UNIVERSITÀ DI TRENTO
Department of Industrial Engineering

Advanced optimization-based robot control
Final Project
Project A: Learning a Terminal Cost

Students:
Claudia Gava [257872]
Matthias Cordioli [258797]

Note:Il seguente report fornisce una valutazione di come è stato portato avanti il progetto assegnato, le scelte adottate nell'implementazione e una revisione dei risultati ottenuti. L'intero esame fa riferimento al codice Python presente nella cartella allegata orc/final_project_, in cui i vari file, essendo numerosi, sono stati organizzati in sottocartelle. Per eseguire correttamente il codice, è necessario aprire l'editor dalla cartella più esterna. I video di come si comportano i due pendoli non sono stati inseriti nel report, ma sono visionabili se vengono eseguiti i file di codice e due esempi sono contenuti nella cartella video.

Academic Year 2025/2026

Contents

1 Introduction **3**

2 Optimization **3**
2.1 Initial conditions . . . 3
2.2 Models . . . 4
2.3 Problem Formulation . . . 4

3 Neural Network **6**
3.1 Neural Network Architecture . . . 6
3.1.1 Activation Function and Scaling . . . 7
3.1.2 Initialization . . . 7
3.1.3 Data Processing . . . 7
3.2 Training . . . 7
3.2.1 Library l4casadi . . . 8
3.2.2 Training Results . . . 8

4 Model Predictive Control **10**
4.1 Problem Formulation . . . 10
4.2 Comparison . . . 12
4.2.1 Different Horizons . . . 12
4.2.2 Different MPC Formulation . . . 14

5 Conclusions **15**
5.0.1 Reintroduction of Constraints . . . 15
5.0.2 Parallel Computing (Multicore Processing) . . . 16
5.0.3 Improvement of Smoothness . . . 16

1 Introduction

L'obiettivo del progetto è far imparare una Value Function approssimata ad una Rete Neurale partendo dall'ottimizzazione di un problema con orizzonte lungo. La rete viene poi usata per predirre un costo terminale per aiutare un MPC con orizzonte corto a comportarsi come se avesse un orizzonte lungo, riuscendo ugualmente a raggiungere il target assegnato.

Il progetto si può dividere in tre fasi diverse:

- Fase 1: Generazione dei dati tramite la risoluzione di molti OCPs con diverse condizioni inizali
- Fase 2: Addestramento della rete neurale partendo dai dati precedentemente generati
- Fase 3: Applicazione di quanto appreso dalla rete per generare un costo finale da aggiungere a un MPC con orizzonte breve. Confronto tra i risultati ottenuti dal MPC con orizzonti di lunghezza diversa e l'utilizzo o meno del costo finale definito dalla rete neurale

2 Optimization

Nella prima parte del progetto ciò che viene richiesto è di risolvere un problema di ottimizzazione in un orizzonte lungo N senza costi finali e senza constraints:

$$\begin{aligned} &minimize_{X,U} \quad \sum_{i=0}^{N-1} l(x_i, u_i) \\ &subject \quad to \quad x_{i+1} = f(x_i, u_i) \quad i = 0...N-1 \\ &\qquad\qquad\quad x_0 = x_{init} \end{aligned} \tag{1}$$

La minimizzazione resta soggetta, invece, alla dinamica importata dal modello utilizzato e l'ottimizzazione viene ripetuta partendo da diverse condizioni iniziali per lo stato. Il codice per questa parte fa riferimento al file *ocp.py*. L'ottimizzazione viene realizzata per 10000 campioni, partendo da altrettante condizioni iniziali randomiche e generando per ciascuna un optimal cost. Questi dati vengono salvati per coppie in due dataset, uno per il pendolo singolo (*dataset_pendulum.npz*) e uno per il pendolo doppio (*dataset_doublependendulum.npz*).

Per risolvere il problema di controllo ottimo, si è scelto di utilizzare la Collocation come Metodo Diretto per passare dall'analisi in tempo continuo a quella in tempo discreto. Il problema continuo è stato perciò discretizzato trasformando gli stati $x_0, \ldots, x_N$ in variabili di decisione del problema di ottimizzazione non lineare (NLP), anziché integrarli sequenzialmente come nel metodo Single Shooting. La dinamica del sistema $\dot{x} = f(x, u)$ è stata imposta nel modo più semplice utilizzando il principio d'integrazione di Eulero Esplicito:

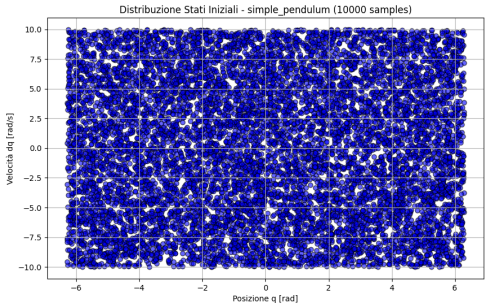
$$x_{k+1} = x_k + f(x_k, u_k) \cdot \Delta t \tag{2}$$

La Collocation come metodo permette di evitare i problemi di instabilità che sarebbero derivati dal Single Shooting e fa si che si possa sfruttare la maggiore "sparsità" ottenuta nella KKT matrix, utilizzata per la successiva risuluzione numerica del solver, rendendolo molto più efficiente.

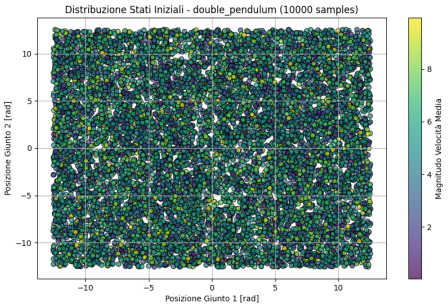
2.1 Initial conditions

Lo stato iniziale per il problema viene calcolato randomicamente della funzione *generate_random_initial_states*() definita nel file *random_generator.py*. La dimensione dello stato iniziale varia a seconda del modello utilizzato: dimensione 2 per il pendolo singolo (posizione e velocità) e 4 per quello doppio (2 posizioni e 2 velocità). I range per la generazione randomica vengono importati dalla definizione del modello e

corrispondono ai valori massimi imposti per gli angoli e le velocità. In questo modo, anche se non vi sono constraints espliciti per lo stato nell'ottimizzazione, la condizione di partenza non risulta assurda.



(a) Random IC Single Pendulum



(b) Random IC Double Pendulum

2.2 Models

Il progetto richiede di definire due modelli diversi, uno più semplice per il pendolo singolo (*pendulum_model.py*) e uno più complesso per il pendolo doppio (*doublependulum_model.py*). Le dimensioni per entrambi i pendoli come anche le masse non erano state imposte quindi sono state scelte arbitrariamente. Entrambi i modelli sono soggetti alla forza di gravità e i giunti non sono assunti come ideali, bensì hanno una propria viscosità interna (attrito viscoso lineare). Vengono definiti anche dei vincoli fisici sugli angoli, sulla velocità e sulle coppie dei motori, sebbene non vi siano dei constraint che li riguardino nel problema di minimizzazione. Come già detto, però, i valori limite vengono utilizzati nel definire il range per le condizioni iniziali randomiche.

