Intelligenza Artificiale

Anno Accademico 2022 - 2023

Esercizi in Python:

Algoritmi Genetici

(caso discreto)

Norme di utilizzo dei materiali didattici

La visione e l'utilizzo del presente materiale didattico è riservato agli utenti iscritti al corso al solo fine di studio e approfondimento didattico.

L'utente che accede alla sezione e-learning e che ne consulta i contenuti è tenuto a rispettare le disposizioni legislative che tutelano il diritto d'autore e pertanto è fatto espresso divieto di riprodurre, pubblicare o distribuire i materiali tratti dal presente sito, anche in forma parziale, fatta salva la possibilità di realizzare un'unica copia del materiale, in formato cartaceo e/o digitale, all'esclusivo fine di studio.

L'utente è responsabile per l'uso improprio o non autorizzato del materiale pubblicato effettuato in violazione delle suddette disposizioni.

Algoritmo Genetico
Popolazione iniziale

- Il primo passo è creare una popolazione di stringhe di bit casuali. Potremmo usare valori booleani True e False, valori stringa "0" e "1" o valori interi 0 e 1. In questo esercizio useremo valori interi.

- Possiamo generare un array usando la funzione **`randint()`**

- Rappresenteremo una soluzione candidata come lista anziché come array NumPy per semplificare le cose.

- Una popolazione iniziale come stringa casuale di bit può dunque essere creata come segue:

```
...
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
```

Algoritmo Genetico
Popolazione iniziale

- Vediamo un esempio di esecuzione della suddetta istruzione, impostando gli iperparametri **n_pop** a 10 e **n_bits** a 15:

In [15]:
```
n_pop = 10
n_bits = 15
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]

pop
```

Out[15]:
```
[[0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0],
 [0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1],
 [1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0],
 [0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1],
 [1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1],
 [0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0],
 [0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1],
 [0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0]]
```

Algoritmo Genetico
Iterazioni e calcolo punteggio

- Successivamente, possiamo effettuare un certo numero di iterazioni dell'algoritmo, controllate da un iperparametro chiamato **`n_iter`**:

```
...
for gen in range(n_iter):
    ...
```

- Il primo passaggio nell'iterazione dell'algoritmo consiste nel valutare tutte le soluzioni candidate. Utilizzeremo una funzione denominata **`objective()`** come funzione obiettivo generica e la chiameremo per ottenere un punteggio di fitness, che ridurremo al minimo:

```
...
scores = [objective(c) for c in pop]
```

Algoritmo Genetico
Tournament Selection

- Possiamo quindi selezionare i genitori che verranno utilizzati per creare figli. La procedura di selezione di torneo può essere implementata come una funzione che prende la popolazione e restituisce un genitore selezionato. Il valore k è fissato a 3 come valore di default, ma possiamo sperimentare anche valori diversi:

In [ ]:
```
# tournament selection
def selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]
```

Algoritmo Genetico
Selezione genitori

- Possiamo quindi chiamare la funzione **`selection()`** una volta per ogni posizione nella popolazione per creare un elenco di genitori selezionati:

```
...
selected = [selection(pop, scores) for _ in range(n_pop)]
```

Algoritmo Genetico
Crossover

- Ora dobbiamo creare la prossima generazione. Ciò richiede innanzitutto una funzione per eseguire il **crossover**.
- Questa funzione richiederà **due genitori** e il **crossover rate**.
- Il crossover rate è un **iperparametro** che determina se il crossover viene eseguito o meno.
- Se non viene eseguito, i genitori vengono copiati nella generazione successiva.
- È dunque una probabilità che in genere ha un valore vicino a 1.0.

