GENERAL NOTES

ANDREW S. HARD[1]

October 14, 2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

[1] *Department of Physics, University of Wisconsin, Madison, United States of America*

1 INTRODUCTION

As a fifth year graduate student, I have come to the realization that significant portions of my skill-set and knowledge base are highly specialized. I intend to pursue a career outside of my field of study (experimental high-energy particle physics). In order to enhance my future job prospects, I decided that it would be useful to review basic concepts in computer science, statistics, mathematics, and machine learning. Most of the derivations and explanations will be borrowed from other sources. A list of references is currently being compiled.

2 BASIC STATISTICS

2.1 Deduction and Induction

Deduction: if $A \rightarrow B$ and A is true, then B is true.
Induction: if $A \rightarrow B$ and B is true, then A is more plausible.

2.2 Conditional Probability and Bayes' Theorem

The **conditional probability** of an event A assuming that B has occured, denoted $P(A|B)$, is given by:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \tag{1}$$

Via multiplication:

$$P(A \cap B) = P(A|B)P(B). \tag{2}$$

This makes sense. The probability of A *and* B is the probability of A *given* B times the probability of B. Or, just as reasonably, the probability of A *and* B is the probability of B *given* A times the probability of A. As an aside, the equation above can be generalized:

$$P(A \cap B \cap C) = P(A|B \cap C)P(B \cap C) = P(A|B \cap C)P(B|C)P(C). \tag{3}$$

Back to the derivation, since $P(A \cap B) = P(B \cap A)$, we can use Equation 2:

$$P(A|B)P(B) = P(B|A)P(A). \tag{4}$$

And re-arranging gives **Bayes' Theorem**:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} = \frac{P(A)}{P(B)}P(B|A). \tag{5}$$

In this equation, $P(A)$ is the **prior probability**, the initial degree of belief in A, or the probability of A before B is observed. It is essentially the probability that hypothesis A is true before evidence is collected. The **posterior probability** $P(A|B)$ is the degree of belief after taking B into account.

In a nutshell, Bayes' theorem is a method for calculating the validity of beliefs based on available evidence. "Initial belief plus new evidence equals new and improved belief."

Bayes' theorem describes the probability of an event, based on conditions that might be relevant to the event. With the Bayesian probability interpretation the theorem expresses how a subjective degree of belief should rationally change to account for evidence (**Bayesian inference**). Note: in the **Frequentist Interpretation**, probability measures a proportion of outcomes, not probability of belief.

Bayes' theorem can also be interpreted in the following manner: A is some hypothesis, and B is the evidence for that hypothesis.

The equation makes intuitive sense, particularly in the last form in which it is presented. If $P(B) \gg P(A)$, the probability $P(A|B)$ will be small because B *usually* occurs without A occurring. Even if $P(B|A)$ were 100%, the posterior probability would still be small. On the other hand, if $P(A) \gg P(B)$, $P(A|B)$ is high, even if the conditional probability $P(B|A)$ is smaller.

Nate Silver has a discussion of Bayesian statistics in "The Signal And The Noise", p245. Let x be the initial estimate of the hypothesis likelihood (**prior probability**). In the book, x is the initial estimate that your spouse is cheating. y is a conditional probability - the probability of an observation being made given that the hypothesis is true. So y in his case was the probability of underwear appearaing conditional on the spouse cheating. z is the probability of an observation being made given that the hypothesis is false. So z was the probability of underwear appearing if the spouse was not cheating. The **posterior probability** is the revised estimate of the hypothesis likelihood. In Silver's case, the likelihood that the spouse is cheating on you, given that underwear was found. The posterior probability is given by the expression:

$$x' = \frac{xy}{xy + z(1-x)}. \tag{6}$$

In Silver's formulation, the denominator is equivalent to the total probability of an observation being made (total probability of underwear being found). So it is the probability of the hypothesis being true times the associated conditional probability of the observation of underwear plus the probability of the hypothesis being false times the associated conditional probability of the observation.

Note that this method can be applied recursively by substituting x' for x in the expression. We can also come up with a recurrent form of Bayes theorem as formulated in Equation 5 by letting $P(A_{n+1}) = P(A_n|B)$.

Yet another formulation, which is explained by the logic in the previous paragraph, is given by:

$$P(A|X) = \frac{P(X|A)P(A)}{P(X|A)P(A) + P(X|notA)P(notA)} \tag{7}$$

$P(A|X)$ is the probability of hypothesis A, given a positive observation X. $P(X|A)$ is the probability of the observation of X under hypothesis A. $P(A)$ is the probability of the hypothesis.

The derivation below is from mathworld's article on Bayes' Theorem. In this example, let A and S be sets. Furthermore, let

$$S \equiv \bigcup_{i=1}^{N} A_i, \tag{8}$$

so that A_i is an event in S, and $A_i \cap A_j = \emptyset \; \forall \; i \neq j$. Then:

$$A = A \bigcap S = A \bigcap (\bigcup_{i=1}^{N} A_i) = \bigcup_{i=1}^{N} (A \cap A_i) \tag{9}$$

Then we can compare the probabilities using the Law of Total Probability:

$$P(A) = P(\bigcup_{i=1}^{N} (A \cap A_i)) = \sum_{i=1}^{N} P(A \cap A_i) = \sum_{i=1}^{N} P(A_i)P(A|A_i) \tag{10}$$

Using substitution into Bayes' Theorem (Equation 5), we have a new form:

$$P(A_j|A) = \frac{P(A_j)P(A|A_j)}{\sum_{i=1}^{N} P(A_i)P(A|A_i)}. \tag{11}$$

What is the meaning of this? The denominator is simply $P(A)$, expressed as the sum of its constituents. So this reduces exactly to Bayes theorem as in Equation 5, with $A \to A_j$ and $B \to A$.

"In many cases, estimating the prior is just guesswork, allowing subjective factors to creep into your calculations. You might be guessing the probability of something that, unlike cancer, does not even exist, such as strings, multiverses, inflation or God. You might then cite dubious evidence to support your dubious belief. In this way, Bayes' theorem can promote pseudoscience and superstition as well as reason." From http://blogs.scientificamerican.com/cross-check/bayes-s-theorem-what-s-the-big-deal/

2.3 Correlation and Covariance

Correlation and covariance are similar. Both concepts describe the degree to which two random variables or sets of random variables tend to deviate from their expected values in similar ways.

Let X and Y be two random variables, with means μ_X and μ_Y and standard deviations σ_X and σ_Y, respectively. The **covariance** is defined (using $\langle\rangle$ to denote expectation value):

$$\sigma_{XY} = \langle (X - \langle X \rangle)(Y - \langle Y \rangle) \rangle. \tag{12}$$

Similarly, the **correlation** of the two variables is defined:

$$\rho_{XY} = \frac{\langle (X - \langle X \rangle)(Y - \langle Y \rangle) \rangle}{\sigma_X \sigma_Y}. \tag{13}$$

So correlation is dimensionless, while covariance is the multiple of the units of the two variables. **Variance** is simply the covariance of a variable with itself (σ_{XX} is usually denoted σ_X^2, the square of the standard deviation). The correlation of a variable with itself is always 1, except in the degenerate case where the two variances are zero and the correlation does not exist.

3 INFORMATION THEORY

Information theory deals with quantifying information. It was developed by Claude Shannon with the goal of establishing limits on signal processing operations such as data compression, reliable storage, and communcation (cite wiki).

The basic unit of information is the **bit**, a portmanteau of binary digit. It can have values of 0 or 1, and could represent boolean states (false/true) or activation states (off/on) among other things. **Bit-length** refers to the length of a binary number. Table 3 lists common data types from the IEEE Standard For Floating Point Arithmetic (IEEE 754).

data type	bit length	bit assignment
signed int	32	Bit string with range $[-2^{31}, 2^{31}-1]$
unsigned int	32	Bit string with range $[0, 2^{32}-1]$
float	32	1 sign, 8 exponent , 24 significand precision
double	64	1 sign, 11 exponent, 53 significand precision

3.1 Entropy

Shannon entroy H - the average number of bits per symbol needed to store or communicate a message. "Bits assumes $\log_2$. "It is analogous to intensive physical entropies like molar and specific entropy S_0 but not extensive physical entropy S."

