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 6, 2025

Abstract

something idk, figure out final formatting later

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.

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 Miscellaneous

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

$$\arg\max_{a} f(a), \tag{1.8}$$

or

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

respectively.

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.

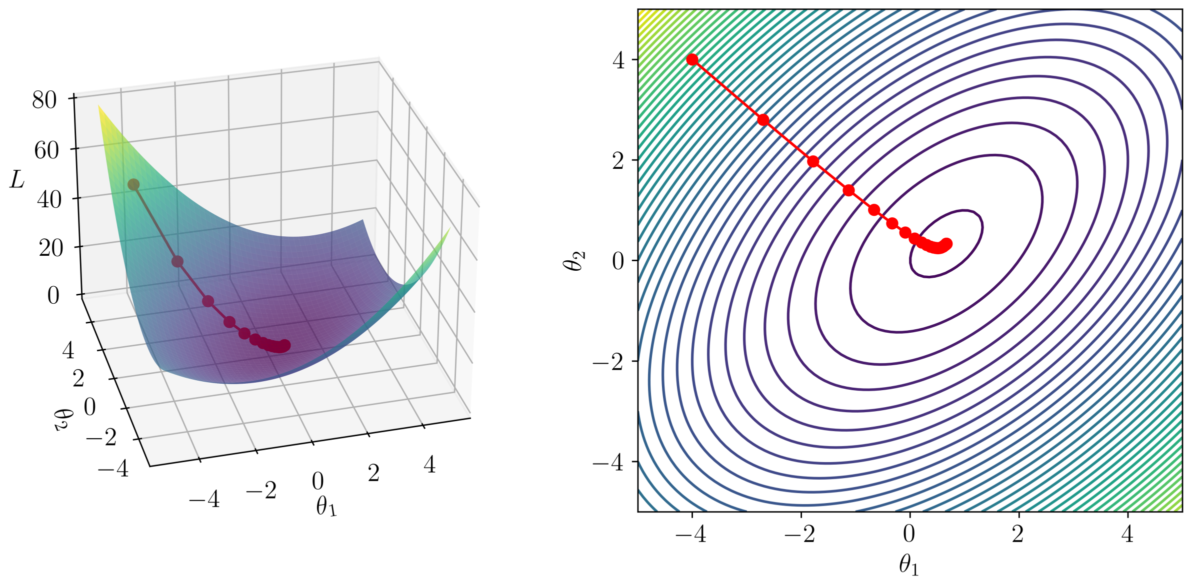

Figure 2.1: Visualization of naive gradient descent on an error surface with two parameters ($L = \theta_1^2 + \theta_2^2 - \theta_1\theta_2 - \theta_1$) by a 3D plot (left), and a contour plot (right). The red line shows 100 iterations of gradient descent, starting from intial parameters $\boldsymbol{\theta}^{(0)} = [-4, 4]$, with $\eta = 0.1$. (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**. NNs are usually fully connected, meaning that each node in a layer is considered by every node in the next layer.

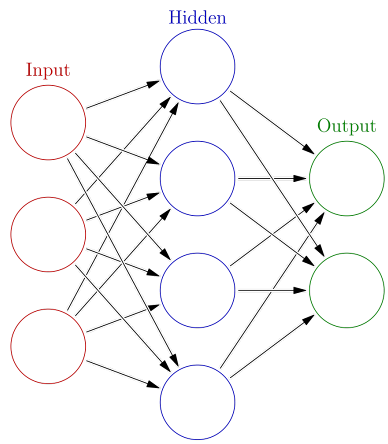

Figure 2.2: A schematic of a neural network with a input layer with three nodes, one hidden layer with four nodes, and an output layer with two nodes [1].

Each node takes a biased weighted sum of the values of the previous layer, applies an activation 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 and the Universal Approximation Theorems

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.

