

GOochelen met kunstmatige intelligentie

Raf Brenninkmeijer & Aron Hardeman

14 januari 2022

Hoofdstuk 1

Inleiding

Van spraakherkenning tot lopende robots, van zelfrijdende auto's tot slimme vertaalmachines, van zoekmachines tot het automatisch interpreteren van MRI-scans: kunstmatige intelligentie komt steeds meer voor in onze samenleving. We komen op een punt dat onze handelingen kunnen worden overgenomen door computers. Denk maar aan bijvoorbeeld zelfrijdende stofzuigers of chatbots. Dit zorgt wel voor de benodigde ethische vragen, zoals hoe we de samenleving moeten vormgeven met robots, of hoe ervoor moet worden gezorgd dat robots niet de wereld overnemen. Voor dit relatief kleinschalige profielwerkstuk maakten we ons niet zoveel zorgen over deze vragen. Wij waren eerder geïnteresseerd in de vraag: "Hoe kan een computer leren?" We hebben zelfs een eigen neurale netwerk gemaakt voor het bordspelletje Go, want onze hoofdvraag luidde: "**Hoe programmeer je een instantie van kunstmatige intelligentie die het spelletje Go speelt?**" We benoemen de verschillende technieken die gebruikt worden om een algoritme te trainen. Dit leggen we ook uit op wiskundig vlak; het is immers een profielwerkstuk voor het vak wiskunde. Een belangrijk onderdeel van het trainen van algoritmes, is backpropagation. Wij hebben in dit profielwerkstuk onder andere de backpropagation-formules bewezen met behulp van de (voor veel variabelen gegeneraliseerde) kettingregel. Verder laten we soms stukjes code van ons zelfgemaakte programma zien, die horen bij het onderwerp waar het dan over gaat.

Ons programma hebben we geschreven in de programmeertaal C++. Het omvat de volgende onderdelen:

- implementatie van de regels van Go
- implementatie van het neurale netwerk en bijbehorende trainingsalgoritmen
- implementatie van een Graphical User Interface (GUI)

Waar wij ons bij dit profielwerkstuk mee proberen te onderscheiden, is het feit dat wij niet een Machine Learningbibliotheek van internet hebben gehaald en die hebben gebruikt, maar dat wij, behalve de regels van Go en de GUI, ook de algoritmen horend bij kunstmatige intelligentie zelf hebben geprogrammeerd. ‘From scratch’ dus, zoals je het zou kunnen noemen. Dit betekende wel dat we ons ook moesten verdiepen in hoe kunstmatige intelligentie werkt, in zijn meest basale vorm.

Omdat het hier kunstmatige intelligentie betreft, waren er zeer veel berekeningen nodig. We merkten zelfs dat onze eigen laptop hiervoor wel een beetje traag werd. Daarom hebben we aangeklopt bij de Rijksuniversiteit Groningen en gevraagd of we gebruik mochten maken van hun rekencluster, en dat mocht, na een video-overleg met hen. Door middel van parallelisatie van de code, was het nu mogelijk om maar liefst tweeeindertig partijen tegelijk gespeeld te laten worden. Dit is meer dan het aantal dat bij een (normale) laptop haalbaar is.

Dit profielwerkstuk is niet in Word of Google Docs gemaakt, maar in L^AT_EX. Dit was handig, want zo konden we relatief gemakkelijk grote vergelijkingen in het document krijgen. Het gebruiken van L^AT_EX betekent echter wel dat je een beetje moet ‘programmeren’, in tegenstelling tot bij Word.

Eindproduct van dit profielwerkstuk is een algoritme waartegen je Go kunt spelen, op een 19x19-bord.

We hopen dat de lezer geniet van het lezen van dit profielwerkstuk en van het spelen tegen ons algoritme.

Inhoudsopgave

1 Inleiding	1
2 Wiskundige achtergronden	6
2.1 De afgeleide van een normale functie	6
2.2 De partiële afgeleide	7
2.3 De gradiënt	8
2.4 De kettingregel	9
2.4.1 Functies met één variabele	9
2.4.2 De kettingregel bij functies met veel variabelen	10
3 De regels van Go	12
3.1 Stenen slaan	12
3.2 Ko	14
3.3 Het einde van de partij	14
3.4 De regels van Go in C++	15
4 Neurale netwerken	18
4.1 Wat is een neuraal netwerk?	18
4.1.1 Verdeling in lagen	19
4.1.2 De informatiestroom door een neuraal netwerk	19
4.1.3 De werking van één neuron	19
4.1.4 De invoer- en uitvoerlaag	21
4.2 Praktisch nut en toepassing van een neuraal netwerk	22
4.3 Precieze naamgeving van variabelen	23
4.3.1 Naamgeving van variabelen in de code	24
4.4 Een compleet overzicht van variabelen	25
4.5 De informatiestroom door het netwerk in C++	25
4.6 Het trainen van een neuraal netwerk	28
4.7 Q-learning	33
4.7.1 Q-learning combineren met een neuraal netwerk	34
4.7.2 Q-learning in code	35

4.8	Xavier initialisatie	37
4.9	Het neurale netwerk speelt altijd met de zwarte stenen	38
4.10	Overfitting	39
4.11	Update: overfitting?	41
5	Het bewijs en de code van backpropagation	44
5.1	Het bepalen van $\frac{\partial C}{\partial b[L][n]}$	45
5.2	Het bepalen van $\frac{\partial C}{\partial w[L][n][m]}$	49
5.3	Van uitvoerlaag naar alle lagen	50
5.4	Het bepalen van $\frac{\partial C}{\partial a[l][n]}$	52
5.5	Het bepalen van $\frac{\partial C}{\partial b[l][n]}$ en $\frac{\partial C}{\partial w[l][n][m]}$	55
5.6	Backpropagation in C++	56
5.6.1	Het bepalen van de gradiënt	56
6	Informatieoverzicht van functies en afgeleiden	60
6.1	De sigmoïdefunctie	60
6.2	De (Leaky) Rectified Linear Unitfunctie	61
6.3	De kwadratische kostenfunctie	62
6.4	De cross-entropy kostenfunctie	64
6.5	Sigmoïde en cross-entropy gaan goed samen	65
7	Het rekencluster van de RUG	67
7.1	Het parallel spelen van meerdere Go-partijen	67
7.2	Toch werkt het niet...	69
7.3	Een oplossing?	71
7.4	Driemaal is scheepsrecht	72
7.5	Meer cores	74
8	Meer halen uit een neuraal netwerk: vooruitkijken	76
8.1	De standaardmanier om met het netwerk een zet te bedenken	76
8.2	Verder vooruitkijken	79
8.2.1	Komi-breinbrekers	83
8.2.2	Geen goede resultaten	84
8.3	Recursief nóg verder vooruitkijken zonder Fixed Athena	84
9	Resultaten: ons eindproduct	89
9.1	Bordgrootte 9x9 en vooruitkijken	89
9.1.1	Standaardmanier	90

9.1.2	Extra vooruitkijken	90
9.2	Bordgrootte 13x13	91
9.3	Bordgrootte 19x19	92
10	Conclusie	94
11	Discussie	95
11.1	Dumb Athena en Fixed Athena	95
11.1.1	Fixed Athena	95
11.1.2	Randomiseren	96
11.1.3	Dumb Athena	97
11.2	Potentiële verbeteringen aan het vooruitkijkalgoritme	98
12	Nawoord	99
12.1	Dankwoord	100

Hoofdstuk 2

Wiskundige achtergronden

In dit profielwerkstuk wordt veel wiskunde gebruikt. Het is nodig om eerst deze wiskundige onderwerpen zelf te begrijpen, zodat daarna ook hun toepassing in de kunstmatige intelligentie kan worden begrepen. In dit hoofdstuk leggen wij de wiskundige onderwerpen uit die we in de rest van het profielwerkstuk zullen gebruiken.

2.1 De afgeleide van een normale functie

Het wordt bekend verondersteld wat de afgeleide functie en haar nut is. Kort gezegd geeft de afgeleide van een functie de helling van de oorspronkelijke functie weer op haar domein. Enkele voorbeelden:

$$f(x) = 5x^4 + 2 \quad \text{geeft} \quad f'(x) = 20x^3 \quad (2.1)$$

$$g(x) = \ln\left(\frac{2}{x}\right) \quad \text{geeft} \quad g'(x) = \frac{1}{\left(\frac{2}{x}\right)} \cdot -\frac{2}{x^2} = -\frac{1}{x} \quad (2.2)$$

$$h(t) = \sqrt{1 + e^t} \quad \text{geeft} \quad h'(t) = \frac{1}{2\sqrt{1 + e^t}} \cdot e^t = \frac{e^t}{2\sqrt{1 + e^t}} \quad (2.3)$$

Let op, bij het differentiëren van de functies $g(x)$ en $h(x)$ is de kettingregel gebruikt. In het vervolg van dit profielwerkstuk veronderstellen we dat de regels van het differentiëren de lezer bekend zijn. We zullen vanaf nu niet meer steeds expliciet vermelden welke differentiatieregels zijn gebruikt.

De afgeleide kan ook anders genoteerd worden, in de Leibniz-notatie, zodat duidelijker is welke variabele er verandert (Khan Academy, z.d.-a):

$$f'(x) = \frac{d}{dx} f(x) \quad (2.4)$$

Hierboven is het bijvoorbeeld de x -variabele die je verandert en de $f(x)$ -functie die reageert op deze verandering. Je krijgt dus (dit zijn twee van dezelfde functies als hierboven):

$$f(x) = 5x^4 + 2 \quad \text{geeft} \quad \frac{d}{dx} f(x) = 20x^3 \quad (2.5)$$

$$h(t) = \sqrt{1 + e^t} \quad \text{geeft} \quad \frac{d}{dt} h(t) = \frac{e^t}{2\sqrt{1 + e^t}} \quad (2.6)$$

Deze notatie blijkt nuttig te zijn bij het differentiëren van functies die meer dan één variabele als *input* krijgen. Dit bekijken we in de volgende sectie.

2.2 De partiële afgeleide

Als bron bij deze sectie hebben we Khan Academy (z.d.-b). Stel nu dat we een functie hebben die drie toevoerwaarden, x , y en z , heeft:

$$f(x, y, z) = x^2 + \sqrt{y} + \ln(z) \quad (2.7)$$

Deze functie heeft niet één, maar drie afgeleiden. Deze afgeleiden noemen we *partiële afgeleiden*. We kunnen bijvoorbeeld de partiële afgeleide nemen als x verandert. De regel die we dan hanteren is dat we de andere variabelen, in dit geval y en z , zien als een constant getal, bijvoorbeeld 3. We differentiëren ook alsof y en z constant zijn. Maar dan zijn \sqrt{y} en $\ln(z)$ ook constant. Deze termen vallen bij het differentiëren dus weg. De partiële afgeleide van f als x verandert, is dus:

$$\frac{\partial}{\partial x} f(x, y, z) = \frac{\partial}{\partial x} f = 2x + 0 + 0 = 2x \quad (2.8)$$

Merk op dat we nu niet meer $\frac{d}{dx}$ gebruiken, maar $\frac{\partial}{\partial x}$. Het ∂ -symbool geeft aan dat het om de partiële afgeleide gaat. In plaats van $\frac{\partial}{\partial x} f(x, y, z)$ hebben we omwille van het gemak $\frac{\partial}{\partial x} f$ opgeschreven. De betekenis blijft hetzelfde.

Zo kunnen we ook de partiële afgeleide berekenen als y verandert. Constant zijn dan x en z , en dus zijn x^2 en $\ln(z)$ ook constant. Hun afgeleide wordt 0. De gezochte partiële afgeleide wordt:

$$\frac{\partial}{\partial y} f = 0 + \frac{1}{2\sqrt{y}} + 0 = \frac{1}{2\sqrt{y}} \quad (2.9)$$

Als laatste differentiëren we de functie als z verandert en x^2 en \sqrt{y} dus constant zijn:

$$\frac{\partial}{\partial z} f = 0 + 0 + \frac{1}{z} = \frac{1}{z} \quad (2.10)$$

Dit is dus de partiële afgeleide van f naar z .

2.3 De gradiënt

Als bron bij deze sectie hebben wij Khan Academy (z.d.-c) gebruikt. We gaan verder met de in de vorige sectie benoemde functie $f(x, y, z) = x^2 + \sqrt{y} + \ln(z)$. Deze functie heeft drie toevoerwaarden (x, y, z) en voert één waarde uit. De drie partiële afgeleiden zijn $\frac{\partial}{\partial x} f$, $\frac{\partial}{\partial y} f$ en $\frac{\partial}{\partial z} f$. Zetten we deze onder elkaar in een vector, dan krijgen we:

$$\nabla f = \begin{pmatrix} \frac{\partial}{\partial x} f \\ \frac{\partial}{\partial y} f \\ \frac{\partial}{\partial z} f \end{pmatrix} = \begin{pmatrix} 2x \\ \frac{1}{2\sqrt{y}} \\ \frac{1}{z} \end{pmatrix} \quad (2.11)$$

Deze vector heet de gradiënt en noteren we met ∇f . Het aantal componenten dat de gradiënt heeft, is altijd gelijk aan het aantal variabelen dat als toevoer dient van de functie. In dit geval zijn dat er drie.

De gradiënt is van groot praktisch nut, ook op het gebied van kunstmatige intelligentie. **Het blijkt namelijk dat multidimensionale functies het snelst stijgen als je de invoerwaarden beweegt in de richting van de gradiënt** op een punt. Voorbeeld (de functie f bestaat voor alle x -waarden, waarbij $y \geq 0$ en $z > 0$): op het punt $(1, 2, 3)$ geldt

$$\nabla f = \begin{pmatrix} 2 \cdot 1 \\ \frac{1}{2\sqrt{2}} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} 2 \\ \frac{1}{4}\sqrt{2} \\ \frac{1}{3} \end{pmatrix}. \quad (2.12)$$

Dit houdt in dat als je op het punt $(1, 2, 3)$ zit, de functie f het snelst stijgt als je aan x , y en z een zeer klein (positief) getal toewoegt in de verhouding respectievelijk $2 : \frac{1}{4}\sqrt{2} : \frac{1}{3}$.

Als we de andere kant op gaan, in de richting van $-\nabla f$, zal de functie onzes inziens dan juist het snelst dalen. Vooral daarin zijn we bij dit profielwerkstuk geïnteresseerd.



Dit verkeersbord ziet eruit als een gradiënt. Het pijtje boven het bord, in het verkeerslicht, valt te interpreteren als geheugensteuntje dat de gradiënt een vector is. De fiets heeft geen wiskundige betekenis. Eigen foto

2.4 De kettingregel

In dit profielwerkstuk gebruiken we functies met meerdere variabelen als invoer. We willen deze functies wel kunnen differentiëren en daarom zullen we nu bespreken hoe de kettingregel in dit soort gevallen werkt. Hiervoor kijken we echter eerst nog even kort naar de kettingregel bij een ‘normale’ functie.

2.4.1 Functies met één variabele

Stel dat we de functie $f(x) = (5x^2 + 3x + 2)^{20}$ hebben, die we willen differentiëren.

We kunnen dit ook als volgt schrijven:

$$g(x) = 5x^2 + 3x + 2 \quad (2.13)$$

$$f(g) = g^{20} \quad (2.14)$$

Als we gaan differentiëren, krijgen we dit:

$$\frac{dg}{dx} = 10x + 3 \quad (2.15)$$

$$\frac{df}{dg} = 20g^{19} \quad (2.16)$$

Differentiëren van de oorspronkelijke functie $f(x) = (5x^2 + 3x + 2)^{19}$ (we gebruiken de kettingregel):

$$\frac{df}{dx} = 20(5x^2 + 3x + 2)^{19} \cdot (10x + 3) \quad (2.17)$$

We zien dat nu geldt dat

$$\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}. \quad (2.18)$$

De bovenstaande regel is de *kettingregel* (Stewart, 2011, p. 199).

2.4.2 De kettingregel bij functies met veel variabelen

Tot zover ging het over functies met één variabele. Nu gaan we dit uitbreiden naar ingewikkeldere functies.

Als f een functie is van de variabelen x_1, x_2, \dots, x_n , en als elke x_j op zijn beurt een functie is van de variabelen t_1, t_2, \dots, t_m , dan is f een functie van de variabelen t_1, t_2, \dots, t_m en vindt men de partiële afgeleide van f als t_i verandert, als volgt ($1 \leq i \leq m$) (Stewart, 2011, p. 927):

$$\boxed{\frac{\partial f}{\partial t_i} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t_i} + \dots + \frac{\partial f}{\partial x_n} \frac{\partial x_n}{\partial t_i}} \quad (2.19)$$

Opdat de lezer dit beter begrijpt, geven we hier een klein voorbeeld van. Zij f een functie van de variabelen x_1 en x_2 , die beide functies zijn van de onafhankelijke variabelen t_1 en t_2 :

$$f = x_1 \cdot e^{x_2} \quad (2.20)$$

$$x_1 = t_1 + 2t_2 \quad (2.21)$$

$$x_2 = (t_1)^2 + t_2 \quad (2.22)$$

We willen $\frac{\partial f}{\partial t_1}$ weten. Volgens de bovenstaande algemene formule geldt nu:

$$\frac{\partial f}{\partial t_1} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t_1} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t_1} \quad (2.23)$$

We kunnen hiervan het antwoord bepalen in tussenstappen:

$$\frac{\partial f}{\partial x_1} = e^{x_2} \quad \frac{\partial x_1}{\partial t_1} = 1 \quad (2.24)$$

$$\frac{\partial f}{\partial x_2} = x_1 \cdot e^{x_2} \quad \frac{\partial x_2}{\partial t_1} = 2t_1 \quad (2.25)$$

Dit weer invullen in de formule boven, geeft:

$$\frac{\partial f}{\partial t_1} = e^{x_2} \cdot 1 + x_1 \cdot e^{x_2} \cdot 2t_1 \quad (2.26)$$

$$\frac{\partial f}{\partial t_1} = e^{x_2} + 2x_1 t_1 e^{x_2}$$

(2.27)

Dit is hoe de kettingregel bij ingewikkeldere functies gebruikt kan worden.

Hoofdstuk 3

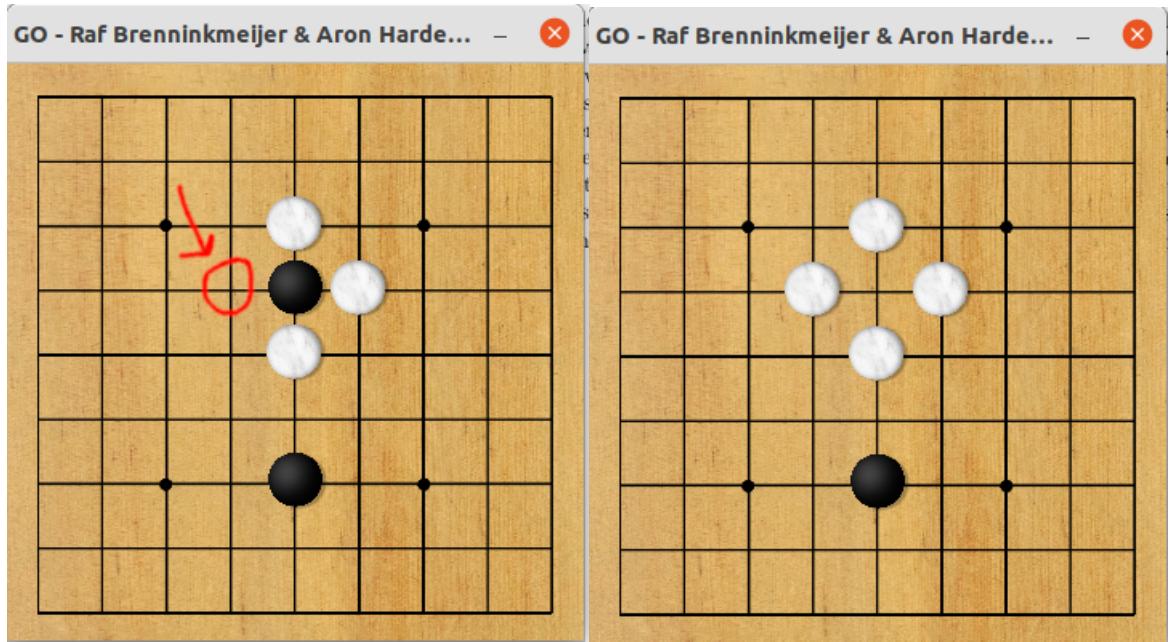
De regels van Go

Voor dit hoofdstuk gebruiken we de bronnen British Go Association (2017) en Sensei's Library (2021).

We maken een kunstmatige intelligentie op het spelletje Go, maar wat is Go eigenlijk? Go is een bordspel oorspronkelijk uit Oost-Azië. Het spel wordt officieel gespeeld op een bord van 19 bij 19. Snelle potjes worden gespeeld op een bord van 13 bij 13. Voor beginners wordt een bord van 9 bij 9 aangeraden. Het is een spel voor twee spelers. Eén speler heeft zwarte stenen en de andere speler heeft witte stenen. De spelers mogen om de beurt een steen op het bord zetten. In tegenstelling tot schaken mag zwart beginnen en is een zet niet in een vakje maar op het snijpunt van de lijnen van het bord. Het doel van het spel is om zo veel mogelijk territorium te verzamelen.

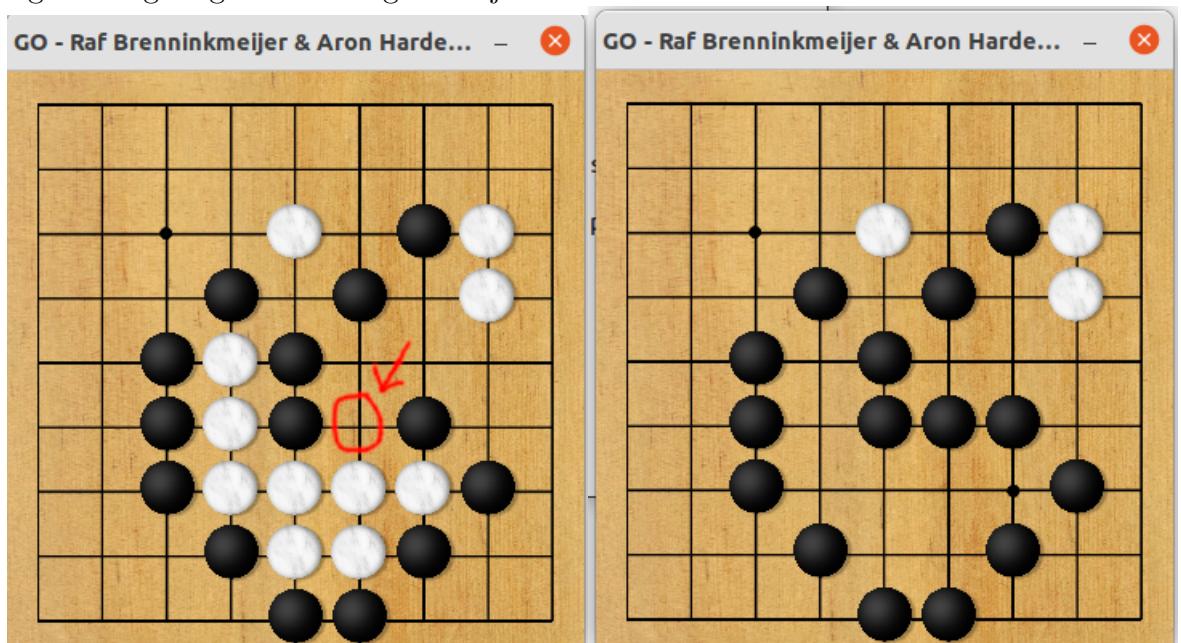
3.1 Stenen slaan

Een belangrijk onderdeel van het spel is het slaan van de stenen. Iedere steen heeft een bepaald aantal vrijheden. Dit zijn de vrije plekken horizontaal en verticaal op het bord. Een steen in het midden van een leeg bord heeft dus vier vrijheden. Een steen aan de zijkant heeft drie vrijheden. En een steen in de hoek heeft maar twee vrijheden. Wanneer een steen geen vrijheden meer heeft, dus deze wordt volledig omringd door de tegenovergestelde kleur, dan is de steen geslagen. Zie de afbeeldingen.



Hier wordt een steen geslagen. Links voor de slagzet, rechts na de slagzet.

Het is ook mogelijk om meerdere stenen van dezelfde kleur aan elkaar te hebben als een lange streng. Deze streng is alleen horizontaal of verticaal verbonden dus niet diagonaal. Het slaan van een streng gaat hetzelfde. Een streng wordt geslagen als deze geen vrijheden meer heeft.



Hier wordt een streng van stenen geslagen. Links voor de slagzet, rechts na de slagzet.

3.2 Ko

Het is niet mogelijk om een zet te doen die ervoor zorgt dat de steen van de speler zelf wordt veroverd. Er is wel een situatie denkbaar dat de speler een zet kan doen die ervoor zorgt dat de steen wordt veroverd, ware het niet dat door die zet de stenen van de tegenstander geen vrijheden meer hebben. Dan is de zet wel legaal. Een erg speciale regel in Go is de ko-regel. Deze regel stelt dat een situatie nooit herhaald mag worden. In de situatie hieronder is te zien dat het spel eindeloos door kan gaan door steeds de stenen te blijven veroveren. Om deze oneindige staat te vermijden, is de regel ko dus bedacht. Er zijn ook wel andere varianten op de ko-regel, maar in dit PWS maken we alleen gebruik van de positionele ko-regel. Ko maakt Go stukken interessanter.



Als de ko-regel er niet was, zou dit oneindig lang door kunnen gaan. De ko-regel zegt dat een positie niet herhaald mag worden en verhindert zodoende een oneindige herhaling.

3.3 Het einde van de partij

Het spel eindigt als beide spelers direct achtereenvolgens hebben gepast, oftewel een zet overgeslagen hebben. Een zet overslaan mag te allen tijde. De spelers tellen al hun territorium bij elkaar op. Het territorium van een bepaalde kleur is het gedeelte van lege velden dat wordt ingesloten door een bepaalde kleur, plus het aantal stenen van de betreffende kleur zelf. De speler met het meeste territorium wint. Omdat zwart begint en dus een voorsprong heeft, wordt wanneer het spel is afgelopen, een bepaalde waarde van zwarts territorium ervan afgehaald. Deze waarde heet komi. Deze waarde is een afspraak tussen de twee spelers. Bij een bord van 19x19, 13x13 en 9x9, wordt in ons profielwerkstuk de waarde 6,5 gebruikt. Bij het 5x5-bord hebben wij als

komi-waarde 24 gekozen, maar in de eindversie van het profielwerkstuk kun je niet meer 5x5 kiezen als bordgrootte (dit was bedoeld als test). Bij elke nu nog beschikbare bordgrootte is de komiwaarde dus 6,5 in dit profielwerkstuk.

