Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Magistrale in
Scienze Statistiche

Aspetti metodologici e computazionali dell'uso della differenziazione automatica per la costruzione di modelli surrogato

Relatore: Prof.ssa Alessandra R. Brazzale
Dipartimento di Scienze Statistiche

Laureando Michele Palese
Matricola N. 2044695

Anno Accademico 2023/2024

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Lo sviluppo delle tecnologie computazionali degli ultimi decenni ha coinvolto i processi di ricerca scientifica, promuovendo l'aggiornamento di procedure e metodi. La simulazione di dati artificiali provenienti da fenomeni complessi è oggi uno strumento essenziale e largamente utilizzato in ingegneria aerospaziale, bioinformatica, fisica delle particelle, finanza.

Da un punto di vista statistico, le procedure di inferenza classiche basate sulla verosimiglianza non sono direttamente utilizzabili per fenomeni molto complessi. Obiettivo di questa tesi è esplorare la costruzione di procedure inferenziali alternative che consentano di fare affermazioni sul fenomeno di interesse a seguito dell'osservazione delle realizzazioni sperimentali. L'attenzione sarà posta su come tali procedure richiedano l'impiego intensivo di metodi computazionali e di come la scelta dei metodi di calcolo determini il successo di tali procedure.

A questo riguardo saranno discussi i vantaggi di cui beneficiano le fasi di ottimizzazione nella costruzione di modelli surrogato apportati dallo sviluppo della differenziazione automatica. L'elaborato è strutturato come segue: Nel Capitolo 1 verranno illustrati i principali algoritmi di ottimizzazione e dei metodi di calcolo delle derivate. Nel Capitolo 2 verranno definiti i simulatori ed il concetto di verosimiglianza intrattabile, discutendo di come un simulatore definisca implicitamente un modello statistico. Nel Capitolo 3 è presente una rassegna delle soluzione bayesiane ai problemi di inferenza senza verosimiglianza, mentre il Capitolo 4 si occupa della costruzione di modelli surrogato. Il Capitolo 5 infine, raccoglie i risultati di alcuni tentativi di implementazione ed uno studio di simulazione nel caso univariato. Nelle Conclusioni invece si presenterà una valutazione dell'efficacia dei metodi e spunti per approfondimenti futuri.

Capitolo 1

La differenziazione automatica

1.1 Algoritmi di ottimizzazione

1.1.1 Principi generali

L'inferenza statistica si occupa di produrre affermazioni rigorose sulle caratteristiche di interesse di una popolazione avendone a disposizione solo un insieme limitato di osservazioni. I dati osservati y_{oss} sono considerati la realizzazione di una variabile aleatoria Y di cui ricostruire, anche parzialmente, la legge di probabilità. Il processo induttivo appena descritto comincia con la specificazione del modello statistico che può essere parametrico, semiparametrico o non parametrico a seconda di quanto si desidera restringere le possibili forme che la legge generatrice dei dati può assumere. In un contesto parametrico il processo generatore dei dati $p_0(y, \theta)$ è univocamente determinato dal parametro θ e per il modello statistico $\mathcal{F}$ si può scrivere $\mathcal{F} = \{p(y; \theta), y \in \mathcal{Y}, \theta \in \Theta\}$, dove $\mathcal{Y}$ indica lo spazio campionario ovvero le possibili configurazioni di y, mentre Θ l'insieme dei valori ammissibili di θ. La ricerca dei valori θ in maggior accordo con i dati osservati costituisce la fase di stima del modello e coincide con l'ottimizzazione di una data funzione obiettivo che in casi ordinari è la funzione di log-verosimiglianza $f(\theta) = \sum_i \log p(y_i, \theta)$. Quando il modello statistico è ben specificato la distribuzione generatrice $p_0(y; \theta)$ appartiene all'insieme di ricerca $\mathcal{F}$ ed asintoticamente l'ottimo trovato $\hat{\theta} = \arg\max_{\theta \in \Theta} \sum_i \log p(y_i, \theta)$ coincide con il vero valore generatore θ_0. Per semplicità nel corso di questo capitolo il modello sarà ritenuto sempre ben specificato.

La soluzione dei problemi di ottimo che si incontrano in statistica è il più delle volte non esplicita e va ricercata iterativamente. Gli algoritmi di ottimizzazione esplorano il dominio della funzione obiettivo $f(.)$, lo spazio parametrico Θ, producendo una successione di punti $\{\theta_r\}_{r \in \mathbb{N}}$ convergente a $\hat{\theta}$. La successione è realizzata alternando un

passo di ricerca di un punto ottimo candidato θ_r ad un passo di valutazione di $f(\theta_r)$, ovvero alla verifica del raggiungimento di una delle due condizioni di arresto:

- differenza del valore della funzione fra due iterazioni successive inferiore ad una certa soglia $\epsilon \in \mathbb{R}^+$: $|f(\theta_r) - f(\theta_{r-1})| \leq \epsilon$ con $\epsilon \approx 0$
- Esecuzione di un numero limite di aggiornamenti di θ^r.

Per il passo di ricerca del punto candidato occorre definire una regola di aggiornamento che si compone di:

- un metodo di calcolo per la direzione d_r lungo cui muovere θ_r,
- un tasso di aggiornamento η_r ovvero di quanto muovere θ_r.

In generale la regola di aggiornamento ha la forma $\theta_{r+1} = \theta_r - \eta d_r$ con $d_r = (B_r)^{-1}\nabla f(\theta_r)$ e dove $(B_r)^{-1}$ e $\nabla f(\theta_r)$ sono una matrice ed il gradiente di f valutati all'iterazione corrente r. La scelta di (B_r) distingue 3 diversi metodi di ricerca del minimo (Nocedal et al. 2006).

1. Se $B_r = I$, si parla di metodi di massima discesa.
2. Se $B_r = H_r$, dove H_r è la matrice *Hessiana* all'iterazione corrente, siamo nel campo dei metodi di Newton.
3. Infine se $B_r \approx H_r$, si rientra nei metodi *quasi-Newton* fra cui ritroviamo i noti algoritmi *DFP (Davidson, Fletcher, Powell)* e *BFGS (Broyden, Fletcher, Goldfarb, Shanno)* per ulteriori dettagli si veda Nocedal et al. (2006).

1.1.2 Metodi di Newton e quasi Newton

L'utilizzo della matrice Hessiana che fanno i metodi di *Newton* e *quasi Newton* trova fondamento teorico nella validità dell'approssimazione lineare ottenuta dalla formula di Taylor. Siano $\theta \in \Theta = \mathbb{R}^n$ e $\xi \in \mathbb{R}^n$, $\delta \in \mathbb{Z}^l$ un multiindice con $\delta_i > 0$ per $0 < i \leq n$. Un multiindice è una tupla collezione ordinata di elementi di numeri per cui vale la seguente notazione:

- Modulo del multiindice $|\delta| = \delta_1 + \delta_2 + \cdots + \delta_n$
- Potenza del multiindice per una variabile θ: $\theta^\delta = \theta_1^{\delta_1}, \theta_2^{\delta_2}, \ldots \theta_n^{\delta_n}$
- Derivata parziale di ordine δ: $D^{|\delta|} = \frac{\partial f^{|\delta|}}{\partial \delta_1^{\delta_1}, \ldots \partial \theta_n^{\delta_n}}$.

La risultante approssimazione di Taylor della funzione obiettivo valutata nel punto $\theta_r + \xi$ arrestata al secondo ordine di derivazione è:

$$f(\theta_r + \xi) = \sum_{|\delta| \geq 0}^{|\delta| \leq 2} \frac{D^{|\delta|} f}{\partial \theta_1^{\delta_1} ... \partial \theta_n^{\delta_n}} = f(\theta^r) + \xi^t \nabla f(\theta^r) + \frac{1}{2} \xi^t H(\theta_r) \xi.$$

Fissando θ^r l'approssimazione è funzione solo di ξ e la direzione ottima lungo cui cercare il minimo si trova in corrispondenza del valore che annulla la derivata di $ni(\xi)$

$$\ni (\xi) = f(\theta) + \xi^t \nabla f(\theta) + \frac{1}{2} \xi^t H(\theta) \xi \approx f(\theta_r + \xi)$$

Risolvendo si ottiene che la direzione ottima all'iterazione corrente r è $d_r^N = -H(\theta_r)^{-1} \nabla f(\theta_r)$, detta direzione di Newton. Si deve sottolineare che affinché d_r^N sia effettivamente la direzione ottima la matrice hessiana $H(\theta_r)$ deve essere definita positiva.

Nei metodi di *quasi Newton* la matrice hessiana esatta è sostituita da una sua approssimazione ottenuta applicando la formula di Taylor alla funzione gradiente $\nabla f(\theta)$

$$\begin{aligned} \nabla f(\theta_r + \xi) &= \nabla f(\theta_r) + \int_0^1 [H(\theta_r + t\xi)] \xi \, dt \\ &= \nabla f(\theta_r) + H_f \xi + \int_0^1 [H(\theta_r + t\xi) - H(\theta_r)] \xi \, dt \\ &= \nabla f(\theta_r) + H\xi + o(||\xi||). \end{aligned}$$

Il termine integrale cattura la variazione del gradiente nel tratto che porta da θ_r a $\theta_r + \xi$ ed è interpretabile come il grado di curvatura della funzione f nell'intervallo in esame, mentre il termine $o(||\xi||)$ indica la trascurabilità di tale integrale al decrescere della norma di ξ. Considerando la natura iterativa del processo sia $\xi = \theta_{r+1} - \theta_r$ la lunghezza del passo, l'approssimazione diventa:

$$\nabla f(\theta_{r+1}) = \nabla f(\theta_r) + H(\theta_r)(\theta_{r+1} - \theta_r) + o(||\theta_{r+1} - \theta_r||).$$

Quando H è definita positiva vale l'approssimazione $H(\theta_r)(\theta_{r+1} - \theta_r) \approx \nabla f(\theta_{r+1}) - \nabla f(\theta_r)$ utilizzata nei metodi di *quasi Newton* in sostituzione alla matrice Hessiana esatta. L'approssimazione viene calcolata risolvendo l'equazione $(H_{r+1})^{-1}(\theta_{r+1} - \theta_r) = B_{r+1}(\theta_{r+1} - \theta_r) = \nabla f(\theta_{r+1}) - \nabla f(\theta_r)$. Ci sono vari modi di aggiornare B_{r+1} ad ogni iterazione che caratterizzano i diversi algoritmi come i già citati *BGFS* e *DPF*, per maggiori dettagli si veda Dai (2002).

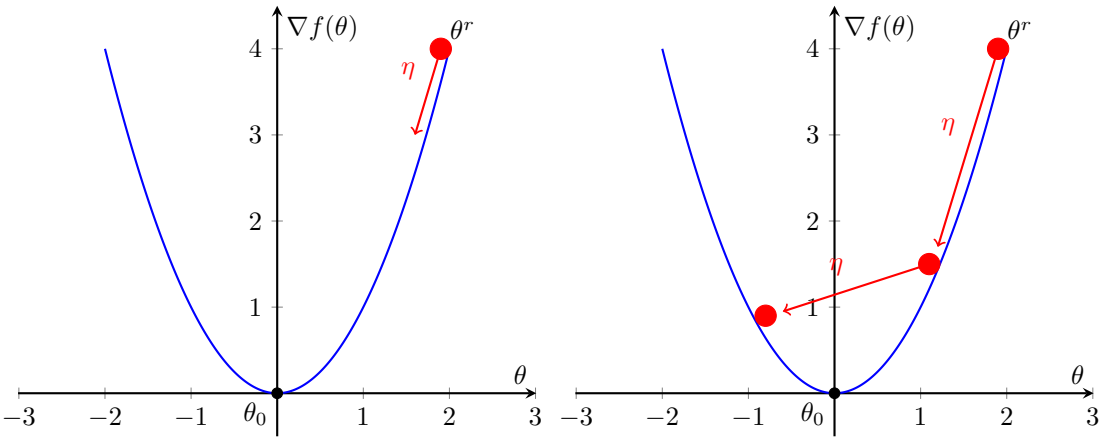

Figura 1.1: Discesa del gradiente per $f(\theta) = \frac{\theta^3}{3}$ con $\frac{\partial f(\theta)}{\partial \theta} = \theta^2$. A destra le conseguenze di un passo η troppo ampio: il punto di minimo viene saltato e l'algoritmo è costretto a rimbalzare da una parte all'altra della funzione.

Trovata la direzione ottimale lungo cui muovere θ_r rimane da determinare il tasso di aggiornamento. Fissato il punto θ_r e la direzione d_r all'iterata corrente la scelta ideale sarebbe quella che minimizza la funzione univariata $\phi(\eta) = f(\theta_r + \eta d_r)$. Tuttavia per la selezione di η si fa affidamento a metodi euristici poiché procedure esatte richiederebbero la risoluzione di un nuovo problema di minimo e molte nuove valutazioni della funzione $f(\theta)$, per ulteriori dettagli si rimanda a Nocedal et al. 2006.

1.1.3 Metodi di massima discesa

I metodi di massima discesa o di discesa del gradiente sono più semplici da implementare e più robusti dei metodi Newtoniani. Infatti richiedono soltanto la valutazione del gradiente e non di derivate di ordine successivo. Nella forma base la regola di aggiornamento è $\theta_{r+1} = \theta_r - \eta \nabla f(\theta_r)$. La scelta di η è un punto cruciale nelle situazioni pratiche:

- una lunghezza dei passi η troppo piccola può tradursi in una discesa troppo lenta e comportare un elevato numero di iterazioni per il raggiungimento del punto di ottimo $\hat{\theta}$.
- Al contrario passi η troppo lunghi possono saltare il punti di arrivo come in figura 1.1.

L'ideale sarebbe adattare il tasso di aggiornamento η alla curvatura della funzione in modo da raggiungere $\hat{\theta}$ col minor numero di iterazioni possibile e ivi fermarsi. Ricorrendo ad un esempio si può pensare a θ_r come ad una palla che rotola sulla superficie $\nabla f(\theta)$, si

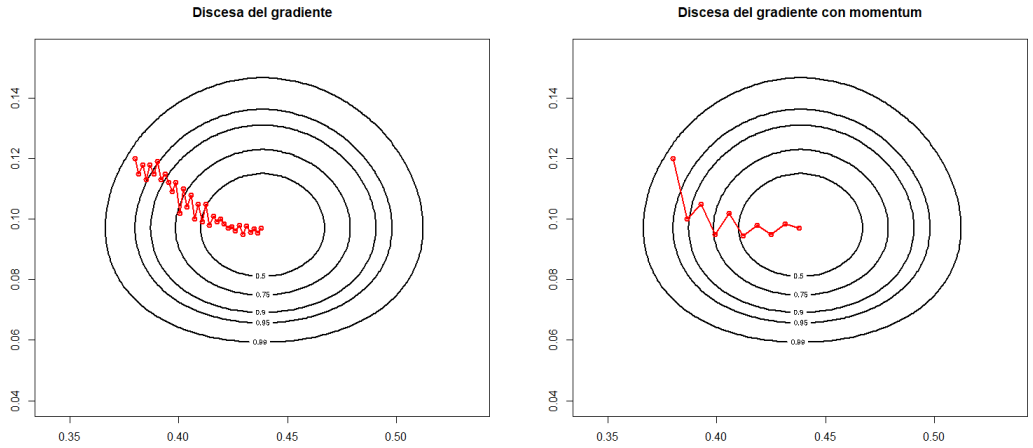

Figura 1.2: Discesa del gradiente e *momentum*, il *momentum* rafforza la discesa lungo le componenti che in cui la funzione $f(\theta)$ si mantiene decrescente per più tempo, sono necessarie quindi meno iterazione per raggiungere l'ottimo.

vuole fare in modo che θ_r raggiunga la valle $\hat{\theta}$ con la maggior velocità possibile prestando attenzione a non procedere oltre. Si vuole quindi:

1. accelerare la discesa dove il gradiente $\nabla f(\theta)$ si mantiene negativo nel tempo,
2. tenere conto della curvatura della superficie e rallentare la discesa nei tratti più ripidi in modo da non saltare il punto di ottimo.

Il primo obiettivo è raggiunto dando inerzia alla palla θ_r ovvero incrementando il tasso η lungo le direzioni in cui il gradiente si mantiene negativo per più iterazioni. In questo modo nelle funzioni multivariate in cui $\theta \in \mathbb{R}^n$, sono rafforzati gli aggiornamenti delle componenti che per più tempo hanno contribuito alla decrescita di $f(\theta)$. A tal fine il calcolo del *momentum* u_r riduce il numero di spostamenti sulla superficie $\nabla f(\theta)$ ed accelera la convergenza come in Figura 1.2:

$$u_r = \gamma\theta_{r-1} + \eta\nabla f(\theta_r)$$

$$\theta_{r+1} = \theta_r - u_r$$

Il termine γ è un iperparametro fissato solitamente a 0.9 come indicato in Ruder (2016). In modo simile si vuole rallentare la discesa lungo le componenti in cui $f(\theta)$ è molto ripida o presenta un comportamento molto variabile. È frequente infatti registrare valori dei gradienti molto elevati negli intorni di un punto di ottimo. (Nocedal et al. 2006). Sia $\{\nabla f(\theta_r)\}_{r\in\mathbb{N}}$ la successione dei gradienti, una stima del grado di curvatura della funzione lungo il percorso $\{\theta_r\}_{r\in\mathbb{N}}$ è data da $\sum_{r\in\mathbb{N}} \nabla f(\theta_r)^2$. È quindi preferibile ridurre

la velocità di discesa lungo le direzioni in cui la somma dei quadrati dei gradienti si fa elevata.

L'algoritmo **ADAM** è un algoritmo di ottimizzazione di massima discesa che tiene in considerazione entrambe le esigenze appena descritte utilizzando stime corrette di media e varianza della successione $\nabla f(\theta_1), \nabla f(\theta_2), , \dots \nabla f(\theta_r), \dots$ nel seguente modo:

$$
\begin{aligned}
m_r &= \alpha_1 m_{r-1} + (1-\alpha_1)\nabla f(\theta_r) \\
v_r &= \alpha_2 v_{r-1} + (1-\alpha_2)(\nabla f(\theta_r))^2 \\
\hat{m}_r &= \frac{m_r}{1-\alpha_1} \\
\hat{v}_r &= \frac{v_r}{1-\alpha_2}
\end{aligned}
$$

Il valore degli iperparametri è usualmente inizializzato con $\alpha_1 = 0.9$, $\alpha_2 = 0.999, m_1 = v_1 = 0$ ed $\epsilon \approx 0$, $\hat{m}_r$ e $\hat{v}_r$ sono correzioni per le stime dei due momenti indicate in Kingma et al. (2014). La risultante regola di aggiornamento è:

$$
\theta_{r+1} = \theta_r - \frac{\eta}{\sqrt{\hat{v}_r} + \epsilon}\hat{m}_r
$$

Come si nota buna parte della performance degli algoritmi di massima discesa in generale, e di **ADAM** in particolare, consiste nel calcolo dei gradienti in corrispondenza dei punti θ_r proposti, motivo per cui la modalità di calcolo delle derivate impatta in maniera significativa la qualità dei risultati.

1.1.4 Valutazioni finali

Al di là di dettagli troppo tecnici ciò che qui sottolineiamo è che i metodi di Newton e quasi Newton registrano una velocità di convergenza quadratica nel senso che la distanza teorica tra θ_r e $\hat{\theta}$ decresce quadraticamente ad ogni iterazione. Sono quindi più rapidi e meno robusti dei metodi di massima discesa, che hanno invece una velocità di convergenza lineare. La violazione dell'ipotesi di definita positività di $H(\theta)$ causa uno sviluppo della successione $\{\theta_r\}_{r\in\mathbb{N}}$ lungo una direzione di non discesa. I metodi di ottimizzazione Newtoniani hanno goduto di un diffuso utilizzo proprio per il minimo numero di valutazioni del gradiente necessario alla ricerca dei punti di ottimo, poiché l'operazione di differenziazione porta con sé rischio di instabilità numerica e di elevato onere computazionale. tali rischi, come si vedrà, sono stati fortemente ridotti dallo sviluppo

dei metodi di differenziazione dutomatica di cui beneficiano maggiormente i metodi di massima discesa.

1.2 Differenziazione automatica

1.2.1 Calcolo delle derivate

L'operazione di differenziazione ricopre un ruolo fondamentale per la buona ricerca di un punto di ottimo. La validità e l'accuratezza dei risultati ottenuti dipende dalla qualità e precisione del metodo di calcolo delle derivate utilizzato. I metodi a disposizione del calcolatore per l'operazione di differenziazione sono tre e si distinguono per punti di forza e carenze.

Il modo più semplice di derivare è applicare la definizione di gradiente come limite ed approssimarne il valore per differenze finite, se $\theta \in \mathbb{R}^n$ ed e_i il vettore canonico, ossia un vettore la cui $i$esima è l' unica componente non nulla ed uguale 1, la derivata parziale rispetto la $i$esima componente del parametro è definita da:

$$\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{h \to 0} \frac{f(\theta + e_i h) - f(\theta)}{h}$$

dove $h \in \mathbb{R}$. La traduzione della definizione di limite in operazione accessibile al calcolatore è ottenuta con l'approssimazione per differenze finite di θ. È sufficiente selezionare h sufficientemente piccolo ed eseguire il calcolo seguente:

$$\frac{\partial f(\theta)}{\partial \theta_i} \approx \frac{f(\theta + e_i h) - f(\theta)}{h}$$

Ci si riferisce a tale metodo con l'espressione **differenziazione numerica**, si tratta del metodo utilizzato dal comando *optim* dell'ambiente di calcolo R quando l'espressione analitica della derivata non è esplicitamente fornita dall'utente. L'implementazione immediata ed intuitiva sconta un'instabilità numerica dovuta alla violazione di due regole auree del calolo numerico:

1. al numeratore viene eseguita una sottrazione fra due numeri molto simili;
2. numeratore e denominatore sono entrambi per definizione di h due numeri vicini allo 0.

