

Sistema intelligente di difesa antiaerea per la simulazione, previsione e calcolo di traiettorie balistiche

Gruppo di lavoro

Studente: Epifania Cristiano

MAT.: 766404

Email: c.epifania5@studenti.uniba.it

[Repository](#)

A.A. 2024-2025

Introduzione

Lo scopo di questo progetto è simulare un sistema intelligente per un dispositivo di difesa antiaerea, capace di prevedere e ottimizzare la precisione, l'affidabilità e l'efficienza delle traiettorie balistiche. Il sistema sarà in grado di gestire le traiettorie di missili nel raggio d'azione del dispositivo, adattandosi a variabili come peso del missile, spinta, resistenza aerodinamica, angolo di lancio, durata della spinta e gravità (quest'ultima costante), prevedendo e applicando principi di balistica.

Sommario

Questo progetto propone l'integrazione di un KBS (*Knowledge Based System*) per la previsione e il controllo delle traiettorie di missili balistici utilizzando reti neurali (non tradizionali, consultare il paragrafo di riferimento). Il KBS combina modelli fisici con l'apprendimento supervisionato per costruire un sistema intelligente in grado di fornire una traiettoria balistica predetta con la maggiore accuratezza possibile. Nel seguito verrà specificata l'architettura, comprendente simulazione e addestramento della rete (d'ora in poi chiamata **PINN**).

Architettura

Il KBS si compone di due moduli principali:

1. Simulazione (*Simulation*)
2. Apprendimento supervisionato (*Supervised Learning*)

Questi moduli consentono la creazione di un dataset iniziale attraverso la simulazione del lancio di un missile con dati iniziali noti (nel seguito si mostrerà un adattamento della PINN alla non conoscenza dei dati) e al successivo apprendimento.

Simulazione

Questo modulo integra la simulazione di un lancio di un missile con correlata creazione di dati e successiva aggiunta di rumore. Per entrambe le tipologie di dati, sarà possibile visualizzarne un grafico.

Simulazione del missile

Le feature scelte per il missile sono le seguenti:

- *gravity*: ossia l'accelerazione gravitazionale, costante fisica con valore **9.81m/s²**;
- *drag_coefficient*: ossia resistenza aerodinamica. Trattasi della forza opposta al moto del missile causata dall'attrito dell'aria. Questa incide molto sul moto del missile, anticipando il momento in cui il missile collide con il terreno. Attualmente impostato a **3kg/s**;
- *launch_angle*: ossia l'angolo di lancio del missile, impostato attualmente a **85rad**;
- *initial_thrust*: ossia la spinta (boost) iniziale del missile. Un valore più alto sposta il vertice della traiettoria più in alto rispetto all'asse y. È stato scelto un valore pari a **5000N** per non esagerare con la simulazione e renderla più contenuta;
- *thrust_duration*: ossia la durata della spinta iniziale che in questa configurazione vale **2s**;
- *mass*: peso del missile impostato a **100kg** (un peso maggiore abbassa il vertice della traiettoria. In tal caso è possibile aumentare la spinta iniziale e la sua durata per una traiettoria più stabile).

Si anticipa che solo *gravity*, *launch_angle* e *mass* sono parametri noti.

Inizialmente questi dati (eccezione fatta per *gravity*) sono stati inizializzati con valori scelti arbitrariamente. Tuttavia, è possibile eliminare i commenti relativi alla generazione probabilistica di questi valori e generare valori casuali attraverso `np.linspace(start, end)`.


```
4 # Uncomment below lines if you desire random values each run or fixed ones (current: fixed)
5 # seed = np.random.randint(0, 10000)
6 # seed = 42
7 # np.random.seed(seed)
8 missile_conf = {
9     'gravity': 9.81, # m/s^2
10    'drag_coefficient': 3, # kg/s
11    'launch_angle': np.radians(85), # radians
12    'initial_thrust': 5000, # N
13    'thrust_duration': 2, # s
14    'mass': 100 # kg
15 }
```

In seguito, è stata definita la classe *Missile* per parametrizzare le chiavi del dizionario e usarle per calcolare variabili utili per la simulazione. Tre componenti sono importanti:

1. *time_arr*: vettore contenente una sequenza di *num_time_steps* (=5000) numeri uniformemente distribuiti da 0 a *max_sim_time* (tempo massimo per la simulazione);
2. *pos_arr*: matrice *num_time_steps* × 2 che modella le coppie di coordinate (x, y);
3. *speed_arr*: matrice *num_time_steps* × 2 che modella le coppie di coordinate (dx, dy) dove **dx** indica la velocità sull'asse x e **dy** sull'asse y.

Queste tre componenti sono essenziali per l'applicazione del metodo di Eulero alle equazioni differenziali ordinarie che compongono il sistema fisico di riferimento e all'interpolazione del punto di collisione tra missile e terreno, condizione per la terminazione della simulazione.

```

41     def simulate(self): 
42         """
43         Builds a simulation for a missile instance.
44         Ordinary Differential Equations are solved by Euler's method.
45         :return:
46         """
47         # Calculate simulation duration
48         time_arr = np.linspace(0, self.max_sim_time, self.num_time_steps)
49
50         # Euler's method application
51         for step in range(1, self.num_time_steps):
52             # Compute forces that alters the projectile trajectory
53             thrust = self.initial_thrust if time_arr[step] < self.thrust_duration else 0
54             drag_force = -self.drag_coefficient * self.speed_arr[step - 1, :]
55             gravitational_force = np.array([0, -self.gravity * self.mass])
56             thrust_force = thrust * np.array([np.cos(self.launch_angle), np.sin(self.launch_angle)])
57
58             # Compute force and acceleration
59             net_force = drag_force + gravitational_force + thrust_force
60             acc = net_force / self.mass
61
62             # Sum acceleration to speed and then speed to position each (time) step
63             self.speed_arr[step, :] = self.speed_arr[step - 1, :] + acc * self.dt
64             self.pos_arr[step, :] = self.pos_arr[step - 1, :] + self.speed_arr[step, :] * self.dt
65

