A SURVEY OF DISCRETE MATHEMATICS IN MACHINE LEARNING

SAMUEL DORCHUCK, AMY KIM, AND ABIGAIL MOSER

Abstract

The fundamental objective of this paper is to provide a survey of discrete mathematical concepts used in various Machine Learning algorithms, with a wide range of applications. To meet that objective, this paper will review fundamental techniques such as Markov chains and graph searching, as well as introduce advanced concepts such as VC-Dimension, Epsilon-Nets, and Hidden Markov Models. The reader should expect to encounter intuitively described proofs, digestible yet detailed algorithmic descriptions and complexity analysis, and closed-form expressions and recursive formulas.

Contents

(1) Introduction .... *Page 1*
(2) Linear Classifiers & The Perceptron .... *Page 2*
(3) Neural Networks: Error-Back Propagation & Gradient Descent . *Page 8*
(4) VC Dimension & Epsilon Nets .... *Page 13*
(5) Markov Decision Processes .... *Page 19*
(6) Hidden Markov Models .... *Page 24*

1. Introduction

A burgeoning scientific field with numerous applications across countless domains, Machine Learning is most commonly perceived as a powerful but opaque black-box tool. At its core, however, Machine Learning is math. **The fundamentals of Machine Learning are deeply rooted in discrete mathematics.** Familiar concepts such as Markov Models, probability theory, graph searching, and discretization of continuous functions appear repeatedly in the algorithms that power the modern revolution of Machine Learning. Throughout this paper, we will demystify some of these algorithms by analyzing several key mathematical concepts.

For the purposes of this paper, **we define Machine Learning as a field of study focused on developing algorithms that possess the ability to improve their performance through experience or data.** This paper will focus on two fundamental types of Machine Learning algorithms: (1) predictive algorithms and (2) decision-making processes. Predictive algorithms attempt to predict future outcomes with a model developed and refined using data, and are often seen in contexts such as financial forecasting or risk analysis. Decision-making processes involve an agent with a goal, outlining a way for the agent to "learn" how best to operate by receiving feedback from its environment, and can be applied to anything

from self-driving cars to Natural Language Processing.

The first half of this paper will focus on predictive algorithms. Beginning with one of the oldest examples of Machine Learning, we will examine the Perceptron in Section 2 to understand the process of classification. Here, we will prove that the Perceptron Algorithm converges under all circumstances, and we will discuss the running time of convergence in terms of the input data set. In Section 3, we will turn to the modern tool of choice for prediction - neural networks. This section will rigorously examine the gradient descent and error-back propagation algorithms, which have been largely responsible for the tremendous growth in importance and use of neural networks in the last two decades. Gradient Descent allows us to calculate the weights of neural network that minimize the loss function of the network, provided we are given a gradient of the loss for each weight. Error Back-Propagation allows us to efficiently compute the gradients of the loss function for each weight in the neural network. Section 4 will introduce the VC-dimension, a discrete-mathematical concept used to analyze the complexity of these prediction algorithms.

In the second half of the paper, we turn to decision-making processes and examine methods which improve a machine's ability to autonomously interact with its environment. In Section 5, we will explore the mathematical foundations of Markov Decision Processes and the closely related Value Iteration Algorithm, to illustrate how computers optimize decisions in a given environment. In particular, the Value Iteration Algorithm allows us to find an decision-making strategy that optimizes our rewards in an environment described by a Markov Decision Process. Lastly, we turn to Hidden Markov Models in Section 6, to examine how computers can begin to understand and interact with the world when they can only partially observe the details of their environment.

Throughout this paper, our central goal is to eliminate the common misconception of Machine Learning's complexity by breaking each algorithm down to its foundational mathematical roots. At its core, this paper emphasizes that mathematics is the language by which we explain the world around us, a concept that persists even into the digital world.

2. Linear Classifiers & The Perceptron

2.1. **Binary Linear Classifiers [1].** A key capability of Machine Learning is the ability to learn how to classify an object. When we classify something, we use its characteristics to identify which class it belongs to. For example, given a photograph of a pet, we can create a *classifier* to determine whether the photo is of a dog, or of a cat. Generally, the objects we hope to classify might be photographs, emails, or even songs, which can be hard to define numerically. In order to make these problems tractable, we will represent the characteristics of a given object as an input vector, $\boldsymbol{x} \in \mathbb{R}^d$, and we will represent the resulting class as an output $y \in \mathbb{R}$. Broadly, a classifier is a process or algorithm that, given input $\boldsymbol{x}$, predicts the resulting class y. While classification can extend to multiple classes, this paper will focus on *binary classifiers* wherein $y \in \{-1, 1\}$.

Classifiers come in many forms, but perhaps the simplest to understand, and the most critical for a strong understanding of Machine Learning, is the linear classifier, which will be the focus of this section. Linear classifiers are relatively powerful alone, and form the basis for a multiplicity of more advanced Machine Learning methods.

Definition 2.1. A *linear classifier* is characterized by a vector $\boldsymbol{\theta} \in \mathbb{R}^d$ and a scalar offset $\theta_0 \in \mathbb{R}$. A *binary linear classifier* $\boldsymbol{\theta}, \theta_0$ maps an input vector $\boldsymbol{x} \in \mathbb{R}^d$ to a predicted output value $y_p \in \{-1, 1\}$ by taking a linear combination of the elements of $\boldsymbol{x}$:

$$y_p = \text{sign}(\boldsymbol{\theta}^T \boldsymbol{x} + \theta_0) \tag{2.1}$$

We can think of $\boldsymbol{\theta}, \theta_0$ as a hyperplane dividing $\mathbb{R}^d$ into two half-spaces. Here we see the essence of classification by a binary linear classifier: the half space on the same side of $\boldsymbol{\theta}, \theta_0$ as the normal vector is the positive half space, where the positively classified points reside, while the half space on the opposite side is negative, and the points there are classified as negative. Combining this notion with the above equation, an input $\boldsymbol{x}$ for which $\boldsymbol{\theta}^T \boldsymbol{x} + \theta_0 > 0$ is classified as positive, otherwise $\boldsymbol{x}$ is classified as negative.

In classification problems, we are generally given a set of data points

$$D_n = \{(\boldsymbol{x}^{(1)}, y^{(1)}), ..., (\boldsymbol{x}^{(n)}, y^{(n)})\}$$

where each $\boldsymbol{x}^{(i)}$ is the vector representation of the characteristics of the ith data point, and $y^{(i)}$ is the **true** classification of the object. Our goal will be to find a binary linear classifier $\boldsymbol{\theta}, \theta_0$ that does the best job at classifying the data points correctly, which we can use to classify points that we haven't seen before. This process of finding a good classifier based on our data is called *training*.

Having described our classification problem, one might wonder how we define success for a classifier. We've said that we want to find a classifier that does the "best job" at classifying the data points correctly – we will now formalize this concept.

Definition 2.2. Given a dataset D_n, the *training error* $E_{\boldsymbol{\theta},\theta_0}$ of a classifier $\boldsymbol{\theta}, \theta_0$ is defined as

$$E_{\boldsymbol{\theta},\theta_0} = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 \text{ if } \text{sign}(\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} + \theta_0) = y_p^{(i)} \neq y^{(i)} \\ 0 \text{ else} \end{cases} . \tag{2.2}$$

Observe that the training error on an individual data point is equal to 1 when it is incorrectly classified by the linear classifier, and 0 when it is correctly classified by the linear classifier. Thus, the training error for a linear classifier represents the average number of classification errors over the entire dataset. The best classifier for a given dataset, and the classifier that we want to find, will be the classifier that minimizes the training error.

Now that we have an understanding of the general process of training and the criteria by which we evaluate a classifier for a given dataset, we can introduce a simple algorithm which, under some conditions, will reliably find a good classifier.

2.2. **The Perceptron Algorithm [3].** Although the Perceptron is largely obsolete in modern Machine Learning, it lays the groundwork for the kind of mathematical reasoning used in Machine Learning, and is often considered one of the first examples of a Machine Learning algorithm. The following pseudocode outlines the Perceptron algorithm, run on the dataset D_n.

Algorithm 1 Perceptron: Find $\boldsymbol{\theta}$ and θ_0 in τ iterations

```
θ = [00...0]^T
θ_0 = 0
for t = 1 to τ do
  for i = 1 to n do
    if y^(i)(θ^T x^(i) + θ_0) ≤ 0 then
      θ = θ + y^(i) x^(i)
      θ_0 = θ_0 + y^(i)
    end if
  end for
end for
return θ, θ_0
```

The Perceptron algorithm runs for some predetermined number of iterations τ. On each iteration, the algorithm runs through each point in the dataset, and checks if the current linear classifier $\boldsymbol{\theta}, \theta_0$ correctly classifies object i. If the point is classified correctly, then no change is made to the current classifier. However, if object i is misclassified, then $\boldsymbol{\theta}, \theta_0$ are shifted such that they are closer to correctly classifying object i. The algorithm terminates either after τ iterations, or if none of the points are misclassified by a given classifier (i.e. if we find a good separator before the τth iteration).

It turns out that the Perceptron can produce the best possible classifier for a dataset in a finite amount of time, as long as the dataset *can* be separated at all.

Definition 2.3. A dataset D_n is *linearly separable* if there exists some classifier $\boldsymbol{\theta}^*, \theta_0^*$ such that

$$y^{(i)}(\boldsymbol{\theta}^{*T}\boldsymbol{x}^{(i)} + \theta_0^*) > 0 \;\forall 1 \leq i \leq n \tag{2.3}$$

i.e. all predictions made by the classifier on the training set are correct. Such a classifier $\boldsymbol{\theta}^*, \theta_0^*$ is called a *perfect separator*.

