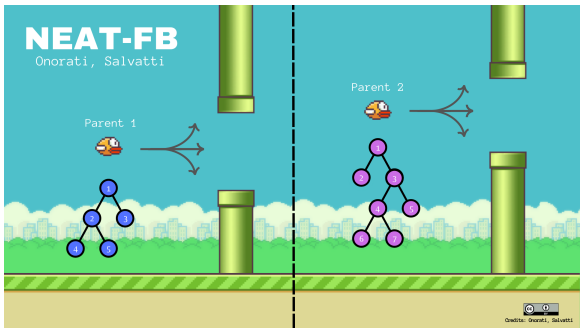
Vincere a Flappy Bird grazie all'IA

di Mattia Salvatti e Claudio Onorati

INTRODUZIONE

Flappy Bird è uno dei giochi arcade più famosi degli ultimi anni, sviluppato da Dong Nguyen nel 2013. Questo gioco, oltre ad essere incredibilmente popolare, è noto anche per la sua grande difficoltà: i giocatori devono controllare un piccolo uccellino mentre vola attraverso una serie di tubi, schivando tutti gli ostacoli lungo il tragitto. Per farlo, devono premere continuamente sullo schermo, facendo sbattere le ali dell'uccellino per mantenerlo in volo.

La grande sfida di Flappy Bird sta nel cercare di ottenere il punteggio più alto possibile, evitando di toccare qualsiasi ostacolo lungo la strada. Ma, come spesso accade nei giochi più difficili, la soluzione a questo problema è stata trovata grazie all'uso dell'intelligenza artificiale.



L'obiettivo del progetto è di impiegare l'algoritmo genetico ***NEAT*** per apprendere a giocare a ***Flappy Bird*** mediante la riproduzione del gioco in Python utilizzando la libreria ***NEAT-Python*** per la realizzazione dell'algoritmo genetico.

Avviare il programma

> **NB** Per avviare il programma è necessario installare le seguenti librerie:
>
> - NEAT-Python https://neat-python.readthedocs.io/en/latest/installation.html
> - PyGame http://www.pygame.org/downloads.shtml
> - Numpy https://numpy.org/install/
> - Matplotlib https://matplotlib.org/stable/users/installing/index.html

- Graphviz https://pypi.org/project/graphviz/

NEAT

NeuroEvolution of Augmenting Topologies (**NEAT**) è un *algoritmo genetico* (GA) di *metaeuristic search* che genera *reti neurali evolutive artificiali*, per la risoluzione di problemi di *ottimizzazione stocastica*. La sua unicità è data dalla capaità di creare una sua struttura sempre più complessa e nel contempo sempre più ottimale (analoga all'evoluzione naturale) attraverso un modello risolutivo a black-box. L'alogritmo è stato originariamente sviluppato Kenneth Stanley e Risto Miikkulainen nel 2002 all'Università del Texas ad Austin.

A partire da reti estremamente semplici, in quanto completamente prive di neuroni intermedi (*Hidden States*), NEAT ha generalmente performance più elevate nella ricerca di soluzioni efficaci e robuste rispetto ad algoritmi di *reinforcement leaening* o a tecniche neuro-evolutive analoghe che però partono da topologie predeterminate o comunque casuali. Ciò è possibile grazie all'uso di strutture iniziali minime che sono facili da ottimizzare e rendono l'algoritmo estremamente veloce nella ricerca di soluzioni.

