Kunstmatige Intelligentie 2008 — opdracht 3

Neurale netwerken

Rick van der Zwet
<hvdzwet@liacs.nl>

LIACS
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

9 april 2008

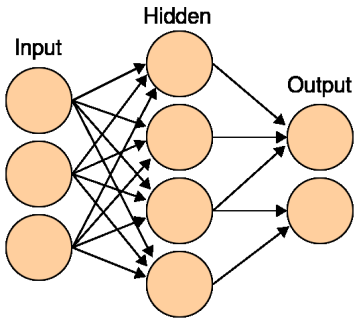
1 Inleiding

Dit verslag gaat over de derde programmeer-opgave van het vak kunstmatige intelligentie [1]. Welke als opdracht heeft om een neuraal netwerk te bouwen, wat met één verborgen laag Poker handen kan classificeren. Dit alle met met als doel de bouw en werking van neurale netwerken (NN) uit het college boek [3] beter te kunnen begrijpen.

2 Uitleg probleem

Om het probleem een visueel gestalte te geven is gekozen om een pokerhand te bepalen. Als invoer wordt er 5 kaarten uitgegeven, waarvan de waarde en kleur apart gecodeerd zijn in respectievelijk 13 en 4 mogelijke waardes.. Met deze kaarten kan er een van de 10 combinaties van een pokerhand gevormd worden. Als er 5 willekeurige kaarten aangeboden worden is het de taak van het NN daar de juiste combinatie bij te zoeken. Er wordt geen informatie gegeven wat de regels zijn om de combinaties te maken, maar er worden wel voorbeelden gegeven van valide combinaties. Nadat alle voorbeelden gezien

zijn worden er willekeurige combinaties kaarten aangeboden, de taak is om deze set zo goed mogelijk te clasifiseren.



Figuur 1: Voorbeeld van een 1-laags neuraal netwerk met 2 invoer nodes, 3 verbonden perceptronen, 2 uitvoer nodes.

3 Theorie

Voor de menselijke beleving is het relatief makkelijk om in één opzicht te zien welke pokerhand er in hand is of om dit aan te leren, door voorbeelden aan te bieden, waarbij dan logische regels worden afgeleid. Om in een algoritme hetzelfde *lerende* gedrag te simuleren/creëren zijn NN uitgevonden in verschillende smaken en stijlen. Dit verslag focust zich in feed-forward netwerken welke als voordeel hebben dat ze een van de simpelste neurale netwerken zijn, waarbij de informatie maar een richting op gaat.

Bij een NN kan uit 3 of meerdere lagen onderscheiden worden. Als eerste is er de *invoer* laag met zijn invoer nodes, waar de gecodeerde invoer op komt te staa. Bij gecodeerd wordt een waarde tussen 0 en 1 bedoeld de reden hiervan komt later aan de orde. De *verborgen* laag kan uit meerdere lagen bestaan, maar tijdens dit verslag zal focust worden op één laag. In deze verborgen laag zitten de *perceptronen*, welke de intelligentie zijn van het NN. In de *uitvoer* laag zitten de uitvoer nodes, welke hun waarde weer tussen 0 en 1 bevinden. Tussen alle nodes in de invoer laag en de uitvoer laag zitten takken, zo ook tussen de verborgen laag en de uitvoer laag. Zie ook figuur 1.

Een perceptronen heeft de eigenschap dat het een *activatie* functie bezit en een drempel waarde. Stel dit voor als een persoon die je slaat, als iemand dit zachtjes doet zal je niets zeggen, als dit een stuk harder is zal je 'auwch' roepen. De kracht die het nodig om je auwch te laten roepen is de drempelwaarde. Van binnen voelde dat dit pijn deed naar mate je harder werd

geslagen ging je meer pijn voelen. Dit kan je het beste vatten in een lineaire functie. Vanwege het feit dat een NN een continue functie is ligt het voor de hand om een logaritmische functie te krijgen, waarbij de activatie functie de vorm krijgt van formule 1. Dit verklaart tevens waarom de invoer en uitvoer tussen 0 en 1 moeten liggen, gezien dit ook het uitvoer bereik is.

$$uit = \frac{1}{1 + e^{in}} \tag{1}$$

De *in* in formule 1 is een gewogen invoer van alle *inkomendetakken* $*$ *hetgewichtvaneentak*. De drempel waarde is echter een vervelend feit, welke makkelijker op te lossen is met een zogenoemde *bias* knoop, de waarde op deze knoop is altijd -1, waardoor deze samen met het gewicht was tussen deze bias knoop en de knoop hangt zorgt voor de drempelwaarde die origineel gebruikt was nu de drempelwaarde 0 gebruikt kan worden.

