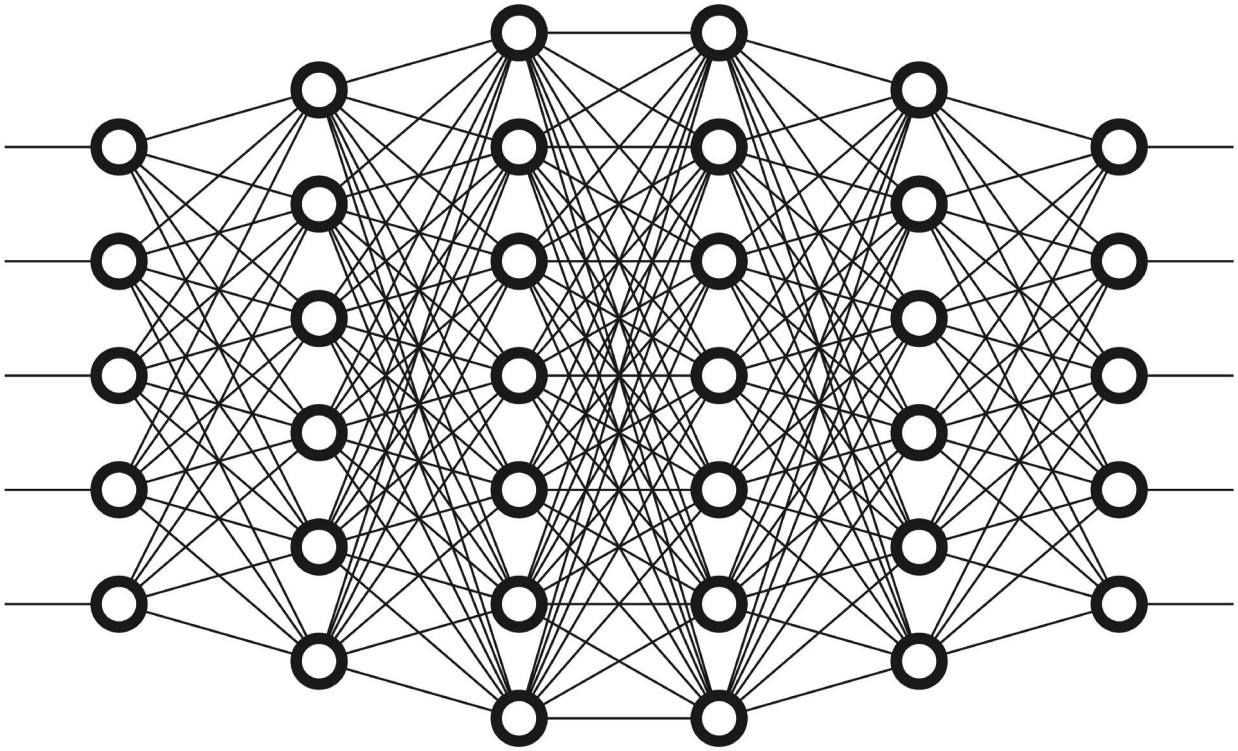
Mercantec

Ressourceoptimering af implementeringer og sammenligning af forskellige maskinlæringsstrategier anvendt på fire-på-stribe

S. F. Jakobsen <sfja2004@gmail.com>

R. P. Browa <mail@reim.ar>

28. February 2025



Indhold

Abstrakt

I machine learning eksisterer der forskellige typer af modeller. Disse modeller har forskellige karakteristika og anvendeligheder. Vi undersøger to forskellige modeltyper; decision tree og neural network. Vi undersøger implementering af disse modeller, der reducerer ressourceforbrug. For at undersøge anvendelighed, implementerer vi et fire-på-stribe-spilsimulation, til at træne og anvende modeller på. Vi afslutter med en diskussion af, hvordan træningsdata kan genereres, og hvordan modellernes teoretiske implementation vil kunne forbedres.

Fire-på-stribe

Spillet fire-på-stribe indeholder konceptuelt et spillebræt med 7 kolonner og 6 rækker. Man kan indsætte brikker i hver kollone. Hver brik fylder ét felt. Brikker indsættes øverst på brættet og fylder op fra bunden. Der er 2 forskellige kulører af brikker, en kulør til hver spiller; rød og gul, en kulør til hver af 2 spillere. En spiller starter, og derefter går turen på skift, hvor hver tur består af, at en spiller indsætter én brik i én kolonne. Når en kolonne er fyldt, dvs. alle 6 felter er fyldte, kan der ikke indsættes flere brikker i kolonnen. En spiller har vundet, hvis spillerens brikker på brættet danner et fire-på-stribe-mønster. Dette gælder både vandret, lodret og diagonalt[1].

For at kunne træne ML-modeller til at spille, kræves der en simulation af spillet. En implementering kræver først og fremmest, at spillet kan spilles korrekt, dvs. reglerne overholdes. Dette inkluderer også at simulationen kan forstå, for hvert træk, om en spiller har vundet. I henhold til træning af ML-modeller er det fordelagtigt at minimere ressourceforbrug, da man derved kan maksimere mængden af træning i en given tidsperiode[3]. Derfor kræves der også af simulation, at ressourceforbrug er minimeret.

Implementation

Implementationen af simulationen tager udgangspunkt i spilbrættet. Klassen **Board** repræsenterer et spilbræt for et enkelt spilt. **Board** har et sæt metoder, **possible_moves**, **insert** og **game_state**.

