Institut für Angewandte und Numerische Mathematik
Sommersemester 2021
Dr. Daniel Weiß, M.Sc. Kevin Ganster

Einführung in Python

Abschlussprojekt

Lernen eines Spiels

Ziel dieses Projektes ist die Implementierung des Trainingsalgorithmus „NeuroEvolution of Augmented Topologies“ (NEAT) [2]. Es wird das Spielen eines Jump'n'Runs (Gadakeco) gelernt. Das Lernproblem wird hierbei mithilfe eines *evolutionären Algorithmus* gelöst. Eine grobe Einführung liefert *SethBling* mit dem Video [1], das außerdem die Motivation für dieses Projekt war.

Mithilfe von evolutionären Algorithmen lassen sich allgemeine Optimierungsprobleme approximativ lösen. Der Ablauf solcher Verfahren erinnert dabei an den Selektionsprozess in der Natur, bei dem sich eine Gruppe von verschiedenen Individuen über mehrere Generationen weiterentwickelt. Entscheidend ist dabei, dass gut an die Umgebung angepasste Individuen eine höhere Überlebens- und Fortpflanzungswahrscheinlichkeit besitzen als weniger gut angepasste Individuen. Der Grad der Anpassung wird dabei mit dem Wert der sogenannten *Fitness-Funktion* gemessen. Durch den Selektionsprozess bilden sich im Laufe der Generationen Individuen heraus, deren Fitness-Werte immer besser werden – sie konvergieren gegen ein (lokales) Optimum der Fitness-Funktion.

In dieser Aufgabe misst die Fitness-Funktion den Erfolg im Spiel Gadakeco und die Individuen sind durch neuronale Netze modellierte Spieler.

1 Ablauf allgemeiner evolutionärer Algorithmen

1. Es wird eine *Population* mit n initialen *Individuen* erzeugt.
2. Jedes Individuum wird aufgrund einer *Fitness-Funktion* bewertet.
3. Die Individuen, die eine hohe Fitness aufweisen, überleben und kommen in die nächste *Generation*. *(Selektion)*
4. Zusätzlich entstehen neue Individuen, welche durch (zufällige) leichte Veränderung der Überlebenden erzeugt werden. *(Mutation)*
5. Außerdem gibt es *Nachkommen*, die durch Kombination von Attributen der Überlebenden entstehen. *(Rekombination)*
6. Die Population soll nach den Schritten 3-5 wieder eine Größe von etwa n besitzen.
7. Wiederhole die Schritte 2-6 bis ein Individuum mit akzeptabler Fitness gefunden wurde.

Für die Abgabe ist der 5. Schritt nicht erforderlich, kann aber als *Bonus* implementiert werden.

2 Das Spiel Gadakeco

Um Gadakeco zu starten werden die Module `pygame` und `pillow` benötigt, die z. B. mit `pip install MODULNAME` oder bei einer vorliegenden Anaconda-Installation mit `conda install MODULNAME` installiert werden können.

Das Ziel des Spiels besteht darin, die Spielfigur vom Startpunkt aus möglichst weit nach rechts zu manövrieren. Die Spielwelt ist dabei nach rechts hin unbeschränkt groß und wird abhängig vom gewählten *Seed* (natürliche Zahl) erzeugt. Dabei erzeugt derselbe Seed immer dieselbe Spielwelt.

