Machine Learning

or: The Unofficial Notes on the Georgia Institute of Technology's **CS7641**: *Machine Learning*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: January 14, 2020

0 Preface **3**

I Supervised Learning **4**

1 Techniques **5**

2 Classification **6**
2.1 Decision Trees 6
2.1.1 Getting Answers 7
2.1.2 Asking Questions: The ID3 Algorithm 8
2.1.3 Considerations 9

3 Regression **12**
3.1 Neural Networks 12
3.1.1 Perceptron 13
3.1.2 Sigmoids 17
3.1.3 Neural Network Structure 17
3.1.4 Bias 18
3.2 Instance-Based Learning 19
3.3 Ensemble Learning 19
3.4 Support Vector Machines 19
3.5 Bayesian Learning 19

II Unsupervised Learning **20**

4 Randomized Optimization **21**

Index of Terms **22**

Preface

> *I read that Teddy Roosevelt once said, "Do what you can with what you have where you are." Of course, I doubt he was in the tub when he said that.*
>
> — Bill Watterson, *Calvin and Hobbes*

Before we begin to dive into all things algorithmic, I'll enumerate a few things I do in this notebook to elaborate on concepts:

- An item that is **highlighted like this** is a "term;" this is some vocabulary that will be used and repeated regularly in subsequent sections. I try to cross-reference these any time they come up again to link back to its first defined usage; most mentions are available in the Index.
- An item in a maroon box, like...

Boxes: A Rigorous Approach

... this often represents fun and interesting asides or examples that pertain to the material being discussed. They are largely optional, but should be interesting to read and have value, even if it's not immediately rewarding.

- An item in a blue box, like...

Quick Maffs: Proving That the Box Exists

... this is a mathematical aside; I only write these if I need to dive deeper into a concept that's mentioned in lecture. This could be proofs, examples, or just a more thorough explanation of something that might've been "assumed knowledge" in the text.

I also sometimes include margin notes like the one here (which just links back here) that reference content sources so you can easily explore the concepts further. Linky

PART I

Supervised Learning

Our first minicourse will dive into **supervised learning**, which is a school of machine learning that relies on human input (or "supervision") to train a model. Examples of supervised learning include anything that has to do with labelling, and it occurs far more often than unsupervised learning. It's often reduced down to **function approximation** (think numpy's polyfit, for example): given enough predetermined pairs of (input, output)s, the trained model can eventually predict a never-before-seen input with reasonable accuracy.

An elementary example of supervised learning would be a model that "learns" that the dataset on the right represents $f(x) = x^2$. Of course, there's no guarantee that this data really does represent $f(x) = x^2$. It certainly could just "look a lot like it." Thus, in supervised learning we will need to a basal assumption about the world: that we have some well-behaved, consistent function behind the data we're seeing.

x	$f(x)$
2	4
9	81
4	16
7	49
...	

Contents

1 Techniques **5**

2 Classification **6**
2.1 Decision Trees 6

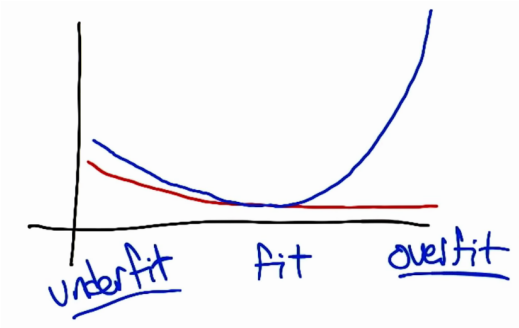
3 Regression **12**
3.1 Neural Networks 12
3.2 Instance-Based Learning 19
3.3 Ensemble Learning 19
3.4 Support Vector Machines 19
3.5 Bayesian Learning 19

Techniques

Supervised learning is typically broken up into two main schools of algorithms. **Classification** involves mapping between complex inputs (like image of faces) and labels (like `True` or `False`, though they don't necessarily need to be binary) which we call "classes". This is in contrast with **regression**, in which we map our complex inputs to an arbitrary, often-continuous, often-numeric value (rather than a discrete value that comes from a small set of labels). Classification leans more towards data with discrete values, whereas regression is more-universal, being applicable to any numeric values.

