Intelligent light control

Diplom afgangs projekt

by
Michel Bøje Randahl Nielsen (s093481)

January 13, 2013

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk
IMM-B.Eng-2014-01

Contents

Abstract (en)

As a part of the upcoming Solar Decathlon project, it is a wish to have some sort of intelligent control of the lighting in the house, which is also more or less generic and requires as little pre-configuration and customization as possible.

The goal of this project has been to investigate and develop a system that solves this task.

The result is a setup, in which one can explore various Artificial Neural Network techniques for setting up a predictive model and then test it in an Intelligent House simulator developed at DTU.

The main purpose of this report is to document my work on this project, but it also serves as inspiration for further development.
In this report, I will also attempt to give a very basic and simple introduction to the artificial neural network techniques which I have utilized for this project, with the target group being students at my own level.

The project was made in the autumn semester of 2013 from 14/10 to 13/01, at the DTU department of Informatics and Mathematical Modeling, under supervision of Christian D. Jensen.

The project is my final project in information technology for the 'Diplom IT' study line at DTU.

Abstrakt (dk)

Som en del af det kommende Solar Decathlon projekt, ønskes der udviklet en form for intelligent styrring af belysningen i huset. Den intelligente styring ønskes at være så generisk som muligt og kræve mindst mulig konfiguration.

Målet med dette projekt har været at undersøge og udvikle et system som kan løse denne opgave.

Resultatet er et miljø, som tillader at man ved hjælp af Kunstig Neural Netværk teknikker kan generere en forudsigende model og teste denne i en Intelligent hus simulator udviklet på DTU.

Formålet med denne rapport er at dokumentere mit arbejde på dette projekt, og derudover være til inspiration for videre udvikling.
Ydermere vil jeg i denne rapport forsøge at give en grundlæggende og simpel introduktion til de kunstig neural netværk teknikker som jeg har benyttet mig af, med studerende på mit eget niveau som målgruppe.

Projektet er udarbejdet i efterårs semestret 2013 fra d. 14/10 til d. 13/01, ved institut for Informatik og Matematisk Modelering på DTU, og under supervision af Christian D. Jensen.

Projektet er mit sidste på Diplom uddannelsen i informations teknologi ved DTU.

Chapter 1

Introduction

Given a family house or maybe an office building, we wish to be able to turn off the lights in the house when it isn't needed. The lights should be turned off as fast as possible with the goal of saving energy, however the lights should be turned off in a way such that they don't cause annoyance for the habitants of the house.

The common solution to this problem is to put up sensors in all rooms, connecting each sensor to the light in the room and then letting the sensor decide when to turn on or off the given light. The sensor will then turn on the light when it registers some motion and at the same time it will start a timer which counts down from a preprogrammed amount of minutes. If the sensor registers motion while the light is on, it will simply reset the timer. When the timer times out, it will turn off the light.

This solution is often acceptable, though, sometimes scenarios occur where the light turns off way to early, or yet other situations where energy could have been saved if the light was turned off earlier.

However, the problem of when exactly to turn off a given light source, is far more complex than one would initially think. Turning on the light is simple given that the system has knowledge about what light source each sensor is related to. Then it is really just a matter of turning on the lights based on sensor registrations.

But autonomously turning off the light again, requires the ability to make predictions based on the behavioral patterns of the habitants of the house.

These behavioral patterns consists of events and data registrations over time, and this hints that the predictive model should be able to handle so called 'time series' prediction.

The development of such a model is close to impossible without using a simulator or a real life setup, or both. A simulator gives the developer the ability to

quickly generate data for the model and test the model after it has been trained. Where a real life setup can require a lot more work, but in the end serves as a better proof of concept.

For this project a simulator has been used.

Chapter 2

Choice of model

Two commonly used models for time series prediction are Markov Models(MM) and Recurrent Neural Networks(RNN), which is a special type of Artificial Neural Network (ANN).

MM seems to be the most widely used, which is most likely due the fact that MM has been successfully applied far earlier than ANN, and therefore often is seen as more mature.

