Introduzione all'Ottimizzazione Black-Box

Progetto IA

Andrea Di Marco, 2022

0. Introduzione

In questa relazione farò un'introduzione all'ottimizzazione black-box. Parlerò delle tecniche principali e infine mostrerò degli esempi pratici utilizzando *Nevergrad* e *NOMAD* per ottimizzare alcuni iperparametri di un sistema di mia creazione.

L'obiettivo è quello di fornire un punto di accesso a chiunque possa essere interessato all'ottimizzazione black-box, campo estremamente affascinante dell'informatica e della matematica la cui vastità potrebbe però scoraggiare i nuovi arrivati.

1. Ottimizzare gli Iperparametri

Parte fondamentale di sistemi informatici e algoritmi sono gli ***iper-parametri***, ovvero parametri decisi *a priori* che influenzano la precisione e le decisioni degli algoritmi.
Parametri come profondità di un albero, numero di nodi da mantenere in frontiera, che peso dare ad un certo valore o scelta e così via.

Molto spesso dare un certo valore piuttosto che un altro ad un iper-parametro porta ad avere vantaggi su un lato ma svantaggi su altri, questo è anche detto ***trade off***. Ad esempio, dato un sistema con più output (come vedremo più avanti), dare più peso ad una certa scelta potrebbe influenzare positivamente un output, ma negativamente gli altri.

Più il sistema cresce e più diventa complesso *aggiustare* gli iper-parametri in modo da ottenere la performance e il risultato migliore. Per questo esistono diversi tool, dotati di molteplici algoritmi, per impostare i suddetti iper-parametri nel modo più efficiente possibile.

L'ideale sarebbe quello di provare tutte le configurazioni possibili di ogni ipe-parametro e poi scegliere la migliore, ma non avendo tempo infinito, possiamo effettuare solo un numero limitato di *trial* (o test).
Ovviamente, diversi algoritmi restituiranno risultati differenti, ma allora in che modo scegliere quello migliore?

I principali algoritmi di ottimizzazione Black-Box sono questi cinque:

I. **Grid Search**
II. **Random Search**
III. **Bayesian Optimization**
IV. **Bandit**
V. **Genetic Search** (population-based training)

Chiaramente ne esistono molti di più che però possono essere ricondotti a varianti dei cinque elencati sopra.

1.1 Grid Search

Il Grid Search è la tecnica più semplice.

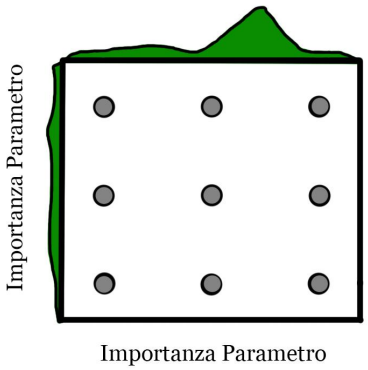
Ogni iperparametro viene diviso in sezioni, *discretizzato*, e valutato *a salti*, in modo da ridurre significativamente il numero di *trial* da dover fare per poter scandire lo spazio delle configurazioni possibili.

```
for param_1 in [0.001, 0.01, 0.1]:
     for param_2 in [2, 3, 4]:
          for param_3 in [“a”, “b”, “c”]:
               trial(param_1, param_2, param_3)
```

Come si può immaginare è facilmente parallelizzabile, con *semi-**embarrassing parallelism***, grazie al fatto che i trial non si influenzano a vicenda, è sufficiente istanziare *p* processi di tipo *slave* che ricevono le configurazioni dal processo *master* il quale alla fine di tutto ritornerà la configurazione che ha ottenuto il risultato migliore.

Inoltre ha un'ottima ***explainability*** in quanto fornisce molte informazioni su come gli iperparametri si influenzano l'un l'altro. L'algoritmo proposto sopra con cicli *for* annidati permette di accorgersi subito come, ad esempio, il `param_1` è influenzato dal `param_2` perché ci permette di osservare come cambia il risultato, fissato nel ciclo il `param_1` e variando `param_2`.

Di contro però, è estremamente inefficiente, dato che prova tutti i *trial* nello spazio delle configurazioni (discretizzato), e questo approccio fa anche sì che l'algoritmo fallisca nel focalizzarsi sulle configurazioni più importanti al soddisfacimento dei vincoli.



1.2 Random Search

Il Random Search è esattamente quello che sembra.
La configurazione di ogni *trial* è scelta prelevando *casualmente* gli assegnamenti degli iper-parametri dal rispettivo dominio, per un numero prefissato di volte, spesso detto *budget*. Dopo di che ritorna la migliore configurazione trovata.

```
for i in budget:
     trial(
          param_1 = sample(0.001, 0.01),
```

```
            param_2 = sample(2, 4),
            param_3 = sample([“a”, “b”, “c”])
            )
```

Come il *Grid Search*, anche il *Random Search* è facilmente parallelizzabile e, nei casi in cui si ha un numero molto elevato di parametri, è estremamente efficiente, in quanto non perde tempo a provare ogni configurazione possibile.
Questi casi sono detti *high dimension*.

Di contro però presenta una serie di problemi, ad esempio l'iper-parametro `budget`. Quanto dovrebbe essere alto per avere una qualche garanzia di ottimalità?
Per questo motivo il *Random Search* rischia spesso di essere una scelta troppo costosa.

1.3 Ottimizzazione Bayesiana

L'*Ottimizzazione Bayesiana* o, più in generale, l'ottimizzazione *model-based* nasce per risolvere il problema principale degli algoritmi visti sopra: la mancanza di memoria.

Sia il *Grid Search* che il *Random Search* fanno trial indiscriminatamente senza preoccuparsi di pesare le configurazioni e di tentare quella *migliore* secondo le informazioni ottenute finora.
Abbiamo quindi un oggetto con conoscenza del mondo (modello) che ricevendo la configurazione e lo ***score*** (punteggio) del *trial* precedente attraverso un ***tell***, è in grado poi di fornire la prossima configurazione da testare tramite un ***ask***, con lo scopo di ottenere uno *score* migliore.

Questi algoritmi non sono parallelizzabili allo stesso livello del Grid Search e del Random Search ma i risultati proposti sono molto più accurati e attendibili.

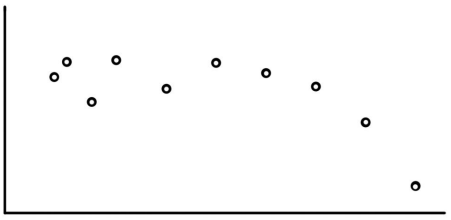
```
opt = Optimizer(
            param_1 = (0.01, 0.1),
            param_2 = (2, 5)
      )

for i in range(budget):
      config = opt.ask()
      score = trial(config)
      opt.tell(config, score)
```

1.3.1 Modello

Per poter effettuare previsioni sui *trial* futuri, questi tipi di algoritmi generano quindi un modello che approssimi la funzione da ottimizzare.

Ricordiamoci che ci troviamo in un contesto Black-Box, ovvero non abbiamo modo di accedere alla funzione ***f***, l'unica cosa che possiamo fare è dare un input ***x*** (gli iper-parametri) ed osservarne l'output ***f(x)***.

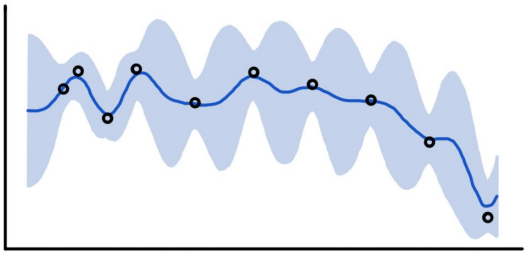


Ci sono diversi metodi per generare un modello per la nostra *f*, quelli utilizzato dalla stragrande maggioranza della letteratura sono i ***Gaussian Processes* (GP)**, ovvero un processo su una funzione *f(x)* tale che ogni collezione finita di **input** $x_1,\ldots,x_n$ abbia una **distribuzione Gaussiana** associata.

$$\begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \underbrace{\begin{bmatrix} k(\mathbf{x}_1,\mathbf{x}_1) & \ldots & k(\mathbf{x}_1,\mathbf{x}_n) \\ \vdots & & \vdots \\ k(\mathbf{x}_n,\mathbf{x}_1) & \ldots & k(\mathbf{x}_n,\mathbf{x}_n) \end{bmatrix}}_{\mathsf{K}}\right)$$

Dove $k(x_i, x_j)$ rappresenta la **covarianza** fra una coppia di input.
La **media** è il primo argomento di *N*, ovvero *0*.

Ottenendo così un modello



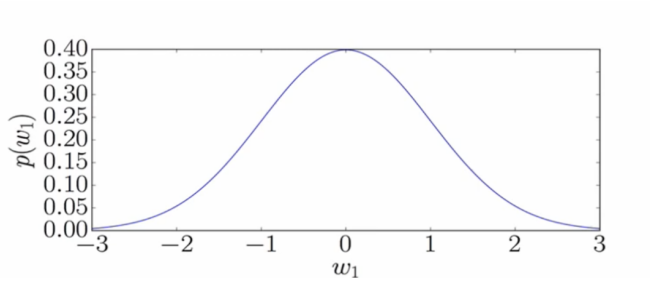
Siamo così riusciti ad ottenere un modello *M* che sia in grado di fare predizioni ma allo stesso tempo avere un grado di incertezza su queste previsioni.