There's also entropy H times the number of symbols in a message. This measure of entropy is in bits. It is analogous to absolute (extensive) physical entropy S.

Landauer's Principle: A change in H*N is equal to a change in physical entropy when the information system is perfectly efficient.

What does entropy mean? It is a measure of the uncertainty involved in predicting the value of a random variable. Example: a coin flip with 50 % probabilities of heads or tails provides less information than a die roll with a $\frac{1}{6}$ probability per side.

The Shannon entropy H in units of bits per symbol is given by

$$H = -\sum_i f_i \log_2(f_i) = -\sum_i \log_2(\frac{c_i}{N}). \tag{14}$$

In this equation, i is an index over distinct symbols, f_i is the frequency of symbols occurring in the message, and c_i is the number of times the i^{th} symbol occurs in the message of length N. HN is the entropy of a message. This has units of bits (not bits per symbol).

Example: a Bernoulli (binary) trial where $X = 1$ occurs with some probability. The entropy $H(X)$ of the trial is maximized when the two possible outcomes are equally probable ($P(X = 1) = 0.5$).

Boltzmann H	**Shannon H**
physical systems of particles	information systems of symbols
entropy per particle	entropy per symbol
probability of a microstate	probability of a symbol
Extensive physical entropy S	Entropy of file or msssage HN

3.2 Numerical Systems

We usually count in base 10 (maybe because we have 10 fingers and 10 toes?). But there are other counting systems available. For instance, computers are programmed with binary instructions because transistors can have "on" or "off" gate positions. Binary, octal, decimal, and hexadecimal are all examples of **positional numerical systems**. The **radix** refers to the number of symbols that can be used at each position in a number. For the decimal system, the radix is 10 because each decimal place can accomodate the numbers 0-9. The radix is 2 for binary, because each position can accomodate the numbers 0 or 1. For hexadecimal, it is 16 (0-9, as well as A-F). And so on...

The equation below shows how to convert a hexadecimal (base 16) number into a decimal (base-10) number.

$$\begin{aligned} 2AF3 &= (2_{16} \cdot 16^3) + (A_{16} \cdot 16^2) + (F_{16} \cdot 16^1) + (3_{16} \cdot 16^0) \\ &= (2 \cdot 16^3) + (10 \cdot 16^2) + (15 \cdot 16^1) + (3 \cdot 16^0) \\ &= 10995 \end{aligned} \tag{15}$$

Here's another example for converting from a decimal base to a binary base.

$$\begin{aligned} 1035 &= (1_{10} \cdot 10^3) + (0_{10} \cdot 10^2) + (3_{10} \cdot 10^1) + (5_{10} \cdot 10^0) \\ &= (1 \cdot 2^{10}) + (1 \cdot 2^3) + (1 \cdot 2^1) + (1 \cdot 2^0) \\ &= 10000001011 \end{aligned} \tag{16}$$

4 DATA STRUCTURES

4.1 Java Basics

In Java, there are **primitive types** and **reference types**. Primitive types are like `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`. Reference types are **arrays** and **classes**.

When you declare a variable with a reference type, you get space for a reference or **pointer** to that type, not for the type itself. Use `new` to get space for the type itself. Non-primitive types are really pointers. Assigning 1 variable to another can cause **aliasing** (two or more different names reference the same object).

For each **primitive type** there is a "box class" or "wrapper class". For instance, the `Integer` class contains one `int` value. Java **autoboxing** converts between the primitive and wrapper classes as appropriate.

Java also has **visibility modifiers**, which specify how code can be accessed.

- *public* accessible to any code
- *private* accessible to code within same class
- *protected* accessible to code within same class or subclasses
- *public* accessible to code within same package (directory)

Numerical comparisons can be performed using the "<", ">", "=" signs for `int`, `float`, `double` primitives. However, the **Comparable** interface should be used for objects in Java (`String`, `Integer`, `Double`). The `compareTo` method returns negative value if "less than", zero if "equal to", and positive if "greater than".

4.2 Complexity

Complexity is a way to gauge the performance of an algorithm: how much of the computing resources are used when a program executes. **Complexity** measures how the resource requirements scale as the problem gets larger. The time requirement is basically proportional to the number of basic operations such as '+', '*', one assignment, one logical test, one read, or one write. The **problem size** is related to the number of important factors. In sorting, for example, this could be the number of elements to sort.

We are usually interested in the worst case number of operations, not the exact number of operations, which can vary. The average and best cases do provide useful information, however. We use the *big-O* notation to write complexity: $O(N)$ is linear complexity, $O(N^2)$ is quadratic complexity, etc.

Amortized complexity is the total expense per operation, but is evaluated over many operations. This is useful when operations are occasionally much more expensive than on average. For instance, insertions in a hash table typically take constant time. However, if the array implementation needs to be expanded, the operation might take $O(N)$ time for a single operation. Amortized complexity is just a way of describing the worst case on-average behavior.

4.3 Abstract Data Types

Abstract Data Types (**ADTs**) separate the *specification* of a data structure from the *implementation* of that data structure. The specification refers to the methods and operations that can be performed on it, while the implementation refers to how those features are programmed. The advantage to using ADTs is that code is more reusable: the implementation of certain code can be changed without changing the programs that use the code. An ADT corresponds to a class or many implementing classes. Operations on an ADT are the class's **public methods**. Example operations on an ADT include: initialize, add data, access data, remove data.

ADTs have 2 parts.

- public/external - conceptual picture & conceptual operations
- private/internal - the representation (how the structure is stored) and the implementation of operations (code)

To implement an interface in Java: (1) include "`implements InterfaceName`" in the class declaration, changing `InterfaceName` as appropriate, (2) define a public method for the class for each method signature in the interface.

To specify that something is a subclass, use `extends`. This says that the current class definition inherits from the specified class. For instance, `MyException` inherits from the `Exception` class in the example below.

```
public class MyException extends Exception {...};
```

4.4 Exceptions

Errors can arise due to *user error* (e.g. wrong inputs) or *programmer error* (e.g. buggy code). It is desirable to pass the error up from low-level methods to a level where it can be handled properly. In C++, methods usually just return a special value to indicate an error. Requires calling code to check for error. In Java, expections allow error testing in the code.

The idea is to **throw** an **exception** when an error is detected. The code causing the error stops executing immediately and control is transferred to the **catch clause** for that type of exception of the first enclosing **try block** that has such a clause. The try block might be in the current method or in a method that called the current one. The exception is passed up the call chain, all the way up to main or the Java virtual machine if necessary (at which point the program stops and an error message appears). The code below provides an example.

```
try {
  // code that might cause exceptions.
} catch( ExceptionType1 id1 ) {
  // statement to handle this exception type.
} catch( ExceptionType2 id2 ) {
  // statement to handle this exception type.
} finally {
  // statement to execute every time this try block executes.
}
```

Each catch clause specifies one type of exception and provides a name for it, like method objects. The finally clause is optional. It *always* executes if the try block is entered. Note: you can have zero catch clauses and still have a finally clause.

Java's standard libraries have several built-in exceptions: ArithmeticException, ClassCastException, IndexOutOfBoundsException, NullPointerException, FileNotFoundException...

All exceptions are either **checked exceptions** or **unchecked exceptions**. If a method includes code that could cause a checked exception to be thrown: (1) the exception must be declared in the method header using a **throws** clause, or (2) the code that might cause the exception to be thrown must be in a try block with a catch clause for that exception. Otherwise you will get a compiler error. Unchecked exceptions do not need to be listed in the throws clause of a method. The code below provides an example.

```
  public static void main( string[] args ) throws FileNotFoundException,
EOFException {
    // those two exceptions might be thrown here.
  }
```

Unchecked exceptions are subclasses of RuntimeException, while checked exceptions are subclasses of Exception only, not RuntimeException. Most built-in Java exceptions are unchecked, with the exception of I/O exceptions, which are checked. user defined exceptions should usually be checked.