There has, however, been a lot of development in ANN, and RNN techniques, and RNN has been applied in handwriting recognition, robotics, speech recognition, protein analysis and stock market prediction, amongst others[1].

The exploration of ANN techniques for the model has been chosen for this project, for two reasons. The first reason, was that I wished to learn about about ANN techniques, -preferably in a practical setting.

And the second reason being that MM has already been tried out multiple times for this specific problem. MM techniques has for example been used in a project at DTU which was about implementing intelligent light control into building 322, and then again in the master thesis "Machine learning in Intelligent Buildings" by Andreas Møller and David Emil Lemvigh [5].

Thus it would be interesting to try out another model, instead of just using the same model again.

I also came across a scientific paper, "Recurrent Neural Network for Human Activity Recognition in Smart Home" [1], in which they evaluate methods for recognizing human patterns in houses.

They gathered data in a real life setup, and tested the predictive power on some models trained with the data.

In the paper they conclude that RNN performs the best for recognizing human patterns in houses, by comparing the results of using RNN (Elman style

[1] `http://www.idsia.ch/~juergen/rnn.html`

setup, with standard backpropagation training), versus a Naive Bayes Classifier and a Hidden Markov Model.

Chapter 3

Simulation

The simulator used for the project, is developed as a product of a bachelor thesis project [4] in 2010 by the students Sune Keller and Martin Skytte Kristensen under supervision of Christian D. Jensen.

3.1 Simulation based evaluation

The simulator is perfectly tailored for the purpose of this project.

In its current state it its capable of fulfilling just one purpose, which is to help with developing and testing intelligent light control.

With the simulator, it is possible to rapidly setup and configure a house with walls, sensors, lights, light-switches and connections between the various sensors and light switches.

Furthermore, one can setup scenarios with one or more actors and then simulate the stimulations of sensors and light switches in a given house, based on the patterns created in these scenarios.

Data can effectively be gathered, and at last, it is possible to manipulate the lights by subscribing to sensor events and scheduling when to turn off a given light.

3.2 Limitations of the simulator in its current state

The simulator gives the developer an excellent overview, and provides him with just enough tools to develop intelligent light control. But it still has a serious limitation, which is that the developer has to manually setup a lot of scenarios.

It is very important that the scenarios created, contain some patterns. And actually this was something which I realised while attempting to automate the pattern generation with random paths.

There really is no patterns in random, and the whole purpose of using the ANN, is to detect patterns. Without patterns, the ANN is useless.

The artificially generated data would also be very different from data gathered in a real life setup, because humans really don't act random. Humans have habits, which cause them to repeat patterns, and this is what we seek to exploit when using an ANN model to solve the problem.

However, I see the fact that the developer has to setup the patterns manually as a limitation.

Not only will the developer make a lot of assumptions when creating these patterns, it is also incredible boring when a lot of patterns has to be generated.

Furthermore it can be very complicated to create scenarios with multiple actors, as one has to think very carefully about when to turn off a given light, -it could be that one of the actors is still inside the given room.

The simulator does a very good job at making the timing sequences for each pattern visible to the user, but it is still cumbersome to setup multi actor scenarios.

Maybe it would be beneficial to extend the simulator with some sort of automated pattern generation in the future, or make more tools to assist with the setup of multi actor scenarios.

For details about encountered bugs in the simulator, please refer to appendix "Known bugs and problems in the simulator"

Chapter 4

Theory about Artificial Neural Networks

When we seek to solve problems in science, we often let us inspire by nature, and the core idea of Artificial Neural Networks is to model the Neural Networks in a brain.

The brain is a brilliant product of nature. It is not only able to perform some very complex pattern recognitions, but also to adept itself to new circumstances. Biological experiments has shown that if you rewire a brain area responsible for, for example, vision such that this area receives input from, for example, the ears instead of eyes, then this area will reprogram itself such that it is able to handle this new kind of input.

With ANN techniques we attempt to, sort of, model how the brain learns patterns.

In this chapter I will briefly describe some of the theory behind the ANN techniques which I have utilized for this project.

