robot doggo can learn how to walk???

An introductory survey of reinforcement learning and its applications to robotic systems.

Sky Hong
Choate Rosemary Hall

May 26, 2025

Abstract

the best abstract youve ever read (real). Everyone knows Mario is cool as [EXPLETIVE DELETED]. But knows what he's thinking? Who knows why he crushes turtles? And why do we think about him as fondly as we think of the mythical (nonexistent?) Dr Pepper? Perchance I believe it was Kant who said "Experience without theory is blind, but theory without experience is mere intellectual play." Mario exhibits experience by crushing turts all day, but he exhibits theory by stating "Let's-a go!" Keep it up, baby! When Mario leaves his place of safety to stomp a turty, he knows that he may Die. And yet, for a man who can purchase lives with money, a life becomes a mere store of value. A tax that can be paid for, much as a rich man feels any law with a fine is a price. We think of Mario as a hero, but he is simply a one percenter of a more privileged variety. The lifekind. Perchance.

Contents

Chapter 1

Introduction

1.1 Preface

This document is a review article for my spring term of the Science Research Program (SRP) at Choate Rosemary Hall. The goal of this article is to provide a overview of machine learning (ML), specifically the framework of reinforcement learning (RL), and its applications to various robotic systems. The first section is intended to provide an understanding of artifical neural networks and reinforcement learning assuming no prior knowledge of the subject except for a grasp of multivariable calculus and linear algbera. Then, the second section will discuss various groups and their work in applying RL to control robotic systems.

The TEX and Python source code for this article is available on GitHub at `https://github.com/skysomorphic/review-article`.

1.1.1 Acknowledgements

1.2 Mathematical Notation

This section will introduce the standard for the mathematical notation used in this document, as well as defining any concepts that are not ubiquitous and may be unfamiliar to the reader. As new topics are introduced, new notation may be defined in their respective sections, as this section is for purely mathematical notation.

These standards which may not necessarily reflect the notation used in the original sources, or in the literature in general, rather they are intended for consistency. Even so, there are not many significant deviations that the author uses from the literature.

1.2.1 Linear Algebra

Scalars, Vectors, Matrices, and Tensors

Vectors are denoted with boldface lowercase symbols, such as $\mathbf{v}$. Matrices and other higher-order tensors are denoted with boldface uppercase symbols, such as $\mathbf{W}$.

Simple scalar values are denoted with regular lowercase symbols, such as b, and are never bolded. The scalar components of a non-scalar tensor are denoted with a subscript affixed to the corresponding unbolded lowercase symbol for the tensor, such as w_{ij} for the i, j component of the matrix $\mathbf{W}$.

Operations

The standard notation for basic vector and matrix operations are used throughout this document.

The **Hadamard product**, or element-wise product, $\odot$ is defined as

$$(\mathbf{A} \odot \mathbf{B})_{ij} = A_{ij} B_{ij}. \tag{1.1}$$

For example,

$$\begin{bmatrix} 1 & 2 \\ -3 & 4 \end{bmatrix} \odot \begin{bmatrix} 5 & -6 \\ -7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & -12 \\ 21 & 32 \end{bmatrix}. \tag{1.2}$$

The **Hadamard division**, or element-wise division, $\oslash$ is defined analagously as

$$(\mathbf{A} \oslash \mathbf{B})_{ij} = \frac{A_{ij}}{B_{ij}}. \tag{1.3}$$

For example,

$$\begin{bmatrix} 12 & 6 \\ -4 & -8 \end{bmatrix} \oslash \begin{bmatrix} 3 & -2 \\ -1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & -3 \\ 4 & -2 \end{bmatrix}. \tag{1.4}$$

The **Kronecker product**, often called the "tensor product" in machine learning, though not exactly the same as the formal linear algebra definition, is denoted with $\otimes$ and is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & A_{12}\mathbf{B} & \cdots & A_{1n}\mathbf{B} \\ A_{21}\mathbf{B} & A_{22}\mathbf{B} & \cdots & A_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1}\mathbf{B} & A_{m2}\mathbf{B} & \cdots & A_{mn}\mathbf{B} \end{bmatrix}. \tag{1.5}$$

1.2.2 Calculus

The gradient of a multivariable scalar function F is denoted with ∇F, and is defined as the vector containing all the first order partial derivatives of F.

$$\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \frac{\partial F}{\partial x_2} \\ \vdots \end{bmatrix} \tag{1.6}$$

The Hessian matrix of a multivariable scalar function F is denoted with $\mathbf{H}(F)$, and is defined as the matrix containing all the second order partial derivatives of F.

$$\mathbf{H}(F) = \nabla \otimes \nabla F = \begin{bmatrix} \frac{\partial^2 F}{\partial x_1^2} & \frac{\partial^2 F}{\partial x_1 \partial x_2} & \cdots \\ \frac{\partial^2 F}{\partial x_2 \partial x_1} & \frac{\partial^2 F}{\partial x_2^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}. \tag{1.7}$$

1.2.3 Probability

The probability of an event A is denoted with $\mathbb{P}(A)$. The conditional probability of an event A given an event B is denoted with $\mathbb{P}(A|B)$. The joint probability of two events A and B is denoted with $\mathbb{P}(A, B)$, or $\mathbb{P}(A \cap B)$.

The expected value of a random variable or function X is denoted with $\mathbb{E}[X]$.

These two operators are related by the following identity.

$$\mathbb{E}[X] = \sum_i x_i \mathbb{P}(X = x_i) = \int x \mathbb{P}(X = x)\ dx \tag{1.8}$$

1.2.4 Miscellaneous

The argument or set of arguments a at the maximum or minimum of a function f may be defined as

$$\arg\max_a f(a), \tag{1.9}$$

or

$$\arg\min_a f(a), \tag{1.10}$$

respectively.

Whenever there is a need to refer to some set of variable that are "optimal," or "best," typically meaning that they maximize or minimize some function, then a superscript $*$ is used to denote this.

Chapter 2

Neural Networks

2.1 Introduction to Machine Learning

The field of machine learning is fundamentally about finding functions that model data. For example, a speech recognition model takes audio as an input and outputs a text, and AlphaZero takes a chessboard state as an input and outputs a move. These functions may be vastly complex, with far too many parameters and relations for any human to reasonably articulate and program, even if the task comes naturally to our evolved biology. Machine learning offers methods for computers to achieve these tasks, without the need for a human to give explicit instructions on *how* it should be accomplished.

This section will give a preliminary peek into machine learning, while specific details will be delved into more deeply in following sections. The general framework of creating a machine learning model is as follows:

1. Identification of the properties of a model. What should this model be able to do? What are its inputs and outputs? What kind of model architecture is best suited to its problem?
2. Defining a loss function, some metric to measure how well a model performs.
3. Optimizing the parameters of a model to minimize the loss function and maximize performance.

2.1.1 Parameters

As an example, consider a simple model consisting of a linear relationship between a data set and the output of the model.

$$f(\mathbf{x}) = b + \sum_i w_i x_i \tag{2.1}$$

Parameters that directly multiply values, w_i, are called **weights**, and parameters that offset

values, b, are called **biases**. The inputs from the data set, $\mathbf{x}$, are called **features** This can be generalized to vector outputs with matrices, and in more complex models, the notation may be extended to tensors.

$$f = \mathbf{b} + \mathbf{W}\mathbf{x} \tag{2.2}$$

The parameters of a model are usually collectively referred to as a vector $\boldsymbol{\theta}$, with components of the individual weights and biases of the model. For complex models, $\boldsymbol{\theta}$ may be millions, billions, or even trillions of parameters long, populated by numerous parameter tensors.

$$\boldsymbol{\theta} = \begin{bmatrix} W_{11} \\ W_{12} \\ \vdots \\ b_1 \\ b_2 \\ \vdots \end{bmatrix} \tag{2.3}$$

For virtually all non-trivial problems, linear relationships are far too reductive to completely capture the complexity of a problem. Such inherent limitations of a model due to its architecture known as **model bias**. The ubiquitous solution to this problem is the neural network, which we will introduce in Section 2.2.

2.1.2 Loss Functions

A **loss function** (also sometimes called a **cost function**), typically denoted $L(\boldsymbol{\theta})$ or just L, is the measure of how "bad" a set of parameters $\boldsymbol{\theta}$ for a model is. A common definition is the deviation between a model's prediction and an actual result. For example, when building a speech recognition model, the loss may be defined as the error rate between its output transcription and the correct transcription. The features of the training data that are provided to the model, in this case, the correct transcriptions, are identified with **labels** that tell the model what it should train towards.

Losses over all labels in the training data are aggregated into an value for the overall loss function, the most common method of which is the **mean square error (MSE)**:

$$L = \frac{1}{N}\sum(y - \hat{y})^2. \tag{2.4}$$

L is associated with an **error surface** that can be understood as the plot of L in a $|\boldsymbol{\theta}|$-dimensional parameter space, though it is usually far too complex to be directly visualized.

2.1.3 Optimization

Since L is continuous, there must exist some set of parameters $\boldsymbol{\theta}^*$ that minimize L and model the training data best.

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L \tag{2.5}$$

The process of improving the model by finding values for $\boldsymbol{\theta}$ that lower the loss function is what it really means for a model to "learn" or "train".

Gradient descent

Since brute-forcing the error surface of L is infeasible with a large number of parameters, the standard algorithmic approach to find minima of L is **gradient descent**. In gradient descent, L is iteratively lowered by stepping $\boldsymbol{\theta}$ against the gradient of L.

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla L(\boldsymbol{\theta}^{(t)}) \tag{2.6}$$

The **learning rate**, η, determines the scaling for the size of each step.

Algorithm 1 Naive gradient descent with leraning rate η over N iterations.

1: Initialize parameters $\boldsymbol{\theta}$
2: **for** $i = 1, \ldots, N$ **do**
3: Compute gradient of loss $\nabla L(\boldsymbol{\theta})$ over the dataset
4: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla L(\boldsymbol{\theta})$
5: **end for**
6: **return** $\boldsymbol{\theta}$

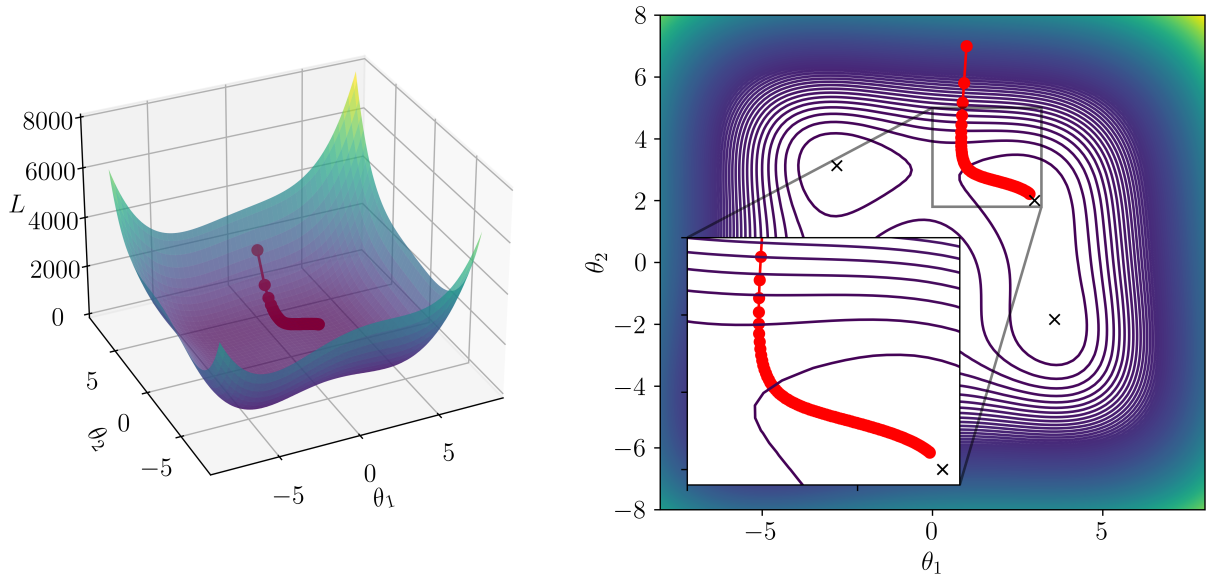


Figure 2.1: Visualization of naive gradient descent on the Himmelblau function [1] by a 3D plot (left), and a contour plot (right). The red line shows 100 iterations of gradient descent, starting from intial parameters $\boldsymbol{\theta}^{(0)} = [1, 7]$, with $\eta = 0.001$. Local minima are marked with $\times$. (Original)

In practice, some conditions may be defined to determine when to cease training, such as setting an upper limit on the number of iterations, setting a time limit, or setting a threshold for a satisfactory value of L. These values that modify aspects of a model's learning are called

hyperparameters. Hyperparameter optimization is itself a complex topic that is beyond the scope of this section.

When $\boldsymbol{\theta}$ reaches a critical point ($\nabla L = 0$), the updates to $\boldsymbol{\theta}$ vanish, and the algorithm has reached a stable ending point. In an ideal case, this would be the global minima of L, the best that the model could possibly reach given the training data. In reality, reaching the global minima is highly improbable, $\boldsymbol{\theta}$ is much more likely to be stuck at a local minima, or a saddle point. We will discuss methods to overcome these challenges in Section 2.6.

2.2 Neural Networks (NNs)

Artificial neural networks (ANNs) often shortened to **neural networks (NNs)**, are one of the largest classes of models used in machine learning. NNs are composed of **nodes** that are arranged in layers. The first layer is called the **input layer**, and the last layer is called the **output layer**. All layers in between are called **hidden layers**. Neural networks are usually fully connected, meaning that each node in a layer is considered by every node in the next layer. Such fully-connected networks are often called **multi-layer perceptrons** (MLPs).