Example 2.4. Consider the dataset $\{([1,1],1),[-1,-1],-1\}$ in the in the $\mathbb{R}^2$ plane. This dataset is clearly linearly separable − an example of a perfect separator is illustrated in Figure 1.

We claim that the Perceptron can return a perfect separator for a linearly separable dataset. We call this property *convergence*. It turns out, however, that due to the simplicity and power of the Perceptron algorithm, we can prove something stronger than convergence. We will show that under a certain set of conditions, we can find an upper bound on the number of iterations it takes for Perceptron to converge (produce a perfect separator).

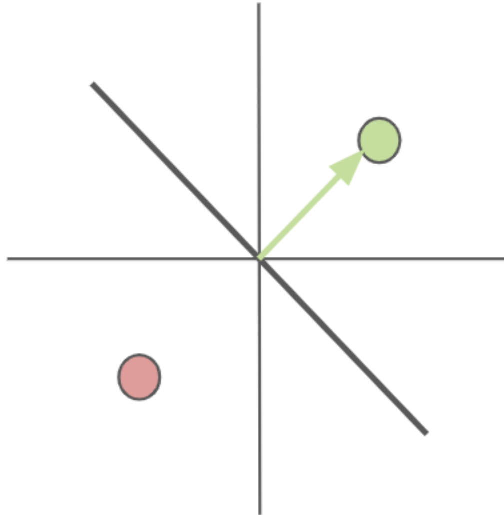

FIGURE 1. An example of a linearly separable dataset. Here, red points correspond to negative points (i.e. points such that $y = -1$), and green points correspond to positive points (i.e. points such that $y = 1$).

Before we can prove this property, called the Perceptron Convergence Theorem, we will define a term that allows us quantify **how well** a linear separator classifies a point.

Definition 2.5. The *margin* $\gamma^{(i)}$ of a data point $(\boldsymbol{x}^{(i)}, y^{(i)})$ with respect to a linear separator $\boldsymbol{\theta}, \theta_0$ is

$$\gamma^{(i)} = y^{(i)} \cdot \frac{\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} + \theta_0}{||\boldsymbol{\theta}||} \tag{2.4}$$

We use $||\boldsymbol{\theta}||$ here to represent the magnitude of the classifier. Note that the term $\frac{\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} + \theta_0}{||\boldsymbol{\theta}||}$ represents the distance from $\boldsymbol{x}^{(i)}$ (which can be visualized as a point in $\mathbb{R}^d$) to the classifier (which can be visualized as a hyperplane in $\mathbb{R}^d$). The margin then represents the positive distance from the point to the classifier if the point is classified correctly, and the negative distance if the point is classified incorrectly.

We can extend the concept of the margin of an individual point to the margin of an entire dataset.

Definition 2.6. The *margin* γ of the dataset D_n with respect to a linear separator $\boldsymbol{\theta}, \theta_0$ is defined as the minimum margin over every point in the dataset:

$$\gamma = \min_i \left(y^{(i)} \cdot \frac{\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} + \theta_0}{||\boldsymbol{\theta}||} \right) \tag{2.5}$$

Thus, only when all objects in the dataset are classified correctly will the margin of D_n be positive. In this case, the margin will represent the distance between the

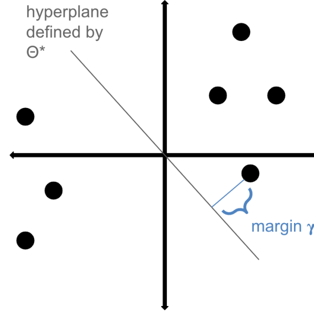

FIGURE 2. Minimum margin γ between the linear separator defined by $\boldsymbol{\theta}^*$ and all points

hyperplane and the closest point in the dataset, as illustrated in Figure 2. We are now ready to state the full theorem.

Theorem 2.7. *(Perceptron Convergence) Given data set D_n such that:*

1. *There exists some linear classifier $\boldsymbol{\theta}^*$ for which the margin of D_n with respect to $\boldsymbol{\theta}^*$ is $\gamma > 0$, and*
2. *$||\boldsymbol{x}^{(i)}|| \leq R \; \forall 1 \leq i \leq n$,*

the Perceptron algorithm will make at most $(\frac{R}{\gamma})^2$ mistakes before convergence.

Note that for the purpose of this theorem and the corresponding proof, we will incorporate the offset θ_0 into $\boldsymbol{\theta}^*$, such that the separator is $(d+1)$-dimensional.

Proof. Let $\boldsymbol{\theta}^{(k)}$ represent the separator after the perceptron algorithm has made k misclassifications. Note that the algorithm initializes $\boldsymbol{\theta}^{(0)} = 0$. We will show that the angle between the perfect separator $\boldsymbol{\theta}^*$ and $\boldsymbol{\theta}^{(k)}$, illustrated in Figure 3, decreases on every iteration of the perceptron algorithm until $\boldsymbol{\theta}^* = \boldsymbol{\theta}^{(n)}$ for some final step n.

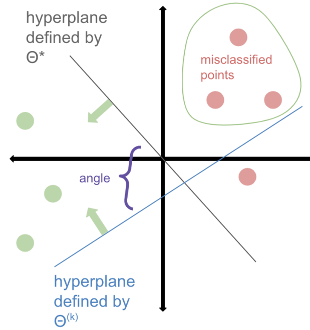

FIGURE 3. Angle between the perfect separator, $\boldsymbol{\theta}^*$, and the separator after k mistakes, $\boldsymbol{\theta}^{(k)}$

The cosine of the angle between $\boldsymbol{\theta}^*$ and $\boldsymbol{\theta}^{(k)}$, is defined as the dot product of the two vectors:

$$\begin{aligned}\cos(\boldsymbol{\theta}^*, \boldsymbol{\theta}^{(\boldsymbol{k})}) &= \frac{\boldsymbol{\theta}^{(\boldsymbol{k})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^{(\boldsymbol{k})}||||\boldsymbol{\theta}^*||} \\ &= \Big(\frac{\boldsymbol{\theta}^{(\boldsymbol{k})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||}\Big)\Big(\frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{k})}||}\Big) \qquad (2.6)\end{aligned}$$

The first term can be decomposed by splitting $\boldsymbol{\theta}^{(\boldsymbol{k})}$ into the previous separator $\boldsymbol{\theta}^{(\boldsymbol{k-1})}$ and $y^{(i)}\boldsymbol{x}^{(i)}$, as defined by the perceptron update step. This assumes that the kth mistake occurs on the ith data point, $\boldsymbol{x}^{(i)}$.

$$\begin{aligned}\frac{\boldsymbol{\theta}^{(\boldsymbol{k})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} &= \frac{(\boldsymbol{\theta}^{(\boldsymbol{k-1})} + y^{(i)}\boldsymbol{x}^{(i)}) \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} \\ &= \frac{\boldsymbol{\theta}^{(\boldsymbol{k-1})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} + \frac{y^{(i)}\boldsymbol{x}^{(i)} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} \\ &\geq \frac{\boldsymbol{\theta}^{(\boldsymbol{k-1})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} + \gamma\end{aligned}$$

The last step follows from the definition of the margin of a dataset. Now, notice that we can repeatedly apply this decomposition:

$$\begin{aligned}\frac{\boldsymbol{\theta}^{(\boldsymbol{k})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} &\geq \frac{\boldsymbol{\theta}^{(\boldsymbol{k-1})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} + \gamma \\ &\geq \frac{\boldsymbol{\theta}^{(\boldsymbol{k-2})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} + \gamma + \gamma \\ &\geq \frac{\boldsymbol{\theta}^{(\boldsymbol{1})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} + \gamma + ... + \gamma \\ \frac{\boldsymbol{\theta}^{(\boldsymbol{k})} \cdot \boldsymbol{\theta}^*}{||\boldsymbol{\theta}^*||} &\geq k\gamma \qquad (2.7)\end{aligned}$$

Using condition (2) of the theorem, we can similarly decompose the second term:

$$\begin{aligned}\frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{k})}||^2} &= \frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{k-1})} + y^{(i)}\boldsymbol{x}^{(i)}||^2} \\ &= \frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{k-1})}||^2 + 2y^{(i)}\boldsymbol{\theta}^{(\boldsymbol{k-1})}\boldsymbol{x}^{(i)} + ||\boldsymbol{x}^{(\boldsymbol{i})}||^2} \\ &\geq \frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{k-1})}||^2 + R^2} \\ &\geq \frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{1})}||^2 + R^2 + ... + R^2} \geq \frac{1}{kR^2} \\ \frac{1}{||\boldsymbol{\theta}^{(\boldsymbol{k})}||} &\geq \frac{1}{\sqrt{k}R}. \qquad (2.8)\end{aligned}$$

Finally, we can substitute the inequalities (2.8) and (2.7) into our original equation (2.6), using the fact that cosine can have a value no larger than 1. Further note that when the cosine between the two linear separators is equal to 1, the two align

with each other perfectly. Simplifying, we obtain

$$\cos(\boldsymbol{\theta}^*, \boldsymbol{\theta}^{(\boldsymbol{k})}) \geq k\gamma\Big(\frac{1}{\sqrt{k}R}\Big)$$
$$1 \geq \frac{\sqrt{k}\gamma}{R}$$
$$k \leq (\frac{R}{\gamma})^2.$$

Therefore, the Perceptron algorithm makes at most $k = (\frac{R}{\gamma})^2$ misclassifications before the angle between the Perceptron-yielded classifier and the perfect separator is such that the Perceptron-yielded classifier makes no mistakes. □

Now armed with the knowledge of linear classifiers and one method by which they are computed, one has the necessary background knowledge to launch into more complicated and powerful classification algorithms.

