Optimization (cont)

Intro2CS – week 13
Information about the exam

• Written exam (3 hours).
  – No extensions except for students with pre approved ones.

• Counts for 50% of your final grade in the course

• Material: *Everything* that we saw during the course.

• חומר סגור (Closed book)
  – also no calculators & no formula sheets
• Be there on time.
• This info is from the shnaton:
• Be sure to check it closer to the exam. Rooms may change!

• Be at the right place!
Information about the exam

• Focus is on:
  – Understanding the concepts taught in class and in tirgulim / targilim
  – Being able to think of basic algorithms

• You need to be able to write code in python, but we will forgive small mistakes (forgetting a comma here or there).

• Remember basic functions like len() and range(), how to write iterators, slicing, lambda expressions, etc.
  – we will provide syntax for more complex functions if they are needed.
Exam Questions

• Structure is different from previous years.
  – All answers must be given on the form.
  – Must be written with dark colored pen (blue / black). No pencils allowed!
  – You will be given exam notebooks for use as draft paper
  – We do not grade or look at the notebooks

• Part A: 10 small questions (4 points each)
  – Answers are short (fit into a small box).
  – Must answer all 10.

• 4 (out of 6) larger questions (15 points each).
  – Answers fit in under 10 lines (usually code)
  – Copy “clean” from draft.
How to practice for the exam?

• Go over the material. Make sure you understand it all.

• Solve exams from last year (there were fewer & longer questions, but the material is similar!)

• Practice writing code on paper
  – without a computer nearby
  – Without looking at the answers

• Time yourself!
  – You should do part A of the exam (short questions) in around an hour
  – Each of the larger questions in around ½ an hour.
Back to...

Optimization
Are there alternatives to brute force search?

• Yes! Unfortunately, for hard problems, they are not much more efficient.

For N-Queen Problems:

• We can try guessing: start with random assignments until we find a good solution.
• We can try to have a solution with conflicts and try to change only conflicted parts.
def set_random(board):
    conflicts = 0
    for col in range(BOARD_SIZE):
        row = random.randrange(BOARD_SIZE)
        if not can_place_queen_at(board, row, col):
            conflicts += 1
            board[row][col] = True
    return conflicts
def try_random_solutions():
    board = [[False]*BOARD_SIZE for _ in range(BOARD_SIZE)]
    counter = 0
    best = BOARD_SIZE+1
    while best:
        counter+=1
        clear_board(board)
        num_conflicts = set_random(board)
        if num_conflicts < best:
            best = num_conflicts
            print("round: ",counter," found:", num_conflicts, "conflicts")
    print_board(board)
```python
def try_changing_conflicted():
    board = [[False]*BOARD_SIZE for _ in range(BOARD_SIZE)]
    num_conflicts = set_random(board)
    best = num_conflicts
    counter = 0
    while best:
        col = random.randrange(BOARD_SIZE)
        for row in range(BOARD_SIZE):
            if board[row][col]:
                if not can_place_queen_at(board, row, col):
                    counter += 1
                    board[row][col] = False
                    board[random.randrange(BOARD_SIZE)][col] = True
                if not counter%1000:
                    print("round:", counter,
                          "  found:", num_conflicts, "conflicts")
                    print_board(board)
        break

    num_conflicts = count_conflicts(board)
    if num_conflicts < best:
        best = num_conflicts
    print_board(board)
```

Is it always this hard?

• Example for an easy constraint satisfaction problem: sort an array.

• Given an array of \( n \) numbers \( x_1, x_2, x_3, \ldots, x_n \) arrange them in order \( x_{\pi(1)}, \ldots, x_{\pi(n)} \) so that
  \[
x_{\pi^{-1}(i)} \leq x_{\pi^{-1}(i+1)}
  \]

• Can you phrase this as an optimization problem?

• If we are too naïve: the search space is all permutations of items
A greedy algorithm:

• Given an arrangement of the array, find two consecutive items that violate the constraint

\[ x_{\pi^{-1}(i)} \leq x_{\pi^{-1}(i+1)} \]

• Swap them.

We already know that this solves the problem – this is basically the bubble-sort algorithm!

\( O(n^2) \) is much better than \( O(n!) \)
Continuous problems

• Think of the following optimization problem:

• \( U = \mathbb{R} \)
• \( V(x) = x^2 \)
• Minimize \( V(x) \) for \( x \in U \)

(Forget for a moment that we know exactly where the minimum is)

• The search space is infinite.
• But, the value function is continuous.
Algorithm idea

• Start with some \( x \in \mathbb{R} \).

• Evaluate \( V'(x) = \frac{dV}{dx} \).

• If \( V'(x) > 0 \) decrease \( x \) (slightly)

• Else if \( V'(x) < 0 \) increase \( x \) (slightly)

\[
x := x - \alpha \cdot V'(x)
\]

This algorithm is known as “Gradient Descent” or “Hill Climbing” (in case of maximization)
Another a very similar algorithm:

- Start with some $x \in \mathbb{R}$.
- Randomly select a small $\epsilon$.
- Evaluate $V(x + \epsilon)$.
- If we improved, set $x := x + \epsilon$.

- Works when the function is shaped “like a bowl” (Convex).
Another a very similar algorithm:

• Start with some $x \in \mathbb{R}$.
• Randomly select a small $\epsilon$
• evaluate $V(x + \epsilon)$
• If we improved, set $x := x + \epsilon$

• May fail when the function has local minima
• We get “stuck”
Another example: Linear regression

• The problem:
• We are given data about the height and weight of many people
• We wish to find a connection between the two parameters.