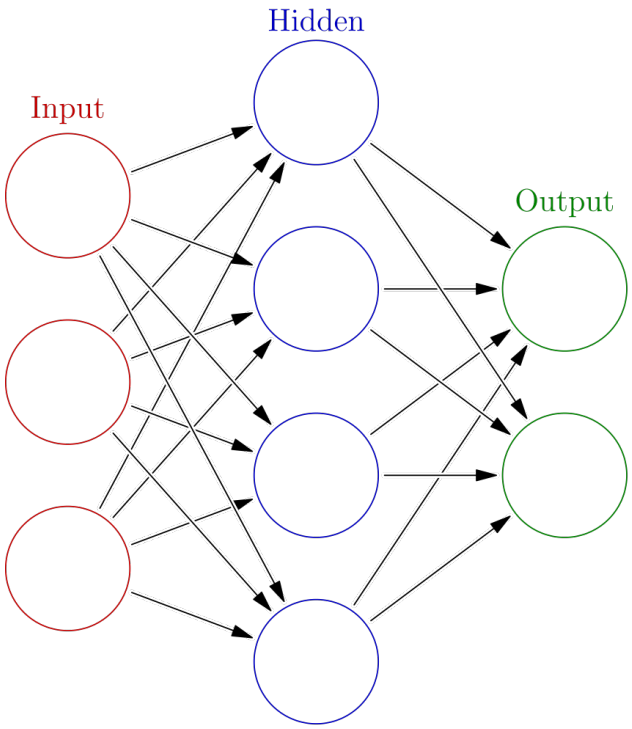


Figure 2.2: A schematic of a neural network with a input layer with three nodes, one hidden layer with four nodes, and an output layer with two nodes [2].

Each node takes a biased weighted sum of the values of the previous layer, applies an acti-

vation function ϕ, and outputs the value to the next layer.

$$a_j^{(l)} = \phi\left(b_j^{(l)} + \sum_i w_{ij}^{(l)} a_i^{(l-1)}\right) \tag{2.7}$$

Or, using matrix notation, where we define ϕ to act component-wise on a vector,

$$\mathbf{a}^{(l)} = \phi\left(\mathbf{b}^{(l)} + \mathbf{W}^{(l)}\mathbf{a}^{(l-1)}\right). \tag{2.8}$$

ϕ is usually the same for every layer of a netowrk, but it may not always be, in which case it may be specified with a subscript, $\phi^{(l)}$. The notable case where this arises is the softmax function for the final layer of a classification network. See Section 2.8 for more details.

In this article, we will use the notation $a_i^{(l)}$ to refer to the output of node i in layer l, and $\mathbf{a}^{(l)}$ to refer to the vector of outputs of all nodes in layer l. The weights and biases applied onto nodes from layer $l-1$ to be inputted into the activation function of layer l will be denoted as $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$, respectively. In general, supercripts will denote a layer in the network, and subscripts will denote a specific component of a vector or matrix, such as a specific node or weight of a layer, as described in Section 1.2. Note that this superscript notation is different from the one we use with $\boldsymbol{\theta}^{(t)}$, which is used to denote the iteration of the gradient descent algorithm.

2.2.1 Activation Functions

For virtually all non-trivial problems, linear relationships are far too reductive to completely capture the complexity of the task at hand. Therefore, a variety of non-linear **activation functions** are applied at each node to eliminate this kind of model bias to allow the model to learn more non-linear relationships.

Sigmoid functions are ubiquitous in this role, and they are also useful since they normalize the output of a node to be between 0 and 1. This is useful for many applications, such as when the output of a node is interpreted as a probability. Sigmoid functions are any function that have an S-shaped curve with horizontal asymptotes as $x \to \pm\infty$. The most common sigmoid function is the logistic function, which is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.9}$$

In literature, one might also other variants of the sigmoid function, such as the hyperbolic tangent,

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.10}$$

For the purposes of this article, the differences between these sigmoid functions are subtle at most, and so we will just use σ to denote a generic sigmoid function.

Another ubiquitous activation function is the **rectified linear unit (ReLU)**.

$$\text{ReLU}(x) = \max(0, x) \tag{2.11}$$

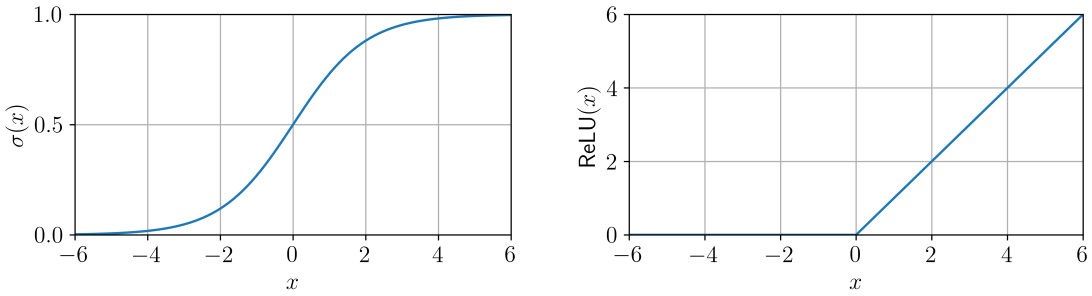


Figure 2.3: Plot of $\sigma(x)$ (left) and ReLU(x) (right). (Original)

ReLUs have the advantage of being computationally easier to calculate, though they are not differentiable at 0 which can introduce complexity during optimization. ReLUs are also unbounded in range and are not constrained by asymptotes. This may be a disadvantage, but not necessarily so. In general, the choice of activation function may be tuned to a specific problem, and there is no one-size-fits-all solution.

Some other common activation functions are listed in the table below.

Table 2.1: Common activation functions [3, 4, 5].

Name	**Equation**	**Range**	**Smoothness**
Logistic	$\sigma(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$	C^∞
Hyperbolic tangent	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$(-1, 1)$	C^∞
Heaviside	$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$	$\{0, 1\}$	C^{-1}
ReLU	$\text{ReLU}(x) = \max(0, x)$	$[0, +\infty)$	C^0
Softplus	$\text{softplus}(x) = \ln(1 + e^x)$	$(0, +\infty)$	C^∞
Softsign	$\text{softsign}(x) = \frac{x}{1+\lvert x\rvert}$	$(-\infty, +\infty)$	C^∞

A thorough catalog of all commonly used activation functions in the literature and comparisions of their properties, advantages, and disadvantages is beyond the scope of this article. The author recommends the following articles: [3, 5].

Universal Approximation Theorem

Any continuous relationship between two variables can be approximated by a *sufficiently large* linear combination of sigmoid functions [6].

$$f \approx b + \mathbf{c} \cdot \sigma(\mathbf{b} + \mathbf{W}\mathbf{x}). \tag{2.12}$$

Equation 2.12 is equivalent to a neural network with one hidden layer and scalar output (omitting the final application of an activation function).

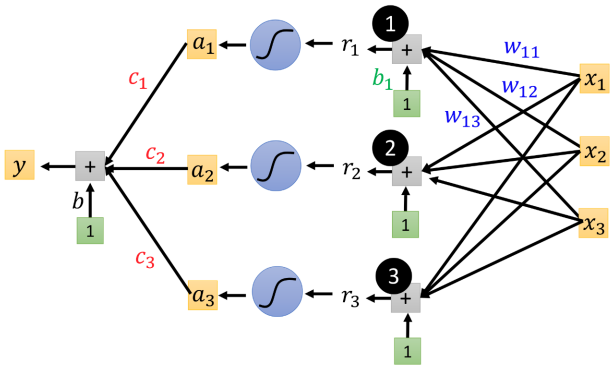


Figure 2.4: Reprinted from [7].

This result can be generalized to any non-polynomial activation function, including ReLUs, in the **universal approximation theorems**, implying that neural networks that use such activation functions may theoretically learn any relationship given *sufficiently large* amounts nodes and layers [8, 9, 10]. This property of neural networks become especially useful in other areas of machine learning where aribtrary function approximators are needed. Though, it is important to note that this result is only theoretical, and in practice, computational and data constraints usually frustrate this ideal. An informal proof of this theorem is given in [11].

2.2.2 Multiple Layers

The addition of layers to a model increases its complexity, the number of which is referred to as its **depth**. The number of nodes in a layer is referred to as the layer's **width**. The width and depth of a model are hyperparameters, and there is a balance to be made between model bias, computational demands, and overfitting, a phenomenon we will discuss in Section 2.4.

The output layer of a model is tailored to the task at hand. A **regression** model predicts a single, continuous value, represented by a single node in the output layer, or a vector of continuous values, represented by multiple nodes in the output layer. In contrast, a **classification** model sorts inputs into discrete categories, usually with the final output layer being a vector of nodes, each representing a class, with the value of each node representing the probability that the input belongs to that class. The specific differences between these two types of models will be discussed in Section 2.8.

2.2.3 Example: MNIST

2.3 Backpropagation

Backpropagation is the textbook algorithm for computing the gradient of the loss function. Essentially, it is the application of the chain rule to neural networks [11, 12].

Recall Equation 2.7,

$$a_j^{(l)} = \phi^{(l)} \left(b_j^{(l)} + \sum_i w_{ij}^{(l)} a_i^{(l-1)} \right). \tag{2.13}$$

For easier comprehension, define $z_j^{(l)}$ to be the weighted and biased sum of the previous layer that is the input to the activation function.

$$z_j^{(l)} = b_j^{(l)} + \sum_i w_{ij}^{(l)} a_i^{(l-1)} \tag{2.14}$$

Then, we can rewrite the equation as

$$a_j^{(l)} = \phi^{(l)}(z_j^{(l)}). \tag{2.15}$$

Assuming a MSE loss, the loss for a single label is the squared difference each node of the output layer $a_j^{(L)}$ and the label y_j.

$$C = \sum_j \left(a_j^{(L)} - y_j\right)^2 \tag{2.16}$$

In this section, we use L to denote the output layer of the network, so we instead use the other standard C to denote the loss function to avoid cconfusion.

We wish to compute the gradient of the loss by computing the partial derivative of the loss with respect to each parameter of the network. First, consider the weights and biases of the output layer, $w_{ij}^{(L)}$ and $b_j^{(L)}$. By the chain rule,

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}}, \tag{2.17}$$

and

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}. \tag{2.18}$$

For paramaters in the previous layers, we can simply apply the chain rule recursively.

$$\frac{\partial C}{\partial a_j^{(L)}} \underbrace{\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_j^{(L-1)}}}_{\text{output layer } L} \underbrace{\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial z_j^{(L-1)}}{\partial a_j^{(L-2)}}}_{\text{layer } L-1} \cdots \underbrace{\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \left(\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \text{ or } \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} \right)}_{\text{layer } l} \tag{2.19}$$

Note the following:

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \phi^{(L)\prime}(z_j^{(L)}), \tag{2.20}$$

$$\frac{\partial z_j^{(l)}}{\partial a_j^{(l-1)}} = \sum_j w_{ij}^{(l)}, \tag{2.21}$$

$$\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}, \tag{2.22}$$

$$\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = 1. \tag{2.23}$$

Substituting,

This notation is somewhat cumbersome, so we will introduce the standard notation used found in literature. Let $\nabla_{a^{(L)}} C$ be the vector of partial derivatives of the loss with respect to the output of the final layer C,

$$\nabla_{a^{(L)}} C = \left[\frac{\partial C}{\partial a_j^{(L)}}\right]. \tag{2.24}$$

Next, define $\delta^{(l)}$ be the vector of partial derivatives of the loss with respect to the weighted and biased sum of the previous layer $z^{(l)}$,

$$\delta^{(l)} = \begin{bmatrix} \frac{\partial C}{\partial z_1^{(l)}} \\ \vdots \end{bmatrix}. \tag{2.25}$$

Thus, we can recursively define $\delta^{(l)}$ by the following two equations:

$$\delta^{(L)} = \nabla_{a^{(L)}} C \odot \phi'(z^{(L)}), \tag{2.26}$$

$$\delta^{(L-1)} = \left(\mathbf{W}^{(L)}\right)^T \delta^{(L)} \odot \phi'(z^{(L-1)}). \tag{2.27}$$

Using Equation 2.14 and Equation 2.15, we can compute these terms as follows:

$$\frac{\partial C}{\partial a_j^{(L)}} = 2(a_j^{(L)} - y_j) \tag{2.28}$$

Substituting these terms into the equations, we have

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = 2(a_j^{(L)} - y_j)\phi'(z_j^{(L)})a_i^{(L-1)}. \tag{2.29}$$

wow this is really complicated to explain with math will come back later

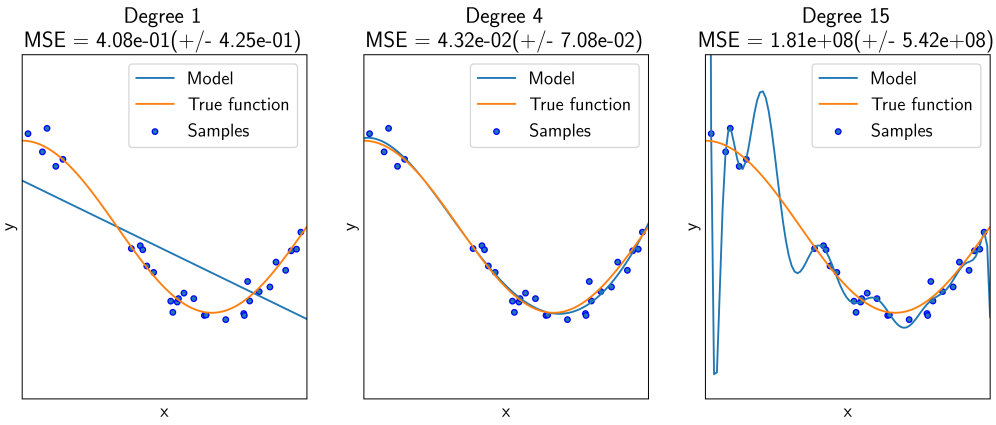


Figure 2.5: Polynomial models of different degrees fitted to sample data. Modified from documentation for `scikit-learn` [13].