Ogni *algorimo genetico* (*GA*) presenta i seguenti step:

1. **Inizializzazione**: Viene creata la prima *generazione* composta da un numero predefinito di elementi, solitamente la popolazione iniziale viene generata in maniera casuale
2. **Selezione**: Alla terminazione di ogni generazione, vengono selezionati alcuni campioni attraverso una *funzione di fitness* che ne valuta la loro performance.
3. **Operatori Genetici**: L'operazione successiva è la riproduzione della popolazine selezionata (*Parents*) formando nuovi individui (*Childs*) attraverso gli *operatori genetici* di *crossover* e di *mutazione*. La nuova generazione presenterà quindi parte delle caratteristiche dei genitori pur avendo delle caratteristiche innovative, rendendo la nuova generazione differente dalla precedente (e teoricamente più efficiente).
4. **Euristica**: Possono essere applicate operazioni *euristiche* aggiuntive per velocizzare i calcoli. Ad esempio l'euristica *speciation* penalizza il *crossover* tra soluzioni candidate che sono troppo simili tra loro.
5. **Terminazione**: Il processo generazionale viene ripetuto fino al raggiungimento di una condizione di terminazione, una delle possibili ccondizioni di terminazione è il superamento del punteggio di *fitness threshold* preventivamente stabilito.

L'algorimo si basa su tre princìpi fondamentali:

1. **OMOLOGIA**: NEAT codifica ciascun nodo e ciascuna connessione della rete attraverso un gene. Ogni volta che una mutazione strutturale sfocia nella creazione di un nuovo gene, quel gene riceve un contrassegno numerico che lo rende permanentemente rintracciabile. Tale marcatura storica è utilizzata in seguito per verificare la conciliabilità di geni omologhi durante l'operazione di crossover, e per definire un operatore di compatibilità;

2. **PROTEZIONE DELL'INNOVAZIONE**: L'operatore precedente di compatibilità divide la popolazione composta da reti neurali in specie differenti, allo scopo di proteggere le soluzioni innovative da un'eliminazione prematura, e di prevenire l'incrocio di materiale genetico incompatibile. Tali innovazioni strutturali presentano una significativa possibilità di raggiungere il loro pieno potenziale, in quanto protette dal resto della popolazione attraverso la suddivisione in specie, ovvero la creazione di nicchie o spazi riservati;

3. **STRUTTURA MINIMA**: da ultimo, il principio secondo cui la ricerca di una soluzione dovrebbe avvenire nel più piccolo spazio possibile (inteso come numero di dimensioni), da espandere poi in maniera graduale. Cominciando il processo evolutivo da una popolazione di elementi a struttura minima, le successive mutazioni topologiche comportano l'aggiunta di nuovi nodi e connessioni alle reti, conducendo pertanto ad una crescita incrementale della popolazione stessa. Dal momento che solo le modifiche strutturali vantaggiose tendono a sopravvivere nel lungo termine, le topologie che vengono raffinate tendono ad essere le minime necessarie alla soluzione del problema assegnato.

L'utilizzo dell'algoritmo porta i seguenti vantaggi rispetto ad analoghi GA:

1. **Velocità**: Strutture più piccole si ottimizzano più velocemente. Inoltre, anche se nel tempo aumenta la complessità, la maggior parte delle strutture preesistenti sono già ottimizzate.
2. **Fuga da un'optima locale**: Non solo NEAT attraverso l'*accoppiamento* e la *mutazione* può cercare la forma del suo ambiente, ma può anche alterare l'ambiente stesso con nuove strutture. Così, quando una specie in NEAT si trova su un optimum locale, è possibile che aggiungendo una nuova connessione, si possa aprire una nuova dimensione di libertà, che porta a un percorso di allontanamento dall'optimum locale.

Limiti di NEAT:

1. **Fitness functions ripetute**: Il calcolo della *fitness function* può essere molto complesso e dispendioso in termini di risorse, il fatto che questa funzione venga ripetuta per ogni generazione porta a rendere l'algoritmo spesso inefficiente (LENTEZZA)
2. **Non scalabilità in complessità**: Quando il numero di elementi esposti alla mutazione è elevato, si verifica spesso un aumento esponenziale delle dimensioni dello spazio di ricerca. Per rendere tali problemi trattabili da *GA* , devono essere scomposti nella rappresentazione più semplice possibile.
3. **La soluzione ottimale NON è SEMPRE la migliore in assoluto**: un problema generale di ogni algorimo di *Reinforcement Learning* è che la ricerca della soluzione deve essere limitata in termini

di risorse (tempo e memoria), in termini di iterazioni (con forte dipendeza della qualità dalle iterazioni precedenti) oppure (in particolare per i *GA*) in termini di *punteggio di fitness* per poter dare una risposta in tempi accettabili per il problema. Ciò comporta che la soluzione ottimale trovata potrebbe non essere la migliore in assoluto se l'algoritmo avesse a disposizione maggiori iterazioni.