La dinamica viene definita direttamente all'interno del modello basandosi sulle equazioni di Newton-Eulero-Lagrange:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}} - \frac{\partial \mathcal{L}}{\partial q} = \tau - F_{attrito} \tag{3}$$

Per il pendolo singolo l'equazione del moto ricavata è quella di un corpo rigido rotante con attrito viscoso, ricavata dal bilancio dei momenti:

$$\tau = I\ddot{q} + b\dot{q} + mgl \cdot \sin(q) \tag{4}$$

dove $I = ml^2$ è il momento d'inerzia, $b\dot{q}$ è il termine per l'attrito viscoso e $mgl\sin(q)$ è la coppia gravitazionale. Per il pendolo doppio, invece, la dinamica è quella di un manipolatore planare a 2 link, descritto in forma matriciale:

$$\tau = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) + F(\dot{q}) \tag{5}$$

dove $M(q)$ è la matrice d'inerzia, $C(q,\dot{q})$ è la matrice per le forze apparenti (Coriolis e Centrifuga), $G(q)$ è il vettore gravità e $F(\dot{q})$ è l'apporto dell'attrito viscoso.

Una scelta implementativa importante riguarda la formulazione dell'OCP: la variabile di controllo u scelta è l'accelerazione angolare ($\ddot{q}$), non la coppia. Di conseguenza:La dinamica inversa ($inv\_dyn()$) viene utilizzata per calcolare a posteriori la coppia τ necessaria per generare l'accelerazione u dato lo stato corrente, verificando che rispetti i limiti fisici. La dinamica forward (f) viene utilizzata per l'evoluzione dello stato tramite integrazione di Eulero Esplicito (eq:2). Quest'ultima equazione definisce il vincolo dinamico.

2.3 Problem Formulation

La cost function da minimizzare in questa fase è composta solo dal running cost, non presentando infatti un final cost. Si compone di tre diverse tasks con pesi specifici ($w\_p >> w\_v, w\_a$)e solo la task sulla

posizione fa riferimanto a un reference non nullo:

$$l(x, u) = w_p \cdot (q - q_{des})^2 + w_v \cdot dq^2 + w_a \cdot u^2 \tag{6}$$

dove $q_{des} = \pi$ per il pendolo singolo e $q_{des} = [\pi, 0]$ per il pendolo doppio. Come spiegato prima la minimizzazione è soggetta unicamente alla dinamica del modello utilizzato e alla posizione iniziale calcolata randomicamente.

L'orizzonte è finito, ma gli viene assegnato un valore abbastanza alto: si è visto che $N = 100$ è un valore sufficientemente grande per garantire una buona ottimizzazione sia nel caso del pendolo singolo che per quello doppio.

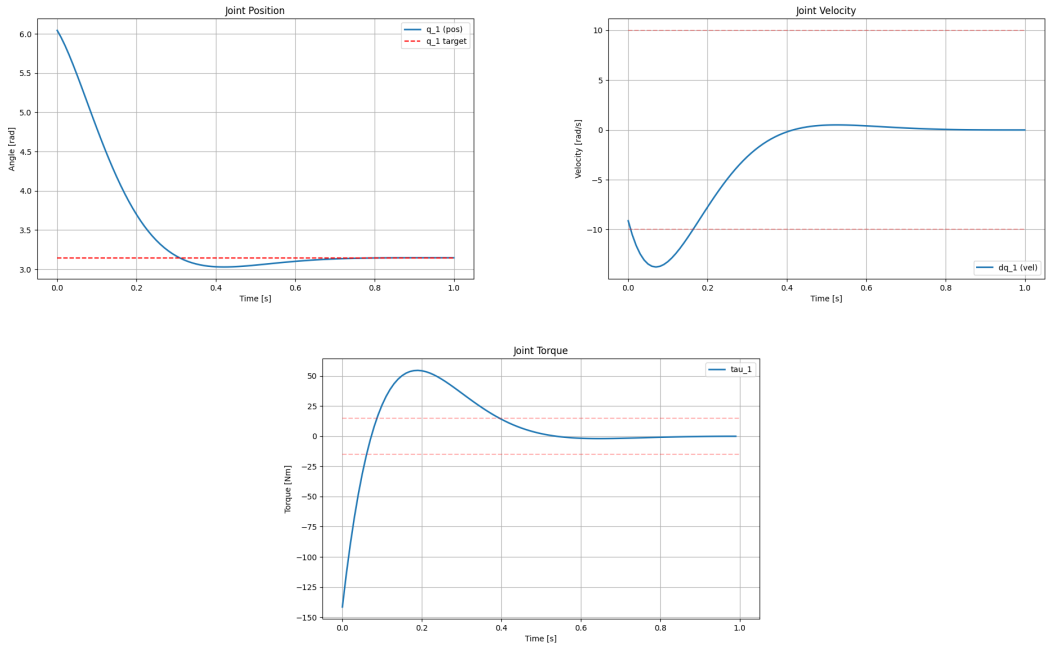


Figure 2: Results of OCP for Single Pendulum Model

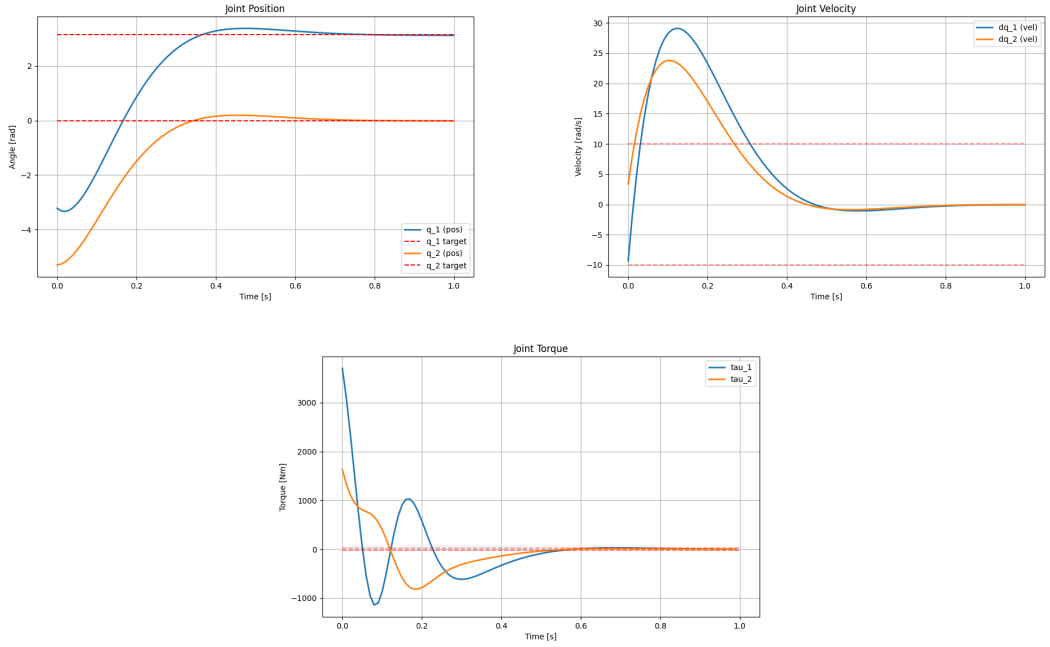


Figure 3: Results of OCP for Duoble Pendulum Model

La scelta dell'orizzonte lungo e di utilizzare un problema unconstained è dettata dal voler garantire la maggiore fattibilità possibile. Aggiungendo dei vincoli sulle variabili si renderebbe sicuramente il problema più simile a un sistema reale, ma questo potrebbe portare l'ottimizzazione a fallire per le condizioni iniziali imposte. Quindi, complessivamente, si otterrebbero meno casi buoni con cui istruire la rete in seguito. Provando a mettere i costraint sulle coppie o sulle velocità per esempio si vede come queste vadano facilmente a saturazione (Fig4).