Java exceptions are objects, and as such a new exception is made by defining a new instantiable class. It must be a subclass of Throwable, usually a subclass of Exception. An example is given below.

```
public class EmptyStackException extends Exception {
  public EmptyStackException() { super(); }
  public EmptyStackException(string message) { super(message); }
};
```

At the point in code where an error is detected, the exception is thrown using a throw statement.

```
if (...)  { throw new EmptyStackException(); }
```

4.5 List Structure

4.5.1 *Basic List*

A **List** structure is an ordered collection of items of some element type *E*. One advantage of lists over arrays is that they are variable size. A disadvantage is that you cannot have an empty list with non-zero size. Lists only hold objects, whereas arrays hold any type (even primitive types such as `int`, `char`).

The table below provides the public interface for the `ListADT`.

`void add(int pos, E item);`	Add an item at a specified position
`void add(E item);`	Append item to end of list
`boolean contains(E item);`	Check if list contains item
`int size();`	Get the number of items in the list
`boolean isEmpty();`	Check if list contains zero items
`E get(int pos);`	Get the item at the specified position
`E remove(int pos);`	Remove, return item at specified pos.

To instantiate `ListADT`, for a list of `Integer` type items named `number` using the `ArrayList` class:

```
ListADT<Integer> numbers = new ArrayList<Integer>();.
```

`Iterator` is a defined interface in `java.util`. Iterating accesses each item in a list in turn. The interface is `Iterable`. Every java class that implements Iterable (from `java.lang` provides an `iterator` method that returns an `Iterator` object for that collection. For example, the `List<E>` interface has an `iterator()` method:

```
Iterator<string> itr = words.iterator(); //words is a List<String>
```

The easiest way to implement an iterator for a class is usually to define a new class. For example, in the `SimpleArrayList` class, the `iterator()` method should return a new `SimpleArrayListIterator` object.

Here's a simple overview of an iterator class:

```
  import java.util.Iterator;
  import java.util.NoSuchElementException;
  public class SimpleArrayListIterator implements Iterator<Object>
{
    // methods:
    public boolean hasNext(){...}
    public Object next(){...}
    public void remove(){...}
  };
```

4.5.2 *Linked Lists*

Linked lists are a type of list in which each item in the list contains the item data as well as a reference to the next element in the list. The list is built using `Listnode` objects that store data as well as the next listnode. The `getNext()` method returns the next item in the list, which can also be set using the `setNext(Listnode<E> n)` method.

Linked lists often have a **header node** – a dummy node at the front of the list. Header nodes do NOT contain data, the only include a reference for the next item. A **tail pointer** references the last listnode.

There are many variants of linked lists, including doubly linked lists (which have a reference to the previous and next elements) and circularly linked lists (where the tail node `next` is linked to the first node).

4.6 Stacks and Queues

Stacks and queues are like lists, but with more limited operations. They can be implemented using an array or linked list.

4.6.1 *Stacks*

A **stack** is a stack of items. Only the top item can be accessed. Items are added to the top of a stack. It is a basic **LIFO** structure: a last-in-first-out data type. A basic stack interface is provided below.

```
public interface StackADT<E> {
  boolean isEmty(); //return true iff empty
  void push(E obj); // add object to top of stack
  E pop() throws EmptyStackException; //remove
  E peek() throws EmptyStackException; // see
};
```

4.6.2 *Queue*

A **queue** is a structure similar to a line or queue. Items go in one side and out the other. It is a basic **FIFO** structure: a first-in-first-out data type. Items are added to the *rear* of the queue and removed from the *front* of the queue.

```
public interface QueueADT<E> {
  boolean isEmty(); //return true iff empty
  void enqueue(E obj); // add object to end of queue
  E dequeue() throws EmptyQueueException; //remove from front
};
```

4.7 Recursion

A method is **recursive** if it can call itself either directly or indirectly. Recursive code is often *simpler*, but not faster or more efficient. There are a few rules associated with recursion.

- *Recursion rule 1*: recursive methods must have a **base case** - a condition under which no recursive call is made. This prevents infinite recursion.

- *Recursion rule 2*: every recursive method **make progress** towards a base case.

At runtime, a stack of activation records (AR) is maintained (one for each active method). This is also named the **call stack**. It includes the method's parameters, local variables, and return address. When a method is called, its AR is pushed onto the stack. When too many methods are called, a **stack overflow** error is produced.

A data structure is recursive if it can be defined in terms of itself.

4.8 Tree Structure

So far, the structures discussed have been **linear data structures**. These incude lists, stacks, queues, and other structures in which items are stored with one item following another. **Trees** are an example of a **nonlinear data structure**. More than one item can follow from a single item. Furthermore, the number of following items can vary from one item to another. Trees are useful because they can provide *fast access* to information in a database.

Some tree terminology is provided in the table below.

Node	A single element of the tree
Edge	Connection between two nodes
Root	Top-most node of the tree (has no **parent**)
Leaf	bottom nodes (has no **children**)
Ancestors	The preceding nodes in a tree
Descendants	The succeeding nodes in the tree
Path	A sequence of conected nodes
Path Length	Number of nodes in the path
Height	Length of *longest* path from root to a leaf
Depth	Path length from a given node back to root

4.8.1 *Binary Trees*

Binary trees are a special type of tree in which (1) each node has 0, 1, or 2 children and (2) each child is either a *right* or *left* child.

Programming a binary tree usually requires a tree class and a tree node class. The `BinaryTreenode` class only requires 3 fields: `data`, `leftChild`, and `rightChild`. A generic tree, by comparison, could just use a list of nodes for the children.

There are several ways to traverse a tree, as shown in the table below. V=view/visit, L=go to left child, R=go to right child.

Name	*for binary tree*	*for other tree*
pre-order	VLR	VC
post-order	LRV	CV
level-order	left to right from top to bottom	
in-order	LVR	(binary only)

4.8.2 *Binary Search Trees*

A **binary search tree** (BST) is a type of binary tree that stores sorted values. For every node *n* with a **key value** (and maybe data):

- all keys in *n*'s left sub-tree are less than the key in *n*,

- all keys in *n*'s right sub-tree are greater than the key in *n*.

A binary search tree has efficient implementations of the insert, contains, remove, and print methods. Example structures for the node and tree classes are provided below.

```
class BSTnode<K> {
  private K key;
  private BSTnode<K> left, right;
  public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {...} ;
  //accessors:
  public K getKey(){...}
  public BSTnode<K> getLeft(){...}
  public BSTnode<K> getRight(){...}
  //mutators:
  void setKey(K key){...}
  void setLeft(BSTnode<K> left){...}
  void setRight(BSTnode<K> right){...}
};

public class BST<K extends Comparable<K> > {
  private BSTnode<K> root;// root of BST
  public void insert(K key){...}
  public void delete(K key){...}
  public boolean lookup(K key){...}
  public void print(PrintStream p){...}
};
```

Of course, it is also possible to design a BST that stores data along with each key.

The `lookup(K key)` method is an example of a recursive algorithm. The base cases are an empty tree, which returns false, or value is in the root node, which returns true. Check the root, left, and right subtrees. The worst-case lookup for a BST is O(N), and corresponds to a stalky shape. Best case performance for a BST lookup is O(log(N)) for a full, balanced tree.

Question: how do you **delete** a node in a tree with children? And with what do you replace the deleted node? Possibilities:

- the largest value in the left subtree (**in-order predecessor**).

- the smallest value in the right subtree (**in-order successor**).

The Java standard library implements a BST with the `TreeSet` and `HashSet` classes. Use `Map` and `TreeMap` to associate data with each key.

A few additional tree types are listed below.

- **Full tree**: all the leaves have the same depth.

- **Complete tree**: full to depth H-1, depth H filled from left to right.

- **Height-balanced tree**: difference in heights of *left* and *right* subtrees is at most 1.