De gewichten zijn het 'magische' van het NN, deze gewichten kunnen namelijk getraind worden waardoor de uitvoer van het neurale netwerk kan veranderen. Deze training word *backpropagation* genoemd en heeft een correlatie met de fout in de uitvoer, het huidige gewicht en de leer-snelheid en de invoer. Om deze fout in een koop te bepalen moet er gekeken worden naar de fout in de uitvoer en het gewicht naar tak van die knoop, zie formule 2. Waarbij $\triangle_{uit}$ gelijk is aan $\triangle_j = g'(in_j) \sum_i W_{j,i} \triangle i$ Waarbij i all opvolgende knopen van j zijn.

$$W_{in,uit} = \alpha * a_{in} * \triangle_{uit} \tag{2}$$

De fouten worden dus omgekeerd berekend, eerst de fouten van de uitvoer laag, dan de verborgen lagen ervoor en als die zijn die van de invoer laag.

4 Aanpak

Er is gekozen gebruik te maken voorbeeld implementatie raamwerk gegeven in de sheet [2] tijdens het college kunstmatige intelligentie welke een pseudo beschrijving geeft voor het programmeren van een neuraal netwerk.

```
herhaal
    voor elke E in trainings set doe
        voorzie de invoer knopen
        bereken de waardes in de perceptronen en uitvoer-knopen
        bereken de delta fouten van de knopen en perceptronen
        pas de takken aan
totdat netwerk "geconvergeerd"
```

Als eerste stap is simpele Perl code gebruikt om van de training en validatie sets een in- en uitvoer data voor het NN te schrijven waarbij alle waarden tussen 0 en 1 liggen. Perl code is een stuk flexibeler als het gaat om tekst verwerking dan C code.

Om ook eens met een kritische noot naar de invoer te kijken en de daadwerkelijke intelligentie van een NN, zal er extra knopen toegevoegd worden om te kijken hoe deze presteren. Voor de poker set zijn dit een paar specifieke combinaties namelijk de setjes -doubles, triples, four- en de aantallen kaarten van elke soort kleur.

Nadat het netwerk geschreven is kan deze getest worden met een het leren van de zogenoemde XOR functie. Deze functie is niet lineair te scheiden, maar is met een NN van 2 invoer, 3 verborgen, 1 uitvoer perfect te berekenen/leren.

5 Implementatie

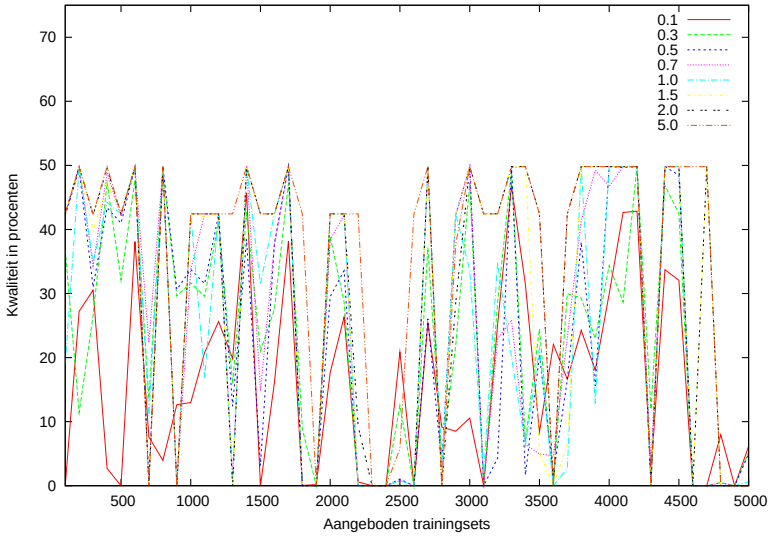
Het NN is geschreven in C, hevig hangend op globale array waar de data in zit en losse functies om de programma flow duidelijker te maken. De pre-parse is geschreven in Perl en Bourne shell code.