Board repræsenterer et spilbræt internt med en 128-bit integer. Hvert felt er repræsenteret af 2 bits: værdien 0 er et tomt felt, 1 er et felt med rød brik, 2 er et felt med gul brik og værdien 3 er ugyldig. En række er derfor repræsenteret som 7*2=14 bits, og hele brættet er derfor repræsenteret som 7 kolonner af 6 rækker af 2 bits = 6*7*2=84 bits i alt. Om det er rækker eller kolonner, der skal ligge sammenhængende, er i stor grad vilkårligt. Vi har valgt at lægge rækker sammenhængende.

Board har et set af interne metoder, som klassens metoder udaftil benytter. Disse inkluderer mest væsentligt **tile** og **set_tile**. **tile**-metoden udtrækker værdien for et givent felt på brættet i form af en **Tile**-værdi, specificeret med en **Pos**-værdi for position. **set_tile**-metoden er førnævntes komplement, som istedet indsætter en **Tile**-værdi på brættet i en givet position.

Begge metoder benytter **offset**-metoden. Denne metode omdanner en position af **Pos**-typen til en forskydning ind i brætrepræsentationen, dvs. 128-bit integeren. Forskydningsberegningen udføres sådan: $\text{offset} = \text{column} \cdot \text{board\_height} + \text{row}$, hvor **row** og **column** er indeksering i boardet nulindekseret med nulpunkt i øverste venstre hjørne.

```
// src/board.hpp:97
inline auto offset(Pos pos) const -> size_t
{
    return pos.col * height + pos.row;
}
```

tile-metoden udtrækker brikværdien ved at forskyde board-integeren ifølge forskydningen og tile-brædden, så positionens værdi ligger som de første 2 bits. Herefter anvendes en maske, som filtrerer alle andre værdier i board-integeren fra. Til sidst omdannes de 2 bits til en værdi af **Tile**-typen og returneres. Beregningen udføres som følgende: $\textit{board} \gg \textit{offset}(\textit{pos}) \cdot \text{tile\_size} \,\&\, \text{tile\_mask}$, hvor board er 128-bit-integeren, >>-operatoren er højre-shifting og **&**-operatoren er bitvis AND-operation.

```
// src/board.hpp:86
inline auto tile(Pos pos) const -> Tile
{
    return static_cast<Tile>(m_val >> offset(pos) * tile_size & tile_mask);
}
```

set_tile-metoden tager udgangspunkt i en tile-værdi, som den omdanner til 2-bit-repræsentation i en 128-bit integer. Herefter forskydes værdien ifølge positionsforskydning og tile-størrelse. Siden værdien originalt kun indeholdte 2 bits, er det ikke nødvendigt at anvende bitmasken. Til sidst bliver værdien indsat i board-integeren med en bitvis OR-operation. Beregning udføres som følgende: $\textit{board} \mid= \textit{tile} \ll \textit{offset}(\textit{pos}) \cdot \text{tile\_size}$.

```
// src/board.hpp:91
inline void set_tile(Pos pos, Tile tile)
{
    m_val |= static_cast<uint128_t>(std::to_underlying(tile))
        << offset(pos) * tile_size;
}
```

possible_moves-metoden returnerer en liste over kolonner, som stadig har plads til, at der kan indsættes en brik. Den tjekker alle kolonner igennem og finder ud af, om den øverste tile i hver kolonne er tom og returnerer hvilke kolonner, der har plads.

insert-metoden udfører konceptuelt et træk i spillet. Man specificerer en kolonne og hvilken tile man vil indsætte, og så fylder den det laveste felt ud i kolonnen, der stadig er tom.

is_draw-metoden returnerer hvorvidt brættet er fuldt, dvs. om spillet er afsluttet uden vindere. Dette gør den, ved at undersøge hver felt i øverste række.

game_state-metoden returnerer hvorvidt rød/blå har vundet, om det står lige, eller om spillet stadig er i gang. Vindere er fundet ved at kigge efter fire-på-stribe-mønstre på brættet.

Diskussion af fire-på-stribe

Spilsimulatorens implementation er optimeret i henhold til minimering af afviklingstid. Implementationens ydeevne i sin partikularitet og sammenligning med alternativer, behandler vi ikke i dette whitepaper. Implementationens ydeevne opnås, ved at spillets mekanismer er implementeret, så de dels minimerer mængden af afviklede CPU-instruktioner og mere væsentligt, så de minimerer mængden af memory-access, som har store ydelsesimplikationer[2].