2.4 Overfitting

Overfitting is a phenomenon that occurs when a model learns the training data too well, and is unable to generalize to new data. Generally, models with more parameters and degrees of freedom are more prone to overfitting. This may be mitigated by simply using a smaller model or by stopping training earlier, but these methods are not always feasible. Sometimes, there may be a fundamental difference between the training data and the test data that prevents any progress from being made. In general, we call problems of this nature **mismatches**, **distribution shifts**, or **dataset shifts**.

Models are usually tested on a seperate data set as a metric of how accurete their predictions are. An indicator of overfitting is when the model performs well on the training data, but poorly on the test data.

The degree 1 polynomial in the left of Figure 2.5 is underfitted and has too much model bias, while the degree 15 polynomial on the right is overfitted and has too much model variance.

2.4.1 Cross-validation

Cross-validation is a technique for estimating the performance of a model on unseen data, from only the training set. This ensures that any signs of overfitting can be monitored during training. Cross-validation splits the training data into a training set and a validation set. The model is trained on the training set, and the performance is evaluated on the validation set. This process is repeated for different partitions of the training data, and the average performance is taken as the final metric.

Cross-validation can be exhaustive where every possible partition is evaluated (leave-one-out cross-validation), though this is usually infeasible for large datasets.

k-fold cross-validation

The most common variant of cross-validation is k-fold cross-validation, a non-exhaustive method that randomly partitions the training data into k equally sized subsets called **folds**.

Algorithm 2 k-fold cross-validation.

1: Shuffle dataset $\mathcal{S}$
2: Partition $\mathcal{S}$ into k folds $f_1, \ldots, f_k$
3: **for** $i = 1, \ldots, k$ **do**
4: Train model on $\mathcal{S} \setminus f_i$
5: Evaluate model on f_i
6: **end forreturn** average performance over all k folds

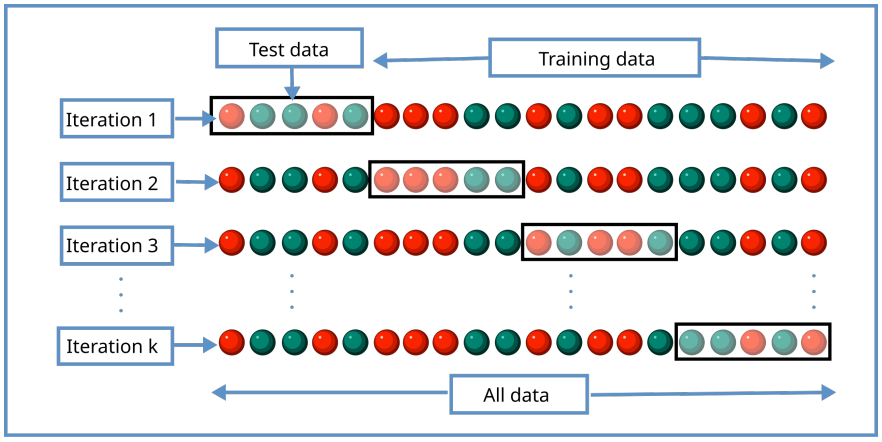


Figure 2.6: [14]

k is a hyperparameter, usually set to 5 or 10. The model is trained on $k-1$ folds, and the performance is evaluated on the remaining fold. This process is repeated for all combinations of folds, for a total of k iterations. Essentially, this is leave-one out applied to blocks of data rather than individual data points. The performance is averaged over all k iterations to give a final performance metric that is much more robust to overfitting than an unpartitioned training set

2.4.2 Regularization

learn more about this

2.4.3 Dropout

learn more about this

2.5 Data Batching

Rather than considering the entire training dataset all at once, we can iterate through small subsets of the training data, called **batches**, and compute gradient descent on that subset, updating parameters every batch. This is called **mini-batch gradient descent**, as opposed to **batch gradient descent**, which considers the entire training data set. (The author despises this inconsistent terminiology as much as you do.) After all batches have been iterated through, the model is said to have completed one **epoch** of training. The dataset is then shuffled, and the process is repeated for a specified number of epochs.

The size of a batch and the number of epochs to train for are both hyperparameters.

Batching also has the advantage of being able to parallelize the computation of the gradient, as each batch can be computed independently. Thus, **graphics processing units** (GPUs) and **tensor processing units** (TPUs) are highly effective for training neural networks, with training times reduced by orders of magnitude with hardware acceleration.

2.5.1 Batch size

2.5.2 Stochastic Gradient Descent

Rather than computing the gradient of the loss function over the entire training data set, **stochastic gradient descent** (SGD) iterates on every single example. This is much faster than the former method, usually called **batch gradient descent**, but it is also much noisier. However, a noisy path may not necessarily be undesirable, as it may help the model escape local minima and saddle points.

Algorithm 3 Stochastic gradient descent with learning rate η over N epochs.

Initialize parameters $\boldsymbol{\theta}$
for $i = 1, \ldots, N$ **do**
    Shuffle dataset $\mathcal{S}$
    for all example $\mathbf{x} \in \mathcal{S}$ **do**
        Compute gradient of loss $\nabla L_{\mathbf{x}}(\boldsymbol{\theta})$ with respect to $\mathbf{x}$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla L_x(\boldsymbol{\theta})$
    end for
end for
return $\boldsymbol{\theta}$

2.6 Gradient Descent Methods

The naive gradient descent algorithm introduced in Section 2.1.3 has many limitations.

Perhaps the most obvious problem is that it is susceptible to critical points that are not the global minima of the loss function. Since $|\boldsymbol{\theta}|$ is usually very large, L will have exceedingly many local minima and saddle points, any one of which may trap gradient descent.

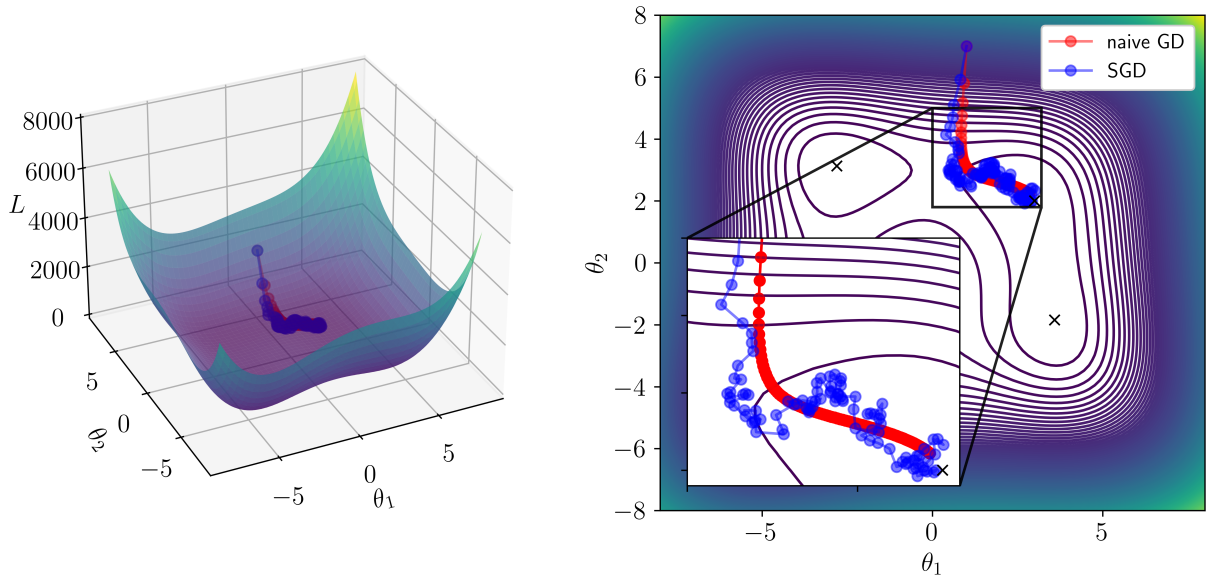


Figure 2.7: Comparison of SDG with naive GD on the Himmelblau function by a 3D plot (left), and a contour plot (right). Artificial noise selected from a normal distribtuion was added to the step direction to simulate stochastic effects. 1 epoch, 100 iterations, $\boldsymbol{\theta}^{(0)} = [1, 7]$, with $\eta = 0.001$. Local minima are marked with $\times$. (Original)

2.6.1 Momentum

The **momentum** modification for gradient descent adds an "inertia" term, by also factoring the previous update into the current update. This is similar to the physical concept of momentum, where an object in motion tends to stay in motion.

Define the change in parameters for an update as $\mathbf{v}$, with $\mathbf{v}^{(0)} = 0$, as

$$\mathbf{v}^{(t)} = \gamma \mathbf{v}^{(t-1)} - \eta \nabla L(\boldsymbol{\theta}^{(t)}), \tag{2.30}$$

where γ is a hyperparameter that determines the weight to which the previous movement is factored into the current movement. The parameters are then updated as

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t)} = \boldsymbol{\theta}^{(t)} + \gamma \mathbf{v}^{(t-1)} - \eta \nabla L(\boldsymbol{\theta}^{(t)}). \tag{2.31}$$

Note that gradient descent without momentum is just a special case of this, where $\gamma = 0$.

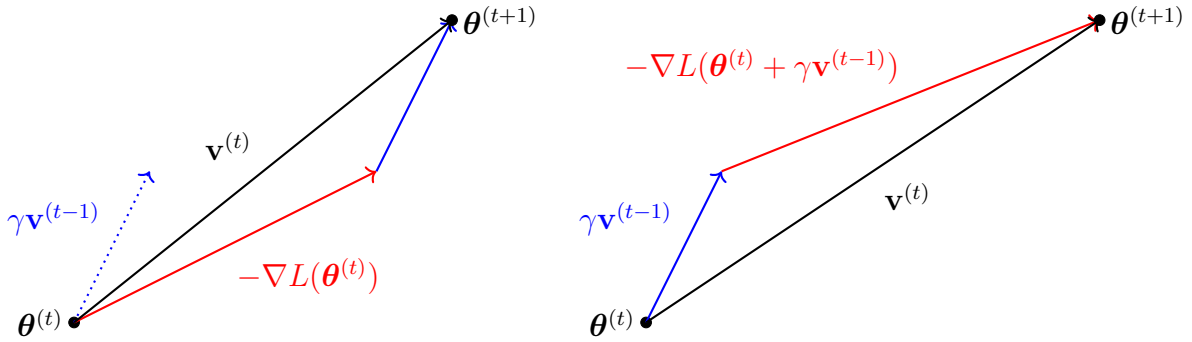
In effect, every movement is actually a weighted sum of all previous gradients, rather than just the most recent one.

$$\mathbf{v}^{(n)} = \sum_{i=0}^{n} -\gamma^{n-i} \eta \nabla L(\boldsymbol{\theta}^{(i)}) = -\eta \nabla L(\boldsymbol{\theta}^{(n)}) - \gamma \eta \nabla L(\boldsymbol{\theta}^{(n-1)}) - \gamma^2 \eta \nabla L(\boldsymbol{\theta}^{(n-2)}) - \cdots \tag{2.32}$$

Nesterov Momentum

In 1983, Russian mathematician Yurii Nesterov intrdouced a variant of momentum gradient descent by instead computing the gradient *after* first applying the intertial term [15].

$$\mathbf{v}^{(t)} = \gamma \mathbf{v}^{(t-1)} - \eta \nabla L(\boldsymbol{\theta}^{(t)} + \gamma \mathbf{v}^{(t-1)}). \tag{2.33}$$



(a) Standard momentum. **(b)** Nesterov momentum.

Figure 2.8: Visual representation of (a) standard momentum and (b) Nesterov momentum algorithms. The blue vector is the previous movement, the red vector is the gradient, and the black vector is the total movement. (Original)

This "look-ahead" method, usually called **Nesterov accelerated gradient (NAG)** or **Nesterov momentum**, allows the algorithm to be more responsive to the current gradient, and it is usually more effective than standard momentum. As such, Nesterov momentum is usually the default momentum implementation in software. Notice how momentum does not necessarily direct the model to the nearest local minima from the initial parameters. This has useful implications for training, as the model may be able to escape small local minima and saddle points, and it may be able to "swing" around large local minima to find a better minima elsewhere.

2.6.2 Adaptive Learning Rate

Since error surfaces are usually complex and non-uniform, a constant learning rate is not always ideal. For example, a small learning rate may be appropiate for a steep slope, but it may also be too small to make any progress on a relatively flat geometry. Conversely, a large learning rate may be appropriate for for flat geometries but may not converge to a minima on steep slopes. Error surfaces may also be anisotropic, where certain parameters are much more senesitive to changes than others.

Adaptive learning rates are a class of methods that introduce a variable σ to be both time-dependent and parameter-dependent to allow for more flexibility in the learning process.