Computazionalmente più stabile è la valutazione della derivata tramite **differenziazione simbolica**. Il calcolatore è istruito delle regole di derivazione atte ad ottenere l'espressione analitica della funzione derivata. Una volta ottenuta l'espressione simbolica

della funzione derivata $\frac{f(\theta)}{\partial\theta}$ si è in grado di ottenere il valore esatto e non approssimato della stessa in un punto. Lo svantaggio è la crescita smisurata dell'onere computazionale con l'aumento dei termini della funzione f originale. Ad esempio, per un semplice polinomio con θ scalare ed $f(\theta) = 9\theta(2-\theta)(1-3\theta)^2$, la risultante derivata ottenuta simbolicamente è: $f'(\theta) = 9(2-\theta)(1-3\theta)^2 - 9\theta(1-3\theta)^2 + 54(2-\theta)(1-3\theta)$. Il fenomeno di aumento ingestibile dei termini coinvolti al crescere del grado del polinomio va sotto il nome di *Expression Swell*.

1.2.2 Modalità forward

Da un punto di vista informatico, il calcolo del valore di una funzione non è che una sequenza più o meno articolata di operazioni elementari. La **differenziazione automatica**(AD) scompone la funzione derivabile f nella successione ordinata di funzioni elementari di cui è composta calcolandone per ciascuna il valore della derivata. In analogia alla differenziazione simbolica è quindi richiesto che il calcolatore sia istruito delle regole di derivazione delle funzioni elementari. Si consideri la funzione tangente iperbolica $f : \Theta^2 \to \mathbb{R}$,con $\Theta \in \mathbb{R}^2$, così definita:

$$f(\theta_1, \theta_2) = \frac{1 - e^{-2(\theta_1 + 0.5\theta_2)}}{1 - e^{-2(\theta_1 + 0.5\theta_2)}}$$

Ogni operazione elementare di cui è composta la funzione f è salvata in una variabile intermedia, cui si fa riferimento con la notazione di Griewank et al. (2008) organizzata come segue:

- $v_{j-n} = \theta_j, \quad j = 1, ..., n$ sono le variabili indipendenti o di input
- $v_j, j = 1, ..., q$ sono le variabili intermedie o *working variables* e corripondono ad ogni operazione elementare eseguita per ottenere $f(\theta)$.
- $y_{m-j} = v_{l-j}, j = m-1, m-2, ..., 0$ sono le variabili di output.

La sequenza ordinata di variabili intermedie compone il flusso di calcolo rappresentato dal grafo computazionale in Figura 1.3

Quando il flusso di calcolo viene attraversato, viene registrato sia il valore di ogni variabile intermedia ottenendo la *traccia primale* sia il valore delle derivate parziali in corrispondenza di ciascuna delle variabili intermedie scrivendo le *traccie tangenti*, una per ciascun input. Le tracce relative all'esempio $f(\theta_1, \theta_2) = \tanh\theta$ calcolate in $\theta = (1, \frac{2}{3})$ sono in Tabella 1.1. Per comodità di notazione si è posto $\dot{\theta_j} = \frac{\partial f}{\partial \theta_j}$ ed allo stesso modo $\dot{v} = \frac{\partial f}{\partial v_j}$ È importante osservare che in questa modalità la scrittura della traccia tangente

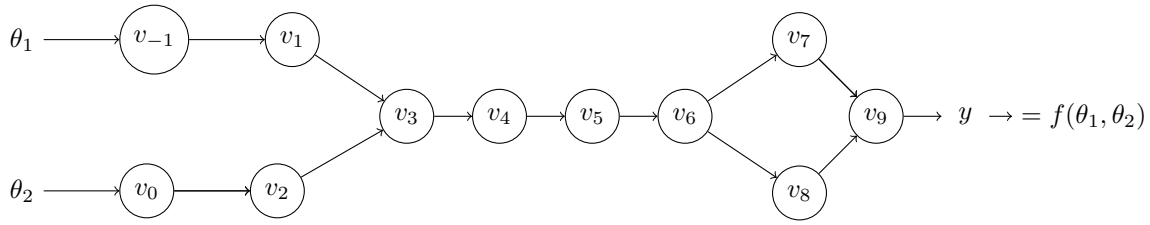

Figura 1.3: Grafo computazionale di $f(\theta_1, \theta_2)$

Traccia primale	Traccia tangente θ_1	Traccia tangente θ_2
$v_{-1} = \theta_1 = 1$ $v_0 = \theta_2 = 0.333$	$\dot{v}_{-1} = \dot{\theta_1} = 1$ $\dot{v_0} = \dot{\theta_1} = 0$	$v_{-1} = \dot{\theta_1} = 0$ $\dot{v_0} = \dot{\theta_2} = 1$
$v_1 = v_{-1} = 1$ $v2 = 0.5v_0 = \frac{1}{6}$ $v_3 = v_1 + v_2 = 1 + \frac{1}{6} = \frac{7}{6}$ $v_4 = 2v_3 = \frac{7}{3}$ $v_5 = -v_4 = -\frac{7}{3}$ $v_6 = e^{v5} = e^{-\frac{7}{3}}$ $v_7 = 1 - v_6 = 1 - \exp\{-\frac{7}{3}\}$ $v_8 = 1 + v_6 = 1 + e^{-\frac{7}{3}}$ $v_9 = \frac{v_7}{v_8} = \frac{1-e^{-\frac{7}{3}}}{1+e^{-\frac{7}{3}}}$	$\dot{v}_1 = \dot{v}_{-1} = 1$ $\dot{v}_2 = 0.5\dot{v}_0 = 0$ $\dot{v}_3 = \dot{v}_1 + \dot{v}_2 = 1$ $\dot{v}_4 = 2\dot{v}_3 = 2$ $\dot{v}_5 = -\dot{v}_4 = -2$ $\dot{v}_6 = \dot{v}_5 e^{v_5} = -2e^{-\frac{7}{3}}$ $\dot{v}_7 = 0 + (-1)\dot{v}_6 = 2\exp\{-\frac{7}{3}\}$ $\dot{v}_8 = 0 + \dot{v}_6 = -2e^{-\frac{7}{3}}$ $\dot{v}_9 = \frac{\dot{v}_7 v_8 - v_7 \dot{v}_8}{v_8^2} = \frac{4e^{-\frac{7}{3}}}{\left(1+e^{-\frac{7}{3}}\right)^2}$	$\dot{v}_1 = \dot{v}_{-1} = 0$ $\dot{v}_2 = 0.5\dot{v}_0 = 0.5$ $\dot{v}_3 = \dot{v}_1 + \dot{v}_3 = 0.5$ $\dot{v}_4 = 2\dot{v}_3 = 1$ $\dot{v}_5 = -\dot{v}_4 = -1$ $\dot{v}_6 = \dot{v}_5 \exp\{v_5\} = -\exp\{-\frac{7}{3}\}$ $\dot{v}_7 = -e^{-1}$ $\dot{v}_8 = e^{-1}$ $\dot{v}_9 = \frac{\dot{v}_7 v_8 - v_7 \dot{v}_8}{v_8^2} = \frac{2e^{-\frac{7}{3}}}{\left(1+e^{-\frac{7}{3}}\right)^2}$
$y_1 = 0.8232006$	$\dot{v_{10}} = 0.3223406$	$\dot{v_{10}} = 0.1611703$

Tabella 1.1: Differenziazione automatica in modalità forward

nei nodi comporta un'operazione di calcolo per ciascuna variabile coinvolta nel nodo. Vuol dire che nei nodi del grafo computazionale in Figura 1.3 in cui sono coinvolte più variabili come $v_3 = 2\theta_1 + \theta_2$ vengono eseguite due operazioni una per θ_1 e l'altra per θ_2. La performance della modalità in avanti è quindi ottima quando la funzione è scalare $f : \mathbb{R} \rightarrow \mathbb{R}$

1.2.3 Modalità Backward

Per funzioni a più variabili come quella dell'esempio conviene scrivere la traccia tangente percorrendo il grafo in senso inverso. In questo modo i nodi intermedi connessi che ricevono connessioni provenienti da più componenti di input sono valutati una sola volta. In questo modo è possibile scrivere una sola traccia valida per tutti gli input. Sia $\bar{v}_j = \frac{\partial f}{\partial v_j}$ la derivata di $f(\theta_1, \theta_2)$ rispetto alla variabile intermedia v_j. Per la regola di

Traccia primale	Traccia tangente
$v_{-1} = \theta_1 = 1$ $v_0 = \theta_2 = 0.333$	$\bar{v}_{-1} = \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = 0.3223406$ $\bar{v}_0 = \bar{v}_2 \frac{\partial v_2}{\partial v_0} = 0.5 \times \bar{v}_3 = 0.1611703$
$v_1 = v_{-1} = 1$ $v2 = 0.5v_0 = \frac{1}{6}$ $v_3 = v_1 + v_2 = 1 + \frac{1}{6} = \frac{7}{6}$ $v_4 = 2v_3 = \frac{7}{3}$ $v_5 = -v_4 = -\frac{7}{3}$ $v_6 = e^{v5} = e^{-\frac{7}{3}}$ $v_7 = 1 - v_6 = 1 - \exp\{-\frac{7}{3}\}$ $v_8 = 1 + v_6 = 1 + e^{-\frac{7}{3}}$ $v_9 = \frac{v_7}{v_8} = \frac{1-e^{-\frac{7}{3}}}{1+e^{-\frac{7}{3}}}$	$\bar{v}_1 = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \bar{v}_3 \times 1$ $\bar{v}_2 = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = \bar{v}_3$ $\bar{v}_3 = \bar{v}_4 \frac{\partial v_4}{\partial v_3} = 2 \times \bar{v}_4 = 0.3223407$ $\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = -1 \times \bar{v}_5$ $\bar{v}_5 = \bar{v}_6 \frac{\partial v_6}{\partial v_6} = \bar{v}_6 e^{v_5} = -0.1611703$ $\bar{v}_6 = \bar{v}_7 \frac{\partial v_7}{\partial v_6} + \bar{v}_8 \frac{\partial v_8}{\partial v_6} = \bar{v}_7 + \bar{v_8} = -1.66203$ $\bar{v}_7 = \bar{v}_9 \frac{\partial v_9}{\partial v_7} = 0.9116003$ $\bar{v}_8 = \bar{v}_9 \frac{\partial v_9}{\partial v_8} = -\frac{v_7}{v_8} = -0.75043$ $\bar{v}_9 \frac{\partial v_{10}}{\partial v_9} = 1$
$y_{10} = 0.8232006$	$\bar{v_{10}} = 1$

Tabella 1.2: Traccia primale e tangente in modalità backward

differenziazione delle funzioni composte avremo:

$$\frac{\partial f}{\partial \theta_1} = \frac{\partial y}{\partial v_9}\frac{\partial v_9}{\partial v_8}\frac{\partial v_8}{\partial v_7}\cdots\frac{\partial v_2}{\partial v_{-1}} \quad \text{e} \quad \frac{\partial f}{\partial \theta_2} = \frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_9}\frac{\partial v_9}{\partial v_8}\frac{\partial v_8}{\partial v_7}\cdots\frac{\partial v_1}{\partial v_0}$$

Il calcolo delle tracce è riportato in Tabella 1.2 La differenziazione automatica in modalità *backward* è adatta sia alle funzioni multivariate che alle funzioni a valore vettoriali.

1.3 Software per la differenziazione automatica

Con la differenziazione automatica si è in grado di calcolare il valore esatto della derivata di una funzione in un punto senza ricorrere ad approssimazioni o all' impiego inefficiente della memoria. I requisiti che la funzione deve rispettare sono soltanto quelli di continuità e derivabilità. La differenziazione automatica è uno strumento relativamente recente per cui non tutti gli ambienti di calcolo sono dotati di strumenti che la implementino, la Tabella 1.3 yoffre una panoramica sui principali strumenti ad oggi disponibili e relativi riferimenti.

Questi strumenti sono stati sviluppati principalmente per applicazioni di *machine learning* e scienze computazionali, dove l'efficienza e la precisione del calcolo delle derivate sono cruciali. Ad esempio, **Autograd** e **TensorFlow** sono ampiamente utilizzati nella comunità del machine learning per la costruzione e l'addestramento di modelli complessi. D'altra parte, strumenti come **ADOL-C**, **Tapenade** e **Stan Math Library** sono più orientati verso applicazioni ingegneristiche, scientifiche e di modellazione statistica, offrendo supporto per linguaggi come C++ e Fortran. Questa selezione di

strumenti dimostra la diversità delle implementazioni di AD disponibili, consentendo agli sviluppatori e ai ricercatori di scegliere l'opzione che meglio si adatta alle loro esigenze specifiche. Una buona parte delle implementazioni illustrate nel Capitolo 5, per la complessità e il numero di parametri coinvolti nelle funzioni da ottimizzare ha richiesto l'impiego di strumenti informatici che supportano la differenziazione automatica. In particolare l'ambiente di calcolo utilizzato è stato Python di cui per la differenziazione automatica si è usata la libreria **torch**.

Tabella 1.3: Elenco riassuntivo degli strumenti informatici per eseguire differenziazione automatica

Strumento	**Tipo di Strumento**	**Linguaggio**	**Riferimenti**
Autograd	Libreria	Python	Maclaurin et al. (2015)
TensorFlow	Libreria	Python, C++	Abadi et al. (2016)
PyTorch	Libreria	Python, C++	Paszke et al. (2017)
Ceres Solver	Libreria	C++	Agarwal et al. (2022)
ForwardDiff.jl	Libreria	Julia	Revels et al. (2016)
DiffSharp	Libreria	F#	Baydin et al. (2015)
Zygote	Libreria	Julia	Innes et al. (2018)
Tapenade	Strumento standalone	Fortran, C	Hascoët et al. (2013)
Stan Math Library	Libreria	C++	Carpenter et al. (2017)

Capitolo 2

Modelli impliciti

2.1 Inferenza di verosimiglianza

La specificazione di un modello parametrico corrisponde all'individuazione di un insieme di densità $\mathbb{F} = \{p(y;\theta), \theta \in \Theta\}$,indicizzate da θ, in cui ricercare la legge generatrice $p_0(y;\theta)$. Il modello è identificato quando la relazione fra θ e $p(\theta;y)$ è biunivoca nel senso che ogni valore di Θ identifica una e una sola densità $p(y;\theta)$. Le densità $p(y;\theta) \in \mathbb{F}$ sono note a meno della componente parametrica. L'accesso alla funzione di densità rende possibile la costruzione di una funzione di verosimiglianza ed una procedura inferenziale rigorosa.

Una volta osservati i dati $p(y_{oss};\theta)$ è una funzione solo di θ e la verosimiglianza è definita come $L(\theta) = c(y_{oss})p(y_{oss};\theta)$. La stima di massima verosimiglianza coincide con la soluzione del seguente problema di ottimo:

$$\hat{\theta} = \arg\max_{\theta}(L(\theta)) = \arg\max_{\theta}(\log L(\theta)).$$

Lo stimatore di massima verosimiglianza gode di proprietà importanti ai fini di affermazioni statistiche accurate e sono:

- **Consistenza Debole**. Errore e varianza dello stimatore convergono a 0 quando la numerosità campionaria cresce

$$n \to \infty \implies \mathbb{E}\{\hat{\theta}\} = \theta_0 \quad e \quad Var(\hat{\theta}) \to 0$$

- **Distribuzione approssimativamente normale**

$$\hat{\theta} \dot{\sim} N\Big(\theta, \sqrt{Var(\hat{\theta})}\Big)$$

- **Invarianza** rispetto a riparametrizzazioni del modello.

- Inoltre $\hat{\theta}$ si trova in corrispondenza del minimo della divergenza di Kullback-Leibler D_{KL}. La quantità D_{LK} è una misura della distanza fra due densità. Sia $p(y;\theta) \in \mathcal{F}$ e $p_0(y) \notin \mathcal{F}$ allora:

$$D_{KL}(p_0(y,\theta), p(y;\theta)) = \mathbb{E}\left\{\log\left(\frac{p(y;\theta)}{p_0(y)}\right)\right\} = \int p_0(y)\log\left(\frac{p(y;\theta)}{p_0(y)}\right)dy$$

  Lo stimatore di massima verosimiglianza $\hat{\theta}$, anche quando il modello non è correttamente specificato individua la distribuzione $p(\theta;y)$ più vicina a $p_0(y)$ in termini della divergenza Kullback-Leibler.

2.1.1 Statistiche sufficienti

La funzione di verosimiglianza opera una partizione dello spazio campionario. Ad ogni elemento $y \in \mathcal{Y}$ dello spazio campionario è associata una verosimiglianza. È tuttavia anche vero che diverse realizzazioni di y possono portare alle stesse conclusioni inferenziali in termini di verosimiglianza. Una statistica $S(y)$ è una trasformazione dei dati originari y ed è detta *sufficiente* per θ se assume lo stesso valore in più punti dello spazio campionario solo se questi punti hanno verosimiglianze che differiscono soltanto per una costante di proporzionalità $c(y)$, cioè se per ogni $y, z \in \mathcal{Y}$

$$s(y) = s(z) \implies L(\theta;y) \propto L(\theta;z) \quad \text{per ogni } \theta \in \Theta$$

Se $S(y)$ è una statistica sufficiente viene operata una riduzione dei dati che conserva tutta l'informazione utile all'inferenza su θ, inoltre $L(\theta)$ dipende dai dati y solo tramite $s(y)$ ovvero esiste una funzione g tale che: $L(\theta) \propto g(S(y);\theta)$. In maniera alternativa una statistica è sufficiente se la distribuzione di Y condizionata a T $\Pr(Y|S = S(y))$ non dipende da θ.

2.2 Verosimiglianze intrattabili

Per alcuni modelli statistici l'accesso alla funzione di densità o di probabilità non è possibile e la strada dell'inferenza di verosimiglianza è non praticabile. Casi di questo

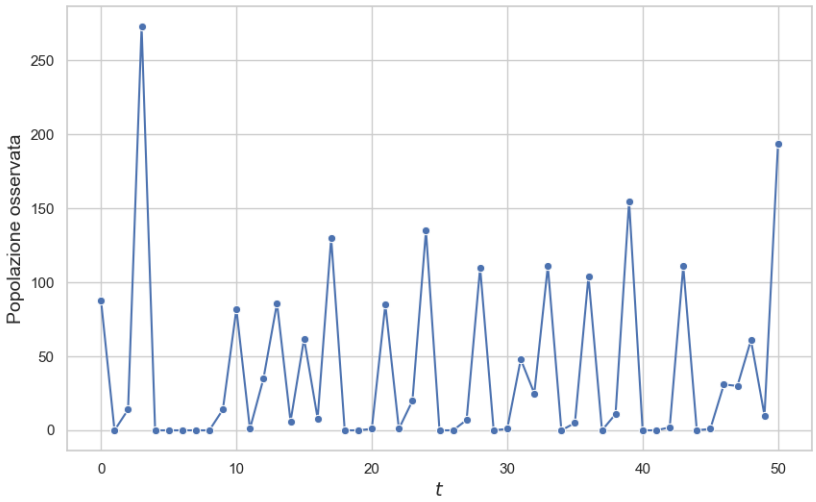

Figura 2.1: Popolazione osservata secondo il modello di Ricker generati a partire da $\theta_0 = (3.8, 0.1^2, 10)$

tipo sono diffusi in fisica, ingegneria e biologia. Un esempio noto di intrattabilità è il modello di Ricker che descrive l'evoluzione del numero di individui N_t nel tempo:

$$N_{t+1} = rN_t \exp\{-N_t + e_t\},$$

dove $e_t \sim N(0, \sigma_e^2)$ è il termine di disturbo ed r un tasso di crescita che regola la dinamica del modello. Sia infine $\phi \in \mathbb{R}^+$, si indichi con $\theta = (\log(r), \sigma_e, \phi)$ Wood 2010 e si supponga di osservare una realizzazione del processo $Y \sim Pois(\phi N_t)$ generata a partire da $\theta_0 = (3.8, 0.1^2, 10)$. Il grafico in Figura **??** riporta il numero di individui y nella popolazione osservato al passare del tempo.

Anche se in linea teorica un modello statistico è disponibile poiché è nota la natura del processo, per l'inferenza di verosimiglianza su θ si dovrebbe risolvere l' integrare rispetto i termini d'errore:

$$p(y; \psi, r, \sigma_e) = \int p(y|e, r, \phi)p(e; \sigma_e^2)\, de. \tag{2.1}$$

La dipendenza delle osservazioni, l'inclusione non lineare dei termini di errore e_t configurano una funzione di verosimiglianza complicata e difficile da gestire. L'integrale in Equazione 2.1 è difficilmente calcolabile per cui si fa ricorso a verosimiglianze surrogato come in Wood (2010) o si ricercano approssimazioni del modello come in Davies (2024). L'ostacolo dell'integrazione in questo caso, e in altri simili, rende inutilizzabili

le procedure inferenziali di verosimiglianza e necessario l'utilizzo di metodi sostitutivi.

In alternativa la funzione di densità può essere del tutto non reperibile. Le distribuzioni $k - g$ sono spesso usate per al loro flessibilità e capacità di modellare dati non standard (Vihola et al. 2022). Pur non disponendo di una funzione di densità chiusa e ben definita è possibile simulare dati Q da tali distribuzioni seguendo lo schema di seguito (Sisson et al. 2018):

$$Q = \mu + \sigma \left[1 + 0.9 \frac{1 - exp(-g \cdot u)}{1 + exp(-g \cdot u)} \right] (1 + u^2)^k \cdot u, \quad u \sim N(0, 1)$$

I parametri μ e σ sono rispettivamente di scala e posizione mentre g e k regolano asimmetria e kurtosi, $u \sim N(0, 1)$. I metodi che eseguono inferenza su θ nella condizione di funzione di verosimiglianza intrattabile o totalmente assente sono detti di tipo *free likelihood inference*.

