WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science

BACHELOR THESIS

SUPERVISOR:
Prof. Dr. Daniela Zaharie

GRADUATE:
Radu-Florin Ciobanu

TIMIŞOARA
2024

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science

Policy Gradient Methods: Insights and Optimization Strategies

SUPERVISOR:
Prof. Dr. Daniela Zaharie

GRADUATE:
Radu-Florin Ciobanu

TIMIŞOARA
2024

Abstract

Reinforcement Learning (RL) represents one fundamental paradigm within the realm of machine learning, dedicated to iteratively approach solutions in an experimental manner, with the objective of maximizing the cumulative reward of an autonomous agent given by an associated environment in which it performs. In contemporary contexts, recalling that RL is the most viable path towards physical artificial intelligence, evolutionary and genetic algorithms using non-differentiable neural networks have yielded precedence to the enhanced performance of Monte Carlo and Temporal Difference (TD) techniques employing usage of expressive approximators such as feedforward neural networks. This paper studies one subclass that directly refines agents' policies through gradient ascent, the Policy Gradient Methods (PGM) branch, sustained by a meticulous ground-up implementation and efficiency strategies, showcasing its relevance in the current state of the art of deep reinforcement learning.

Contents

Chapter 1

Introduction

> "A computer would deserve to be called intelligent if it could deceive a human into believing that it was human."
>
> — Alan Turing

Modern machine learning techniques implies the use of any function approximators for problems described as a function that cannot be expressed directly as a simple to put down equation. In supervised learning, the process of shaping a function that delivers good predictions on unseen data is straight-forward: architecture a model and initialize it randomly, decide which loss function to use depending on whether it's a regression or classification problem, then optimize the parameters of the model gradually using any gradient descent algorithm that you see fit until convergence. In the last decades, the research attended interest for the development of deep neural networks for their formidable performance of discern complex non-linear dependencies, in comparison with linear models, decision trees or support vector machines (SVM), driven by the continuously growing complexity and scale of real-world problems, as well as the increasing availability of large datasets and computational resources. By having at our disposal a large dataset we basically have a partial access to the objective function we are trying to approximate, the only task remained is to generalize our estimate by regularizing the model between data key-frames using different methods, like data augmentation or regularization functions.

Mapping inputs to outputs it's not as easy when we do not have a target output that we are trying to learn, thus by deciding a reward function assembled as an environment that manages to offer an agent a general clue to the right improvement direction, we can estimate a solution by interfering more or less stochastically within that training environment. Tournament based methods between mutated policies depicted the naive incipient options, further expanded towards topology preservation as in NEAT [SM02]. One does showing scalability capabilities, so the research left derivative free optimization (DFO) in exchange for algorithms that make use of differentiable neural networks as a better approach that still keeps the focus on the policy.

In recent years, transcending the old days of achievements such as AlphaGo by Google DeepMind and the extraordinary performance exhibited in video games through RL, research endeavors have increasingly ventured into hybrid approaches that amalgamate both supervised learning and RL. A notable example of this fusion is evident in large language models where training methodologies draw inspiration from a dual paradigm, leveraging the strengths of both supervised learning, with its large dataset availability ran on Transformer architecture [VSP+17], and RL, with its capacity to learn through human feedback and even surpass the average human-level capabilities.

1.1 Motivation

Attempting to implement a system that does replicate either a simple supervised learning training or even a state-of-the-art deep reinforcement learning algorithm will end up with a limited knowledge overhead if there is no prior understanding or differential programming dependencies. This paper proposes to deliver a comprehensive explanation of the inner processes behind deep neural networks (presented in Chapter 2), moreover, provides all needs to successfully rise a DRL training tool. Chapter 3 is planning to give implementation details and loss differentiation explanation for two different state-of-the-art algorithms, Proximal Policy Optimization [SWD+17] and Soft-Actor Critic [HZAL18], alongside various improvement techniques inspired from their authors reference implementation. In the appendix, brief descriptions are provided for two other algorithms, Deep Deterministic Policy Gradient [SLH+14] and Twin Delayed DDPG [FHM18], distinguished by their use of deterministic policies. Naturally, this paper includes reference to a comprehensive implementation from the ground up which can be found at https://github.com/smtmRadu/DeepUnity that thoroughly encompasses all the information elucidated in these chapters in C# programming language packaged as a framework called DeepUnity, without using an external tensor library with differentiable programming support. Last but not least, packaged within the repository, there are RL environments examples with 3D modeled agents running by the means of Unity physics system, trained with DeepUnity using custom setups, personal modular utilities and experimental methods for optimization.

1.2 Overview and Objective

Reinforcement learning involves two key components: the agent and the environment. The agent, acting as a character to which is assigned a "brain", interacts within a world crafted by its creators—the environment. This world is purposefully designed, featuring obstacles and rewards to facilitate learning. Successfully solving a reinforcement learning problem occurs when the agent accumulates a predetermined amount of rewards or surpasses initial expectations within a defined timeframe, referred to as maximizing the expected reward. The agent's interaction with the environment unfolds through transitions between states, triggered by specific actions. Each transition yields a reward, a numerical signal reflecting the quality of the action—a principle encapsulated in the Markov Decision Process (MDP). Temporal difference learning dictates that all preceding actions contributing to a reward are deemed favorable. Even after achieving success, the agent persists in exploring the environment, seeking to unveil concealed elements.

Deep Reinforcement Learning grapples with the challenge of balancing exploration and exploitation. Unlike earlier scenarios where noise facilitated exploration, introducing excessive noise in deep reinforcement learning may impede meaningful learning. In this context, the agent executes actions with limited understanding, potentially hindering the comprehension of their efficacy.

The aim of this paper is to elucidate the functioning of agent's "virtual brain" in the 2^{nd} Chapter, detailing its implementation from the ground up through various techniques tailored for specific agents. Moving forward to the 3^{rd} Chapter, the focus shifts to the process of enabling the agent to learn, incorporating a discussion on state-of-the-art algorithms, finally describing in the 4^{th} Chapther the practical part of the project: the reference implementation, with already trained agents as examples.

Chapter 2

Deep Learning

This chapter introduces notions regarded to the necessary deep learning tools and their implementation respectively. Multi-dimensional non-linear problems performing with an unmanageable space of states and actions are no longer solvable using classic tabular methods, linear regression or default algorithms, typically requiring an advanced alternative to approximate the function by which it's defined. State of the art solutions imply the use of differentiable neural networks models, including all their variations dedicated to a specific type of information. In RL, the agent may deal, based on the environment he acts, with diverse classes of input observations, that must be approached using the right architecture. Popular frameworks like PyTorch or TensorFlow provide a tensor library with automatic gradient computation for deep learning, though this paper proposes not only to expound the basics, but also to unveil the reasons and logic behind common utility components. Starting with Section 2.1 we dive into an implementation-wise explanation of deep neural networks and other related tools that form a general foundation of usage, especially in relation with any DRL algorithm. The reference implementation can be found at https://github.com/smtmRadu/DeepUnity, which integrates the usage of deep learning tools within *Unity* game engine for simulating internally the behaviour of physical, intelligent agents.

2.1 Neural Networks

The scope of the neural network is to approximate the objective function $\mathcal{J}(\theta)$, or strictly said, to minimize the mean squared error (MSE) over the number of training samples m on the parameters of the network defined by θ (check Equation 1). Note that we introduce the fraction $1/2$ in the equation in order to simplify the math, knowing that the derivative of the MSE loss function with respect to the output y is $2 \cdot (\hat{y} - y)$ (see Section 2.3.1). Why don't we take the absolute value (MAE) of the difference between $\hat{y}$ and y , or even to raise the difference to the fourth power? The problem of mean absolute error is that it is not continuously differentiable, and does not penalize large errors in the network with large gradients as MSE does. On the other hand, raising the error to the fourth power on the other hand is less computationally efficient, and there are other solutions for that (see Cross Entropy in Subsection 2.3.1). The error minimization over the parameters θ is done by using the Gradient Descent algorithm, explained in Section 2.3.2.

$$\mathcal{J}(\theta) = \underset{\theta}{\text{minimize}} \frac{1}{2m} \sum_{i=1}^{m} (\hat{y_i} - y_i)^2 \tag{1}$$

For the sake of simplicity, we describe a deep neural network as a sequence of layers that flow data between them in two ways, forwards and backwards, as in Figure 1.

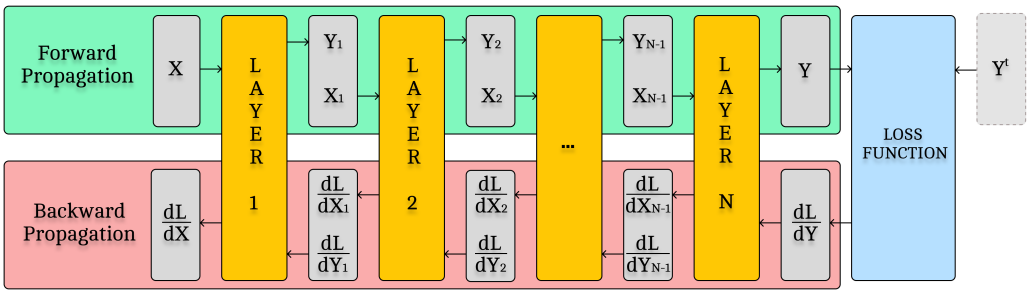


Figure 1: Layers communication of X and $\frac{\partial L}{\partial X}$

We propagate forward the input X to obtain the prediction $\hat{Y}$ (see explanation in Section 2.1.1). The error (also known as loss or cost) is calculated between the output $\hat{Y}$ of the network and Y^t (the label value our model wants to target). Afterwards, the model's parameters must be modified θ accordingly, by subtracting their gradients (that form the Jacobian matrix). The Jacobian matrix cannot in one swoop, so Section 2.1.2 explains how the loss is backpropagated, letting each of the layer to compute the partial derivatives of the loss function with respect to their parameters.

Algorithm 1 Sequential Neural Network, *Forward* and *Backward* methods implementations without tensor automatic gradient computation. Layers (see Section 2.2) are individually responsible for computing the gradients of their own parameters ∇_θ (if they have trainable parameters) and to return the derivative of the loss function L with respect to the previous layer's output in consequence to the chain rule.

Require: $\theta = l_{\theta_1}, l_{\theta_2}, \ldots, l_{\theta_n}$: Layer sequence of the network

procedure FORWARD(x)
  $y \leftarrow x$
  for each $l \in \theta$ **do**
    $y \leftarrow l.Forward(y)$
  end for
  return y
end procedure
procedure BACKWARD($\frac{\partial L}{\partial y}$)
  $\frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial y}$
  for each $l \in \theta$ in reverse order **do**
    $\frac{\partial L}{\partial x} \leftarrow l.Backward(\frac{\partial L}{\partial x})$
  end for
  return $\frac{\partial L}{\partial x}$
end procedure

2.1.1 Forward Propagation

The process of forward propagation involves the traversal of an input vector through the function encapsulated by our neural network. Expanding on this conceptual notion, the input undergoes sequential parsing through each layer, with each layer serving as a distinct function, particularly evident in the case of a straightforward sequential deep neural network. It's worth noting that deep models may incorporate layer skips. In practical terms, an input represented by x is subjected to activation by the initial layer of the neural network, producing an output denoted as $\hat{y}$. This output then serves as the input for the subsequent layer, and this sequential process continues iteratively until the final layer's output is obtained. This progression is illustrated in Figure 1. This sequential activation through layers allows the network to progressively capture intricate features and representations within the input data. Each layer refines the information passed from the preceding one, enabling the network to learn hierarchical representations comparing to a simple artificial neural network (ANN), that consists of a single hidden layer. The intricate interplay of parameters and activation functions in each layer contributes to the network's ability to discern complex patterns, furtherly generalized in order to make good predictions on unseen data. An aspect that is not covered in the following subsections algorithmic representations is that in the case of an implementation that does not uses an autograd system, the input received by each layer must be cached in order to compute the derivative of the loss function with respect to that input.

2.1.2 Backward Propagation for Gradients Computation

Backward propagation, or backpropagation, is the complementary process to forward propagation that makes half of the magic behind the actual learning process of neural networks. While forward propagation moves input data through the network to produce an output, backpropagation involves computing the gradients of the parameters ∇_θ to thereafter update the model's parameters θ to minimize the difference between the predicted output and the actual target. The fundamental idea behind backpropagation is to calculate the gradient of the loss function L with respect to the model's parameters efficiently. This involves applying the chain rule to compute how much each parameter contributed to the overall error. The loss (also called *error* or *cost*) will be minimized by subtracting a portion of the gradient (denoted as α, known as learning rate) from the actual parameter, this process being outlined in detail in Section 2.3, along with advanced optimization and acceleration methods.

The gradient of the loss function is propagated similarly with forward propagation but in reverse, as depicted in Figure 1. As a running example, we can suppose a sequence composed of n layers (functions) denoted as y_1, y_2, ... , y_n, parametrized by θ_1,θ_2, ... , θ_n respectively. Of course, most often one layer may have multiple parameters (weights and biases), but we abstract for a general case. Computing $\partial L/\theta_n$, which is the gradient of the last layer's parameter, cannot be computed directly, thus we need to expand the equation, making it directly computable:

$$\frac{\partial L}{\partial \theta_n} = \frac{\partial L}{\partial y_n} \cdot \frac{\partial y_n}{\partial \theta_n} \tag{2}$$

Computing $\partial L/\partial \theta_{n-1}$ works the same, by extending 2 formula as follows:

$$\frac{\partial L}{\partial \theta_{n-1}} = \frac{\partial L}{\partial y_n} \cdot \frac{\partial y_n}{\partial y_{n-1}} \cdot \frac{\partial y_{n-1}}{\partial \theta_{n-1}} \tag{3}$$

This process is available for all other parameters. It's important to remark that not all the layers have parameters (because typically parametrized layers are followed

by nonlinear activations, which are often non-parametrized) and, in Eq. 3 the first two partial derivatives of the product were already computed in Eq. 2 forming $\partial L/\partial y_{n-1}$ and there is no need for extra calculations so we can simplify it to:

$$\frac{\partial L}{\partial \theta_{n-1}} = \frac{\partial L}{\partial y_{n-1}} \cdot \frac{\partial y_{n-1}}{\partial \theta_{n-1}}$$

Now we can see inductively how the process runs sequencially for the rest of the parameters in this reversed order. Also, to avoid any potential confusion, may be necessary to clarify that the output of a layer y_i is the same entity with the input of the next layer x_{i+1}, as a result of the fact that we do not use the input notation in our equations.

$$\frac{\partial L}{\partial \theta_{n-2}} = \frac{\partial L}{\partial y_{n-2}} \cdot \frac{\partial y_{n-2}}{\partial \theta_{n-2}}$$

$$\vdots$$

$$\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial y_2} \cdot \frac{\partial y_2}{\partial \theta_2}$$

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial y_1} \cdot \frac{\partial y_1}{\partial \theta_1}$$

2.1.3 Weight Initialization

On any neural network's initialization, the parameters can be stochastically initialized from any kind of distribution (excepting *Batch Normalization* or similar layers, where γ and β are initialized with neutral values, see subsection 2.2.3). Though, the nonlinear activation function that follows the parametrized layer directly affects it's behavior depending on the scale of the parameters, that destabilizes the model leading to underfitting or even divergence from the solution. To overcome this issue, multiple initialization methods were introduced, which produces values relative to the features dimension of the layer, so we will define fan_{in} and fan_{out}, whom represent the size of the input features in the layer, and the size of the output features respectively. One was designed in [HZRS15] for non-saturated activation functions that may produce gradient explosion, defined in the following distributions:

$$\mathcal{N}(0, \sigma)\,, \text{ where } \sigma = \sqrt{\frac{2}{fan_{in}}} \qquad \text{(Kaiming - Normal distribution)}$$

$$\mathcal{U}(-k, k)\,, \text{ where } k = \sqrt{\frac{6}{fan_{in}}} \qquad \text{(Kaiming - Uniform distribution)}$$

The difficulty of training deep neural networks is strongly debated in [GB10] by observing the influence of the nonlinear activation functions and how random initialization is unsuited for deep neural networks with saturated activations like logistic sigmoid activation (because for high weight values the gradients vanish), coming up with the following initialization method:

$$\mathcal{N}(0, \sigma)\,, \text{ where } \sigma = \sqrt{\frac{2}{fan_{in} + fan_{out}}} \qquad \text{(Xavier - Normal distribution)}$$

$$\mathcal{U}(-k, k)\,, \text{ where } k = \sqrt{\frac{6}{fan_{in} + fan_{out}}} \qquad \text{(Xavier - Uniform distribution)}$$

A common practice when using $Xavier$ is to use a $Zero$ initialization for biases, like in TensorFlow Dense default settings. PyTorch uses *LeCun* initializer as it's default, which is similar to $Kaiming$ though the numerator is 1 and 3 respectively.

The orthogonal initialization of the weights matrix is worth discussing as it helps mitigate the vanishing and exploding gradients problem. This initialization works by Gram-Schmidt process of decomposing a matrix A into a product $A = QR$, where Q is orthogonal ($Q^TQ = I$) and R is an upper triangular matrix.

Algorithm 2 QR algorithm. Generates an orthogonal matrix Q and an upper triangular matrix R given fan_{in} and fan_{out}.

Initialize $A \in \mathcal{R}^{fan_{out} \times fan_{in}} \sim \mathcal{N}(0, 1)$
Initialize $Q \in 0_{fan_{out} \times fan_{in}}$
Initialize $R \in 0_{fan_{in} \times fan_{in}}$
for $i = 1, 2, \ldots, fan_{in}$ **do**
    $Q_i \leftarrow A_i$
    for $j = 1, 2, \ldots, i$ **do**
        $R_{j,i} \leftarrow Q_j^T A_i$
        $Q_i \leftarrow Q_i - R_{j,i} Q_j$
    end for
    $R_{i,i} \leftarrow ||Q_i||_2$
    $Q_i \leftarrow \dfrac{Q_i}{R_{i,i}}$
end for
return Q

2.2 Layers

A deep neural network comprises multiple layers, each with specific roles in learning intricate data patterns. These layers facilitate data flow within the network through two essential paths: $Forward$ (towards the next layer) and $Backward$ (towards the previous layer), as an entire deep neural network does. Layers can be categorized as either learnable (parametrized) or non-learnable (with fixed parameters or without parameters). Learnable layers are those that adapt and optimize their parameters θ during training to capture data patterns effectively, while non-learnable layers serve as fixed components with no trainable parameters, though they are in complement with the others in order to improve the training session and also the final performance of the model. Additionally, not all layers are directly chained together, there are different architectures, such as ResNets [HZRS16], that imply skip connections in order to overcome vanishing gradient problem on extremely deep neural networks, multi-headed networks sharing the same *backbone* or forks the outputs and joins them again like in Variantional Auto-Encoders [KW13].

In the upcoming subsections, we will delve into several common components that form the basics of deep learning, with direct utilization in reinforcement learning. The reference implementation includes more modules that are not covered in this paper, used in related DL problems. It's important to note that the $Forward$ method always preserves (caches) the input data x before applying the layer's function. This preservation is therefore used in the $Backward$ method to compute the partial derivative of the loss function L with respect to x. This gradient computation is necessary to be passed to the previous layers in order to update their parameters as well. Usually the loss function is reduced by mean over the batch, so in the forthcoming explanations, the gradients of the parameters will be averaged over the batch (we will denote the batch size as m).

2.2.1 Insight into Function Approximators

To build a neural network capable of capturing complex non-linear functions, we start by considering the universal approximation theorem [HSW89], which states that any multivariate continuous function arbitrarily well by a feed forward neural network with at least one hidden layer, provided it uses any squashing activation function and has a sufficient number of neurons in the hidden layer. So, for a function $f : \mathbb{R}^n \to \mathbb{R}$,

$$f(x) \approx \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) = \sigma\left(w^T x + b\right) \tag{4}$$

where $x = (x_1, x_2, \ldots, x_n)$ is the input vector, $W = (w_1, w_2, \ldots, w_n)$ is the weight vector, b is the bias (scalar), and σ any nonlinear function (see Subsection 2.2.2).

In a general sense, the input is linearly projected into a higher dimensional embedding (up linear projection) since our problems typically involve multiple input and output dimensions, thus w will be a extended to a matrix, and b to a vector. Interestingly, higher-dimensional hyperplanes of the data exhibits structures that are effectively simpler (even almost linear) compared to lower-dimensional spaces. This linear/affine transformation is integrated as a layer into a neural network, named *Linear*, *Dense* or *Fully Connected* layer, followed up by a nonlinear activation function. Stacking up linear and nonlinear transformations enables the neural network to learn and represent increasingly complex relationships within the data, making it capable of solving real-world nonlinear problems.

Algorithm 3 Linear Activation. Defined by H_{in} and H_{out}, where H represents the size of the input and the output respectively. For parameter initialization, see section 2.1.3. The Linear layer is typically followed by a nonlinear activation function.

Require: W: Weights
Require: B: Biases
Require: $\frac{\partial L}{\partial W}$: ∇_W - Weights' gradients
Require: $\frac{\partial L}{\partial B}$: ∇_B - Biases' gradients

Initialize $W \in \mathcal{R}^{H_{out} \times H_{in}}$ using an initializer (see Section 2.1.3)
Initialize $B \in \mathcal{R}^{H_{out}}$ using an initializer (see Section 2.1.3)
procedure FORWARD(x)
    $y \leftarrow xW^{\mathrm{T}} + B_{expanded}$ ▷ Biases are expanded on the batch dimension
    return y
end procedure
procedure BACKWARD($\frac{\partial L}{\partial y}$)
    $\frac{\partial L}{\partial W} \leftarrow (\frac{\partial L}{\partial y})^T \frac{1}{m} \sum_{i=1}^{m} x_i$
    $\frac{\partial L}{\partial B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$
    $\frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial y} W$
    return $\frac{\partial L}{\partial x}$
end procedure

2.2.2 Nonlinear Activation Functions

As previously explained in Subsection 2.2.1, the nonlinearity is essential for the expressivity of the model to learn the intricate representation of our problem. It gives the ability of the approximation function to capture complex shapes that can align with the objective function, inserted as a layer that follows any other layer in the network. Since deep neural networks are designed on differentiation, all activation functions must be differentiable (not necessary continuously differentiable). To backpropagate the loss of the function with respect to the input of a nonlinear function $f(x)$ layer, we need to compute:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f(x)} \cdot \frac{\partial f(x)}{\partial x} \tag{5}$$

The first element of the equation is the partial derivative of the loss function L with respect to the output of the nonlinear layer $f(x)$ (received as parameter on the *Backward* method), and the second one is the partial derivative of the activation function wrt. the input x. It is important to note that if we are using a parametric activation layer (with trainable parameters), we also need to compute first the partial derivative of the loss function with respect to that parameter $\partial L/\partial \theta$, before continuing the backpropagation of $\partial L/\partial x$ to the subsequent layer.

The easiest way to make a nonlinear function is to create one from two linear functions, by rectifying the linear activation in $x = 0$; we call it Rectified Linear Unit, or ReLU for short. This activation is one of the most popular ones due to its low computational efficiency, but it comes with its problems rised from its non-continuous differentiability and zero thresholding. Since it is non-saturated towards $+\infty$, it leads to numerical instability and exploding gradients on large models if wrong weight initialization methods are used (Kaiming initialization is recommended, see Subsection 2.1.3).

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \qquad \mathrm{ReLU}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

The Hyperbolic Tangent activation, or Tanh for short, presents a similar nature with logistic Sigmoid function, due to its ability to map input values to a specific range, in it's case in between -1 and 1, offering advantages in certain scenarios, such as squashing continuous actions in RL. Tanh addresses the previously mentioned issue of exploding gradients visible in recurrent architectures, which is particularly beneficial as it helps stabilize and control the flow of gradients, facilitating more effective weight updates, though it can fall into the vanishing gradient problem, born from the opposite end (is it noticeable on the very left or right of the function, where the gradient approaches zero).

$$\mathrm{Tanh}(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \qquad \mathrm{Tanh}'(x) = 1 - \mathrm{Tanh}^2(x)$$

It is imperative to recognize that the final approximation of the objective function, either it is regression or classification, will look different based on the activation function used (curve fitting is smooth with Tanh and rectangular with ReLU, see Figure 2).