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

Another very common activation function is the **rectified linear unit (ReLU)**, which has the advantage of being computationally easier to calculate.

$$\text{ReLU}(x) = \max(0, x) \tag{2.10}$$

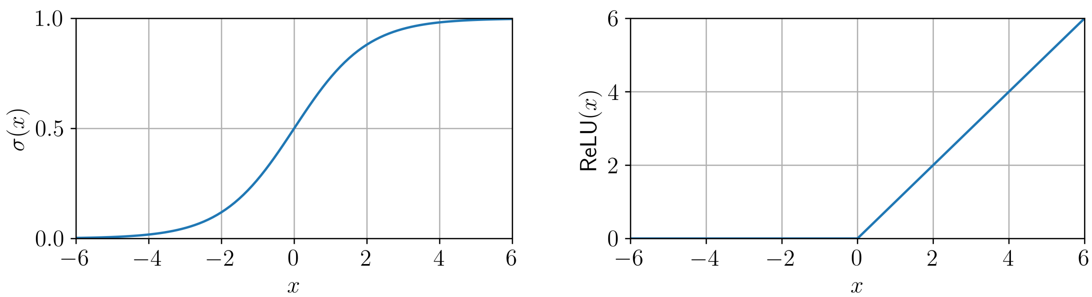

Figure 2.3: Plot of $\sigma(x)$ (left) and ReLU(x) (right). (Original)

Any continuous relationship between two variables can be approximated by a *sufficiently large* linear combination of sigmoid functions [2].

$$f \approx b + \mathbf{c} \cdot \sigma(\mathbf{b} + \mathbf{W}\mathbf{x}). \tag{2.11}$$

Equation 2.11 is equivalent to a neural network with one hidden layer and scalar output (omitting the final application of an activation function).

This result can be generalized to any non-polynomial activation function, including ReLUs, in the **universal approximation theorems**, implying that NNs that use such activation functions may theoretically learn any relationship given *sufficiently large* amounts nodes and layers [4, 5, 6]. It is important to note that this result is only theoretical, and in practice, computational and data constraints usually frustrate this ideal.

2.2.2 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.

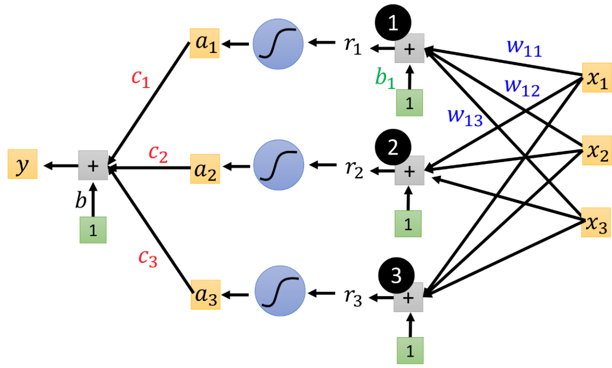

Figure 2.4: Reprinted from [3].

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.

Recall Equation 2.7,

$$a_j^{(l)} = \phi^{(l)} \left(b_j^{(l)} + \sum_i w_{ij}^{(l)} a_i^{(l-1)} \right). \tag{2.12}$$

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.

$$z_j^{(l)} = b_j^{(l)} + \sum_i w_{ij}^{(l)} a_i^{(l-1)} \tag{2.13}$$

Then, we can rewrite the equation as

$$a_j^{(l)} = \phi^{(l)}(z_j^{(l)}). \tag{2.14}$$

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.

$$C = \sum_j \left(a_j^{(L)} - y_j\right)^2 \tag{2.15}$$

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,

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}}, \tag{2.16}$$

and

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}. \tag{2.17}$$

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

$$\frac{\partial C}{\partial a_j^{(L)}} \underbrace{\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_j^{(L-1)}}}_{\text{output layer } L} \underbrace{\frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial z_j^{(L-1)}}{\partial a_j^{(L-2)}}}_{\text{layer } L-1} \cdots \underbrace{\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \left(\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \text{ or } \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} \right)}_{\text{layer } l} \tag{2.18}$$

Note the following:

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \phi^{(L)\prime}(z_j^{(L)}), \tag{2.19}$$

$$\frac{\partial z_j^{(l)}}{\partial a_j^{(l-1)}} = \sum_j w_{ij}^{(l)}, \tag{2.20}$$

$$\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}, \tag{2.21}$$

$$\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = 1. \tag{2.22}$$

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,

$$\nabla_{a^{(L)}} C = \left[\frac{\partial C}{\partial a_j^{(L)}} \right]. \tag{2.23}$$

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 $z^{(l)}$,

$$\delta^{(l)} = \begin{bmatrix} \frac{\partial C}{\partial z_1^{(l)}} \\ \vdots \end{bmatrix}. \tag{2.24}$$

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

$$\delta^{(L)} = \nabla_{a^{(L)}} C \odot \phi'(z^{(L)}), \tag{2.25}$$

$$\delta^{(L-1)} = \left(\mathbf{W}^{(L)}\right)^T \delta^{(L)} \odot \phi'(z^{(L-1)}). \tag{2.26}$$

Using Equation 2.13 and Equation 2.14, we can compute these terms as follows:

$$\frac{\partial C}{\partial a_j^{(L)}} = 2(a_j^{(L)} - y_j) \tag{2.27}$$

Substituting these terms into the equations, we have

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = 2(a_j^{(L)} - y_j)\phi'(z_j^{(L)})a_i^{(L-1)}. \tag{2.28}$$

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

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.