6 Experimenten

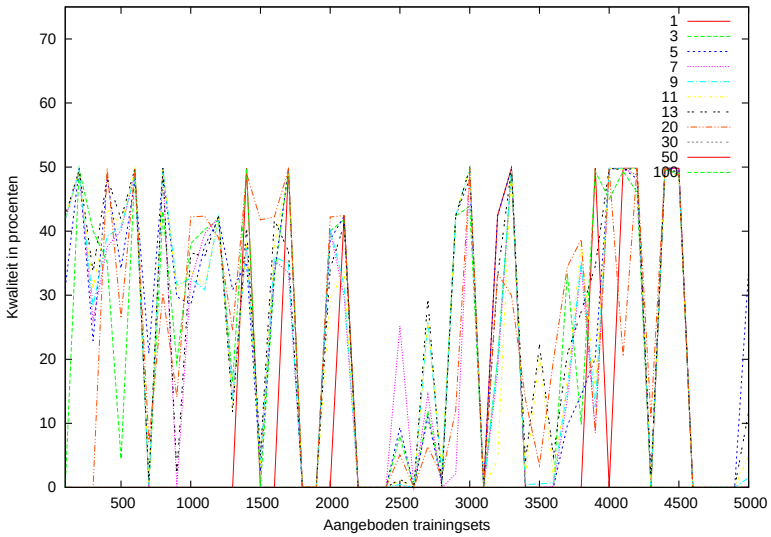
Alle testen zijn uitgevoerd met een computer, waarbij gcc 4.01 als compiler aanwezig was en Perl 5.8.8 als parser . Bij wijziging van de input knopen of uitvoer knopen, is de pre-parse aangeroepen en de code opnieuw gecompileerd met de nieuwe waardes.

Tijdens alle simulaties het de trainingsset een grootte van 25000. De eind validatie set -welke de eind kwaliteit van het NN bepaald- 1 miljoen. En de tussentijdse kwaliteit set heeft een grootte van 1000, welke na elke 100 training invoeren draaien.

Tijdens de leersnelheid experiment is gebruik gemaakt van het de standaard poker set, met 10 invoeren, 10 uitvoeren en 10 verborgen perceptronen. De resultaten zijn te zien in figuur 2. De verborgen perceptronen test is gedaan om dezelfde data set, hierbij is de leer-snelheid 0.5 gekozen, deze resultaten staan in figuur 3.



Figuur 2: NN training verloop bij variabele leersnelheden



Figuur 3: NN training verloop bij variabele verborgen invoer laag grootte

Tabel 1: Prestaties NN bij verschillende invoer data types

Invoer variatie	Maximale kwaliteit in %	Voorbeelden nodig
1	49.92	400
2	49.82	600
3	92.26	1800
4	92.26	6800
5	92.26	600

Tijdens de invoer variatie is gekozen voor vijf verschillende combinaties, de randvoorwaarden zijn 10 uitvoer knopen, aantal verborgen knopen 20, leersnelheid 0.5 De standaard set hierna afgekort met *STD* (1). STD met kleur tellingen (2), STD met paar tellingen (3), STD met kleur en paar tellingen (4), de paar tellingen + kleur tellingen (5) . Resultaten staan in tabel 1.

7 Conclusie

Er is gekeken naar verschillende aspecten van het neurale netwerk. Heeft toevoegen van nieuwe knopen met extra informatie invloed op de uitvoer? Heeft de leer-snelheid een positief effect op het NN? Heeft het aantal verborgen perceptronen invloed op het netwerk? Dit alle gecombineerd tot het originele probleem: Is het mogelijk goed poker-handen te voorspellen aan te hand van een 1-laags NN.

Het variëren van leer-snelheid levert geen beter kwaliteit NN op, als de leer-snelheid echter tussen 0 en 1 ligt zal het verloop minder grillig zijn en zal het NN niet 'te ver' doorschieten in een bepaalde richting.

Het variëren van het aantal verborgen knopen levert een maximum op als het aantal verborgen knopen groter of gelijk is aan het aantal invoer knopen. In het kader van de efficient is het aan te raden, deze rond de waarde van de invoer knopen te houden om zo het aantal berekeningen dat uitgevoerd moet worden zo minimaal mogelijk te houden.