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sigma_i^t} \frac{\partial L}{\partial \theta_i^{(t)}} \tag{2.34}$$

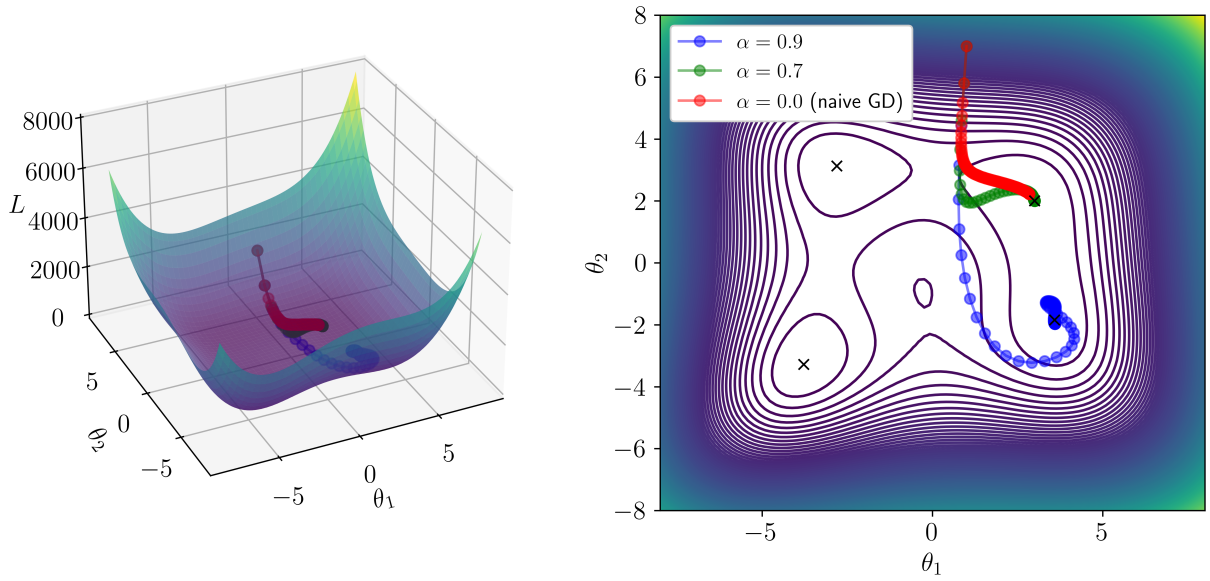


Figure 2.9: Comparison of Nesterov momentum optimizers on the Himmelblau function with various values of α visualized by a 3D plot (left), and a contour plot (right). 100 iterations, $\boldsymbol{\theta}^{(0)} = [1, 7]$, with $\eta = 0.001$. Local minima are marked with $\times$. (Original)

Adagrad

Adapative Gradient Descent, more commonly known as **Adagrad**, is a basic adapative learning rate method that sets $\sigma_i^{(t)}$ to be the sum of the squares of each iteration of the gradient with respect to the parameter i.

$$\sigma_i^t = \sqrt{\sum_{j=0}^{t} \left(\frac{\partial L}{\partial \theta_i^{(j)}} \right)^2 + \epsilon} \tag{2.35}$$

A very small $\epsilon \lll 1$ is included to prevent singularity problems when $\sigma_i^{(t)}$ is 0.

RMSprop

σ can only increase in Adagrad, which can cause problems if if the effective learning rate becomes too small. **RMSprop** is a modification of Adagrad that weights previous iterations of the gradient and the most recent gradient with a hyperparameter decay factor α.

$$\sigma_i^t = \sqrt{\alpha(\sigma_i^{(t-1)})^2 + (1-\alpha) \left(\frac{\partial L}{\partial \theta_i^{(t)}} \right)^2 + \epsilon} \tag{2.36}$$

This allows RMSprop to be more flexible and responsive than Adagrad, slowing down when the gradient is large and speeding up when the gradient is small.

Adam

Adaptive Moment Estimation, or **Adam**, is a more advanced adaptive learning rate method that combines the ideas of momentum and RMSprop. It uses two decay factors, β_1 and β_2, to weight the previous iterations of the gradient and the squared gradient, respectively.

[16]

read paper come back later

This algorithm is a standard in the field, and it is the default optimizer in many machine learning libraries, such as PyTorch.

2.6.3 Learning Rate Scheduling

The learning rate η is a hyperparameter that is usually set to a constant value. However, it may be beneficial to instead define η to vary with respect to iteration, $\eta^{(t)}$. This is called **learning rate scheduling**.

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta^{(t)}}{\sigma_i^t} \frac{\partial L}{\partial \theta_i^{(t)}} \tag{2.37}$$

Typically, $\eta^{(t)}$ is set to decay as the number of iterations increases to prevent sudden explosions in step size after many iterations of small σ.

Warmup

lmao let the model warm up???

2.7 Normalization

So far, we have only discussed optimization methods that do not directly change the loss surface. By appling statistical **normalization**, we can make the loss sufrace easier to optimize, independently of the optimization method used.

2.7.1 Feature Scaling

Feature scaling, also called **data normalization** by some authors, is a technique that the features of the training data such that they are around the same magnitude. This as a variety of benefits, such as faster convergence gradient descent convergence, and in general, greater stability in training.

A common method is **standardization**, where features are normalized component-wise with their z-score with respect to the training data.

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i} \tag{2.38}$$

Or, with vectors,

$$\tilde{\mathbf{x}} = (\mathbf{x} - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma} \tag{2.39}$$

2.7.2 Activation Normalization

In the case of deep learning, normalization can also be applied to the activations of a layer.

Batch Normalization

Batch normalization (BatchNorm) is a very common implementation of activation normalization, applied before the actiavation function of a layer. BatchNorm takes advantage of the parallel computafortion of features within a batch and interleaves standardization simultaneously with the forward pass of the network.

insert figure here

Since normalization with z-scores moves the mean to 0, this introduces some model bias. Parameters $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are introduced to get rid of this bias. They are *not* hyperparameters and can be learned by the network. It is possible that these parameters will reintroduce disparities in magnitude, but they are usually initialized with values of $\mathbf{1}$ and $\mathbf{0}$ and only change the scaling slowly.

In testing, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are replaced with moving averages, as it is infeasible to wait around for a full batch of data.

It is still debated why exactly BatchNorm is so effective, but the prevailing hypothesis is that it has the effect of smoothing the loss surface, making it easier to optimize. [17][18][19]

2.8 Classification

Some tasks require a model to classify an input into one of a finite number of classes. Perhaps the most well known example is identifying handwritten digits from the MNIST dataset. In such cases, it is far from optimal to represent classes as scalar value features to train towards. Instead, classes are usually represented as one-hot vectors, where the index of the class is set to 1, and all other indices are set to 0.

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.40}$$

2.8.1 Softmax

The output vector in classification is usually passed through the **softmax** function. The softmax function is a generalization of the sigmoid/logistic function to vector values, mapping all vectors to be wthin the unit hypercube. The softmax function is useful for classification problems, as it normalizes the output of a model to be a probability distribution over the

classes, where the sum of all components is 1. This is useful for many applications, such as when the output of a node is interpreted as a probability.

Formally defined, it is a function $\sigma\colon \mathbb{R}^K \to (0,1)^K$, where $K > 1$, takes a vector $\mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$ and computes each component of vector $\sigma(\mathbf{z}) \in (0,1)^K$ with

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\,. \tag{2.41}$$

In other words, every component is exponentiated and then normalized by dividing with the sum of all exponentiated components.

The inputs to the softmax function are also called **logits**.

When implemented in code, the final normalized vector after being passed through the softmax function is then called by **argmax** to identify the index of the class with the highest probability, and thus the predicted class.

$$\begin{array}{ccccc} \text{penultimate activation/logits } \mathbf{z} & & \text{final layer probablilities } \sigma(\mathbf{z}) & & \text{identified class } \mathbf{y} \\ \begin{bmatrix} 1.2 \\ 0.3 \\ 3.0 \\ -0.9 \end{bmatrix} & \xrightarrow{\text{softmax}} & \begin{bmatrix} 0.13 \\ 0.05 \\ 0.80 \\ 0.02 \end{bmatrix} & \xrightarrow{\text{argmax}} & \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array} \tag{2.42}$$

2.8.2 Cross Entropy Loss

Recall the formula for the ubiquitous MSE loss for vector outputs.

$$L = \frac{1}{N} \sum \left(\sum_i (\hat{y}_i - y'_i)^2 \right) \tag{2.43}$$

For classification purpose, another loss called **cross entropy** is more commonly used instead, often paired with softmax. Minimizing cross entropy is equivalent to maximizing likelihood, for various information theory reasons. `https://www.youtube.com/watch?v=fZAZUYEeIMg`

$$L = \frac{1}{N} \sum \left(-\sum_i \hat{y}_i \ln y'_i \right) \tag{2.44}$$

Note that $\ln y'_i$ is always negative because y' is the output of the softmax, and thus the term inside the parentheses is always positive.

Cross entropy allso usually gives smoother loss surfaces compared to MSE for classification problems.

2.9 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) are a class of neural networks that are particularly well-suited for image processing tasks. Consider an image with dimensions of $w \times h$ and c channels. This can be represented as a rank 3 tensor $\mathbf{X} \in \mathbb{R}^{w \times h \times c}$, which will be the input to a neural network. If we use a fully connected neural network, the number of parameters in the first hidden layer will be enormous, as each node will be connected to every pixel (or even every channel) in the image. This is not only computationally expensive, but it also makes the model very prone to overfitting.

[20] steal figures from this textbook its goated

insert figure here

CNNs are designed to take advantage of the structure of images, and other types of data that may be "grid-like" in nature.

2.9.1 Convolutional Layers

The main building block of a CNN is the **convolutional layer**, which applies a convolution operation to the input data. One may be familiar with the continuous definition of the convolution operation,

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\ d\tau. \tag{2.45}$$

Note that the convolution is commutative, so $f * g = g * f$.

In the discrete case, the convolution may be defined analogously as

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m) g(n - m), \tag{2.46}$$

where f and g are discrete functions or vectors.

insert figure here

This can be generalized to higher dimensions with f and g being matrices or tensors. Without loss of generality, we can visualize this as a sliding window, called a **receptive field**, that moves across f and computes the inner product with g at each position, and mapping the result to a new tensor $h = f * g$, called the **feature map**. g is usually called a **kernel**, **filter**, **mask**, or a **template** [21]. The idea of this process is that a kernel may be able to detect certain patterns in the input data, such as edges, corners, or textures, and extract them into the feature map.

The size of the receptive field and the kernel is usually much smaller than the input data, and it is usually square for image processing tasks. The specific offset of the receptive field for each iteration of its movement can also be controlled by a hyperparameter called the **stride**. The strictly orthodox defintion of the discrete convolution in Equation 2.46 has a stride of 1, meaning that the receptive field moves in offsets of 1 pixel at a time. Theoretically,

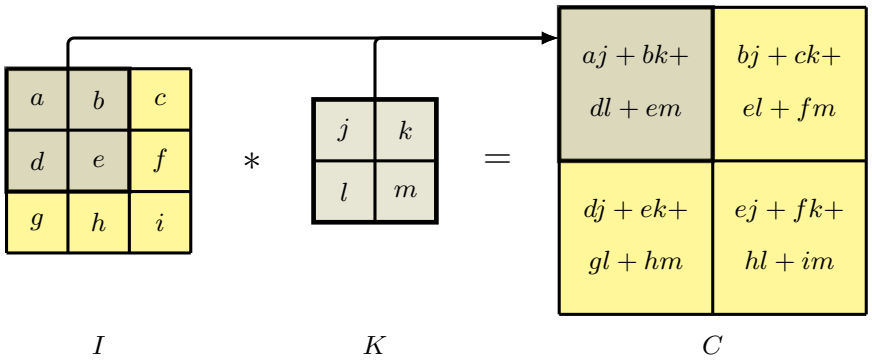


Figure 2.10: A 2×2 kernel is convolved with a 3×3 input tensor, producing a 2×2 feature map. Reprinted from [20].

0	0	0	0	0	0
0	X_{11}	X_{12}	X_{13}	X_{14}	0
0	X_{21}	X_{22}	X_{23}	X_{24}	0
0	X_{31}	X_{32}	X_{33}	X_{34}	0
0	X_{41}	X_{42}	X_{43}	X_{44}	0
0	0	0	0	0	0

Figure 2.11: A 4×4 tensor is zero padded to a 6×6 tensor, allowing a 3×3 kernel to be applied to the edges of the original input tensor. Reprinted from [20].

convolution stride may be set to any positive integer or even vary with respect to specific axes. Though, in practice, the stride is for most cases kept to 1, or sometimes some other very small integer, and is also isotropic.

In addition, the input is padded with extra values around the edges such that the convolution is defined for all positions of the receptive field, including the edges. **Zero padding** is the most common method of padding, where the extra values are set to 0, though other methods such as copying the edge values or using a constant value are also possible.

Each kernel consists of individual parameters that may be learned. Thus, a convolutional layer with k individual kernels will have k individual feature maps. For an RGB image with dimensions $w \times h \times 3$, a typical kernel size is $3 \times 3 \times 3$, and the output feature map will have dimensions of $w \times h \times k$.

Using the traditional visualization of a neural network, a convolutional layer consists of nodes that each correspond to a single kernel at single position of the receptive field. Each node is only connected to the values in the receptive field rather than the entire input layer with the

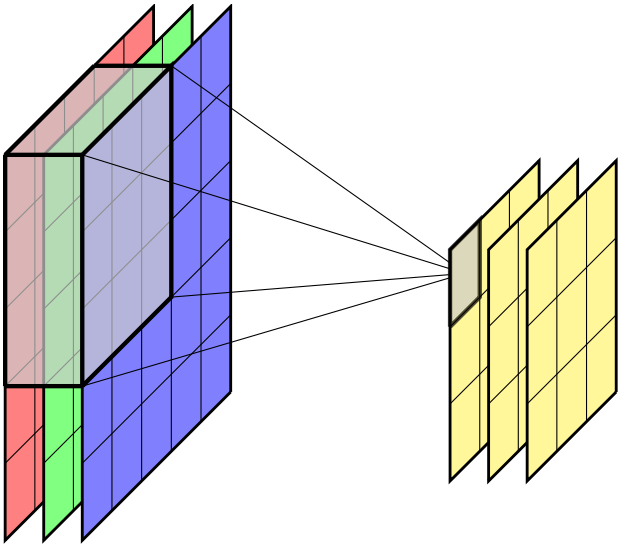


Figure 2.12: 3 kernels with dimension $3 \times 3 \times 3$ are applied to a $6 \times 6 \times 3$ input tensor, producing a combined feature map with dimension $3 \times 3 \times 3$. Reprinted from [20].

weights being the parameters of the kernel, with different nodes *sharing the same weights* if they use the same kernel. These two ways of thinking about a convolutional layer are fundamentally the same.

