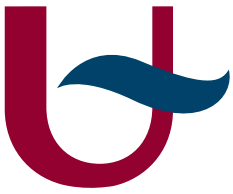


Universiteit Antwerpen

Faculteit Wetenschappen
Departement Fysica

Academiejaar 2017-2018

Herkennen van handgeschreven karakters

Een toepassing van Machine Learning: het feed forward neural network

gesuperviseerd door

Prof. Dr. Nick Van Remortel
Dr. Yamiel Abreu
Davide Di Croce

Verslag Examenopdracht Programmeren voor Fysici

Inhoudsopgave

1 Inleiding

Als examenopdracht voor het vak *programmeren voor fysici* werd gekozen voor het implementeren van een neuraal netwerk dat in staat is om, door middel van backpropagation, handgeschreven cijfers te identificeren. Om dit te realiseren werd de klas onderverdeeld in verschillende groepen die elk een eigen deel van de code moesten implementeren. Deze groepsverdeling zal verder besproken worden in sectie 2. Wegens de grote hoeveelheid tijd die nodig zou zijn om een netwerk voor het hele alfabet te trainen werd de opdracht beperkt tot het herkennen van cijfers. Een uitbreiding tot letters is via een analoog proces mogelijk, maar was niet realiseerbaar binnen de opgegeven periode.

Het herkennen van cijfers heeft verschillende toepassingen in de hedendaagse wereld. Zo wordt deze techniek gebruikt in de industrie, om nummerplaten te herkennen, om barcodes te scannen, enzovoort. Ook neurale netwerken zelf hebben ontelbare toepassingen. Zo worden ze gebruikt voor gezichtsherkenning, in de procesindustrie, voor spraakherkenning, enzovoort.

2 Groepsindeling

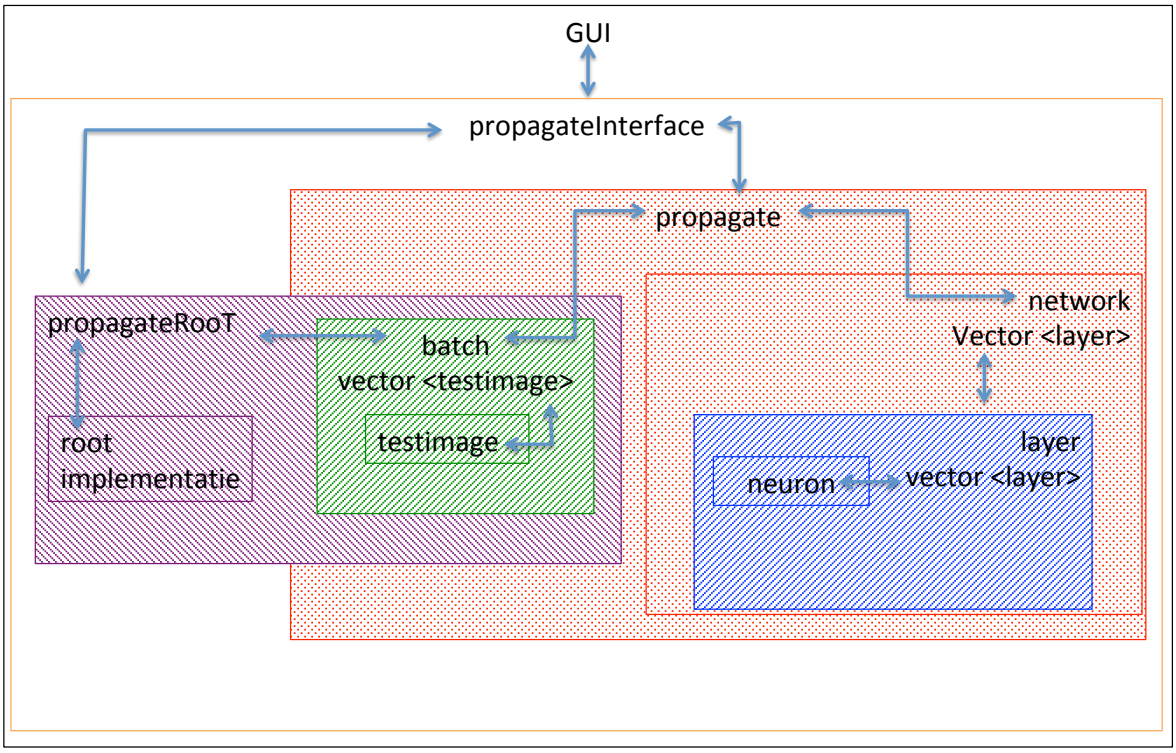
Elke student mocht zelf beslissen in welke groep zij/hij wou werken. Tijdens het verdere verloop van de opdracht zijn deze groepen niet gewijzigd. Beschouw hieronder de groepenverdeling.

- **Groep 1**: Ivan-Jago Coric, Jana Christopher en Stefan Koekkoek.
- **Groep 2**: Marjan Demuynck, Robbe Ceulemans, Tycho Cools, Yannick van den Steen.
- **Groep 3**: Maxim Pavlov, Robin Brabants, Toon De Schutter, Wietse Van Goethem.
- **Groep 4**: Ben Huyge, Luca Serafini, Robine Cleirbaut, Roy Janssens, Stefanie Truyen.
- **Groep 5**: Pieter Schillemans, Storm Brabants.
- **Groep 6**: Bert Jorissen, Matthias Quintelier en Quinten van den Bremen.

Groep 6 bestaat uit de eindverantwoordelijken voor het hele project. Deze groep heeft de algemene structuur bedacht, samen met een eerste versie voor de headers van de nodige klassen. Tevens werden de algemene ontwerpregels opgesteld door deze groep. Verder was deze groep verantwoordelijk voor het testen van de code en de opmaak van het verslag. Groep 6 stond tevens in voor de communicatie tussen de verschillende groepen. De verdere taakverdeling wordt besproken in sectie 3.

3 Structuur van de code

Wat volgt is een bespreking van de top-level structuur van het programma. Figuur 1 geeft een schematische voorstelling weer van de algemene structuur van de gehele code. Deze figuur geeft tevens een voorstelling van de taakverdeling voor de verschillende groepen. De communicatie tussen de verschillende groepen verliep via de website *Trello*. De interne communicatie binnen de groepen gebeurde via een middel naar keuze.



Figuur 1: Schematische voorstelling van de structuur van de code.

Groep 1 was verantwoordelijk voor de implementatie van de klassen *neuron* en *layer*. De klasse *neuron* is de klasse die alle functionaliteiten van een enkelvoudig neuron bevat. Deze functionaliteiten omvatten; een activatiefunctie, een sigmoidfunctie, een functie voor het bepalen van de weights en de bias, enzovoort. Deze klasse interageert met de klasse *layer*. Deze klasse bevat een vector van neuronen. De gedetailleerde beschrijving van de klassen staat beschreven in sectie 4.

Groep 2 was verantwoordelijk voor de implementatie van de klassen *network* en *propagate*. In de klasse *network* bevindt zich de opbouw van het hele neurale netwerk. Deze klasse bevat een vector van verschillende layers. Het aantal layers kan vooraf gekozen worden. Deze klasse bevat tevens een functie die de verschillende layers kan opslaan naar een apart bestand. Tevens werd een functie geïmplementeerd die reeds bestaande layers kan laden in het huidige netwerk. De klasse *propagate*, is de klasse waar de eigenlijke backpropagtion plaatsvindt. Deze klasse interageert intensief met de klasse *network*. De gedetailleerde beschrijving van de klassen en hun werking staat beschreven in sectie 5.

Groep 3 was verantwoordelijk voor de RooT-implementatie voor de backpropagatie. Deze groep moest analoog te werk gaan aan groep 2, maar moest gebruik maken van de TMVL-toolkit van RooT. Groep 3 was tevens verantwoordelijk voor een de klasse *propagateRooT*. Deze klasse moet de berekende data doorgeven naar de klasse *propagateInterface*. Deze klasse werd ontwikkeld door Bert Jorissen en wordt verder beschreven in onderdeel 9. De gedetailleerde beschrijving van de klassen ontwikkeld door groep 3 staat beschreven in onderdeel 6.

Groep 4 was verantwoordelijk voor het ontwerp en de implementatie van de Graphical User Interface (GUI). De structuur van de code is zo ontworpen dat de GUI enkel interageert met de klasse *propagateinterface*. Deze klasse krijgt de verkregen data van de klassen *propagateRooT* en *propagate*. De gedetailleerde beschrijving van de klassen en hun werking staat beschreven in sectie 7.

Groep 5 was verantwoordelijk voor het uitlezen en inladen van de MNIST database. Hiervoor zijn er twee klassen geschreven namelijk *batch* en *testimage*. In de klasse *batch* worden de bestanden van de MNIST database ingelezen. Deze klasse maakt een vector van shared pointers naar testimages. De klasse *testimage* is verantwoordelijk voor de verdere verwerking van de verkregen data uit *batch*. *testimage* maakt zowel een vector en een matrix van de data. Enkel de matrix wordt opgeslagen voor verder gebruik. Er is echter een functie die de matrix omzet in een vector. De gedetailleerde beschrijving van de klassen en hun werking staat beschreven in sectie 8.

4 Groep 1

4.1 Algemene Info & Werkverdeling

De taak van groep 1 is om de klasse *neuron* en de klasse *layer* te implementeren met alle nodige in- en outputs. Om de veiligheid en efficiëntie van vectoroperaties te optimaliseren, is er gekozen om gebruik te maken van algoritmes uit de 'algorithm' bibliotheek. Deze zijn doorgaans efficiënter dan ruwe loops. Het gebruik van iteratoren voorkomt op zijn beurt 'out-of-range'- fouten. Er waren echter enkele situaties waar het gebruik van ruwe loopsniet vermeden kon worden.

In de context van de werkverdeling is alles ongeveer evenredig verlopen. In enkele gevallen werd er in parallel gewerkt aan verschillende versies van implementaties waarbij uiteindelijk alles samengevoegd werd tot één finale versie. Dit was mogelijk minder efficiënt, maar had als bijkomend voordeel wel dat de voor-en nadelen van de verschillende technieken vergeleken konden worden zodat het voor ons een leerrijk proces was. De communicatie binnen de groep verliep voornamelijk via Discord, een modern chatprogramma.

4.2 Neuron

De klasse *neuron* simuleert een perceptron. Voor het neuron zijn twee constructors voorzien. Bij de eerste moeten de gewichten en de bias expliciet worden meegeven zodat het neuron met deze parameters wordt aangemaakt. Deze constructor dient gebruikt te worden indien een netwerk aangemaakt moet worden op basis van reeds opgeslagen parameters in een bestand. Voor de tweede constructor moet enkel het aantal inputs meegeven worden en wordt op basis daarvan een neuron aangemaakt met willekeurige waarden voor de gewichten en de bias. Dit is handig wanneer men wil starten met een volledig nieuw neuron. Naast de constructors is er nog een simpele copy constructor aanwezig, bijgestaan door een overladen assignment operator.