3.4 De regels van Go in C++

We hebben een programma geschreven waarin Go gespeeld kan worden. In deze paragraaf worden twee belangrijke functies beschreven: het veroveren van stenen en de ko regel. Hieronder staat de functie voor het veroveren van stenen. Dit is een bool-functie, dus als uitvoer geeft het of true of false. De functie kijkt op de plaats x,y of er vrijheden zijn. Als die er niet zijn dan gaat de functie verder door naar boven, onder, links en rechts te kijken of deze steentjes vrijheden hebben. In dit programma wordt dit mogelijk door het gebruik van een recursieve functie. Hierin wordt in de functie naar de functie zelf verwezen. Dus als er geen vrijheid op het blokje zelf is, dan wordt de functie zelf weer aangeroepen, maar dan met andere x en y waarden. Er wordt ook gekeken of de initial_function_call true is. Als dat zo is, dan heeft de functie nog geen ander blokje bezocht en wordt visited helemaal leeg gemaakt. De visited vector is een tweedimensionale vector die wordt veranderd als het steentje al is bezocht. Als de streng geen vrijheden meer heeft, dan worden alle steentjes die zijn bezocht van het bord afgehaald.

```
bool has_liberties(int y, int x, bool initial_function_call) {
    if(initial_function_call) {
        // ^ if this function is now NOT recursively called
        // but was called from another function
        //we must now 'reset' the following things:
        at_least_one_liberty_is_found = false;
        for(int q = 0; q < N; q++) {
            for(int w = 0; w < N; w++) {
                visited[q][w] = false;
            }
        }
    }
    //Notice that, after this function has been called,
    //the visited-vector is NOT yet reset.
    //This means that we can see the string of
    //connected Go stones evenafter this function is ended.
    //These have visited[y][x] = true.
    //This can be useful: when we place a block,
    //we can check whether
    //its neighbors have liberties or not.
    //And if this function says the neighbor has no liberties,
    //then we can at once delete
}
```

```

//the stones for which visited[y][x] is true.
//These stones have thenbeen captured.
visited[y][x] = true;
//^this means that the current position/block has been visited.
if (x > 0) {
    if(board[y][x-1] == 0) {
        at_least_one_liberty_is_found = true;
    }
    if(board[y][x-1] == board[y][x] && visited[y][x-1] == false) {
        has_liberties(y,x-1, false);
    }
}
if (x < N-1) {
    if(board[y][x+1] == 0) {
        at_least_one_liberty_is_found = true;
    }
    if(board[y][x+1] == board[y][x] && visited[y][x+1] == false) {
        has_liberties(y,x+1, false);
    }
}
if (y > 0) {
    if(board[y-1][x] == 0) {
        at_least_one_liberty_is_found = true;
    }
    if(board[y-1][x] == board[y][x] && visited[y-1][x] == false) {
        has_liberties(y-1,x, false);
    }
}
if (y < N-1) {
    if(board[y+1][x] == 0) {
        at_least_one_liberty_is_found = true;
    }
    if(board[y+1][x] == board[y][x] && visited[y+1][x] == false) {
        has_liberties(y+1,x, false);
    }
}
if(initial_function_call) {
    return at_least_one_liberty_is_found;
} else {
    return false;
}
}

```

De volgende functie is de ko regel. Het lastige aan deze functie is dat een oude positie vergeleken moet worden met de huidige positie. Zoals te zien is hieronder, wordt de ko regel aangeroepen in de functie move_is_legal. Deze functie kijkt of een zet legal is. In deze functie wordt het bord nog niet veranderd. Er moet dus een nieuwe functie gecreëerd worden die kijkt

wat de positie wordt wanneer de speler een zet. Deze doet precies hetzelfde als de has_liberties functie, maar verandert een potentieel bord in plaats van het echte bord. Vervolgens vergelijkt de functie vectorcontains alle oude bordposities v1 met de potentiële bordpositie. v1 of former_positions is een driedimensionale vector die voor elke zet een bord bevat. Met een for-loop gaat de functie elk bord bij langs. Als het potentiële bord hetzelfde is als het oude bord, dan geeft de functie een true uitvoer. Dan zegt de ko regel dat deze zet niet kan en dan moet de speler een nieuwe zet voorstellen.

```

bool vectorcontains(std::vector<std::vector<std::vector<int> > > &v1,
                    std::vector<std::vector<int> > &v2) {
    for(int i = 0; i < v1.size(); i++) {
        if(v1[i] == v2) {
            return true;
        }
    }
    return false;
}

bool move_is_legal(int choice_y, int choice_x, int player) {
    ...
    ...
    ...
    //KO RULE:
    if (vectorcontains(former_positions, potential_board)) {
        return false;
    }
}

```

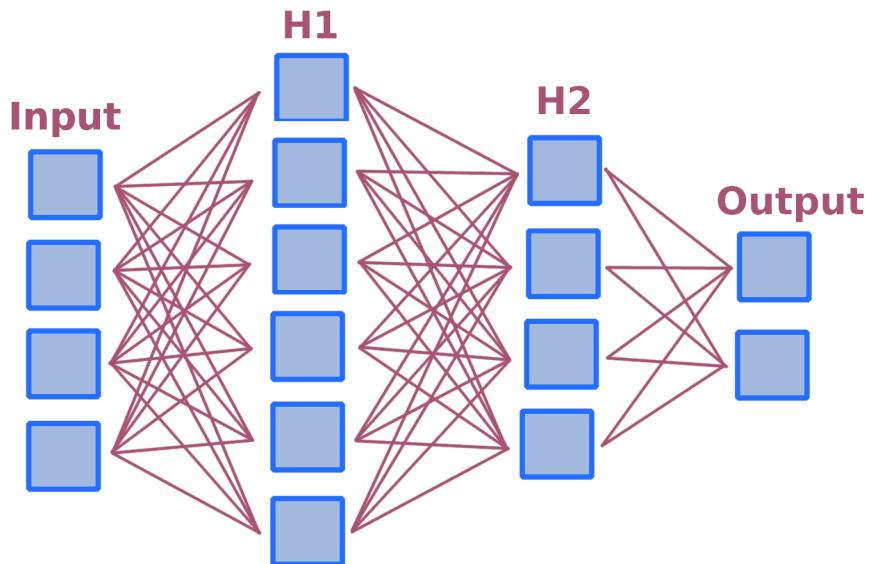
Hoofdstuk 4

Neurale netwerken

We gaan nu de wiskunde die we tot nog toe hebben besproken, concreet toepassen op het gebied van kunstmatige intelligentie (KI).

4.1 Wat is een neuraal netwerk?

Deze sectie is ontleend aan Nielsen (2015, Hoofdstuk 1). Een neuraal netwerk bestaat kort gezegd uit twee onderdelen: neuronen, en de verbindingen tussen neuronen. Hieronder staat een weergave van een neuraal netwerk:



4.1.1 Verdeling in lagen

De blauwe rechthoekjes stellen de neuronen voor, die in de tekening met paarse lijntjes verbonden zijn. We zien dat in de tekening de neuronen in lagen (kolommen) gerangschikt zijn. Boven die lagen staat de naam van de laag. Helemaal links heb je de vier neuronen die samen de *input layer/toevoerlaag* vormen, en helemaal rechts heb je de twee neuronen die samen de *output layer/uitvoerlaag* vormen. Tussen deze twee lagen in zijn twee *hidden layers/verborgen lagen* getekend; deze zijn in de tekening aangegeven met H1 en H2. Het idee is dat het netwerk van links naar rechts wordt doorlopen om van een invoer een uitvoer te maken. We zullen straks laten zien hoe dat in zijn werk gaat.

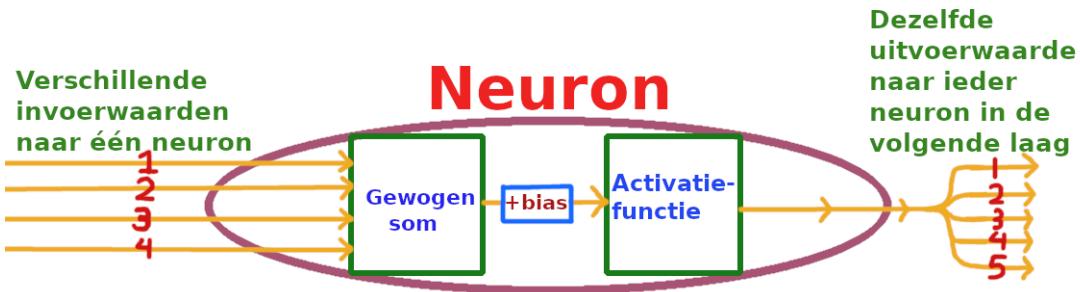
4.1.2 De informatiestroom door een neuraal netwerk

We gaan verder met het hierboven afgebeeld netwerk. Het doet er nu niet toe waar deze vier getallen voor staan en wat het nut is van dit netwerk; het gaat er nu om dat we begrijpen *hoe* het netwerk werkt.

We hebben vier invoerneuronen; dit betekent dat het netwerk vier getallen als invoer neemt. De vier neuronen in de invoerlaag helemaal links geven deze vier getallen als uitvoer door aan de neuronen rechts van hen. Ieder neuron in deze eerste verborgen laag (de tweede laag van links) krijgt zo vier invoerwaarden. Deze invoerwaarden zet het neuron met een functie om naar één getal. Op dat ene getal wordt de zogeheten *activation function/activatiefunctie* losgelaten. De uitvoer van deze functie is de uitvoer van het neuron. Deze uitvoeren worden vervolgens weer doorgegeven aan de neuronen in de tweede verborgen laag van het netwerk. Hetzelfde wordt voor deze laag herhaald en de uitvoeren van de neuronen in de tweede verborgen laag, worden de invoeren van de neuronen in de uitvoerlaag. De neuronen in de uitvoerlaag zetten dit weer om naar een uitvoer. De uitvoer van de neuronen in de uitvoerlaag, is de uiteindelijke uitvoer van het hele netwerk, bij deze ene gegeven invoer.

4.1.3 De werking van één neuron

Hieronder staat in een plaatje de werking van één neuron aangegeven.



Als we naar het plaatje kijken, zien we dat er vier pijltjes in het neuron gaan. Dit betekent dat we ervan uit zijn gegaan dat er in de laag links van het getekende neuron, vier neuronen zijn die alle vier hun uitvoerwaarde aan dit neuron geven.

De vier pijltjes die vanuit een ander neuron, dit neuron ingaan, zou je interneuronale verbindingen kunnen noemen. Bij elk van deze verbindingen hoort ook een zogenaamd *weight/gewicht*, een getal dat aangeeft hoe zwaar deze verbinding meebeleeft in de berekening. We kunnen dan de gewogen som van de toevoerwaarden nemen. Als we de toevoerwaarden i_1, i_2, i_3 en i_4 noemen, en de bijbehorende gewichten w_1, w_2, w_3 en w_4 , is hun gewogen som

$$\sum_k i_k w_k = i_1 w_1 + i_2 w_2 + i_3 w_3 + i_4 w_4 \quad (4.1)$$

Merk op dat de notatie links hier aangeeft dat je sommeert over alle toevoerwaarden k .

Aan deze gewogen som voegen wij dan nog een zogenaamde *bias* toe. Dit is één getal dat per neuron verschillend is en bij het neuron hoort. We noemen de bias b . De gewogen som van de toevoerwaarden met de bias erbij opgeteld, noemen we hier s (van bijvoorbeeld het woord ‘som/sum’, maar vergeet de bias niet). We krijgen dus:

$$s = b + i_1 w_1 + i_2 w_2 + i_3 w_3 + i_4 w_4 \quad (4.2)$$

of algemener (bij een variabel aantal toevoerwaarden),

$$s = b + \sum_k i_k w_k \quad (4.3)$$

Zoals in de schematische tekening van een neuron hierboven te zien is, voeren we deze s weer in in de zogenaamde *activatiefunctie* (activation function). Deze activatiefunctie noemen we $a(s)$. Hierop gaan we later in.

De uitvoer $a(s)$ van deze activatiefunctie is ook de uitvoer van het neuron zelf. In de schematische weergave staan vijf pijlen. Hieraan kan men zien dat

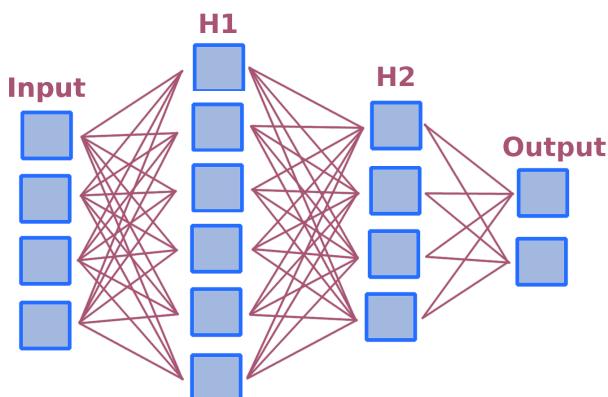
er zich in de volgende laag van het netwerk blijkbaar vijf neuronen bevinden. Deze vijf neuronen krijgen van het neuron dat we nu hebben behandeld alle vijf dezelfde invoerwaarde, maar de gewichten van de interneuronale verbindingen naar elk van de vijf neuronen, zijn in principe wel verschillend.

De neuronen in de volgende laag herhalen het zojuist beschreven proces, en zo stroomt de informatie van links naar rechts door het netwerk.

4.1.4 De invoer- en uitvoerlaag

We weten nu (globaal) hoe de informatie van links naar rechts door een netwerk stroomt. We weten echter twee belangrijke zaken nog niet: hoe dit precies werkt aan het ‘begin’ en ‘eind’ van een netwerk: helemaal links waar de data wordt ingevoerd, en helemaal rechts, waar het netwerk zijn uitvoer moet geven. Dat behandelen we in deze subsectie.

Hieronder volgt nogmaals de schematische weergave van een neuraal netwerk (of ‘net’), iets verkleind.



De laag helemaal links kan men de invoerlaag noemen; hier wordt immers data ‘gevoerd’ aan het netwerk, waar het netwerk vervolgens aan mag rekenen. De vierkantjes links kunnen omschreven worden als zeer eenvoudige neuronen. Zij hebben in het model geen invoerwaarde, maar alleen een uitvoerwaarde; die is gelijk aan het getal dat men op die plek als invoer voor het netwerk had. De neuronen in de tweede laag (H1) maken met deze uitvoerwaardes weer gewogen sommen en zo gaat het verder zoals reeds beschreven.

Het is belangrijk te begrijpen dat de uitvoer van een neuron, dezelfde is naar alle neuronen in de volgende laag. De gewichten die bij de ‘lijntjes’ in de tekening horen, verschillen echter gewoonlijk wel allemaal van elkaar, waardoor de gewogen sommen van de neuronen in de volgende laag ook verschillen.

In de tekening hierboven is te zien, dat men vier getallen als invoer aan dit netwerk kan geven.

Dan kijken we nu naar de uitvoerlaag. Dit is zeker niet al te moeilijk, want er geldt: de uitvoer van de neuronen in de uitvoerlaag is gelijk aan de uitvoer van het netwerk. In de tekening hierboven is dus te zien dat dit netwerk twee uitvoerwaardes zal geven (bij vier invoerwaardes).

4.2 Praktisch nut en toepassing van een neuraal netwerk

Het algemene begrip, nodig om deze sectie te kunnen schrijven, is vergaard door het bestuderen van de bron Nielsen (2015, Hoofdstuk 1).

Nu we weten hoe de informatie door een neuraal netwerk stroomt van invoerlaag naar uitvoerlaag, willen we graag weten wat die informatie eigenlijk precies inhoudt. Dat zullen we beschrijven in deze sectie.

Het zit zo: men heeft eerst data, zoals een foto gemaakt door een beveiligingscamera. Men vraagt zich dan af, of er zich een mensengezicht op de foto bevindt of niet, en men wil dat een neuraal netwerk dit bepaalt. Wat men dan doet is ten eerste de foto omzetten in data die het neurale netwerk kan begrijpen, dat wil zeggen: getallen. Men zou bijvoorbeeld iedere pixel kunnen omzetten naar een of meerdere getallen die aangeven wat voor kleur die pixel heeft. Men heeft dan een zeer lange lijst getallen.

Zeg dat men de foto omgezet heeft naar 400 000 getallen. Er is dan een neuraal netwerk nodig dat 400 000 invoerneuronen heeft in de invoerlaag. Ieder neuron krijgt (in volgorde) als invoerwaarde één van deze 400 000 getallen. Zoals hierboven beschreven, kan onder invloed van de gewichten en biases, oftewel de parameters van het netwerk, de informatie nu van links naar rechts door het netwerk stromen, waarbij steeds een laag wordt opgeschoven.

Zodoende wordt ook de uitvoerlaag van het netwerk bereikt. We kunnen voor dit voorbeeld met de beveiligingscamera zeggen dat het netwerk één uitvoerneuron heeft. Men kan dan bijvoorbeeld als afspraak hebben bedacht: is de bijbehorende uitvoerwaarde een getal dichtbij de 1, dan bevindt zich er tenminste één menselijk gezicht op de foto. Is de uitvoerwaarde een getal dichtbij de 0, dan is dit niet zo. Het kan ook andersom zijn, dit ligt er puur aan hoe men het wil.

Samenvattend heeft men data waarover men iets wil weten. Men zet deze data dan om naar getallen en voert deze in in het netwerk. Het netwerk geeft dan een uitvoer. Deze uitvoer zet men weer om naar een vorm van data waar

men wat aan heeft. Men heeft nu informatie vergaard: dit is het nut van het goed gebruiken van een neuraal netwerk.

Natuurlijk is een neuraal netwerk niet alleen handig om te kijken of er een gezicht op een foto staat; de mogelijkheden zijn veel groter. Een neuraal netwerk zou bijvoorbeeld kunnen vaststellen in welke taal een gegeven tekst geschreven is, voorspellen of het gaat regenen of niet gegeven de data van weerstations, of (en hier gaat onze interesse het meest naar uit): **voorspellen wat de beste zet is of wie er beter staat, gegeven een stelling in Go.**

4.3 Precieze naamgeving van variabelen

Voor het vervolg van dit profielwerkstuk is het nodig dat we nu voor iedere individuele bias en gewicht in het netwerk een aparte naam hebben. In deze sectie definiëren we een precieze naamgeving.

Te beginnen met de lagen in het netwerk. We nummeren deze lagen, **te beginnen bij -1 bij de invoerlaag**. De eerste verborgen laag heeft dan nummer 0, de tweede verborgen laag heeft nummer 1, enzovorts. Dit klinkt als een erg onlogische nummering, maar in ons codebestand nummeren wij ook zo. In het profielwerkstuk nemen wij deze nummering daarom over.

Overigens begint C++ altijd met nummeren bij 0. Als je een lijst hebt van n getallen, nummert C++ deze van 0 tot en met $n - 1$.

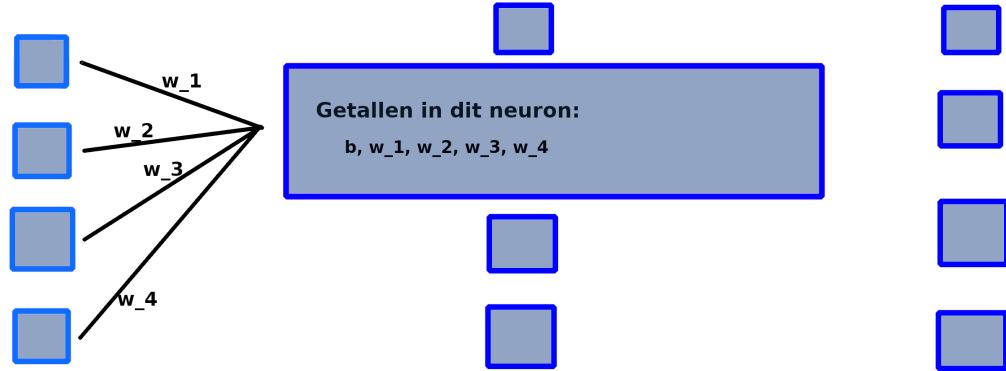
De invoerlaag is wiskundig niet zo interessant: hij bestaat uit ‘neuronen’ die eigenlijk niets doen, behalve een uitvoer geven. Daarom heeft in de code de invoerlaag geen nummer, en noemen we die in het profielwerkstuk maar laag -1 .

Vervolgens kijken we naar een willekeurige laag in het netwerk. De neuronen in deze laag nummeren we van boven naar beneden (in de tekening), beginnend bij 0, wederom omdat C++ dit ook zo doet.

Vervolgens gaan we kijken naar het specifieke neuron op een willekeurige plaats in het netwerk (maar niet in de invoerlaag).

Wiskundig gezien is dit neuron gewoon een lijst van getallen, te beginnen met de bias van dat neuron, die nummer 0 krijgt. De laag links van dit specifieke neuron bevat k neuronen. Dan geldt: na de bias volgen k getallen, genummerd 1 tot en met k , die respectievelijk de gewichten aangeven van de interneuronale verbindingen van de neuronen in de laag links van dit

specifieke neuron van boven naar beneden, naar dit neuron. Met andere woorden: in een neuron staat een lijst getallen. Het eerste van deze getallen is de bias, en daarna volgen alle gewichten van alle neuronen in de vorige laag, naar deze laag. Dit is ingewikkeld, dus daarom hebben wij ter verduidelijking een plaatje gemaakt:



In het plaatje wordt gefocust op het groot getekende neuron in het midden. Omwille van de duidelijkheid zijn alleen de relevante interneuronale verbindingen getekend; de hoop is dat de lezer onderhand zelf doorheeft waar de overige zwarte lijntjes getekend zouden moeten zijn.

Nu we deze nummering hebben bedacht om te gebruiken in dit profielwerkstuk, kunnen we ook variabelen een ‘naam’ gaan geven met behulp van deze nummering.

We gebruiken in dit profielwerkstuk de volgende ‘naamgeving’:

$b[l][n]$ = bias van neuron nummer n in laag l

$w[l][n][m]$ = gewicht van neuron m in laag $l - 1$ naar neuron n in laag l

Wij hebben de keuze gemaakt om vierkante haken te gebruiken in plaats van bijvoorbeeld subscripts, vanwege twee redenen. De eerste reden is simpelweg de leesbaarheid: grote letters zijn beter leesbaar dan letters in subscript. Ten tweede werkt ook de programmeertaal C++ met vierkante haken en leek het ons daarom wel zo consistent om die ook gewoon te gebruiken in het werkstuk zelf.

4.3.1 Naamgeving van variabelen in de code

We hebben twee plekken waar we naamgeving hebben: ten eerste in deze pdf, ten tweede in de C++-code. Hiervoor gebruiken we soms verschillende

notaties. In sommige gevallen, zoals bij de vectoren \vec{a} en \vec{s} , is de notatie in de code hetzelfde als in de pdf. Dat betekent dat iets als $a[5][2]$ in de code precies hetzelfde betekent als $a[5][2]$ in deze pdf.

We willen de lezer wijzen op een belangrijk verschil: in de pdf hebben we het over $b[l][n]$ en $w[l][n][m]$. In de code werkt dit anders: daar hebben we de driedimensionale vector `net` die zowel de biasen \vec{b} als de gewichten \vec{w} bevat.

Dit gaat via de volgende regels:

$$b[l][n] = \text{net}[l][n][0] \quad (4.4)$$

$$w[l][n][m] = \text{net}[l][n][m + 1] \quad (4.5)$$

4.4 Een compleet overzicht van variabelen

In deze sectie geven we een compleet overzicht van alle variabelen met hun namen, die we in dit profielwerkstuk gebruiken of nog zullen gebruiken.

$b[l][n]$ = bias van neuron nummer n in laag l

C = kostenfunctie

$s[l][n]$ = gewogen som van de invoerwaarden met bias erbij opgeteld, in neuron n in laag l

$w[l][n][m]$ = gewicht van neuron m in laag $l - 1$ naar neuron n in laag l

$a[l][n] = s[l][n]$ = activatiefunctie met s als invoer, van neuron n in laag l = de uitvoerwaarde van neuron n in laag l

$u[n]$ = de uitvoerwaarde van neuron n in de uitvoerlaag

$w[n]$ = de gewenste uitvoerwaarde van neuron n in de uitvoerlaag (de uitvoerwaarde waar het netwerk naartoe moet streven)

4.5 De informatiestroom door het netwerk in C++

We zullen de lezer eens verblijden met wat C++-code. Hieronder is de functie afgedrukt, die verantwoordelijk is voor de informatiestroom door het neurale netwerk van links naar rechts. Deze functie maakt, bij een gegeven training input (vd `input_data`, helemaal bovenin), de vectoren \vec{s} en \vec{a} .

```
// Sets the vectors weighted sums (s) and activations (a).
void Athena::get_weighted_sums_and_activations(vd input_data, vvd &s, vvd &a) {
    s.clear();
    a.clear();
```

```

// the s vector is a vector with all the weighted sums (plus bias) of the
// neurons. The big vector contains all layers except the input layer.
// Therefore the first hidden layer gets index 0. All the subvectors contain
// the weighted sums (+biases) of those neurons in their layers.

// The vector a is the same as the vector s which is defined right above.
// But this vector has had the activation function applied to all its
// values.

for (int l = 0; l < net.size(); l++) { // go through all layers
    s.push_back(std::vector<double>{});
    for (int n = 0; n < net[l].size();
         n++) { // loop through all neurons in layer
        double weighted_sum_plus_bias = 0;
        if (l > 0) {
            for (int d = 0; d < a[l - 1].size(); d++) {
                weighted_sum_plus_bias += a[l - 1][d] * net[l][n][d + 1];
                if (std::isnan(a[l - 1][d])) {
                    std::cout << "nan a[l-1][d]\n";
                    exit(0);
                }
                if (std::isnan(net[l][n][d + 1])) {
                    std::cout << "nan net[l][n][d+1]\n";
                    exit(0);
                }
            }
        }
    }
} else if (l == 0) {
    for (int d = 0; d < input_data.size(); d++) {
        weighted_sum_plus_bias += input_data[d] * net[l][n][d + 1];
        if (std::isnan(input_data[d])) {
            std::cout << "nan input data\n";
            exit(0);
        }
        if (std::isnan(net[l][n][d + 1])) {
            std::cout << "nan weight\n";
            exit(0);
        }
    }
}
weighted_sum_plus_bias += net[l][n][0]; // add the bias

if (std::isnan(weighted_sum_plus_bias)) {

```

```

        std::cout << "nan weighted sum plus bias\n";
        exit(0);
    }
    s[1].push_back(weighted_sum_plus_bias);
}
a.push_back(s[1]);
activation_function(a[1]);
}

return;
}

```

De bovenstaande functie berekent dus onder andere de gewogensommen-vector \vec{s} . Vrij ver onderaan zie je het volgende staan:

```
a.push_back(s[1]);
activation_function(a[1]);
```

Op deze twee regels gaan we iets dieper in. $\vec{s}[l]$, de vector van gewogen sommen inclusief biasen van de neuronen van laag l , is op dat moment net berekend. $\vec{s}[l]$ bestaat uit de componenten $s[l][0]$, $s[l][1]$, et cetera. Met `a.push_back(s[1]);`, voegen we aan de tweedimensionale \vec{a} -vector aan het einde van het rijtje vectoren dat er al is, de vector $\vec{s}[l]$ toe.

Zodoende zijn $\vec{a}[l]$ en $\vec{s}[l]$ eerst gelijk aan elkaar. Maar dat willen we niet; we moeten nog de activatiefunctie toepassen op alle waarden van de vector $\vec{a}[l]$. Dat doen we met de coderegel `activation_function(a[1]);`.

Die coderegel roept vervolgens een andere door ons gemaakte functie aan:

```
void Athena::activation_function(
    std::vector<double> &input) { // for an entire vector
    for (int i = 0; i < input.size(); i++) {
        input[i] = activation_function(input[i]);
    }
    return;
}
```

Deze korte functie krijgt als invoer een vector met getallen, een `std::vector<double>`. Het enige dat deze functie doet, is ervoor zorgen dat op elk afzonderlijke element van deze invoervector, de activatiefunctie wordt toegepast. Dat gebeurt met de regel `input[i] = activation_function(input[i]);`. Deze regel roept op zijn beurt weer een `activation_function` aan. Deze krijgt echter slechts een getal als invoer, en niet een vector met getallen. Door

de regel `input[i] = activation_function(input[i]);` wordt de volgende functie aangeroepen:

```
double Athena::activation_function(double x) { // for a single value
    if (std::isnan(x)) {
        std::cout << "nan input for activation_function\n";
        exit(0);
    }
    if (which_activation_function == "sigmoid") {
        return 1.0 / (1.0 + exp(-x));
    } else if (which_activation_function == "relu") {
        if (x < 0) {
            return 0;
        } else {
            return x;
        }
    } else if (which_activation_function == "leakyrelu") {
        if (x < 0) {
            return LEAKYRELU_PARAMETER * x;
        } else {
            return x;
        }
    }

    std::cout << "ERROR in line number " << __LINE__ << ".\n";
    return 0.0;
}
```

Deze functie krijgt als invoer een getal, en doorloopt vervolgens een if-else-schema om de activatiewaarde te *returnen*.

