CSI 4103

Great Algorithms

Study Guide

Fall 2024

Gaussian Elimination

- Used to solve systems of linear equations
- Can be applied to any system with the same number of equations and unknowns
- Here's how it works

Algorithm GE
Input: A system S of n linear equations with n unknowns
1 if n=1 and the equation is ax=b, output b/a
2 Otherwise, pick a variable x and an equation e in S in which x has nonzero coefficient
3 Create a new system S' of n-1 equations and n-1 unknowns
4 For every equation e'≠S other than e
5 subtract enough copies of e from e' to eliminate x and
6 include the resulting equation in S'
7 Apply GE to S' to obtain an assignment to all variables except for x
8 Plug the partial assignment into e to recover x
9 Output the completed assignment

How to Analyze Algorithms

We ask questions like

- Why does it work? Does it ever fail?
- How efficient? What makes it attainable?
- How robust?
- Why is it useful?
- Can we apply the algorithm? Look for nails to hammer in

Problems vs Algorithms

- A computational problem is a relation R on instance-solution pairs
- An algorithm is a procedure that takes an instance x and produces solution(s) y such that $(x,y)\in R$
- Types of algorithms:
  - Search: output any solution y as long as $(x,y)\in R$
  - Decision: only need to answer if such solutions exist (yes/no)
  - Enumeration: output all possible solutions

In the case for Gaussian Elimination, an instance-solution pair is (S, sol), S is a system of linear equations and sol is an assignment

Correctness of Gaussian Elimination

- Algorithm GE has its issues, but we do expect it to work on instances with a unique solution
- The procedure of adding or subtracting multiples of one row from another is called an elementary row operation. They are invertible

Claim: Let S be a system of linear equations and e be an equation in S. Let S' be obtained from S by adding a multiple of e to some other equation e'. Then S' has the same solutions as S

Solution for S $\rightleftarrows$ Solution for S'

The reason for this is because being a solution is closed under taking linear combinations of equations

Proposition: Assuming S has a unique solution, then GE finds it

We can also tackle the problem where an equation reduces to 0=0. We add the line: if S has the form 0=0 but there are still unused variables, set all of them free. This should be added before line 2

There is also the problem of having a contradiction in the form 0=b for some nonzero b. Then insert the line: if S has the form 0=b for nonzero b, output no solution

Gauss-Jordan Elimination

Algorithm GJE
Input: A system S of m linear equations with n unknowns
1 If S has the form 0=0 or is empty, output the assignment in which all variables are free
If S has the form 0=b for b≠0, output no solution
2 Otherwise, pick a variable x and an equation e in S in which x has nonzero coefficient
3 Create a new system S' of m-1 equations and n-1 unknowns
4 For every equation e'≠S other than e
5 Subtract enough copies of e from e' to eliminate x and
6 include the resulting equation in S'
7 Apply GE to S' to obtain an assignment to all variables except for x
8 Plug this partial assignment into e to recover x in terms of the free variables
9 Output the completed assignment

Number of equations and number of unknowns no longer have to be the same

Theorem: GJE outputs a description of all the solutions of S if there are any, and no solutions if there aren't

Efficiency and Limitations of GE

Suppose a system of equations has m equations and n unknowns

Worst case: m-1 elementary row operations, then runs recursively on an instance with m-1 equations and n-1 unknowns, then completes assignment. There are $\Theta(mn)$ operations for step 4-6 and $\Theta(n^2)$ operations for step 8. The number of operations is then

$$C(m,n)=C(m-1,n-1)+\Theta(mn+n^2)$$

If m=n, we get

$$C(n)=C(n-1)+\Theta(n^2)$$
$$=\Theta(n^3)$$

In general, it takes $mn+m$ numbers to describe a system with m equations so the worst case complexity is at most $\Theta(\text{(input size)}^{1.5})$

A linear system is sparse if the number of variables that participates in a typical equation is much smaller than n

For such sparse systems, the worst-case running time of GE is cubic in the size of the instance

Some Features of Gaussian Elimination

- Gaussian Elimination allows for in-place implementation. The space used to describe the instance can be reused to describe all intermediate states and the solution
- Can be implemented with perfect precision. If the input coefficients are provided as rational numbers and the additions and divisions in the elementary row operations are implemented without loss of accuracy, the algorithm will output an exact solution

Unsatisfiability of Gaussian Elimination

If we run Gaussian Elimination on a system and it outputs "no solution", how do we know there's indeed no solution?

A system cannot have a solution by coming up with some linear combination of the equations that makes the left-hand side vanish but not the right hand side. Such a linear combination is called a contradictory linear combination

Theorem: A system of equations has no solutions if and only if there exists a contradictory linear combination of the equations

If we use matrix notation and our system is $A\vec{x}=\vec{b}$, then if $\vec{y}^T$ is a row vector, we get that $A\vec{x}=\vec{b}$ has no solution if and only if $\vec{y}^TA=\vec{0}$, $\vec{y}^T\vec{b}=1$ has a solution

Learning Linear Functions

We can use data to train linear functions.