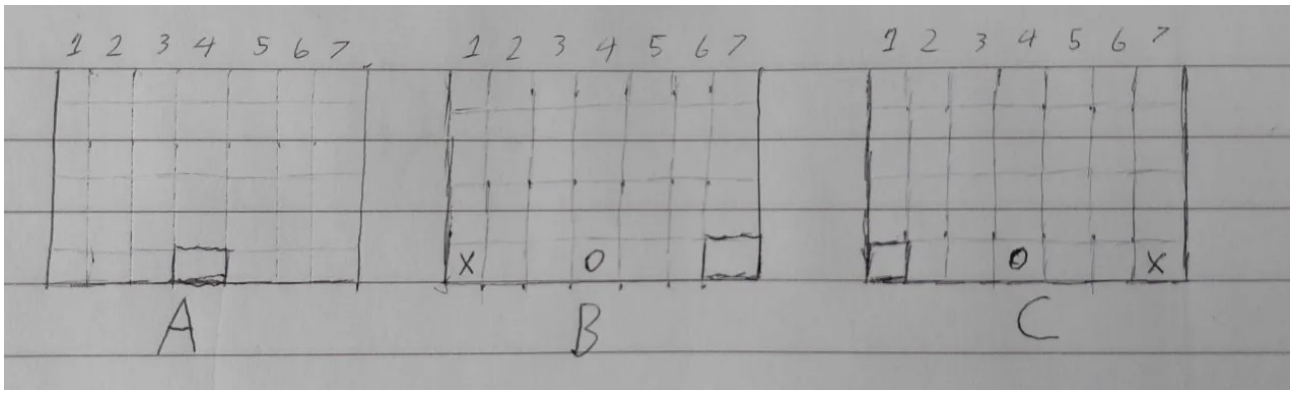
Decision-tree-modellen afhænger af, at kunne skelne mellem forskellige brætstadier. Det fordelagtige i dette tilfælde, er at repræsentationen er minimal, samtidig med at tilgang ikke begrænses markant. Dette opnås, ved at konvertere et givet brætstadie til en 64-bit integer. Denne repræsentation er bygget op som følgende. Hver kolonne behandles hver for sig. 3 bit bruges til at repræsentere højden på en kolonne. 6 bit bruges så til at repræsentere hver brik i kolonnen indenfor højden. Dette er muligt, fordi brikker altid fylder op fra bunden. Herved kan hver kolonne repræsenteres med 3+6=9 bit. Altså kan bræt-stadiet med brættets 7 kolonner repræsenteres med 7*9=63 bit, som kan opbevares i en 64-bit integer. Siden et fire-på-stribe-spil kan spejlvendes, er der funktionalitet, så 64-bit-repræsentationen kan spejlvendes. Dette halverer den totale mængde af mulige kombinationer.

Decision tree

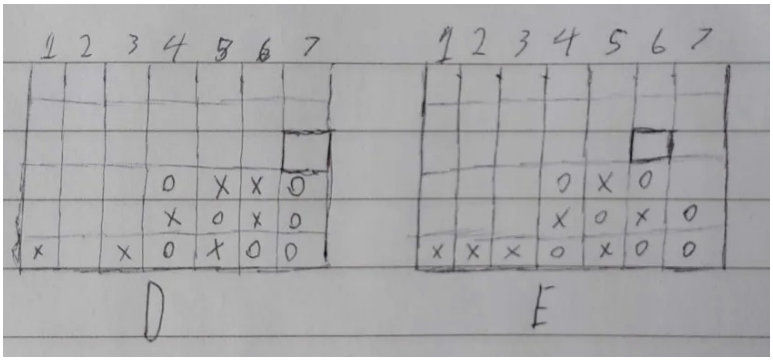
Konceptuelt

Decision tree er en ML-strategi. Princippet er, at modellen gemmer hver muligt stadie, dvs. hvert kombination i fire-på-stribe-spillet, sammen med det optimale valg i situationen. Altså danner modellen et træ over alle beslutningerne i et fire-på-stribe-spil.

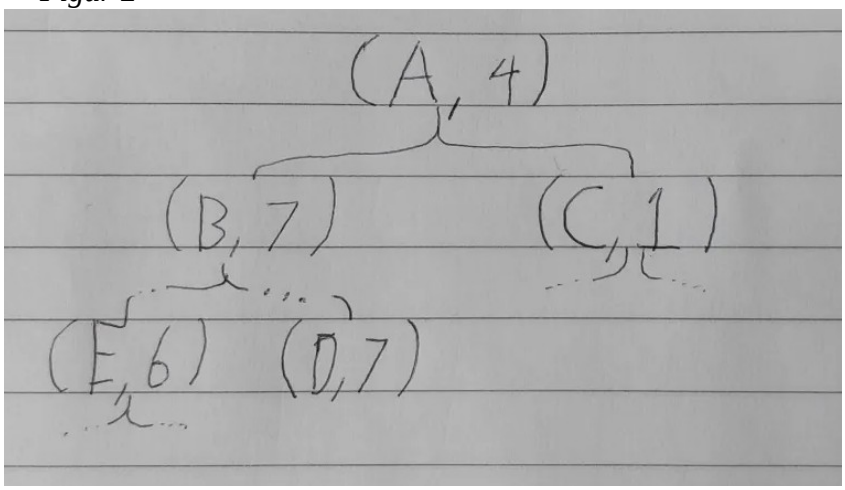
På figur 1 og 2 ses forskellige brætstadier. Modellen spiller som O og modstanderen spiller som X (istedet for gul og rød). I A skal modellen vælge en startposition. Siden kolonne 4 er den bedste startposition, bør modellen vælge denne kolonne. På figur 3 ses beslutningstræet, hvis modellen starter. Her kan man se, at modellen vil vælge kolonne 4 i situation A. Det kan ses, at situationerne B og C har forskellige korrosponderende valg af kolonner. Situationerne D og E opstår flere træk inde i spillet. I D kan det ses, at modellen har mulighed for at vinde, ved at placere i kolonne 7, hvilket modellen bør vælge. I E kan det ses, at modellen skal forsøge at undgå modstanderen vinder, ved at placere en blokerende O i kolonne 6.



Figur 1



Figur 2



Figur 3

Implementering

Modellen er implementeret, så den gemmer en vægt for hver kolonne, for hvert spilstadie, i stedet for kun at gemme kolonnetallet. Kolonnen med største vægt er i stedet den valgte kolonne.