Het grillige verloop van de grafiek in figuur 2 als ook in figuur 3 is een gedag wat niet verklaard kan worden. Verder onderzoek met bijvoorbeeld een andere data-set of de data in een andere volgorde aan te bieden zou hier misschien uitsluitsel voor kunnen bieden.

Het variëren van de invoer heeft groot effect op te effectiviteit van het netwerk. Bij het kiezen van de juiste data verdubbeld het netwerk zijn effectiviteit. Dit kan verklaard worden doordat op het moment de paren van te

voren bepaald worden deze intelligentie niet meer door het systeem ontdekt hoeft te worden. Het probleem wordt dus eigenlijk versimpeld. Wel is het noodzakelijk relevante versimpelingen te maken. Het tellen van het aantal kaarten van een bepaalde kleur heeft duidelijk geen enkel effect in het netwerk, wat logisch is als we dichter naar de poker kaarten gaan kijken. Hiervoor is het enkel interessant om te weten of er 5 dezelfde kleuren in zitten of niet. Elke andere tussenvorm levert geen extra informatie op.

Voor een vervolg verslag zou gekeken kunnen worden of een meer laags NN een beter resultaat boekt bij het ontdekken van de poker kaarten. Verder zouden er ook 'debug' sets ontwikkeld kunnen worden waarbij een willekeurig NN netwerk getest kan worden op juiste werking, gegeven alle randvoorwaarden. Deze optie wordt momenteel enkel voor een XOR voorbeeld gegeven welke enkel maar een validatie set is van een zeer beperkt probleem. Het is namelijk niet aan te tonen of het netwerk fout geimplementeerd is.

Referenties

[1] W.A. Kosters, Kunstmatige intelligentie Programmeer-opgave 3 van 2008 – Neurale netwerken, `http://www.liacs.nl/~kosters/AI/nn08.html`

[2] W.A. Kosters, Kunstmatige intelligentie College neurale netwerken 2008, `http://www.liacs.nl/~kosters/AI/neuraal.pdf`

[3] S.J. Russell en P. Norvig, Artificial Intelligence, A Modern Approach, Second ediion, Prentice Hall, 2003.

[4] Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository `http://www.ics.uci.edu/~mlearn/MLRepository.html`. Irvine, CA: University of California, School of Information and Computer Science.

[5] Robert Cattral (cattral@gmail.com) and Franz Oppacher (oppacher@scs.carleton.ca) Carleton University, Department of Computer Science, Intelligent Systems Research Unit `http://archive.ics.uci.edu/ml/datasets/Poker+Hand`

