Miniprojekt – Aufgabe
Genetische Algorithmen und neuronale Netze

Einführende Bemerkungen

Ich will die einführenden Sätze des iypthon-Notebooks wiederholen:

Wir befassen uns mit einer Situation, in der es Wesen in einer Umwelt gibt. Die Wesen haben eine Gestalt und ein Verhalten, das durch eine Art Genom codiert wird. Ein gegebenes Wesen bewährt sich in der Umwelt in gewissem Maß, das numerisch bewertet wird (unter Umständen auch in der Form 1=lebendig/0=tot).

In der natürlichen Evolution ist die besagte Codierung durch die DNA in Zellkern und Mitochondrien unbeschreiblich komplex. Man ist weit davon entfernt, diese beschreiben zu können, auch wenn man den Schritt DNA→Protein halbwegs verstanden hat.

- Für 'algorithmische Evolution' ist es entscheidend, eine Codierung für die Eigenschaften (Verhalten/Gestalt) zu finden, die flexibel genug ist, interessante Anpassungsphänomene zu erlauben und doch wieder nicht so flexibel, dass die meisten Codes zu völlig unsinnigen Wesen führen.

- Von der Idee der 'algorithmischen Evolution' inspiriert entstand die Idee der 'evolutionären Algorithmen'. In diesem Fall ist anstelle eines Wesens ein bestimmtes Optimierungsproblem gegeben, d.h. die Parameter, die ein Verhalten codieren, sind ebenso gegeben, wie die 'Umwelt', die dieses Verhalten bewertet.

In formaler Hinsicht ist kein großer Unterschied zwischen 'algorithmischer Evolution' und 'evolutionären Algorithmen', aber das Ziel und die Bedeutung unterscheiden sich dennoch stark.

Wer Evolution im Computer erproben will, hat kein bestimmtes Optimierungsproblem im Blick, sondern will Anpassungsphänomene durch Mutation und Selektion an eine Umwelt beobachten. Eine geeignete Wahl der Codierung ist dabei entscheidend & sehr schwierig.

Ein evolutionärer Algorithmus wird dagegen zur Lösung eines bestimmten, gegebenen Optimierungsproblems verwendet. Aber statt herkömmlicher numerischer Optimiertungsmethoden wird auf 'Mutation und Selektion' zurückgegriffen. Das ist, wenn man mehr über das Optimierungsproblem weiß, kaum jemals die beste Methode, sie ist aber in ihrer Einfachheit bestechend und in gewissem Sinn universell.

Ein Beispielproblem: Cartpole

Das Beispiel im iypthon-Notebook verwendet das Paket openAI-gym, um eine Wagen, auf dem ein drehbar gelagerter Balken steht ('cartpole'), zu simulieren. Der Zustand des Wagens ist gegeben durch vier Zahlen (Ort, Geschwindigkeit, Winkel des Balkens, Winkelgeschwindigkeit des Balkens). Zu jedem Zeitpunkt gibt es nur zwei mögliche Handlungen, eine kleine Kraft nach rechts oder nach links auf den Wagen wirken zu lassen. Das Experiment gilt als gescheitert, sobald der Balken mehr als 12 Grad geneigt ist oder der Wagen zu weit rechts oder links ist. Solange es noch nicht gescheitert ist, gibt es eine Belohnung.

Das Optimierungsproblem besteht nun darin, eine Handlungsfunktion ('Policy') $a : \mathbb{R}^4 \to \{\text{links}, \text{rechts}\}$ zu lernen, die in jedem Zustand eine Handlung empfiehlt, so dass der Balken balanciert wird und das Fahrzeug im Bildausschnitt bleibt, d.h. insgesamt möglichst viele 'Belohnungen' zu sammeln.

Im Notebook wurde die Handlungsfunktion *probabilistisch* modelliert, d.h. $f : \mathbb{R}^4 \to \mathbb{R}$ gibt die Wahrscheinlichkeit für die erste der beiden möglichen Aktionen (links oder rechts drücken) aus.

Nun ist die optimale Handlungsfunktion ja vorher unbekannt, wir müssen also eine allgemeine Beschreibung vieler möglicher Funktionen voraussetzen und darunter die beste finden. Dazu gibt man eine *parametrisierte Familie von Funktionen* an, d.h. für jeden Wert der Parameter $p \in \mathbb{R}^N$ gibt es eine andere Funktion $f_p : \mathbb{R}^4 \to \mathbb{R}$. Diese Familie lässt sich also als Funktion

$$\begin{aligned} \mathbb{R}^N \to &\quad \text{Funktionen}(\mathbb{R}^4, \mathbb{R}) \, . \\ p \mapsto &\quad f_p \end{aligned}$$

auffassen, oder aber als Funktion

$$\begin{aligned} \mathbb{R}^N \times \mathbb{R}^4 \to &\mathbb{R} \\ (p, x) \mapsto &f_p(x) \, . \end{aligned}$$

Es ist ein Parameter $p \in R^N$ zu finden, so dass die Belohnung für den 'Akteur' maximal wird.