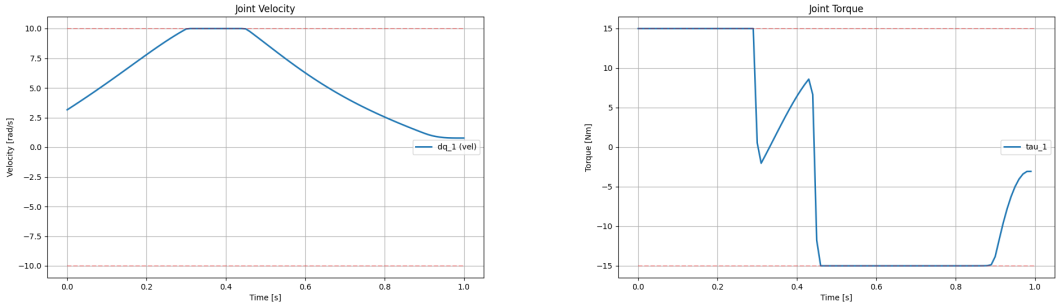


Figure 4: Results of OCP for Single Pendulum Model with Constraints

3 Neural Network

La seconda parte del progetto prevede di addestrare una rete neurale basandosi sui dataset precedentemente generati, in modo che data una condizione iniziale essa calcoli qual è la value function associata.

3.1 Neural Network Architecture

L'architettura utilizzata è un Multi-Layer Perceptron (MLP) feedforward completamente connesso, definito nella classe *NeuralNetwork* nel file *neural_network.py*.

Layers:

- Input Layer: la dimensione dell'input corrisponde alla dimensione dello stato del robot (n_x): il pendolo singolo l'input ha dimensione 2 (posizione q e velocità $\dot{q}$), mentre per il doppio pendolo ha dimensione 4 ($q_1, q_2, \dot{q}_1, \dot{q}_2$)
- Hidden Layers: La rete è composta da 2 layer nascosti, ciascuno contenente 64 neuroni
- Output Layer: L'ultimo layer ha una dimensione di uscita pari a 1, poiché la rete deve predire un singolo valore scalare: il costo ottimo $V(x)$

3.1.1 Activation Function and Scaling

Viene utilizzata la Tangente Iperbolica ($nn.Tanh()$) come funzione di attivazione dopo ogni layer lineare, incluso l'ultimo layer di output. Questa scelta è motivata dalle proprietà matematiche della Tanh, che è una funzione liscia (C^∞). Sebbene nel solver (IPOPT) si utilizzi un'approssimazione dell'Hessiano (L-BFGS) per efficienza computazionale, la liscezza della Tanh garantisce l'esistenza di gradienti continui ovunque, prevenendo le instabilità numeriche tipiche di funzioni non lisce come la ReLU e rendendo l'ottimizzazione più robusta. Poiché l'ultimo layer include l'attivazione Tanh, l'uscita della rete sarebbe intrinsecamente limitata nel range $[-1, 1]$. Per questo motivo, il risultato finale viene moltiplicato per un parametro scalare ub (Upper Bound, impostato a 10.0). Questo fattore di scala estende il codominio della rete al range $[-10, 10]$, permettendo di predire valori di costo che eccedono l'unità, come richiesto dalla natura del problema.

3.1.2 Initialization

I pesi dei layer lineari sono inizializzati utilizzando il Metodo Xavier Normal, mentre i bias sono inizializzati a zero. Questo aiuta a prevenire problemi di gradienti che esplodono o svaniscono all'inizio del training.

3.1.3 Data Processing

Prima di essere forniti in input alla rete, i dati contenuti nei dataset (sia gli stati x che i target V) vengono normalizzati. Questa operazione è fondamentale per garantire che tutte le feature abbiano lo stesso ordine di grandezza, facilitando la convergenza dell'algoritmo di ottimizzazione (Gradient Descent).

$$x_{norm} = \frac{x - \mu_X}{\sigma_X}, \quad y_{norm} = \frac{y - \mu_Y}{\sigma_Y} \tag{7}$$

Le statistiche di media (μ) e deviazione standard (σ) sono calcolate sull'intero dataset di training e salvate insieme ai pesi del modello per essere riutilizzate in seguito per la denormalizzazione.

Questo passaggio di normalizzazione e denormalizzazione sarà importante anche in seguito quando nel Model Predictive Control verrà usata la funzione *L4CasADi*: lo stato in input dovrà essere normalizzato prima di essere passato alla funzione e l'output dovrà invece essere denormalizzato per ottenere il final cost reale.

3.2 Training

Il training è stato implementato utilizzando il framework `PyTorch` nel file *training_neural_network.py*. Le epoche impostate per l'apprendimento sono 1000 e risultano sufficienti per ottenere un buon risultato (Fig5, Fig6). Il dataset generato nella fase precedente, contenente $N = 10000$ campioni, è stato suddiviso casualmente in due sottoinsiemi:

- Training Set (80%): Utilizzato per l'aggiornamento dei pesi della rete (apprendimento guidato)
- Validation Set (20%): Utilizzato per monitorare la capacità di generalizzazione e prevenire l'overfitting (test di verifica)

Il processo di minimizzazione dell'errore è stato guidato dalla funzione di costo Mean Squared Error (MSE) calcolata tra il costo predetto dalla rete e il costo ottimo J_{opt} calcolato dall'OCP. Per prevenire l'overfitting, è stata implementata una strategia di Best Model Checkpoint: ad ogni epoca viene salvato il modello che ottiene la Loss minore sul Validation Set, per evitare di salvare un modello in overfitting nelle ultime epoche.

Parameter	**Value**
Optimizer	Adam
Learning Rate	$1 \cdot 10^{-3}$
Batch Size	64
Epochs	1000
Loss Function	MSE

Table 1: Training Hyperparams

Le scelte riportate in Tabella 1 sono state motivate da considerazioni teoriche e pratiche. La Mean Squared Error è stata scelta in quanto rappresenta lo standard per problemi di regressione non lineare (Least Squares Prediction), penalizzando maggiormente i grandi errori di predizione. Per l'optimizer si è preferito Adam rispetto al classico SGD per la sua capacità di adattare il learning rate per ogni parametro, garantendo una convergenza più rapida e stabile. La velocità di apprendimento resta costante, ma l'aggiornamento è soggetto ad altri parametri che Adam calcola per ogni singolo peso della rete basandosi sulla storia passata dei gradienti:

$$w_{new} = w_{old} - \frac{\alpha}{\sqrt{v} + \epsilon} \cdot m \tag{8}$$

L'utilizzo di mini-batch, unito al mescolamento casuale (shuffling) del dataset, è stato fondamentale per rompere la correlazione temporale tra campioni consecutivi generati dall'OCP, approssimando l'ipotesi di dati i.i.d. (Independently Identically Distributed) e rendendo il calcolo del gradiente computazionalmente efficiente.