Though data is everything in machine learning, it isn't perfect. Errors can come from a variety of places:

- hardware (sensors, precision)
- human element (mistakes)
- malicious intent (willful misrepresentation)
- unmodeled influences



These hidden errors will factor into the resulting model if they're present in the training data. Similarly, they'll cause inaccuracies in evaluation if they're present in the testing data. We need to be careful about how accurately we "fit" the training data, ideally keeping it general enough to flourish on real-world data. A method for reducing this risk of **overfitting** is called **cross-validation**: we can use some of the training data as a "fake" testing set. The "Goldilocks zone" of training is between **underfitting** and overfitting, where the error across both training data and cross-validation data are relatively similar.

Classification

> *Science is the systematic classification of experience.*
>
> — George Henry Lewes, *Physical Basis of Mind*

Breaking down a classification problem requires a number of important elements:

- **instances**, representing the input data from which the overall model will "learn;"
- the **concept**, which is the abstract concept that the data represents (hopefully representable by a well-formed function);
- a **target concept**, which is the "answer" we want: the ability to classify based on our concept;
- the **hypotheses** are all of the possible functions (ideally, we can restrict ourselves from literally *all* functions) we're willing to entertain that may describe our concept;
- some input **samples** pulled from our instances and paired (by someone who "knows") with the *correct* output;
- some **candidate** which is a potential target concept; and
- a **testing set** from our instances that our candidate concept has not yet seen in order to evaluate how close it is to the ideal target concept.

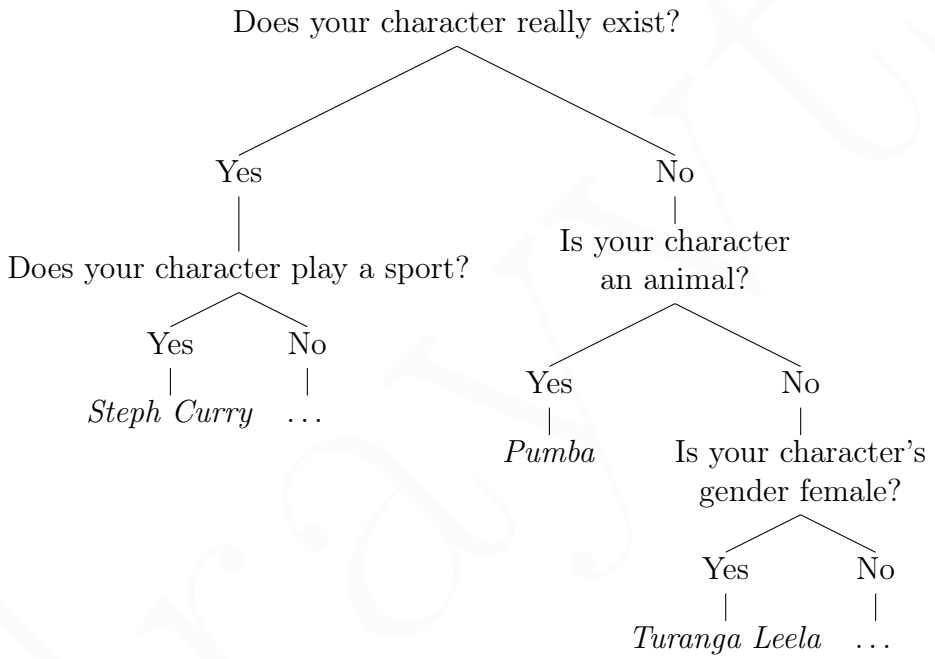
2.1 Decision Trees

Quinlan '86

Decision trees are a form of classification learning. They are exactly what they sound like: trees of decisions, in which branches are particular questions (in which each path down a branch represents a different answer to said question) based on the input, and leaves are final decisions to be made. It maps various choices to diverging paths that end with some decision.

To create a decision tree for a concept, we need to identify pertinent **features** that would describe it well. For example, if we wanted to decide whether or not to eat at a restaurant, we could use the weather, particular cuisine, average cost, atmosphere, or even occupancy level as features.

For a great example of "intelligence" being driven by a decision tree in popular culture, consider the famous "character guessing" AI Akinator. For each yes-or-no question it asks, there are branches the answers that lead down a tree of further questions until it can make a confident guess. One could imagine the following (incredibly oversimplified) tree in Akinator's "brain:"



It's important to note that decision trees are a **representation** of our features. Only after we've formed a representation can we start talking about the **algorithm** that will use the tree to make a decision.