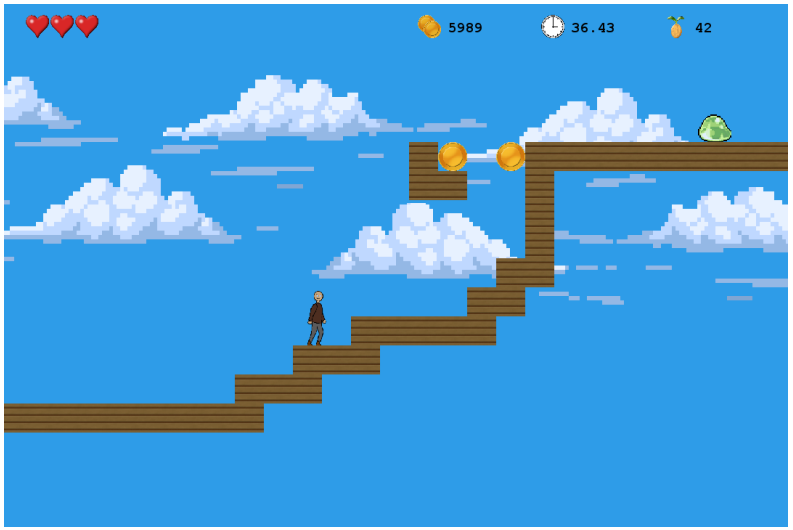

Abbildung 1: Der Screenshot zeigt eine typische Spielsituation in Gadakeco. Vor dem Hintergrund bewegt sich die Spielfigur auf rechteckigen Blöcken von links nach rechts. Oben rechts ist ein Gegner (Schleim) zu sehen. Am rechten oberen Bildschirmrand werden die aktuell erreichte Punktzahl, die bereits verstrichene Zeit und der Seed für die aktive Spielwelt angezeigt. Die Punktzahl bemisst sich daran, wie weit der Spieler nach rechts vorgedrungen ist und wie viele Münzen er eingesammelt hat. Oben links ist die aktuelle Lebensenergie des Spielers mit drei Herzen symbolisiert, die durch Kontakt mit einem Gegner reduziert wird. Das Spiel ist verloren, wenn die Spielfigur nach unten aus dem Bild fällt oder ihre Lebensenergie auf null sinkt.

a) Starten Sie das Programm `main.py` im Ordner `src` und spielen Sie eine, oder auch mehrere Runden, indem Sie auf *Start Game* und im neuen Fenster ebenfalls auf *Start Game* klicken. Mit den folgenden Tasten können Sie steuern: **A** bzw. **D**, um nach Links bzw. Rechts zu laufen und **Leertaste** zum Springen.

Für das Lernen des Spiels werden wir spezielle neuronale Netze verwenden, welche wir nun einführen.

3 Aufbau künstlicher neuronaler Netze (KNN)

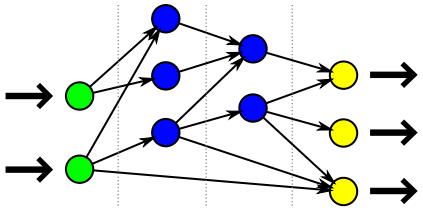

Abbildung 2: Vereinfachte Darstellung eines künstlichen neuronalen Netzes mit zwei versteckten Schichten (hidden layers). Hier wird die Information von links (Eingabeneuronen) nach rechts (Ausgabeneuronen) entlang der Kanten propagiert. Die Trennung der Schichten ist hierbei mit gestrichelten vertikalen Linien visualisiert.

Ein KNN ist ein kantengewichteter, gerichteter und zykelfreier Graph, wobei die Knoten *Neuronen* genannt werden. Jedes Neuron h speichert eine Zahl out_h, die wir seine Ausgabe nennen. Es gibt unterschiedliche Neuronentypen:

- Eingabeneuronen (in Abb. 2 grün), welche zum Einspeisen der Daten in das Netz dienen.
- Versteckte Neuronen (in Abb. 2 blau), die eingehende „Signale“ verarbeiten und an die nächsten Schichten weiterleiten.
- Ausgabeneuronen (in Abb. 2 gelb), welche nach der Berechnung das Ergebnis des Netzes beinhalten.

Die Kanten verlaufen hierbei immer vorwärtsgerichtet (feedforward), das heißt „von links nach rechts“ wie etwa in Abb. 2 zu sehen: Eingabeneuronen können nur Anfangsknoten einer Kante, während Ausgabeneuronen nur Endknoten einer Kante sein können. Die Neuronen bekommen auf diese Weise eine „Schicht“ zugewiesen:

- Die *Eingabeschicht*, bestehend aus allen Eingabeneuronen.
- Die *Ausgabeschicht*, welche alle Ausgabeneuronen umfasst.
- Die *versteckten Schichten*, die jeweils aus allen versteckten Neuronen bestehen, welche die gleiche maximale Pfadlänge zur Eingabeschicht besitzen.