2.3 Simulatori e modelli impliciti

Nei metodi *free likelihood inference* l'assenza della verosimiglianza o di un modello statistico esplicito è rimediata dalla capacità di simulare dati artificiali con l'utilizzo di un simulatore. Un simulatore $\mathbb{S}\{y|\theta\}$ è un programma informatico che prende in ingresso un parametro θ e restituisce un vettore di dati y *pseudocausali*. La proprietà di pseudocausalità dei dati generati dal simulatore deriva dal campionamento di strati interni z che definiscono la componente latente del simulatore. In generale un simulatore opera una sequenza ordinata di operazioni. Siano $p_{\psi_1}, p_{\psi_2}, \ldots, p_{\psi_K}$ e p_ϕ, delle distribuzioni di probabilità note parametrizzate da $\psi_1, \psi_2, \ldots, \psi_K$, ed $f_1, f_2, \ldots f_k, g$ delle funzioni arbitrarie, si può riassumere il funzionamento di un simulatore come di seguito:

1. Si ottiene ϕ_1 come trasformazione del parametro di ingresso θ:

$$\phi_1 = f_1(\theta)$$

2. Si campiona il primo elemento latente z_1 dalla distribuzione p_{ψ_1}:

$$z_1 \sim p_{\phi_1}$$

3. Si ottiene ψ_2 come trasformazione degli elementi precedenti:

$$\psi_2 = f_2(z_1, \theta)$$

4. Si campiona il secondo strato latente da p_{ψ_2} e si procede così fino a z_K

$$z_2 \sim p(\psi_2)$$
$$\dots$$
$$\psi_K = f_K(z_{1:K-1}, \theta)$$
$$z_K \sim p_{\psi_K}$$

5. In ultimo si ottiene ϕ e si campiona y da p_ϕ

$$\phi = g(z_{1:K}, \theta)$$
$$y \sim p_\phi$$

Si nota ora che se tutte le distribuzioni coinvolte $p_{\psi_1}, p_{\psi_2}, \dots p_{\psi_K}, p_\phi$ fossero trattabili lo sarebbe anche la distribuzione congiunta di y e $z_{1:K}$ dato θ:

$$p(y, z_{1:k}|\theta) = p_\psi(y) \prod_k p_{\phi_k}(z_k),$$

ed una volta fissati i dati y_{oss} si potrebbe ottenere la verosimiglianza integrando la quantità:

$$p(y|\theta) = \int p(y_{oss}, z_{1:k}|\theta)\, dz_{1:k}.$$

Si può concludere quindi che un simulatore definisce implicitamente un modello statistico da cui potenzialmente si può ricavare una funzione di verosimiglianza. Tuttavia l'operazione di integrazione è spesso non eseguibile sia analiticamente sia numericamente. La difficoltà nel calcolo dell'integrale è dovuta:

- alla dimensione dello strato latente $z_{1:K}$, ed è il caso dei simulatori che riproducono il moto di particelle in fisica
- alla complessità delle strutture informatiche che compongono il simulatore. La possibilità di conoscere tutte le distribuzioni coinvolte $\psi_1, \psi_2, \dots, \psi_K, p_\phi$ è spesso solo teorica.

L'inferenza basata su un modello di simulazione va quindi eseguita con metodi e strumenti non riconducibili ai noti metodi di verosimiglianza. Nel seguito di questa tesi con la scrittura θ^* si indicheranno i valori di θ dati in ingresso al simulatore $\mathbb{S}\{y|\theta^*\}$ che quindi restituisce in uscita i dati artificiali $y^* \sim \mathbb{S}\{y|\theta^*\}$. Spesso ci sarà l'esigenza di campionare i valori θ^* da una distribuzione proposta indicata con $\pi(\theta)$ oppure con $u(\theta)$

a seconda che si tratti della distribuzione a priori o di una distribuzione proposta più generale non meglio specificata.

Capitolo 3

Metodi bayesiani

3.1 Inferenza bayesiana

In questo capitolo, dopo una breve rassegna dei principi dell'inferenza bayesiana, si discuterà di come alcuni metodi computazionali sviluppati in ambito bayesiano possano essere adattati ai problema di *free likelihood inference* fornendo soluzioni all' assenza di una verosimiglianza dei modelli impliciti.

Nella statistica bayesiana la verosimiglianza è utilizzata per aggiornare l'informazione pre-sperimentale contenuta nella distribuzione a priori $\pi(\theta)$. Nei modelli espliciti continui il problema inferenziale si risolve con la determinazione della distribuzione a posteriori $\pi(\theta|y_{oss})$ che combina l'informazione a priori $\pi(\theta)$ con l'evidenza empirica fornita dalla verosimiglianza:

$$\pi(\theta|y_{oss}) = \frac{\pi(\theta)p(y|\theta)}{\int \pi(\theta)p(y|\theta)\,d\theta}.$$

Affinché sia possibile ottenere la forma analitica della distribuzione a posteriori è necessario scegliere $\pi(\theta)$ in modo che la costante di integrazione sia analiticamente o numericamente calcolabile. A tal fine è di solito conveniente scegliere $\pi(\theta)$ fra le distribuzioni coniugate in modo che $\pi(\theta|y_{oss})$ appartenga alla stessa famiglia di $\pi(\theta)$. Ancora una volta il vincolo della scelta della distribuzione a priori sta nell'integrabilità del denominatore.

3.2 Simulazione Markov chain Monte Carlo

Se la scelta di $\pi(\theta)$ è tale da rendere la costante di normalizzazione al denominatore irrisolvibile si possono usare metodi computazionali per ottenere una versione empirica della posteriori da usare per le conclusioni inferenziali. La possibilità di ottenere una

Riquadro 1: Markov Chain Monte Carlo per l'inferenza bayesiana su θ

Input: Valore iniziale θ_1, distribuzione $\pi(\theta|y_{oss})$, distribuzione proposta $\omega(.|\theta_1, \epsilon)$, lunghezza della catena C

Output: Catena di Markov $\{\theta_1, \theta_2, \ldots, \theta_C\}$

1: **for** $i = 2$ to C **do**
2: Genera un candidato θ da $\omega(.|\theta_{i-1}; \epsilon)$
3: Calcola il rapporto di accettazione $\tau = \frac{\pi(\theta|y_{oss})}{\pi(\theta_{i-1}|y_{oss})}$
4: Estrai un numero u da una distribuzione uniforme $U(0, 1)$
5: **if** $u \leq \min(1, \tau)$ **then**
6: Accetta il candidato: $\theta_i = \theta$
7: **else**
8: Rifiuta il candidato: $\theta_i = \theta_{i-1}$
9: **end if**
10: **end for**

valutazione empirica della posteriori $\pi(\theta|y_{oss})$ richiede la conoscenza della densità a meno della costante di normalizzazione. È sufficiente quindi disporre di:

$$\pi(\theta|y_{oss}) \propto \pi(\theta)p(y_{oss}|\theta).$$

I metodi Monte Carlo basati su Catene di Markov (MCMC) sono una classe di algoritmi per il campionamento da distribuzioni di probabilità basati sulla costruzione di una catena di Markov le cui realizzazioni correnti dipendono solo da quelle immediatamente precedenti. Lo spazio parametrico Θ viene esplorato con una camminata aleatoria. Alla $j$esima iterazione il valore proposto θ viene quindi generato da una distribuzione $\omega(\cdot|\theta_{j-1}; \epsilon)$, di cui ϵ regola la varianza. Il criterio di accettazione del valore candidato riguarda la quantità:

$$\tau = \frac{\pi(\theta)p(y_{oss}|\theta)\omega(\theta_{j-1}|\theta)}{\pi(\theta_{j-1})p(y_{oss}|\theta_{j-1})\omega(\theta|\theta_{j-1})},$$

detta probabilità di accettazione. Nel seguito si assumerà $\omega(\cdot|\cdot, \epsilon)$ essere una densità simmetrica per cui il fattore $\frac{\omega(\theta_{j-1}|\theta)}{\omega(\theta|\theta_{j-1})}$ non sarà considerato poiché pari ad 1. L'algoritmo è riportato in dettaglio nel Riquadro 1. La scelta del parametro di varianza ϵ delle realizzazioni che compongono la camminata aleatoria è da farsi prima della partenza dell'algoritmo ed è una scelta che determina la qualità della distribuzione a posteriori empirica ottenuta. Valori di ϵ troppo piccoli potrebbero comportare un elevato tasso di accettazione e quindi una scarsa esplorazione dello spazio Θ. Al contrario valori troppo elevati potrebbero causare la rapida fuga dai punti di maggiore densità e quindi una versione empirica poco fedele alla vera distribuzione a posteriori. Regole euristiche prevedono che ϵ vada scelto in modo che il tasso di accettazione sia compreso fra il 25% ed il 50%. Nei problemi multivariati si tende a preferire un tasso di accettazione più vicino al 25%.

3.3 Calcoli bayesiani approssimati

Gli algoritmi *MCMC* forniscono un metodo per esplorare lo spazio parametrico sulla base di un criterio quantitativo di accettazione dei valori proposti. Nel caso dei modelli impliciti l'impossibilità di reperire una funzione di verosimiglianza invalida il criterio di accettazione dei valori candidati illustrato in precedenza. In generale, un algoritmo di accettazione rifiuto ha l'obiettivo di simulare campioni da una distribuzione di probabilità obiettivo $f(\theta)$ a partire da una distribuzione proposta $u(\theta)$. Sia $X \geq \max_\theta \frac{f(\theta)}{u(\theta)}$ e $u(\theta)$ scelta in modo tale che $u(\theta) > 0$ se $f(\theta) > 0$, un algoritmo di questo tipo esegue i passi seguenti:

- Genera θ^* da $u(\theta)$
- Accetta θ^* con probabilità $\tau = \frac{f(\theta^*)}{Xu(\theta^*)}$

Si noti che se $f(\theta) = \pi(\theta)p(y_{oss}|\theta)$ e $u(\theta) = \pi(\theta)$ risulta che la probabilità di accettazione $\tau = \frac{\pi(\theta)p(y_{oss}|\theta)}{X\pi(\theta)} \propto p(y_{oss}|\theta)$, al contrario se $u(\theta) \neq \pi(\theta)$ allora . Se si considera per il momento y realizzazione di una variabile aleatoria discreta è possibile interpretare la probabilità di accettazione come la probabilità che il processo implicito al simulatore $\mathbb{S}\{y|\theta\}$ generi i dati osservati sotto un fissato valore di θ. Oppure, in altre parole, se fissiamo θ^* e generiamo osservazioni y^* la probabilità di osservare precisamente $y^* = y_{oss}$ è proporzionale al tasso di accettazione. Il punto cruciale di queste considerazioni è che in un contesto generale in cui $u(\theta) \neq \pi(\theta)$ da cui $\tau = \frac{\pi(\theta)}{u(\theta)} \frac{p(y_{oss}|\theta)}{X}$, una volta proposto un valore candidato θ^* estratto da $u(\theta)$, accettiamo tale valore proposto con probabilità $\frac{\pi(\theta^*)}{u(\theta^*)X}$ solo se per le osservazioni simulate $y^* \sim \mathbb{S}\{y|\theta^*\}$ vale $y^* = y_{oss}$ (Sisson et al. 2018). L'uguaglianza $y_{oss} = y^*$ sostituisce il ruolo della verosimiglianza nella regola di accettazione.

Nel caso continuo, che è di nostro interesse, l'evento $y^* = y_{oss}$ ha probabilità 0 per cui è necessario ricorrere ad un altro criterio quantitativo di similarità. I metodi di calcolo bayesiano approssimato (ABC) utilizzano quindi il simulatore per generare dati artificiali $y^* \sim \mathbb{S}\{y|\theta^*\}$ da confrontare con i dati reali valutando la plausibilità di θ^* come valore generatore dei dati y_{oss} sulla base della somiglianza fra dati simulati y^* e dati realmente osservati y_{oss} definita da un criterio quantitativo $\rho(y_{oss}, y^*)$. In seguito assumeremo che il criterio si similarità $\rho(\cdot, \cdot)$ sia la norma euclidea $||.||$.

Sia $\varepsilon > 0$ e $D_\varepsilon(y_{oss})$ l'intorno di y_{oss} definito come l'insieme dei punti y che hanno distanza da y_{oss} al massimo uguale ad ε:

$$D_\varepsilon(y_{oss}) = \{y : ||y - y_{oss}||) \leq \varepsilon\} \tag{3.1}$$

Possiamo approssimare localmente[1] la verosimiglianza $p(y_{oss}|\theta)$ con (Prangle 2017):

$$p(y_{oss}|\theta) \approx \frac{\Pr(||y - y_{oss}|| \leq \varepsilon|\theta)}{|D_\varepsilon(y_{oss})|}, \tag{3.2}$$

$D_\varepsilon(y_{oss})$ indica il volume dell'intorno. Si può interpretare la quantità appena descritta come la concentrazione dei campioni osservabili vicini a y_{oss} in corrispondenza di un preciso valore di θ. Nella pratica si è interessati ad una stima del solo numeratore, facilmente reperibile con l'utilizzo del simulatore $\mathbb{S}\{y|\theta\}$. Sia N il numero di simulazioni in corrispondenza di un valore proposto θ^*, il calcolo di una stima del nominatore è operato via:

$$\Pr(||y - y_{oss}|| \leq \varepsilon|\theta^*) \approx \frac{1}{N}\sum_{l=1}^{N} I(||y_l^* - y_{oss}|| \leq \varepsilon), \quad \text{dove} \quad y_i^* \sim \mathbb{S}\{y|\theta^*\}; \tag{3.3}$$

$I(.)$ è la funzione indicatrice e vale 1 se $||y_l^* - y_{oss}|| \leq \varepsilon$ e 0 altrimenti. Si può quindi definire un' approssimazione della distribuzione a posteriori:

$$\pi(\theta|y_{oss}) \approx \frac{\Pr(||y - y_{oss}|| \leq \varepsilon|\theta)\pi(\theta)}{\int_\Theta \Pr(||y - y_{oss}|| \leq \varepsilon|\theta)\pi(\theta)\, d\theta} = \pi(\theta \mid ||y - y_{oss}|| \leq \varepsilon), \tag{3.4}$$

e proporne una versione empirica collezionando i valori proposti θ^* da una distribuzione $u(\theta)$ in corrispondenza dei quali sono generati dati sintetici y^* che soddisfano il criterio di distanza $||y^* - y_{oss}|| \leq \varepsilon$. Nel seguito la posteriori approssimata $\pi(\theta \mid ||y - y_{oss}|| \leq \varepsilon)$ definita nell'Equazione 3.4 sarà indicata con la scrittura $\pi_\varepsilon(\theta|y_{oss})$ e la si può pensare come la posteriori esatta che avremmo ottenuto con osservazioni y alternative ma vicine ad y_{oss}. Mentre per la versione empirica di $\pi_\varepsilon(\theta|y_{oss})$ si scriverà $\hat{\pi}_\varepsilon(\theta|y_{oss})$, ad indicare sia la natura empirica sia la dipendenza dalla soglia ε da cui dipende la qualità della distribuzione a posteriori ottenuta. Più ε si avvicina a 0 più la versione empirica ottenuta sarà simile alla vera distribuzione a posteriori. Infatti, svolgere le operazioni di: 1) campionare θ da $u(\theta)$, 2) simulare dati y da $\mathbb{S}\{y|\theta\}$ in cui è implicitamente definita $p(y|\theta)$, e 3) rifiutare θ se $||y - y_{oss}|| > \varepsilon$, corrisponde ad estrarre coppie (θ, y) da una distribuzione congiunta proporzionale a:

$$I(||y - y_{oss}|| \leq \varepsilon)p(y|\theta)u(\theta).$$

Se inoltre, come visto prima, la generica coppia candidata (y, θ) è accettata con probabilità proporzionale a $\frac{\pi(\theta)}{u(\theta)}$, allora l'algoritmo di accettazione rifiuto in versione ABC sta

[1] località rispetto a θ

Riquadro 2: Accettazione rifiuto ABC con $u(\theta) = \pi(\theta)$

Input: $\pi(\theta)$, y_{oss}, soglia ε
Output: Versione empirica dell'approssimazione $\hat{\pi}_\varepsilon(\theta|y = y_{oss})$
1: **repeat**
2: Campiona $\theta^* \sim \pi(\theta)$
3: Simula $y^* \sim \mathbb{S}\{(y|\theta^*)\}$
4: **until** $||y^* - y_{oss}|| \leq \varepsilon$
5: **return** θ^*

campionando dati provenienti da una densità proporzionale a:

$$I(||y - y_{oss}|| \leq \varepsilon)p(y|\theta)u(\theta)\frac{\pi(\theta)}{u(\theta)} = I(||y - y_{oss}|| \leq \varepsilon)p(y|\theta)\pi(\theta).$$

A questo punto è immediato osservare che se $\varepsilon \to 0$ allora $\pi_\varepsilon(\theta|y_{oss}) \to \pi(\theta|y_{oss})$:

$$\lim_{h\to 0} \int \mathbb{I}(||y - y_{obs}|| \leq \varepsilon)p(y|\theta)\pi(\theta)\, dy = \pi(\theta|y_{oss}).$$

Uno schema per campionare valori dalla distribuzione a posteriori tramite un algoritmo di accettazione rifiuto con metodi bayesiani approssimati e distribuzione proposta $u(\theta) = \pi(\theta)$ è nel Riquadro 2.

3.3.1 Markov chain Monte Carlo ABC

I metodi di campionamento accettazione-rifiuto ed in particolare per inferenza bayesiana approssimata nei casi di verosimiglianza non disponibile sono particolarmente onerosi da un punto di vista computazionale. È richiesto infatti che i valori proposti θ^* da valutare tramite simulazione provengano da tutto lo spazio parametrico Θ. Per questo motivo è utile adattare gli algoritmi di simulazione *MCMC* ai metodi ABC in modo da avere un criterio di esplorazione di Θ.

Come nell'algoritmo MCMC descritto nel Riquadro 1 possiamo perturbare i valori θ accettati per candidarne di nuovi, di modo che, come prima, la successione di valori candidati $\{\theta_1, \theta_2, \ldots \theta_C\}$ sia la realizzazione di una camminata aleatoria sulla superficie dello spazio parametrico. Siccome l'obiettivo in questo caso è ottenere una versione empirica di $\pi_\varepsilon(\theta|y_{oss})$ definita nell'Equazione 3.4 la probabilità di accettazione τ del punto 4 del Riquadro 1 per il $j$esimo candidato θ diventa:

$$\tau = \frac{\Pr(||y - y_{oss}|| \leq \varepsilon|\theta_j)\pi(\theta)}{\Pr(||y - y_{oss}|| \leq \varepsilon|\theta_{j-1})\pi(\theta_{j-1})}.$$

La quantità $\Pr(||y - y_{oss}|| \leq \varepsilon|\theta)$ va stimata come in Equazione 3.2. Tuttavia la simulazione di un numero N di campioni per ogni valore proposto θ caricherebbe il calcolatore

Riquadro 3: Markov Chain Monte Carlo ABC (MCMC ABC)

Input: Valore iniziale θ_1, simulatore $\mathbb{S}\{y|\theta\}$, proposta $\omega(\cdot|\theta_1;\epsilon)$, soglia ε, dati reali y_{oss}, lunghezza C della catena,
Output: Catena di Markov $\theta_1, \theta_2, \ldots, \theta_C$

for i in 2 to C **do**
  Proponi $\theta^* \sim \omega(.|\theta_{i-1};\epsilon)$
  Simula $y^* \sim \mathbb{S}\{y|\theta^*\}$
  if $||y^* - y_{oss}|| \leq \varepsilon$ **then**
    Calcola $\tau = \frac{\pi(\theta^*)}{\pi(\theta_{i-1})}$
    Estrai un numero $u \sim U(0,1)$
    if $u \leq \tau$ **then**
      $\theta_i = \theta^*$ {Accetta θ^*}
    else
      $\theta_i = \theta_{i-1}$ {Rifiuta θ^*, rimani su θ_{i-1}}
    end if
  else
    $\theta_i = \theta_{i-1}$ {Rifiuta perché $||y^* - y_{oss}|| > \varepsilon$}
  end if
end for
return $\theta_1, \theta_2, \ldots, \theta_C$

di un lavoro computazionale troppo impegnativo, per cui il più delle volte ci si limita ad $N = 1$ come suggerito in Bornn et al. (2017) ed un valore proposto θ^* viene accettato con probabilità $\tau = \frac{\pi(\theta^*)}{\pi(\theta)}$ solo se è rispettata la condizione $||y^* - y_{oss}|| \leq \varepsilon$ dove y^* indica il vettore di osservazioni generato dal simulatore $\mathbb{S}\{y|\theta^*\}$ che riceve in ingresso il valore proposto θ^*. La procedura è riportata in dettaglio nel Riquadro 3

3.4 Limiti dei metodi bayesiani

3.4.1 Maledizione della dimensionalità

I metodi **ABC** riescono a rimediare l'assenza di una funzione di verosimiglianza cercando valori del parametro θ che generano osservazioni artificiali vicine a quelle osservate y_{oss}. Questa strategia si rivela però poco efficace quando dimensione e numerosità dei dati crescono. Sia $y \in \mathbb{R}^{N \times K}$ il vettore di dati osservati e si supponga per adesso $d = 1$. La norma euclidea è una quantità crescente rispetto ad N in quanto somma di quantità tutte positive:

$$||y_{oss} - y^*|| = \sqrt{\sum_{i=1}^{N} (y_i^* - y_{oss_i})^2}. \tag{3.5}$$

Ne risulta che è difficile adeguare ε in maniera adeguata al crescere di N. Per questo motivo più che considerare i dati y_{oss} ed y^* così come osservati, si tenta una riduzione a statistiche ovvero a statistiche $S(.)$ che trasformino il generico vettore di

dati y da un punto in $\mathbb{R}^N$ ad un punto in uno spazio minore. Così facendo il criterio $||y^* - y_{oss}|| \leq \varepsilon$ è sostituito con $||S(y^*) - S(y_{oss})|| \leq \varepsilon$ (Turner et al. 2012). Se il modello fosse noto le scelte ideali di $S(.)$ sarebbero le statistiche sufficienti di cui si è discusso ne paragrafo 2.1.1. Tuttavia nei modelli impliciti è difficile individuare trasformazioni $S(.)$ che trattengano, riducendone la dimensione, tutta l'informazione contenuta nei dati y riguardo il parametro di interesse θ. Come indicato in Vihola et al. (2022) scelte ragionevoli di $S(.)$ sono percentili, coefficienti di autoregressione o loro trasformazioni.