3. Neural Networks: Error Back Propagation & Gradient Descent

3.1. **Introduction to Neural Networks [2].** So far, we have been looking at the types of Machine Learning problems that can be solved using linear classifiers: given a set of dimensions, or variables, how can we construct a particular model that optimizes our predictive power? While incredibly important and very powerful, however, linear classifiers are limited by their functional form. Often, in Machine Learning, we are faced with problems where we do not know what our variables will be, or even what kind of model we might want to use. For these dynamic problems, neural networks, with their flexible structure, are an invaluable tool.

We begin by defining the components of a neural network.

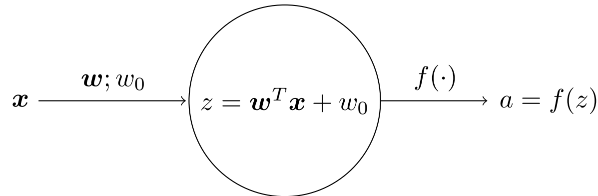

Figure 4. Diagram of a neuron with weights $\boldsymbol{w}$, offset w_0, and activation function f.

Definition 3.1. A *neuron* is a function that maps an input vector $\boldsymbol{x} \in \mathbb{R}^m$ to a single output a, as illustrated in Figure 4. We characterize a neuron by a vector of weights $\boldsymbol{w} \in \mathbb{R}^m$ and an offset $w_0 \in \mathbb{R}$, which is used to calculate the *pre-activation* $z = \boldsymbol{w}^T\boldsymbol{x} + w_0$. The pre-activation is then passed through an activation function $f : \mathbb{R} \to \mathbb{R}$, to produce the output $a = f(z)$. The output of the neuron can be written as:

$$a = f(\boldsymbol{w}^T\boldsymbol{x} + w_0) \tag{3.1}$$

Example 3.2. We've already seen one example of a neuron-like function – a binary linear classifier. Recall that in Section 2, we looked at binary linear classifiers in $\mathbb{R}^d$,

characterized by the model $y_p = \text{sign}(\theta^T x + \theta_0)$. Note that this closely resembles the structure of the neuron – in fact, if we choose weights $\boldsymbol{w} = \theta$, offset $w_0 = \theta_0$, and f to be the sign function, we have a neuron that replicates the behavior of a binary linear classifier.

A neuron describes a function that "learns" one piece of information about an input. As promised, however, neural networks are much more powerful than simple linear classifiers. As a first step, we can combine neurons to produce a vector of outputs, representing multiple pieces of information about an input.

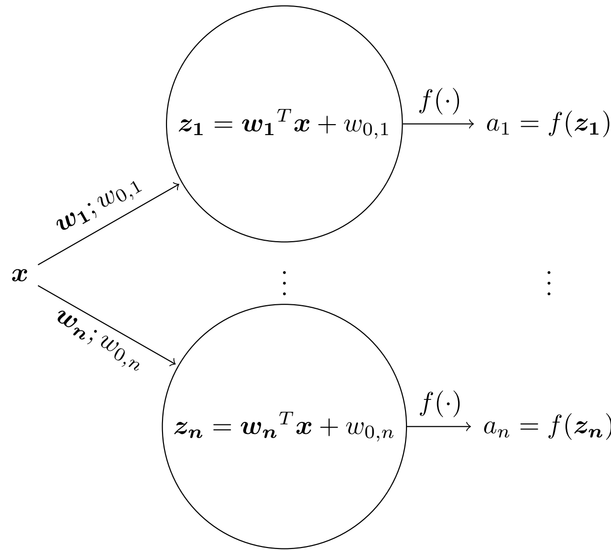

FIGURE 5. Diagram of a layer of a neural network with weights W, offset $\boldsymbol{w_0}$, and activation function f.

Definition 3.3. A *layer* of a neural network is a function that maps an input vector $\boldsymbol{x} \in \mathbb{R}^m$ to an output vector $\boldsymbol{a} \in \mathbb{R}^n$. We can visualize this as a combination of n neurons, as depicted in Figure 5. Accordingly, we can combine the weights $\boldsymbol{w_1}, \boldsymbol{w_2}, ..., \boldsymbol{w_n}$ of each neuron into a matrix $W \in \mathbb{R}^m \times \mathbb{R}^n$ and combine the offsets $w_{0,1}, w_{0,2}, ..., w_{0,n}$ of each neuron into a vector $\boldsymbol{w_0} \in \mathbb{R}^n$, to produce preactivation vector $\boldsymbol{z} = W^T\boldsymbol{x} + w_0$. We can then apply the activation function pointwise to each preactivation, to produce output vector $\boldsymbol{a}$. We can now denote the output of the layer as:

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{z_1}) \\ f(\boldsymbol{z_2}) \\ \vdots \\ f(\boldsymbol{z_n}) \end{bmatrix} = f\left(\begin{bmatrix} \boldsymbol{w_1}^T\boldsymbol{x} + w_0^1 \\ \boldsymbol{w_2}^T\boldsymbol{x} + w_0^2 \\ \vdots \\ \boldsymbol{w_n}^T\boldsymbol{x} + w_0^n \end{bmatrix} \right) = f(W^T\boldsymbol{x} + w_0) \tag{3.2}$$

Finally, we can put together multiple layers of neurons to form a neural network. This allows us to continually extrapolate information from our data, then extrapolate information from the results of the previous layer, resulting in a finely-tuned, extremely powerful network.

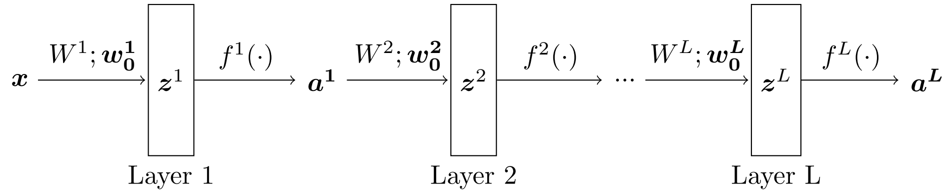

FIGURE 6. Diagram of a neural network with L layers.

Definition 3.4. A *neural network* is a connected series of layers, where we use the output $\boldsymbol{a}^l$ of layer l as the input to layer $l+1$. This is depicted visually in Figure 6. We specify the weights, offsets, and activation function for each layer with a superscript, so we say that layer l has input $\boldsymbol{a}^{l-1}$, weights W^l, offset $\boldsymbol{w_0}^l$, preactivation $\boldsymbol{z}^l$, and activation function f^l. We can now denote the output of layer l as:

$$\boldsymbol{a}^l = f^l(W^{lT}\boldsymbol{a}^{l-1} + \boldsymbol{w_0}^l) \tag{3.3}$$

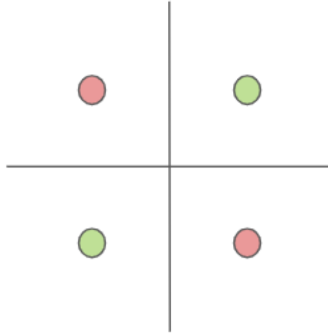

FIGURE 7. XOR Classification in $\mathbb{R}^2$.

Example 3.5. Let's look at an example of a slightly more challenging problem. The XOR classification problem, as illustrated in Figure 7, cannot be solved by a linear classifier in $\mathbb{R}^2$, but it *can* be solved using a simple neural network. Here, our input $\boldsymbol{x}$ is a point $[x_1, x_2] \in \mathbb{R}^2$. Qualitatively, XOR requires us to compare the signs of our two inputs. In a neural network, we can achieve this by extracting the sign of each input in the first layer, then comparing them in the second layer, providing a second level of analysis which is not possible with a linear classifier. This neural network is illustrated in Figure 8, and we can see that it produces output

$$a^2 = |z_1^2| = |\text{sign}(x_1) - \text{sign}(x_2)|$$

This output will be equal to 0 if x_1 and x_2 have the same sign, and 1 if they have different signs, as required.

Now that we have defined this new type of model, we need to determine a way to set the weights which correspond to each layer of the network. Note that the

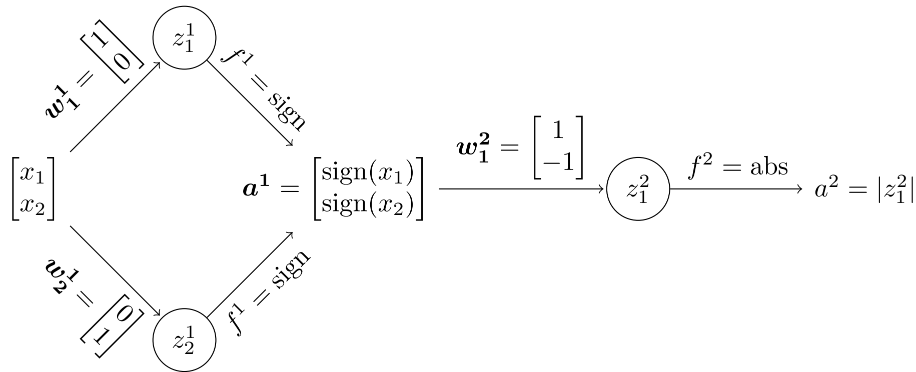

Figure 8. A neural network for the XOR classification problem in $\mathbb{R}^2$. Note that all offsets are set to zero.

activation function also plays a key role - we will not discuss choosing an activation function here, but the choice will depend on the type of data we have, and the problem we are trying to solve. For example, logarithmic activation functions tend to be used when we are seeking to output a probability. In order to introduce an algorithm for determining the optimal weights, we will first look at optimization techniques.