For example, for CSI4103 we want to know the grade weighting

h = homework
m = midterm
p = project
f = final grade

We have $f=hx_1+mx_2+px_3$ and we can solve for h, m, and p using past grade data and GE

However, the data is often noisy and rounding can significantly impact the accuracy of GE

Modular Gaussian Elimination

Problem: We have inputs $x_1,\dots,x_n$ that can be +1 or −1, then we have $f(x)$ is the product of a subset of the input. We want to find the subset.

We can use data to learn the subset.

This can be done by replacing −1 with 1 and 1 with 0 mod 2. Then GE can be used to solve the system and the subset can be found

Gradient Descent

The gradient of a function $f(\vec{x})$ where $\vec{x}=(x_1,x_2,\dots,x_n)$ is the vector of partial derivatives

$$\nabla f(\vec{x})=\left(\frac{\partial f}{\partial x_1},\frac{\partial f}{\partial x_2},\dots,\frac{\partial f}{\partial x_n}\right)$$

The gradient descent algorithm is an algorithm used for minimizing a function f. Used for solving system of linear equations when the solution minimizes a loss function f.

Algorithm GD (Simple Gradient Descent)
Input: a function f with multiple inputs
1 Choose a starting point $\vec{x}$
2 Choose a rate $\rho>0$
3 Until $\vec{x}$ is sufficiently small
4 Move $\vec{x}$ to $\vec{x}-\rho\cdot\nabla f(\vec{x})$

Choosing the right ρ is very important. ρ too large can cause the algorithm to never make progress. ρ too small can slow everything down

If we have a linear system $A\vec{x}=\vec{b}$ and loss function $\|A\vec{x}-\vec{b}\|$, then $\nabla f(\vec{x})=2A^T(A\vec{x}-\vec{b})$

Why Gradient Descent Works?

For one variable: if we have an equation $ax=b$. Then the loss function is $f(x)=(ax-b)^2$ and $\frac{df}{dx}=2a(ax-b)$. We have that as long as $\rho<\frac{1}{a^2}$, Gradient descent will make progress

For a system $A\vec{x}=\vec{b}$, $f(\vec{x})=\|A\vec{x}-\vec{b}\|^2$, we have if $\|A\|$ is the spectral norm of A and if $\rho<\frac{1}{\|A\|^2}$, then GD will make progress

How Fast is Gradient Descent?

- If we move from x to x', we have $x'=x-\rho\cdot2a(ax-b)$ which homogenizes to $y'=(1-2\rho a^2)y$ for $y=ax-b$ and $y'=ax-b$. y is square root of the loss.

The root-loss shrinks by a factor of $1-2\rho a^2$ in each step, an exponential rate

- For system of multiple equations $A\vec{x}=\vec{b}$ with loss $f(\vec{x})=\|A\vec{x}-\vec{b}\|^2$, then if we assume $\rho<\frac{1}{\|A\|^2}$ and our initial guess is at distance d from the actual solution, then at iteration t, we have

$$f(x_t)\le\frac{d^2}{(1-\rho\|A\|^2)\rho t}$$

This guarantees an inverse linear rate of progress. Can be faster but not slower.

Spectral Norm and Condition Number

For a symmetric matrix B, the spectral norm is defined as $\|B\|=\max_{\vec{v}\neq\vec{0}}\frac{\|B\vec{v}\|}{\|\vec{v}\|}$. If $\|B\|<1$ then B shrinks every vector.

We have that if we have a system $A\vec{x}=\vec{b}$, then if we move from $\vec{x}$ to $\vec{x}'$, we get

$$\vec{x}'-\vec{x}^*=(I-2\rho A^TA)(\vec{x}-\vec{x}^*)$$

$\vec{x}^*$ is actual solution to $A\vec{x}=\vec{b}$

The spectral norm of $I-2\rho A^TA$ would determine how much it shrinks the distance to the actual solution

The condition number of a matrix A, denote by K, measures how close the rows of A are to being linearly independent

Theorem: If A is a square matrix with linearly independent rows, $\rho<\frac{1}{\|A\|^2}$, after t steps with initial guess $\vec{x}$, the state $\vec{x}_t$ satisfies

$$\|\vec{x}_t-\vec{x}^*\|\le(1-2\rho K^{-1})\|\vec{x}-\vec{x}^*\|$$

K condition number of A^TA

Working with noisy data

Gradient descent gives the best estimations for solutions to a linear system even with noisy data. It minimizes the error sum of squares.