Meno si può fare quando a crescere è K. È noto che la distanza fra punti in spazi di grande dimensione sia tale da invalidare i metodi di stima di densità fondati sulla prossimità dei punti osservati (Bishop 2021). Infatti in uno spazio di grande dimensione i punti tendono ad essere tutti distribuiti sulla frontiera dello spazio e molto distanti fra loro. Si consideri l'ipercubo di dimensione $d = 10$ con lato di lunghezza unitaria in cui sono uniformemente distribuiti N punti, si può dimostrare che per catturare l'1% delle N osservazioni è necessario estrarre un cubo di lato 0.63 dall'ipercubo unitario (Hastie et al. 2008). La lunghezza del lato dell'ipercubo da estrarre per catturare l'1% delle osservazioni aumenta al crescere della dimensione K. Allo stesso modo, se N sono i punti distribuiti uniformemente in una sfera di dimensione d con centro nell'origine, allora la distanza mediana dall'origine al punto più vicino è data da:

$$\left(1 - 2^{-\frac{1}{N}}\right)^{1/K},$$

ed è quindi pari circa a 0.52 se $K = 10$ ed $N = 500$, a 0.52 se $K = 11$ ed $N = 1000$ ed a 0.93 se $K = 100$ ed $N = 1000$.

Il fenomeno di allontanamento dei punti al crescere di K è noto come *Maledizione della dimensionalità* e la conseguenza sui metodi ABC è immediata: se tutti i punti nello spazio $\mathbb{R}^{N \times K}$, con K grande, sono distanti, allora è improbabile osservare una quantità $||y^* - y_{oss}||$ minore di un valore positivo ε arbitrariamente piccolo. Quando la dimensione dei dati cresce, molti dei valori proposti θ non riescono a produrre dati sintetici y^* sufficientemente vicini a quelli reali e ciò determina un aumento poco controllabile del numero di simulazioni necessarie a collezionare sufficienti realizzazione della distribuzione a posteriori empirica approssimata $\hat{\pi}_\varepsilon(\theta|y_{oss})$.

Alla luce di quanto discusso in questo paragrafo si riportano i risultati di un piccolo esperimento. Si è eseguito un campionamento di tipo **MCMC ABC** al fine di ottenere una versione empirica delle distribuzioni a posteriori approssimate $\pi_\epsilon(\theta|y_{oss})$ e $\pi_\varepsilon(\theta|S(y_{oss}))$. Il simulatore utilizzato è definito nel listato in Figura 3.1, mentre la riduzione è operata da $S(y) = (P_{25,y}, P_{50,y}, P_{75,y})$ dove $P_{u,y}$ indica l' *u-esimo* percentile

```python
import torch
def simulator(theta, nsim=15):
    mean = torch.tensor([[theta[0], theta[1]]])
    SIGMA = torch.tensor([[theta[2], 0.5], [0.5, theta[3]]])
    distribution = MultivariateNormal(loc=mean,
    covariance_matrix=SIGMA)
    out = distribution.sample((nsim,))
    return out.squeeze(1)
```

Figura 3.1: Funzione definita in `Python` che riceve in ingresso il valore del parametro θ, costruisce la distribuzione di probabilità e restituisce `nsim` osservazioni provenienti dalla normale parametrizzata da θ

di y, infine $\varepsilon = 0.05$. I dati y_{oss} sono stati generati da $\theta_0 = (-0.5, 3.0, 2.0, 0.9)$ parametro di interesse.

In figura 3.2 sono riportati i grafici riassuntivi delle distribuzioni empiriche $\hat{\pi}_\varepsilon(\theta|y_{oss}$ e $\hat{\pi}_\varepsilon(\theta|S(y_{oss}))$ mentre alcune statistiche descrittive sono disponibili in Appendice alla Tabella B.1 e alla Tabella B.2.

Entrambi i metodi producono distribuzioni a posteriori su un supporto adeguato al problema in esame. Come ci si attendeva la riduzione di dimensione operata da $S(y)$ rende più facile discriminare punti vicini da punti lontani, ciò si traduce in posteriori con meno variabilità. Entrambi gli algoritmi eseguono un numero di proposte pari a 30000 a cui sono da sottrarre le prime 5000 osservazioni. Lo scarto iniziale serve ad eliminare la dipendenza della distribuzione ottenuta dal valore iniziale della catena. Il tasso di accettazione per il campionamento ABC basato sui dati originali y è dello $0,09\%$ mentre per quello che utilizza una riduzione a statistiche dei dati dello $0,2\%$. I tassi di accettazione così bassi danno un'idea della quantità di valori proposti scartati e quindi del notevole carico computazionale.

3.4.2 Assenza di un meccanismo di aggiornamento

Stabilito che si può parzialmente contrastare l'aumento della numerosità N ricercando opportune trasformazioni dei dati $S(y)$, si segnala come altro limite dei metodi ABC l'impossibilità di aggiornare le distribuzioni empiriche $\hat{\pi}_\varepsilon(\theta|y_{oss})$ o $\hat{\pi}_\varepsilon(\theta|s(y_{oss}))$ ottenute all'occorrenza di nuove osservazioni reali. Se y_{oss} è la realizzazione di un vettore aleatorio formato da n osservazioni, che si possono assumere indipendenti ed identicamente distribuite, la distanza euclidea da un generico vettore simulato y^* è data dall'Equazione 3.5.

Qualora, dopo l'esecuzione di uno qualsiasi degli algoritmi di ABC visti in precedenza, si rendesse disponibile una nuova osservazione reale $y_{oss_{n+1}}$, questa non sa-

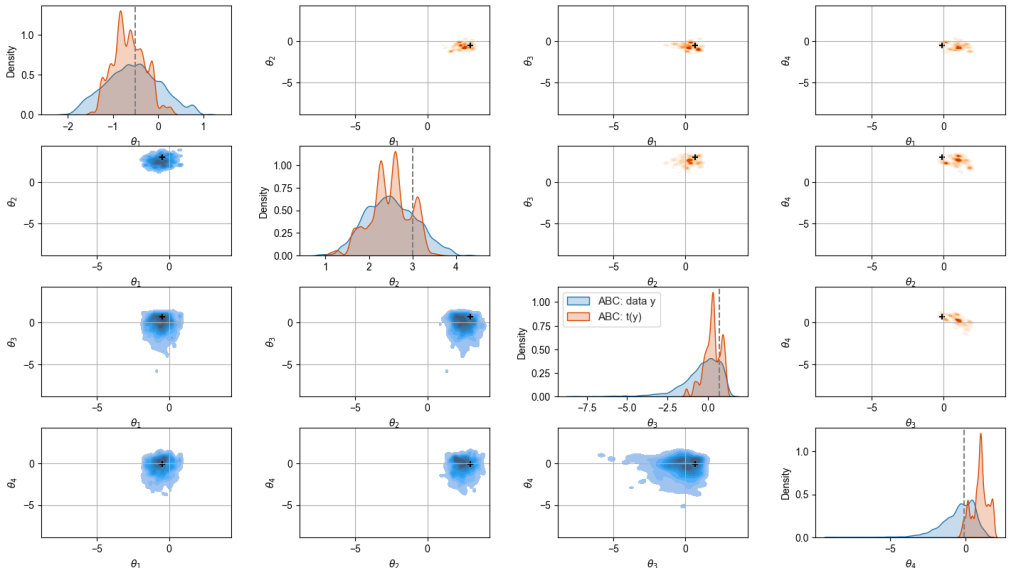

Figura 3.2: ABC blu ed ABC con riduzione della dimensionalità operata da $s(y)$ arancione. Viene riportata una rappresentazione delle distribuzioni marginali e bivariate delle componenti di θ. L'uso delle statistiche $s(y)$ riduce la dimensione, quindi la distanza misurata delle osservazioni e rende più semplice discriminare valori plausibili o meno di θ come generatori di y_{oss}

rebbe direttamente utilizzabile per aggiornare $\hat{\pi}_\varepsilon(\theta|y_{oss})$. Affinché la distribuzione a posteriori giovi della maggior disponibilità di dati empirici è necessario ripetere tutto il procedimento da capo.

Per concludere, l'impossibilità di aggiornare la distribuzione a posteriori $\pi_\varepsilon(\theta|y_{oss})$ qualora si disponga di nuove osservazioni reali e la difficoltà di gestire il crescere della distanza dei punti all'aumento delle dimensioni delle osservazioni rende i metodi ABC applicabili a contesti specifici di dimensioni limitate.

Capitolo 4

Modelli surrogato

4.1 Definizione ed utilità di un surrogato

Oltre ai limiti tecnici discussi nel capitolo precedente, ai metodi ABC si può contestare il procedere, nella ricerca della distribuzione a posteriori, per prova ed errore. Nel discriminare valori di θ accettati e non il legame fra y e parametro non è indagato, ci si limita a selezionare i valori proposti di θ che generano dati artificiali ritenuti vicini ai dati osservati sotto l'ipotesi che dati simili siano generati da θ simili. Tutte le simulazioni rifiutate vengono scartate ed ignorate, si tratta di informazione utile che non viene utilizzata.

Nel corso di questo capitolo verrà illustrata la procedura di costruzione di un modello surrogato. Tale procedura usa il simulatore per produrre coppie artificiali (y^*, θ^*) da utilizzare al fine di apprendere in che modo θ determina y e derivarne un modello esplicito e trattabile $q_w(y|\theta)$, detto surrogato, con cui sostituire la verosimiglianza nelle procedure inferenziali. La scrittura $q_w(y|\theta)$ segnala che il surrogato dovrà essere regolato da parametri esterni w da selezionare adeguatamente, e deve inoltre restituire una misura quantitativa della plausibilità di θ come parametro generatore di y. La figura 4.1 offre uno schema riassuntivo della procedura di inferenza realizzata con ABC e della procedura realizzata con la costruzione di un surrogato. Nel secondo caso il confronto con i risultati sperimentali y_{oss} è rimandato: prima si costruisce il surrogato $q_w(y|\theta)$ e solo in seguito le procedure inferenziali proseguono considerando i dati reali y_{oss} come se il modello fosse noto. Il secondo passo si traduce nella sostituzione della verosimiglianza con il surrogato $q_w(y|\theta)$, in un contesto bayesiano invece di:

$$\pi(\theta|y_{oss}) \propto \pi(\theta)p(y_{oss}|\theta) \quad \text{si avrà} \quad \pi(\theta|y_{oss}) \propto \pi(\theta)q_w(y_{oss}|\theta)$$

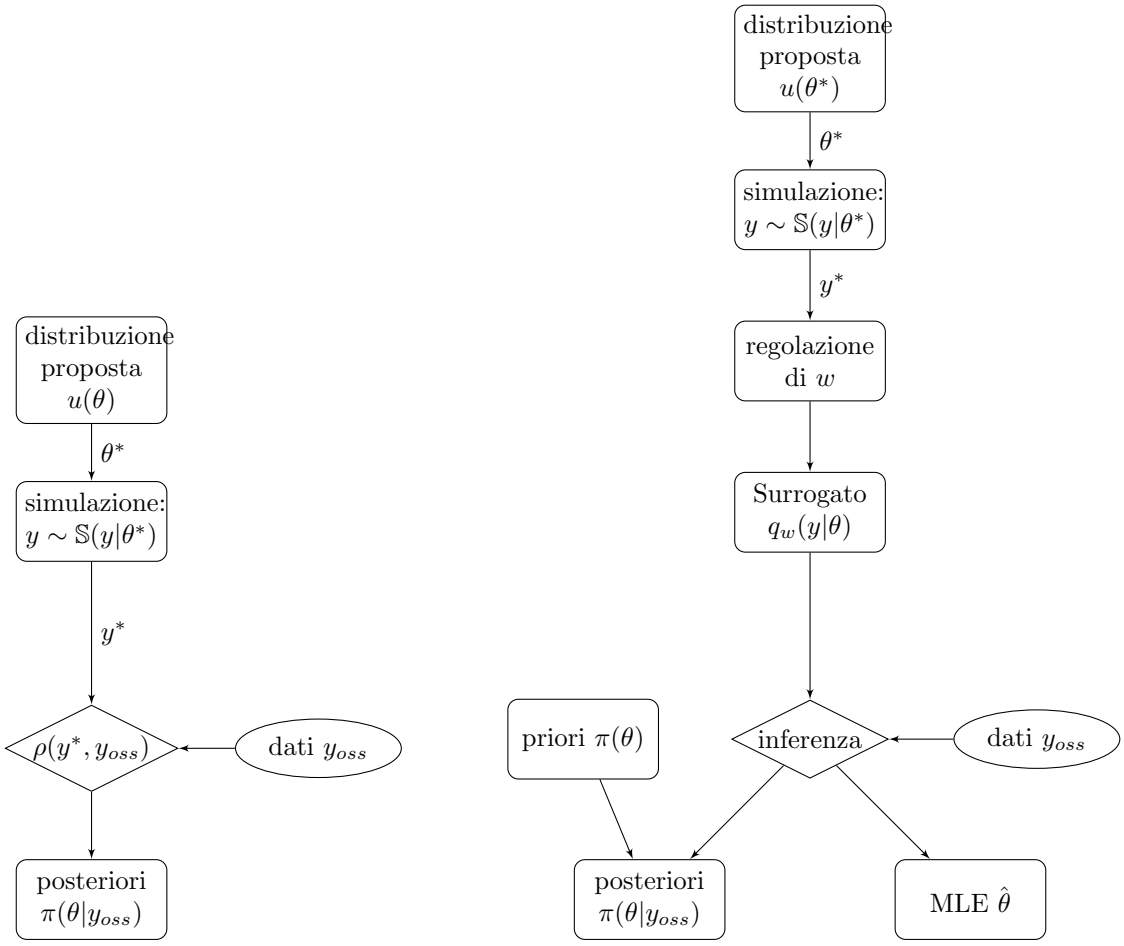

Figura 4.1: Confronto delle procedure di inferenza ottenute tramite ABC (a sinistra) e modello surrogato (a destra). Nel primo caso il confronto con i dati osservati y_{oss} è immediato mentre nel secondo caso si passa prima per la costruzione del surrogato.

In termini intuitivi il surrogato $q_w(y|\theta)$ disegna una mappa $\theta \xrightarrow{q_w} y$ in cui ricercare $\theta_0 \longrightarrow y_{oss}$. Prima di di passare alla costruzione effettiva della mappa è necessario introdurre gli strumenti utili a tale scopi.

4.2 Costruzione di un modello surrogato

4.2.1 Reti neurali per la regressione

Per indagare il legame fra θ ed y si può ricorrere a tecniche di regressione da adeguare opportunamente ai nostri scopi. In generale, in un modello di regressione si ricerca la funzione f della variabile indipendente, nel nostro caso θ, che meglio prevede il valore della risposta y. In base alle forme che si ritiene possa assumere f si distinguono diversi modelli di regressione; f può essere lineare, quadratica, a gradini e così via. Nel seguito

di questa tesi il modello di regressione di cui si discuterà è la rete neurale: si tratta di uno strumento particolarmente flessibile che consente ad f di approssimare qualsiasi funzione continua con un margine di errore arbitrariamente piccolo a patto che la struttura della rete sia sufficientemente complessa (Hornik et al. 1989). In sintesi, f è una composizione di funzioni di θ (detto input) organizzata ed ottenuta come descritto di seguito, dove è riportato il caso generale di $dim(\theta) = p$ e $dim(y) = K$:

1. Si calcolano M combinazioni lineari delle variabili di input cui si applica una funzione $g()$ non lineare $Z_1, Z_2, \dots Z_M$, ottenendo le variabili latenti:

$$Z_m = g(\alpha_{0m} + \alpha_m^T \theta) \quad m = 1, \dots, M.$$

   Scelte frequenti di g sono la funzione sigmoide, $g = \frac{\exp(Z)}{1+\exp(Z)}$, oppure la funzione tangente iperbolica $g = \frac{\exp(Z)-\exp(-Z)}{\exp(Z)+\exp(-Z)}$, funzioni di questo tipo servono ad introdurre non linearità nel modello.

2. La variabile risposta Y è modellata come funzione h di una combinazione lineare delle variabili latenti, h può essere o meno lineare, nel caso sia lineare:

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \dots K$$

3. Alla fine il $k - esimo$ output risulta quindi modellato da una composizione di funzioni di θ

$$f_k(\theta) = h_k(T) = h_k(g(\alpha^T \theta))$$

Fra θ ed y possono essere presenti più strati nascosti, una rappresentazione generale di rete neurale è in Figura 4.2

4.2.2 Stima dei parametri

Stimare i parametri di una rete neurale significa individuare i valori $\hat{\alpha}$ e $\hat{\beta}$, che minimizzano una data funzione di perdita. Dato che nei problemi di regressione l'obiettivo è la previsione di y, bisogna determinare α e β in modo che i valori previsti $\hat{y_i} = \hat{f}(\theta_i)$ siano il più vicino possibile alle vere realizzazioni y_i. Il criterio di stima è quindi l'errore quadratico medio (EQM), se si hanno a disposizione n osservazioni (θ_i, y_i), $i = 1, 2, \dots, n$, risulta:

$$EQM(\alpha, \beta) = \frac{1}{2n} \sum_{k=1}^{K} \sum_{i=1}^{n} (y_{ik} - f_k(\theta_i; \alpha, \beta))^2.$$

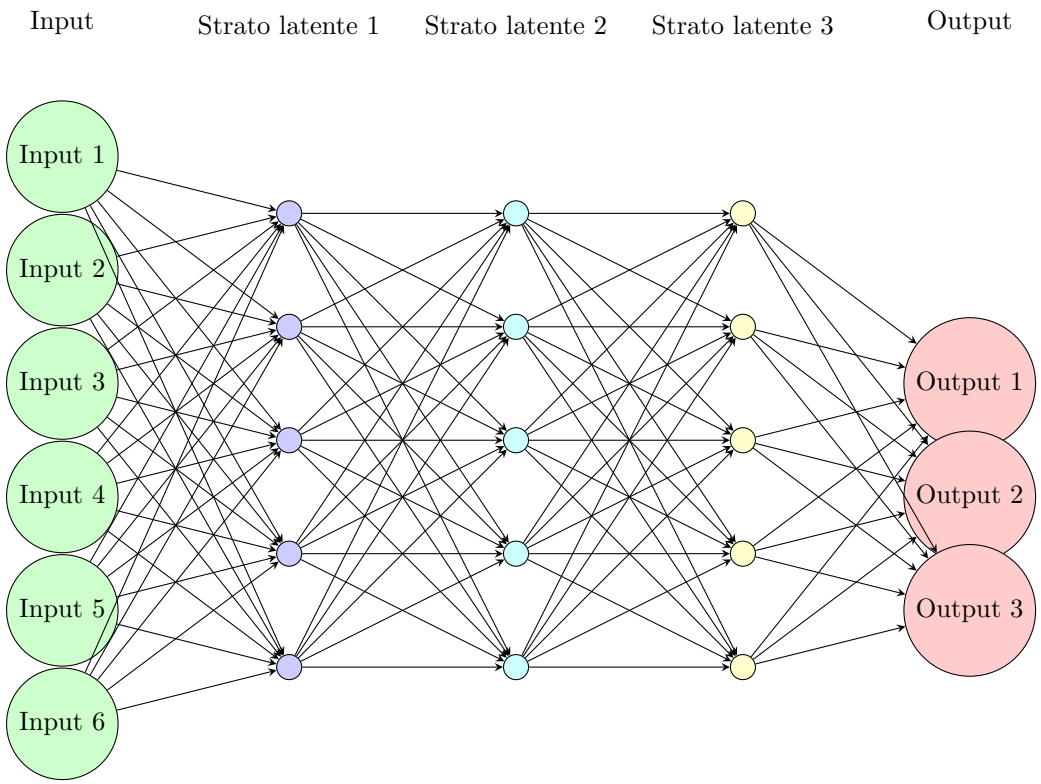

Figura 4.2: Rete neurale con 6 nodi di input, 3 strati latenti e 3 nodi di output. La complessità della struttura consente di catturare strutture più complicate presenti nei dati ma comporta un aumento del numero di *pesi* da stimare.