3.2.1 Library l4casadi

Una volta completato il training, il modello PyTorch non può essere utilizzato direttamente all'interno del framework di ottimizzazione CasADi, poiché quest'ultimo necessita di espressioni simboliche per il calcolo dei gradienti (Automatic Differentiation). Per risolvere questo problema, è stata utilizzata la libreria *l4casadi*. Essa permette di compilare il grafo computazionale dinamico di PyTorch in una funzione statica CasADi. In questo modo, la rete neurale appresa $V_{NN}(x)$ può essere inserita come termine di costo terminale simbolico all'interno dell'OCP, mantenendo la capacità di calcolare le derivate prime e seconde necessarie al solutore IPOPT.

3.2.2 Training Results

I risultati ottenuti confermano l'efficacia del training sia per il pendolo singolo (Fig5) che per il doppio (Fig6).

L'addestramento è stato eseguito utilizzando unicamente la CPU. Data la ridotta complessità dell'architettura (MLP con pochi layer) e la dimensione contenuta del dataset, non è stato necessario ricorrere all'accelerazione hardware tramite GPU. I tempi richiesti dal training sono sufficientemente limitati (Tab2), confermando l'efficienza computazionale dell'approccio scelto.
Analizzando le curve di Loss, si nota un rapido abbattimento dell'errore (MSE) che si stabilizza intorno all'ordine di 10^{-4}, con la Validation Loss che segue la Training Loss, indicando l'assenza di overfitting. Il grafico di accuratezza (Scatter Plot) mostra una correlazione positiva quasi perfetta tra i costi predetti dalla rete neurale (allievo) e quelli calcolati da CasADi (maestro), con i punti allineati sulla bisettrice

Training Time	
Model	**Value**
Single Pendulum	7.24 minuts
Double Pendulum	11.26 minuts

Table 2: Training Times

ideale. Non si osservano, infatti, overfitting o underfitting significativi. Se la correlazione non fosse ottimale quello che si osserverebbe sarebbe una nuvola di punti sparsi. L'Heatmap è una mappa topografica del costo: dice quanto è "costoso" (difficile) raggiungere l'obiettivo partendo da un certo stato. La zona viola scuro/blu scuro ("zona facile") rappresenta un costo basso, il che significa che il robot è già sul target o molto vicino, quindi non deve fare quasi nulla per raggiungerlo. La zona verde/giallo ("zona difficile") rappresenta un costo alto. È la zona "difficile". Significa che perchè il robot è lontano dal target e dovrà usare molta energia e tempo per arrivarci (swing-up per salire). Per il pendolo singolo questo grafico si adatta bene alla dimensione dello stato, mentre per il pendolo doppio la stessa rappresentazione andrebbe fatta in 4 dimensioni, per questo, solo in questa rappresentazione, si impostano le velocità a 0 e si riporta il peso della value function in funzione dei soli due angoli dei joints.

Infine, i pesi migliori ottenuti dal training vengono salvati in altri due dataset, insieme ai valori di media e standard deviation: *learned_value_pendulum.pth* e *learned_value_double_pendulum.pth*.

