*, retrieves an *observation* of the current state of the environment and uses its policy to compute an *action* from this observation. Once the action is computed, it is applied in the environment, which *transitions* to a new state. The agent observes this new state and receives an instantaneous reward. The new observation

[20] More precisely, real-world environments are usually partially observable Markov decision processes.

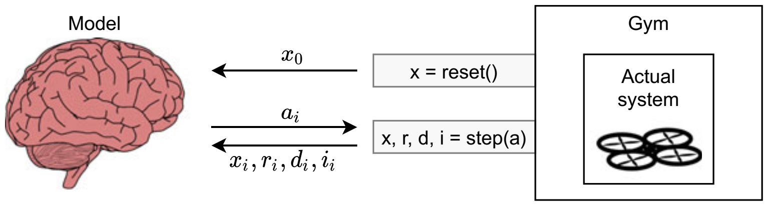


Fig. 15.14 Gym interface

can then be used to compute a new action from the agent's policy and so on. As implied by the name, the observations outputted by an MDP must have the Markov property. In other words, the observation actually fed to the policy must contain the whole history of past observations reasonably relevant to the task. Alternatively, the policy can be an RNN …

In practice, most RL environments are implemented using the Gym Python interface (Brockman et al., 2016) (Fig. 15.14).

The Gym interface is very simple and essentially consists of two methods. The *reset* method sets the environment to its initial state and returns an initial observation. The agent computes an action from this observation and feeds this action to the *step* method. The step method performs a transition of the environment and returns four values: a new observation, an instantaneous reward, a Boolean that tells whether the task is complete, and a Python dictionary that can usually be ignored (it may contain debugging information).

Importantly for robot applications, note that MDPs do not naturally take real-time considerations into account. The world is simply "stepped" from one fixed state to the next, and time is supposed to be "paused" between transitions. We can achieve a real-time behavior by means of a timer that clocks our Gym environment, but this comes with delayed dynamics that we need to handle properly.[21]

Reinforcement Learning Objective

The general philosophy of RL is to maximize the accumulated reward signal that the agent gets from the environment. More precisely, we want to find an optimal policy which, from any given observation, maximizes the sum of future rewards that the agent can *expect*. In other words, we want to find optimal parameters θ^* for the policy of the agent, such that:

$$\theta^* = \operatorname{argmax} \sum_{t=t_0}^{\infty} \mathbb{E}[r_t],$$

where $\mathbb{E}[r_t]$ is the *expectation* over the instantaneous reward r_t received from the environment at timestep t when following the policy π_{θ^*}, starting from an arbitrary initial observation.

[21] For a helper that handles these dynamics automatically, see rtgym (pypi.org/project/rtgym/).

Note that this sum can be infinite (e.g., if $r_t > 0$ for all t), and thus θ^* may be undefined. To alleviate this issue, we introduce a *discount factor* $0 \leq \gamma < 1$ in the rewards. This hyperparameter is very often used in RL, and understanding its role is important. Instead of trying to achieve the maximum sum of expected rewards, the agent tries to achieve the maximum sum of expected γ*-discounted* rewards:

$$\theta^* = \operatorname{argmax} \sum_{t=t_0}^{\infty} \mathbb{E}[\gamma^t r_t].$$

Since $\gamma < 1$, this makes the optimal policy relatively *greedy*: instead of optimizing for long-term rewards, we give more importance to the rewards that are not too far in the future. The closer γ is to 0, the greedier the optimal policy becomes. We usually set γ very close to 1 (0.95 or 0.99 are common values), but the effect is still noticeable. For instance, let us imagine we design the reward to be 0 everywhere except for the single timestep when the robot completes the task, where the reward is 1. Without a discount factor (i.e., with $\gamma = 1$), any policy would be optimal as long as it completes the task someday, and it should take ten thousand years. On the other hand, when $\gamma < 1$, the only optimal policies are the ones that complete the task as fast as possible.

Several ways of maximizing this sum exist. We will focus on an algorithm published in 2015, called "Deep Q-Network" (DQN) (Mnih et al., 2015), because it introduces many of the basic concepts that more advanced deep RL techniques used nowadays.

Deep Q-Network Policy

In the DQN algorithm, we train a near-deterministic policy that maps complex observations to discrete actions.[22] For this matter, we use the concept of *Q-value*.