Furthermore, it is very important to delve into the Softmax activation function, distinguished by its unique purpose compared to other activation functions. Unlike other activation functions that focus solely on capturing nonlinearities or squashing input values within a certain range, the Softmax function serves a distinct role, especially in the context of multi-class classification problems. Shortly said, the primary objective of Softmax is to transform a vector of arbitrary real values (logits) into a probability distribution.

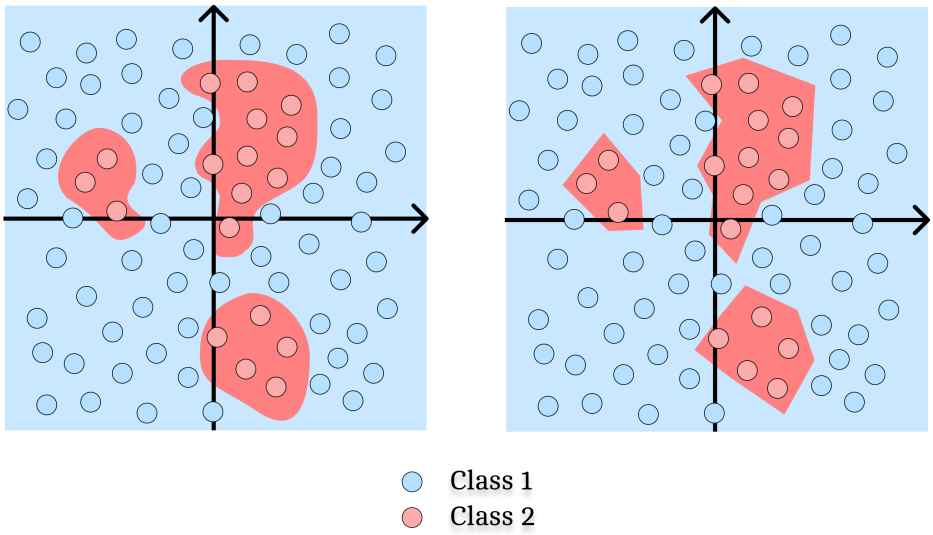


Figure 2: Tanh (left) and ReLU (right) nonlinearities

This is done by exponentiating the vector to get rid of negative values, then normalize it by dividing each element by its element's sum, leading to a discrete probability distribution vector, whose elements sum to 1. The given probability distribution can be softened (shifted towards an uniform distribution) by dividing the input vector before exponentiation with a temperature value $\tau > 1$, to encourage a higher entropy output, commonly present in RL for exploration purposes or LLMs to promote more diverse output.

The derivative calculation for the Softmax activation implies finding the partial derivatives of each output value w.r.t. each input value, generating a Jacobian matrix $J_{\text{Softmax}}(x)$. Computing the gradient of the loss function w.r.t. the input x is obtained by computing the matrix multiplication between the partial derivative of the loss function with respect to Softmax(x) with the Jacobian matrix $\left(\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \text{Softmax}(x)} \cdot J_{\text{Softmax}(x)}\right)$.

$$\text{Softmax}\left(x, \tau\right) = \frac{\exp\left(\frac{x}{\tau}\right)}{\sum_i \exp\left(\frac{x_i}{\tau}\right)} \qquad \frac{\partial \text{Softmax}_i\left(x, \tau\right)}{\partial x_j} = \text{Softmax}_i\left(x, \tau\right) \cdot \left(\delta_{ij} - \text{Softmax}_j\left(x, \tau\right)\right)$$

where δ_{ij} denotes the Kronecker delta, which equals 1 if $i = j$ and 0 otherwise. We recall that activation functions exhibit unstable behavior based on the architecture of the model and the preliminary values of the parameters, so initializing them properly 2.1.3 will diminish any nasty performance that affects convergence.

2.2.3 Normalization for accelerated training

The complications of training deep neural networks arise in the distribution of each layer, that changes as the previous layer parameters change. Authors of [IS15] whom firstly approached and named this issue as *internal covariate shift* which slows down the convergence of the network, reduced it by normalizing the data over the batch dimension, introduced as a layer that can be placed before or after nonlinear activations. While part of a deep neural network it enables higher learning rates with no need for additional regularization techniques.

Algorithm 4 Batch Normalization. Defined by H which is the size of input features, η - momentum used in μ_{running} and ${\sigma^2}_{\text{running}}$ computation, ϵ - value added to the denominator for numerical stability. The Batch Normalization layer is placed before or after the nonlinear activation function.

Require: $\eta \in (0, 1)$: Momentum (default: 0.9)
Require: ϵ: Value for numerical stability (default: 10^{-5})
Require: μ_{running}: Running mean
Require: $\sigma^2_{running}$: Running variance
Require: γ: Affine scale
Require: β: Affine shift
Require: $\dfrac{\partial L}{\partial \gamma}$: ∇_γ - Weights' gradients
Require: $\dfrac{\partial L}{\partial \beta}$: ∇_β - Biases' gradients

Initialize $\mu_{running} \in 0_H$
Initialize $\sigma^2_{running} \in 1_H$
Initialize $\gamma \in 1_H$
Initialize $\beta \in 0_H$
procedure FORWARD(x)
  if inference **then**
    $x_{\text{normalized}} \leftarrow \dfrac{x - \mu_{\text{running}}}{\sqrt{\sigma^2_{running} + \epsilon}}$
    $y \leftarrow x_{\text{normalized}} \odot \gamma + \beta$
    return y
  else
    $\mu_B \leftarrow \dfrac{1}{m}\sum_{i=1}^{m} x_i$ ▷ Mean across the batch
    $\sigma^2_{B,biased} \leftarrow \dfrac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{Bi})^2$ ▷ Biased variance across the batch
    $\sigma^2_{B,unbiased} \leftarrow \dfrac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_{Bi})^2$ ▷ Unbiased variance across the batch
    $x_{centered} \leftarrow x - \mu_B$
    $\sigma \leftarrow \sqrt{\sigma^2_{B,biased} + \epsilon}$
    $\hat{x} \leftarrow x_{centered}/\sigma$
    $\mu_{running_t} \leftarrow \eta \mu_{running_{t-1}} + \mu_B(1 - \eta)$
    $\sigma^2_{running_t} \leftarrow \eta \sigma^2_{running_{t-1}} + \sigma^2_{B,unbiased}(1 - \eta)$
    $y \leftarrow \hat{x} \odot \gamma + \beta$
    return y
  end if
end procedure

At train time in the forward pass, the variance is calculated via the biased estimator (the Maximum Likelihood Estimator (MLE) for the variance is biased), however the value used for updating the running variance $\sigma^2_{running}$ is calculated via the unbiased estimator. Affine transformation is optional, so γ and β can be fixed parameters. Note that BN layer cancels out the biases (if any) of a previous parametrized layer (e.g. the Dense layer), so they should be ommited.

23: **procedure** BACKWARD($\frac{\partial L}{\partial y}$)

24: $\quad \frac{\partial L}{\partial \hat{x}} \leftarrow \frac{\partial L}{\partial y} \cdot \gamma$

25: $\quad \frac{\partial L}{\partial \sigma_B^2} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot x_{centered_i} \cdot \frac{-1}{2}(\sigma + \epsilon)^{-3/2}$

26: $\quad \frac{\partial L}{\partial \mu_B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{-1}{\sigma} + \frac{\partial L}{\partial \sigma_B^2} \cdot \frac{1}{m} \sum_{i=1}^{m} -2 \cdot x_{centered_i}$

27: $\quad \frac{\partial L}{\partial \gamma} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \cdot \hat{x}_i$

28: $\quad \frac{\partial L}{\partial \beta} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$

29: $\quad \frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial \hat{x}} \cdot \frac{-1}{\sigma} + \frac{\partial L}{\partial \sigma_B^2} \cdot \frac{2 \cdot x_{centered}}{m} + \frac{\partial L}{\partial \mu_B} \cdot \frac{1}{m}$

30: $\quad$ **return** $\frac{\partial L}{\partial x}$

31: **end procedure**

Authors suggest to accelerate the learning rate decay, shuffle "within-shard" the training samples more thoroughly to prevent the same examples from always appearing in a mini-batch together (authors found that it improves by 1% the regularization capabilities) and for computer vision tasks, reduce the photometric distorsion of the images (because batch normalized networks are training faster, they observe each training example fewer times so they focus on more *real* images by distorting them less). The idea of normalizing a one dimensional array of input features H can be optionally extended towards higher dimensions instead of reshaping the input in a vectorized form for the sake of efficiency and consistency.

Layer Normalization (LN) [BKH16], Root Mean Squared (RMS) Normalization [ZS19] and Group Normalization (GN) [WH18] are more popular alternatives especially in deep sequence modelling (batch normalization isn't straightforward when it comes to the influence of previous timesteps in a hidden state for recurrent models and is basically sensitive to the batch size, but is still in used in computer vision), with the main difference they impose being the transposition of the normalization axis; instead of normalizing across the batch dimension, they are computed over the features dimension (note that parameters' shape must be adjusted accordingly). Over that, they do not depend on a running mean and variance, so the input is normalized independently in *inference* mode.

2.2.4 Stochastic Regularization via Node Dropout

Since large models are more efficient in approximating $\mathcal{J}(\theta)$, they fall into the generalization problem. By fitting too well over the training data, it loses the capabilities to return good results on novel data. To prevent this "overfitting" issue, multiple methods addresses it by reducing the capabilities of the parameters during training or inference. One introduces a penalty directly into the loss function (explained in Section 2.3.3) that forces the weights to attain certain values, but one simpler way was introduced in [SHK+14] whose effect runs during training by cutting off random neurons/parameters, inserted as a hidden layer after nonlinear activation functions, disabled on inference. In order to simulate the dropout effect, we scale and filter out the outputs of the previous layer through a binary mask M with the strength of a probability factor $p \in [0, 1)$.

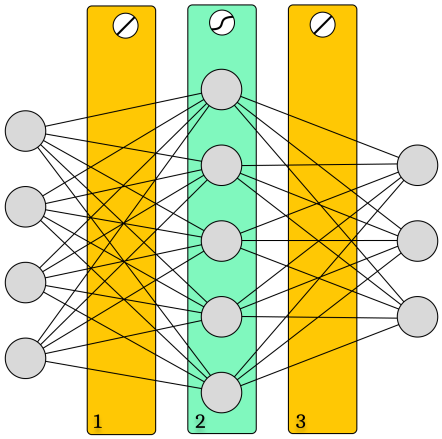


Figure 3: Fully Connected

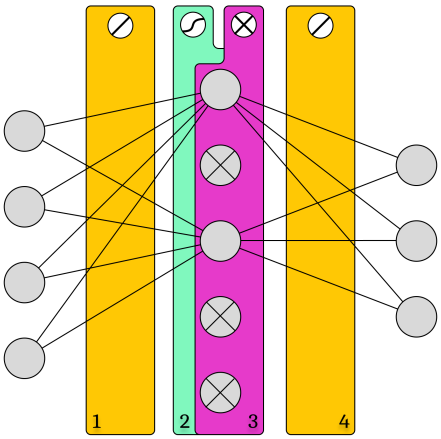


Figure 4: With Dropout

Algorithm 5 Dropout. The regularization strength is defined by a continuous variable $p \in [0, 1)$, that parametrizes the Bernoulli Distrbution that generates the mask M, active only on training (on evaluation, dropout is disabled). The Dropout layer is always placed after the nonlinear activation function. For channeled input $\in \mathcal{R}^{C \times \Pi_{i=1}^{n} D_i}$, Spatial Dropout variant zeroes out channels individually, because activations of adjacent pixels within feature maps with high correlation are not simply regularized by normal dropout.

Require: $p \in [0, 1)$: Dropout strength probability (typically < 0.5)
Require: M: Masking matrix with the same shape as x

procedure FORWARD(x)
  if inference **then**
    return x
  else
    $M \leftarrow GenerateMask(p)$
    $y \leftarrow x \odot M \cdot \dfrac{1}{1-p}$
    return y
  end if
end procedure
procedure BACKWARD($\dfrac{\partial L}{\partial y}$)
  $\dfrac{\partial L}{\partial x} \leftarrow \dfrac{\partial L}{\partial y} \odot M$
  return $\dfrac{\partial L}{\partial x}$
end procedure

2.2.5 Feature Extraction and Pooling

In computer vision tasks or sequence modelling, enabling systems to derive abstract information from digital images or sequential information plays a huge role in achieving generalization. One key architectural component introduced many years back in [LB+95] facilitating this process is the convolutional layer. Through the cross-correlation of the input with a set of kernels for each channel like in Figure 5, the features are effectively

passed forward. The kernels are trainable parameters that optimise in such way to remodel the input into a multi-chanelled representation containing simpler and more general variants of the input and to establish a logical connections between neighbouring features in the input. When dealing with very large kernels, the cross-correlation $\star$ can be replaced by Fast Fourier Transformations (FFT) to reduce the complexity from $\mathcal{O}(NK)$ to $\mathcal{O}(NlogN)$, where N is the input size. To enhance expressiveness, an activation is placed immediately afterward to ensure the caption of nonlinear characteristics of the features. The same principles can be applied for any number of dimensions in the received input, by setting up the kernels shape accordingly (see the note at the end of this subsection).

The computation is often streamlined by pairing the convolutional layers with pooling layers, that used to reduce the dimension of the convoluted batch while retaining important aspects. The pooling process basically works by grouping neighbouring features into equal packs, where the maximum or the mean value from each contributes to the final pooled output. To retain the dimensionality in the transformation, the input should be padded based on the kernels' sizes. This involves adding zero values or mirroring the neighboring features of the edges to ensure a more comprehensive representation and prevent the loss of relevant information. The gradient of a convolutional layer with respect to its input can be viewed as a deconvolution (also known as fractionally-strided convolution or transposed convolution), that functions independently as a layer, particularly present in generative models to upsample the features.

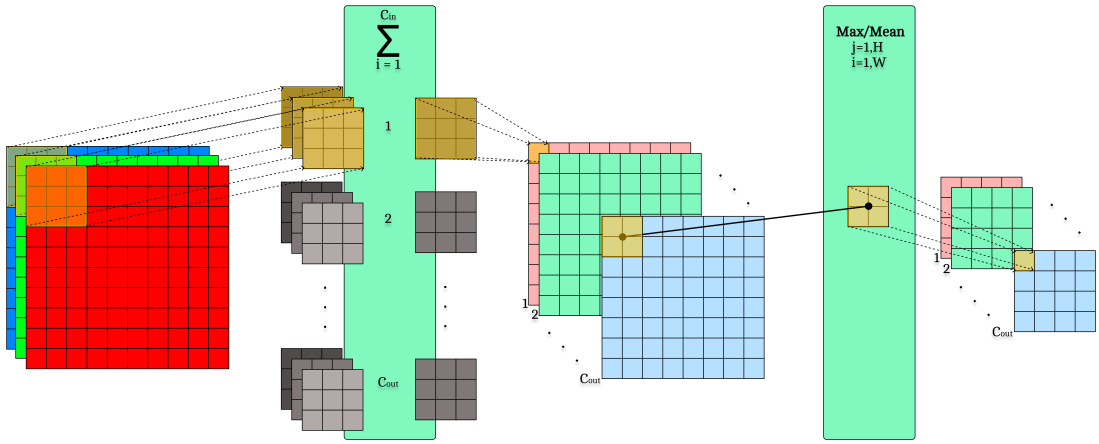


Figure 5: 2D Convolution with follow up Pooling

For processing inputs in higher resolutions and having broader view of the image with an overall faster run-time with fewer parameters, dilated convolution (also known as à trous convolution) or pooling proposes to introduce holes for more coverage instead of increasing the size of the kernel. Increasing the stride, which involves moving the filter by more steps both horizontally and vertically, can enhance the reduction of the feature map and computational complexity. The default convolution has a stride of 1 (the filter moves by one step) and a dilation of 1 (the filter has no gaps).

Figure 6: dilation 1

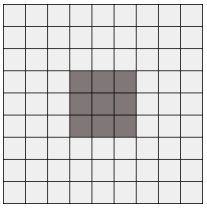


Figure 7: dilation 2

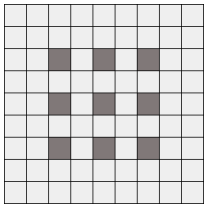
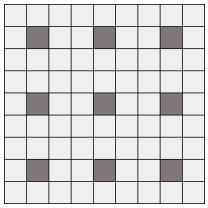


Figure 8: dilation 3



Algorithm 6 Convolution 2D. Defined by the input shape C_{in} and C_{out} which represents the number of channels of the input and the output respectively, and the shape of the kernels (K_H, K_W), where K_H and K_W are the kernel's height and width respectively. For parameter initialization, $fan_{in} = C_{in} \cdot K_H \cdot K_W$ and $fan_{out} = C_{out}$. The output will be of shape $(C_{out}, H_{out}, W_{out})$, where $H_{out} = H_{in} - K_H + 1$ and $W_{out} = W_{in} - K_W + 1$. The Convolution layer is typically followed by an activation function and a Pooling layer.

Require: $C_{in}, H_{in}, W_{in} > 0$: Input's shape
Require: $C_{out} > 0$: Number of output channels
Require: $K_H > 1$: Kernels' height (typically 3 or 5)
Require: $K_W > 1$: Kernels' width (typically 3 or 5)
Require: K: Kernels
Require: B: Biases
Require: $\frac{\partial L}{\partial K}$: ∇_K - Kernels gradients
Require: $\frac{\partial L}{\partial B}$: ∇_B - Biases gradients

Initialize $K \in \mathcal{R}^{C_{out} \times C_{in} \times K_H \times K_W}$ using an initializer (see Section 2.1.3)
Initialize $B \in \mathcal{R}^{C_{out}}$ using an initializer (see Section 2.1.3)
procedure FORWARD(x)
    Initialize y tensor of shape $(C_{out}, H_{out}, W_{out})$ with 0
    for each $x_b \in x$ **do** ▷ For each batch element
        for o = 0, 1, ... , C_{out} **do**
            for i = 0, 1, ... , C_{in} **do**
                $y_{b,o} \leftarrow y_{b,o} + Correlate^{2D}_{valid}(x_b, K_{o,i}) + \frac{B_o}{C_{in}}$ ▷ Summation over C_{in}
            end for
        end for
    end for
    return y
end procedure
procedure BACKWARD($\frac{\partial L}{\partial y}$)
    $s \leftarrow 1/(m \cdot C_{in} \cdot C_{out} \cdot K_H \cdot K_W \cdot H_{in} \cdot W_{in})$ ▷ Gradient scale coeff.
    for each $(x_b, L_b) \in \left(x, \frac{\partial L}{\partial y}\right)$ **do** ▷ For each batch elements
        for o = 0, 1, ... , C_{out} **do**
            for i = 0, 1, ... , C_{in} **do**
                $\frac{\partial L_b}{\partial K_{o,i}} \leftarrow \frac{\partial L_b}{\partial K_{o,i}} + s \cdot Correlate^{2D}_{valid}(x_{b,i}, L_b)$
                $\frac{\partial L_b}{\partial x_{b,i}} \leftarrow \frac{\partial L_b}{\partial x_{b,i}} + s \cdot Convolve^{2D}_{full}(L_{b,o}, K_{o,i})$
            end for
            $\frac{\partial L_b}{\partial B_o} \leftarrow \frac{\partial L_b}{\partial B_o} + s \cdot \sum L_{b,o}$
        end for
    end for
    return $\frac{\partial L}{\partial x}$
end procedure

Algorithm 7 MaxPool 2D. Defined by the kernel height K_W and width K_W. The output shape is $(C, \lfloor (H_{in} - K_H)/K_H + 1 \rfloor, \lfloor (W_{in} - K_W)/K_W + 1 \rfloor)$. The Pooling layer is typically placed after a Convolution layer, followed by a nonlinear activation function and optionally preceded by a padding layer.

Require: $K_H > 1$: Kernel's height (default: 2)
Require: $K_W > 1$: Kernel's width (default: 2)

procedure FORWARD(x)
  for each $x_b \in x$ **do** ▷ For each batch element
    for c = 0, 1 ... C **do** ▷ For each channel
      for j = 0, 1, ... $\lfloor (H_{in} - K_H)/K_H + 1 \rfloor$ **do**
        for i = 0, 1, ... $\lfloor (W_{in} - K_W)/K_W + 1 \rfloor$ **do**
          $y_{b,c,j,i} \leftarrow \max(x_{b,c})$
        end for
      end for
    end for
  end for
  return y
end procedure
procedure BACKWARD($\frac{\partial L}{\partial y}$)
  for each $(x_b, L_b) \in \left(x, \frac{\partial L}{\partial y}\right)$ **do** ▷ For each batch elements
    for c = 0, 1 ... C **do** ▷ For each channel
      for j = 0, 1, ... , $\lfloor (H_{in} - K_H)/K_H + 1 \rfloor$ **do**
        for i = 0, 1, ... , $\lfloor (W_{in} - K_W)/K_W + 1 \rfloor$ **do**
          $\frac{\partial L}{\partial x_{b,c}} \leftarrow \text{onehot}^{2D}(\text{x}_{b,c}) \cdot \frac{\partial L_b}{\partial y_{b,c,j,i}}$
        end for
      end for
    end for
  end for
  return $\frac{\partial L}{\partial x}$
end procedure

The presented Convolution and Pooling algorithms are simplified, presented with a stride of 1 and the dilation of 1 with no integrated padding (padding can be implemented as a layer itself that precedes them). They can be adapted to one-dimensional inputs, using one-dimensional filters of shape (K), where K is the kernel's length. The weight and bias tensors of a Conv1D layer will be initialized with the shape (C_{out}, C_{in}, K) and (C_{out}) respectively. While variations such as transposed, depthwise, separable depthwise, or pointwise convolutions are noteworthy, their purposes lie beyond the scope of this paper, which focuses on reinforcement learning. For visual representations, see guide [DV18].

2.2.6 Processing Time Sequences

Managing sequences of variables with varying lengths presents a formidable challenge when relying on standard layers within neural network architectures. The intricacies of handling dynamic temporal data require specialized mechanisms to capture dependencies and patterns over time. Thus, we cover the Simple Recurrent Network (SRN), first state space model introduced in [Elm90] designed for sequence modelling. It works by integrating recurrent parameters, so any recurrent activation takes into consideration it's last output, also known as the hidden state, denoted as h_{t-1}. It's important to note that a hidden state h_t is purely dependent by the input x_t and the previous hidden state h_{t-1} (a general linear construction can be defined as $h_t = Rh_{t-1} + Wx_t$, where W and R are trainable parameters). So, the forward method receives an input x consisting of (variable) sequences of features S inputs $x_1, x_2, \ldots, x_S$, and returns either the last hidden state, or all hidden states computed (the least might be needed if the following layer is another RNN cell). Note that the hidden state h_0 is considered 0 for the purpose of computing the first element in the sequence, and all input and hidden states must be cached in order to compute the gradients (also the linear function denoted as l_t in the presented algorithm).

The Recurrent Layer can yield either the final hidden state h_T or all computed hidden states, depending on specific requirements. It commonly employs Tanh as a nonlinearity or, occasionally, utilizes ReLU activation for scenarios involving short sequence lengths. The gradient computation implies the back propagation "through time" of the loss in reverse by passing the gradient of the hidden state to the previous time step. When all hidden states are returned by this layer, their corresponding gradient are obtained, which are added to the previously mentioned gradient, resulting from two distinct sources (for a better comprehension, check the algorithm at line 19).