1.3.1.1 Gaussian Process (GP)

L'idea è che, dato un insieme Y di risultanti da $f(x)$, quello che si fa è tracciare tutte le possibili funzioni h tali che $h(x) = f(x) = y \in Y$ e assegnare ad ognuna una probabilità p.
Questo si fa con i *processi Gaussiani*.

La distribuzione classica di probabilità di *Gauss* è definita con una media μ, e una covariante Σ.

$$w \sim N(\mu, \Sigma)$$



La distribuzione sopra è stata ottenuta con $\mu = 0$, $\Sigma = 1$ e si estende su un vettore ***uni**dimensionale w*.

Un *Gaussian Process* (*GP*) è quindi una distribuzione di *Gauss* ma bidimensionale, estesa su una funzione

$$f \sim GP(\mu, K(X, X))$$

dove μ è il valore atteso

$$\mu(x) = \mathbb{E}[f(x)]$$

e *K* è la covarianza delle coppie di valori

$$K(x_1, x_2) = \mathbb{E}\Big[\big(f(x_1) - \mu(x_1)\big)\big(f(x_2) - \mu(x_2)\big)\Big]$$

Un *GP* è quindi un insieme di variabili aleatorie.
La ***f*** che vogliamo trovare è una delle tante funzioni nella distribuzione.

La covarianza ***K(x₁, x₂)*** ci da quindi una misura di quanto x_1 e x_2 sono **simili** e quindi di quanto ***f(x₁)*** e ***f(x₂)*** *dovrebbero* essere simili.
Stiamo infatti assumendo che con **leggere perturbazioni** dell'input *x*, anche l'output *f(x)* cambierà di poco. Quindi che dato un $f(x_1)$ **noto**, $f(x_2)$ non ancora noto sarà nell'intorno di $f(x_1)$ se la covarianza ***K(x₁, x₂)*** ci dice che sono simili.

Una funzione alternativa per il calcolo della covarianza è la *squared exponential kernel*:

$$K(x_1, x_2) = \sigma^2 exp\left(-\frac{(x_1 - x_2)^T (x_1 - x_2)}{2l^2}\right)$$

Dove ***l*** influisce sulla larghezza della curva, mentre ***σ*** influisce sull'altezza.
Entrambi i parametri definiscono l'influenza generale della distribuzione.

La media ***μ(x)*** invece ci mostra il valore atteso della funzione sconosciuta. Se non ci sono $K(x_1, x_2)$ abbastanza simili, sarà la media $\mu(x)$ a decidere il risultato della Gaussiana che, nella maggior parte dei casi, non avendo abbastanza informazioni su *f*, si usa $\mu(x) = 0$.

1.3.2 Euristiche

Una volta ottenuto il modello della funzione, ci occorre definire un criterio, che per forza di cose non può garantire l'ottimalità, secondo cui scegliere la prossima configurazione da provare nel *trial*.

Questa euristica è tradizionalmente chiamata ***α(x)***, una funzione che prende in input la configurazione x e restituisce *quanto* quella configurazione sia *buona*.

Alcune euristiche non sono sempre fattibili, in quanto hanno bisogno che il modello permetta delle azioni specifiche, nelle prossime sezioni entrerò nel dettaglio su queste azioni.
Ogni euristica però, necessita che il modello permetta due azioni:

I. **Observe(M, x, y)**: registrare le osservazioni
II. **Predict(M,x)**: fare una predizione futura, ovvero $\alpha(x)$

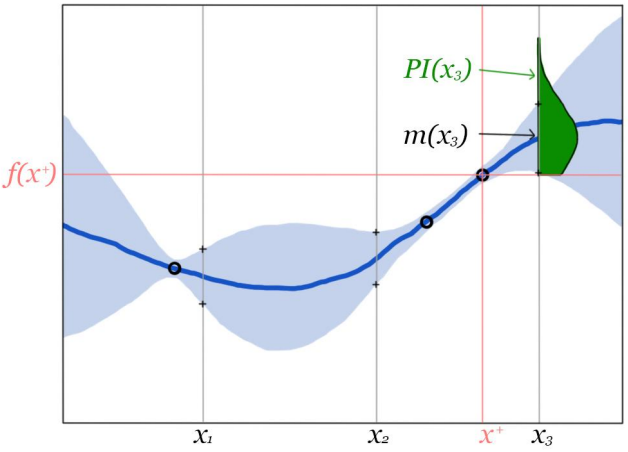
Senza queste due capacità non è possibile fare Ottimizzazione Bayesiana.

1.3.2.1 Probability of Improvement (PI)

La funzione ***α(x)*** consiglia la prossima x con più alta probabilità di essere migliore del risultato migliore trovato finora ***f(x⁺)***.

$$P\left[f(x) > f^{+} + \xi\right] = \phi\left(\frac{\mu(x) - f^{+} - \xi}{\sigma(x)}\right)$$

Dove $\phi(Z)$ è il Normal *Cumulative Density Function* (*CDF*) ovvero una Normale di Gauss con media $\mu = 0$.



Nella figura l'area verde è l'area ritornata da $\boldsymbol{\phi(x_3)}$ ovvero la probabilità di ottenere un valore più grande di $f(x^{+})$ utilizzando $\mathbf{x_3}$ come configurazione.
Con questa tecnica $\alpha(x)$ non considera nemmeno le configurazioni con $f(x_i) < f(x^{+})$.

Il problema di questa tecnica è che è molto, molto greedy, e questo non è sempre un bene, soprattutto nel caso di ottimi locali.

Per poter effettuare *PI* deve essere possibile chiedere al modello:

GetTail(M, v, x): probabilità di superare il valore v con la configurazione x

1.3.2.2 Expected Improvement (EI)

Consiglia la configurazione x con la media dei risultati $f(x)$ più alta.
Questa euristica è efficiente e facile da implementare.

$$\alpha(x) = \mathbb{E}\left[max(0,\ f(x) - f^{+} - \xi)|M\right]$$

Per poter effettuare *EI* deve essere possibile chiedere al modello:

GetImprovement(M, v, x): la probabilità di superare v con la configurazione x

1.3.2.3 Upper Confidence Bound

Consiglia la configurazione x con limite superiore più alto, ovvero la configurazione che può ritornare il più alto *f(x)* con probabilità $\boldsymbol{\delta}$.

L'idea di questa tecnica è quella di avere una probabilità di *Upper Bound* sulla funzione vera *f(x)*

$$\alpha(x) = q \qquad dove \qquad Pr[f(x) \leq q|M] = 1 - \delta$$

Questa tecnica è anche detta *ottimistica* di fronte alle incertezze.

Per poter effettuare *UPB* deve essere possibile chiedere al modello:

GetQuantile(M, q, x): il q-esimo quantile

I quantili, in statistica, sono valori che indicano la distribuzione di una popolazione. Il q-esimo quantile della variabile aleatoria x indica quindi quanti *f(x)* sono dentro a quel quantile.

1.3.2.4 Thompson Sampling

Considera un'acquisizione randomizzata dal modello con *Sample(M,x)*, e ritorna il valore più alto trovato.

La componente randomica fa sì che l'algoritmo non si *"incastri"* in un plateau, come gli altri precedenti.

Per poter effettuare *Thompson Sampling* deve essere possibile chiedere al modello:

Sample(M,x): estrarre un risultato *f(x)* dal modello a partire dall'input x

1.3.2.5 Predictive Entropy Search

Fra tutte le euristiche citate, questa è quella più recente e quindi la meno utilizzata. Diversi paper però sembrerebbero mostrare che sia una tecnica molto promettente.

Consiglia la configurazione x con il guadagno informativo più alto atteso, ovvero la configurazione x il cui risultato *f(x)* ci dovrebbe permette di ottenere più informazioni sulla funzione f e quindi di poter rifinire meglio il modello.

$$\alpha(x) = \; - \mathbb{E}_{y_x}\left[\mathbb{H}\left[f|M \cup \{y_x\}\right]|M\right]$$

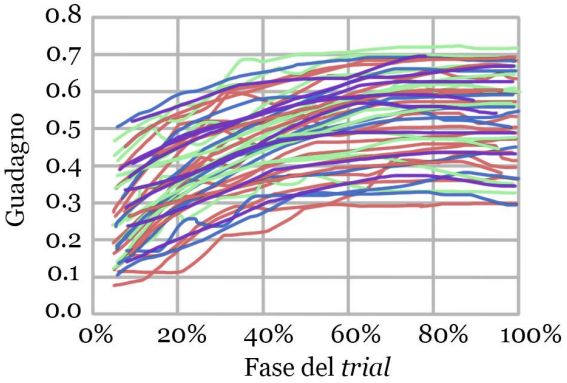
Il calcolo dell'entropia però può essere dispendioso, per questo motivo questa tecnica si accompagna spesso al Thompson Sampling per velocizzare i calcoli.