In het 'protected' gedeelte van de klassen bevinden zich vooral de parameters die het neuron definiëren. Namelijk de gewichten (*Weights*), de bias (*Bias*), het aantal inputs (*NumberOfInputs*, geïnitialiseerd door ("setNumberOfInputs") en de output (*Output*). NumberOfInputs is een unsigned integer. Deze keuze volgt uit de logische tests bij de exception handling. NumberOfInputs wordt namelijk vergeleken met de lengte van een vector en daar *vector::size()* een unsigned integer teruggeeft, is het beter dat deze van hetzelfde type zijn om fouten te vermijden.[1]

De functies "setWeights", "setBias" en "setNumberOfInputs" worden gebruikt om de aanpassing van de parameters op gecontroleerde wijze te laten gebeuren. De functies "getWeights", "getBias" en "getNumberOfInputs" zijn de complementen van deze set-functies die de gebruiker toelaten de parameters op veilige wijze te raadplegen.

De functie "randomize" genereert (pseudo-)willekeurige gewichten voor de eerste iteratie van het neuron indien er geen waarden worden meegegeven door het netwerk. Voor de implementatie van "randomize", is er gebruik gemaakt van het Mersenne Twister algoritme dat willekeurige trekkingen uit een uniforme distributie geeft. Initieel is er gewerkt met het interval [-1,1], maar uiteindelijk is er na raadpleging van de literatuur[2] gekozen voor de zogenaamde Xavier initializatie. Hierbij is het interval afhankelijk van de structuur van het netwerk.

[1] Naast het vermijden van fouten voorkomt het tevens dat kwaadaardige gebruikers schade kunnen toebrengen. `https://stackoverflow.com/a/3261019/6739146` (20/05/2018)

[2] `https://github.com/BVLC/caffe/blob/737ea5e936821b5c69f9c3952d72693ae5843370/include/caffe/filler.hpp#L129-143`

Bij de eenvoudige variant wordt de variantie gekozen als $\frac{1}{n_{in}}$ waarbij n_{in} het aantal inputs is voor elk neuron. In dit geval vertaalt dit zich voor een uniforme distrubutie naar het interval $[\frac{-1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}]$. Bij het aanroepen van "randomize" moeten enkel de grenswaarden meegeven worden.

De functie "activateFunc" berekent $z = w \cdot x + b$. Dit is de som van de bias en het inproduct van de gewichten met de input. Het inproduct wordt berekend volgens het *std::inner_product* algoritme. De functie "sigmoid" berekent vervolgens het resultaat van de logistische (sigmoïd) functie: $\sigma(z) = \frac{1}{1-exp(-z)}$. Vervolgens berekent "dsigmoid" de afgeleide van de sigmoïdfunctie.

De bovenstaande functies worden automatisch opgeroepen door de functie "resultFunc", waarbij het berekende resultaat wordt weggeschreven naar de variabele Output. Daar "dsigmoid" gebruik maakt van de waarde van Output heeft deze een default waarde van 0. Dit voorkomt problemen indien "dsigmoid" voor "resultFunc" opgeroepen zou worden.

Figuur 2 geeft de header van de klasse neuron weer.

```
class neuron {
public:
        neuron(const vector<float>&, const float&);
        neuron(const int&);
        ~neuron();
        neuron(const neuron&);
        neuron& operator = (const neuron&);


        void setWeights(const vector<float>&);
        void setBias(const float&);

        vector<float> getWeights();
        float getBias();
        const int getNumberOfInputs();

        float sigmoid(const float&);
        float dsigmoid();
        void activateFunc(const vector <float>&);
        float resultFunc(const vector <float>&);
        float operator()(const vector<float>& NeuronInput) {
                return resultFunc(NeuronInput); }
        float randomize(const float&, const float&);

protected:
        void setNumberOfInputs(const int&);
        vector<float> Weights;
        float Bias;
        size_t NumberOfInputs;
        float Output = 0;
};
```

Figuur 2: Figuur die de header voor de klasse *neuron* weergeeft.

4.3 Layer

De layer klasse maakt een laag van neuronen aan. Dit is in principe gewoon een vector (*Neurons*) die gevuld wordt met objecten van het type 'neuron' uit de klasse *neuron*. Net zoals bij het neuron zijn er hier ook twee constructors, een copy constructor en een overladen assignment operator voorzien. De memberfuncties zijn conceptueel allemaal vergelijkbaar. Ze doorlopen de vector en op elk neuron roepen ze dan de relevante members op. Het aantal neuronen van de layer wordt respectievelijk ingesteld en geraadpleegd door de functies "setNumberOfNeurons" en "getNumberOfNeurons".

De functie "setWeights" en "setBias" itereren over een laag en veranderen de gewichten en de bias van elk neuron. "getWeights" en "getBias" doorlopen vervolgens de laag en roepen van elk neuron de bijhorende gewichten en bias op. Analoog itereren de functies "dsigmoid" en "resultFunc" over de laag en roepen ze "neuron::dsigmoid" en "neuron::resulFunc" aan voor elk van de neuronen.

Figuur 3 geeft de header van de klasse layer weer.

```
class layer {
public:
        layer(const vector<vector<float>>&, const vector<float>&);
        layer(const int&, const int&);
        ~layer();

        void setWeights(const vector<vector<float>>&);
        void setBias(const vector<float>&);
        vector<vector<float>> getWeights();
        vector<float> getBias();
        const int getNumberOfNeurons();

        vector<float> resultFunc(const vector<float>&);
        vector<float> dsigmoid();
        vector<float> operator()(const vector<float> LayerInput) {
                                return resultFunc(LayerInput); }
        layer(const layer&);
        layer& operator = (const layer&);

protected:
        void setNumberOfNeurons(const int&);
        vector<neuron> Neurons;
        size_t NumberOfNeurons;
        size_t NumberOfInputs;

};
```

Figuur 3: Figuur die de header voor de klasse *layer* weergeeft.

5 Groep 2

5.1 Algemene info & werkverdeling

Er werden twee klassen geïmplementeerd door deze groep namelijk, *network* en *propagate*. Bovendien werd er meegewerkt aan de klasse propagateInterface (zie sectie 9). De klasse *network* werd geschreven en getest door Robbe en Tycho. De errorfunctie van *network* en de klasse *propagate* werden door Yannick en Marjan geschreven. Het testen en opsporen van de fouten van deze klasse werd gezamelijk uitgevoerd. Marjan was verantwoordelijk voor het verslag.

5.2 Klasse network

Aangezien er twee verschillende manieren mogelijk zijn om de structuur van het netwerk vast te leggen, werden er twee verschillende constructors geschreven. De eerste constructor zal een netwerk creëren aan de hand van het aantal neuronen per laag en het aantal inputs. De weights en biases zullen in dit geval random gegenereerd worden. De tweede constructor implementeert het netwerk aan de hand van een meegegeven filename. Met dit bestand worden eerder berekende weights en bias ingeladen. Deze maakt gebruik van de functie "loadLayers(string)". De bias en weights kunnen worden opgeslagen met de functie "saveLayers(string)" onder een meegegeven filename. In beide constructors wordt de member variabele *vector<layer> Layers* aangemaakt. Deze vector bevat voor elke laag een layer-object.
Vervolgens bevat de klasse nodige get- en setfuncties voor de Weights, Bias, het aantal lagen (NumberofLayers) en de resultaat van de laag (LayerResult).
Verder wordt ook het resultaat van het netwerk voor een bepaalde input berekend met de functie "resultFunc". Deze maakt gebruik van de gelijknamige functie uit klasse *layer*. De output van een laag l kan worden berekend aan de hand van de output van de vorige laag $\mathbf{a}^{l-1}$ met vergelijking (1).

$$\mathbf{a}^l = \sigma(\mathbf{w}^l \cdot \mathbf{a}^{l-1} + \mathbf{b}) \tag{1}$$

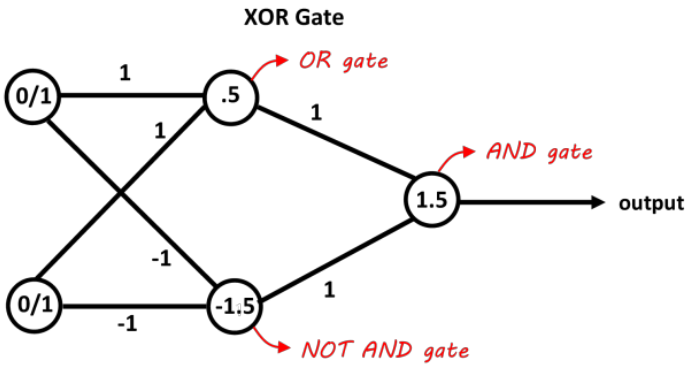
De resultaten van elke laag, alsook de input van het netwerk, worden opgeslagen in de variabele *vector<vector<float>> LayerResult*. Voor elk resultaat kan de zogenaamde costfunctie worden berekend met de functie "costFunc(vector<float> y,vector<float> a)". Deze berekent het verschil tussen het berekende resultaat $\mathbf{a}^L$ en de werkelijke waarde $\mathbf{y}$ via $C = \frac{\|\mathbf{y}-\mathbf{a}^L\|^2}{2}$. Bovendien kan voor elk neuron j in laag l de error functie worden gedefiniëerd zoals weergegeven wordt in vergelijking (2).

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = w_{kj}^{l+1}\delta_k^{l+1}\sigma'(z_j^l) \tag{2}$$

Dit recursief voorschrift vereist dat de berekening van de errorfunctie voor de laatste laag als afzonderlijk wordt beschouwd. Deze kan immers berekend worden via $\delta_j^L = (a_j^L - y_j)\sigma'(z_j^l)$. Deze errorfunctie kan overigens worden gebruikt voor de backpropagatie.

5.2.1 Testen en validatie

De network klasse werd getest door een netwerk op te stellen dat functioneert als een XOR-poort. Dit is een netwerk bestaande uit twee lagen en twee inputs. De eerste laag bevat twee neuronen, de laatste laag bevat één neuron. De weights en biases voor beide lagen worden weergegeven in figuur 4[3].



Figuur 4: Het neurale network voor een XOR-poort.

Op deze manier kunnen verscheidene functies getest worden, waaronder de "resultFunc". Voor een goed geselecteerde input is de uitkomst namelijk bekend. In deze opzet wordt de sigmoid functie vervangen door een stapfunctie. De input kan slechts de waarden 1 of 0 aannemen. De output van "resultFunc" voor verschillende inputs wordt gegeven in tabel 1. Deze tabel stemt overeen met de waarheidstabel voor een logische XOR-poort. Verder werd ook de errorfunctie en de costfunctie berekend per input. Deze waren zoals verwacht steeds 0.

Tabel 1: Het resultaat van het netwerk voor verschillende inputs

Input		**Output**
0	0	0
1	0	1
0	1	1
1	1	0

Tenslotte werd tevens een eenvoudig netwerk gecreëerd met een beperkt aantal lagen en neuronen. In deze opzet werd er echter wel gebruik gemaakt van de sigmoid functie. Er worden random gegenereerde weights en biassen gekozen, alsook een zelfgekozen input. Hiermee kan het resultaat en de errorfunctie handmatig[4] berekend worden. Deze correspondeerden met de output van de functies "resultFunc" en "errorFunc".

Door deze testen van een versimpeld netwerk is de implementatie van de gebruikte functies geverifiëerd. Hieruit kon geconcludeerd worden dat het netwerk op een juiste manier is opgesteld. Tevens kan geconcludeerd worden dat er op een correcte manier gebruik gemaakt werd

[3] `https://stats.stackexchange.com/questions/328488/how-does-the-xor-neural-net-work` (22/05/2018)

[4] Hierbij werd gebruik gemaakt van Matlab

van de klassen *layer* en *neuron*.