When Learning about ANN techniques, I have made heavy usage of the book "Introduction to the Math of Neural Networks" [2] and to some extent the "Heaton Research wiki page" [3], thus this chapter and the code implementation itself will reflect the content presented in these two resources.

4.1 ANN structure

An ANN consists of two things, which are neurons that basically encapsulates a activation function, and weights which are weighted connections from the output of one neuron to the input of another.

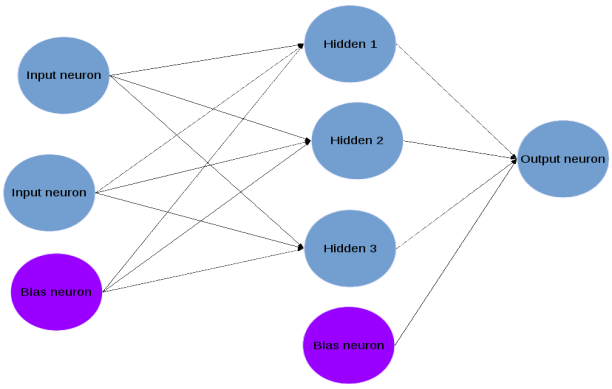

Figure 4.1: Basic ANN structure

The neurons are arranged into layers, and the most common basic structure consists of an input layer, minimum one hidden layer and one output layer. There is also a bias neuron in all layers except the output layer, for reasons explained later.

To calculate the output(s) of an ANN, one goes through the layers calculating the output of each neuron in the layer using the weighted outputs from neurons connected from the previous layer.

The output of the input neurons are simply the input values fed into them, and the value of the bias neuron is always the value 1.0. These output values are multiplied by the weight that connects them to the next neuron, and in this next neuron they are summed up and used as argument for an activation function.

This process goes on for all the neurons in each layer until the outputs for the last layer (the output neurons) has been calculated.

The following formula expresses the calculation of the output of a neuron.

$$n\_output = A(\sum_{c=0}^{c<prev\_ns} prev\_n_c * \omega_c)$$

Where A is the activation function used in the neuron, $prev\_ns$ is the number of neurons in the previous layer, ω_n is the weight from the nth neuron in the previous layer to the current and $prev\_n_n$ is the output of the nth neuron in the previous layer.

4.1.1 Activation function

The chosen activation function for all neurons (except bias and input neurons) for this project, is the sigmoid function.

$$f(x) = 1.0/(1.0 - exp(-x))$$

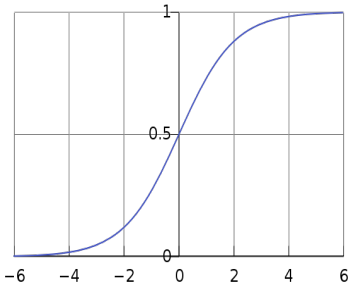

Figure 4.2: The sigmoid function (logistic function)

The sigmoid function, has a nice property that given any negative or positive value as argument the output values always are between 0.0 and 1.0.

One of the reasons why this function is nice to use as activation function, is the same reason as why I normalize the input values, as will be discussed in the next chapter.

Regarding the sigmoid function, it is now that the importance of the bias neuron comes into the picture.

Because the bias neuron output a static value of one, it will allow for the sigmoid function to be shifted left or right.

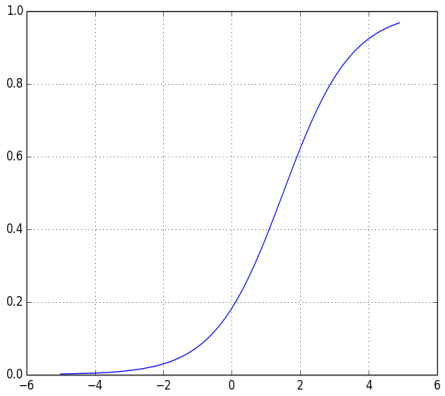

Figure 4.3: shifted sigmoid function

If no bias neuron is used, it will be harder for the network to learn because it will have problems with adjusting itself to situations where all the inputs are zero. The bias neuron helps to solve this problem.

4.1.2 The weights

Before the training of the ANN begins, the weights needs to have some values.