I parametri da stimare sono quindi:

$$\{\alpha_{0m}, \alpha_m; m = 1, 2, \dots M\} \text{ per un totale di } M(p+1) \text{ elementi}$$
$$\{\beta_{0k}; k = 1, 2, \dots K\} \text{ per un totale di } K(M+1) \text{ parametri}$$

La ricerca del minimo della funzione di perdita è eseguita con un algoritmo di massima discesa; è quindi necessario il calcolo delle derivate rispetto ad α e β. Data la natura delle funzioni considerate, utilizzando la regola della catena per la derivazione di funzioni composte, otteniamo (Hastie et al. 2008):

$$\frac{\partial EQM_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(\theta_i))h'_k(z_i\beta_k)z_{mi}$$
$$\frac{\partial EQM_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(\theta_i))h'_k(z_i\beta_k)\beta_{km}g'(\theta_i\alpha_m)\theta_{il}$$

Il calcolo di queste derivate è particolarmente agevole tramite differenziazione automatica in modalità backward, mentre la regola di aggiornamento all'iterazione corrente r (questa volta indicati tra parentesi nell'apice) di un algoritmo di massima discesa, come visto nel capitolo 1, è della forma:

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \eta \sum_{i=1}^{N} \frac{\partial EQM_i}{\partial \beta_{km}^{(r)}}$$
$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \eta \sum_{i=1}^{N} \frac{\partial EQM_i}{\partial \alpha_{ml}^{(r)}}$$

L'algoritmo di ottimizzazione può essere **ADAM** in modo da adattare ad ogni iterazione r il tasso di aggiornamento η. I parametri da stimare di una rete sono comunemente chiamati pesi, nel seguito ci si riferirà a loro con w (*weigths*) senza più necessità di ricorrere alla notazione $\alpha_0, \alpha, \beta_0, \beta$.

4.2.3 Limiti dei metodi di regressione

Stabilito come funzionano le reti neurali si può procedere con l'adattare questi strumenti alla costruzione di un modello surrogato, ovvero ad apprendere e fare emergere dalle simulazioni il modo in cui θ genera y, che non vuol dire prevedere. O meglio, non sempre. Si consideri il problema presente in Bishop (1994), rappresentato in figura 4.3 dove i valori θ sono estratti da una distribuzione uniforme $U(0, 1)$ ed i dati y sono

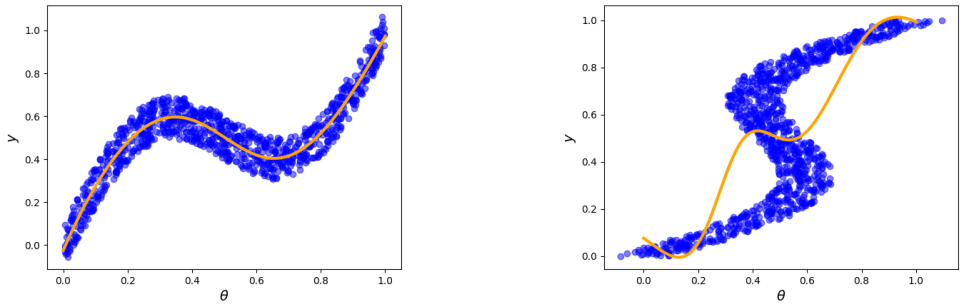

Figura 4.3: Valori previsti (arancione) dalla rete neurale di regressione con 1 strato nascosto a 5 nodi stimata minimizzando EQM sul problema diretto (destra) e sul problema inverso (sinistra)

ottenuti con la trasformazione:

$$y = \theta + 0.3\sin(2\pi\theta) + \zeta, \text{ dove } \zeta \sim U(-0.1, 0.1), \tag{4.1}$$

si consideri anche il problema inverso ottenuto scambiando i valori di θ con quelli di y. In entrambi i casi è stata eseguita la regressione di y su θ. I risultati non buoni sul problema inverso sono visibili in Figura 4.3. La criticità del metodo nel problema inverso sta nella multimodalità della distribuzione condizionata $y|\theta$. Nei contesti di regressione infatti siamo alla ricerca di una stima puntuale del valore previsto $\hat{y_i}$ come funzione di θ_i. Cerchiamo pertanto i pesi w che restituiscono le previsioni più vicine ai valori effettivamente osservati minimizzando l'EQM. Supponendo di avere infinite osservazioni possiamo scrivere:

$$\begin{aligned} EQM &= \lim_{n\to\infty} \frac{1}{2n} \sum_{i=1}^{n} \sum_{k=1}^{K} [f_k(\theta_i; w) - y_{ik}]^2 \\ &= \frac{1}{2} \sum_{k=1}^{K} \iint [f_k[\theta; w) - y_k]^2 p(y, \theta)\, d\theta\, dy \end{aligned}$$

Il minimo di EQM è la media condizionata di y rispetto a θ indicata con $\bar{y}|\theta$. Un'altra osservazione importante è che l'errore quadratico medio può essere scomposto in:

$$EQM = \frac{1}{2} \sum_{k=1}^{K} \int [f_k(\theta_i; w) - \bar{y}_k|\theta]^2 p(\theta)\, d\theta \tag{4.2}$$

$$+ \frac{1}{2} \sum_{k=1}^{K} \int [\bar{y}_k^2|\theta - \bar{y}_k|\theta]^2 p(\theta)\, d\theta \tag{4.3}$$

Si osserva che l'errore quadratico medio dipende dai pesi w solo per il termine in 4.2 che è minimo in corrispondenza di $\bar{y}|\theta$ (Bishop 2021), mentre il termine 4.3 restituisce la varianza media dei dati target y_i intorno al loro valore medio condizionale. Si vuole sottolineare quindi che stimando i parametri di una rete neurale con criterio EQM si stanno stimando due quantità:

- La media condizionata come funzione della variabile dipendente θ regolata da w.
- La media della varianza dei dati attorno la media condizionata stimata.

Si sta quindi implicitamente assumendo che la distribuzione $Y|\theta$ abbia una media che cambia con θ_i ed una varianza costante. Per concludere, nel caso in cui la distribuzione condizionata $Y|\theta$ sia bimodale (o multimodale) la funzione di regressione individua come valore previsto di $\hat{y}$ un punto intermedio alle mode, verosimilmente in corrispondenza di una zona della distribuzione con poca densità di probabilità. Di conseguenza il valore previsto per $\hat{y}$ fa riferimento ad una realizzazione in verità rara ed improbabile del processo.

4.2.4 Reti neurali per densità mistura

Si intuisce quindi che il surrogato deve avere come priorità l'individuazione della distribuzione di probabilità $Y|\theta$ più coerente con i dati osservati $y_i|\theta$ e non invece, come nei problemi di regressione, la maggior riduzione possibile di una misura di errore fra dati osservati e previsti dal modello. Nel caso $Y|\theta$ sia unimodale le due cose coincidono, ma non sempre. Occorre nel nostro caso concentrarsi sul primo obiettivo.

Nel seguito si assumerà quindi che il surrogato $q_w(y|\theta)$ sia una distribuzione mistura di G componenti gaussiane i cui parametri saranno opportune funzioni di θ. Una mistura di gaussiane con un numero sufficiente di componenti è in grado di approssimare qualsiasi distribuzione di probabilità (Ferguson 1983), è quindi la struttura ideale da dare a $Y|\theta$.

$$q_w(y|\theta) \sim \sum_{j=1}^{G} \lambda_j(\theta; w) N\big(\mu_j(\theta; w), \sigma_j(\theta; w)\big)$$

Per comodità di notazione i parametri di mistura sono organizzati in 3 vettori:

- coefficienti di mistura: $b^{\lambda} = (\lambda_1, \lambda_2, \dots \lambda_G)$
- medie di mistura: $b^{\mu} = (\mu_1, \mu_2, \dots \mu_G)$
- parametri di varianza: $b^{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_G)$

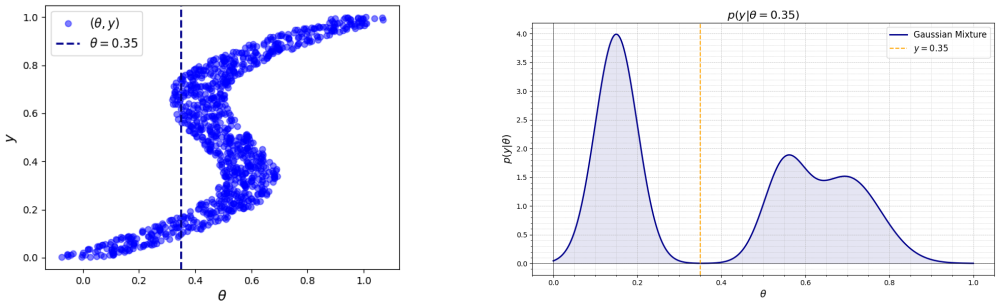

Figura 4.4: Mistura che descrive la distribuzione dei dati $y|\theta = 0.35$. In corrispondenza di $\theta = 0.35$ y il valore previsto dal modello di regressione è circa 0.3, un valore con probabilità quasi nulla.

Per trasformare θ nei parametri di mistura si userà viene usata una rete neurale che prende in ingresso il parametro θ e lo trasformerà nelle componenti di mistura. Per ogni θ dato in ingresso alla rete, questa restituirà 3 vettori di G componenti. I parametri della mistura sono sottoposti a dei vincoli e affinché siano rispettati si devono applicare dovute trasformazioni diverse ai vettori che contengono i parametri di varianza e e coefficienti di mistura (Bishop 1994), in particolare:

- Il vincolo di somma ad 1 dei coefficienti di mistura $\sum_{j=1}^{G} \lambda_j = 1$ è soddisfatto applicando la trasformazione *softmax* ai relativi output.

$$\lambda_l = \frac{\exp\left(b_l^{\lambda}\right)}{\sum_{j=1}^{G} \exp\left(b_j^{\lambda}\right)}$$

- Il vincolo di stretta positività delle varianze è soddisfatto applicando la funzione esponenziale

$$\sigma_l = \exp\left(b_l^{\sigma}\right)$$

Una efficace rappresentazione della rete neurale è in figura 4.5. Il simulatore $\mathbb{S}\{y|\theta\}$ entra in gioco nella fase di ottimizzazione della rete, questo dovrà fornire i dati sintetici (θ^*, y^*) da utilizzare per selezionare i pesi w. I pesi devono regolare le trasformazioni applicate a θ di modo che, mediamente, il surrogato $q_w(y|\theta)$ sia il più vicino possibile alla versione empirica simulata di $y|\theta$. Per questo motivo la funzione da minimizzare è il negativo della log-verosimiglianza di mistura valutata nei dati sintetici, se si dispone

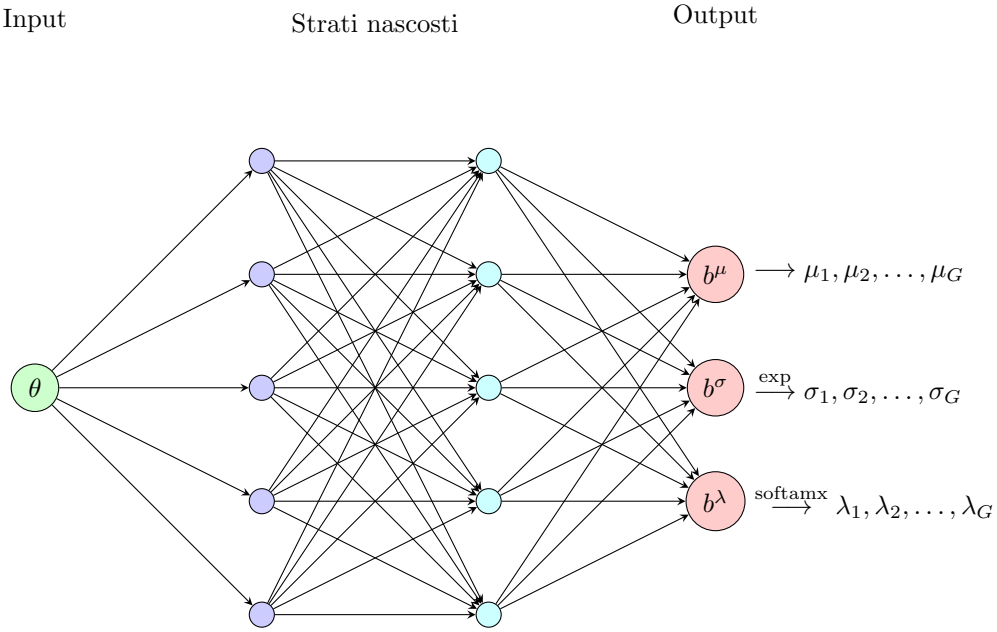

Figura 4.5: Rete neurale per stimare i parametri di una mistura di gaussiane con input θ di dimensione 1. Ciascun output $b^\mu, b^\sigma, b^\lambda$ subisce una ulteriore trasfromazione per assicurare il rispetto dei vincoli di somma ad 1 dei pesi di mistura $\sum_{j=1}^{G} \lambda_j$ e stretta positività dei paraemtri di varianza $\sigma_1, \sigma_2, \ldots, \sigma_G$

di N dati artificiali (y, θ) :

$$-\log L(w, y, \theta) = -\sum_{i=1}^{N} \log \left(\sum_{j=1}^{G} \lambda_j(w;\theta) \frac{1}{\sqrt{2\pi\sigma_j^2(w;\theta)}} \exp\left(-\frac{(y_i - \mu_j(\mu;\theta))^2}{2\sigma_j^2(w;\theta)} \right) \right) \tag{4.4}$$

Terminata la fase di ottimizzazione, individuati quindi i pesi ottimi $\hat{w}$, si possono sostituire w ed y con $\hat{w}$ ed y_{oss} nell'Equazione 4.4 e reperire le regioni di Θ a più alta densità o massimizzando rispetto a θ o ottenendo una versione empirica della verosimiglianza con metodi MCMC.

4.3 Esplorazione dello spazio parametrico

Nella costruzione del modello surrogato il simulatore fornisce dati (θ, y) da utilizzare per addestrare la rete neurale ed individuare i pesi w che trasformano θ nella mistura che meglio descrive $y|\theta$. Nei casi generali lo spazio parametrico Θ può essere vasto, rendendosi necessaria una selezione accurata dei valori θ in corrispondenza di cui simulare. Se $\Theta = \mathbb{R}$ o $\Theta = \mathbb{R}^n$ dovranno necessariamente esistere delle zone inesplorate, la capacità di simulare dati è una risorsa finita ed occorre guidare il simulatore verso le aree più vicine a θ_0 valore generatore di y_{oss} La produzione di dati con cui addestrare la rete e

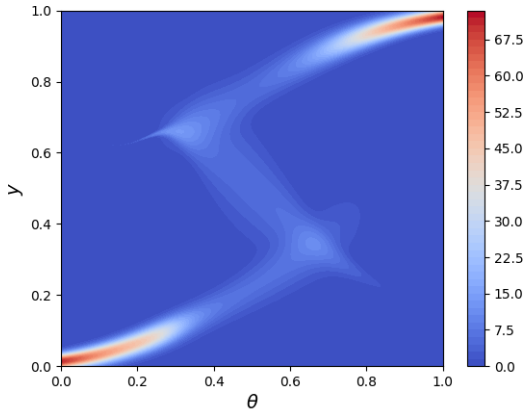

Figura 4.6: Mappa di densità del surrogato costruita per i dati del problema inverso in Figura 4.3, come illustrato in questo Capitolo. Il surrogato cattura perfettamente la distribuzione $y|\theta$

Riquadro 4: Verosimiglianza neurale sequenziale

Input: dati y_{oss}, rete $q_w(y \mid \theta)$, numero di iterazioni R, simulazioni per iterazione N, simulatore $\mathbb{S}\{y|\theta\}$
Output: surrogato $q_w(y_{oss}|\theta)$

Inizializza $q_{w_1}(y|\theta) = u(\theta)$ e $D = \emptyset$ $\{w_1$ indica pesi inizializzati casualmente$\}$
for $r = 1$ **to** R **do**
  for $n = 1$ **to** N **do**
    Campiona $\theta_n \sim u(\theta)$ con MCMC
    Simula $y_n \sim \mathbb{S}\{(y\theta_n)\}$
    Aggiungi (θ_n, y_n) a D
  end for
  Stima nuovamente $q_w(y \mid \theta)$ su D e aggiorna $u(\theta) \propto q_w(y_{oss}|\theta)u(\theta)$
end for
return $q_w(y_oss|\theta)$

disegnare la mappa è da svolgersi un po' per volta avvicinandosi progressivamente alle regioni di Θ di maggior rilevanza.

L' unico modo per farlo è come proposto in Papamakarios et al. (2019) lasciare che il surrogato $q_w(y_{oss}|\theta)$ guidi il campionamento dallo spazio Θ. Dopo un primo campionamento da una distribuzione proposta generale $u(\theta)$, la seguente generazione di corrispondenti dati simulati e quindi di una stima iniziale di $\hat{w}$, si può aggiornare la proposta con la conoscenza di Θ estratta dalla simulazione precedente campionando quindi in un secondo momento da $u(\theta)q_w(y_{oss}|\theta)$. E procedere così per un numero definito di volte. La procedura di questo tipo, descritta in dettaglio nel riquadro 4 prende il nome di Verosimiglianza neurale sequenziale.

Capitolo 5

Casi di studio

5.1 Premesse

Nel corso di questo capitolo verranno illustrati i risultati ottenuti applicando le metodologie discusse in questa tesi che operano inferenza utilizzando la capacità di simulare dati artificiale in sostituzione alla verosimiglianza. I modelli sono noti ma le procedure inferenziali eseguite sono proprie dell'inferenza senza versomiglianza per cui i dati verranno trattati come se il modello fosse ignoto ed intrattabile. Tutti i casi analizzati hanno notazione e caratteristiche comuni discusse di seguito:

- k indica la dimensione della singola osservazione y_i
- p è la dimensione del parametro di interesse θ, nonché numero di input della rete neurale per la costruzione dei surrogati.
- n il numero di osservazioni di cui sono composti $y_o ss$ ed i campioni simulati da $\mathbb{S}\{y|\theta\}$
- I simulatori utilizzati sono quelli già presenti e definiti negli ambienti di calcolo e dipendono solo da θ. In particolare si è ricorso a:
  - di `R: rgamma; rmvnorm` (libreria `rmvtnorm`);`runif0` *runif.*
  - di `Python`: `MultivariateNormal; Gamma` (modulo `distribution` della libreria `torch`)
- Con il termine versione empirica della verosimiglianza si intende quella ottenuta con algoritmi di simulazione MCMC, ciò vale sia nei casi di modello noto sia nei casi di modello surrogato. La scelta è motivata dal fatto che per le verosimiglianza

surrogato, dato l'elevato numero di pesi w coinvolti, risulta complicato ottenere una forma analitica. Nello studio di simulazione invece la costruzione degli intervalli con la statistica test log-rapporto di verosimiglianza è realizzata ricorrendo alla forma analitica della verosimiglianza.

5.2 Caso unidimensionale monoparametrico

Il simulatore utilizzato è `rgamma`. L'obiettivo è valutare la sostituibilità del surrogato alla funzione di verosimiglianza nel caso di inferenza sul solo parametro di forma θ_0. Lo studio di simulazione è stato così organizzato:

1. Estrazione casuale di 500 valori generatori θ_0 da `runif(500 0.2,10)`
2. Campionamento di 20 osservazioni, che interpreteranno il ruolo di y_{oss}, da ciascuno dei 500 valori generatori al punto 1. Estrazione eseguita con `rgamma(20,`θ_0`,1)`.
3. Costruzione dell'intervallo di confidenza con livello di fiducia 0.95 basato sulla statistica test *log*-rapporto di verosimiglianza per ognuno dei 500 campioni. Il modello in questo punto è supposto noto.
4. Costruzione del surrogato $q_w(y_{oss}|\theta)$ e collezione di una sua versione empirica $\hat{q}_w(y_{oss}|\theta)$.
5. Controllo della condizione: moda del surrogato interna alla regione di confidenza individuata al punto 3.

Il surrogato è costruito come illustrato nel Capitolo 4, in particolare si è costruita una rete per densità mistura con uno strato nascosto a 5 nodi e funzione di attivazione tangente iperbolica. Il numero di componenti G della mistura è pari a 3. Per $u(\theta)$ si è scelta una distribuzione gamma con parametro di forma 3 e parametro di scala 0.1, si noti la varianza e la possibile generazione di valori da $u(\theta)$ anche molto distanti da θ_0: tale scelta è motivata dal voler fare emergere le capacità di convergenza del metodo. Lo schema seguito è quello delle verosimiglianze neurali sequenziali illustrate nel Riquadro 4. Si sono quindi campionati all'inizio 60 valori θ dalla distribuzione proposta iniziale $u(\theta)$ da aggiornare di volta in volta con la collezione delle simulazioni. Dopo il campionamento iniziale di θ, la simulazione ed una prima stima del surrogato si continua alternando nuove fasi di campionamento di θ, simulazione da $\mathbb{S}\{y|\theta\}$, ristima del surrogato e aggiornamento della proposta $u(\theta)q_w(y_{oss}|\theta)$. In tutto, questo procedimento viene reiterato per 12 volte. Per ogni campionamento successivo al primo i valori θ estratti sono 20 e non 60. Il totale di campioni simulati per la stima di un surrogato

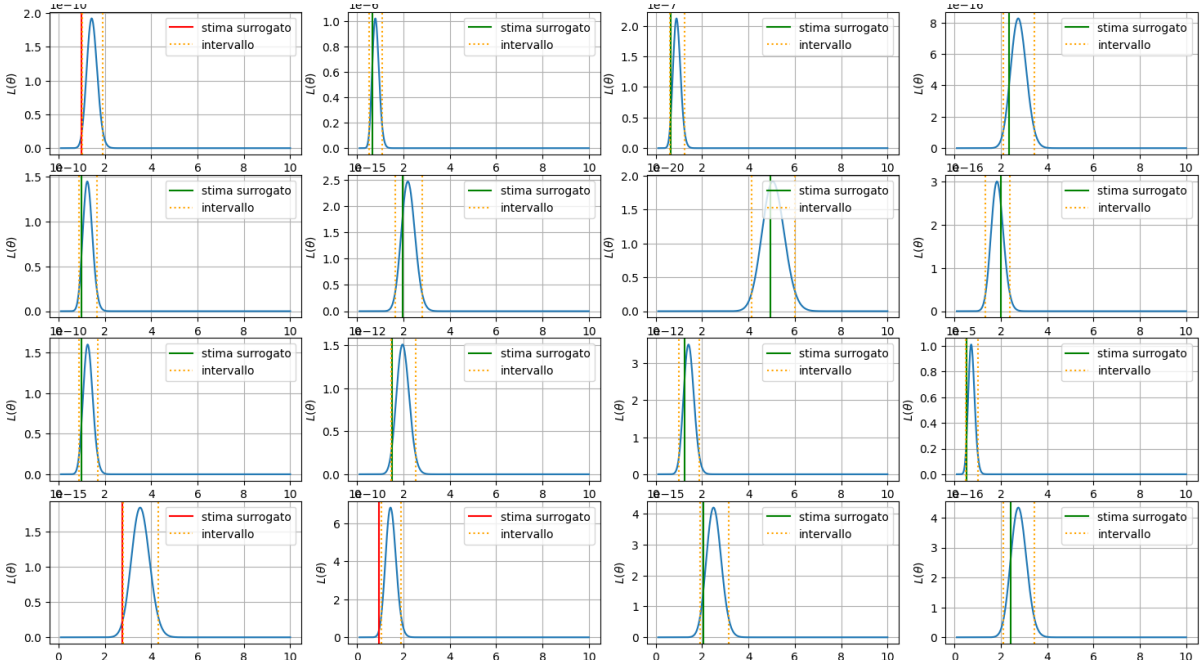

Figura 5.1: Rappresentazione grafica di alcuni intervalli dello studio di simulazione

è di 200. I risultati mostrano che la stima puntuale del surrogato è vicina a quella di massima verosimiglianza, infatti l' $86,4\%$ delle volte il massimo del surrogato cade nella regione di confidenza individuata dalla verosimiglianza.