Figuur 5 geeft de header van de klasse weer.

```
class network {
public:
        network(const vector<int>& nNeurons, const int nInputs);
        network(string filename);
        ~network();
        network(const network& net);
        network& operator = (const network& net);

        void setWeights(const vector<vector<vector<float>>>&);
        void setBias(const vector<vector<float>>&)
        void setNumberofLayers(const int);

        vector<vector<vector<float>>> getWeights();
        vector<vector<float>> getBias();
        int getNumberofLayers();
        vector<vector<float>> getLayerResult();

        void loadLayers(const string);
        void saveLayers(const string);

        vector<vector<float>> errorFunc(const vector<float>&);
        vector<float> resultFunc(const vector<float>&);
        float costFunc(const vector<float>& ,const vector<float>& );

protected:
        vector<layer> Layers;
        int NumberofLayers;
        vector<vector<float>> LayerResult;
};
```

Figuur 5: Figuur die de header van de klasse *network* weergeeft.

5.3 Klasse propagate

De klasse *propagate* voert de training van het netwerk uit aan de hand van backpropagatie. Deze klasse maakt gebruik van de klassen *neuron*, *layer*, *network* en *batch*. Opnieuw worden er twee constructors geïmplementeerd die beide gebruik maken van zowel een constructor uit klasse *network* als de constructor uit klasse *batch*. De eerste constructor construeert het netwerk met random weights en biases, terwijl de tweede constructor het netwerk opstelt aan de hand van een ingeladen file. Beide constructors maken het *network CurrentNetwork* aan, alsook de *batch EvaluationBatch* en de *TrainingBatch*. De training wordt gedefinieerd door een vooraf bepaalde learning rate η en *MiniBatchSize*.

De nodige get- en setfuncties zijn geschreven voor η, *MiniBatchSize* en *Threshold*. De functie van de waarde threshold wordt later toegelicht. Ook hier kan een eerder berekend netwerk gebruikt worden door middel van de "save-" en "loadNetwerk(string)"-functies.

Aan de hand van de errorfunctie uit *network*, kunnen de biases en weights na m trainingsamples worden aangepast door gebruik te maken van vergelijking (3). De nieuwe weights en biases worden berekend in de functie "deriveWeightsAndBias()", waarna deze worden aangepast in "PropagateMiniBatch()".

$$\begin{cases} w'_k = w_k - \dfrac{\eta}{m} \displaystyle\sum_j a_k^{l-1} \delta_j^l \\ b'_l = b_l - \dfrac{\eta}{m} \displaystyle\sum_j \delta_j^l \end{cases} \tag{3}$$

Hierin worden twee member variabelen geïntroduceerd, *SumError* en *SumAError*. Deze zijn gelijk aan de sommaties van de errorfuncties voor respectievelijk de biases en weights. Vergelijking (4) geeft dit weer voor laag j.

$$\begin{cases} \text{SumError}_j = \displaystyle\sum_j \delta_j^l \\ \text{SumAError}_j = \displaystyle\sum_j a_k^{l-1} \delta_j^l \end{cases} \tag{4}$$

De functie "propagateEpoch()" propageert meerdere minibatches, voorafgegaand door een herschikking van alle trainingsamples. De herschikking wordt verwezenlijkt door een vector te creëren met gehele getallen gaande van 0 tot het aantal trainingsamples. Deze member variabele *Numbers* wordt dan opnieuw gerangschikt. In de functie "propagateMiniBatch" worden steeds de eerste *MiniBatchSize* getallen van de variabele *Numbers* gebruikt, waarna ze worden verwijderd uit de vector. Deze getallen geven aan welk trainingsample uit de vector wordt gebruikt. Bij het kiezen van de *MiniBatchSize* zijn er situaties mogelijk waarbij $\frac{\text{aantal trainingimages}}{\text{grootte minibatch}}$ geen geheel getal is. In deze situaties worden de overige figuren, die geen minibatch meer kunnen vormen, niet gebruikt.

Na elke epoch wordt er tevens een evaluatie uitgevoerd door de functie “evaluate()”. Deze berekent de *ValidationError*, t_p, t_n, f_p en f_n. Hierbij wordt gebruik gemaakt van de member variabele *Threshold*. Deze wordt geïnitialiseerd op 0.5, maar kan via de set-functie aangepast worden. Samen met de variabele *TrainingError*, worden deze variabelen per epoch berekend. Er kunnen meerdere epochs worden uitgevoerd door “propagateNepochs(int)”. De output bevat steeds de reeds berekende errors per epoch[5]. Bovendien bezit de klasse de functie “evaluateHistogram()”. Deze tuple bevat voor elke testsample het resultaat van de “resultFunc-functie en het werkelijke getal.

Tenslotte bezit de klasse de functies “resultFunc” en ”IntToVec”. Deze eerste roept de “resultFunc”-functie uit klasse *network* aan. De functie “intToVec”zet een getal om naar een vector bestaande uit 0, met een 1 op de correcte plaats, zoals weergegeven wordt in tabel 2.

Tabel 2: IntToVec voor het getal 5. Bovenste rij geeft de resulterende vector weer.

0	0	0	0	0	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9

[5]Voor de opslag is de volgorde per epoch; *TrainingError*, *ValidationError*, t_p, t_n, f_p,f_n

5.3.1 Testen en validatie

Deze klasse kan tevens getest worden aan de hand van een versimpeld netwerk. Hiervoor is er gekozen om het netwerk te trainen zodat het de werking van een logische XOR-poort volgt. Om dit te realiseren wordt de code herschreven zodat het mogelijk is om zelf opgestelde trainingsamples mee te geven aan het programma. In het geval van een XOR-poort zijn er vier mogelijke inputs met bijbehorende output gegeven. Een voorbeeld hiervan wordt getoond in tabel 1. In deze situatie zijn er veel epochs doorlopen, aangezien er maar gebruik werd gemaakt van een klein aantal trainingsamples.

Na enkele testen bleek het netwerk in staat zichzelf te trainen om te werken als een XOR-poort. Dit was echter niet na elke test het resultaat. Er werd geconstateerd dat de training erg afhankelijk is van de beginwaarden voor de weights en biases en van de parameter η. In het geval van convergentie bereikte de trainingerror een orde van 10^{-5}. Wanneer het netwerk niet convergeerde, bereikte het slechts een trainingerror van orde 10^{-2}. De trainingerror bleef dan tevens fluctueren rond deze waarde. In tabel 3 worden de outputs gegeven voor de situaties met en zonder convergentie. Het is opmerkelijk dat, wanneer de training niet convergeerde, er steeds maar één waarde sterk afweek van de juiste waarde. Bij het kiezen van een juiste learning rate en een juiste grootte van de minibatch, was de snelheid waarmee het netwerk convergeert veel groter dan wanneer dit niet het geval was.

Tabel 3: Het resultaat van het netwerk na training met convergentie (C) en zonder convergentie (NC).

Input		**Output C**	**Output NC**
0	0	0.012301	0.00753
1	0	0.989385	0.66265
0	1	0.989387	0.66265
1	1	0.0126636	0.66644

Uit deze test kan geconcludeerd worden dat de klasse *propagate* op een correcte manier de biases en weights aanpast.

Figuur 6 geeft de header van de klasse propagate weer.

```
class propagate{
public:
        propagate(float E, int Size,const vector<int>& nNeurons,
                const int nInputs,  const string EvalImageString,
                const string EvalLabelString, const string TrainImageString,
                const string TrainLabelString);

        propagate(const string fileName, float E, int Size,
                const string EvalImageString, const string EvalLabelString,
                const string TrainImageString,const string TrainLabelString);

        ~propagate();
        propagate(const propagate& prop);
        propagate& operator = (const propagate& net);

        void setEta(float);
        void setMiniBatchSize(int);
        void setThreshold(float);

        void setTrainImages(batch&);
        void setEvalImages(batch&);

        float getEta() const;
        int getMiniBatchSize() const;
        float getThreshold() const;

        void loadNetwork(const string );
        void saveNetwork(const string );

        vector<float> resultFunc(const vector<float>& Input);
        tuple<vector<int>,vector<vector<float>>> evaluateHistogram();

        vector<vector<float>> propagateNEpochs(const int N);
        vector<float> evaluate();
```

```
protected:
        vector<float> IntToVec(const float);

        float propagateEpoch();
        void propagateMiniBatch();
        void deriveWeightsAndBias(const vector<vector<float>>&,
                const vector<vector<vector<float>>>&);

        float Eta;
        int MiniBatchSize;

        float CostMiniBatch;
        float TrainingError;
        float ValidationError;

        float Threshold = 0.5;
        vector<int> ChooseTrainingSample;

        network CurrentNetwork;

        batch EvaluationBatch;
        batch TrainingBatch;
        vector<shared_ptr<testimage>> TrainImages;
        vector<shared_ptr<testimage>> EvalImages;

};
```

Figuur 6: Figuur die de header van de klasse *propagate* weergeeft.

6 Groep 3

6.1 Algemene info & werkverdeling

Voor deze groep was het niet eenvoudig om een duidelijke werkverdeling op te stellen. Elk onderdeel van de code werd steeds gezamelijk geschreven. Hierdoor heeft dus iedereen een gelijke hoeveelheid werk geleverd.

In dit onderdeel wordt een equivalent neuraal netwerk besproken dat gemaakt werd aan de hand van de RooT bibliotheek. Hiervoor gebruiken we een aparte module genaamd 'The Toolkit for Multivariate Data Analysis with RooT' (TMVA). Dit is een alleenstaand project dat een omgeving voor supervised machine-learning in RooT aanbiedt. Deze module bevat verschillende implementaties van multivariate classificatiemethodes. Het bevat ook een implementatie van artificiële neurale netwerken, wat zeer interessant is met betrekking tot het project.

De implementatie van onze code is gegeven in de propagateRoot.ccp en propagateRoot.h files. Vooraleer de training aan te vatten moeten de test- en training data in een .root file om deze te kunnen gebruiken. In de functie "makeRootFile" wordt een *TTree* object aangemaakt voor elk nummer. Een *TTree* is een datastructuur in RooT waar ook de TMVA module mee werkt. Elke *TTree* krijgt tevens een branch mee voor elke pixel in de afbeeldingen. Dit betekent dat uiteindelijk een .root file aangemaakt wordt met 10 *TTrees* met elk 784 branches. In elke branch wordt de grayscale waarde van de pixel opgeslagen voor alle afbeeldingen. We houden de training- en test-data gescheiden door eerst alle trainingdata in de TTree te plaatsen, gevolgd door alle testdata.

Het trainen van de data gebeurt in de functie "propagateNEpochs" en het evalueren in de functie "Application". Er wordt respectievelijk een *TFactory* en *TReader* object aangemaakt die de gepaste methodes toepassen op de ingegeven *TTrees*.

Het was de intentie om de RooT interface te linken aan de zelfgemaakte GUI van groep 4. In Qt creator werd echter een segmentation violation verkregen die niet opgelost kon worden. Om deze reden werd het programma uitgevoerd aan de hand van een main.cpp en een zelfgemaakte makefile.