Theorem: For any matrix A, $\rho<\frac{1}{\|A\|^2}$, after t steps with initial guess $\vec{x}$, the state $\vec{x}_t$ of gradient descent satisfies

$$\|\vec{x}_t-\vec{x}^*\|\le(1-2\rho K^{-1})^t\|\vec{x}-\vec{x}^*\|$$

$\vec{x}^*$ is the unique input that minimizes $f(\vec{x})=\|A\vec{x}-\vec{b}\|^2$

Variants of Gradient Descent

- Adaptive Gradient Descent

Algorithm AGD
Input: A function f with multiple inputs
1 Choose starting point $\vec{x}=(x_1,x_2,\dots,x_n)$
2 Until $\vec{x}$ sufficiently small
3 Choose rate ρ depending on $\vec{x}$, $\nabla f(\vec{x})$, $f(\vec{x})$, and previous ρ
4 Move $\vec{x}$ to $\vec{x}-\rho\cdot\nabla f(\vec{x})$

- Steepest Descent chooses ρ that minimizes $\vec{x}-2\rho\nabla f(\vec{x})$
- Conjugate Gradient descent chooses ρ such that it minimizes $(1-2\rho\lambda^{-1})$. Guaranteed to terminate.
- Preconditioning: apply a transformation on the rows of the original system to make them less dependent
- Gradient descent interacts with its input by evaluating the gradient $2A^TA\vec{x}-2A^T\vec{b}=L\vec{x}-b'$ where $L=2A^TA$ and $\vec{b}'=2A^T\vec{b}$. It is effectly solving the system $L\vec{x}=\vec{b}'$. A^TA is symmetric positive-definite
- Stochastic Gradient Descent uses a small sample of records to approximate the gradient, where each row of the matrix is a record

Convex Minimization

- Gradient descent works on all convex functions
- A function is convex if the averaging of 2 evaluations is never smaller than evaluating the average
- Also used for non-convex minimization

Graphs

Consist of vertices and edges that connect pairs of vertices

Can be represented by symmetric matrices with 0s and 1s, where a 1 at position (i,j) means that there being an edge between vertices V_i and V_j.

Graphs can be realized geometrically as a dot product of vectors. We can associate a vector to every vertex so that entries in the matrix are a dot product of the corresponding vectors.

Edges represented by vectors with dot product 1 while non-edges are represented by vectors with dot product 0

Any graph with n vertices has a geometric representation in at most n-dimensional space. We want lower dimensional embeddings.

We can do this using algebra

Eigenvalues and Eigenvectors

- An eigenvector of S is a nonzero vector $\vec{v}$ and λ is its corresponding eigenvalue if we have

$$S\vec{v}=\lambda\vec{v}$$

- If a matrix is symmetric, then we have $S\vec{v}$ and $\vec{v}^TS$ produce the same result, but $S\vec{v}$ is a column vector and $\vec{v}^TS$ is a row vector. We can then get that eigenvectors associated with different eigenvalues are orthogonal
- The eigenvectors of a symmetric matrix break up into mutually orthogonal subspaces, with each subspace labeled by a distinct eigenvalue
- The collection of subspaces with their corresponding eigenvector is called the spectrum of S.

Theorem: Every nxn symmetric matrix S has an orthonormal basis of eigenvectors $\vec{v}_1,\vec{v}_2,..,\vec{v}_n$

Every vector can be written as a linear combination of $\vec{v}_1,\vec{v}_2,...,\vec{v}_n$

We can write

$$\vec{x}=\alpha_1\vec{v}_1+\alpha_2\vec{v}_2+...+\alpha_n\vec{v}_n \qquad \alpha_i=\vec{x}\cdot\vec{v}_i$$

$$S\vec{x}=\alpha_1\lambda_1\vec{v}_1+\alpha_2\lambda_2\vec{v}_2+...+\alpha_n\lambda_n\vec{v}_n$$

Applying S to $\vec{x}$ scales the coefficients by their corresponding eigenvalue

If we shift our perspective from the standard basis, S looks like a diagonal matrix: It affects all directions independently, each scaled by the corresponding eigenvalue.

The transformation $\vec{x}\to S\vec{x}$ in the eigenvector basis scales all coordinates by their eigenvalues. The maximum amount by which a vector $\vec{x}$ can grow is its largest eigenvalue in absolute value.