Per poter effettuare *Thompson Sampling* deve essere possibile chiedere al modello:

GetEntropy(M, x): l'entropia della predizione x

1.4 Algoritmi Bandit

Da dati sperimentali è possibile vedere come la maggior parte dei trial vanno male, quindi perché sprecare risorse su trial non *buoni*?



Vorremmo quindi comparare le performance relative dei trial ed interrompere preventivamente quelli con punteggi *peggiori*, mantenendo però attivi i trial con i punteggi *migliori*.

Negli algoritmi bandit, ogni trial viene eseguito fino ad un certo punto di *cutoff*, a quel punto viene messo a confronto con i trial arrivati nello stesso punto. Se le performance del trial in considerazione sono nella *fetta migliore* finora, allora si estende il punto di *cutoff* e si fa continuare il trial, altrimenti lo si termina e si manda in esecuzione un nuovo trial.

```
for i in range(budget):
      # get a new trial
      trial = sample_from(hyperparameter_space)
      while trial.iter < max_epochs:
            # epoch = fase del trial
            trial.run_one_epoch()
            if trial.at_cutoff():
                  if is_top_fraction(trial, trial.iter):
                        trial.extend_cutoff()
                  else:
                        # allow new trials to start
                        trial.pause()
                        break
```

1.4.1 Problemi Bandit

Non tutte le funzioni possono essere ottimizzate utilizzando algoritmi Bandit, soltanto le funzioni che rispecchiano i criteri dei problemi Bandit e che, ovviamente, possono essere scomposte in *epoch*.

I *Bandit Problems* presentano le seguenti caratteristiche:

- *Baby reinforcement learning*: Non si hanno conoscenze pregresse sul mondo
- *Acting in the face of uncertainty*: Non si hanno conoscenze sui risultati delle nostre azioni
- *No planning*: la natura imprevedibile del problema ed il rumore non ci permette di pianificare nel lungo termine

Abbiamo quindi un insieme finito di *n* azioni *A* da compiere in sequenza, ciascuna con diversi modi in cui compierla, per ogni $a \in A$ c'è quindi una distribuzione sconosciuta (dovuta al rumore e alle scelte) P_a. Il learner può solo scegliere un'azione $a \in A$ e osservare la ricompensa $R_a \sim P_a$.
L'obiettivo è massimizzare

$$\sum_{t=1}^{n} R_t$$

Ovvero massimizzare la somma delle ricompense delle azioni compiute.
La natura astratta di questi algoritmi li rende molto utili in diversi settori. Alcune delle applicazioni più famose sono:

- *Clinical trials* (terapie)
- *A/B testing* (GUI, …)
- *Ad placement*
- *Network routing*
- *Game tree search*

Il nostro obiettivo di massimizzare il guadagno di ogni azione, può essere tradotto con l'obiettivo di minimizzare il *regret* (*rimpianto*), ovvero la perdita di guadagno.
Sia μ_a il valore atteso di P_a e $\mu^\star = max_{a \in A} \mu_a$ allora il *regret* dell'*n*-esima azione è

$$\mathfrak{R}_n = n\mu^\star - \mathbb{E}\left[\sum_{t=1}^{n} R_t\right] = \mathbb{E}\left[\sum_{t=1}^{n} \left(\mu^\star - R_t\right)\right] = \mathbb{E}\left[\sum_{t=1}^{n} \Delta_{A_t}\right]$$

Dove $n\mu^\star$ rappresenta il guadagno massimo possibile.

Ci occorre sapere con che tipo di distribuzione abbiamo a che fare (*Bernoulli, Gaussiana, etc*) perché a seconda del tipo di distribuzione si avranno diversi algoritmi.

1.4.1.1 Contextual Linear Bandits

Rappresentano una versione *ridotta* dei problemi Bandit, ma più efficiente e facile da gestire.

Si ha un insieme di tutte le azioni possibili $A_t \subset R^d$
Scegliendo un'azione $a_t \in A_t$, la ricompensa sarà

$$X_t = \left(A_t, \theta\right) + \eta_t$$

dove $\theta \in R^d$ è sconosciuto e η_t è il rumore.
Abbiamo quindi un'infinità di azioni ma solo *d* parametri da dover scegliere.

1.4.2 Euristiche

Una volta assicurato che il nostro problema sia effettivamente risolvibile con algoritmi Bandit, ci resta da chiedere: come poter confrontare i *trial* fra loro? Come bilanciare *exploration* ed *exploitation*?

1.4.2.1 ε-First

Fissati ε ed n punti di *cutoff*. Decidiamo allora di fare *exploration* randomica per i primi $(n*\varepsilon)$ *trial*. (**exploration**)
A quel punto si salveranno gli $\varepsilon*(n-1)$ migliori *trial* e si riprende fino al prossimo punto di *cutoff*. Alla fine si ritorna la configurazione migliore tra gli ε trial rimasti (**exploitation**)

1.4.2.2 ε-Greedy

La scelta di exploration/exploitation sarà decisa dal parametro ε.
Con probabilità $(1-\varepsilon)$ si faranno continuare solo i *trial* più promettenti, altrimenti si sceglie randomicamente un *trial* fra quelli poco performanti da far continuare al posto di un *trial* più performante.

1.4.2.3 Upper Confidence Bound

Vogliamo dare una stima del valore atteso di ogni azione ed eseguire soltanto le azioni che sono *statisticamente plausibilmente* ottimali (**exploitation**).
Però allo stesso tempo vogliamo incoraggiare l'esplorazione entro un certo livello, ovvero vogliamo eseguire alcune azioni, anche se non *statisticamente plausibilmente* ottimali. (**exploration**)

Le strategie ottimistiche sono pericolose nelle situazioni in cui non è possibile "tornare indietro" ma questi non sono problemi *Bandit*, infatti noi assumeremo sempre che una sequenza di azioni "sbagliate" ci faccia al massimo perdere punti ma che non sia mai distruttiva. Sarà sempre possibile "riprovare". Lanciarsi da un grattacielo NON è un problema Bandit!

Assumendo la distribuzione come una *Gaussiana*, allora dato un insieme $A_1, \ldots, A_T$ di variabili aleatorie, il valore atteso previsto sarà semplicemente

$$\hat{\mu} = \frac{1}{T}\sum_{t=1}^{T} A_t$$

Ma quanto è accurata questa predizione? Dipende dalla varianza, normalizzata $\delta \in (0, 1)$. Allora abbiamo che

$$P\left(\hat{\mu} \leq \mu - \sqrt{\frac{2\log\left(\frac{1}{\delta}\right)}{T}}\right) \leq \delta$$ che la previsione sia più piccola del valore corretto

$$P\left(\hat{\mu} \geq \mu + \sqrt{\frac{2\log\left(\frac{1}{\delta}\right)}{T}}\right) \leq \delta$$ che la previsione sia più grande del valore corretto

Negli algoritmi Bandit vogliamo comportarci come se ci trovassimo nel mondo migliore esistente, dove i valori attesi sono il più alto possibile restando però nel limite del plausibile. Il punteggio (*reward*) atteso di un singolo epoch è quindi calcolato

$$R_t = argmax_a \hat{\mu}_a(t) + \sqrt{\frac{2\,log\left(\frac{1}{\delta}\right)}{T_a(t)}}$$

Dove:

$\hat{\mu}_a(t)$ = valore atteso dell'azione a dopo lo step t

$T_a(t)$ = numero di step rimanenti dopo aver eseguito a al passo t

δ = livello di confidenza, solitamente $\delta = \frac{1}{n^2}$

Ovvero considerando il valore atteso dell'azione che ha il valore atteso migliore (*ottimismo*), sommato all'errore di aver sottostimato il risultato. (**exploitation**)

L'algoritmo però considererà una a sub-ottimale (**exploration**) se

$$\mu_a + 2\sqrt{\frac{2\,log\left(\frac{1}{\delta}\right)}{T_a(t-1)}} \geq \mu_{a^*}$$

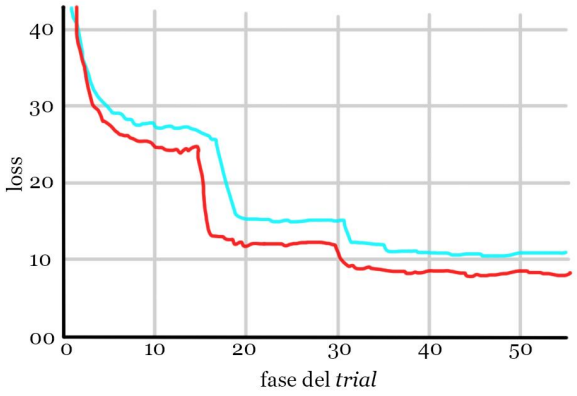
Si moltiplica il fattore di sicurezza per 2 perché si considerano entrambi i lati dell'errore nella previsione (*sotto*-stima e *sopr*a-stima).