Het netwerk is reeds volledig getraind en deze training is opgeslagen in verschillende MulticlassNumbers.root files. De functie propagateNEpochs detecteert automatisch of het netwerk al dan niet getraind is door te zien of deze .root files al aangemaakt zijn. Indien de gebruiker het netwerk opnieuw wil trainen zal hij deze .root files eerst moeten verwijderen.

6.2 Multi Layer Perceptron (MLP)

De originele opzet was om een neuraal netwerk te trainen met behulp van de TMVA methode MLP. Bij de implementatie van deze methode zijn er echter enkele problemen ontdekt. Een eerste probleem is dat de RooT netwerken geen constante input variabelen accepteren. Dit probleem is eenvoudig op te lossen door deze constante variabelen niet te gebruiken bij het trainen en evalueren. Het significante probleem is dat de gewichten van de nodes worden opgeslagen in een .xml file, maar in elke versie van het programma geven deze gewichten een NaN (Not a Number) weer. De oorzaak van dit probleem is een groot aantal pixelwaarden waarvan de meerderheid zeer dicht bij elkaar ligt en het blijkt dat de MLP methode hier niet goed mee overweg kan. Er werd met verschillende methoden getracht dit probleem op te lossen.

1. Het probleem oplossen op een multiclass methode was een eerste poging. Hierbij worden er tien verschillende klassen aangemaakt, één voor elk nummer. Dat maakt dat er in totaal tien output neuronen aanwezig zijn in het netwerk. Deze methode werkt dus analoog aan de methode van groep 2.

2. Een tweede manier om het gestelde probleem op te lossen was om te werken met een doorsnee classifier en geen multiclass. De classifier van RooT wordt vaak gebruikt om een signaal van achtergrond te kunnen onderscheiden. In dit voorbeeld, waarbij een netwerk getraind wordt om nummers te herkennen, worden de klassen *signal* en *background* dus anders gedefinieerd. Eerst wordt alle data voor het nummer 0 als signal beschouwd, terwijl de andere nummers onder background vallen. Daarna worden de negen andere neurale netwerken aangemaakt, waarbij telkens één van de nummers als *signal* gedefinieerd wordt en de overige nummers als *background*. In totaal verkrijgt men dus tien verschillende netwerken, waarbij elk netwerk de functionaliteit heeft om één bepaald nummer te herkennen. Deze netwerken verschillen van de netwerken uit de opgave, aangezien er hier telkens maar twee output neuronen aanwezig zijn per netwerk in plaats van tien. Het programma zal dus telkens nummer per nummer afgaan en daarbij de kans geven op *signal* (het nummer overeenkomend met het netwerk) en *background* (de andere negen cijfers).

3. Tevens werd getracht de pixels te verdelen in zwart en wit. De grayscale waarde onder en boven een bepaalde drempelwaarden worden respectievelijk op 0 en 1 gezet. Dit hebben we toegepast op zowel de eerste als de tweede methode die hierboven vermeldt staan.

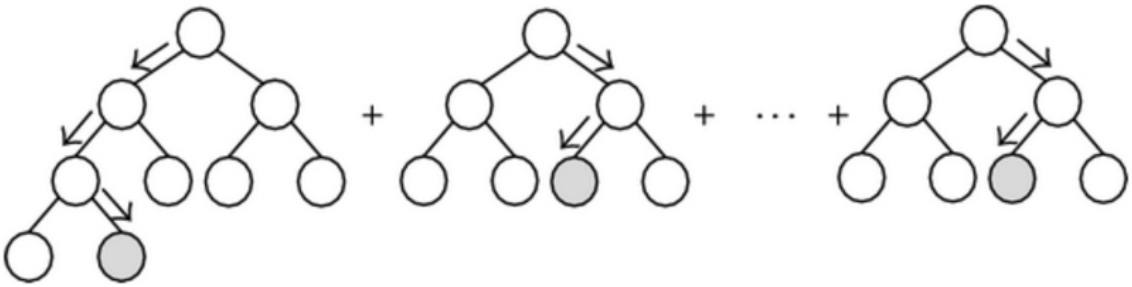
4. Aangezien het probleem te wijten is aan te weinig variatie tussen de input neuronen, werd getracht het netwerk simpeler te maken. Hierbij worden de zwart-wit pixelwaarden (0 of 1) rij per rij opgeteld. Zo verkrijgt men een netwerk met maar 28 input neuronen in plaats van de originele 784. Aangezien deze methode ook een negatief resultaat opleverde, is ook getracht om elke afbeelding in vier rijen op te delen en de som van de pixels op deze rijen te gebruiken als input neuronen. Ook dit netwerk, met dus maar vier input neuronen, was niet in staat het gestelde probleem op te lossen.

5. Tijdens het initialiseren van de TMVA methode (in dit geval dus MLP) kunnen er verschillende opties worden meegegeven. Eén van deze opties is bijvoorbeeld om het aantal hidden layers aan te passen. Een voldoende aantal hidden layers zou in staat moeten zijn om de kleine variaties tussen de input neuronen beter te onderscheiden van elkaar. Dit werd getest tot tien hidden layers met een variërend aantal neuronen, maar ook hier waren de resultaten negatief.

Geen van de bovenstaande methoden lost het probleem van de gewichten op, terwijl er echter wel correct getrainde en geëvalueerde resultaten verkregen werden wanneer er een andere methode gebruikt werd dan MLP.

6.3 Boosted Decision Trees (BDT)

In het uiteindelijke programma wordt een alternatief aangeboden dat weliswaar niet op een multilayer perceptron manier te werk gaat, maar wel een correct resultaat teruggeeft. Hiervoor wordt de TMVA methode BDT gebruikt. Een decision tree is een classifier die eenvoudig in een 2D boomstructuur kan worden weergegeven. In deze tree worden alle nodes overlopen en, aan de hand van de inputwaarden, wordt een correct pad gekozen. Afhankelijk van de uiteindelijke positie in de tree, kan een enkele decision tree bepalen of een bepaalde input als *signal*- of *background*-achtig beschouwd moet worden. Figuur 7[6] toont een schematische voorstelling van deze methode.



Figuur 7: Schematische voorstelling van een Gradient Boosted Decision Tree ensemble.

Decision trees zijn echter gevoelig voor statistische fluctuaties in het trainingsample. Om dit op te lossen zijn boosting algoritmes gebruikt, meer bepaald gradient boosting. Het te optimaliseren model wordt als een ensemble van gewogen, zwakke modellen genomen. Bij onze casus zijn deze zwakke modellen dus decision trees.

Bij het trainen, of growing van een decision tree, wordt elke node samen met enkele criteria overlopen. Aan de hand van een 'separation criterium' wordt bepaald hoeveel nauwkeurigheid er wordt opgegeven bij het aanbrengen van een splitsing. Hiervoor wordt gebruik gemaakt van het Gini criterium: $p(1-p)$, waarbij p een getal is dat de samenstelling van de 2 nieuwe paden vergelijkt. Bij $p = 0.5$ zijn signal en background volledig gemixt, terwijl bij $p = 0$ het signaal zich volledig aan één kant bevindt en de achtergrond aan de andere.

Dit biedt een model dat vanuit de inputwaarden de waarde van het target kan afschatten. Bij onze casus is dit te gebruiken om, vanuit de grayscale waarden van de pixels, te zien hoe dicht een afbeelding aanleunt bij elk van de 10 getallen.

Deze methode werd op 2 verschillende manieren getest namelijk; op een multiclass manier en op een individuele signal-background manier. Bij deze implementaties werd er voor een meegegeven afbeelding 10 waarden verkregen. Het hoogste van deze getallen wijst op het nummer waar de afbeelding de meeste gelijkenissen mee vertoont. Na beide manieren getest te hebben voor verschillende afbeeldingen blijkt de multiclass manier vaker foutieve beslissingen te nemen dan de signal-background manier. Het nadeel van deze manier is echter dat de code minder overzichtelijk is. Deze implementatie bestaat uit meer lijnen code omdat de objecten (TFactory en TReader) 10 keer aangemaakt moeten worden. Dit is niet het geval bij de andere manier.

[6]Shin, Yoonseok. Application of Boosting Regression Trees to Preliminary Cost Estimation in Building Construction Projects. Computational intelligence and neuroscience. 2015. 149702. 10.1155/2015/149702.

Er werd gekozen om prioriteit te geven aan correctere resultaten, hierdoor werd er voor de signal-background implementatie gekozen. Figuur 8 toont de verkregen output van deze methode. Zoals te zien is wordt het 'real number' correct herkent door het programma.

```
real number: 7
evaluation 0: -0.626809
evaluation 1: -0.757652
evaluation 2: -0.365982
evaluation 3: -0.391184
evaluation 4: -0.513821
evaluation 5: -0.387841
evaluation 6: -0.673309
evaluation 7: 0.378486
evaluation 8: -0.531877
evaluation 9: -0.457013
```

Figuur 8: Resultaat van het testen van een geschreven 7. De zevende evaluatie heeft de hoogste waarde, dus de afbeelding wordt correct herkend.

Aangezien de BDT-methode een andere structuur heeft dan de MLP-methode, kunnen de twee netwerken moeilijk vergeleken worden met elkaar. Het MLP netwerk werkt met een aantal epochs, hidden layers, een bepaalde batch size en een learning rate. De BDT heeft deze opties niet. Wat wel aangepast kan worden, is het aantal trees. Dit staat standaard op 800 trees. Bij BDT is tevens een learning rate aanwezig, maar deze verschilt van de learning rate bij MLP. Bij BDT bedraagt deze waarde standaard 0,5, terwijl bij MLP deze waarde gelijk is aan 0,02. Er zijn dus wel wat verschillen tussen de opties voor deze twee types, waardoor ze niet met elkaar vergeleken kunnen worden. Hierdoor worden bij de BDT methode geen opties meegegeven, de default waarden worden telkens gebruikt.[7]

De default optie voor boosting is Adaptive Boost (AdaBoost). Bij deze boosting methode wordt een hoger gewicht meegegeven aan afbeeldingen die verkeerd geclassificeerd zijn in een vorige tree. Dit heeft als gevolg dat de gewichten in de volgende tree vermenigvuldigd worden met een *common boost weight* α. Deze α wordt als volgt gedefinieerd:

$$\alpha = \frac{1 - err}{err} \tag{5}$$

De error err is de misclassificatie van de vorige tree.

[7] A. Hoecker, P. Speckmayer, J. Stelzer, J. Therhaag, E. von Toerne, H. Voss, arXiv:physics/0703039

6.4 Resultaten

Indien het programma uitgevoerd werd aan de hand van RooT, werkte het trainen en testen van de gehele database perfect. Na het uitvoeren van "propagateNEpochs", werd "Application" toegepast om enkele testimages te evalueren. Via de RooT interface is het eenvoudig om informatie op te vragen over de training. Voor dit project is het echter gewenst dat resultaten zoals TrainingError, ValidationError en alle true/false positives/negatives berekent worden zonder gebruik te maken van de RooT interface. Om dit te realiseren, werd de functie "Application" op een groot aantal testimages uitgevoerd. Hier werden echter bepaalde geheugenlekken vastgesteld. Tijdens het herhaaldelijk uitvoeren van "Application" treedt er een stijging van het RAM geheugen op (RSS in Task Manager). Het geheugengebruik neemt constant toe totdat het maximale geheugen van de VirtualBox bereikt wordt, waarop het programma gestopt wordt voordat het volledig is afgerond. Tevens is het aan te raden om het RAM geheugen van de Virtual Box te vergroten (liefst boven de 7 GB).