- **Balanced tree**: has a height of O(log(N)), where N=# nodes.

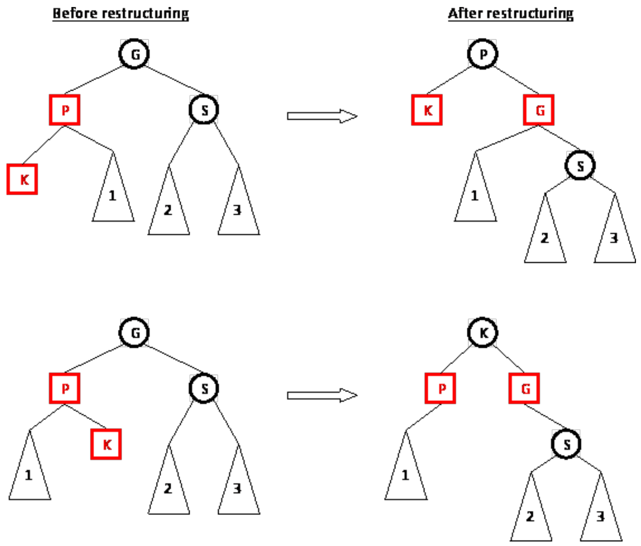

Figure 1: An example of a **tri-node restructuring** procedure for a red-black tree (c.o. James D. Skrentny, University of Wisconsin CS367).

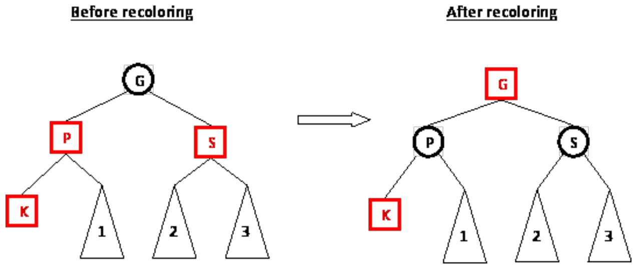

Figure 2: An example of a **recoloring** procedure for a red-black tree (c.o. James D. Skrentny, University of Wisconsin CS367).

4.8.3 *Red-Black Trees*

Red-black trees (RBT) are a type of balanced tree. In order to preserve balance, the `insert` and `delete` operations may restructure the tree. Each node of the tree has a **color** (red or black), and 3 properties hold:

- **Root property**: the root of the tree is black.
- **Red property**: children of a red node are black.
- **Black property**: for each node with at least 1 null child, the number of black nodes on the path from the root to the null child is the same.

We want to perform insert, lookup, delete, and print operations on this tree. Lookup and print for the RBT are identical to those of the BST. The lookup, insert, delete operations have $O(\log(N))$ complexity for the worst-case.

Here's an overview of the RBT insert algorithm for node with key K.

- Use the BST insert algorithm to add K to the tree.
- Color the node containing K **red**.

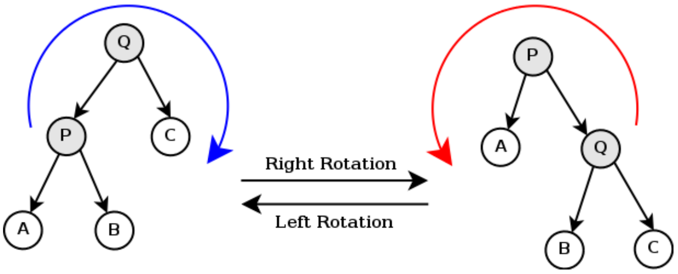

Figure 3: An example of a tree rotation (By Ramasamy at the English language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4043513).

- Restore the RBT properties if necessary.

Insertion can lead to a red property violation **RPV**. Here's how to deal with that.

- *Case 1*: if K's parent P is black, then do nothing (no RPV).
- *Case 2a*: if P's sibling is black or null, do a **trinode restructuring**.
- *Case 2b*: if P's sibling S is red, do a **recoloring** of P, S, G (unless G is the root).

4.8.4 *Balanced Search Trees*

There are other types of search trees that preserve the $O(\log(N))$ tree height that produces efficient operations for N nodes. Other than red-black trees there are also AVL and BTree objects.

An **AVL** is a height-balanced binary search tree. It was the first such structure to be invented, and is named after its Soviet inventors.

- Keeps a balance value for each node (-1, 0, +1).
- Detect problem when balance becomes ± 2.
- Fix by **rotation**.

BTrees relax the binary tree structure by allowing 2, 3, 4 descendant nodes (the number of allowed nodes is called the order). BTrees are also referred to sometimes as 2-3 trees or 2-3-4 trees. When 4 nodes are detected, they are split.

4.9 Priority Queues

Priority queues are a data structure where priorities are put into the object and highest priorities are removed from the object. Priority queues are useful for sorting as well as other operations where minimum or maximum elements need to be accesssed quickly. They typically have `push` and `pop` operations with $\log(N)$ time complexity. Here are the basic priority queue operations:

```
boolean isEmpty()
void insert(Comparable p);
void insert(Comparable p);
Comparable removeMax();
Comparable getMax();
```

To implement the priority queue efficiently, a new data structure called a heap will be implemented. A **heap** is a binary tree with two special properties.

- **Order property** for every node N, the value N is greater than or equal to the values in its children.
- **Shape property**
  - All leaves are at depth d or $(d - 1)$.
  - All of the leaves at depth $(d - 1)$ are to the right of the leaves at depth d.
  - There is at most 1 node with just 1 child. That child is the left child of its parent. It is the rightmost leaf at depth d.

The **root** of the heap is always in `array[1]`. If a node is in `array[k]`, the left child is in `array[2*k]`, the right child is in `array[2*k+1]`, and the parent is in `array[k/2]`.

Implementing **heap insert**:

- add new value at the end of the array (preserve shape),
- swap with parent until order property holds, or get to root.

Implementing **heap removeMax**:

- replace the value with the value at the end of the array (removing leaf),
- work way down the tree, swapping value with the larger child to restore the order property.

The insert and remove operations described above are for a **max heap**. Of course, it is also possible to have a **min heap** along the same lines.

4.10 Hashing

Hashing is a technique that is $O(1)$ average complexity for insert, lookup, and delete operations. The basic idea is to store values (*keys*) in an array, computing each key's position in the array as a function of its value. The array storing data is referred to as the **hash table**, while the function that maps keys to array indices is called the **hash function**. It is good to use efficient hash functions. For example, one can convert the key value to an integer modulo the table size.

One issue that arises with hashing is that multiple data items might map to the same array index. **Collisions** occur when key values map to the same index. There are a few tricks that help to avoid collisions. First, it is best to use a table that is 1.25 times the size of the expected number of items and then to expand the hash table as necessary. Second, it is good to have a *prime* table size, so that hashing functions that include multiplication by some factor don't send all keys to the same bucket after the hash function modulation. Ideally, you want the generator for the hash index to use a constant that is co-prime with the hash table length.

Buckets are used to handle keys that get mapped to the same array index. *LinkedList* buckets can be used for the buckets. This structure would give

$O(1)$ complexity for insertion but $O(N)$ worst-case complexity for other operations. Another option would be to use a BST bucket, which would give $O(\log(N))$ worst-case complexity when all keys are mapped to one index.

Java has a `hashCode()` method for every Object, including `Strings`.

Another way to handle collisions is through open addressing. If there is a collision, search for an unused element elsewhere in the table using a **probe sequence**. The list below uses H to represent the hash function result for an item k:

- *linear probing*: $H(k), H(k)+1, H(k)+2, ...$ % table size
- *quadratic probing*: $H(k), (H(k)+1)^2, (H(k)+2)^2, ...$ % table size
- *double hashing*: $H(k), (H(k)+\mathtt{step}), (H(k)+2\cdot\mathtt{step}), ...$ % table size

4.11 Graph Structure

Graphs are a generalization of the tree structure. They have **nodes** connected by **edges** (also called *vertices* or *arcs*). A nide can have any number of incoming or outgoing edges. There are directed and undirected graphs.