Theorem: The spectral norm of a symmetric matrix S equals the largest eigenvalue in absolute value

Power Iteration

Algorithm PI

Input: Symmetric matrix S

1 Choose an initial vector $\vec{x}$ (unit length)

2 Until 2 successive values are close

3 Replace $\vec{x}$ by $S\vec{x}$

4 Normalize $\vec{x}$ to unit length

5 Output the last $\vec{x}$ as eigenvector and last normalization factor as the spectral norm

Power iteration converges rapidly to a dominant eigenvector provided S exhibits a gap in the magnitudes of its largest and second largest eigenvalues. If $\lambda_1,\vec{v}_1$ is the dominant eigenvalue-eigenvector pair, after t steps the projection of $\vec{x}$ on $\vec{v}_1$ equals

$$\vec{x}\cdot\vec{v}_1=\frac{1}{\sqrt{1+(\alpha_2/\alpha_1)^2\cdot(\lambda_2/\lambda_1)^{2t}+...+(\alpha_n/\alpha_1)(\lambda_n/\lambda_1)^{2t}}}$$

and $\vec{x}$ approaches $\vec{v}_1$ at an exponential rate with base $\max_{i\neq1}|\lambda_i|/|\lambda_1|$

If a matrix fails to exhibit a spectral gap, then $\vec{x}$ would keep alternating between 2 possibilities. To avoid this, we can shift the diagonal entries of the matrix by the same amount

Spectral Decomposition

Any symmetric matrix can be decomposed as

$$S=\lambda_1\vec{v}_1\otimes\vec{v}_1+\lambda_2\vec{v}_2\otimes\vec{v}_2+...+\lambda_n\vec{v}_n\otimes\vec{v}_n$$

$\otimes$ represents outer product

Theorem: S is symmetric with nonnegative eigenvalues if and only if $S=A^TA$ for some matrix A

The geometric realization of S can be read off from the columns of the matrix A. The entries of S are the dot products of the columns of A. Thus, geometric realizations can be derived from the spectral decomposition

If we can shift the diagonal such that the negative eigenvalues disappear, then we can lower the number of dimensions in the geometric representation

We also have: graph regular $\to$ Top eigenvector $=\begin{pmatrix}1\\1\\\vdots\\1\end{pmatrix}$

Approximate Embeddings

We sometimes cannot find exact low dimensional embeddings. We need to find a good approximation.

Picking the best possible low-dimensional representation L of PSD matrix S is an optimization problem. We can either minimize $\|S-L\|$ or the sum of squares of S-L. We use the truncated spectral decomposition

Theorem: Assume S is PSD with spectral decomposition $\lambda_1\vec{v}_1\otimes\vec{v}_1+...+\lambda_n\vec{v}_n\otimes\vec{v}_n$ and $\lambda_1\geq...\geq\lambda_n\geq0$. Among all d by n matrices A, the spectral norm $\|S-A^TA\|$ and the sum of squares of the entries of $S-A^TA$ are both minimized when A is the matrix with rows $\sqrt{\lambda_1}\vec{v}_1,...,\sqrt{\lambda_d}\vec{v}_d$

Each successive term in the expansion provides a finer approximation of S

Orthogonalization

We can use power iteration to find the eigenvalue-eigenvector pairs of a symmetric matrix S. If we found an eigenvector and we want to find the next, power iteration won't make any progress if we don't make $\vec{x}$ perfectly orthogonal with the previous eigenvector.

We can enforce orthogonality between $\vec{x}$ and the previous eigenvector at every step of power iteration

To do this, we can use Gram-Schmidt orthogonalization

Algorithm GS

Input: Vectors $\vec{v}_1,...,\vec{v}_n$

1 For i ranging from 1 to n

2 Replace $\vec{v}_i$ with $\vec{v}_i-\sum_{j=1}^{i-1}(\vec{v}_i\cdot\vec{v}_j)\vec{v}_j$

3 Normalize $\vec{v}_i$ to unit length

Subspace Iteration

Algorithm SI

Input: nxn symmetric matrix S

1 Choose an initial orthonormal basis $\vec{x}_1,...,\vec{x}_n$

2 Until $\vec{x}_1,...,\vec{x}_n$ stabilize

3 Replace $\vec{x}_i$ by $S\vec{x}_i$ for all i

4 Orthogonalize $\vec{x}_1,...,\vec{x}_n$ using GS

Eigenvalues can be recovered by calculating $\frac{\|S\vec{v}_i\|}{\|\vec{v}_i\|}$

QR Algorithm

This algorithm only finds the eigenvalues

Algorithm QR

Input: A nxn symmetric matrix S

1 Until the diagonal entries of S stabilize

2 Calculate QR decomposition of S

3 Replace S with RQ

Theorem: Let $X=(\vec{x}_1,...,\vec{x}_n)$ and S be the states of SI and QR after t iterations, respectively. If both are run on input S_0 and SI is initialized with the standard basis, then $S=X^TS_0X$