In RL, the *state-value* function $V_{\pi_\theta}(x)$ maps the observation x to the sum of γ-discounted rewards that the agent can expect from following its current policy π_θ after observing x. It is defined recursively as follows:

$$V_{\pi_\theta}(x) = \mathbb{E}_{a \sim \pi_\theta(\cdot|x)} \mathbb{E}_{x', r' \sim p(\cdot|x,a)} [r' + \gamma V_{\pi_\theta}(x')],$$

where p is the transition distribution of the environment, i.e., the statistical distribution of the new observation x' and the new reward r' when observing x and taking action a. The policy π_θ is also written as a distribution because it is not deterministic. The Q-value, or *action-value* function, is almost the same thing, except it "forces" the first action:

$$Q_{\pi_\theta}(x, a) = \mathbb{E}_{x', r' \sim p(\cdot|x,a)} [r' + \gamma V_{\pi_\theta}(x')].$$

[22] Determinism is not really suitable for robotics in practice: We may get stuck in unseen situations. More advanced deep RL techniques are able to train stochastic policies that output real-valued actions.

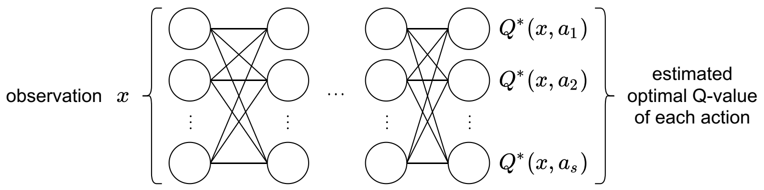


Fig. 15.15 Deep Q-network model

In plain English, the Q-value $Q_{\pi_\theta}(x, a)$ is the sum of discounted rewards that the robot can expect when it observes x, takes action a, and then follows it current policy π_θ ever after. We use the Q-value to discriminate good and bad actions from a given observation.

Let us imagine that we magically have access to the optimal Q-value function Q^*, i.e., the Q-value function under the optimal policy π_{θ^*}. When actions are discrete, the optimal policy π_{θ^*} is obviously to choose the action with the highest Q^* at each timestep, i.e.,

$$\pi_{\theta^*}(x) = \underset{a}{\operatorname{argmax}} Q^*(x, a).$$

This is exactly how the DQN policy works. We train a DNN with parameters θ that maps observations to the optimal Q-value Q^* of each action (Fig. 15.15).

Once the DQN model is trained, the robot uses it with the observation received from the environment at each timestep. The optimal policy π_{θ^*} is simply to select the action with the highest estimated Q^*. Ties are broken randomly, which is why the DQN policy is not entirely deterministic.

Deep Q-Network Training

Of course, we do not magically have access to the optimal Q-value function for training our DQN model in practice. But there is a well-known method for approximating this function: *Q-learning*. The previous equations can be combined into the following form:

$$Q^*(x, a) = \mathbb{E}_{x', r' \sim p(\cdot|x,a)}[r' + \gamma \max_{a'} Q^*(x', a')].$$

This identity, called the *Bellman equation*, is very important for RL. It enables identifying the optimal Q-value function by performing *Bellman backups*. For any transition (x, a, x', r') performed in the environment, we can improve our approximator (e.g., DQN model) of the optimal Q-value function with a simple operation:

$$Q^*(x, a) \leftarrow r' + \gamma \max_{a'} Q^*(x', a')$$

Let us take a moment to unveil the full potential of this operation. The Bellman backup improves the estimate of $Q^*(x, a)$ by aggregating the actual reward r' and the estimated best Q^* under the next observation x'. We can select any transition

(x, a, x', r') to perform this backup, as long as the transition has been sampled from the environment at some point. This includes transitions collected under a policy that is not the current policy of the robot. In particular, we can use transitions collected by an expert, or simply by an older version of the current policy. In RL, this important property is called *off-policy*. An off-policy algorithm, such as DQN, is able to improve the current policy with transitions collected under another policy. This is in contrast to *on-policy* algorithms, which can only use transitions collected under the current policy. The main reason why the off-policy property is important is that it enables reusing old transitions several times at different stages of training. This is particularly important in robotic applications, where it is costly to collect environment transitions.