Questa strategia ci permette di monitorare le diverse azioni e capire quale potrebbe beneficiare più dall'exploration e quale dall'exploitation.

É bene notare come questa implementazione non è ottimale in quanto si ferma a "scegliere" l'azione migliore ad ogni passo. Nei problemi di ottimizzazione black-box, si utilizza la modellazione vista sopra del *Contextual Linear Bandits*, in cui i parametri sono numeri reali scelti all'inizio e non cambiano nel corso degli *epoch*, quindi il calcolo risulta molto più veloce!

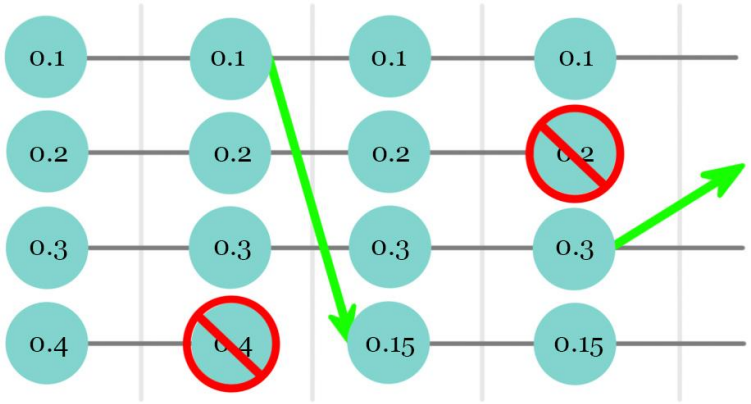
1.5 Algoritmi Genetici

Sperimentalmente l'ottimizzazione incappa spesso in plateau, e quindi gli algoritmi *Bandit* potrebbero non essere sempre la scelta migliore.



Come risolvere questo problema? Con gli algoritmi genetici o *PBT* (*population-based training*) che nel nostro caso specifico dell'ottimizzazione black-box, sono variazioni degli algoritmi *Bandit* chiamati *Population-Based Bandit Optimization* (*PBB*).

Proprio come i normali algoritmi *Bandit* vogliamo eseguire più *trial* fino al punto di *cutoff*, terminare i peggiori e continuare i migliori. La differenza è che gli algoritmi PBB, oltre a continuare i trial migliori, li duplicano. Si fanno partire nuovi *trial* le cui configurazioni si trovano nell'intorno del *trial* genitore appartenente alla popolazione migliore trovata finora. (**exploitation**)
In più c'è la possibilità di clonare alcuni dei *trial* peggiori oppure di far variare uno o più parametri dei *trial* figli, molto di più che l'intorno del padre. (**exploration**)



La difficoltà, come al solito, è decidere in che modo alterare i parametri nel momento della clonazione del *trial*. Farli variare nell'intorno, ma nell'intorno di cosa?
L'estrema versatilità di questi algoritmi rende facile associarli ad altre tecniche viste prima, come (appunto) gli algoritmi *Bandit* ma anche gli algoritmi *model-based* come l'Ottimizzazione Bayesiana.

Alcune tecniche infatti prevedono di variare i parametri nei figli nell'intorno del punteggio dei parametri, ad esempio prendendo una configurazione con valori molto diversi ma con lo stesso valore di *Entropy Improvement* oppure di *Upper Confidence Bound*. Per questo motivo le euristiche utilizzate non sono diverse da quelle viste fino a questo punto.

2. Il Modello

In questa sezione parlerò brevemente del modello creato con *OpenModelica* i cui iper-parametri sono intenzionato ad ottimizzare, in modo da rendere il resto della relazione di più facile comprensione.

Il modello in questione gestisce *n* droni all'interno di un'arena di dimensioni 200^3.
Gli *n* droni sono gestiti da un unico ***Centro di Controllo*** (***CC***) che si occupa di impostare le rotte dei droni e dirigerli verso le *aree* dell'arena con l'obiettivo di sorvegliare la zona.

2.1 Drone

Ogni drone parte dalla *docking bay* esterna all'arena e ha una batteria che si scarica nel tempo. Appena raggiunge un livello di carica pericolosamente basso detto *dangerousBatteryLevel*, questo ritornerà alla docking bay per ricaricare la batteria.
Finché la batteria è sufficientemente carica il drone si dirigerà verso il *set point* fornito dal *CC*.

Di default, il numero di droni è *3*.

2.2 Centro di Controllo

Parte fondamentale del sistema, nonché la componente di cui vogliamo ottimizzare gli iperparametri.

Ogni ***Tcc*** secondi (di default *1.5*) controlla la posizione di tutti i droni e quali aree sono state visitate e comunica con il database *SQL* nel *server*. A quel punto assegna ad ogni drone non *occupato* il nuovo *set point*. Un drone è da considerarsi *occupato* se non ha ancora raggiunto l'ultimo *set point* assegnato.

Quale area assegnare a quale drone è deciso pesando due possibili aree con un parametro normalizzato ***cc_choice***.

A quale delle due scelte il *CC* dovrebbe dare più peso?

I. L'area non ancora visitata, **più vicina al drone**.
II. L'area non ancora visitata, **lasciata scoperta da più a lungo**.

Di default il parametro *cc_choice* ha valore *0.5*, ovvero pesa entrambe le scelte allo stesso modo in quanto il calcolo è:

*I. choice_value[1] = distanza_area * cc_choice*
*II. choice_value[2] = tempo_scoperto * (1 – cc_choice)*

cc_choice e ***Tcc***, saranno i due parametri che proveremo ad ottimizzare nel corso di questa relazione.

2.3 Monitor

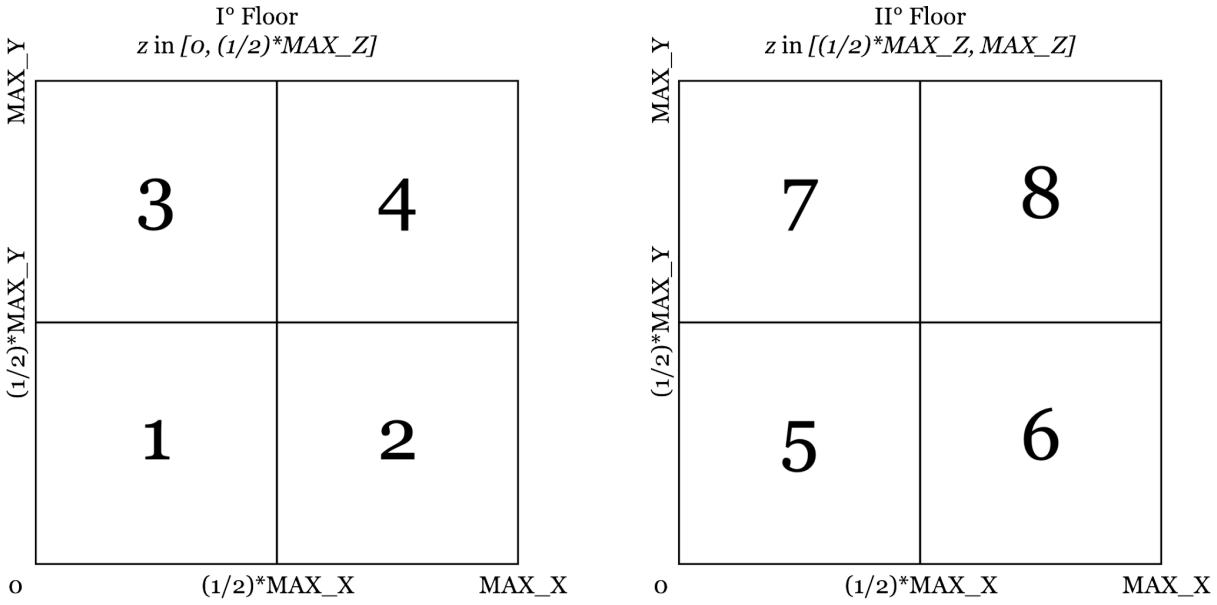
Il monitor è una componente scollegata da tutte le altre, prende le informazioni che gli servono dal *server* (quindi con *query* sul *DB*) e calcola i seguenti fattori:

I. ***AvgDrones***: Numero atteso di droni fra tutte le aree
II. ***PrDrones***: Probabilità di trovare un drone nell'area
III. ***StdDev***: Deviazione standard del numero di droni di ogni area
IV. ***AvgNoDrone***: La media del tempo per cui ogni area è lasciata scoperta
V. ***MaxNoDrone***: Il tempo massimo per cui un'area è stata lasciata scoperta

Questi fattori saranno l'*output* finale del sistema e rappresentano le diverse funzioni obiettivo che, con i tool di ottimizzazione, vorremmo minimizzare (o massimizzare).