Wenn die Belohnung für den Akteur sich in Abhängigkeit von $p \in \mathbb{R}^N$ beispielsweise als k mal differenzierbare Funktion schreiben ließe, könnten effiziente Methoden der Analysis zur Anwendung kommen. Wenn jedoch die Belohnung sich nur aus der Simulation einer Umwelt ergibt, so dass sie sich zwar für jedes p ermitteln, aber nicht explizit darstellen lässt, lässt sich mit genetischen Algorithmen etwas ausrichten. Der Parameter p ist sozusagen das *Genom*, das zufällig mutiert wird. Die Varianten, die sich bewähren (Selektion), werden erneut mutiert.

Genetische Algorithmen als 'stochastische Optimierung'

Um das Problem etwas besser zu verstehen, setzen wir noch voraus, dass die Komponenten p_i alle in einem bestimmten abgeschlossenen Intervall $I = [a,b] \subset \mathbb{R}$ liegen müssen, also $p \in I^N \subset \mathbb{R}^N$. I^N ist also ein N-dimensionaler Würfel.

Wir stellen nun eine Überlegung an, um zu verstehen, warum die Dimension N des Parameterraums entscheidend für die Möglichkeit 'evolutionären Lernens' ist. Wenn wir annehmen, dass wir uns bei den optimalen Parametern p_i jeweils einen Fehler $\varepsilon = (b-a)/K$ leisten können, so müssen wir also für jeden der Parameter p_i K verschiedene Teilintervalle von $[a,b]$ untersuchen, bzw. jeweils einen Punkt in einem solchen Teilintervall. Da wir dass für alle N Parameter unabhängig machen müssen, haben wir insgesamt den Würfel in K^N Teilwürfelchen zerlegt. Im schlimmstmöglichen Fall müssten wir alle K^N Würfel ein Mal besuchen. Wenn nun der evolutionäre Algorithmus die Mutationen nach Art einer 'Irrfahrt' erzeugt, wird er früher oder später jedes Würfelchen erreichen. Nur wächst leider K^N sehr stark mit N, was bedeutet, dass mit wachsendem N eben nicht der ganze Suchraum in einer sinnvollen Zeit durchlaufen werden kann – und dass ein evolutionärer Algorithmus im schlimmstmöglichen Fall scheitern muss. Man nennt das auch im Zusammenhang mit maschinellem Lernen immer wieder auftretende Phänomen den *curse of dimensionality*.

Aber glücklicherweise haben wir es oftmals nicht mit dem schlimmstmöglichen Fall zu tun. Zur Erleichterung der Vorstellung denken wir uns den Fall $N = 2$. Die Funktion, die für jeden Parameterwert die Belohnung angibt, lässt sich als Fläche darstellen. Wir suchen den höchsten Berg in dieser Fläche. Was mit dem schlimmstmöglichen Fall gemeint war, wäre eine unbeschreiblich zerklüftete Landschaft mit lauter schmalen aber hohen Felsnadeln. Bessere Fälle sehen so aus, dass es nur wenige ausgeprägte Berge gibt, und dass man sich diesen schon von weitem nähern kann, indem man bergauf geht.

Die Anfangspopulation sollte groß genug sein, dass sich mit großer Wahrscheinlichkeit einige der Wesen im Einzugsbereich dieses Optimums befinden. Indem nun in jedem Durchgang eine Teilpopulation ausgewählt und diese mutiert wird, sollte dafür gesorgt werden, dass in jeder weiteren Generation immer mehr Wesen sich dem Optimum immer mehr nähern.

Die Aufgaben

Es gibt nun sehr viel Freiheit bei der Konzeption eines solchen Algorithmus:

- bei der Wahl der Funktionenfamilie
- bei der Bestimmung der Mutationen
- bei der Selektion