Fourier Transform over Boolean cubes

- Fourier transform is a conversion between two representations of a function: list representation and polynomial representation
- Start with functions over the n-dimensional Boolean cube $\{-1,+1\}^n$. Inputs are all 2^n bit strings of length n
- The list representation of a function on n-bit inputs is defined by listing its 2^n evaluations in some order
- The polynomial representation is built using a linear combination of all 2^n parity functions. We can also write the polynomial representation by listing all 2^n coefficients $\hat{f}(S)$.
- Any function is a linear combination of point functions $\text{Point}_{\vec{a}}$, they are functions that evaluate to 1 if $\vec{x}=\vec{a}$ and 0 otherwise
- We have that the polynomial representation for a univariate point function is

$$\text{Point}_a(x)=\frac{1+ax}{2}$$

- Point functions are also multiplicative

$$\text{Point}_{a_1,a_2}(x_1,x_2)=\text{Point}_{a_1}(x_1)\text{Point}_{a_2}(x_2)=\frac{1+a_1x_1}{2}\cdot\frac{1+a_2x_2}{2}$$

- Then the polynomial function of a function given by its 2^n evaluations $f(\vec{a})$ as $\vec{a}$ ranges over $\{-1,+1\}^n$ is obtained by simplifying the expression

$$f(\vec{x})=\sum_{\vec{a}\in\{-1,+1\}^n}f(\vec{a})\,\text{Point}_{\vec{a}}(\vec{x})=\sum_{a_1,...,a_n\in\{-1,+1\}}f(a_1,...,a_n)\cdot\frac{1+a_1x_1}{2}\cdots\frac{1+a_nx_n}{2}=\sum_{S\in P([1,n])}\hat{f}(S)\prod_{i\in S}x_i$$

- From a geometric perspective, the Fourier transform is a change of basis formula. We define a vector space where each axis is labeled by an input. The list representation can be represented using a vector of dimension 2^n. The polynomial representation is just a representation of the same vector in a different basis
- Fourier transform converts the standard representation of f as a linear combination of the 2^n point functions to the polynomial representation of f as a linear combination of the 2^n parity functions $\text{parity}_S=\prod_{i\in S}x_i$
- Lemma: The 2^n parity functions are mutually orthogonal when viewed as 2^n-dimensional vectors
- We have that in general

$$\hat{f}(S)=\text{average value of }(f\cdot\text{parity}_S)=\frac{1}{2^n}\sum_{\vec{a}\in\{-1,+1\}^n}f(\vec{a})\,\text{parity}_S(\vec{a})$$

Fast Fourier-Walsh Transform

- Computing the Fourier Transform means taking the list of 2^n values $f(\vec{a})$ and producing the list of 2^n Fourier coefficients $\hat{f}(S)$.
- Both input and output has size $N=2^n$ so it makes sense to express the complexity in terms of N
- Evaluating the fourier coefficient would entail N^2 operations: It takes $N=2^n$ additions to calculate each Fourier coefficient and there are $N=2^n$ of them to calculate
- There's a faster recursive algorithm. Suppose we have figured out the polynomial representations of

$$f_-(x_1,...,x_{n-1})=f(x_1,...,x_{n-1},-1)$$
$$f_+(x_1,...,x_{n-1})=f(x_1,...,x_{n-1},+1)$$

We can get that

$$f=\tfrac{1}{2}(f_++f_-)+\tfrac{1}{2}(f_+-f_-)x_n$$

Then we get

$$\hat{f}(S)=\tfrac{1}{2}(\hat{f}_+(S)+\hat{f}_-(S)) \quad \text{for every } S\subseteq\{1,...,n-1\}$$
$$\hat{f}(S\cup\{n\})=\tfrac{1}{2}(\hat{f}_+(S)-\hat{f}_-(S))$$

Algorithm FFW (Fast Fourier-Walsh Transform)
Input: A list representation of $f: \{-1,+1\}^n \to \mathbb{R}$

1 If n=0, output the value f()

2 Calculate $\hat{f}_+ = FFW(f_+)$ and $\hat{f}_- = FFW(f_-)$

3 Calculate $\hat{f}$ and output it

Complexity

- The Fourier representation is sometimes a helpful indicator of computational complexity of the function
- One natural complexity measure of a polynomial is its degree
- Low degree polynomials have relatively few "degrees of freedom"
- A linear function over the n-dimensional Boolean cube is completely specified by its mean $\hat{f}(\emptyset)$ and its n level-1 Fourier coefficients $\hat{f}(\{1\}), \ldots, \hat{f}(\{n\})$
- A degree-d polynomial is determined by its Fourier coefficients of size d or less. There are at most $2^{nH(d/n)}$ of them, where H(p) is the binary entropy function
- When d is smaller than n by some factor this is much less than the 2^n values it takes to specify the function as a whole
- The compactness of polynomial representations can be exploited by algorithms
- Suppose we're given input-output examples from some function f. If f is a degree-d polynomial we can try to reconstruct the polynomial representing f by interpolating its Fourier coefficients from the examples. This can be accomplished in $2^{nH(d/n)}$