Il totale di parametri coinvolti è 40 e la funzione di perdita è:

$$\log L(w,y) = \sum_{i=1}^{N} \log \left(\sum_{j=1}^{G} \lambda_j(w;\theta^*) \frac{1}{\sqrt{2\pi\sigma_j^2(w;\theta^*)}} \exp\left(-\frac{(y_i^* - \mu_j(\mu;\theta^*))^2}{2\sigma_j^2(w;\theta^*)} \right) \right)$$

La ridotta dimensione del problema consente di utilizzare `optim`. Nei casi di dimensioni maggiori, per via dell'elevato tempo di calcolo necessario e le risorse computazionali richieste, non è stato fatto uno studio di simulazione nel senso che non si è testata l'efficacia del metodo un numero di volte sufficiente per affermazioni rigorose . Si sono comunque realizzati dei tentativi di implementazione provando il metodo su un solo campione osservato.

5.3 Caso bidimensionale multiparametrico

Qui si è supposto il modello ignoto essere una normale bivariata con parametri di covarianza noti pari a 0.5. Il simulatore utilizzato è `MultivariateNormal` del modulo `distriubtion`. L'obiettivo è l'inferenza sui parametri di media e varianza. La distribuzione proposta $u(\theta)$ da cui estrarre i θ^* in corrispondenza di cui simulare i dati y^* è una

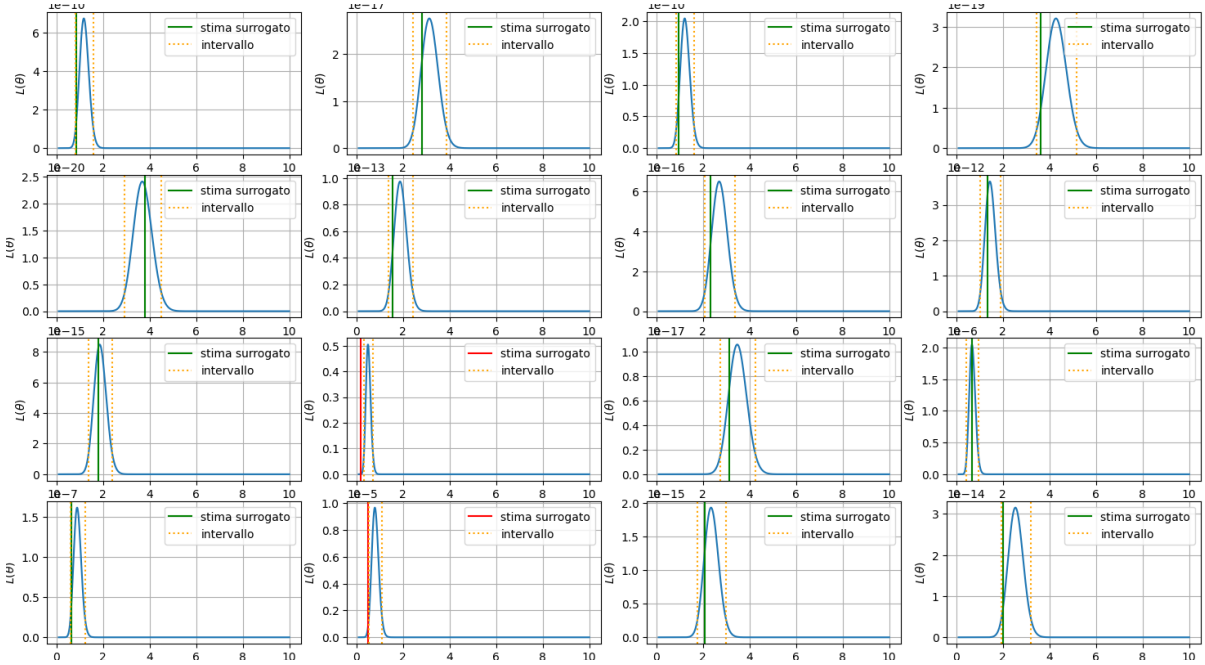

Figura 5.2: Rappresentazione grafica di alcuni intervalli dello studio di simulazione

distribuzione Uniforme di dimensione 4 con estremi $-4, 4$. $u(\theta) \sim U_2(-4, 4)$, i parametri di varianza sono in scala logaritmica.

Viene stimato un surrogato con $G = 4$ componenti di mistura. Un' ipotesi ragionevole è che le componenti di mistura abbiano tutte la stessa matrice di varianza, in modo da stimarne una sola, si tratta di un buon compromesso fra flessibilità del surrogato e quantità di parametri da stimare. La rete ha due strati latenti, ciascuno di 90 nodi con funzioni di attivazione rispettivamente di tangente iperbolica e $\text{ReLU}(x) = \max(0, x)$.L'output ha dimensione $k \times G + G + k^2 = 8 + 4 + 4 = 16$:2 medie per ciascuna componente μ_j, 4 pesi di mistura π_j, una matrice diagonale di varianza Σ. La funzione obiettivo da massimizzare è quindi:

$$L_{\theta^*}(w) = \prod_{i=1}^{n} \left\{ \sum_{j=1}^{G} \lambda_{\theta_j^*}(w)(2\pi)^{-\frac{k}{2}} |\Sigma_{\theta^*}(w)|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(y_i^* - \mu_{\theta^* j}(w))^T \Sigma_{\theta}^{-1}(w)(y_i^* - \mu_{\theta^* j}(w)) \right) \right\}$$

oppure la sua equivalente trasformazione logaritmica:

$$l_{\theta^*}(w) = \sum_{i=1}^{n} \log \left\{ \sum_{j=1}^{G} \lambda_{\theta^* j}(w)(2\pi)^{-\frac{k}{2}} |\Sigma_{\theta^*}(w)|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(y_i^* - \mu_{\theta^* j}(w))^T \Sigma_{\theta^*}^{-1}(w)(y_i^* - \mu_{\theta^* j}(w)) \right) \right\}$$

I pesi w da ottimizzare sono più di 9000, un compito di questa portata è impraticabile

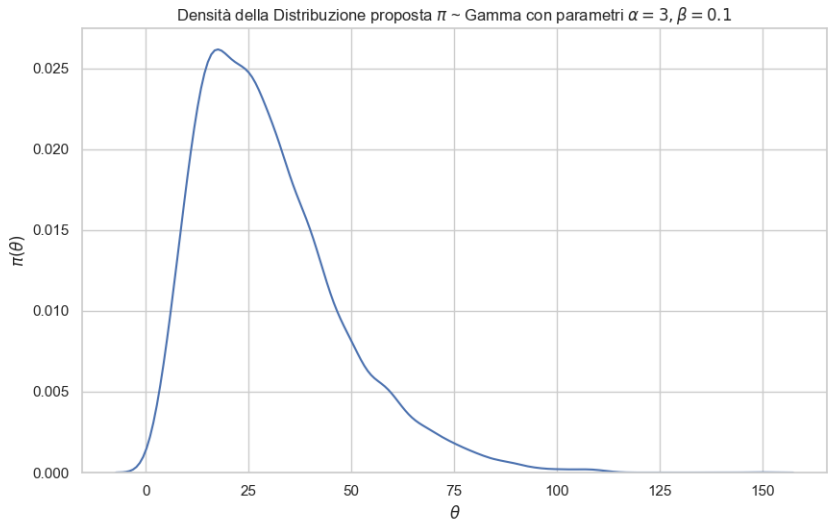

Figura 5.3: Densità della distribuzione proposta iniziale $u(\theta)$: $Gamma(3, 0.1)$

con la discesa quasi Newton e calcolo delle derivate tramite differenze finite. L'unico modo per ricercare il massimo è calcolare le derivate con differenziazione automatica e procedere alla ricerca del massimo con discesa del gradiente minimizzando $-l_\theta(w)$. La scrittura y^* e θ^* ricorda che i dati utilizzati per stimare i pesi della rete sono i dati artificiali provenienti dal simulatore. Un punto delicato della stima dei pesi di una rete neurale, soprattutto quando la struttura è molto complicata, è il rischio del sovra adattamento. Questo si verifica quando un modello si adatta eccessivamente ai dati di stima, catturando oltre al segnale anche le oscillazione casuali dei dati. La conseguenza è una performance eccellente sui dati di stima ma poco generalizzabile. Per evitare questo rischio si è usata la tecnica della frenata anticipata[1]. Si divide quindi l'insieme di dati in insieme di stima ed insieme di verifica. Ad ogni iterazione dell'algoritmo di ottimizzazione di massima discesa si valuta il valore della funzione di perdita sull'insieme dei dati di verifica. La fase di ottimizzazione si arresta se la funzione obiettivo sui dati di verifica non migliora o peggiora per un numero consecutivo di iterazioni di discesa, detto *patience*. Nelle implementazioni questo numero è stato posto uguale a 10. Si tratta di una scelta empirica da valutare in base alle specifiche de problema e con qualche tentativo. Una rappresentazione grafica dei risultati è in figura 5.5, mentre le Tabelle 5.1 e 5.2 riportano alcune statistiche descrittive delle versioni empiriche della verosimiglianza e della verosimiglianza surrogato. Emerge che i risultati ottenuti con il surrogato siano in linea con quelli di verosimiglianza per i parametri media, mentre ci sono difficoltà per i parametri di varianza, per la stima puntuale del surrogato in corrispondenza di θ_{03}

[1] Frenata ricorda l'esempio della discesa della palla nel Capitolo 1 per i metodi di massima discesa

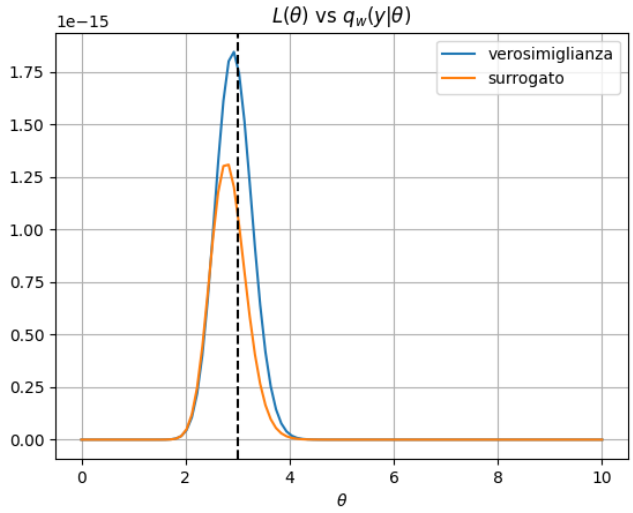

Figura 5.4: Confronto funzione di verosimiglianza e funzione di verosimiglianza surrogato per uno dei campioni dello studio di simulazione, la linea verticale tratteggiata in nero indica θ_0

si registra una distorsione stimata pari a 2. La difficoltà nella stima dei parametri di varianza si estende ai casi con dimensione maggiore del prossimo paragrafo.

5.4 Caso 8-dimensionale multiparametrico

Quando la dimensione dei dati k cresce i metodi di ABC perdono efficacia per via della maledizione della dimensionalità ed il loro carico computazionale aumenta, mettendone in discussione la fattibilità pratica. Grazie alla differenziazione automatica rimane invece aperta la possibilità di addestrare una rete neurale per definire un surrogato. Si sono simulati dati y_{oss} con $k = 8$ da una normale multivariata di dimensione 8 parametrizzata come di seguito:

Tabella 5.1: Tabella descrittiva verosimiglianza empirica con modello supposto noto per caso 8-dimensionale con 16 parametri da stimare. Gli intervalli sono intervalli quantile di livello 0.95 delle distribuzioni ottenute con MCMC.

Parametro	θ_1	θ_2	θ_3	θ_4
θ_0	-0.5	3.0	0.693	-0.1053
intervallo	$(-1.1225; 1.223)$	$(2.5591; 3.614)$	$(1.495; 3.118)$	$(0.67; 1.56)$
Media emp.	0.11	3.107	2.15	0.995
sd. emp.	0.97	0.891	0.4359	0.2255
moda emp.	0.022	3.11	1.921	0.8961
min	-1.7356	2.0345	1.238	0.5606
max	1.759	4.342	3.8265	0.5606

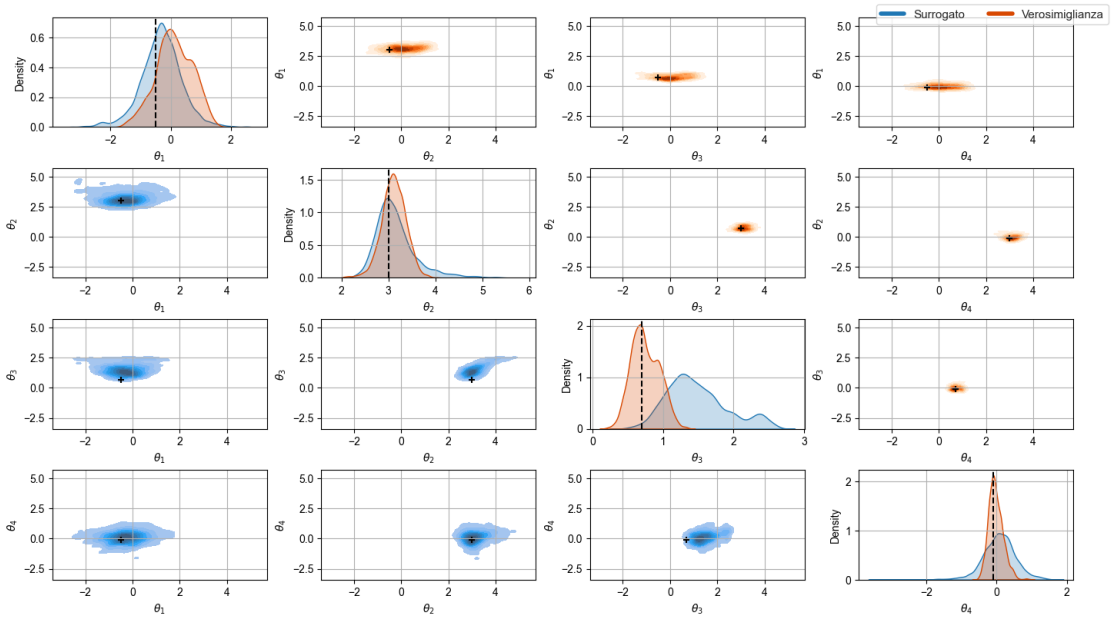

Figura 5.5: Confronto verosimiglianza empirica del modello noto (arancione) e verosimiglianza empirica del modello surrogato (blu).

- medie $\mu_0 = (-0.5, 2.4, 2.0, 0.9, 0.5, 2.5, -1.0, -1.0)$,
- elementi fuori dalla diagonale Σ_0 tutti uguali a 0.3, supposti noti,
- elementi sulla diagonale di Σ pari a

$$diag(\Sigma_0) = (1.0, 2.0, 0.4, 1.0, 2.0, 1.5, 0.3, 1.4).$$

Dunque $\theta_0 = (\mu_0, \log(\Sigma_0))$. Le dimensioni degli spazi coinvolti rendono necessario l'impiego intensivo del simulatore. Per la costruzione del surrogato sono state effettuate in totale 100000 chiamate del simulatore, la distribuzione proposta $u(\theta)$ da cui campionare inizialmente i θ^* è una uniforme $U_{16}(-4, 4)$ ed il modello $q_w(y|\theta)$ è una mistura di normali a $G = 8$ componenti, il numero di pesi coinvolti è elevato: più di 100000, in reti così

Tabella 5.2: statistiche descrittive versione empirica della verosimiglianza per il modello bidimensionale multiparametrico.

Parametro	θ_1	θ_2	θ_3	θ_4
θ_0	-0.5	3.0	0.693	-0.1053
intervallo	$(-1.777; 1.0023)$	$(2.51; 4.40)$	$(2.221; 11.4995)$	$(0.38, 2.725)$
Media emp.	-0.31	3.155	4.8545	1.191
sd. emp.	0.686	0.455	2.4153	0.57
moda emp.	-0.23	3.01	3.50	0.96
min	-1.7356	2.0345	1.238	0.5606
max	-3.260	2.00	1.5869	0.03342

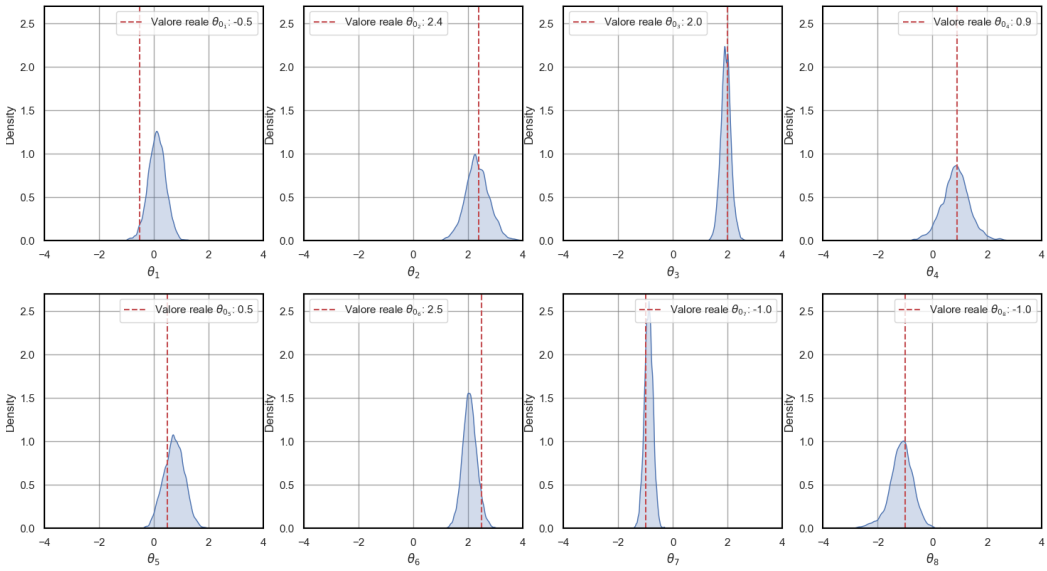

Figura 5.6: Versione empirica marginale della verosimiglianza per i parametri di media μ_0. In rosso è indicata la posizione del vero valore del parametro generatore.

complesse una della funzioni di attivazione sempre presente è $ReLU(w) = max(0, w)$ che rende nulli i pesi poco utili alleggerendo il processo di stima. La Tabella 5.3 riporta le statistiche riassuntive della verosimiglianza empirica ottenuta mentre nei grafici in Figura 5.6 e Figura 5.7 sono rappresentate marginalmente le densità delle versioni empiriche delle verosimiglianze ottenute rispettivamente per i parametri di media e i parametri di varianza. Osserviamo qui uno schema simile a quello del caso bidimensionale: la stima dei parametri di media da parte del surrogato non presenta grandi deviazioni dal vero valore μ_0. Mentre per alcuni dei parametri di varianza il surrogato presente una distorsione significativa che per l'11esima componente del parametro è stimata pari a 1.

Per concludere, nel caso univariato è più semplice gestire le varie fasi di costruzione del surrogato, a partire dalla scelta del numero di componenti di mistura G che può avvenire sulla base di valutazioni grafiche, non disponibili nei casi di dimensioni maggiori. Allo stesso modo il legame fra θ ed y può assumere forme meno complicate, ciò si traduce in reti neurali con struttura più semplice e gestibile. In ultimo, gli algoritmi MCMC sono più efficienti nel caso univariato è più semplice la selezione del parametro di varianza della camminata aleatoria e così via. Più difficili si fanno le scelte di questi iperparametri del surrogato in dimensioni grandi. Studi ulteriori potranno essere eseguiti per indagare criteri di selezione di questi iperparametri nei contesi di grandi dimensioni.