Undirected graphs have nodes which are connected by edges without any direction. Connected (adjacent) nodes are simply referred to as **neighbors**.

Directed graphs have neighbor nodes that are connected by edges with a **direction**. The source of the edge is referred to as the **predecessor** node, while the target of the edge is referred to as the **successor** node.

Paths can be taken through the nodes in a graph. A *cyclic path* is a path in which the same node occurs twice. An *acyclic path* path is a path in which no node is repeated. Note: a node can connect an edge to itself.

Here are some types of graphs:

- *directed acyclic graph (DAG)*: a directed graph that has NO cyclic paths.
- *Complete graph*: an undirected graph that has an edge between every pair of nodes, or a directed graph that has an edge from every node to every other node.
- *Weighted graph*: A graph with values associated with the edges.
- *Network*: A weighted di-graph (directed graph) with non-negative weights.

4.11.1 *Depth-First Search*

A **depth-first search**, or DFS, is a method for searching through the nodes in a graph. It usually is implemented with a *stack* or *recursive* structure. The idea is to start at some node n, then follow an edge of of n, then another edge, getting as far away from n as possible before visiting any more of n's successors.

- start with all nodes marked "unvisited"
- mark n "visited"
- recursively do a DFS from each of n's unvisited successors.

The time for this search is $O(N+E)$, where N=# nodes and E=# edges in the graph. A boolean in the node class could store visited information.

There are many applications of the depth-first search. It can answer questions like the following.

- Is a graph connected?
- Does the graph contain a cycle?
- Is there a path between two nodes?
- What nodes are reachable from the given node?
- Can the nodes be ordered so that, for every node j, j comes before all of its successors?

Depth-first search is similar to the *pre-order* traversal of the tree structure.

4.11.2 *Breadth-First Search*

A **breadth-first search**, or BFS, is a method for searching through the nodes in a graph. The idea is to visit all nodes at the same distance from the starting node before visiting farther nodes. It is usually implemented with a *queue* structure rather than through recursion.

- Start with all nodes "unvisited" and enqueue(n).
- Dequeue, mark node as visited, add the node's children to the queue if unvisited.
- Repeat previous line until queue is empty.

Breadth-first search is similar to the *level-order* traversal of the tree structure. It can be used to find the shortest path between points. *Djikstra's algorithm* is an example of this. It finds the shortest path through a weighted non-negative graph and has complexity $O(E\log(N))$.

4.12 Data Structures in C++

This section provides notes on data structures and coding generally in C++. A list of primitive types in C++ is given in Table 4.12. Also, Table 4.12 provides a description of some basic C++ data structures.

Primitive Type	**Description**
`bool`	a boolean (1 bit)
`char`	a character (1 byte)
`int`	an integer (4 bytes)
`float`	a floating point number (4 bytes)
`double`	a double floating point number (8 bytes)
`void`	a valueless type
`wchar_t`	a wide character (2-4 bytes)

The basic primitive types can be modified using one or more of the type modifiers: `signed`, `unsigned`, `short`, `long`. When a `char` or `int` is `unsigned`, the maximum value is doubled, but the possibility for negative values is removed. The bit size is unchanged. Using `short` reduces the number of bits by a factor of 2 (short int is 2 bytes, whereas int is 4 bytes). Using `long` increases the number of bits by (up to) a factor of 2 (long int is 4-8 bytes).

Name	Description
`deque`	The double-ended queue is a sequence container with dynamic size that can be expanded or contracted on both ends. Usually implemented as some dynamic array. Unlike vectors, deques are not guaranteed to store all elements in contiguous storage locations. Any elements can be accessed in constant time. Worse than lists when frequent internal insertions or removals are needed.
`list`	Lists are sequence containers that allow constant time insertion and erase operations anywhere within the sequence, as well as iteration in both directions (doubly-linked). Similar to `forward_list` except that `forward_list` objects are single-linked lists, and can only be iterated forwards. They lack direct access to elements by their position, and require linear time in the distance between iterator position and target to lookup. They also consume extra memory to store linking information.
`map`	Slower than `unordered_map` for lookup, but faster at iterating over (ordered) elements. Often implemented as a binary search tree, with O(log(N)) lookup time. Generally slower than `unordered_map` containers at accessing individual elements by key. However, `map` allows direct iteration on subsets based on ordering.
`queue`	A FIFO data structure, where items are deposited on one end and withdrawn from the other. Generally, the `push`, `pop`, `front`, and `back` operations are done in constant time.
`stack`	A LIFO structure, container adaptor. Standard container classes are `vector`, `dequeue`, `list`.
`unordered_map`	Implemented as a hash map in C++. Constant time lookup for the most part, amortized linear time lookup. Generally faster than regular `map` for lookup, but slower for range iteration through a subset of elements.
`vector`	A sequence container representing array that can change in size. Use dynamically allocated array to store elements. Memory reallocations should only happen at logarithmically growing intervals of size, so that insertion at the end of a vector can be provided with amortized constant time complexity.

4.12.1 *Exceptions in C++*

Concerning exceptions, C++ allows the user to use `try`, the `throw`, and catch as in Java. The difference is that in the catch clause, you list the exception *type*:

```
try {
  throw 20;
} catch (int e) {
  cout << "An exception occurred, no.  " << e << endl;
}
```

4.12.2 *The const keyword in C++*

The `const` keyword in C++ is used to specify that an object's state is constant (as opposed to `volatile` or `mutable`. Objects whose type is const-qualified cannot be modified. Direct violations usually result in compile time errors.

The `const` keyword usually references the type to its *left*. The exception is when `const` is all the way to the left in a declaration, in which case it applies to the first part of the type. Some examples and their meaning are listed below. The meaning is clear if the definitions are *read in reverse*.

- **`const int x`** - Same as `int const x`. Stores a constant integer value.
- **`int const * x`** - A pointer to a constant integer.
- **`const int * x`** - Equivalent to `int const * x`.
- **`int * const x`** - A constant pointer to an integer.

4.12.3 *Increment and decrement operators in C++*

Given an incrementable object, say `int x`, there are two ways to increment: `x++` and `++x`. What is the difference?

It turns out that `x++` makes a copy of the underlying object in memory, while `++x` does not. For simply incrementing, this might not matter. However, the return value for the operation will be different, as shown in the example below.

```
int x = 5;
int y = x++;
x = 5;
int z = ++x;
```

It turns out that y = 5, whereas z = 6!

5 ALGORITHMS

5.1 Cache Algorithms

Cache algorithms are instructions that a computer can follow in order to maintain a cache of stored information. A **cache** is just a data storage system that can serve up data faster. When the cache is full, the algorithm must decide which item to discard in order to make room for the new item.

A few examples of caching algorithms are LRU, MRU, and RR. The **least recently used** (LRU) algorithm discards the least recently used item in the cache when making space. The **most recently used** (MRU) algorithm discards the most-recently used item in the cache to make space. The **random replacement** (RR) just randomly selects a candidate item to discard.

5.2 Dynamic Programming

Dynamic programming is a method of solving problems. It is similar to a *divide and conquer* strategy, in that a large problem is solved by breaking it down into similar, smaller problems of the same type. However, dynamic programming is typically used with polynomial complexity in scenarios where straightforward divide and conquer or recursion would require exponential complexity.

5.3 Sorting Algorithms

Sorting involves putting the values of an array into some order. **Comparison sorts** work by comparing values. Simple algorithms are $O(N^2)$, though the best-possible performance is $O(N \log(N))$.

Name	Complexity
selection sort	w.c. $O(N^2)$
insertion sort	w.c. $O(N^2)$
merge sort	w.c. $O(N \log(N))$
quick sort	w.c. $O(N^2)$, average $O(N \log(N))$
heap sort	$O(N \log(N))$

Stable sorting algorithms also preseve the relative ordering for duplicate keys from the previous sorting.

5.3.1 *Selection Sort*

Selection sort is an $O(N^2)$ complexity sorting algorithm. The basic approach is to find the smallest value in an array `A` and then put it in `A[0]`. Then find the n^{th} smallest value in `A` and put it in `A[n-1]`.