In dieser Aufgabe soll ein künstlicher Spieler implementiert werden, der ähnliche Ein- und Ausgabemöglichkeiten besitzt wie ein menschlicher Spieler. Dazu wird ihm eine vereinfachte Ansicht der aktuellen Spielsituation, vgl. Abb. 3 zur Verfügung gestellt und er kann zu jedem Zeitpunkt festlegen, welche Aktionen (**Links**, **Rechts**, **Springen**) er momentan ausführt.

Intern soll der künstliche Spieler seine Aktionen mithilfe eines neuronalen Netzes auswählen, das für jedes der 486 Felder der vereinfachten Ansicht der aktuellen Spielsituation ein Eingabeneuron und für jede der Aktionen **Links**, **Rechts**, **Springen** ein Ausgabeneuron besitzt.

Um den künstlichen Spieler in Gadakeco einzubinden, dient der Menüpunkt „Neuronal Networking“, mit dem Sie sich folgendermaßen vertraut machen können:

a) Klicken Sie im Hauptmenü auf *Neuronal Networking* und anschließend auf *Create New Network*, um das Training eines neuen Spielers zu starten.

Da der künstliche Spieler und das Training noch nicht implementiert sind, bewegt sich die Spielfigur nicht. Verbessert der künstliche Spieler für 3.5 Sekunden (in beschleunigter Spielzeit) den Fitness-Wert, vgl. Abb. 3, nicht, wird seine Bewertung beendet und das Spiel mit einem neuen Spieler erneut gestartet.

Das Ziel dieser Aufgabe besteht darin, diesen Modus des Spiels zu vervollständigen. Dafür wird die Auswertung der verwendeten neuronalen Netze beschrieben.

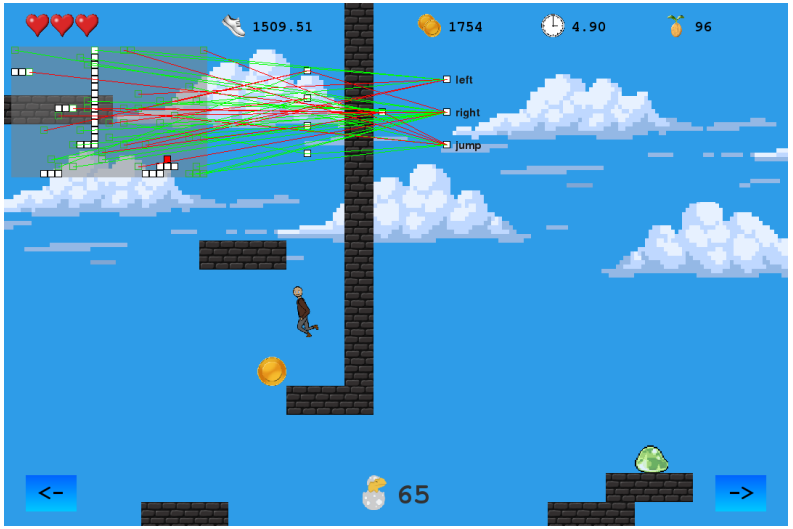

Abbildung 3: Bildschirmausgabe der zweiten Ansicht im Modus „Neuronal Networking“. Oben links sehen Sie die vereinfachte Ansicht der aktuellen Spielsituation mit $27 \times 18 = 486$ Feldern. Hierbei bedeutet ein weißes Feld einen begehbaren Block, während rote Felder Feinde darstellen. Daneben wird eine Darstellung des aktuellen neuronalen Netzes angezeigt. Hierüber wird der aktuelle Wert der Fitness-Funktion dargestellt, im Beispiel 1509.51. Unten in der Mitte wird angezeigt, dass aktuell die 65. Generation der Population im evolutionären Algorithmus spielt. Mit den Pfeil-Buttons (oder der **TAB**-Taste) können sie zwischen den verschiedenen Zeichenmodi wechseln.

4 Berechnung der Netzwerkausgabe

Jedes Eingabeneuron ist für einen festen Bildschirmausschnitt verantwortlich und das Netz hat drei Ausgabeneuronen für die möglichen Aktionen des Spielers, vgl. Abb. 4. Ist der Wert out_a eines Ausgabeneurons a für eine gegebene Spielsituation positiv, bedeutet dies, dass der künstliche Spieler in dieser Situation die entsprechende Aktion ausführt.