We want to use the Bellman backup for training our DQN model, which is a DNN. Therefore, we need to translate this backup into a loss function, so as to enable SGD. This is pretty straightforward: We can simply use the MSE loss. We select a transition (x, a, x', r'), we feed x and x' separately to our DQN model so as to retrieve $Q^*(x, a)$ and $Q^*(x', \cdot)$, and then we perform an SGD step for the $Q^*(x, a)$ output only, using the following loss:

$$L_{\text{MSE}}(Q^*, \{(x, a, x', r')\}) = |Q^*(x, a) - (r' + \gamma \max_{a'} \underline{Q}^*(x', a'))|^2$$

The reason why we underline Q^* in the right-hand part of this equation is a bit subtle. Notice that, in supervised learning, the quantity between parenthesis would correspond to the ground truth label y, which would be a constant. Here, however, this quantity depends on the parameters of the DNN, because the DNN is used to compute $Q^*(x', \cdot)$. Thus, if we are not careful, performing SGD will modify the DNN parameters such that our "ground truth" gets closer to our estimate rather than the other way round! To avoid this issue, we do not back-propagate gradients through the computation graph of $\underline{Q}^*$.[23] However, this leaves yet another issue: Updating our DNN parameters still updates the very target that we are trying to reach! Indeed, we are updating these parameters such that our estimate of $Q^*(x, a)$ gets closer to the quantity between parentheses, but this collaterally changes $\underline{Q}^*$ and thus this very quantity, making training unstable. We instead keep an old copy of our DQN model that we periodically update. This copy, called the *target network*, is used to compute $\underline{Q}^*(x', \cdot)$. Since it is only updated once in a while, it becomes easier to track.

It is straightforward to generalize our loss to minibatch gradient descent. Using a minibatch $\mathcal{M} = \{(x_i, a_i, x'_i, r'_i)\}$ of n transitions, the MSE loss becomes

$$L_{\text{MSE}}(Q^*, \mathcal{M}) = \frac{1}{n} \sum_{i=1}^{n} |Q^*(x_i, a_i) - (r'_i + \gamma \max_{a'_i} \underline{Q}^*(x'_i, a'_i))|^2.$$

Since DQN is off-policy, we can sample these transitions randomly from a dataset, as we would do in supervised learning. Sampling from a fixed dataset (e.g., of expert demonstrations) is possible and known as *offline RL*. Theoretically, this could match

[23] In PyTorch and TensorFlow, the "no gradient" option ensures this constant-like behavior.

and even outperform the expert level, because we are not imitating the expert anymore: We are literately learning from their experience. However, this approach comes with practical difficulties, in particular because it tends to overestimate the value of unexplored situations. We often prefer letting the agent collect its own experience, by *exploring* the environment while learning. In this situation, there is virtually no theoretical limit to how good the agent can get.

In DQN, the agent explores its environment by following an *ϵ-greedy* policy: With a certain probability ϵ, the agent selects a random action, and with probability $(1 - \epsilon)$, it selects the best action as estimated by the DQN model. This scheme *exploits* the current knowledge of the environment by drawing exploration toward promising trajectories only. A higher ϵ yields a higher tendency to explore and thus slower convergence, whereas a smaller ϵ yields faster convergence at the price of being more likely to converge to local optima, i.e., to poor policies. This trade-off is known as the *exploration/exploitation dilemma*.

In practice, we store the transitions collected by the agent in a huge circular dataset called *replay buffer* (1–100 million transitions are common sizes). In parallel, we randomly sample minibatches of transitions from this buffer, and we train our DQN model by minimizing the $L_{\mathrm{MSE}}(Q^*, \mathcal{M})$ loss via SGD.

15.5.3 Improvements of Deep Q-Learning

DQN was published in 2015 and has popularized the idea of deep Q-learning, which is leveraged in many state-of-the-art deep RL algorithms nowadays. Over the years, improvements have been introduced that increased the performance of these methods. In particular, we usually dampen the updates of the target network and train two DQN models in parallel.

Slowly Moving Target
Using an old copy of our DQN model that we only periodically update makes the target easier to track. An even better strategy is to update the target slowly in a dampened fashion, by mean of an *exponentially moving average*. Instead of only periodically updating the target parameters, we update them with each SGD step, but only by pulling them weakly toward those of the current DNQ model. More precisely, the parameters of the target are updated according to:

$$\underline{\theta} \leftarrow (1 - \tau)\underline{\theta} + \tau\theta,$$

where θ are the parameters of the current Q-network, $\underline{\theta}$ are those of the target network, and τ is a small attraction coefficient (commonly 0.005). With this strategy, the $\underline{Q}^*$ target does not move erratically due to the stochasticity of SGD updates.

Double Q-Networks
Another well-known issue of the original DQN algorithm is that the Q-network tends to converge to overestimated values for some actions. This issue, known as the

overestimation bias, comes from the fact that the target network $\underline{Q}^*$ is an estimator and has a noisy error. When selecting the maximum $\underline{Q}^*$ over all actions, we also select the maximum over … the noisy error!