```

Simulazione

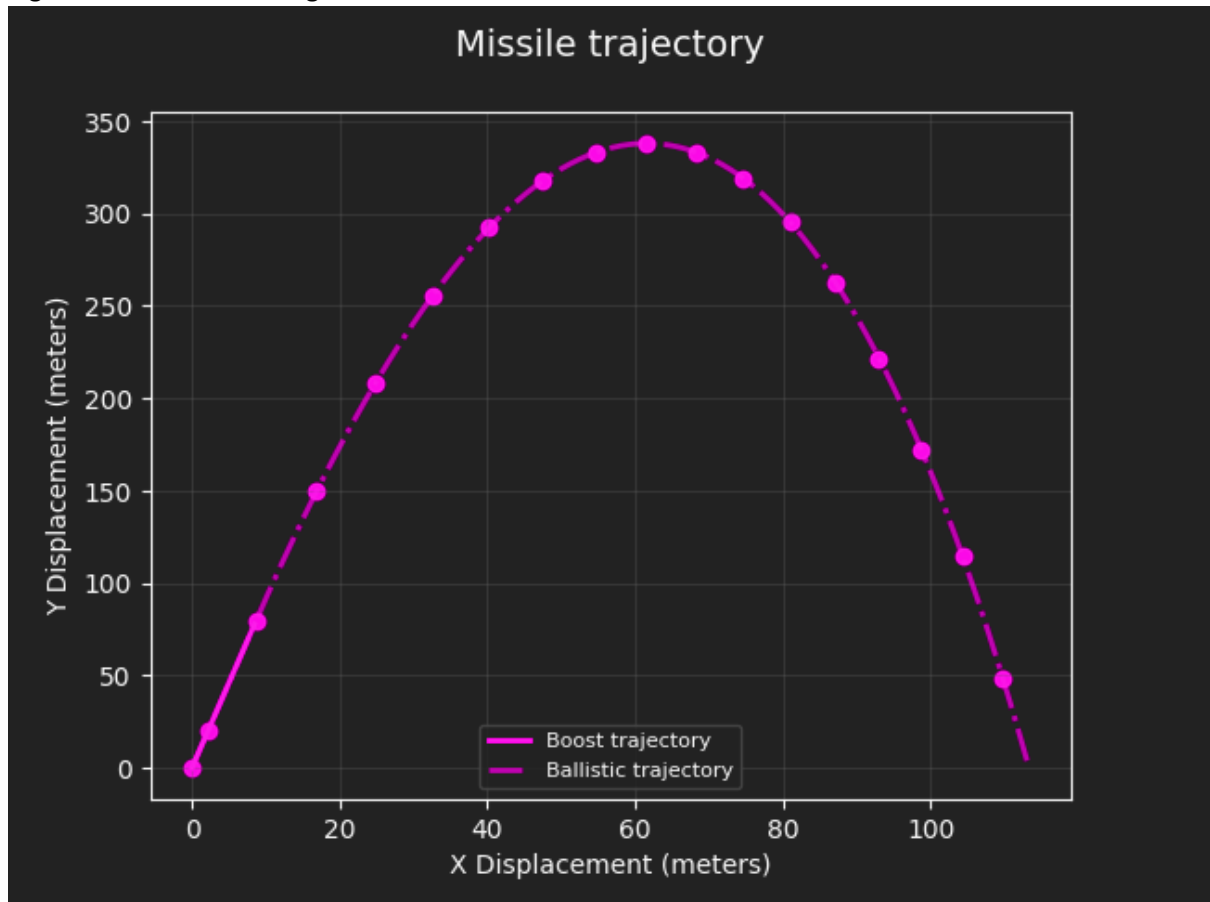
```

66         # End simulation if missile hits the ground interpolating the ground impact point
67         if self.pos_arr[step, 1] < 0:
68             self.pos_arr = self.pos_arr[:step, :]
69             break

```

Termina simulazione

Il grafico risultante è il seguente:

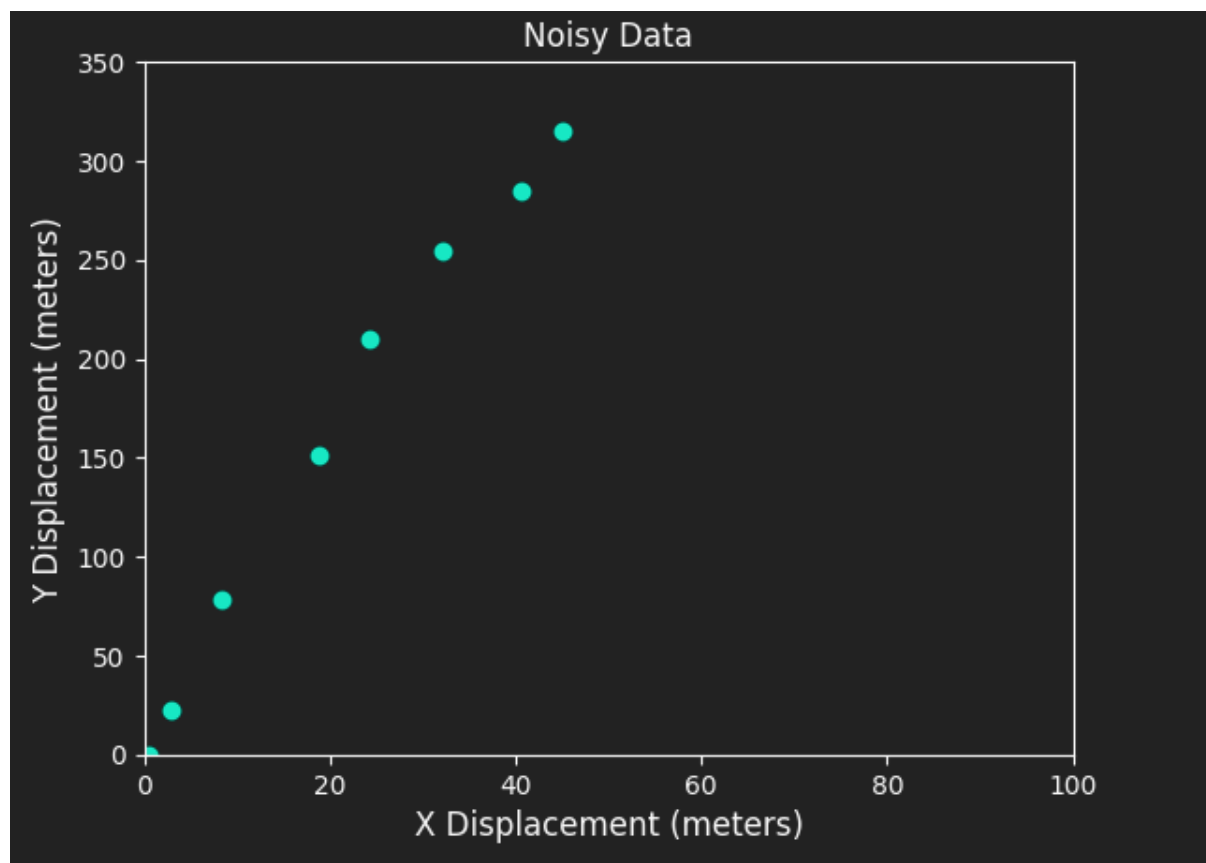


Completata la simulazione, il KBS ha una struttura del mondo su cui creare il dataset iniziale.

Generazione di rumore

Per una resa più realistica e per un principio di aleatorietà della generazione dei dati, è stato aggiunto rumore ai dati simulativi, modellando 8 punti:

```
8 def noisy_data(missile, n_points=8):  # CittiX
9     """
10     Generate noisy data for the missile ballistic trajectory
11     :param missile: A missile object
12     :param n_points: Point limit
13     :return: Noisy data
14     """
15     # Compute indices for every second
16     actual_timesteps = missile.pos_arr.shape[0]
17     # Indices for every second
18     second_indices = np.arange(0, actual_timesteps, int(1 / missile.dt))
19
20     # Every passed second find the position and limit the points to n_points
21     selected_data = missile.pos_arr[second_indices][:n_points]
22
23     # Gaussian distribution
24     noise = np.random.normal(0, (1 + selected_data / 100))
25     noisy_data = selected_data + noise
26
27     return noisy_data
```



Si può notare come i punti rumorosi si discostano da quelli effettivi.

Questo dataset viene salvato in un file denominato *dataset.csv*, non molto utile all'atto pratico di questo progetto sebbene sia efficace per una visualizzazione standard dei dati e per eventuali estensioni future di questo progetto alla stesura di altri.

X Displacement	Y Displacement
0.496714153	-0.138264301
3.084300509	24.58597317
9.237681352	88.64539848
22.99205925	200.0864531
33.90199858	321.1692631
46.90556063	424.2916264
60.64722644	509.8420756
69.57840821	600.2950869

Apprendimento

Questo modulo prepara la PINN all'addestramento dei dati generati nella fase precedente. Tuttavia va chiarito cosa sia una PINN.

Physical Informed Neural Network

Sebbene simili, ANN (Artificial Neural Network) e PINN si differenziano per alcuni aspetti. Entrambe sono reti neurali, ossia modelli parametrizzati per predizioni, tipicamente composte da diversi livelli di funzioni lineari parametrizzate e funzioni di attivazione non lineari. Questi modelli (d'ora in poi abbreviate con NN) restituiscono una funzione lineare per una predizione a valori reali. In base al tipo di predizione le NN calcolano funzioni per la predizione, come nel caso di una predizione booleana che fa uso di sigmoide, o softmax per quelle categoriche. Per quanto concerne predizioni strutturate vengono introdotti metodi speciali.

Le NN sono composte da $n - 1$ *hidden layers* (livelli/strati nascosti) ed n livelli, dove n è la profondità della rete, ognuno dei quali mappa un vettore (o strutture dati affini come array o lista). I componenti dei vettori sono detti *unità*. Nel caso di questo progetto la PINN è composta da 4 livelli.

Le NN fanno uso di una funzione di attivazione non lineare che calcola l'output in funzione degli input del livello corrente.