2.4 Aree

Le aree sono *8* e si dispongono nell'arena nel seguente modo



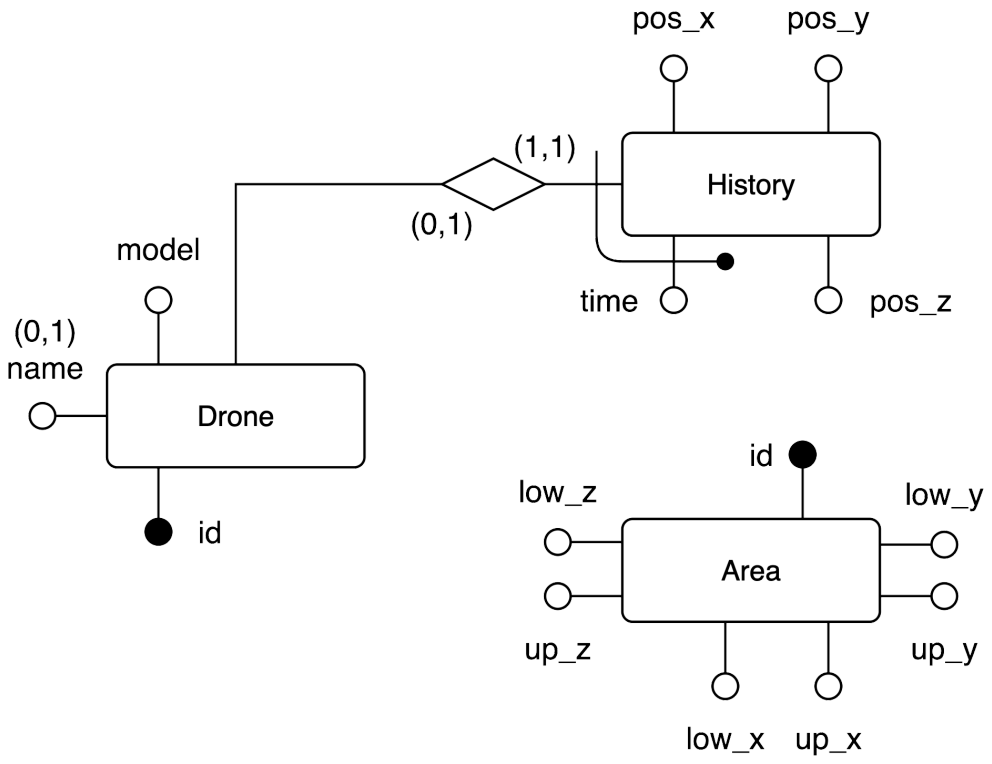
2.5 Server

Un server scritto in *C* che si occupa di fare da tramite fra il *CC* e il *Database*.

2.6 Database SQL

Un database implementato con *PostgreSQL* che si occupa di mantenere lo storico delle posizioni dei droni e i limiti geografici delle aree.

Il *CC* dovrà fare quindi diverse query per inserire le posizioni dei droni del *DB* e per sapere se una certa area è stata visitata o meno.



Che

3. Nevergrad

3.1 Introduzione

Nevergrad è una libreria di *Python*, sviluppata da *Meta*, che utilizza algoritmi di *IA* (simili a quelli visti sopra) per ottimizzazioni Black-Box senza gradienti (da qui il nome).

L'obiettivo di *Nevergrad* è trovare la combinazione migliore di iperparametri per minimizzare l'errore (***loss***) di un modello, o massimizzare le performance (***gain***) offrendo una vasta gamma di algoritmi diversi di ottimizzazione tra cui scegliere.

3.2 Modellazione

Per prima cosa occorre la funzione da ottimizzare. Nel caso di *Nevergrad* questa funzione può prendere in input diversi parametri, di diverso tipo, ma deve ritornare in output rigorosamente un ***float***, che sarà il valore di *loss* o *gain*.
Non tutti gli algoritmi di *Nevergrad* permettono di massimizzare il gain, in quei casi basta invertire il segno del gain e spacciarlo per funzione di loss, chiedendo all'ottimizzatore di minimizzare l'output.

Si deve poi definire una struttura chiamata ***Instrumentation*** dove poter specificare i parametri della funzione con i vincoli di *upper bound* e *lower bound*.

```
parametrization = ng.p.Instrumentation(
          # un reale distribuito tra 0.001 e 0.1
     param_1 = ng.p.Log(lower=0.001, upper=0.1),
          # un intero da 2 a 6
     param_2 = ng.p.Scalar(lower=2, upper=4).set_integer_casting(),
          # parametro con dominio enumerativo
     param_3 = ng.p.Choice(["a", "b", "c"])
)
```

A questo punto occorre scegliere l'algoritmo di ottimizzazione detto ***optimizer***, comunicargli i parametri, il *budget* e il numero di *worker*, ovvero il numero di processi da eseguire in parallelo.

```
optimizer = ng.optimizers.NGOpt(parametrization=parametrization,
budget=100, n_workers=1)
```

Infine possiamo eseguire ed ottimizzare, specificando la funzione obiettivo e in che modo ottimizzare (*maximize* o *minimize*).

```
recommendation = optimizer.minimize(our_function)
```

Per stampare i risultati si usa la keyword kwargs dell'oggetto ritornato dall'ottimizzatore.

```
# mostra i valori consigliati insieme alle keyword associate
print(recommendation.kwargs)
```

In alternativa è possibile utilizzare l'*API* di cui è dotato ogni ottimizzatore: le funzioni di ***ask()*** e ***tell()***

```
# Crea un metodo di Ottimizzazione di Nevergrad
optimizer = ng.optimizers.registry[optimizer_name](instrumentation,
budget=budget, num_workers=num_workers)
recommendation = optimizer.minimize(model)

# Interfaccia iterativa
for i in range(budget):
     # chiedi l'iper-parametro
     nevergrad_hp[j] = optimizer.ask()
     nevergrad_hp_val[j] = nevergrad_hp[j].value

     for j in range(num_workers): # versione parallela
          score[j] = norm(nevergrad_hp_val[j]

     for j in range(num_workers):
          optimizer.tell(nevergrad_hp[j], score[j])
```

Su Nevergrad i diversi ottimizzatori sono elencati nel dizionario ***registry***, per questo si possono chiamare dando in input il nome dell'algoritmo come stringa.

3.3 Algoritmi

I molteplici algoritmi di *Nevergrad* hanno diversi pro e contro, per questo ho modellato il problema più volte.

La funzione da ottimizzare si chiama ***model*** e crea un file testuale con i valori dei parametri passati in input in modo che Modelica possa leggerlo (in particolare, in modo che *System* possa leggerlo). A quel punto esegue la simulazione e infine ritorna il valore delle variabili del *monitor* che ci interessano in output.

3.3.1 NGOpt

La documentazione di *Nevergrad* suggerisce di iniziare con l'ottimizzatore *NGOpt*.

NGOpt sta per *Nevergrad Optimizer* ed è un *wizard* che seleziona l'algoritmo migliore al posto nostro, dato quello che il programma sa della funzione da ottimizzare.
Tiene conto di:

- Numero di variabili da ottimizzare
- Tipo di variabili da ottimizzare

- Numero di trial (*budget*)
- Possibilità di parallelizzare (*n_worker*)

Nel primo test ho chiesto all'ottimizzatore di scegliere il numero di droni in modo massimizzare la media di droni per area (*AvgDrones*). Essendo un test ho fatto pochi trial (10) e ho limitato il numero di droni in [1, 8].

parameter	**suggested value**	**gain (AvgDrones)**
n_drones	6	0.955

Riprovando lo stesso test ma con *50 trial*.

parameter	**suggested value**	**gain (AvgDrones)**
n_drones	4	0.79

Aggiungendo un po' di verbosity alla funzione possiamo notare come l'algoritmo scelto da *NGOpt* faccia diversi trial con valori che ha già provato prima.

n° trial	**n_drones**	**gain (AvgDrones)**
1	4	0.79
2	3	0.5775
3	6	0.955
4	5	0.8825
5	7	0.9775
6	4	0.79
7	5	0.8825
8	6	0.955
9	3	0.5775
10	4	0.79

Questo perché *Nevergrad* non è fatto per scegliere parametri discreti con spazi piccoli, in quanto non garantisce l'ottimalità. In questi casi, in cui c'è un numero ridotto di opzioni, si fa prima a provarle tutte e ritornare quella migliore senza scomodare *Nevergrad*.

Per questo motivo ho provato di nuovo ma con un altro parametro, questa volta con dominio Reale: ***cc_choice***
Ho esteso il *budget* di *Nevergrad* a *100*, che è un budget comunque molto basso.

parameter	suggested value	gain (AvgDrones)
cc_choice	0.6018698230890397	0.575

Ho poi ridotto l'orizzonte di simulazione a *500* ma esteso il budget a *1000*. Perché lasciando l'orizzonte di simulazione a *1000*, ogni *trial* impiega circa *30* secondi, questo moltiplicato per *1000 trial* avrebbe fatto sì che il test completo impiegasse *16 ore*.

parameter	suggested value	gain (AvgDrones)
cc_choice	0.05847010233963334	0.58

Sfortunatamente la mia macchina non è fra le più performanti e la necessità di collegarsi al un server per accedere al *DB* rende la parallelizzazione pressoché impossibile senza riscrivere l'intero sistema (server compreso) da capo, cosa che non dovremmo poter essere in grado di fare, per il paradigma della *Black-Box Optimization*. Non possiamo mettere mano sul sistema.