Zo zien we dat de coderegels

```
a.push_back(s[1]);
activation_function(a[1]);
```

veel doen, ook al zijn het maar twee regels.

4.6 Het trainen van een neuraal netwerk

De mogelijkheden van een neuraal netwerk zijn dus eindeloos. Echter hebben we één cruciaal principe nog niet besproken: de vraag hóé je aan een neuraal netwerk komt dat werkt. Nu zet het neuraal netwerk alleen een bepaalde

invoer om in een bepaalde uitvoer. We weten helemaal niet of deze uitvoer goed is. Hoe zorg je ervoor dat het neuraal netwerk wordt aangepast dat de uitvoer beter is?

Dit gebeurt aan de hand van negatieve gradiënt van de kostenfunctie. De kostenfunctie, die we C noemen, is een functie die het verschil tussen de wenselijke uitvoer, die we \vec{w} noemen, en de daadwerkelijke uitvoer, die we \vec{u} noemen, uitrekent. Wanneer we een neuraal netwerk willen verbeteren, dan willen we dat de kostenfunctie zo klein mogelijk wordt. De daadwerkelijke uitvoer is dan bijna gelijk aan de gewenste uitvoer. Zoals al eerder is uitgelegd, zegt de negatieve gradiënt welke kant de functie moet opgaan om het snelst te dalen om zo uiteindelijk (na herhaaldelijk bewegen in de richting van de gradiënt) bij een minimum te komen. In een neuraal netwerk gebeurt dit door de partiële afgeleiden te nemen van de kostenfunctie naar de gewichten en de biasen. Het bereken van de gradiënt heet backpropagation. In dit profielwerkstuk hebben wij zelf de backpropagation-formules bewezen, zie het betreffende hoofdstuk.

Laten we als voorbeeld een neuraal netwerk nemen. Alle gewichten en biasen kloppen nog niet, dus de uitvoer verschilt erg veel van de gewenste uitvoer. Eerst kijken we hoeveel elke neuron aangepast moet worden om een juiste uitvoer te geven. Dit kan direct door de biasen aan te passen. Het kan ook via de gewichten. We kijken dan naar de vorige laag van het neuraal netwerk. We kijken dan hoeveel elk gewicht aangepast moet worden om de juiste uitvoer te geven. Hoeveel een gewicht aangepast moet worden, hangt af van de activatiewaarde van de neuron van de vorige laag. Als deze namelijk erg klein is, heeft het weinig invloed. Bij grotere activatiewaarden wordt de kostenfunctie meer aangepast. Als we weten hoeveel de gewichten en biasen moeten aangepast worden van de laag voor de uitvoerlaag, kunnen we weer kijken hoeveel invloed elk gewicht heeft op de uiteindelijke kostenfunctie. Dan gaan we weer een laag terug en doen we precies hetzelfde. Dit is het principe van backpropagation. Vanuit de uitvoerlaag wordt teruggewerkt naar de invoerlaag, terwijl de gewichten en biasen worden aangepast. Dit aanpassen gebeurt door middel van de updateregel die verder naar onder wordt uitgelegd.

Om voor elke trainingsinvoer de partiële afgeleiden uit te rekenen en dan de gradiënt te middelen is zeer inefficiënt, want als je het gemiddelde neemt over 100000 gradiënten, zal dat ongeveer wel hetzelfde zijn als de gemiddelde gradiënt over 100 training inputs. Als je al je training inputs opdeelt in mini batches, kun je veel vaker je biasen en gewichten updaten naar de gemiddelde gradiënt over je mini batch. Je kunt dan wel 1000 keer je gewichten en biasen updaten naar een vrij accurate gemiddelde gradiënt over 100 training inputs, en dat is veel en veel efficiënter dan maar een keer je gewichten en biasen

updaten naar een slechts iets preciezere gemiddelde gradiënt over 100000 training inputs. Dus het aantal trainingsvoorbeelden wordt willekeurig verdeeld en opgedeeld in mini-batches. Je neemt dan steeds de gemiddelde gradiënt over alle trainingsvoorbeelden van een mini-batch en dan update je de biasen en gewichten, en dat doe je voor elke mini-batch. Deze methode heet **stochastic gradient descent**, oftewel stochastische gradiëntdaling.

Als je alle mini-batches één keer hebt gehad, en dus je totale training data één keer bent doorlopen, dan heb je een **epoch** doorlopen. En dat laatste kun je natuurlijk ook weer meerdere keren doen.

De learning rate is een getal dat aangeeft hoe hard je beweegt in de richting van de negatieve gradiënt. De updateregel, bij een gegeven (gemiddelde) gradiënt ∇C , luidt (in vectornotatie) (onthoud dat `net` zowel de biasen als de gewichten bevat):

$$\overrightarrow{\text{net}} \leftarrow \overrightarrow{\text{net}} - \eta \nabla C \quad (4.6)$$

Hierbij is η , ofwel eta, de learning rate. C zien we hier als functie van alle componenten van $\overrightarrow{\text{net}}$, en dus is de gradiënt van C een vector met partiële afgeleiden van C naar de componenten van $\overrightarrow{\text{net}}$, oftewel naar alle biasen en gewichten.

De bovenstaande updateregel gebruiken wij altijd, ook bij stochastische gradiëntdaling. Dit is ook te zien in de code hieronder.

We weten dus dat ∇C bestaat uit $\frac{\partial C}{\partial b}$ en $\frac{\partial C}{\partial w}$ voor alle biasen en gewichten. De vraag is nu dus: hoe vinden we $\frac{\partial C}{\partial b}$ en $\frac{\partial C}{\partial w}$ voor alle biasen en gewichten? Deze vraag staat centraal in het hoofdstuk waarin wij backpropagation bewijzen.

Voor het schrijven van deze sectie, vanaf het begin van de sectie tot aan hier, gebruikten wij de volgende bronnen: McGonagle et al. (z.d.), Nielsen (2015, Hoofdstukken 1–3) en Sanderson (2017).

Uiteraard hebben we dit algoritme ook in C++ geïmplementeerd:

```
// This is the 'learning' function, in which we update the network parameters
// (biases and weights)
void Athena::update_biases_and_weights(vvd input_datas,
                                         vvd desired_output_datas,
                                         double learning_rate) {
    double avgcost_before = 0.0;
    double avgcost_after = 0.0;
    if (do_enable_alrcs) { // TODO: hierover nog wat zeggen in pws?
        avgcost_before = cost_value(input_datas, desired_output_datas);
    }
}
```

```

network gradient_times_negative_learning_rate =
    multiplyNetworkWithConstantValue(
        gradient(input_datas, desired_output_datas), -learning_rate);
// the learning rate is NEGATIVE here because we want to SUBTRACT
// (learning_rate*gradient) from our net vector which contains the biases
// and weights.

net = networkSum(net, gradient_times_negative_learning_rate);

if (do_enable_alrcs) {
    avgcost_after = cost_value(input_datas, desired_output_datas);
}

// the following few lines of code are exclusively usable if we are using
// athena in combination with the rest of the program that plays go !
if (avgcost_before < avgcost_after && do_enable_alrcs) {
    NETWORK_LEARNING_RATE *= 0.2;
} else if (do_enable_alrcs) {
    NETWORK_LEARNING_RATE *= 1.0015;
}

return;
}

```

Hierboven zie je de functie `Athena::update_biases_and_weights`. Deze functie roept weer wat andere functies aan, zoals `multiplyNetworkWithConstantValue` en `networkSum`, maar daar gaan we nu even niet verder op in. De lezer kan, indien gewenst, deze code zelf opzoeken in de codebestanden (in dit geval in `athena.cpp`) die zijn meegeleverd met dit profielwerkstuk.

De functie `Athena::update_biases_and_weights` wordt aangeroepen door de volgende functie:

```

void Athena::stochastic_gradient_descent(
    std::vector<std::vector<double>> input_datas,
    std::vector<std::vector<double>> desired_output_datas,
    double learning_rate, // IMPORTANT NOTE: this parameter is replaced by the
// global NETWORK_LEARNING_RATE when ALRCS is enabled.
    int mini_batch_size) {
    std::deque<int> order;
    for (int i = 0; i < input_datas.size(); i++) {
        order.push_back(i);
    }
}

```

```

    std::random_device r_d;
    std::mt19937 g(r_d());
    std::shuffle(order.begin(), order.end(), g);

    std::vector<std::vector<double>> input_datas_mini_batch;
    std::vector<std::vector<double>> desired_output_datas_mini_batch;

    for (int i = 0; i < input_datas.size(); i++) {
        input_datas_mini_batch.push_back(input_datas[order[i]]);
        desired_output_datas_mini_batch.push_back(
            desired_output_datas[order[i]]);

        if ((i + 1) % mini_batch_size == 0 || i + 1 == input_datas.size()) {
            if (do_enable_alrcs) {
                update_biases_and_weights(input_datas_mini_batch,
                                           desired_output_datas_mini_batch,
                                           NETWORK_LEARNING_RATE);
            } else {
                update_biases_and_weights(input_datas_mini_batch,
                                           desired_output_datas_mini_batch,
                                           learning_rate);
            }
        }

        input_datas_mini_batch.clear();
        desired_output_datas_mini_batch.clear();
    }
}
}

```

Hierboven zie je onze implementatie van stochastische gradiëntdaling, of stochastic gradient descent.

Vervolgens willen we de lezer wijzen op een klein zelfbedacht amateurisch algoritmetje: Automatic Learning Rate Correction System, oftewel ALRCS. (Die naam is ook zelfbedacht.)

Als dit is ingeschakeld, past het automatisch de learning rate aan waar dat nodig is. Als de kost opeens door trainen hoger is geworden, wat absoluut niet zou moeten, is de learning rate waarschijnlijk te groot geweest en ben je te ver doorgeshoten in je beweging in de richting van de negatieve gradiënt¹. ALRCS zorgt er in dat geval voor dat de learning rate flink wordt verlaagd.

¹De gradiënt bestaat uit afgeleiden en afgeleiden gaan natuurlijk over kleine veranderingen op kleine schaal. De gradiënt wijst dan wel in de richting van de snelste stijging, maar dat is op kleine schaal. Je kunt natuurlijk niet ontzettend ver gaan bewegen in de richting van dezelfde gradiënt, want als je te ver weg bent zijn de afgeleiden misschien wel

Als de kost echter, zoals het hoort, kleiner is geworden na het trainen, dan is dat mogelijk een indicatie dat de learning rate wel iets hoger mag (of wellicht zou mogen). Daarom wordt, in het geval van een verlaging van de kost, de learning rate net een heel klein beetje groter gemaakt door ALRCS.

De bedoeling van dit ALRCS is dat het op lange termijn zorgt voor een (enigszins) stabiele learning rate, waardoor het trainen efficiënt verloopt zonder dat we erover na hoeven te denken.

4.7 Q-learning

Voor het schrijven van deze sectie hebben we de bron van Shyalika (2019) en het volgende Wikipedia-artikel als aanvulling en bevestiging gebruikt: (“Q-Learning”, 2021).

Een ding hebben we nog niet besproken: wat geven we in de context van Go als invoer aan het neurale netwerk en wat moet het netwerk uitvoeren? Als invoer hebben wij bedacht dat we de bordstelling aan het neurale netwerk geven, gecodeerd in getallen die het netwerk begrijpt. Dit is dan de stelling nadat het netwerk een zet heeft gedaan. Wat we het netwerk als uitvoer willen laten geven, is echter een moeilijkere kwestie.

We willen dat het netwerk bij een ingevoerde stelling kan uitvoeren hoe goed het netwerk staat. Eerst hadden we bedacht dat we gewoon willen dat het netwerk 1 uitvoert als het denkt compleet gewonnen te staan, en 0 als het zeker denkt te weten te gaan verliezen. We lieten het netwerk dan trainen door het heel vaak te laten spelen tegen een speler die altijd een willekeurige zet speelt. Als het netwerk een potje had gewonnen, gaven we als desired_output een 1, als het netwerk had verloren, gaven we als desired_output om mee te gaan trainen een 0.

Deze techniek werkte al aardig goed: op 26 oktober 2021 hadden we een netwerk dat aardig goede resultaten behaalde met de hierboven beschreven methode.

Toch hebben wij ervoor gekozen om ook een iets professioneler algoritme te implementeren dat bepaalt met welke desired_output het netwerk moet gaan trainen. Deze methode heet **Q-learning**.

De update-formule bij Q-learning luidt als volgt:

$$Q \leftarrow Q + \alpha(R + \gamma \max Q_{\text{toekomst}} - Q) \quad (4.7)$$

helemaal anders geworden en moet je weer opnieuw de gradiënt bepalen, wil je nog verder stijgen. Het is dan logisch dat eenzelfde verhaal geldt bij dalen, gradient descent. Vandaar de logica van ALRCS.

Hierin is Q een getal dat bij een gegeven staat en actie (oftewel: een bordpositie en een gespeelde zet) de kwaliteit aangeeft. Q wordt steeds vernieuwd: daarom hebben we een pijl naar links neergezet en geen $=$ -teken; het is een algoritmische updateregel. α is de learning rate. Maar let goed op: we gebruiken in dit profielwerkstuk ook al een andere learning rate. Daarom noemen we deze learning rate vanaf nu geen learning rate meer, maar alleen alfa of α . Als we het dus ergens anders in dit profielwerkstuk over een learning rate hebben, bedoelen we nooit alfa!

Verder is R de reward of beloning die het algoritme krijgt uitgedeeld bij de gegeven actie in de gegeven staat (de gegeven zet in een bepaalde bordstelling).

Gamma, γ , is de ‘discount rate’. Dit geeft aan hoeveel waarde hij hecht aan toekomstige rewards. Als je het algoritme dus heel ‘hebzuchtig’ en kortetermijn-denkend wilt maken, moet je gamma heel laag zetten, zodat het algoritme vooral kijkt naar wat hij nu, en binnen korte tijd, kan bereiken.

Tenslotte wordt met $\max Q_{\text{toekomst}}$ de verwachte maximale reward bedoeld voor de toekomst, een tijdstap verder.

Als het Go-spel voorbij is en de laatste zet gespeeld is, geef je die laatste zet als Q -value gewoon alleen de reward. Bij ons is dat 1 als het netwerk het potje gewonnen heeft en anders 0.

Iets wat achteraf een beetje jammer gevonden zou kunnen worden, is dat wij Q-learning niet volledig hebben benut. Wij kennen het algoritme namelijk alleen een reward toe aan het einde van het spel, als zwart of wit gewonnen heeft, en tussentijds (als het spel nog bezig is) geven wij het algoritme nooit een reward, terwijl dat juist zo’n mooie mogelijkheid is die Q-learning ons geeft. Wellicht is dit iets dat we later nog kunnen toevoegen.

4.7.1 Q-learning combineren met een neuraal netwerk

Als we weer even kijken naar het Q-learning-algoritme, zien we dat je dus steeds bij staat-actie-paren een Q -waarde moet bepalen. Nu is Go echter een extreem complex spelletje met ontzettend veel bordmogelijkheden, dat valt zelf wel na te gaan. Het is dus niet te doen, zelfs niet met een computer, om bij alle bordmogelijkheden en zetten, een Q -waarde te berekenen.

Dit is waar het neurale netwerk nuttig wordt: we gebruiken gewoon bij alle stellingen een neurale netwerk om de Q -waarde te benaderen! Dan lukt het opeens wel, en heb je geen oneindig lange tijd nodig om miljarden en miljarden Q -waarden te berekenen voordat je er iets mee kunt. Dat neurale netwerk moet dan uiteraard wel getraind worden, maar dat kan bij een 5x5-Go-bord binnen acceptabele tijd, blijkt uit onze ervaring.

4.7.2 Q-learning in code

Om Q-learning goed te implementeren in het specifieke geval van onze Go-applicatie, was wat creativiteit nodig, maar het is wel gelukt. Hieronder geven we de functie weer die het programma één Go-potje laat spelen. Hierbij wordt ook Q-learning toegepast, indien dat is ingeschakeld. Uit het oorspronkelijke codefragment hebben we wel wat minder relevante dingen, zoals uitgeschakelde coderegels, weggeknippt, omdat het anders te lang werd om hier weer te geven.

```
// plays a game, but DOESN'T call Athena functions or so
bool fully_play_one_game(bool do_not_write_anything = false) {
    reset_all_global_variables();
    std::vector<std::tuple<int, int>>
        moves_done; // all moves which have been played. By either black or
                    // white.
    std::vector<double> q;
    while (!game_ended && !quit) {
        auto move_data = player_choice();
        moves_done.push_back(std::make_tuple(
            std::get<0>(move_data),
            std::get<1>(move_data))); // you now get all moves which have been
                                      // played. By either black or white.
        q.push_back(std::get<2>(move_data));
    }
    bool black_wins =
        (double)count_area_black() > (double)count_area_white() + komi;

    // fix Q-values in final states
    if (moves_done.size() % 2 == 1) {
        // last move was from black
        q.back() = (black_wins ? 1.0 : 0.0);
    } else {
        // last move was from white
        q[q.size() - 1 - 1] =
            (black_wins ? 1.0 : 0.0); // change q for black's last move
    }
    if (do_not_write_anything) {
        std::cout << ((black_wins != komi < 0) ? "\n\nZwart heeft gewonnen.\n"
                                                : "\n\nWit heeft gewonnen.\n");
        // if komi is negative, present to the user the result which the user
        // sees: invert colors.
    }
    return black_wins;
}
```

```

}

// now, we want to save the game and it's result in a file.
std::string game_manager_file = std::to_string(N) + "-" +
                                std::to_string(komi) +
                                "game_count_manager.txt";
std::ifstream file(game_manager_file, std::ios::in);
if (!file.is_open()) {
    std::cout << "ERROR in line " << __LINE__ << __FILE__ << ".\n";
    std::cout << game_manager_file << "\n";
    exit(0);
}
int number_of_games_played_totally;
file >> number_of_games_played_totally;
file.close();
std::ofstream file2(game_manager_file, std::ios::out | std::ios::trunc);
if (!file2.is_open()) {
    std::cout << "ERROR in line " << __LINE__ << __FILE__ << ".\n";
    exit(0);
}
file2 << ++number_of_games_played_totally;
file2.close();

// now we know which file to write to.
std::string curr_game_file =
    std::to_string(N) + "-" + std::to_string(komi) + "game" +
    std::to_string(number_of_games_played_totally) + ".txt";
std::ofstream gamefile(curr_game_file, std::ios::trunc);
if (!gamefile.is_open()) {
    std::cout << "ERROR in line " << __LINE__ << __FILE__ << ".\n";
    exit(0);
}
for (int i = 0; i < (int)former_positions.size(); i += 2) {
    for (int y = 0; y < N; y++) {
        for (int x = 0; x < N; x++) {
            gamefile << former_positions[i][y][x] << " ";
        }
    }
    if (!Q_LEARNING) {
        gamefile << (black_wins ? 1 : 0); // who eventually won the game
    } else {
        double new_q_value;
        if (q.size() >= i + 3) {
            new_q_value =

```

```

        q[i] + ALFA * (DISCOUNT * q[i + 2] - q[i]);
    } else {
        new_q_value = black_wins;
    }
    gamefile << new_q_value;
}
gamefile << "\n";
}
gamefile.close();
return black_wins;
}

```

In de bovenstaande code zijn Q-learning-elementen duidelijk herkenbaar.

4.8 Xavier initialisatie

Voor deze sectie hebben we de bron van Dellinger (2019) gebruikt.

Hoe initialiseer je een netwerk? Gebruik je grote willekeurige groottes voor de gewichten of begin je juist met kleine willekeurige groottes? Of moet je juist een andere methode gebruiken? Het lijkt op het eerste gezicht niet erg belangrijk, maar dit heeft grote invloed op het uiteindelijke netwerk. Stel we kiezen willekeurige waardes voor de gewichten die van dezelfde grootte zijn als de invoerlaag. Het blijkt dan dat de uitvoerlaag van het netwerk veel te veel uitvoer geeft. Wat als we nu de gewichten met een bepaalde factor naar beneden schalen? Het blijkt dat dan de uitvoerlaag bijna geen uitvoer meer geeft. Wanneer de gewichten zo worden geïnitialiseerd dat ze of te klein zijn of te groot, dan zal de uitvoerlaag dus geen uitvoer geven of juist te veel. De backpropagatie zal dan niet meer efficiënt verlopen, omdat de negatieve gradiënt te klein of te groot. Het zal dus langer duren om een lokaal minimum te bereiken. Bij netwerken met een grote hoeveelheid lagen gebeurt dit veel eerder. Een laag is natuurlijk verbonden is met de volgende laag en die laag is ook weer verbonden. Dus wanneer bijvoorbeeld de gewichten gemiddeld te klein zijn zal iedere laag weer een beetje kleiner worden, totdat er bijna geen uitvoer is. Bij een grote hoeveelheid valt dit dus meer op.

We moeten dus een manier vinden dat de uitvoerlaag niet te groot en te klein wordt. De onderzoekers Xavier Glorot en Yoshua Bengio vonden een oplossing voor functies zoals de sigmoïdefunctie. De gewichten krijgen een willekeurige waarde tussen

$$\pm \frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}} \quad (4.8)$$

Hierbij is n_l het aantal verbindingen die met de laag verbonden zijn (aan de linkerkant, ze gaan de laag in). n_{l+1} is het aantal verbindingen die uit laag l gaan. Het blijkt dat de gradiënten niet te groot worden, maar ook niet te klein worden. De uitvoerlaag geeft dus niet meer bijna geen uitvoer en ook niet veel te veel uitvoer. Voor ons neuraal netwerk hebben wij deze initialisatietechniek ook gebruikt; deze is te vinden in de als bijlage opgenomen codebestanden.

4.9 Het neurale netwerk speelt altijd met de zwarte stenen

Hier lijkt het ons goed, het volgende nog te vermelden. **In de code speelt het (trainende) neurale netwerk altijd met de zwarte stenen**, parodoxaal genoeg zelfs als het met de witte stenen speelt. Dit hebben wij gedaan om efficiënt te kunnen programmeren, en niet sommige dingen twee keer te hoeven programmeren, voor zwart en voor wit.

Dit werkt als volgt: het netwerk speelt in de kern van de computercode gewoon altijd zwart. Maar voor de gebruiker lijkt het alsof het netwerk zowel zwart als wit kan spelen. Lijkt het voor de gebruiker alsof het netwerk zwart speelt, dan doet het dat in de kern van de code ook en is er niets geks aan de hand.

Als het voor de gebruiker lijkt alsof het netwerk wit speelt, dan doet het dat eigenlijk in de kern van de code niet. Het netwerk speelt toch gewoon zwart, maar voor de gebruiker worden de kleuren wit en zwart omgekeerd op het scherm gezet. Ook worden in dit geval de winstmeldingen voor de gebruiker omgedraaid: ‘Wit heeft gewonnen!’. Ook krijgt de witspeler in deze situatie een extra zet, nog voordat de zwarte speler zijn ‘eerste’ zet speelt. Maar deze witspeler lijkt zwart en dus klopt alles toch.

Er gebeurt in de situatie dat het moet lijken alsof het netwerk wit speelt, nog iets bijzonders: **de komi-waarde is intern negatief**. De (negatieve) komi-waarde wordt namelijk in de code opgeteld bij de (echte) witspeler, en dat is in deze situatie equivalent aan het optellen van de gepositiveerde komiwaarde bij de (schijnbare) witspeler, hetgeen precies is wat je wil.

Dit is allemaal inderdaad erg ingewikkeld. Toch is het handig. Nu we dit eenmaal hebben bedacht, hoeven we in het vervolg niet vaak meer na te denken over het verschil tussen zwart en wit. Als we nieuwe dingen programmeren, kunnen we in ons hoofd houden dat het trainende netwerk altijd zwart speelt, en dan hoeven we daar niet moeilijk over na te denken. Het

enige dat we ons nog hoeven te bedenken is het volgende: **als we willen dat het trainende netwerk wit speelt, moeten we een negatieve komi-waarde invullen.**

Dit heeft wel de consequentie dat we voor zwart en wit twee verschillende netwerken hebben, vanwege de verschillende komi-waarden.

4.10 Overfitting

Toen we uiteindelijk ons programma aan het trainen waren, gebeurde er wel iets heel vreemds: na een trainingsepoch had het ALRCS-algoritme de learning rate niet naar beneden bijgesteld, wat een indicatie was dat de kosten tijdens het trainen gewoon, zoals wenselijk, waren gedaald. Maar: ons netwerk functioneerde opeens slechter; het verloor meer partijen. Deze twee waarnemingen lijken totaal met elkaar in tegenspraak. Wat wij denken dat hiervan de oorzaak is, is *overfitting*. Overfitting is wat er gebeurt als je netwerk goede prestaties levert bij de data waarmee het traint, maar slechte prestaties op data waarmee het niet traint (Carremans, 2018). Dit is natuurlijk een zeer ongewenste situatie, want je hebt niet zoveel aan een netwerk, als het niet kan generaliseren over willekeurige nieuwe data. Volgens onze eigen interpretatie zou in ons geval overfitting inhouden: het programma speelt een aantal partijen en traint daarop, door middel van een combinatie van stochastische gradiëntdaling en Q-learning. Dit netwerk wordt vervolgens gebruikt om meer partijen te spelen. Maar die partijen zijn anders dan de vorige; je kunt dit zien als nieuwe data. Door overfitting presteert het netwerk op nieuwe data niet goed, en verliest het dus partijen, ook al heeft je netwerk zich goed bekwaamd in de data van de vorige set partijen en ook al is daar de learning rate gedaald.

Dit is onwenselijk! Maar we wisten nog steeds niet of overfitting wel de werkelijke oorzaak was van het hierboven beschreven probleem. Om te controleren of wij inderdaad te maken hadden met overfitting, hebben we het volgende bedacht.

We hebben de data gescheiden in trainingdata (logischerwijs de data waarmee getraind wordt) en testdata (data waarmee niet getraind wordt). Er wordt dus een aantal training-partijen gespeeld, maar ook een aantal test-partijen. De test-data-partijen worden alleen gebruikt om te kijken of ook op de testdata de kosten lager worden. Hierbij hoort onder andere het volgende blokje code:

```
double training_data_cost_first = athena.cost_value(athena_inp, athena_desired_outp);
double test_data_cost_first = athena.cost_value(athena_inp_test, athena_desired_outp_test);
```

```

//now perform the actual training:
for(int e = 0; e < number_of_epochs; e++) {
    athena.stochastic_gradient_descent(athena_inp,
        athena_desired_outp, learning_rate, mini_batch_size);
}

double training_data_cost_after = athena.cost_value(athena_inp, athena_desired_outp);
double test_data_cost_after = athena.cost_value(athena_inp_test, athena_desired_outp_test);

std::cout << "Training data cost before training: " << training_data_cost_first
    << "\nTraining data cost after training: " << training_data_cost_after << "\n";
std::cout << "Test data cost before training: " << test_data_cost_first
    << "\nTest data cost after training: " << test_data_cost_after << "\n\n";

```

Toen alles geïmplementeerd was, kregen we bijvoorbeeld de volgende uitvoer:

```

THREAD 21 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 31 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 10 : Won 9 out of 10 games and 5 out of 5 games.
THREAD 20 : Won 10 out of 10 games and 5 out of 5 games.

Games have been played. Now training...
Training data cost before training: 0.361637
Training data cost after training: 0.29287
Test data cost before training: 0.341431
Test data cost after training: 0.302358

```

```

Training finished.
Training finished. The current learning rate is 0.0408733

```

```

THREAD 20 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 4 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 5 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 3 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 13 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 9 : Won 10 out of 10 games and 5 out of 5 games.

Games have been played. Now training...
Training data cost before training: 0.0560635
Training data cost after training: 0.0540381
Test data cost before training: 0.0284877
Test data cost after training: 0.0295837

```

```

Training finished.
Training finished. The current learning rate is 0.0404707

```

In het bovenste plaatje zie je dat de kosten bij zowel de trainingdata als de testdata (!) kleiner zijn geworden. Het netwerk is gedurende het trainen met de trainingdata dus ook beter geworden in het benaderen van de test data, zonder dat het daarmee getraind had! Dit is niet wat je bij overfitting zou verwachten. Wij begonnen ons dus af te vragen of we wel te maken hadden met overfitting. Overigens zie je bij beide plaatjes ook dat er ontzettend veel partijen worden gewonnen; het leek dus wel alsof ons hele probleem weg was.