Beslutningstræet konstrueres, ved at træne modellen. Dette gøres, ved at lade modellen spille mod en modstander. Hver stadie spillet kommer igennem gemmes. Når spillet er afsluttet, finder man ud af, om modellen har vundet, tabt eller spillet er uafgjort. Hvis modellen har vundet, går træningsalgoritmen igennem alle de gemte stadier, der førte til at modellen vandt. For hvert stadie belønner algoritmen beslutningen, ved at øge vægten for den valgte kolonne. Omvendt, hvis modellen taber, straffer algoritmen modellen, ved at sænke vægten for de valgte kolonner for hvert stadie. Dette gælder generelt for algoritmer til at konstruere beslutningstræer. Hvad algoritmen gør i tilfælde af uafgjort afhænger af den specifikke algoritmes detaljer.

Implementationen består hovedsageligt af klassen DeciTreeAi. Klassen repræsenterer en simuleret spiller, der benytter en decision tree-model til at beslutte træk. DeciTreeAi-klassens primære metode er next_move, som beslutter et træk ud fra et brætstadie i form af Board-typen. Denne metode bruges både under træning og efter træning til spil. Til træning har DeciTreeAi-klassen yderligere metoderne new_game, report_win, report_loss, report_draw. Disse bruges under træning af modellen.

Klassen er implementeret med udgangspunkt i beslutningstræet, som er repræsenteret med en key/value-store. Beslutningstræet har C++-typen std::unordered_map<Board::Hash, ColWeights>. Typen Board::Hash er brættet repræsenteret i en 64-bit integer. Typen ColWeights repræsenterer vægtene for hvert kolonne, og er et alias for C++-typen std::array<Weight, Board::width>, hvor Weight er et alias for int16_t, dvs. en 16-bits integer. Typen std::unordered_map er en key/value-store, hvor værdierne er organiseret så lookup er optimeret. Dvs. tiden det tager, at finde kolonnevægte ud fra et brætstadie minimeres. Dette kommer på bekostning af tiden det tager, at indsætte nye værdier ind i kollektionen. Men insert-tiden er mindre væsentlig end lookup-tiden, da der højst er én insertion per træningsspil og der kan være op til 42/2=21 lookups i ét spil.

```
// src/deci_tree_ai.hpp:14
using Weight = int16_t;
using ColWeights = std::array<Weight, Board::width>;
// ...
// src/deci_tree_ai.hpp:64
  std::unordered_map<Board::Hash, ColWeights> m_choice_weights {};
```

Metoden next_move implementerer en algoritme, for at finde prefered kolonne med beslutningstræet ud fra et brætstadie. Ud af mulige træk på boardet, finder algoritmen alle kandidater, dvs. trækket eller trækkene med størst vægt. Algoritmen vælger kolonner med størst vægt med en tollerence fra den største vægt, som kaldes exploration rate. Exploration rate er en variabel, der kan justeres for at øge kvaliteten af træningen. Et træk vælges ud af alle kandidater, med et tilfældigt genereret tal. Beslutningen gemmes og kolonnetallet returneres. Når algoritmen kigger i beslutningstræet, tester den bræt-hash'et. Hvis en værdi ikke eksisterer for bræt-hashet, tester algoritmen for bræt-hash'et af et spejlvendt bræt. Dette er muligt, da fire-på-stribe er symmetrisk, og gør at beslutningstræet kun behøver at gemme halvdelen af alle mulige beslutninger. Hvis hverken eksisterer i træet, oprettes en ny entry med nulstillede vægte.

```
// src/deci_tree_ai.hpp:14
using Weight = int16_t;
using ColWeights = std::array<Weight, Board::width>;
// ...
// src/deci_tree_ai.hpp:64
std::unordered_map<Board::Hash, ColWeights> m_choice_weights {};

// src/deci_tree_ai.cpp:18 DeciTreeAi::next_move
for (uint8_t col = 0; col < board.width; ++col) {
    if (!possible_moves.at(col))
        continue;

    auto weight = weights->at(col);
    if (weight > cand_weight) {
        cand_weight = weight;
        cand_size = 0;
    }
    // includes exploration rate
    if (choice_is_candidate(weight, cand_weight)) {
        candidates[cand_size] = col;
        cand_size += 1;
    }
}
// ...
m_current_choices.push_back({ hash, col });

// src/deci_tree_ai.cpp:47 DeciTreeAi::lookup_choices
auto hash = board.hash();
if (m_choice_weights.contains(hash))
    return /* ... */;

auto flipped = board.flipped_hash();
if (m_choice_weights.contains(flipped))
    return /* ... */;

    m_choice_weights.insert_or_assign(hash, ColWeights { 0 });
```

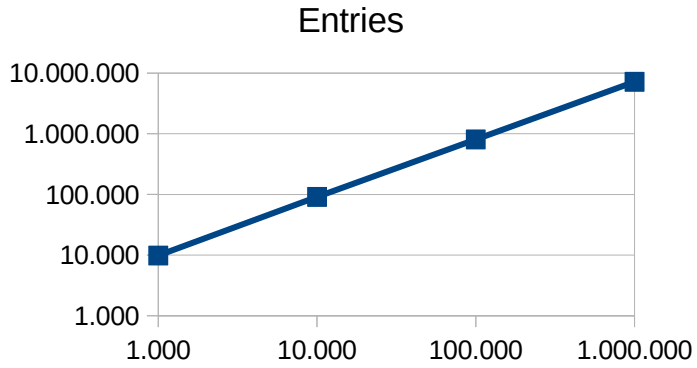
For at træne modellen, laves der 2 instanser af DeciTreeAi-klassen. Disse 2 instanser spiller mod hinden i et bestemt antal træningsiterationer. Instanserne skiftes til at starte. For hvert træk tjekker træningsalgoritmen om en af instanserne har vundet. Hvis en instans har vundet eller spillet er end uafgjort påbegynder spillet påny.

```
// src/main.cpp:346
for (int i = 0; i < training_iters; ++i) {
    auto board = Board();

    bot1.new_game();
    bot2.new_game();

    auto* current = &bot1;
    auto* other = &bot2;

    while (true) {
        size_t col = current->next_move(board);
        board.insert(col, current->tile());

        auto state = board.game_state();
        if (state == color_win_state(turnee.color())) {
            turnee.report_win();
            other.report_loss();
            break;
        }
        // ... check loss ...
        if (state == GameState::Draw) {
            turnee.report_draw();
            other.report_draw();
            break;
        }

        std::swap(current, other);
    }
}
```