• Lets assume connection is linear:

\[ \text{weight} = a \cdot \text{height} + b \]

Or

\[ y = a \cdot x + b \]
• In reality data is noisy.
What is the best fit?

• The question: Find parameters $a, b$ such that $y = a \cdot x + b$ is a “good fit” to the data.

• Consider a point $x_1, y_1$ how well does a given set of parameters fit?

• $error_1 = (y_1 - (a \cdot x_1 + b))^2$
Phrase as an optimization problem

- \( U = \{(a, b) \in \mathbb{R}^2\} \)
- \( V(a, b) = \sum_i \text{error} (a, b, x_i, y_i) \)
  \[= \sum_i (y_i - a \cdot x_i - b)^2 \]
# the function:

def func(a, b, x_data):
    return [a*x+b for x in x_data]

def error(a, b, x_data, y_data):
    expected = func(a, b, x_data)
    total_error = 0
    for i in range(len(expected)):
        total_error += (expected[i]-y_data[i])**2
    return total_error
def optimize(func, start, num_rounds, step):
    current = start
    value = func(current)

    for _ in range(num_rounds):
        new = [x + (random.random() - 0.5) * 2 * step for x in current]
        new_value = func(new)

        if new_value < value:
            current = new
            value = new_value

    return current, value
def parametrized_error(x_data, y_data):
    return lambda vars: error(vars[0], vars[1], x_data, y_data)

# find a best fit
params, score = optimize(parametrized_error(x_data1, y_data1),
                          [0, 0], 1000, 0.01)
print(params, score)

import matplotlib.pyplot as plt
# show the data with the fit
plt.plot(x_data1, y_data1, 'o', x_data1,
         func(params[0], params[1], x_data1), '-r')
plt.show()
Linear Regression

- Lucky for us, linear regression always converges.
- The reason: the error is convex ("bowl shaped")
Solving Analytically

• For linear regression, we have (linear) equations that we can solve to get the line’s parameters

\[ V(a, b) = \sum_i (y_i - a \cdot x_i - b)^2 \]

• \[ 0 = \frac{\partial V}{\partial a} = 2 \sum_i (y_i - a \cdot x_i - b) \cdot (-x_i) \]

• \[ 0 = \frac{\partial V}{\partial b} = 2 \sum_i (y_i - a \cdot x_i - b) \cdot (-1) \]

• These are two equations for 2 variables (a,b). Solve them and get the best fit!

• But for other functions, we don’t such a formula, and if the function isn’t convex, it is harder...
\[ y = \sin(a \cdot \pi \cdot x) + b \cdot x^2 \]

\[ a = 3 \quad ; \quad b = 0.5 \]
• Most problems are either convex (or can be translated into a convex problem somehow)

• Or really hard (we need to go over lots of options).

This is the “tragedy” of computer science.

“Everything interesting is hard.”
• Some functions may be hard for gradient descent
# the function:

def func(x_data):
    a = 3
    b = 0.5
    return [b*(math.sin(a*math.pi*x1) +
             math.sin(a*math.pi*x2) + (x1**2 + x2**2))
            for x1, x2 in x_data]
Many random starting points
Do gradient descent on each one...
• We are stuck at local minima.

• We don’t know that we have enough “resolution” and that we’ve seen all the minimums
Simulated annealing

• Accept improving steps always
• Accept a bad step with probability
  \[ p = \exp \left( \frac{-(v_{t+1} - v_t)}{T} \right) \]
• Slowly decrease T (to decrease p to 0).
Simulated annealing

• When \( p = 0 \), we are back to regular gradient descent.

• Probability of leaving local minimum is greater than probability of leaving global minimum.
Simulated annealing
def optimize_n(func, start, num_rounds, step):
    value = func(start)
    temp = 2

    for round in range(num_rounds):
        temp *= 0.9999
        change = [((random.random()-0.5)*2*step+x1,
                   (random.random()-0.5)*2*step+x2) for x1, x2 in start]
        new_value = func(change)

        for ind in range(len(start)):
            if new_value[ind]<value[ind] or 
               random.random() < math.exp((value[ind]-new_value[ind])/temp):
                start[ind] = change[ind]
                value[ind] = new_value[ind]

        if round%(num_rounds/5) == 0:
            plot_it(start, value)
            print(temp)

plot_it(start, value)
What is optimization useful for?
EMPLOYEES OF THE MONTH

JANUARY
ETHEL

FEBRUARY
MATT

MARCH
FRANK

APRIL
CAROL

MAY
MARTON

JULY
TOM

AUGUST
AUTOTRON

SEPTEMBER
AUTOTRON

OCTOBER
AUTOTRON

NOVEMBER
AUTOTRON

Kanin
\[ y = \sigma \left( \sum_{i} x_i \cdot w_i \right) \]
Deep neural network

\[ \vec{x}, \vec{y} \]
• A neural network is a function $f(\hat{x}, \hat{w})$.
• Given data $\hat{x}, \hat{y}$ we can define the error for some weights:

$$Error(\hat{w}, \hat{x}, \hat{y}) = \|f(\hat{x}, \hat{w}) - \hat{y}\|^2$$

• The problem in neural networks:
• Find weights $w_{i,j}$ such that the error of the network on data $\hat{x}, \hat{y}$ is minimized.
• This is typically solved through gradient descent!