

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 (1) per un dispositivo di difesa antiaerea, capace di prevedere e ottimizzare precisione, affidabilità ed 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*) (2) per la previsione e il controllo delle traiettorie di missili balistici utilizzando reti neurali (3) (non tradizionali, consultare il paragrafo di riferimento). Il KBS combina modelli fisici con l'apprendimento supervisionato (4) 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, creazione di dati con la conseguente 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 (5). 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*: massa del missile impostato a **100kg** (una massa 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* saranno 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

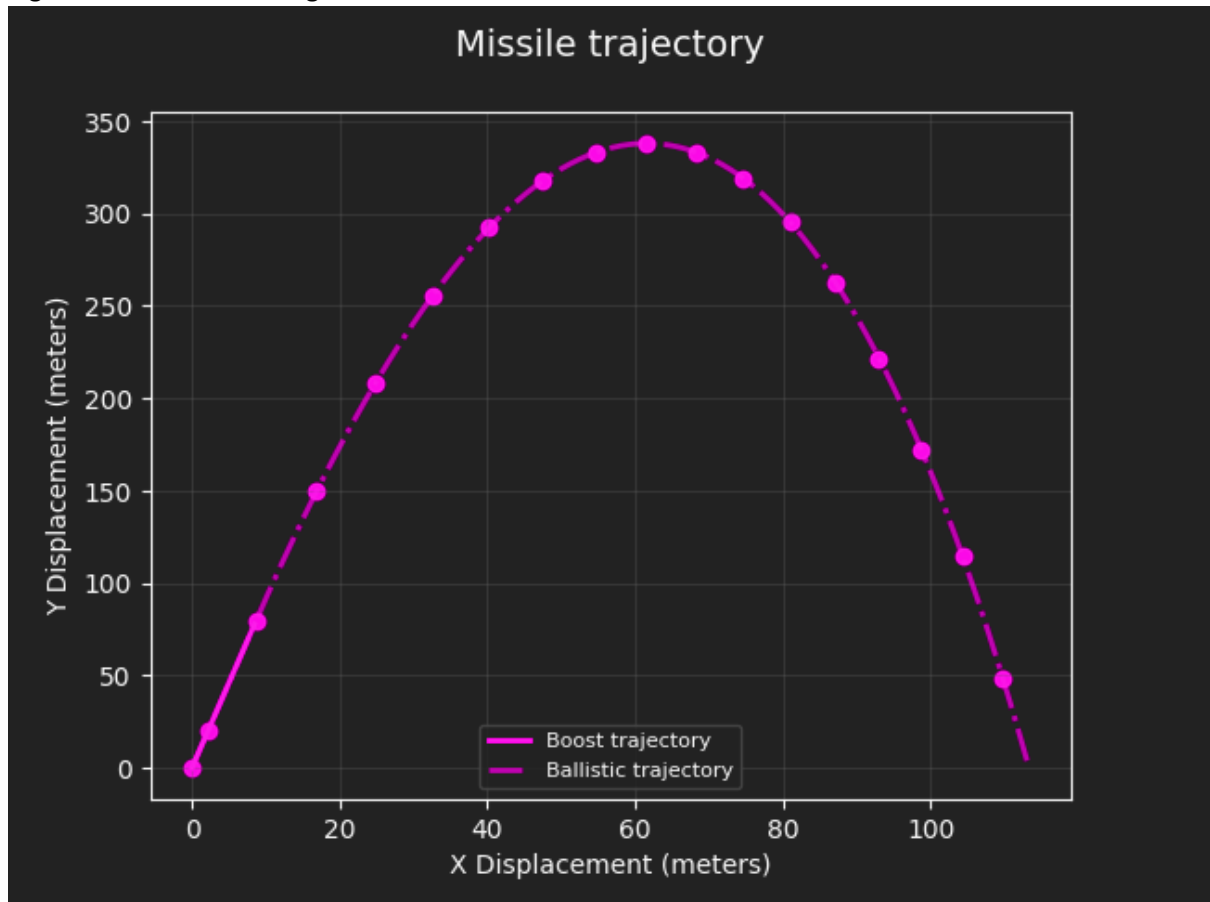
```

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

```

Termine simulazione

Il grafico risultante è il seguente:

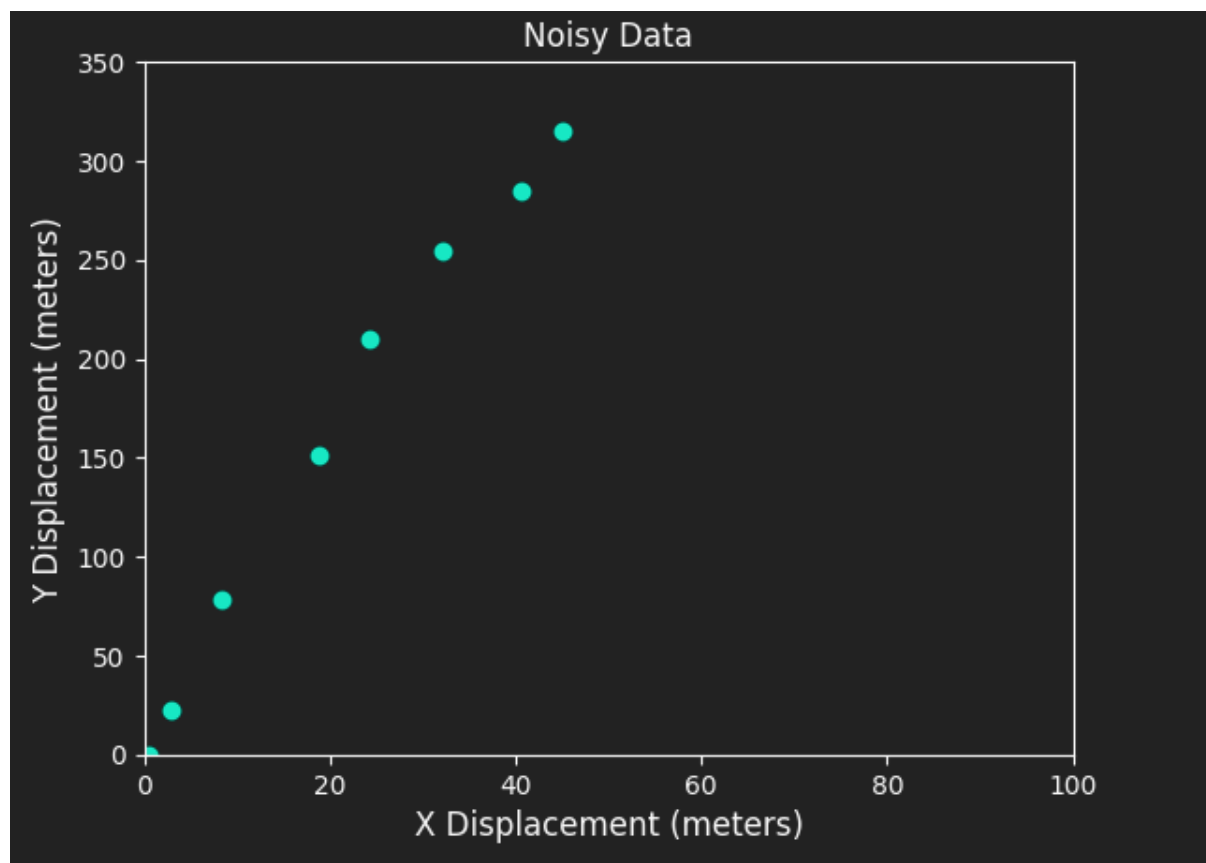


Completata la simulazione, il KBS ha una struttura del mondo su cui creare il dataset iniziale.

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 discostino 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 (in forma compatta) e per eventuali estensioni future di questo progetto o 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 (6)

Sebbene simili, ANN (*Artificial Neural Network*) (7) e PINN (*Physical-Informed Neural Network*) 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 abbreviati 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 quattro 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 (o errore), ad esempio MSE (Mean Squared Error, *Errore quadratico medio*, o MSL, Mean Squared Loss) (8) o Entropia Incrociata (*Cross Entropy*) (9).

In questo progetto sono state adottate le PINN che si differenziano dalle ANN tradizionali per il loro dominio di applicazione, 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 (10)

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. Un livello di input;
- II. Due livelli nascosti che fanno uso di una funzione di attivazione detta *GELU()*;
- III. Un 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) (11) 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.

La funzione GELU è così definita:

$$GeLU(x) = xP(X \leq x) = x\phi(x) = x \cdot \frac{1}{2} \left[1 + \operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) \right]$$