Appendix

De NN code en pre-parse.pl code zagen er als volgt uit:

```
/*
 * Rick van der Zwet
 * 0433373
 * OS Assigment 3
 * Licence: BSD
 * $Id: nn.c 555 2008-04-07 21:59:55Z rick $
 */

#include <sysexits.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/* NOTE: All first knobs are bias knobs or hidden stale knobs
 * - Validation is done using rounding, please make outputs discrete or
 * alter validation function
 */

/* Allow uniform and easy calls at functions */
#define TRUE 1
#define FALSE 0


/* Network variables */
/*NOTE: first node is 'hidden' bias knob */
#ifndef INPUT_SIZE
#define INPUT_SIZE 11
#endif

/*NOTE: first node is 'hidden' bias knob */
#ifndef HIDDEN_SIZE
#define HIDDEN_SIZE 11
#endif

/*NOTE: first node is 'hidden' 'lame' knob */
#ifndef OUTPUT_SIZE
#define OUTPUT_SIZE 11
#endif

/* Learn speed alpha of network */
#ifndef LEARN_SPEED
#define LEARN_SPEED 0.5
#endif

/* After QUALITY_ROUND trainingset check quality of network */
#define QUALITY_ROUND 100

/* Training set, to be used to train network */
char * file_training = "data/training.txt";
/* Validation set, to be used to test end result of network */
char * file_validate = "data/validate.txt";
/* Quality set, to be used to do quick testing whether network is
 * improving
 */
char * file_quality = "data/quality.txt";

/* Globally defined arrays, which represent the network */
double hidden[HIDDEN_SIZE];
double input[INPUT_SIZE];
double output[OUTPUT_SIZE];
double target[OUTPUT_SIZE];
double weight_HtoO[HIDDEN_SIZE][OUTPUT_SIZE];
double weight_ItoH[INPUT_SIZE][HIDDEN_SIZE];

#define WEIGHT_NOT_USED -99999

void stdInit() {
    int i;
    /* Should never change, been using */
    for (i = 0; i < INPUT_SIZE; i++)
        weight_ItoH[i][0] = WEIGHT_NOT_USED;
    for (i = 0; i < HIDDEN_SIZE; i++)
        weight_HtoO[i][0] = WEIGHT_NOT_USED;
}


/* Random init of weights */
void randInit() {
    int i,j;

    /* Different numbers every call */
    srandom(time(NULL));

    for (i = 0; i < INPUT_SIZE; i++)
        for ( j = 1; j < HIDDEN_SIZE; j++) {
            weight_ItoH[i][j] = (double)(random() % 100) / 100;
        }

    for (i = 0; i < HIDDEN_SIZE; i++)
        for (j = 1; j < OUTPUT_SIZE; j++)
            weight_HtoO[i][j] = (double)(random() % 100) / 100;

    stdInit();
}

/* Fixed init of weights */
void fixedInit() {
    int i,j;
    for (i = 0; i < INPUT_SIZE; i++)
        for ( j = 1; j < HIDDEN_SIZE; j++) {
            weight_ItoH[i][j] = 0.5;
        }

    for (i = 0; i < HIDDEN_SIZE; i++)
        for (j = 1; j < OUTPUT_SIZE; j++)
            weight_HtoO[i][j] = 0.5;

    stdInit();
}

/* Define exact wights, used for debugging calculations
 * other flags INPUT = 2, HIDDEN = 2, OUTPUT = 1
 */
void debugInit() {
    stdInit();
    weight_ItoH[0][1] = 1;
    weight_ItoH[0][2] = 1;
    weight_ItoH[1][1] = 0.62;
    weight_ItoH[1][2] = 0.42;
    weight_ItoH[2][1] = 0.55;
    weight_ItoH[2][2] = -0.17;

    weight_HtoO[0][1] = 1;
    weight_HtoO[1][1] = 0.35;
    weight_HtoO[2][1] = 0.81;
}

/*  calculate Aj's and Ai's (outputs) */
void nnCalc() {
    int i,j;
    double total;
    for (i = 1; i < HIDDEN_SIZE; i++) {
        total = 0;
        for (j = 0; j < INPUT_SIZE; j++)
            total += weight_ItoH[j][i] * input[j];
        hidden[i] = 1 / ( 1 + exp(total * (-1)));
    }

    for (i = 1; i < OUTPUT_SIZE; i++) {
        total = 0;
        for (j = 0; j < HIDDEN_SIZE; j++)
            total += weight_HtoO[j][i] * hidden[j];
        output[i] = 1 / ( 1 + exp(total * (-1)));
    }

}

/* train network, NOTE: nnCalc needs to be called first */
void nnTrain() {
    int i,j;
    double hidden_delta[HIDDEN_SIZE];
    double output_delta[OUTPUT_SIZE];
    double output_error[OUTPUT_SIZE];
    double hidden_sum_delta[HIDDEN_SIZE];

    for (i = 1; i < OUTPUT_SIZE; i++) {
        output_error[i] = target[i] - output[i];
        output_delta[i] = output_error[i] * output[i] * (1 - output[i]);
    }

    for (i = 0; i < HIDDEN_SIZE; i++) {
```

```
        hidden_sum_delta[i] = 0;