The order in which we apply each feature to our should be correlated with its ability to reduce our space. Just like Akinator divides the space of characters in the world into fiction and non-fiction right off the bat, we should aim to start our decision tree with questions whose answers can sweep away swaths of finer decision-making. For our restaurant example, if we want to spend $\leq \$10$ no matter what, that would eliminate a massive amount of restaurants immediately from the first question.

2.1.1 Getting Answers

The notion of a "best" question is obviously subjective, but we can make an attempt to define it with a little more mathematical rigor. Taking some inspiration from binary search, we could define a question as being good if it divides our data roughly

in half. Regardless of our final decision (heh) regarding the definition of "best," the algorithm is roughly the same:

1. Pick the "best" attribute.
2. Ask the question.
3. Follow the answer path.
4. If we haven't hit a leaf, go to step 1.

Of course the flaw is that we want to *learn* our decision tree based on the data. The above algorithm is for *using* the tree to make decisions. How do we create the tree in the first place? Do we need to search over the (massive) space of all possible decision trees and use some criteria to filter out the best ones?

Given n boolean attributes, there are 2^n possible ways to arrange the attributes, and 2^{2^n} possible answers (since there are 2^n different decisions for each of those arrangements)... we probably want to be a little smarter than that.

2.1.2 Asking Questions: The ID3 Algorithm

ID3 Algorithm, Udacity

If we approach the feature-ranking process greedily, a simple top-down approach emerges:

- $A \leftarrow$ best attribute
- Assign A as the decision attribute (the "question" we're asking) for the particular node n we're working with (initially, this would be the tree's root node).
- For each $v \in A$, create a branch from n.
- Lump the training examples that correspond to the particular attribute value, v, to their respective branch.
- If the examples are perfectly classified with this arrangement (that is, we have one training example per leaf), we can stop.
- Otherwise, repeat this process on each of these branches.

The "information gain" from a particular attribute A can be a good metric for qualifying attributes. Namely, we measure how much the attribute can reduce the overall entropy:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in A} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v) \tag{2.1}$$

where the entropy is calculated based on the probability of seeing values in A:

$$\text{Entropy}(A) = \sum_{v \in A} \Pr[v] \cdot \log \Pr[v] \tag{2.2}$$

We'll dive into these more later when we get to Randomized Optimization, but for now we should just think of this as a measure of how much information an attribute gives us about a system. Attributes that give a lot of information are more valuable, and should thus be higher on the decision tree. Then, the "best attribute" is the one that gives us the maximum information gain: $\max_{A \in \mathcal{A}} \text{Gain}(S, A)$.

Inductive Bias

There are two kinds of bias we need to worry about when designing any classifier:

- **restriction bias**, which automatically occurs when we decide our hypothesis set, $\mathcal{H}$. In this case, our bias comes from the fact that we're only considering functions that can be represented with a decision tree.
- **preference bias**, which tells us what sort of hypotheses *from* our hypothesis set, $h \in \mathcal{H}$, we prefer.

The latter of these is at the heart of inductive bias. Which decision trees—out of all of the possible decision trees in the universe that can represent our target concept—will the ID3 algorithm prefer?

Splits Since it's greedily choosing the attributes with the most information gain from the top down, we can confidently say that it will prefer trees with good splits at the top.

Correctness Critically, the ID3 algorithm repeats until the labels are correctly classified. And though it may be obvious, it's still important to note that it will hence prefer correct decision trees to incorrect ones.

Depth This arises naturally out of the top-heavy split preference, but again, it's still worth noting that ID3 will prefer trees that are shallower or "shorter."