Table 2.2: Equivalent concepts between different visualizations of a convolutional layer. Adapted from [22].

Tensors and Kernels	Network Nodes and Weights
Input tensor	Input layer
Individual pixel or channel	Node
Kernel	Shared weights of nodes
Convolution	Weighted sum of connections
Kernel size	Number of connections to a node
Feature map	Output layer

insert figure here

2.9.2 Activation Function Layers

Just like in a fully connected neural network, the output of a convolutional layer is usually passed through an activation function to introduce non-linearity into the model. By far the most common choice of activation function for CNNs is the ReLU.

2.9.3 Pooling Layers

Pooling is a technique used to reduce the size of feature maps in a CNN, a process called **downsampling**. Notice that scaling an image down to a smaler size does not fundamentally

change the information in a image by much, we are still able to recognize the image, even if it is more pixelated. We can apply the same concept by taking a small region of the feature map applying some "aggregate" function to the values in that region to get a single value. This very similar to a convolutional layer, with the receptive field feeding to some scalar function rather than being convolved with a learned kernel. If we wish to "tile" the feature map with no overlap, we set the stride to be equal to the size of the receptive field.

The most common type of pooling is **max pooling**, where the maximum value in each receptive field is taken as the representative value. Other common operations may be taking the mean or the median.

Pooling prevents feature maps from growing too large, which would be computationally expensive and introduce too many parameters. For example, a pooling layer with a 2×2 receptive field and a stride of 2 will reduce the size of a 100×100 feature map to 50×50, decreasing the number of parameters for the next layer by a factor of 4.

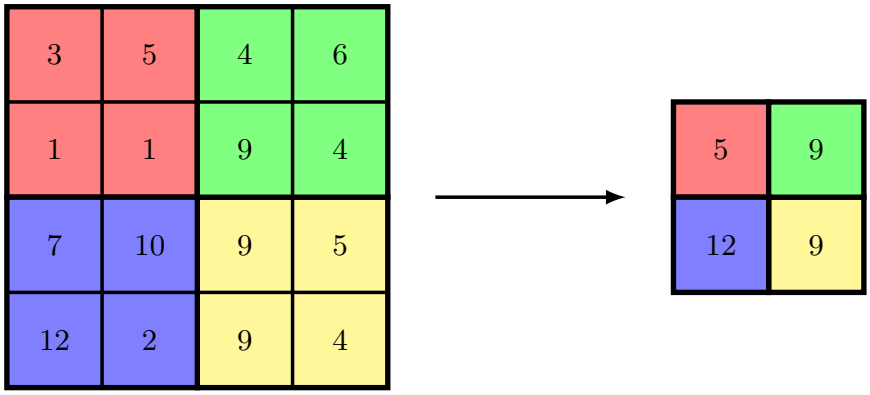


Figure 2.13: Max pooling with a 2×2 receptive field and a stride of 2. The dimensions of the tensor is reduced from 4×4 to 2×2. Reprinted from [20].

In addtion, pooling also has the effect of making the model more robust to small translations and distortions in the input data, as the pooling operation is usually invariant to small changes in the input.

In general, the loss of information from pooling is not as big of a problem as it may seem since pooling is usually applied after a convolutional layer, when relevant features of the input have already been extracted. Though, there are some examples where pooling is not used, particularly in cases where such small details may be important. As computing power becomes cheaper and more available, pooling is becoming less common in modern architectures, with other methods of downsampling becoming more common, such as increasing the stride of convolutional layers.

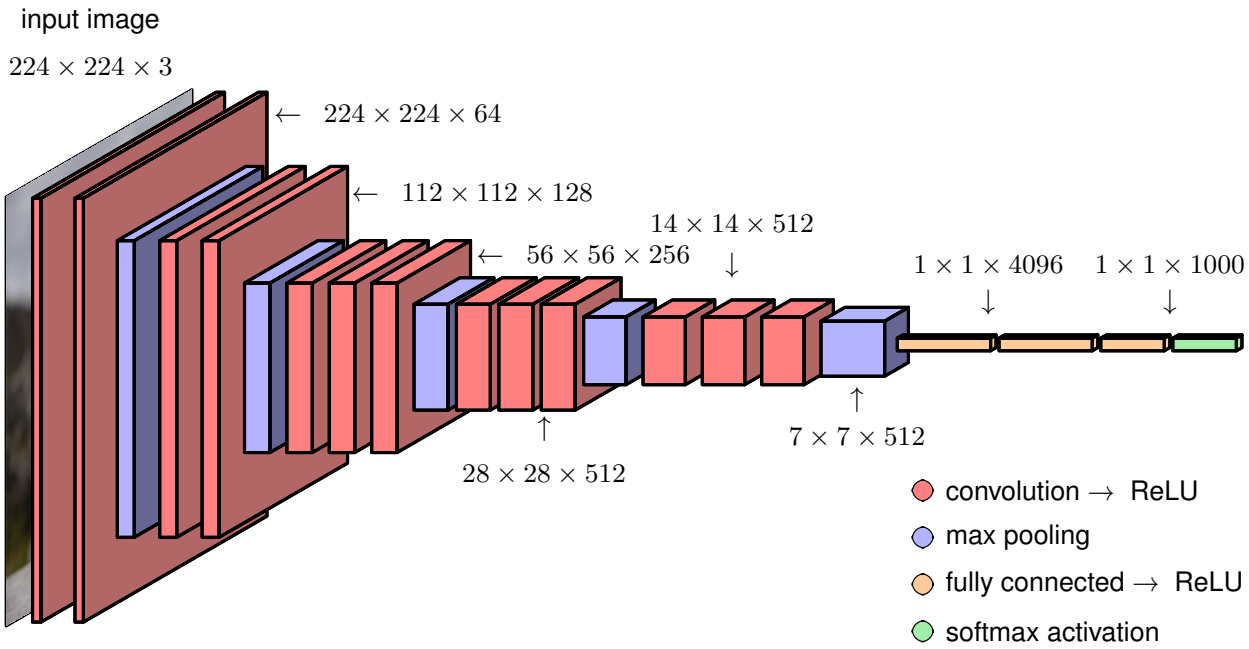


Figure 2.14: The architecture of VGG-16 from Simonyan & Zisserman (2015) [23]. The CNN consists of 13 convolutional layers, 5 max pooling layers, and 3 fully connected layers. Reprinted from [20].

2.9.4 Upsampling Layers

2.9.5 CNN Architecture

Convolutional, activation, and pooling layers are stacked together to form the architecture of a CNN model. In addition, fully connected layers are usually added at the end of the model to make the final predictions. All the hyperparameters of the model, such as the number of layers, the specific combination and ordering of layers, the number of kernels, the kernel size, the stride, and the pooling size, can all be tuned to improve the performance of the model for a specific task. Thus, CNNs are highly flexible and have found success in a wide variety of applications, such as image classification, object detection, and image segmentation.

It is important to note that CNNs have extensive applications in non-image data as well, such as spectrograms and text. Essentially, any data that can be represented in a sequence or grid of values and are "spatially related" (not necessarily in the literal sense, e.g. in temporal data, we convolve with respect to time rather than space) can be processed with CNNs.

maybe talk about dilation layers?

Chapter 3

Reinforcement Learning (RL)

So far, we have only discussed ML techniques that are **supervised**, where the model is trained on a dataset with labels directing the model on what the desired output should be. The task of labeling is often done manually by a human, which is often infeasible or impractical for many applications. Sometimes, even humans may not even know what the best answer is.

The framework of **reinforcement learning (RL)** is a class of ML techniques that do not require labeled data, and thus are generally unsupervised. Instead, RL models learn by interacting with an environment, and receiving feedback in the form of rewards or penalties. The goal of an RL model is to learn a **policy** that maximizes the expected reward over time.

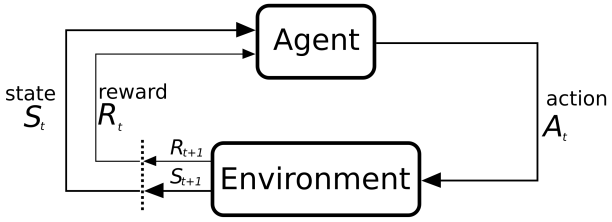


Figure 3.1: [24]

put tikz figure here

In RL, there are two main components: the **agent** and the **environment**. The agent is a model that takes in observations from the environment and outputs actions. The environment then judges the action and returns a **reward**. The reward may be negative, in which case it may be called a **penalty**, **cost**, or **loss**. The agent then uses the reward as feedback to learn how to improve its performance.

3.1 Markov Decision Processes (MDPs)

A **Markov descision process** (MDP) is a mathematical framework that can be used to model decisionmaking in a RL agent. MDPs are generalizations of **Markov chains**, which are stochastic processes that are **Markovian**, meaning that the future state of the process only depends on the current state, and not on any previous states. In other words, the future is independent of the past given the present; there is no hysteresis. Note that this does not mean that Markovian processes are deterministic.

Formally, an MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$, the **state space**, is the set of all possible states of the agent, which corresponds to the observations that the agent receives from the environment.
- $\mathcal{A}$, the **action space**, is the set of all possible actions that the agent can take.
- $\mathcal{P}$, the **transition function**, outputs the probability of transitioning from one state to another via a specific action.
- $\mathcal{R}$, the **reward function**, outputs the reward received by the agent after transitioning from one state to another given a specific action.
- $\gamma \in [0, 1]$, the **discount factor**, is a hyperparameter that determines how much the agent should care about future rewards.

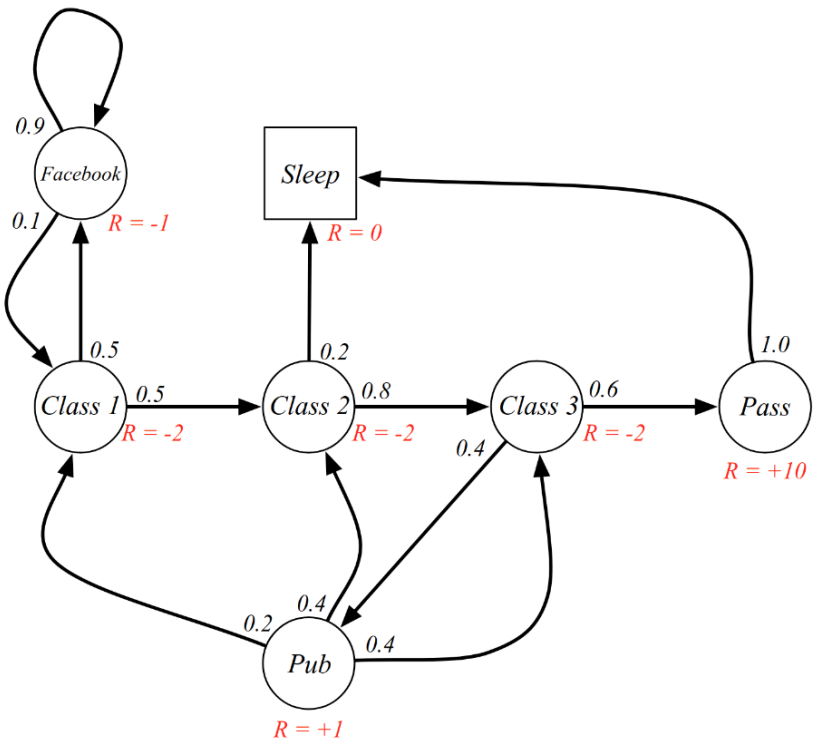


Figure 3.2: Reprinted from [25]. Gotta make one of my own lol.

put tikz figure here

3.1.1 Episodes and Trajectories

An **episode** is a single session of the agent in the environment, starting from an initial state s_0 and ending when the agent reaches a terminal state s_T. The agent interacts with the environment by taking actions a_t and receiving rewards r_t at each time step t according to $\mathcal{R}$. The specific sequence of states, actions, and rewards that the agent experiences during an episode is called a **trajectory**.

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T, a_T, r_T) \tag{3.1}$$

A set of trajectories experienced by an agent may be collectively considered as $\mathcal{D}$ in notation. Typically, $\mathcal{D}$ is the training data used for optimizing the agent.

3.1.2 Policies

The **policy** $\pi(s, a) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a function that outputs the probability of taking action a in state s.

$$\pi(a|s) = \mathbb{P}(a_t = a|s_t = s) \tag{3.2}$$

Equivalently, the policy can be thought of as a function that maps from the state space to a probability distribution over the action space.

Recall the universal approximation theorem from Section 2.2. Since π is a function that maps from the state space to a probability distribution over the action space, it can be approximated by a classification neural network. An optimal π is almost always highly complex and non-linear, and thus for most applications of RL, π is represented by a neural network. The neural network then called the **policy network** of the agent.

The pairing of deep learning with RL is called **deep reinforcement learning** (DRL). We will mainly discuss RL concepts and methods in the context of DRL, as it is widespread in the field of robotics, however it is important to note that RL is not limited to DRL. In fact, many of the RL methods discussed in this article can be applied to non-neural network policies as well.

Evaluating Policies

The goal of the agent is to learn a policy that maximizes the expected reward over time. How can we encapuslate this idea mathematically?