        for (j = 1; j < OUTPUT_SIZE; j++)
            hidden_sum_delta[i] += weight_HtoO[i][j] * output_delta[j];
        hidden_delta[i] = hidden[i] * (1 - hidden[i]) *
        hidden_sum_delta[i];
     }

    for (i = 0; i < HIDDEN_SIZE; i++)
        for (j = 1; j < OUTPUT_SIZE; j++) {
            weight_HtoO[i][j] = weight_HtoO[i][j] + LEARN_SPEED *
            hidden[i] * output_delta[j];
        }

    for (i = 0; i < INPUT_SIZE; i++)
        for (j = 1; j < HIDDEN_SIZE; j++) {
            weight_ItoH[i][j] = weight_ItoH[i][j] + LEARN_SPEED *
            input[i] * hidden_delta[j];
        }
}

/* Verify wether target, matches output */
int nnValidate() {
    int i;
    //printf ("Rounding: %lf - %lf\n",output[1], target[1]);
    for (i = 1; i < OUTPUT_SIZE; i++)
        if (round(output[i]) != round(target[i]))
            return FALSE;
    return TRUE;
}

/* Pretty print of output */
void nnOutput() {
    int i;
    for(i = 0; i < INPUT_SIZE; i++)
        printf("%lf, ", input[i]);
    printf("= %lf - %lf - ", output[1], target[1]);
    if (nnValidate() == TRUE)
        printf("OK");
    else
        printf("ERROR");
    printf("\n");
}

/* Pretty print of hidden knobs */
void nnHiddenOutput() {
    int i;
    for(i = 0; i < HIDDEN_SIZE; i++)
        printf("%lf, ", hidden[i]);
    printf(" - HIDDEN\n");
}


/* Pretty print of all weights */
void nnNeuronOutput() {
    int i,j;
    for (i = 0; i < INPUT_SIZE; i++)
        for(j = 0; j < HIDDEN_SIZE; j++)
            if (weight_ItoH[i][j] != WEIGHT_NOT_USED)
                printf("weight_ItoH[%i][%i] = %lf\n", i, j,
                weight_ItoH[i][j]);
    printf("---\n");
    for (i = 0; i < HIDDEN_SIZE; i++)
        for(j = 0; j < OUTPUT_SIZE; j++)
            if (weight_ItoH[i][j] != WEIGHT_NOT_USED)
                printf("weight_HtoO[%i][%i] = %lf\n", i, j,
                weight_HtoO[i][j]);
}

int nnReadInput(FILE * handle) {
    int i = 1;
    double finput;
    while (fscanf(handle, "%lf", &finput) != EOF) {
        if (i < INPUT_SIZE)
            input[i] = finput;
        else if (i < (INPUT_SIZE + OUTPUT_SIZE))
            target[i - INPUT_SIZE] = finput;

        /* Calc next input */
        i++;
        /* Skip hidden output knob */
        if (i == INPUT_SIZE)
            i++;
        if (i == (INPUT_SIZE + OUTPUT_SIZE))
            return TRUE;
    }

    /* Input not complete */
    return FALSE;
}

/* Verify quality of current network */
double nnQualityCheck(char * file) {
    double validate_total = 0;
    double validate_ok = 0;
    double validate_percent = 0;
    FILE * handle;

    handle = fopen(file,"r");
    while (nnReadInput(handle) == TRUE) {
            validate_total++;
            nnCalc();
            if (nnValidate() == TRUE)
                validate_ok++;
            //else
            //  nnOutput();
    }
    fclose(handle);
    validate_percent = (validate_ok / validate_total) * 100;
    printf("Validating: %.0lf/%.0lf - %.2lf %%\n",
    validate_ok,validate_total,validate_percent);

    return(validate_percent);
}

/* Main program */
int main (int argc, char * argv[]) {
    int i,training_total, training_best;
    double quality_max, quality;
    FILE * handle;

    /* Set the bias knob */
    input[0] = -1;
    hidden[0] = -1;

    /* Init set of all wights */
    //debugInit();
    fixedInit();
    //randInit();

    /* Set initial quality */
    quality_max = nnQualityCheck(file_quality);
    training_best = 0;
    training_total = 0;

    printf("Running neural network with following parameters\n");
    printf("Input  nodes    : %i\n", INPUT_SIZE);
    printf("Hidden nodes    : %i\n", HIDDEN_SIZE);
    printf("Output nodes    : %i\n", OUTPUT_SIZE);
    printf("Learning rate   : %lf\n", LEARN_SPEED);
    printf("Quality check   : %i\n", QUALITY_ROUND);
    printf("Initial quality : %lf %%\n", quality_max);
    /* Start training */
    //nnNeuronOutput();
    i = 1;
    handle = fopen(file_training,"r");
    while ( nnReadInput(handle) == TRUE) {
        training_total++;