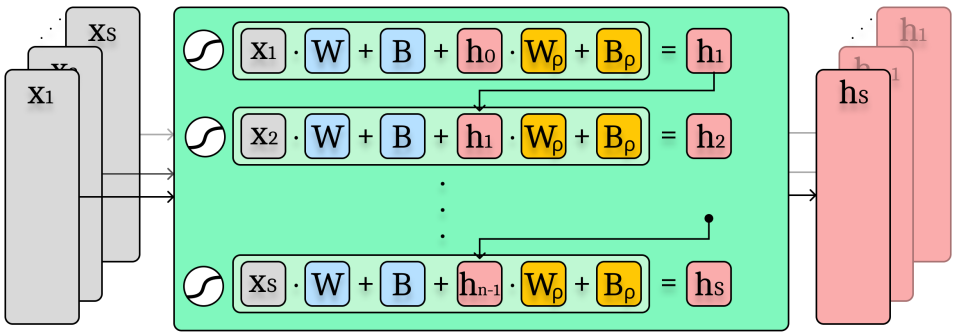


Figure 9: Forwarding a sequential input in RNN Cell

RNN cells struggle to capture long-range dependencies effectively, as they primarily rely on time dependence when computing a hidden state. Additionally, they suffer of vanishing or exploding gradients when it comes to large sequences, due to hidden state overwriting though saturated or non-saturated activation functions respectively, so gated architectures like LSTM [HS97] or GRU [CVMBB14] are designed to avoid that by adding at the hidden state, so the gradients are distributed back evenly. As part of Mamba architecture, as introduced in [GD23], selective state space models (S6), which can be interpreted as a parallelizable linear RNN, represent the cutting edge since they proximately achieve the sequence modeling power of Transformers though with a linear scalability. However, since DRL does not currently scale with very-long sequences, RNNs are still used due to their inherent simplicity.

Algorithm 8 RNN Cell. Defined by H_{in}, H_{out} and φ, where H_{in} represents the input size, H_{out} the output size and φ the nonlinearity.

Require: φ - Nonlinearity (default: Tanh)
Require: W: Weights
Require: R: Recurrent Weights
Require: B: Biases
Require: $\frac{\partial L}{\partial W}$: ∇_W - Weights' gradients
Require: $\frac{\partial L}{\partial R}$: ∇_R - Recurrent Weights' gradients
Require: $\frac{\partial L}{\partial B}$: ∇_B - Biases' gradients

Initialize $W \in \mathcal{R}^{H_{out} \times H_{in}}$ using an initializer (see Section 2.1.3)
Initialize $R \in \mathcal{R}^{H_{out} \times H_{out}}$ using an initializer (see Section 2.1.3)
Initialize $B \in \mathcal{R}^{H_{out}}$ using an initializer (see Section 2.1.3)
$h_0 \leftarrow 0$
procedure FORWARD(x_1, x_2, ... ,x_S)
    for t = 1, 2, ... , S **do**
        $l_t \leftarrow x_t W^T + h_{t-1} R^T + + B_{expanded}$
        $h_t \leftarrow \varphi(l_t)$
    end for
    return h_1, h_2, ... , h_S **or** h_S
end procedure
procedure BACKWARD($\frac{\partial L}{\partial h_1}, \frac{\partial L}{\partial h_2}, \ ... \ , \frac{\partial L}{\partial h_S}$ **or** $\frac{\partial L}{\partial h_S}$)
    for t = S, $S-1$, ... , 1 **do** ▷ Unfold in reverse
        $\frac{\partial L}{\partial l_t} \leftarrow \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial l_t}$ ▷ Note that $\frac{\partial h_t}{\partial l_t}$ equals $\frac{\partial \varphi(l_t)}{\partial l_t}$
        $\frac{\partial L}{\partial W} \leftarrow \frac{\partial L}{\partial W} + (\frac{\partial L}{\partial l_t})^T \frac{1}{m} \sum_{i=1}^{m} x_{t,i}$
        $\frac{\partial L}{\partial R} \leftarrow \frac{\partial L}{\partial R} + (\frac{\partial L}{\partial l_t})^T \frac{1}{m} \sum_{i=1}^{m} h_{t-1,i}$
        $\frac{\partial L}{\partial B} \leftarrow \frac{\partial L}{\partial B} + \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial l_{t,i}}$
        $\frac{\partial L}{\partial x_t} \leftarrow \frac{\partial L}{\partial l_t} W$
        $\frac{\partial L}{\partial h_{t-1}} \leftarrow \frac{\partial L}{\partial h_{t-1}} + \frac{\partial L}{\partial l_t} W_\rho$ ▷ If not received as arg., $\frac{\partial L}{\partial h_{t-1}}$ is 0 by default
    end for
    return $\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \ ... \ , \frac{\partial L}{\partial x_S}$
end procedure

When stacking sequence to sequence layers (e.g., RNN/LSTM/GRU cells), the last one typically has to return only the last hidden state, as it is considered the predicted next token; i.e., the gradients for all other sequence elements but the last are zero. This reduces computational cost if the model has any other layers afterwards.

2.2.7 Self-Attention using Scaled Dot-Product approach

Attention mechanisms were designed as a sequence model replacement for encoder-decoder models in neural machine translation, because they highlight from the input the most important parts, regardless of their position in the sequence and their neighbourhoods (unlike convolutional layers). It was firstly introduced by Bahdanau et al. in [BCB14] as a bidirectional RNN (BiRNN) that emulates as both an encoder and decoder. A general adaptation is Scaled Dot-Product Attention (SDPA), part of the Transformer architecture [VSP+17], used also in sequence handling, especially focusing on dependencies between distant elements in a sequence or between the elements of two different sequences. The three components that make the analogy to the BiRNN are the Query, Key and Value matrices, obtained by multiplying the input x with W_Q, W_K and W_V (in Self-Attention). In Cross-Attention, the Query is obtained instead from the second input y and W_Q.

Scaled Dot-Product Attention works by matching the Query with Key in order to obtain the score values (which determine the relevance given in the attendance of each token to another). The score is scaled by $1/\sqrt{d_k}$ for stability, were d_k is the embedding dimension that determines the actual size of the Key matrix (for the sake of simplicity, we will assume all matrices will have the same dimension d). Optionally, a masking operation for "future" tokens can be used right after by setting the values in the upper triangular elements as $-\infty$. The result is passed through a Softmax activation to make the scores positive, and the masked zones will get a value of 0. The output comes from the multiplication of the obtained positive scores matrix with the Value matrix.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{6}$$

The authors propose creating more smaller individual Query, Key and Value pairs to leverage the advantages of "mixture of experts". The context matrices of each SDPA head are concatenated, then multiplied with a learnable matrix W^O of shape (d, d) $(d = H)$.

Algorithm 9 Causal Attention on a single Scaled Dot-Product head. Defined by H and d, where H represents the features size of the input and d is the embedding dimension.

Require: d: Embedding dimension

Require: W_Q, W_K, W_V: Query, Key and Value weight matrices

Require: $\frac{\partial L}{\partial W_Q}$, $\frac{\partial L}{\partial W_K}$, $\frac{\partial L}{\partial W_V}$: ∇_{W_Q}, ∇_{W_K}, ∇_{W_V} - Query, Key and Value weights' grad.

Initialize W_Q, W_K and $W_V \in \mathcal{R}^{H \times \frac{d}{num.\ heads}}$ using an initializer (see Section 2.1.3) where $fan_{in} = H$ and $fan_{out} = \frac{d}{num.\ heads}$

procedure FORWARD(x)

  for $x_b \in x$ **do** ▷ For each batch element

    $Q_b \leftarrow x_b W_Q$

    $K_b \leftarrow x_b W_K$

    $V_b \leftarrow x_b W_V$

    $y_b \leftarrow \text{Softmax}\left(\frac{Q_b K_b^T}{\sqrt{d}} \odot M\right)V$ ▷ $M_{ij}^{L \times L} = \begin{cases} -\infty & \text{if } i < j \\ 1 & \text{otherwise} \end{cases}$

  end for

  $y \leftarrow y_1, y_2, \ldots, y_m$

  return y

end procedure

12: **procedure** BACKWARD($\frac{\partial L}{\partial y}$)

13: **for** $x_b \in x$ **do** ▷ for each batch element

14: $\frac{\partial L}{\partial V_b} \leftarrow \text{Softmax}\left(\frac{Q_b K_b^T}{\sqrt{d}}\right)\frac{\partial L}{\partial y_b}$

15: $\frac{\partial L}{\partial Q_b} \leftarrow V\left(\frac{\partial L}{\partial y_b}\right)^T J_{\text{Softmax}}\left(\frac{Q_b K_b^T}{\sqrt{d}}\right)\frac{1}{\sqrt{d}}K_b$

16: $\frac{\partial L}{\partial K_b} \leftarrow V\left(\frac{\partial L}{\partial y_b}\right)^T J_{\text{Softmax}}\left(\frac{Q_b K_b^T}{\sqrt{d}}\right)\frac{1}{\sqrt{d}}Q_b$

17: $\frac{\partial L}{\partial W_Q} \leftarrow \frac{\partial L}{\partial W_Q} + \frac{1}{m}x_b^T\frac{\partial L}{\partial Q_b}$

18: $\frac{\partial L}{\partial W_K} \leftarrow \frac{\partial L}{\partial W_K} + \frac{1}{m}x_b^T\frac{\partial L}{\partial K_b}$

19: $\frac{\partial L}{\partial W_V} \leftarrow \frac{\partial L}{\partial W_V} + \frac{1}{m}x_b^T\frac{\partial L}{\partial V_b}$

20: $\frac{\partial L}{\partial x_b} \leftarrow \frac{\partial L}{\partial Q_b}W_Q^T + \frac{\partial L}{\partial K_b}W_K^T + \frac{\partial L}{\partial V_b}W_V^T$

21: **end for**

22: $\frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \dots, \frac{\partial L}{\partial x_m}$

23: **return** $\frac{\partial L}{\partial x}$

24: **end procedure**

2.2.8 Residual Connections for deeper neural networks

Deeper neural networks tend to forget features of the dataset samples as well of showing signs of gradient vanishing, thus making them very difficult to train. Along substantial empirical results, [HZRS16] introduces residual learning through residual blocks to mitigate this degradation problem. They work by creating shortcut connections between every few stack of layers $\mathcal{F}(x)$, in which the input x is added to the output of the stack ($y = \mathcal{F}(x) + x$, see Figure 10). The gradient of the residual block will flow equally on two different paths, so the connection gradient will be directly added to the gradient of the block's input. During the final addition of the block, there might be a mismatch between the shapes of the skip connection output and the residual mapping. To address this, the skip connection can employ a linear projection to ensure the shapes align properly.

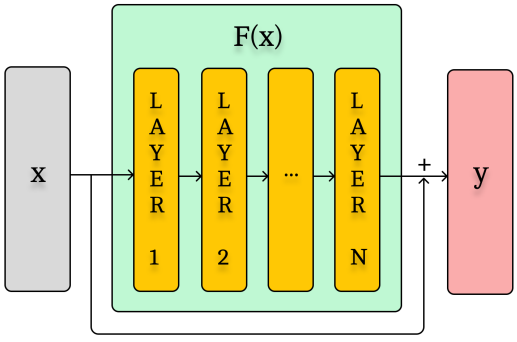


Figure 10: A building block in ResNets

2.3 Optimization

In the subsequent subsections, we discuss the optimization process, elucidating how the parameters of the models are updated based on the objective function we are trying to minimize or maximize, followed by improvement mechanisms in terms of convergence speed and generalization.

2.3.1 Understanding Model Error

Loss functions, also known as cost or objective functions, play a pivotal role in the training of machine learning models, with the purpose of quantifying the disparity between the predicted output of a model and the actual ground truth. This error serves as a signal guiding the optimization process during training. In the context of supervised learning, where models are trained on labeled datasets, the loss function evaluates the dissimilarity between the predicted output and the true target values. The objective is to minimize this loss, effectively enhancing the model's ability to make accurate predictions. Reinforcement Learning problems are solutionated by maximizing cumulative reward, involving the minimization of negative log-likelihood (NLL) loss associated to the actions' probabilities, covered in Chapter 3. For now, we will strictly bound to supervised learning. As preliminary described at the beginning of the paper, approximating the objective function $\mathcal{J}(\theta)$ involves the minimization of the mean squared error, which, for two datapoints, is:

$$\text{MSE}\left(\hat{y}, y\right) = \left(\hat{y} - y\right)^2 \qquad \frac{\partial \text{MSE}\left(\hat{y}, y\right)}{\partial \hat{y}} = 2 \cdot \left(\hat{y} - y\right)$$

Classification tasks often demand the model to make predictions about the likelihood of different classes, and this is where Cross Entropy loss comes into play. Cross Entropy is well-suited for scenarios where the goal is to measure the dissimilarity between predicted probability distributions and the true distribution of class labels, encouraging the model to assign high probabilities to the correct class. It penalizes the model more rigorously for confidently incorrect predictions in comparison with MSE, making it particularly effective for multi-class classification problems. Note that we are using a very small value $\epsilon > 0$ to avoid division by 0 in the loss calculation in cases where the output features contains values of 0.

$$\text{Cross Entropy}\left(\hat{y}, y\right) = -y \cdot \log\left(\hat{y} + \epsilon\right) \qquad \frac{\partial \text{Cross Entropy}\left(\hat{y}, y\right)}{\partial \hat{y}} = \frac{-y}{\hat{y} + \epsilon}$$

In binary $outputs - labels$ scenarios, where we want to measure the dissimilarity between the predicted probability distribution and the true binary labels of a dataset, Binary Cross Entropy (BCE)/Log Loss is a go-to choice. BCE is an adaptation of Cross Entropy that is particularly used in binary classification problems, that evaluates the divergence between predicted probabilities and the true binary labels, by forcing the model to return a probability close to 1 for true samples and close to 0 for false ones. Cross Entropy loss functions yield higher gradient values as depicted in Table 2.3.1, improving the convergence speed of the model. Again, we are inserting a small value $\epsilon > 0$ to avoid division by 0 or log 0 that yield NaN values.

$$\text{BCE}\left(\hat{y}, y\right) = -\left[y \cdot \log\left(\hat{y} + \epsilon\right) + \left(1 - y\right) \cdot \log\left(1 - \hat{y} + \epsilon\right)\right] \quad \frac{\partial \text{BCE}\left(\hat{y}, y\right)}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y} \cdot \left(1 - \hat{y}\right) + \epsilon}$$

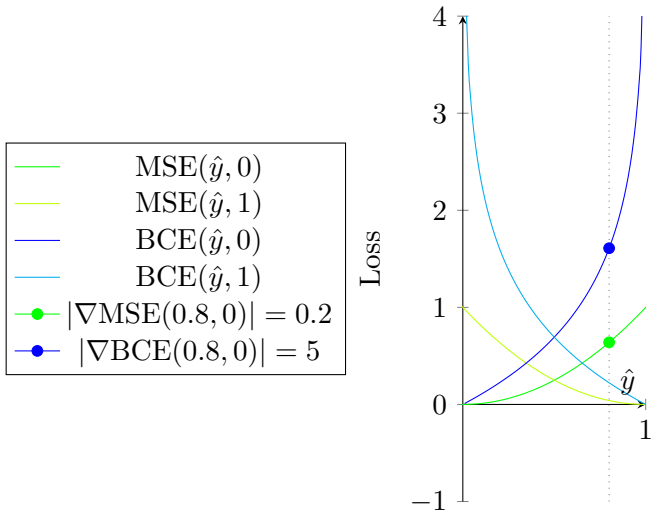


Figure 11: MSE and BCE gradient comparison

2.3.2 Gradient-based optimization

Finding the global minimum of any arbitrary function is a challenging problem, and there is no general mathematical method that guarantees a solution for all types of functions. The difficulty arises from the diverse nature of functions, which can be discountinuous, multivariate, and can have multiple minima when presenting non-convex shapes. The No-Free-Lunch (NFL) [WM97] theorem in optimization theory also suggests that no algorithm can outperform all others for all possible functions. Methods like simulated annealing or genetic algorithms are probabilistic solutions that are able to find global minimums since they are attaining the entire search space, though, in practice, heuristic approaches are commonly employed, accepting the trade-off between settling for a local minimum and computational effort. The associated cost is relatively inconsequential, as models trained on local minima often demonstrate empirical success and perform well in practical applications. Attaining a global minimum does not necessarily guarantee improved generalization or performance, making this compromise a pragmatic choice.

As the preeminent algorithm in state-of-the-art optimization problems, gradient descent is prominently employed due to its effectiveness in minimizing the aforementioned loss functions. The idea is to iteratively take small steps towards the opposite direction of the gradient of the differentiable objective function starting from an initial stochastic point, until reaching a local minimum, where any other direction is no longer heading towards a lower position on the function hyperplane. In contrast, we can employ a similar motion in the direction of the gradient to seek a local maximum; this process is referred to as gradient ascent. Supposing a multivariate function $\mathcal{J}(x)$ differentiable in a point θ_t, we can observe that by taking a minimal step in the direction of the negative gradient $-\nabla_{\theta_t}\mathcal{J}(\theta_t)$ will automatically end up in a lower point θ_{t+1} (see Figure 12).

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{J}(\theta_t), \text{ where } \alpha > 0 \tag{7}$$

Choosing the right step size α is not easy since either too small or too high values will end in requiring more steps until reaching a local minimum point. High learning rates have better chance of finding lower local minima, but the convergence will become unstable if the learning rate isn't decayed during the training process, and the parameter will oscillate between the opposite gradients (see Figure 12).

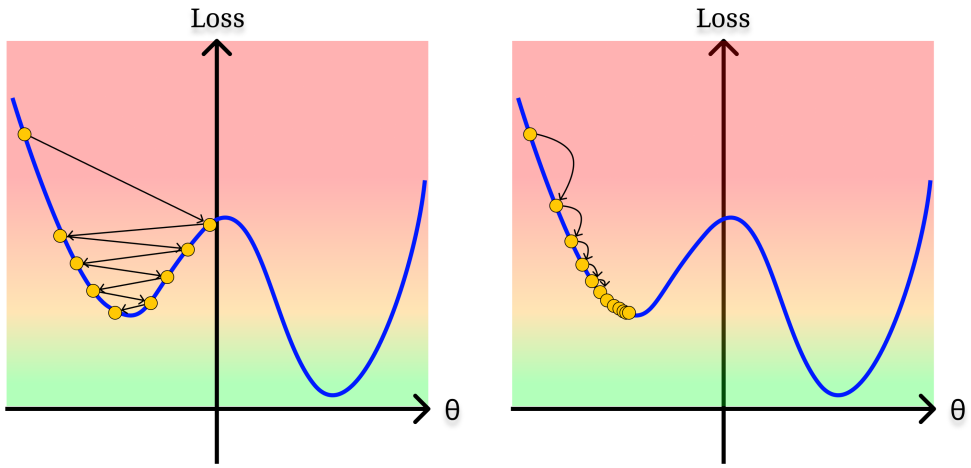


Figure 12: Gradient Descent with low and high learning rates

In supervised and reinforcement learning problems, the objective multivariate function $\mathcal{J}(\theta)$ that we are trying to minimize or maximize is unknown directly, but it can be approximated with any function approximator by the points given in the dataset (where each data sample represents a single point in the function). The process of error minimization between our model approximator and the objective function consists of three parts: compute the gradient of the loss function with respect to the model's parameters (back propagation), update the parameters using the their gradients (optimization) and repeat until convergence. The optimization of parameters can occur through two main approaches: computing the mean gradients over all samples in the dataset, termed batch training, or computing the gradient of a single randomly sampled point, known as stochastic training. Yet, the most employed method is a hybridization of the two because it resemble the sweet spot between oscillations over the hyperplane, fast convergence, better local minima and account of computational limits and efforts (see Figure 13), referred to as mini-batch training. This involves optimizing parameters using the mean of the gradients computed from a randomly selected small batch of data from the entire dataset. Full-batch training usually ensures stability and works well with high learning rates, while stochastic approaches present unstable convergence so they require low learning rates.

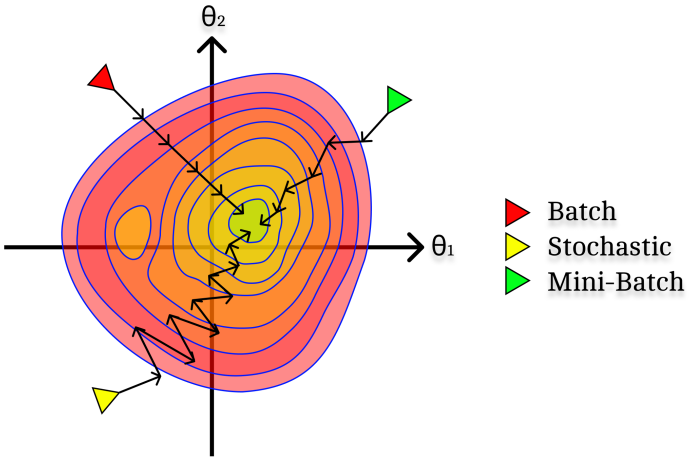


Figure 13: Gradient Descent methods

2.3.3 Mitigating the bias-variance tradeoff

Regularization is a technique in machine learning that is used to balance the bias-variance tradeoff. Overparametrized models in consequence of having low bias, they tend to exhibit high variance, more specifically, they manage to learn the training data extremely well while also capturing noise and specific patterns, leading to poor generalization. Classical regime models (underparametrized) require additional precautions, such as early stopping, to overcome the *double descent* [NKB+21] phenomenon.

Regularization adds constraints/penalties to the model during training to encourage it to be simpler or have smaller weights, leading to a more or less strong flattening of the model function (see Figure 14), reducing the risk of overfitting. Common types of regularizations are controlled by a hyperparameter $\lambda > 0$ (with common values $\in [1e-4, 1e-2]$) that is directly proportional with the regularization strength we want to apply in the training process. Note that regularization is applied only to the weights of the model, because the biases do not introduce the same level of model approximation complexity as weights do, and just one should be used at a time depending on the situation.

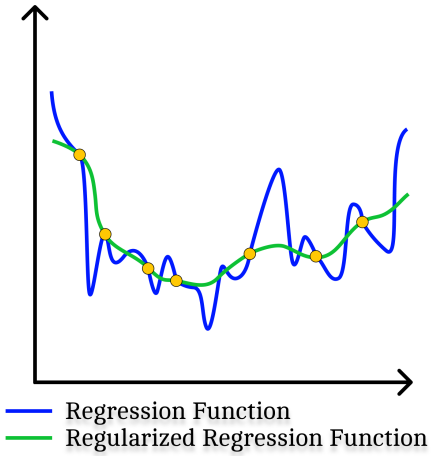


Figure 14: Generalization

L1 & L2 regularization techniques offer different benefits for the model. While L2 encourages small weights in order to stop the model to rely too heavily on any single weight, L1 does the opposite, encouraging sparsity in the model by forcing some weights to zero, thus making it simpler. The regularization is added into the loss function when employing a library that facilitates any autograd system. Implementing these through back propagation can be challenging; however, Weight Decay (WD) is a differentiable-free method inserted directly in the stochastic gradient descent and its variants algorithms' parameter update step, generally equivalent to $L2$ Regularization (excepting for adaptive optimizers). The logic is simply defined by subtracting a minor proportion λ from the old parameter θ_{t-1} on updating step, thus forcing θ to head towards small values.

$$\theta_t \leftarrow \theta_{t-1} - \alpha \nabla_\theta \mathcal{J}_t(\theta_{t-1}) - \lambda \theta_{t-1} \tag{8}$$

where $\nabla_\theta \mathcal{J}(\theta)$ is the gradient of the objective function w.r.t. the parameter θ and α is the step size (learning rate).

