Project Report

Connect 4 Neural Network

Cedric Mingneau (1232611)

William Thenaers (1746077)

Machine Learning incl. Deep Learning
Academiejaar 2019-2020
2020-06-12

Introductie

In volgend verslag bespreken we welk werk en testen we verrichtten om met behulp van machine learning een model te creëren om vier op een rij te spelen, en hopelijk te winnen, tegen op zijn minst een random speler.

Opbouw van het netwerk

Data input

Inputdata bestaat uit een (X, Y)-koppel waarbij X de inputdata van het netwerk voorstelt en Y het label dat we eraan geven. Als Y-waarde nemen we telkens de speler die uiteindelijk gaat winnen en als X-waarde dachten we aan één van de volgende mogelijkheden:

- Bordstaat, het ID van de huidige speler en bijkomend als label ook welke zet de speler gedaan heeft bovenop de uiteindelijke winnaar;
- Bordstaat en het huidige speler ID per cell meegeven.
    - Het speler ID zou dan 42 extra, maar dezelfde, waarden toevoegen zodat het netwerk kan zien dat dit zwaarder doorweegt.
- Bordstaat en het huidige speler ID (in 1D lijst);

We kozen uiteindelijk voor de laatste optie. Het spelbord van 7 op 6 vakjes wordt telkens naar een 1-dimensionale lijst omgevormd en de speler wordt aan het einde toegevoegd, voor een totaal van 43 input nodes. Voor het voorspellen van de winstkans geven we dus buiten het huidige bord ook de speler mee. We kozen hier ook vooral voor omdat de aangeleverde code games omzet naar deze representatie.

Een bijkomende uitbreiding die we later toepasten, was om elk bord te normaliseren naar de huidige speler. Bijvoorbeeld door het volledige bord (en de uiteindelijke winnaar) te vermenigvuldigen met de huidige speler, zodat de huidige speler telkens speler 1 wordt. Zo kunnen we het 43^{ste} element weglaten en in plaats daarvan telkens hetzelfde soort input aan het netwerk geven. Dit zou mogelijk de accuraatheid kunnen verbeteren (zie resultaten), omdat er nu niet meer geleerd moet worden dat de laatste node de huidige speler voorstelt.

Voor bovenstaande inputs maakten we telkens een Fully Connected Neural Network, maar we experimenteerden later ook met de bordstaat in de 2D-matrixvoorstelling te laten staan en in plaats van een FCN een Convolutional NN te gebruiken. Deze had als eerste laag een 4x4 kernel size met 32 filters, maar we probeerden verschillende configuraties en combinaties met extra lagen. We vermoedden dat dit netwerk beter in staat zou zijn om patronen van 4 schijfjes op een rij te herkennen.

Hidden layers

Voor de binnenste lagen van het FCN probeerden we wat topologieën uit met verschillende groottes van lagen en verschillende parameters voor activatiefuncties. We zagen in het algemeen weinig tot geen verschillen tussen de mogelijkheden, dus kozen we voor de standaard `RELU`-activatie op elke laag.

Het eerste FCN dat we trainden had een `Dense` layer met 64 nodes gevolgd door één met 7 nodes. Dit netwerk haalde 60% training accuracy en 59% value accuracy op de testdata. Na nog wat meer uitproberen, gebruikten we uiteindelijk 4 hidden layers met respectievelijk 64, 256, 256 en 42 nodes. Dit haalde 80% training accuracy en rond de 61% value accuracy. Meer dan dat we tot nu toe gezien hadden, maar niet zoveel meer. De veel hogere trainings accuracy duidt wel op een zekere overfitting.

Omdat we aan het begin niet direct wisten hoe we een goed netwerk voor deze toepassing konden opbouwen, kozen we voor een arbitrair aantal lagen en nodes die we daarna probeerden te optimaliseren. Toen we later terug gingen naar het begin en wel eens uittestten wat een FCN zonder hidden layers zou geven, haalde dit een relatief grote winrate tegen de random speler wat ons deed vermoeden dat het met veel minder en kleinere lagen ook wel goed zou kunnen werken. Een nieuw versimpeld FCN werd dan samengesteld bestaande uit de input, 1 `Dense` layer met 84 nodes en de output. Uiteindelijk leek dit simpele netwerk beter te presteren dan de voorgaande, zoals later zal blijken uit de resultaten.