Er werd getracht om de oorzaak van deze geheugenlekken te achterhalen door middel van het reduceren van de "Application" functie door de memberfuncties onmiddellijk toe te passen op het Reader object. Hiermee wordt het gebruik van andere variabelen, die ruimte in het geheugen opnemen en misschien niet verwijderd worden, vermeden. Ondanks deze maatregelen blijft het geheugenlek aanweg. Hierdoor wordt verwacht dat het verwachten we dat het geheugenlek zich in de memberfuncties van het Reader object zelf bevindt. Er is echter geen zekerheid over deze conclusie.

De optimale manier om het geheugenlek te vermijden is minder afbeeldingen gebruiken zodat het programma afrondt voordat het geheugen een probleem wordt. We raden wel aan om andere applicaties te sluiten bij het uitvoeren van het programma. Tevens wordt aangeraden om via de Task manager het gebruikte geheugen op te volgen om eventuele problemen te vermijden.

Voor het bepalen van de extra informatie omtrent de training werden 200 test images en 50 train images gebruikt. De functie "evaluate" (die wordt aangeroepen in PropagateNepochs) berekent de training error, validation error en de true/false positives/negatives. De training error en validation error worden bepaald met de costfunctie, analoog aan de andere delen van het project. Voor de true/false positives/negatives wordt gebruik gemaakt van de "GetSignalReferenceCut" functie voor het bepalen van een threshold waarboven een event als signaal wordt beschouwd. Hiermee kan via eenvoudige statements bepaalt worden in welke categorie een event thuishoort. Een true negative heeft bijvoorbeeld een waarde die onder de cut valt, terwijl het nummer dat bij deze waarde hoort niet overeenkomt met het werkelijk geschreven nummer (en dus behoort tot de background). Dit leidt tot de resultaten in figuur 9.

```
TrainingError 0.00239564
ValidationError 0.0581286
TruePositive 196
TrueNegative 1793
FalsePositive 7
FalseNegative 4
```

Figuur 9: Resultaten bij 200 test- en 50 training images

7 Groep 4

7.1 Algemene info & werkverdeling

De interface bestaat uit vier tabbladen. De eerste versie van de interface is volledig door Roy geïmplementeerd. De grafieken op de Training- en Additional Graphs-tabbladen zijn door Ben en Robine geïmplementeerd. Stefanie maakte de laatste aanpassingen en was tevens verantwoordelijk voor kleurenschema van de interface.

Roy heeft gezorgd voor de functies waarmee afbeeldingen ingeladen kunnen worden en de functies waar de afbeeldingen naar zwart-wit of grijswaarden geconverteerd kunnen worden. Eveneens was hij verantwoordelijk voor de functie die de afbeeldingen naar 28x28 pixels kan herschalen. Luca heeft de veschillende versies van de propagate klasse up-to-date gehouden. Tevens heeft hij samen met Bert de *propagateInterface* verbeterd, de interface voor het creëren van een nieuw netwerk mee geïmplementeerd en bepaalde onderdelen verbeterd. Ook heeft hij gezorgd voor de dynamische spinboxes die verschijnen als de 'number of neurons per layer' vastgelegd moeten worden. Robine heeft een begin geschreven aan een functie voor het opslaan van de grafieken, deze werd later afgemaakt door Ben. Tevens waren Robine en Ben verantwoordelijke voor dummy-data voor bepaalde testen die gebruikt konden worden om bepaalde grafieken te verkrijgen. Later werd door hen de dummy-data vervangen door functies die de accuracy, specificity, precision en sensitivity berekenen. Tevens hebben ze ook vier functies die de verkregen grafieken kunnen opslaan geschreven. Ben heeft gezorgd voor de checkboxen waarmee gekozen kan worden welke grafiek getekend moet worden. Iedereen heeft geholpen bij het linken van de verschillende functies. De code om de histogrammen te plotten en op te slaan werd geïmplementeerd door Luca, Ben, Robine en Roy. Robine heeft het verslag van groep 4 geschreven.

7.2 Gebruikte klassen & functies

Er worden drie verschillende klassen gebruikt in de GUI; *MainWindow*, *plotgraph* en *propagateinterface*. Uit de Qt-documentatie werd een headerfile gedownload, genaamd *QCustomPlot*, die functies bevat waarmee grafieken geplot kunnen worden en legendes en assen aangepast kunnen worden.

MainWindow is de hoofdklasse waarin alle functies voor de knopfunctionaliteiten van de interface geïmplementeerd worden. Deze klasse bevat ook een functie om de plots op te slaan en vier functies die de accuracy, precision, specificity en sensitivity berekenen. De klasse *plotgraph* bevat een functie die een normale vector naar een 'qvector' kan omzetten. Alsook één die data van een grafiek kan verwijderen wanneer op de 'clear' knop gedrukt wordt. Deze klasse bevatte tijdens het testen van de interface ook de functies en implementatie van de dummy-data. In de klasse *propagateInterface* staan voornamelijk set- en getfuncties voor de parameters die nodig zijn voor het programma. Ook zijn er nog twee functies; één die een netwerk kan opslaan en één die een bestaand netwerk kan inladen.

7.3 Toelichting van de GUI

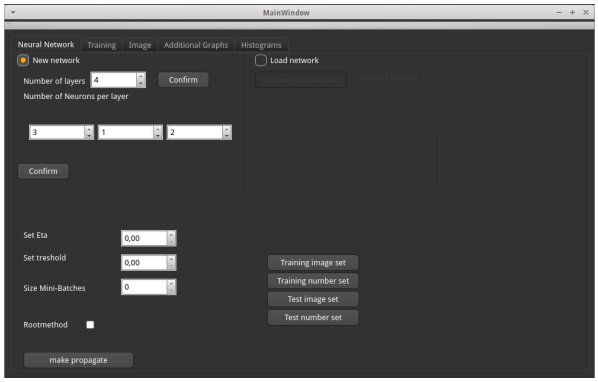
In figuur 10 worden de verschillende tabbladen van de interface afgebeeld. Op de eerste tab '*Neural Network*' kan gekozen worden of men een bestaand netwerk wil inladen of één manueel wil invoeren. Een bestaand netwerk kan ingeladen worden door de *weights*, *biases* en de *Random Values* in te laden. Als manueel een netwerk aangemaakt moet worden, kan men de *number of layers* en de *number of neurons per layer* ingeven. Op dit tabblad kan ook de waarde voor η gekozen worden. Met de overige vier knoppen kunnen de *training image en number* en *test image en number* ingeladen worden.

Op het tweede tabblad staan grafieken van de *Training* en *validation error*. Met de start en stop knoppen kan de training gestart en gestopt worden. Op dit tabblad kunnen de grootte en het aantal van de *mini-batches* gekozen worden, alsook het aantal *epochs*. Tenslotte kunnen op dit tabblad het netwerk en de twee grafieken opgeslagen worden in een pad naar keuze.

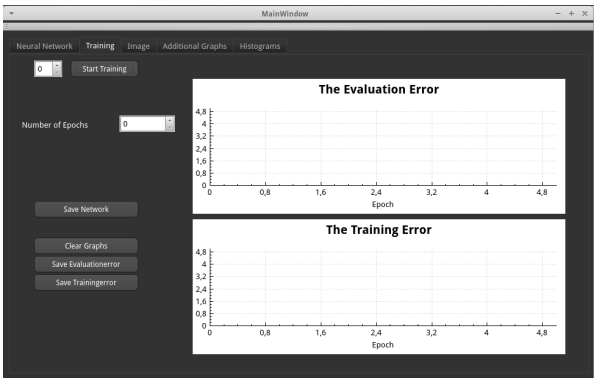
Op het derde tabblad '*Image*' kunnen de afbeeldingen van de cijfers ingeladen worden en kan de grootte herschaald worden naar 28x28 pixels. Er kan gekozen worden of de afbeeldingen naar zwart-wit of grijswaarden geconverteerd moeten worden. Voor zwart-wit waarden kan ook de drempelwaarde ingegeven worden waarboven of waaronder de pixels respectievelijk wit of zwart zijn. Met de *identify* knop kan het cijfer herkend worden. Bij de identify knop wordt de onderste afbeelding ingelezen en opgeslagen als een Mat object. Deze matrix wordt omgezet in een vector en meegegeven aan de functie EvaluateSingleImage.

Het vierde tabblad '*Additional Graphs*' toont de grafieken van de accuracy, sensitivity, specificity en precision. Met de checkboxen linksboven kan gekozen worden welke curves geplot moeten worden. Alle vier de afbeeldingen kunnen in een pad naar keuze opgeslagen worden.

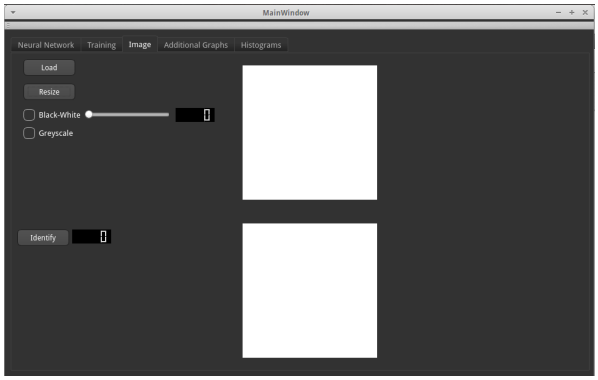
Het laatste tabblad *Histograms* geeft histogrammen weer. Eerst kan op *Evaluate Histogram* gedrukt worden om het nodige te berekenen om achteraf alle mogelijke histogrammen te maken. Als tweede dient de *number* gezet te worden. Als vervolgens op *Plot Histogram* wordt gedrukt dan worden 10 histogrammen op elkaar afgebeeld. Elk histogram hoort bij ëën outputneuron. De x-as bij de histogrammen heeft waarden van 0 tot 1, wat een kans voorstelt. De histogrammen geven namelijk de output (kans) van het beschouwde neuron bij de verschillende testafbeeldingen van het cijfer ingevuld in 'numbers'. De *numbers* kan meerdere keren aangepast worden om met de *Plot Histogram* voor een ander cijfer (andere testafbeeldingen van een ander cijfer) de 10 histogrammen te creëren. De vorige 10 histogrammen worden dan verwijdert. Het getoonde histogram kan ook worden opgeslagen door op de knop *Save as* te klikken.