2.3.4 Momentum, Nesterov Acceleration and Adaptive Gradient

Most of the time, due to the immersive size of the dataset, we cannot minimize the objective function $\mathcal{J}(\theta)$ using gradient descent over the entire batch at a time due to hardware limitations. Thus, it's is mandatory to call for stochastic gradient descent (SGD), in which we split our training dataset into randomly sampled mini-batches. Due the fact that we can fall into a local minimum with SGD, the "Heavy Ball Method" proposes to accelerate the descent in order to fasten the descent or to even "jump" over local maxima in search for lower local minima. In order to generate momentum (also called velocity), we add to the current descent step a portion from the acceleration of the previous update step. There are multiple ways of writing the update formula, though, the

presented algorithms formulae were taken from [SMDH13], despite that one might find counter-intuitive to add the gradient to the parameter instead of subtracting it.

Nesterov Accelerated Gradient (NAG) [Nes83] is an extension of the normal momentum-based gradient descent that stops the momentum by computing the gradient of the parameter in the ahead position $\nabla_\theta \mathcal{J}_t(\theta_{t-1} - \eta v_{t-1})$ rather then the current position of θ. This prevents large steps that might lead to overshooting and oscillations if the step becomes too large by looking in the future of the performance of the updated parameter θ, thus improving the speed of the convergence (see Figure 15).

Algorithm 10 Stochastic Gradient Descent

Require: α: Learning rate (default: 0.01)
Require: $\eta \in [0, 1)$: Momentum (typical value: 0.9)
Require: $\lambda \geq 0$: Weight decay
Require: v: Moment buffer
Initialize v_0 with 0 or with g_1
for t = 1, 2, ... , K steps **do**
    $g_t \leftarrow \nabla_\theta \mathcal{J}_t(\theta_{t-1}) + \lambda\theta_{t-1}$ ▷ Estimate gradient **g** of the objective function $\mathcal{J}(\theta)$
    $v_t \leftarrow \eta v_{t-1} - \alpha g_t$ ▷ Update momentum buffer
    if nesterov **then**
        $\theta_t \leftarrow \theta_{t-1} - (g_t + \eta v_t)$
    else
        $\theta_t \leftarrow \theta_{t-1} + v_t$
    end if
end for

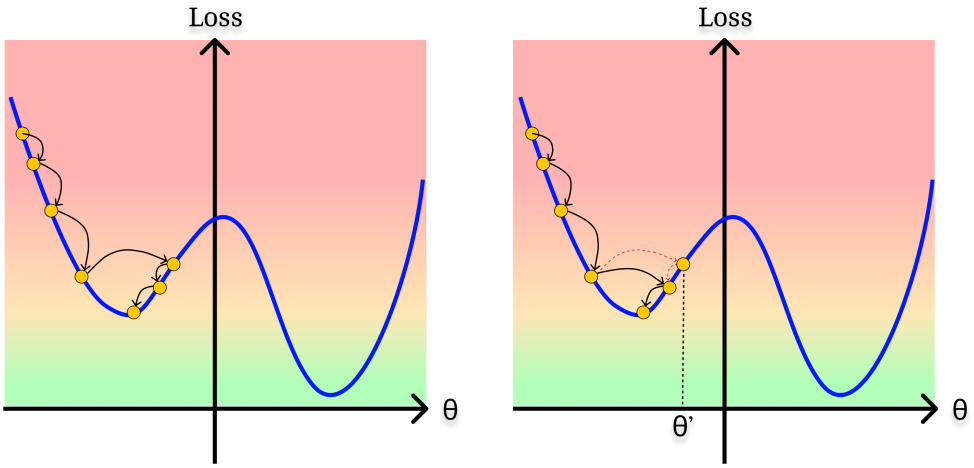


Figure 15: SGD with Momentum (left) and Nesterov Accelerated (right)

All gradient descent techniques are typically using a learning rate whose value is equivalent for each individual parameter, yet certain parameters may require more frequent adjustments to expedite convergence, whereas others may necessitate smaller modifications to avoid surpassing the optimal value. Duchi et al. [DHS11] propose an algorithm with flexible learning rate called AdaGrad, mainly determined by the objective of hastening up the convergence, by keeping up the track of the previous gradients values aiming assign lower step sizes to parameters with rapidly varying gradients and vice versa. This idea led to the development of Adam [KB14] optimizer (one that is mostly used cur-

rently), which incorporates the moving average of past gradients (similar to RMSProp [GNK12]) and also maintains an exponentially decaying average of past squared gradients (similar to the concept in AdaGrad).

The mechanism of adaptive optimizers influences the weight decay formulation, because when the WD is added directly to the gradient in these optimizers, it alters the adaptive gradients. Decoupled Weight Decay [LH17] separates the weight decay term from the moment buffers. The convergence of methods with a constant sized window of past gradients to highly suboptimal solutions was demonstrated in [RKK19] and it was simply covered by holding the maximum of all v_t, used to normalize the running average of the gradient of the current update step.

Algorithm 11 AMSGradW

Require: α: Learning rate (default: 0.001)
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates (typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$)
Require: $\epsilon > 0$: Value for numerical stability (default: 10^{-8})
Require: $\lambda \geq 0$: Weight decay
Require: $m, v, \hat{v}^{max}$: First and second moment
Initialize m_0, v_0 and $\widehat{v_0}^{max}$ with 0.
for t = 1, 2, ... , K steps **do**
  $g_t \leftarrow \nabla_\theta \mathcal{J}_t(\theta_{t-1})$ ▷ Estimate gradient **g** of the objective function $\mathcal{J}(\theta)$
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
  $\widehat{m_t} \leftarrow m_t / (1 - \beta_1^t)$ ▷ 1^{st} moment estimate bias towards 0 correction
  $\widehat{v_t} \leftarrow v_t / (1 - \beta_2^t)$ ▷ 2^{nd} moment estimate bias towards 0 correction
  $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\widehat{v_t}^{max}} + \epsilon) - \lambda \theta_{t-1}$ ▷ Update the parameter θ
end for

2.3.5 Gradient Clipping

Gradient Clipping is a method that modifies the already computed gradients after back propagation which overcomes the exploding gradient problem, thereby decreasing the likelihood of numerical instability during training. Basically, it means that the gradients that surpass a certain numerical value are scaled down to a more manageable value. There are two ways of clipping the gradients, by value and by norm. If we consider clipping by value, we need to decide a threshold value c, and before updating the parameters, all parameters gradients will be reasigned as $g_i \leftarrow Clip(g_i, -c, c)$. Clipping the norm of the gradients simply scales down all the parameters' gradients by choosing a max norm threshold c, then after computing the norm of the gradient vector, each gradient will be reassigned as $g \leftarrow g \cdot c / ||g||$. Below is shown how does the Infinity, Manhattan and Euclidean norm respectively are computed for a gradients vector g.

$$||g||_\infty = \max_{i=1,2,...n} |g_i| \qquad ||g||_1 = \sum_{i=1}^{n} |g_i| \qquad ||g||_2 = \sqrt{\sum_{i=1}^{n} g_i^2 + \epsilon}, \quad \epsilon > 0$$

Chapter 3

Reinforcement Learning

The goal of any Reinforcement Learning (RL) algorithm is to maximize the expected return given by the environment in which and agent performs. At its core, the fundamental objective is to enhance the agent's decision-making abilities by optimizing the cumulative reward obtained through interactions with its surroundings. The overarching aim can be expressed through the maximization of a performance metric, often denoted as the objective function $\mathcal{J}(\theta)$, where θ represents the parameters guiding the agent's behavior:

$$\mathcal{J}(\theta) = \underset{\theta}{\text{maximize}} \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[\sum_{t=1}^{T} r_t \right] \tag{1}$$

In this expression, $\mathbb{E}$ denotes the expectation operator, and the summation runs over discrete time steps t, with r_t signifying the reward accrued at timestep t for taking the action a_t from the state s_t in a given trajectory τ of length T sampled using the policy π parametrized by θ. The ultimate goal is to find the optimal policy π^* that has a set of parameters θ that maximizes the expected sum of rewards over a trajectory.

The pursuit of this optimization involves various approaches, each contributing to the rich landscape of RL algorithms. Policy optimization strategies firstly includes Derivative-Free Optimization (DFO), enumerating evolutionary and genetic algorithms (or others that combine these strategies such as NEAT [SM02] and it's hypercube-based readaptation), which are simple to implement and work very well for small policies, though they scale poorly with their dimensions. These techniques seek to optimize the policy in a tournament based manner between different evolved architectures or variants with stochastically perturbed parameters. Another prominent avenue is the exploration of Actor-Critic architectures, where an agent is divided into an actor responsible for selecting actions and a critic tasked with evaluating the chosen actions. This dual perspective allows for a more nuanced learning process. Furthermore, RL encompasses classical methods like Value Iteration, Policy Iteration and Q-learning, a model-free technique, which stands out as a pivotal algorithm, facilitating the estimation of action values and aiding in decision-making. The multifaceted nature of these approaches contributes to the adaptability and effectiveness of RL algorithms in various real-world scenarios.

The Section 3.3 exclusively delves into an in-depth exploration of state-of-the-art stochastic Policy Gradient Methods (PGMs), ones that are estimating the improvement direction of the policy by using certain measurements. Stochastic PGMs represent a powerful subset of algorithms that operate by directly optimizing the parameters of a policy, making them particularly well-suited for problems where the action space is continuous or high-dimensional.

3.1 Action Spaces

There are PGMs that rely on a deterministic policy (which produces the action directly by the model), and others that are using stochastic policies, in which the actions are not obtained directly by feed-forwarding the observation input, but instead the model's output parametrizes a distribution, from which we sample actions based on the entropy given by the policy. The following subsections describe how continuous and discrete actions are generated stochastically via sampling over the distributions parametrized by our models.

3.1.1 Discrete Actions

In the context of a RL, the agent interacts within his environment by using a given finite set of distinct actions. For example, in a 2D game, one branch that can define the character movement contains the following discrete actions set: move *upwards*, *downwards*, *right* or $left$. We can augment his behaviour by creating a new branch of discrete actions that let's him manipulate environmental objects, like *pick*, *throw*, *drop* etc. Each action is assigned a value $\in [0, 1]$ by the policy π that defines it's probability of being selected. Thus, for one discrete actions branch, our policy network yields for a given state s_t a vector ϕ_t with probabilities values that sum up to 1, with the length equal to the number of discrete actions in that branch. This vector parameterizes a Multinomial Distribution (also known as Categorical Distribution) that outputs a one-hot vector a_t that outlines which action was sampled (see Figure 1). The output vector returned by the policy's model that contains unnormalized values is transformed into a probability vector by passing it through a $Softmax$ activation, inserted as a layer at the end of the network.

Recalling the multi-action sampling, even though branching is a cleaner method of using more discrete actions at a time, this can also be done by defining all discrete actions and their combinations in a single discrete branch, or by multi-sampling without replacement from the same Categorical Distribution.

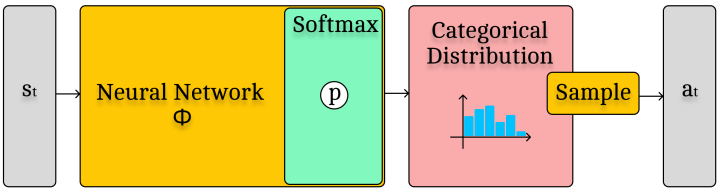


Figure 1: Discrete Actions

3.1.2 Continuous Actions

In contrast to discrete actions, we might need more precise control over the agent's actions. As an example, for a humanoid agent, each joint that connects it's body parts cannot be accurately moved by discrete commands, and instead by a gradient of movement strength and direction vectors in a real space that controls it's acceleration. This values are stochastically drawn using Box-Muller transform method from multiple one-dimensional Normal Distributions parametrized by our policy π that yields at a given state s_t one $\mu_t \in \mathbb{R}$ and $\sigma_t > 0$ value for each continuous action, therefore passed through a squashing function like Tanh for finite support. So, we will have two different neural networks (or one sharing the same backbone but with two different heads), one's ending with a linear activation to obtain μ, and the other with a positive activation to generate the stochasticity given by σ. NaN values occurrence is usually magnified by the exponential function, so it is preferred to use the Softplus activation (with configurable

parameters 4.3) instead. Alternatively, the secondary policy network can generate the log standard deviation, and then the value can be safely exponentiated separately before multiplied with an exploratory coefficient. Some algorithms like Vanilla PG or PPO allow using a fixed standard deviation set as a hyperparameter that can be linearly decayed over the training session, some are defined by a deterministic policy on which noise is applied (DDPG or TD3), while others are basically defined by their maximum-entropy characteristics like SAC, so lacking a trainable standard deviation limits it's exploratory potential. The probability of an action a_t is given by the product of all continuous actions' probabilities, obtained from the probability density function (PDF):

$$Probability(a, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right) \tag{2}$$

There are also algorithms especially designed for continuous control, however we can still use them for discrete actions by partitioning the continuous space of (-1, 1). After training, the stochasticity can be further removed (by using the argmax over the probability distribution for discrete actions and taking only the mean on continuous actions respectively) to serve as a precise, though deterministic behaviour on inference.

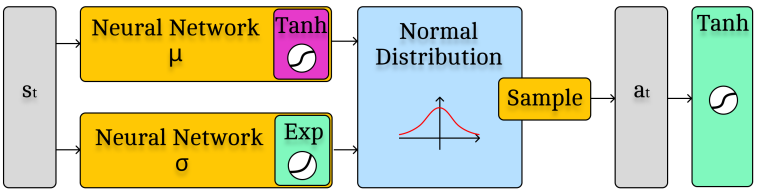


Figure 2: Continuous Actions

Note: Neural Network μ_θ lacks the last Tanh activation in PPO, A2C, A3C etc.

3.2 Entropy Bonus

The most important aspect in RL is the Exploration/Exploitation ratio. The agent cannot converge to a pleasant solution if he doesn't explore enough the environment and he will be stuck into a local maximum. On the other hand, if his actions are too diverse, his experiences will not be exploited enough, thus the agent will never learn. We will define our entropy regularized objective function $\mathcal{J}(\theta)$ that we are trying to maximize over the parameters θ as

$$\mathcal{J}(\theta) = \underset{\theta}{\text{maximize}} \underset{\tau\sim\pi_\theta}{\mathbb{E}} \left[\sum_{t=1}^{T} \gamma^t \Big(r_t + \beta\mathcal{H}\left(\pi_\theta(\cdot|s_t)\right)\Big)\right] \tag{3}$$

where $\mathcal{H}(\pi_\theta) = \mathbb{E}[\mathcal{H}^{Shannon}(\pi_\theta)]$ is the entropy bonus and $\beta > 0$ denotes the trade-off, which forces proportionally the policy to yield less biased probabilities (in case of discrete actions) and high standard deviation values (in case of continuous actions). Basically, the agent is extra rewarded proportionally with the entropy of π, so the goal of the agent is not only to learn an optimal policy π^* that achieves the highest discounted rewards, but also to put itself in states where its future entropy is the highest (we will call it a maximum entropy policy).

$$\pi^* = \text{argmax}_\pi \underset{\tau\sim\pi_\theta}{\mathbb{E}} \left[\sum_{t=1}^{T} \gamma^t \Big(r_t + \beta\mathcal{H}\left(\pi_\theta(\cdot|s_t)\right)\Big)\right] \tag{4}$$

Since our new entropy regularized objective function implies the entropy function $\mathcal{H}$, the partial derivative of $\mathcal{H}$ with respect to the distribution parameters of π takes part in the gradient of $\mathcal{J}(\theta)$, and is going to be covered in 3.2.2 and 3.2.1.

3.2.1 Exploration in Discrete Actions Spaces

Even if in discrete actions spaces the actions a are sampled from a Categorical Distribution parameterized by the output probabilities vector ϕ of the network, we can enhance the exploratory behaviour of the agent by integrating the entropy bonus $\mathcal{H}$ to the policy objective function, controlled by the entropy coefficient $\beta > 0$. For a given categorical probability distribution ϕ on a single sample, the entropy is calculated using the formula:

$$\mathcal{H}^{Shannon}(\phi_t) = -\sum_{x \in \mathcal{X}} \phi_{t,x} \cdot \log \phi_{t,x} \tag{5}$$

Similarly with the previous case, we add to the gradient of the objective function the partial derivative of the negative entropy with respect to the probability distribution's ϕ parameters, multiplied by the regularization hyperparameter β.

$$\frac{\partial \mathcal{H}^{Shannon}(\phi_t)}{\partial \phi_t} = -(\log \phi_t + 1) \tag{6}$$

3.2.2 Exploration in Continuous Actions Spaces

For continuous actions, if σ is a fixed hyperparameter it directly induces the entropy in agent's actions since the actions are directly sampled from the normal distribution using Box-Muller method with a constant entropy, so there is no need for any regularization in this sense because we can control the entropy by modifing σ's value. On the other hand, if we choose σ to be a trainable parameter, we may introduce the regularization into the loss function. The complete equation of entropy bonus is the average over all samples from a minibatch, and the equation for a single example (also known as the Shannon entropy, extended for continuous variables) is defined as

$$\mathcal{H}^{Shannon}\left(\pi_\theta(a_t|s_t)\right) = -\int_{-\infty}^{\infty} \pi_\theta(a_t|s_t) \cdot \log \pi_\theta(a_t|s_t) = \frac{1}{2} \log\left(2\pi e \sigma_t^2\right) \tag{7}$$

Since we are using gradient ascent to optimize our model, when computing the partial derivative of the negative loss function with respect to σ, we also add the derivative of the negative entropy $\mathcal{H}$ with respect to σ (multiplied by the entropy coefficient $\beta > 0$).

$$\frac{\partial \mathcal{H}^{Shannon}\left(\pi_\theta(a_t|s_t)\right)}{\partial \sigma_t} = \frac{1}{\sigma_t} \tag{8}$$

3.3 Policy Gradient Methods

Policy Gradient Methods (PGM) represent one subclass in RL algorithms that directly optimizes a policy π parametrized by θ using gradient ascent. In any PGM method, the parameters of the policy are updated gradually towards the direction of the policy gradient, defined by the partial derivative of the expected cumulative reward with respect to each parameter θ. They are characterized by their ability to efficiently handle uncertainty and randomness in the environment. Unlike deterministic policies, which prescribe a single action for a given state, stochastic policies provide a probability distribution over the possible actions. This inherent stochasticity introduces the crucial element of exploration, enabling agents to discover more effective strategies in complex environments.

Like all Deep Reinforcement Learning techniques, they imply finding possible ways to solve the problem by letting multiple agents to explore new experiences hence optimize their actions. Each agent's experience is a single step t in the environment, and is generally denoted by a tuple of few important components: the state, the action performed for the given state, the reward received based on the action quality, the next state in which the agent enters after performing the action and a done flag that shows whether or not the next state is terminal. One agent's attempt to solve the problem is considered an episode, and it consists of multiple experiences, until reaching a final point (known as the terminal state) that determines either he managed to successfully finish his task, or he failed, and after that, the environment resets either to it's initial state or stochastically. As previously said, the agent is rewarded for the action he takes, but most of the time the reward function is sparse (for example, the agent receives one reward point only when he reached the terminal state), and according to the collected transitions/timesteps, all previous actions that led to obtain the reward were not gratified, as them should be. To accomplish rewarding all actions that contributed to the cumulated reward at a given timestep t in the episode, we will discount all of them in time by $\gamma \in (0, 1]$ for all of them, and it will be known as return R.

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i \tag{9}$$

In this context, the return represents the real quality of an action a_t took from a state s_t, which will be furtherly used in optimization algorithms. However, accurately determining the return involves employing a function approximator, which is trained on accumulated experiences. This function approximator is associated with a policy that, in turn, provides an estimate known as the Q value:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t] \tag{10}$$

Here, $Q^{\pi}(s_t, a_t)$ represents the expected return given a particular state-action pair (s_t, a_t). Several algorithms necessitate attributing a value solely to the state, disregarding any specific action taken. This leads to the formulation of the state-value function, denoted as $V^{\pi}(s_t)$:

$$V^{\pi}(s_t) = \mathbb{E}[R_t | s_t] \tag{11}$$

Expressing $V^{\pi}(s_t)$ in terms of $Q^{\pi}(s, a)$, where a is sampled from the policy π associated with the state s_t, results in the following relationship:

$$V^{\pi}(s_t) = \mathbb{E}_{a \sim \pi(\cdot|s)}[Q^{\pi}(s, a)] \tag{12}$$

In essence, 12 illustrates that the state-value function can be represented as the expected value of the Q values associated with all possible actions sampled from the policy at the given state. It's important to note that even if V^{π} and Q^{π} are trying to approximate the same value, the difference lies in the action a, in which for V is strictly dependent on π. As a real example, imagine an agent in the last step s_{T-1} before reaching a terminal state in s_T. For each step taken, the agent receives -1 penalty, and 10 reward if he reaches s_T. In this situation, $V(s_{T-1})$ will be equal to 9, and, if we consider he takes an action $\tilde{a}$ that moves the agent to another one step away state from s_T, he requires to do 2 steps in total to reach the terminal state, therefore $Q(s_{T-1}, \tilde{a})$ is 8.

In the next subsections will be presented two different state-of-the-art stochastic algorithms, members of Policy Gradient Methods, which exhibit unique characteristics that contribute to their effectiveness in addressing challenges within the field of deep reinforcement learning. The deterministic ones are briefly included in the Appendix.

3.3.1 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [SWD+17] is DRL algorithm from the class of PGMs that follows the same training procedure as vanilla PG, but using a stochastic policy. It's a on-policy algorithms since it uses only the recent collected experience trajectory to optimise it's parameters, and also a model-free algorithm. It was developed to resolve the problems of Trust Region Policy Optimization methods, previously introduced in [SLA+15], PPO requiring less computational resources for updating the model, allowing different architecture techniques, but most importantly, a conceptually simpler algorithm. Even though is a simpler algorithm to understand and implement, the original paper is sub-optimally documented, so [BW21] explains in depth all the details of PPO and enriched this documentation as well. In this subsection, we will cover the essential parts of the algorithm necessary for implementing it. PPO is not considered an Actor-Critic method because the policy π_θ and the value function V_ω can share parameters (in comparison with TRPO [SLA+15] algorithm that doesn't allow that) since they have the same state vector s, but in practice they are implemented as separate neural networks, where V_ω is associated as the critic. $V_\omega(s_t)$ returns a scalar value that represents the expected reward of the state s at the moment of time t. On the other hand π_θ, is represented by a neural network that outputs values based on the action spaces we require for our agent (see 3.1). The training process of the policy π_θ involves calculating advantages through the value function, which abstractly indicates whether the actions taken by the agent were favorable or unfavorable.

The main loop of the algorithm can be shortly described as follows: collect a certain amount of trajectories, compute the target values V^{target}, compute the advantages, minimize the V_ω mean squared error and maximize the objective function $L^{CLIP}(\theta)$.