2.1.3 Considerations

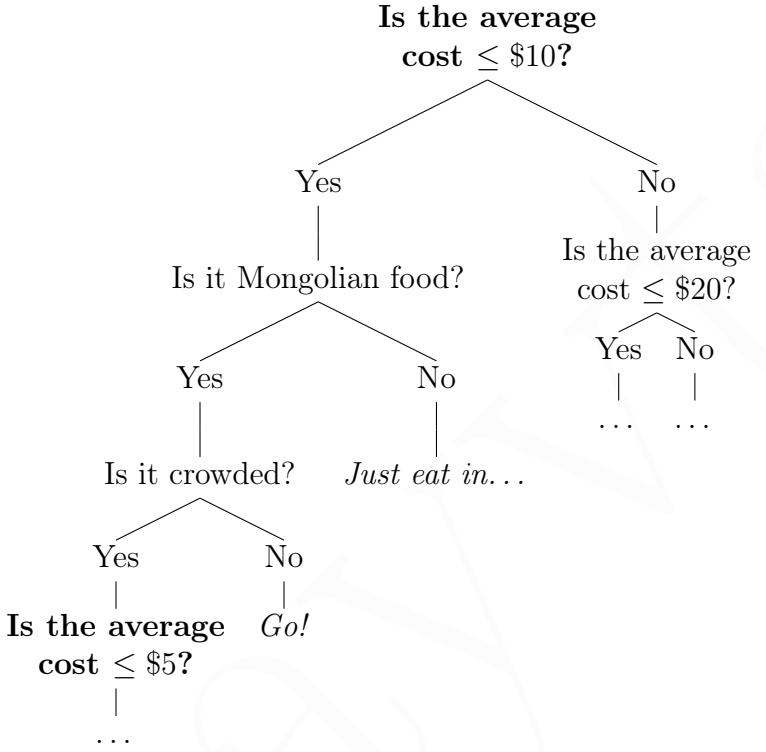
Asking (Continuous) Questions

The careful reader may have noticed the explicit mention of branching on attributes based on *every possible value* of an attribute: $v \in A$. This is infeasible for many features, especially **continuous** ones. For our earlier restaurant example, we may have discretized our "cost" feature into one, two, or three dollar signs (à la Yelp), but what if we wanted to keep them as a raw average dish dollar value instead?

Well, if we're sticking to the "only ask Boolean questions" model, then binning is a viable approach. Instead of making decisions based on a precise cost, we instead make decisions based on a place being "cheap," which we might subjectively define as cost $\in [0, 10)$, for example.

Repeating Attributes

Does it make sense to ask about an attribute more than once down its branch? That is, if we ask about cost somewhere down a path, can (or should) we ask again, later?



With our "proof by example," it's pretty apparent that **yes**, it's acceptable to ask about the same attribute twice. **However**, it really depends on the attribute. For example, we wouldn't want to ask about the weather twice, since the weather will be constant throughout the duration of the decision-making process. With our bucketed continuous values (cost, age, etc.), though, it does make sense to potentially refine our buckets as we go further down a branch.

Stopping Point

The ID3 algorithm tells us to stop creating our decision tree when all of our training examples are classified correctly. That's... a lot of leaf nodes... It may actually be pretty problematic to refine the decision tree to such a point: when we leave our training set, there may be examples that don't fall into an exact leaf. There may also be examples that have identical features but actually have a different outcome; when we're talking about restaurant choices, opinions may differ:

	Weather	**Cost**	**Cuisine**	**Go?**
Alice:	Cloudy	$	Mexican	✓
Bob:	Cloudy	$	Mexican	×

If both of these rows were in our training set, we'd actually get an infinite loop in the naïve ID3 algorithm: it's impossible to classify every example correctly. It makes sense to adopt a termination approach that is a little more general and robust. We want to avoid overfitting our training examples!

If we bubble up the decisions down the branch of a tree back up to its parent node, then **prune** the branch entirely, we can avoid overfitting. Of course, we'd need to make sure that the generalized decision does not increase our training error by too much. For example, if a cost-based branch had all of its children branches based on weather, and all but one of those resulted in the go-ahead to eat, we could generalize and say that we should always eat for the cost branch without incurring a very large penalty for the one specific "don't eat" case.

Adapting to Regression

Decision trees as we've defined them here don't transfer directly to regression problems. We no longer have a useful notion of information gain, so our approach at attribute sorting falls through. Instead, we can rely on purely statistical methods (like variance and correlation) to determine how important an attribute is. For leaves, too, we can do averages, local linear fit, or a host of other approaches that mathematically generalize with no regard for the "meaning" of the data.

Regression

> *We stand the risk of regression, because you refused to take risks. So life demands risks.*
>
> — Sunday Adelaja

When we free ourselves from the limitation of small discrete quantities of labels, we are open to approximate a much larger range of functions. Machine learning techniques that use regression can approximate real-valued and continuous functions.

In supervised learning, we are trying to perform *inductive* reasoning, in which we try to figure out the abstract bigger picture from tiny snapshots of the world in which we don't know most of the rules (that is, approximate a function from input-output pairs). This is in stark contrast with *deductive* reasoning, through which individual facts are combined through strict, rigorous logic to come to bigger conclusions (think "if *this*, then *that*").