Approximation using Polynomials

- Some simple types of calculations are naturally captured by low-degree polynomials
- Other, more complex functions can be approximated by polynomials of fairly low degree
- The degree-d approximation of any function f that minimizes the mean-squared error of the first d Fourier levels $\sum_{S:|S|\le d} \hat{f}(S)^2$ is $\sum_{i=0}^{d} \hat{f}(\{i\})x^i$. This follows from orthogonality of the parities
- For a function f, the mean-square error of the degree-d approximation equals

$$\left(\text{average of } f(\vec{x})^2\right) - \sum_{S:|S|\le d} \hat{f}(S)^2$$

- Parseval's inequality

$$\text{average of } f(\vec{x})^2 = \sum_S \hat{f}(S)^2$$

for ±1 valued function, we get

$$\sum_S \hat{f}(S)^2 = 1$$

- Disadvantage: dense parity functions do not qualify as simple

Modular Fourier Transform

- If we have a Boolean function $f:\{-1,+1\}^n \to \mathbb{R}$ with a one-bit input. Their Fourier transform is

$$f(x) = \hat{f}(\emptyset) + \hat{f}(1)x \qquad \hat{f}(\emptyset) = \frac{f(1)+f(-1)}{2} \quad \hat{f}(1) = \frac{f(1)-f(-1)}{2}$$

- The parity basis $1=(1,1)$ and $x=(1,-1)$ is not only orthogonal as a basis of functions, but it is a group. The pointwise product of two basis functions is a basis function. The multiplication table is

·	1	x
1	1	x
x	x	1

≡

	+1	−1
+1	+1	−1
−1	−1	+1

Consider functions that take three input values, or trits. The trits are represented by $\omega_0, \omega_1, \omega_2$

- We want to represent functions f over domain $\{\omega_0, \omega_1, \omega_2\}$ by polynomials
- f should have three "degrees of freedom", suggesting a quadratic representation

$$f(x) = \hat{f}(0) + \hat{f}(1)x + \hat{f}(2)x^2$$

- $1, x, x^2$ should multiply exactly like $\omega_0, \omega_1, \omega_2$

Solution: Associate monomial x^i by the complex root of unity $\omega_i = e^{2\pi i/3}$

	ω_0	ω_1	ω_2
ω_0	ω_0	ω_1	ω_2
ω_1	ω_1	ω_2	ω_0
ω_2	ω_2	ω_0	ω_1

- For dot product with complex entries, we have

$$\vec{a}\cdot\vec{b} = (a_1,\ldots,a_n)\cdot(b_1,\ldots,b_n) = a_1\overline{b_1} + \ldots + a_n\overline{b_n}$$

$\overline{b_j}$ is the complex conjugate of b_j

- Using the above definition, we get $1, x, x^2$ are orthogonal
- The Fourier transform modulo 3 is defined for complex-valued functions f over the third roots of unity $\{1, e^{2\pi i/3}, e^{4\pi i/3}\}$. These functions have a unique quadratic polynomial representation.
- The Fourier coefficients are

$$\hat{f}(j) = \text{average of } (f\cdot x^{-j}) = \frac{1}{3}\left(f(1)\cdot 1 + f(e^{2\pi i/3})\cdot e^{-2\pi i j/3} + f(e^{4\pi i/3})e^{-4\pi i j/3}\right)$$

- $\omega = e^{2\pi i/3}$. Then the domain of f is $\{1,\omega,\omega^2\}$ and its Fourier Transform is

$$\hat{f}(j) = \frac{1}{3}\left(f(1) + f(\omega)\omega^{-j} + f(\omega^2)\omega^{-2j}\right)$$

- Generalizes to arbitrary modulus $N \ge 2$. Functions $f:\{1,\omega,\ldots,\omega^{N-1}\} \to \mathbb{C}$ have a unique degree-(N−1) polynomial representation

$$f(x) = \hat{f}(0) + \hat{f}(1)x + \ldots + \hat{f}(N-1)x^{N-1}$$

$$\hat{f}(j) = \frac{1}{N}\sum_{k=0}^{N-1} f(\omega^k)\omega^{-kj}$$

Fast Fourier Transform

- Takes time O(NlogN) assuming $N=2^n$
- If we take a function f, we can split it into even and odd parts

$$f(x) = \hat{f}(0) + \hat{f}(1)x + \hat{f}(2)x^2 + \ldots + \hat{f}(N-1)x^{N-1}$$

$$= \left(\hat{f}(0) + \hat{f}(2)x^2 + \ldots\right) + \left(\hat{f}(1)x + \hat{f}(3)x^3 + \ldots\right)$$

$$= f_+(x^2) + x f_-(x^2)$$

$$f_+(y) = \hat{f}(0) + \hat{f}(2)y + \hat{f}(4)y^2 + \ldots$$

$$f_-(y) = \hat{f}(1) + \hat{f}(3)y + \hat{f}(5)y^2 + \ldots$$

We can then get

$$f_+(x^2) = \frac{f(x)+f(-x)}{2}$$

$$f_-(x^2) = \frac{f(x)-f(-x)}{2x}$$

We also get that

$$\hat{f}_+(j) = \hat{f}(2j) \qquad \hat{f}_-(j) = \hat{f}(2j+1)$$

Conversely, we get

$$\hat{f}(j') = \begin{cases} \hat{f}_+(j'/2) & j' \text{ even} \\ \hat{f}_-((j'-1)/2) & j' \text{ odd} \end{cases}$$