3.2. **Gradient Descent [2].** Recall that the goal of our neural network is to output a prediction given our input data. In order to evaluate how "good" our prediction is, we need some way of measuring the penalty, or the "loss" of an incorrect prediction.

Definition 3.6. A *loss function* $L(y, a)$ describes how much we training error is incurred for predicting output a, when the actual output should have been y. Then, a neural network is considered optimal if it has a minimized total loss summed over all data points in the data set.

We want to train a model that minimizes this loss. This is fairly easy when our loss function is simple, and our data has very few dimensions: we know from multivariable calculus that a function can be minimized or maximized over several variables by taking first order derivatives and setting them to zero. As our datasets get larger, however, we run into problems with this technique. Taking partial derivatives of the loss function for each data point in a large dataset is incredibly computationally intensive, and additionally, it is often the case that our loss function is too complicated for us to actually calculate the derivative. We introduce a popular method to find the minimum point of a multi-dimensional function, called Gradient Descent, which discretizes the continuous problem of optimization, and makes it computationally tractable.

Below, we describe the pseudo-code for the Gradient Descent Algorithm. In short, the Gradient Descent Algorithm optimizes the weights $\boldsymbol{w}$ of a Neural Network by repeatedly taking steps in the opposite direction of the gradient of the loss function. The goal of Gradient Descent is to find the weights of a Neural Network that minimize the loss function over some data set.

Algorithm 2 Gradient Descent: Determining parameters $\boldsymbol{w}$ to minimize loss $L(\boldsymbol{w}, \boldsymbol{y})$ with initial value $\boldsymbol{w}_{init}$, step-size η, and accuracy parameter ϵ

$\boldsymbol{w}_0 = \boldsymbol{w}_{init}$
$t = 0$
while $|L(\boldsymbol{w}_t, y) - L(\boldsymbol{w}_{t-1}, y)| > \epsilon$ **do**
  $t = t + 1$
  $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_w L$
end while
return $\boldsymbol{w}_t$

Intuitively, at every timestep of the Gradient Descent algorithm, we are stepping in the direction of greatest decrease in the loss function, until we get close enough to the minimum. The step size parameter η controls how big of a step we take, and the accuracy parameter ϵ controls how close we want to get to the true minimum. When η is too large, it is possible to overstep the true minimum, and when it is too large, it is possible to never reach the minimum, so this must be a carefully chosen value. We will not discuss how to choose η in this paper, but this is useful intuition to understand the nuances lurking in this algorithm. Epsilon can be thought of as a margin of error, and is up to the algorithm user to determine the level of error they are willing to tolerate. Unfortunately, we still have to calculate the gradient of the loss function at each step, $\nabla_w L$, which is a vector of the partial derivatives of L with respect to each component of $\boldsymbol{w}$.

Note, however, that since this algorithm only evaluates the gradient at a specific point, we can easily approximate the gradient local to that point, which is incredibly useful if we have a complicated loss function. Furthermore, if we have a large dataset, we can use a technique called *stochastic gradient descent*, which allows us to pick a random point at every time step instead of running the algorithm on the entire dataset every time. It turns out that this randomized method still does a very good job of approximating the optimal value, while requiring far less computational power.

Algorithm 3 Stochastic Gradient Descent: Determining parameters $\boldsymbol{w}$ to minimize loss $L(\boldsymbol{w}, \boldsymbol{y})$ with initial value $\boldsymbol{w}_{init}$, step-size η, and accuracy parameter ϵ

$\boldsymbol{w}_0 = \boldsymbol{w}_{init}$
$t = 0$
while $|L(\boldsymbol{w}_t, y) - L(\boldsymbol{w}_{t-1}, y)| > \epsilon$ **do**
  $t = t + 1$
  Pick random point w_i in $\boldsymbol{w}$
  $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \nabla_{w_i} L$
end while
return $\boldsymbol{w}_t$

3.3. **Error Back-Propagation [2].** In the context of a neural network, we will often have a set of "training data": recall from Section 2 that this is a sample set of data with inputs $\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_n}$, which we can combine into a matrix X, and their associated outputs $y_1, y_2, ..., y_n$, which we can combine into a vector $\boldsymbol{y}$. We use

our training data to optimize the weights (by reducing the error) of, or "train" our neural network, so that we can use our network to reliably predict outputs given data where the actual output is unknown. We can use stochastic gradient descent, optimizing over the weights of each layer, to train our network.

The difficult part will be calculating the gradient of the loss $L(NN(x;W), y)$, where $NN(x;W)$ represents the output of the neural network with input $\boldsymbol{x}$ and weights W, with respect to the weights of all the layers. We will start by calculating the gradient of loss with respect to the last layer, W^L, by applying the chain rule.

$$\frac{\partial L}{\partial W^L} = \frac{\partial L}{\partial \boldsymbol{a}^L} \cdot \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \cdot \frac{\partial \boldsymbol{z}^L}{\partial W^L}$$

Note that $\boldsymbol{z}^L = W^{LT}\boldsymbol{a}^{L-1}$, so $\frac{\partial \boldsymbol{z}^L}{\partial W^L} = \boldsymbol{a}^{L-1}$. This is true for any layer, so we can write the partial derivative of loss with respect to layer l, rearranging to match dimensions, as

$$\frac{\partial L}{\partial W^l} = \boldsymbol{a}^{l-1}\Big(\frac{\partial L}{\partial \boldsymbol{z}^l}\Big)^T \tag{3.4}$$

Now it remains to find $\frac{\partial L}{\partial \boldsymbol{z}^l}$. We can again apply the chain rule:

$$\frac{\partial L}{\partial \boldsymbol{z}^l} = \frac{\partial L}{\partial \boldsymbol{a}^L} \cdot \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \cdot \frac{\partial \boldsymbol{z}^L}{\partial \boldsymbol{a}^{L-1}} \cdot \frac{\partial \boldsymbol{a}^{L-1}}{\partial \boldsymbol{z}^{L-1}} \cdot ... \cdot \frac{\partial \boldsymbol{a}^{l+1}}{\partial \boldsymbol{z}^{l+1}} \cdot \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l} \cdot \frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l}$$

Again, since $\boldsymbol{z}^k = W^{kT}\boldsymbol{a}^{k-1}$, we have $\frac{\partial \boldsymbol{z}^k}{\partial \boldsymbol{a}^{k-1}} = W^k$ for all $1 \le k \le L$. $\frac{\partial \boldsymbol{a}^k}{\partial \boldsymbol{z}^k}$ depends solely on f^k, so given the function f^k and the entry z_j^k, we can calculate all the entries in this matrix. Finally, the last part of the equation, $\frac{\partial L}{\partial \boldsymbol{a}^L}$, depends on the loss function and the output of the neural network. Putting it all together and rearranging to match dimensions:

$$\frac{\partial L}{\partial \boldsymbol{z}^l} = \frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \cdot W^{l+1} \cdot \frac{\partial \boldsymbol{a}^{l+1}}{\partial \boldsymbol{z}^{l+1}} \cdot ... \cdot \frac{\partial \boldsymbol{a}^{L-1}}{\partial \boldsymbol{z}^{L-1}} \cdot W^L \cdot \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \cdot \frac{\partial L}{\partial \boldsymbol{a}^L} \tag{3.5}$$

Combining this with Equation 3.4, we obtain a closed-form expression for the gradient of loss with respect to any layer l.

$$\frac{\partial L}{\partial W^l} = \boldsymbol{a}^{l-1}\Big(\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \cdot W^{l+1} \cdot \frac{\partial \boldsymbol{a}^{l+1}}{\partial \boldsymbol{z}^{l+1}} \cdot ... \cdot \frac{\partial \boldsymbol{a}^{L-1}}{\partial \boldsymbol{z}^{L-1}} \cdot W^L \cdot \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \cdot \frac{\partial L}{\partial \boldsymbol{a}^L}\Big)^T \tag{3.6}$$

At each step of the stochastic gradient descent algorithm, we update the weights by first doing a *forward pass* to compute all of the $\boldsymbol{a}$ and $\boldsymbol{z}$ values at each layer, and eventually the final output and the corresponding loss. Then we work backwards, using *error back-propagation* to compute the gradient of the loss with respect to the weights in each layer. This can be thought of intuitively as "assigning blame" to the weights in each layer; that is, how much did that layer contribute to the overall loss. We use this gradient to complete the gradient descent step on our chosen random point, then start all over again with another random point. Eventually, we obtain a neural network that minimizes loss given our training dataset.

4. VC Dimension & Epsilon Nets

4.1. **VC Dimension [6; 9].** Having introduced a series of Machine Learning models, including linear classifiers and neural networks, it is critical that we have some means of *quantifying* how powerful these models are. **The Vapnik–Chervonenkis**

(VC) dimension is one such way of quantifying the capacity (power, complexity, expressive power, flexibility) of a binary classification model. Recall that a binary classification model is a model which classifies objects into one of two classes (e.g. +1 or -1). Capacity is somewhat of a vague notion, so it helps to think of it as the following verbal descriptions:

- Ability to handle complexity
- Representational power
- How complicated of a pattern or relationship the classifier can express
- How well the classifier generalizes

Using the VC dimension, we have place a quantitative value on these notions for a given model.

First, we will define the concept of *shattering*.

Definition 4.1. A binary classification model *shatters* a set of points in $\mathbb{R}^d$ if for all assignments of labels (negative or positive) to those points, there exists some parameter vector θ such that the model correctly classifies every point.

Example 4.2. We will break down this definition visually through a series of examples. The following dataset of two points is shattered by $\text{sign}(\theta_1 x_1 + \theta_2 x_2 + \theta_0)$, a linear classifier in $\mathbb{R}^2$. First, notice the assignments of true values: there are always 2^n assignments when objects are classified in a binary fashion.