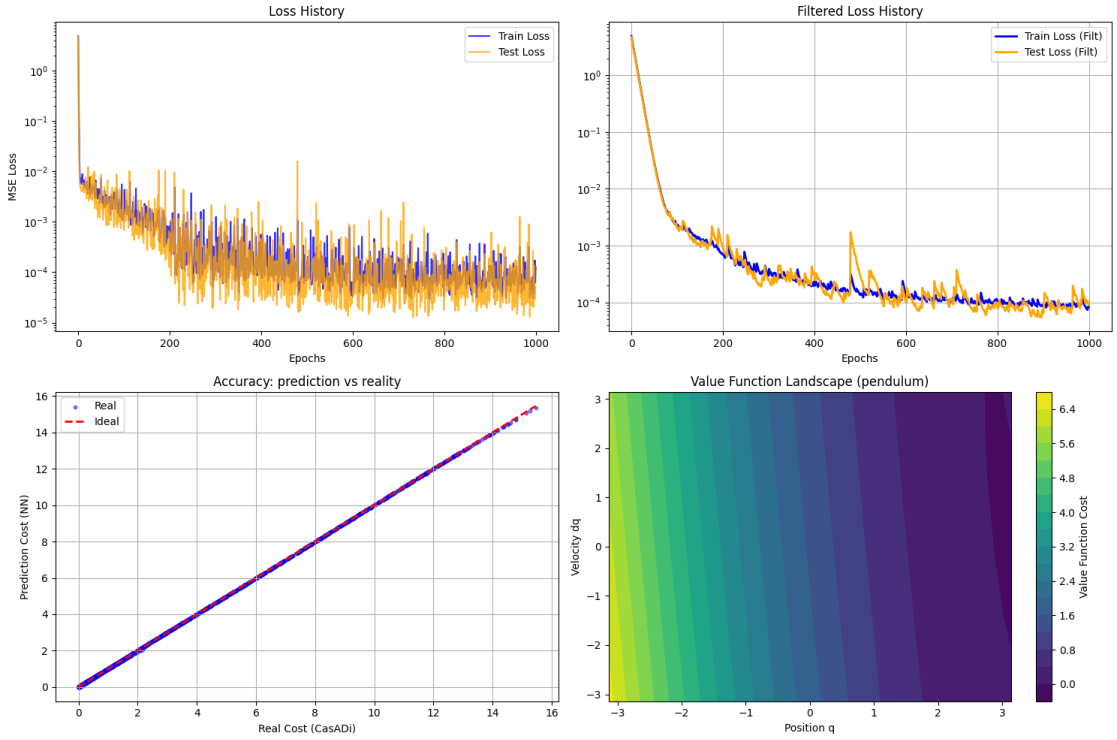


Figure 5: Training of Neural Network for Single Pendulum

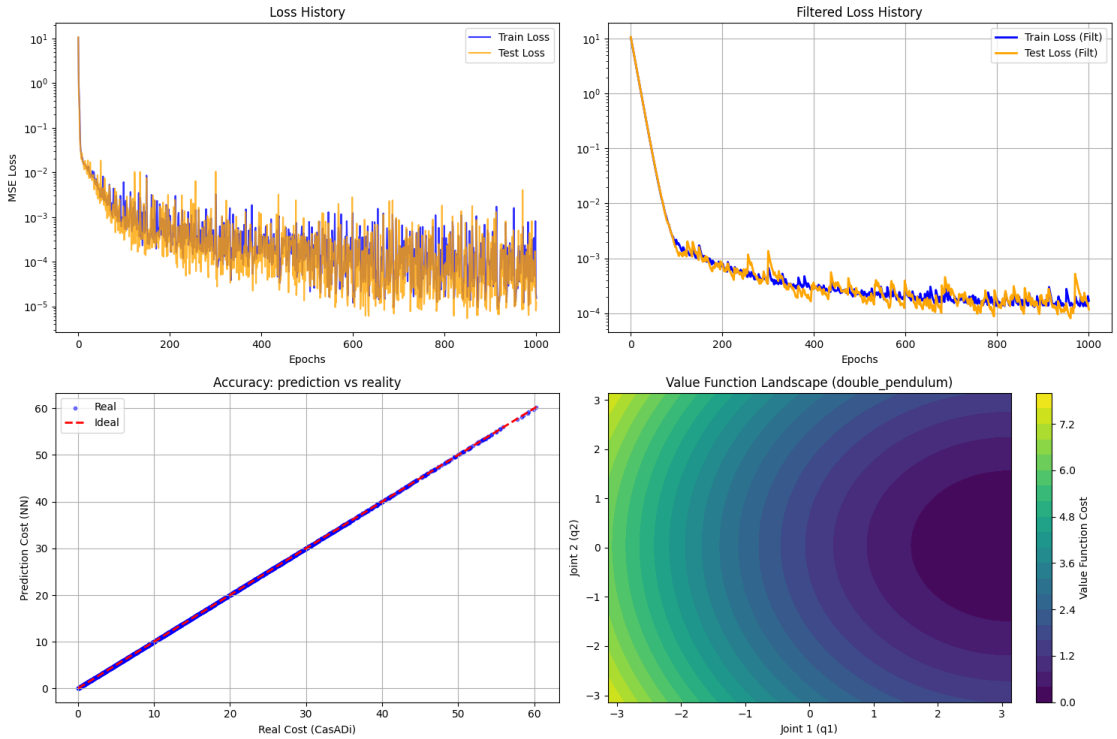


Figure 6: Training of Neural Network for Double Pendulum

4 Model Predictive Control

4.1 Problem Formulation

Nel passo 3 del progetto ciò che viene richiesto è di riformulare il probelma di ottimizzazione iniziale aggiungendo un costo finale, calcolato dalla rete istruita, e risolvendolo in un orizzonte molto più breve ($M << N$).

$$\begin{aligned} &minimize_{X,U} \quad \sum_{i=0}^{M-1} l(x_i, u_i) + J(x_M) \\ &subject \quad to \quad x_{i+1} = f(x_i, u_i) \quad i = 0...M-1 \\ &\qquad\qquad\quad x_0 = x_{init} \end{aligned} \tag{9}$$

La condizione iniziale viene generata randomicamente come in precedenza facendo riferimento al file *random_generator.py*. Il problema viena mantenuto senza constraints e la dinamica utilizzata è sempre definita tramite Eulero Esplicito (eq2), basandosi sui modelli del pendolo (*pendulum_model.py*) e doppio pendolo (*doublependulum_model.py*). Anche il running cost mantiene la definizione e i pesi usati in precedenza (eq6).

Quel che cambia è che viene aggiunto un costo finale, calcolato direttamente dalla funzione *L4CasADi* della libreria omonima. La funzione prende in input lo stato corrent e restituisce la value function associata a tale stato, basandosi su quanto appreso dalla rete neurale. Questa funzione lavora correttamente con valori di input piccoli, per questo lo stato che le viene passato in input viene precedentemente normalizzato. L'optimal cost ottenuto in output è anch'esso un valore normalizzato, quindi, prima di aggiungerlo alla funzione di costo, va denormalizzato. Per l'importazione della funzione *L4CasADi* e per entrambi i passaggi di trasformazione dei dati si fa riferimento ai database creati dopo il training (*learned_value_pundulum.pth*, *learned_value_double_pendulum.pth.*), che contengono i pesi migliori ottenuti dal training e i valori di media e diviazione standard.

Questa nuova ottimizzazione viene realizzata con un Model Predictive Control (MPC), in cui lo stato

corrente viene utilizzato come condizione iniziale e l'ottimizzazione viene reiterata ad ogni passo. Ecco perchè la soluzione completa generata da ogni ottimizzazione non è necessaria per intero, bensì è sufficiente salvare i risultati fino allo step successivo, da cui verrà rieseguita l'ottimizzazione. Solo al termine di tutto il processo di simulazione sarà quindi definita l'actual trajectory, la quale potrebbe discostarsi da quelle predette in precedenza. L'uso del MPC fa si che il sistema sia in grado di aggiornare la propria traiettoria mentre la sta percorrendo, vedendo e aggiustando step by step l'errore commesso, rendendo il sistema più robusto rispetto all'ottimizzazione standard.

Il file di riferimento per questa prima parte, in cui viene testata la validità del MPC sia per il pendolo singolo che doppio è *test_mpc.py*, con un orizzonte $M = 20$. Con tale valore l'ottimizzazione produce risultati buoni per entrambi i modelli.