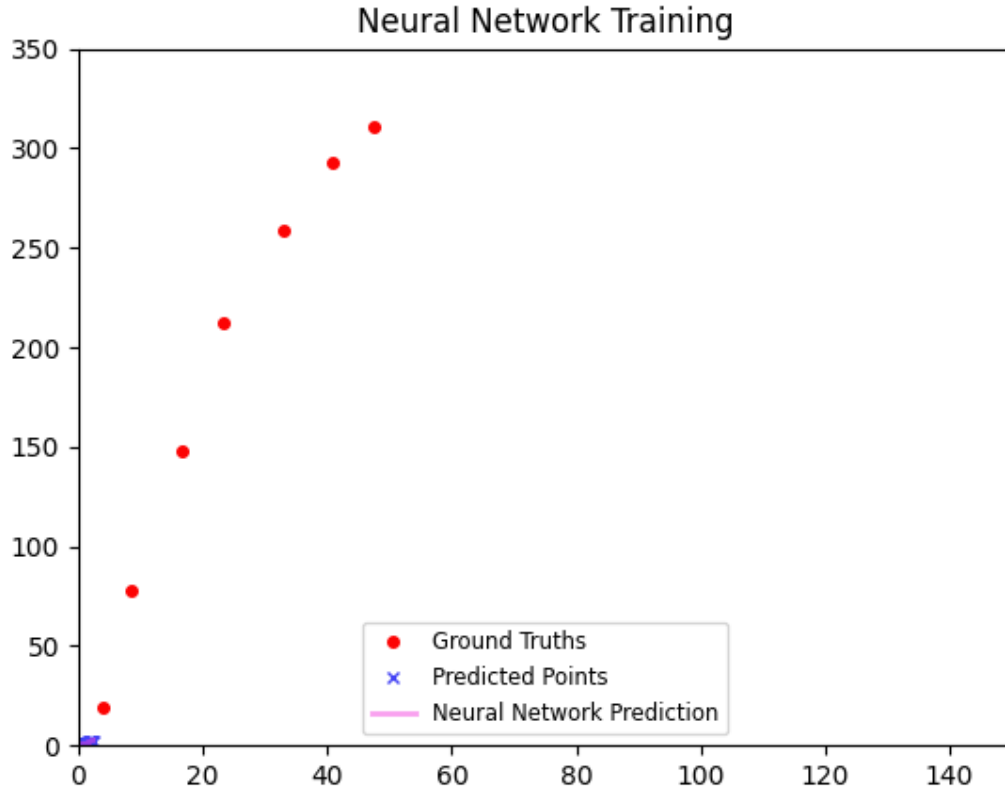
dove $\operatorname{erf} \left(\frac{x}{\sqrt{2}} \right)$ è la funzione di errore. A causa dell'elevato costo computazionale e tempo richiesto per calcolarla, essa risulterà più pesante da calcolare e dunque può essere approssimata attraverso la sigmoide o tangente iperbolica:

- I) $0.5x \left(1 + \tanh \left[\sqrt{\frac{2}{\pi}} (x + 0.044715x^3) \right] \right)$
- II) $x \cdot \operatorname{sigmoid}(1.702x)$

Tuttavia, in questo progetto verrà utilizzata la funzione d'errore:

$$\operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) = \frac{2}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} e^{-t^2} dt$$

Di seguito si mostra graficamente il risultato della predizione del learner se fosse stata applicata la ReLU:



Layer nascosti con ReLU

Si può notare come la traiettoria del missile risulti più “spigolosa” e quindi non assumibile secondo l’oggetto di questo progetto.

Come ottimizzatore è stato scelto **adam** (adaptive moments, *momentum adattivi*) (12) 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*) (13) 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 e output sono state convertite in tensori che integrano il calcolo con GPU.