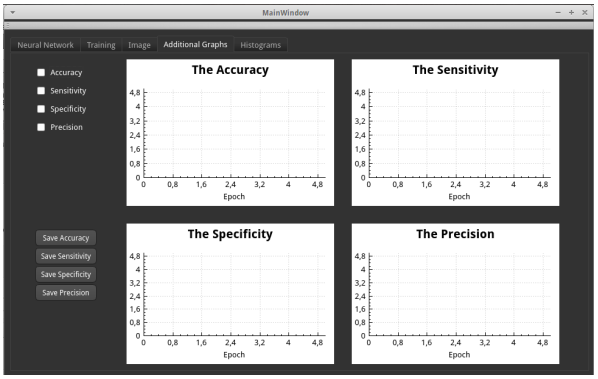
(a) Neural Network



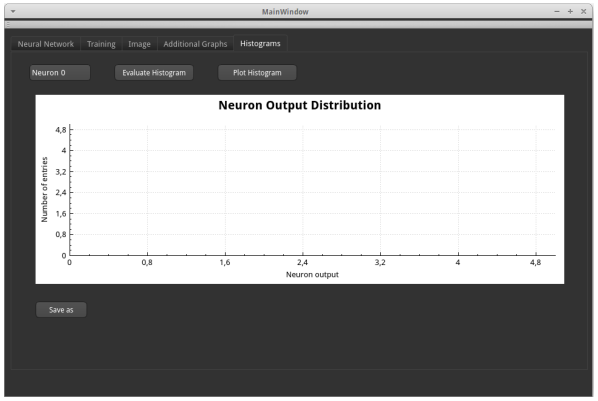
(b) Training



(c) Image



(d) Additional Graphs



(e) Histograms

Figuur 10: De verschillende tabbladen van de grafische userinterface.

8 Groep 5

8.1 Werkverdeling & informatie over MNIST

Aangezien beide klassen erg verstrengeld waren en groep 5 slechts bestond uit 2 personen, is alles samen op één computer gedaan. Met behulp van de screenshare optie op *Discord* kon de code samen geschreven, besproken, getest en verbeterd worden.

De MNIST dataset bestaat uit vier datasets. Deze zijn geen afbeelding- (.png, .jpg ...) of tekstfiles (.txt), maar een binaire opsomming van pixel- en labelwaarden, voorafgegaan door één header per dataset. Twee van de datasets bevatten pixelwaarden, de andere twee bevatten de labels die bij de afbeeldingen horen. Er is een kleine test set (10 000 afbeeldingen) en een grotere training set (60 000 afbeeldingen), elk met bijbehorende labelsets.

De datasets zijn opgebouwd uit delen van andere datasets, NIST. Dit is een oudere database bestaande uit handgeschreven cijfers. De database bestond uit afbeeldingen van 20x20 pixels, maar MNIST heeft dit omgezet naar afbeeldingen bestaande uit 28x28 pixels. De helft van de MNIST datasets bestaat uit getallen van NIST groep 3, de andere helft uit NIST groep 1. Groep 3 is "mooier" neergeschreven dan groep 1 en dus makkelijker te herkennen.

De twee groepen zijn echter wel gemixt in MNIST. De afbeeldingen bestaan uit 28x28 pixels, dus de eerste 784 waarden na de header van een file die pixelwaarden bevat, vormen de eerste afbeelding, 785 tot 1568 de tweede enz. In de header zijn verschillende delen informatie te vinden. Een eerste deel is een "magic number" dat informatie geeft of men met een label- of afbeeldingset werkt. Tevens bevat de header de grootte van de set en, indien met met afbeeldingen werkt, het aantal rijen en kolommen.

Volgens de site is MNIST opgesteld met het gebruik van "high endian" processors (waarvan de meeste non-Intel zijn) in gedachten. Deze lezen binaire code anders in dan "low endian" processors. Hierom moet bij het gebruik van een Intel-processor een inversie van getallen van de header worden uitgevoerd. [8]

[8] http://yann.lecun.com/exdb/mnist//. Geraadpleegd op 16 mei 2018.

8.2 testimage

De vectoren die gemaakt worden door de readfuncties van *batch*, worden element per element aan de constructor van *testimage* doorgegeven. Hier wordt het label opgeslagen. Tevens wordt de vector van uchars omgezet in een matrix bestaande uit uchars door de functie "convertToMat". Er wordt een 28x28 nulmatrix gegenereerd en deze wordt element per element een pixelwaarde van de vector gevuld. Deze matrix wordt gebruikt om de afbeelding later te visualiseren.

Om de verkregen data in te voeren in een neuraal netwerk, moet de functie "getConcatenatedColumns" aangeroepen worden op de matrix. Deze functie plaatst de matrixkolommen onder elkaar in een vector van uchars. Aansluitend wordt deze vector geconverteerd naar een vector van floats. Hierbij zijn beide vectoren nodig vermits een directe conversie van Mat<uchar> naar vector<float> niet mogelijk is.

Een afbeelding van een *testimage*-matrix is terug te vinden in Figuur 11. Het label is erbij geprint bij wijze van illustratie. Dit gebeurt niet in de eigenlijke code.

Figuur 11: Willekeurige getallen uit de MNIST database

Figuur 12 geeft de header van de klasse testimage weer.

```
class testimage{
public:
        testimage(vector<uchar> Image, int RN);
        ~testimage();
        int getRealNumber();
        vector<float> getConcatenatedColumns();
        Mat getMat();

protected:
        Mat convertToMat(vector<uchar> ImageVec);
        int RealNumber;
        Mat ImageMat;

private:
        testimage(const testimage&);

}
```

Figuur 12: Figuur die de header voor de klasse *testimage* weergeeft.

8.3 batch

Een batch is een vector van shared pointers die wijzen naar testimages. Om een batch op te stellen moeten twee files aan de constructor worden meegegeven; de file met pixelwaarden en de file met de labels van de set die men wenst te gebruiken (zie ook 8.1). Deze files worden dan doorgegeven aan de functie "loadBatch". Hier worden eerst de headers ingelezen met readfuncties, daarna worden de afbeeldingen en labels zelf ingelezen met behulp van andere readfuncties. Na omzetting in testimage-objecten, genereert "loadBatch" ook een vector met shared pointers naar deze objecten.

De readfuncties van de headers ("readImageHeader" en "readLabelHeader") inverteren eerst de bytes van de header (zie 8.1). Deze inversie is overgenomen van de code van Jürgen Brauer[9]. Welke info er te vinden is in de headers, staat beschreven in 8.1.

De overige files hebben één byte per waarde voor een pixel/label. Hierom lezen de readfuncties de data in als uchars, daar deze even groot zijn. Om de header niet te gebruiken bij het inlezen van de overige files, wordt gebruik gemaakt van het commando "seekg". De afbeeldingen worden opgeslagen als een vector. De readfunctie voor de afbeeldingen geeft een vector van vectoren terug. De readfunctie voor de labels levert een vector van integers.

Figuur 13 geeft de header van de klasse batch weer.

```
class batch{
public:
        batch(string ImageSetName, string LabelSetName);
        ~batch();

        void loadBatch(string ImageSetName, string LabelSetName);
        vector<shared_ptr<testimage>> getTestImages();

protected:
        vector<int> readImageHeader(string ImageSetName);
        vector<int> readLabelHeader(string LabelSetName);
        vector<vector<uchar>> readImages(string ImageSetName, int Rows,
                int Cols, int NumberOfImages);
        vector<int> readLabels(string LabelSetName, int NumberOfLabels);
        vector<shared_ptr<testimage>> TestImages;

private:
        batch(const batch&);

}
```

Figuur 13: Figuur die de header voor de klasse *batch* weergeeft.

[9] http://www.juergenwiki.de/notes/machine_learning_reading_in_mnist_dataset.html. Geraadpleegd op 28 april 2018.

9 De klasse propagateInterface

Deze klasse werd ontwikkeld door Bert Jorissen en verbindt de klasse *propagate* met de klasse *propagateRooT*. De voornaamste reden voor de ontwikkeling van deze klasse is een eenvoudigere communicatie tussen de verschillende implementaties en de GUI. In deze klasse wordt de definitieve keuze gemaakt tussen de RooT-implementatie en de eigen implementatie. Deze keuze wordt verricht in de functie "makePropagate()" door middel van een bool. Het is dus niet mogelijk om beiden simultaan te verrichten.

In *propagateInterface* worden tevens de verschillende fouten berekent per Epoch. Deze fouten zijn: de TrainingErrors, de EvaluationErrors, de TruePositives, de TrueNegatives, de FalsePositives en de FalseNegative. Voor elk van deze fouten zijn er de nodige get- en setfuncties. Tevens worden de Accuracy, de Sensitivity, de Specificity en de Precision berekend in deze klasse. Ook voor deze variabelen zijn de nodige get- en setfuncties geïmplementeerd.

Figuur 14 geeft de header van de klasse propagateInterface.

```
class propagateinterface
{
public:
        propagateinterface();
        ~propagateinterface();
        propagateinterface(const propagateinterface&);
        propagateinterface& operator = (const propagateinterface&);

        void makePropagate();
        void propagateNEpochs(int);
        vector<float> evaluateSingleImage (vector<float>);

        void useLoadNetwork(bool);
        void useRooT(bool);
        void setNumberOfNeuronsPerLayer(vector<int>);
        void setMiniBatchSize(int);
        void setEta(float); //construeren

        void setIterations(int);
        void setEvaluationError(float);
        void setTrainingError(float);
        void setTruePositive(float);
        void setTrueNegative(float);
        void setFalsePositive(float);
        void setFalseNegative(float);
        void setThreshold(float);

        void setEvaluationImages(string);
        void setEvaluationNumbers(string);
        void setTrainingImages(string);
        void setTrainingNumbers(string);
        void setLoadNetworkPath(string);
```

```
        int getMiniBatchSize();
        void saveNetwork(string);
        vector<int> getNumberOfNeuronsPerLayer();

        vector<int> getIterations();
        vector<float> getEvaluationError();
        vector<float> getTrainingError();
        vector<vector<vector<float>>> getRawHistogram();


        vector<float> getTruePositive();
        vector<float> getTrueNegative();
        vector<float> getFalsePositive();
        vector<float> getFalseNegative();
        vector<float> getAccuracy();
        vector<float> getSensitivity();
        vector<float> getSpecificity();
        vector<float> getPrecision();
        float getThreshold();
        string getEvaluationImages();
        string getEvaluationNumbers();
        string getTrainingImages();
        string getTrainingNumbers();

protected:
        int MiniBatchSize;
        int NumberOfLayers=0;
        float Eta = 0;
        vector<int> NumberOfNeuronsPerLayer;
        bool RootBool = false;
        string EvaluationImages;
        string EvaluationNumbers;
        string TrainingImages;
        string TrainingNumbers;
        string LoadNetworkPath;

        vector<int> Iterations;
        vector<float> EvaluationError;
        vector<float> TrainingError;

        vector<float> TruePositive;
        vector<float> TrueNegative;
        vector<float> FalsePositive;
        vector<float> FalseNegative;

        propagate* Prop;
        propagateRoot* RootProp;
        bool make = false;
        int nInputs = 784;
        float Threshold;
        void destroyCurrentNetwork();

};
```

Figuur 14: Figuur die de header van de klasse *propagateInterface* weergeeft.

10 Testen van de implementatie

De verschillende stukken code werden uitvoerig getest door verschillende personen. De functionaliteiten van elke klasse werden afzonderlijk getest alvorens deze gebruikt werden tijdens het trainen. Wanneer alle afzonderlijke delen naar behoren functioneerden, werd getracht een eenvoudig netwerk (XOR-netwerk) te trainen. Hierdoor werden bepaalde problemen opgemerkt en verbeterd. Wanneer dit eenvoudig netwerk het correcte gedrag vertoonde werd het netwerk uitgebreid naar de te behandelen netwerken.

11 Resultaten

Er worden verschillende netwerken gecreëerd. Hiermee worden verschillende testen uitgevoerd en de karakterisatie van het netwerk wordt steeds onderzocht.

