Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Pong, gespielt von KNNs

Praktikum Paper

Dokumentation der erstellten Applikation

Daniel Speck und Florian Kock
Matr.Nr. 6321317, 6346646
2speck@informatik.uni-hamburg.de, 2kock@informatik.uni-hamburg.de

2. Februar 2015

Abstrakt

Unsere Aufgabe des letzten Semesters war es, in dem Praktikum Neuronale Netze eine Aufgabe zu lösen. Wir haben uns für das in den 70ern populäre Spiel Pong entschieden. Dieses war dahingehend interessant, da es gleich mehrere Probleme zu lösen gab:

- die zeitverzögerte Bewertung
- das Erkennen der Flugrichtung durch Rekurrenz

Das Spielprinzip von Pong ist simpel und ähnelt dem des Tischtennis: Ein Punkt („Ball“) bewegt sich auf dem Bildschirm hin und her. Jeder der beiden Spieler steuert einen senkrechten Strich („Schläger“), den er mit einem Drehknopf (Paddle) nach oben und unten verschieben kann. Lässt man den „Ball“ am „Schläger“ vorbei, erhält der Gegner einen Punkt. (Wikipedia)

Inhaltsverzeichnis

1 Vorbedingungen und Anforderungen

Um die Applikation erfolgreich ausführen zu können sind folgende Anforderungen an die Laufzeitumgebung notwendig:

- Python3.4

- NumPy
- The Python Standard Library
  - multiprocessing
  - threading
  - sys
  - random
  - os.path
  - logging
  - json
  - socketserver
  - tkinter
  - socket
  - time
  - datetime
  - copy

2 Schnelleinstieg

Der Schnelleinstieg führt über folgende Befehle: (Wenn die Vorbedingungen gegeben sind!)

Starten des Hauptprogramms:

```
> ls
__init__.py         knnframe.py      telegramframe.py
__pycache__         main.py                    visu.py
concol.py           recneunet.py
court.py            save
> python3.4 main.py
... starting application ...
```

Starten der Visualisierung zum Hauptprogramm:

```
> ls
__init__.py        knnframe.py        telegramframe.py
__pycache__        main.py                  visu.py
concol.py          recneunet.py
court.py           save
> python3.4 visu.py
... starting visualisation ...
```

3 Aufbau der Applikation

Unsere Applikation ist in mehrere Dateien und Module aufgeteilt. In der Abbildung 1 auf Seite 4 ist die Grundstruktur dargestellt. Die einzelnen Module sind separiert in die Haupt-Anwendung (main.py) 3.1 und die Visualisierung (visu.py) 3.2.

3.1 Die Haupanwendung - main.py

Die Hauptapplikation besteht im Wesentlichen aus einer Hauptschleife, die die Spielzüge (im Quellcode 'Ticks' genannt) kontrolliert. Der relevante Teil befindet sich hierzu in der *main.py* in den Zeilen von ca. 360 bis 460.

Sie liefert in jedem Spielzug vom Spielfeld (siehe 3.1.1) die Ballpositionen an die Spieler (siehe 3.1.2), um anschließend deren Aktion wieder dem Spielfeld zuzuführen.

3.1.1 Das Spielfeld - curt.py

Das Spielfeld implementiert eine Simulation der Spielumgebung. Diese besteht aus zwei Schlägern und einem Ball in einem rechteckigen Rahmen. Bei jedem Aufruf von *tick()* berechnet das Spielfeld den nächsten Zustand des Balles, hierbei wird ebenfalls geprüft, ob er die Bande, den Rahmen, überschritten hat und korrigiert dies. Ein solcher Abprall ist der Physik nachempfunden ('Einfallswinkel = Ausfallswinkel').

Das Ziel des Spieles ist es, den Ball mit dem Schläger zurückzuspielen, gelangt der Ball jedoch über die Linie, bedeutet dies, das der Ball wieder in der Mitte neu initialisiert wird. Hierfür ist die Funktion *__initvectors()* zuständig. Sie setzt den Ball mit einer zufälligen Höhe auf die Mittellinie (Ortsvektor), um ihm dann mit einem zufälligen Einheitsvektor, welcher jeweils zwischen -45 bis + 45 Grad zu jedem Spieler zeigen kann, eine Richtung zu geben.