Le ANN sono adatte al riconoscimento di pattern complessi nei dati e all'approssimazione di funzioni non lineari, tuttavia, richiedendo un'importante mole di dati. Le ANN sono addestrate mediante metodi di ottimizzazione come la discesa di gradiente per la minimizzazione di una funzione di perdita, ad esempio MSE (Mean Squared Error, *Errore quadratico medio*, o MSL, Mean Squared Loss) o Cross-Entropia.

In questo progetto sono state adottate le PINN che si differenziano dalle ANN tradizionali per il loro dominio, ossia problemi di carattere fisico. Rispetto alla controparte tradizionale, le PINN sono composte di livelli i cui input sono coordinate spaziali e temporali, integrando equazioni fisiche nella

funzione di perdita, inserendo vincoli fisici, per l'addestramento. Non necessita di grandi quantità di dati grazie alla conoscenza incorporata della fisica.

Preparazione dei dati per l'addestramento

Vengono preparati i dati (rumorosi) per l'addestramento della PINN trattandoli come *target feature*

```
15 def pre_training(noisy_data):  # CittàX
16     """
17     This function prepares the noisy data for PINN training.
18     Noisy data are treated as the target (or samples or simply output).
19     :param noisy_data: Noisy data for PINN training
20     :return: Input-Output pairs consisting of time and observed positions
21     """
22     # The output/target to be predicted
23     samples = noisy_data
24     n_samples = len(noisy_data)
25     inputs = np.linspace(0, n_samples - 1, n_samples)
26
27     return inputs, samples
28
29
30 inputs, samples = pre_training(noisy_targets)
```

e definito un dizionario contenente gli iperparametri.

```
5 # Dictionary containing hyperparameters for model training
6 training_conf = {
7     "net_depth": 50, # Number of hidden layers
8     "learning_rate": 0.025,
9     "epochs": 1000,
10    "anim_record_freq": 3, # The higher the number the fewer animation frames are recorded
11    "anim_frame_duration": 20, # Duration (ms) of each frame
12 }
```

Definizione della PINN feed-forward

Come anticipato, la NN utilizzata in questo progetto è la PINN. Essa è stata definita attraverso la libreria *PyTorch*.

La PINN è composta di 4 livelli, di cui:

- i. 1 livello di input;
- ii. 2 livelli nascosti che fanno uso di una funzione di attivazione detta GELU;
- iii. 1 livello di output.

Il **livello di input** prende il tempo come input monodimensionale e predice la posizione del missile come **output** bidimensionale.

I **livelli nascosti** sono stati configurati attualmente con 50 neuroni e fanno uso della **GELU** come funzione di attivazione.

La **funzione di attivazione** utilizzata per questo modello è la *Gaussian Error Linear Unit* (GELU) e non la ReLU. La scelta è stata fatta tenendo in considerazione la gradualità che fornisce rispetto alla ReLU, oltre al fatto che rappresenta una soluzione più moderna e adatta alle PINN, in particolare quella definita in questo progetto che è poco profonda.

Come ottimizzatore è stato scelto **adam** (adaptive moments, *momentum adattivi*) data la presenza di scale differenti tra i parametri, la necessità di una rapida convergenza, la presenza di molti parametri e tempo richiesto per l'addestramento. Adam è un ottimizzatore che usa sia i momentum che il quadrato dei gradienti, apportando correzioni ai parametri da ottimizzare che sono stati eventualmente inizializzati a 0, dato che non rappresentano una buona stima su cui calcolare la media. Rappresenta una soluzione migliore per questo progetto rispetto, ad esempio, alla **SGD** (Stochastic Gradient Descent, *Discesa di Gradiente Stocastica*) standard che converge più lentamente.

```
24 # It uses ADAM as optimizer to guarantee fast convergence
25 optimizer = torch.optim.Adam(model.parameters(), lr=training_conf["learning_rate"])
26
27 # Use tensors to integrate GPU computing
28 in_tensor = torch.tensor(inputs, dtype=torch.float).view(-1, 1)
29 out_tensor = torch.tensor(targets, dtype=torch.float).view(-1, 2)
```

Per una computazione più rapida le feature in input ed output sono state convertite in tensori che integrano il calcolo con GPU.