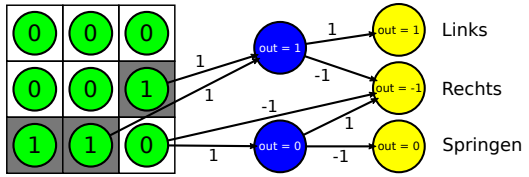

Abbildung 4: Schematisches Beispiel für die Auswertung eines KNN für Gadakeco. Das 3x3-Raster auf der linken Seite symbolisiert die vereinfachte Spielansicht, in der begehbare Blöcke dunkel eingezeichnet sind. Jedes Kästchen dieses Rasters ist mit einem Eingabeneuron verbunden, wobei die Neuronen zu begehbaren Blöcken den Wert 1 erhalten. Die Zahlen über den Kanten des KNN geben das Gewicht w der jeweiligen Kante an. So werden die Werte der Eingabeneuronen durch das Netz zu den drei Ausgabeneuronen propagiert. In diesem Beispiel führt der künstliche Spieler nur die Aktion **Links** aus.

Die Ausgabe out_i jedes Eingabeneurons i wird mit dem Wert des zugehörigen Bildschirmausschnitts in der vereinfachten Darstellung initialisiert. Im nächsten Schritt werden die versteckten Schichten und anschließend die Ausgabeschicht ausgewertet, wobei hier die Vorwärtsrichtung beachtet wird. (Das heißt zuerst wird die Schicht mit Distanz 1 zur Eingabeschicht bearbeitet.)

Jedes versteckte bzw. Ausgabeneuron aus der aktuellen Schicht berechnet die gewichtete Summe aller Zustände der Vorgänger gewichtet mit dem Kantengewicht. Anschließend wird auf diese gewichtete Summe die *Aktivierungsfunktion* angewendet. Wir verwenden hierfür die Signumsfunktion, das heißt:

$$\sigma(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}.$$

Insgesamt berechnet also ein Neuron h seine Ausgabe out_h wie folgt:

$$\text{out}_h = \sigma\left(\sum_{(v,h,w)\in K(h)} w \cdot \text{out}_v\right),$$

hierbei enthält $K(h)$ die gewichteten Kanten (v, h) des KNN, welche in h enden, als Dreier-Tupel. Dabei gibt der dritte Wert w das Gewicht der Kante (v, h) an.

Am Ende kann die Ausgabe des Netzes an den Ausgabeneuronen abgelesen werden. Ist der Zustand positiv, so gilt die zugehörige Taste als gedrückt.

5 Mutation von Netzwerken

Es gibt sehr viele Arten von Mutationen für KNN, wir beschränken uns auf die folgenden zwei:

- *Kanten-Hinzufügen-Mutation* Hierbei wird eine Kante in Vorwärtsrichtung mit einem zufälligen Gewicht, das heißt 1 oder -1, eingefügt. Beachten Sie, dass Eingabeneuronen nur ausgehende und Ausgabeneuronen nur eingehende Kanten besitzen. Versteckte Neuronen können sowohl Eingangs- wie auch Ausgangsknoten sein, solange der Graph zykelfrei bleibt. Die Auswahl des Startknotens wird durch eine Wahrscheinlichkeitsverteilung bestimmt: Die Eingabeneuronen, welche für die direkte Umgebung der Spielfigur zuständig sind, sollen eine höhere Wahrscheinlichkeit besitzen, ausgewählt zu werden als die Eingabeneuronen, die für den Rand des Spielfeldes zuständig sind. Die Spielfigur befindet sich an den Positionen $(10, 9)$ und $(10, 10)$. Die versteckten Neuronen erhalten eine mittlere Wahrscheinlichkeit als Startknoten ausgewählt zu werden.
- *Knoten-Hinzufügen-Mutation* Hierbei wird eine vorhandene Kante (a, b, w) zwischen den Neuronen a und b mit Gewicht w durch zwei Kanten $(a, c, 1), (c, b, w)$ und einem neuen versteckten Neuron c ersetzt. Insgesamt wird also die Ausgabe des Netzes durch diese Mutation nicht verändert.