Recall that the environment returns a reward according to R each time the agent takes an action a_t. It would be unreasonable to assume that the immediate reward for that action is the only thing that matters. An action may have long-term consequences that affect the future rewards of the agent. Therefore rather than only considering a specific r_t, we consider the **discounted return** G as the metric to judge an action.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{3.3}$$

G can be defined on a trajectory τ as

$$G(\tau) = \sum_{t=0}^{T} \gamma^t r_t. \tag{3.4}$$

The **expected return** J_π of a policy π is then formally defined as the expected value G over all possible trajectories that the agent may take.

$$J_\pi = \mathbb{E}_\pi[G(\tau)] = \int_\tau G(\tau)\mathbb{P}_\pi(\tau)\ d\tau. \tag{3.5}$$

This value, J_π, is the value that we want to maximize to optimize the policy π.

Additionally we may define two other closely related functions, the **state-value function** $V_\pi(s) : \mathcal{S} \to \mathbb{R}$ defined as the expected discounted return of the agent at state s,

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s], \tag{3.6}$$

and the **action-value function** $Q_\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ defined as the expected discounted return of the agent at state s and taking action a,

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]. \tag{3.7}$$

Note the following relationship between the two functions,

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s, a). \tag{3.8}$$

Bellman Equation

Learning Policies

3.1.3 Continuous State and Action Spaces

Up to this point, our notation has been limited to discrete state and action spaces. It is important to take note that for many applications $\mathcal{S}$ and $\mathcal{A}$ may be continuous spaces, such as the position and velocity of a robot arm, or the torque to apply to a joint. It is rather easy in most cases to extend the concepts that we have discussed to the continuous case: replacing sums with integral, probabilities with probability density functions, and so on. So, we will not go into much detail here about the specific nuances between discrete and continuous domains, just keep in mind that there may be some abuses of notation here and there.

3.2 Policy Gradient

Policy gradient methods are a class of RL algorithms that learn a policy by directly optimizing the expected return J_π. This is typically accomplished by parameterizing the policy π with $\boldsymbol{\theta}$ and then using gradient descent/ascent to optimize J_π with respect to $\boldsymbol{\theta}$. Note that this is exactly the same as the optimization problem we have discussed in the previous chapter. In fact, in DRL, the parameters of the policy *are precisely* the parameters of its neural network.

3.2.1 Policy Gradient Theorem

The **policy gradient theorem** (see proof in Appendix A.1) states that the gradient of J_π with respect to the parameters $\boldsymbol{\theta}$ can be expressed as

$$\nabla_{\boldsymbol{\theta}} J_\pi = \mathbb{E}_\pi \left[G(\tau) \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \right], \tag{3.9}$$

where the summation represents all pairs of (s_t, a_t) in a trajectory. This is a important result, as it allows us to compute the gradient of J_π, and thus optimizing $\boldsymbol{\theta}$, without having to compute the state-value function V_π. Inaddition, policy gradient methods are **model-free** methods, as they do not require any knowledge of the environment, its associated value functions or the transition function. The theorem only presupposes that $\pi_{\boldsymbol{\theta}}$ is differentiable, which is a reasonable assumption for almost all neural networks.

In practice, the right-hand side of the equation is estimated by sampling trajectories from the environment in a Monte Carlo fashion, and then computing the average over all sampled trajectories.

$$\nabla_{\boldsymbol{\theta}} J_\pi \approx \frac{1}{N} \sum_{i=1}^{N} G(\tau^{(i)}) \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t^{(i)}|s_t^{(i)}) \tag{3.10}$$

3.2.2 `REINFORCE`

In 1992, Ronald J. Williams introduced the basic `REINFORCE` algorithm, which uses an even simpler version of the policy gradient theorem in combination with sampling trajectories in a Monte Carlo manner. [26] Note that line `8` is equivalent to gradient descent on $-J_\pi$, which means that all the the optimization techniques that we have discussed in Section 2.6 can be applied here, up to a difference in sign.

3.2.3 Trust Region Policy Optimization (TRPO)

[27]

3.2.4 Proximal Policy Optimization (PPO)

[28]

Algorithm 4 `REINFORCE` with policy $\pi_{\boldsymbol{\theta}}$, learning rate η, and N episodes.

Initialize parameters $\boldsymbol{\theta}$
for $i = 1, \ldots, N$ **do**
    Play episode i
    $\tau^{(i)} \leftarrow (s_0^{(i)}, a_0^{(i)}, r_0^{(i)}, \ldots, s_T^{(i)}, a_T^{(i)}, r_T^{(i)})$
    for $t = 0, \ldots, T$ **do**
        $G_t^{(i)} \leftarrow \sum_{k=0}^{T} \gamma^k r_{t+k}^{(i)}$
        $\nabla_{\boldsymbol{\theta}} J_\pi \leftarrow G_t^{(i)} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t^{(i)} | s_t^{(i)})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J_\pi$
    end for
end for
return $\boldsymbol{\theta}$

3.3 Actor-Critic Methods

In contrast to policy gradient methods, **actor-critic** methods learns the state-value function V_π directly. We define a **critic** as a seperate entity from the the agent, which also may be a neural network.

3.3.1 Advantage Actor-Critic (A2C)

3.3.2 Asynchronous Advantage Actor-Critic (A3C)

3.4 Q-Learning

3.5 Reward Shaping

Some environments have very sparse rewards, such as playing chess or go, where the agent only receives a definitive reward at the end of the game based on whether it wins or loses. This is a problem for learning, as the agent have trouble associating the actions it took during the game with the final reward. This is an example of the **credit assignment problem**, which is the fundamental reason why RL is so difficult.

Thus, we may define additional rewards to guide the agent, by **reward shaping**. By applying domain knowledge, we can notice that certain actions are more likely to lead to a positive outcome, and assign a small reward for those actions, even if they are not inherent in the environment.

3.5.1 Example: VizDoom

VizDoom is a adaptation of the classic 1996 video game *Doom* for use as a RL research environment [29]. *Doom* is a first-person shooter game where the player must navigate through a maze and defeat enemies. The game is highly complex, but rewards are sparse, as the player only receives a reward when they beat the level. Therefore, researchers introduce

additional rewards for picking up medkits, killing enemies, and even just surviving and moving around.

Figure 3.3: Reprinted from [29].

3.5.2 Curiosity

Curiosity is a form of intristic reward that incentivizes the agent to explore its environment and to encounter novel states. The idea behind curiosity is that increasing the seen state space of the agent will allow it learn a better policy, as it has a more complete understanding of the environment. When there are sparse extrinsic rewards, or even a complete absence of extrinsic rewards, curiosity can be effective for training an agent in such environments [30]. This idea finds applications where the environment is not well defined and lacking of any obvious reward structure.

maybe put fig here?

3.6 Imitation Learning

In cases where an optimal or desired behavior is known, such as in the case of human demonstrations, it may be more efficient to train a model to imitate the behavior from an expert rather than learning from scratch. This is called **imitation learning**. Immitation learning is a form of supervised learning, where the model is trained on a dataset of demonstrations.

Behavior cloning (BC) is a form of imitation learning, where the policy network $\pi(a|s)$ simply learns from state-action pairs (s, a) from expert demonstrations. This technique is very straightforward to implement but it is also very limited. For example, mismatches may occur when the learned policy is placed in an unfamiliar environment that was never reached by the expert. A simple analogy, is learning how to play soccer by watching Messi. You are not Messi. Messi will never make basic mistakes, but you will, and you will not know what to do in those situations.

put tikz figure here

3.6.1 Dataset Aggregation (DAgger)

3.6.2 more methods look into them

3.6.3 Inverse Reinforcement Learning (IRL)

In contrast to behavior cloning, **inverse reinforcement learning (IRL)** is a form of imitation learning where the model learns a reward function (usually in the form of a neural network) from expert demonstrations, rather than the expert policy. The learned reward function can then be used to train a from-scratch agent using standard RL methods.

put tikz figure here

3.7 Cirriculum Learning

[31]

[32]

Chapter 4

Applications of RL in Robotics Literature

Now that we have built up a solid foundation for the basics of reinforcement learning, we will now review examples of applications of RL in the field of robotics. This section is by no means extensive or comprehensive, it is only meant to serve as a tasting menu of the state of the art in the field. To acomplish this, we will approach a class of robots as a case study. We will then examine methods that have been used in the literature for that platform, and how they relate to the concepts we have discussed previously in this article.

4.1 ANYmal: Quadrupedal Legged Robot

Originally developed for the DARPA Robotics Challenge, the ANYmal is a quadrupedal legged robot developed by ANYbotics. It is a highly capable platform that has been used in a variety of applications, including search and rescue, inspection, and exploration [33, 34].

The ANYmal is a highly articulated robot with four legs, each with three joints and degrees of freedom (DoF). Ordered from the hip to the tip of the leg, these DoF are named in literature as Hip Abduction/Adduction (HAA), Hip Flexion/Extension (HFE), and Knee Flexion/Extension (KFE). Each joint is driven by an electric motor actuator, totaling to 12 actuators controlling the robot's motion. The full range of motion of a single leg is shown in Figure 4.2.

4.1.1 Locomotion of Legged Robots

At a high level, robotic locomotion may be controlled by commanding the robot to move at a specific velocity in a specific direction, such as through inputs with a joystick or a keyboard. It is then the job of the software to translate these commands into torques to apply to the actuators of the robot.

Traditional methods for locomotion of legged robots are often computationally expensive and require a lot of manual tuning to work well in practice. In addition, they are often

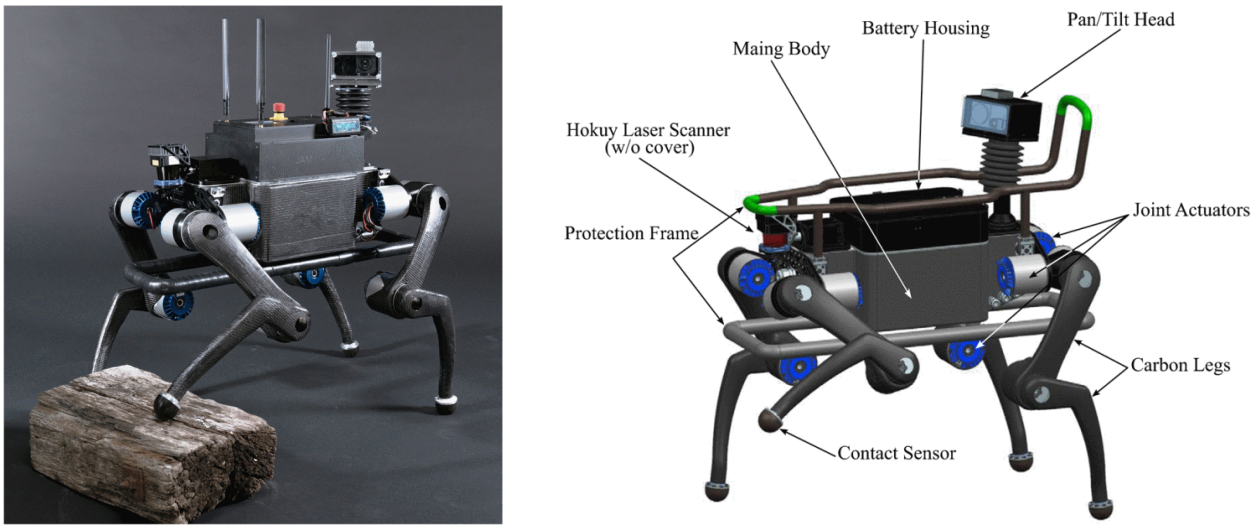


Figure 4.1: ANYmal quadrupedal robot. Reprinted from [33].

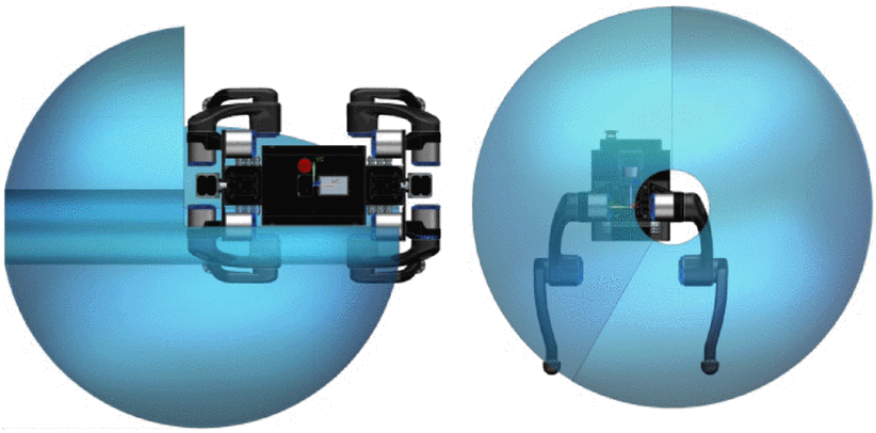


Figure 4.2: Reprinted from [33].

not robust to changes in the environment. For example, robots that have different payloads may have markedly different mass distributions and physical dynamics, and thus require additional tuning to work well. In contrast, reinforcement learning (RL) methods are more flexible and allows us to black-box the control of the robot with a policy network.

Table 4.1: Comparison of methods for locomotion and navigation of legged robots.

Aspect	**Traditional Methods**	**Data-Driven RL**
Real-time Computational Demand	High	Low
Training Computational Demand	None	Very High
Human Labor Demand	High	Low
Robustness to Environment Changes	Low	High

Hwangbo et al. (2019)

We examine a model-free RL approach first published in Hwangbo et al. (2019) [35] for controlling the ANYmal. With RL, the authors were able to achieve state-of-the-art performance in a variety of locomotion tasks, including walking and fall recovery. Compared to previous non-RL methods. The learned policy matched commanded velocities with greater accuracy, were more energy efficient, and exerted less stress on the robot's joints. In addtion, the learned policy was able to achieve a significantly higher top speed, without comprimising gait stability.