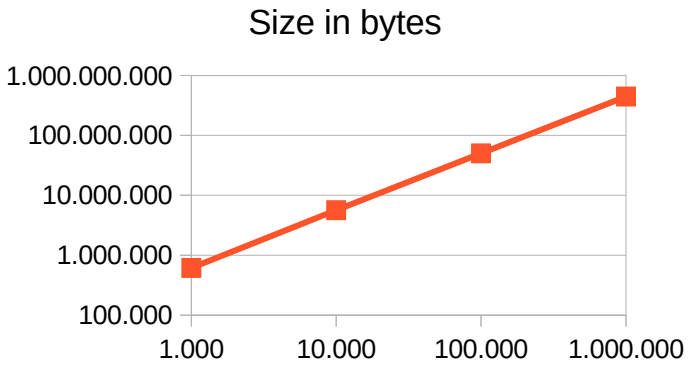
Træning

Hvis denne algoritme afvikles, kan man sammenligne forskellige iterationer, både i forhold til tid det tager, men også i forhold til størrelse.

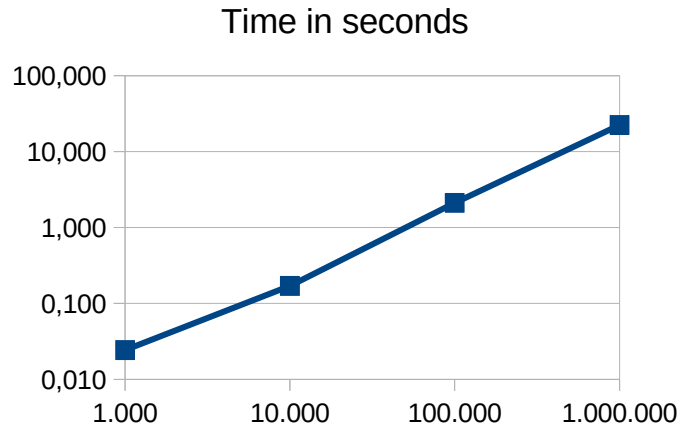
iterations	**entries** avg.	**bytes** avg.	**time** avg.
1.000	9.842	613.170	0,024
10.000	91.201	5.654.478	0,170
100.000	807.168	50.044.401	2,114
1.000.000	7.173.009	444.726.581	22,381



Figur 4: Entries per tid i sekunder, begge akser er logaritmiske.



Figur 5: Modelstørrelse i bytes per tid i sekunder, begge akser er logaritmiske.



Figur 6: Modelstørrelse i bytes per tid i sekunder, begge akser er logaritmiske.

Det ses at korrolationen mellem træningsiterationer er linear, både i forhold tid og størrelse i bytes. Kvaliteten af modellen afhænger af mængden af iterationer. En model trænet med 1.000.000 iterationer er godkendt anekdotisk, men vi vil ikke behandle modellens dygtig yderligere.

Diskussion af decision tree

Den primære fordel ved decision tree-modeller, er deres anvendelighed til reinforcement learning. Et decision tree bliver efter få iterationer gode til de scenarier den har spillet. Dette er det, der gør det muligt at træne 2 instanser op ad hinanden i dette tilfælde. Den største ulempe ved decision tree-modeller ressourceforbruget, både i forehold til størrelse og træningstid. Ved komplekse problemer er der mange kombinationer. En decision tree-model skal træne hver eneste kombination, dvs. gemme hver eneste kombination, for at kunne tage en fornuftig beslutning ved den kombination.

Altså kan decision tree-modeller ikke lave regression. Decision tree-modeller er derfor ressourcekrævene, sammenlignet med andre strategier, som eksempelvis neurale netværk.