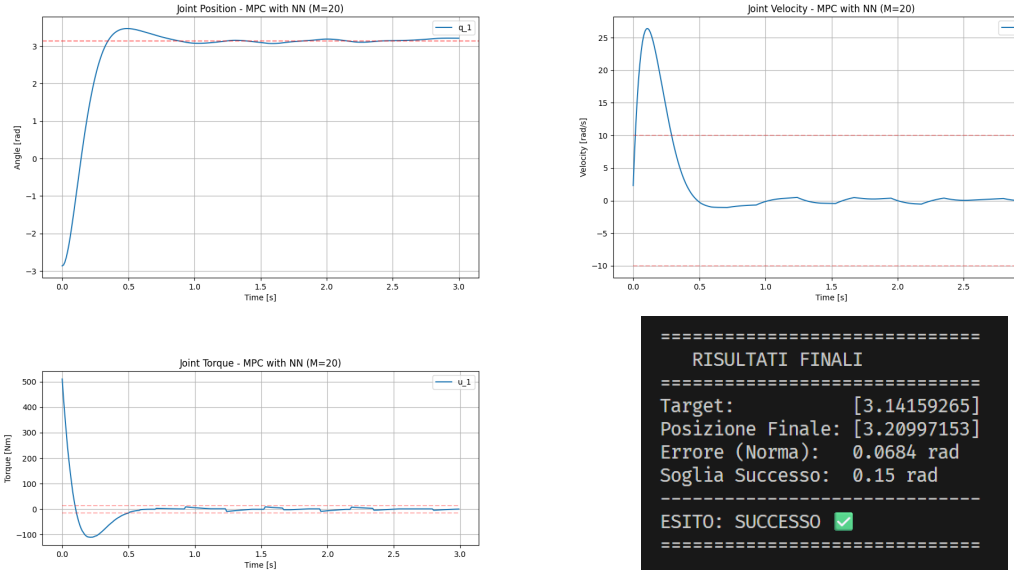


Figure 7: Results of MPC for Single Pendulum Model with M=20

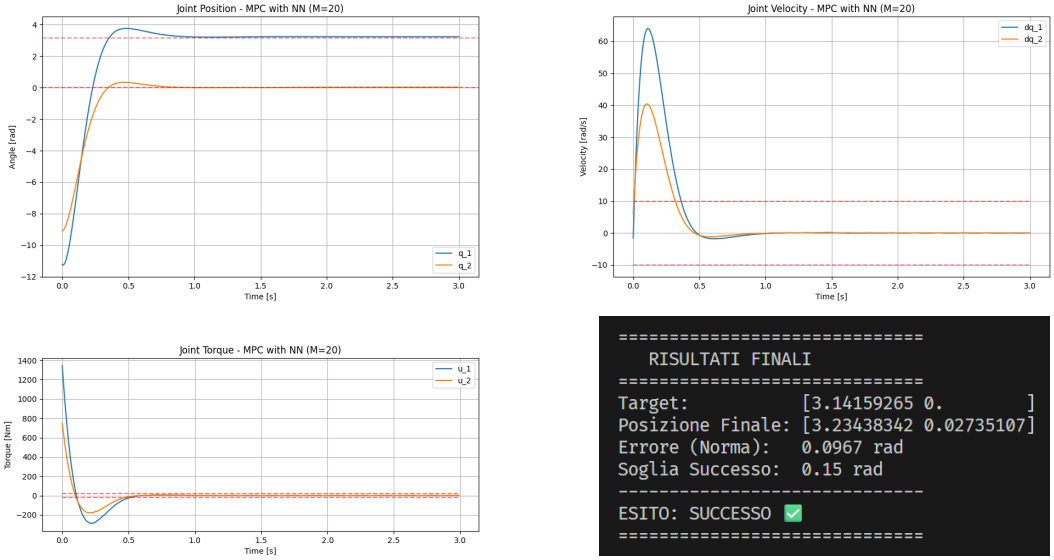


Figure 8: Results of MPC for Single Pendulum Model with M=20

4.2 Comparison

Dopo aver comprovato che il MPC funzionava correttamente, partendo dal file *test_mpc.py* è stato creato un secondo file *mpc_funzione.py*. L'unico scopo che ha questo secondo file è di vedere il MPC come funzione di diversi parametri tra cui l'orizzonte, l'uso della rete neurale e la condizione iniziale. Ciò permette di utilizzare il MPC in un analisi più ampia, confrontando in modo diretto le diverse simulazioni.

4.2.1 Different Horizons

La prima analisi riguarda la dimensione dell'orizzonte per il MPC dotato di costo finale definito dalla rete neurale (eq9). Nel file *mpc_diff_M.py* lo stesso MPC, a parità di condizioni iniziali, è stato risolto per diversi valori di M, sia per il pendolo singolo sia per il pendolo doppio. Le diverse traiettorie sono state poi sovrapposte in un unico grafico.

L'obiettivo principale dell'approccio MPC-Neural Network è ridurre drasticamente il costo computazionale minimizzando l'orizzonte di predizione M, mantenendo però la capacità di completare il task grazie all'apporto della rete neurale. Arrivato al termine dell'orizzonte che può vedere il risolutore si fida della rete per predirre il passo mancante. Osservando i grafici comparativi (Fig9, Fig10), emerge chiaramente come la qualità del controllo degradi al diminuire di M. Per orizzonti lunghi le traiettorie sono fluide, mentre, riducendo M si notano lievi overshoot e tempi di assestamento più lunghi, sebbene il sistema riesce ancora a raggiungere il target. La perdita di smoothness è dovuta alla natura approssimata della Value Function appresa dalla rete neurale ed è evidente sia negli andamenti delle velocità che delle coppie.

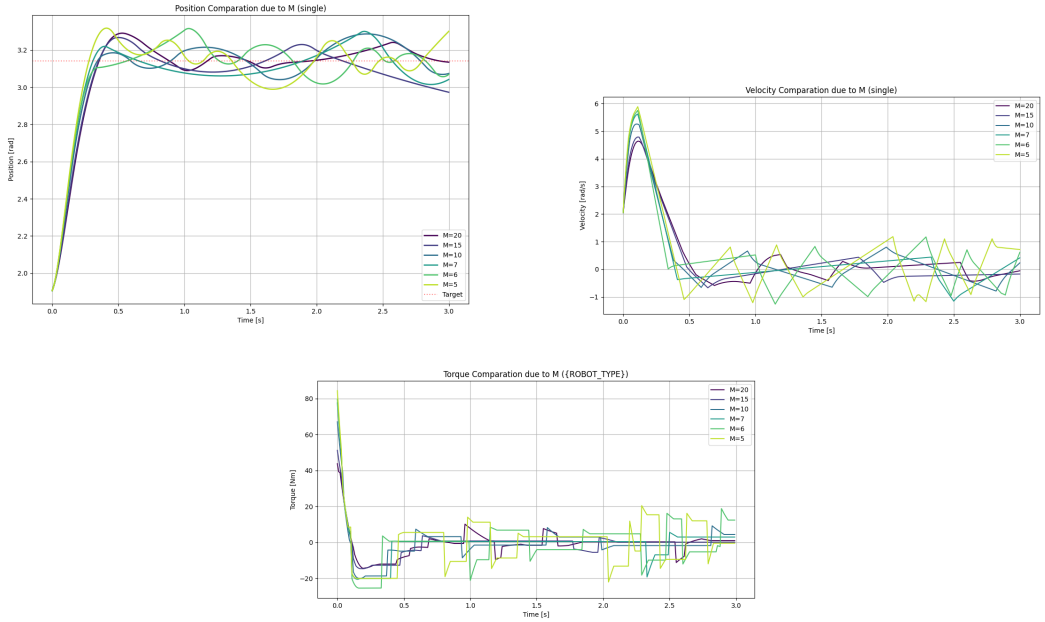


Figure 9: Results of MPC for Single Pendulum Model with different M

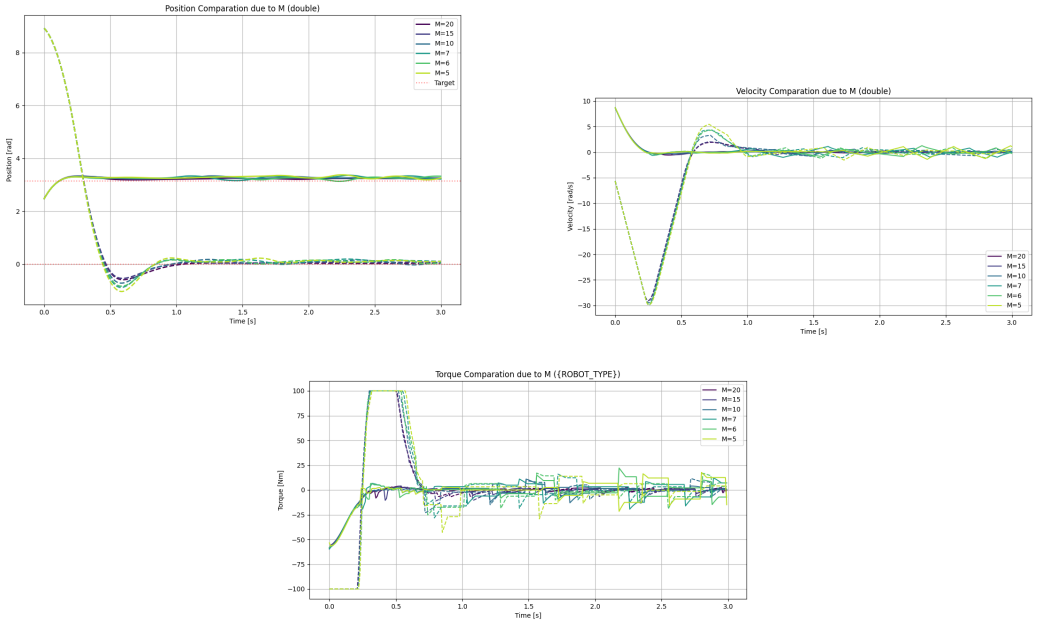


Figure 10: Results of MPC for Double Pendulum Model with different M

Il successo o meno dell'ottimizzazione è sicuramente legato alle condizioni di partenza e talvolta l'ottimizzazione fallisce in modo randomico. Gli orizzonti ampi hanno meno problemi di quelli brevi da questo punto di vista. Per riportare un'analisi statistica di questo aspetto si fa riferimento al file *diff_M_statistic.py* che fa un analisi su più campioni (100 test) con diverse condizioni iniziali, al variare dell'orizzonte e calcola la percentuale di successo.