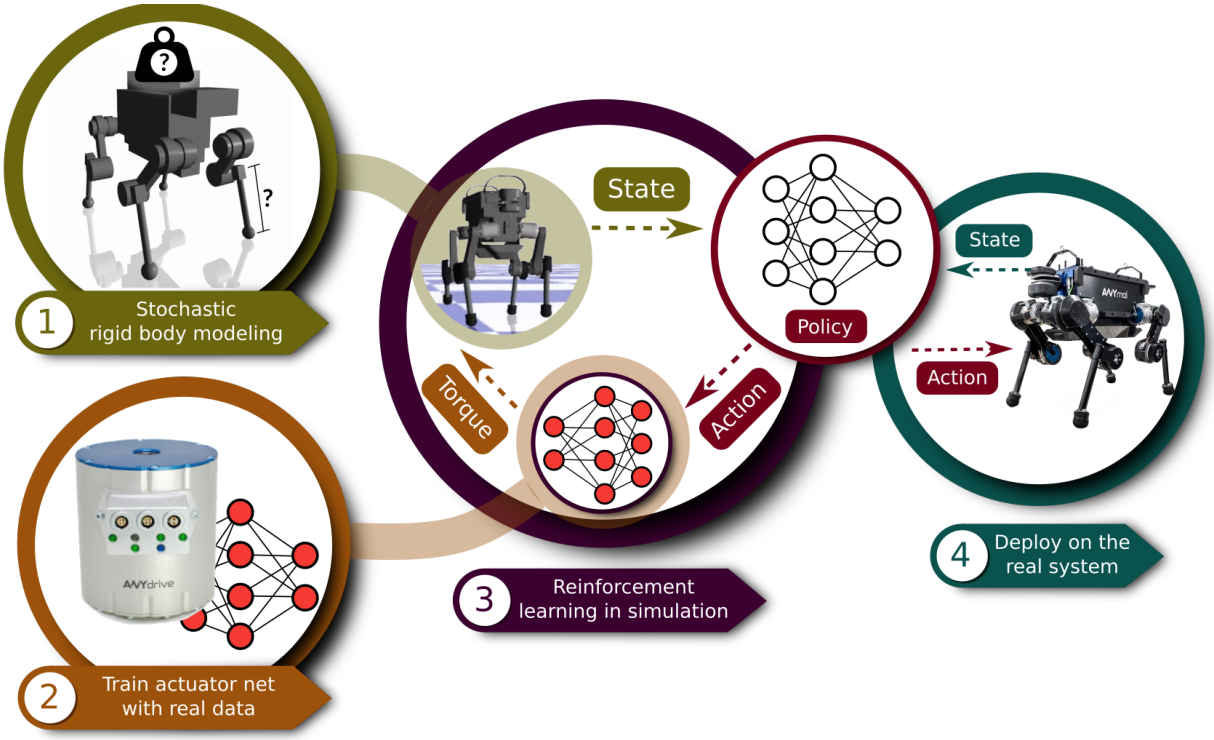


Figure 4.3: Repreinted from [35].

Figure 4.3 shows an overview of the training process. Because of the infeasibility of training a a policy on a physical robot, the authors trained the policy in a simulated environment, and then transferred the policy to the real robot.

The policy network outputs the desired joint positions of the robot, which are passed to a controller that computes the actuator torques to apply to the robot. Each observation to the policy network consisted of relevant joint state information, its history, the previous action of the policy, and the command passed to the robot. The controller that takes the joint positions and computes the actuator torques is commonly a proptional-derivative (PD) controller or proportional-integral-derivative (PID) controller.

LOOK INTO THIS??? inverse kinematics???

Since realistic actuators are very difficult to model accurately in simulation, the authors used supervised learning to train a standard feedforward neural network to learn the actuator

torques from desired and actual joint positions and velocities that is usually passed to the controller *in situ*. This actuator net essentially acts as the vrtual controller *in silico*. The torques are then passed to the rigid body simulation to provided realistic dynamics for the policy to learn from. The architecture of the training process is shown in Figure 4.4.

In simulation, the authors also include models of the robot with different mass distributions to make the policy robust to such changes in intertial properties. Since trajectories were generated in simulation, the authors were able to considerably speed up training as they were not constrained to the temporal limitations of the material world. Using TRPO, the published policy was trained in less than 24 hours of real time, with a cirrciulum in place such that the costs imposed on the policy gradually increased as the policy improved. This allowed the policy to learn to walk in the first place, and then optimize its gait later in training. For comparison, previous non-RL methods may take weeks or months to tune.

Lee et al. (2020)

The same group of authors published a successor paper in 2020 [36] that builds on the previous work, using a completely different architecture to improve the performance of the policy over highly diverse terrains. Using purley propioceptive internal state information, the policy was able to learn to walk on a variety of terrains with different textures, slopes, and coefficients of friction, as shown in Figure **??**, without any prior knowledge of the environment.

Instead of training a traditional feedforward neural network on state information, the authors instead use a temporal convolutional network (TCN) to process the state history, rather than just the most recent state. In addition, another policy is trained with privileged **ground-truth** information about the environment, such as the terrain type and external forces acting on the robot. This privileged policy is designated as the teacher policy, which is used to train the student policy via DAgger imitation learning. According to the authors, this approach was necessary to learn effective policies, as this locomotion task is markedly more difficult than the previous work, and outright learning a policy from scratch was not feasible. In simulation, diverse terrain was proceduralllly generated to to populate the environment with terrain types that the robot may encounter in the real world. The novel architecture presented in Lee et al. (2020) is visualized in Figure **??**.

[34]

4.2 Exoskeletons

[37] [38]

4.3 more examples will go here

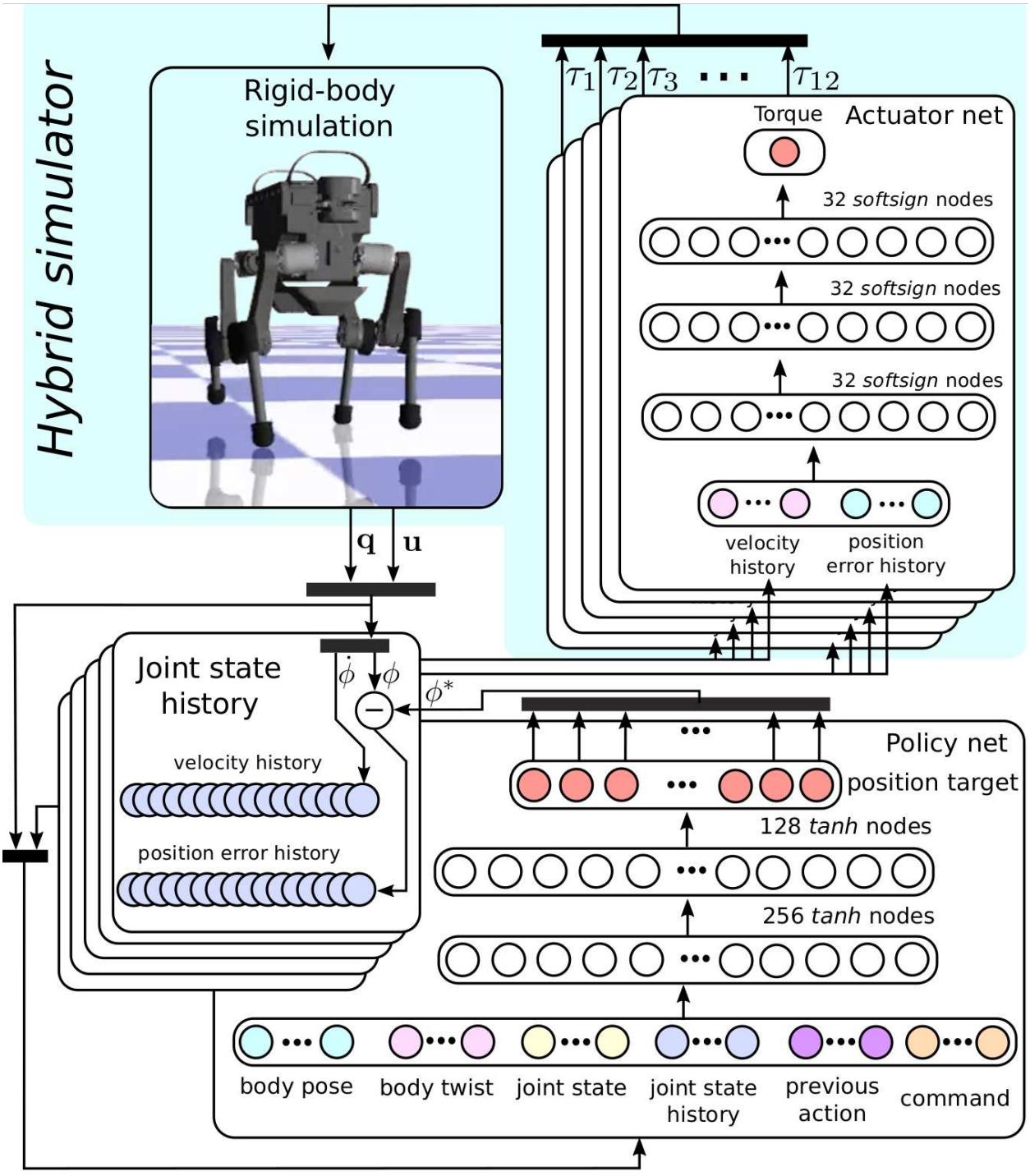


Figure 4.4: Reprinted from [35].

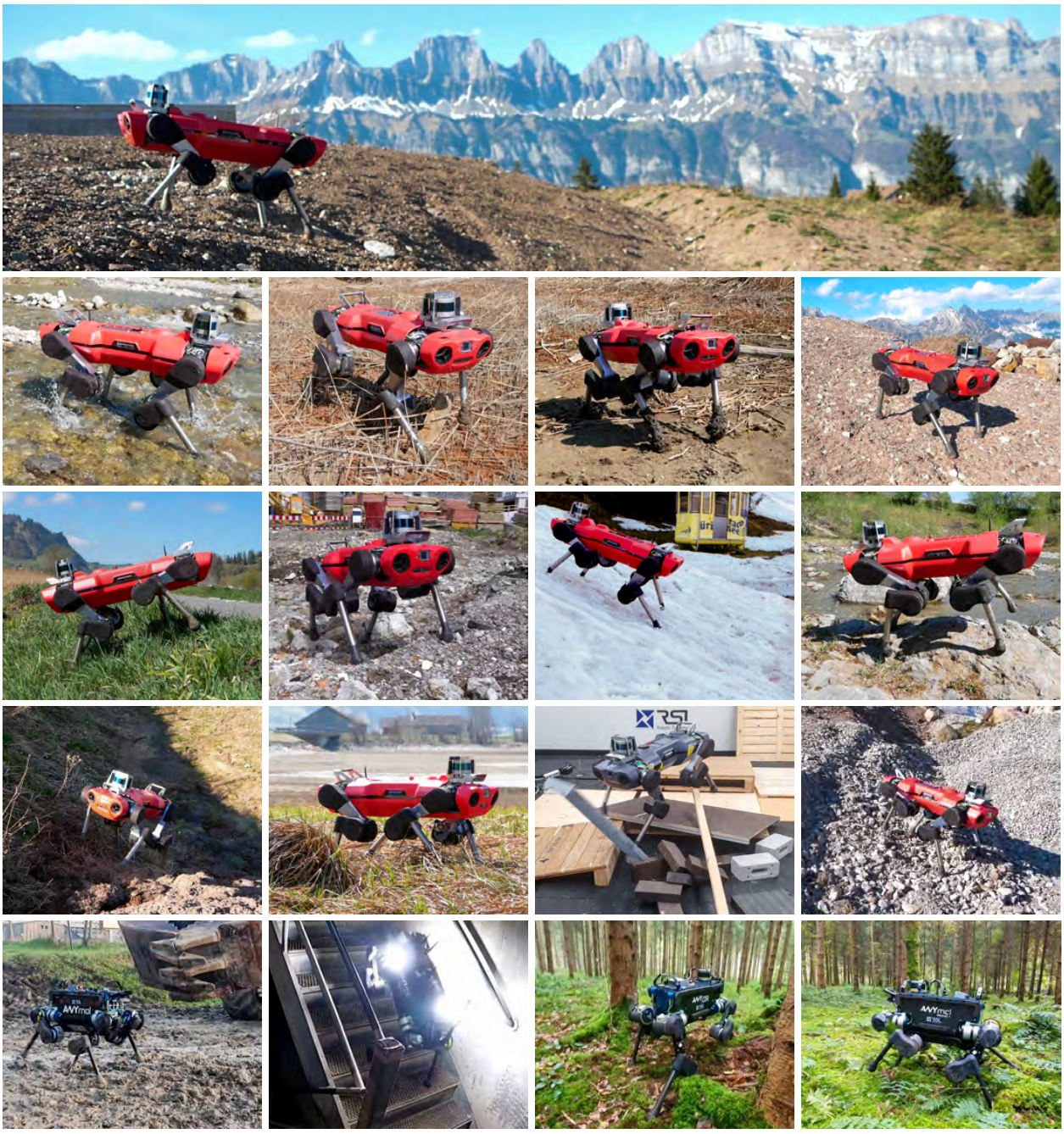


Figure 4.5: Reprinted from [36].