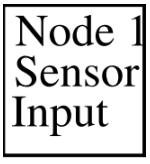
4. **Fitness function difficile da implementare** in problemi complessi non è sempre facile implementare una funzione di fitness che risolva il problema in questione (NB: la *fitness funcrion* è specifica per ogni problema affrontato)
5. **Selezione casuale della popolazione iniziale**: Dato che la prima generazione è selezionata casualmente, è SPESSO necessario molto più tempo rispetto ad altri algoritmi, inoltre non è prevedibile il tempo (o il numero di generazioni) necessario per raggiungere la "situazione ottimale"

NEAT non è un algoritmo di RL in ML
L'alogritmo di Reinforcement Learning (RL) fa parte della categoria del Machine Learning (ML), oltre al *Supervised Learning* e all'*Unsupervised Learning*. NEAT in quanto GA fa parte della categoria di *Metaheuristic Stochastic Search*, in quanto framework ad alto livello indipendente dal problema.
Le principali differenze sono:

1. RL utilizza una struttura relativamente ben compresa e matematicamente fondata dei processi decisionali di Markov, mentre NEAT si basa in gran parte sull'euristica e su mutazioni casuali.
2. RL cerca di massimizzare la somma di ricompense dell'agente aggiornando la sua *action value function* appresa durante l' *exploration* e *exploitation* con l'ambiente attraverso l'uso dei gradienti, mentre NEAT presenta una fitness function statica e l'operazione di massimizzazione viene effettuata tra agenti di una generazione.
3. NEAT è un *Inter-life algorithm* in quanto gli agenti devono morire per progredire mentre RL + un *Intra-life algorithm* in quanto apprende attivamente dall'ambiente (*Continual Learning*) e in alcuni casi presenta funzioni aggiuntive per preservarsi (*Safe RL*).

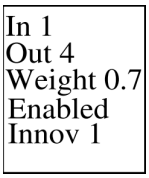
DIZIONARIO NEAT:

- **Gene Nodo**: Rappresenta il nodo vero e proprio, ognuno di essi trasporta l'informazione di *Numero di Nodo*, *Tipologia di Nodo* intesa come Input/Hidden/Output e altre informazioni.

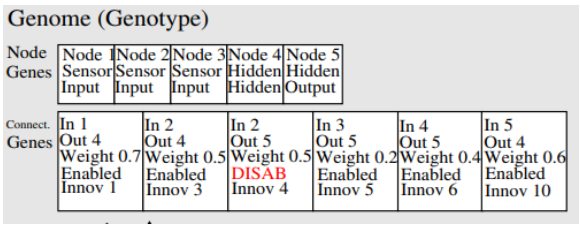


- **Gene di Connessione**: Collegamento tra due *Geni Nodo*, ognuno di essi trasposta l'informazione di *Nodo d'Ingresso*, *Nodo d'Uscita*, *Peso della Connessione*, un *Enable Bit* per l'espressione o non