Value Function

The role of the value function is to estimate the expected return, based on the data collected by the agent. Thus, after the experience buffer if fullfiled, we need to firstly compute the value of each trajectory in the buffer, which are then used as targets for our value function.

$$V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V_\omega(s_T) \tag{13}$$

As previously said, the scope of the value network is to approximate as good as possible the expected return, so we are going to use the mean squared error loss to minimize its error.

$$L_t^V = \left(V_\omega(s_t) - V_t^{target}\right)^2 \tag{14}$$

The partial derivative of Equation 14 w.r.t. the output of the value function is:

$$\frac{\partial L_t^V}{\partial V_\omega(s_t)} = 2 \cdot \left(V_\omega(s_t) - V_t^{target}\right) \tag{15}$$

Advantage Estimate

We know that V_ω offers an estimative information about the value of a state s_t. Though, if in our freshly collected experience buffer, we compute that the value of s_t is higher than what the value function expected to be, it means the action a_t he took this time is a good one, comparing to the average actions took previously in his history, so we say that the advantage of that a_t is positive (this logic applies similarly for bad actions). The advantages are later used in the loss function of the policy to indicate the sign and the size

of the gradient. The precise value of the advantage for each timestep t can be calculated by subtracting our value estimation from the actual value of the state previously computed as V^{target}:

$$A_t = V_t^{target}(s_t) - V_\omega(s_t) \tag{16}$$

The problem of on-policy algorithms, where the data available for optimization is not as steady since it varies over time with the agent's learning process, is addressed in [SML+15], where authors introduce a new estimator for the advantage function to mitigate these challenges. Their proposed solution known as Generalized Advantage Estimation (GAE), is particularly effective in on-policy RL scenarios. GAE extends the traditional advantage function by introducing a discounting parameter $\lambda > 0$ that allows for a balance between the one-step advantage and a more generalized advantage computed over multiple time steps. Their experiments with the best performances were obtained using intermediate values of $\gamma \in [0.99, 0.995]$ and $\lambda \in [0.96, 0.99]$. In the evaluation equation, $\hat{A}_t$ represents the advantage estimate at time step t, incorporating the immediate advantage δ_t and additional terms that account for the expected cumulative advantages over future time steps. Each term is discounted by $\gamma\lambda$, reflecting the impact of future events on the current advantage estimation.

$$\hat{A}_t = \sum_{k=t}^{T} (\gamma\lambda)^l \delta_{t+k}, \quad \text{where } \delta_t = r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t) \tag{17}$$

We can highlight that T may be infinite, it can represent the timestep of a terminal state, or it can be the minimum of the distance between the current timestep t with the timestep of the terminal state T and the *horizon* (explained in Subsection 3.4.7). When GAE is used to estimate the advantages, V^{target} is simply computed afterwards using the Equation 16 instead of 13 by switching the sides. The form of Equation 17 can be expanded to provide a clearer understanding:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \ldots + (\gamma\lambda)^{T-t+1}\delta_T$$

Policy

As said previously, advantages have their own weight to the policy loss function, which is defined as a surrogate loss function as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{18}$$

where $r_t(\theta)$ is the ratio between the probabilities given by the policy using the new and the old parameters respectively $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$.

The surrogate loss function is formulated to encourage an increase in the probability of actions that have positive advantage values and a decrease in the probability of actions with negative advantages. Its objective is to move the policy in the direction that improves performance by gradient ascent, within a limited step range by ceiling the probability ratio $r_t(\theta)$ to $1+\epsilon$. Mathematically speaking, the *min* and *clip* operators involved by the ceiling are not differentiable if their arguments are equal, though we are considering them differentiable by straight-through gradient estimate, similar to *ReLU* activation (which is not differentiable when $x = 0$). Equation 19 describes the partial derivative of surrogate L^{CLIP} w.r.t. the probability value $\pi_\theta(a_t|s_t)$ which evaluates Equation 18.

$$\frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} = \left(A_t \frac{\partial L_t^{CLIP}}{\partial r_t(\theta)} + A_t \frac{\partial L_t^{CLIP}}{\partial \text{clip}(r_t(\theta))} \frac{\partial \text{clip}(r_t(\theta))}{\partial r_t(\theta)} \right) \frac{\partial r_t(\theta)}{\partial \pi_\theta(a_t|s_t)} \tag{19}$$

Covering the derivatives of *min* and *clip* operators, the Equation 19 can be explicitly defined as:

$$\begin{aligned} \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} = \Bigg(& A_t \cdot \begin{cases} 1 & \text{if } r_t(\theta) A_t \leq \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t \\ 0 & \text{otherwise} \end{cases} \\ & + A_t \cdot \begin{cases} 1 & \text{if } \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t < r_t(\theta) A_t \\ 0 & \text{otherwise} \end{cases} \\ & \cdot \begin{cases} 1 & \text{if } 1-\epsilon \leq r_t(\theta) \leq 1+\epsilon \\ 0 & \text{otherwise} \end{cases} \Bigg) \cdot \frac{1}{\pi_{\theta_{\text{old}}}(a_t, s_t)} \end{aligned} \tag{20}$$

After computing the partial derivative of the surrogate loss function w.r.t. the probability distribution, we can differentiate it further w.r.t. the outputs of the neural networks in order to compute the parameters' gradients though back propagation. For continuous action spaces, the gradients of L^{CLIP} are:

$$\frac{\partial L_t^{CLIP}}{\partial \mu_\theta(s_t)} = \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_\theta(s_t)} = \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \cdot \pi_\theta(a_t|s_t) \cdot \frac{a_t - \mu_\theta(s_t)}{\sigma_\theta(s_t)} \tag{21}$$

$$\frac{\partial L_t^{CLIP}}{\partial \sigma_\theta(s_t)} = \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \frac{\partial \pi_\theta(a_t|s_t)}{\partial \sigma_\theta(s_t)} = \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \cdot \pi_\theta(a_t|s_t) \cdot \frac{(a_t - \mu_\theta(s_t))^2 - \sigma_\theta^2(s_t)}{\sigma_\theta^3(s_t)} \tag{22}$$

In case of discrete actions, since $\phi_\theta(s_t)$ returns the probabilities of each individual action, the gradient of $\pi_\theta(a_t|s_t)$ is 1 if the discrete action $i \in \mathcal{X}$ at the timestep t marked as $a_{t,i}$ was sampled, and 0 otherwise. In other words, we multiply the partial derivative of L^{CLIP} w.r.t. $\pi_\theta(a_t|s_t)$ with the one hot vector representation of a_t.

$$\frac{\partial L_t^{CLIP}}{\partial \phi_\theta(s_t)} = \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \frac{\partial \pi_\theta(a_t|s_t)}{\partial \phi_\theta(s_t)} = \frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \cdot \begin{cases} 1 & \text{if } \phi_{\theta,i}(s_t) \sim a_t \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

Since we want to maximize L_{CLIP}, we need to do gradient ascent by multiplying the derivatives from Equations 21, 22 and 23 with -1 right before backpropagating them, unless a *maximization* setting is already implemented in the optimizer, which adds the gradient instead of subtracting it from the parameter.

Algorithm 12 Proximal Policy Optimization (PPO)

Require: N: Number of parallel agents
Require: T: Trajectory's length
Require: π_θ and $optim_\theta$: Policy ($\pi_\theta.\mu_\theta$ and $\pi_\theta.\sigma_\theta$ networks) and it's optimizer
Require: V_ω and $optim_\omega$: Value network and it's optimizer
Require: Ω: Observations online normalizer (see Subsection 3.4.1)
Require: $\alpha \in [5 \times 10^{-6}, 3 \times 10^{-4}]$, $\gamma \in [0.99, 0.995]$, $\lambda \in [0.92, 0.98]$, $\epsilon \in [0.1, 0.3]$, $h \in [64, 2048]$, $\beta \in [5 \times 10^{-3}, 2 \times 10^{-2}]$, $l_\infty \in [0.1, 0.5]$, $d_{target} \in [0.015, 0.1]$, $v_{coef} \in [0.5, 1]$: Learning rate, discount factor, bias-variance trade-off coefficient, clip factor, time horizon, entropy bonus coefficient, max infinity norm, KL divergence threshold and value coefficient

Initialize π_θ and V_ω and their optimizers
Initialize N parallel agents in N different environments
repeat
    buffer $\leftarrow$ []
    // Collect trajectories
    for t = 1, 2, ... , T **do**
        for each agent $\in$ parallel agents **do**
            $s_t \leftarrow agent.CollectObservations()$
            $s_t \leftarrow \Omega.UpdateThenNormalize(s_t).Clip()$
            $a_t \in \mathbb{R} \leftarrow \pi_\theta(\cdot|s_t)$
            $\pi_{\theta_{old}}(a_t|s_t) \leftarrow \pi_\theta.distibution.probability(a_t)$
            $r_t \leftarrow$ agent.$OnActionReceived$(Tanh(a_t)) ▷ Use a_t in the environment
            $s_{t+1} \leftarrow$ agent.$CollectObservations()$
            $s_{t+1} \leftarrow \Omega.Normalize(s_{t+1}).Clip()$
            if s_{t+1} is terminal **then**
                agent.$environment.Reset()$
            end if
            agent.$trajectory \leftarrow$ agent.$trajectory$ + (s_t, a_t, $\pi_{\theta_{old}}(a_t|s_t)$, r_t, s_{t+1})
        end for
    end for
    // Compute rewards-to-go and advantages
    for each agent $\in$ parallel agents **do**
        for t = 1, 2, ... , T in agent.$trajectory$ **do**
            $\hat{A}_t \leftarrow \sum_{i=t}^{min(t+h,T) \text{ or } s_{i+1} \text{ is terminal}} (\gamma\lambda)^i (r_i + \gamma V_\omega(s_{i+1}) - V_\omega(s_i))$
            $V_t^{target} \leftarrow \hat{A}_t + V_\omega(s_t)$
        end for
        buffer $\leftarrow$ buffer + $agent.trajectory$
        agent.$trajectory \leftarrow$ []
    end for

// Update parameters θ and ω
for however many epochs **do**
  buffer $\leftarrow Shuffle$(buffer)
  minibatches $\leftarrow SplitIntoMinibatches$(buffer)
  for each m $\in$ minibatches **do**
    // Update the value network by gradient descent
    $\frac{\partial L_t^V}{\partial V_\omega(s_t)} \leftarrow 2 \cdot \left(V_\omega(s_t) - V_t^{target}\right)$
    $optim_\omega.ZeroGrad()$
    $V_\omega.Backward\left(v_{coef} \cdot \frac{\partial L_t^V}{\partial V_\omega(s_t)}\right)$
    $optim_\omega.GradClipNorm(l_\infty)$
    $optim_\omega.Step(\alpha)$
    // Update the policy network by gradient ascent
    m $\leftarrow NormalizeAdvantages$(m) ▷ See Subsection 3.4.3
    $\mu_\theta(s_t) \in \mathbb{R}, \sigma_\theta(s_t) \in \mathbb{R}+ \mid \phi_\theta(s_t) \leftarrow \pi_\theta(\cdot|s_t)$ ▷ Reparam. π_θ distribution
    $\pi_\theta(a_t|s_t) \leftarrow PDF(a_t, \mu_\theta(s_t), \sigma_\theta(s_t))$
    $\frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \leftarrow \left(A_t \frac{\partial L_t^{CLIP}}{\partial r_t(\theta)} + A_t \frac{\partial L_t^{CLIP}}{\partial \text{clip}(r_t(\theta))} \frac{\partial \text{clip}(r_t(\theta))}{\partial r_t(\theta)}\right) \frac{\partial r_t(\theta)}{\partial \pi_\theta(a_t|s_t)}$
    $optim_\theta.ZeroGrad()$
    if using continuous actions **then**
      $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_\theta(s_t)} \leftarrow \pi_\theta(a_t|s_t) \cdot \frac{a_t - \mu_\theta(s_t)}{\sigma_\theta(s_t)}$
      $\pi_\theta.\mu_\theta.Backward\left(-\frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \cdot \frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_\theta(s_t)}\right)$
      if σ is trainable **then**
        $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \sigma_\theta(s_t)} \leftarrow \pi_\theta(a_t|s_t) \cdot \frac{(a_t - \mu_\theta(s_t))^2 - \sigma_\theta^2(s_t)}{\sigma_\theta^3(s_t)}$
        $\frac{\partial \mathcal{H}(\pi_\theta(a_t|s_t))}{\partial \sigma_\theta(s_t)} \leftarrow \frac{1}{\sigma_\theta(s_t)}$
        $\pi_\theta.\sigma_\theta.Backward\left(-\frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \cdot \frac{\partial \pi_\theta(a_t|s_t)}{\partial \sigma_\theta(s_t)} - \beta \frac{\partial \mathcal{H}(\pi_\theta(a_t|s_t))}{\partial \sigma_\theta(s_t)}\right)$
      end if
    end if
    if using discrete actions **then**
      $\frac{\partial \mathcal{H}(\phi_\theta(s_t))}{\partial \phi_\theta(s_t)} \leftarrow -\log \phi_\theta(s_t) - 1$
      $\pi_\theta.\phi_\theta.Backward\left(-\frac{\partial L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} \odot \vec{a_t} - \beta \frac{\partial \mathcal{H}(\phi_\theta(s_t))}{\partial \phi_\theta(s_t)}\right)$
    end if
    $optim_\theta.GradClipNorm(l_\infty)$
    $optim_\theta.Step(\alpha)$
    // Check for early stopping
    if m last minibatch **and** $KL(\pi_\theta(a_t|s_t), \pi_{\theta_{old}}(a_t|s_t)) > d_{target}$ **then**
      halt and optionally Rollback θ
    end if
  end for
  $\alpha, \beta, \epsilon \leftarrow Schedule(\alpha, \beta, \epsilon)$ ▷ Linear decay of hyperparameters
end for
until π_θ converged
return π_θ

3.3.2 Soft Actor-Critic

The Soft Actor Critic (SAC) algorithm, initially introduced in [HZAL18], operates as an off-policy learning algorithm. This implies that the agent can leverage historical data, not necessarily encountered by the current policy—here, it assimilates knowledge from previously collected experiences. The algorithm employs an extensive replay buffer to store all experiences, still used till the end of the training (in contrast to on-policy methods that clears the buffer post-update). The maximum-entropy characteristic in contrast to DDPG [SLH+14] signifies SAC's dual objective: it doesn't only seek to maximize the expected cumulative rewards, but it also strives to maximize the entropy of the policy. SAC optimizes a trade-off between these objectives, introducing stochasticity to the agent's behavior so it becomes less deterministic resulting in a more exploratory behaviour that doesn't limit the convergence to a local optima. The SAC algorithm comprises two entities, the Actor and the Critic, presented furtherly based on the modern adaptation from OpenAI's SpinningUp documentation[1].

Two noteworthy points distinguish SAC from PPO. Firstly, SAC demands more computational effort for network updates in exchange for a slightly better final policy. Secondly though, it exhibits higher sample efficiency, making it suitable for environments that pose more effort in terms of simulation.

Critic

Twin SAC's Critic consists of two different networks Q_{ϕ_1} and Q_{ϕ_2} similar to Double Q-learning and TD3 algorithms, parametrized by ϕ_1 and ϕ_2 respectively. This is due the fact the Q-function is slowly overestimating Q-values leading to disrupting the policy, and the problem was addressed by taking the minimum between the two. The parameters are linearly updated with a soft transition of Q networks towards their objective, so we will also require to use a set of target parameters, $\phi_{targ,1}$ and $\phi_{targ,2}$. The regularized formula that adds discounted entropy along the reward for all timesteps but first is:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t + \beta\sum_{t=1}^{T}\gamma^t \mathcal{H}\left(\pi(\cdot|s_t)\right)\Big| s_0 = s, a_0 = a\right] \tag{24}$$

Note that s_0 and a_0 are actually the received parameters s and a, and not the first elements of an episode, so the computation takes in account only the states and actions that are after s and a. The right hand side part in the equation is an expectation over next states (which come from the replay buffer) and next actions (which come from the current policy, and not the replay buffer). Since it's an expectation, we can approximate it with samples of actions:

$$Q^{\pi}(s,a) \approx r + \gamma\left(Q^{\pi}(s',\tilde{a}') - \beta\log\pi(s'|\tilde{a}')\right), \quad \tilde{a}' \sim \pi(\cdot|s') \tag{25}$$

where $\tilde{a}_t$ represents a newly sampled action from the policy on s_t. Note that the scalar probability of an action a in a state s denoted as $\pi(s|a)$ is the product of all independent continuous actions probabilities in the action vector a (extracted from the probability mass function 2). The target of the Q function is defined as:

$$Q_{\phi,t}^{target} = \begin{cases} r_t & \text{if } s_{t+1} \text{ is terminal} \\ r_t + \gamma V_{\omega}(s_{t+1}) & \text{otherwise} \end{cases} \tag{26}$$

where $V_{\omega}(s_{t+1})$ is the value of the state at the next time step. Excluding the learning of a Value function, the Equation 26 can use the Q networks by sampling a new action from the policy as follows:

[1] `https://spinningup.openai.com/en/latest/algorithms/sac.html`

$$Q_{\phi,t}^{target} \leftarrow \begin{cases} r_t \text{ if } s_{t+1} \text{ is terminal, else} \\ r_t + \gamma \left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s_{t+1}, \tilde{a}_{t+1}) - \beta \log \pi_\theta(\tilde{a}_{t+1}|s_{t+1}) \right), \quad \tilde{a}_{t+1} \sim \pi_\theta(\cdot|s_{t+1}) \end{cases} \tag{27}$$

After calculating the Q functions targets for each individual timestep in the extracted minibatch, the Q networks are updated via gradient descent by minimizing the Mean Squared Bellman Error (MSBE), so the gradient of the loss function w.r.t. the output of each Q network is

$$\frac{\partial}{\partial Q_{\phi_i}^{\pi}(s_t, a_t)} \left(Q_{\phi_i}^{\pi}(s_t, a_t) - Q_{\phi,t}^{target} \right)^2 = 2 \cdot \left(Q_{\phi_i}^{\pi}(s_t, a_t) - Q_{\phi,t}^{target} \right), \text{ for i = 1,2} \tag{28}$$

usually multiplied by a coefficient of $1/2$. When ϕ_1 and ϕ_2 parameters were optimised, we are softly updating the target parameters $\phi_{targ,1}$ and $\phi_{targ,2}$ by Polyak averaging using a fixed coefficient $\tau > 0$.

$$\phi_{targ,i} \leftarrow (1 - \tau)\phi_{targ,i} + \tau\phi_i, \quad \text{for i = 1, 2} \tag{29}$$

Actor

We define a new action $\tilde{a}_\theta(s, \xi)$ using the reparametrization trick firstly introduced in [KW13] thus to make the loss differentiable w.r.t. the actor network, which is represented by two neural networks μ_θ and σ_θ that are generating actions in the continuous space (it is not necessary that these networks share the same parameters θ). In order to update them, we need to generate such actions by passing s as input to for the actor, hence obtaining $\mu_\theta(s)$ and $\sigma_\theta(s)$, then sampling a random vector $\xi \sim \mathcal{N}(0, I)$, where I is the identity matrix. Discrete actions can be obtained only by partitionating the $(-1, 1)$ range.

$$\tilde{a}_\theta(s) = \text{Tanh}(u_\theta(s)), \text{ where } u_\theta(s) = \mu_\theta(s) + \sigma_\theta(s) \odot \xi, \; \xi \sim \mathcal{N}(0, I) \tag{30}$$

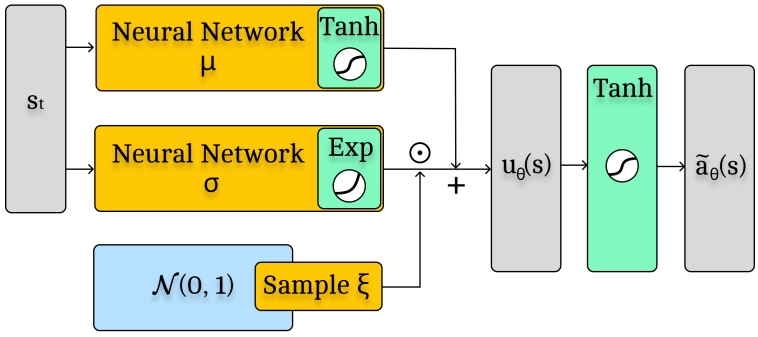


Figure 3: Generation of $u_\theta(s)$ and $\tilde{a}_\theta(s)$

In SAC, the objective function, which is the loss of the policy network, is defined as the negative log-likelihood of an action a in a state s times an entropy coefficient $\beta \sim 0.2$ added to the minimum of the Q values estimators given by $(s, \tilde{a}_\theta(s))$ pair:

$$L^\pi(\theta) = \mathbb{E}\left[\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \beta \log \pi_\theta(\tilde{a}_\theta(s)|s) \right] \tag{31}$$

Checking the Appendix C of [HZAL18], the authors use an unbounded Gaussian $\mu(u|s)$ for action distribution on which they apply the squashing function $Tanh$ for bounding the actions to the finite support $(-1, 1)$. The log-likelihood has the following form:

$$\log \pi_\theta(\tilde{a}_\theta(s)|s) = \log \mu(u_\theta(s)|s) - \sum_{i=1}^{D} \log(1 - \text{Tanh}^2(u_{\theta,i}(s))) \tag{32}$$

where D is the actions dimension and $\mu(u_\theta(s)|s)$ is a multivariate Gaussian (normal) distribution. The density function for a MVN distribution [Duc07] with mean μ and covariance matrix Σ is:

$$p(x) = \frac{1}{(2\pi)^{n/2}\det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{33}$$

Since our covariance matrix Σ has the diagonal with σ^2 values and all off-diagonal elements are zero, the determinant is simply the product of all diagonal elements. To simplify the calculations, we can observe that $(x-\mu)^T\Sigma^{-1}(x-\mu)$ which is $\sum_{i=1}^{D}(u_\theta(s) - \mu_i)^2/\sigma_i^2$ can be replaced by $\sum_{i=1}^{D}\xi_i^2$, since $u_{\theta,i} = \mu_{\theta,i}(s) + \sigma_{\theta,i}(s)\cdot\xi_i$.

$$\mu(u|s) = \frac{1}{\sqrt{(2\pi)^D}\prod_{i=1}^{D}\sigma_i} \exp\left(-\frac{1}{2}\sum_{i=1}^{D}\xi_i^2\right) \tag{34}$$

So, to compute the gradients of the parameters for SAC loss function, we need to differentiate it w.r.t. the output of each neural network, $\mu_\theta(s_t)$ and $\sigma_\theta(s_t)$. The objective loss function must be differentiated w.r.t. the output of μ_θ and σ_θ networks, but since we cannot compute it directly, we expand the differential equations:

$$\frac{\partial L_t^\pi}{\partial u_\theta(s_t)} = \frac{\partial \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s_t))}{\partial u_\theta(s_t))} - \beta \cdot \frac{\partial \log \pi_\theta(\tilde{a}_\theta(s_t)|s_t)}{\partial u_\theta(s_t)} \tag{35}$$

The first element in both differential equations can be obtained by firstly computing the partial derivative of $\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s_t))$ w.r.t. $\tilde{a}_\theta(s_t)$, which is the backpropagation through the network of $\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s_t))$ value, and then extracting only the $\tilde{a}_\theta(s_t)$ part from the state-action compound gradient. Therefore, we multiply by the gradient of Tanh w.r.t. $u_\theta(s_t)$ to obtain the partial derivative of $\tilde{a}_\theta(s_t)$ w.r.t. $u_\theta(s_t)$, then a final multiplication with the partial derivative of $u_\theta(s_t)$ w.r.t. $\mu_\theta(s_t)$ for the μ_θ head, which is simply 1. The same principle is used similarly for the σ_θ head, but it is multiplied by ξ_t instead. The second part of Equation 35 is expanded into another two partial derivatives w.r.t. $u_\theta(s_t)$ since Equation 32 it is composed of two different elements.