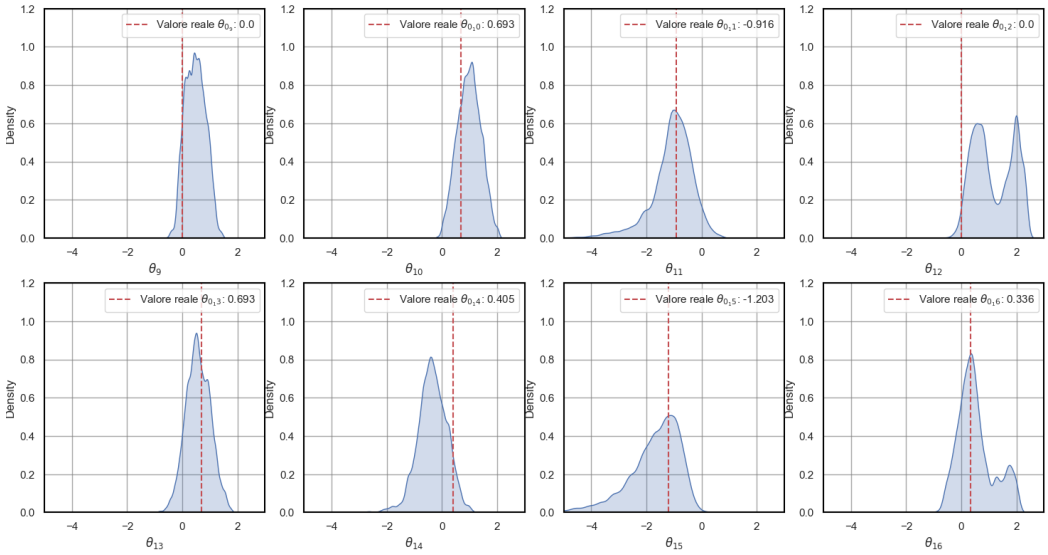

Figura 5.7: Versione empirica della verosimiglianza per i parametri di varianza $diag(\Sigma_0)$. Le linea in rosso indica il vero valore del parametro generatore θ_0.

Tabella 5.3: Tabella descrittiva della versione empirica della verosimiglianza surrogato per modello 8-dimensionale con 16 parametri da stimare.

Parametro	θ_1	θ_2	θ_3	θ_4
θ_0	-0.5	2.4	2.0	0.9
intervallo	$(-0.8679; 0.9992)$	$(1.4605; 3.7132)$	$(1.5127; 2.5488)$	$(0.2457; 1.7432)$
Media emp.	0.0404	2.5645	2.0234	0.9829
sd. emp.	0.4693	0.5791	0.2601	0.3750
moda emp.	0.0888	2.5077	1.6538	1.1129
min	-1.9023	0.6487	1.1402	-0.7209
max	2.7855	5.6040	3.0375	2.5040

Parametro	θ_5	θ_6	θ_7	θ_8
θ_0	0.50	2.50	-1.00	-1.00
Intervallo	$(0.0228; 1.1024)$	$(1.8670; 3.2343)$	$(-1.813; 0.8125)$	$(-2.11; -0.53)$
Media	0.5831	2.5618	-1.2996	-1.3429
Dev. std.	0.2678	0.3794	0.2597	0.3996
moda emp.	0.5165	2.5986	-1.3843	-1.4103
min	-0.5255	1.1799	-2.2441	-3.0196
max	1.6529	5.6151	-0.3386	0.3005

Parametro	θ_9	θ_{10}	θ_{11}	θ_{12}
θ_0	0.000	0.693	-0.916	0.000
Intervallo	$(-0.4282; 1.8703)$	$(0.3087; 1.7594)$	$(-4.1206; 0.4886)$	$(-0.3686; 1.4972)$
Media emp.	0.6913	1.0159	-1.6239	0.4932
sd emp.	0.5823	0.3994	1.2663	0.4677
moda emp.	0.2015	1.0327	-3.1429	0.3716
min	-3.6857	-0.2678	-9.3283	-2.8849
max	2.1232	2.0132	0.7935	1.9601

Parametro	θ_{13}	θ_{14}	θ_{15}	θ_{16}
θ_0	0.0000000	0.4054651	-1.2039728	0.0000
Intervallo	$(-1.95785; 1.0626)$	$(-2.67836; 1.2533)$	$(-4.1814; 0.1492)$	$(-0.8999; 1.2157)$
Media emp.	-0.3416	-0.6776	-1.7566	0.1884
sd emp.	0.7886	0.9837	1.1915	0.5751
moda emp.	0.1411	-0.1642	-3.3069	0.2250
min	-6.6356	-9.0523	-10.1112	-6.6125
max	1.5947	1.7347	0.4789	1.6244

Conclusioni

In questa tesi sono stati esaminati e analizzati i principali metodi di inferenza per modelli impliciti ai simulatori, con un'attenzione particolare agli approcci ABC e ai modelli surrogato. I metodi della classe ABC, sebbene dimostrino una grande potenzialità, sono ostacolati dalla maledizione della dimensionalità e dalla loro incapacità di essere aggiornati in maniera semplice, rendendoli quindi meno pratici per problemi ad alta dimensionalità. D'altra parte, i modelli surrogato offrono un'applicabilità più generale ma richiedono una selezione meticolosa degli iperparametri di regolazione, che deve essere adeguata alla specificità di ogni problema.

Nel Capitolo 5 è stata studiata la possibilità di sostituire la funzione di verosimiglianza con un modello surrogato nel contesto di un problema monoparametrico univariato. I risultati ottenuti nei casi a più dimensioni indicano che questa sostituibilità potrebbe essere estesa anche a problemi di dimensioni maggiori. Tuttavia, è necessario condurre ulteriori esperimenti per confermare questa ipotesi. La fattibilità pratica della costruzione di un modello surrogato nel caso di $dim(\theta) > 1$ e dimensione K delle osservazioni maggiore di 1 è legata agli strumenti di ottimizzazione utilizzati nella fase d stima dei pesi della rete neurale. In questi scenari, per garantire al surrogato la flessibilità necessaria a catturare adeguatamente il legame fra dati y e parametro θ contenuto nel simulatore, la struttura della rete neurale non può essere banale, ma deve contenere un numero sufficiente di strati e nodi nascosti e parametrizzare inoltre un numero di componenti di mistura adeguato. In problemi di ottimizzazione di questa portata l'algoritmo di discesa non può essere di tipo Newtoniano: la valutazione della matrice Hessiana o di una sua approssimazione può essere proibitiva. In aggiunta, come visto nel Capitolo 1, il mancato verificarsi della condizione di definita positività di H può condurre gli algoritmi Newtoniani su direzioni di non discesa.

Di conseguenza occorre utilizzare metodi di massima discesa come ADAM. La valutazione efficiente del valore della derivata in un punto effettuata dalla differenziazione automatica è lo strumento ideale per questo tipo di metodi. È importante sottolineare

ancora che la stabilità numerica portata dalla differenziazione automatica rende i metodi di massima discesa più robusti rispetto i metodi Newtoniani.

In definitiva si conclude che le potenzialità dei modelli surrogato di raggiungere obiettivi inferenziali nei modelli impliciti può esprimersi a pieno solo se accompagnata da strumenti di calcolo precisi e flessibili come la differenziazione automatica.

Appendice A

Codice Python sviluppato

A.1 Rete neurale di regressione e rete neurale per densità mistura utilizzate nei paragrafi 4.2.3 e 4.2.4

```
import torch
import math
import numpy as np
from torch import nn
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torch.distributions import Gamma, Normal, Uniform
import torch.optim as optim

torch.manual_seed(3298)
n = 1000

dim_input = 1      # dimensione dell'input
dim_output = 1     # dimensione dell'output
dim_hidden = 50    # numero di nodi nello strato latente
k = 3              # Numero componenti d mistura

# Creo i dati
```

```
19
20  x_train = Uniform(0.0,1.0).sample((n,1))  # vettore di 1000 osservazioni
21                                            # da uniforme(0,1)
22
23  noise = Uniform(-0.1,0.1).sample((n,1)) #
24  y_train = x_train + 0.3*torch.sin(2*x_train*torch.tensor(
25  [math.pi])) + noise
26  x_test  = torch.linspace(0,1, steps= n).reshape((n,dim_input))
27
28  # Costrusico la classe della rete
29  class NeuralNetwork(nn.Module):
30      def __init__(self):
31          super().__init__()
32          self.hidden_layers = nn.Sequential(
33              nn.Linear(dim_input, dim_hidden),
34              nn.Tanh()
35          )
36          self.output = nn.Sequential(
37
38              nn.Linear(dim_hidden, dim_output)
39          )
40
41      def forward(self, x):
42          hid = self.hidden_layers(x)
43          out = self.output(hid)
44          return out
45  # funzione obiettivo: in questo caso MSE
46  loss_fn = nn.MSELoss()
47
48  model = NeuralNetwork()
49  # Algoritmo di Ottimizzazione ADAM
50  optimizer = optim.Adam(model.parameters(), lr = 0.001)
51  n_epochs = 6000
52
53  for e in range(n_epochs):
54      optimizer.zero_grad()    # azzera i gradienti
```

```
55      outputs = model(x_train) # calcola i valori previsti
56      loss = loss_fn(outputs, y_train)        # Calcola il valore della
57      loss.backward()                         # Calcola i gradienti
58
59      # Stampa il valore di loss ogni 100 iterazioni/poche
60      if (e) % 100 == 0:
61          print(f'{e}, Loss: {loss.item():.4f}')
62
63  predictions = model(x_test) # valori predetti dal modello
64
65  # grafico dei punti x,y e dei valori previsti dalla rete 'addestrata'
66  plt.plot(x_train, y_train, 'go', alpha = 0.5)
67  plt.plot(x_test, predictions.detach().numpy(),'r', linewidth = 3.0)
68  plt.show()
69
70  # Creo la classe che inizializza la rete:
71  class MixtureDensityNetwork(nn.Module):
72      def __init__(self):
73          super().__init__()
74          self.hidden_layers = nn.Sequential(
75          # Lo strato latente è definito come prima
76              nn.Linear(dim_input, dim_hidden),
77              nn.Tanh()
78          )
79          self.mean_output = nn.Sequential(
80          nn.Linear(dim_hidden, k)
81          )
82
83          self.dev_output = nn.Sequential(
84
85              nn.Linear(dim_hidden, k)
86          )
87          self.pesi_output = nn.Sequential(
88          # Per i pesi occorre che la loro somma
89          # sia 1 quindi quindi si applica la funzione
```

```
90          # Softmax: Softamx(u) = exp(u)/sum(exp(u))
91              nn.Linear(dim_hidden, k),
92              nn.Softmax()
93          )
94      def forward(self, x): # trasorma x nei 9 output
95          hid  = self.hidden_layers(x)
96          mean = self.mean_output(hid)
97          sigma = torch.exp(self.dev_output(hid))
98          pesi = self.pesi_output(hid)
99          return mean, sigma, pesi
100
101 # IN questo caso la funzione da ottimizzare è
102 # la log-verosimiglianza della mistura.
103 # In altre parole si devono trovare i PESI W1 W2
104 # tali per cui y sia trasformato nei parametri
105 # della mistura più vicina ai dati osservati x/y
106 def loss_mixture(x, mu, sigma, pi):
107     z_score_sq = torch.pow((x - mu) /sigma, 2)
108     normal_loglik = ( # calcolo le log_densità nelle k componenti
109      -0.5*z_score_sq - torch.log(sigma) + torch.log(pi)
110     )
111     loglik = torch.logsumexp((normal_loglik), dim = 1) # sommo le densità
112     # delle componoenti
113     # log(sum(exp( log_densità delle componenti )))
114     return  -sum(loglik) # sommo per i dati osservati
115
116
117 MDN = MixtureDensityNetwork() # costruisco la rete
118
119 optimizer = optim.Adam(MDN.parameters(), lr = 0.008)
120 # ottimizzo come prima
121 n_epochs = 5000
122 for e in range(n_epochs):
123     optimizer.zero_grad()
124     out = MDN(y_train)
```

```
    loss2 = loss_mixture(x = x_train, mu = out[0],
            sigma = out[1], pi = out[2])  # Calcola la perdita
    loss2.backward()
    optimizer.step() # Calcola i gradienti

    # Stampa il risultato ogni 100 epoche
    if (e) % 100 == 0:
        print(f'{e}, Loss: {loss2.item():.4f}')
# costrusico un insieme di punti su cui valutare
#la densità condizionata per vedere se si adatta ai dati

y_test = torch.linspace(0,1,steps = 500).reshape((500,1))
predictions = MDN(y_test) # do in input alla rete y_test
                          #in modo da ottenere i
                          # parametri della mistura

mu = predictions[0] # medie
sd = predictions[1] # deviazioni standard
alpha = predictions[2] # pesi

def mixture_density(dati, medie, sd, pesi):
    # funzione che valuta la desnità di una
    # mistura in un punto dati i paraemtri
    z = torch.pow((dati - medie)/sd, 2)
    out = torch.exp(-.5*z)/sd*pesi
    return torch.sum(out,1)

# costruisco una matrice dove salvare i valori delle
# densità per una griglia di punti y x
Z_matrix= torch.zeros(500,500)
# riempo la matrice
for j in range(500):
    Z_matrix[:,j] = mixture_density(dati  = y_test,
                                    medie = mu[j,:],
                                    sd    = sd[j,:],
```

```
                                    pesi  = alpha[j,:])

# Faccio un contour
Z_np =  Z_matrix.detach().numpy()
x_seq_np = np.linspace(0,1,500)
y_seq_np = np.linspace(0,1,500)
plt.contourf(x_seq_np, y_seq_np, Z_np, levels=50, cmap='coolwarm')
plt.colorbar()  # Aggiunge una barra dei colori per indicare i valori
plt.xlabel('$\\theta$', fontsize=14, fontweight='bold', color='black')
plt.ylabel('$y$', fontsize=14, fontweight='bold', color='black')
plt.show()
```

A.2 Caso bivariato multiparametrico

```

torch.manual_seed(394)

nsim = 8
def simulator(theta, nsim = nsim):
    mean = torch.tensor([[theta[0], theta[1]]])
    SIGMA = torch.tensor([[1.0, theta[2]],
                          [theta[2], 1.0]])
    distribution = MultivariateNormal(loc=mean, covariance_matrix=SIGMA)
    out = distribution.sample((nsim,))
    return out.squeeze(1)

theta_true = torch.tensor([0.5, 1.0, 0.5])
x_oss = simulator(theta=theta_true, nsim = nsim) # tenosore 10,1,2
n_star = 5000
theta_star = torch.cat((Uniform(-3,3).sample((n_star,2)),
                        #  Uniform(0.5,2.0).sample((n_star,2)),
                         Uniform(-0.5,0.8).sample((n_star,1))), dim = 1)

x_star = torch.empty((n_star,nsim,2))
for j in range(theta_star.shape[0]):
```

```
    x_star[j,:,:] = simulator(theta = theta_star[j,:], nsim = nsim)


class MixtureDensityNetwork(nn.Module):
        def __init__(self, dim_in, dim_hid, k, dim_out):
            super().__init__()
            self.dim_in = dim_in
            self.dim_hid = dim_hid
            self.dim_out = dim_out
            self.k = k
            self.hidden_layers = nn.Sequential(
                nn.Linear(dim_in,dim_hid),
                nn.ReLU(),
                nn.Linear(dim_hid,dim_hid),
                nn.Tanh()
                )
            self.mean_out = nn.Sequential(
                nn.Linear(dim_hid, dim_out * k)
                )
            self.dev_out = nn.Sequential(
                nn.Linear(dim_hid, dim_out)
                )
            self.pi_out = nn.Sequential(
                nn.Linear(dim_hid, k),
                nn.Softmax(dim = 0)
                )

        def forward(self,x):
            hid = self.hidden_layers(x)
            mu  = self.mean_out(hid).view(-1,self.k, self.dim_out)
            sd = torch.exp(self.dev_out(hid))
            sigma = torch.diag_embed(sd)
            pi  = self.pi_out(hid)
            return mu, sigma , pi

```

```
        def loss(self, input, target):
            mu, sigma, pi = self.forward(input)
            mean = mu.unsqueeze(2)
            alpha = pi.unsqueeze(2)
            d = target.shape[2]
            det_S = torch.det(sigma)
            S_inv = torch.inverse(sigma)
            const = -0.5*d*torch.log(torch.tensor([2*math.pi]))                    (-0
            const = const.unsqueeze(1)
            likelihood = torch.empty(input.shape[0],
            target.shape[1], self.k)
            for c in range(self.k):
                mu_c = mean[:,c,:]
                Z_t = torch.transpose(target-mu_c,1,2)
                ZS = torch.matmul((target-mu_c),S_inv)
                ZSZt = torch.diagonal(torch.matmul(ZS,Z_t),
                offset=0, dim1=-2, dim2=-1)
                likelihood[:,:,c] =  const-0.5*ZSZt +
                                        torch.log(alpha[:,c])
            log_lik = torch.logsumexp(likelihood,dim = 2)
            return  -torch.sum(log_lik,1)

        def point_density_evaluation(self, input, target = x_oss):
            mu, sigma, pi = self.forward(input)
            mean = mu.squeeze(0)
            alpha = pi.squeeze(0)
            d = target.shape[1]
            det_S = torch.det(sigma)
            S_inv = torch.inverse(sigma)
            const = -0.5*d*torch.log(torch.tensor(
            [2*math.pi]))- 0.5*torch.log(det_S)
            likelihood = torch.empty(target.shape[0],self.k)
            for c in range(self.k):
                mu_c = mean[c,:]
                Z_t = torch.transpose(target-mu_c,0,1)
```

```
                ZS = torch.matmul((target-mu_c),S_inv)
                ZSZt = torch.diagonal(torch.matmul(ZS,Z_t),
                offset=0, dim1=-2, dim2=-1)
                likelihood[:,c] =  const-0.5*ZSZt + torch.log(alpha[c])
            log_lik = torch.logsumexp(likelihood,dim = 1)
            return  torch.sum(log_lik)



MDN = MixtureDensityNetwork(dim_in = 3, dim_out = 2,
dim_hid = 90, k = 8)
out = MDN(theta_star)
out_single = MDN(theta_star[1,:])

optimizer = optim.Adam(MDN.parameters(), lr = 0.005)
n_epochs = 2000
for e in range(n_epochs):
    optimizer.zero_grad()
    out = MDN(theta_star)
    loss = MDN.loss(theta_star, x_star).mean()
    loss.backward()
    optimizer.step()
    if (e) % 10 == 0:
        print(f'{e}, Loss: {loss.item():.4f}')

torch.manual_seed(83)
theta_0 = theta_star[torch.randint(0,n_star,size = (1,))]
theta_0 = theta_0.squeeze(0)
print(theta_0)
def MCMC(n_values, eps, th_0, burnin = 500):
    acc = 0
    theta_values = torch.empty(n_values, th_0.shape[0])
    theta_values[0,:] = th_0
    rw_dist = Uniform(-eps,eps)
```

```
    U_dist  = Uniform(0,1)
    th = th_0
    for j in range(1,n_values):
        th_star = th + rw_dist.sample((3,))
        if th_star[2].abs() > 1:
            alpha = 0
        else:
            alpha =  torch.exp(MDN.point_density_evaluation(
            input = th_star)
                                - MDN.point_density_evaluation(th))
        if U_dist.sample((1,)) < alpha:
            th = th_star
            acc += 1
        theta_values[j,:] = th
        acc_rate = acc/j
        if j % 100 == 0:
            print(f'iteration: {j}, accpetance_rate = {acc_rate}')
    return theta_values[(n_values - burnin):,:], acc/n_values



versione_empirica_surrogato = MCMC(n_values = 10000,
eps = 0.3, th_0 = theta_0, burnin = 4500)
```