Die Schläger könnnen, über die die Funktion *move(player, action)* verschoben werden. Der Parameter *action* kann entweder ein *float* mit genauen Positionsdaten (-1 bis +1) oder ein *string* mit Aktionen:

- *'u':str* für einen Schritt aufwärts

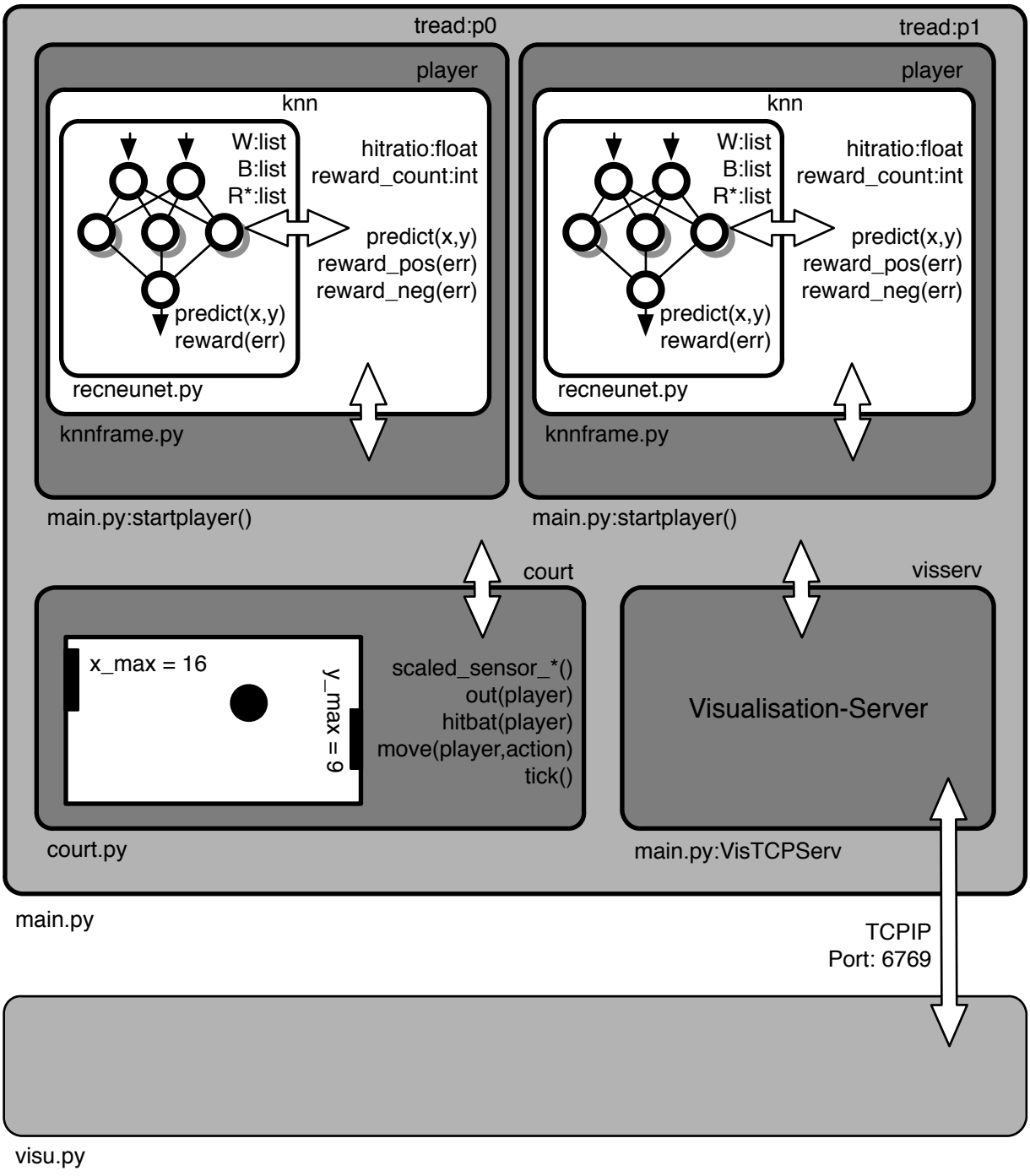

Abbildung 1: Schematischer Aufbau der Pong-Applikation

- *'d':str* für einen Schritt abwärts
- *'n':str* für stehenbleiben

In der Applikation nutzen wir jedoch nur die einfache, direkte Positionierung über eine *float*-Zahl.

Für die Visualisierung stellt das Spielfeld Inspektionsfunktionen zur verfügung die immer mit *'v_'* beginnen. Sie werden ausschließlich zu diesem Zweck genutzt und finden keine weitere Anwendung in der Logik der Applikation.