The Equation 31 can be differentiated as follows:

$$\frac{\partial L_t^\pi}{\partial u_\theta(s_t)} = \frac{\partial \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s_t))}{\partial \tilde{a}_\theta(s_t)} \frac{\partial \tilde{a}_\theta(s_t)}{\partial u_\theta(s_t)} \tag{36}$$

$$- \beta \left(\frac{\partial \log \mu(u_\theta(s_t)|s)}{\partial u_\theta(s_t)} - \frac{\partial \log(1 - \text{Tanh}^2(u_\theta(s_t)))}{\partial u_\theta(s_t)}\right) \tag{37}$$

That is:

$$\frac{\partial L_t^\pi}{\partial u_\theta(s_t)} = \nabla_{\tilde{a}_\theta(s_t)} \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s_t)) \cdot \left(1 - \text{Tanh}^2(u_\theta(s_t))\right)$$
$$- \beta \left(\frac{u_\theta(s_t) - \mu_\theta(s_t)}{\sigma_\theta^2(s_t)} - \frac{-2 \cdot \text{Tanh}(u_\theta(s_t)) \cdot \text{Sech}^2(u_\theta(s_t))}{1 - \text{Tanh}^2(u_\theta(s_t))}\right) \tag{38}$$

Algorithm 13 Soft Actor-Critic (SAC)

Require: N: Number of parallel agents
Require: π_θ and $optim_\theta$: Policy ($\pi_\theta.\mu_\theta$ and $\pi_\theta.\sigma_\theta$ networks) and it's optimizer
Require: Q_{ϕ_1}, Q_{ϕ_2} and $optim_\phi$: Q_1 and Q_2 network and their optimizer
Require: $Q_{\phi_{targ,1}}$ and $Q_{\phi_{targ,2}}$ networks
Require: $\alpha \in [10^{-4},\ 10^{-3}]$, $\gamma \in [0.99,\ 0.995]$, $\beta \in [0.1,\ 0.3]$, $\tau \in [10^{-3},\ 5 \times 10^{-3}]$: Learning rate, discount factor, entropy coefficient and interpolation factor

Initialize π_θ, Q_{ϕ_1}, Q_{ϕ_2} networks and their optimizers
Initialize $Q_{\phi_{targ,1}}$, $Q_{\phi_{targ,2}}$ networks and set $\phi_{targ,1} \leftarrow \phi_1$, $\phi_{targ,2} \leftarrow \phi_2$
Initialize N parallel agents in N different environments
Initialize buffer $\leftarrow []$
repeat
    // Collect trajectories
    for each agent $\in$ parallel agents **do**
        $s_t \leftarrow agent.CollectObservations()$
        $a_t \in \mathbb{R} \leftarrow \pi_\theta(\cdot|s_t)$
        $r_t \leftarrow$ agent.$OnActionReceived(\text{Tanh}(a_t))$ ▷ Use a_t in the environment
        $s_{t+1} \leftarrow$ agent.$CollectObservations()$
        if s_{t+1} is terminal **then**
            agent.$environment.Reset()$
        end if
        buffer $\leftarrow$ buffer + timestep$_t(s_t,\ a_t,\ r_t,\ s_{t+1})$
    end for
    if new k transitions were collected **then** ▷ $k \sim N$ is the update frequency
        for however many epochs **do**
            minibatch $\leftarrow SampleMinibatch(\text{buffer})$
            // Update Q functions by gradient descent
            $\tilde{a}_{t+1} \leftarrow \pi_\theta(\cdot|s_{t+1})$
            $$Q_{\phi,t}^{target} \leftarrow \begin{cases} r_t \text{ if } s_{t+1} \text{ is terminal, else} \\ r_t + \gamma \left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s_{t+1}, \tilde{a}_{t+1}) - \beta \log \pi_\theta(\tilde{a}_{t+1}|s_{t+1}) \right) \end{cases}$$
            $$\frac{\partial \left(Q_{\phi_i}(s_t,a_t) - Q_{\phi,t}^{target}\right)^2}{\partial Q_{\phi_i}(s_t,a_t)} \leftarrow 2 \cdot \left(Q_{\phi_i}(s_t,a_t) - Q_{\phi,t}^{target} \right), \quad \text{for i} = 1,\ 2$$
            $optim_\phi.ZeroGrad()$
            $$Q_{\phi_i}.Backward\left(\frac{\partial \left(Q_{\phi_i}(s_t,a_t) - Q_{\phi,t}^{target}\right)^2}{\partial Q_{\phi_i}(s_t,a_t)} \right), \quad \text{for i} = 1,\ 2$$
            $optim_\phi.Step(\alpha)$

// Update Policy function by gradient ascent
$u_\theta(s_t) = \pi_\theta.\mu_\theta(s_t) + \pi_\theta.\sigma_\theta(s_t) \odot \xi_t\,, \quad \xi \sim \mathcal{N}(0, I) \qquad \triangleright\ \pi_\theta.\mu_\theta(s_t) \in (-1, 1)$
$\tilde{a}_\theta(s_t) = \text{Tanh}(u_\theta(s_t))$
$\frac{\partial \min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s_t))}{\partial \tilde{a}_\theta(s_t)} \leftarrow Q_{\phi_1}.Backward\left(\frac{\partial \min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s_t))}{\partial Q_{\phi_1}(s,\tilde{a}_\theta(s_t))}\right) +$
$Q_{\phi_2}.Backward\left(\frac{\partial \min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s_t))}{\partial Q_{\phi_2}(s,\tilde{a}_\theta(s_t))}\right)$
$\frac{\partial \min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s_t))}{\partial u_\theta(s_t)} \leftarrow \frac{\partial \min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s_t))}{\partial \tilde{a}_\theta(s_t)} \cdot \left(1 - \text{Tanh}^2(u_\theta(s_t))\right)$
$\frac{\partial L_t^\pi}{\partial u_\theta(s_t)} \leftarrow \frac{\partial \min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s_t))}{\partial u_\theta(s_t)} - \beta\left(\frac{u_\theta(s_t) - \mu_\theta(s_t)}{\sigma_\theta^2(s_t)} - \frac{-2 \cdot \text{Tanh}(u_\theta(s_t)) \cdot \text{Sech}^2(u_\theta(s_t))}{1 - \text{Tanh}^2(u_\theta(s_t))}\right)$
$optim_\theta.ZeroGrad()$
$\pi_\theta.\mu_\theta.Backward\left(-\frac{\partial L_t^\pi}{\partial u_\theta(s_t)} \cdot 1\right)$
$\pi_\theta.\sigma_\theta.Backward\left(-\frac{\partial L_t^\pi}{\partial u_\theta(s_t)} \cdot \xi_t\right)$
$optim_\theta.Step(\alpha)$
$\phi_{targ,i} \leftarrow (1 - \tau)\phi_{targ,i} + \tau\phi_i\,, \quad \text{for i} = 1, 2 \qquad \triangleright$ Update target parameters
end for
end if
until convergence
return π_θ

Hyperparameters	
Nonlinearity	Tanh
Hidden Layers	2
Hidden Units	64
Optimizer	Adam
Learning rate (α)	3×10^{-4}
Discount (γ)	0.99
Learning rate decay	✗
PPO specific	
Batch size	512
Buffer size	10240
Horizon	256
Num. epochs	8
Entropy coeff. (β)	5×10^{-3}
Clip factor (ϵ)	0.2
GAE parameter (λ)	0.96
Max norm (l_∞)	0.5
Value coeff.	0.5
Target KL (d_{target})	0.015
Advantage normalization	✓
SAC specific	
Batch size	128
Replay buffer size	10^6
Update interval (*steps*)	64
Update after (*steps*)	1024
Num. updates	1
Entropy coeff. (β)	0.2
Target smooth coeff. (τ)	5×10^{-3}

Table 3.1: Default hyperparameters used in DeepUnity

3.4 Optimizations

Small tricks and adjustments are necessary for the algorithms to operate on our available hardware in a reasonable amount of time. These enhancements improve stability, efficiency, and convergence, and can even solve previously impossible tasks.

3.4.1 Preprocessing Observations

Neural networks can be sensitive to high input values for several reasons that can lead to difficulties in training and convergence. When input values are very large, the gradients can also become large, causing weight updates to be large and unstable. This can lead to convergence problems, such as the network failing to converge or overshooting the optimal weights. Many activation functions, such as the logistic sigmoid or hyperbolic tangent (Tanh), saturate as their input values become very large. When an activation function saturates (see Section 2.2.2), its derivative becomes close to zero, which results in very small gradients during back propagation. Normalizing the input data within [-1, 1] or [0, 1] is the way to go to mitigate such issues:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{39}$$

Even though the range limits (if unknown) can be found using a *min-max online normalizer*, the training process will become unstable especially if abnormally large or small values are met. This problem can be handled using an online standardizer, which is more responsive to unusual and less frequent values and becomes lineary stable during the training process, and can be further combined with clipping, preventing rare, exceptionally large values. Each observation is normalized by subtracting a running mean and then divide it by a running variance (or standard deviation). It is important to note that for off-policy algorithms, the raw states must be stored and normalized each time they are fed into the models.

Algorithm 14 Online Normalizer. Updated only during training, before normalization.

Require: n: Total number of data points
Require: $\mu \in \mathcal{R}^{|x|}$: Mean
Require: $M2 \in \mathcal{R}^{|x|}$: 2nd moment
Initialize $n \leftarrow 0$, $\mu \leftarrow 0$, $M2 \leftarrow 0$
procedure UPDATE(x)
    $n_t \leftarrow n_{t-1} + 1$
    $\Delta_1 \leftarrow x - \mu_{t-1}$
    $\mu_t \leftarrow \mu_{t-1} + \frac{\Delta_1}{n}$
    $\Delta_2 \leftarrow x - \mu_t$
    $\mathrm{M2}_t \leftarrow \mathrm{M2}_{t-1} + \Delta_1 \cdot \Delta_2$
end procedure
procedure NORMALIZE(x)
    if $n \leq 1$ **then**
        return x
    else
        return $\frac{x - \mu}{\sigma^2}$, where $\sigma^2 \leftarrow \frac{M2}{n-1}$ ▷ Optional post-clipping in [-5, 5]
    end if
end procedure

3.4.2 Early Stopping

Early Stopping optimization technique can be adapted to PPO [SWD+17] (or other on-policy methods as introduced in the openai/spinningup library's implementation of the algorithm, which halts the policy update within an epoch if the mean Kullback-Leibler (KL) Divergence between the current policy and the target policy computed over the last batch of the epoch becomes too high (it surpasses a threshold value d_{target}), more precisely prevents updates that change the policy to abrubtly. [DHOM21] studies it's effect used in conjunction with other common code-level optimizations, showing that PPO is sensitive to the number of epochs per iteration and Early Stopping mitigates such sensitivity by dynamically adjusting the number of policy update iterations applied within an epoch.

KL Divergence, denoted $D_{KL}(P||Q)$, represents the statistical non-negative distance between ones probability distribution P and a reference probability distribution Q. If two distributions are exactly the same, the distance is equal to 0. The relative entropy from the discrete probability distributions P and Q, defined on the common sample space $\mathcal{X}$, is expressed as a sum over the discrete probabilities:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{40}$$

For continuous random variables, where we have normal distributions $P = \mathcal{N}(\mu_p, \sigma_p)$ and $Q = \mathcal{N}(\mu_q, \sigma_q)$, the relative entropy is expressed as:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} P(x) \log \left(\frac{P(x)}{Q(x)} \right) = \log \frac{\sigma_q}{\sigma_p} + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} - \frac{1}{2} \tag{41}$$

The divergence is computed at every epoch K over the last batch M, and if $D_{KL}(\theta_K, \theta_{k-1}) > d_{target}$, we stop the update at the epoch K (KLE-Stop). Though, this means that the policy π_{θ_K} already surpassed at epoch K our current threshold d_{target}. To keep π entropy under d_{target}, we can rollback θ to θ_{K-1} before stopping, which makes the policy "less susceptible to degenerate performance".

Algorithm 15 Early Stopping for Proximal Policy Optimization

Require: d_{target}: KL divergence threshold (default: 0.015)

repeat

- Fill experience buffer D using $\pi_{\theta_{\text{old}}}$

for however many epochs **do**

- Update ω_{k-1} to ω_k for each minibatch $M \in D$
- Update θ_{k-1} to θ_k for each minibatch $M \in D$

Compute $D_{\text{KL}}(\theta_k, \theta_{\text{old}})$ for the last minibatch $M \in D$

if $D_{\text{KL}}(\theta_k, \theta_{\text{old}}) > d_{target}$ **then**

break if using *KLE-Stopping*

break if using *KLE-Rollback* and then $\theta_k \leftarrow \theta_{k-1}$

end if

end for

until convergence

3.4.3 Advantage Normalization

Especially if the rewards are not scaled, normalizing the advantages in advantage based algorithms stabilizes the training, and even offer better generalization, since the advantage A has direct weight in the objective functions. In [ARS+20], the normalization is done per-minibatch (instead over the entire buffer), and it seem to be better for not so small minibatches. The advantages are normalized by subtracting their mean and divide the difference by their standard deviation. Consider shuffling the training data every epoch.

$$A_t = \frac{A_t - \mu_m}{\sigma_m} = \frac{A_t - \frac{1}{m}\sum_{i=1}^{m} A_i}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}(A_i - \mu_A)^2 + \epsilon}} \text{ where } m \text{ is the minibatch size.} \tag{42}$$

3.4.4 Geometric Reward

The reward function can assign the agent multiple partial reward points simultaneously in a single step, reflecting various aspects of its actions. For instance, a walker might receive rewards at each time step for maintaining an upright head position, progressing towards a target, and sustaining the correct orientation relative to the target, each with distinct weighting proportions. In practical scenarios, the agent may exhibit a bias toward maximizing simpler or more weighted rewards. In the given example, the walker might opt for a strategy of standing straight with an upright head to accumulate more points, rather than taking the risk of moving forward. To mitigate the tendency to prioritize specific aspects over others without excessively promoting exploratory behavior at the cost of learning capabilities, the step reward is determined as the product of all the partial rewards for that step multiplied by a scaling coefficient, getting rid of any partial reward weighting. Be aware that this approach is sensitive to negative rewards, due to the fact that multiplying an odd number of rewards results in a positive reward.

3.4.5 Actions in between Sparse Decisions

For each step in the environment, the agent's observation input must pass through the network in order to obtain actions. This is a huge computational effort, especially for large neural networks and high physics frame rates. Reducing the physics update frequency may end up in poor, inaccurate simulations, and instead, we will reduce the inference frequency. Instead of taking a decision every single step (which means generating a new action), we will make one every $x > 1$ steps, and the intermediate steps will use the action determined by the last decision (note that the agent is forced to take a decision in the first step of an environment episode) As an example, let's say that we have an environment that runs a step every 0.01 seconds, which means a rate of 100 frames per second. We decide that the agent will take a decision every 5 steps, so for the steps $s_1, \ ... \ , s_5$ the agent will use the action a_1 generated at step s_1, then the next steps $s_6, s_7, \ ... \ , s_{10}$ the agent uses the action a_2 generated at step s_6 and so on. This decision reduction should not be overly exaggerated otherwise the agent will not be capable to converge to a solution. We will consider that the reward is cumulative in between decisions, so the reward r_1 for a_1 is computed over all actions determined by that decision. Continuing on our previous example, r_1 associated to a_1 equals to the sum of all rewards from steps $s_1, s_2, \ ... \ , s_5$; r_2 associated to a_2 equals to the sum of all rewards from steps $s_6, s_7, \ ... \ , s_{10}$ etc.

3.4.6 Stacked Inputs

One method that does give to the agent a limited memory of size $k > 1$ is to use a sequential input $[s_{t-k}, s_{t-k+1}, \ldots, s_{t-1}, s_t]$. By using recurrent cells 2.2.6 (or any other variant like LSTM [HS97] or GRU [CVMBB14]) to handle the sequence of inputs may give an unnecessary computational overhead, and instead, due to their complexity, we can concatenate them in a single input vector, thus maintaining our neural network models as simple feed forward networks. At the beginning of an episode, all precedent states are set neutral to 0.

$$s_1 = \begin{cases} [0, 0, \ \ldots \ , 0] \\ [0, 0, \ \ldots \ , 0] \\ \quad \vdots \\ [0, 0, \ \ldots \ , 0] \\ [x_1^1, x_2^1, \ \ldots \ , x_n^1] \end{cases} \quad s_2 = \begin{cases} [0, 0, \ \ldots \ , 0] \\ [0, 0, \ \ldots \ , 0] \\ \quad \vdots \\ [0, 0, \ \ldots \ , 0] \\ [x_1^1, x_2^1, \ \ldots \ , x_n^1] \\ [x_1^2, x_2^2, \ \ldots \ , x_n^2] \end{cases} \quad \ldots \quad s_t = \begin{cases} [x_1^{t-k}, x_2^{t-k}, \ \ldots \ , x_n^{t-k}] \\ [x_1^{t-k+1}, x_2^{t-k+1}, \ \ldots \ , x_n^{t-k+1}] \\ \quad \vdots \\ [x_1^{t-1}, x_2^{t-1}, \ \ldots \ , x_n^{t-1}] \\ [x_1^t, x_2^t, \ \ldots \ , x_n^t] \end{cases}$$

3.4.7 Time Horizon

Computing the expected returns can really slow down the process if the episodes are very long or even infinite, because this process goes ahead until reaching the final step or the end of the buffer. Since we are using a discounting factor $\gamma \approx 0.99$, the reward at a very far step into the future will be discounted so drastically as it can be omitted. Reminding that the value of a state at timestep t is calculated within a loop ranging $t \to T$, by adding a time horizon h the loop will go $t \to min(t + h, T)$.

Note that the same principle can be used when computing the generalized advantage estimate (GAE). The time horizon can be used as well to close the range for looking over long term rewards, but it is not a popular choice because this can be simply controlled by reducing the value of the discounting factor γ.

3.4.8 Spectral Normalization

Spectral Normalization (SN) [MKKY18] is a type of weight normalization that maintaints the Lipschitz continuity of the layers. Controlling the Lipschitz constant keeps neural network gradients in check, mitigating issues like gradient explosion in GANs [GPAM$^+$14] training and helps stabilize PGMs' policies as well.

Algorithm 16 Spectral Normalization. Applied before every forward and backward propagations. The weights are denormalized afterwards, immediately before SGD step.

Require: $\tilde{u} \in \mathcal{R}^{H_{out}}$ for each weight matrix $W^{H_{out} \times H_{in}} \in \theta$

1: Initialize every $\tilde{u} \sim \mathcal{N}(0, 1)$
2: **repeat**
3: **for** however many iterations **do** ▷ Determine largest singular value of each W
4: $\tilde{v} \leftarrow W^T \tilde{u} / ||W^T \tilde{u}||_2$
5: $\tilde{u} \leftarrow W \tilde{v} / ||W \tilde{v}||_2$
6: **end for**
7: $W_{SN} \leftarrow W / \left(\tilde{u}^T W \tilde{v}\right)$ ▷ Normalize each set of weights

- Compute $\nabla_\theta \mathcal{J}(\theta_{SN})$ using normalized weights $W_{SN} \in \theta_{SN}$
- Update unnormalized $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{J}(\theta_{SN})$ (with SGD variant)

8: **until** convergence

3.4.9 Distributed Priority of Experiences

Uniform experience sampling for policy updates in off-policy algorithms often will include outdated and insignificant experiences that fail to contribute to the policy convergence. The authors of [SQAS15] proprose a stochastic prioritization of the timesteps, making the most effective use of the replay buffer considering also it's finite space. Their idea is to assign a priority value to each timestep in the buffer, and those priorities will parametrize a distribution from which minibatches are sampled for the update step while overcoming bias and overfitting.

Algorithm 17 Combined Experience Replay (CER)

Require: $\mathcal{A}$: An off-policy algorithm (SAC, DDPG, TD3 etc.)
Require: $\tau \in [0, 1]$: Dampening factor (default: 0.7)
Require: $\beta \in [0, 1]$: Importance-sampling annealing exponent (default: 0.5)
Require: $\epsilon > 0$: Error offset (default: 0.1)

Initialize $\mathcal{A}$ with lower α ▷ initialize buffer, parallel agents, environments, networks
Initialize $p_1 \leftarrow 1$ and $\delta \leftarrow 0$
repeat
  for each agent $\in$ parallel agents **do** ▷ One step in each parallel environment
    $s_t \leftarrow agent.CollectObservations()$
    $a_t \in (-1, 1) \leftarrow \pi_\theta(\cdot|s_t)$
    $r_t \leftarrow$ agent.$OnActionReceived(a_t)$ ▷ Use a_t in the environment
    $s_{t+1} \leftarrow$ agent.$CollectObservations()$
    $p_t \leftarrow \max_p$ (buffer) ▷ Set p_t as maximal proprity
    if s_{t+1} is terminal **then**
      agent.$environment.Reset()$
    end if
    buffer $\leftarrow$ buffer + timestep$_t(s_t, a_t, r_t, s_{t+1}, p_t)$
  end for
  if new k timesteps were collected **then** ▷ $k \sim N$ is the update frequency
    for however many epochs **do** ▷ $\mathcal{A}$ optimization steps
      • Compute Q_ϕ^{target}
      $p_{normalized} \leftarrow p_i^\tau / \sum_j^{|\text{buffer}|} p_j^\tau$ ▷ Compute timesteps' normalized priorities
      minibatch $\leftarrow SampleMinibatchWeighted$(buffer, $p_{normalized}$)
      minibatch $\leftarrow$ minibatch + $LastKTimesteps$(buffer) ▷ CER
      $w \leftarrow (|\text{buffer}| \cdot p_{normalized})^{-\beta}$ ▷ Compute importance-sampling weights
      $w \leftarrow w / \max(w)$ ▷ Normalize weights by largest element in batch
      $\delta \leftarrow Q_\phi(s, a) - Q_\phi^{target}$ ▷ Compute timesteps batch errors
      $p \leftarrow |\delta| + \epsilon$ ▷ Update timesteps batch priorities
      • Update Q-function(s) parameters ϕ by gradient descent
      $\mu_\theta.Backward\left(w \cdot \delta \cdot \nabla_{\mu_\theta(s)} \mathcal{J}(\theta)\right)$ ▷ Compute gradients of θ
      $optim_\theta.Step(\alpha_\theta)$
      • Update target parameters
    end for
    $\beta \leftarrow Schedule(\beta)$ ▷ Linear anneal towards 1
  end if
until convergence
return π_θ

After the minibatch timesteps are used, their priorities update based on the error of the Q function denoted as δ. One direct option to control the greedy prioritization and maintain diversity so high error samples are not drawn as frequently (to avoid overfitting) is to introduce a small positive ϵ to prevent zero error of used timesteps. The bias introduced by this prioritization is solved using importance sampling weights w, annealed linearly by the β exponent, scheduled linearly towards 1 till the end of the training.

Note that to stabilize their experiments (on Atari games), the authors used a lower learning rate $\alpha = \alpha_{baseline}/4$ (their $\alpha_{baseline} = 0.00025$) to avoid potential issues associated with biased sampling and large steps that might lead to overfitting.

Given PER is a stochastic method, assigning a high priority to newer timesteps does not guarantee they will be used in the immediate next batch (especially when dealing with very large buffers), thus [ZS17] proposes the Combined Experience Replay (CER), which is simply adding the latest timestep(s) of the agent(s) to the sampled batched (weighted by the priorities).

3.4.10 Hindsight Experience Replay

Hindsight Experience Replay (HER) [AWR$^+$17] is an innovative technique used with off-policy algorithms that addresses sparse or delayed rewards environments, without creating a dense supervised setting of auxiliary tasks or additional losses. Given a final goal is hard to reach, creating vector representations g of sub-goals from the agent's previous failures proves to leverage failed attempts as learning opportunities. In practice, the goals are represented by states s that the agent experienced, sampled from the experience buffer following a sampling strategy $\mathbb{S}$.

Replay Strategy

For each experience collected, the authors consider four replay strategies $\mathbb{S}$:

- `final` − the sampled additional goals are the terminal states of the environments
- `future` − the k sampled additional goals are random states from the agent trajectory that were experienced after the current timestep
- `episode` − the k sampled additional goals are random states from the agent trajectory
- `random` − the k sampled additional goals are random states from the buffer (including the agent trajectory)

Authors prove the performance improvements of HER with experiments that rely mostly on robotic arms environments (push, slide, pick-and-place) with binary and sparse rewards. Their empirical results show that the `future` strategy is the best, with $k = 4$ or 8 (higher k degrades performance).