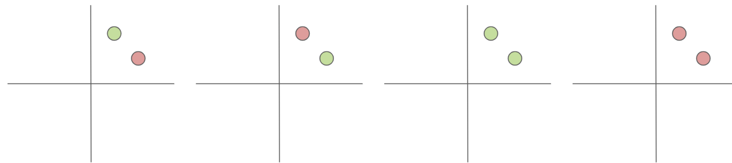

Figure 9. A graphical illustration of D_1.

We construct a dataset D_1 of all 2^n assignments of truth values to two points in $\mathbb{R}^2$, as illustrated in Figure 9. Having enumerated each possible training set, we can now show that D_1 can be shattered by drawing linear separators which are perfect separators. Since D_1 consists only of two points, and thus there are only 4 assignments of truth values we must consider, we can easily and visually draw classifiers by hand. However, in more complicated cases (e.g. more complicated neural networks as our model or larger datasets) we must use a machine and an algorithm which determines the proper linear separator (if it exists) in each case. We could use any of the algorithms discussed in the previous parts of this paper, including the Perceptron, Gradient Descent, and Error Back Propagation.

In this simple example, it is clear that the dataset can be shattered by a linear classifier, as depicted by Figure 10. However, there are many scenarios under which this is not the case.

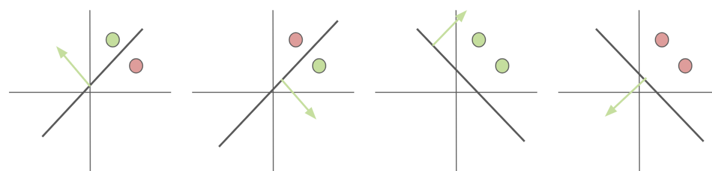

FIGURE 10. Constructing linear classifiers using the parameter values θ, θ_0 to perfectly separate the data between +1 (green) and -1 (red).

Example 4.3. Consider a model which is a circle centered at the origin, where the only changeable parameter is the radius, r. This model always has normal vector pointing inwards. Can this model shatter the following two points, enumerated in Figure 11, in $\mathbb{R}^2$?

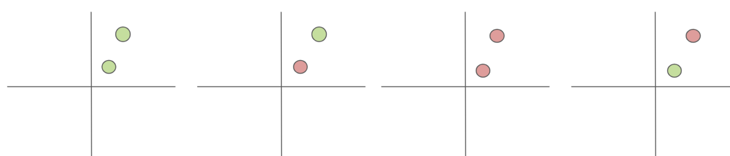

FIGURE 11. The 2^n assignments of truth values to two other points in $\mathbb{R}^2$. Define this dataset D_2.

We now seek to determine a classifier for each possible training dataset which contains only the positively classified points inside the circle (in the direction of the positive normal to the circle), and the negatively classified points outside.

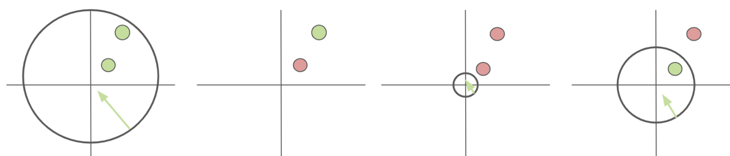

FIGURE 12. Attempting to construct circular classifiers which perfectly separate the data between +1 (green) and -1 (red).

However, unlike the previous example, we cannot do so! According to Figure 12, we see the following successful cases. When both points are classified positively, the circle can encompass both, with normal vector pointing inwards. When both points are classified negatively, we have a small circle which doesn't encompass either point, such that outside the circle represents the negative class. Finally,

when there is one positive point and one negative point where the positive point is closer to the origin, the circle's perimeter splits the two points into two sub-planes, with the positive point encapsulated by the circle. Now, it is clear why the second assignment of values to points in the above image cannot be classified by this type of model - there is no way to encompass only the positive (green) point by the circle, but not the negative point. As a result, we say that the circle centered at the origin with inward pointing normal vector *does not* shatter D_2.

Having concluded these two informative examples, we have laid the groundwork to define *VC Dimension.*

Definition 4.4. The *VC Dimension* of a model f is the maximum number of points that can be arranged such that some instance of f can shatter them.

It is key to observe that the definition mentions *any* arrangement of points; that is, it suffices to show that a single arrangement of points forces us to be unable to find a classifier, thus making that quantity of points un-shatterable, and thus a non-inclusive upper bound for the VC-Dimension. Given this definition, we can look to the previous example 4.3. Since the circular function cannot shatter the D_2 which has two points, we say that the VC Dimension of our origin-centered circle is 1. It is true by observation that this model can shatter a single data point, but simply encompassing the point, if positive, by the circle, otherwise not. Thus, 1 point is the maximum number of points that can be arranged such that the circular origin-centered model cannot shatter them.

Let us shift focus to a more generalized example, connecting the previous Perceptron algorithm and linear classifiers to the VC-Dimension.

Example 4.5. Recall that the Perceptron yields a binary linear classifier of the form:

$$f(x;\theta) = \text{sign}(\theta_1 x_1 + \theta_2 x_2 + ... + \theta_{d+1} x_{d+1} + \theta_0),$$

for $N = d + 1$ points in $\mathbb{R}^d$. We will now introduce a theorem regarding the VC-Dimension of a classifier of this form.

Theorem 4.6. *The VC Dimension of a linear classifier of the form* $f(x;\theta) = sign(theta_1 x_1 + \theta_2 x_2 + ... + \theta_{d+1} x_{d+1} + \theta_0)$ *on* $N = d + 1$ *points in* $\mathbb{R}^d$ *is equal to exactly* $d + 1$.

Proof. In order to show that the VC-Dimension, d_{VC}, is exactly equal to $d + 1$, we will prove two bounds: (1) $d_{VC} \geq d + 1$, and (2) $d_{VC} \leq d + 1$, thus implying that $d_{VC} = d + 1$.

(1) $d_{VC} \geq d + 1$. It suffices to show that there exists a set of $d + 1$ points such that $f(x;\theta)$ can produce any pre-specified $\{+1, 11\}$ assignment of values. Since each of the $n + 1$ input vectors $x^{(i)}$ is a vector in $\mathbb{R}^d$, we can construct an $(n + 1)$ by $(n + 1)$

input matrix X.

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ . \\ . \\ x_{d+1}^T \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & ... & 0 \\ 1 & 1 & 0 & ... & 0 \\ 1 & 0 & 1 & ... & 0 \\ . & . & . & ... & . \\ . & . & . & ... & . \\ 1 & 0 & 0 & ... & 1 \end{bmatrix}$$

The first coordinate of each row is 1 in order for the bias term θ_0 to be produced. By observing the matrix X, we notice that it is invertible, because its columns are all linearly independent. As a result, we guarantee that $f(x;\theta)$ shatters any $d+1$ points by choosing the θ, θ_0 which guarantees $\text{sign}(\theta_1 x_1 + \theta_2 x_2 + ... + \theta_{d+1} x_{d+1} + \theta_0)$ always equals y. First write the combination of θ and θ_0 as the matrix θ' for simplicity. Now, choose $\theta' = X^{-1} y$ such that $X\theta' = y$ implies $\text{sign}(X\theta') = y$. Thus due to X's invertibility, we have shown that $f(x;\theta)$ shatters at least $d+1$ points.

(2) $d_{VC} \leq d+1$. Now, we must show that no $d+2$ points can be shattered by $f(x;\theta)$. That is, there is some assignment of y values to $d+2$ points in $(d+1)$-dimensional space which are not separable by $f(x;\theta)$. First, observe that this requires the set of points to be linearly dependent, since we have more points than dimensions in our matrix X. This means that at least one $x^{(j)}$ can be constructed as a linear combination of the other x vectors.

We can formalize this relationship as: $x^{(j)} = \sum_{i \neq j} a_i x^{(i)}$ such that not all a_i are 0. Since every assignment of y values to each point must be perfectly classified by $f(x;\theta)$ in order for the dataset to be shatterable, we can choose any assignment of y values to show that this fails. Choose $y^{(j)}$ = -1 and $y^{(i)} = \text{sign}(a_i) = \text{sign}(\theta_1 x_1 + \theta_2 x_2 + ... + \theta_{d+1} x_{d+1} + \theta_0)$. We proceed by calculating $\text{sign}(\theta^T x^{(i)})$, which is the predicted value of $y_{(i)}$, to show a contradiction. We can rewrite as follows:

$$\begin{aligned} y_{Pred}^{(j)} &= \text{sign}(\theta^T x^{(j)}) \\ &= \text{sign}(\sum_{i \neq j} a_i \theta^T x^{(i)}) \\ &> 0 \\ &\implies y_{Pred}^{(j)} \neq y^{(j)} = -1. \end{aligned}$$

To summarize the above math, the sign of a_i and the sign of $\theta^T x^{(i)}$ are by definition equivalent, so multiplying the two quantities yields a positive number. Thus, the prediction on data point j must be positive, so the assignment of truth values to each y where $y^{(j)} = -1$ is *not* perfectly separable by the linear classifier with $d+1$ parameters. Thus, there exists an assignment of $d+2$ points not separable by $f(x;\theta)$, so $d_{VC} \leq d+1$.

Combining (1) and (2) which state respectively that $d_{VC} \geq d+1$ and $d_{VC} \leq d+1$, we have only one remaining quantity for the VC-Dimension of $f(x;\theta)$, which is $d+1$. In other words, the VC-dimension of a linear classifier is proportional to the

data dimension, and equals the number of parameters of the classifiers. Note that although the data is in $\mathbb{R}^d$, there is one additional parameter, θ_0, contributing to the increase of 1 in the VC-Dimension. □

Recall our initial definition of capacity, which states that a model's capacity is a function of its representational power, and how complicated of a relationship the classifier can express. It is therefore intuitive that a linear model has capacity (as quantified by VC-Dimension) proportional to the number of parameters, or degrees of freedom!