3.1.2 Das Framework um das KNN - knnframe.py

Das Rekurente Netz in der Datei *recneunet.py* (siehe 3.1.3) ist möglichst allgemein gehalten. Es sollte nicht spezifisch für dieses Problem geschrieben sein. Es könnte, so war unsere Intention, leicht portiert werden um weitere Probleme lösen zu können, deren Lösungswege ebenfalls die Rekurrenz bedingen.

Das Framework adaptiert und konfiguriert das Netz an die Probleme des Spieles Pong. Es sorgt dafür, das jeweils *positive* und *negative* Belohnungen (siehe 3.1.4) dem Netz zugeführt werden und es entsprechend daraus lernen kann.

Eine weitere Aufgabe ist es, Daten über den aktuellen Zustand zu sammeln um damit eine Aussage zu haben, wie gut das Netz trainiert ist und welche Qualität die Vorhersagen haben. Ursprünglich war es ebenfalls abgedacht, diese für einen dynamischen Lernenfortschritt zu nutzen wovon wir jedoch während des Verlaufes aus Zeitgründen wieder abgekommen sind.

3.1.3 rekurrentes künstliches neuronales Netzwerk - recneunet.py

Das rekurrente künstliche neuronale Netzwerk ist von der Grundstruktur her aufgebaut wie ein MLP welches wir zu Anfang des Kurses kennengelernt haben. Wir haben jedoch ein paar Modifikationen eingeführt um dem Netz es möglich zu machen, das gestellte Problem zu lösen.:

Unser Netz kann Daten aus der Vergangenheit speichern und somit für zukünftige Aussagen nutzen. Dies ist wichtig, da eines der Haupt-Probleme darin bestand, dass über einen langen Zeitraum keine Aussage über die Qualität der Vorhersagen getroffen werden kann. Im Spiel Pong bewegt sich der Ball eine lange Zeit lang über das Spielfeld und es ist nicht ohne physikalisches Wissen möglich einen Aufprallpunkt zu berechnen. So kann keine Aussage darüber getroffen werden, ob die aktuelle Schlägerposition gut oder schlecht ist bzw. wie sie am besten korrigiert werden sollte. (siehe Abbildung 2 auf Seite 6) Durch das Aufnehmen der Daten in einem *Ringpuffer* entsteht eine gewisse *Sicht* auf vergangene Aktionen und Vorhersagen wenn eine, die Letzte, der Aussagen bewertet werden kann. Hierdurch werden dann Rückschlüsse auf die letzten Vorhersagen generiert und daraus kann gelernt werden. Die Anzahl der Lernschritte in die Vergangenheit ist in der Variable *t_max* abgebildet. Sie kann beim Erstellen des KNN-Objektes durch *knnframe.py* (siehe 3.1.2) neben der Struktur (Layer und Anzahl der Neuronen je Layer) fest-

gelegt werden. Die Daten werden in *Listen* und *Numpy-Arrays* gespeichert welche immer mit einem *R* im Namen beginnen:

- RW: Rekurrente Gewichte
- RH: Rekurrente Aktivierungen
- RS: Rekurrenter Output

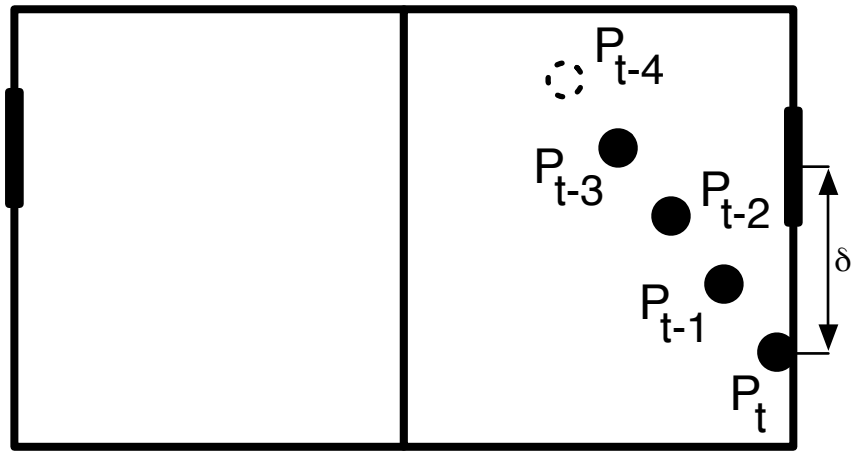

Abbildung 2: Lernen aus Daten der Vergangenheit

Um keine Daten doppelt zu Speichern greift auch die nächste Modifikation auf diese Daten zurück: Das Netz nutzt für zukünftige Vorhersagen Daten aus vergangenen Aussagen. Hierzu wird für jedes *Hidden-Layer* auf den internen Datenspeicher zugegriffen, welcher vom lernen aus der Vergangenheit aus dem vorherigen Absatz angelegt wurde. Auf diese Daten, greift das Netz bei jeder neuen Vorhersage zu und kann somit, in unserem Fall, eine Ballrichtung erkennen.