A common approach is to initialize all the weights to random floating point values in the range [-1.0 to 1.0].

One reason to do this, is simply due to the observation that most of the weights will be closely distributed around zero after a neural network has been trained. Thus it is beneficial to have them initialized to values around zero from the beginning.

Nguyen-Widrow, advanced weight initialization

We generally wish the training of the ANN to be as fast as possible.

If we initialize the weights in a clever way before we even start the training then there might be less work for the training algorithm to do. The Ngyuen-Widrow weight initialization (NGWI) [2, ch. 6] is such an approach.

Purely random initialized weights will start out with a weight distribution similar to the one in the following bar plot.

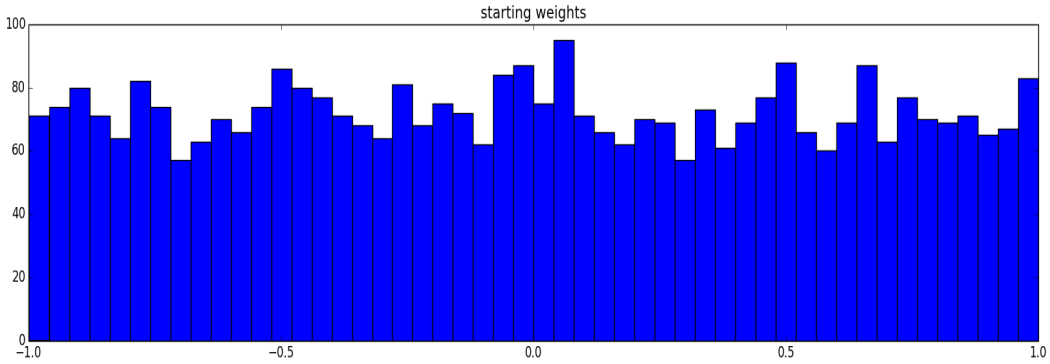

Figure 4.4: Starting weights randomized in the range [-1.0 to 1.0]

And after training it can often be observed that the weights form a distribution clustered around zero.

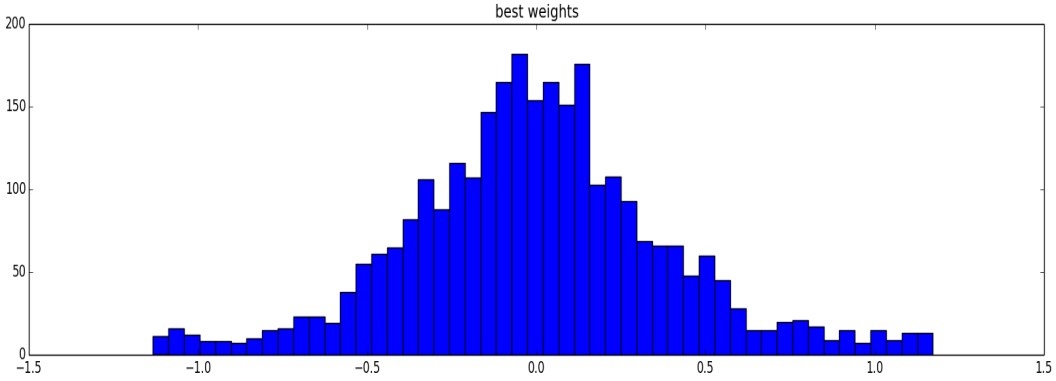

Figure 4.5: Resulting best weights after the model has been trained.

By using the NGWI technique, we get a distribution looking approximately like the following.