Senza alterare drasticamente il sistema sono riuscito però ad ottimizzarlo, riducendo il tempo di ogni *trial* a *10* secondi, in questo modo il tempo totale del test con budget *1000* è di sole *3 ore*. Inoltre ho chiesto a *NGOpt* di ottimizzare non uno ma ben due parametri, ***n_drones*** e ***cc_choice***.

parameter	suggested value	gain (AvgDrones)
n_drones	3	0.5775
cc_choice	0.04014053335382976	

NGOpt suggerisce quindi di dare molto più peso all'area lasciata scoperta più a lungo (**scelta 2**), piuttosto che all'area più vicina al drone (**scelta 1**).

3.3.2 SHIWA

Come ho già detto, *Nevergrad* è dotato di numerosi ottimizzatori, studiarli tutti richiederebbe molto tempo, esiste forse un modo più efficiente per decidere?

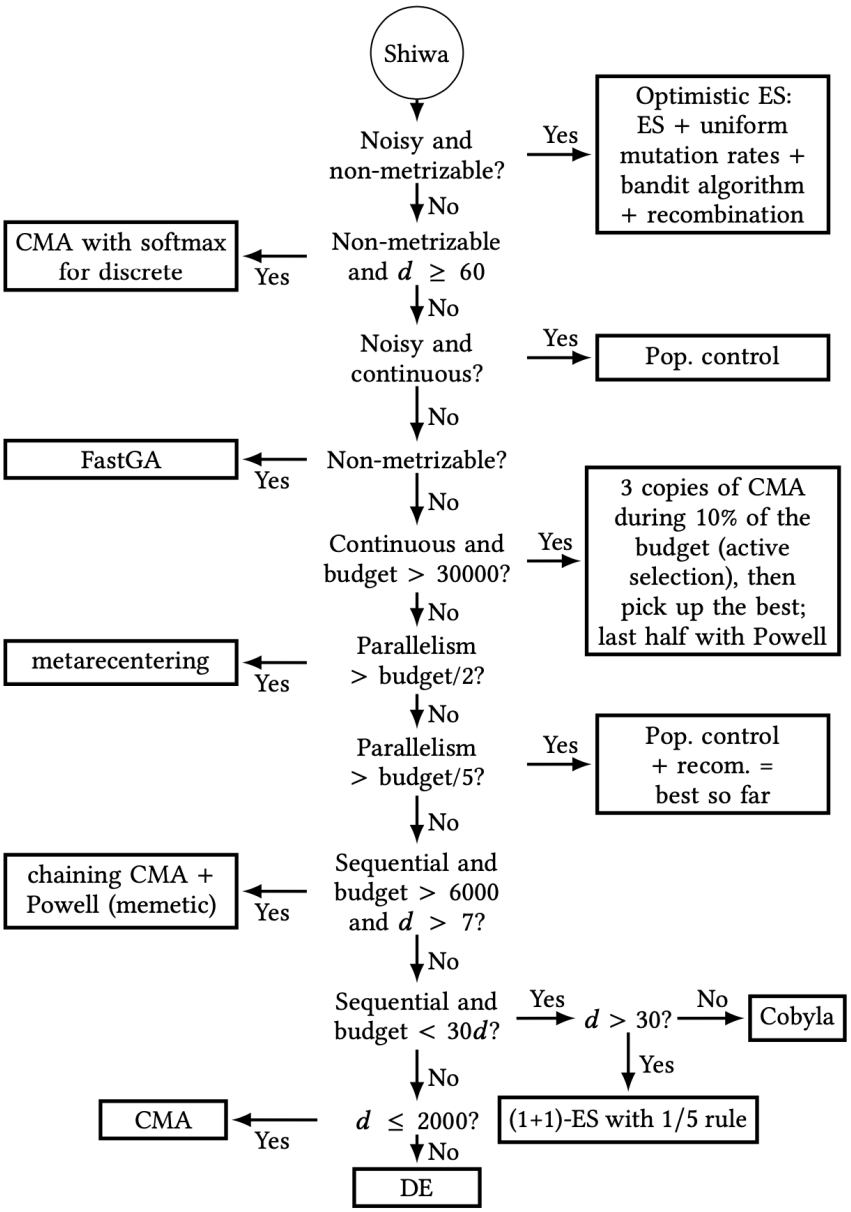
```
# Choose your optimization method below.
optimizer_name = "NGOpt"
optimizer_name = "DoubleFastGADiscreteOnePlusOne"
optimizer_name = "OnePlusOne"
optimizer_name = "DE"
optimizer_name = "RandomSearch"
optimizer_name = "TwoPointsDE"
optimizer_name = "Powell"
optimizer_name = "MetaModel"
optimizer_name = "SQP"
optimizer_name = "Cobyla"
optimizer_name = "NaiveTBPSA"
optimizer_name = "DiscreteOnePlusOne"
optimizer_name = "cGA"
optimizer_name = "ScrHammersleySearch"
```

Soltanto alcuni degli ottimizzatori di Nevergrad

Ideato dai ricercatori Antonie Moreau, Mike Preuss e Jialin Liu nel paper *Versatile Black-Box Optimization*, l'algoritmo ***SHIWA*** ha lo scopo di amalgamare più ottimizzatori insieme e aiutarci a prendere la scelta migliore per ottimizzare il nostro sistema.

SHIWA chiede di sapere della funzione da ottimizzare:

- Il tipo di iperparametri (discreti/continui)
- Il numero di iperparametri
- Se c'è rumore
- Il numero di trial da poter effettuare
- Il livello di parallelismo (quanti trial posso eseguire in contemporanea)



Seguendo l'albero di decisione viene fuori che gli algoritmi migliori da utilizzare nel nostro caso specifico sono ***Cobyla*** (con poco budget) oppure ***CMA + Powell***. La concatenazione "+" suggerisce di dividere il budget fra i due algoritmi e dare al secondo l'output del primo.

Non avendo a disposizione una macchina particolarmente performante ho scelto *Cobyla*.

3.3.3 Cobyla

Cobyla sta per *Constrained Optimization BY Linear Approximation*.
É un algoritmo *model-based* (capitolo 1.3) che approssima iterativamente il problema utilizzando tecniche di *linear programming* (LP) e alla fine di ogni *trial* estrae la prossima configurazione da provare, basandosi sulle approssimazioni fatte fino a quel punto.

Ad ogni passo *Cobyla* pesca la configurazione che (secondo il modello) si avvicina di più all'obiettivo prefissato (*massimizzare* o *minimizzare* l'output della funzione), rispettando però i vincoli di *upper bound* e *lower bound* imposti sui parametri.

Quando l'algoritmo pensa di non poter fare più miglioramenti significativi, riduce le dimensioni dei "salti", aumentando la **granulosità** della ricerca. Quando questa granulosità diventa sufficientemente fine (o finisce il *budget*), l'algoritmo termina.
Questo algoritmo è stato ideato da Michael J. D. Powell e distribuito sotto la *GNU Lesser General Public Licence* (LGPL).

Come al solito per iniziare ho deciso di fare un test con pochissimo budget (10), questa volta però, invece di utilizzare come funzione obiettivo il numero atteso di droni (*AvgDrones*), ho utilizzato il numero massimo di tempo (fra tutte le aree) in cui un'area è stata lasciata scoperta (***MaxNoDrone***). L'iperparametro da ottimizzare è sempre ***cc_choice***.
Come potevamo aspettarci, *Cobyla* suggerisce di dare molto più peso all'area lasciata scoperta più a lungo (**scelta 2**).

parameter	**suggested value**	**loss (MaxNoDrone)**
cc_choice	0.03162277877171417	15

Dopo di che sono passato ad un *budget* di *1000*:

parameter	**suggested value**	**loss (MaxNoDrone)**
cc_choice	0.0316227766016838	15

Si può notare come il loss sia identico, questo perché i parametri suggeriti sono molto "simili" e una volta passati al nostro sistema diventano uguali dato che Open*Modelica* approssima i numeri reali fino alla quinta cifra decimale e quindi, non essendoci *noise*, dato lo stesso input ritorna lo stesso output.

Infine ho provato ad ottimizzare più parametri, sempre con budget di *1000*.

parameter	**suggested value**	**loss (AvgNoDrone)**
cc_choice	0.029971976529311172	0.625
Tcc	2.0564670136484198	

Cobyla suggerisce quindi di dare più peso alla scelta 2 e di aumentare il tempo di aggiornamento del *CC* risultando quindi in meno richieste inviate.