3.1.4 Belohnungen

Positive und *negative* Belohnungen sind eigentlich aus dem SARSA-Algorithmus bekannt. Wir haben Anfangs gedacht eben diesen für unser Problem einzusetzen. Da jedoch durch die vielen Zustände ein sehr großer *Statespace* (alle Positionen multipliziert mit allen Winkel) erhalten würden, haben wir schnell diese Idee verwerfen müssen. Geblieben sind jedoch die Belohnungen. Diese werden im Framework um das KNN (siehe 3.1.2) dazu genutzt, das Netz anzuweisen aus seinen gespeicherten Daten entsprechend zu lernen. Der Algorithmus kann als Abgewandeltes *überwachtes Lernen* bezeichnet werden. Dies ist im neuronalen Netzwerk (siehe 3.1.3) beschrieben.

3.2 Die Visualisierung - visu.py

Die Visualisierung der dient der einfachen Diagnose, des Zustands der Hauptapplikation welche keine eigene Visualisierung bereitstellt. Die Verbindung wird via TCP/IP über den *Port 6769* hergestellt und ist aktuell nur vom *localhost* erreichbar.

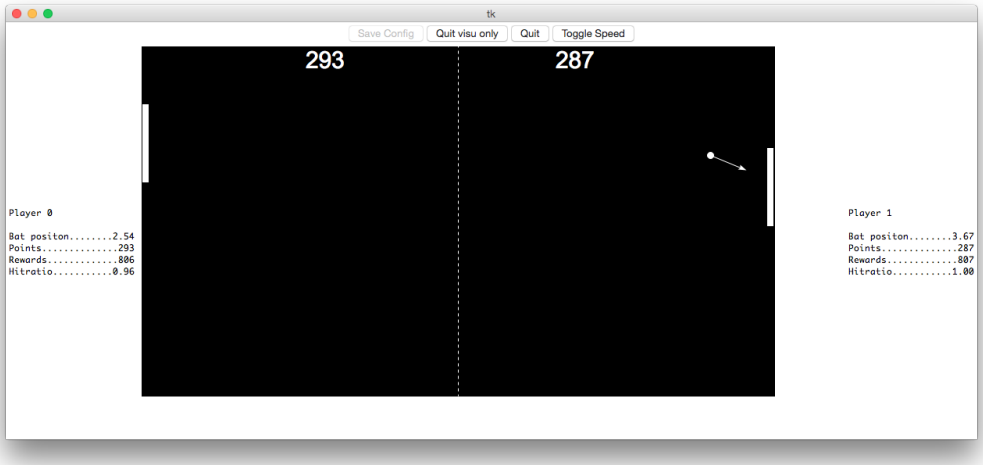

Abbildung 3: Bildschirmfoto der Visualisierung

Probleme beim Beenden entstehen je nach Konstellation durch noch geöffnete Ports, die einem Prozess der durch den Benutzer terminiert wurde noch reserviert sind. Hier hilft es, einen Moment zu warten oder die Visualisierung zu starten. Durch den versuch die Verbindung zwischen Client und Server herzustellen, merkt das Betriebssystem schnell ob der Server-Thread noch lebt. Wir empfehlen den Server und den Klienten über die Buttons in der Visu zu beenden. Hierbei werden die entsprechenden Ports wie vorgesehen geschlossen und wieder freigegeben.

3.3 telegramframe.py - telegramframe.py

Da die Applikation modularisiert und parallelisiert ausgeführt wird, wird ein einfaches Protokoll benötigt, welches die Module miteinander kommunizieren lassen kann. Dieses ist in der Datei *telegramframe.py* definiert.

3.4 Weitere Datein

Neben den angesprochenen Datein gibt es noch ein paar weitere die für das ausführen des Programms benötigt werden:

- *concol.py* - Enthält Definitionen für farbige Ausgaben in der Konsole

- *__init__.py* - Systemdatei, die es Python erlaubt im Verzeichnis nach benötigten Paketen (imports) zu suchen.
- *log_player_0.log* - Wird von der Applikation erstellt, Protokoll-Datei von Spieler 0
- *log_player_1.log* - Wird von der Applikation erstellt, Protokoll-Datei von Spieler 1
- *log_pong.log* - Wird von der Applikation erstellt, Protokoll-Datei der Hauptschleife