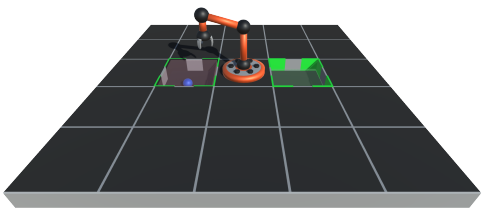


Figure 4: Robotic Arm in DeepUnity

Rewarding the subgoals

In the original paper, the authors introduce two types of sub-goal reward functions:

- `shaped` $-\ \mathcal{R}(s_t, s_{t+1}, g) = \lambda||g' - s_t||_p - ||g' - s_{t+1}||_p$, where $\lambda \in \{0, 1\}$ and $p \in \{1, 2\}$
- `binary`[1] $-\ \mathcal{R}(s_t, g) = -\left[||g - s_t||_p > \epsilon\right]$, where $\epsilon > 0$ is a threshold

Algorithm 18 Hindsight Experience Replay (HER)

Require: $\mathcal{A}$: An off-policy algorithm (SAC, DDPG, TD3 etc.)
Require: $\mathbb{S}$: Strategy for sub-goal sampling for replay (default: `future`)
Require: $k \geq 1$: Hindsight sub-goals per episode (default: 4) ▷ k = 1 if $\mathbb{S}$ = `final`
Require: $\mathcal{R}$: Sub-goal reward function

Initialize $\mathcal{A}$ ▷ Initialize buffer, parallel agents, environments, networks
repeat
  for each agent ∈ parallel agents **do**
    repeat
      $g \leftarrow SampleGoal$(buffer)
      $s_t \leftarrow agent.CollectObservations()$
      $a_t \in (-1, 1) \leftarrow \pi_\theta(\cdot|s_t||g)$ ▷ Concatenate s_t and g
      $r_t \leftarrow$ agent.$OnActionReceived(a_t)$ ▷ Use a_t in the environment
      $s_{t+1} \leftarrow$ agent.$CollectObservations()$
      agent.$trajectory \leftarrow$ agent.$trajectory$ + timestep$_t(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$
    until s_{t+1} is terminal
    buffer ← buffer + agent.$trajectory$
    for t = 0, 1, ... , |agent.$trajectory$| **do**
      $G \leftarrow SampleGoals(k, \mathbb{S}, \text{buffer})$ ▷ Consider agent.$trajectory \in$ buffer
      for each $g' \in G$ **do**
        $r'_t \leftarrow \mathcal{R}(s_t, s_{t+1}, g)$ ▷ Sub-goal reward
        buffer ← buffer + timestep$_{\text{HER}}(s_t||g',\ a_t,\ r'_t,\ s_{t+1}||g',\ g)$
      end for
    end for
    agent.$trajectory \leftarrow$ []
    agent.environment.$Reset()$
  end for
  for however many epochs **do** ▷ $\mathcal{A}$ optimization steps
    minibatch ← $SampleMinibatch$(buffer)
    - Compute Q_ϕ^{target}

    $Q_\phi^{target} \leftarrow Clip\left(Q_\phi^{target}, -\frac{1}{1-\gamma}, 0\right)$ ▷ Clip Q_ϕ^{target} in possible range

    - Update Q-function(s) parameters ϕ by gradient descent
    - Update Policy parameters θ by gradient ascent
    - Update target parameters

  end for
until convergence
return π_θ

[1] If the norm is outside the threshold $> \epsilon$, the binary reward evaluates to -1, and 0 otherwise.

Chapter 4

Reference Implementation

Both Chapter 2 and 3 are complemented by a from scratch implementation in C# programming language for Unity game engine as a standalone add-on project named *DeepUnity*, along with specific advanced modules that were not explained in the previous subsections, but ones that worth to be mentioned are going to be referenced in Section 4.3. *DeepUnity*'s objective is to offer an implementation example support for the 3^{rd} Chapter that covers Reinforcement Learning and to deliver all particularities regarded to the abstract algorithms presented previously in the second one. The project is open-sourced, greeted by the documentation and tutorial examples that can be found at `https://github.com/smtmRadu/DeepUnity`. Section 4.1 grants partial details of the entire program, depicted by all implemented deep learning tools, followed by Section 4.3 that emphasizes useful specific improvements, ending up with final results over some made up environments in Subsection 4.4.

4.1 Description of Reference Implementation

DeepUnity is an add-on framework that provides tensor computation (with GPU acceleration support) and deep neural networks, along with reinforcement learning tools that enable training for intelligent agents within Unity environments using state-of-the-art RL algorithms. Due to its inherent limitations in general performance, which hinder its competitiveness against frameworks enjoying robust community engagement and operating without constraints, its implementation has been exclusively tailored to provide modest enhancements in efficiency, ease of use, and educational value for the purposes of this paper, devoid of any competitive orientation.

Generally speaking, training agents with DRL within an Unity environment requires the use of a deep learning framework, though .NET NuGet packages (like TorchSharp or TensorFlowSharp/TensorFlow.NET) installation is cumbersome when needed for each separate project. Rather than running a separate .NET application process in background that handles all deep learning necessities and transfer information using sockets (as ML-Agents Unity add-on does), a framework that resolves all mechanics internally without any external dependencies stands as the most streamlined solution. This ensures that the project remains standalone, avoiding the requirement for intertwining two distinct applications.

4.1.1 Deep Learning Library

Tensor

Firstly, all operations in any deep learning frameworks are computed with tensors (generally called n dimensional arrays, or ND Arrays). In DeepUnity, the class *Tensor* constructs

non-static tensor that facilitates all necessary functions like random initialization, operations on axis, matrix multiplication etc. Even though multithreading offer a significant increase for parallel operations such as matrix multiplication or convolution, DeepUnity provides GPU acceleration support with the help of Compute Shader scripts, by loading up the matrices in the VRAM, dispatch the thread groups for operation and then bring back the result in RAM. *TensorGPU* class allows tensors living permanently in VRAM for even greater performance, but is currently deprecated due to difficulty of use (the tensors had to be manually deallocated).

Module

Every deep neural network, as explained in Chapter 2, contains a sequence of layers (linear and non-linear activations, batch normalization, dropout, pooling etc.), here determined by the classes that implement the *IModule* interface, which provide the *Forward()* and *Backward()* methods. Additionally, the layers have a *Predict()* method used exclusively during evaluation to distinguish between training and inference (e.g. BatchNorm or Dropout require this distinction). Modules with trainable parameters also implement the *ILearnable* interface, which includes methods for accessing the parameters' tensors.

Model

The *Model* class represents the base class for different neural network architectures, that beside the functions previously enumerated for the previous interfaces, considers serialization methods as scriptable objects (for easy management in Unity Editor).

Optimizer

DeepUnity's neural networks are trained by the use of optimizers, all of them following the same functionality, inherited from the base class *Optimizer*. The base class ensures the reference to the learnable parameters of the models (as expressed in Figure 1 , and provides functions like *ZeroGrad()* (resetting the gradients to 0), *ClipGradValue()* (clipping gradients by value), *ClipGradNorm()* (clipping gradients by norm), and *Step()* (updating parameters). The *Step()* function is purely abstract and is implemented separately by each optimizer based on its specific algorithm.

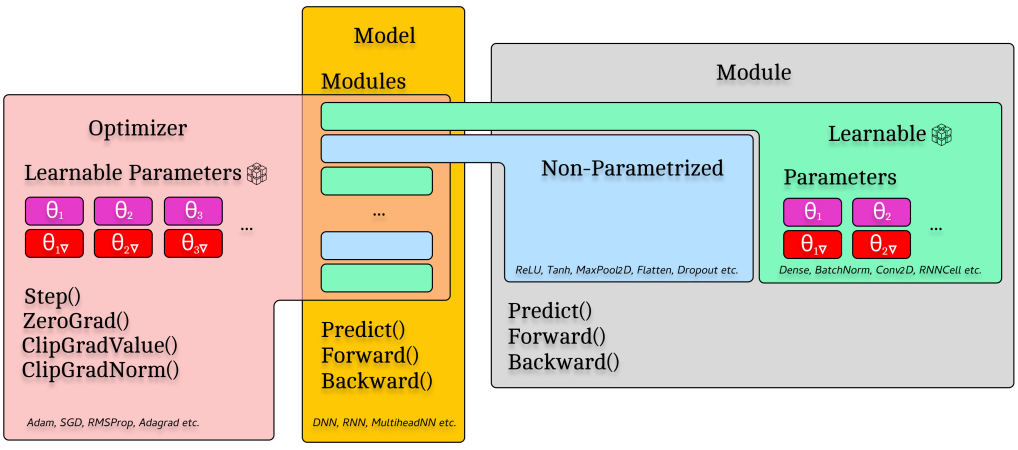


Figure 1: Architecture of the Model

Loss & Metrics

For procedures in which supervised learning is involved, the *Loss* class allows automatic differentiation of different loss functions (MSE, BCE, KLD etc.) with respect to the output of the model, that is furtherly backpropagated to compute the gradients of the parameters, instead of writing the partial derivative manually. *Metrics* class manages the computation for model's performance, offering methods to calculate the accuracy, F1 score and mean squared error.

4.1.2 Deep Reinforcement Learning Interface

Trainer

Each deep reinforcement learning algorithm has its own class that implements its specific optimization methods. Though all share in common similar jobs, defined in the base class *Trainer* which they inherit. The purpose of this base class is to handle all parallel agents learning in the scene then capturing their experiences, the auto-saving process and policy's update trigger.

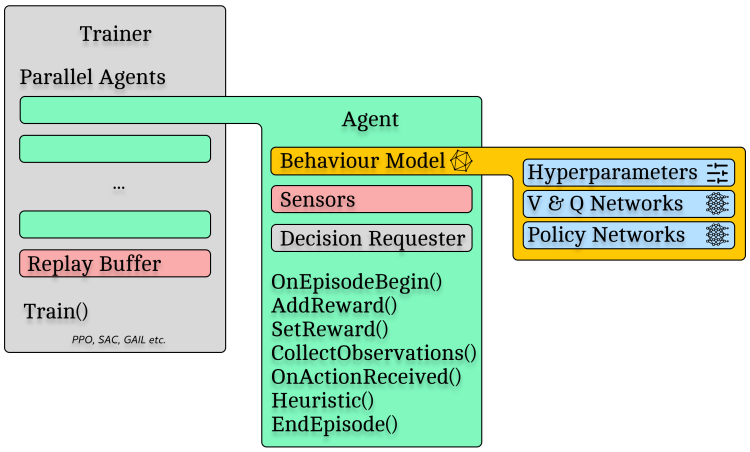


Figure 2: RL Trainer Architecture

Agent

Similar to ML-Agent's interface, any novel agent's behaviour must have it's own script that inherits from the base *Agent* class, which is of type *MonoBehaviour*, so it can be attached to the agent *GameObject*. The behaviour script is packed with the decision requester class. The *Agent* class allows overriding methods such as *CollectObservations()* for creating an observation vector or *OnActionReceived()* to use the yielded action values to simulate interaction in the environment. The logic of generating actions is handled by the *DecisionRequester* class that is automatically attached along our behaviour class, that extends the usability on different natures of simulations.

Behaviour Model

To keep the interface cleaner and modular, the *AgentBehaviour* class plays the logic of a box that contains the references to all neural networks and the training configuration asset. Likewise, it is a control point for the behaviour running characteristics, for instance the physics update frequency, observations normalization, standard deviation (fixed or

trainable) for continuous actions etc. It is created when the bake button is pressed on the agent's behaviour script in Unity's Inspector that inherits the *Agent* class, serialized as a *ScriptableObject* with the same name as the agent's behaviour script, along with the *Hyperparameters* asset (named *Config*) and neural networks assets (named specifically by their role) in the same folder (again, with the same name as the agent's behaviour) in Unity main *Assets* folder.

Sensors, Body Controllers and more

Agents might need complex ways to observe the environments' features, like proximity sensors or visual observations. To simulate these sensorial devices faster *Unity Sensors*, were created (inspired from ML-Agents framework), running in both 2D and 3D environments. They can be found both inside *DeepUnity* package, or as a separate repository[1]. One does uses multiple Configurable Joints for controlling itself, with custom joint spring, damper or force; *BodyController* class was made to speed things up, which takes care of all body parts management. There are numerous other features, like the *TrainingStatistics* class which builds up all data information (cumulative reward, loss, episode time progress etc.) resulted from training sessions, exported at the end in SVG format. Nonetheless, a small *Utils* library useful for random sampling, image processing (image resize, noising, masking etc.), advanced math operations, conversions and tensors manipulation is present within the DeepUnity's namespace.

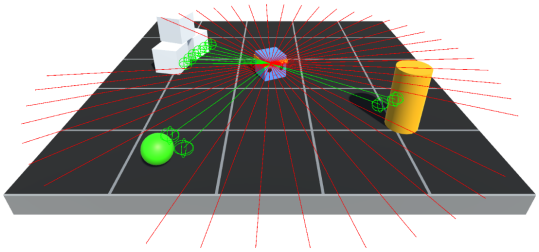


Figure 3: Ray Sensor

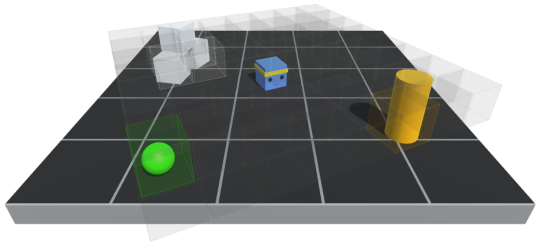


Figure 4: Grid Sensor

It's worth mentioning that DeepUnity encompasses numerous additional features that, due to certain constraints, aren't detailed here, and it offers comprehensive documentation for all classes and methods. The code is well documented taking into account various use cases, restrictions and recommendations. For additional information about the framework, please refer to the project's GitHub repository[1].

4.2 Related Work

In the preliminary stages of development, it is essential to acknowledge that DeepUnity drew significant inspiration from various frameworks, benefiting from their well-documented features, theoretical explanations and implementation paradigms, the key ones highlighted in the forthcoming subsections.

4.2.1 PyTorch and TensorFlow

PyTorch [PGM+19] and TensorFlow [AAB+15], as leading open-source deep learning frameworks, served as invaluable resources for guiding the development of DeepUnity.

[1] https://github.com/smtmRadu/UnitySensors
[1] https://github.com/smtmRadu/DeepUnity

These frameworks, predominantly implemented for Python, offer comprehensive support, albeit with no guarantee of API stability in their multi-language implementations. DeepUnity integrates syntax elements from both PyTorch and TensorFlow, although with simplifications, to enhance accessibility for new users. A notable distinction lies in DeepUnity's lack of a tensor autograd system, a feature present in all advanced deep learning libraries. While DeepUnity's layers support auto-differentiation when computing their own parameters' gradients, users must differentiate their custom Loss Function (refer to Section 2.3.1), excepting ones that are used in classic problems like regression or classification, such as L1, L2, Cross Entropy etc. For accelerated computing on a graphics card, rather than using CUDA platform from NVIDIA for parallel computing, DeepUnity uses the integrated Unity support of Compute Shaders using HLSL programming language (here, it is used in highly parallelizable operations − multiplication, convolution etc.).

4.2.2 ML-Agents

Furthermore, ML-Agents[1] is an officially endorsed package by Unity Technologies, providing both PyTorch and TensorFlow implementations for training intelligent Reinforcement Learning agents, supporting Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). ML-Agents also incorporates Multi-Agent POsthumous Credit Assignment (MA-POCA), covered in [CTB+22], that addresses the problem in classic Multi-Agent Reinforcement Learning (MARL) algorithms that doesn't consider the fact that agents can terminate their episode before their teammates and also that they will not learn from the collective success or failure of the group. DeepUnity simplifies the models management by serializing them as Scriptable Objects saved inside the assets of the project. The utility interface of ML-Agents significantly influenced the design and functionality of DeepUnity RL part. This influence has resulted in a more consolidated and user-friendly alternative, as DeepUnity conducts all requisite processes internally, eliminating the need for external interference with processes and paths, also int encompasses an alternative implementation for ML-Agent's RL utilities such as ray and grid sensors or joint controllers. The learning environments examples available in DeepUnity's repository, also presented in Subsection 4.4 were also partially inspired from ML-Agents examples[2] however, certainly, with different models, behaviour parameters, physics properties and training setups.

4.2.3 Gym and MuJoCo

Gym[3] is an open-sourced library developed by OpenAI generally created to accelerate their own RL research and is available for anyone. It consists of a growing suite of environments (from simulated robots to Atari games) written in Python, integrated with PyTorch and TensorFlow, and a site for benchmarking the performance of RL algorithms. Currently, it became a standard toolkit for researchers working on RL projects.

MuJoCo[4] (Multi-Joint dynamics with Contact) is an open-sourced physics engine developed by Emo Todorov at the University of Washington (currently maintained by Google DeepMind) designed for the purpose of simulating accurate physics simulations, enumarating collisions, rigid bodies interactions and joint systems. It can be integrated in Unity for complex environments that require precision or to edit the environments. Though, Unity provides a simpler interface for creating and simulating physics environments, thus DeepUnity manages to combine the benefits and features of all contextual

[1] https://github.com/Unity-Technologies/ml-agents

[2] https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Learning-Environment-Examples.md

[3] https://github.com/openai/gym

[4] https://mujoco.org

applications presented up to this point.

4.3 Advanced Features of Reference Implementation

DeepUnity includes various additional features dedicated for different tasks, diversity and user preference, that couldn't be presented in this paper since they are optional and predominantly manifest as improved versions of the already presented activation functions or stochastic gradient descent. Though, some notable ones that embody intriguing approaches are presented in the below.

Other nonlinear activation functions

Leaky ReLU is an extension of the ReLU activation function that addresses *dying ReLU problem*, in which neurons are becoming entirely inactive when their input it's less then 0 since ReLU is constant in 0 for that range. This issue is prevented by allowing a small positive slope $\hbar$ (default 0.01) for the negative input space, maintaining continuously their use during the training. The $\hbar$ parameter can be a learnable parameter (updated concurrently with the model's other parameters) as well, offering a dynamic adjustment of the small positive gradient during the learning process, known as the Parametric Rectified Linear Unit (PReLU), or, as firstly introduced in [XWCL15] as RReLU, randomly sampled from an uniform distribution $\mathcal{U}(\alpha, \beta)$ on training (on evaluation $h = \frac{\alpha+\beta}{2}$).

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ x \cdot \hbar, & \text{if } x < 0 \end{cases} \qquad \text{LeakyReLU}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ \hbar, & \text{if } x < 0 \end{cases}$$

Softplus is a smooth approximation of ReLU function that constraints the output to be positive. Below is presented the generalized Softplus formula that takes into account a β hyperparameter, as well with a modification that helps stabilizing the σ network in agent's continuous actions (β and ψ default values: 1, DeepUnity uses $\beta = 1.2$ and $\psi = 6$ to allow higher initial exploration strength and rapid stochasticity adaptation contrary to the exponential function). Making the non-staturated activation functions' hyperparameter a learnable parameter might improve covergence, this paper [CNDM20] emphasizes the advantages for such parameterization with several experiments and a novel function that extends Softplus and PReLU, named Soft++.

$$\text{Softplus}(x) = \frac{\psi \log\left(1 + e^{\beta x}\right)}{\beta} \qquad \text{Softplus}'(x) = \frac{\psi e^{\beta x}}{1 + e^{\beta x}}$$

Gaussian Error Linear Unit (GELU), introduced in [HG16] is an activation function in which it's nonlinearity weights inputs by their percentile rather than gating them by their sign as in ReLU (the tail of the activation is not completely flat, so it allows gradients to flow). It evaluates to $x\Phi(x)$, where $\Phi(x)$ is the Gaussian cumulative distribution function (CDF - for continuous random variables). For computation improvements, GELU can be approximated either with SiLU $= x\sigma(x)$ [EUD18] (a.k.a. Swish activation), where $\sigma(x)$ is the logistic sigmoid function, or as follows:

$$\text{GELU}\left(x\right) \approx \frac{1}{2} \cdot x \cdot \left(1 + \text{Tanh}\left(\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right)\right)$$

$$\begin{aligned}\text{GELU}'\left(x\right) \approx \frac{1}{2}\left(1 + \text{Tanh}\left(\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right)\right) \\ + \frac{1}{2} \cdot x \cdot \text{Sech}^2\left(\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right)\end{aligned}$$

Mish, proposed in [Mis19], draws inspiration from Swish and exhibits consistent and better accuracy on comparison with Swish and its variants, serving as a robust choice for neural networks. Its characteristics, including smoothness, non-monotonic behavior (decreasing and increasing as input changes), and lower bounding, all together contribute to its overall performance.

$$\text{Mish}\left(x\right) = x \cdot \text{Tanh}\left(\text{Softplus}(x)\right)$$

$$\text{Mish}'\left(x\right) = \text{Tanh}\left(\text{Softplus}(x)\right) + x \cdot \text{Sech}\left(\text{Softplus}\left(x\right)\right)$$

Other loss functions

L1 and L2 loss functions, Root Mean Squared Error, Cross Entropy, Binary Cross Entropy, Hinge Embedding (measures L1 distance between the input and the target $\in \{-1, 1\}$) and Kullback-Leibler Divergence loss functions are available in a dedicated class that differentiates the loss with respect to the output of the network.

$$\text{Hinge Embedded}\left(\hat{y}, y\right) = \max\left(0, 1 - \hat{y} * y\right)$$

$$\text{Hinge Embedded'}\left(\hat{y}, y\right) = \begin{cases} -y, & \text{if } 1 - \hat{y} \cdot y > 0 \\ 0, & \text{otherwise} \end{cases}$$

Stochastic gradient descent variants

DeepUnity provides usage of SGD, RMSProp, Adam & AdamW (+AMSGrad), Adamax, Nadam, RAdam, Lion, Adagrad, Adadelta and Adan optimizers, of which some algorithms have been taken from PyTorch documentation[1]. ADAptive Nesterov momentum [XZL+22] (Adan for short) is a worth mentioning new opimization algorithm that avoids the extra overhead of computing gradient at the look-ahead point and shows better results thanAdam [KB14] optimizer in terms of convergence speed (especially in reinforcement learning). As any other stochastic gradient descent algorithm, it's implementation is straight forward, running it does not require any deep understanding of paper details. Below the algorithm is presented without the reset condition block (that restarts the momentum strategy), even though from authors paper shows slight improvements in practice. Be aware that the default values for the beta parameters in Adan are not universally standardized across different types of architectures or optimization problems.

[1] `https://pytorch.org/docs/stable/optim.html`

Algorithm 19 Adan (without momentum restart)

Require: α: Learning Rate (default: 0.001)
Require: $\beta_1, \beta_2, \beta_3$: Momentum (default: $\beta_1 = 0.02$, $\beta_2 = 0.08$, $\beta_3 = 0.01$)
Require: $\epsilon > 0$: Value for numerical stability (default: 10^{-7})
Require: $\lambda \geq 0$: Weight decay
Require: m, v, n: Moment buffers
Require: g_{old}: Previous step's gradient cache

for t = 1 to ... **do**
  $g_t \leftarrow \nabla_\theta \mathcal{J}_t(\theta_{t-1})$ ▷ Estimate gradient **g** of the objective function $\mathcal{J}$
  if t = 1 **then** ▷ Initialize momentum buffers
    $g_{old} \leftarrow 0$
    $m_t \leftarrow g_t$
    $v_t \leftarrow 0$
    $n_t \leftarrow g_t^2$
  end if
  $v_t \leftarrow g_t - g_{old}$ **if t = 2**
  $m_t \leftarrow (1 - \beta_1)m_{t-1} + \beta_1 g_t$
  $v_t \leftarrow (1 - \beta_2)v_{t-1} + \beta_2(g_t - g_{old})$
  $n_t \leftarrow (1 - \beta_3)n_{t-1} + \beta_3 \left[g_t + (1 - \beta_2)(g_t - g_{old})\right]^2$
  $\eta_t \leftarrow \alpha/(\sqrt{n_k} + \epsilon)$
  $\theta_t \leftarrow (1 + \lambda\alpha)^{-1} \left[\theta_{t-1} - \eta_t \odot (m_t + (1 - \beta_2)v_t\right]$
  $g_{old} \leftarrow g_t$
end for
return θ_t

4.4 Evaluation of Reference Implementation

This section is allocated to showcase the capabilities of the DeepUnity framework. Various tasks were conducted to assess the modularity and use cases of the deep learning tools. These tasks include the implementation and evaluation over the MNIST dataset of architectures beyond a conventional Sequential model, enumerating GANs for digit images generation from latent space, CNNs for handwritten digit classification, VAEs for image reconstruction/denoising and ResNets.