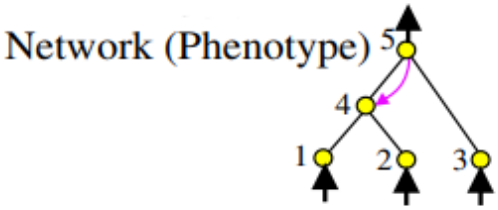
del gene, e un *Numero di Innovazione*



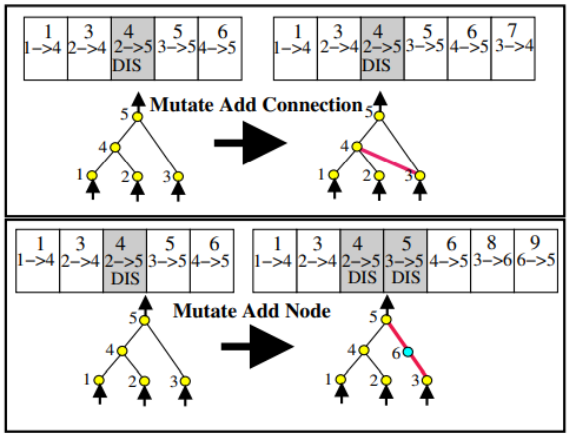
- **Genotipo**: Singolo insieme di *Geni Nodo* e *Geni di Connessione*.



- **Fenotipo**: Singola rappresentazione della topologia di una singola rete di *Geni Nodo* e *Geni di Connessione*



- **Generazione**: singolo insieme di un *Genotipo* e di un *Fenotipo* che rappresenta la singola iterazione
- **Popolazione**: parte o totalità di una *Generazione*
- **Funzione di fitness**: è la funzione che a seconda di parametri di input forniti, produce un output che valuta la "bontà" degli elementi rispetto al problema in questione
- **Funzione di attivazione**: funzione che trasfotma la media pesata dei valori di input in valori di output
- **Crossover**: incrocio di *Genotipi*
- **Mutazione strutturale**: evento che può cambiare sia il *Peso della connessione* che la *Topologia della Rete*, avviene in 2 modalità, *aggiungendo una connessione* tra due nodi non connessi precedentemente oppure *aggiungendo un nodo* in una connessione preesistente dividendola in due.



- **fitness threshold**: valore di ricomensa della *popolazione* oltre il quale termina l'algoritmo (raggiungimento situazione ottimale)

COME VIENE USATO NEAT IN FLAPPY BIRD

Le reti neurali

Per realizzare un'IA in grado di giocare a Flappy Bird, sono stati utilizzati modelli di apprendimento automatico noti come "reti neurali". Questi strumenti si ispirano alla struttura e al funzionamento del cervello umano, e sono composti da diversi livelli, ognuno dei quali ha uno scopo specifico.

Nel caso di Flappy Bird, la rete neurale ha due livelli principali: il primo è chiamato "Input Layer", ed è costituito dalle informazioni che l'IA ha a disposizione.

Il secondo livello, invece, è il "Output Layer", che ha il compito di decidere quale azione intraprendere in base alle informazioni raccolte nel primo livello.

Ma la vera sfida di Flappy Bird sta nella capacità dell'IA di migliorarsi continuamente, grazie a una serie di "generazioni" in cui viene testata e valutata la sua capacità di sconfiggere il gioco. In questo modo, l'IA impara ad adattarsi a situazioni sempre più difficili, fino a raggiungere un livello di abilità tale da non poter essere più battuta.

Le informazioni che la rete neurale ha a disposzione nell'input layer sono la posizione dell'uccelino e la distanza tra quest'ultimo e i tubi, mentre l'output layer avrà il compito di decidere se far saltare o meno l'uccelino.