Addestramento e Valutazione

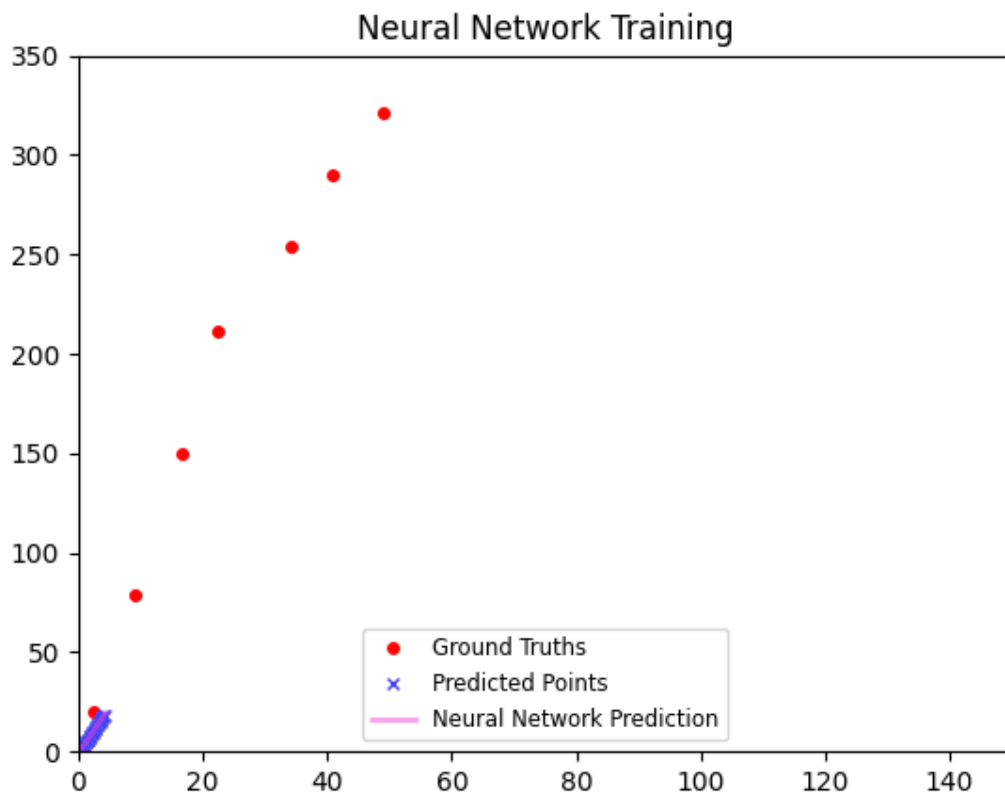
La PINN, ogni epoca, addestra le valutazioni sulle predizioni (**train_one_epoch**) e ad una fissata frequenza le registra (**train_and_record**) in un array che verrà poi utilizzato per visualizzare i risultati su un grafico.

```
11 def train_one_epoch(model, inputs, targets, optimizer):  # CittiX
12     """
13     It trains the NN for one epoch
14     :param model: Model of NN to be trained
15     :param inputs: Input features
16     :param targets: Target features
17     :param optimizer: Optimizer
18     """
19     predictions = model(inputs)
20
21     # Compute the mean squared error loss
22     criterion = nn.MSELoss(reduction='sum')
23     loss = criterion(predictions, targets) # To print loss use loss.item()
24
25     # Backward pass
26     optimizer.zero_grad()
27     loss.backward()
28     optimizer.step()
```

L'addestramento del modello in un'epoca usa come funzione di perdita la media quadratica tra predizioni e dati effettivi e ad ogni step vengono calcolati i gradienti.

```
50 def train_and_record(model, optimizer, in_tensor, out_tensor, dom_tensor, epochs, freq, train_one_epoch):
51     """
52     It trains the NN for given number of epochs recording evaluation predictions for visualization
53     :param model: Model of NN to be trained
54     :param optimizer: Optimizer
55     :param in_tensor: Input tensor
56     :param out_tensor: Output tensor
57     :param dom_tensor: Domain tensor
58     :param epochs: Number of epochs the NN will be trained
59     :param freq: Frequency of the recording
60     """
61     eval_pred = []
62
63     for epoch in range(epochs):
64         train_one_epoch(model, in_tensor, out_tensor, optimizer)
65         if epoch % freq == 0:
66             eval_pred.append(make_eval_pred(model, dom_tensor))
67
68     return eval_pred
69
70 # Domain over which NN will be evaluated
71 dom_tensor = gen_eval_domain(30, 300)
72
73 # Train the NN and record predictions over the whole domain for visualization
74 eval_pred = train_and_record(model, optimizer, in_tensor, out_tensor, dom_tensor, training_conf["epochs"],
75                               training_conf["anim_record_freq"], train_one_epoch)
```

Come primo risultato viene mostrato un grafico animato dell'addestramento della PINN, tenendo conto delle *ground truth* e dei punti predetti:



Problemi

Dal grafico si può notare come la PINN presenti sovraddattamento e scarsa estrapolazione.

La curva si adatta troppo ai punti di training e questo causerà scarsa capacità di generalizzazione. Inoltre, al di fuori della regione di training, le predizioni sono scorrette, violano leggi fisiche e vincoli e i dati non sono utilizzabili, anche se da un certo punto di vista, forniti i dati di addestramento, questo sia assumibile ma non da un punto di vista cinematico.

Soluzione

Come è stato già asserito, le PINN non sono totalmente diverse dalle ANN tradizionali. Tuttavia, oltre alle differenze già citate, vi è l'inclusione di operatori differenziali, importanti per il calcolo di leggi fisiche che coinvolgono l'oggetto di questo progetto.

Tuttavia, le PINN non risolvono le leggi fisiche direttamente, bensì imparano ad approssimare una funzione $f(x)$ tale che le equazioni coinvolte nel calcolo della fisica vengano minimizzate.

Si introduce dunque la funzione di perdita definita come segue:

$$L = L_{dati} + \lambda L_{fisica}$$

Dove:

1. L_{dati} : termine per l'errore della predizione, in questo caso l'errore quadratico;

2. L_{fisica} : termine per l'errore calcolato sulla fisica, indicatore di quanto l'output della PINN soddisfi le leggi fisiche;
3. λ : regolarizzatore dell'errore fisico.

Un'altra importante differenza che emerge tra PINN e ANN tradizionali è nell'errore fisico che è calcolato sull'intero dominio in input e non solo dove ci sono i dati di training. Queste importanti differenze rendono le PINN più adatte al problema affrontato in questo progetto rispetto alle ANN tradizionali, fornendo ottime capacità di estrapolazione e facendo emergere ottime capacità nell'apprendere dati al di fuori della regione di addestramento.