11.1 Netwerk zonder hidden layer

Voor het eerste deel van de opdracht wordt een netwerk zonder hidden layers opgesteld. Deze bevat dus slechts een laag met 10 neuronen. De invloed van de verschillende parameters worden onderzocht, waaronder de *MiniBatchSize* en de learning rate η. Het training- en validatie-errors tonen aan dat het gebouwde netwerk een correcte werking volgt. Beide errors convergeren naar een limiet van 0.4. Dit is een vrij hoge waarde, wat duidt op een matige perfomantie.

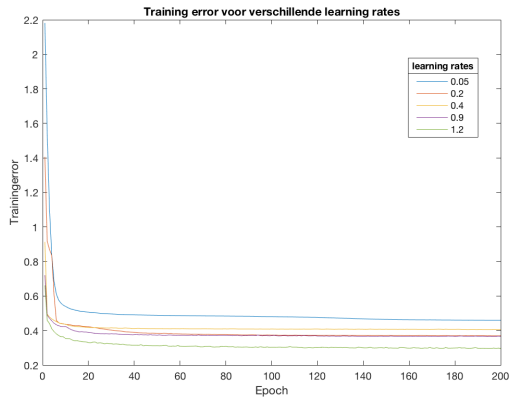
11.1.1 Invloed η

Er wordt steeds een *MiniBatchSize* van 1000 en een *Threshold* van 0.5 gebruikt.
Figuur 15 toont het verloop van de grootheden voor verschillende waardes van η. De invloed van de parameter η wordt hierin duidelijk geïllustreerd. Het valt op dat het verloop bij $\eta = 0.05$ sterk afwijkt. Deze afwijking uit zich vooral in het verloop van de sensitiveit en van de precisie.

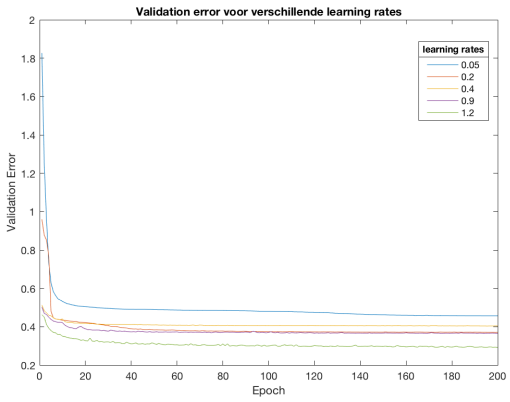
Algemeen kan men concluderen dat hoe groter de waarde voor η, hoe kleiner de trainingerror zal zijn. Dit is echter niet het geval voor $\eta = 0.2$. Deze convergeert naar een lagere waarde dan die voor $\eta = 0.4$. Een conforme conclusie kan bekomen worden voor de Validation error.

Voor de specificity kan besloten worden dat η slechts een minimieme invloed uitoefent op de convergentiesnelheid van het netwerk. η blijkt echter een grote invloed te hebben op de sensitiviteit van het netwerk, zoals te zien is in figuur 15f. Hier is tevens te concluderen dat, hoe groter de waarde voor η gekozen wordt, hoe groter de sensitivity zal zijn. Opnieuw vormt $\eta = 0.2$ een uitzondering op deze regel. Bij deze waarde zal de sensitivity hoger zijn dan bij $\eta = 0.4$. In figuur 15e is te zien dat de learning rate een belangrijke invloed op de precisie heeft. Wanneer deze te klein gekozen wordt (bijvoorbeeld $\eta = 0.05$), zal de Precision zeer traag convergeren. De convergentiesnelheid voor grotere waarden zal ongeveer gelijk zijn. Ook de Accuracy is sterk afhankelijk van de waarde voor η. Hoe groter deze waarde gekozen wordt, hoe hoger de Accuracy. Voor $\eta = 0.2$ is er echter weer een uitzondering.

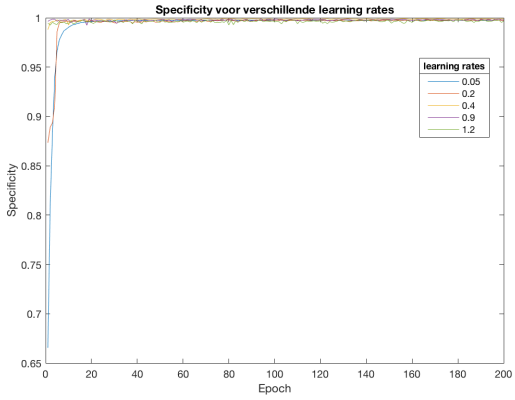
In de opgave werd gevraagd dit ook voor het RooT-netwerk te onderzoeken. Door de verandering van type implementatie is dit echter niet meer mogelijk. De gebruikte implementatie is niet afhankelijk van een parameter η.



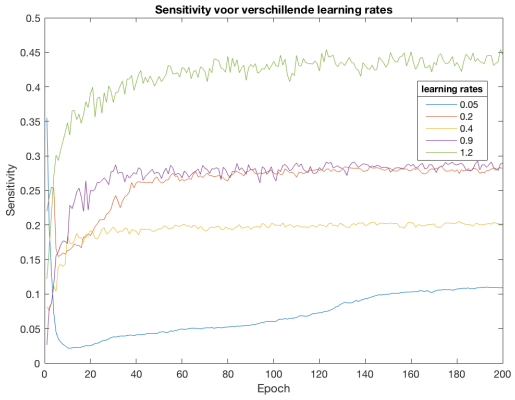
(a) Trainingerror



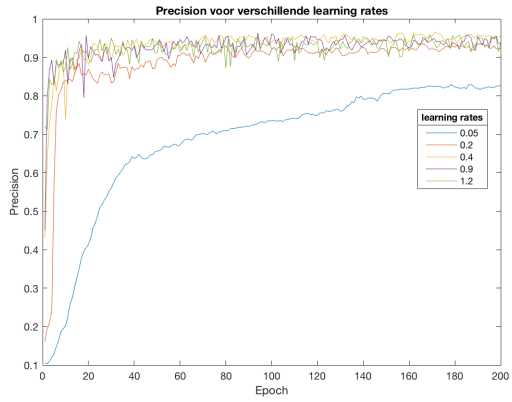
(b) Validation error



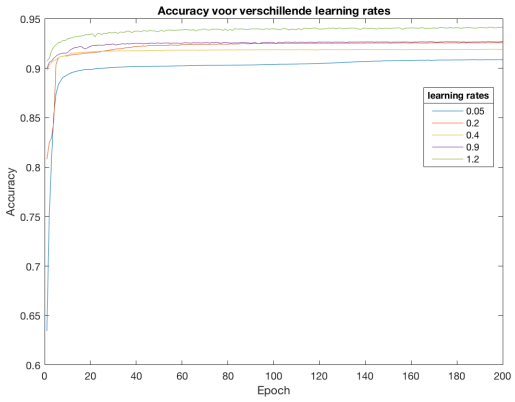
(c) Specificity



(d) Sensitivity



(e) Precision



(f) Accuracy

Figuur 15: De Trainingerror, Validation error, Specificity, Sensitivity, Precision en Accuracy voor verschillende learning rates. Het gebruikte netwerk bevat geen verborgen lagen. De gebruikte *MiniBatchSize* en *Threshold* zijn respectievelijk 1000 en 0.5 .

11.1.2 Invloed *MiniBatchSize*

Voor hetzelfde netwerk wordt de invloed van de *MiniBatchSize* onderzocht. Hierbij wordt voor de learning rate steeds een waarde van 0.9 gebruikt. Uit de vorige sectie bleek dat dit een goede waarde is. Uit figuur 16 blijkt dat de *MiniBatchSize* een grote invloed uitoefend op de eindresultaten. Het belang van een juist gekozen *MiniBatchSize* wordt duidelijk.

Bij een te klein gekozen *MiniBatchSize* toont het netwerk al snel een slechte performantie. Zowel de training- en validatie-error, alsook de specificity en dergelijke veranderen niet doorheen het trainingsproces. Het netwerk behoudt hoge error-waardes. De specificiteit blijft heel de tijd 1, wat betekent dat de 0-waardes in de output steeds op de juiste plaats staan. Bovendien is de sensitiveit steeds 0. Er wordt dus nooit een correcte 1-waarde gegeven als output. Hierdoor behaalt het netwerk met deze grootte van de minibatch een zeer lage precisie en accuracy.

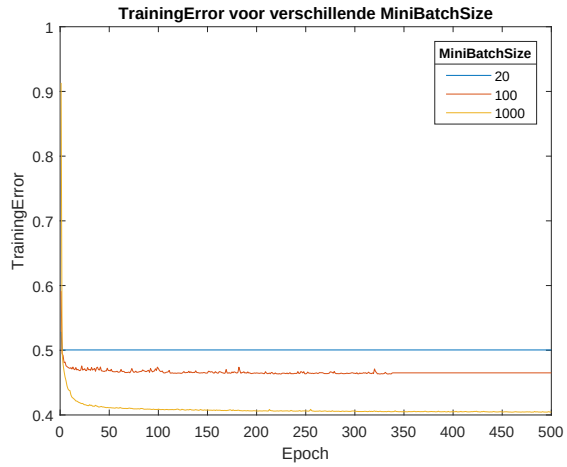
Een *MiniBatchSize* van 100 trainingssamples behaalt een matige performantie. Het scoort op elk vlak beter dan bij een kleine *MiniBatchSize*, maar de behaalde errors zijn niet ideaal. Na een bepaald aantal epochs (≈ 350), stopt het netwerk met veranderen. Het verloop van de errors en karakteristieke vormen een perfect horizontale rechte. Waarom het dit gedrag vertoond is niet te verklaren.

Het netwerk met 1000 *MiniBatchSize*s behaalt de laagste training- en validatie-error. Het is opvallend dat bij deze waarde de karakteristieke grootheden veel meer fluctueren dan bij de andere netwerken. Met deze keuze van *MiniBatchSize* bezit het netwerk een redelijke performantie, waarna er gekozen wordt om met deze waarde verder te werken.

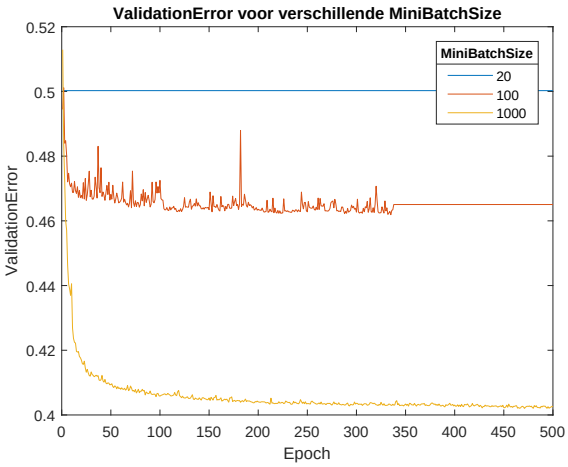
11.2 Netwerk met hidden layers

Wegens tijdsgebrek werden voor het trainen van deze netwerken maar 100 epochs uitgevoerd. Tevens werd er gekozen om het zelf-geïmplementeerde netwerk te gebruiken. Er wordt steeds een *MiniBatchSize*, η, *Threshold* van respectievelijk 1000, 0.9 en 0.5 gebruikt. Er worden twee netwerken gecreëerd met steeds één hidden layer. In het eerste geval bevat deze verborgen laag 150 neuronen en in het tweede geval is dit 300. Figuur 17 toont het verloop van de verschillende karakteristieken en errors voor beide netwerken.