To tackle this issue, we train two DQN models in parallel, with different sets of initial parameters. This yields two target networks, each with different error distributions. In recent algorithms, we compute the value of each action as its minimum across both networks, which cancels the overestimation bias.

15.5.4 Deep Reinforcement Learning for Practical Applications

Despite DQN being relatively recent (2015), the research effort has been so intense in the deep RL community over the last few years that it is already vastly outperformed by more advanced techniques.

– Soft Actor Critic (SAC). Since its publication in 2018, SAC (Haarnoja et al., 2018a, 2018b) has been one of the main players in deep RL for robotics. SAC is an *Actor Critic* algorithm. This means that two neural networks are trained in an interleaved fashion: an *actor* (policy network similar to behavioral cloning) and a *critic* (Q-network similar to DQN). The policy trained by SAC is stochastic and able to output continuous actions. Moreover, its entropy[24] is maximized in parallel to the RL objective. This yields a policy that is very robust to unseen situations, which is particularly important in the real world. SAC is also an off-policy algorithm.
– Model-based reinforcement learning. More recently, another class of deep RL algorithms, called *model based*, has seen a dramatic surge of interest from the community. Model-based algorithms use a model of the world to predict the response of the environment from a given observation. In state-of-the-art approaches, this model is learnt from interacting with the environment. Once learnt, the policy can be trained without interacting with the environment anymore, and the model can be used for planning. The MuZero (Schrittwieser et al., 2020) algorithm is currently the winning player in this field.
– Non-stationary environments. Let us point out a real-world concern that we have silenced so far. We have focused on off-policy algorithms because these have tremendous advantages in situations where the collection of transitions is costly, which is typically the case in robot applications. However, “naive” off-policy algorithms only work when the environment is stationary. Indeed, these algorithms rely on a dataset of past transitions to train their current policy. But think of what happens in the real world, where other agents are continuously learning and changing their behavior. Old transitions may become obsolete, and learning from those may become counterproductive. In this situation, you might want to draw inspiration from on-policy approaches such as Proximal Policy Optimization (PPO) (Schulman et al., 2017), which only rely on the present.

[24] The entropy is the amount of randomness of a stochastic function.

– Sim-to-real. It is often practical to train an RL algorithm in simulation, especially for robotics. RL being fundamentally based on trial-and-error, we do not want our robot to break because of crazy random actions during training. Simulation instead provides a safe environment where arbitrarily bad actions can be tried out. Moreover, it is possible to accelerate time in simulation and collect transitions faster than real time. But this comes at a price: Simulation is never the same as reality, and a policy trained in simulation typically fails in the real world. This concern, called the *sim-to-real* gap, is one of the main challenges that has long kept deep RL from being really useful in practical robotics. However, it has recently been alleviated impressively by techniques such as Rapid Motor Adaptation (RMA) (Kumar et al., 2021), which uses a combination of deep RL and supervised learning for this matter.

15.6 Wrapping It Up: How to Deeply Understand the World

Deep RL is an elegant illustration of how deep learning enables a robot to produce its own understanding of the world. When learning a task such as driving autonomously from camera images through deep RL, the robot is never explicitly told what a car, a pedestrian, or a road is. Instead, it learns these concepts on the fly, only from maintaining its own belief about which decisions are good or bad for each possible observation. This belief is formed by the principle of gradient back-propagation in a DNN. In the case of DQN, this DNN is the DQN model, which can be seen as a black box, mapping observations to the corresponding Q-value of each available action. For instance, an image in which there is a pedestrian would likely be mapped to low Q-values for actions that lead to run over the pedestrian and to higher Q-values for actions that do not. This implies that DQN builds its own way of detecting pedestrians (with convolutional kernels that detect pedestrian features for instance) and grasps a certain understanding of what a pedestrian is, how it moves, how it interacts with the world, etc. Now, there is no magic at play here: This understanding is entirely statistical, and it is defined as the complex projections performed by the DNN in its successive layers of artificial neurons. How this fundamentally differs from a human understanding of the world, however, is a real question.

15.7 Summary

In this chapter, an introduction to the fundamentals of modern machine learning has been provided, in its aspects most relevant to robotics. We have started from simple linear regressions and have built our way up to highly expressive and nonlinear deep neural networks. This allows us to approximate complex mappings, such as optimal controls from sensor readings. Furthermore, we have introduced the basics of deep reinforcement learning, a popular approach that enables robots to look for such controls autonomously. In the course of this chapter, we have seen how a dataset can be used for training a model and how it is possible to ensure that this model generalizes well. Finally, we have introduced state-of-the-art supervised models that the reader can use out of the box for robotic perception and state-of-the-art reinforcement learning algorithms that are becoming increasingly relevant for robot control. Keep in mind however that all these techniques are uncertain in nature, due to the use of statistical approximators (e.g., neural networks). Safety is therefore always an important concern when applying modern ML techniques in the real world.