Het kwam ook wel eens voor, zoals op het onderste plaatje, dat de trainingkosten lager werden maar de kosten bij de testdata juist hoger. Dit zou dus toch wel kunnen duiden op overfitting. Maar omdat het programma onderhand zoveel partijen won, maakten we ons daar niet meer zo druk over.

Maar hoe kan het dan dat we eerst het probleem hadden dat het programma slechter werd maar toch de kosten (vermoedelijk) ook omlaag gingen? Dat weten we dus niet zeker. Een alternatieve hypothese die we hebben is de volgende.

Wij dachten: als de kosten omlaag gaan, wordt het netwerk beter. Maar de Q-waardes die het netwerk moet benaderen, veranderen ook, onder invloed van de Q-learning rate, alfa. Misschien, dachten wij, hebben we deze alfa wel veel te heftig ingesteld, en veranderen de Q-waardes zo snel dat ‘het netwerk het niet bij kan houden’ en de hele zaak niet meer stabiel is. We hebben de alfa dus omlaag gezet, uiteindelijk naar 0,015 op het moment van schrijven. Het kan dus ook zijn dat dit ons oorspronkelijke probleem heeft opgelost. Het blijft allemaal een beetje gissen hoe het kan, maar het werkt.

4.11 Update: overfitting?

Na het bovenstaande stuk te hebben geschreven, keken wij terug op de terminal en zagen het volgende:

```

THREAD 2 : Won 5 out of 10 games and 5 out of 5 games.
THREAD 23 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 5 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 21 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 26 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 16 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 3 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 18 : Won 10 out of 10 games and 4 out of 5 games.

Games have been played. Now training...
Training data cost before training: 0.147012
Training data cost after training: 0.196401
Test data cost before training: 0.131644
Test data cost after training: 0.202961

Training finished.
Training finished. The current learning rate is 0.0583466
THREAD 3 : Won 6 out of 10 games and 1 out of 5 games.
THREAD 16 : Won 5 out of 10 games and 2 out of 5 games.
THREAD 20 : Won 8 out of 10 games and 2 out of 5 games.
THREAD 30 : Won 5 out of 10 games and 4 out of 5 games.
THREAD 24 : Won 6 out of 10 games and 5 out of 5 games.
THREAD 19 : Won 7 out of 10 games and 4 out of 5 games.
THREAD 4 : Won 7 out of 10 games and 1 out of 5 games.

```

Dit vonden we vrij vreemd. Niet alleen de kosten op de testdata zijn *gestegen*, maar ook de kosten op de trainingdata! Je ziet hier ons oorspronkelijke probleem weer terug als je kijkt naar het aantal gewonnen partijen door het netwerk: eerst wint hij (bijna) alles en na trainen wint hij minder partijen. Wat helemaal raar is, is dat de vorige learning rate (die niet op het plaatje zichtbaar is), gelijk was aan 0,0576507. Na trainen werd dat (zoals in het plaatje) 0,0583466. ALRCS heeft dus de learning rate verhoogd, nota bene terwijl de kosten zijn gestegen. De bedoeling van ALRCS is echter juist, zoals eerder beschreven, om in dergelijke gevallen als de kosten stijgen, de learning rate naar beneden bij te schroeven. We hadden er weer een nieuw vraagstuk bij.

Maar in ons hoofd hadden we ook al weer snel een antwoord bedacht: ALRCS gebruikt de kosten van de data *in de mini-batch* die de gradiëntdaling stochastisch maakt. Echter worden in de terminal geprint de kosten over de gehele dataset, en dus niet over een willekeurige mini-batch.

We vermoeden dat ons programma een te kleine mini-batchgrootte gebruikte: in dat geval zou de mini-batch niet representatief zijn voor de gehele dataset en de bijbehorende mini-batchgradiënt dus ook niet. De gewichten en biasen in de richting van de negatieve gradiënt bewegen, heeft dan uiteraard ook weinig zin. Toch kon het ALRCS-algoritme hebben gedacht dat alles goed ging, omdat de kosten op de te kleine mini-batch waarschijnlijk wel daalden.

Om het trainen voort te zetten, hebben wij de mini-batchgrootte dus veranderd van 1500 naar 7500, in de hoop dat een mini-batch van deze grootte

wel representatief zou zijn voor de gehele dataset.

Hierna zagen we wat we wilden: in de meeste gevallen daalden tijdens het trainen de kosten, zowel op de trainingsdata als op de testdata. En die theoretische getallen betekenden in de praktijk gelukkig dat ons programma inderdaad bijna alle partijen won.

Het probleem dat we hebben geschetst aan het begin van de vorige sectie, is nu dus opgelost.

```
Games have been played. Now training...
Training data cost before training: 0.0477155
Training data cost after training: 0.0466783
Test data cost before training: 0.0688719
Test data cost after training: 0.0670594

Training finished.
Training finished. The current learning rate is 0.0411687
THREAD 28 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 1 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 31 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 11 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 14 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 9 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 19 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 2 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 4 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 6 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 25 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 16 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 8 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 22 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 5 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 7 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 24 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 13 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 29 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 17 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 27 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 10 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 15 : Won 9 out of 10 games and 5 out of 5 games.
THREAD 12 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 0 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 30 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 18 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 23 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 21 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 3 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 26 : Won 10 out of 10 games and 5 out of 5 games.
THREAD 20 : Won 10 out of 10 games and 5 out of 5 games.
```

Het programma wint 479 van de 480 gespeelde partijen tegen de ‘smartened random’ in deze partijenset. (Ziet u waar hij er een heeft verloren?)

Hoofdstuk 5

Het bewijs en de code van backpropagation

We gaan kijken naar de kostenfunctie C . We **willen weten** wat $\frac{\partial C}{\partial b}$ en $\frac{\partial C}{\partial w}$ zijn, voor alle biasen en gewichten in het hele netwerk. Dit is ons doel in dit hoofdstuk. Vervolgens laten we een deel van onze C++-code zien, met betrekking tot het bepalen van de gradiënt van de kostenfunctie, de vector met als componenten de partiële afgeleiden die we bepalen in dit hoofdstuk.

Graag zouden wij de lezer erop attenderen dat wij dit bewijs zelf hebben bedacht, terwijl we dit hoofdstuk aan het schrijven waren. We hebben het bewijs dus niet overgenomen van internet. Tegelijk met het ontdekken van het bewijs, zijn wij dit hoofdstuk al gaan schrijven. Daarom staat in dit hoofdstuk niet alleen het bewijs zelf, maar zie je ook de denkstappen die we hebben genomen om dit bewijs te bedenken.

Wij denken dat we hiervoor de kettingregel voor functies met veel variabelen nodig hebben. Zie voor meer uitleg hierover de sectie over de kettingregel in het hoofdstuk met voorbereidende wiskunde voor dit profielwerkstuk.

We gaan ervan uit dat we de vorm weten van de kostenfunctie, als functie van de uitvoerwaarden en de gewenste uitvoerwaarden van de uitvoerneuronen van het netwerk. We zeggen ook dat de gewenste uitvoerwaarden constant zijn (bij een gegeven invoer). Aangezien we de vorm weten van de kostenfunctie (die bepalen we namelijk zelf), is het dan niet zo moeilijk om de volgende afgeleide te bepalen, voor de uitvoerwaarden $u[n]$ van alle uitvoerneuronen n :

$$\frac{\partial C}{\partial u[n]} \tag{5.1}$$

We introduceren nu de hoofdletternotatie L , hiermee bedoelen we dat het

om de uitvoerlaag gaat. We weten dat geldt $u[n] = a[L][n] = a(s[L][n])$. Voor een gegeven neuron n in de uitvoerlaag zijn de waarden $s[L][n]$ afhankelijk ten eerste van alle activatiewaarden $a[L - 1][m]$ voor alle neuronen m in de voorlaatste laag, ten tweede van de bias van dit uitvoerneuron, $b[L][n]$, en ten derde van de gewichten naar dit uitvoerneuron, dus $w[L][n][m]$ voor alle m . De afhankelijkheid schematisch:

$$\begin{array}{c} a[L-1][m] \text{ voor alle } m \\ b[L][n] \\ w[L][n][m] \text{ voor alle } m \end{array} \rightarrow s[L][n] \quad (5.2)$$

5.1 Het bepalen van $\frac{\partial C}{\partial b[L][n]}$

Laten we nu proberen om van neuron n in de uitvoerlaag te bepalen wat $\frac{\partial C}{\partial b[L][n]}$ is, dus de partiële afgeleide van de kostenfunctie als de bias van uitvoerneuron n verandert. Als we deze partiële afgeleide opstellen, zijn alle andere gewichten en biasen in het hele netwerk constant (zie de sectie over de partiële afgeleide van het hoofdstuk over wiskunde achtergronden). De invoerwaarden voor het netwerk zijn ook constant. De eerste verborgen laag van het netwerk krijgt dan dus constante invoerwaarden binnen, en met constante gewichten en biasen zijn ook de gewogen sommen van alle neuronen in de eerste verborgen laag gelijk. De uitvoerwaarden van deze eerste verborgen laag zijn dus ook constant. Dit betekent dat de tweede verborgen laag constante invoerwaarden krijgt. Onder invloed van constante biasen en gewichten, zijn ook de gewogen sommen constant, en dus zijn alle uitvoerwaarden van de neuronen van de tweede verborgen laag ook constant.

Alle uitvoerwaarden/activaties van de eerste paar lagen van het netwerk, zijn dus steeds constant. Dit gaat net zolang door, tot we uitkomen bij de laatste laag, waarin een van de biasen niet constant is (we zijn immers de partiële afgeleide aan het bepalen met als veranderende variabele die bias). We gaan nu weer specifiek naar die laatste laag kijken. Door het verhaal van net, weten we nu dat **$a[L - 1][m]$ constant** is voor alle neuronen m in laag $L - 1$.

Vanaf nu zullen we de notatie `.size()` gebruiken om ergens de grootte van aan te geven. Dus `net[L].size()` is de grootte van laag L van het netwerk, in dit geval het aantal neuronen in laag L. Het laatste neuron heeft dan index `net[L].size() - 1`, omdat we beginnen te tellen bij nul.

We willen $\frac{\partial C}{\partial b[L][n]}$ weten, en het lijkt handig om hier de kettingregel voor te gebruiken. De kostenfunctie C is afhankelijk van de variabelen $a[L][0], a[L][1], a[L][2], \dots, a[L][n], \dots, a[L][\text{net}[L].\text{size}() - 1]$. Deze variabelen zijn, vanwege het verhaal van net, allemaal constant, behalve $a[L][n]$ (dit komt omdat de veranderende bias $b[L][n]$ alleen invloed heeft op $a[L][n]$, en niet op de andere activaties in laag L).

De variabelen $a[L][0], a[L][1], \dots, a[L][n], \dots, a[L][\text{net}[L].\text{size}() - 1]$, zijn dus allemaal constant behalve $a[L][n]$: die is afhankelijk van $b[L][n]$ en $w[L][n][m]$ voor alle neuronen m in laag $L - 1$.

We kunnen het volgende ‘afhankelijkheidsschema’ maken om de kettingregel toe te passen:

$$\begin{array}{c}
 C \\
 \uparrow \\
 \underbrace{a[L][0], a[L][1], \dots, a[L][n], \dots, a[L][\text{net}[L].\text{size}() - 1]}_{\text{constant}} \\
 \uparrow \qquad \qquad \qquad \uparrow \\
 b[L][n] \text{ en } \underbrace{w[L][n][k] \text{ voor alle } k}_{\text{constant}}
 \end{array}$$

Je ziet dat de onderste regel van dit schema bestaat uit allemaal onafhankelijke variabelen. Als je de bias verandert, blijven alle gewichten gewoon hetzelfde. $w[L][n][k]$ (voor alle k) is constant, want we zijn de partiële afgeleide $\frac{\partial C}{\partial b[L][n]}$ aan het proberen te bepalen. Alleen $b[L][n]$ verandert dus, en alle andere gewichten en biasen in het hele netwerk veranderen niet en zijn dan constant. We kunnen nu de kettingregel toepassen (zie ook de sectie over de kettingregel in het hoofdstuk met voorbereidende wiskunde).

De kettingregel zegt:

$$\underbrace{\frac{\partial C}{\partial b[L][n]}}_{\text{willen we uiteindelijk weten}} = \underbrace{\frac{\partial C}{\partial a[L][n]}}_{\text{bekend}} \frac{\partial a[L][n]}{\partial b[L][n]} \tag{5.3}$$

(We zitten in de uitvoerlaag, en er geldt $a[L][n] = u[n]$. Dan geldt ook $\frac{\partial C}{\partial a[L][n]} = \frac{\partial C}{\partial u[n]}$. Aangezien we onze kostenfunctie zelf kunnen bedenken, kunnen we dan ook bepalen wat de partiële afgeleide $\frac{\partial C}{\partial u[n]}$ van die kostenfunctie is. Zodoende weten we ook $\frac{\partial C}{\partial a[L][n]}$. Dit is een handige denkstap,

omdat het ons op het idee brengt om $\frac{\partial a[L][n]}{\partial b[L][n]}$ te bepalen (zie vergelijking (5.3)). Deze denkstap is echter **niet noodzakelijk** voor het bewijs zelf.)

Om $\frac{\partial C}{\partial b[L][n]}$ te weten te komen, moeten we dus $\frac{\partial a[L][n]}{\partial b[L][n]}$ bepalen. Dat gaan we nu doen, met wat tussenstappen:

$$a[L][n] = a(s[L][n]) \quad (5.4)$$

$$a[L][n] = a \left(b[L][n] + \sum_{k=0}^{\text{net}[L-1].\text{size}()-1} (a[L-1][k] \cdot w[L][n][k]) \right) \quad (5.5)$$

Dit laatste ziet er ingewikkeld uit, maar het houdt eigenlijk alleen in dat we $s[L][n]$ hebben herschreven volgens zijn definitie (namelijk bias + gewogen som van invoerwaarden van de neuronen).

Vervolgens willen we er uiteindelijk naartoe dat we $\frac{\partial a[L][n]}{\partial b[L][n]}$ bepalen:

$$\frac{\partial a[L][n]}{\partial b[L][n]} = \frac{\partial}{\partial b[L][n]} a \left(b[L][n] + \sum_{k=0}^{\text{net}[L-1].\text{size}()-1} (a[L-1][k] \cdot w[L][n][k]) \right) \quad (5.6)$$

Merk op dat bij het bepalen van deze partiële afgeleide, alleen $b[L][n]$ verandert; alle andere gewichten en biasen in het hele netwerk blijven constant. Om die reden is $a[L-1][k]$ constant voor elke k , dat is namelijk de laag vòòr de laag waarin de veranderende bias zit. Ook is $w[L][n][k]$ constant voor elke k . Dit betekent dus dat $a[L-1][k] \cdot w[L][n][k]$ ook constant is voor elke k .

En dit betekent vervolgens weer dat $\sum_{k=0}^{\text{net}[L-1].\text{size}()-1} (a[L-1][k] \cdot w[L][n][k])$ óók constant is. Oftewel:

$$\frac{\partial}{\partial b[L][n]} \left(\sum_{k=0}^{\text{net}[L-1].\text{size}()-1} (a[L-1][k] \cdot w[L][n][k]) \right) = 0 \quad (5.7)$$

Hieruit volgt:

$$\frac{\partial}{\partial b[L][n]} \left(b[L][n] + \sum_{k=0}^{\text{net}[L-1].\text{size}()-1} (a[L-1][k] \cdot w[L][n][k]) \right) = 1 \quad (5.8)$$

Dit bevat precies de definitie van $s[L][n]$ en dus kunnen we het bovenstaande véél compacter schrijven:

$$\frac{\partial s[L][n]}{\partial b[L][n]} = 1 \quad (5.9)$$

We gaan verder met het bepalen van $\frac{\partial a[L][n]}{\partial b[L][n]}$:

$$\frac{\partial a[L][n]}{\partial b[L][n]} = \frac{\partial}{\partial b[L][n]} a(s[L][n]) = \underbrace{\frac{da[L][n]}{ds[L][n]}}_{\text{kettingregel}} \underbrace{\frac{\partial s[L][n]}{\partial b[L][n]}}_{= 1} \quad (5.10)$$

$$\frac{\partial a[L][n]}{\partial b[L][n]} = \frac{da[L][n]}{ds[L][n]} \quad (5.11)$$

Aha! We moeten opmerken dat deze laatste term ons gewoon bekend is. Hij betekent niets anders dan de afgeleide van onze activatiefunctie als zijn directe invoer verandert. $s[L][n]$ is namelijk de directe invoer van de activatiefunctie. En onze activatiefunctie kunnen we nu eenmaal zelf kiezen, dus weten we er ook de afgeleide van.

(Overigens: we noteren $\frac{da[L][n]}{ds[L][n]}$ en niet $\frac{\partial a[L][n]}{\partial s[L][n]}$. Dit doen we omdat $a[L][n]$, wanneer het een functie is van $s[L][n]$, **niet** ook nog andere variabelen als invoer nodig heeft. We hebben in dit geval dus niet te maken met een partiële afgeleide, maar met een ‘normale’ afgeleide.)

Als we $\frac{da[L][n]}{ds[L][n]}$ weten, is dat gelijk aan $\frac{\partial a[L][n]}{\partial b[L][n]}$: die weten we nu dus ook.

Kijk nog eens naar vergelijking (5.3):

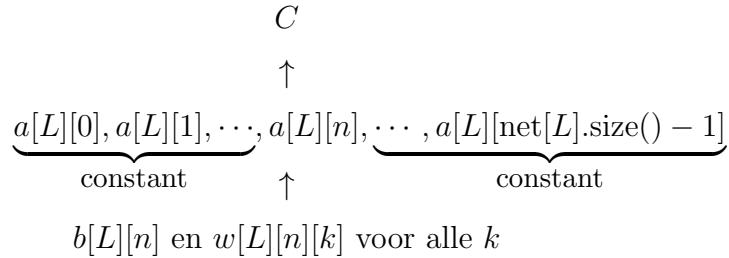
$$\underbrace{\frac{\partial C}{\partial b[L][n]}}_{\text{willen we uiteindelijk weten}} = \underbrace{\frac{\partial C}{\partial a[L][n]}}_{\text{deze wisten we al}} \underbrace{\frac{\partial a[L][n]}{\partial b[L][n]}}_{\text{deze weten we nu ook!}}$$

De laatste partiële afgeleide ‘invullen’ geeft ons de volgende vergelijking:

$$\boxed{\frac{\partial C}{\partial b[L][n]} = \frac{\partial C}{\partial a[L][n]} \frac{da[L][n]}{ds[L][n]}} \quad (5.12)$$

5.2 Het bepalen van $\frac{\partial C}{\partial w[L][n][m]}$

Laten we nu eens kijken of het ons lukt om $\frac{\partial C}{\partial w[L][n][m]}$ te vinden voor willekeurige n en m . We hebben nog steeds het afhankelijkheidsschema:



In de middelste laag van het afhankelijkheidsschema zijn alle variabelen constant, behalve $a[L][n]$, want dat is de enige variabele waarop de veranderende variabele $w[L][n][m]$ invloed heeft.

Alle variabelen in de onderste laag zijn constant, behalve $w[L][n][m]$, omdat we die nu laten veranderen om de partiële afgeleide $\frac{\partial C}{\partial w[L][n][m]}$ te vinden. Alle variabelen in de onderste laag van bovenstaand schema zijn onafhankelijke variabelen. We zullen zodra de kettingregel toepassen.

We krijgen nu voor een groot deel herhaling van wat we al gedaan hebben toen we $\frac{\partial C}{\partial b[L][n]}$ aan het bepalen waren. We gaan er nu dus wat sneller doorheen:

$$\frac{\partial C}{\partial w[L][n][m]} = \overbrace{\frac{\partial C}{\partial a[L][n]} \frac{\partial a[L][n]}{\partial w[L][n][m]}}^{\text{kettingregel, zie afhankelijkheidsschema}} \quad (5.13)$$

$$= \frac{\partial C}{\partial a[L][n]} \overbrace{\frac{\text{da}[L][n]}{\text{ds}[L][n]} \frac{\partial s[L][n]}{\partial w[L][n][m]}}^{\text{kettingregel}} \quad (5.14)$$

$$= \frac{\partial C}{\partial a[L][n]} \frac{\text{da}[L][n]}{\text{ds}[L][n]} \frac{\partial}{\partial w[L][n][m]} \left(\underbrace{b[L][n]}_{\text{constant}} + \sum_{k=0}^{\text{net}[L-1].\text{size}() - 1} (a[L-1][k] \cdot w[L][n][k]) \right) \quad (5.15)$$

$$= \frac{\partial C}{\partial a[L][n]} \frac{da[L][n]}{ds[L][n]} \frac{\partial}{\partial w[L][n][m]} \left(\sum_{k=0}^{\text{net}[L-1].\text{size}() - 1} \left(\underbrace{a[L-1][k]}_{\text{constant}} \cdot \underbrace{w[L][n][k]}_{\substack{\text{constant} \\ \text{als } k \neq m}} \right) \right) \quad (5.16)$$

$$= \frac{\partial C}{\partial a[L][n]} \frac{da[L][n]}{ds[L][n]} \frac{\partial}{\partial w[L][n][m]} \left(\underbrace{a[L-1][m]}_{\text{constant}} \cdot w[L][n][m] \right) \quad (5.17)$$

Dit geeft ons het volgende:

$$\frac{\partial C}{\partial w[L][n][m]} = \frac{\partial C}{\partial a[L][n]} \frac{da[L][n]}{ds[L][n]} \cdot a[L-1][m] \quad (5.18)$$

We herhalen vergelijking 5.12:

$$\boxed{\frac{\partial C}{\partial b[L][n]} = \frac{\partial C}{\partial a[L][n]} \frac{da[L][n]}{ds[L][n]}}$$

Daaraan zien we dat we vergelijking 5.18 ook als volgt kunnen noteren:

$$\boxed{\frac{\partial C}{\partial w[L][n][m]} = \frac{\partial C}{\partial b[L][n]} \cdot a[L-1][m]} \quad (5.19)$$

5.3 Van uitvoerlaag naar alle lagen

De bovenstaande twee vergelijkingen zijn extreem belangrijk. Met deze twee vergelijkingen hebben we namelijk ons doel voor dit hoofdstuk behaald, maar wel alleen nog voor de uitvoerlaag. Nu volgt een cruciale denkstap. We moeten inzien dat we in het bewijzen van de bovenstaande twee vergelijkingen, **geen gebruik hebben gemaakt van het feit dat het om de uitvoerlaag ging**. Zoals eerder ook al besproken was dit feit wel handig en kwamen we zo op het idee om bepaalde denkstappen uit te voeren, maar het feit dat je in de uitvoerlaag zat, was voor het bewijs zelf niet relevant.

We kunnen in de vergelijkingen dus gewoon de hoofdletter L door een kleine letter l vervangen en dan klopt het nog steeds. We weten nu dus dat

voor elke laag l met $0 \leq l \leq L$, en voor elke willekeurige n en m , de volgende vergelijkingen gelden:

$$\boxed{\frac{\partial C}{\partial b[l][n]} = \frac{\partial C}{\partial a[l][n]} \frac{da[L][n]}{ds[L][n]}} \quad (5.20)$$

$$\boxed{\frac{\partial C}{\partial w[l][n][m]} = \frac{\partial C}{\partial b[l][n]} \cdot a[l-1][m]} \quad (5.21)$$

Laten we nu in de bovenstaande twee vergelijkingen kijken wat we al weten en wat nog niet:

$$\begin{array}{ccc} \text{willen we uiteindelijk weten} & & \text{bekend (zie toelichting 1)} \\ \overbrace{\frac{\partial C}{\partial b[l][n]}} & = & \frac{\partial C}{\partial a[l][n]} \quad \overbrace{\frac{da[L][n]}{ds[L][n]}} \\ & & \end{array} \quad (5.22)$$

$$\begin{array}{ccc} \underbrace{\frac{\partial C}{\partial w[l][n][m]}} & = & \frac{\partial C}{\partial b[l][n]} \quad \underbrace{a[l-1][m]} \\ \text{willen we ook weten} & & \text{bekend (zie toelichting 2)} \end{array} \quad (5.23)$$

Toelichting 1: $s[l][n]$ is de directe invoer van de activatiefunctie; $a[l][n] = a(s[l][n])$. We kunnen zelf kiezen welke activatiefunctie we gebruiken, en we weten dan dus ook de afgeleide. Daarom is $\frac{da[L][n]}{ds[L][n]}$ bekend.

Toelichting 2: we passen backpropagation toe als we al een netwerk hebben met biasen en gewichten, en bij een gegeven invoer aan het netwerk. Alle data is dan al door het hele netwerk gestroomd van links naar rechts en pas dan gaan we kijken wat de gradiënt van de kostenfunctie is (Nielsen, 2015, Hoofdstuk 2). Op het moment dat we backpropagation toepassen, is $a[l-1][m]$ dus bekend, want alle activaties zijn bekend als de data al door het netwerk is gestroomd (Nielsen, 2015, Hoofdstuk 2).

Overigens zal het de oplettende lezer wellicht zijn opgevallen dat $a[l-1][m] = a[-1][m]$ als $l = 0$. Je hebt dan index -1 . Wij kiezen er voor om dit toch gewoon zo te gebruiken in het profielwerkstuk, ook al ziet het er wellicht niet zo netjes uit. Toch klopt het wel. We hadden namelijk al afgesproken dat de laagnummering begint bij -1 voor de invoerlaag, zodat de eerste verborgen laag index 0 krijgt. (De invoerlaag is namelijk een best vreemde laag. De neuronen hebben geen bias en geen gewichten; ze hebben alleen uitvoerwaarden. Dit is de reden dat we de invoerlaag laag -1 noemen.) $a[-1][m]$ betekent dan: de activatie/uitvoerwaarde van neuron m in de invoerlaag.

Als we weer kijken naar onze twee vergelijkingen, valt ons het volgende op: als we weten wat $\frac{\partial C}{\partial a[l][n]}$ is, weten we opeens alles wat we willen weten in dit hoofdstuk, namelijk $\frac{\partial C}{\partial b[l][n]}$ en daardoor ook $\frac{\partial C}{\partial w[l][n][m]}$. Onze volgende tussenstap is dus om $\frac{\partial C}{\partial a[l][n]}$ te bepalen.

5.4 Het bepalen van $\frac{\partial C}{\partial a[l][n]}$

We gaan in deze sectie proberen om $\frac{\partial C}{\partial a[l][n]}$ te bepalen. Als eerste merken we op dat we deze partiële afgeleide in het geval van de uitvoerlaag reeds weten, want $\frac{\partial C}{\partial a[L][p]} = \frac{\partial C}{\partial u[p]}$ voor elke p en we hebben zelf onze kostenfunctie bepaald dus we weten daarvan ook de afgeleide. Kortom: $\frac{\partial C}{\partial a[L][p]}$ voor elke p weten we (let op de hoofdletter L die aangeeft dat het om de uitvoerlaag gaat).