4. NOMAD

4.1 Introduzione

NOMAD è un implementazione in C/*C++/Python* degli algoritmi di *Mash Adaptive Direct Search* (MADS), ovvero algoritmi ideati per l'ottimizzazione Black-Box di funzioni particolarmente costose con possibilità di rumore.
Il nome *NOMAD* infatti è l'acronimo di *Nonlinear Optimization by Mesh Adaptive Direct Search*. Esistono diverse versioni di NOMAD, in questa relazione utilizzerò NOMAD 4. La più recente.

4.2 Algoritmi MADS

NOMAD risolve problemi del tipo

$$x_{final} = min\, f(x)$$

Ovvero trovare l'input per il quale la funzione f ritorna il valore più piccolo in $Im(f)$.
Come fare per trovare il massimo invece? basta invertire il segno del valore in output, analogamente a come abbiamo già visto con *Nevergrad*.

Gli algoritmi *MADS* iterano su una serie di *Mesh* di diverse dimensioni. Un mesh è una discretizzazione dello spazio delle configurazioni, ovvero dei domini degli iperparametri.
Similmente ad un *Grid Search* ma in questo caso vi è anche una **ricerca adattiva** che varia la granulosità dei mesh, come per *Cobyla*.
Lo scopo di ogni iterazione è trovare un *trial* con configurazione *x* che generi un punteggio migliore di $\boldsymbol{f}^+$. Se il trial fallisce, si aumenta la granulosità del mesh.

Ogni iterazione è composta da due step:

I. *Search*
II. *Poll*

Come suggerisce il nome, la fase di *Search*, si occupa di cercare il prossimo *x* da provare, nello spazio delle configurazioni. Questa è la fase più *malleabile* in quanto è dettata dalle euristiche. Abbiamo infatti diversi modi di fare *Search*. (**exploration**)

La fase di *Poll* invece cerca la prossima configurazione da provare nell'intorno di $\boldsymbol{x}^+$. (**exploitation**)

Sia x_0 una possibile configurazione iniziale (degli iperparametri) e sia x_k la configurazione alla *k*-esima iterazione.

```
while ( criterio di fermata non soddisfatto ) loop
     SEARCH sul mesh per trovare una soluzione migliore di x_k
     if ( SEARCH ha fallito ) then
          POLL sul mesh per trovare una soluzione migliore di x_k
```

```
if ( SEARCH or POLL hanno avuto successo )
      nomina il trial x_{k+1} e diminuire la granulosità
else
      imposta x_{k+1} = x_k e aumenta la granulosità
Aggiorna i parametri e k = k+1
```

4.3 Modellazione

Per prima occorre la funzione ***obj*** da ottimizzare. Avendo già collegato il sistema a *Python* per *Nevergrad*, per *NOMAD* ho utilizzato lo stesso linguaggio.
A differenza di *Nevergrad*, su *NOMAD* la funzione da ottimizzare deve prendere in input un oggetto di *PyNomad* e deve dare in output una stringa del formato corretto.

In compenso però NOMAD permette di definire ulteriori vincoli sulla funzione oltre a quelli di *upper bound* e *lower bound*. Questi vincoli devono essere valutati all'interno della funzione e scritti nella stringa dell'output. Se il vincolo ha un valore > *0* allora è considerato violato.

Per sfruttare al meglio *NOMAD* ho deciso di scrivere un vincolo c_i per ogni variabile nell'output del *monitor*. I vincoli sono:

- $\boldsymbol{c_1}$*: AvgDrones* deve essere maggiore di 0,5
- $\boldsymbol{c_2}$*: AvgNoDrone* deve essere minore di 0,5
- $\boldsymbol{c_3}$*: MaxNoDrone* deve essere minore di 20
- $\boldsymbol{c_4}$*: PrDrones* deve essere maggiore di 0,5
- $\boldsymbol{c_5}$*: StdDev* minore di 0,5

```
def model(x):
    # i parametri si trovano nell'oggetto x
    dim = x.size()
    # estraggo i parametri nell'array f
    f = [x.get_coord(i)**2 for i in range(dim)]
    # estraggo i parametri
    cc_choice = f[0]
    Tcc = f[1]

    # eseguo il modello
    with open ("modelica_parameters.in", 'wt') as f:
    f.write("p.cc_choice="+str(cc_choice)+"\n"
        +"p.Tcc="+str(Tcc)+"\n")
    f.flush()
    os.fsync(f)

    # execute simulation
    os.system("./System -overrideFile=modelica_parameters.in")
```

```
    # leggi i valori dell'output
    with open ("outputs.txt", 'r') as f:
      i = 1
      w = 0
      for line in f:
           if i == 2:
                # spacchetta la riga
                tokens = line.split()
                # prendi i valori
                AvgDrones = float(tokens[0])
                AvgNoDrone = float(tokens[1])
                MaxNoDrone = float(tokens[2])
                PrDrones = float(tokens[3])
                StdDev = float(tokens[4])
           else:
                i += 1
      f.close()

    # calcola risultato finale e constraint
    obj = -AvgDrones # funzione obiettivo
    c1 = 0.5 - AvgDrones
    c2 = AvgNoDrone - 5.0
    c3 = MaxNoDrone - 20
    c4 = 0.5 - PrDrones
    c5 = StdDev - 0.5

    #unisci i valori nella stringa in output
    f = str(obj) + " " + str(c1)
        + " " + str(c2) + " " + str(c3)
        + " " + str(c4) + " " + str(c5)

    # formatta output
    x.setBBO(str(f).encode("UTF-8"))
    # fine
    return 1 # successo
```

La parte di *NOMAD* che si occupa dell'ottimizzazione invece è questa:

```
# cc_choice, Tcc
x0 = [0.5, 1.5] # condizione iniziale
lb = [0.0, 1.0] # lower-bound
ub = [1.0, 3.0] # upper bound

# PB := progressive barrier
budget = 10000;
params = ["BB_OUTPUT_TYPE OBJ PB PB PB PB PB", # formato output
        "MAX_BB_EVAL "+str(budget), # numero di trial
```

```
        "DISPLAY_DEGREE 2",
        "DISPLAY_ALL_EVAL false", # se mostrare ogni trial
        "DISPLAY_STATS BBE OBJ"]

# prendi suggerimenti
result = PyNomad.optimize(model, x0, lb, ub, params)

# stampa i risultati
fmt = ["{} = {}".format(n,v) for (n,v) in result.items()]
output = "\n".join(fmt)
print("\nNOMAD results \n" + output + " \n")
```

NOMAD cerca la configurazione migliore che rispetti tutti i vincoli ma comunica nell'output anche la configurazione migliore che però non ha rispettato i vincoli, e di quanto.

Si può notare che ho definito il formato di output come *OBJ PB PB PB PB PB* ovvero che la funzione darà in output un valore da minimizzare accompagnato da *5 constraint* (1 per ogni variabile di output) da dover rispettare. Occorre specificare a priori quanti saranno i constraint in modo che NOMAD sappia come *leggere* la stringa data in output dal modello.

I vincoli scelti sono di tipo *Progressive Barrier* (*PB*), l'alternativa sarebbero vincoli di tipo *Extreme Barrier* (*EB*). La differenza fra vincoli *PB* e *EB* è che i primi permettono una violazione graduale (e quindi valutata) del vincolo. I secondi non tollerano alcuna violazione del vincolo, non considerando nessuna configurazione che violi (anche se di poco) il vincolo.

4.4 Algoritmo

Per iniziare ho eseguito pochi test (10) su un solo parametro e senza vincoli, ovvero $\forall i\, c_i = 0$ fisso.

parameter	**suggested value**	**gain (AvgDrones)**
cc_choice	0.426	0.68

Una volta verificato che lo script funzioni, ho aumentato il numero di test da effettuare a 1000 e ho provato ad ottimizzare i due parametri utilizzando invece che un'unica funzione di loss, un constraint per ogni valore dato in output dal modello.
Come funzione obiettivo *obj* invece ho utilizzato ***AvgNoDrone***. I due suggerimenti sono stati:

Type	**Tcc**	**cc_choice**	**obj**	**h**
Feasible	1.49211	0.499958	3.125	0.0
Infeasible	1.09195	0.1135	1.875	0.00001444

La *h* è la stima di *NOMAD* di quanto la connfigurazione ha violato i vincoli imposti.

5. Confronto

5.1 Facilità di utilizzo

5.1.1 Nevergrad

Nevergrad è molto facile da installare ed utilizzare, senza contare che i numerosi esempi forniscono un ottimo framework da cui poter espandere ed apprendere gradualmente.
La comunità online di *Nevergrad* è disponibile al dialogo e si trovano innumerevoli esempi, discussioni, paper, etc... che aiutano non poco ad imparare e mi hanno tenuto per mano nel corso di questa relazione.