Input layer

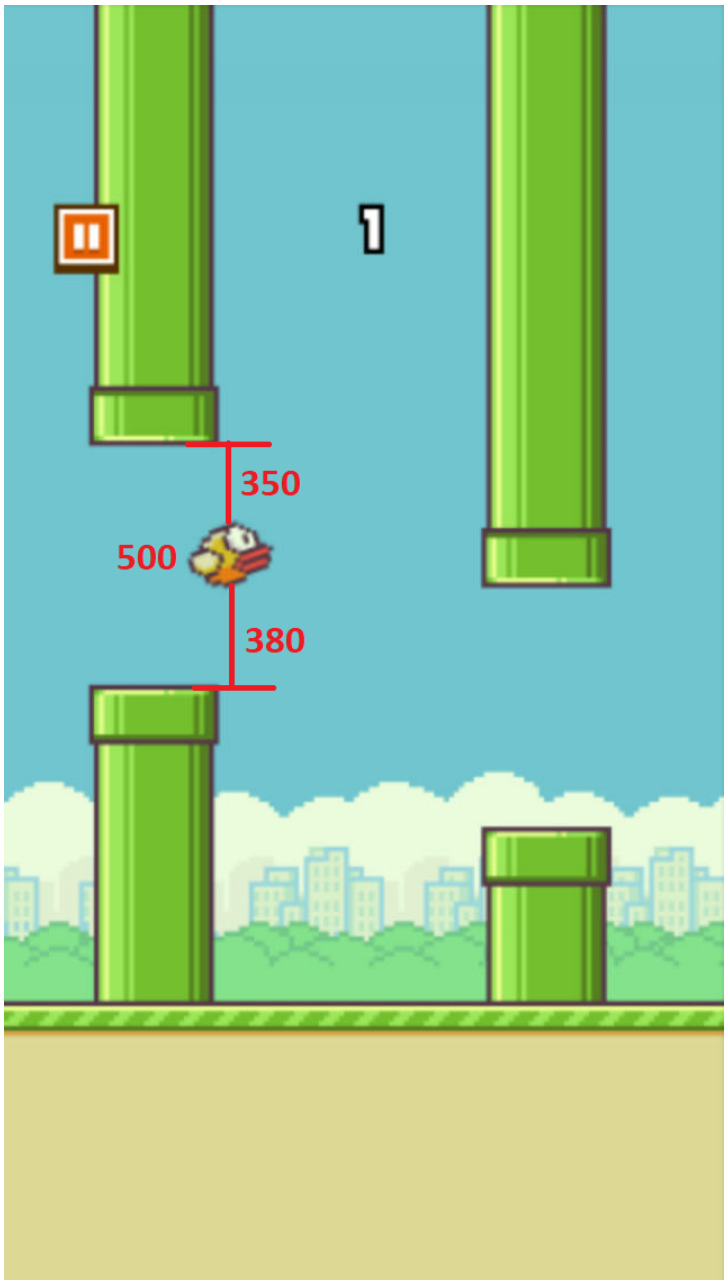
Output layer

Posizione Uccellino

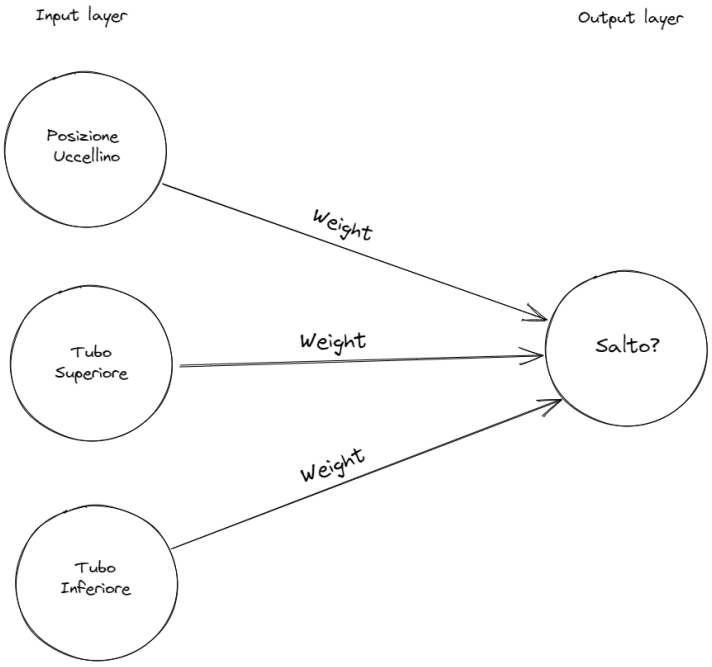
Tubo Superiore

Tubo Inferiore

Salto?