Neural Network

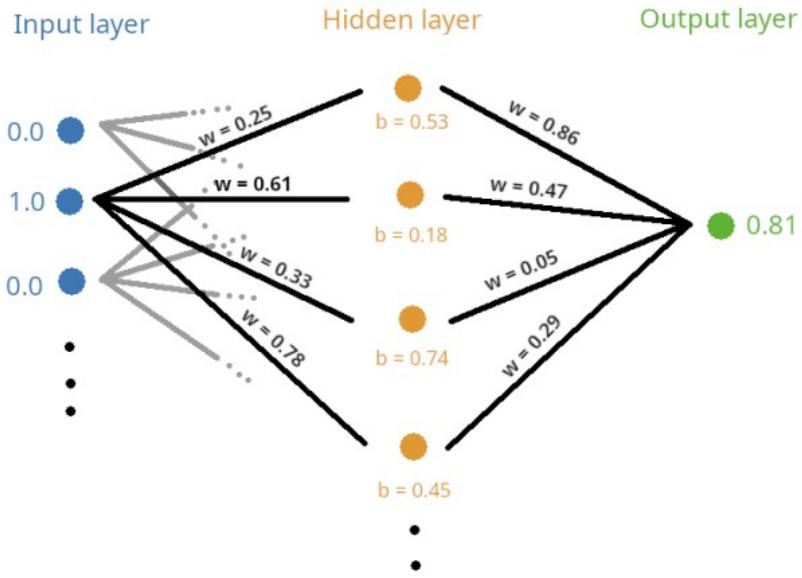
Konceptet

Et neuralt netværk består forskellige komponenter. For det første består neurale netværk af input- og output-parametre. For at kunne anvende et neuralt netværk til et problem, kræves det at problemet kan oversættes til inputparametre og at svaret til problemet kan repræsenteres i form af output-parametre.

Parametre i et neuralt netværk er floating-pointværdier, ofte med værdier mellem 0 og 1 eller -1 og 1, dog afhænger dette af implementationen. I dette tilfælde har parametre en værdi mellem 0 og 1, enderne inklusive. Det kræves altså, at problemet kan oversættes dvs. repræsenteres som et sæt af floating-point værdier.

Neurale netværk består af lag af neuroner. Inputs og outputs udgør hver sit lag. Mellem input- og output-laget er der gemte lag. Hver lag har assignerede bias'er, dvs. en enkelt værdier per neuron i lagene. Input-laget skiller sig ud, ved at den ikke har biaser.

Hver neuron er forbundet med alle neuroner både i laget før og laget efter. Hver forbindelse har en vægt, som ligesom bias er en værdi. Figur 7 skitserer et neuralt netværk.



Figur 7

Et neuralt netværk har et sæt operationer, som bruges i dets anvendelse. Disse består hovedsageligt af feed forward-operationen. Feed forward anvender netværket på et givet input. Feed forward udregnes med følgende beregning:

$$layer = sigmoid\left(layer_{i-1} \cdot weights_i + biases_i\right)$$

Denne beregning anvendes på alle lagene efter input-laget. Man ender derefter ud med output-parametrene.

Træningsdata består af input-parametre, hvor man også har de korrospenderende korrekte output-parametre. Dette kan anvendes til at måle, hvor godt netværket er. Dette kaldes cost. Cost-funktionen udregner fejl, kaldet loss, med følgende beregning:

$$error = output_{guess} - output_{corret}$$

For at gøre fejl mere markant, anvendes mean-square-error (MSE) oftest istedet. Denne beregnes med følgende beregning:

$$MSE = \frac{\sum error^2}{length_{data}}$$

Træning af et netværk tager udgangspunkt i et set træningsdata. Træningsalgoritmen forward-feeder input gennem netværket og udregner MSE for træningsdataet. Netværket bliver derefter vurderet ud fra MSE. Hvis MSE er for høj, ændre algoritmen vægtende i netværket, så netwærket producerer et andet resultat.

Anvende på fire-på-stribe

For at anvende en neuralt netværk-model på fire-på-stribe, kræves det, at spillet reduceres til input- og output-værdier. Vi konverterer boardet til inputværdier, ved at omdanne alle 42 felter til hver deres floating-point værdi. Værdien for hvert felt bliver 0,5, hvis feltet er tomt, 0,0 hvis feltet indeholder en gul (blå) brik og 1.0 hvis feltet indeholder en rød brik.

Outputparametrene anvendes til at vælge spillerens kolonne. Der er 7 outputparametre i alt, en til hver kolonne. Programmet vælger kolonnen med størst værdi, hvor kolonnen også er et gyldigt træk.

Implementering

Implementeringen tager udgangspunkt i Model-klassen. Denne klasse repræsenterer et generisk neuralt netværk. Klassen konstueres med en liste over størrelserne på hvert lag. Klassen indeholder primært 2 metoder: feed og mutate.

En instans af Model-klassen indeholder en liste af vægte og en liste af biaser; et sæt for hvert lag udover input-laget. Vægte for et lag er repræsenteret som en 2-dimensionel matrice. Biaser for et lag er repræsenteret som en 1-dimensionel matrice[i], med en bias for hver neuron (herfra node).

```
// src/nn_model.hpp:263
class Model {
public:
    Model(std::vector<size_t> layers);

    auto feed(const Mx1& input) -> Mx1;
    void mutate();
    // ...
    std::vector<size_t> m_layers;
    std::vector<Mx2> m_weights;
    std::vector<Mx1> m_biases;
};
```

Værdier i det neurale netværk er repræsenteret af matricer. Implementation definerer datatyper for 1- og 2-dimensionelle matricer. I begge typer gemmes de indre værdier i en sammenhængende datastruktur, specifikt har de C++-typen std::vector<double>. Typen Mx1 er en 1-dimensionel matrice. Den gemmer antallet af kolonner i matricen. Typen Mx2 er en 2-dimensionel matrice, som både gemmer antaller af rækker og kolonner. For hver matrice definerer implementationen de matematiske operationer, det neurale netværk benytter.

```
// src/nn_model.hpp:22
```

i Matricer er per definition 2-dimensionelle. Vektorer ville istedet være den korrekte betegnelse. Vi anvender begrebet af n-dimensielle matricer, da det passer meget godt med vores implementering af matematiske operationer. Yderligere kan man for 1-dimensionelle matricer, forestille sig en 2-dimensionel matrice med 1-kolonne og n-rækker[4].

```
class Mx1 {
public:
    // ...
    auto operator[](size_t col) const -> double
    {
        return m_data[col];
    }
    // ...
    void operator*=(const Mx1& rhs)
    {
        ASSERT_EQ(m_cols, rhs.m_cols);
        for (size_t i = 0; i < m_data.size(); ++i)
            m_data[i] *= rhs.m_data[i];
    }
    // ...
private:
    size_t m_cols;
    std::vector<double> m_data;
};
```