- Use outer loop from `0` to `n-1`, letting `k` represent the index.
- Use a nested loop from `k+1` to `n-1` to find index of smallest value.
- Swap that value with `A[k]`.
- After i^{th} iteration, `A[0]` through `A[i-1]` are ordered.

Complexity = $(N-1) + (N-2) + ... + 1 + 0 = O(N^2)$.

5.3.2 *Insertion Sort*

Insertion sort is another $O(N^2)$ complexity sorting algorithm. The basic approach is to put the first two items in correct relative order. Then insert the 3^{rd} item in the correct place relative to the first two. Then insert the n^{th} item in the correct place relative to the previous $n-1$.

- Use outer loop from `k=1` to `k=n-1`.
- Use inner loop from `j=k-1` to `0` *as long as* `A[j] > A[k]`.

- Each time, shift higher numbers up (`A[j+1]=A[j]`) to make space for eventual insertion.
- Insert `A[k]` into final `A[j]`.
- After the i^{th} iteration, `A[0]` through `A[i-1]` are relatively ordered but are not in the final position.

5.3.3 *Merge Sort*

Merge sort is an $O(N\log(N))$ *divide and conquer* algorithm. It takes advantage of the fact that it is possible to merge two sorted arrays, each containing $N/2$ items, in $O(N)$ time. It works by simultaneously stepping through the two arrays and always choosing the smaller value to put in the final array.

- Divide the array into two halves.
- Recursively sort the left half.
- Recursively sort the right half.
- Merge the two sorted halves (requires temporary auxiliary array).

5.3.4 *Quick Sort*

Quick sort is another on average $O(N\log(N)$ divide and conquer sorting algorithm. Compared with merge sort, it does more work during the "divide" part in order to avoid work in the "combine" part. The idea is to start by *partitioning* the array using some *median* value (or *pivot* value). Then use 2 pointers at opposite ends of the array to perform swaps of values.

- Choose a pivot value – put pivot at end of array (swap with existing).
- Partition the array. Put all entries less than pivot in the left part and all entries greater than the pivot in the right part, with the pivot in the middle.
- Recursively, sort the values less than or equal to the pivot.
- Recursively, sort the values greater than or equal to the pivot.

Note: a poorly chosen pivot can lead to $O(N^2)$ complexity. The **median of three** technique can be useful for choosing a pivot: pick median value from sampling beginning, middle, and end of array. Upper value placed at end of array, median placed next to end, low value placed at beginning of array.

Note that the quick sort algorithm does not require extra storage space, unlike the merge sort algorithm.

5.3.5 *Heap Sort*

Heap sort is an $O(N\log(N))$ complexity sorting algorithm. The idea is to insert each item into an initially empty heap. Then fill the array right-to-left as follows: while the heap is not empty, do one `removeMax` operation and put the returned value into the next position of the array.

5.3.6 *Radix Sort*

Radix sort is not a comparison sort. It is useful on sequences of *comparable* values, like sequences of characters or numbers. The time is $O((N + R) * L)$, where N is the number of sequences, R is the range of values ieach item could have, L is the maximum length of the sequences. The approach uses an array of queues.

- Process each sequence right-to-left (least to most significant digit).
- Each pass, values taken from original array and stored in a queue in the auxiliary array based on the value of the current digit. Note: the auxiliary array has length L, and the queues at each array position handle duplicates etx.
- Queues are dequeued back into the original array, ready for next pass.

The queue structure preserves previous ordering. It is often *better* than $O(N \log(N))$.

5.3.7 *Bubble Sort*

Bubble sort is a sorting algorithm of complexity $O(N^2)$. Each pass through the unsorted part "bubbles" the next smallest item from unsorted to the back of the sorted part.

- swap `A[j]` with `A[j-1]` if it is smaller, so that small values bubble all the way down.

5.4 Searching in an Array

There are two basic approaches to searching in an array: **sequential** search and **binary** search. Sequential search involves looking at each value in turn. It is faster for sorted arrays $O(N)$. Binary search (for sorted array) involves looking at the middle item, comparing with the value of interest, then eliminating half of the array from the search $O(\log(N))$.

5.5 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest path between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

The original algorithm does not use a min-priority queue, and it runs in time $O(N^2)$, where N is the number of nodes. When implemented with a min-priority queue, the algorithm can be sped up to $O(N \log N + E)$, where E is the number of edges.

Start at an initial node. For a given node Y, define distance as the distance from the initial node to Y.

- Assign an infinite distance value to all nodes other than start, which is zero.

- Set the initial node as current. Mark other nodes as unvisited. Create an unvisited set.
- For the current node, consider all of its neighbors and calculate their tentative distances. Compare newly calculated tentative distance to the current value and keep the smaller one.
- Mark the current node as visited and remove it from the unvisited set.
- Stop if the destination node has been marked visited
- Otherwise, select the unvisited node with the smallest tentative distance, set it as the new current node, and return to step 3.

5.6 A* Search Algorithm

The A* search algorithm is an algorithm that is widely used in pathfinding and graph traversal (plotting an efficient path between multiple nodes). It is an extension of Dijkstra's algorithm, and has been noted for its performance and accuracy.

A* is a best-first search, which means that is solves problems by searching among all possible paths to the solution for the one that incurs the smallest cost, and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs. Starting from a specific node, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost still to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n) \tag{17}$$

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is the tricky part, and is problem-specific. In order for the algorithm to find the actual shortest path, the heuristic must never overestimate the actual cost to get to the nearest goal node. One example of the heuristic is the straight-line distance between two places on a map, in the context of a search for the shortest distance via roads.

Typically, A* is implemented with a *priority queue* to perform the repeated selection of minimum (estimated) cost nodes to expand. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue. From there, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the length of the shortest path.

6 DEEP LEARNING

This section discusses the basics of deep learning. The material is based largely on notes from the Google Udacity course "Deep Learning". On a basic level, deep learning involves training multi-layer neural networks to transform an input data vector into a particular output vector. The networks can be trained in a supervised manner by providing the input data along with the desired output vector, or in an unsupervised way by training the network to build internal, compressed representations of the data in order to reconstruct the input data.

6.1 Softmax

In neural networks and other machine learning algorithms, the **softmax** function is used as the final layer of a classification network. The softmax function is a generalization of the logistic function. It transforms a K-dimensional vector **z** of arbitrary real values into a K-dimensional vector $\sigma(\mathbf{z})$ of real values in the range $(0,1)$ which sum to 1.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \text{ for j=1,...,K} \tag{18}$$

The result of the softmax function can, for example, be used as the predicted probability for the j^{th} class in a classification task.

Softmax score probabilities get close to 0 or 1 as scores get larger. Similarly, score probabilities get closer to uniform as all scores get smaller. A classifier is less confident in a result with low scores and more confident in a result with high scores. The training of a classifier is designed to increase confidence over time.

6.2 Initializing variables

Generally, it is useful to have variables with a mean near zero and equal variance. This will help avoid numerical instability issues and ill-conditioned matrices. Remember: matrix algebra really underpins all of this stuff!

Weight and bias initialization needs to be done carefully, otherwise the training of the model is inefficient. A useful method is to randomly sample from a Gaussian distribution centered at zero with a given standard deviation. The standard deviation is related to the order of magnitude of hte outputs. For a less certain initialization, it is a good idea to use a small standard deviation (much less than 0.1).

6.3 Cross-Validation

Cross-validation, sometimes called rotation estimation, is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent set of data. The idea is that the performance of a model needs to be measured using a dataset that is separate from the one used for training. If the same data are used for training and testing a model, the performance will be falsely inflated, since the model will have trained on some of the statistical variations in the training data. Cross-validation can prevent **over-fitting**.

A simple cross-validation system involves splitting up the data into a **training**, **validation**, and **testing** dataset.

- Training set: a data sample for training the neural network via backpropagation.
- Validation set: a data sample for validating and measuring the performance.
- Testing set: a data sample for preventing over-training.