Algorithm FFT (Fast Fourier Transform)
Input: N (power of 2) and list $f(1), f(\omega), \ldots, f(\omega^{N-1})$

1 If N=1, output value of f(1)

2 Set $\omega = e^{2\pi i/N}$

3 Calculate list representations of $f_+, f_-: \{1,\omega^2,\omega^4,\ldots,\omega^{2(N-1)}\} \to \mathbb{C}$

4 Calculate $\hat{f}_+ = FFT(N/2, f_+)$ and $\hat{f}_- = FFT(N/2, f_-)$

5 Construct $\hat{f}: \{0,\ldots,N-1\} \to \mathbb{C}$ and output it

Cosine Transform

- In signal processing, the function $f(\omega^t)$ might represent a signal like the amplitude of a sound sampled at times $t = 0, 1, \ldots, N-1$

$$f(\omega^t) = \hat{f}(0) + \hat{f}(1)\omega^t + \ldots + \hat{f}(N-1)\omega^{t(N-1)}$$

$\omega = e^{2\pi i/N}$

- Even though the signal is real valued, the Fourier coefficients and the basis functions are complex
- $f(\omega^t)$ is periodic modulo n, but signals are not likely periodic, discontinuity might exist at $t=N-1$ to $t=N$
- The problems can be eliminated by concatenating with mirror image before taking the Fourier transform

$$g(\omega^t) = 2\hat{g}(0) + 4\hat{g}(1)\cdot\cos\left(\frac{\pi t}{2N}\right) + \ldots + 4\hat{g}(N-1)\cos\left(\frac{(2N-1)\pi t}{2N}\right)$$

$\omega = e^{\pi i/2N}$ $\quad t$ odd modulo $4N$

- j in $\hat{g}(j)$ represents frequency of corresponding cosine wave

Two-dimensional Transforms

- Constructed by taking the "product" of one-dimensional Fourier transforms
- Suppose we know a Fourier representation for functions in x and one for functions in y. Then functions in x and y have a unique representation

$$f(x,y) = \sum_{i,j} \hat{f}(i,j)x^i y^j$$

- The embeddings of the monomials $x^i\cdot y^j$ are obtained by taking the outer product of the embeddings of x^i and y^j individually: the $(s,t)^{th}$ entry of the vector representing $x^i y^j$ is the product of the s^{th} entry of x^i and the t^{th} entry of y^j.
- $\hat{f}(i,j)$ is the average of $f(x,y)x^{-i}y^{-j}$ for x,y ranging over their respective domains
- If x and y range over the M^{th} and M^{th} roots of unity, respectively, the Fourier representation of f(x,y) has the form

$$f(e^{2\pi i s/N}, e^{2\pi i t/M}) = \sum_{j,k} \hat{f}(j,k)e^{2\pi i(sj/N + tk/M)}$$

j ranging from 1 to N−1

k ranging from 1 to M−1

- Fourier coefficient obtained by averaging $f(e^{2\pi i s/N}, e^{2\pi i t/M})e^{2\pi i(sj/N+tk/M)}$ over $s \in \{0,\ldots,N-1\}$ and $t \in \{0,\ldots,M-1\}$

Continuous Transforms

- It is sometimes useful to think of the signal as a continuous function
- The amplitude of a sound wave can in principle be measured at any continuous instance t
- Discrete-time approximations that are amenable to data processing can be obtained by sampling the signal at regularly spaced points
- As N gets larger and larger, the Fourier transform modulo N approaches a continuous Fourier transform.
- Its inputs are functions f that take values on the continuous unit circle
- Any reasonable f of this type can be expanded as a possibly infinite Fourier series $f(\omega) = \sum_{j=-\infty}^{\infty} \hat{f}(j)\omega^j$. ω denotes any point on complex unit circle and j ranges over all integers. $\hat{f}(i)$ average of $f(\omega)\omega^{-i}$

Quantum Fourier Sampling

- Fourier sampling: Given the list representation of f, output set S with probability $\hat{f}(S)^2$
- A computer would have to at least inspect all N values
- The Quantum Fourier Sampling algorithm solves the problem using logN elementary quantum steps

Forward Propagation for Polynomials

Algorithm FP (Forward Propagation for Polynomials)
Input: A polynomial $f(x,y,\ldots,z)$