Modellen anvendes og trænes med feed forward-operationen. Denne operation er implementeret i Model-klassens feed-metode. Metoden anvender beregningen, som beskrevet ovenfor. Implementationen benytter de implementerede matriceoperationer via. operator overloading. For at undgå kopieret data, laver implementationen in-place-operationer, dvs. den originale matriceværdi bliver muteret.

```
// src/nn_model.hpp:99
auto Model::feed(const Mx1& inputs) -> Mx1
{
    auto outputs = inputs;
    for (size_t i = 0; i < m_layers.size() - 1; ++i) {
        auto l1 = m_weights[i].dot(outputs);
        l1 += m_biases[i];
        l1.apply(sigmoid);
        outputs = l1;
    }
    return outputs;
}
```

Til træning af modellen benyttes mutation. Mutation er en operation, hvorpå tilfældig bliver tilført netværkets vægte og biaser. Mutation er implementeret ved at generere en matrice for hver lags vægte og biaser. Hver matrice bliver fyldt med tilfældige værdier. Værdierne er genereret med en bestemt karakteristik, som i dette tilfælde er, at 50% af værdierne er 0, 25% er mellem -2.0 og 0 og 25% er mellem 0 og 2.0. De 50% der ikke er 0, generes efter en approximeret normalfordeling med gennemsnit på 0. Mutationen ganges på alle vægte og biaser med en bestemt learning rate, som kan justeres på, for at ændre træningskarakteristikken.

```
// src/nn_model.hpp:128
for (auto& layer_weights : m_weights) {
    auto mutation = Mx2(layer_weights.rows(), layer_weights.cols());
    mutation.apply(mutation_dec);
    mutation *= learning_rate;
```

```
    layer_weights += mutation;
}
```

Træning

Neurale netværk er anderledes fra decision tree-modeller i forhold til træning, i det neurale netværk er mindre egnede til reinforcement learning. Det der ligger til grund for dette, er måden hvorpå de to modellers hukommelse. I decision tree-modeller, er der en en-til-en-sammenhæng mellem et træk i spillet og en entry i beslutningstræet. Dette gør, at modellen relativt hurtigt får grundlæggende kompetencer for problemstillingen. Neurale netværk har ikke samme en-til-en-sammenhæng. Dette betyder, at modellen vil være relativt langsom, til at nå et grundlæggende niveau.

Reinforcement learning, specielt[ii] i vores tilfælde, kræver at modellen af sig selv får et grundlæggende kompetenceniveau. Dette er, fordi de spil modellen spiller også er modellens træningsdata. Dvs. træningsdataens kvalitet afhænger af modellens niveau.

Vi har forsøgt at træne en neural netværk (NN)-model med reinforcement learning på samme måde som decision tree-modellen. Modellen har 4 lag, af 42, 42, 18 og 6 nodes. De 42 og 7 lagstørrelser kommer fra de 42 felters input og 7 kolonners output. Resultaterne reflekterede ingen sammenhæng mellem input og output. Vi vurderer, at træningsmekanismen ikke er anvendelig på sådanne NN-model.

Traditionelt trænes NN-modeller på et datasæt, hvor man designere en del som træningsæt og en anden del som testsæt[5]. Dette fungerer, at anvende gradient descent, til at træne modellen med træningsdataet, dvs. træning som afhænger cost-funktionen.

Ressourceforbrug

NN-modellen i denne situation har et markant mindre ressourceforbrug, sammenlignet med decision tree (DT)-modellen, med antagelsen at modellen kan trænes, til at være konkurrencedygtig. En forskel på NN-modeller og DT-modeller, er at DT-modeller, som vi har set tidligere, stiger i memory-footprint. Dette begrænser mængden af træning, det er muligt at udføre i et givet miljø. Yderligere gør det også modellen langsomere ift. afviklingstid. Det tager længere tid både at træne, men også at anvende modellen.

NN-modellen er derimod statisk. NN-modellen ændre sig ikke med træning, da der ikke bliver tilføjet eller fjernet lag, og mængden af nodes i hver lag ændres hellller ikke. Derved er modellen

ii Måske også specifikt i vores tilfælde. Kommende an på reward-systemer kan neurale netværk sansynligvis anvendes til reinforcement learning, men dette er udenfor forfatternes viden.

lige hurtig ved iteration 1 såvel som iteration 1.000.000. Dette kan gøre NN-modellen mere anvendelig i forskellige situationer, som vi ikke vil undersøge yderligere.

Diskussion

Manglen på træningsdata reducerede mængden af mulige konklusioner markant. Der er forskellige måder, hvorpå træningsdata kan anskaffes. En mulighed er at undersøge offentlige databaser, som opbevare dataer for spillede spil. Det kritiske for sådanne data, er at det indeholder forskellige identificerbare spilstadier, og så med det korrekte træk i hver situation. En anden mulighed er at generere data ud fra en spille-algoritme. Dvs. en algoritme der er implementeret, til at beslutte korrekte træk analytisk. Den åbenlyse algoritme til dette formål er Minimax-algoritmen[6]. Anvendelsen ville så være, at minimax-algoritmen spiller et antal spil, mod en magen til instans. Hver spilstadie gemmes sammen med trækket som algoritmen beslutter. Siden minimax-algoritmen opererer analytisk, behøver denne datagenerering kun at foregå én gang.