15.8 Quiz

Please find the quiz for this chapter in the Jupyter notebooks available online.[25]

15.9 Further Reading

To learn more about deep learning, we recommend "Deep Learning" (Goodfellow et al., 2016) by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. To learn more about reinforcement learning, we recommend "Reinforcement Learning: An introduction" (Sutton & Barto, 2018) by Richard Sutton and Andrew Barto. Both references are available online for free.

References

Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). Yolov4: Optimal speed and accuracy of object detection. arXiv preprint arXiv:200410934

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. arXiv preprint arXiv:160601540

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J, Winter, C., ... Amodei, D. (2020). Language models are few-shot learners. arXiv preprint arXiv:200514165

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:14123555

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 248–255). IEEE.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:181004805

[25] https://github.com/Foundations-of-Robotics/ML-Quiz.

Elharrouss, O., Almaadeed, N., Al-Maadeed, S., & Akbari, Y. (2020). Image inpainting: A review. *Neural Processing Letters, 51*(2), 2007–2028.

Fitch, F. B. (1944). McCulloch Warren S. and Pitts Walter. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, vol. 5, pp. 115–133. *Journal of Symbolic Logic, 9*(2).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018a). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning, PMLR* (pp. 1861–1870).

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., & Levine, S. (2018b). Soft actor-critic algorithms and applications. arXiv preprint arXiv:181205905

He, F., Liu, T., & Tao, D. (2019). Control batch size and learning rate to generalize well: Theoretical and empirical evidence. *Advances in Neural Information Processing Systems, 32*, 1143–1152.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2961–2969).

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks, 4*(2), 251–257.

Ivakhnenko, A. G., & Lapa, V. G. (1965). *Cybernetic predicting devices*. CCM Information Corporation.

Kendall, A., Grimes, M., & Cipolla, R. (2015). PoseNet: A convolutional network for real-time 6-DOF camera relocalization. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2938–2946).

Kingma, D. P.,& Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:14126980

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (Vol. 25, pp. 1097–1105).

Kumar, A., Fu, Z., Pathak, D., & Malik, J. (2021). RMA: Rapid motor adaptation for legged robots. arXiv preprint arXiv:210704034

Linnainmaa, S. (1970). *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors* (Master's Thesis), University of Helsinki, pp. 6–7 (in Finnish).

Long, X., Deng, K., Wang, G., Zhang, Y., Dang, Q., Gao, Y., Shen, H., Ren, J., Han, S., Ding, E., & Wen, S. (2020). PP-YOLO: An effective and efficient implementation of object detector. arXiv preprint arXiv:200712099

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., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533.

Moon, G., Chang, J. Y., & Lee, K. M. (2018). V2V-PoseNet: Voxel-to-voxel prediction network for accurate 3D hand and human pose estimation from a single depth map. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5079–5088).

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). Determination Press.

Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., & Kavukcuoglu, K. (2016). WaveNet: A generative model for raw audio. arXiv preprint arXiv:160903499

Pham, H., Dai, Z., Xie, Q., & Le, Q. V. (2021). Meta pseudo labels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 11557–11568).

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 779–788).

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T, & Silver, D. (2020). Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature, 588*(7839), 604–609.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:170706347

Stigler, S. M. (1981). Gauss and the invention of least squares. *The Annals of Statistics,* 465–474.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.

Tan, M., & Le, Q. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In: *International Conference on Machine Learning, PMLR* (pp. 6105–6114).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998–6008).

Wang, M., & Deng, W. (2018). Deep visual domain adaptation: A survey. *Neurocomputing, 312*, 135–153.

Wang, Z., Chen, J., & Hoi, S. C. (2020). Deep learning for image super-resolution: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2223–2232).

Yann Bouteiller is an engineer from École des Mines de Saint-Étienne (France) working as a research associate at the Computer Science Department of Polytechnique Montreal (Canada). His research focuses on machine learning (ML) and more specifically on designing deep reinforcement learning algorithms for real-world applications. At the junction between theoretical and practical ML, he aims at facilitating the transfer of recent, simulation-based deep learning successes to the industry. His work includes advances in reinforcement learning theory as well as practical deep learning-based advances in neuroscience, autonomous driving, robot learning, video games, real-time embedded systems, etc.