Figure 5: Image reconstruction (2nd row) using VAE (not cherry-picked)

Since the focus of this framework is set into Deep Reinforcement Learning research, DeepUnity framework includes an extensive collection of environments[1] responsible to

[1] `https://github.com/smtmRadu/DeepUnity/tree/main/Assets/DeepUnity/Tutorials`

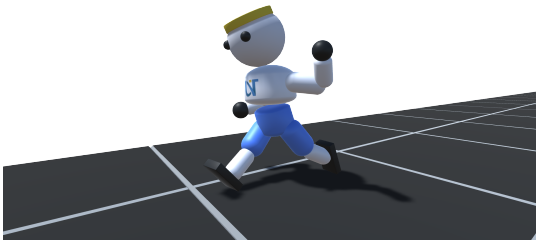
highlight what it is possible to recreate and train with the current implemented features, serving as well as templates for other environments and experiments on different algorithms. A brief overview of some of these environments (some of them inspired from ML-Agents) is presented below.

Sorter

Task: The agent must visit all tiles in ascending order as fast as possible.
Observations: 70 variables corresponding to 5 ray-casts each detecting all types of tiles and a one-hot embedding of the current tile he must search for.
Actions: 7 discrete actions, corresponding to moving along 4 directions, clockwise and counterclockwise rotation, or do nothing.
Reward Function: -0.0025 for every step, +1 if the agent visits the correct tile and -1 is it hits the wall. In the latter case, the episode also ends.

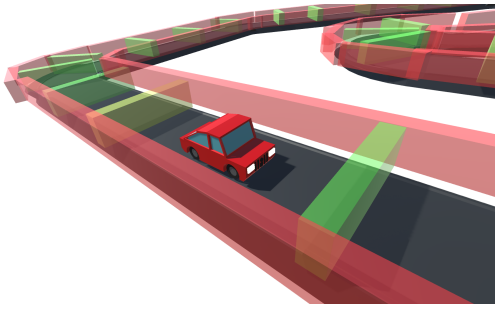
Walker

Task: The agent must walk forward as fast as possible without falling over.
Observations: 120 continuous variables, corresponding to the rotation, velocities and angular velocities of each body part.
Actions: 36 continuous actions, corresponding to joint's target rotations and strength.
Reward Function: +0.1 for every meter reached. The episode ends if the walker's upper body touches the ground.



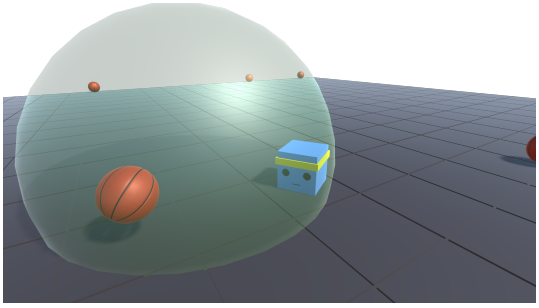
Driver

Task: The driver must complete one lap of the circuit as fast as possible without hitting the side walls.
Observations: 72 variables, corresponding to 18 ray-casts detecting the proximity to the walls or another car.
Actions: 3 continuous actions, corresponding to steering, acceleration and brake.
Reward Function: -0.001 for each step, +0.5 for each checkpoint triggered and -1 if the car touches a side wall. In the latter scenario, the episode also ends.



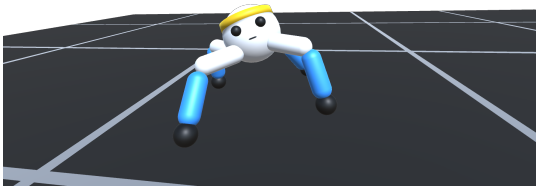
Dodger

Task: The agent must dodge all basketballs that are targeting him while keeping his position in the center of the arena.
Observations: 435 variables, corresponding to his relative position to the center of the arena and a 6×3×6 grid that detects basketballs positions in his proximity.
Actions: 7 discrete actions, corresponding to moving along 4 directions, clockwise and counterclockwise rotation, or do nothing.
Reward Function: +0.0025 for keeping within the center zone and -1 if a basketball hits him. In this case, the episode resets.



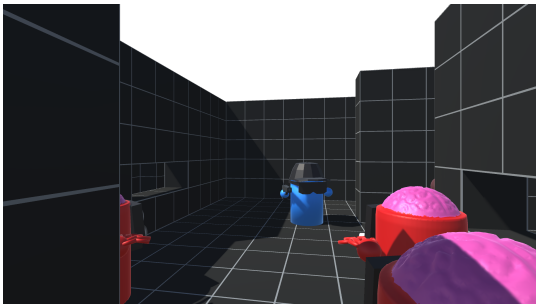
Crawler

Task: A four-legged crawler must walk forward as fast as possible without falling over.
Observations: 94 continuous variables, corresponding to the rotation, velocities and angular velocities of the head and each limb.
Actions: 20 continuous actions, corresponding to joint's target rotations and strength.
Reward Function: +0.1 for every meter reached. The episode ends if the crawler's head touches the ground.



Survivor

Task: The agent must survive for as long as possible inside an arena while being attacked by continuous waves of enemies.
Observations: 157 variables, corresponding to 50 ray-casts each detecting the enemies and the gun state, a ray-cast along its barrel direction and its current ammunition.
Actions: 9 discrete actions, corresponding to moving along 4 directions, rotation, gun fire, gun reload, and do nothing.
Reward Function: +0.0025 for each step alive, +0.25 for every enemy shot. The episode ends if one enemy reaches him.

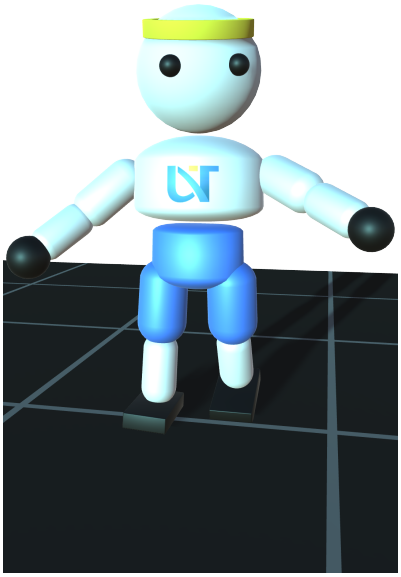


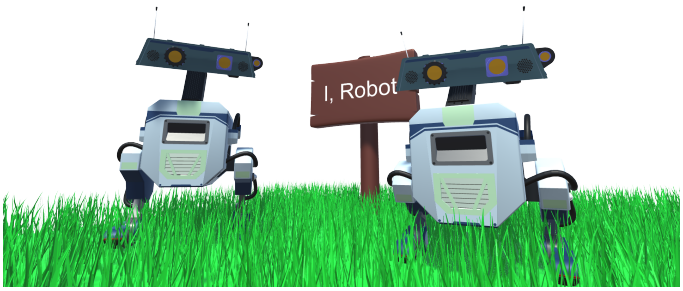
Chapter 5

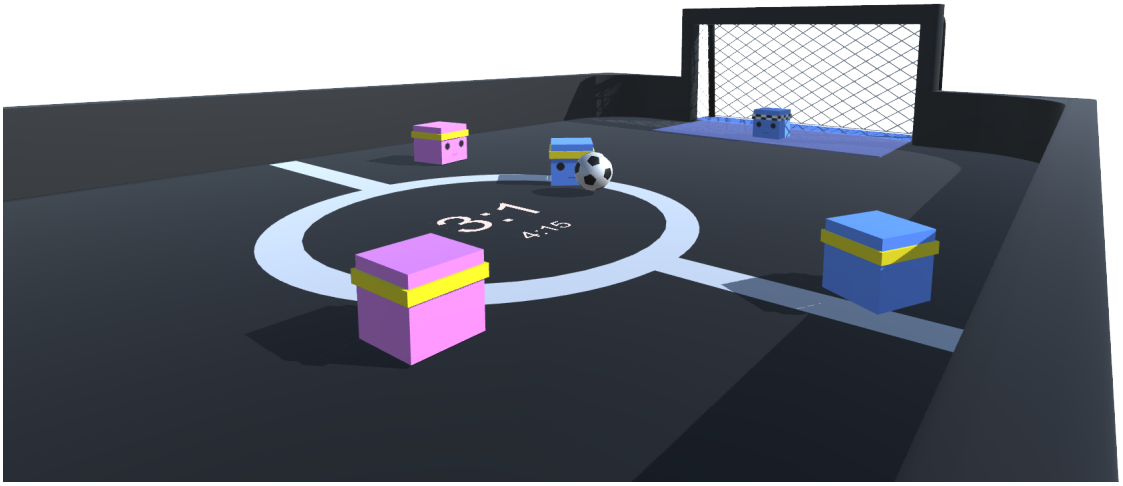
Conclusion

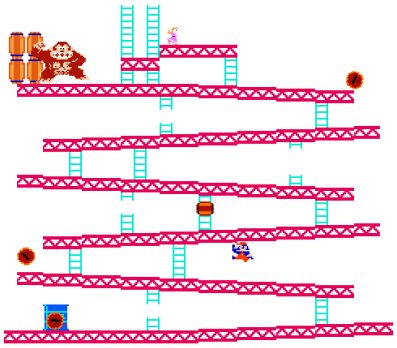
At first glance, this paper aims to provide a brief overview of the current elements and methodologies utilized in deep learning relevant to DRL research. Following this introduction, the paper delves deeper into the exploration of two prominent deep reinforcement learning algorithms belonging to the Policy Gradient Methods, along with varied code-level optimizations and good practices, covered by a ground-up optimized implementation. Due to efficiency issues and optimization limitations, handling visual observations or time sequences with convolution, recurrent or self-attention mechanisms within RL remain mostly unexplored at the time this paper was written. As a future endeavor through newer versions of Unity, the framework might benefit from Unity's Data-Oriented Technology Stack (DOTS) for extensive physics simulations, alongside TorchSharp integration allowing to explore more complex environments.

Conclusively, considering the current advancements and achievements, it can be stated that Reinforcement Learning is an exploratory process of both the agent and the trainer. The current algorithms remain susceptible to the hyperparameters tuning, and a universal solutions applicable to all environment scenarios has yet to be found. While "Reward is enough" suggests adequacy in certain contexts, most of the time computational requirements and demands do not scale proportionally when employing Trial and Error approaches. Therefore, alongside post-training models using Reinforcement Learning with Human Feedback (RLHF), robotics still heavily relies on heuristic methods to enhance RL implementations. The AGI research entails modular enhancements and algorithms, each possessing unique strengths and weaknesses, therefore, a robust path towards developing autonomous intelligent systems involves integrating the benefits of effective architectures within modern methodologies like Joint Embeddings [LeC22] or other that might have to come in the literature.









Presentation of some other environments, not elaborated upon here, to showcase the framework's capabilities in training various agents on a single machine.

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015.

[ARS+20] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2020.

[AWR+17] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[BW21] Daniel Bick and MA Wiering. *Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization.* PhD thesis, Master's thesis, 2021.[Online]. Available: https://fse. studenttheses. ub . . . , 2021.

[CNDM20] Andrei Ciuparu, Adriana Nagy-Dăbâcan, and Raul C Mureşan. Soft++, a multi-parametric non-saturating non-linearity that improves convergence in deep neural architectures. *Neurocomputing*, 384:376–388, 2020.

[CTB+22] Andrew Cohen, Ervin Teng, Vincent-Pierre Berges, Ruo-Ping Dong, Hunter Henry, Marwan Mattar, Alexander Zook, and Sujoy Ganguly. On the use and misuse of absorbing states in multi-agent reinforcement learning. *arXiv:2111.05992v2*, 2022.

[CVMBB14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[DHOM21] Rousslan Fernand Julien Dossa, Shengyi Huang, Santiago Ontañón, and Takashi Matsubara. An empirical investigation of early stopping optimizations in proximal policy optimization. *IEEE Access*, 9:117981–117992, 2021.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[Duc07] John Duchi. Derivations for linear algebra and optimization. *Berkeley, California*, 3(1):2325–5870, 2007.

[DV18] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, 2018.

[Elm90] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[EUD18] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks*, 107:3–11, 2018.

[FHM18] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[GD23] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[GNK12] Hinton Geoffrey, Srivastava Nitish, and Swersky Kevin. Overview of mini-batch gradient descent. 2012.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[HG16] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[LeC22] Yann LeCun. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review*, 62(1), 2022.

[LH17] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[Mis19] Diganta Misra. Mish: A self regularized non-monotonic activation function. *arXiv preprint arXiv:1908.08681*, 2019.

[MKKY18] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.

[Nes83] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o(1/k**2). *Doklady Akademii Nauk SSSR*, 269(3):543, 1983.

[NKB+21] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, 2021.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[RKK19] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

[SM02] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[SML+15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[UO30] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[WH18] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[WM97] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[XWCL15] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[XZL+22] Xingyu Xie, Pan Zhou, Huan Li, Zhouchen Lin, and Shuicheng Yan. Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. *arXiv preprint arXiv:2208.06677*, 2022.

[ZS17] Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.

[ZS19] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

Appendix

The appendix includes off-policy algorithms with deterministic policies, which are part of Policy Gradient Methods. These algorithms are no more complex than their stochastic policy alternatives and the paper is expanded here for completeness.

A Deep Deterministic Policy Gradient (DDPG)

DDPG is the Actor-Critic version of Deep Q-learning introduced in [SLH+14] designed for continuous actions spaces, that relies on a deterministic policy μ_θ optimized with gradient ascent using a Q-value estimator Q_ϕ. Because μ_θ is deterministic, Gaussian noise with zero-mean (denoted below as ξ) is applied over the actions at training time.

Algorithm 20 Deep Deterministic Policy Gradient (DDPG)

Require: N: Number of parallel agents
Require: μ_θ and $optim_\theta$: Policy network and it's optimizer
Require: Q_ϕ and $optim_\phi$: Q network and it's optimizer
Require: $\mu_{\theta_{targ}}$ and $Q_{\phi_{targ}}$ networks
Require: α_θ & $\alpha_\phi \in [10^{-4}, 10^{-3}]$, $\gamma \in [0.99, 0.995]$, $\tau \sim 5 \times 10^{-3}$, $\sigma \sim 0.1$: Learning rate, discount factor, interpolation factor and standard deviation for Gaussian noise

Initialize μ_θ and Q_ϕ networks and their optimizers
Initialize $\mu_{\theta_{targ}}$, $Q_{\phi_{targ}}$ networks and set $\theta_{targ} \leftarrow \theta$, $\phi_{targ} \leftarrow \phi$
Initialize N parallel agents in N different environments
Initialize buffer $\leftarrow []$
repeat
    // Collect trajectories
    for each agent $\in$ parallel agents **do**
        $s_t \leftarrow agent.CollectObservations()$
        $a_t \leftarrow Clip(\mu_\theta(s_t) + \xi, -1, 1)\,, \quad \xi \sim \mathcal{N}(0, \sigma)$ ▷ $\mu_\theta(s_t) \in$ (-1, 1)
        $r_t \leftarrow$ agent.$OnActionReceived(a_t)$ ▷ Use a_t in the environment
        $s_{t+1} \leftarrow$ agent.$CollectObservations()$
        if s_{t+1} is terminal **then**
            agent.$environment.Reset()$
        end if
        buffer $\leftarrow$ buffer + timestep$_t$(s_t, a_t, r_t, s_{t+1})
    end for
    if new k timesteps were collected **then** ▷ $k \sim N$ is the update frequency
        for however many epochs **do**
            minibatch $\leftarrow SampleMinibatch$(buffer)
            // Update Q function by gradient descent
            $Q_{\phi,t}^{target} \leftarrow \begin{cases} r_t \text{ if } s_{t+1} \text{ is terminal, else} \\ r_t + \gamma Q_{\phi_{\text{targ}}}\left(s_{t+1}, \mu_{\theta_{\text{targ}}}(s_{t+1})\right) \end{cases}$
            $\frac{\partial \left(Q_\phi(s_t,a_t) - Q_{\phi,t}^{target}\right)^2}{\partial Q_\phi(s_t,a_t)} \leftarrow 2 \cdot \left(Q_\phi(s_t,a_t) - Q_{\phi,t}^{target}\right)$
            $optim_\phi.ZeroGrad()$
            $Q_\phi.Backward\left(\frac{\partial \left(Q_\phi(s_t,a_t) - Q_{\phi,t}^{target}\right)^2}{\partial Q_\phi(s_t,a_t)}\right)$
            $optim_\phi.Step(\alpha_\phi)$

// Update Policy function by gradient ascent
$L_t(\theta) \leftarrow -Q_\phi(s_t, \mu_\theta(s_t) \in (-1, 1))$ ▷ Forward process
$optim_\theta.ZeroGrad()$
$\frac{\partial L_t(\theta)}{\partial \mu_\theta(s_t)} \leftarrow Q_\phi.Backward(-1)$
$\mu_\theta.Backward\left(\frac{\partial L_t(\theta)}{\partial \mu_\theta(s_t)}\right)$
$optim_\theta.Step(\alpha_\theta)$
// Update target parameters
$\theta_{targ} \leftarrow (1-\tau)\theta_{targ} + \tau\theta$
$\phi_{targ} \leftarrow (1-\tau)\phi_{targ} + \tau\phi$
end for
end if
until convergence
return π_θ

B Twin Delayed Deep Deterministic Policy Gradient (TD3)

Twin DDPG [FHM18] overcomes the Q-values overestimate by using two critics over DDPG (see why in Subsection 3.3.2) and introduces a less frequent update of the policy (known as a policy delay, below denoted by $\delta \sim 2$). A critic update occurs once every step (or once every N steps for N parallel agents), though updating the policy once every two or more steps contributes to a more stable policy, preventing abrupt changes. The authors found that noise drawn from Ornstein-Uhlenbeck process [UO30] (used in the original DDPG implementation) offers no performance benefits, so they apply uncorrelated Gaussian noise $\mathcal{N}(0, 0.1)$ to each action.

Algorithm 21 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Require: N: Number of parallel agents
Require: π_θ and $optim_\theta$: Policy network and it's optimizer
Require: Q_{ϕ_1}, Q_{ϕ_2} and $optim_\phi$: Q_1 and Q_2 network and their optimizer
Require: $\mu_{\theta_{targ}}$, $Q_{\phi_{targ,1}}$ and $Q_{\phi_{targ,2}}$ networks
Require: α_θ & $\alpha_\phi \in [10^{-4}, 10^{-3}]$, $\gamma \in [0.99, 0.995]$, $\tau \sim 5 \times 10^{-3}$, $\delta \sim 2$, $\sigma \sim 0.1$, $c \sim 0.5$: Learning rate, discount factor, interpolation factor, policy delay, standard deviation for Gaussian noise and noise clipping value
Initialize π_θ, Q_{ϕ_1}, Q_{ϕ_2} networks and their optimizers
Initialize $\mu_{\theta_{targ}}$, $Q_{\phi_{targ,1}}$, $Q_{\phi_{targ,2}}$ nets and set $\theta_{targ} \leftarrow \theta$, $\phi_{targ,1} \leftarrow \phi_1$, $\phi_{targ,2} \leftarrow \phi_2$
Initialize N parallel agents in N different environments
Initialize buffer $\leftarrow []$
repeat
    // Collect trajectories
    for each agent $\in$ parallel agents **do**
        $s_t \leftarrow agent.CollectObservations()$
        $a_t \leftarrow Clip(\mu_\theta(s_t) + \xi, -1, 1), \quad \xi \sim \mathcal{N}(0, \sigma)$ ▷ $\mu_\theta(s_t) \in$ (-1, 1)
        $r_t \leftarrow$ agent.$OnActionReceived(a_t)$ ▷ Use a_t in the environment
        $s_{t+1} \leftarrow$ agent.$CollectObservations()$
        if s_{t+1} is terminal **then**
            agent.$environment.Reset()$
        end if
        buffer $\leftarrow$ buffer + timestep$_t(s_t, a_t, r_t, s_{t+1})$
    end for

if new k timesteps were collected **then** ▷ $k \sim N$ is the update frequency
  for however many epochs **do**
    minibatch $\leftarrow$ $SampleMinibatch$(buffer)
    // Compute target actions
    $a_{t+1}(s_{t+1}) \leftarrow Clip\left(\mu_{\theta_{targ}}(s_{t+1}\right) + \xi, -1, 1), \quad \xi \sim Clip\left(\mathcal{N}(0, \sigma), -c, c\right)$
    // Update Q functions by gradient descent
    $Q_{\phi,t}^{target} \leftarrow \begin{cases} r_t \text{ if } s_{t+1} \text{ is terminal, else} \\ r_t + \gamma \min\limits_{i=1,2} Q_{\phi_{\text{targ},i}}\left(s_{t+1}, a_{t+1}(s_{t+1})\right) \end{cases}$
    $\frac{\partial\left(Q_{\phi_i}(s_t,a_t) - Q_{\phi,t}^{target}\right)^2}{\partial Q_{\phi_i}(s_t,a_t)} \leftarrow 2 \cdot \left(Q_{\phi_i}(s_t, a_t) - Q_{\phi,t}^{target}\right), \quad \text{for i = 1, 2}$
    $optim_\phi.ZeroGrad()$
    $Q_{\phi_i}.Backward\left(\frac{\partial\left(Q_{\phi_i}(s_t,a_t) - Q_{\phi,t}^{target}\right)^2}{\partial Q_{\phi_i}(s_t,a_t)}\right), \quad \text{for i = 1, 2}$
    $optim_\phi.Step(\alpha_\phi)$
    // Update Policy function with delay by gradient ascent
    if $optim_\phi.step\_count$ mod $\delta = 0$ **then**
      $L_t(\theta) \leftarrow -Q_\phi\left(s_t, \mu_\theta(s_t) \in (-1, 1)\right)$ ▷ Forward process
      $optim_\theta.ZeroGrad()$
      $\frac{\partial L_t(\theta)}{\partial \mu_\theta(s_t)} \leftarrow Q_\phi.Backward(-1)$
      $\mu_\theta.Backward\left(\frac{\partial L_t(\theta)}{\partial \mu_\theta(s_t)}\right)$
      $optim_\theta.Step(\alpha_\theta)$
    end if
    // Update target parameters
    $\theta_{targ} \leftarrow (1 - \tau)\theta_{targ} + \tau\theta$
    $\phi_{targ,i} \leftarrow (1 - \tau)\phi_{targ,i} + \tau\phi_i, \quad \text{for i = 1, 2}$
  end for
end if
until convergence
return π_θ

C Effective architectures

Based on the agent's needs and observations, choosing the right network architecture impacts substantially the success of the agent and the training efficiency. Whether the agents receive a visual observation (a 2D multi-channeled image) or sequential inputs that can simulate a short memory, enhancing the models with convolutional or recurrent layers can prove intelligent behaviour. Difficult tasks require larger policies, Q and Value models, but the last two stated can be enhanced with normalization layers, higher learning rates and lower epochs number due to their stable regressive task. When dealing with actions sampled from the normal distribution, the σ network (or head) can be reduced to a lower dimension of layers and neurons comparing with the rest, on the grounds that adapting the entropy doesn't require accuracy, or simply just use a fixed standard deviation.

Experiments show that Softmax and Exponential activation function, used for generating probabilities for a Categorical Distribution and positive values as standard deviations, are sensitive to large values, thus using normalization layers, Glorot Intialization, or a saturated hidden activation function like Tanh prevents abnormally large values or distributions, therefore preventing NaN values occurrence in floating point arithmetic.