I nodi dell'input layer sono collegati al nodo dell'output layer tramite una connessione, ognuna di queste connessioni ha un peso chiamato, ***Weight***. Il valore del wheight è diverso per ogni connessione è il suo scopo è quello di migliorare, o in alcuni casi, peggiorare la rete neurale.



Come funziona la rete neurale in Flappy Bird?

Il processo per far funzionare una rete neurale inizia con il passaggio di valori ai nodi di input. Ciascuna connessione riceverà poi un peso e questi valori saranno elaborati attraverso funzioni che consentiranno ai nodi di output di eseguire azioni specifiche.

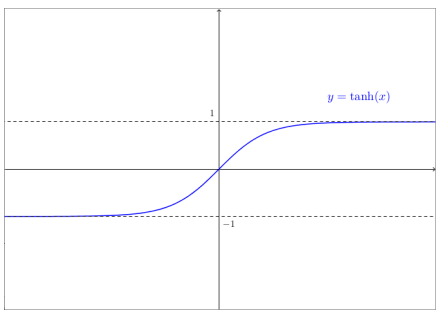
La prima azione è la somma ponderata, che viene poi passata al nodo di output. Successivamente, verrà sommato il BIAS, un parametro che consente di controllare la flessibilità e l'adattabilità delle reti neurali. In questo modo, se i pesi non funzionano correttamente, si può spostare la funzione di attivazione della rete nella direzione desiderata.

Il risultato ottenuto viene quindi passato alla funzione di attivazione, che permette di ottenere un valore compreso tra due livelli prestabiliti. Tra le varie funzioni disponibili, la Tanh(x), ossia la tangente iperbolica, è quella più adatta per il gioco Flappy Bird. Infatti, permette di ottenere un valore compreso tra -1 e 1. Se il valore in uscita dalla somma ponderata è un numero positivo grande, tenderà a 1; se è un numero negativo grande, tenderà a -1. In ogni altro caso, il valore sarà compreso tra -1 e 1.

$$Somma\ ponderata = \sum_{i=0}^{k} (Input_i * Weight_i) + BIAS$$

$$F(x) = TanH(x)$$

$$Output = F(Somma\ Ponderata)$$



Grazie a questo valore il nodo di output sarà in grado di eseguire una determinata azione, per il caso di Flappy Bird se il valore sarà maggiore di 0.5 l'uccellino salterà.

Come vengono scelti Weights e Bias?

Il Weight e il bias per il caso di Flappy Bird vengono impostati in modo automatico dalla macchina, per fare ciò è stato utilizzato l'algoritmo ***NEAT***(NeuroEvolution of Augmenting Topologies) è un algoritmo di intelligenza artificiale che si basa sulla selezione naturale.

Per il primo step non sappiamo quale può essere il valore giusto per i Weights e per il Bias e perciò è necessario eseguire dei test in modo completamente casuale, creando una popolazione di uccellini ampia.

Ogni popolazione è composta da X uccellini(agenti intelligenti), dove ogn'uno di questo ha una propria rete neurale che lo controlla, ogni rete neurale verrà testata ed allenata e successivamente verrà valutato il loro Fitness.

Il fitness si riferisce alla capacità di un algoritmo o di un modello di adattarsi e migliorare le proprie prestazioni attraverso l'allenamento e l'ottimizzazione, viene spesso utilizzata come metrica di valutazione per algoritmi di apprendimento automatico, come le reti neurali, per selezionare i modelli più performanti e migliorare le loro capacità predittive.

In Flappy Bird, il fitness viene valutato in base a quanto l'uccellino avanza nel gioco, in particolare ogni fotogramma che avanza nel gioco o tubo che supera ottiene dei punti.

Alla fine della simulazione si prendono gli uccellini con il punteggio di fitness più elevato e si crea una nuova popolazione partendo da questi. La nuova popolazione sarà migliore della precedente e continuiamo con questo processo fino a quando non saremo soddisfatti.