1 If f is a constant, output $\nabla f = (0,0,\ldots,0)$

2 If f is a variable, say x, output $\nabla f = (1,0,\ldots,0)$

3 If f is of the form g+h, output FP(g)+FP(h)

4 If f is of the form g×h, output g×FP(h)+h×FP(g)

Circuits

- In computer science it is often useful to represent expressions as labeled graphs called circuits
- A circuit is a directed acyclic graph in which every source vertex is labeled by an input variable or a constant. Every other vertex (gate) is labeled by a basic operation
- The basic operations of an arithmetic circuit are + and ×. Richer circuits may allow others like divide and exponentiate

General Forward Propagation

- Given a circuit computing f as input, it computes circuits for its partial derivatives as output
- If a function $f(x_1,x_2,\ldots,x_n)$ has circuit size s, then ∇f has size Θ(ns)
- In a circuit C, each gate computes a function whose inputs are the functions computed at its children
- Denote by g both the node in the graph and the function computed by it. Let [g] stand for the type of gate at node g
- Given a gate operation [g] with d inputs and one output, its gradient [∇g] can be viewed as some other gate operation with (at most) d inputs and d outputs
- Chain rule for Forward Propagation

$$\frac{\partial g}{\partial x} = \left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \ldots, \frac{\partial w}{\partial x}\right)\cdot[\nabla g](u,v,\ldots,w)$$

$$= \frac{\partial u}{\partial x}\cdot\partial_u[g] + \frac{\partial v}{\partial x}\cdot\partial_v[g] + \ldots + \frac{\partial w}{\partial x}\cdot\partial_w[g]$$

- General Forward Propagation algorithm:

Algorithm FP
Input: A circuit C with n inputs

1 For every gate g in C with children u,v,…,w

2 Create a new gate ∇g $\quad (\partial_u[g], \partial_v[g], \ldots, \partial_w[g])$

3 For every input variable x

4 Create a new node $\frac{\partial g}{\partial x}$

5 Connect $\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \ldots, \frac{\partial w}{\partial x}$ and ∇g to $\frac{\partial g}{\partial x}$ using chain rule

6 Output resulting circuit ∇C

- Theorem: Each gate $\frac{\partial g}{\partial x}$ in ∇C computes the partial derivative of g w.r.t. input variable x
- In step 4 of the algorithm, each gate is augmented with Θ(n) new gates. In step 5 each wire contributes an extra times gate.
- If C has s gates and w wires, then ∇C will have Θ(n(s+w)) gates

Backpropagation

- Calculates partial derivatives by visiting the vertices in reverse topological order, starting at the output y and ending at sources $x_1, \dots, x_n$
- Upon visiting node z, backpropagation constructs a new gate that calculates $\frac{\partial y}{\partial z}$
- Computes ∇C of size $O(n+s+w)$
- Definition: Assume C is a circuit, y is a gate, and z is a vertex. Let $(C-z)$ be the circuit obtained by removing all edges that point to z from C and turning z into an input. Then y computes some function f in $C-z$. The partial derivative $\frac{\partial y}{\partial z}$ is the derivative of f w.r.t. z
- Suppose if we have a gate g whose out-edges point to gates $u, v, \dots, w$. In the circuit $(C-g)$, the output y depends on g via the gates $u, v, \dots, w$, each of which depends on g.
- Chain rule for backpropagation:

$$\frac{\partial y}{\partial g} = \frac{\partial y}{\partial u} \cdot \partial_g[u] + \frac{\partial y}{\partial v} \cdot \partial_g[v] + \dots + \frac{\partial y}{\partial w} \partial_g[w]$$

- Backpropagation algorithm:

  Algorithm BP (Backpropagation)

  Input: A circuit C

  1 Compute a reverse topological sort s of C's vertices starting with y

  2 For every gate g in s, construct a subcircuit for $\frac{\partial y}{\partial g}$ using chain rule

  3 Output resulting circuit ∇C

- Theorem: For any circuit C with designated output y and operations coming from B, the output ∇C of Backpropagation is a circuit with operation set $B \cup \nabla B \cup \{+, \times\}$ that contains gates computing $\frac{\partial y}{\partial u}$ for every node u of C. The number of gates in ∇C is at most three times the number of wires plus the number of nodes in C

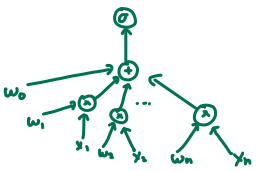
Neural Networks

- A perceptron is a model of a single neuron. It consists of two types of inputs: n signals $x_1, \dots, x_n$ and $n+1$ weights $w_0, w_1, \dots, w_n$. It outputs

$$y = \sigma(w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n)$$

σ activation function

- Perceptron circuit



- If we want to estimate the parameters, we can construct the loss function and minimize it. This is where forward and backpropagation can be useful
- When there's not many parameters, forward propagation and backpropagation produce circuits of similar size
- However, when there are many parameters, backpropagation produces much smaller circuits