        nnCalc();
        //nnOutput();
        //nnHiddenOutput();

        if (nnValidate() == FALSE) {
            nnTrain();
            //nnNeuronOutput();
        }

        /* Verifiy quality, stop training when quality is going down */
        if ((training_total % QUALITY_ROUND) == 0) {
            printf("Learned: %i - ", training_total);
            quality = nnQualityCheck(file_quality);
            if (quality > quality_max) {
                quality_max = quality;
                training_best = training_total;
            }
        }
    }
    fclose(handle);
    printf("Max quality: %.2lf%% at training round: %i\n", quality_max,
        training_best);
    quality = nnQualityCheck(file_validate);
    return(EX_OK);
}
```

```
#!/usr/bin/env perl -w

while ($line = <STDIN>) {
    #Input in way more every workable format
    @numbers = split(',',$line);

    #Get the number of every colour in on every line
    @colour = (0, 0, 0, 0, 0);
    foreach (0,2,4,6,8) {
        $colour[$numbers[$_]]++;
    }

    #Find pairs, triples, four
    @cards = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    foreach (1,3,5,7,9) {
        $cards[$numbers[$_]]++;
    }

    ### BEGIN Specials ###
    #triple = 1, four = 2
    $special = 0;
    #one pair = 1, two pair = 2
    $pairs = 0;
    ### END Specials ###

    ### BEGIN decoded specials ###
```

```
027:     $onepair = 0;
028:     $twopair = 0;
029:     $triple = 0;
030:     $four = 0;
031:     ### END decoded specials ###
032:
033:     foreach (1 .. 13) {
034:         if ($cards[$_] == 2) {
035:             $pairs++;
036:             if ($pairs == 1) {
037:                 $onepair = 1;
038:             } elsif ($pairs == 2) {
039:                 $twopair = 1;
040:             }
041:         } elsif ($cards[$_] > 2) {
042:             $special = $cards[$_];
043:             if ( $special == 3) {
044:                 $triple = 1;
045:             } elsif ($special == 4) {
046:                 $four = 1;
047:             }
048:         }
049:     }
050:
051:     #Decode into split option array
052:     @output = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
053:     $output[$numbers[10]]++;
054:
055:     $pairs /= 10;
056:     $special /= 10;
057:     foreach (0 .. 10) {
058:         $numbers[$_] /= 10;
059:     }
060:     foreach (0 .. 4) {
061:         $colour[$_] /= 10;
062:     }
063:
064:     # Input 10  : all cards
065:     # Output 1  : decoded version of result
066:     #print "@numbers[0..10]\n";
067:
068:     # Input 10  : all cards
069:     # Output 10 : all options, one node
070:     print "@numbers[0..9] @output[0..9]\n";
071:
072:     # Input 14  : all cards + colours
073:     # Output 10 : all options, one node
074:     #print "@numbers[0..9] @colour[1..4] @output[0..9]\n";
075:
076:     # Input 16  : all cards + specials
077:     # Output 10 : all options, every has it's one node
078:     #print "@numbers[0..9] @colour[1..4] $pairs $special @output[0..9]\n";
079:
080:     # Input 14  : all cards + decoded specials
081:     # Output 10 : all options, every has it's one node
082:     #print "@numbers[0..9] $onepair $twopair $triple $four @output[0..9]\n";
083:
084:     # Input 18  : all cards + colour + decoded specials
085:     # Output 10 : all options, every has it's one node
086:     #print "@numbers[0..9] @colour[1..4] $onepair $twopair $triple $four @output[0..9]\n";
087:
088:     # Input 10: all cards
089:     # output 2: full house, four of a kind
090:     #print "@numbers[0..9] $output[6] $output[7]\n";
091:
092:     # Input 16 : all cards + specials
093:     # output 1 : full house
094:     #print "@numbers[0..9] @colour[1..4] $pairs $special $output[7]\n";
095:
096:     # Input 2  : Calculated set
097:     # output 1 : full house
098:     #print "$pair $special $output[6]\n";
099:
100:     # Input  8  : colour + decoded specials
101:     # Output 10 : all options, every has it's one node
102:     #print "@colour[1..4] $onepair $twopair $triple $four @output[0..9]\n";
103: }
```