FLAPPY BIRD IA CODE

Creazione della grafica

Di seguito vengono riportate le classi principali utilizzate per implementare la grafica di Flappy Bird, tra cui la classe *Bird* per la creazione degli uccellini, la classe *Pipe* per l'implementazione casuale dei tubi in basso e in alto e la classe *Base* per la costruzione del terreno di gioco

Creazione dell'uccellino

```
class Bird:
  MAX_ROTATION = 25           #Valore massimo di rotazione (per non farlo mai andare a testa
  ROT_VEL = 20                #Velocità di rotazione dell'immagine dell'uccellino (quando sal
  ANIMATION_TIME = 5          #Ogni quanto cambia il frame e l'uccellino muove le ali

  def __init__(self, x, y):    #Inizializzazione dell'uccellino

  def jump(self):              #Descrizione dei parametri di salto

  def move(self):              #Descrizione del movimento parabolico dell'uccellino, inclusa l

  def draw(self, win):         #Descrizione del battito di ali dell'uccellino

   def get_mask(self):         #Associazione dell'immagine dell'uccellino alla maschera di coll
```

Creazione dei tubi

```
class Pipe:
  GAP = 200                   #Distanza costante della fessura tra due tubi
  VEL = 5                     #Velocità costante di movimento del tubo

  def __init__(self, x):      #Inizializzazione del tubo

  def set_height(self):       #Gestisce l'altezza dei tubi

  def move(self):             #Descrive il movimento orizzontale regolare del tubo

  def draw(self, win):        #Disegno del tubo

  def collide(self, bird):    #Descrive il modo in cui l'uccellino può collidere con il tubo
```

Creazione del terreno

```
class Base:
  VEL = 2.5                  #Velocità di movimento del terreno (Associata per essere coerente

  def __init__(self,y):      #Inizializzazione del terreno

  def move(self):            #Descrizione del movimento del terreno (coerente con il movimento

  def draw(self, win):       #Associazione dell'immagine all'elemento terreno
```

Creazione della finestra complessiva di gioco

```
def draw_window(win, birds, pipes, base, score):  #Creazione degli elementi quali uccelli, tub
```

Implementazione di NEAT

Il primo step è impostare i parametri nel file di configurazione di neat. Per modificare i parametri dell'algoritmo è necessario, impostarli nel file di configurazione, denominato *config.txt*. Di seguito i parametri più importanti:

```
fitness_criterion     = max                #Decidiamo quali uccellini tenere in base al  valo
fitness_threshold     = 100                #Soglia che deve raggiungere il fitness prima che
pop_size              = 50                 #La dimensione dell popolazione per ogni generazio
activation_default    = tanh               #Funzione di attivazione, nel nostro caso la tange
bias_max_value        = 30.0               #Valore massimo che il  bias può assumere alla pri
bias_min_value        = -30.0              #Valore minimo che il  bias può assumere alla prim
```

Successivamente il file di configurazione viene caricato nel programma, viene creata la popolzione utilizzando i parametri del file di configurazione. Una volta creata, viene lanciata la funzione di fitness chiamata **main**, per una massimo di 50 volte, che sarà il numero massimo di generazioni prima che il programma venga terminato.

```
def run(config_path):

    #Viene passato il path del file di configurazione e venogno settate le diverse proprietà
    config = neat.config.Config(neat.DefaultGenome, neat.DefaultReproduction, neat.DefaultSpeci

    #Creazione della popolazione
    p = neat.Population(config)

    #Opzionale: serve per visualizzare nella console le statistche
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
```

```
    #configurazione della fitness function, 50 è il numero di volte che chiamerà la funzione ma
    w = p.run(main,50)


if __name__ == "__main__":

    #Caricamento del file di configurazione

    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, "config.txt")
    run(config_path)
```