4.2. **Epsilon Nets [5].** In Machine Learning, we often deal with very large training datasets, which require a lot of computational power, even with the efficient algorithms we have already discussed. One way that we can reduce the computational complexity of a problem is by randomly sampling some of the points in our data. However, in predictive problems, especially for classification problems, we want to make sure that we sample enough points to get a representative picture of the data. For instance, in the previous section we discussed the XOR classification problem. We can easily see (in Figure 7) that we have four "types" of points: the two positive quadrants and the two negative quadrants. So when we're sampling from a dataset used for the XOR problem, we would want to make sure we sample enough points to be reasonably certain that we get a point from each of the four "groups". We call a representative sample of points an *epsilon net*, i.e. a set of points that cover all sufficiently large "groups".

Definition 4.7. An *ϵ-net* is a sample N of sample space X and set space S, such that N intersects all sets that contain at least $\epsilon|X|$ points.

The question is, how do we calculate the number of points to sample for an arbitrary model? It turns out that we can use the VC dimension of a model to solve this problem.

Theorem 4.8. *If sample space X with set space S has VC Dimension d, then a uniform random sample N of size $O(\frac{d}{\epsilon}\ln(1/\epsilon))$ is an ϵ-net with probability at least* $1/2$.

Proof. First, set $s := c \cdot \frac{d}{\epsilon} ln(1/\epsilon)$. This will be the number of points we want to sample for N. Instead of directly sampling s points, however, we'll try to sample set A with twice as many points as we need – i.e. pick $|A| = 2s$ random points. Then pick half of the points in A at random to be in N. The remaining points will be set $M = A \setminus N$. We'll also assume that all sets have size at least $\epsilon|X|$ (if there are any small sets, we can simply ignore them).
Now we'll define two important events:

$$E_1 = \text{There exists a set } S^* \in S \text{ that does not intersect } N$$
$$E_2 = \text{There exists a set } S^* \in S \text{ that does not intersect } N \text{ and that intersects } M \text{ at least } k := 1/2\epsilon s \text{ times}$$

Claim 1: $\frac{1}{2}\mathbb{P}[E_1] \leq \mathbb{P}[E_2] \leq \mathbb{P}[E_1]$.

Note that $\mathbb{P}[E_2] \leq \mathbb{P}[E_1]$ is immediate. Furthermore, this claim is true if N is an epsilon net: by definition, an epsilon-net must intersect all sets of size at least $\epsilon|X|$, so if N is an epsilon-net, then N cannot fail to intersect any set S, and so

$\frac{1}{2}\mathbb{P}[E_1] = \mathbb{P}[E_2] = 0$. So it remains to show that this is true if N is not an epsilon net.

Now fix set S^* such that S^* does not intersect N. Let $y := |M \cap S^*|$, the number of points in M that lie in S^*. Note that we can simply show $\mathbb{P}[y \geq k] \geq \frac{1}{2}$, since this implies that the probability of E_2 is at least $1/2$ of the probability of E_1.

Recall that $|M| = s$. If we select a random point from X, the probability that the point is in S^* is **at least** ϵ. Since we draw s points **independently** to create M, $\mathbb{E}[y]$ (the expected number of points in M that lie in S^*) is at least $\epsilon \cdot s = 2k$. Using Chernoff bounds, we can simplify our expression to obtain:

$$\mathbb{P}[y \geq k] \geq \mathbb{P}[y \geq \frac{1}{2}\mathbb{E}[y]] \geq 1 - \exp(-1/8\mathbb{E}[y])$$

Note that $\mathbb{E}[y] = c \cdot d \ln(1/\epsilon)$, so we can choose constant c large enough that this is bounded by $1/2$.

Claim 2: For every fixed choice of A, we have $\mathbb{P}[E_2|A] < \frac{1}{4}$.
Once we fix A, we only need to consider sets in A. Formally, we only consider sets $S' \in \{S_i \cap A | S_i \in S\}$. Note that this new set system still has VC dimension bounded by d.

Next, consider some specific $S' \in A$. E_2 requires that there are at least k points in S', and also that the points we choose for N don't come from S'. Then there are at most $\binom{2s-k}{s}$ ways to choose N such that E_2 is true, and $\binom{2s}{s}$ ways to choose N in general. So we can write:

$$\mathbb{P}[E_2|A, S'] \leq \frac{\binom{2s-k}{s}}{\binom{2s}{s}} < \exp(-k/2) = (1/\epsilon)^{-cd/4}$$

Finally, note that since our set system has VC dimension bounded by d, we can bound the number of sets S' from above by $(\frac{12s}{d})^d = (12c\ln(1/\epsilon)/\epsilon)^d$ (again, we won't prove this here). We can now apply the union bound:

$$\begin{aligned}\mathbb{P}[E_2|A] \leq [\text{number of sets } S'] \cdot \mathbb{P}[E_2|A, S'] &\leq (1/\epsilon)^{-cd/4} \cdot (12c\ln(1/\epsilon)/\epsilon)^d \\ &\leq (12c(1/\epsilon)^2 \cdot (1/\epsilon)^{-C/4})^d\end{aligned}$$

We can pick arbitrary C large enough that this is bounded by $1/4$. So we have $\mathbb{P}[E_1] \leq 2\mathbb{P}[E_2] < 2(1/4) = 1/2$, as required. □

With the introduction of VC-Dimension and Epsilon Nets, we have introduced both a method to quantify the complexity of a classifier and a method to calculate the number of points to sample for a model. Using these tools, we can tune our Machine Learning models to precisely as powerful as desired.

5. Markov Decision Processes

5.1. Introduction to Decision-Making Processes and Markov Models [8].

At this point in the paper, we shift from analyzing predictive algorithms and their complexity to examine decision-making processes. In short, **decision-making processes enable a machine to learn how to optimize a utility function through repeated interactions with an environment.** Therefore, any decision-making process requires feeding a machine a utility function and allowing

the machine to interact with some environment for some time. Through experience, the machine will identify possible decisions, obtain a reward for selecting an optimal action, and reinforce its proclivity to select the same action in the future. That is, when the machine experiences a positive reward for taking some action, they will continue to take that action when in the same circumstance in the future, and they will avoid actions that yield lesser or negative rewards.

Markov Models are a common structure taught in any introductory discrete mathematics course, and are often used to simulate decision-making processes.

Definition 5.1. A *Markov Model* is a discrete-time stochastic process satisfying the following condition, which we call the *Markov Property*:

$$\mathbb{P}(s_{t+1}|s_1, s_2, ...s_t) = \mathbb{P}(s_{t+1}|s_t). \tag{5.1}$$

That is, the conditional probability of future states depends only on the current state.

We begin by introducing the Markov Chain, which is an example of a type of Markov Model.

Definition 5.2. A *Markov Chain* is a temporal sequence of states, in which transitions between each state occurs based on a probability distribution. A Markov Chain is characterized by:

- **S**: A finite set of possible states, where s_t represents the state at time t.
- **T**: A stochastic state transition matrix, where the jth entry in the ith row $T_{ij} = \mathbb{P}[s_{t+1} = j|s_t = i]$ represents the probability of transitioning from state i at time t to state j at time $t+1$.

Additionally, the Markov Chain is an example of a Markov Model, and so it must also satisfy the Markov Property.

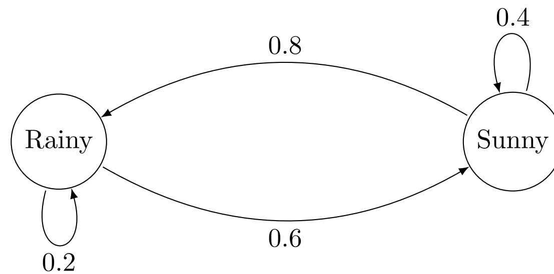

FIGURE 13. Above is a standard Markov Chain highlighting the stochastic transitions between states. In this example, the Markov Property means tomorrow's weather depends only on today's weather, and not on last week's blizzard!

Example 5.3. Consider the Markov Chain illustrated in Figure 13, which can be defined by the following components:

- The set of states $\mathbf{S} = \{\text{Rainy}, \text{Sunny}\}$
- The transition matrix

$$\mathbf{T} = \begin{bmatrix} 0.2 & 0.8 \\ 0.6 & 0.4 \end{bmatrix}$$

With some intuition for Markov Chains and Markov Models, we can now define several properties of Markov Chains.

Definition 5.4. A Markov Chain is *autonomous* and *fully observable*, because one can simply completely observe the process and current state, but cannot take actions to modify the state. As we will see in the next two sections, various types of Markov Models emerge as we vary the user's ability to take actions, obtain rewards, and observe the state of the system. This is illustrated in Table 1.

Types of Markov Models		
	Fully Observable State	Partially Observable State
System is Autonomous	Markov Chain	Hidden Markov Model
System is Controlled	Markov Decision Process	Partially Observable MDP

TABLE 1. Types of Markov Models, categorized by whether the model is autonomous or controlled, and fully observable or partially observable.

5.2. **Markov Decision Processes [8].** In a Markov Chain, an agent may observe the current state. In Markov Decision Processes (MDPs), an agent may also choose an action and receive an immediate reward for that action. Furthermore, in MDPs, the actions chosen by the agent affect the probability distribution of the next state.

