Attacchi avversari con l'algoritmo evoluzione differenziale

Raffaele Russo, Alessandro Vanacore

Indice

1 Introduzione

Il Deep Learning ha impattato numerosi aspetti della società moderna, apportando progressi decisivi in problemi irrisolti nell'intelligenza artificiale. Le tecniche tradizionali di Machine Learning richiedono l'estrazione manuale e onerosa di feature dai dati. Inoltre, la scelta delle feature discriminanti da estrarre può essere complessa e limitata a un problema specifico. L'idea chiave del Deep Learning è fare in modo che sia un algoritmo di apprendimento a estrarre la migliore rappresentazione dei dati.

La disponibilità di enormi quantità di dati, i progressi in hardware, software e computazione parallela hanno reso possibile l'addestramento di reti con milioni di parametri in poche ore. Le reti neurali convoluzionali, in particolare, hanno contribuito allo stato dell'arte nell'ambito della computer vision in attività come classificazione di immagini, segmentazione e rilevamento di oggetti, riuscendo rispetto ad altre architetture a trarre vantaggio dalla correlazione tra pixel adiacenti.

Sono stati progettati modelli sempre più complessi e accurati, trascurandone inizialmente la robustezza. Questo fin quando alcuni ricercatori [1] hanno scoperto come in realtà fosse piuttosto semplice ingannare una rete neurale, inserendo nelle immagini in ingresso una quantità di rumore impercettibile per un osservatore umano. Da quando i modelli di Deep Learning hanno eguagliato lo stato dell'arte in molte applicazioni, la robustezza e la sicurezza delle reti è diventata la priorità dei ricercatori. Numerosi studi hanno infatti mostrato come la maggior parte delle reti neurali sia vulnerabile ad attacchi avversari. Un attacco avversario consiste nel generare un esempio avversario, cioè un immagine modificata ad hoc per essere classificata erroneamente dalla rete target con un elevato livello di confidenza. In molti casi la nuova immagine non è distinguibile a occhio nudo da quella originale, ma è comunque tale da portare il modello a una predizione errata. Inizialmente si credeva che tali vulnerabilità fossero dovute a una condizione di overfitting, cioè all'utilizzo di un modello con regioni di decisioni troppo complesse e irregolari per i dati di training a disposizione. Come conseguenza si ottiene un algoritmo che non è in grado di generalizzare su nuovi dati e le cui predizioni sul test-set sono soggette a errori randomici, che possono essere sfruttati dall'attaccante. Se la causa fosse stata realmente l'overfitting, ogni esempio avversario sarebbe un caso isolato e unico. Scegliendo un diverso modello ci aspetteremmo di commettere differenti errori sul test-set.

In realtà Goodfellow et al. in [1] mettono in luce l'aspetto critico della trasferibilità degli attacchi. Esempi avversari, generati opportunamente per funzionare su un certo modello, restano avversari anche per altri modelli sconosciuti. Ciò è da associare al fatto che modelli differenti, addestrati sugli stessi dati, tendono comunque a formare regioni di decisione simili tra loro. Le vulnerabilità delle reti sono quindi da attribuire a fenomeni sistematici, piuttosto che randomici, e ciò deriva dall'eccessiva linearità dei modelli, e dunque una condizione di underfitting piuttosto che di overfitting.

2 Evoluzione differenziale

L'algoritmo evoluzione differenziale (DE) è un'euristica utilizzata per risolvere complessi problemi di ottimizzazione, introdotta per la prima volta in [2]. Si tratta di un algoritmo evolutivo, che usa meccanismi ispirati alla teoria dell'evoluzione. Gli individui migliori di una popolazione sono quelli che producono più figli, che a loro volta ereditano le caratteristiche dei genitori. Questo fa in modo che la popolazione tenda a migliorare nel tempo, generazione dopo generazione. Ciò è possibile attraverso differenti meccanismi presenti in natura, quali mutazione, ricombinazione e selezione. L'idea è quella di considerare una popolazione di soluzioni che evolve in accordo con un meccanismo di selezione per produrre soluzioni con buoni valori di funzione obiettivo.