A.3 Caso multivariato multiparametrico

```

torch.manual_seed(394)

# Costruisco il simulatore si tratta di una normale bivariata
# con covarianze fissate a 0.5. 4 parametri da stimare
nsim = 15
def simulator(theta, nsim = nsim):
    mean = torch.tensor([theta[0], theta[1],
```

```
                        theta[2], theta[3],theta[4],
                        theta[5],theta[6], theta[7]])
    # SIGMA = torch.tensor((8,8))
    varianze = theta[ -8:]
    SIGMA = torch.diag(varianze)
    distribution = MultivariateNormal(loc=mean, covariance_matrix=SIGMA)
    out = distribution.sample((nsim,))
    return out
# Il vero valore generatore dei dati è theta_true
theta_true = torch.tensor([-0.5, 2.4,
                           2.0, 0.9,0.5,
                           2.5,-1.0,-1.0,
                           1.0,2.0,0.4,
                           1.0,2.0,1.5,
                           0.3,1.4])
x_oss = simulator(theta=theta_true, nsim = nsim)
print(x_oss)
print(theta_true[-8:])
print(torch.diag(theta_true[-8:]))
n_star = 100000

theta_star = torch.cat((Uniform(-3,3.7).sample((n_star,8)),
                        #Uniform(-0.5,0.8).sample((n_star,1)),
                        Uniform(0.2,3.5).sample((n_star,8))), dim = 1)
print(theta_star)

x_star = torch.empty((n_star,nsim,8))
for j in range(theta_star.shape[0]):
    x_star[j,:,:] = simulator(theta = theta_star[j,:], nsim = nsim)
    if j % 10000 == 0:
        print(f'iterazione n {j}')

# CREAZIONE DELLA RETE che restituisce i paraemtri
# della mistura di gaussiane
# surrogato per la verosimiglianza
```

```
44 class MixtureDensityNetwork(nn.Module):
45         def __init__(self, dim_in, dim_hid, k, dim_out):
46             super().__init__()
47             self.dim_in = dim_in
48             self.dim_hid = dim_hid
49             self.dim_out = dim_out
50             self.k = k
51 
52             self.hidden_layers = nn.Sequential(
53                 nn.Linear(dim_in,dim_hid),
54                 nn.ReLU(), # ReLU disattiva alcuni legami
55                 nn.Linear(dim_hid,dim_hid),
56                 nn.Tanh()
57                 )
58             self.mean_out = nn.Sequential(
59                 nn.Linear(dim_hid, dim_out * k)
60                 )
61             self.dev_out = nn.Sequential(
62                 nn.Linear(dim_hid, dim_out)
63                 )
64             self.pi_out = nn.Sequential(
65                 nn.Linear(dim_hid, k),
66                 nn.Softmax(dim = 0) # vincolo di somma ad 1 dei pesi d
67                 )
68 
69         def forward(self,x):
70 
71             hid = self.hidden_layers(x)
72             mu  = self.mean_out(hid).view(-1,self.k, self.dim_out)
73             sd = torch.exp(self.dev_out(hid))
74             sigma = torch.diag_embed(sd)
75             pi  = self.pi_out(hid)
76             return mu, sigma , pi
77 
78         def loss(self, input, target):
```

```
79
80              mu, sigma, pi = self.forward(input)
81              mean = mu.unsqueeze(2)
82              alpha = pi.unsqueeze(2)
83              d = target.shape[2]
84              det_S = torch.det(sigma)
85              S_inv = torch.inverse(sigma)
86              const = -0.5*d*torch.log(torch.tensor(
87              [2*math.pi])) -0.5*torch.log(det_S)
88              const = const.unsqueeze(1)
89              likelihood = torch.empty(input.shape[0],
90              target.shape[1], self.k)
91              '''
92              calcolo il valore della densità di x_star in
93              tutte le componenti della mistura
94              e le peso
95              '''
96              for c in range(self.k):
97                  mu_c = mean[:,c,:]
98                  Z_t = torch.transpose(target-mu_c,1,2)
99                  ZS = torch.matmul((target-mu_c),S_inv)
100                 ZSZt = torch.diagonal(torch.matmul(ZS,Z_t),
101                 offset=0, dim1=-2, dim2=-1)
102                 likelihood[:,:,c] =  torch.exp(const-0.5*ZSZt)*(
103                 alpha[:,c])
104             lik= torch.sum(likelihood,2) # sommo sulla componenti
105             log_lik =  torch.sum(torch.log(lik),1) # log verosimiglianza
106             return  -log_lik # negativo perché gradient
107                              # descent minimizza
108
109         def point_density_evaluation(self, input, target = x_oss):
110             '''
111             Questa funzione è identica a quella precedente però
112             consente la valutazione
113             della densità della normale in un singolo
114             punto di dimensione 2:
```

```
            serve per l'MCMC successivo
            '''
            mu, sigma, pi = self.forward(input)
            mean = mu.squeeze(0)
            alpha = pi.squeeze(0)
            d = target.shape[1]
            det_S = torch.det(sigma)
            S_inv = torch.inverse(sigma)
            const = -0.5*d*torch.log(torch.tensor(
            [2*math.pi]))- 0.5*torch.log(det_S)
            likelihood = torch.empty(target.shape[0],self.k)
            for c in range(self.k):
                z = target -  mean[c,:]
                # Z_t = torch.transpose(target-mu_c,0,1)
                ZS = torch.matmul(z,S_inv)
                ZSZt = torch.diagonal(torch.matmul(ZS,z.t()))
                likelihood[:,c] =  torch.exp(const-0.5*ZSZt)*(alpha[c])
            lik = torch.sum(likelihood,dim = 1)
            return torch.sum(torch.log(lik))


# creazione della rete
MDN = MixtureDensityNetwork(dim_in = 16, dim_out = 8,
dim_hid = 90, k = 4) # 9450 parametri qualcuno
                     # disattivato da ReLU !!

# divisione stima e validazione
theta_train, theta_val, out_train, out_val,
= train_test_split(x_star, theta_star,
test_size=0.2, random_state=42)

train_dataset = TensorDataset(out_train, theta_train)
val_dataset = TensorDataset(out_val, theta_val)

optimizer = optim.Adam(MDN.parameters(), lr = 0.005)
```

```
batch_size = 212  # i dati non vengono in input tutti insieme
train_loader = DataLoader(train_dataset,
batch_size = batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)

# Early stopping
patience = 7 # iniziale 10/20
best_val_loss = float('inf')
counter = 0

# Addestra il modello
num_epochs = 1000 # numero massimo di passi/aggiornamenti di w
for epoch in range(num_epochs):
    # Addestramento
    MDN.train()
    for inputs, targets in train_loader:
        outputs = MDN(inputs)

        # Calcola la loss/verosimiglianza
        loss = MDN.loss(inputs, targets).sum()

        # Backward pass e aggiornamento dei pesi
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Valutazione sul set di validazione
    MDN.eval()
    with torch.no_grad():
        val_loss = 0
        for inputs, targets in val_loader:
            outputs = MDN(inputs)
            val_loss += MDN.loss(inputs, targets).sum()
    # Calcola la media della loss sul set di validazione
    val_loss /= len(val_loader)
```

```
    # Stampa la loss di addestramento e di validazione
    print(f'Epoch [{epoch+1}/{num_epochs}],
    Training Loss: {loss.item():},
    Validation Loss: {val_loss:}')

    # Controlla se la loss sul set di validazione è migliorata
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        counter = 0
    else:
        counter += 1

    # Se la loss sul set di validazione non migliora
    # per il numero di epoche specificato,
    # interrompi l'addestramento
    if counter >= patience:
        print(f'Early stopping at epoch {epoch+1}')
        break

torch.manual_seed(83)
# Prendo a caso un theta* come punto iniziale per una MCMC
theta_0 = theta_star[torch.randint(0,n_star,size = (1,))]
theta_0 = theta_0.squeeze(0)
print(theta_0)
def MCMC(n_values, eps, th_0, burnin = 500):
    '''
    Ora che abbiamo una densità surrogato paraemtrica e
    valutabile possiamo simulare per ottenerne
    una versione empirica
    '''
    acc = 0
    theta_values = torch.empty(n_values, th_0.shape[0])
    theta_values[0,:] = th_0

    rw_dist = Uniform(-eps,eps)
```

```
    U_dist  = Uniform(0,1)
    th = th_0
    for j in range(1,n_values):
        th_star = th + rw_dist.sample((16,))
        th_star[-8:] = torch.abs(th_star[-8:])

        alpha =  torch.exp(MDN.point_density_evaluation(input = th_star)
                                - MDN.point_density_evaluation(th))
        if U_dist.sample((1,)) < alpha:
            th = th_star
            acc += 1
        theta_values[j,:] = th
        acc_rate = acc/j
        if j % 500 == 0:
            print(f'num proposal: {j}, acceptance_rate = {acc_rate}')
    return theta_values[burnin:], acc/n_values

# PROVARE DIVERSE CATENE CON DIVERSI PUNTI DI PARTENZA


posterior = MCMC(n_values = 40000, eps = 0.3,
th_0 = theta_0, burnin = 7500)
theta_posterior = posterior[0]
print(torch.mean(theta_posterior,0))
# Calcoliamo gli intervalli di confidenza quantile
quantile_025 = torch.kthvalue(theta_posterior,
int(0.025 * theta_posterior.size(0)), dim=0).values
quantile_975 = torch.kthvalue(theta_posterior,
int(0.975 * theta_posterior.size(0)), dim=0).values
```

Appendice B

Codice R sviluppato

B.1 Studio di simulazione per sostituibilità dei modelli surrogato alla verosimiglianza

```r

sigmoid <- function(x) 1/(1 + exp(-x))
softmax <- function(hidden) exp(hidden)/rowSums(exp(hidden))
dim_input <- 1
k <- 3
hidden_size <- 5
# Verosimiglianza ed intervalli
logL <- function(theta, dati) -sum(dgamma(dati, theta, 1, log = T))
simulatore <- function(theta, n_sim = 20)  rgamma(n_sim, theta,1)
studio_simulazione <- function(numero_simulazioni = 500)
{
  TRUE_THETA <- numeric(numero_simulazioni)
  Y_OSS <- matrix(NA,nrow = numero_simulazioni, ncol = 20)
  out <- numeric(numero_simulazioni)
  intervalli <- matrix(NA, nrow = numero_simulazioni, ncol = 2)
  for (i in 1:numero_simulazioni)
  { set.seed(i)   # fisso il seme per la riproducibilità
    true_theta <- runif(1,0.5,4)
    TRUE_THETA[i] <- true_theta
    y_oss <- simulatore(theta = true_theta, n_sim = 20)
```

```
    Y_OSS[i,] <- y_oss
    mle <- nlminb(start = 1, function(u) logL(u,dati = y_oss))
    out[i] <- mle$par
    IC_l <- uniroot(function(u) - logL(u, dati = y_oss) + mle$objective
                    + qchisq(0.95,1)/2, c(1e-07,mle$par))$root
    IC_u <- uniroot(function(u) - logL(u, dati = y_oss) +  mle$objective
                    + qchisq(0.95,1)/2, c(mle$par,10))$root
    intervalli[i,] <- c(IC_l,IC_u)
  }
  list(TRUE_THETA = TRUE_THETA,Y_OSS = Y_OSS,
  result =cbind(intervalli, out) )
}
intervalli_W <- studio_simulazione()
Y_OSS <- intervalli_W$Y_OSS

# write.csv(as.data.frame(Y_OSS), 'campioni_simulati.csv')
# grafici in pyton
TRUE_THETA <-intervalli_W$TRUE_THETA
set.seed(299)  ; theta_star <- rgamma(60,3,0.5)
set.seed(9389) ; x_star     <- unlist(lapply(theta_star,
                               function(x) simulatore(x)))
set.seed(982)  ; initial_params <- runif(40)


IC <- intervalli_W$result[,1:2]


library(modeest)
# Funzione che trasorma l'input theta nei
# parametri della mistura p(x|theta) (surrogato di verosimiglianza)
# a partire da valori causali dei pesi W (sono 50 in tutto)

# costruzione della rete
SYNT_logL <- function(input, param, target)
{
```

```
  w1      <- matrix(param[1:5],
                    nrow = dim_input, ncol = hidden_size)
  w_mu    <- matrix(param[6:20], nrow = hidden_size, ncol = k)
  w_sigma <- matrix(param[21:25],
                    nrow = hidden_size, ncol = 1)
  w_pi    <- matrix(param[26:40],
                    nrow = hidden_size, ncol = k)
  h       <- input %*% w1
  media   <- h %*% w_mu
  sigma   <- exp(h %*% w_sigma)
  pi      <- softmax(h %*% w_pi)
  # verosimiglianza della mistura
  sum(log(dnorm(target, media[1], sigma)*pi[1] +
            dnorm(target, media[2], sigma)*pi[2]+
            dnorm(target, media[3], sigma)*pi[3]))
}

# Funzione che fa MCMC sulla u(\theta)
# aggiornata ottenuta ad ogni passo di SNL
MCMC <- function(n_sim = 3000, burnin = 1000,
                 eps = 2.5, th_0 = runif(1,1,5), param, target)
{
  acc <- 0
  th_values <- numeric(n_sim)
  th <- th_0
  th_values[1] <- th_0
  for (j in 2:n_sim)
  {
      th_star <- th + runif(1,-eps,eps)
      if (th_star <= 0) {alpha <- 0}
      else
        {
            alpha   <- exp(SYNT_logL(param = param,
                                     input = th_star, target = target))/
                  (exp(SYNT_logL(param = param,
```

```
                                    input = th, target = target)))
        }
      # print(alpha)

      if (runif(1) < alpha)
      {
        th <- th_star
        acc <- acc + 1
      }
    th_values[j] <- th
  }
  list(th_dist = th_values[500:n_sim], acc = acc/n_sim)
}

studio_simulazione_surrogato <- function(n_simulazioni)
{
  mode <- numeric(n_simulazioni)
  for (i in 1:n_simulazioni)
    {
      obs <- Y_OSS[i,]
      set.seed(299)  ; theta_star <- rgamma(50,3,0.5)
      set.seed(9389) ; x_star     <- unlist(lapply(theta_star,
                                      function(x) simulatore(x)))
      set.seed(982)  ; initial_params <- runif(40)
      opt <- optim(initial_params, function(x) -SYNT_logL(x,
                              input = theta_star, target = x_star),
                method = 'BFGS', control = list(trace = 0, maxit = 2000))

      par_update <- opt$par
      theta_dist <- MCMC(n_sim = 3000, burnin = 1500, eps = 1,
      th_0 = 5.60, param = par_update, target = obs)

      for (r in 1:8)
        {
          new_theta <- sample(theta_dist$th_dist, 20, replace = F)
```

```
126           theta_star <- c(theta_star,new_theta)
127           x_star     <- c(x_star,unlist(lapply(new_theta,
128                                  function(x) simulatore(x))))
129
130
131       # aggiorno i pesi della reta
132
133           opt <- optim(initial_params, function(x) -SYNT_logL(x,
134             input = theta_star, target = x_star),
135              method = 'BFGS', control = list(trace = 0, maxit = 2000))
136           par_update <- opt$par
137           # campiono dalla posteriori aggiornata
138
139           theta_dist <- MCMC(n_sim = 2000, eps= 1, burnin = 500,
140                              param = par_update, target = obs)
141           cat('=')
142         }
143   mode[i] <- mlv(theta_dist$th_dist, method = 'shorth')
144   print(IC[i,1] <=  mode[i] & IC[i,2] >= mode[i] )
145   print(c(IC[i,1],mode[i] ,IC[i,2]))
146   theta_dist <- NULL
147   }
148   out <- mode
149 }
150 stime_surrogato <- studio_simulazione_surrogato(n_simulazioni = 500)
151 stime_surrogato
152
153 studio_simulazione_risultati <- cbind(inizio_surrogato, IC)
154
155 sum(inizio_surrogato >= IC[,1] & inizio_surrogato <= IC[,2])/500
156
157
```

B.2 ABC con riduzione a statistiche ed ABC con dati non trasformati del paragrafo 3.4.2.

```
    theta_true <- c(-0.5,3, 2.0,0.9)
x_oss  <- simulatore(theta = c(-0.5,3, 2.0,0.9),nsim = nsim)
quant_oss <- apply(x_oss, 2, function(u) quantile(u,
probs = c(.25,.5,.75)))
euclidean_norm <- function(u) sqrt(sum(u^2))
s_oss <- suff_statistics(x_oss)
set.seed(303)

th_0 <- jitter(theta_true, amount = 1)




ABC <- function(n_values, th_0, eps = 0.8,
                        x_oss = x_oss,
                        tolerance = 5e-1,
                        burnin = 5000,
                        statistics = FALSE)
{
  th_values <- matrix(NA, ncol = 4, nrow = n_values)
  th_values[1,] <- th_0
  th <- th_values[1,]
  acc   <- 0
  it   <- 0
  cond <- FALSE
  x_oss <- cbind(sort(x_oss[,1]),sort(x_oss[,2]))
  s_oss <- quantile(x_oss,probs = c(0.25,0.5,0.75))
  for (j in 2:n_values)
  {
    th_star <- th + rmvnorm(1,mean = rep(0,4), sigma = diag(eps,4))
    if (statistics == FALSE){

```

```
        x_star <- simulatore(th_star, nsim = 15)
        x_star <- cbind(sort(x_star[,1]),sort(x_star[,2]))
        rho    <- euclidean_norm((x_oss - x_star))
        cond   <- rho < tolerance
    }
    if (statistics == TRUE) {
      x_star <- simulatore(th_star, nsim = 15)
      s_star <- quantile(x_star, probs = c(0.25,0.5,0.75))
      rho <- euclidean_norm((s_star - s_oss))
      cond <- rho < tolerance
    }
    if (cond == TRUE)
    {
      th <- th_star
      acc <- acc +1
    }
    th_values[j,] <- th
    if (j %% 10000 == 0) {print(paste('n_iteration:' , as.character(j),
                           'acceptance rate: ', as.character(acc/j),
                          'accettati:',
                         as.character(acc)),
                              )}
    if(j %% 1000 == 0) {cat('=')}

  }
  out <- list(theta_dist =
  th_values[burnin:n_values,]
  , acceptance_rate = acc/n_values)
}
```

Tabella B.1: statistiche descrittive distribuzione a posteriori empirica $\hat{\pi}_\varepsilon(\theta|y_{oss})$ ottenuta con ABC, la tabella fa riferimento alla Figura 3.2

Parametro	θ_1	θ_2	θ_3	θ_4
θ_0	-0.5	3.0	0.693	-0.1053
intervallo	$(-1.652; 0.749)$	$(1.448; 3.698)$	$(-3.53; 1.0882)$	$(-3.23; 1.168)$
Media emp.	-0.55	2.5043	-0.3893	-0.4862
sd. emp.	0.6043	0.5859	1.2058	1.1289
moda emp.	-0.572	2.3554	0.158	0.0469
min	-2.12	0.864	-8.221	-8.87
max	1.15	4.302	1.368	1.371

Tabella B.2: statistiche descrittive distribuzione a posteriori empirica $\hat{\pi}_\varepsilon(\theta|S(y_{oss}))$ ottenuta con ABC e riduzione dei dati y a statistiche $S(\cdot)$. La tabella fa riferimento alla Figura 3.2

Parametro	θ_1	θ_2	θ_3	θ_4
θ_0	-0.5	3.0	0.693	-0.1053
intervallo	$(-1.2448; 0.0996)$	$(1.633; 3.2532)$	$(-0.873; 1.1270)$	$(-0.139; 1.830)$
Media emp.	-0.6466	2.4725	0.2655	0.9439
sd. emp.	0.354	0.461	0.5323	0.5346
moda emp.	-0.6901	2.446	0.166	0.954
min	-1.47	1.11	-1.336	-0.284
max	0.352	3.631	1.127	1.830

Bibliografia

Abadi, Martín et al. (2016). "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283.

Agarwal, Sameer, Keir Mierle et al. (2022). *Ceres Solver: Tutorial and Reference*. Rapp. tecn. Available at: `http://ceres-solver.org`. Google Inc.

Baydin, Atilim Gunes et al. (2015). "DiffSharp: an AD library for .NET languages". In: *arXiv preprint arXiv:1509.00160*.

Bishop, Christopher M. (1994). "Mixture Density Networks". In: *Neural Computing Research Group* 10, pp. 1–12.

— (2021). *Pattern Recognition and Machine Learning*. Springer.

Bornn, Luke et al. (2017). "The use of a single pseudo-sample in approximate Bayesian computation". In: *Statistics and Computing* 27.3, pp. 583–590.

Carpenter, Bob et al. (2017). "Stan: A probabilistic programming language". In: *Journal of Statistical Software* 76.1, pp. 1–32.

Dai, Yu-Hong (2002). "Convergence Properties of the BFGS Algorithm". In: *SIAM Journal on Optimization* 13.3, pp. 693–702.

Davies, Laurie (2024). "Statistical Analysis of the Ricker Model". In: *Faculty of Mathematics, University of Duisburg-Essen*. eprint: `laurie.davies@uni-due.de`.

Ferguson, Thomas S. (1983). "Bayesian density estimation by mixtures of normal distributions". In: *Recent Advances in Statistics*, pp. 287–302.

Griewank, Andreas e Andrea Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia: Society for Industrial e Applied Mathematics. DOI: `10.1137/1.9780898717`.

Hascoët, Laurent e Valéry Pascual (2013). "The Tapenade Automatic Differentiation tool: Principles, Model, and Specification". In: *ACM Transactions on Mathematical Software (TOMS)* 39.3, pp. 1–43.

Hastie, Trevor, Robert Tibshirani e Jerome Friedman (2008). *The Elements of Statistical Learning*. 2nd. Springer Series in Statistics. Springer.

Hornik, Kurt, Maxwell Stinchcombe e Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5, pp. 359–366.

Innes, Mike et al. (2018). "Don't Unroll Adjoint: Differentiating SSA-Form Programs". In: *arXiv preprint arXiv:1810.07951*.

Kingma, Diederik P e Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Maclaurin, Dougal, David Duvenaud e Matthew Johnson (2015). "Autograd: Effortless gradients in Numpy". In: *ICML 2015 AutoML Workshop*.

Nocedal, Jorge e Stephen J. Wright (2006). *Numerical Optimization*. 2nd. Springer.

Papamakarios, George et al. (2019). "Sequential Neural Likelihood: Fast Likelihood-Free Inference with Autoregressive Flows". In: *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pp. 837–848.

Paszke, Adam et al. (2017). "Automatic differentiation in PyTorch". In: *NIPS-W*.

Prangle, Dennis (2017). "Adapting the ABC distance function". In: *Bayesian Analysis* 12.1, pp. 289–309.

Revels, Jarrett, Miles Lubin e Theodore Papamarkou (2016). "Forward-Mode Automatic Differentiation in Julia". In: *arXiv preprint arXiv:1607.07892*.

Ruder, Sebastian (2016). "An Overview of Gradient Descent Optimization Algorithms". In: *arXiv preprint arXiv:1609.04747*.

Sisson, S. A., Y. Fan e M. A. Beaumont (feb. 2018). "Overview of Approximate Bayesian Computation". In: Available at: `https://example.com`.

Turner, Brandon M e Trisha Van Zandt (2012). "A tutorial on approximate Bayesian computation". In: *Journal of Mathematical Psychology* 56.2, pp. 69–85.

Vihola, Matti et al. (2022). "Sequentially Guided MCMC Proposals for Synthetic Likelihoods and Correlated Synthetic Likelihoods". In: *Bayesian Analysis* 17.1, pp. 43–69.

Wood, Simon N. (2010). "Statistical Inference for Noisy Nonlinear Dynamic Systems". In: *Nature* 466, pp. 1102–1104.