UL01. Randomized Optimization

Optimization:

- An optimization problem typically includes:
  - Input space X
  - Objective (Fitness) function $f: x \to \mathbb{R}$
  - Goal: Find $x^* \in X$ such that $f(x^*) = max_x f(x)$
- Optimization is meant to help us:
  - Find the best process.
  - Find the best route.
  - Find the root.
  - Find the best parameters of the learning algorithms.

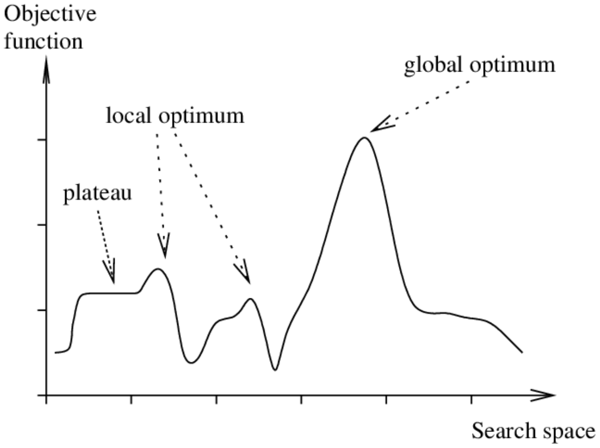
Optimization Approaches:

- Generate and test: Small input space – Complex function.
- Calculus: Function has derivative – solvable derivative = 0
- Newton's method: Function has derivative – Iteratively improve – Single optima.
- What if we have:
  - Big input space.
  - Complex function.
  - No (or hard to find) derivative.
  - Many local optima.
- In this case we use Randomized Optimization.

Hill Climbing:

- Guess $x \in X$.
- Repeat:

```
    Let neighbor $n^* = argmax_{n \in N(x)} f(n)$
    If $f(n^*) > f(x): x = n$
    Else: stop (Local minima)
```



Random Restart Hill Climbing:

- To avoid ending up with a local optima, once a local minima is reached, try again starting from a randomly chosen x.
  - Advantages:
    1. Multiple tries to find a good starting place.
    2. Not much more expensive (constant factor).
  - If there's only one optima, doing random restart will keep giving us the same answer.
  - Randomized Hill Climbing may not do better than evaluating all the space in the worst case, but it won't be worse.
  - Randomized Hill Climbing depends a lot on the size of the attraction basin around the global optima. Bigger basin results in better performance.

Simulated Annealing:

- Similar to Random Restart Hill Climbing.
- Difference: Don't always improve, sometimes you should explore the space instead of going upward the hill.
- For a finite set of iterations:
  1. Sample new point $x_+ \, in \, N(x)$ (Neighborhood of x).
  2. Jump to new sample with probability given by an acceptance probability function $P(x, x_+, T)$

$$P(x, x_+, T) = \begin{cases} 1 & if \ f(x_+) \geq f(x) \\ e^{\frac{f(x_+) - f(x)}{T}} & otherwise \end{cases}$$

  - We basically check if the fitness of the new point $f(x_+)$ is greater than or equal the fitness of the old point $f(x)$, jump to x_+.
  - If it's not improving $(f(x_+) < f(x))$, then we evaluate $e^{\frac{f(x_+) - f(x)}{T}}$ and take it as the probability:
    - → If the two points are really close to each other, the difference would be almost zero, and since
      $e^0 = 1$ we will make the move to x_+.
    - → If it's a big step down, $(f(x_+) < f(x))$ will evaluate to a big negative number, then the term $e^{\frac{f(x_+) - f(x)}{T}}$ will be very close to zero, and we will not make the jump.
    - → If it's a moderate step down, then the value of T will determine the decision:
      1. If T is big ($T \to \infty$), the term $e^{\frac{f(x_+) - f(x)}{T}}$ will evaluate to almost 1, and we'll always accept the next point (facilitates exploring the space).
      2. If T is small ($T \to 0$), the term $e^{\frac{f(x_+) - f(x)}{T}}$ will evaluate to 0, and we'll always reject the point (perform like hill climbing).
  3. Decrease temperature T but maintain $T > 0$
     - Decreasing T slowly gives the algorithm the change to reach the global minima basin of attraction before starting to act like hill climbing.

- The probability of ending at x:

$$P_r(ending\ at\ x) = \frac{e^{\frac{f(x)}{T}}}{Z_T}$$

- Decreasing T puts all the weight on $f(x)$ and eventually, pushes the probability to its maximum.
- However, we need to decrease T slowly to avoid ending up in a local minima.
- This is called Boltzmann Distribution.