L'algoritmo si articola nei seguenti passi:

1. Si inizializza in maniera randomica una popolazione di NP individui, ognuno di essi caratterizzato da un insieme di geni rappresentanti i parametri della soluzione.

2. Per ogni elemento della popolazione, denominato individuo target R1, si procede con il meccanismo di mutazione scegliendo casualmente tre individui della popolazione $R2$, $R3$ e $R4$ (purchè diversi da $R1$). Lavorando su questi ultimi si ottiene l'individuo mutante U:

   $$U = R2 + F(R3 - R4)$$

   Il fattore F, detto costante di mutazione, descrive di quanto il vettore mutante differisce rispetto agli individui selezionati.

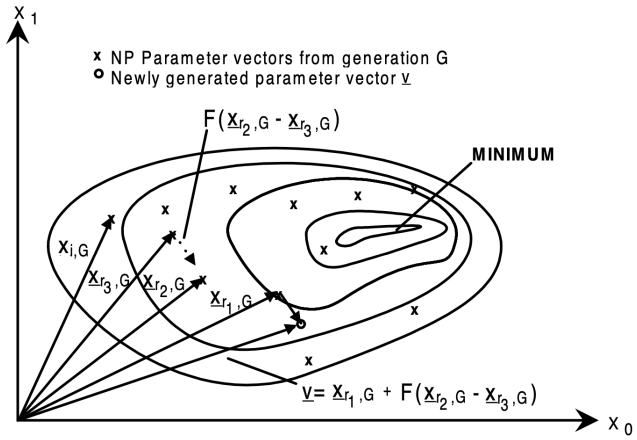

Fig. 1: Generazione vettore DE

3. Avendo ottenuto a questo punto il vettore U, è possibile effettuare il passo di crossover, utilizzando l'individuo target $R1$. L'idea è quella di considerare gli individui U e $R1$ come due genitori, a partire dai quali, tramite

l'operatore di crossover, viene determinata la quantità di informazione da ereditare da ognuno di essi. Ogni gene dell'individuo figlio *trial* è ottenuto confrontando il crossover rate (CR) con numero casuale tra 0 e 1. Se quest'ultimo è più grande del CR allora viene ereditato il gene dell'individuo target $R1$, viceversa il figlio lo erediterà dall'individuo mutante U.

4. Il passaggio finale prevede di calcolare il valore della funzione obiettivo in corrispondenza dell'individuo target $R1$ e dell'individuo ottenuto tramite crossover. Se *trial* è associato a un valore migliore della funzione obiettivo prenderà il posto dell'individuo target $R1$ nella generazione successiva.

   L'algoritmo continua finchè non viene raggiunto il numero massimo di generazioni o in caso anticipato se viene soddisfatto un criterio di successo.

3 Implementazione DE

```
def de(f,len_x,lb_x,ub_x,NP=10,F=0.5,CR=0.5,it=100):

    x = np.random.uniform(low=lb_x, high=ub_x, size=(NP,len_x))
    u = np.zeros(len_x)
    trial = np.zeros(len_x)
    opt = sys.maxsize
    x_opt = np.zeros(len_x)*sys.maxsize

    for k in range(it):
        for i in range(NP):
            r = random.sample([k for k in range(NP) if k != i], 3)
            r1 = r[0]
            r2 = r[1]
            r3 = r[2]
            u = x[r1] + F * (x[r2]-x[r3])
            u = np.clip(u, lb_x, ub_x)
            idx = np.random.rand(len_x)
            for id in range(len(idx)):
                if idx[id] < CR:
                 trial[id] = u[id]
                else:
                 trial[id] = x[i][id]
            if(f(trial)<f(x[i,:])):
                 x[i] = trial
            curr = f(x[i])
            if(curr<opt):
               opt = curr
               x_opt = x[i]

    return opt,x_opt
```