Success Rate		
	Single Pendulum	Double Pendulum
M=20	99.0%	100.0%
M=15	99.0%	78.0%
M=10	90.0%	36.0%
M=7	87.0%	24.0%
M=6	93.0%	28.0%
M=5	96.0%	25.0%

Table 3: Success Rate

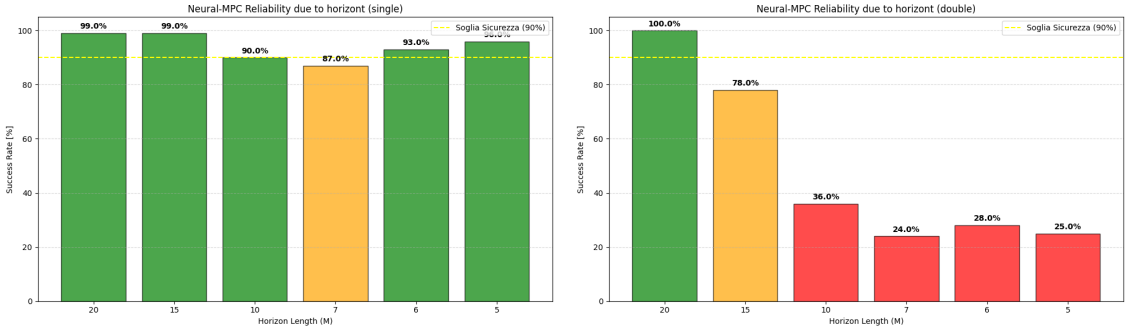


Figure 11: Success Rate Histograms

4.2.2 Different MPC Formulation

L'ultimo confronto che viene fatto vuole sottolineare i diversi risultati che si ottengono dal MPC se viene usata o meno la rete neurale (definizione costo finale) e se si cambia l'orizzonte (lungo o breve). Vengono confrontati tre casi diversi:

- CASE A: MPC Miope (Short Horizon M) - senza Neural Network
- CASE B: MPC Miope (Short Horizon M) - con Neural Network
- CASE C: MPC Far-Sighted (Long Horizon M+N) - senza Neural Network

Quello che ci si aspetta e che si vuole dimostrare è che il risolutore con un orizzonte troppo breve e senza aiuti (case A) fallisce molto spesso o comunque funziona male. Al contrario esso riesce a completare l'obiettivo se, seppur con una vista limitata, riceve un aiuto esterno che lo guida nella parte finale (case B). Il risultato ottenuto in questo caso è lo stesso che si avrebbe ampliando la vista del risolutore e non dovendo più ricorrere all'aiuto esterno (case C). Tuttavia, anche se i risultati sono pressocchè gli stessi, ridurre di molto l'orizzonte del problema fa calare drasticamente anche i tempi computazionali. In conclusione, il caso B rappresenta un'alternativa computazionalmente migliore e che offre gli stessi risultati del "caso sicuro" (case C).

Per dimostrare tutto ciò è stata eseguita una statistica sia per il pendolo singolo sia per quello doppio, risolvendo diversi MPCs (100 test) con altrettante condizioni iniziali randomiche, nei tre diversi casi presentati. Per la medesima simulazione la stessa condizione iniziale è stata usata in modo equo per tutte e tre le configurazioni. Per il valore di N si è usato lo stesso valore defito nell'ottimizzazione iniziale ($N = 100$). Il valore di M, invece, è stato deciso dall'analisi statistica precedente (Tab3), mantendo un valore con una percentuale di successo buona ($> 90\%$), ma allo stesso abbastanza breve: $M = 5$ per il pendolo singolo e $M = 20$ per il pendolo doppio.

I risultati statistici (Fig12, Fig13) rispettano le aspettative e fanno riferimento al file *comparison_mpc.py*: il caso B ha una percuntuale di successo sufficientemente alta e computazionalmente parlando viene risolto in tempi più brevi, a parità di errore e costo medi, rispetto al caso C (safe case). Questo rende la soluzione implementata con l'aggiunta della rete neurale (case B) una valida alternativa all'ottimizzazione semplice (case C).

Note: Si sottolinea che i valori di costo e errore medi sono calcolati sui soli casi riusciti e non c'è una penalizzazione legata ai fallimenti.

Il caso A, invece, è una soluzione del tutto scartabile perchè fallisce il più delle volte. Inoltre, i costi medi che essa genera (calcolati solo su i casi riusciti) sono più alti degli altri due casi come anche l'errore. Questo si vede bene nell'analisi del pendolo semplice, ma non in quella di quello doppio. Ciò si verifica probabilmente perchè nel secondo caso l'orizzonte breve è comunque abbastanza ampio ($M = 20$): se viene dimezzato i risultati di costi ed errori sono più simili al pendolo semplice, ma la soluzione B perde nettamente in affidabilità. L'ultima osservazione non è trascurabile ed è quella che ci fa preferire l'orizzonte $M = 20$.