We kunnen nu met de kettingregel voor complexe functies, ook $\frac{\partial C}{\partial a[L-1][q]}$ voor elke willekeurige q bepalen. We stellen weer een afhankelijkheidsschema op, om de kettingregel toe te kunnen passen:

$$\begin{array}{c}
 C \\
 \uparrow \\
 a[L][0], a[L][1], \dots, a[L][\text{net}[L].\text{size}() - 1] \\
 \uparrow \\
 a[L-1][0], a[L-1][1], \dots, a[L-1][\text{net}[L-1].\text{size}() - 1]
 \end{array}$$

We gebruiken de gealgemeiniseerde kettingregel zoals deze is uitgelegd in

het hoofdstuk over de wiskundige achtergronden. Wij krijgen dan:

$$\begin{aligned}
 & \text{willen we uiteindelijk weten} \\
 & \overbrace{\frac{\partial C}{\partial a[L-1][q]}}^{\text{bekend}} = \overbrace{\frac{\partial C}{\partial a[L][0]}}^{\text{bekend}} \frac{\partial a[L][0]}{\partial a[L-1][q]} + \overbrace{\frac{\partial C}{\partial a[L][1]}}^{\text{bekend}} \frac{\partial a[L][1]}{\partial a[L-1][q]} + \\
 & \quad \cdots + \underbrace{\frac{\partial C}{\partial a[L][\text{net}[L].\text{size}() - 1]}}_{\text{bekend}} \frac{\partial a[L][\text{net}[L].\text{size}() - 1]}{\partial a[L-1][q]}
 \end{aligned} \tag{5.24}$$

$\frac{\partial C}{\partial a[L][p]}$ voor elke p is bekend; dit is hierboven reeds uitgelegd. Om $\frac{\partial C}{\partial a[L-1][q]}$ te weten te komen, moeten we dus eerst als tussenstap $\frac{\partial a[L][0]}{\partial a[L-1][q]}$, $\frac{\partial a[L][1]}{\partial a[L-1][q]}$, etcetera, oftewel in het algemeen $\frac{\partial a[L][p]}{\partial a[L-1][q]}$, bepalen. Dat gaan we nu doen.

$$\frac{\partial a[L][p]}{\partial a[L-1][q]} = \underbrace{\frac{\text{da}[L][p]}{\text{ds}[L][p]}}_{\text{kettingregel}} \underbrace{\frac{\partial s[L][p]}{\partial a[L-1][q]}}^{\text{bekend}} \tag{5.25}$$

$$= \frac{\text{da}[L][p]}{\text{ds}[L][p]} \frac{\partial}{\partial a[L-1][q]} \left(\underbrace{b[L][p]}_{\text{constant}} + \sum_{k=0}^{\text{net}[L-1].\text{size}() - 1} (a[L-1][k] \cdot w[L][p][k]) \right) \tag{5.26}$$

$$= \frac{\text{da}[L][p]}{\text{ds}[L][p]} \frac{\partial}{\partial a[L-1][q]} \left(\sum_{k=0}^{\text{net}[L-1].\text{size}() - 1} \left(\underbrace{a[L-1][k]}_{\text{constant}} \cdot \underbrace{w[L][p][k]}_{\text{constant}} \right) \right) \tag{5.27}$$

$$= \frac{\text{da}[L][p]}{\text{ds}[L][p]} \frac{\partial}{\partial a[L-1][q]} \left(a[L-1][q] \cdot \underbrace{w[L][p][q]}_{\text{constant}} \right) \tag{5.28}$$

Conclusie:

$$\frac{\partial a[L][p]}{\partial a[L-1][q]} = \frac{da[L][p]}{ds[L][p]} \cdot w[L][p][q] \quad (5.29)$$

We kunnen nu vergelijking 5.24 invullen. Voordat we dit doen, schrijven we echter eerst vergelijking 5.24 even anders, om te voorkomen dat we té lange en onoverzichtelijke vergelijkingen krijgen. Vergelijking 5.24 schrijven we als volgt om:

$$\frac{\partial C}{\partial a[L-1][q]} = \sum_{p=0}^{\text{net}[L].\text{size}()-1} \left(\frac{\partial C}{\partial a[L][p]} \frac{\partial a[L][p]}{\partial a[L-1][q]} \right) \quad (5.30)$$

Hierin kunnen we vergelijking 5.29 invullen om de volgende vergelijking te verkrijgen:

$$\frac{\partial C}{\partial a[L-1][q]} = \sum_{p=0}^{\text{net}[L].\text{size}()-1} \left(\frac{\partial C}{\partial a[L][p]} \frac{da[L][p]}{ds[L][p]} \cdot w[L][p][q] \right) \quad (5.31)$$

Maar nu kunnen we precies dezelfde ‘grap’ uithalen als eerder in dit hoofdstuk: de formule generaliseren naar andere lagen. We hebben immers wederom nergens in de bovenstaande afleiding gebruik gemaakt van het feit dat het om de uitvoerlaag (L) en de laag daarvoor ($L - 1$) ging. Dit feit heeft ons wel op ideeën gebracht, maar voor het bewijs zelf was het wederom niet relevant. We kunnen onze hoofdletter L dus veranderen in een kleine letter l :

$$\frac{\partial C}{\partial a[l-1][q]} = \sum_{p=0}^{\text{net}[l].\text{size}()-1} \left(\frac{\partial C}{\partial a[l][p]} \frac{da[l][p]}{ds[l][p]} \cdot w[l][p][q] \right) \quad (5.32)$$

We kunnen nu de q veranderen in een n en bij alle laag-indexen één optellen, om de volgende vergelijking te vinden:

$$\frac{\partial C}{\partial a[l][n]} = \sum_{p=0}^{\text{net}[l+1].\text{size}()-1} \left(\frac{\partial C}{\partial a[l+1][p]} \frac{da[l+1][p]}{ds[l+1][p]} \cdot w[l+1][p][n] \right) \quad (5.33)$$

Nu hebben we $\frac{\partial C}{\partial a[l][n]}$ bepaald, het doel van deze sectie!

Opeens heb ik iets door (en dat is echt zo, ik zie het echt plots op dit moment): vergelijk het bovenstaande met vergelijking 5.20. We zien dat we bovenstaande vergelijking wat compacter kunnen schrijven, namelijk als volgt:

$$\frac{\partial C}{\partial a[l][n]} = \sum_{p=0}^{\text{net}[l+1].\text{size}()-1} \left(\frac{\partial C}{\partial b[l+1][p]} \cdot w[l+1][p][n] \right) \quad (5.34)$$

We moeten wel blijven opletten: de bovenstaande vergelijking geldt natuurlijk alleen als $0 \leq l \leq L - 1$. Omdat in de vergelijking een paar keer $l + 1$ voorkomt, mag l niet gelijk zijn aan L .

5.5 Het bepalen van $\frac{\partial C}{\partial b[l][n]}$ en $\frac{\partial C}{\partial w[l][n][m]}$

We hadden al de vergelijkingen 5.20 en 5.21 gevonden:

$$\begin{aligned} \frac{\partial C}{\partial b[l][n]} &= \frac{\partial C}{\partial a[l][n]} \frac{da[L][n]}{ds[L][n]} \\ \frac{\partial C}{\partial w[l][n][m]} &= \frac{\partial C}{\partial b[l][n]} \cdot a[l-1][m] \end{aligned}$$

Als $l = L$, is $\frac{\partial C}{\partial a[l][n]}$ al bekend; eerder in dit hoofdstuk is dit al besproken. Als $0 \leq l \leq L - 1$, kunnen we de vergelijking 5.34 invullen in vergelijking 5.20, die ook hierboven is afgedrukt. Al met al vinden we het volgende.

Voor l met $0 \leq l \leq L$, voor n met $0 \leq n < \text{net}[l].\text{size}()$, en voor m met $0 \leq m < \text{net}[l-1].\text{size}()$, hebben wij zelf bewezen dat de volgende vergelijkingen gelden:

$$\frac{\partial C}{\partial b[l][n]} = \begin{cases} \frac{da[l][n]}{ds[l][n]} \cdot \sum_{p=0}^{\text{net}[l+1].\text{size}()-1} \left(\frac{\partial C}{\partial b[l+1][p]} \cdot w[l+1][p][n] \right), & \text{als } 0 \leq l < L \\ \frac{da[l][n]}{ds[l][n]} \cdot \frac{\partial C}{\partial a[l][n]}, & \text{als } l = L \end{cases} \quad (5.35)$$

EN

$$\boxed{\frac{\partial C}{\partial w[l][n][m]} = \frac{\partial C}{\partial b[l][n]} \cdot a[l - 1][m]} \quad (5.36)$$

Zo. Ons doel in dit hoofdstuk was om $\frac{\partial C}{\partial b}$ en $\frac{\partial C}{\partial w}$ te bepalen, voor alle biasen en gewichten van het netwerk. Wie kijkt naar de linkerleden van de twee bovenstaande vergelijkingen, ziet dat dat gelukt is, en dat we dus ons hoofdstukdoel behaald hebben. Nu volgt de C++-code.

5.6 Backpropagation in C++

Nu we zelf een bewijs hebben bedacht voor de bovenstaande twee vergelijkingen, laten we ook zien hoe we dit in onze code hebben geïmplementeerd.

5.6.1 Het bepalen van de gradiënt

Hieronder staat de functie `Athena::gradient()` afgedrukt. In deze C++-functie wordt de *gemiddelde* (over een aantal training inputs) gradiënt van de kostenfunctie naar alle biasen en gewichten bepaald. Je ziet in de functie ook ons oorspronkelijke commentaar, dat we voor onszelf hebben geschreven terwijl we aan het programmeren waren. Overigens is dit in het Engels geschreven, omdat Engels een wereldtaal is. Programmeurs van over de hele wereld kunnen je code begrijpen als je er Engels commentaar bij plaatst. Daarom, en als voorbereiding op onze toekomst, hebben wij ons commentaar ook in het Engels geschreven.

Wie goed kijkt, herkent in de hieronder afgedrukte functie de formules 5.35 en 5.36.

```
// averaged gradient for variable number of training input(s)
network Athena::gradient(vvd input_datas, vvd desired_output_datas) {
    double number_of_training_inputs = input_datas.size() + 0.0;
    if (number_of_training_inputs == 0.0) {
        std::cout << "Error line " << __LINE__ << __FILE__ << "\n";
        exit(0);
    }

    // BEGIN construct the to-be AVERAGE gradient vector (correct size) with
    // zeroes This vector will contain the AVERAGE gradient, AVERAGED over
    // training inputs passed on to this function.
    network avggrad;
```

```

avggrad.reserve(number_of_layers - 1);
for (int l = 0; l <= number_of_layers - 2; l++) {
    int neurons_in_current_layer_l = layer_sizes[l + 1];
    int neurons_in_layer_left_to_current_layer_l = layer_sizes[l];
    avggrad.push_back(
        layer(neurons_in_current_layer_l,
              neuron(1 + neurons_in_layer_left_to_current_layer_l, 0.0)));
}
// END construct the to-be AVERAGE gradient vector (correct size) with
// zeroes

// for every training input...
for (int t = 0; t < input_datas.size(); t++) {
    auto input_data =
        input_datas[t]; // vector of numbers; one current training input
    auto desired_output_data =
        desired_output_datas[t]; // vector of numbers; the desired output
                                // for one current training input
    auto w = desired_output_data;

    // BEGIN compute all the weighted sums in the network and all the
    // activations
    vvd s;
    // a vector with all the weighted sums (plus bias) of the neurons.
    // The big vector contains all layers except the input layer. Therefore
    // the first hidden layer gets index 0. All the subvectors contain the
    // weighted sums (+bias) of those neurons in their layers.

    vvd a;
    // The same as s which is defined right above. But this vector has had
    // the activation function applied to all its values.

    get_weighted_sums_and_activations(input_data, s,
                                      a); // this does the magic
}
// END compute all the weighted sums in the network and all the
// activations

// BEGIN construct the gradient vector (correct size) with zeroes for
// this current SINGLE training input
network grad;
grad.reserve(number_of_layers - 1);
for (int l = 0; l <= number_of_layers - 2; l++) {
    int neurons_in_current_layer_l =
        layer_sizes[l + 1]; // the layer_sizes numbering is kind of
                            // weird, because it includes the input layer
                            // and gives that index 0.
    int neurons_in_layer_left_to_current_layer_l =
        layer_sizes[l]; // therefore, when using the layer_sizes array,
                      // WE MUST BE EXTREMELY CAREFUL OF THE NUMBERING.
}

```

```

grad.push_back(layer(
    neurons_in_current_layer_l,
    neuron(1 + neurons_in_layer_left_to_current_layer_l, 0.0)));
}

// END construct the gradient vector (correct size) with zeroes for
// this current SINGLE training input

// BEGIN actually computing the gradient for this single training input

// We will go backward through the layers of the network.

int L = net.size() - 1; // output layer
for (int l = L; l >= 0;
l--) { // the loop starts at output layer and stops when it has
// passed the first hidden layer
#pragma omp parallel for
    for (int n = 0; n < net[l].size(); n++) { // loop through all
        // neurons
        // we are looking at neuron: net[l][n]

        // BEGIN compute grad[l][n][0]
        // the bias of this neuron is: net[l][n][0] = b[l][n] (the first
        // one is used in the code, the second in the pdf) this means
        // that we are going to locate (partial C)/(partial b[l][n]) in
        // the grad vector, at: grad[l][n][0]. Furthermore, the
        // beautiful thing is that we can just write s[l][n] in this
        // code. It is exactly the same notation as in the pdf :)
        if (l == L) {
            if (which_cost_function == "cross-entropy" &&
                which_activation_function == "sigmoid") {
                grad[l][n][0] = a[l][n] - w[n];
            } else {
                grad[l][n][0] =
                    derivative_activation_function(s[l][n]) *
                    partial_C_over_a_L_n(a[l][n], w[n]);
            }
        } else {
            grad[l][n][0] = 0.0;
            for (int p = 0; p <= net[l + 1].size() - 1; p++) {
                grad[l][n][0] +=
                    (grad[l + 1][p][0] * net[l + 1][p][n + 1]);
            }
            grad[l][n][0] *= derivative_activation_function(s[l][n]);
        }
        if (std::isnan(grad[l][n][0])) {
            std::cout << "Error line " << __LINE__ << __FILE__ << "\n";
            exit(0);
        }
        // So, above, we computed the partial derivative of C with
    }
}

```

```

// respect to the bias  $b[l][n]$ .
avggrad[1][n][0] += grad[1][n][0] / number_of_training_inputs;
// END compute grad[l][n][0]

// BEGIN compute grad[l][n][1, 2, 3, ...]
// Now we are going to compute the p. derivative of C with
// respect to the weights  $w[l][n][m]$ . Note that  $w[l][n][m] =$ 
//  $net[l][n][m+1]$ . And therefore, the partial derivative of C
// with respect to  $w[l][n][m]$  is to be put to the variable
// grad[l][n][m+1].
for (int m = 0; m < net[1][n].size() - 1; m++) {
    if (l == 0) {
        grad[1][n][m + 1] = grad[1][n][0] * input_data[m];
        if (std::isnan(input_data[m])) {
            std::cout << "Error line " << __LINE__ << __FILE__
            << "\n";
            exit(0);
        }
    } else {
        grad[1][n][m + 1] =
            grad[1][n][0] *
            a[l - 1]
            [m]; // a[l-1][m] doesn't exist when l==0, because
                  // C++ can't handle negative index.
        // That's why we have a separate case for when l==0, see
        // above.
    }
    avggrad[1][n][m + 1] +=
        grad[1][n][m + 1] / number_of_training_inputs;
    if (std::isnan(avggrad[1][n][m + 1])) {
        std::cout << "Error line " << __LINE__ << __FILE__
        << "\n";
        exit(0);
    }
}
// END compute grad[l][n][1, 2, 3, ...]
}

// END actually computing the gradient for this single training input
}
return avggrad;
}

```

Hoofdstuk 6

Informatieoverzicht van functies en afgeleiden

In dit profielwerkstuk gebruiken we een hoop wiskundige functies, waar we vaak ook de afgeleide van willen weten. In dit hoofdstuk introduceren we deze functies en stellen we hun afgeleide functies op.

6.1 De sigmoïdefunctie

In dit profielwerkstuk zullen we de functie

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (6.1)$$

gebruiken, die we de sigmoïdefunctie noemen (Nielsen, 2015, Hoofdstuk 1). Deze functie gaan we differentiëren:

$$\sigma(x) = (1 + e^{-x})^{-1} \quad (6.2)$$

$$\sigma'(x) = -(1 + e^{-x})^{-2} \cdot e^{-x} \cdot -1 \quad (6.3)$$

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \quad (6.4)$$

Wat nu blijkt is dat ook geldt $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ (Nielsen, 2015, Hoofdstuk 3). We gaan nu (zelf) laten zien dat dat inderdaad klopt.

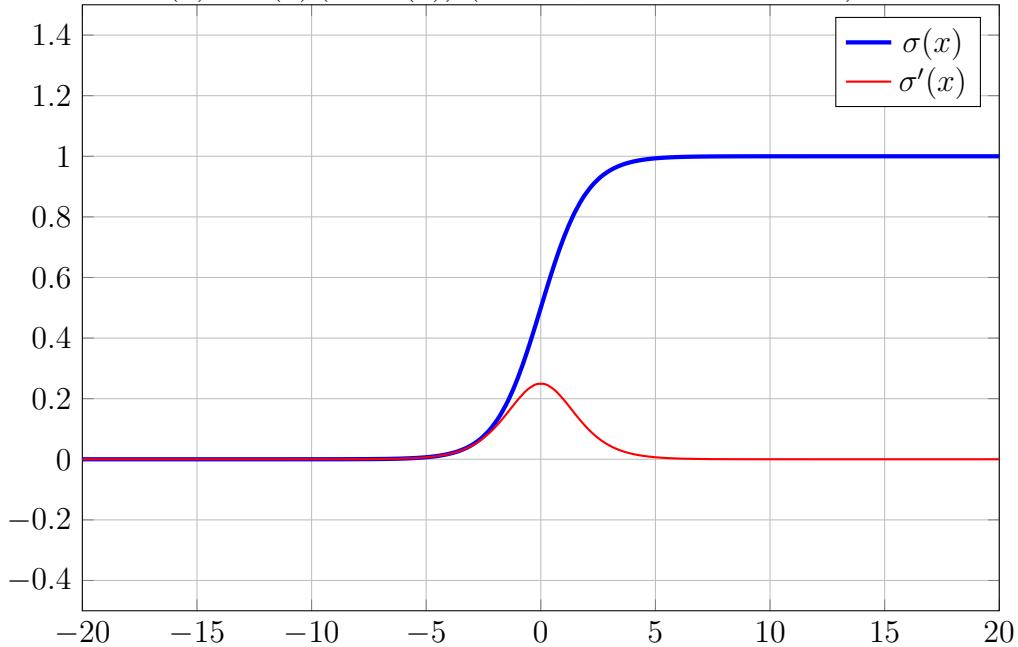
$$\sigma(x)(1 - \sigma(x)) = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) \quad (6.5)$$

$$= \frac{1}{1+e^{-x}} \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}} \right) \quad (6.6)$$

$$= \frac{1}{1+e^{-x}} \left(\frac{e^{-x}}{1+e^{-x}} \right) \quad (6.7)$$

$$= \frac{e^{-x}}{(1+e^{-x})^2} \quad (6.8)$$

Dit komt overeen met de formule voor $\sigma'(x)$ die we al hadden laten zien. We weten nu dus $\sigma'(x) = \sigma(x)(1-\sigma(x))$ (Nielsen, 2015, Hoofdstuk 3).



Hierboven staat een grafiek van de sigmoïdefunctie (blauw) en haar afgeleide (rood). Vooral de vorm van de sigmoïdefunctie (blauw) is belangrijk en komt later nog aan bod.

6.2 De (Leaky) Rectified Linear Unitfunctie

Als activatiefunctie bestaat ook de Rectified Linear Unitfunctie, oftewel ReLU.

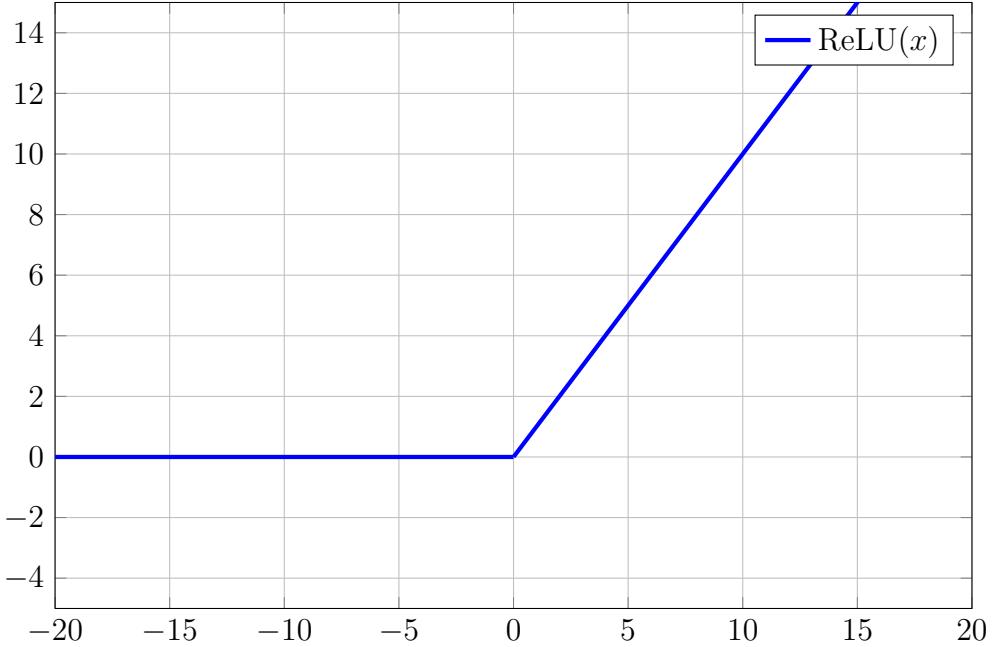
De formule van deze functie is erg simpel:

$$\text{ReLU}(x) = \max(0, x) \quad (6.9)$$

(Hansen, 2019).

Ook het opstellen van de afgeleide functie is niet moeilijk:

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{als } x < 0 \\ 1 & \text{als } x > 0 \end{cases} \quad (6.10)$$



Overigens bestaat er ook Leaky ReLU. Dat functievoorschrift luidt

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{als } x > 0 \\ px & \text{als } x \leq 0 \end{cases} \quad (6.11)$$

waarbij p een parameter is, een positief getal, vaak dichtbij de nul gekozen (SG, 2019).

Deze Leaky ReLU-functie lijkt erg op de normale ReLU en daarom achten we het niet nodig om hier ook de LeakyReLU-afgeleide op te stellen: de lezer kan dat zelf ook, indien gewenst.

6.3 De kwadratische kostenfunctie

Er bestaat een zogenaamde kwadratische kostenfunctie. In deze sectie zullen we deze functie definiëren en de afgeleide ervan opstellen.

De kwadratische kostenfunctie voor meerdere training inputs is

$$C = \frac{1}{2 \cdot \text{aantal training inputs}} \sum_{\text{alle training inputs}} |\vec{w} - \vec{u}|^2 \quad (6.12)$$

(Nielsen, 2015, Hoofdstuk 1).

We gebruiken hier pijltjes boven de w en u , om aan te geven dat het vectoren zijn. $\vec{w} - \vec{u}$ is dan ook een vector, en $|\vec{w} - \vec{u}|$ geeft dan de lengte aan van de vector $\vec{w} - \vec{u}$. $|\vec{w} - \vec{u}|$ is dus een maat voor het verschil tussen \vec{w} , de verlangde netwerkuitvoer, en \vec{u} , de eigenlijke netwerkuitvoer. Daarom is het intuïtief niet lastig te begrijpen dat deze functie kan worden gebruikt als kostenfunctie: hoe groter het verschil tussen \vec{w} en \vec{u} , hoe slechter het netwerk het doet en hoe hoger de kosten idealiter moeten zijn. Deze functie eerbiedigt die regel.

Laten we ervan uitgaan dat we kijken naar één training input. De kostenfunctie wordt dan logischerwijs:

$$C = \frac{1}{2} |\vec{w} - \vec{u}|^2 \quad (6.13)$$

Anders schrijven geeft:

$$C = \frac{1}{2} \sqrt{(w[0] - u[0])^2 + (w[1] - u[1])^2 + (w[2] - u[2])^2 + \dots + (w.back() - u.back())^2}^2 \quad (6.14)$$

De wortel en het kwadraat vallen tegen elkaar weg:

$$C = \frac{1}{2} ((w[0] - u[0])^2 + (w[1] - u[1])^2 + (w[2] - u[2])^2 + \dots + (w.back() - u.back())^2) \quad (6.15)$$

We willen de partiële afgeleide naar $u[n]$ voor willekeurige n weten:

$$\frac{\partial C}{\partial u[n]} = \frac{1}{2} \cdot \frac{\partial}{\partial u[n]} ((w[n] - u[n])^2) \quad (6.16)$$

$$\frac{\partial C}{\partial u[n]} = \frac{1}{2} \cdot 2(w[n] - u[n]) \cdot -1 \quad (6.17)$$

We draaien onder andere de u en de w om en krijgen:

$$\frac{\partial C}{\partial u[n]} = u[n] - w[n]$$

(6.18)

En de partiële afgeleide $\frac{\partial C}{\partial u[n]}$, oftewel $\frac{\partial C}{\partial a[L][n]}$, is opgesteld: handig voor bij de backpropagation!

6.4 De cross-entropy kostenfunctie

In dit profielwerkstuk maken we gebruik van een kostenfunctie die Nielsen (2015, Hoofdstuk 3) de cross-entropy kostenfunctie noemt. Deze kostenfunctie is volgens Nielsen (2015, Hoofdstuk 3) als volgt (omgezet naar onze eigen notatie):

$$C = -\frac{1}{\text{aantal training inputs}} \sum_{\text{alle training inputs}} \sum_n (w[n]\ln(u[n]) + (1-w[n])\ln(1-u[n])) \quad (6.19)$$

Maar dit is over meerdere training inputs. Laten we gauw de formule wat korter maken door uit te gaan van één training input. We krijgen:

$$C = -\sum_n (w[n]\ln(u[n]) + (1-w[n])\ln(1-u[n])) \quad (6.20)$$

Deze kostenfunctie is, zoals te zien, een functie van $w[n]$ (de verlangde uitvoer van het netwerk) en $u[n]$ (de eigenlijke uitvoer van het netwerk), voor alle uitvoerlaagneuronen n .

We zullen nu de partiële afgeleide van deze functie maken, naar $u[p]$ voor elke willekeurige p .

$$\frac{\partial C}{\partial u[p]} = \frac{\partial \left(-\sum_n (w[n]\ln(u[n]) + (1-w[n])\ln(1-u[n])) \right)}{\partial u[p]} \quad (6.21)$$

Toepassen van de somregel geeft (let op de minus):

$$\frac{\partial C}{\partial u[p]} = -\sum_n \frac{\partial (w[n]\ln(u[n]) + (1-w[n])\ln(1-u[n]))}{\partial u[p]} \quad (6.22)$$

Wanneer geldt $n \neq p$, zien we dat $u[n]$ en $w[n]$ constant zijn. Dan is $w[n]\ln(u[n]) + (1-w[n])\ln(1-u[n])$ ook constant en is de partiële afgeleide naar $u[p]$ dus nul. Met andere woorden, heel veel afgeleiden vallen weg en we houden over:

$$\frac{\partial C}{\partial u[p]} = -\frac{\partial (w[p]\ln(u[p]) + (1-w[p])\ln(1-u[p]))}{\partial u[p]} \quad (6.23)$$

Dit kunnen we gewoon verder uitwerken, gebruik makend van onze basis-kennis dat de afgeleide van de natuurlijke logaritme van x gelijk is aan $1/x$. We moeten wel even opletten: $\ln(x)$ bestaat alleen voor $x > 0$. In ons geval moet dus gelden dat zowel $u[p] > 0$ als $1 - u[p] > 0$. Deze twee combineren geeft dat **bij deze kostenfunctie moet gelden:** $0 < u[p] < 1$, anders gaat het mis. (Merk op dat dit bij de uitvoer van de sigmoïde-activatiefunctie het geval is!)