Addestramento e Valutazione

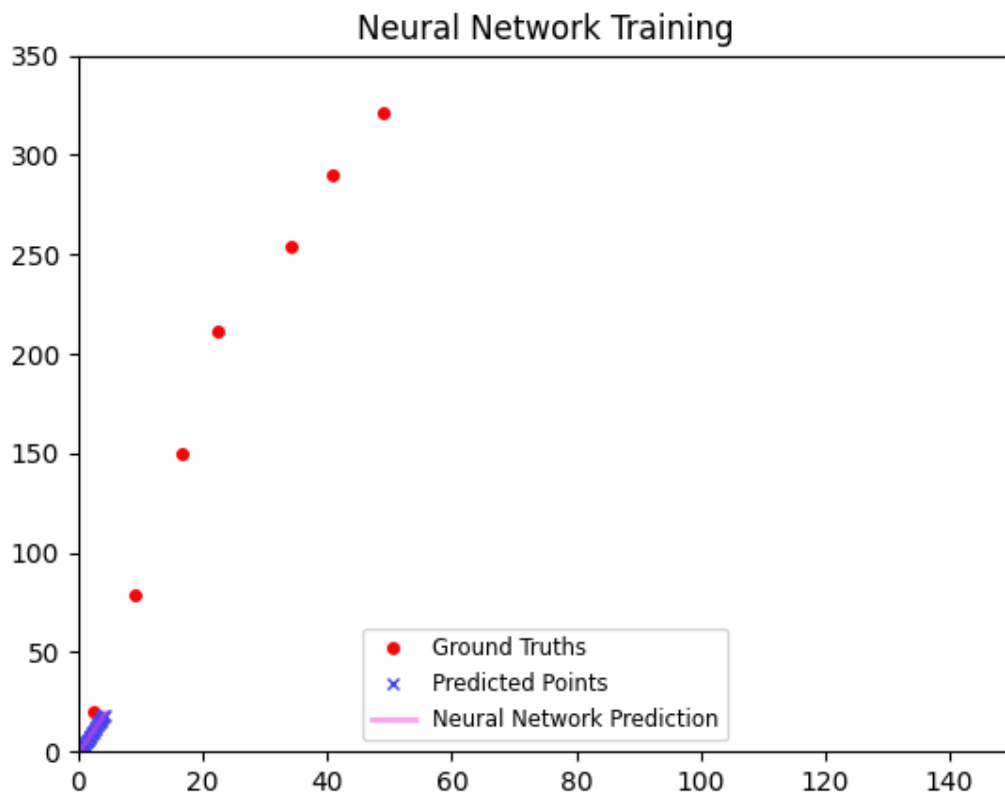
La PINN, ogni epoca, addestra le valutazioni sulle predizioni (*train_one_epoch*) e ad una fissata frequenza le registra (*train_and_record*) in un array che verrà poi utilizzato per visualizzare i risultati su un grafico.

```
11 def train_one_epoch(model, inputs, targets, optimizer):  # CittàX
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 sovradattamento e scarsa estrapolazione, risultando quindi in un eccesso di fiducia.

La curva si adatta troppo ai punti di training e questo causerà una scarsa capacità di generalizzazione. Inoltre, al di fuori della regione di training, le predizioni sono scorrette; violano leggi fisiche e vincoli, ne risulta quindi che i dati non sono utilizzabili. Dunque, la PINN prevede che il missile continui a muoversi sempre più in alto. Sebbene questo sia assumibile secondo i dati di training, da un punto di vista cinematico è scorretto e improbabile.

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 (14) 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:

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

Dove:

1. C_t è la resistenza aerodinamica (*drag*),
2. C_d è la spinta (*thrust*),
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 e 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, $step(thrust)$ (15) 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, risolvendo il problema della differenziabilità:

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

Dove:

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

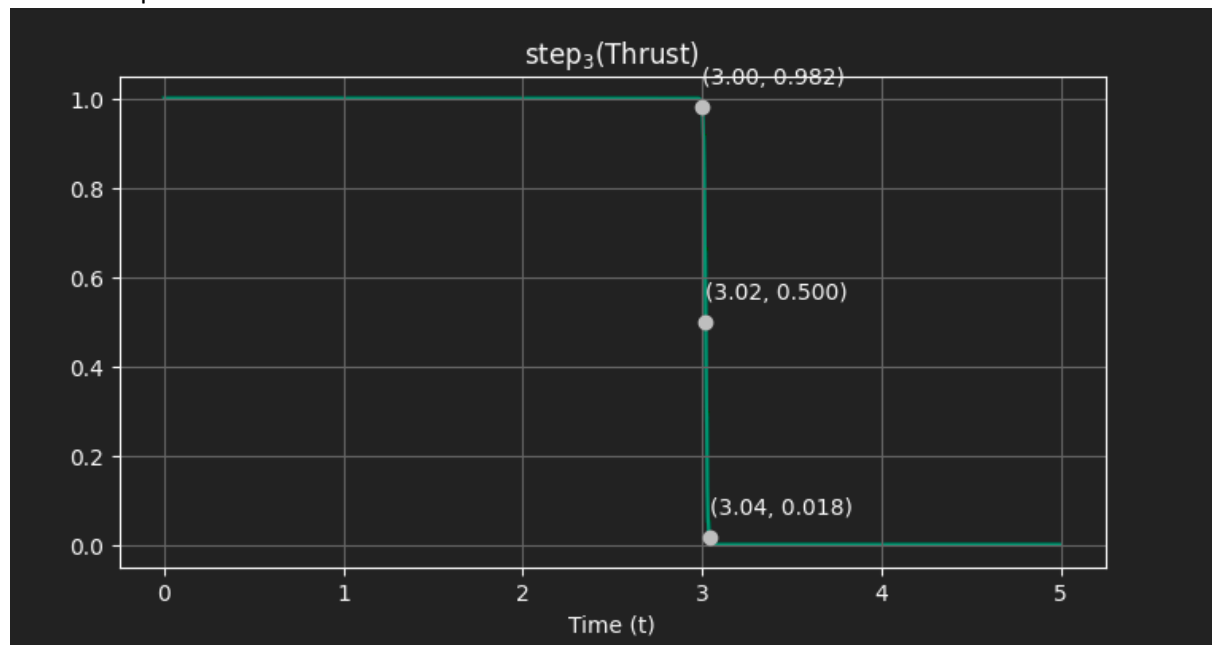
k, ϵ sono stati scelti attraverso dei test empirici.