NN-modellen er implementeret med tilfældig mutation til træning. Mere effektive metoder eksistere, til at ændre modellen ift. træning såsom backpropagation[7]. Denne metode analyserer hvilke vægte og biaser, der bør ændres, for at forbedre resultatet. Dette gør den ud fra cost-funktionen, dvs. en sammenligning af modellens resultat med det korrekte resultet. Yderligere undersøgelser, kunne udforske forskellen på tilfældig mutation og backpropagation. Både i forhold til effektivitet og anvendelighed, men også i forhold til kompleksitet af implementering.

DT-modellen er implementeret, så den er nødt til at løbe gennem alle kombinationer, for at blive trænet til potientiale. Metoder såsom alpha-beta-pruning eksisterer, hvis formål er at undgå mængden af kombinationer, modellen er nødt til at træne[8]. Yderligere undesøgelse kunne udforske alpha-beta-pruning i lignende implementation. Her er det både relevant at undersøge ressourceforbruget og det totale niveau, for en træningsperiode. Men det kan også være relevant, at undersøge forskelle på læringskarakteriska mellem en model med og uden pruning.

Perspektivering

Machine learning kan anvendes til løsning af mange moderne problemer. Ofte er ML-modellers effektivitet og anvendelighed begrænset af begrænsinger af ressourcer. Dette kan både være ressourcer ift. regnekraft og tid. Dette gælder både for træning og anvendelse af modeller. Det er populært, at bygge intelligens ind i små devices. Her er regnekraften oftest begrænset. Der er også

populært, at træne modeller på lejet computerkraft via. cloud-computing. I disse tilfælde er minimering af ressourceforbrug fordelagtigt.

For at kunne arbejde med ressourceforbrug, er det nødvendigt med viden om forskellige typer af ML-modeller, og hvad forskellen er på dem. Derudover er det nødvendigt med viden om, hvordan forskellige typer af modeller dels kan trænes, og hvordan de kan anvendes.

Efter man har valgt en type af model på teoretisk grundlag, kræves der derudover, at man kan træffe fordelagtige implementeringsbeslutninger på praktisk grundlag, for at minimere ressourceforbruget.

Kilder

1. Neil Sloane, ed. (2012). "Number of legal 7 X 6 Connect-Four positions after n plies". Online Encyclopedia of Integer Sequences. sequence A212693. Retrieved 2023-02-12.
2. Aho, Lam, Sethi, and Ullman. "Compilers: Principles, Techniques & Tools" 2nd ed. Pearson Education, Inc. 2007
3. S. Ponnusamy and M. Khoje, "Optimizing Cloud Costs with Machine Learning: Predictive Resource Scaling Strategies," 2024 5th International Conference on Innovative Trends in Information Technology (ICITIIT), Kottayam, India, 2024, pp. 1-8, doi: 10.1109/ICITIIT61487.2024.10580717.
4. Lang, Serge (2002), Algebra, Graduate Texts in Mathematics, vol. 211 (Revised third ed.), New York: Springer-Verlag, ISBN 978-0-387-95385-4, MR 1878556
5. Bishop, Christopher M. (2006-08-17). Pattern Recognition and Machine Learning. New York: Springer. ISBN 978-0-387-31073-2
6. Maschler, Michael; Solan, Eilon; Zamir, Shmuel (2013). Game Theory. Cambridge University Press. pp. 176–180. ISBN 9781107005488.
7. Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986a). "Learning representations by back-propagating errors". Nature. 323 (6088): 533–536. Bibcode:1986Natur.323..533R
8. Russell, Stuart J.; Norvig, Peter. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Hoboken: Pearson. ISBN 9780134610993. LCCN 20190474.

Bilag

Bilag 1 – Decision tree-træning med forskellige iterationer

iterations 1.000

entries	**bytes**	**time** (s)
9.998	619.876	0,026
10.000	620.000	0,026
9.695	601.090	0,015
9.748	604.376	0,015
9.695	601.090	0,026
9.748	604.376	0,026
9.970	618.140	0,03
10.006	620.372	0,03
9.970	618.140	0,024
10.006	620.372	0,024
9.906	614.172	0,025
9.360	616.032	0,025

iterations 10.000

entries	**bytes**	**time** (s)
90.912	5.636.544	0,163
90.954	5.639.148	0,163
90.912	5.636.544	0,171
90.954	5.639.148	0,171
91.068	5.646.216	0,17
91.175	5.652.850	0,17
91.837	5.693.894	0,176
91.798	5.691.476	0,176

iterations 100.000

entries	**bytes**	**time** (s)
800.246	49.615.252	2,138
793.239	49.180.818	2,138
802.220	49.737.640	2,099
802.532	49.756.984	2,099
809.317	50.177.654	2,093
808.896	50.151.552	2,093
819.654	50.818.548	2,125
821.238	50.916.756	2,125

iterations 1.000.000

entries	**bytes**	**time** (s)
7.299.705	452.581.710	22,764
7.274.664	451.029.168	22,764
7.279.936	451.356.032	22,468
7.292.180	452.115.160	22,468
7.104.551	440.482.162	22,163
7.019.558	435.212.596	22,163
7.051.239	437.176.818	22,127
7.062.242	437.859.004	22,127