4 Esempio applicazioni

4.1 First De Jong Function

$$\min_x \left(f_1(x) = \sum_{i=0}^{2} x_i^2 \right)$$

s.t.

$$x_i \in [-100, 100]$$

dove

- Valore ottimo: $z^* = 0$
- Soluzione ottima: $x^* = [0, 0, 0]$

$OUTPUT$:

- $Z = 0$
- $X = [0 \quad 0 \quad 0]$

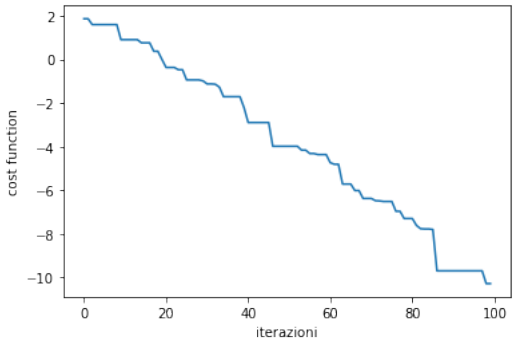

Fig. 2: Log10 della funzione obiettivo al variare delle iterazioni

4.2 Second De Jong Function

$$\min_x f_2(x) = 100 \times (x_0^2 - x_1)^2 + (1 - x_0)^2$$

s.t.

$$x_i \in [-2.048, 2.048]$$

dove:

- Valore ottimo: $z^* = 0$
- Soluzione ottima: $x^* = [1 \quad 1]$

$OUTPUT$:

- $Z = 0$
- $X = [1 \quad 1]$

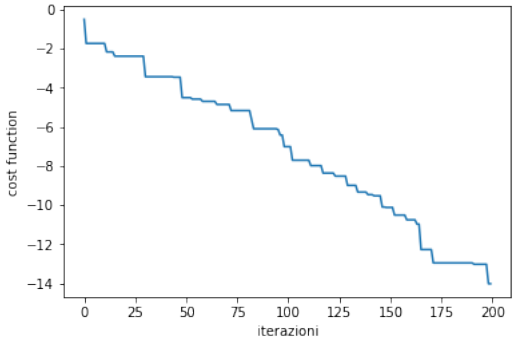

Fig. 3: Log10 della funzione obiettivo al variare delle iterazioni

4.3 Third De Jong Function

$$\min_x \left(f_3(x) = 30 + \sum_{i=0}^{4} \lfloor x_i \rfloor \right)$$

s.t.

$$x_i \in [-5.12, 5.12]$$

dove:

- Valore ottimo: $z^* = 0$
- Soluzione ottima: $x_i^* = -5 - \epsilon$ con $\epsilon \in [0, 0.12]$

$OUTPUT$:

- $Z = 0.0$
- $X = [-5.12 \quad -5.12 \quad -5.12 \quad -5.12 \quad -5.12]$

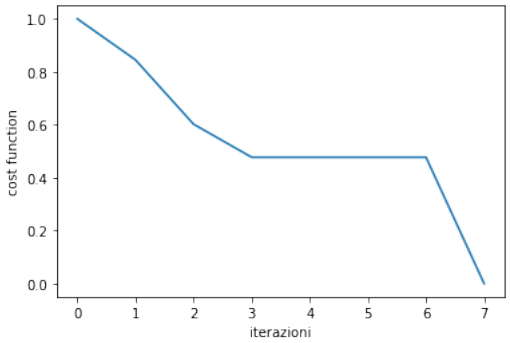

Fig. 4: Log10 della funzione obiettivo al variare delle iterazioni

4.4 Ackley Function

$$\min_x f4(x) = -20exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi x_i)\right) + 20 + e$$

con

$$x_i \in [-32.768, 32.768]$$

dove:

- Valore ottimo: $z^* = 0$
- Soluzione ottima: $x_i^* = 0$

$OUTPUT$ per $n = 20$:

- $Z = 0.0$
- $X = [0...0]$

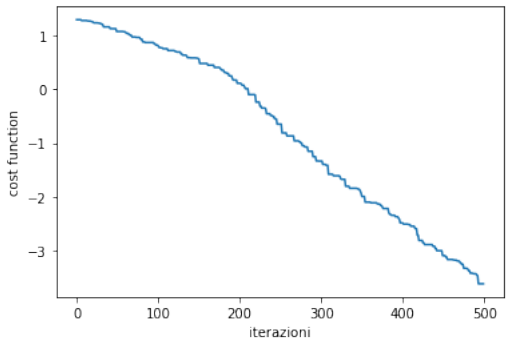

Fig. 5: Log10 della funzione obiettivo al variare delle iterazioni

4.5 One Pixel Attack

One Pixel Attack è una tecnica di attacco, introdotta nel paper [3], che consente di generare esempi avversari modificando un solo pixel nell'immagine originale. La scelta del pixel e del tipo di modifica avviene mediante l'algoritmo di evoluzione differenziale, introdotto da Storn e Price negli anni 90. Si tratta di un attacco black-box, che richiede come unica informazione i livelli di confidenza delle predizioni della rete target. Inoltre, rispetto alla tecniche basate su calcolo del gradiente come FGSM, si rivela flessibile, potendo attaccare reti neurali convoluzionali non differenziabili o per le quali il calcolo del gradiente risulta complesso. L'obiettivo dell'attacco è generare un esempio avversario perturbando un pixel nell'immagine originale. Definiamo la perturbazione di un pixel come una tupla di tre parametri:

$$t = (x, y, z)$$

dove x e y rappresentano le coordinate del pixel da 0 a 27 mentre z rappresenta un livello di grigio pari a 0 o 1. Perturbazioni multiple possono essere ottenute come concatenazione di tuple:

$$X = (x_1, y_1, z_1, x_2, y_2, z_2, ...)$$

La ricerca dei pixel che porteranno al successo dell'attacco è un problema di ottimizzazione, che risolviamo con l'algoritmo di evoluzione differenziale. Viene quindi inizializzata una popolazione casuale di n perturbazioni:

$$P = (X_1, X_2, ..., X_n)$$

Per ciascuna iterazione calcoliamo n nuovi figli mutanti usando la formula:

$$X_i = X_{r_1} + F(X_{r_2} - X_{r_3})$$

dove $r1 \neq r2 \neq r3$.
$r1$, $r2$, $r3$ sono indici casuali nella popolazione P e F è un parametro di mutazione.
Quindi prendiamo tre perturbazioni casuali dalla precedente generazione e le ricombiniamo per formare una nuova soluzione candidata.

La funzione costo utilizzata restituisce il livello di confidenza dell'immagine modificata con la perturbazione casuale. Vogliamo perturbare un'immagine in modo da minimizzare il livello di confidenza della classe corretta, massimizzando quindi quello dell'altra classe. Se il nuovo candidato X_i è associato a un valore più basso della funzione costo, sostituiamo il vecchio X_i con questo nuovo. Tale processo prosegue per un numero massimo di iterazioni o finchè non viene soddisfatto un criterio di successo, cioè quando troviamo una perturbazione che porta il livello di confidenza della classe iniziale corretta al di sotto di quello di un'altra classe, facendo così sbagliare il modello.

5 Risultati sperimentali One Pixel Attack

5.1 Dataset

Il dataset utilizzato è memorizzato in un file CSV, costituito da 5517 righe di 785 elementi. I primi 784 di essi assumono un valore pari a 0 o 1, relativo a livello di grigio di un'immagine 28×28 pixel, rappresentante un numero da 0 a 9 come in figura 6. L'ultimo elemento assume un valore da 0 a 9, ossia la classe corretta dell'immagine considerata.