```

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 d'errore fisica è così definita:

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

L_{fisica} necessita delle derivate prime e seconde (dove le derivate prime descrivono la velocità lungo gli assi, mentre le derivate seconde descrivono l'accelerazione lungo gli assi)

```

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      # Generate physics constants and convert them to tensors
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      drag_coefficient = torch.tensor(missile.drag_coefficient, dtype=torch.float, requires_grad=False)
30      thrust_magnitude = torch.tensor(missile.initial_thrust, dtype=torch.float, requires_grad=False)
31      mass = torch.tensor(missile.mass, dtype=torch.float, requires_grad=False)
```

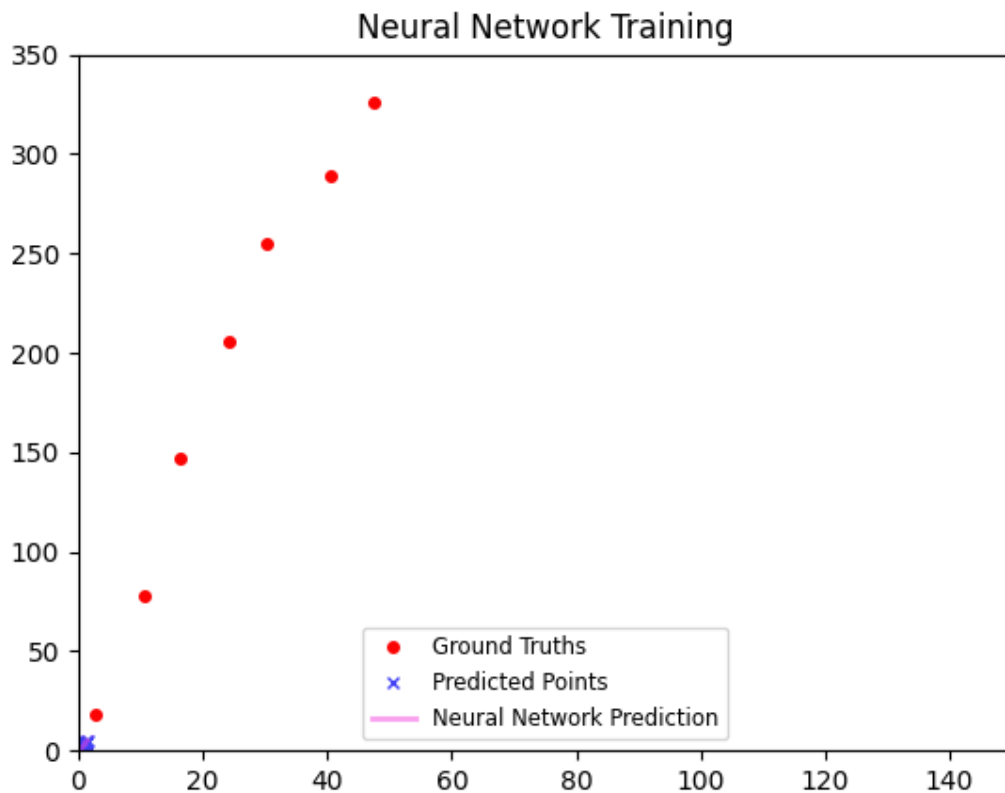
Infine, viene calcolata L_{fisica} :

```
47      # Calculate the final physics loss function starting with x-axis and y-axis
48      x_axis = (thrust_term * np.cos(launch_angle) - drag_coefficient * dx_dt) / mass
49      y_axis = (thrust_term * np.sin(launch_angle) - drag_coefficient * dy_dt - mass * gravity) / mass
50
51      physics_loss = ((d2x_dt2 - x_axis) ** 2 + (d2y_dt2 - y_axis) ** 2).mean()
52
```