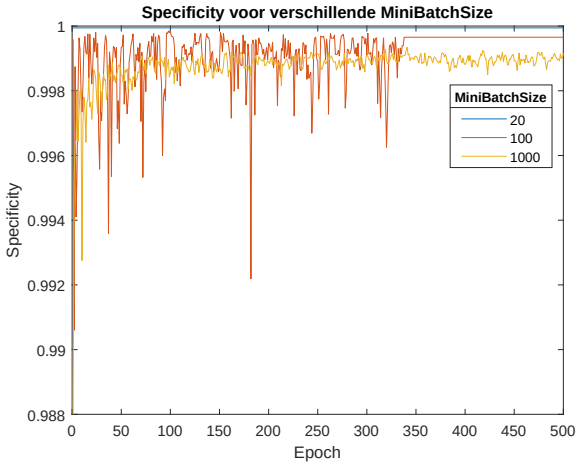
Zoals te zien is in figuur 17a zal de Trainingerror sneller convergeren bij een netwerk met 150 neuronen. Bovendien zal dit netwerk convergeren naar een kleinere waarde. Uit de overige figuren is verder te concluderen dat het netwerk met 150 neuronen performanter is dan het netwerk met 300 neuronen. Slechts bij de specificity en de precisie behaalt het netwerk met 300 neuronen een betere waarde. Echter, beide netwerken convergeren naar eenzelfde waarde. Hierdoor wordt het netwerk met 150 neuronen in de hidden layer verkozen als het meest performante netwerk.



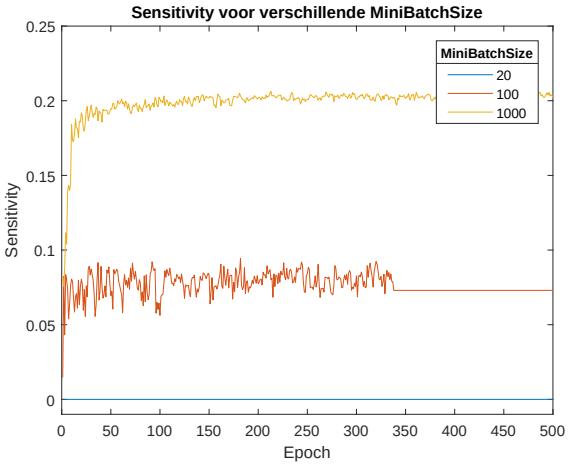
(a) Trainingerror



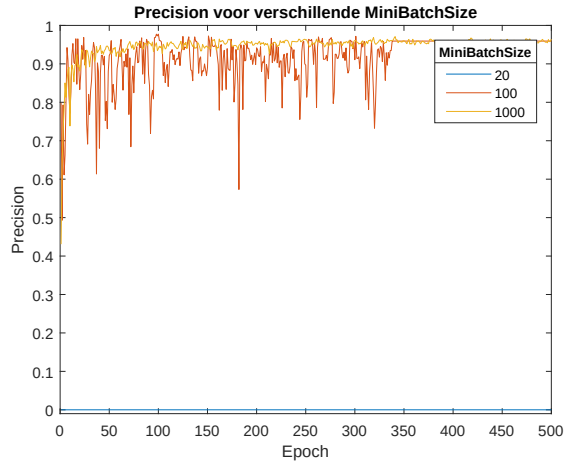
(b) Validation error



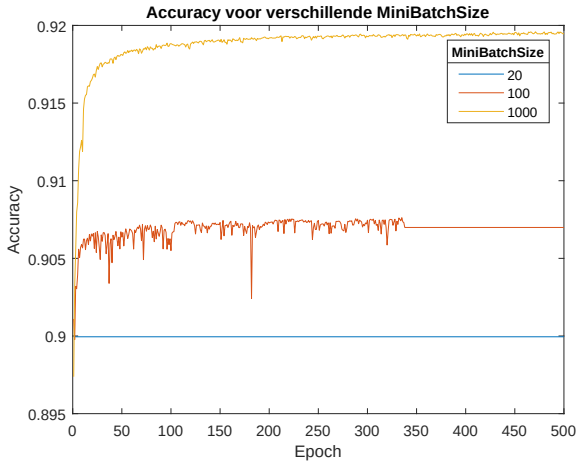
(c) Specificity



(d) Sensitivity

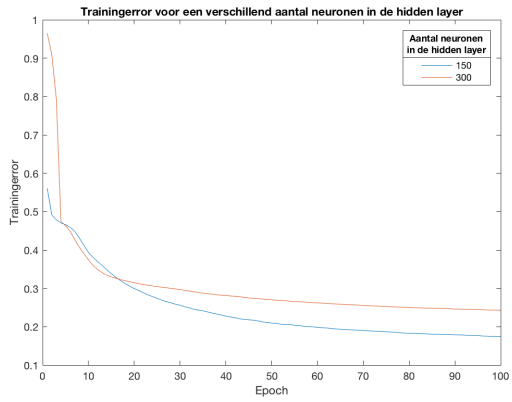


(e) Precision

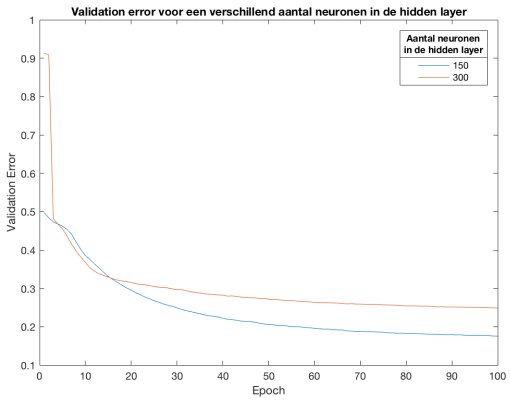


(f) Accuracy

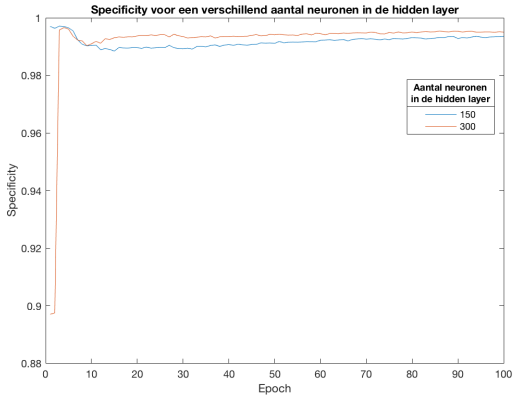
Figuur 16: De Trainingerror, Validation error, Specificity, Sensitivity, Precision en Accuracy voor verschillende *MiniBatchSize*s. Het gebruikte netwerk bevat geen verborgen lagen. De gebruikte η en *Threshold* zijn respectievelijk 0.9 en 0.5 .



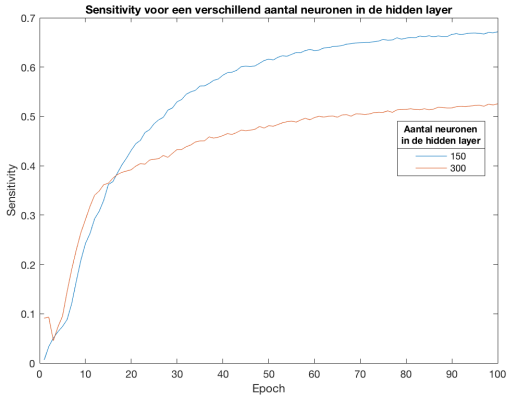
(a) Trainingerror



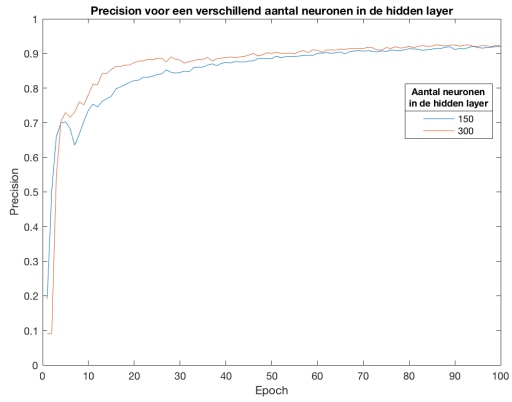
(b) Validation error



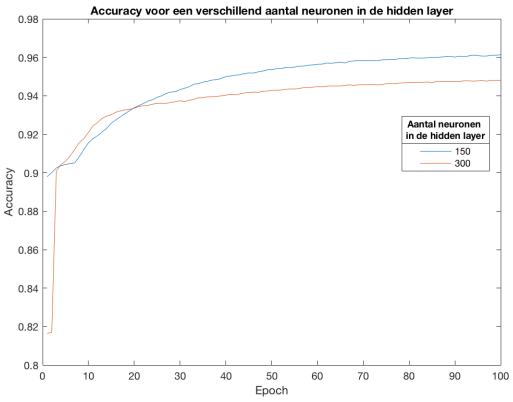
(c) Specificity



(d) Sensitivity



(e) Precision



(f) Accuracy

Figuur 17: De trainingerror, Validation error, Specificity Sensitivity, precision en accuracy voor een verschillend aantal neuronen in de hidden layer. De gebruikte η, *MiniBatchSize* en *Threshold* zijn respectievelijk, 0.9, 1000 en 0.5 .

11.3 Extra netwerk

In de opgave stond dat er tevens een netwerk ontworpen moest worden met beduidend minder inputs dan het normale netwerk. In essentie is dit het XOR-netwerk dat gebruik werd voor het testen van de code. Zoals reeds vermeld werd ook hier convergerend gedrag vastgesteld. Voor dit deel werden er echter geen grafieken bijgehouden.

12 Conclusie

Neurale netwerken kennen een breed spectrum van toepassingen inde moderne maatschappij. Een van deze toepassingen is het herkennen van handgeschreven letters en cijfers. Als examenopdracht voor het vak *programmeren voor fysici* is er een neuraal netwerk geïmplementeerd dat handgeschreven cijfers kan herkennen. Het trainen van dit netwerk verloopt via een backpropagatie algoritme. De studenten hebben zich verdeeld in verschillende groepen die elk een eigen opdracht hadden. Er zijn twee verschillende netwerken ontworpen die in staat zijn om deze handgeschreven cijfers te herkennen; één netwerk dat gebruik maakt van RooT en één zelfgemaakt netwerk. Er werd tevens een GUI ontworpen dat interageert met het zelf opgebouwde netwerk. De link tussen het RooT-netwerk en de GUI is echter niet gelukt.

Er is onderzocht wat de invloed van de parameter η is op de training van het netwerk. De resultaten staan weergegeven in sectie 11. Er werd geconcludeerd dat er een optimale waarde voor η is. Voor dit netwerk is $\eta = 0.2$ gebruikt.

Verder is er nog gekeken naar de invloed van het aantal neuronen in de verborgen laag. Dit werd onderzocht voor twee netwerken; één met 150 neuronen in de verborgen laag en één met 300 neuronen. De resultaten van dit deel staan gegeven in sectie 11.2. Uit de resultaten kan er besloten worden dat het netwerk met 150 neuronen in de verborgen laag performanter is dan het netwerk met 300 neuronen.