We gaan verder:

$$\frac{\partial C}{\partial u[p]} = -\frac{\partial (w[p]\ln(u[p]))}{\partial u[p]} - \frac{\partial ((1 - w[p])\ln(1 - u[p]))}{\partial u[p]} \quad (6.24)$$

$$\frac{\partial C}{\partial u[p]} = -\frac{w[p]}{u[p]} - \frac{1 - w[p]}{1 - u[p]} \frac{\partial (1 - u[p])}{\partial u[p]} \quad (6.25)$$

$$\frac{\partial C}{\partial u[p]} = -\frac{w[p]}{u[p]} + \frac{1 - w[p]}{1 - u[p]} \quad (6.26)$$

6.5 Sigmoïde en cross-entropy gaan goed samen

Wat blijkt is dat er mooie wiskunde ontstaat bij het samenvoegen van de sigmoïdeactivatiefunctie en de cross-entropy kostenfunctie. Deze samenhang hebben wij niet zelf bedacht en dus citeren we Nielsen (2015, Hoofdstuk 3) als bron bij deze sectie.

We gaan verder bij de laatste vergelijking waar we gebleven waren. Als het in één breuk willen hebben, moeten we de twee breuken gelijknamig maken:

$$\frac{\partial C}{\partial u[p]} = -\frac{w[p](1 - u[p])}{u[p](1 - u[p])} + \frac{u[p](1 - w[p])}{u[p](1 - u[p])} \quad (6.27)$$

$$\frac{\partial C}{\partial u[p]} = \frac{-w[p](1 - u[p]) + u[p](1 - w[p])}{u[p](1 - u[p])} \quad (6.28)$$

We zien dat we de haakjes kunnen uitwerken, en dat dan het een en ander tegen elkaar wegvalt:

$$\frac{\partial C}{\partial u[p]} = \frac{-w[p] + w[p]u[p] + u[p] - u[p]w[p]}{u[p](1 - u[p])} \quad (6.29)$$

$$\frac{\partial C}{\partial u[p]} = \frac{u[p] - w[p]}{u[p](1 - u[p])} \quad (6.30)$$

En nu komt iets moois. We gingen uit van de sigmoïdeactivatiefunctie. Dit betekent dat $u[p] = \sigma(s[L][p])$. Dit gaan we invullen:

$$\frac{\partial C}{\partial u[p]} = \frac{u[p] - w[p]}{\sigma(s[L][p])(1 - \sigma(s[L][p]))} \quad (6.31)$$

Maar we wisten reeds dat $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Dit kunnen we nu gebruiken:

$$\frac{\partial C}{\partial u[p]} = \frac{u[p] - w[p]}{\sigma'(s[L][p])} \quad (6.32)$$

Let nu goed op. We nemen nog een keer de vergelijking over die we in het backpropagation-hoofdstuk zelf hebben bewezen:

$$\frac{\partial C}{\partial b[l][n]} = \begin{cases} \frac{da[l][n]}{ds[l][n]} \cdot \sum_{p=0}^{\text{net}[l+1].\text{size}() - 1} \left(\frac{\partial C}{\partial b[l+1][p]} \cdot w[l+1][p][n] \right), & \text{als } 0 \leq l < L \\ \frac{da[l][n]}{ds[l][n]} \cdot \frac{\partial C}{\partial a[l][n]}, & \text{als } l = L \end{cases} \quad (6.33)$$

Dus bij $l = L$, oftewel, als je in de uitvoerlaag zit, geldt het volgende:

$$\frac{\partial C}{\partial b[L][n]} = \frac{da[L][n]}{ds[L][n]} \cdot \frac{\partial C}{\partial a[L][n]} \quad (6.34)$$

Met vergelijking 6.32 kunnen we het bovenstaande als volgt schrijven (bedenk daarbij dat we de sigmoïdefunctie gebruiken als activatie en dat $u[n] = a[L][n]$):

$$\frac{\partial C}{\partial b[L][n]} = \underline{\sigma'(s[L][n])} \cdot \frac{u[n] - w[n]}{\underline{\sigma'(s[L][n])}} \quad (6.35)$$

$$\frac{\partial C}{\partial b[L][n]} = u[n] - w[n] \quad (6.36)$$

Wie had gedacht, dat er bij het samenvoegen van de ingewikkeld ogende sigmoïdefunctie en de cross-entropy kostenfunctie, zo'n eenvoudige partiële afgeleide zou ontstaan? De hoop is dat de computer deze som sneller kan uitrekenen, omdat hij zo makkelijk is.

Zoals net getoond, is er dus een mooie samenhang tussen de sigmoïdeactivatiefunctie en de cross-entropy kostenfunctie.

Hoofdstuk 7

Het rekencluster van de RUG

In de eerste versie van dit profielwerkstuk, hadden wij het programma getraind op een 5x5-bord met komi-waarde 24. Zoals echter al is uitgelegd, is het gebruikelijk om Go te spelen op een groter bord, zoals 19x19, of 13x13 voor een snel potje. Toen wij echter het programma lieten trainen op een 19x19-bord, duurde dat vreselijk lang. Het leek ons dus een goed idee om te vragen of wij het krachtige rekencluster van de Rijksuniversiteit Groningen mochten gebruiken. Wij hebben een online video-meeting gehad met twee mensen van de universiteit, F. Dijkstra en H. Zilverberg. In die meeting hebben wij uitgelegd wat ons programma doet en waarom we vroegen om toegang tot het rekencluster.

Tot ons grote plezier, hebben de twee heren van de universiteit voor ons een account aangemaakt. We hebben dus toegang gekregen tot het rekencluster.

7.1 Het parallel spelen van meerdere Go-partijen

Zij hebben ons echter ook erop gewezen, dat het bij krachtigere rekenclusters belangrijk is om code parallel te kunnen draaien, en onze code was daar niet op gemaakt. Zo hadden we een aantal globale variabelen, bijvoorbeeld `board` en `potential_board`, die slechts relevant waren voor één partij Go. We konden dus niet meerdere Go-partijen tegelijk spelen.

Dit hebben we opgelost door elke gespeelde trainingspartij een ID te geven, een geheel getal; de eerste partij kreeg ID 0, de tweede partij kreeg ID 1, enzovoorts.

Vervolgens hebben we al die globale variabelen een beetje aangepast. Wat bijvoorbeeld eerst de variabele `potential_board` was, is nu een vector geworden, gevuld met ‘oude’ `potential_board`-eenheden, die corresponderen

met de ID van een partij. Hetzelfde hebben wij gedaan bij de andere globale variabelen waarvoor dat nodig was.

In de oude situatie, de situatie van de eerste versie van dit profielwerkstuk, speelde het programma dus één partij en dan keek het naar de globale variabelen `board`, `potential_board`, `visited`, et cetera. In de nieuwe situatie speelt het programma een aantal partijen tegelijkertijd. Stel dat die partijen ID's 0, 1, 2 en 3 hebben. Voor de gespeelde partij met ID 2, wordt er gebruik gemaakt van `board[2]`, `potential_board[2]`, `visited[2]`, et cetera. Voor de gespeelde partij met ID 3, wordt er gebruik gemaakt van `board[3]`, `potential_board[3]`, `visited[3]`, et cetera. Zodoende kunnen er in de nieuwe codesituatie meerdere partijen tegelijk gespeeld worden, omdat ze gebruik maken van hun eigen variabelen, en niet dezelfde variabelen hoeven te delen met andere tegelijk gespeelde partijen.

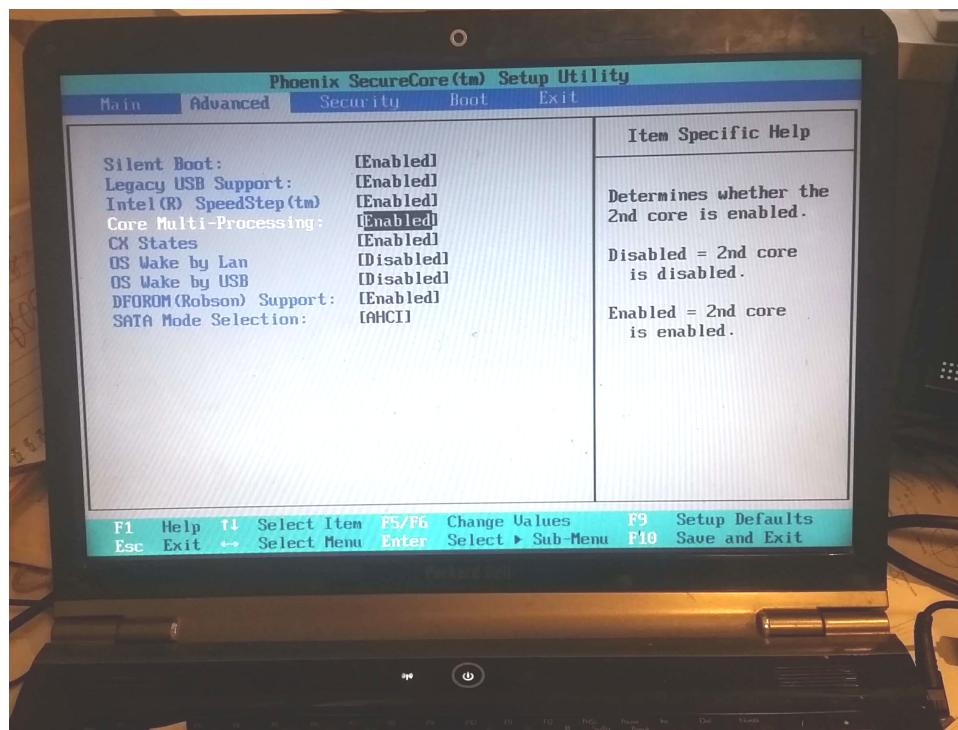


Foto ter illustratie. Arons programmeerlaptop is zo oud dat er in de BIOS nog een optie te vinden is om parallel rekenen uit te schakelen.

7.2 Toch werkt het niet...

We hadden het zo leuk bedacht, die parallelisatie, maar: we kregen foutmeldingen. Van deze foutmeldingen begrepen we niet zo heel veel.

Voorbeelden van foutmeldingen die we kregen:

De onleesbare brei aan karakters boven de foutmelding, is de gewone, gewenste output van ons programma. We hadden hem al die karakters laten outputten om te debuggen, met het doel om de foutmeldingen op te lossen, en dat zie je in het screenshot terug.

Ons viel ook op dat we niet altijd dezelfde foutmelding kregen. Dit was een andere foutmelding:

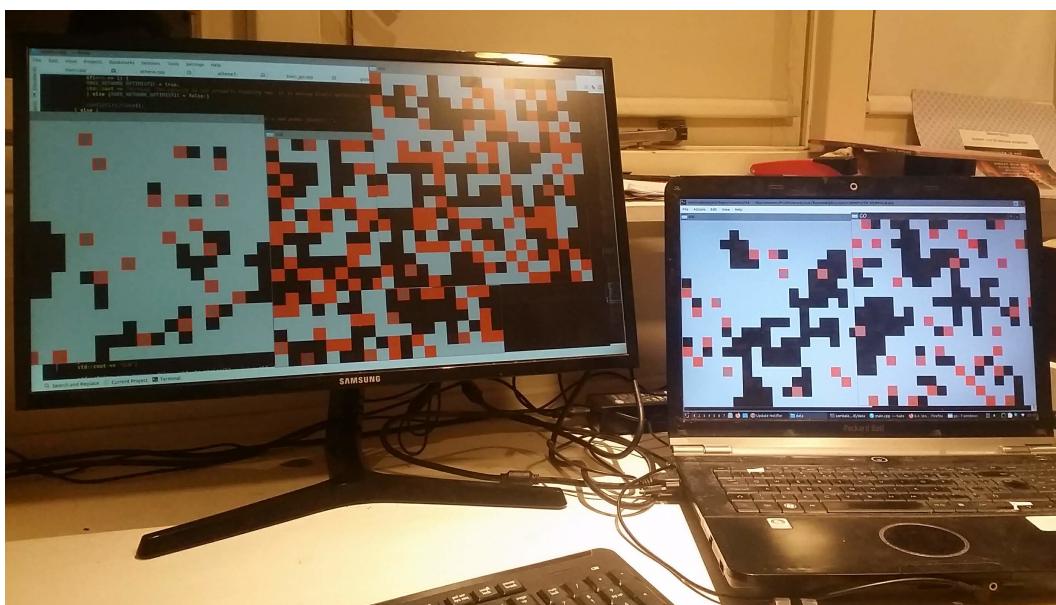
```

b6b6a11b11a17b17a6b6a6b6a11b11a6b11b11a6b6a11a6b6a11a6b6a11b11a6b6a6
1a6b6a11b11a6b6a1b1a6b6a6b6a11a11b11a6b6a6b6a11a6b6a11b11a6b6a6
a6b6a6b6a11b11a6b6a11b11a6b6a1b1a11b11a1b1a6b6a11b11a6b6a6b
b11a6a11b11b6a6b6a6b6a11b11a11b11a6b6a6b6a11b11a6b6a6b6a11b11a6b6
b6a11b11a6b6a11b11a6b6a1b1a6b6a1b1a6b6a11b11a6b6a11b11a6b6a6
Finished playing game 6-7double free or corruption (fasttop)
Aborted (core dumped)
gotratner@gotratner:~/PWS/NIEUWER/data$ 

```

Bovendien viel het ons op, dat de foutmelding slechts ‘soms’/incidenteel leek te ontstaan. Dat wil zeggen: soms konden we het programma runnen en crashte het meteen; soms deed hij het eerst (vermoedelijk zelfs naar behoren) en duurde het wat langer totdat het crashte. Er kwam uiteindelijk helaas wel altijd zo’n crash.

Tenslotte merkten wij, dat deze foutmelding zich niet voordeed als we parallelisatie uit hadden staan. Dit suggereerde dat de fout echt in de parallelisatie zat.



Vijf partijen worden gespeeld. Eigen foto

We hebben voor het ontstaan van de fout een verklaring bedacht. Deze is wel enigszins vaag, omdat we zelf natuurlijk ook niet snapten waar deze bug vandaan kwam. Onze hypothetische verklaring luidde als volgt (wij hebben geprobeerd het zo duidelijk mogelijk te formuleren):

“Bij het paralleliseren worden er veel globale variabelen ‘gekopieerd’ door

er honderd versies van in een vector te zetten die vervolgens allemaal hun eigen (verschillende) ding doen. Wat eerst bijvoorbeeld de driedimensionale vector `former_positions` was, is nu een vierdimensionale vector, bestaande uit driedimensionale vectoren die allemaal hetzelfde doen als de oorspronkelijke driedimensionale vector `former_positions`.”

Aan de oorspronkelijke vector werd wel eens een tweedimensionale vector toegevoegd door middel van de `push_back()`-functie. Dit betekent dat nu aan iedere driedimensionale component van de vierdimensionale vector, wel eens een tweedimensionale vector wordt toegevoegd. Als thread 2 zit te werken aan `former_positions[2]` en thread 17 aan `former_positions[17]`, en als aan `former_positions[2]` een tweedimensionale vector als component wordt toegevoegd, moet er wellicht incidenteel geheugen worden verplaatst (als de huidige geheugenplek vol zit). Dat mogelijke verplaatsen is zonder parallelisatie kennelijk geen probleem, anders zou het programma ook zonder parallelisatie wel crashen.

Wellicht is de oorzaak van het probleem het volgende: als een thread probeert geheugen te lezen of te schrijven terwijl een andere thread net bezig is met het verplaatsen van geheugen, dan is het natuurlijk niet mogelijk om het verplaatst wordende geheugen te lezen of ernaar te schrijven. Dit zou wel eens de oorzaak kunnen zijn van de errors die wij niet begrijpen.

7.3 Een oplossing?

Toen wij deze mogelijke verklaring neerschreven in dit document, wisten wij nog niet of dat wel de werkelijke oorzaak was. We zijn na het opschrijven van het bovenstaande, dus gaan programmeren om te zien of het zou werken.

Wat wij hebben gedaan is als volgt:

Wat eerst bijvoorbeeld de vierdimensionale

```
std::vector<std::vector<std::vector<std::vector<int>>> former_positions  
was, of, in andere ‘woorden’, wat eerst  
std::vector<vvvi> former_positions  
was, hebben we veranderd in  
std::vector<vvvi*> former_positions.
```

Dit hebben we ook gedaan voor een aantal andere variabelen waarbij dit nodig werd geacht, zoals `board` en `potential_board`.

In plaats van een vector van `vvvi`’s, waarvan wij dachten dat die een niet-constante grootte hadden, hebben we nu dus een vector van *pointers naar `vvvi`’s*. Het leek ons namelijk een logische gedachte dat pointers niet veel van grootte kunnen veranderen. Ze wijzen (pointen) leuk naar een wellicht van grootte veranderende `vvvi`, maar zelf veranderen pointers niet van grootte,

dachten wij. Een vector van een constant aantal pointers zou dan gewoon stabiel zijn, zonder dat er steeds geheugen wordt verplaatst.

Nogmaals: wij zijn geen “die-hard” C++-programmeurs en daarom is deze verklaring slechts hypothetisch. Of hij C++-technisch klopt, wisten en weten we niet zeker. We hebben het dus maar gewoon zo geprogrammeerd, in de hoop dat het zou werken.

Dit had de consequentie dat er in de code rekening moest worden gehouden met de pointers. Als er eerst `former_positions[3][4][0][0]` in de code stond, moest dat veranderd worden in `(*former_positions[3])[4][0][0]`. Volgens dit principe hebben we dus de code veranderd.

We hoopten dat dit zou werken, maar tot onze grote teleurstelling bleven de foutmeldingen komen.



Een gebouw op het Zernike-universiteitscomplex in Groningen. Eigen foto

7.4 Driemaal is scheepsrecht

Nu volgt een kort en enigszins anekdotisch, doch waargebeurd verhaaltje. Van ons tweeën was Aron degene die zich bezighield met het paralleliseren. Nadat hij dus twee keer een poging had gedaan om parallelisatie te

implementeren, en terwijl de deadline¹ naderde, had Aron besloten de parallelisatie dan maar te laten liggen. Raf was hiermee al akkoord gegaan. Wij vonden dit ontzettend jammer, want we hadden nu toegang tot een rekencluster, maar zonder parallelisatie konden we niet volledig gebruik maken van de kracht van dat cluster.

Even later kwam Aron opeens toch op een idee waarmee het paralleliseren wellicht zou lukken.

Dit idee was nogal anders dan de vorige methodes, en daarom is hij het snel gaan implementeren om te zien of dat het waard zou zijn.

Het idee was als volgt.

In plaats van het runnen van één instantie van het programma, moest er één ‘primaire’ instantie van het programma komen, die op zijn beurt een met het aantal threads overeenkomend aantal ‘secundaire’ instanties zou openen. Het programma moest zichzelf dus aanroepen. Wij riepen zelf via de terminal de primaire instantie op via `./go`. Als de primaire instantie vervolgens weer een aantal keer parallel `./go` zou oproepen, zou dit zich oneindig diep herhalen. We hebben dus een manier bedacht om aan een proces te laten weten dat het een secundaire instantie is, en die manier is met programmaparameters/-argumenten. De primaire instantie werd nog steeds aangeroepen met `./go`, maar de secundaire instanties werden door de primaire instantie aangeroepen met het commando `./go thread X`, waarbij X het nummer van de betreffende thread was. Dit moest dan voor elke thread, en het moest ook nog parallel. Dit laatste hebben we gedaan door gebruik te maken van `std::thread`, als volgt:

```
std::vector<std::thread> thread_vec;
for(int i = 0; i < NUMBER_OF_THREADS; i++) {
    thread_vec.push_back(std::thread(run_go, i));
}
for(int i = 0; i < NUMBER_OF_THREADS; i++) {
    thread_vec[i].join();
}
```

We hadden meerdere threads (het aantal was zelf instelbaar), en dus was er een vector van `std::threads` nodig. Onderaan moeten de threads weer joinen, oftewel samenkommen. Overigens werd alleen het partijen spelen geparalleliseerd. Het automatisch spelen van partijen kostte namelijk extreem veel tijd, zeker op een 19x19-bord, voor ons gevoel bijvoorbeeld 1,5 seconden per partij (maar het varieert en is ook afhankelijk van de netwerkafmetingen). Dat is veel tijd, omdat het programma natuurlijk vél meer dan één partij

¹Vanwege persoonlijke omstandigheden van onze begeleider, is onze deadline uiteindelijk later geworden, maar dat wisten we toen nog niet.

moest spelen. De andere stap, het aanpassen van de gewichten en biasen, trainend met behulp van de zojuist gespeelde partijen, kostte natuurlijk ook veel tijd, maar het leek ons beter eerst het partijen spelen te paralleliseren.

In de bovenstaande code worden threads gemaakt die `run_go` uitvoeren. `run_go` is de volgende functie:

```
void run_go(int thrnum) {
    std::system((std::string("./go thread ") + std::to_string(thrnum)).c_str());
    //run this program with parameters
}
```

Dit is de functie die verantwoordelijk is voor het aanroepen van de secundaire go-instanties met de systeemcommando's `./go thread X`.

Er is nog meer veranderd in de code. Zo heeft elke thread nu een eigen map waarin hij zijn eigen bestanden schrijft. Het neurale netwerk blijft wel thread-onafhankelijk, maar de gespeelde partijen uiteraard niet; elke thread speelt zijn eigen partijen, maar allemaal met hetzelfde neurale netwerk, dat traint met de partijen van *alle* threads.

Zoals de naam van deze sectie al zei, is driemaal scheepsrecht. We hebben dit geïmplementeerd en, tot ons geluk, werkte het!

7.5 Meer cores

Wij zijn er toen mee gaan experimenteren. Gebruikten we één thread, dan werkte het. Gebruikten we vier threads, dan ging het partijen spelen op gevoel ook ongeveer vier keer zo snel als met één thread. Gebruikten we echter vijftig threads, dan konden we ontzettend goed merken dat het partijen spelen niet vijftig keer zo snel ging als met één thread. Dit deed vermoeden dat we tegen de parallelisatiegrens van de hardware aanliepen. Toen parallelisatie nog niet werkte maar we het al wel hadden geprobeerd, zette OpenMP (de parallelisatiemethode die we toen gebruikten) automatisch het aantal threads op 4. Maar ook nu parallelisatie wel werkte, probeerden wij eens de process viewer `htop` op het rekencluster te openen. Wij kwamen erachter dat deze was geïnstalleerd, en ook `htop` gaf iets aan dat erop leek dat we 'maar' 4 cores hadden. Dit deed ons vermoeden dat aan ons een deel van het cluster was toegewezen met 4 cores. Maar nu we multithreading eindelijk werkend hadden gekregen, leek het ons ook wel leuk om wat meer cores te mogen gebruiken, zodat we meer threads konden hebben en dus meer dingen parallel konden laten lopen.

Wij hebben toen de heren van de Rijksuniversiteit Groningen benaderd met de vraag of wij wellicht een groter deel van het rekencluster mochten gebruiken. Dit mocht, en hier waren we zeer blij mee.

Zij hebben ons account opgeschaald tot 32 cores. Het aanroepen van `lscpu` op het rekencluster gaf de volgende informatie: 32 CPU's, 32 sockets, 1 core per socket en 1 thread per core. Toen we dit zagen, leek het ons vrij duidelijk dat we ook 32 threads konden aanmaken op het rekencluster, die allemaal parallel zouden draaien.

Met deze **32 threads op het rekencluster van de Rijksuniversiteit Groningen** zijn we heel blij. Ter vergelijking: Arons (minstens 10 jaar oude) programmeerlaptop heeft volgens `lscpu` maar 2 threads. De in november 2021 aangeschafte werkcomputer (een middenklasse-laptop) van zijn vader heeft er 6, en zijn vorige werkcomputer 8. Daarentegen is 32 threads veel meer. En waar we helemaal trots op zijn, is dat we al deze threads ook daadwerkelijk hebben gebruikt, door middel van het implementeren van parallelisatie van onze C++-code. Wij konden dus maar liefst 32 partijen tegelijk laten spelen. We kunnen dus wel stellen dat we met het rekencluster meer hebben kunnen doen dan met een gemiddelde laptop. We zijn hier echt trots op.



Het prachtige industrieterrein Eemspoort in Groningen. Ergens op dit industrieterrein stond het (fysieke) deel van het rekencluster waarop onze berekeningen zijn uitgevoerd. Dit was op de Merlin-cloud. Eigen foto

Hoofdstuk 8

Meer halen uit een neurale netwerk: vooruitkijken

Uiteindelijk, terwijl we bezig waren met dit profielwerkstuk, hadden we een aantal goedwerkende neurale netwerken getraind. ‘Goedwerkend’ betekent in dit geval dat het neurale netwerk bijvoorbeeld in 99% van de partijen wint, tegen een smartened random. We kregen de netwerken niet eenvoudig nog beter getraind. We vroegen ons dus af: kunnen we de speelprestaties niet op een andere manier nog beter krijgen?

Wij dachten hierbij aan dieper vooruitkijken. Om dit te kunnen uitleggen, moeten we echter eerst uitleggen wat het programma normaal gesproken zou doen, volgens de standaardmanier, en de manier waar het netwerk mee is getraind.

8.1 De standaardmanier om met het netwerk een zet te bedenken

In de situatie eerst werd als volgt bedacht welke zet het netwerk zou spelen: het programma kijkt voor elke zet wat de kwaliteit van die zet volgens het neurale netwerk is, en kiest zodoende de zet met de hoogste evaluatie.

Hiervoor hadden we de volgende lambdafunctie bedacht:

```
auto choose_black_best_move = [&player, &choice_y, &choice_x, &max_eval]
<class Function, class ... Types>(Function evaluation_function, Types ... Args) {
    //is 'smartened' //for BLACK!
    auto leg_moves = all_legal_moves(1); //all legal moves of BLACK
    double cab = (double)count_area_black();
    double caw = (double)count_area_white();
    if((int)leg_moves.size() == 1 || (cab > caw + komi && last_move_was_a_pass)) {
```

```

choice_y = -1;
choice_x = -1;
potential_board = board;
int playernow = 1; //BLACK!!
update_potential_board(-1,-1, playernow);
max_eval = evaluation_function(Args...);

} else
{
    max_eval = -std::numeric_limits<double>::infinity();
    int corresponding_index = -1;
    int i = 0;
    if(cab < caw + komi) {
        i = 1;
    //skip the option of passing a move: that would make the network lose.
    }
    for(; i < (int)leg_moves.size(); i++) {
        auto board_first = board;
        potential_board = board;
        int playernow = 1; //BLACK!!!
        update_potential_board(std::get<0>(leg_moves[i]),
                               std::get<1>(leg_moves[i]), playernow);
        board = potential_board;
        double eval = evaluation_function(Args...);
        if(eval >= max_eval) {
            max_eval = eval;
            corresponding_index = i;
        }
        board = board_first;
        potential_board = board_first;
    }
    choice_y = std::get<0>(leg_moves[corresponding_index]);
    choice_x = std::get<1>(leg_moves[corresponding_index]);
}
};


```

Dit is een heel bijzondere lambdafunctie. Het is namelijk een lambdafunctie met een normale template parameter (`class Function`) en dan ook nog een template parameter pack, namelijk `class ... Types`. Dit maakt het mogelijk om als argumenten aan deze functie (`choose_black_best_move`), een andere functie te geven, namelijk de `Function evaluation_function`, en de argumenten voor die `evaluation_function`, namelijk `Types ... Args`. De lambdafunctie `choose_black_best_move` roept dan intern de functie `evaluation_function` aan, met de argumenten die `choose_black_best_move` verder heeft ontvangen. Dit gebeurt door de regel `double eval = evaluation_function(Args...);`.

Dit klinkt allemaal best ingewikkeld en dat is het ook; we hebben zelf erg lang over dit principe moeten nadenken. Maar het werkt wel en het was een zeer goede oefening in C++ voor ons.

Ter verduidelijking: de (lambda)functie `choose_black_best_move` kijkt in het algemeen bij een bepaalde stelling waar zwart aan de beurt is voor elke zet die zwart kan doen, hoe goed die zet is, en kiest dan de beste. Dit kijken ‘hoe goed een zet is’, gebeurt door een functie die je zelf bedenkt.