Fig. 6: Esempi dataset

5.2 Risultati ottenuti

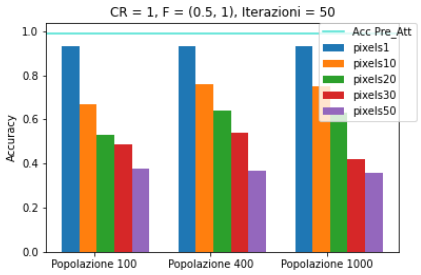

Fig. 7: Andamento dell'accuracy post attacco al variare dei pixel modificati

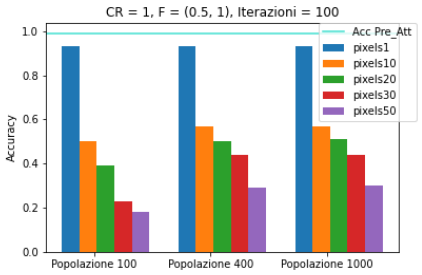

Fig. 8: Andamento dell'accuracy post attacco al variare dei pixel modificati

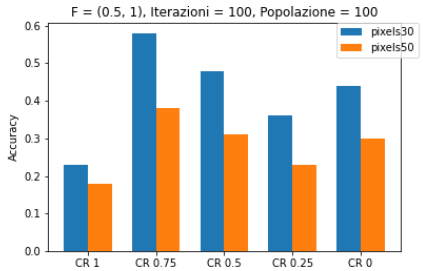

Fig. 9: Andamento dell'accuracy post attacco al variare del parametro CR

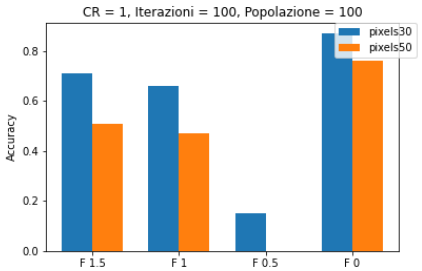

Fig. 10: Andamento dell'accuracy post attacco al variare del parametro F

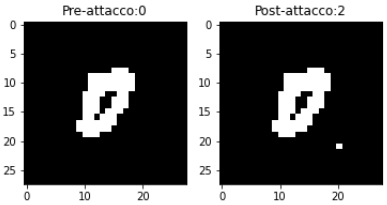

Fig. 11: 1 pixel

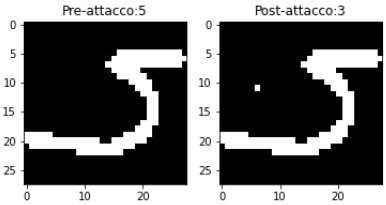

Fig. 12: 1 pixel

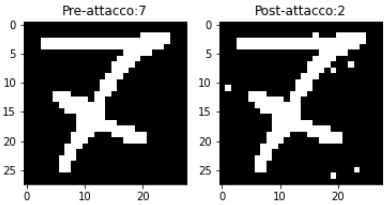

Fig. 13: 10 pixel

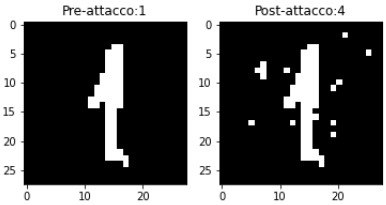

Fig. 14: 20 pixel

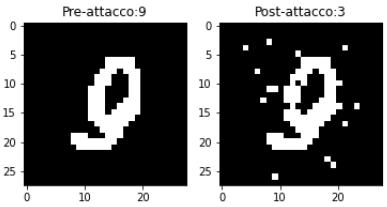

Fig. 15: 30 pixel

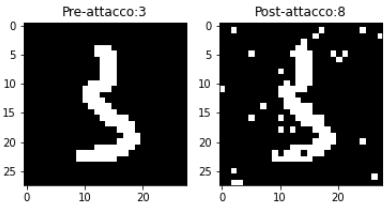

Fig. 16: 50 pixel

Pixels	Accuracy	Loss
1	0.93	0.106
10	0.67	0.458
20	0.53	0.642
30	0.49	0.804
50	0.38	1.07