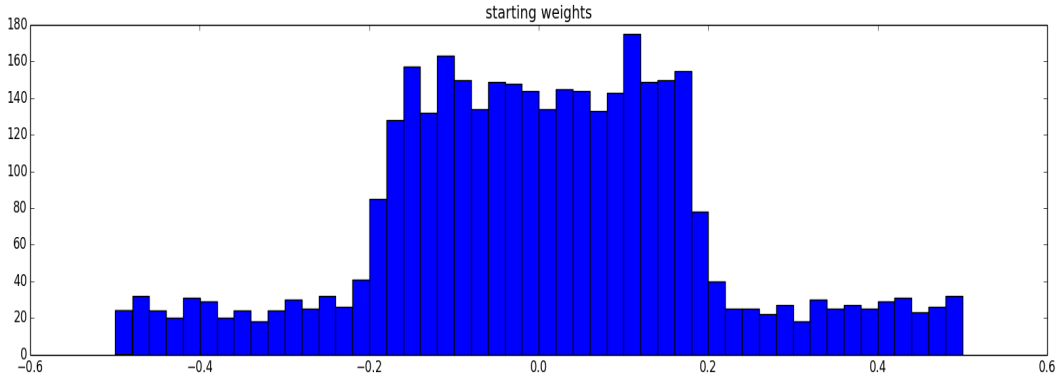

Figure 4.6: starting weights initialized with NGWI

In the NGWI, we start by initializing all the weights, between the first hidden layer and the output layer, with values in the range [-.5 to .5], and the weights between the input layer and the first hidden layer in the range [-1.0 to 1.0]. After this we adjust the weights located between the input layer and the first hidden layer.

The first thing we calculate is a so called 'beta' value.

$$\beta = 0.7 * hn * in^{-1}$$

Where hn is the number off hidden neurons in the first hidden layer, and in is the number of neurons in the input layer.

Next we calculate the 'euclidean norm' for every neuron in the first hidden layer.

$$n = \sqrt{\sum_{i=0}^{i<\omega_{max}} \omega_i^2}$$

Where ω_n represents the nth weight connected to the given neuron.
At last it is time to change to weight values. This is done with following formula for each individual weight.

$$\omega_t = \frac{\beta * \omega_t}{n}$$

Remember that we have previously initialized the weights connected to the input layer in the range [-1.0 to 1.0], we use this value in the formula and overwrite the weight value with the new calculated value for each of the weights between the input layer and the first hidden layer.

4.1.3 Training

When we have settled on an ANN structure and initialized the weights, it is time for the training.

The training can basically be summed up, as being the search for a satisfying minima on a hyperplane defined by the weights.

For this, backpropagation is often used, and backpropagation techniques is also what I have utilized in this project.

The error functions

Since backpropagation starts at the end, it makes since to first describe the 'global' error function, and the 'output' error function. (What I refer to as 'output' error function is often referred to as 'the error function'. But I find that this term easily can make a lot of confusion)

The output error function is both used for the backpropagation algorithm and as input to the global error function.

The output error function is essentially an evaluation of how well it is going with the training, and it is the resulting value of the output error function we seek to minimize during the backpropagation training.

If we were to create and train an ANN to be used in robotics, we would probably make up some complex function based on for example rules of physics and the vision of the robot, and then use this as the output error function.

However, when learning from a data set containing target/expected values it suffices to use a simple linear error function.

$$E = (t - a)$$

Where t is the target value and a is the actual value.

The global error function uses the values from the output error function.

The purpose of the global error function is to inform the observant of the progress and to allow him to define an acceptable level for how much error he will tolerate in the trained model.

For this project, the root mean square error has been used, and it is given a list of output errors collected during batch training.

$$RMS = \sqrt{\frac{1}{n}\sum_{i=0}^{n} E^2}$$

However, I could just as well have used the mean square error.

$$MSE = \frac{1}{n}\sum_{i=0}^{n} E^2$$

The difference between these to formulas, are that the RMS value will be higher.

As this only is used for the observant, it isn't too important as long as the observant knows which global error function is used. RMS might reveal more details when plotting the error, but there might be a minor performance gain by using MSE instead.

Backpropagation

The backpropagation, is the training process itself.

After the output error has been calculated, the resulting error is propagated back in the network such that the individual weights can be adjusted.

We begin the backpropagation by first calculating node deltas for the output neurons. Following is calculated for each of the output neurons.

$$\delta_i = -E * A'(a)$$

Where E is the previously calculated output error, a is the actual output value of the given neuron and A' is the derivative of the activation function, which in my implementation is the derivative of the sigmoid function.

$$A'(x) = x * (1.0 - x)$$

After calculating the deltas for the output neurons, we proceed to calculate the deltas for the interior nodes. For this we use following formula.

$$\delta_i = A'(a_i)\sum_{k} \omega_{ki}\delta_k$$

Where ω_{ki} represents the weights going out from the current node i to a node k in the next layer, and δ_k is the calculated delta for the node k in the next layer.