Voor de oplettende lezer: de functie `choose_black_best_move` doet, zoals te zien in de code, nog iets. Hij is ‘smartened’, staat in het commentaar achter een dubbele slash. Dit houdt in dat hij, als hij kan winnen door nu een zet over te slaan, ook een zet over zal slaan, zonder enige andere zet te overwegen. Een overgeslagen zet wordt in ons programma aangegeven met de ‘coördinaten’ $(-1, -1)$. Je ziet dit terug in de code als volgt: als `cab > caw + komi`, met andere woorden, als zwart qua oppervlakte gewonnen staat, en (`&&`) als wit de laatste zet ook al had overgeslagen (`last_move_was_a_pass`), dan worden `choice_y` en `choice_x` gezet tot -1 , want als zwart dan een zet overslaat, zal hij gegarandeerd winnen. Dit is vrij bovenaan de code. Overigens staat er nog iets binnen de if: `(int)leg_moves.size() == 1`. Dit houdt in dat er hetzelfde gebeurt wanneer er slechts één zet mogelijk is (dat is dan namelijk automatisch de ‘zet’ om een zet over te slaan).

Het feit dat deze functie ‘smartened’ is, betekent nog iets. In de functie stond namelijk het volgende stukje code:

```
int i = 0;
if(cab < caw + komi) {
    i = 1; //skip the option of passing a move: that would make the network lose.
}
for(; i < (int)leg_moves.size(); i++) {
    ....
```

Oftewel: vlak voordat je alle mogelijke zetten begint te doorlopen (de for-loop), gaat zwart, in het geval dat hij qua oppervlakte verloren staat, de waarde van de variabele `i` veranderen naar 1 . De functie die alle legale zetten opsomt, is namelijk zo geprogrammeerd dat de zet $(-1, -1)$ altijd als eerste in het rijtje komt, dus bij index 0 . Dit betekent dat de for-loop beginnen bij `i==1`, gelijkstaat aan het niet overwegen van de mogelijkheid om een zet over te slaan. Immers, als zwart in de situatie dat hij qua bordoppervlakte verloren staat, een zet zou overslaan, zou wit dat natuurlijk ook doen, en dan verliest zwart, want dan zijn er twee zetten achter elkaar overgeslagen. Daarom is het slim om in die situatie niet eens het neurale netwerk te laten overwegen om een zet over te slaan.

We keren weer terug naar ons oorspronkelijke verhaal. De meest basale manier om een zet te bedenken, is om `choose_black_best_move` aan te roepen met als doorgegeven functie `athena_output_on_given_position`, de functie die domweg kijkt wat het neurale netwerk (‘Athena’) als evaluatie aan de stelling (inclusief gespeelde zet) geeft.

```
<.....> else if(player_input_mode == "a") { //athena that plays black (internally)
    choose_black_best_move(athena_output_on_given_position, false);
}
```

De bovenste regel betekent hier: “Als het neurale netwerk (a) aan de beurt is om een zet te doen”. Dus als het neurale netwerk aan de beurt is, wordt de functie `choose_black_best_move` aangeroepen, met als eerste argument de naam van de functie die de evaluatie moet bepalen; in dit geval is dat `athena_output_on_given_position`. Het `false` wordt vervolgens weer doorgegeven als argument aan de functie `athena_output_on_given_position`; dit geeft aan dat hier de normale Athena moet worden gebruikt, en niet de fixed athena, een ander programmeerbaar waar we nu niet dieper op ingaan.

Er wordt in de standaardsituatie, overigens ook de situatie waarmee het netwerk wordt getraind, dus telkens als het netwerk aan de beurt is, een zet bedacht door van alle direct mogelijke zetten met het netwerk de evaluatie te bepalen, en de zet te spelen met de hoogste evaluatie.

Dit was de standaardsituatie. Je kijkt hier dus maar één zet vooruit. We hebben vervolgens iets bedacht om het algoritme nog slimmer te maken, en dat is door verder vooruit te kijken.

8.2 Verder vooruitkijken

Je kunt je afvragen: waarom zo moeilijk doen, met een (lambda)functie met template parameter pack en een functie als argument?

Nou, het enorme voordeel dat dit brengt, is dat we nu gewoon een andere evaluatiefunctie kunnen bedenken die iets verder vooruitkijkt, en dat we die aan `choose_black_best_move` kunnen geven als argument. We hoeven dan geen twee functies `choose_black_best_move` te hebben, maar kunnen voor alle doeleinden altijd dezelfde functie `choose_black_best_move` blijven gebruiken, alleen met andere argumenten. Dit gaat het probleem van copy-pasten tegen. Als je namelijk teveel copy-pastet, loop je het risico dat je bepaalde kopieën niet meer gebruikt, waardoor je alleen de nieuwe kopieën blijft aanpassen. De oude kopieën raken dan in de vergetelheid, en na verloop van tijd heb je er niks meer aan. Door het maken van een generieke functie `choose_black_best_move`, lopen we niet tegen dit probleem aan.

In de zojuist beschreven standaardsituatie maakten we gebruik van de coderegel

```
choose_black_best_move(athena_output_on_given_position, false);
```

In de nieuwere situatie hebben we daarop de volgende variant bedacht:

```
choose_black_best_move(evaluation_now_look_half_move_deep,
    (last_move_was_a_pass ? 1 : 0));
```

Hierbij is `evaluation_now_look_half_move_deep` de volgende functie:

```
//this uses fixed athena
//consec_passes is the number of consecutive passes
//that have occurred, including the just-played move by black
//when this function is called, black has played the last move.
//Now, for every white move, it is evaluated with fixed athena
//how white thinks about the position. The maximum gainable
//evaluation for white can thus be calculated. Good for white
//is bad for black, so the MINUS whites_max_eval is returned.
//The return value of this function therefore as usual is an
//evaluation of quality of the move that black just played,
//seen from black perspective.
double evaluation_now_look_half_move_deep(int consec_passes) {
    potential_board = board;
    if(consec_passes == 2) {
        if((double)count_area_black() < (double)count_area_white() + komi) {
            return -std::numeric_limits<double>::infinity();
        } else {
            return std::numeric_limits<double>::infinity();
        }
    }
    if(consec_passes == 1 && ((double)count_area_black() <
        (double)count_area_white() + komi)) {
        //white will also pass and win. black's evaluation is minus infinity
        return -std::numeric_limits<double>::infinity();
    }
    auto leg_movesw = all_legal_moves(2); //all legal moves of WHITE
    double whites_max_eval = -std::numeric_limits<double>::infinity();
    //white tries to make blacks evaluation as small as
    //possible because white wants that black loses
    for(int i = 0; i < (int)leg_movesw.size(); i++) {
        auto potential_board_first = potential_board;
        update_potential_board(std::get<0>(leg_movesw[i]),
        std::get<1>(leg_movesw[i]), 2); //WHITE!
        auto consec_passes_first = consec_passes;
        board = potential_board;
        if(std::get<0>(leg_movesw[i]) == -1) {
            consec_passes++;
        } else {
            consec_passes = 0;
        }

        double whites_eval;
        if(consec_passes == 2) { //white has passed
            if((double)count_area_black() < (double)count_area_white() + komi) {
```

```

        whites_eval = std::numeric_limits<double>::infinity();
    } else {
        whites_eval = -std::numeric_limits<double>::infinity();
    }
} else {
    //one move of white is now done. Now look at
//evaluation of fixed athena, white athena.
    invert_colors();
    whites_eval = athena_output_on_given_position(true);
    invert_colors();
}
if(whites_max_eval < whites_eval) whites_max_eval = whites_eval;
board = potential_board_first;
potential_board = potential_board_first;
consec_passes = consec_passes_first;
}

return -whites_max_eval;
}

```

De bovenstaande code is in de kern heel simpel: het berekent voor alle mogelijke witte zetten de netwerkevaluatie. Hieronder is de functie opnieuw opgenomen, maar nu alleen met de belangrijkste punten en deels in pseudocode:

```

double evaluation_now_look_half_move_deep(int consec_passes) {
    auto leg_movesw = all_legal_moves(2); //all legal moves of WHITE
    double whites_max_eval = -std::numeric_limits<double>::infinity();
    for(int i = 0; i < (int)leg_movesw.size(); i++) {
        <<Speel de zet leg_movesw[i]>>
        invert_colors();
        double whites_eval = athena_output_on_given_position(true);
        invert_colors();
        if(whites_max_eval < whites_eval) whites_max_eval = whites_eval;
        <<Maak de zojuist gespeelde zet leg_movesw[i] ongedaan>>
    }
    return -whites_max_eval;
}

```

Allereerst: let goed op. Het bovenstaande codefragment komt **niet** precies zo voor in onze code, omdat het versimpeld is. Dit kortere codefragment is niet volledig en zal dus niet goed functioneren, maar bevat wel het belangrijkste principe van deze functie. Laten we er dus eens kort iets dieper op ingaan.

Bovenin de code wordt de functie `all_legal_moves` aangeroepen met als argument 2; dit geeft aan dat wit aan de beurt is. De returnwaarde van deze functie, een vector van coördinatenparen, wordt geschreven naar de variabele

`leg_movesw`. Vervolgens wordt er voor wit een ‘maximale’ evaluatie gesteld op minus oneindig. Nu komt het programma in een for-loop terecht. Voor elke zet die wit kan spelen, wordt de evaluatie `whites_eval` berekend volgens het neurale netwerk. Is die evaluatie groter dan de ‘maximale’ evaluatie tot nu toe, dan wordt de maximale evaluatie aangepast naar boven. Dit is de evaluatie die wit maximaal verwacht als hij in de gegeven stelling de beste zet zou spelen. Wanneer alle van de voor wit mogelijke zetten bij langs zijn gegaan, is de maximale evaluatie die wit verwacht, uitgerekend. Wit weet dan wat zijn evaluatie zou zijn als hij in de gegeven stelling de beste zet (volgens zichzelf) zou doen.

Maar, even terug: dit was bij een gegeven zwarte zet. Ons idee was dat we dit hele proces zouden uitvoeren voor elke zwarte zet en zodoende bedenken wat voor zwart de beste zet is. We hebben nu voor wit een evaluatie berekend, maar wat voor wit goed is, is voor zwart juist slecht, en andersom. Daarom luidt de laatste regel van deze functie

```
return -whites_max_eval;
```

De *negatieve* maximale evaluatie voor wit wordt dus gereturnd als evaluatie voor zwart, na de gespeelde gegeven zet van zwart. Dit is logisch, volgens het principe ‘goed voor wit is slecht voor zwart’. Dit is hoe deze functie werkt. Voor het totaalbeeld: de functie `evaluation_now_look_half_move_deep` is vergelijkbaar met de functie `athena_output_on_given_position(false)`. Deze functies worden beide aangeroepen wanneer er een zojuist gespeelde potentiële zwarte zet moet worden geëvalueerd. Beide functies returnen dan de evaluatie - vanuit zwart perspectief (hoger is beter) - van deze zet.

Beide functies doen dus hetzelfde. In de oudere situatie, de standaardsituatie, werd alleen de functie `athena_output_on_given_position(false)` gebruikt. Deze deed zijn taak. In de nieuwere, modernere en hopelijk betere situatie, kan nu als vervanging ook de functie `evaluation_now_look_half_move_deep` worden gebruikt. Dit is een verbetering ten opzichte van de standaardsituatie, omdat de laatstgenoemde functie nog een zet meer vooruitkijkt en dan pas het neurale netwerk raadpleegt, terwijl `athena_output_on_given_position(false)` niet verder vooruitkijkt, maar direct het neurale netwerk raadpleegt.

En wie verder vooruitkijkt, ziet uiteraard meer. Zo hebben wij geprobeerd ons algoritme nog sterker te maken.

Voor de die-hards: in de bovenstaande code kwam ook nog tweemaal de regel `invert_colors()`; voor. Deze functie draait de kleuren van alle stenen op het bord om. Hier hebben we het tot nog toe niet over gehad. In de volgende subsectie staat uitgelegd waar dit voor is.

8.2.1 Komi-breinbrekers

Deze functie `evaluation_now_look_half_move_deep` wordt aangeroepen door `choose_black_best_move`, op het moment dat zwart de laatste zet gespeeld heeft. Houd in herinnering dat - tot nu toe - **het netwerk altijd zwart speelde**. Dit is eerder uitgelegd, namelijk in hoofdstuk 4.9. Als het netwerk - zwart - nu dus een zet heeft gespeeld, zou in de normale situatie op dit moment de functie `athena_output_on_given_position` worden aangeroepen. Als we hier met positieve komi spelen, wordt op dat moment dus het netwerk geraadpleegd dat is getraind om zwart te spelen bij positieve komi. Hetzelfde geldt bij een negatieve komi.

Nu echter, plakken we direct na de zet van zwart, er ook nog een zet van wit achteraan; we zijn immers verder aan het vooruitkijken. Als zwart speelt met positieve komi, en als we willen dat een neuraal netwerk een door wit gespeelde zet evalueert, dan moet dat netwerk dus getraind zijn met *negatieve komi*. Dat netwerk denkt natuurlijk dat hij zwart is, maar hier speelt het weer wit. De kleuren van het bord moeten dus ook worden omgedraaid, voor én na het berekenen van de evaluatie. Dit hele idee is extreem tricky.

Het is zelfs nog iets gecompliceerder als we met zwart aan het spelen zijn met negatieve komi: het algoritme lijkt dan voor de gebruiker wit te spelen, maar het speelt intern natuurlijk altijd zwart. De coderegel

```
choose_black_best_move(evaluation_now_look_half_move_deep,  
    (last_move_was_a_pass ? 1 : 0));
```

gaat dan alle zetten voor (intern) zwart bij langs en roept, nadat het zo'n zet van zwart gespeeld heeft, de functie `evaluation_now_look_half_move_deep` op, die op zijn beurt weer alle witte zetten bij langs speurt na de gespeelde zwarte zet. Omdat zwart hier gespeeld wordt door een netwerk met negatieve komi, moeten alle door wit gespeelde zetten worden geëvalueerd door een netwerk met positieve komi. Wederom denkt dit netwerk juist dat het zwart is, waardoor het nodig is om voor én na de evaluatie de kleuren van het bord om te draaien.

Dit zijn moeilijke denkstappen; er was grote zorgvuldigheid nodig bij het nadenken hierover.

Wat we echter zagen, is het volgende patroon. Als we met een bepaalde (positieve respectievelijk negatieve) komi speelden, was er voor de functie `evaluation_now_look_half_move_deep` juist een netwerk nodig met de komi met het omgedraaide teken (negatief respectievelijk positief). En ongeacht de komi-waardes, was het altijd nodig om twee keer de kleuren van de stenen op het bord om te wisselen.

Dit netwerk met de komi met het omgedraaide teken, noemen we *Fixed Athena*. Deze naam heeft weinig te maken met wat het doet, maar om historische redenen¹ heet het toch zo.

8.2.2 Geen goede resultaten

Toen wij het bovenstaande hadden geprogrammeerd, viel ons iets erg vreemds op: het algoritme leek nauwelijks sterker te worden, en in sommige gevallen zelfs minder sterk! Eerst wisten we niet hoe dit kon komen, maar al vrij snel hadden we een hypothese geformuleerd. De oorzaak van het probleem leek ons het volgende:

Je gebruikt de Fixed Athena (FA) in situaties waar zwart de laatste zet heeft gespeeld en wit in het vooruitkijkalgoritme de FA is. De FA controleert de kwaliteit van een door wit gespeelde zet. Maar omdat de partij tot dusverre tot stand is gekomen door het algoritme (zwart) tegen een smartened random (wit) te laten spelen, zal zwart waarschijnlijk aan het winnen zijn. Het FA-netwerk moet dan dus een zet controleren in een stelling die - vanuit de FA gezien - waarschijnlijk tot verlies zal leiden. Maar de FA is zelf ook een slim algoritme: toen de FA zelf getraind werd, won hij ook een groot percentage van de partijen, en onder die omstandigheden is de FA getraind! Het probleem is volgens ons dat de FA is getraind om goed te presteren *in situaties waarin hij gewonnen staat*, maar dat hij in deze vooruitkijksituatie juist ingezet wordt wanneer hij verloren staat. Op verliezende situaties is de FA niet getraind en daarom zal de evaluatie ook wel wat slordiger zijn.

Dit was volgens ons het probleem, waardoor het inzetten van FA bij vooruitkijken niet tot verbetering van spelkwaliteit leidde.

8.3 Recursief nóg verder vooruitkijken zonder Fixed Athena

Met de zojuist uitgelegde codefuncties, hadden we het programma nu in staat gesteld om één zet verder vooruit te kijken dan normaal. Maar hoe verder je vooruitkijkt, hoe beter de zetten van het programma zouden moeten zijn, dus daarom hebben we ook een C++-functie gemaakt die recursief is; deze roept zichzelf aan. Hierdoor kan het programma in theorie zo ver vooruitkijken als wij maar willen. (In de praktijk echter wordt het aantal mogelijke stellingen

¹Deze Fixed Athena was oorspronkelijk geïmplementeerd met een ander doeinde, waar de naam wel bij paste. In onze code is dit principe echter Fixed Athena blijven heten, en dus hebben we die naam ook voor in dit document maar behouden.

al gauw veel te groot om binnen redelijke tijd te evalueren, dus veel zetten vooruitkijken lukt niet.)

Dit berust op het volgende principe. Als zwart altijd de zet pakt met de hoogste evaluatie, moet wit de zet pakken met de laagste evaluatie. Immers, wit wil dat de evaluatie van zwart zo laag mogelijk is. De evaluatie van een zet wordt als volgt bepaald:

- Als we vanaf de gegeven stelling vooruit willen kijken:
 - De evaluatie van de gegeven stelling is gelijk aan de evaluatie van de stelling nadat de beste zet is gespeeld. De beste zet wordt bepaald door na te gaan welke zet de stelling met de hoogste/laagste* evaluatie genereert.
- Als we vanaf de gegeven stelling **niet** nog verder vooruit willen kijken:
 - De evaluatie van de gegeven stelling is gelijk aan de evaluatie die het neurale netwerk aan de stelling geeft.

* *Dit is afhankelijk van welke speler er aan de beurt is.*

Zodoende hebben wij de volgende functie geïmplementeerd:

```
//consec_passes is the number of consecutive passes
//that have occurred, including the just-played move by black
double tree_evaluation_after_black_move(bool initial_func_call,
                                         int depth, int consec_passes) {
    if(initial_func_call) potential_board = board;
    if(consec_passes == 2) {
        if((double)count_area_black() < (double)count_area_white() + komi) {
            return -std::numeric_limits<double>::infinity();
        } else {
            return std::numeric_limits<double>::infinity();
        }
    }
    if(consec_passes == 1 && ((double)count_area_black()
        < (double)count_area_white() + komi)) {
        //white will also pass and win. black's evaluation is minus infinity
        return -std::numeric_limits<double>::infinity();
    }
    auto leg_movesw = all_legal_moves(2); //all legal moves of WHITE
    double min_eval = std::numeric_limits<double>::infinity();
    //white tries to make blacks evaluation as small as possible
    //because white wants that black loses

    /*if(leg_movesw.size() > 13) {
        return athena_output_on_given_position(false);
   }*/
}
```

```

for(int i = 0; i < (int)leg_movesw.size(); i++) {
    auto potential_board_first = potential_board;
    update_potential_board(std::get<0>(leg_movesw[i]),
                           std::get<1>(leg_movesw[i]), 2); //WHITE!
    auto consec_passes_first = consec_passes;
    board = potential_board;
    if(std::get<0>(leg_movesw[i]) == -1) {
        consec_passes++;
    } else {
        consec_passes = 0;
    }

    double max_eval_here = -std::numeric_limits<double>::infinity();
    if(consec_passes == 2) {
        if((double)count_area_black() < (double)count_area_white() + komi) {
            max_eval_here = -std::numeric_limits<double>::infinity();
        } else {
            max_eval_here = std::numeric_limits<double>::infinity();
        }
    } else {
        auto potential_board_first2 = potential_board;
        auto consec_passes_first2 = consec_passes;
        auto leg_movesb = all_legal_moves(1); //all legal moves of BLACK
        for(int j = 0; j < (int)leg_movesb.size(); j++) {
            update_potential_board(std::get<0>(leg_movesb[j]),
                                   std::get<1>(leg_movesb[j]), 1); //BLACK!
            board = potential_board;
            if(std::get<0>(leg_movesb[j]) == -1) {
                consec_passes++;
            } else {
                consec_passes = 0;
            }
            double evalb;
            if(consec_passes == 2) {
                if((double)count_area_black() < (double)count_area_white() + komi) {
                    evalb = -std::numeric_limits<double>::infinity();
                } else {
                    evalb = std::numeric_limits<double>::infinity();
                }
            } else if(depth == 0) {
                evalb = athena_output_on_given_position(false);
            } else {
                evalb = tree_evaluation_after_black_move(false, depth-1, consec_passes);
            }
            if(evalb > max_eval_here) {
                max_eval_here = evalb;
            }
        }
        potential_board = potential_board_first2;
    }
}

```

```

        board = potential_board_first2;
        consec_passes = consec_passes_first2;
    }
}
if(max_eval_here < min_eval) min_eval = max_eval_here;
board = potential_board_first;
potential_board = potential_board_first;
consec_passes = consec_passes_first;
}
return min_eval;
}

```

Dit is wederom een grote (en daarom enigszins onoverzichtelijke) functie. Hij is nogal groot, omdat er ook rekening moet worden gehouden met alle randgevallen, wanneer de evaluatie naar plus- of minus-oneindig moet worden gezet. Zulke situaties doen zich voor, ingeval er bijvoorbeeld tweemaal achter elkaar een zet is overgeslagen, oftewel `if(consec_passes == 2)`. Het belangrijkste van de hierboven geprinte functie zijn de volgende regels:

```

double tree_evaluation_after_black_move(n.v.t., int depth, n.v.t.) {
    auto leg_movesw = all_legal_moves(2); //all legal moves of WHITE
    double min_eval = std::numeric_limits<double>::infinity();
    //white tries to make blacks evaluation as small as possible,
    //because white wants that black loses

    for(int i = 0; i < (int)leg_movesw.size(); i++) {
        <<Speel de zet leg_movesw[i] en update het bord>>
        double max_eval_here = -std::numeric_limits<double>::infinity();
        auto leg_movesb = all_legal_moves(1); //all legal moves of BLACK
        for(int j = 0; j < (int)leg_movesb.size(); j++) {
            <<Speel de zet leg_movesb[j] en update het bord>>
            double evalb;
            if(depth == 0) {
                evalb = athena_output_on_given_position(false);
            } else {
                evalb = tree_evaluation_after_black_move(false, depth-1, consec_passes);
            }
            if(evalb > max_eval_here) {
                max_eval_here = evalb;
            }
            <<Neem de zet leg_movesb[j] terug en zet de bordstelling terug>>
        }
        if(max_eval_here < min_eval) min_eval = max_eval_here;
        <<Neem de zet leg_movesw[i] terug en zet de bordstelling terug>>
    }
    return min_eval;
}

```

Wanneer men dit ziet, wordt duidelijk dat de variabelen `evalb`, `max_eval_here` en `min_eval` gebruikt worden om de evaluaties te bepalen. Er zit een for-loop in een for-loop, omdat eerst wit een zet moet spelen (je itereert over alle zetten), en na elke zet van wit moet er weer worden geïtereerd over alle zetten die zwart kan spelen.

De recursiviteit dezer functie ziet men vooral terug in de volgende code-regel:

```
evalb = tree_evaluation_after_black_move(false, depth-1, consec_passes);
```

Dit houdt in dat de functie zichzelf blijft aanroepen, maar steeds met één zetdiepte minder. Uiteindelijk kom je bij zetdiepte nul en dan wordt het neurale netwerk geraadpleegd.

Echter, in de praktijk hebben wij deze functie bijna alleen maar gebruikt bij *depths* niet hoger dan 1. Wanneer we diepte twee of drie (of hoger) instelden, duurde het erg lang om een partij te spelen, omdat het aantal neurale-netwerk-raadplegingen sterk toeneemt bij iedere verhoging van de kijkgrootte. Er zijn dan immers veel meer bordposities mogelijk.

In subsectie 8.2.2 hebben we verteld dat we niet zoveel betere resultaten behaalden, wanneer we het algoritme met de Fixed Athena gebruikten, dat net iets verder vooruitkeek dan in de meest basale situatie. Met de zojuist beschreven functie hebben we echter wél betere resultaten behaald. Dit vooruitkijken was dus nuttig. De precieze verbetering van speelsterkte zie je in cijfers terug in hoofdstuk 9.1.

Om enigszins goede en betrouwbare cijfers te genereren, moesten we héél veel partijen gespeeld laten worden. Helaas kostte dit teveel tijd bij grote Go-borden. We hebben de resultaten dus slechts in cijfers beschikbaar voor bordgrootte 9x9.

Hoofdstuk 9

Resultaten: ons eindproduct

Het eindproduct van dit profielwerkstuk, is een algoritme dat getraind is om goed Go te kunnen spelen. Dit trainen hebben wij laten plaatsvinden op verschillende bordgroottes. We laten nu per bordgrootte zien, welke resultaten wij behaald hebben met het algoritme. Om de gegeven percentages te berekenen, hebben wij het programma gewoon heel veel partijen laten spelen.

De winstpercentages zijn van het neurale algoritme tegen een *smartened random*. Dit laatste is een algoritme dat altijd een willekeurige zet speelt, met de volgende uitzonderingen:

- Indien de vorige zet is overgeslagen en de smartened random gewonnen staat wat betreft bordoppervlakte, dan slaat de smartened random altijd een zet over om zodoende direct te winnen.
- Indien de smartened random verloren staat wat betreft bordoppervlakte en er tenminste één andere legale zet mogelijk is, dan zal de smartened random hoe dan ook *geen* zet overslaan. (Zou de smartened random dit wel doen, dan zou zijn tegenstander daarna ook een zet overslaan, waardoor de smartened random zou verliezen.)

9.1 Bordgrootte 9x9 en vooruitkijken

Bij bordgrootte 9x9 hebben wij als absolute komiwaarde 6,5 gebruikt. Dezelfde komiwaarde gebruiken we ook bij alle andere bordgroottes, voor het gemak.

Deze bordgrootte hebben wij vooral gebruikt om te testen in hoeverre het gebruik van ons vooruitkijkalgoritme een verbetering in speelsterkte zou opleveren. Daarom hebben wij hier slechts de cijfers voor een positieve komiwaarde, oftewel: het algoritme speelt zwart. Wij zijn ervan overtuigd dat vergelijkbare resultaten zouden zijn behaald bij een negatieve komiwaarde,

maar omwille van de tijd hebben wij het vooruitkijkalgoritme niet ook nog getest met een negatieve komiwaarde.

9.1.1 Standaardmanier

Bij de standaardmanier, wanneer we dus **niet** de vooruitkijkalgoritmen uit hoofdstuk 8 gebruiken, behaalden wij de volgende resultaten:

```
THREAD 6 : Won 0 out of 0 games and 987 out of 1000 games.  
THREAD 23 : Won 0 out of 0 games and 991 out of 1000 games.  
THREAD 13 : Won 0 out of 0 games and 992 out of 1000 games.  
Results: algorithm won 31703 out of 32000 games. That is 99.0719 percent.
```

(Natuurlijk werden alle 32 threads gebruikt, maar we kunnen omwille van de ruimte niet steeds een volledig screenshot geven.)

9.1.2 Extra vooruitkijken

Wanneer we wel extra vooruitkeken, behaalde het algoritme de volgende resultaten:

```
THREAD 1 : Won 0 out of 0 games and 999 out of 1000 games.  
THREAD 24 : Won 0 out of 0 games and 1000 out of 1000 games.  
THREAD 10 : Won 0 out of 0 games and 1000 out of 1000 games.  
Results: algorithm won 31955 out of 32000 games. That is 99.8594 percent.
```

Dat is dus een forse verbetering in speelsterkte! We concluderen dat het gebruiken van extra vooruitkijfuncties, het algoritme sterker maakt. Onderliggens moeten we zeggen dat de bovenstaande cijfers niet extreem nauwkeurig zijn, aangezien er voor het genereren ervan ‘maar’ tweehonderdveertigduizend partijen werden gespeeld. We kunnen echter wel de conclusie trekken dat vooruitkijken het algoritme beter maakt.