In particolare gli esempi sono stati pensati apposta per poter essere letti da qualcuno che si sta cimentando per la prima volta in questo campo. Sono pieni di commenti, istruzione per istruzione, che spiegano passo passo lo scopo di ogni azione.

```
# Library and environment for Reticulate/Nevergrad.
library("reticulate")
conda_create("r-reticulate")
conda_install("r-reticulate", "nevergrad", pip=TRUE)
use_condaenv("r-reticulate")

# Only if you use parallelism.
library(doParallel)

# Choose your optimization method below.
optimizer_name <- "NGOpt"
# optimizer_name <- "DoubleFastGADiscreteOnePlusOne"
# optimizer_name <- "OnePlusOne"
# optimizer_name <- "DE"
# optimizer_name <- "RandomSearch"
# optimizer_name <- "TwoPointsDE"
# optimizer_name <- "Powell"
# optimizer_name <- "MetaModel"  CRASH !!!!
# optimizer_name <- "SQP"
# optimizer_name <- "Cobyla"
# optimizer_name <- "NaiveTBPSA"
# optimizer_name <- "DiscreteOnePlusOne"
# optimizer_name <- "cGA"
# optimizer_name <- "ScrHammersleySearch"

# Now we can play with Nevergrad as usual.
# We assume here that we have 17 continuous hyperparameters with values in [0, 1].
# We can do other instrumentations, as discussed below.
my_tuple <- tuple(17)
instrumentation <- ng$p$Array(shape=my_tuple)
instrumentation$set_bounds(0., 1.)
num_workers <- 3  # We want to be able to evaluate 3 hyperparametrizations simultaneously.
num_iterations <- 100 * num_workers  # Let us say we have a budget of 100xnum_workers hyperpar

# Let us create a Nevergrad optimization method.
optimizer <-  ng$optimizers$registry[optimizer_name](instrumentation, budget=num_iterations, n

# Dummy initializations.
nevergrad_hp <- 0
nevergrad_hp_val <- 0
score <- 0

for (i in 1:num_iterations) {
    for (j in 1:num_workers) {
       nevergrad_hp[j] <- optimizer$ask()
       nevergrad_hp_val[j] <- nevergrad_hp[j]$value
    }

    # Sequential version.
    # for (j in 1:num_workers) {  # In a perfect world this would be parallel.
    #    score[j] <- norm(nevergrad_hp_val[j]
    # }

    # Parallel version.
    # Actually this could be asynchronous, Nevergrad is ok for that, you do not have to
    # do the tell's in the same order as the ask's.
    registerDoParallel(cores=num_workers)
    getDoParWorkers()
    foreach(i=1:num_workers) %dopar% score[j] <- norm(nevergrad_hp_val[j])

    for (j in 1:num_workers) {
       optimizer$tell(nevergrad_hp[j], score[j])
    }
}
print(optimizer$recommend()$value)
```

Senza contare che *Nevergrad* offre un ottimo wizard (*NGOpt*), questo è un vantaggio non indifferente che permette a chiunque di ottimizzare la propria funzione senza doversi studiare tutti gli algoritmi per capire quale sia il migliore.

L'interfaccia unificata (*ask* e *tell*) fra gli algoritmi fa anche sì che sia facile cambiare algoritmo per un nuovo test, senza dover riscrivere tutto.

Risulta però difficile fare ottimizzazioni più *sofisticate* di *minimize* o *maximize*, ad esempio specificando diversi vincoli da dover rispettare. In quei casi infatti occorre inventarsi degli stratagemmi come ad esempio degli if interni alla funzione che controllano i vincoli e se un dato vincolo non è rispettato fanno *schizzare* la *loss* alle stelle.

Infine se una funzione ritorna più parametri da dover ottimizzare è necessario dare in output una combinazione lineare di questi, in quanto *Nevergrad* tollera soltanto un singolo *float* come funzione di loss

5.1.2 NOMAD

NOMAD non è semplice da installare. richiede numerosi passaggi soltanto per poter compilare ed eseguire il primo test. La documentazione disponibile è a dir poco scarna e quel poco che si riesce a trovare non è stato pensato per essere di facile comprensione a qualcuno che si stia cimentando in questo campo per la prima volta.

Dal sito ufficiale è possibile scaricare diversi esempi ma anche questi sono estremamente difficili da comprendere in quanto privi di commenti.
Come si può vedere sotto, nell'esempio proposto non risulta chiaro quali siano i vincoli, quali siano gli iperparametri, quale sia la funzione da ottimizzare, quale sia il budget, in che modo i vengano eseguiti i trial, etc...

```
#include <cmath>
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main ( int argc , char ** argv ) {

double f = 1e20, c1 = 1e20 , c2 = 1e20;
double x[5];

if ( argc >= 2 ) {
    c1 = 0.0 , c2 = 0.0;
    ifstream in ( argv[1] );
    for ( int i = 0 ; i < 5 ; i++ ) {
        in >> x[i];
        c1 += pow ( x[i]-1 , 2 );
        c2 += pow ( x[i]+1 , 2 );
    }
    f = x[4];
    if ( in.fail() )
        f = c1 = c2 = 1e20;
    else {
        c1 = c1 - 25;
        c2 = 25 - c2;
    }
    in.close();
}
cout << f << " " << c1 << " " << c2 << endl;
return 0;
}
```

Inoltre non aiuta il fatto che questo tool non sia utilizzato tanto quando *Nevergrad* e quindi che la comunità online sia pressoché inesistente.

Però, una volta capito come funziona e dopo essersi fatti degli script framework propri da compilare all'occasione, risulta molto veloce e facile da utilizzare.

Infine, il fatto che NOMAD abbia un unico algoritmo estremamente veloce fa sì che per alcuni problemi sia la soluzione perfetta.

5.2 Risultati migliori

La configurazione che ha portato ad avere risultati migliori è stata quella suggerita da NOMAD, come si può vedere dalla tabella.

I parametri suggeriti da NOMAD hanno una media di droni per area maggiore, tempo massimo di attesa ridotto, probabilità di trovare un drone più alta e deviazione standard (del numero di droni per area) minore!

É sicuramente interessante come *Nevergrad* abbia scelto di aumentare *Tcc* del 30% rispetto al parametro iniziale, ma diminuire drasticamente (del 96%) il parametro di scelta, mentre *NOMAD* abbia proposto di aumentare *Tcc* del 30% e ridurre *cc_choice* dell'80%

Opt	**Tcc**	**cc choice**	**Avg Drones**	**Avg No Drone**	**Max No Drone**	**Pr Drones**	**Std Dev**
Nevergrad	2.0564	0.0299	0.545	0.625	20	0.53	0.49957
NOMAD	1.0919	0.1135	0.6075	1.875	15	0.59	0.48875

NOTA: La soluzione proposta di *NOMAD* è la migliore trovata senza tenere conto dei vincoli (*best infeasible).*

5.3 Conclusioni

L'approfondimento di questo settore dell'informatica ha avuto i suoi alti e i suoi bassi. Essendo un'area di nicchia, è stato molto difficile trovare materiale di supporto e consigli sulle *best practices* da utilizzare.

Mi sento di consigliare *Nevergrad* ai principianti e a chiunque voglia anche solo *dare un'occhiata* a questo modo in quanto è facile da utilizzare e la comunità online risulta molto amichevole. La sua natura *user-friendly* però, come spesso accade, può risultare troppo rigida e difficile da adattare a problemi di natura più complessa, diversi dalla semplice minimizzazione della funzione di *loss*.

Per questo motivo credo che sia davvero un peccato che *NOMAD* abbia una curva di apprendimento così ripida, perché da un punto di vista della modellazione del problema, è quello che offre molta più libertà.

Chiaramente c'è molto di più da dire sia sulla BBO in generale, che su *Nevergrad* e *NOMAD* nello specifico. Questo documento (insieme al codice sorgente ricco di commenti) vuole essere un'introduzione di più facile lettura possibile, in modo che qualcuno il lettore che vorrà cimentarsi in questo settore potrà farlo con una visione generale più chiara e (si spera) senza commettere tutti gli errori che ho commesso anche io nel corso dello sviluppo e della stesura di questo progetto. :)

Andrea Di Marco, 2022

6. Ringraziamenti

Ringrazio di cuore il Dott. Marco Esposito, il Prof. Toni Mancini e il Prof. Enrico Tronci per avermi seguito e sopportato nel corso della stesura di questo documento, cosa che non sarei mai riuscito a fare da solo e senza la giusta motivazione.

7. Note

Tutti i test sono stati eseguiti su una macchina virtuale Parallels (versione 17.1.4) con Ubuntu sopra un MacBook Pro (2018) con processore 2,2 GHz Intel Core i7 6 core e memoria 16GB 2400 MHz DDR4.

8. Bibliografia

Nevergrad Documentation by Meta
NOMAD Documentation by GERAD
Versatile Black-Box Optimization by Antonie Moreau, Mike Preuss e Jialin Liu
A Modern Guide to Hyperparameter Optimization by Richard Liaw
Bayesian Optimization by Matthew W. Hoffman from UAI 2018
Gaussian Processes in Machine Learning by Carl Edward Rasmussen
Understanding Quantiles by ThoughtCo.
Bandit Algorithms from ICTP Quantitative Life Sciences
Bandit Algorithms book by Tor Lattimore and Csaba Szepesvári
Provably efficient online hyperparameter optimization with population-based bandits by Jack Parker-Holder, Vu Nguyen, Stephen J. Roberts