Algoritmo Genetico
Crossover

- La seguente funzione **`crossover()`** implementa il crossover utilizzando un'estrazione di un numero casuale nell'intervallo [0,1] per determinare se eseguire o meno il crossover, quindi selezionando un punto di divisione valido nel caso in cui debba essere eseguito:

In [ ]:
```
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string
        pt = randint(1, len(p1)-2)
        # perform crossover
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]
```

Algoritmo Genetico
Mutazione

- Ora occorre eseguire la mutazione. La funzione che segue inverte semplicemente i bit con una bassa probabilità controllati dall'iperparametro **`r_mut`**:

In [ ]:
```
# mutation operator
def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        # check for a mutation
        if rand() < r_mut:
            # flip the bit
            bitstring[i] = 1 - bitstring[i]
```

Algoritmo Genetico
Creazione nuova generazione

- Possiamo quindi scorrere l'elenco dei genitori e creare un elenco di figli da utilizzare come generazione successiva, chiamando le funzioni di **crossover** e **mutazione**:

```
...
children = list()
for i in range(0, n_pop, 2):
    # get selected parents in pairs
    p1, p2 = selected[i], selected[i+1]
    # crossover and mutation
    for c in crossover(p1, p2, r_cross):
        # mutation
        mutation(c, r_mut)
        # store for next generation
        children.append(c)
```

Algoritmo Genetico

Codice Algoritmo

In [ ]:

```
# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(pop[0])
    # enumerate generations
    for gen in range(n_iter):
        # evaluate all candidates in the population
        scores = [objective(c) for c in pop]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
        # select parents
        selected = [selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]
```

Algoritmo Genetico
Codice (1a parte)

In [ ]:
```
# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(pop[0])
```

Algoritmo Genetico
Codice (2a parte)

```
# enumerate generations
for gen in range(n_iter):
    # evaluate all candidates in the population
    scores = [objective(c) for c in pop]
    # check for new best solution
    for i in range(n_pop):
        if scores[i] < best_eval:
            best, best_eval = pop[i], scores[i]
            print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
    # select parents
    selected = [selection(pop, scores) for _ in range(n_pop)]
```

Algoritmo Genetico
Codice (3a parte)

```
    # create the next generation
    children = list()
    for i in range(0, n_pop, 2):
        # get selected parents in pairs
        p1, p2 = selected[i], selected[i+1]
        # crossover and mutation
        for c in crossover(p1, p2, r_cross):
            # mutation
            mutation(c, r_mut)
            # store for next generation
            children.append(c)
    # replace population
    pop = children
return [best, best_eval]
```

Problema OneMax
Definizione della Funzione Obiettivo

- In questa sezione applicheremo l'algoritmo genetico a un problema di ottimizzazione binaria basato su stringhe. Il problema si chiama **OneMax** e valuta una stringa binaria in base al numero di 1 nella stringa.

- Dato che abbiamo implementato l'algoritmo genetico per minimizzare la funzione obiettivo, possiamo aggiungere un segno negativo a questa valutazione in modo che grandi valori positivi diventino grandi valori negativi.

- Possiamo pertanto definire la funzione obiettivo da minimizzare come segue:

In [ ]:
```
# objective function
def onemax(x):
    return -sum(x)
```

Problema OneMax
Inizializzazione Iperparametri

- Definiamo gli iperparametri come segue:

In [ ]:
```
# define the total iterations
n_iter = 100
# bits
n_bits = 20
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / float(n_bits)
```

Problema OneMax
Esecuzione dell'Algoritmo

- Eseguiamo l'algoritmo, passandogli come funzione obiettivo la funzione **onemax**:

In [ ]:
```
# perform the genetic algorithm search
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Problema OneMax
Esecuzione Algoritmo

- Risultato dell'esecuzione:

```
>0, new best f([1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0]) = -10.000
>0, new best f([1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1]) = -12.000
>0, new best f([0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]) = -13.000
>0, new best f([1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -15.000
>1, new best f([1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1]) = -17.000
>4, new best f([1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]) = -18.000
>5, new best f([1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -19.000
>8, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Done!
f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000
```

Riferimenti

Wirsansky, E. *Hands-On Genetic Algorithms with Python*, Packt, 2020.

Brownlee, J. *Optimization for Machine Learning - Finding Function Optima with Python*, Machine Learning Mastery, 2021.

Gridin, I. *Learning Genetic Algorithms with Python*, BPB Publications, 2021.

Luke, S. *Essentials of Metaheuristics*, Second Edition, 2013.