Er konden ‘maar’ zo weinig partijen worden gespeeld met het vooruitkijkalgoritme, vanwege tijdsdruk aan onze zijde, maar ook omdat het gebruiken van het vooruitkijkalgoritme extreem veel tijd kostte. Er moesten immers veel meer berekeningen worden uitgevoerd.

Het vooruitkijkalgoritme werd hier - wederom omwille van de hoge hoeveelheid tijd die het kostte om een partij te spelen - slechts gebruikt indien er minder dan zeventien legale zetten waren. Het vooruitkijkalgoritme werd dan aangeroepen op diepte 1.

9.2 Bordgrootte 13x13

Bij een 13x13-bord hebben wij als (absolute) komiwaarde 6,5 gebruikt.

Lieten we het neurale netwerk op de standaardmanier (zie hoofdstuk 8) spelen tegen de smartened random, met een positieve komi (het netwerk speelt dan ‘echt’ zwart), dan kregen wij de volgende resultaten:

```
gotrainer@gotrainer:~/PWS/code/data$ ./go
Enable multithreading? (Also available with just one thread) [y/n] : y
Number of threads : 32

Write data to files and train? [y/n] : n
THREAD 24 : Won 4993 out of 5000 games and 4996 out of 5000 games.
THREAD 14 : Won 4995 out of 5000 games and 4994 out of 5000 games.
THREAD 4 : Won 4991 out of 5000 games and 4995 out of 5000 games.
THREAD 11 : Won 4992 out of 5000 games and 4990 out of 5000 games.
THREAD 15 : Won 4994 out of 5000 games and 4993 out of 5000 games.
THREAD 17 : Won 4996 out of 5000 games and 4994 out of 5000 games.
THREAD 26 : Won 4993 out of 5000 games and 4992 out of 5000 games.
THREAD 29 : Won 4995 out of 5000 games and 4991 out of 5000 games.
THREAD 16 : Won 4993 out of 5000 games and 4989 out of 5000 games.
THREAD 2 : Won 4996 out of 5000 games and 4989 out of 5000 games.
THREAD 27 : Won 4993 out of 5000 games and 4994 out of 5000 games.
THREAD 18 : Won 4991 out of 5000 games and 4995 out of 5000 games.
THREAD 12 : Won 4993 out of 5000 games and 4996 out of 5000 games.
THREAD 0 : Won 4991 out of 5000 games and 4993 out of 5000 games.
THREAD 5 : Won 4988 out of 5000 games and 4993 out of 5000 games.
THREAD 30 : Won 4996 out of 5000 games and 4993 out of 5000 games.
THREAD 8 : Won 4990 out of 5000 games and 4995 out of 5000 games.
THREAD 1 : Won 4992 out of 5000 games and 4989 out of 5000 games.
THREAD 3 : Won 4995 out of 5000 games and 4998 out of 5000 games.
THREAD 20 : Won 4995 out of 5000 games and 4992 out of 5000 games.
THREAD 23 : Won 4995 out of 5000 games and 4996 out of 5000 games.
THREAD 28 : Won 4993 out of 5000 games and 4990 out of 5000 games.
THREAD 31 : Won 4994 out of 5000 games and 4990 out of 5000 games.
THREAD 25 : Won 4990 out of 5000 games and 4991 out of 5000 games.
THREAD 13 : Won 4996 out of 5000 games and 4994 out of 5000 games.
THREAD 19 : Won 4997 out of 5000 games and 4990 out of 5000 games.
THREAD 22 : Won 4989 out of 5000 games and 4990 out of 5000 games.
THREAD 6 : Won 4991 out of 5000 games and 4989 out of 5000 games.
THREAD 9 : Won 4995 out of 5000 games and 4994 out of 5000 games.
THREAD 10 : Won 4994 out of 5000 games and 4992 out of 5000 games.
THREAD 7 : Won 4994 out of 5000 games and 4996 out of 5000 games.
THREAD 21 : Won 4993 out of 5000 games and 4985 out of 5000 games.
Results: algorithm won 319541 out of 320000 games. That is 99.8566 percent.
```

Merk trouwens op dat wij ook bij het genereren van deze resultaten, dankbaar gebruik hebben gemaakt van de 32 parallelle threads die we op het rekencluster toegewezen hebben gekregen.

Vervolgens hebben we met een negatieve komi, dus -6,5, gespeeld. In de echte situatie (buiten onze interne code) houdt dit in dat het netwerk nu wit heeft gespeeld. Dit was weer op de ‘standaardmanier’, waarbij er niet zo

wordt gelet op verder vooruitkijken. We kregen de volgende resultaten:

```
THREAD 1 : Won 4991 out of 5000 games and 4993 out of 5000 games.  
THREAD 25 : Won 4994 out of 5000 games and 4995 out of 5000 games.  
THREAD 29 : Won 4994 out of 5000 games and 4995 out of 5000 games.  
Results: algorithm won 319620 out of 320000 games. That is 99.8812 percent.
```

9.3 Bordgrootte 19x19

Nu gaan we ‘voor het eggie’. Bij de bordgrootte 19x19 behaalde ons neurale netwerk de volgende win-rates (komi 6,5 werd wederom gebruikt).

Positieve komiwaarde (netwerk #321703):

```
THREAD 14 : Won 3590 out of 3600 games and 0 out of 0 games.  
Results: algorithm won 114803 out of 115200 games. That is 99.6554 percent.
```

```
THREAD 26 : Won 3585 out of 3600 games and 0 out of 0 games.  
Results: algorithm won 114790 out of 115200 games. That is 99.6441 percent.
```

```
THREAD 16 : Won 3585 out of 3600 games and 0 out of 0 games.  
Results: algorithm won 114807 out of 115200 games. That is 99.6589 percent.
```

Negatieve komiwaarde (netwerk #333994):

```
THREAD 8 : Won 3600 out of 3600 games and 0 out of 0 games.  
Results: algorithm won 115184 out of 115200 games. That is 99.9861 percent.
```

```
THREAD 3 : Won 3600 out of 3600 games and 0 out of 0 games.  
Results: algorithm won 115191 out of 115200 games. That is 99.9922 percent.
```

```
THREAD 3 : Won 3600 out of 3600 games and 0 out of 0 games.  
Results: algorithm won 115186 out of 115200 games. That is 99.9878 percent.
```

Nemen we de gemiddelden, dan vinden we dat het netwerk wanneer het zwart speelt (positieve komi), in afgerond ongeveer 99,65 procent van de gevallen tegen een smartened random wint.

Bij negatieve komi speelt het netwerk wit. Als het netwerk wit speelt, wint het in afgerond ongeveer 99,99 procent van de partijen tegen een smartened random.

Nemen we nu van zwart en wit de gemiddelde winstrate, dan vinden we het volgende:

In afgerond ongeveer 99,82 procent van de partijen op een 19x19-Go-bord wint ons algoritme - zonder extra vooruitkijkfuncties - van een smartened-random-zettenspeler.

Dit beschouwen we als onze hoofdprestatie geleverd bij dit profielwerkstuk.

Hoofdstuk 10

Conclusie

Onze hoofdvraag was: “Hoe programmeer je een instantie van kunstmatige intelligentie die het spelletje Go speelt?”

Nou, in de afgelopen pagina’s hebben we die vraag beantwoord. Dit is een grote onderzoeksvraag, waarop we niet zomaar even een antwoord kunnen geven. Voor het antwoord: zie de rest van het profielwerkstuk. De theorie is echt te uitgebreid om hier in de conclusie samen te vatten.

De ‘conclusie’ die we wel hebben gevonden, is dat het kennelijk voor twee middelbare scholieren mogelijk is om zelf te begrijpen hoe je vanaf de wiskundige kern een neuraal netwerk programmeert, dat het mogelijk is dat ook te doen, en een Graphical User Interface te programmeren, en de Go-regels te implementeren. Neurale netwerken klinken als iets heel moeilijks, maar het is ons wel gelukt om er tenminste een beetje van te begrijpen.

Hoofdstuk 11

Discussie

Eerlijk gezegd zijn we best trots op ons profielwerkstuk. Toch is het natuurlijk nooit perfect. In dit hoofdstuk bediscussiëren wij ons werkstuk.

11.1 Dumb Athena en Fixed Athena

We trainden ons programma in de meeste gevallen op zo'n manier, dat het neurale netwerk tegen een (smartened) random agent moest spelen. Dit werkte aardig goed: hiermee hebben we hoge winstrates behaald. Dit is ook de methode die achteraf het beste heeft gewerkt.

Het nadeel van deze methode is echter: na verloop van tijd is het winstpercentage boven de 99 procent. Het netwerk wint dan bijna alles. Maar om te leren, is er natuurlijk winst én verlies nodig: als het netwerk (bijna) alles wint, ziet het zijn eigen fouten niet en kan het dus ook niet sterker worden of leren.

Om dit probleem op te proberen te lossen, hebben we ons in een aantal bochten gewrongen.

11.1.1 Fixed Athena

Het (luxe-)probleem was dat het netwerk tegen een randomachtig algoritme bijna alles won. Om dit op te lossen, moesten we dus een manier bedenken om het netwerk kunstmatig wat vaker te laten verliezen. De eerste oplossing die we hadden bedacht was een Fixed Athena. Het lerende algoritme speelt dan niet tegen een randomachtige agent, maar tegen een ander neurale netwerk dat moeilijker te verslaan is dan een random agent. Dit andere neurale netwerk, de Fixed Athena, is natuurlijk verkregen door eerdere training. De Fixed Athena verandert tijdens de training niet en is daarom 'Fixed'.

Omdat de Fixed Athena (*FA*) moeilijker te verslaan is dan een randomachtige zettenspeler, verliest het in training zijnde netwerk nu meer partijen. Het ‘probleem’ dat het alles wint, is dan voorbij. Toch bevonden zich op onze weg enige obstakels.

11.1.2 Randomiseren

Het eerste obstakel was zeer logisch: omdat Fixed Athena een neuraal netwerk is, speelt het altijd dezelfde zet gegeven dezelfde bordstelling. Dit betekent dat je steeds 100 van precies dezelfde partijen krijgt wanneer je steeds 100 partijen speelt en dan traint. Aan 100 dezelfde partijen heb je natuurlijk niet zoveel. Dit hebben we opgelost door de Fixed Athena te randomiseren:

```
else if(player_input_mode == "fa") {
    if(uniform_random_real_number(0.0, 100.0) < 25.0) {
        smartened_random_mv();
        //we need to have a randomized aspect in fixed athena,
        //otherwise it will always play the same
    } else {
        choose_white_best_move(athena_output_on_given_position, true);
    }
}
```

Het bovenstaande FA-algoritme speelt in 25 procent van de gevallen een smartened-random-zet, en in 75 procent van de gevallen wordt het FA-neurale netwerk geraadpleegd. De constante 25 hebben we natuurlijk soms ook op andere waarden gehad.

Toch hadden we een beetje pech. Op deze manier werkte het wel, maar er gebeurde het volgende:

De trainende Athena, die al was getraind en al zeer sterk was geworden tegen een random, verloor in het begin iets meer partijen. Dat zag er goed uit, gezien het probleem dat we hadden. Na verloop van tijd won het in training zijnde algoritme meer partijen. Dat was ook goed. De teleurstelling was echter wanneer we het nieuwe algoritme lieten spelen tegen een random om te testen of het sterker was geworden: dat bleek niet zo te zijn. Tegen het specifieke FA-algoritme was ons trainende netwerk sterker geworden, maar niet tegen een willekeurig-achtige zettenspeler.

Hoe kan dit nu weer? Onze hypothese is als volgt: misschien had het FA-algoritme als het ware een bepaalde speelstrategie. Het in training zijnde algoritme krijgt na verloop van tijd die FA-spelstrategie door, en bedenkt een nieuwe strategie, die de FA-strategie vaak verslaat. De nieuwbedachte strategie mag dan wel beter werken tegen de FA-strategie, maar tegen een (smartened) random agent hoeft het de nieuwe strategie helemaal niet beter

te werken. Bedenk wel dat dit slechts een hypothese is en dat we niet zeker weten of dit de juiste verklaring is.

11.1.3 Dumb Athena

Edoch, onze purperen geest vaarwel gezegd hadden niet de overwegingen der zoete ijver, met behulp van welke wij plachten te trachten, te verbeteren de speelprestaties van ons algoritme. In niet-pseudo-Homerische taal: we hadden weer een nieuw idee om te verbeteren bedacht, genaamd Dumb Athena. Dumb Athena is, zoals de naam al zegt, een variant van het neurale netwerk die kunstmatig wat dommer is gemaakt door te randomiseren, bijvoorbeeld als volgt:

```
else if(player_input_mode == "d") { //dumb athena used for training
    if(uniform_random_real_number(0.0, 100.0) < 90.0) {
        smartened_random_mv();
    } else {
        choose_black_best_move(athena_output_on_given_position, false);
    }
}
```

Oftewel, in slechts 10 procent van de gevallen wordt het in training zijnde neurale netwerk geraadpleegd: in de overige gevallen wordt een smartened-random-zet gespeeld.

Het doel hiervan is om het netwerk meer partijen te laten verliezen. Als dat gebeurt en er een balans ontstaat tussen winnen en verliezen, dan ziet het dommer gemaakte netwerk zijn fouten beter en kan het dus beter leren. Als je het netwerk op een kunstmatige manier verdomt, is dat precies wat er gebeurt. Het netwerk kan beter trainen en wordt dan beter. Je gaat verder met trainen met het domme algoritme en dat algoritme wordt sterker. Ons idee was dan: haal je de domheidsfactor weer weg, dan moet je wel een heel slim neuraal netwerk overhouden.

Toch is dit niet helemaal wat er gebeurde. Het netwerk werd niet extreem veel slimmer. Bij de Fixed Athena hadden we hier een verklaring voor kunnen bedenken, maar hier was dat wel moeilijker. Er zijn zoveel parameters: netwerkgrootte, learning rate, de 90%-factor van de Dumb Athena... Wellicht hadden we, als we meer tijd hadden gehad, meer kunnen bereiken met deze Dumb Athena. Toch zijn we trots op wat we zonder Dumb Athena hebben bereikt.

11.2 Potentiële verbeteringen aan het vooruitkijkalgoritme

Het grootste probleem bij het vooruitkijkalgoritme waar we tegenaan liepen, was dat het in bepaalde gevallen extreem veel berekeningstijd kostte, waardoor het onmogelijk was om ermee te trainen; hiervoor zijn namelijk veel partijen nodig. We hebben dit probleem proberen op te lossen door bijvoorbeeld te coderen dat het algoritme alleen vooruitkijkt als er minder dan een gegeven aantal legale zetten is. Dit is echter wel een lelijke oplossing: het kan zijn dat je een bordstelling hebt waarin er maar drie zetten mogelijk zijn, maar waarin na een willekeurige zet, er weer heel veel zetten mogelijk zijn. Dan gebruikt het algoritme toch weer teveel rekentijd.

We hebben dit proberen op te lossen door een tijdslimiet te geven aan de vooruitkijkfunctie. Hierbij moest het vooruitkijkalgoritme stoppen als de tijdslimiet bereikt werd. In dat geval werd er dan overgeschakeld op het neurale netwerk zonder extra vooruitkijken.

We hebben dit geprobeerd te programmeren, maar eerlijk gezegd is de domme reden dat we hiermee zijn opgehouden, de deadline die naderde. Als we het hadden doorgezet, hadden we met de tijdslimiet wellicht iets bereikt, maar nu dus niet.

Hoofdstuk 12

Nwoord

De afgelopen tijd hebben wij gewerkt aan dit profielwerkstuk. In ons geval is deze periode eigenlijk langer dan een jaar: op 24 december 2020 (!) kwam onze samenwerking voorzichtig tot stand. Later werd dit definitief.

We hebben heel hard gewerkt aan dit profielwerkstuk. Voor Aron geldt ook dat dit de studierichting is waarnaar hij kijkt, en dus was dat voor hem nog een extra motivatie om hard te werken, naast dat we dit beiden een leuk onderwerp vonden.

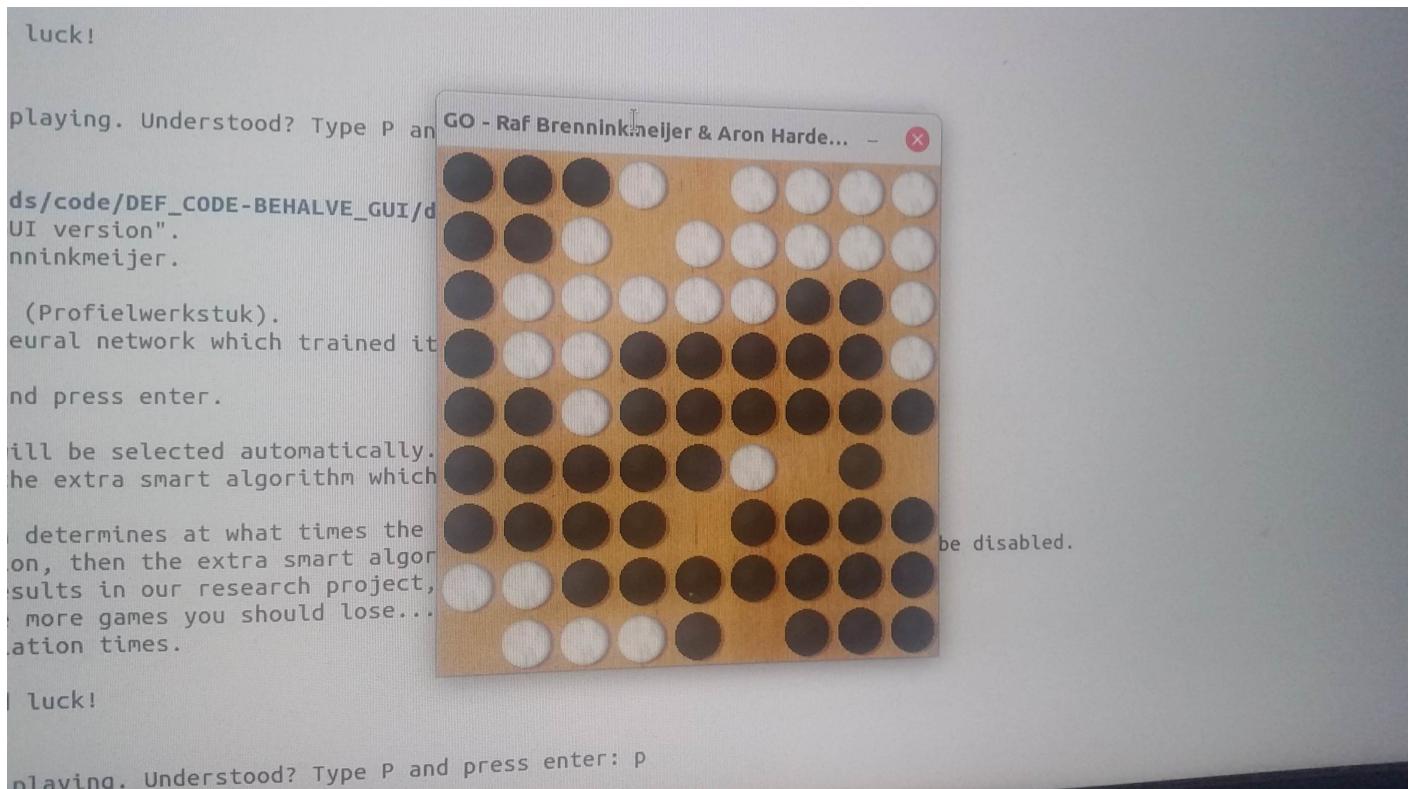
Het profielwerkstuk heeft een bijna belachelijk groot aantal pagina's, daarvan zijn wij ons bewust. Maar ja, we hadden zo veel leer- en programmerwerk gedaan, en toen we dat alles op papier zetten, kwamen we erachter dat daar iets meer pagina's voor nodig waren dan we eigenlijk hadden gedacht. Hopelijk is het een pluspunt.

Door dit profielwerkstuk te maken, hebben wij beiden geleerd om beter te kunnen programmeren. Ook onze wiskundige kennis werd uitgebreid, met name op het gebied van multivariabele calculus. Denk aan het bewijs van backpropagation: dat is een heel hoofdstuk met als onderwerp het bepalen van twee soorten partiële afgeleiden.

Verder hebben we nog iets anders geleerd en dat is het gebruiken van L^AT_EX. Dit pdf-bestand is gemaakt met behulp van L^AT_EX. Dankzij L^AT_EX hebben we zulke mooie wiskundige formules in de pdf kunnen zetten; anders was dit veel moeilijker geweest.

Nu nog even een mooie anekdote over onze eigen ervaringen. Natuurlijk hebben wij ook zelf wel eens tegen ons algoritme gespeeld. Speelden we expres een beetje dom, dan was het voor het algoritme niet zo moeilijk om ons te verslaan. Het is ons echter ook wel eens overkomen dat we verloren terwijl we wel serieus probeerden te spelen zonder expres dom te doen! Dit gebeurde vooral toen we nog aan het testen waren met bordgrootte 5x5, maar ook bij 9x9 is het een keer gebeurd. (Achteraf is het jammer dat we in de

definitieve versie geen 5x5-neuraal netwerk meer ingevoegd hebben...) Op het moment dat je dan zo'n domme fout maakt (die het algoritme afstraf!) schrik je een beetje. Je kunt het zien als jammer om te verliezen, maar voor ons is dit eigenlijk juist prachtig. Want hoe ontzettend mooi is het (in dit geval) als je verslagen wordt door je eigen computerprogramma?!



Aron speelde hier, wit zijnde, tegen het extra-vooruitkijkende algoritme met bordgrootte 9. Hij had een domme fout gemaakt en verloor spoedig. Deze partij was prachtig.

12.1 Dankwoord

We zouden ook graag nog een aantal mensen specifiek willen bedanken.

Bedankt mevrouw Jacobs, wiskundeleraar op het Willem Lodewijk Gymnasium te Groningen en tevens onze profielwerkstukbegeleider. Zij heeft ons begeleid gedurende het maken van dit profielwerkstuk. Ze heeft onze eerste versie gelezen en daarbij goed commentaar en tips gegeven. Ook mochten we maandenlang haar Calculusboek lenen met meer dan 1000 pagina's. En in een periode dat ze niet op school kon zijn, heeft ze toch nog een keer met

ons willen videobellen.

We willen ook hartelijk bedanken de heren Fokke Dijkstra, Henk-Jan Zilverberg en Cristian Marocico, van de Rijksuniversiteit Groningen. Zij hebben ons enorm geholpen: in de eerste instantie is het geweldig dat zij openstonden voor een videogesprek en dat we vervolgens hun rekencluster mochten gebruiken, maar vervolgens hebben we ze nog een paar keer gemaaid en ze bleven ons helpen, bijvoorbeeld door de debugger GDB te installeren en onze virtuele machine te vergroten tot 32 cores. Dankzij dit rekencluster hebben we een heel nieuwe dimensie kunnen toevoegen aan het werkstuk. Heel erg bedankt!

Aron wil nog specifiek zijn ouders bedanken, die graag aandacht gaven en luisterden als hij bijvoorbeeld vertelde hoeveel pagina's het werkstuk nu dan wel weer had. Raf wil ook zijn ouders bedanken die ondanks hun weinige kennis over het onderwerp met veel interesse hebben geluisterd. Raf wil ook graag zijn broer bedanken, die met de toffe titel kwam van ons profielwerkstuk.

Tot slot volgt nog een algemene dankbetuiging aan alle mensen die - direct of indirect - hebben meegeholpen aan de totstandkoming van dit profielwerkstuk. Bedankt, wetenschappers die kunstmatige neurale netwerken hebben bedacht. Bedankt, ontwikkelaars van C++, ontwikkelaars van L^AT_EX, van (Ubuntu) Linux, van de debuggers GDB en Valgrind, van de codeprofilers Gprof, van de C++-compiler g++, van de graphicsbibliotheek SDL2, et cetera. Eer komt ook toe aan de natuurkundigen en wetenschappers dankzij wie er nu computers bestaan, en het internet, et cetera. Wij, de auteurs van dit werkstuk, zijn slechts kleine wezentjes op de schouders van deze reuzen.

Literatuurlijst

- British Go Association. (2017, 26 oktober). *How to Play*. Geraadpleegd op 28 november 2021, van <https://www.britgo.org/intro/intro2.html>
- Carremans, B. (2018, 23 augustus). *Handling overfitting in deep learning models*. Towards Data Science. Geraadpleegd op 20 december 2021, van <https://towardsdatascience.com/handling-overfitting-in-deep-learning-models-c760ee047c6e>
- Dellinger, J. (2019, 3 april). *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming*. Towards Data Science. Geraadpleegd op 14 januari 2022, van <https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79>
- Hansen, C. (2019, 22 augustus). *Activation Functions Explained - GELU, SELU, ELU, ReLU and more*. Machine Learning From Scratch. Geraadpleegd op 1 november 2021, van <https://mlfromscratch.com/activation-functions-explained/#/>
- Khan Academy. (z.d.-a). *Derivative notation review*. Geraadpleegd op 24 november 2021, van <https://www.khanacademy.org/math/ap-calculus-ab/ab-differentiation-1-new/ab-2-1/a/derivative-notation-review>
- Khan Academy. (z.d.-b). *Introduction to partial derivatives*. Geraadpleegd op 24 november 2021, van <https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/introduction-to-partial-derivatives>
- Khan Academy. (z.d.-c). *The gradient*. Geraadpleegd op 24 november 2021, van <https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/the-gradient>
- McGonagle, J., Shaikouski, G., Williams, C., Hsu, A., Khim, J., & Miller, A. (z.d.). *Backpropagation*. Brilliant. Geraadpleegd op 28 november 2021, van <https://brilliant.org/wiki/backpropagation/>
- Nielsen, M. A. (2015). *Neural Networks and Deep Learning* [E-book]. Determination Press.
- Q-learning. (2021, 25 november). In *Wikipedia*. <https://web.archive.org/web/20211129151044/https://en.wikipedia.org/wiki/Q-learning>
- Sanderson, G. [3Blue1Brown]. (2017, 3 november). *What is backpropagation really doing? | Chapter 3, Deep learning* [Video]. YouTube. <https://www.youtube.com/watch?v=Ilg3gGewQ5U>
- Sensei's Library. (2021, 15 oktober). *Superko*. Geraadpleegd op 28 november 2021, van <https://senseis.xmp.net/?Superko>
- SG. (2019, 10 juli). *PReLU activation*. Medium. Geraadpleegd op 1 november 2021, van <https://medium.com/@shoray.goel/prelu-activation-e294bb21fef>
- Shyalika, C. (2019, 15 november). *A Beginners Guide to Q-Learning*. Towards Data Science. Geraadpleegd op 28 november 2021, van <https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c>
- Stewart, J. (2011). *Calculus Early Transcendentals* (7de editie). Thomson Brooks/Cole.

Bijlagen

Bij dit profielwerkstuk zijn enkele bijlagen opgenomen. Dit zijn voornamelijk codebestanden en bestanden waarin de parameters van de gebruikte neurale netwerken staan.

De belangrijkste codebestanden in de bijlage zijn:

main.cpp
go_rules.cpp
athena.cpp
train_go.cpp
gui.cpp

De codebestanden horen bij het profielwerkstuk.

Hoe opent u het programma?

In de map met codebestanden bevindt zich een map genaamd '**data**'. In de map 'data' bevindt zich het bestand '**go**'. Dit is een voor Linux pregecompileerde versie van de GUI-versie van ons programma. Door dit bestand te openen, kunt u bijvoorbeeld tegen onze neurale netwerken spelen.

Zelf compileren

Let op: zelf compileren zou niet nodig moeten zijn! Er bestaat een pregecompileerde versie van het programma, zie hierboven. Indien toch gewenst, kunt u het programma ook zelf compileren. In de map met codebestanden bevindt zich het bestand '**fc**'. Door 'fc' te runnen, op Linux, worden alle codebestanden gecompileerd en gelinkt. Het uiteindelijke bestand wordt geschreven naar data/go. Bij het compileren ontstaat een klein aantal warnings. Deze kunnen geen kwaad.

Let op: hernoem het bestand 'go' niet!