Output

Aangezien de labels die we bij trainen gebruikten drie waarden kunnen hebben, namelijk -1, 0 en 1 afhankelijk van welke speler er wint en of er een gelijkspel is, zal de output van het model dus ook een indicatie moeten geven welke van deze drie categorieën de hoogste kans heeft gegeven de input. De laatste `Dense` layer bestaat dus uit 3 nodes met de `softmax` activatie om af te leiden welke speler het meeste kans maakt om te winnen. Omdat elke outputklasse een integerwaarde heeft en we de kansen willen kunnen vergelijken tussen de 3, kozen we voor de `sparse_categorical_crossentropy` lossfunctie. Voor de optimizerfunctie probeerden we ook de mogelijkheden uit, en kwamen we tot de conclusie dat een goed gevalideerde functie die men tegenwoordig het vaakst gebruikt, het NAdam algoritme implementeert. De gebruikte `Nadam` optimizer functie hebben we ingesteld met een learning rate van `0.001`, iets lager dan de standaard `0.002` volgens keras. Hierdoor duurt de convergentie van het netwerk iets langer, vandaar ook de 15 epochs bij het trainen.

We experimenteerden ook met de parameters van de activatie- en lossfunctie, maar konden daar geen belangrijke verschillen ontdekken.

Het hele punt bij het maken van een model om vier op een rij te spelen, is om te zien hoe goed het speelt tegen andere spelers, of in dit geval een random en een monte-carlo-speler. Waar we bij het spelen kijken of het model wint of verliest, kijken we tijdens het trainen hoe goed het kan bepalen welke speler er uiteindelijk gaat winnen. Met de `accuracy` metric kunnen we dit dan ook nagaan. Hierbij wordt er gekeken in hoeveel van de gevallen een prediction overeenkomt met het werkelijke label van de trainingsset. Zoals eerder vermeld zagen we de trainings accuracy soms wel tot 80% gaan, terwijl de value accuracy op de testdata meestal achterblijft tussen de 60% tot 64%. Om een betere indicatie te krijgen hoe goed een model werkelijk presteert, laten we het telkens spelen tegen de computer. De resultaten hiervan staan in het volgende deel.

Een andere mogelijke output waar we aan dachten was om een laatste laag met 7 nodes te hebben, het aantal kolommen van het bord, die direct de kansen zou geven dat een move in één van die kolommen zou resulteren in een win. Aangezien er in de datasets ook telkens gegeven wordt welke move er gemaakt werd en welke speler zou winnen, zou dit een mogelijke piste zijn geweest. We wisten echter niet precies hoe we dit moesten voorstellen om het in het model te gieten, dus hebben we hier niet verder naar gekeken.

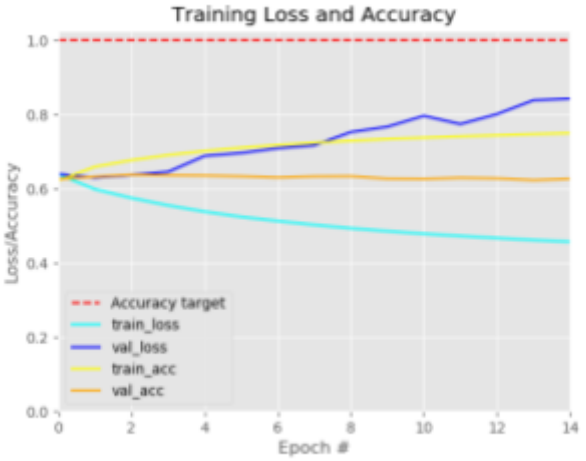
Resultaten

Trainen

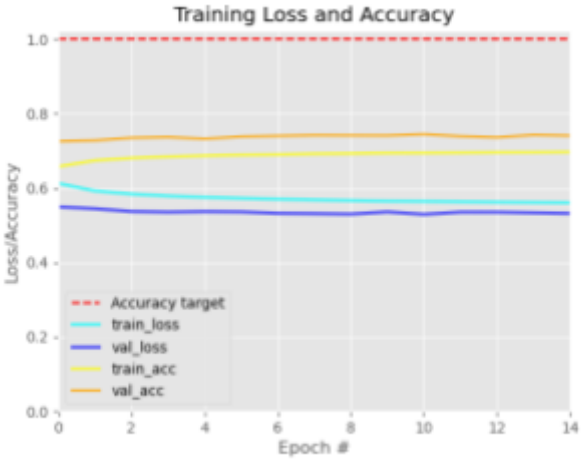
Het trainen gebeurde telkens met 15 epochs en een 85% ratio tussen trainings- en testdata. Dus 85% van de gegenereerde bordstaten (resultaat van het `convert.py` script) worden