Funzione fitness

La prima parte della funzione fitness si occupa della creazione della rete neurale di ogni uccellino. Nel codice abbiamo tre array, per le reti neurali, per i genomi e per gli uccellini. Tramite gli indici di ogni array sono in grado di sapere sempre quale rete neurale e genoma è associato ad ogni uccellino. Succesivamene viene creata la finestra di gioco contenente le varie componenti del gioco.

```
def main(genomes, config):

    nets = []                                                   #Array delle re
    ge = []                                                     #Array contenen
    birds = []                                                  #Array degli uc

    #Creazione della rete neurale per ogni genomaD
    for _, g in genomes:
        g.fitness = 0                                           #Valore del fit
        net = neat.nn.FeedForwardNetwork.create(g, config)      #Creazione dell
        birds.append(Bird(230, 350))                            #Creazione dell
```

Proseguendo la funzone di fitness, avvia un ciclo while che si interrompe solamente se viene chiusa la finestra di gioco oppure se non ci sono piu uccellini in vita. ##INSERIRE LA CONDIZIONE DA GUARDARE

All'interno di questo ciclo troviamo un ciclo for che si occupa di incrementare di 0.1 il fitness di ogni genoma, per ogni frame che l'uccellino avanza. Successivamente viene attivata la rete neurale di ogni uccellino passandogli la posizione di quest'ultimo e le distanze tra l'uccellino e il tubo superiore ed inferiore, viene eseguita la funzione di attivazione, tanh, che ritornerà un valore compreso tra -1 e 1, se il valore è maggiore di 0.5, l'uccellino salta.

```
while run:
  clock.tick(30)                                                          #Gestisce il f
              for event in pygame.event.get():                            #Si occupa di
                  if event.type == pygame.QUIT:

              pipe_ind = 0
              if len(birds) > 0:
                  if len(pipes) > 1 and birds[0].x > pipes[0].x + pipes[0].PIPE_TOP.get_width(
                      pipe_ind = 1

              else:                                                        #Se non ci so
                  run = False
                  break

              #Per ogni frame che l'uccellino avanza viene aumentato il fitness di 0.1
              for x, bird in enumerate(birds):
                  bird.move()
                  ge[x].fitness += 0.1

                  #Attivazione della rete neurale
                  output = nets[x].activate((bird.y, abs(bird.y - pipes[pipe_ind].height), abs

                  if output[0] > 0.5:
                      bird.jump()
```

Le righe successive si occupano di gestire la logica del gioco, in particolare le prime righe gestiscono la movimentazione dei tubi nella schermata di gioco e la collissione degli agenti intelligenti con quest'ultimi. Ogni volta che l'agente intelligente oltrepassa un tubo il suo punteggio di fitness viene incrementato di 3, successivamente il tubo passato viene rimosso dalla schermata e viene aggiunto un altro tubo. Le ultime righe controllano le collissioni con il suolo e il limite superiore della schermata di gioco

```
              add_pipe = False
              rem = []

              for pipe in pipes:
```

```
            pipe.move()                                                 #Muove i tubi

            for x,bird in enumerate(birds):                             #Gestisce le
                                                                        #uccellini ch
                if pipe.collide(bird):
                    ge[x].fitness -= 1                                  #il fitness d
                    birds.pop(x)
                    nets.pop(x)
                    ge.pop(x)

                if not pipe.passed and pipe.x < bird.x:                 #Controlla ch
                    pipe.passed = True
                    add_pipe = True

            if pipe.x + pipe.PIPE_TOP.get_width() < 0:                  #Si occupa di
                rem.append(pipe)

        if add_pipe:                                                    #Se supera i
            score +=1
            for g in ge:
                g.fitness += 3
            pipes.append(Pipe(600))                                     #Aggiunge un

        for r in rem:
            pipes.remove(r)

        for x, bird in enumerate(birds):
            if bird.y + bird.img.get_height() - 10 >= 730 or bird.y < -50:   #Controlla le
                birds.pop(x)                                            #parte superi
                nets.pop(x)
                ge.pop(x)

        base.move()                                                      #Muove il te
        draw_window(win, birds, pipes, base, score)                      #Crea i vari
```

Avvia il codice in Gitpod

Open in Gitpod