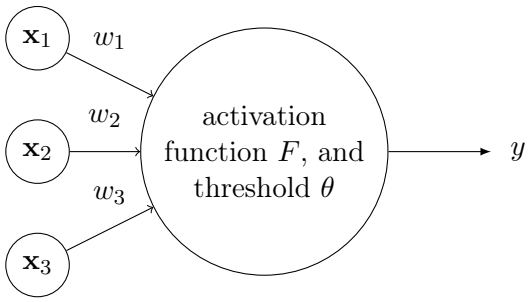
3.1 Neural Networks

Let's dive right into the deep end and learn about how a **neural network** works.

Neurons in the brain can be described relatively succinctly from a high level: a single neuron is connected to a bunch of other neurons. It accumulates energy and once the amount is bigger than the "**activation threshold**," the neuron fires, sending energy to the neurons its connected to (potentially causing a cascade of firings). Artificial neural networks take inspiration from biology and are somewhat analogous to neuron interactions in the brain, but there's really not much benefit to looking at the analogy further.

The basic model of an "artificial neuron" is a function powered by a series of inputs $\mathbf{x}_i$, and weights w_i, that somehow run through F and produce an output y:

Typically, the activation function "fires" based on a **firing threshold** θ.



3.1.1 Perceptron

The simplest, most fundamental activation function procudes a binary output (so $y \in \{0, 1\}$) based on the weighted sum of the inputs:

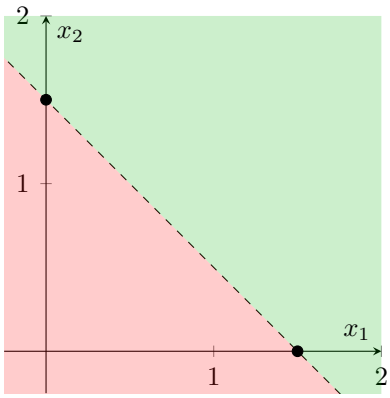
$$F(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n, w_1, w_2, \dots, w_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i \mathbf{x}_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

This is called a **perceptron**, and it's the foundational building block of neural networks going back to the 1950s.

> Example 3.1: **Basic Perceptron**
>
> Let's quickly validate our understanding. Given the following input state:
>
> $$x_1 = 1, w_1 = \frac{1}{2}$$
> $$x_2 = 0, w_2 = \frac{3}{5}$$
> $$x_3 = -1.5, w_3 = 1$$
>
> and the firing threshold $\theta = 0$, what's the output?
>
> Well, pretty simply, we get
>
> $$y = \left[1\left(\frac{1}{2}\right) + 0\left(\frac{3}{5}\right) + (-1.5) \cdot 1 = -\frac{1}{2}\right] < 0 = 0$$
>
> The perceptron doesn't fire!

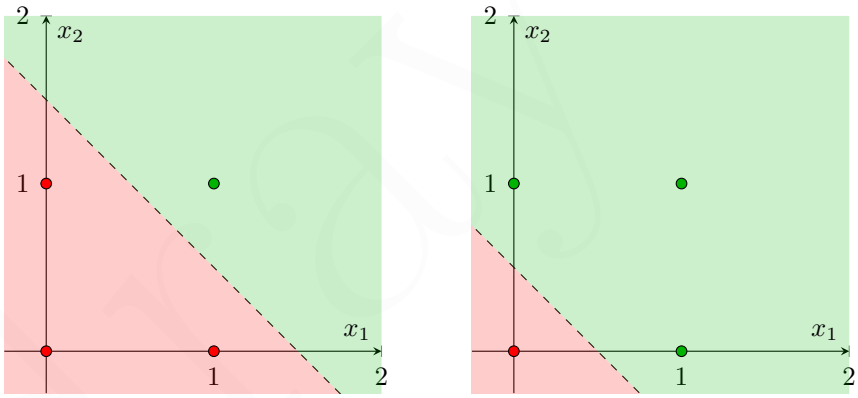
What kind of functions can we represent with a perceptron? Suppose we have two inputs, x_1 and x_2, along with equal weights $w_1 = w_2 = \frac{1}{2}, \theta = \frac{3}{4}$. For what values of x_i will we get an activation?