voor trainen gebruikt en de overige 15% voor evaluatie tijdens het trainen. In de resultaten hieronder staat ook telkens aangegeven wat de accuraatheid was op de trainingsdata ten opzichte van de testdata bij het fitten van het netwerk.

We gebruikten telkens eerst de gegeven dataset van 10k games en testten de output vervolgens door te spelen tegen één van de geïmplementeerde spelers. Daarna trainden we het beste netwerk opnieuw met de 50k dataset. Hieronder de resultaten van het FCN te trainen met de 50k dataset in Figuur 1 en de resultaten met een samengevoegde dataset in Figuur 2.



Figuur 1: `trained_50k model` resultaten



Figuur 2: `trained_100k-simple` model resultaten

Gebruik

We voerden voor elke testreeks 1000 spellen uit tegen de random speler en tegen de monte-carlo simulatie speler met een diepgang `n` van 3, 5 en 100. Het simuleren van de `n=100` speler duurde het langst, dus hier beperkten we het op 100 spellen. De AI speler van het netwerk is telkens speler `-1`, en speler `1` is dus de voorgemaakte strategie. Natuurlijk kan het netwerk ook predicties maken voor de andere speler. Zo voegden we ook een mogelijkheid in om het AI tegen zichzelf te laten spelen. Zoals gevraagd in de opgave, hebben we alle random number generators geseed op een vaste waarde, namelijk 1232611. Zo zijn de experimenten beter te vergelijken. De training gebeurde wel randomized. Het testen van een model duurt ongeveer 25 tot 30 minuten (op i7 7700k @ 4,6GHz), waarvan de helft van de tijd genomen wordt voor de `n=100` monte-carlo speler..

Om het netwerk effectief te gebruiken, gaan we eigenlijk elke kolom van het huidige spel af en doen een zet in die kolom. Vervolgens vragen we aan het netwerk wat de kans is dat de AI speler (-1) wint na deze zet gedaan te hebben. We selecteren dan de move die de hoogste kans had als resultaat. We voegden ook nog twee extra opties toe aan onze predict functie:

- `check_early_win`: Indien een move resulteert in een 100% win, namelijk als `game.status` de AI speler wordt, dan stoppen we met uitvoeren en gebruiken direct die kolom;
- `prevent_other_win`: Hierbij zoeken we niet enkel de hoogste kans dat de AI speler wint, maar ook de kans dat de andere speler wint, door gewoon hetzelfde te doen, maar nu de andere als extra input te geven aan het model. Indien de andere speler in een kolom een hogere kans heeft dan de AI speler, spelen we in die kolom in de

plaats. Zo voorkomen we dat de andere speler direct zou kunnen winnen en speelt de AI dus iets defensiever.

De eerste optie lijkt heel voor de hand liggend en gebruiken we ook altijd bij het testen. De tweede optie zou mogelijk iets zijn dat een reinforcement learning network wel aangeleerd kan hebben door een negatieve reward te geven als het een move had kunnen doen om niet te verliezen. Voor onze tests hebben we twee keer gemeten, met de optie uit en aan.

Testen

De verkregen accuracies staan in volgende tabellen, telkens met `prevent_other_win` aan. De resultaten zonder deze optie zijn te vinden in de meegeleverde bestanden, maar hier weggelaten. De geteste modellen in Tabel 1 werden getraind op de 10k en 50k datasets, die datasets, maar met de input genormaliseerd naar één speler (zoals eerder vermeld), een nieuwe dataset waar het model zelf tegen de smart player speelde en het versimpeld model. Verder voegden we de 50k datasets samen en trainden het oorspronkelijke netwerk en het versimpelde netwerk. In de "Fitted accuracy" kolom staat de accuracy tijdens trainen ten opzichte van het deel waar het netwerk op traint en dat waarmee het evalueert.