Genetic Algorithms:

- Genetic Algorithms conducts a randomized, parallel, hill-climbing search for the hypotheses that optimize a predefined fitness function.
- The algorithm operates by iteratively updating a pool of hypotheses, called the population. On each iteration, all members of the population are evaluated according to the fitness function (A predefined numerical measure for the problem). A new population is then generated by probabilistically selecting the most fit individuals from the current population.
- Algorithm:
  - Initialize population: Generate an initial hypotheses population P_t of size K, where t indicates the t^{th} generation.
  - Evaluate: Compute fitness for all $h \in P_t$
  - Create a new generation P_{t+1}:
    1. Select "most fit" individuals according to the fitness function:
       → Truncation Selection: We take the top half of the population in terms of their scores and declare them to be the most fit.
       → Roulette Wheel Selection: We select individuals at random, but we give the higher scoring individuals a higher probability to be selected (similar to having a temperature parameter close to ∞).

$$P_r(h_i) = \frac{Fitness(h_i)}{\sum_{h_j \in P_0} Fitness(h_j)}$$

    2. Crossover: Probabilistically select pairs of hypotheses from P_0 according to $P_r(h_i)$. For each pair, produce an offspring by applying the crossover operator (Copying selected bits from each parent).
    3. Mutate: Choose a percentage of the members of P_0 with uniform probability. For each, invert one randomly selected bit in its representation.
    4. Update $P_t \leftarrow P_{t+1}$
    5. Evaluate: Compute fitness for all $h \in P$
    6. Repeat till converge.
- Genetic Algorithms is less likely to fall into a local minima, because it moves abruptly, replacing parents with offspring that might be radically different. As opposed to Gradient Descent which moves smoothly from one hypothesis to another that is very similar.

Problems with Randomized Optimization Algorithms:

- There's no structure or learning. You start with a point and end up with a point that is closer to the global optima.
- It's not clear what kind of probability distribution we're dealing with.

MIMIC - Finding Optima by Estimating Probability Densities:

- By directly modeling the distribution and successfully refining the model, we'll end up with a structure.

$$P^{\theta_t}(x) = \begin{cases} 1/z_\theta & if\ f(x) \geq 0 \\ 0 & otherwise \end{cases}$$

  - This probability is uniform over all values of x whose fitness is above some threshold θ.

- Pseudo code:
  - Generate samples from probability distribution $P^{\theta_t}(x)$ → Generate population.
  - Set θ_{t+1} to n^{th} percentile.
  - Retain only those samples such that $f(x) \geq \theta_{t+1}$ → Retain fittest.
  - Estimate $P^{\theta_{t+1}}(x)$ → Estimate a new distribution.
  - Repeat.
- What we're doing here is:
  1. We have some threshold θ.
  2. We generate a probability distribution that is uniform over all points that have a fitness value $\geq \theta$.
     - This means we generates all the points whose fitness is at least as good as θ.
  3. Take from those the points whose fitness is much higher than θ (Maybe highest 50%).
  4. Keep repeating till you reach θ_{max}.
- This way helps us retain the structure from time step to time step.
- This should work as intended if:
  - We can estimate $P^{\theta_{t+1}}(x)$ given a finite set of data.
  - $P^{\theta_t}(x) \cong P^{\theta_{t+1}}(x)$. That is, when I generate $P^{\theta_t}(x)$, it also gives me samples for the next distribution $P^{\theta_{t+1}}(x)$, because both distributions are relatively close.
- This will eventually lead us to θ_{max}, which convey the global optima.

MIMIC – Estimating Distributions:

- The chain rule version of a probability distribution would look like that:

$$P(x) = P(x_1|x_2 \dots x_n)P(x_2|x_3 \dots x_n) \dots P(x_n)$$

  - Every x is a vector of features $[x_1, x_2, x_3 \dots x_n]$
  - $P(x)$ would be the probability of all of the features of example x being a joint distribution of all the features $[x_1, x_2, x_3 \dots x_n]$.