Hinweis: Durch diese Operationen können neue Schichten des Netzes entstehen und Knoten können die Schicht wechseln. Sie müssen also gegebenenfalls Ihre Netzstruktur entsprechend anpassen/aktualisieren.

Die Bearbeitung der folgenden Aufgaben können Sie ausschließlich im Ordner `src/neat` bewerkstelligen.

6 Implementierung des Netzwerks

Gadakeco erwartet, dass eine Klasse `Network` existiert, die einen Spieler als neuronales Netz implementiert. Diese Klasse wird von Gadakeco folgendermaßen verwendet:

- Die Methode `evaluate` soll für eine vereinfachte Ansicht der aktuellen Spielsituation die gedrückten Tasten als Aktion des Spielers ermitteln.
- Das Attribut `fitness` speichert den Wert der Fitness-Funktion, den der Spieler mit seinem aktuellen Verhalten erreicht. Dieser Wert wird über die Methode `update_fitness`, die Gadakeco während des Trainings aufruft, berechnet und aktualisiert.

Bearbeiten Sie die folgenden Aufgaben in der Datei `network.py`, die schon ein Grundgerüst für das neuronale Netz enthält.

a) Implementieren Sie Klassen für Ein- und Ausgabeneuronen, sowie für versteckte Neuronen und erstellen Sie mit deren Hilfe in der Klasse `Network` das initiale Netzwerk. Achten Sie darauf, dass die oben beschriebenen Mutationen in Ihrer Implementierung möglich sind.

b) Implementieren Sie die Methode `evaluate` der Klasse `Network`, welche obige Neuronen benutzt, um die Ausgaben des Netzes zu berechnen. Beachten Sie hierfür auch die Kommentare im Quellcode in `neat/network.py`.

c) Erstellen Sie jeweils eine Methode für die zwei genannten Netzwerkmutationsarten. Beachten Sie die Beschreibung in Abschnitt 5 (inklusive Hinweis!). Sie können zunächst eine Gleichverteilung verwenden, um den Code zu testen. Anschließend sollte jedoch eine passende Verteilung, wie z.B. eine um den Spieler zentrierte Gaußverteilung, gewählt werden.
Hinweis: Achten Sie darauf, dass das Netz nach dem Hinzufügen einer neuen Kante zykelfrei bleibt.

7 Implementierung der Netzwerkdarstellung

Zur Überprüfung und Visualisierung des Zustandes der Netzwerke soll das derzeitig aktive Netz gezeichnet werden. Während des Trainings einer Population kann die Taste **TAB**, oder alternativ die Pfeil-Buttons, gedrückt werden, um verschiedene Zeichenmodi zu durchlaufen. Im zweiten Zeichenmodus wird die vereinfachte Spielfeldansicht, welche als Eingabe für das Netz dient, angezeigt.

a) Machen Sie sich mit dem Modul `neat/networkrenderer.py` vertraut. Die Funktion `render_network` zeichnet bereits die vereinfachte Spielfeldansicht.

b) Erweitern Sie die Funktion `render_network`, sodass das übergebene Netzwerk auf das übergebene Surface gezeichnet wird. Für eine mögliche Darstellung siehe Abb. 3. Beachten Sie, dass wiederholte Aufrufe dieser Funktion zum gleichen Inhalt des Surface führen sollen. Ein Blick in die Dokumentation von PyGame [3] bietet sich an.

8 Implementierung der Population

Eine Population ist eine Menge von `size` Individuen, die jeweils durch ein neuronales Netz modelliert werden. Gadakeco erwartet, dass eine Klasse `Population` existiert, die im Modul `neat/population.py` implementiert ist. Sie besitzt ein Attribut `current_generation`, welches eine Liste mit Individuen der aktuellen Generation enthält. Diese Klasse wird von Gadakeco folgendermaßen verwendet:

- Bestimmt bei jedem Update die Fitness und die gedrückten Tasten für jedes Element aus `current_generation` mittels `update_fitness` und `evaluate`.
- Ruft `create_next_generation` auf, um die Schritte 3, 4 und 6 des evolutionären Algorithmus auszuführen.
- Benutzt das Attribut `name`, um die Population im Menü anzuzeigen.
- `load_from_file` bzw. `save_to_file` um die Population zum Spielen/Weitertrainieren zu laden bzw. nach dem Trainieren zu speichern.