La PINN viene adattata all'addestramento con la funzione di loss L_{fisica} :

```
16 # Arrange config settings
17 training_conf["epochs"] = 3000
18 training_conf["anim_record_freq"] = 10
19 training_conf["learning_rate"] = 0.005
20
21 # Reinitialize network, optimizer, input and target tensors
22 model = nn.Sequential(
23     nn.Linear(input_dim, hidden_dim),
24     nn.GELU(),
25     nn.Linear(hidden_dim, hidden_dim),
26     nn.GELU(),
27     nn.Linear(hidden_dim, output_dim)
28 )
29
30 optimizer = optim.Adam(model.parameters(), lr=training_conf["learning_rate"])
31 input_tensor = torch.tensor(inputs, dtype=torch.float).view(-1, 1)
32 target_tensor = torch.tensor(targets, dtype=torch.float).view(-1, 2)
33
34
35 def train_one_epoch_supervised(model, inputs, targets, optimizer): 
36     """
37     A remake of train_one_epoch function of training.py.
38     This uses mean squared error and physics loss.
39     """
40     # Get predictions from NN
41     predictions = model(inputs)
42
43     # Mean squared loss
44     criterion = nn.MSELoss(reduction="sum")
45     mse_loss = criterion(predictions, targets)
46
47     # Physics loss
48     physics_loss = compute_physics_loss(model, missile)
49
50     # Combined loss
51     loss = mse_loss + physics_loss
52
53     # Backward pass
54     optimizer.zero_grad()
55     loss.backward()
56     optimizer.step()
57
58
59 # Domain over which NN will be evaluated
60 dom_tensor = gen_eval_domain(30, 300)
61
62 # Train NN and record predictions over entire domain for visualization
63 eval_pred = train_and_record(model, optimizer, input_tensor, target_tensor, dom_tensor,
64                             training_conf["epochs"],
65                             training_conf["anim_record_freq"], train_one_epoch_supervised)
66
67 # Animation
68 anim = show_animation(eval_pred, targets, training_conf)
```


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



È stato così risolto il sovraddattamento e migliorata la capacità di estrapolazione della PINN.

Apprendimento di parametri sconosciuti

Fino ad ora si è assunta la conoscenza a priori di tutte le costanti fisiche ma questo non è realistico. Nella realtà è molto probabile non avere accesso a informazioni come la massa del missile, intensità della spinta, coefficiente di resistenza aerodinamica o l'angolo di lancio. Le NN sono in grado di apprendere parametri sconosciuti. In particolare, la PINN del progetto imparerà direttamente dai dati i parametri sconosciuti, il che rappresenta uno strumento importante nel trattare dati rumorosi o osservazioni incomplete.

Adattamento della PINN alla non disponibilità di dati

Vengono dunque inizializzati i parametri sconosciuti per renderle addestrabili nella rete. In questo modo verranno adattati ai pesi e bias della NN.

Si suppone quindi di conoscere il modello del missile, e quindi la massa, e calcolare l'angolo di lancio.

In questo modello del mondo, la PINN dovrà ricavare il coefficiente di aerodinamicità (*drag_coefficient*) e il coefficiente di spinta missilistica (*thrust_coefficient*).

Dunque, vengono rielaborati tutti i calcoli relativi a questi parametri:

```
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}
```

Lo snippet di codice alla riga 31 affronta il problema delle diverse scale su cui sono definiti i due parametri che affligge i metodi di ottimizzazione basati su gradienti.

Vengono quindi fornite due inizializzazioni che fungeranno da coefficienti da moltiplicare ai corrispondenti parametri durante la computazione della fisica.

Viene riadattato l'ottimizzatore ADAM con la differenza che esso aggiornerà anche i due parametri.

Sarebbe stato possibile fornire un'inizializzazione casuale ai due parametri ma per ragioni di velocità di convergenza sono stati scelti due valori arbitrari.

```
43  optimizer = optim.Adam([{"params": model.parameters(), "lr": training_conf["learning_rate"]},
44                           {"params": learnable_constants, "lr": training_conf["learning_rate"] / 10}]
45  )
```

Anche il calcolo della funzione L_{fisica} è stata rielaborata, adattandola alla non conoscenza dei due parametri:

```
7 def compute_physics_loss(model, constants, constant_scale_factors, physics_weight=10):  # CittiX
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
22     # Mean squared loss term for initial position and initial speed, both should be [0, 0]
23     initial_pos_loss = x_pred[0] ** 2 + y_pred[0] ** 2
24     initial_speed_loss = dx_dt[0] ** 2 + dy_dt[0] ** 2
25
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)
30
31     # Unknown parameters with scale factors
32     drag_coefficient, thrust_magnitude = constants
33     drag_scale_factor = constant_scale_factors["drag_coefficient"]
34     thrust_scale_factor = constant_scale_factors["thrust_magnitude"]
35
36     def compute_thrust(t, thrust_cease=missile.thrust_duration + 1, sharpness=200, offset=0.02):  # CittiX
37         """
38         This is a reworked version of training.py function to be better adapted to torch library
39         :param t: Time
40         :param thrust_cease: The time at which thrust ceases
41         :param sharpness: Constant that monitors the step sharpness
42         :param offset: Position corrector for the step
43         :return: A differentiable step function
44         """
45         return 1 / (1 + torch.exp(-sharpness * (thrust_cease - t + offset)))
46
47     # Compute thrust for time inputs
48     thrust_term = thrust_scale_factor * thrust_magnitude * compute_thrust(dom_tensor)
49
50     # Calculate the final physics loss function starting with x-axis and y-axis
51     x_axis = (thrust_term * torch.cos(launch_angle) - drag_scale_factor * drag_coefficient * dx_dt) / mass
52     y_axis = (thrust_term * torch.sin(launch_angle) - drag_scale_factor * drag_coefficient * dy_dt - mass * gravity) / mass
53
54     physics_loss = ((d2x_dt2 - x_axis) ** 2 + (d2y_dt2 - y_axis) ** 2).mean()
55
56     # Add initial position losses and return total loss multiplied by lambda weight
57     return physics_weight * (physics_loss + initial_pos_loss + initial_speed_loss)
```