1. Im ipython-Notebook ist die Funktionenfamilie durch ein so genanntes *künstliches neuronales Netz* mit einem *hidden layer* mit h Knoten gegeben. Ein Approximationssatz garantiert, dass ein neuronales Netz mit einer *hidden layer* jede stetige Funktion auf einer kompakten Menge beliebig genau approximieren kann, wenn nur die Zahl der verborgenen Knoten groß genug ist. Es ist nichts Geheimnisvolles an diesen Funktionen. Heißen die Eingabevariablen $x_1, \ldots, x_4$, so berechnen die zehn verborgenen Knoten $z_i,\ j = 1, \ldots, 10$ den folgenden Wert:

$$z_i = \mathrm{ReLU}(b_i + \sum_{j=1}^{4} w_{ij} x_j);.$$

Der Ausgangsknoten y berechnet den Wert:

$$y(x) = \sigma(b + \sum_{i=1}^{10} w_i z_i)\ .$$

Insgesamt also stellt das neuronale Netz für jeden festen Wert der Parameter w_{ij} ($4h$ Stück), b_i (h Stück), w_i (h Stück) und b (1 Parameter) die Funktion

$$\begin{aligned} \mathbb{R}^4 \to &\quad \mathbb{R} \\ x = (x_1, \ldots, x_4) \mapsto &\quad \sigma\left(b + \sum_{i=1}^{10} w_i\ \mathrm{ReLU}(b_i + \sum_{j=1}^{4} w_{ij} x_j)\right) \end{aligned}$$

dar.

a) Ersetzen Sie die Funktion `make_model` durch eine Funktion `make_model_2`, die ein Netz ohne verborgene Knoten erzeugt. Damit stellt das Netz nur die einfache Funktion

$$x \mapsto \sigma(b + \sum_{i=1}^{4} w_i x_i)$$

dar und hat nur fünf Parameter. Dieses Modell ist auch als 'logistische Regression' bekannt. Es besteht nur aus der Verknüpfung von σ mit einer linearen Funktion. Probieren Sie aus, ob dieses Modell ausreicht, um den Cartpole zu steuern. Dokumentieren Sie außerdem, wie schnell dieses Modell im Vergleich zum anderen lernt. (Sie können die Rechenzeit messen und außerdem die Zahl der Generationen.)

b) Lassen Sie beide Modelle mit verschieden starker Mutation lernen. Das wird in dem Beispielprogramm durch den Parameter σ der Funktion `evolve` gesteuert, der die Standardaweichung der Gauß'schen Störung angibt, die auf die Parameterwerte addiert wird. Probieren Sie verschiedene Werte von σ aus und dokumentieren Sie, wie schnell das Modell jeweils lernt. Probieren Sie ebenfalls aus, was passiert, wenn der Mutationsparameter von Generation zu Generation kleiner wird. (Das entspricht der Idee, am Anfang den Parameterraum möglichst weiträumig zu durchforschen, später aber die Mutationen so klein zu halten, dass der Algorithmus sich von bereits 'guten' Modellen zu in der Nähe liegenden verbessert, statt ganz woanders hin zu springen.)

c) Ändern Sie ebenfalls den Parameter k, mit dem `evolve` die Methode `keep_best` aufruft. Er bestimmt, wie viele 'Wesen' einer Generation auf die nächste übergehen und mutiert werden. Dokumentieren Sie wieder, wie ihr Modell lernt.

d) Lassen Sie für eines der Modelle und feste Parameterwerte das Modell 20 Mal lernen, den Pfahl zu stabilisieren. Merken Sie sich (in einer geeigneten Datenstruktur) für jeden Durchlauf den höchsten, niedrigsten und mittleren Reward in der der i. Generation für $i = 0\ldots,99$. Plotten Sie diese Verläufe in einem Graphen. Plotten Sie außerdem den über die zwanzig Durchläufe gemittelten höchsten Reward.

2. Die zweite Aufgabe können Sie nur lösen, wenn Sie auf Ihrem Betriebssystem auch noch das Python-Paket box2d installieren können. Falls ja, bietet Ihnen openAI-gym dann auch das Environment 'BipedalWalker-v2', das ein zweibeiniges Wesen simuliert (s. auch `https://gym.openai.com/envs/BipedalWalker-v2/`.)

   Es hat einen 24-dimensionalen Zustandsraum (der die Stellungen verschiedener Gelenke und Körperteile, bzw. auch deren Ableitung, beschreibt) und als Handlungsraum $[-1,1]^4$, d.h. vier reelle Zahlen beschreiben die möglichen Handlungen, die Drehmomente an Knien und Hüften beschreiben. (Das ist natürlich ein stark vereinfachtes Modelle eines wirklichen Fußgängers, sieht auch etwas seltsam aus.)

   Versuchen Sie sich daran, den zweibeinigen Gang 'evolutionär' zu lernen. Wählen Sie ein geeignetes Modell, Mutationsparameter etc. und dokumentieren Sie Ihre Ergebnisse. Ich weiß nicht, wie schwer dieses Problem ist, da die mir bekannten und funktionierenden Lösungen gerade nicht auf evolutionären Algorithmen, sondern auf so genanntem 'Reinforcement Learning' beruht. Good Luck!

   Nachtrag: Da ich niemanden sozusagen ins offene Messer laufen lassen möchte, habe ich ausprobiert, ob das klappen kann – es kann.