Definition 5.5. A *Markov Decision Process (MDP)* is a type of Markov Model. It is characterized by:

- **S**, a finite set of states
- **A**, a finite set of actions
- $\delta : S \times A \to S$, a stochastic state transition function for each state-action pair
- $\mathbf{R} : S \times A \to \mathbb{R}$, a reward function for each state-action pair

Example 5.6. At this point, it may be helpful to illustrate a problem which can be cast into an MDP. In robotics, path planning is a crucial task in which the robot interacts with the environment, observes its current state, and obtains rewards based on reaching a certain state. Figure 14 illustrates the path planning task for a robot in a simple environment. We can introduce a stochastic element in the transitions by noting that a robot is most likely, with 0.8 probability, to move in the direction of its action, with a small 0.1 probability of malfunction causing it to move in either adjacent direction. Note that it is possible the robot may attempt to move into a wall: if the robot moves into a wall, it simply remains in the same

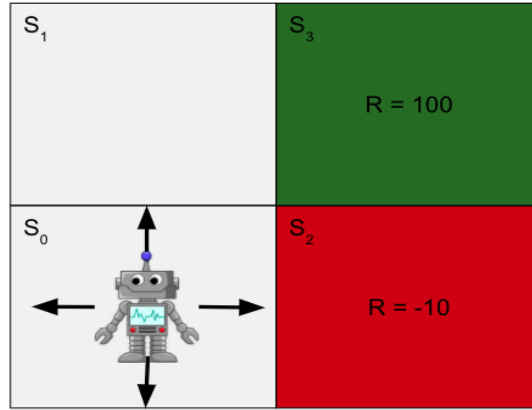

FIGURE 14. Above is a path planning problem highlighting states, actions, and rewards.

state. Now, we can cast the path planning example problem into a Markov Decision Process:

- $\mathbf{S} = \{ \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3 \}$
- $\mathbf{A} = \{$ Up, Down, Left, Right $\}$
- $\delta(s, a)$:
  - $\delta(S_0, UP) = [0.1, 0.8, 0.1, 0.0]$. That is, moving up from S_0 causes the robot's next state to be S_1 with probability 0.8, S_2 with probability 0.1 (malfunction to go right), and S_0 with probability 0.1 (malfunction to go right, hits wall, stays in place)
  - $\delta(S_1, UP) = [0.0, 0.9, 0.0, 0.1]$
- $\mathbf{R}(\mathbf{s}, \mathbf{a}) = \begin{cases} -10 \text{ if } s = S_2 \\ +100 \text{ if } s = S_3 \\ 0 \text{ otherwise} \end{cases}$

Therefore, we have effectively cast the robot path planning problem into a Markov Decision Process, complete with a stochastic transition function, a series of actions, and rewards.

Definition 5.7. In an MDP, a *policy* is a function $\pi : S \to A$ that prescribes which action the agent will take from a given state. Given an environment model of an MDP, **our goal is to determine an optimal policy π^* that maximizes the total reward earned by the agent over all time periods**. This is called the agent's *lifetime reward*.

When defining lifetime reward, we encounter an obstacle - if an agent has a series of actions that yield a positive reward, then the agent can repeat those actions forever to obtain an infinite reward! To resolve this issue, we introduce the concept of discounting. The principle behind discounting is simple - rewards obtained more

immediately should have a higher weight than the same reward obtained in the future.

Definition 5.8. A *discount factor* $\gamma \in [0, 1)$ is a factor applied to all future rewards so as to enforce a finite reward for an infinite lifetime MDP. A reward r, obtained t periods in the future, is then *discounted* by γ^t.

Lastly, let us introduce notation used for the remainder of this section. As a reminder, subscripts are used to indicate the time step; therefore, s_1 is the state in the 1st time step, a_2 is the action taken in the 2nd time step, and r_3 is the immediate reward received from taking action a_3 in state s_3 during the 3rd time step, ie $R(s_3, a_3)$.

Definition 5.9. A *value function* $V^\pi : S \to \mathbb{R}$ maps a starting state s_0 to the total lifetime reward accrued by executing policy π at each time step, i.e. the *value* of the policy from state s_0.

$$V^\pi(s_0) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... = \sum_t \gamma^t * r_t \tag{5.2}$$

5.3. **The Value Iteration Algorithm [8].** Now that we have defined the lifetime reward of a policy for a given state, our goal is to find the optimal policy, that is, the optimal decision for our agent in each state. Fundamentally, our algorithm to determine the optimal policy $\pi^*(s)$ is as follows:

1. Examine all possible policies π from s
2. Select the policy π^* with the greatest value. Expressed as a formula:

$$\pi^* = \arg\max_\pi V^\pi(s)$$

As such, we have shown how to extract the optimal policy from a series of Value Functions for all possible policies. Therefore, in order to find the optimal policy of an agent in the MDP environment, we are interested in only one Value Function - the optimal Value Function V^*. Let us definite the optimal Value Function V^*:

$$V^* = \max_\pi V^\pi(s)$$

Then, our algorithm to find the optimal value function $V^*(s)$ is as follows:

1. Examine all possible actions a' in set of actions A
2. Compute r, the expected immediate reward for action a' in state s
3. Compute l, the expected lifetime reward for the next state s', assuming our agent acts optimally from state s' onwards
4. Select the action that maximizes $r + l$.

$$V^*(s) = \max_{a' \in A}[R(s, a_i) + \gamma V^*(\delta(s, a'))] \tag{5.3}$$

Equation (5.3) is the formal statement of Bellman's Equation. In short, Bellman's Equation states that the optimal value of a state is the immediate reward for that state, plus the expected discounted reward of the next state. Note that our above definition then relies on a pre-computed optimal Value Function V^*. When implementing, this recursive definition of the optimal value function requires dynamic programming to compute efficiently and effectively.

The most common algorithm to compute the optimal value function of an MDP is known as Value Iteration. In this algorithm, we iteratively update our optimal value function, stopping when the updates become negligible. Expressed in pseudo-code format, the Value Iteration algorithm is as follows:

Algorithm 4 Value Iteration: Calculate Optimal Value Function V^*

$\Delta = 2\epsilon$
$t = 0$
$\forall s \in S : V_0^*(s) = 0$
while $\Delta \geq \epsilon$ **do**
  $V_{t+1}^*(s) = \max_{a' \in A}[R(s, a') + \gamma V_t^*(\delta(s, a'))]$
  $\Delta = |V_{t+1}^*(s) - V_t^*(s)|$
  $t = t + 1$
end while
return V_t^*

Armed with the Value Iteration Algorithm, we can now efficiently compute the optimal Value Function V^* for each state using the recursive power of Dynamic Programming. By definition V^*, we can now extract the optimal policy from the optimal Value Function using a one-step lookahead process. That is,

$$\pi^* = \arg\max_{a' \in A}[R(s, a') + \gamma V^*(\delta(s, a'))].$$

Recall that our goal was to find the optimal policy π^*, which is the best move the agent can make in each state in order to optimize its lifetime reward. Therefore, we have constructed a series of algorithms by which a machine interacting with an environment can find an optimal series of decisions. The basis for this ability is for the machine to iteratively interact with its environment in the Value Iteration Algorithm, and deduce optimal policies after calculating optimal values for each state.

In summary, Markov Decision Processes represent a powerful means to abstract decision problems, and provide a robust method for optimal decision-making by a machine. The foundation of this powerful capability is rooted in random walks over Markov Models, probability theory, and recursion.

6. Hidden Markov Models

6.1. **Hidden Markov Models [7].** So far, we have been introduced to Markov models with fully observable states. Both Markov Chains and Markov Decision Processes learn by observing the sequence of states – they can directly use this information to make predictions and draw conclusions about the system. In contrast, a Hidden Markov Model (HMM), while still satisfying the Markov Property, does not allow us to directly observe the state at any point in time. We are, however, given access to a series of related observations, where each observation depends on the current state. Therefore, an HMM requires an additional input of an observation model, which highlights the probability of each observation for any given state.

Definition 6.1. A *Hidden Markov Model* is a type of Markov Model. It is characterized by:

- **S**: A finite set of states, where s_t represents the state at time t.
- **V**: A finite set of observations, where o_t represents the state at time t.
- **T**: A stochastic state transition matrix where $\mathbf{T_{ij}} = \mathbb{P}(s_{t+1} = j | s_t = i)$, as in a Markov Chain.
- **O**: A stochastic observation matrix where the jth entry in the ith row $\mathbf{O_{ij}} = \mathbb{P}(o_t = j | s_t = i)$ represents the probability of observing observation j at time t given that we are at state i.

Example 6.2. At this point, it is worthwhile to provide an example of a Hidden Markov Model. First, let us refresh our weather Markov Chain example from Figure 13. Now, let us say that we are unable to directly detect the state - in this case, the day's weather - because we are a motivated Marine guarding the President's underground bunker. However, we are able to observe the President walking by each morning either carrying an umbrella or wearing a baseball cap. Naturally, the President's attire choices are conditionally dependent on the weather outside, but not perfectly correlated; for example, the President may be wearing a ball cap not to protect from the sun but because the second greatest organization known to mankind (behind the United States Marine Corps), the New York Yankees, have a game tonight. In this example, the components of the HMM are as follows:

- The set of states $\mathbf{S} = \{\text{Rainy}, \text{Sunny}\}$
- The set of observations $\mathbf{O} = \{\text{Umbrella}, \text{Ball-Cap}\}$
- The stochastic state transition matrix
$$\mathbf{T} = \begin{bmatrix} 0.2 & 0.8 \\ 0.6 & 0.4 \end{bmatrix}$$
- The stochastic observation matrix:
$$\mathbf{O} = \begin{array}{c} \\ \end{array}\begin{array}{cc} \text{Umbrella} & \text{Ball-Cap} \\ \end{array}$$
$$\mathbf{O} = \left[\begin{array}{cc} 0.9 & 0.1 \\ 0.4 & 0.6 \end{array} \right] \begin{array}{l} \text{Rainy} \\ \text{Sunny} \end{array}$$

6.2. **Inference Tasks on Hidden Markov Models [7].** Given a Hidden Markov Model, there are known algorithms to solve a series of fundamental inference tasks:

(1) **Filtering: Given, the available sequence of observations, what is our belief about the current hidden state?** Our goal with filtering is to compute our current *belief state*, a probability distribution across all possible states at time t given a series of observations from time step 1 to time step t. Expressed mathematically, we want to find
$$\mathbb{P}(s_t | o_1, o_2, ..., o_t) = \mathbb{P}(s_t | o_{1:t}).$$
In our weather HMM example from above, the filtering inference task is computing the probability the weather is rainy or sunny outside today, given the past week's observations of the President's umbrella or ball cap accessories. Filtering is a fundamental inference task and solved using a single forward pass to calculate a belief state, as we describe in the next section.