Seconda Legge di Newton e Applicazione della Funzione d'Errore Fisico

La traiettoria del missile è calcolata dalla Seconda Legge di Newton che considera forze come accelerazione gravitazionale, resistenza aerodinamica e spinta:

$$\frac{d^2x}{dt^2} = \frac{C_t \cos(\theta) - C_d \frac{dx}{dt}}{m}, \quad \frac{d^2y}{dt^2} = \frac{C_t \sin(\theta) - C_d \frac{dy}{dt} - mg}{m}$$

Dove:

1. C_t è la resistenza aerodinamica,
2. C_d è la spinta,
3. m è la massa,
4. g è l'accelerazione gravitazionale,
5. θ è l'angolo di lancio del missile.

Queste due equazioni descrivono il comportamento dell'accelerazione lungo l'asse x ed y.

Dunque, la PINN ha lo scopo di imparare una funzione $f(t)$ che predice le coordinate (x, y) al tempo t.

La retropropagazione necessita di una funzione differenziabile per il calcolo dei gradienti. La funzione utilizzata in questo progetto, $step(thrust)$ coinvolge la resistenza aerodinamica ed è attiva solo per i primi 2 secondi ($t \leq 2$). La funzione è così definita:

$$step_2(C_t) = \begin{cases} 1, & C_t \leq 2 \\ 0, & C_t > 2 \end{cases}$$

che non è differenziabile in $t=3$. Viene dunque fornita una funzione logistica come approssimazione della precedente:

$$step(C_t) = \frac{1}{1 + \exp(-k \cdot (a - t + \epsilon))}$$

Dove:

1. $k = 200$, una costante che controlla la precisione dello step
2. $a = 3$, il momento in cui la resistenza aerodinamica cede,
3. t , momento attuale,
4. $\epsilon = 0.02$, una piccola costante che adatta meglio la posizione dello step.

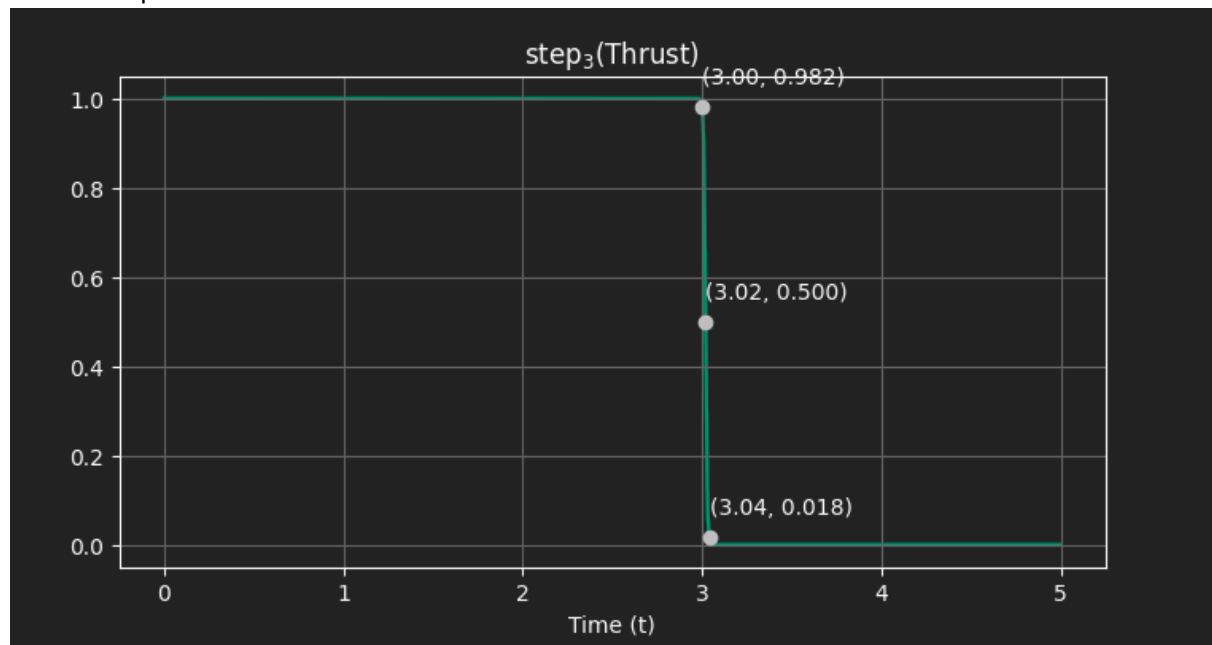
k, ϵ sono stati scelti attraverso dei test empirici.

```

77 def compute_thrust(t, thrust_cease=missile_conf["thrust_duration"] + 1, sharpness=200, offset=0.02):
78     """
79     It calculates thrust values using a differentiable step function (sigmoid)
80     :param t: Time
81     :param thrust_cease: The time at which thrust ceases
82     :param sharpness: Constant that monitors the step sharpness
83     :param offset: Position corrector for the step
84     :return: A differentiable step function
85     """
86     return 1 / (1 + np.exp(-sharpness * (thrust_cease - t + offset)))

```

Ne risulta quindi:



La funzione di perdita fisica è così definita:

$$\sum_{i=0}^N \left(\left(\frac{d^2x}{dt^2} - \frac{C_t \cos(\theta) - C_d \frac{dx}{dt}}{m} \right)^2 + \left(\frac{d^2y}{dt^2} - \frac{C_t \sin(\theta) - C_d \frac{dy}{dt} - mg}{m} \right)^2 \right)$$