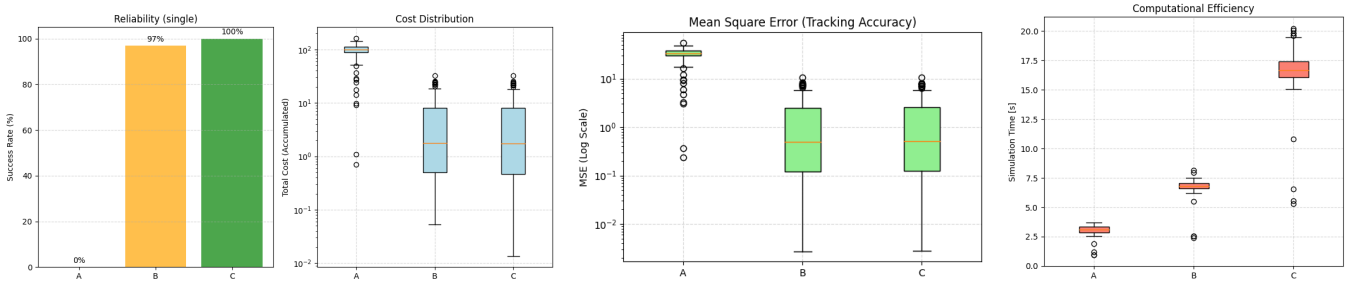


Figure 12: Statistical Analysis of Case A,B,C for Single Pendulum

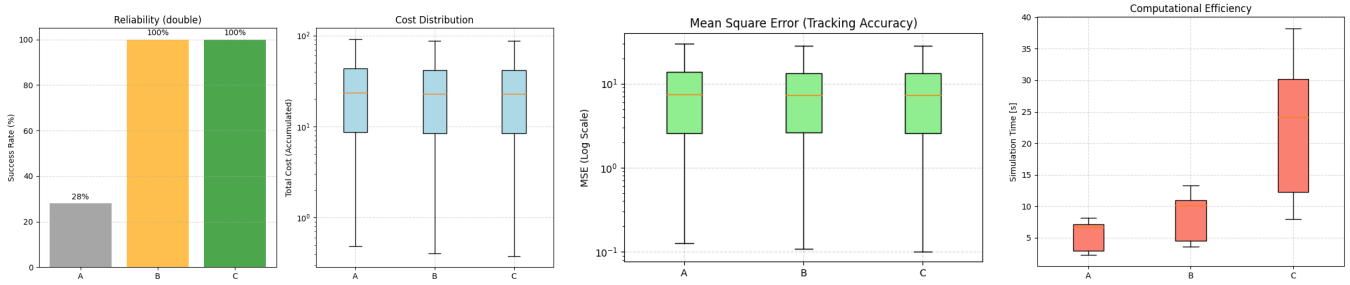


Figure 13: Statistical Analysis of Case A,B,C for Double Pendulum

L'unico aspetto limitante del caso con l'orizzonte breve e l'uso della rete neurale (case B) è la perdità di smoothness nelle traiettorie, in particolare per posizione e velocità. Questo aspetto veniva già osservato quando il MPC era stato risolto con diversi orizzonti sempre più piccoli (Fig9, Fig10). Un risultato simile si ottiene se vengono confrontati i casi A, B e C risolti, per esempio, per la condizione iniziale nulla (*final_plot_comparison.py*).

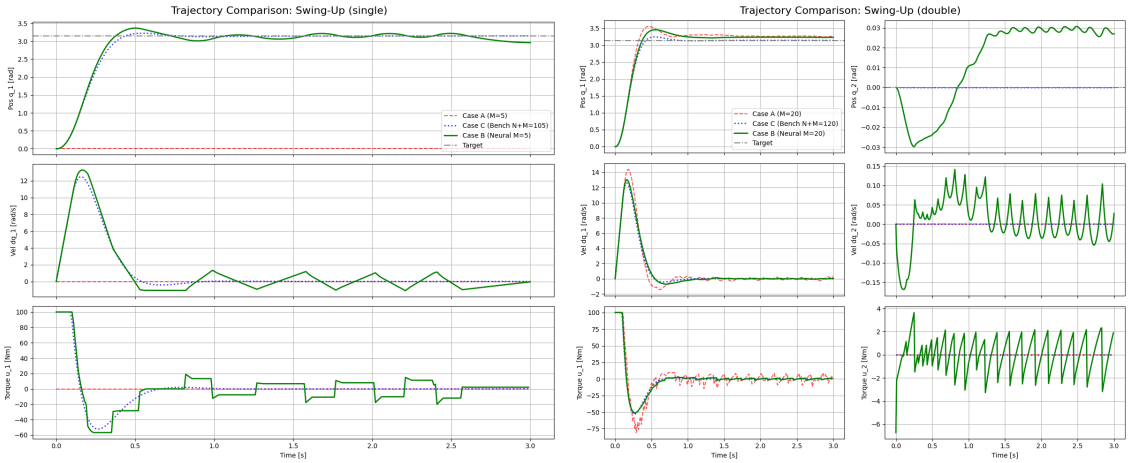


Figure 14: Trajectories of Case A,B,C for Single and Double Pendulum

5 Conclusions

Il progetto ha dimostrato l'efficacia dell'approccio Learning for Control per il controllo di sistemi robotici non lineari. L'addestramento di una Rete Neurale per apprendere la Value Function ha permesso di implementare un controllore MPC a orizzonte ridotto (Neural MPC, Case B) capace di replicare le prestazioni di un controllore a orizzonte lungo (Case C), ma con un costo computazionale drasticamente inferiore. Le analisi statistiche confermano che il Neural-MPC raggiunge tassi di successo elevati e rappresenta il miglior compromesso tra l'affidabilità del "safe case" (C) e la velocità del "myopic case" (A), rendendolo una soluzione praticabile per l'implementazione su hardware con risorse limitate.

L'analisi critica ha tuttavia evidenziato aree di miglioramento per scenari più realistici.

5.0.1 Reintroduction of Constraints

Nel progetto attuale, la generazione del dataset (Fase 1) è avvenuta risolvendo OCP senza vincoli stringenti sugli attuatori per massimizzare la raccolta dati. In uno scenario reale, l'imposizione di vincoli fisici (es. limiti di coppia e velocità) comporterebbe un "filtraggio" naturale delle condizioni iniziali: molti stati di partenza risulterebbero infattibili (non potrebbero raggiungere il target rispettando i vincoli). Di conseguenza, per ottenere un dataset valido della stessa dimensione ($N = 10.000$), sarebbe

necessario simulare un numero di combinazioni iniziali significativamente maggiore, scartando i casi di fallimento. Tuttavia, allenare la rete su traiettorie constrained permetterebbe al Neural-MPC di apprendere implicitamente i limiti del sistema, migliorando la fattibilità fisica delle predizioni in queste condizioni.

5.0.2 Parallel Computing (Multicore Processing)

Il codice presentato esegue la generazione dei dati e le analisi statistiche in modo sequenziale per mantenere una struttura software lineare. Tuttavia, questi processi sono altamente parallelizzabili. L'implementazione del multiprocessing (sfruttando tutti i core della CPU) ridurrebbe drasticamente i tempi di calcolo della Fase 1. Questo upgrade tecnico sarebbe fondamentale per gestire la problematica citata sopra: permetterebbe di generare massicci volumi di dati (es. 10^5 o 10^6 tentativi) in tempi ragionevoli, compensando l'alto tasso di scarto dovuto ai vincoli attivi.

I file che potrebbe essere utile riformulare con il multicore sono racchiusi in $orc/final\_project\_/multiprocessin$
ha senso applicare questa modifica, a mio parere, solo al file dell'ottimizzazione iniziale per la generazione
di dati (*ocp_multicore.py*) e ai file che eseguono delle statistiche su molti campioni ($diff\_M\_statistic\_multicore.p$
comparison_mpc_multicore.py).

5.0.3 Improvement of Smoothness

Dal confronto delle traiettorie (Case B vs Case C), si nota che l'orizzonte ridotto ($M = 5$) introduce talvolta un comportamento oscillatorio a scatti ("chattering") sia nelle velocità che nei comandi di controllo, dovuto alla sensibilità del solver alle imperfezioni locali della Value Function appresa. Per mitigare questo fenomeno e ottenere andamenti più smooth, si potrebbe:

- riaumentare leggemente l'orizzonte (es. $M = 10$), allungando la vista del solver che eviterebbe di effettuare cambi decisionali improvvisi
- introdurre nel costo una penalità sulla variazione del controllo ($\Delta u^2 = (u_k - u_{k-1})^2$)
- raffinare il training della rete focalizzando il campionamento (Active Learning) nelle zone dello spazio di stato più critiche, dove il gradiente della Value Function è più ripido

.

I risultati evidenziano come il Neural-MPC sia in grado di garantire stabilità e convergenza al target con un orizzonte di predizione minimo. Questo trade-off favorevole tra accuratezza e velocità di esecuzione rende la soluzione proposta particolarmente adatta per applicazioni robotiche, che richiedono alta frequenza di controllo.