(2) **Smoothing: Given the available sequence of observations, what is our belief about a past hidden state?** Our goal with smoothing is

to compute the belief state for time $k < t$ using all observations from time step 1 to time step t. Expressed mathematically, we want to find

$$\mathbb{P}(s_k|o_1, o_2, ..., o_t) = \mathbb{P}(s_k|o_{1:t}) \text{ such that } k < t.$$

In our weather HMM example from above, the smoothing inference task is to compute the belief state of last Tuesday's weather, given the past week's observations of the President's umbrella or ball cap accessories. Smoothing is a challenging inference task and is most commonly solved using the Forward-Backward Algorithm.

(3) **Prediction: Given the available sequence of observations, what is our belief about a future hidden state?** Prediction is an inference task in which we project our current belief state in order to compute a future belief state for time $k > t$, given observations from time step 1 to time step t. Expressed mathematically, the structure for prediction is very similar to that of smoothing. We want to find

$$\mathbb{P}(s_k|o_1, o_2, ..., o_t) = \mathbb{P}(s_k|o_{1:t}) \text{ such that } k > t.$$

In our weather HMM example from above, the prediction inference task is computing the belief state for tomorrow's weather, given the past week's observations. Once we have found the current belief state for time t, any prediction inference relies solely on the transition model applied to the current belief state. Therefore, solving prediction for HMMs is equivalent to prediction in Markov Chains.

(4) **Most likely explanation: Given the available sequence of observations, what is the most likely sequence of states to have caused those observations?** Commonly called decoding, this task requires computing a sequence of states that are the most likely to correspond to the sequence of observations. Decoding is most commonly accomplished via the Viterbi Algorithm and finds numerous applications in speech recognition and computational linguistics.

Lastly, it is worth mentioning that serious effort has been made to develop algorithms for a machine to learn the transition or observation models of a Hidden Markov Model. Although this paper will not dive into these advanced methods, the reader may choose to research parameter learning in the Expectation-Maximization (EM) Algorithm or for specific application to HMMs, the Baum-Welch Algorithm.

6.3. **Derivation of Forward-Algorithm for Filtering [7].** Given the above definitions of inference tasks to be performed on a Hidden Markov Model, we now turn to constructing and analyzing their corresponding algorithms. We will first examine the most fundamental inference task of a Hidden Markov Model. Given that we are unable to directly determine the state of the system, we need a method to determine the current state we believe the system is likely to be in – the belief state at time t denoted as $\hat{p}_t$. As such, we will now apply fundamental rules of Probability Theory – such as Bayes' Rule, the Chain Rule, and the Markov Property (5.1) – to establish the Forward-Algorithm used for the inference process of filtering.

First, we recall the mathematical definition of the filtering inference task, given by

$$\hat{p}_t = \mathbb{P}(s_t | o_{1:t}) = \mathbb{P}(s_t | o_1, o_2, ...o_{t-1}, o_t). \tag{6.1}$$

Now, we apply Bayes' Rule with a normalizing constant, as follows:

$$\mathbb{P}(s_t | o_1, o_2, ...o_{t-1}, o_t) = \eta \cdot \mathbb{P}(o_t | s_t, o_{1:t-1}) \cdot \mathbb{P}(s_t | o_{1:t-1}).$$

Then, by our definition of the observation model, the observation at time t depends only on the state at time t, given by the equalities

$$\mathbb{P}(s_t | o_{1:t}) = \eta \cdot \mathbb{P}(o_t | s_t, o_{1:t-1}) \cdot \mathbb{P}(s_t | o_{1:t-1}) = \eta \cdot \mathbb{P}(o_t | s_t) \cdot \mathbb{P}(s_t | o_{1:t-1}).$$

At this point, we note that the first term of our equation, $\mathbb{P}(o_t | s_t)$ is directly obtainable from the observation model of the HMM! Therefore, we simply need to solve for the second term, by expanding the second probability term using a marginal distribution over all possible prior states s_{t-1} as follows:

$$\mathbb{P}(s_t | o_{1:t-1}) = \sum_{i=1}^{d} \mathbb{P}(s_t, s_{t-1} = i | o_{1:t-1}). \tag{6.2}$$

Again, we apply Probability Chain Rule within the summation to separate our two events:

$$\sum_{i=1}^{d} \mathbb{P}(s_t, s_{t-1} = i | o_{1:t-1}) = \sum_{i=1}^{d} \mathbb{P}(s_t | s_{t-1} = i, o_{1:t-1}) \cdot \mathbb{P}(s_{t-1} = i | o_{1:t-1}). \tag{6.3}$$

Once again, we examine the conditional dependencies in a Hidden Markov Model to simplify our formula. By the Markov Property, the current state s_t depends only on the prior state s_{t-1}, which allows us to simplify the first term of (6.3) as follows:

$$\sum_{i=1}^{d} \mathbb{P}(s_t | s_{t-1} = i, o_{1:t-1}) = \sum_{i=1}^{d} \mathbb{P}(s_t | s_{t-1} = i). \tag{6.4}$$

Substituting (6.4) back into (6.3), we will now undo the marginalization, as follows:

$$\sum_{i=1}^{d} \mathbb{P}(s_t | s_{t-1} = i) \cdot \mathbb{P}(s_{t-1} = i | o_{1:t-1}) = \mathbb{P}(s_t | s_{t-1}) \cdot \mathbb{P}(s_{t-1} | o_{1:t-1}). \tag{6.5}$$

At this point, we can assert that the first term of (6.5) can clearly be obtained from the transition matrix. Similarly, the second term is simply the definition of the goal of the filtering process, but for a lower t value. That is,

$$\mathbb{P}(s_{t-1} | o_{1:t-1}) = \hat{p}_{t-1}.$$

Therefore, we have constructed a recursive definition for our belief state at time t, relying on a belief state at time $t-1$. Then, so long as our Hidden Markov Model has a base case belief state at time $t = 0$, before our observations begin, we can use a sequence of observations to compute our current belief state, with the following update step:

$$\hat{p}_t = \eta \cdot \mathbb{P}(o_t | s_t) \cdot \mathbb{P}(s_t | s_{t-1}) \cdot \hat{p}_{t-1}. \tag{6.6}$$

Using recursion, we can compute the belief states for each time step t' for which we have obtained an observation, culminating in our current belief state. This process is frequently called the Forward Algorithm because it makes a single forward pass to compute each belief state for each time step.

In addition to deriving a closed form formula for the filtering inference task, Hidden Markov Models allow us to calculate the current belief state using an elegant matrix representation. Above, we defined $\mathbf{T}$, the stochastic transition matrix. In addition to T, we will create a diagonalized observation matrix $\mathbf{D_{t-1}}$, corresponding to the observation at time step $t-1$.

We construct $\mathbf{D_{t-1}}$, a diagonal matrix whose ith row/column entry is $\mathbb{P}(o_{t-1}|s_{t-1} = i)$. That is, we take the column of $\mathbf{O}$ corresponding to the observation at time $t-1$, o_{t-1}, and construct a diagonal matrix $\mathbf{D_{t-1}}$ from that column vector.

Then, in a HMM, the current belief state at time t, given a series of observations is

$$\hat{\mathbf{p}}_{\mathbf{t}} = \eta \mathbf{D_{t-1}} \mathbf{T}^T \hat{\mathbf{p}}_{\mathbf{t-1}}. \tag{6.7}$$

This derivation of the forward algorithm for filtering reminds us of our theme throughout this paper: even complex tasks in a Machine Learning pipeline can be reduced down to their mathematical foundations.

References

1. Kaelbling, Leslie (2019, February). Linear Classifiers. 6.036: Machine Learning. Lecture, Cambridge, MA.

2. Kaelbling, Leslie (2019, March). Neural Networks. 6.036: Machine Learning. Lecture, Cambridge, MA.

3. Kaelbling, Leslie (2019, February). The Perceptron. 6.036: Machine Learning. Lecture, Cambridge, MA.

4. Kaelbling, Leslie (2019, March). Reinforcement Learning. 6.036: Machine Learning. Lecture, Cambridge, MA.

5. Mustafa, Nabil, et. al. “A Simple Proof of Optimal Epsilon Nets.” Combinatorica, Springer Verlag, 2017.

6. Rothvoss, Thomas. “Probabilistic Combinatorics.” University of Washington, 2019.

7. Shrobe, Howard. (2020, October). Hidden Markov Model Inference. 6.877: Principles of Autonomy and Decision Making. Lecture, Cambridge, MA.

8. Shrobe, Howard. (2020, October). Markov Decision Processes. 6.877: Principles of Autonomy and Decision Making. Lecture, Cambridge, MA.

9. Sontag, Eduardo D. “VC Dimension of Neural Networks.” Northeastern University, August 1998.