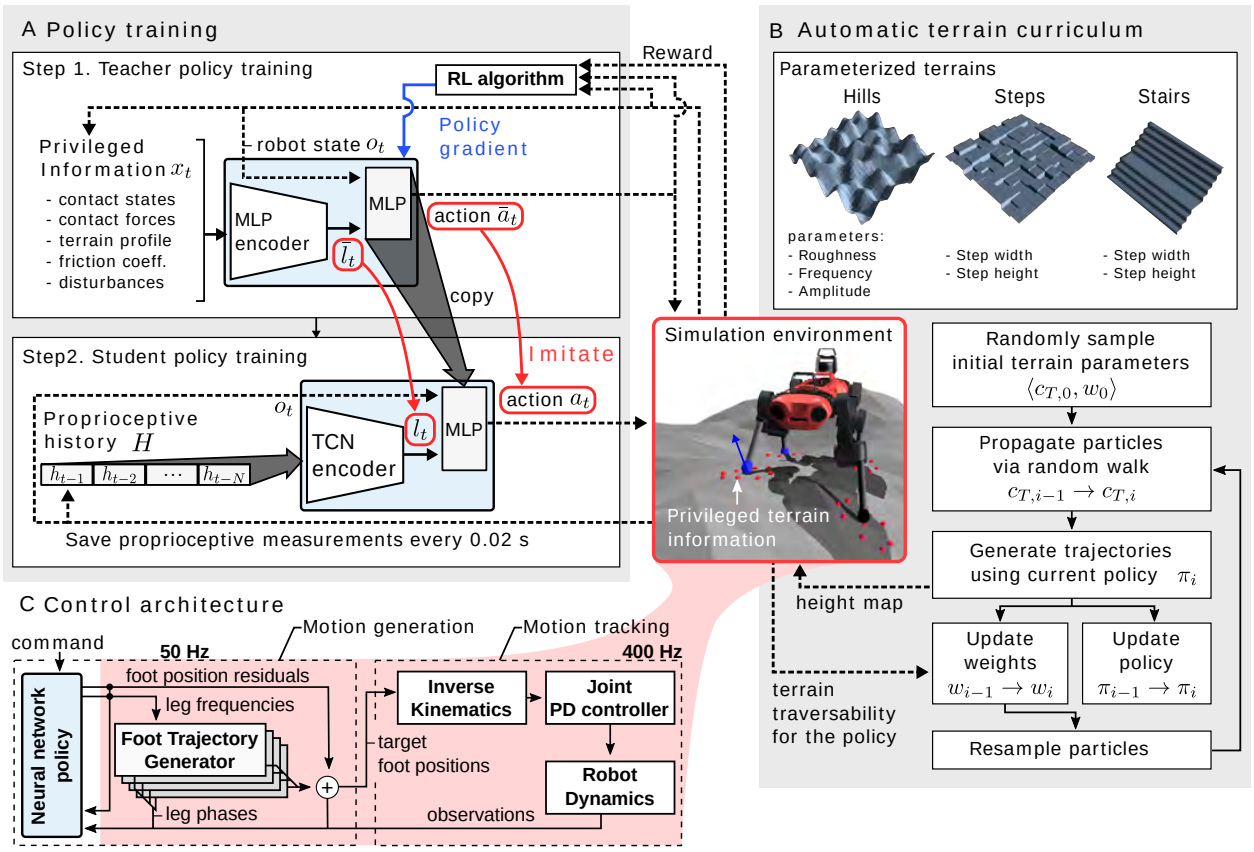


Figure 4.6: Reprinted from [36].

Appendix A

Proofs

A.1 Policy Gradient Theorem (Equation 3.9)

Proof. By definition (Equation 3.5), the expected return J_π is defined as

$$J_\pi = \mathbb{E}_\pi[G(\tau)] = \int_\tau G(\tau)\mathbb{P}_{\boldsymbol{\theta}}(\tau)\ d\tau.$$

Next, take the gradient with respect to the parameters $\boldsymbol{\theta}$ of the policy π to both sides of the equation.

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J_\pi &= \nabla_{\boldsymbol{\theta}} \int_\tau G(\tau)\mathbb{P}_{\boldsymbol{\theta}}(\tau)\ d\tau \\
&= \int_\tau G(\tau)\nabla_{\boldsymbol{\theta}}\mathbb{P}_{\boldsymbol{\theta}}(\tau)\ d\tau && \text{(by Leibniz integral rule)} \\
&= \int_\tau G(\tau)\mathbb{P}_{\boldsymbol{\theta}}(\tau)\nabla_{\boldsymbol{\theta}} \log \mathbb{P}_{\boldsymbol{\theta}}(\tau)\ d\tau && (\partial \log f(x) = \tfrac{1}{f(x)}\partial f(x)) \\
&= \mathbb{E}_{\boldsymbol{\theta}}\left[G(\tau)\nabla_{\boldsymbol{\theta}} \log \mathbb{P}_{\boldsymbol{\theta}}(\tau)\right] && \text{(by the definition of expectation)} \\
&= \mathbb{E}_{\boldsymbol{\theta}}\left[G(\tau)\nabla_{\boldsymbol{\theta}} \log \left(\prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(a_t|s_t)\right)\right] && \text{(deconstructing probability of } \tau) \\
\nabla_{\boldsymbol{\theta}} J_\pi &= \mathbb{E}_{\boldsymbol{\theta}}\left[G(\tau)\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right]
\end{aligned}$$

□

Glossary

ANN artificial neural network. 9

batch . 17

bias Parameters that add/offset to values in a model. Adds to a value add a node in neural networks. Not to be confused with *model bias*. 7

error surface The surface that a loss function plots in parameter space. 7

feature . 7

hyperparameter Values that specify aspects of a model's inherent architecture or its learning. 9

label The identifiers attached to training data that function as an "answer key" for what it is supposed to be. 7

loss function Function defined to measure the performance of a model, lower is better. 7

model bias Inherent limitations of a model's performance to its architecture. Not to be confused with *bias*. 7

MSE mean square error. 7

NN neural network. 9

node . 9

ReLU rectified linear unit. 11

weight Parameters that are direct coefficients to values in a model. Determines strength of connections between nodes in neural networks. 6

Bibliography

[1] Momin Jamil and Xin-She Yang. "A literature survey of benchmark functions for global optimisation problems". In: *International Journal of Mathematical Modelling and Numerical Optimisation* 4.2 (2013), pp. 150–194. DOI: 10.1504/IJMMNO.2013.055204. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJMMNO.2013.055204.

[2] Wikimedia Commons. *File:Colored neural network.svg — Wikimedia Commons, the free media repository*. 2025. URL: https://commons.wikimedia.org/w/index.php?title=File:Colored_neural_network.svg&oldid=995727191.

[3] Andrea Apicella et al. "A survey on modern trainable activation functions". In: *Neural Networks* 138 (2021), pp. 14–32. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2021.01.026. URL: https://www.sciencedirect.com/science/article/pii/S0893608021000344.

[4] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. _eprint: 1811.03378. 2018. URL: https://arxiv.org/abs/1811.03378.

[5] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. "Activation functions in deep learning: A comprehensive survey and benchmark". In: *Neurocomputing* 503 (2022), pp. 92–108. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2022.06.111. URL: https://www.sciencedirect.com/science/article/pii/S0925231222008426.

[6] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1, 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[7] Hung-Yi Lee. "Introduction of Machine / Deep Learning". 2021. URL: https://speech.ee.ntu.edu.tw/~hylee/ml/ml2021-course-data/regression%20(v16).pdf.

[8] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[9] Patrick Kidger and Terry Lyons. "Universal Approximation with Deep Narrow Networks". In: *Proceedings of Thirty Third Conference on Learning Theory*. Ed. by Jacob Abernethy and Shivani Agarwal. Vol. 125. Proceedings of Machine Learning Research. PMLR, July 9, 2020, pp. 2306–2327. URL: https://proceedings.mlr.press/v125/kidger20a.html.

[10] Allan Pinkus. "Approximation theory of the MLP model in neural networks". In: *Acta Numerica* 8 (1999), pp. 143–195. DOI: 10.1017/S0962492900002919.

[11] Michael Nielsen. *Neural Networks and Deep Learning*. Dec. 2019. URL: https://neuralnetworksanddeeplearning.com/.

[12] Grant Sanderson. "Backpropagation calculus — DL4". Nov. 3, 2017. URL: https://www.youtube.com/watch?v=tIeHLnjs5U8.

[13] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

[14] Wikimedia Commons. *File:K-fold cross validation EN.svg — Wikimedia Commons, the free media repository*. 2024. URL: https://commons.wikimedia.org/w/index.php?title=File:K-fold_cross_validation_EN.svg&oldid=932198002.

[15] Yurii Nesterov. "A method for unconstrained convex minimization problem with the rate of convergence O (1/k2)". In: *Dokl. Akad. Nauk. SSSR*. Vol. 269. Issue: 3. 1983, p. 543.

[16] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. _eprint: 1412.6980. 2017. URL: https://arxiv.org/abs/1412.6980.

[17] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. _eprint: 1502.03167. 2015. URL: https://arxiv.org/abs/1502.03167.

[18] Sergey Ioffe. *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*. _eprint: 1702.03275. 2017. URL: https://arxiv.org/abs/1702.03275.

[19] Shibani Santurkar et al. *How Does Batch Normalization Help Optimization?* _eprint: 1805.11604. 2019. URL: https://arxiv.org/abs/1805.11604.

[20] Christopher M Bishop and Hugh Bishop. *Deep learning: Foundations and concepts*. Springer Nature, 2023. URL: https://www.bishopbook.com/.

[21] Changil Moon. *General Overview for Computer Vision*. May 2025. DOI: 10.5281/zenodo.15420905. URL: https://doi.org/10.5281/zenodo.15420905.

[22] Hung-Yi Lee. "Convolutional Neural Network (CNN)". 2021. URL: https://speech.ee.ntu.edu.tw/~hylee/ml/ml2021-course-data/cnn_v4.pdf.

[23] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. _eprint: 1409.1556. 2015. URL: https://arxiv.org/abs/1409.1556.

[24] Wikimedia Commons. *File:Markov diagram v2.svg — Wikimedia Commons, the free media repository*. 2023. URL: https://commons.wikimedia.org/w/index.php?title=File:Markov_diagram_v2.svg&oldid=757108625.

[25] David Silver. *Lectures on Reinforcement Learning*. 2015. URL: https://www.davidsilver.uk/teaching/.

[26] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (May 1, 1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[27] John Schulman et al. "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei.

Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 7, 2015, pp. 1889–1897. URL: https://proceedings.mlr.press/v37/schulman15.html.

[28] John Schulman et al. *Proximal Policy Optimization Algorithms*. _eprint: 1707.06347. 2017. URL: https://arxiv.org/abs/1707.06347.

[29] Michał Kempka et al. *ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning*. _eprint: 1605.02097. 2016. URL: https://arxiv.org/abs/1605.02097.

[30] Deepak Pathak et al. *Curiosity-driven Exploration by Self-supervised Prediction*. _eprint: 1705.05363. 2017. URL: https://arxiv.org/abs/1705.05363.

[31] Sanmit Narvekar et al. *Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey*. _eprint: 2003.04960. 2020. URL: https://arxiv.org/abs/2003.04960.

[32] Saurabh Arora and Prashant Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress". In: *Artificial Intelligence* 297 (2021), p. 103500. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103500. URL: https://www.sciencedirect.com/science/article/pii/S0004370221000515.

[33] Marco Hutter et al. "ANYmal - a highly mobile and dynamic quadrupedal robot". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 38–44. DOI: 10.1109/IROS.2016.7758092.

[34] Marco Hutter. "Legged Robots on the way from subterranean". 2022 IEEE International Conference on Robotics and Automation (ICRA), May 24, 2022. URL: https://www.youtube.com/watch?v=XwheB2_dyMQ.

[35] Jemin Hwangbo et al. "Learning agile and dynamic motor skills for legged robots". In: *Science Robotics* 4.26 (Jan. 2019). Publisher: American Association for the Advancement of Science (AAAS). ISSN: 2470-9476. DOI: 10.1126/scirobotics.aau5872. URL: http://dx.doi.org/10.1126/scirobotics.aau5872.

[36] Joonho Lee et al. "Learning quadrupedal locomotion over challenging terrain". In: *Science Robotics* 5.47 (Oct. 2020). Publisher: American Association for the Advancement of Science (AAAS). ISSN: 2470-9476. DOI: 10.1126/scirobotics.abc5986. URL: http://dx.doi.org/10.1126/scirobotics.abc5986.

[37] Shuzhen Luo et al. "Reinforcement Learning and Control of a Lower Extremity Exoskeleton for Squat Assistance". In: *Frontiers in Robotics and AI* Volume 8 - 2021 (2021). ISSN: 2296-9144. DOI: 10.3389/frobt.2021.702845. URL: https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2021.702845.

[38] Shuzhen Luo et al. "Experiment-free exoskeleton assistance via learning in simulation". In: *Nature* 630.8016 (June 1, 2024), pp. 353–359. ISSN: 1476-4687. DOI: 10.1038/s41586-024-07382-4. URL: https://doi.org/10.1038/s41586-024-07382-4.

[39] Liyuan Liu et al. *On the Variance of the Adaptive Learning Rate and Beyond*. _eprint: 1908.03265. 2021. URL: https://arxiv.org/abs/1908.03265.

[40] Hao Li et al. *Visualizing the Loss Landscape of Neural Nets*. _eprint: 1712.09913. 2018. URL: https://arxiv.org/abs/1712.09913.

[41] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. _eprint: 1710.02298. 2017. URL: https://arxiv.org/abs/1710.02298.

[42] Anna Choromanska et al. *The Loss Surfaces of Multilayer Networks*. _eprint: 1412.0233. 2015. URL: https://arxiv.org/abs/1412.0233.

[43] Chiyuan Zhang et al. *Understanding deep learning requires rethinking generalization*. _eprint: 1611.03530. 2017. URL: https://arxiv.org/abs/1611.03530.

[44] Devansh Arpit et al. *A Closer Look at Memorization in Deep Networks*. _eprint: 1706.05394. 2017. URL: https://arxiv.org/abs/1706.05394.

[45] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. _eprint: 1609.04747. 2017. URL: https://arxiv.org/abs/1609.04747.

[46] Ashwin Rao. "Policy Gradient Algorithms". Stanford University. URL: https://web.stanford.edu/~ashlearn/RLForFinanceBook/PolicyGradient.pdf.

[47] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.