Tab. 1: CR = 1 F = (0.5,1) it = 50 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.107
10	0.76	0.45
20	0.64	0.666
30	0.54	0.813
50	0.37	1.149

Tab. 2: CR = 1 F = (0.5,1) it = 50 pop = 400

Pixels	Accuracy	Loss
1	0.93	0.11
10	0.75	0.473
20	0.63	0.698
30	0.42	1.135
50	0.36	1.25

Tab. 3: CR = 1 F = (0.5,1) it = 50 pop = 1000

Pixels	Accuracy	Loss
1	0.93	0.106
10	0.5	0.645
20	0.39	0.77
30	0.23	1.01
50	0.18	1.307

Tab. 4: CR = 1 F = (0.5,1) it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.11
10	0.57	0.621
20	0.50	0.709
30	0.44	0.925
50	0.29	1.315

Tab. 5: CR = 1 F=(0.5,1) it = 100 pop = 400

Pixels	Accuracy	Loss
1	0.93	0.11
10	0.57	0.616
20	0.51	0.866
30	0.44	0.977
50	0.3	1.33

Tab. 6: CR = 1 F = (0.5,1) it = 100 pop = 1000

Pixels	Accuracy	Loss
1	0.93	0.098
10	0.7	0.5
20	0.63	0.603
30	0.58	0.793
50	0.38	1.125

Tab. 7: CR = 0.75 F=(0.5,1) it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.101
10	0.73	0.447
20	0.62	0.632
30	0.48	0.802
50	0.31	1.155

Tab. 8: CR = 0.5 F = (0.5,1) it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.097
10	0.63	0.557
20	0.51	0.704
30	0.36	0.864
50	0.38	1.208

Tab. 9: CR = 0.25 F=(0.5,1) it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.106
10	0.68	0.467
20	0.55	0.601
30	0.44	0.748
50	0.3	0.999

Tab. 10: CR = 0 F = (0.5,1) it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.104
10	0.88	0.288
20	0.82	0.421
30	0.71	0.582
50	0.51	0.917

Tab. 11: CR = 1 F = 1.5 it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.105
10	0.84	0.302
20	0.83	0.407
30	0.66	0.602
50	0.47	0.976

Tab. 12: CR = 1 F = 1 it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.93	0.106
10	0.55	0.593
20	0.37	0.823
30	0.15	1.105
50	0.0	1.427

Tab. 13: CR = 1 F = 0.5 it = 100 pop = 100

Pixels	Accuracy	Loss
1	0.96	0.085
10	0.93	0.19
20	0.91	0.25
30	0.87	0.38
50	0.76	0.63

Tab. 14: CR = 1 F = 0 it = 100 pop = 100

Riferimenti bibliografici

[1] Ian J Goodfellow, Jonathon Shlens e Christian Szegedy. ≪Explaining and harnessing adversarial examples≫. In: *arXiv preprint arXiv:1412.6572* (2014).

[2] R. Storn e K. Price. ≪Differential evolution a simple and efficient adaptive scheme for global optimization over continu≫. In: *Journal of Global Optimization* (1997).

[3] Jiawei Su, Danilo Vasconcellos Vargas e Kouichi Sakurai. ≪One Pixel Attack for Fooling Deep Neural Networks≫. In: *IEEE Transactions on Evolutionary Computation* 23.5 (ott. 2019), pp. 828–841. ISSN: 1941-0026. DOI: 10.1109/tevc.2019.2890858. URL: http://dx.doi.org/10.1109/TEVC.2019.2890858.