Tutte le derivate sono calcolate dalla PINN, il resto invece sono costanti fisiche. Le derivate vengono predette attraverso un calcolo di gradienti:

```

7 def compute_physics_loss(model, constants, constant_scale_factors, physics_weight=10):
8     # input_tensor is time
9     dom_tensor = gen_eval_domain(30, 300).requires_grad_(True)
10
11     # Take (x, y) predictions from NN
12     phys_pred = model(dom_tensor)
13     x_pred = phys_pred[:, 0]
14     y_pred = phys_pred[:, 1]
15
16     # Compute first and second order NN derivatives with respect to time using autograd
17     dx_dt = torch.autograd.grad(x_pred, dom_tensor, grad_outputs=torch.ones_like(x_pred), create_graph=True)[0]
18     dy_dt = torch.autograd.grad(y_pred, dom_tensor, grad_outputs=torch.ones_like(y_pred), create_graph=True)[0]
19     d2x_dt2 = torch.autograd.grad(dx_dt, dom_tensor, grad_outputs=torch.ones_like(dx_dt), create_graph=True)[0]
20     d2y_dt2 = torch.autograd.grad(dy_dt, dom_tensor, grad_outputs=torch.ones_like(dy_dt), create_graph=True)[0]
21

```

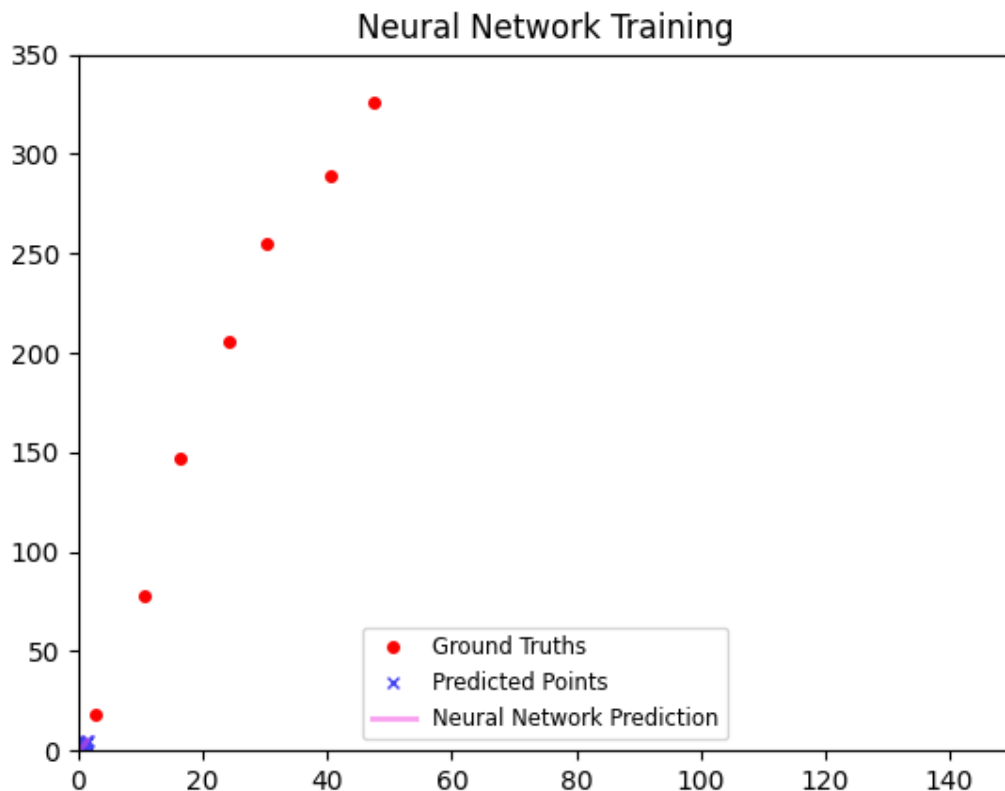
Mentre le caratteristiche note del missile vengono così calcolate:

```
26 # Known parameters
27 gravity = torch.tensor(missile.gravity, dtype=torch.float, requires_grad=False)
28 launch_angle = torch.tensor(missile.launch_angle, dtype=torch.float, requires_grad=False)
29 mass = torch.tensor(missile.mass, dtype=torch.float, requires_grad=False)
```

Tuttavia il modello non conosce parametri come *drag_coefficient* e *thrust_magnitude*, dunque vengono inizializzati a 1 e calcolati attraverso gradienti:

```
21 # Initial guesses for parameters are provided
22 init_drag_coef = 1
23 init_thrust_coef = 1
24
25 # Unknown parameters
26 drag_coefficient = torch.tensor(init_drag_coef, dtype=torch.float, requires_grad=True)
27 thrust_magnitude = torch.tensor(init_thrust_coef, dtype=torch.float, requires_grad=True)
28 learnable_constants = [drag_coefficient, thrust_magnitude]
29
30 # Scale factors
31 constant_scale_factors = {"drag_coefficient": 1,
32                           "thrust_magnitude": 3000}
```

Il risultato dell'introduzione della funzione di errore fisica è dato dal seguente grafico animato:




È stato così risolto il sovradattamento e la potenza di estrapolazione della PINN.