La PINN verrà addestrata seguendo gli stessi principi, tenendo conto anche dei due parametri non osservati in precedenza:

```
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()
72
73
74     # Domain over which NN will be evaluated
75     dom_tensor = gen_eval_domain(30, 300)
76
77     # Train NN and record predictions over entire domain for visualization
78     eval_pred = train_and_record(model, optimizer, input_tensor, target_tensor, dom_tensor,
79                                training_conf["epochs"],
80                                training_conf["anim_record_freq"], train_one_epoch_supervised)
81
82     drag_param_pred = learnable_constants[0].item() * constant_scale_factors["drag_coefficient"]
83     thrust_param_pred = learnable_constants[1].item() * constant_scale_factors["thrust_magnitude"]
```

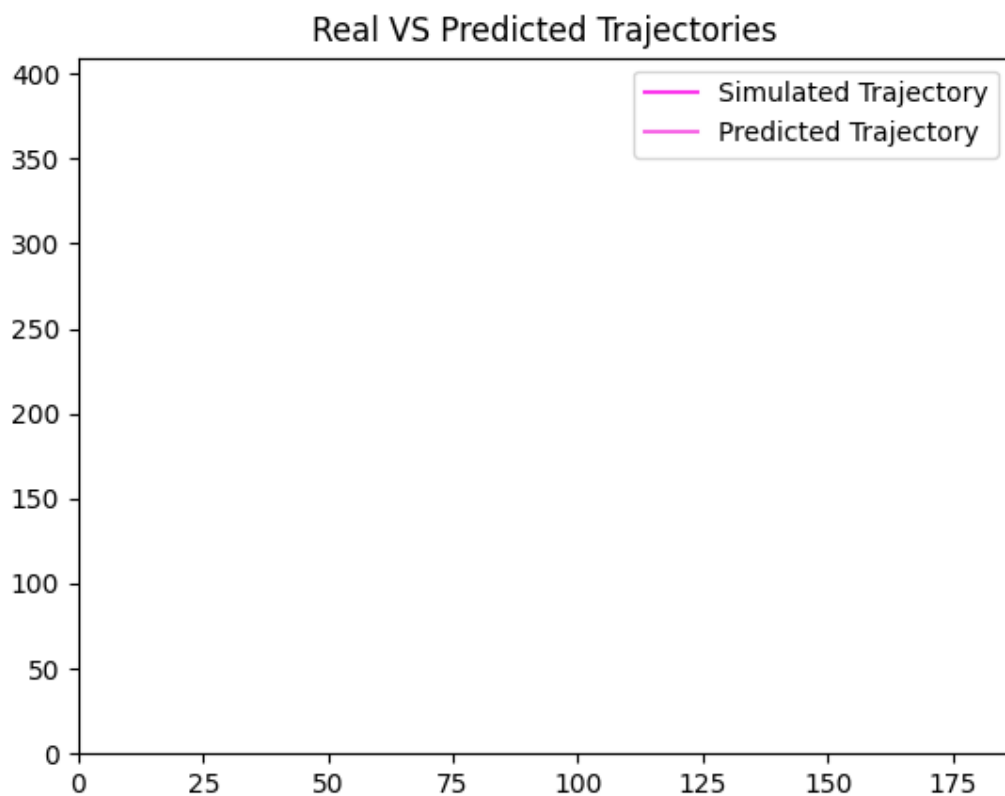
Vengono inoltre monitorate le stime effettuate dalla PINN rispetto alle ground truth:

```
85 print(f"Predicted drag force: {drag_param_pred:.2f}\\tActual drag force: {missile.drag_coefficient}")
86 print(f"Predicted thrust: {thrust_param_pred:.2f}\\tActual thrust force: {missile.initial_thrust}")
```

```
Predicted drag force: 1.53   Actual drag force: 3
Predicted thrust: 2900.62   Actual thrust force: 5000
```

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.

Eventuali estensioni

Il progetto può essere esteso facendo comparire casualmente un oggetto volante, identificabile come *Neutral*, *Dangerous*, *Friendly*, e far sì che il modello in questione sia in grado di effettuare una scelta e intervenire (qualora sia necessario) obliterando l'oggetto prevedendo e calcolando una buona balistica che tenga conto delle posizioni che il target compie nello spazio e nel tempo.