The bigger the validation set, the more precise the estimate of the model performance. The performance measurement is susceptible to statistical fluctuations, after all. More data always helps. Having a large training dataset is the most important thing.

6.4 Loss function

Machine learning is often about desigining the correct loss-function to use for model optimization.

In order to train a model, it is necessary to measure how far away the output of a model is from the desired output. One way to do this is to minimize the **cross-entropy**. Use the average cross-entropy as the loss function.

Sometimes it is difficult to scale the regression model, where the loss function and gradient is computed for each example. The gradient computation, for example, typically takes 3 times as long as the loss computation.

One solution, called **stochastic gradient descent**,

6.5 Stochastic Gradient Descent

Stochastic gradient descent (SGD) involves computing the loss for a tiny random sample of data instead of the whole dataset (between 1 and 1000 training samples each time). This method requires random sampling of the training data, otherwise it won't work. On average, the gradient will point in the right direction. But the approach requires many small steps, each of which might be partially misguided.

Stochastic gradient descent is at the core of deep learning, since it *scales very well* with data and model size. Bigger data and bigger models are better, after all. However, SGD is a fast but bad optimizer, and it comes with some issues.

First, in order to work properly, the input variables need to have a mean of zero and equal, small, variances. Similarly, the initial weights and biases should have a mean of zero and very small variance.

The training algorithm can also use knowledge from previous steps to know where it should be going. This method, called **momentum**, keeps a running average of the gradient direction and uses that for optimization instead of the single gradient measurement.

It is also useful to make smaller steps as the training continues. This procedure, called **learning rate decay**, lowers the learning rate over time, typically by exponential damping.

The learning rate and momentum are examples of **hyper-parameters** that can be tuned for a model. Some other hyper-parameters include the initial learning rate, the batch size for SGD, and the weight initialization. **AdaGrad** is a modification of SGD that implicitly does momentum and learning rate decay. It makes learning less sensitive to hyper-parameters, but is slightly worse than tuned SGD with momentum.

One useful suggestion: *"Keep calm and lower your learning rate"*.

6.6 Linear Models

A linear classifier creates a discriminant based on the value of a linear combination of the input features. For an input feature vector of real values, **x**, the output score y is given by:

$$y = f(\mathbf{w} \cdot \mathbf{x}) = f(\sum_j w_j x_j), \tag{19}$$

In this expression, **w** is a real vector of weights and f is a function that converts the dot-product of the two vectors into the desired output. **w** is a linear function mapping of **x** onto $\mathbb{R}$.

Numerically, linear operations are very stable. This means that small changes to the inputs do not lead to big changes in the outputs. The derivatives are constant, so they are also quite stable. The problem is that many problems cannot be represented by a linear model. For example, addition can be modeled, but multiplication cannot be modeled.

The trick is to create a non-linear model from linear components. We want to keep our parameters inside big linear functions. After all, GPUs are designed for linear operations (e.g. matrix multiplication). And it is really useful to harness GPUs for training.

6.7 Non-Linear Models

One way to introduce non-linearities into a model is to use **rectified linear units**. A RELU returns 0 if $x < 0$ and x if $x \geqslant 0$. One advantage of RELU is that the derivative is constant.

A logistic model can then be extended by inserting a RELU layer in the middle: the first layer connects the input features to the RELU, while the second layer connects the RELU to the classifier. The resulting model is nonlinear.

The first layer of the model consists of the set of weights and biases applied to the input features and passed through the RELUs. The output layer of the RELU layer is fed to the next one, but it is not observable from outside the network. Hence, it is known as a **hidden** layer. The second layer consists of the weights and biases applied to these intermediate outputs, followed by the softmax function to generate probabilities.

A network can be built by stacking simple operations on top of each other. It makes the back-propagation math, which is based on the chain rule, very simple!

6.8 Back-Propagation

Back-propagation is the mechanism for using differences between the desired and actual model output to update the model parameters and improve the prediction. Basically, the loss function is used to compute a gradient for each model parameter using the chain rule. The model parameters (weights and biases) are then updated. As a reminder, the **chain rule** states that:

$$[g(f(x))]' = g'(f(x)) \cdot f'(x). \tag{20}$$

Note that each block of the back-propagation takes twice the memory and twice the computation time of the forward-propagation.

6.9 Deep Models

The accuracy of a model can improved by adding more parameters to the model. This could be done by increasing the size of each layer of the model, but this turns out to be inefficient. Increasing the number of nodes per layer makes the network harder to train. A better approach is to make the model deeper by adding more layers. It is more efficient in terms of the number of parameters; you can get more performance with fewer parameters by going deeper rather than wider. Furthermore, deep models capture data with hierarchical structures better. Deeper layers are able to learn more abstract features of the data.

However, deep models only shine if you have enough data to train them. But it is hard to optimize a network that is the right size. So it is often easier to train a big network and then try to prevent over-fitting. This is sometimes called the "skinny jeans" problem. Skinny jeans look great, but it is hard to fit in them. So it is usually easier to wear jeans that are just a bit too big.

6.10 Preventing Over-Fitting

One way to prevent over-fitting is **early termination**. During training, keep track of the validation set performance versus training time. As soon as improvement stops being made on the validation set performance, terminate the training.

Another way to prevent overfitting is through **regularization**. With regularization, artificial constraints are applied to the network that implicitly reduce the number of free parameters while not making it more difficult to optimize.

L2 regularization adds a new term to the loss function that penalizes large weights. The new term is just the sum of the square of the weights multiplied by a new hyperparameter constant c:

$$\text{Loss}' = \text{Loss} + \frac{c}{2}(w_1^2 + w_2^2 + ... + w_M^2). \tag{21}$$

L2 regularization is a simple technique that does not change the structure of the network. Furthermore, the derivative of teh L2 norm of the vector is just the weights:

$$W = w_1 + w_2 + ... + w_M. \tag{22}$$

Perhaps the most important recent technique is called **dropout**, which randomly sets some of the activations in the neural net to zero during each training. Activations are the values that connect one layer to another. Dropout forces the neural network to learn redundant internal representations of the data. It sounds inefficient, but it prevents overfitting and makes the model more robust. Essentially, the network becomes a consensus of many networks. If dropout doesn't help, the network probably needs to be larger.

Of course, the randomness of dropout is undesirable when evaluating a network that has been trained with dropout. You don't want randomness to affect the prediction. The network is written to take the consensus of all the evaluations during evaluation. This is done by scaling the non-zero outputs during training by a factor of c for a dropout rate of $\frac{1}{c}$, for example.

6.11 Models for Structured Data

Simple deep neural networks are great, but the performance of deep models can be improved if the data are structured like an image or sequence. If the data have a structure, you can simplify the model training by building that structure into the neural network to begin with.

One concept that is important when considering structures in data is **statistical invariance**. As an example, consider a model that is designed to look for cats in photographs. It doesn't matter *where* the cat appears in the photo, just that the cat appears somewhere. In this position, we want our neural network to have translational invariance (different position, same kitteh). **Statistical invariants** are things that don't change on average across time and or space.

The search for statistical invariants will lead to **convolutional neural networks** for images and **embeddings** and **recurrent neural networks** for text and speech.

6.12 Convolutional Neural Networks

Convolutional neural networks (CNNs, or convnets) share their parameters across space. Images have height, width, and depth (color), and nearby patches of images are related.

Imagine running a patch of an image and running a neural network on it. Then slide the NN across the image, patch by patch, without changing the weights. The output that the neural network will have produced is also an image, just one with a different width, heigh, and depth!

In a convolutional neural network, we have stacks of convolutions. The convolutions will form a pyramid, with the original image at the bottom. The original image is large spatially but relatively shallow. The successive convolutional layers will squeeze the image but also make it deeper. A classifier can be placed at the top of the pyramid. After all of the squeezing and deepening, only features corresponding to the classification task should remain. Depth in the network correponds approximately to *semantic representation*.

Below are some definitions for convnets.