- We cannot (very hard) estimate this because it’s an exponential size probability table.
- However, we can estimate it by making an assumption about conditional independence. That assumption would be that we only care about “Dependency Trees”:
  - A Dependency Tree is a special case of Bayesian Networks where the network itself is a tree. That means that every variable has exactly one parent.
- Having made this assumption, we can write the distribution as:

$$\hat{P}_\pi = \prod P(x_i | \pi(x_i))$$

  - Where $\pi(x_i)$ is the parent of x_i.
  - Here each variable has exactly one parent, which means that no feature is ever conditioned on more than one other feature. This makes the conditional probability stays very small (quadratic in the number of features).
- The simplest structure we can use is assuming that each variable has no parent $\prod P(x_i)$, but this means we assume that the features are completely independent of each other, which is not realistic all the time.
- Assuming a Dependency Tree (each variable has only one parent) is simple yet helps us capture the underlying relationships.
- The Dependency Tree also helps us capture the same locality information that we get from Cross Over in Genetic Algorithms.

Information Theory:

- In a machine learning setting, we need to know how each feature of the input examples relates to the output, and which feature has the highest effect. In other worlds, we need to figure out which of these features gives the most information about the output.
- These input/output vectors can be considered as a probability density function.
- Information Theory is a mathematical framework used to compare these density functions to determine:
  - Mutual information: The similarity between different vectors.
  - Entropy: Does the feature in hand carry any information about the output?
- To calculate the Entropy of variable x:

$$Entropy = -\sum P(x) \, log \, P(x)$$

- Joint Entropy: Randomness contained in two variables together.

$$H(x, y) = -\sum P(x, y) \, log \, P(y, x)$$

- Conditional Entropy: Randomness of one variable given another variable.

$$H(y|x) = -\sum P(x, y) \, log \, P(y|x)$$

- If x and y are independent $(x \| y)$:

$$H(y|x) = H(y)$$
$$H(x, y) = H(x) + H(y)$$

- Conditional Entropy is not an indicative measure of dependence:
  - $H(y|x)$ will be small if y is highly dependent on x, or if $H(y)$ is very small to begin with.
- This is why we measure the Mutual Information:

$$I(x; y) = H(y) - H(x|y)$$

- Kullback–Leibler Divergence: Measures the difference between any two distributions:

$$D_{kl}(p\|q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

  - It's a distance measure. But It's a non-negative quantity and equals zero only if $p(x) = q(x)$, since the log term will equal zero.

Finding Dependency Trees:

- Given an underlying probability distribution P, we want to find $\hat{P}_\pi$ that is the closest we can get to P.
- To get a sense of the similarity between these two distributions, we will measure the Kullback–Leibler divergence (from Information Theory) between the two of them:

$$D_{kl}(P\|\hat{P}_\pi) = \sum P[\log P - \log \hat{P}_\pi]$$

- To find the closest approximate distribution $\hat{P}_\pi$, we need to minimize $D_{kl}(P\|\hat{P}_\pi)$.
  - Substituting in $D_{kl}(P\|\hat{P}_\pi)$ using the information theory equations:

$$D_{kl}\left(P\|\hat{P}_\pi\right) = -H(P) + \sum H(x_i|\pi(x_i))$$

  - Given that we need to find π, the first term $-H(P)$ doesn't matter. We end up with a cost function J_π that we need to minimize:

$$J_\pi = \sum H(x_i|\pi(x_i))$$

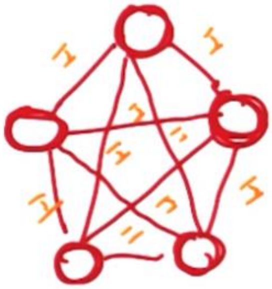
  - So, the best Dependency Tree would be the one that minimizes the sum of all Entropy of each feature given its parent.
  - To make easier to minimize J_π, we'll add another term to the equation that is not dependent on π and hence will not affect our calculations:

$$\grave{J}_\pi = -\sum H(x_i) + \sum H(x_i|\pi(x_i))$$

    Substituting in the Mutual Information function:

$$\grave{J}_\pi = -\sum I(x_i;\ \pi(x_i))$$

  - This means that minimizing the cost function is the same thing as maximizing the sum of the Mutual Information between every feature and its parent. Which makes sense, we want to associate every feature to the parent that gives the most information about it.
- How to implement this?
  - We start with a graph that has all the features as nodes and the mutual information between each feature and all the other features as edges.
  - What we need to do is to reduce that graph to the dependency tree that maximizes the sum of mutual information values. This is equivalent to finding the Maximum Spanning Tree.



MIMIC in Practice:

- When the optimal value we’re looking for depends only on the structure as opposed to the exact values, MIMIC tends to do well.
- We don’t only to represent the optima, we also represent everything in between.
- MIMIC finds the optima in orders of magnitude fewer iterations. But each iteration might take longer time.