2.4.1 Cross-validation

2.4.2 Regularization

2.4.3 Dropout

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.

put a figure here or smth

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.

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

$$\mathbf{v}^{(t)} = \gamma \mathbf{v}^{(t-1)} - \eta \nabla L(\boldsymbol{\theta}^{(t)}), \tag{2.29}$$

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

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t)}. \tag{2.30}$$

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

put a figure here or smth

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

$$\mathbf{v}^{(n)} = \sum_{i=0}^{n} -\gamma^{n-i} \eta \nabla L(\boldsymbol{\theta}^{(i)}) = -\eta \nabla L(\boldsymbol{\theta}^{(n)}) - \gamma \eta \nabla L(\boldsymbol{\theta}^{(n-1)}) - \gamma^2 \eta \nabla L(\boldsymbol{\theta}^{(n-2)}) - \cdots \tag{2.31}$$

Nesterov Momentum

Russian mathematician Yurii Nesterov improved on the standard momentum method by instead computing the gradient *after* first applying the intertial term.

$$\mathbf{v}^{(t)} = \gamma \mathbf{v}^{(t-1)} - \eta \nabla L(\boldsymbol{\theta}^{(t)} + \gamma \mathbf{v}^{(t-1)}). \tag{2.32}$$

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.

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sigma_i^t} \frac{\partial L}{\partial \theta_i^{(t)}} \tag{2.33}$$

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.

$$\sigma_i^t = \sqrt{\sum_{j=0}^{t} \left(\frac{\partial L}{\partial \theta_i^{(j)}} \right)^2 + \epsilon} \tag{2.34}$$

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.

$$\sigma_i^t = \sqrt{\alpha (\sigma_i^{(t-1)})^2 + (1-\alpha) \left(\frac{\partial L}{\partial \theta_i^{(t)}} \right)^2 + \epsilon} \tag{2.35}$$

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.

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**.

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta^{(t)}}{\sigma_i^t} \frac{\partial L}{\partial \theta_i^{(t)}} \tag{2.36}$$

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.

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i} \tag{2.37}$$

Or, with vectors,

$$\tilde{\mathbf{x}} = (\mathbf{x} - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma} \tag{2.38}$$

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 **1** and **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.

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.

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.39}$$

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 $\mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K$ and computes each component of vector $\sigma(\mathbf{z}) \in (0,1)^K$ with

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \,. \tag{2.40}$$

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

insert figure here

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.

2.8.2 Cross Entropy Loss

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

$$L = \frac{1}{N} \sum \left(\sum_i (\hat{y}_i - y'_i)^2 \right) \tag{2.41}$$

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`

$$L = \frac{1}{N} \sum \left(- \sum_i \hat{y}_i \ln y'_i \right) \tag{2.42}$$

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.

Chapter 3

Reinforcement Learning

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. 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.

insert 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.

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. Formally, an MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R)$, 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.
- P, the **transition function**, outputs the probability of transitioning from one state to another via a specific action.
- R, the **reward function**, outputs the reward received by the agent after transitioning from one state to another given a specific action.

insert figure here

3.1.1 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.

Equivalently, the policy can be thought of as a function that maps from the state space to the action space, $\pi : \mathcal{S} \to \mathcal{A}$. More precisely, the policy maps not to just a single action, but a probability distribution over the action space.

The goal of the agent is to learn a policy that maximizes the expected reward over time.

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.

3.2 Policy Gradient

3.2.1 Value Functions

3.3 Actor-Critic Methods

3.4 Reward Shaping

3.5 Inverse Reinforcement Learning (IRL)

Chapter 4

Robotic Systems and Control

4.1 Hutter, Lee, Hwangbo et al. and the ANYmal by ANYbotics

[7]

4.2 Luo et al. and Exoskeletons

4.3 more examples will go here

Glossary

ANN artificial neural network. 8

batch . 13

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*. 6

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

feature . 6

hyperparameter Values that specify aspects of a model's inherent architecture or its learning. 8

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. 8

node . 8

ReLU rectified linear unit. 10

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

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.