Il modello è stato poi reinizializzato insieme all'ottimizzatore:

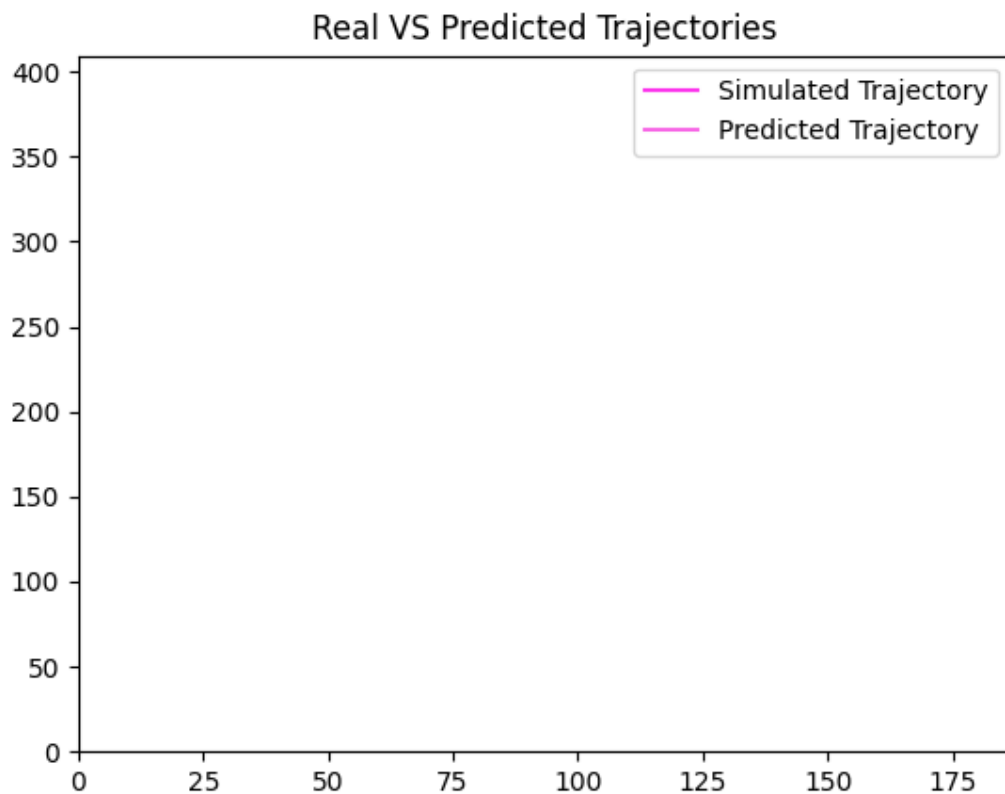
```
34 # Reinitialize network, optimizer, input and target tensors
35 model = nn.Sequential(
36     nn.Linear(input_dim, hidden_dim),
37     nn.GELU(),
38     nn.Linear(hidden_dim, hidden_dim),
39     nn.GELU(),
40     nn.Linear(hidden_dim, output_dim)
41 )
42
43 optimizer = optim.Adam([{"params": model.parameters(), "lr": training_conf["learning_rate"]},
44                         {"params": learnable_constants, "lr": training_conf["learning_rate"] / 10}]
45 )
46 input_tensor = torch.tensor(inputs, dtype=torch.float).view(-1, 1)
47 target_tensor = torch.tensor(targets, dtype=torch.float).view(-1, 2)
```

In seguito all'aggiunta della funzione di errore, è stato adattato l'addestramento di un'epoca:

```
50 def train_one_epoch_supervised(model, inputs, targets, optimizer):  CittiX
51     """
52     A remake of train_one_epoch function of training.py.
53     This uses mean squared error and physics loss.
54     """
55     # Get predictions from NN
56     predictions = model(inputs)
57
58     # Mean squared loss
59     criterion = nn.MSELoss(reduction="sum")
60     mse_loss = criterion(predictions, targets)
61
62     # Physics loss
63     physics_loss = compute_physics_loss(model, learnable_constants, constant_scale_factors)
64
65     # Combined loss
66     loss = mse_loss + physics_loss
67
68     # Backward pass
69     optimizer.zero_grad()
70     loss.backward()
71     optimizer.step()
```

Risultato Finale

Si mostra il risultato finale del progetto:



Come si può notare, la PINN sottostima leggermente le due forze ignote, spinta e resistenza aerodinamica, risultando più “piatta” rispetto a quella effettiva. Viene anche anticipata la collisione del missile con il terreno; tuttavia, è poco meno di un secondo.

Conclusioni

In questo progetto si è dimostrata la potenza e l'utilità di un modello di rete neurale feedforward noto come PINN per casi che coinvolgono la fisica. Utilizzando le PINN si è scavalcata la difficoltà del sovradattamento e della scarsa capacità di estrapolazione del modello risultando in uno affidabile, credibile e potente, nonostante la presenza di rumore e la non conoscenza di due parametri fisici.

Elenco argomenti di interesse

- Feature Engineering
- Reti neurali feed forward
- Apprendimento dei parametri
- Migliore l'ottimizzazione
- Apprendimento supervisionato
- Sovradattamento
- Regolarizzatori
- Valutare le previsioni

- Modelli di conoscenza incerta
- Ricerca di soluzioni in spazi di stati

Strumenti utilizzati

- PyTorch
- NumPy
- Matplotlib
- Pandas
- IPython
- Jupyter

IDE: PyCharm