Well, we know that if $x_2 = 0$, then we'll get an activation for anything that makes $x_1 w_1 \geq \theta$, so $x_1 \geq \theta/w_1 \geq 1.5$. The same rationale applies for x_2, and since we know that the inequality is linear, we can just connect the dots:



Thus, a perceptron is a linear function (each $w_i \mathbf{x}_i$ term is linear), and so it can be used to compute the **half-plane** boundary between its inputs.

This very example actually computes an interesting function: if the inputs are binary (that is, $x_1, x_2 \in \{0, 1\}$), then **this actually computes binary AND** operation. The only possible outputs are marked accordingly; only when $x_1 = x_2 = 1$ does $y = 1$! We can actually model OR the same way with different weights (like $w_1 = w_2 = \frac{3}{2}$).



Note that we "derived" OR by adjusting $w_{1,2}$ until it worked, though we could've also adjusted θ. This might trigger a small "aha!" moment if the idea of induction from stuck with you: if we have some known input/output pairs (like the truth tables for the binary operators), then we can use a computer to rapidly guess-and-check the weights of a perceptron (or perhaps an entire neural network...?) until they accurately describe the training pairs as expected.

Combining Perceptrons

To build up an intuition for how perceptrons can be combined, let's binary XOR. It can't be described by a single perceptron because it's not a linear function; however, it *can* be described by several!

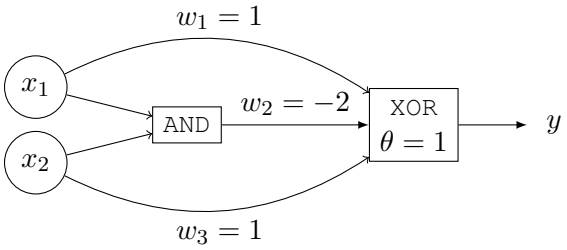
Intuitively, XOR is like OR, except when the inputs succeed

x	y	$x \oplus y$
1	1	0
1	0	1
0	1	1
0	0	0

Table 3.1: The truth table for XOR.

under AND... so we might imagine that XOR $\approx$ OR$-$AND.

So if x_1 and x_2 are both "on," we should take away the result of the AND perceptron so that we fall under the activation threshold. However, if only one of them is "on," we can proceed as normal. Note that the "deactivation weight" needs to be equal to the sum of the other weights in order to effectively cancel them out, as shown Figure 3.1.



x_1	x_2	AND	y
1	1	1	$1 \cdot 1 + -2 \cdot 1 + 1 \cdot 1 = 0$
1	0	0	$1 \cdot 1 + -2 \cdot 0 + 1 \cdot 0 = 1$
0	1	0	$1 \cdot 0 + -2 \cdot 0 + 1 \cdot 1 = 1$
0	0	0	$1 \cdot 0 + -2 \cdot 0 + 1 \cdot 0 = 0$

Figure 3.1: A neural network modeling XOR and its summations for all possible bit inputs. The final column in the table is the summation expression for perceptron activation, $w_1 x_1 + w_2 F_{\text{AND}}(x_1, x_2) + w_3 x_2$.

Learning Perceptrons

Let's delve further into the notion of "training" a perceptron network that we alluded to earlier: twiddling the w_is and θs to fit some known inputs and outputs. There are two approaches to this: the **perceptron rule**, which operates on the post-activated y-values, and **gradient descent**, which operates on the raw summation.

Perceptron Rule Suppose $\hat{y}$ is our perceptron's current output, while y is its desired output. In order to "move towards" the goal y, we adjust our w_is accordingly based on the "error" between y and $\hat{y}$. That is,

$$\begin{gathered} \text{define } \hat{y} = \left(\textstyle\sum_i w_i x_i \geq 0\right) \\ \text{then use } \Delta w_i = \eta \left(y - \hat{y}\right) x_i \\ \text{to adjust } w_i \leftarrow w_i + \Delta w_i \end{gathered}$$

where $\eta > 0$ is a **learning rate** which influences the size of the Δw_i adjustment made every iteration.

Notice the absense of the activation threshold, θ. To simplify the math, we can actually treat it as part of the summation: a "fake" input with a fixed weight of -1, since:

$$\begin{aligned} \textstyle\sum_i^n w_i x_i = \theta &\implies \textstyle\sum_i^n w_i x_i - \theta = 0 \\ &\implies \textstyle\sum_i^{n+1} w_i x_i = 0 \qquad \text{where } w_{n+1} = -1 \text{ and } x_{n+1} = \theta \end{aligned}$$

This extra input value is now called the **bias** and its weight can be tweaked just like the other inputs.