Bearbeiten Sie die folgenden Aufgaben in der Datei `neat/population.py`, die schon ein Grundgerüst für die Klasse `Population` enthält.

a) Nach dem Konstruktoraufruf soll im Attribut `current_generation` eine Liste von `size` Netzwerken angelegt werden. Bei diesen soll nach dem Erstellen unabhängig voneinander einmalig die *Kanten-Hinzufügen-Mutation* durchgeführt werden.

b) Setzen Sie im Konstruktor außerdem das Attribut `name` auf einen eindeutigen Namen der Population, wobei z. B. ein Zeitstempel ausreicht.

c) Implementieren Sie die Methode `save_to_file`. Eine einfache Möglichkeit bietet das Modul `pickle`.

d) Implementieren Sie die statische Methode `load_from_file`, welche es ermöglicht die Population mit dem übergebenen Dateinamen `filepath` zu laden.

8.1 Parametrisierung des evolutionären Algorithmus

Es gibt viele Parameter, die ein schnelles Training ermöglichen. Folgende Parameterwerte liefern eine gute Geschwindigkeit:

- Die besten 10% aller Individuen werden unverändert in die nächste Generation übernommen (vgl. Abschnitt 1 Punkt 3.).
- Die restlichen 90% der Population werden erzeugt, indem auf Kopien der 10% Überlebenden zu jeweils 80% die *Kanten-Hinzufügen-Mutation* und zu 20% die *Knoten-Hinzufügen-Mutation* angewandt wird (vgl. Abschnitt 1 Punkt 4.).

a) Implementieren Sie die Methode `create_next_generation`, die automatisch von Gadakeco aufgerufen wird, sobald alle Netzwerke aus `current_generation` bewertet wurden.

Erstellen Sie die nächste Generation im Attribut `current_generation` nach den Regeln in „Ablauf allgemeiner evolutionärer Algorithmen“ (siehe Seite 1). Verwenden Sie dazu die oben genannten Parameter.

Hinweis: Da die Mutationen das Netzwerk verändern, sollten Sie vorher eine Kopie anlegen. Die Funktion `deepcopy` *des Moduls* `copy` *ist hierfür hilfreich.*

Wenn Sie nun eine Population nur trainieren möchten, können Sie herfür das Modul `main_simulation.py` benutzen. Hier erhalten Sie keine graphische Ausgabe. Sie können dafür die Referenz `pop_name` auf den Namen einer bereits bestehender Populationsdatei setzen. Gibt es diese Datei nicht, wird automatisch eine neue Population erstellt. Zur Darstellung der Trainingsergebnisse können Sie die trainierte Population in Gadakeco laden.

Bonusaufgaben

a) Experimentieren Sie mit anderen Fitness-Funktionen. Die Fitness-Funktion ist in der Methode `update_fitness` der Klasse `Network` in `neat/network.py` implementiert. Man könnte zum Beispiel größere Netzwerke *bestrafen*, sodass einfache Netze bevorzugt werden.

b) Führen Sie eine Wahrscheinlichkeitsverteilung ein, die es ermöglicht, dass jedes Netz in die nächste Generation gelangen kann. Hierbei werden Netze mit hoher Fitness bevorzugt.

c) Speichern Sie das bisher beste Netz in der Population. Sollte sich mehrere Generationen ein Netz nicht wesentlich verbessert haben, ersetzen Sie es durch das bisher Beste. (Vermeidung von Stagnation und unnötiger Netzwerkstruktur)

d) *Beachten Sie, dass die folgenden Aufgaben evtl. sehr umfrangreich werden können.*

   (i) Greifen Sie die Idee der Originalarbeit [2] auf und implementieren Sie *Spezies*.

   (ii) Implementieren Sie den Schritt 5) des evolutionären Algorithmus. Schauen Sie auch hier für Anregungen in die Originalarbeit [2].

Literatur

[1] SethBling. MarI/O - Machine Learning for Video Games. https://www.youtube.com/watch?v=qv6UVOQ0F44.

[2] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[3] Pete Shinners *et al.* Pygame. http://pygame.org/, 2011.

Material: ILIAS