Elenco argomenti di interesse

- ❖ 7.3.1 Valutare le Predizioni

- ❖ 7.2 Apprendimento: Problematiche
- ❖ 7.4.2 Regressione e Classificazione Lineari
- ❖ 7.5 Sovradattamento
- ❖ 7.5.2 Regolarizzazione
- ❖ 8.1 Feature Engineering
- ❖ 8.2.1 Reti Neurali Feed Forward
- ❖ 8.2.2 Apprendimento dei Parametri
- ❖ 8.3.1 Migliorare l'Ottimizzazione
- ❖ 8.3.2 Migliorare la Generalizzazione
- ❖ 3.10.3 Domini Continui: Discesa di Gradiente

Riferimenti

1. **David L. Poole. Alan K. Mackworth.** What is Artificial Intelligence? *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024, p. 3-4.
2. **David L. Poole. Alan K. Mackworth.** Knowledge Base. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024, p. 179.
3. **David L. Poole. Alan K. Mackworth.** Neural Networks and Deep Learning. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024, p. 327.
4. **David L. Poole. Alan K. Mackworth.** Supervised Learning Foundations. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024, p. 266.
5. **Resistenza Fluidodinamica.** *Wikipedia*. [Online] https://it.wikipedia.org/wiki/Resistenza_fluidodinamica.
6. **Physics-informed neural networks.** *Wikipedia*. [Online] https://en.wikipedia.org/wiki/Physics-informed_neural_networks.
7. **Mackworth, David L. Poole. Alan K.** Neural Networks and Deep Learning. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024, p. 328.
8. **David L. Poole. Alan K. Mackworth.** Evaluating Predictions - The squared loss (L2 loss). *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024, p. 270.
9. **David L. Poole. Alan K. Mackworth.** Information Theory. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024.
10. **David L. Poole. Alan K. Mackworth.** Feedforward Neural Networks. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024.
11. **Dan Hendrycks, Kevin Gimpel.** *Gaussian Error Linear Units (GELUs)*. s.l. : arxiv.org, 2023.

12. Mackworth, David L. Poole. Alan K. Adam. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024.

13. —. Stochastic Gradient Descent. *Artificial Intelligence Foundations of Computational Agents, 3rd Edition-Cambridge University Press (2024)*. s.l. : Cambridge University Press, 2024.

14. Principi della Dinamica. *Wikipedia*. [Online]
https://it.wikipedia.org/wiki/Principi_della_dinamica#Secondo_principio.

15. Heaviside, Oliver. Funzione gradino di Heaviside. *Wikipedia*. [Online] Funzione gradino di Heaviside.

Strumenti utilizzati

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

Guida all'installazione

Clonare il repository dal link della prima pagina e importare il progetto su PyCharm (o qualsiasi IDE che supporti l'integrazione di Jupyter Notebook).

Assenza del notebook Jupyter

Il file .ipynb è già incluso nella cartella del progetto, tuttavia è possibile crearne uno ex novo aprendo un nuovo notebook di Jupyter e aggiungere le seguenti celle di codici:

```
1 from IPython.display import HTML
2 from matplotlib.animation import PillowWriter
3
4 from Architecture.Supervised_learning.training_visualizer import anim
5
6 %matplotlib inline
7
8 HTML(anim.to_jshtml())
9 anim.save("MNtraining.gif", writer=PillowWriter(fps=10))
10
11 from IPython.display import HTML
12 from matplotlib.animation import PillowWriter
13 from Architecture.Supervised_learning.supervised_training import anim
14
15 %matplotlib inline
16
17 anim.save("MNtraining_loss.gif", writer=PillowWriter(fps=20))
18 HTML(anim.to_jshtml())
19
20 from IPython.display import HTML
21 from Architecture.Supervised_learning.supervised_training_visualizer import anim
22
23 %matplotlib inline
24
25 anim.save("final.gif", writer=PillowWriter(fps=20))
26 HTML(anim.to_jshtml())
```

Dove la prima cella si occupa dell'addestramento con sovradattamento e scarsa estrapolazione, la seconda invece si occupa dell'addestramento supervisionato con funzione di loss fisica, mentre la terza cella si occupa semplicemente di mettere a confronto balistica effettiva e predetta.

Dal notebook è possibile avviare in maniera indipendente ogni cella oppure tutte insieme come unico blocco di codice. A prescindere dalla modalità di esecuzione, verranno generati tutti i grafici necessari e mostrati in questo progetto, ossia il grafico dei dati simulati, il grafico dei dati rumorosi e il grafico della funzione a gradini approssimata.