Notice that when our output is correct, $y - \hat{y} = 0$ so there is no effect on the weights. If $\hat{y}$ is too big, then our $\Delta w_i < 0$, making that $w_i x_i$ term smaller, and vice-versa if $\hat{y}$ is too small. The learning rate controls our adjustments so that we take small steps in the right direction rather than overshooting, since we don't know how close we are to fixing $\hat{y}$.

Claim 3.1. *If a dataset is **linearly-separable** (that is, able to be separated by a line), then a perceptron can find it with a finite number of iterations by using the perceptron rule.*

Of course, it's impossible to know whether a sufficiently-large "finite" number of a iterations has occurred, so we can't use this fact to determine whether or not a dataset is linearly-separable, but it's still good to know that it will terminate when it can.

Gradient Descent We still need something in our toolkit for datasets that aren't linearly-separable. This time, we'll operate on the unthresholded summation since it gives us far more information about how close (or far) we are from triggering an activation. We'll use a as shorthand for the summation: $a = \sum_i w_i x_i$.

Then we can define an error metric based on the difference between a and each expected output: we sum the error for each of our input/output pairs in the dataset D. That is,

$$E(\mathbf{w}) = \frac{1}{2} \sum_{(x,y) \in D} (y - a)^2$$

Let's use our good old friend calculus to solve this via **gradient descent**. A function is at its minimum when its derivative is zero, so we'll take the derivative with respect to a weight:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \left[\frac{1}{2} \sum_{(x,y) \in D} (y - a)^2 \right] \\
&= \sum_{(x,y) \in D} (y - a) \cdot \frac{\partial}{\partial w_i} \left[- \sum_j w_j x_j \right] && \text{chain rule, and only } a \text{ is in terms of } w_i \\
&= \sum_{(x,y) \in D} (y - a)(-x_i) && \text{when } j \neq i, \text{ the derivative will be zero} \\
&= - \sum_{(x,y) \in D} (y - a) x_i && \text{rearranged to look like the perceptron rule}
\end{aligned}$$

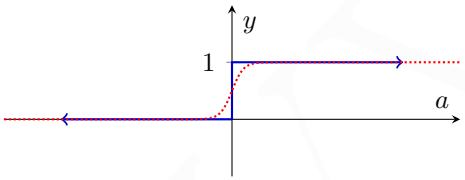
Notice that we essentially end up with a version of the perceptron rule where $\eta = -1$, except we now use the summation a instead of the binary output $\hat{y}$. Unlike the perceptron rule, we have no guarantees about finding a separation, but it is far more robust to non-separable data. In the limit (thank u Newton very cool), though, it will converge to a local optimum.

To reiterate our learning rules, we have:

$$\Delta w_i = \eta(y - \hat{y})x_i \tag{3.1}$$
$$\Delta w_i = \eta(y - a)x_i \tag{3.2}$$

3.1.2 Sigmoids

The similarity between the learning rules in (3.1) and (3.2) begs the question, why didn't we just use calculus on the thresholded $\hat{y}$?



The simple answer is that the function isn't differentiable. Wouldn't it be nice, though if we had one that was very similar to it, but smooth at the hard corners that give us problems? Enter the **sigmoid** function:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{3.3}$$

By introducing this as our activation function, we can use gradient descent all over the place. Furthermore, the sigmoid's derivative itself is beautiful (thanks to the fact that $\frac{d}{dx}e^x = e^x$):

$$\dot{\sigma}(a) = \sigma(a)(1 - \sigma(a))$$

Note that this isn't the only function that smoothly transitions between 0 and 1; there are other activation functions out there that behave similarly.

3.1.3 Neural Network Structure

Now that we have the ability to put together differentiable individual neurons, let's look at what a large neural network presents itself as. In Figure 3.2 we see a collection of sigmoid units arranged in an arbitrary pattern. Just like we combined two perceptrons to represent the non-linear function XOR, we can combine a larger number of these sigmoid units to approximate an arbitrary function.