- **Kernel**: a patch of a convnet.
- **Feature map**: each "pancake" in the stack of convnets.
- **Stride**: the number of pixels moved each time the filter is shifted.
- **Padding**: the technique for dealing with the edges of images.

Valid padding is where the convnet never goes past the edge of the image. As a result, the output image is necessarily smaller than the input image. **Same padding** allows patches to go off the edges of the image. The missing inputs are simply padded with zeros. It is called same padding because the output size is the same as the input size, assuming a stride of 1.

A few things can be done to improve the CNN architecture. **Pooling** is a better way to reduce the spatial extent of the features map in the convnet pyramid. As mentioned before, large strides (large pixel shifts between kernels) are a very aggressive way of down-sizing the image. The pooling method is less aggressive: small strides are used, but the convolutions in a neighborhood are combined. **Max-pooling** uses the maximum output from a neighborhood of outputs. One advantages to max-pooling is that it does not add to the number of model parameters. However, it still has a small stride and is expensive to compute. It also adds to the number of hyper-parameters: pooling size and pooling stride must be tuned.

A typical network has the following structure:

$$\text{image} \rightarrow \text{conv.} \rightarrow \text{pool} \rightarrow \text{conv.} \rightarrow \text{pool} \rightarrow \text{fully-conn.} \rightarrow \text{classifier.} \quad (23)$$

There are other examples of pooling, such as **average-pooling**, which takes the average value in a neighborhood instead of a max value. It is similar to providing a blurred view.

Another improvement to the CNN architecture comes from 1×1 **convolutions**. Why would we want to do a 1×1 convolution? They just look at 1 pixel and not a patch, after all. The reason is that, if you add a 1×1 convolution into the middle of the network, you have a mini-deep network over the patch instead of a linear classifier. So it is an *inexpensive* way to make models deeper with more parameters without changing their structure. 1×1 are really just matrix multiplications and not convolutions.

A third way to improve the CNN architecture is via the **inception module**. At each layer of the convnet, we have many options when it comes to pooling or convolution. All of these are beneificial and provide modeling power, so why choose between them? We can just use all of the operations and concatenate the output. The combined output of average pooling, 1×1, 3×3 , and 5×5 convolutions, etc. often performs better than a simple convolution.

6.13 Text in a deep model

How do you classify a document? You can start by having a look at the words in the document. But words are difficult. There are many words, and it is often the most rare words that are the most important. Rare events are a problem for deep learning, which relies on the availability of numerous trainging examples.

Another problem that text presents: we often use different words to mean the same thing. Cat and kitty, for instance, are often interchangeable. We have to train a model that know that kitty and cat are related. We would like to see important words often enough so that we can learn their meanings automatically and see relationships so that we can share parameters between them.

The most straightforward approach involves **unsupervised learning**, or providing training data without labels. It is easy to find text for training on the web or wikipedia, if you can figure out what to learn from it.

One very powerful idea for training is that *similar words tend to occur in similar contexts*. A model can be built to use words to predict word context. Such a model would have to treat cat and kitty similarly, for instance. And you don't even have to know what individual words mean – word meanings can be inferred directly from the company they keep.

To do this, map words to small vectors called *embeddings*. Embedding vectors will be close to each other when words have similar meanings and will be far apart when they have different meanings. Embeddings solve some of the problem with sparsity. Once you have embedded a word into a small vector, you have a word representation where all cat-like things are represented by vectors that are very similar. As a result, the model doesn't have to learn new things for every way of talking about a cat. It can generalize from patterns of cat-like things.

Word2Vec is a simple and effective way to learn embeddings. Each word of a sentence is mapped to an (initially random) embedding vector. Then the embedding is used to predict the context of the word, where the context is just nearby words. Then the model is trained as if it were a supervised learner. A logistic classifier is used to predict nearby words.

It would be useful to take a look at the model and see which embeddings are grouped closely together. One option would be to do nearest neighbor lookup for any given word. Or you could reduce the dimensionality of the embedding to 2 dimensions for representation with a principal component analysis (PCA). But much of the useful information is lost in such a compression, so the result is usually not great. **tSNE** is a visualization technique for embeddings. It provides a way of projecting the embedding that preserves the neighborhood structure.

For training embeddings, it is also often useful to measure the closeness of two vectors using cos() and not L2. This is because the length of the embedding vector is not nearly as relevant to the classification as direction. In fact, it is often better to normalize all embedding vectors to have unit norm.

- $L2 = ||V_{cat} - V_{kitten}||_2^2$
- $\cos() = \frac{(V_{cat} \cdot V_{kitten})}{||V_{cat}||||V_{kitten}||}$

Here's how to make an actual model for Word2Vec. Starting with a word, say "cat", embed the word into a small vector V_{cat}. Then feed V_{cat} into a simple linear model with weights and biases. That model will output a softmax probability that can be compared to the target, which is another word in the context of the input word.

The problem is that there are many words in our vocabulary, and computing the softmax over all the words is difficult. One remedy to this is to *randomly sample* the targets. This is called **sampled softmax**.

One interesting feature of embeddings is that analogies can be expressed in terms of vector operations. An example of a semantic analogy would be "*puppy - dog + cat = kitty*". An example of a syntactic analogy would be *taller - tall + short = shorter*.

There are two ways to train an embedding. The **skip-gram** model uses a word to predict a context. **CBOW**, or continuous bag of words, uses context to predict words. The two approaches are essentially mirror images of each other.

6.14 Recurrent Neural Networks

Of course, there are many way sto learn embeddings. One question: what to do with input sequences of varying lengths?

Recurrent neural networks use shared parameters across time to extract patterns, and are great for dealing with input sequences. They are similar to convolutional networks, in that they share parameters across some dimension. It's just that CNNs share parameters across *space*, while RNNs share parameters across *time*.

Imagine a sequence of events. At each point in time, we want the model to make a decision about what has happened so far in the sequence. We want to take into account the past. One option is to use the state of the previous classifier as a summary of the past.

The idea is to use two classifiers: a single model connected to the summary of the past inputs, and a model connected to the inputs from the current step. The network thus has a simple repeating pattern.

Back-propagation is difficult to do through time, though. The architecture is bad for stochastic gradient descent, since the parameters are correlated. As a result, gradients explode or vanish. Exploding gradients can be handled by a procedure called **gradient clipping**. The idea is to compute the norm and shrink the step when the norm grows too big. It acts as a cheap and effective cap on gradients.

Vanishing gradients are a bigger problem. Practically, vanishing gradients are equivalent to memory loss, where the model quickly forgets its summary of past events. The solution to the problem came from a breakthrough called **LSTM**, or **long short-term memory**. In the basic recurrent model, a summary of the past M as well as current inputs X are fed into a weight module that produces a prediction Y as well as a summary for the future, M'. With LSTM, the weight module is replaced by a LSTM cell. It is still functionally a neural network.

To describe the LSTM, let's first create a simple memory model.

Instruction	Input	Output
Write	X	M
Read	M	Y
Forget	M	

Each logic gate can be thought of as a multiplication by 1 or 0. But what if continuous decisions were made at each gate instead, with values between 0 and 1? This would give us a continuous, differentiable, back-prop-able memory model! You can perform a logistic regression of gate values. The gates help the model keep memory longer when needed and forget memory when not needed.

Regularization techniques are needed for LSTM as with other neural networks. L2 regularization works. So does dropout, as long as you use it on the input or the output but not on the recurrent connections.

6.14.1 *Prediction with recurrent neural networks*

A model that predicts the next item of a sequence can also be used to *generate* a sequence. This can be done by aternating predictions with samplings to build a sequence. A more sophisticated generative method is to sample multiple times at every step. Then you have multiple sequences (a.k.a. multiple hypotheses) with which you can continue predicting at each step. By looking at the total probability of multiple time steps at a time, you prevent the sampling from accidentally making one bad choice and being stuck with it for the rest of the time sequence.

One way of doing this is **beam search**, where only a few of the most likely sequences are kept at each step. This approach works well for generating sequences. It also prevents the number of hypotheses from growing exponentially.