Source training	Fitted accuracy	Test vs random	vs smart (n=3)	(n=5)	(n=100)
10k	81.00/59.51%	98.30%	88.00%	79.00%	13.00%
10k-norm	75.74/62.09%	93,95%	74,90%	61,20%	6.50%
10k AI vs smart	85.69/72.53%	97.50%	87.50%	76.90%	9.00%
10k-simple	68.31/61.33%	99.40%	93.20%	86.10%	17.00%
50k	74.98/62.66%	99.50%	96.50%	90.40%	18.00%
50k-norm	71.48/64.51%	95.40%	79.30%	66.10%	8.00%
50k AI vs smart	82.64/74.68%	98.20%	88.70%	84.40%	15.00%
100k merged	75.92/73.90%	99.70%	97.30%	93.10%	22.00%
100k-simple	69.65/74.04%	99.80%	98.50%	94.90%	33.00%

Tabel 1: testresultaten met verschillende modellen

We zien dat het normaliseren van de input data (door het te vermenigvuldigen met de huidige speler, aangeduid met de `-norm` suffix), voor een slechter resultaat zorgt. De resultaten zonder de `prevent_other_win` waren nog slechter. Verder zien we dat de fitted accuracy veel hoger ligt dan die ten opzichte van de test set, dit duidt op overfitting. Het lukt het netwerk niet om de test accuracy te verbeteren bij training. Deze blijft redelijk constant en telkens ~10% lager dan die van de trainingsset. Bij het versimpelde model lijkt dit veel beter.

Het model getraind met de 50k dataset wint al 90% van de spellen tegen de monte-carlo speler met `n=5`, wat ons toch een goed resultaat lijkt. Tegen de random speler, verloor het slechts 5 spellen van de 1000. Het model getraind met de 100k dataset presteert zoals verwacht ook beter door een grotere sample set. Het versimpelde model met de 100k dataset presteert ronduit het beste in elke test. Al deze resultaten zijn veel beter dan het geval waar 2 random spelers het tegen elkaar opnemen. In dat geval is de winkans gemiddeld rond de 50%.

De resultaten van het CNN dat we uittestten, staan niet in bovenstaande tabel, maar presteerde veel slechter, met een winrate van slechts 60% tegen de random speler. Dit is

waarschijnlijk het geval omdat convolueren over zo'n klein "beeld" het niet waard is om te doen en veel precisie verliest over de staat van elk board.

De test logs worden meegeleverd met dit verslag, indien men wilt nakijken bij hoeveel spellen het gelijkspel was.

Nieuwe data genereren

We gebruikten het voorheen best presterende model (50k) om een nieuwe dataset te genereren door het terug tegen de monte-carlo speler met `n=10` te laten spelen. Omdat python met MPI niet werkte op Windows (zo werkt de oorspronkelijke dataset gegenereerd), gebruikten we een `ProcessPoolExecutor` in de plaats waarbij het model dan op de CPU in 8 processen wordt uitgevoerd. Het genereren van 50k nieuwe spellen duurde ongeveer 2 uur en 41 minuten. Opnieuw gebeurde enkel het testen met de eerder vermelde vaste random seed. Ook stonden de twee extra opties aan, omdat dit steeds de beste resultaten opleverde.

We merken dat de training en test accuracy veel groter is dan voorheen, wellicht te wijten aan het feit dat het netwerk beter in staat is om zijn eigen spel opnieuw te leren dan een andere speler. De accuraatheid ten opzichte van de andere spelers is nu iets lager dan voorheen. Een betere dataset was dit dus niet, maar we hebben ze gebruikt om samen te voegen met de eerste 50k dataset om een 100k set te maken. Dit model levert zoals hierboven in de tabel te zien is, het beste resultaat op tegen alle spelers.

Conclusie

Na het onderzoeken van de mogelijkheden voor FCNs en CNNs binnen tensorflow en keras, kwamen we tot een relatief goed werkend model en een beter begrip van alle technieken die men gebruikt om deze modellen op te stellen.