Because each individual unit is differentiable, the entire mapping from $\mathbf{x} \mapsto y$ is differentiable! The overall error of the system enables a bidirectional flow of information:

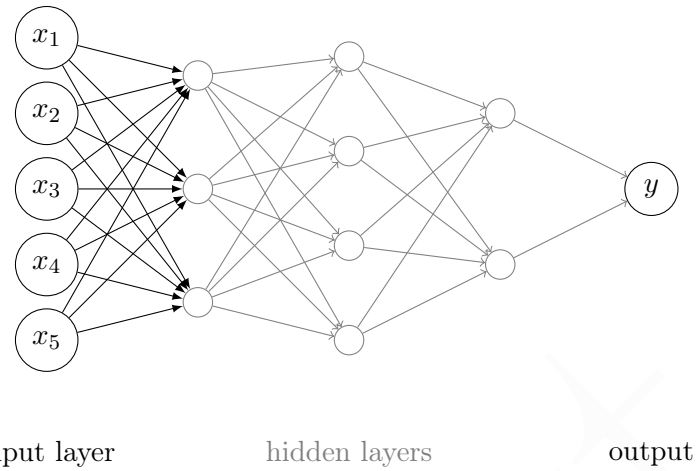


Figure 3.2: A 4-layer neural network with 5 inputs and 3 hidden layers.

the error of y impacts the last hidden layer, which results in its own error, which impacts the second-to-last hidden layer, etc. This layout of computationally-beneficial organizations of the chain rule is called **back-propagation**: the error of the network propogates to adjust each unit's weight individually.

Optimization Methods

It's worth reiterating that we've departed from the guarantees of perceptrons since we're using $\sigma(a)$ instead of the binary activation function; this means that gradient descent can get stuck in local optima and not necessarily result in the best *global* approximation of the function in question.

Gradient descent isn't the only approach to training a neural network. Other, more advanced methods are researched heavily. Some of these include **momentum**, which allows gradient descent to "gain speed" if it's descending down steep areas in the function; higher-order derivatives, which look at combinations of weight changes to try to grasp the bigger picture of how the function is changing; randomized optimization; and the idea of penalizing "complexity," so that the network avoids overfitting with too many nodes, layers, or even too-large of weights.

3.1.4 Bias

What kind of problems are neural networks appropriate for solving?

Restriction Bias A neural network's restriction bias (which, if you recall, is the representation's ability to consider hypotheses) is basically non-existent if you use sigmoids, though certain models may require arbitrarily-complex structure.

We can clearly represent Boolean functions with threshold-like units. Continuous

functions with no "jumps" can actually be represented with a single hidden layer. We can think of a hidden layer as a way to stitch together "patches" of the function as they approach the output layer. Even arbitrary functions can be approximated with a neural network! They require two hidden layers, one stitching at seams and the other stitching patches.

This lack of restriction does mean that there's a significant danger of overfitting, but by carefully limiting things that add complexity (as before, this might be layers, nodes, or even the weights themselves), we can stay relatively generalized.

Preference Bias On the other hand, we can't yet answer the question of the preference bias of a neural network (which, if you recall, describes which hypotheses *from the restricted space* are preferred). We discussed the algorithm for updating weights (gradient descent), but have yet to discuss how the weights should be initialized in the first place.

Common practice is choosing small, random values for our initial weights. The randomness allows for variability so that the algorithm doesn't get stuck at the same local minima each time; the smallness allows for relative adjustments to be impactful and reduces complexity.

Given this knowledge, we can say that neural networks—when all other things are equal—prefer simpler explanations to complex ones.

This idea is an embodiment of **Occam's Razor**:

> *Entities should not be multiplied unnecessarily.*

More colloquially, it's often expressed as the idea that the simpler explanation is likelier to be true.

3.2 Instance-Based Learning

3.3 Ensemble Learning

3.4 Support Vector Machines

3.5 Bayesian Learning

PART II

Unsupervised Learning

Randomized Optimization

Index of Terms

A

activation threshold **12**

B

back-propagation **18**
bias **16**

C

classification **5**
cross-validation **5**

F

features **7**
firing threshold **12**
function approximation **4**

G

gradient descent **15**, **16**, 19

L

learning rate **15**

M

momentum **18**

N

neural network **12**

O

Occam's Razor **19**
overfitting **5**, 11

P

perceptron **13**
perceptron rule **15**, 16
preference bias **9**, 19

R

randomized optimization 18
regression **5**, 12
restriction bias **9**, 18

S

sigmoid **17**
supervised learning **4**

U

underfitting **5**