Note that we do not calculate delta for the bias neurons and for the input neurons.

Next we can calculate the gradients, which will be used to adjust the weights in the end.

The gradient of a weight is defined as the partial derivative of the local error function E with respect to the given weight. The local error for the output neurons is the linear output error function mentioned earlier.

$$weight\_gradient = \frac{\partial E}{\partial \omega_{ik}} = \delta_k * o_i$$

Where k again represents the connected node in the next layer, o_i is the output of the given node and ω_{ik} represents the weight between the two nodes.

This calculation tells us how much each individual weight is responsible for the output error. The calculation is applied to all the weights in the model, and after this we should have a gradient value for each of the weights.

The next thing to do, is the adjustment of the individual weights.

The gradients, surely tells us how much the related weight is responsible for the output error in the model. But if we merely add each of these values to their weight, there is a good chance that we will move too far and then we might end up diverging instead of converging. Therefore we use a so called learning rate to scale the gradient before we adjust the weight. Furthermore, there is a neat trick of saving the previously calculated gradient delta and then add this value scaled with a momentum value. This will force the weight in the direction of the previously calculated gradient delta.

$$weight\_delta = weight\_grad * learning\_rate + weight\_delta * momentum$$

In the first iteration of the training there won't be any previous *weight_delta* so they are initialized to zero values from the beginning.

An example of a value used for the *learning_rate* could be 0.7 and a value for the *momentum* could be 0.3. Though the optimal setup for these values can vary from data set to data set.

When experimenting with the values I have in general observed that it is most efficient to keep the momentum low and the learning rate high.

Batch learning

When we perform the previous described backpropagation, we take one entry from our data set, containing input and target values (the values which we wish to predict), and then we use these values for the backpropagation.

Adjusting the weights after each backpropagation iteration is called online learning, but this method is inefficient.

When doing this we can imagine that there will be a lot of unnecessary moving back of forth in the weights.

Instead of performing online learning, we can use a technique called batch learning.

In batch learning we generally sum up the gradients after having calculated them for the whole data set, and then we use these summed up gradients for the weight adjustments.

I found this method to be inefficient for my time series data sets, as the values grew too big, so I have made my own interpretation of batch learning.

As my data sets consists of sequences of data, I take one of these sequences at a time, run all the entries through the backpropagation and sum up the gradients. When this is done for all the sequences, I take the average of each of all these sums. This method has proved more efficient.

The method can be formulated following way.

$$weight\_grad = \frac{1}{\#DS} \sum_{ds \in DS} \sum_{en \in ds} prop(en)$$

Where DS is the data set containing all the data sequences, ds represents a data sequence, en represents an entry in a data sequence and $\#DS$ evaluates the length of DS which results in the number of total data sequences in the data set. $prop$ is the backpropagation method which calculate the gradient for each weight given a data entry containing input and target values.

When the algorithm has been through all the data sequences in the data set, and the $weight\_grad$ has been calculated for all the weights, then the weights are adjusted with either the method previously described (using learning rate and momentum), or the more efficient method which I will describe next.

Resilient backpropagation (RPROP)

Resilient backpropagation (RPROP) [2, ch. 5][1], is not only a faster training method it also needs less configuration than the old form of backpropagation. In the old backpropagation method, we need to adjust the learning rate and the momentum, we don't need to do that for RPROP.

In my implementation, I have ended up using an enhanced version of RPROP, simply called RPROP+.

With the RPROP method, an individual update value is used for each weight, in place of using one static global learning rate. This update value

[1] `http://www.heatonresearch.com/wiki/RPROP`

is adjusted as the learning progresses.

It is determined how to adjust the individual update values, based on the individual calculated gradients.

All these update values are initially set to a value of 0.1, which works pretty well. However, when resuming training of an already trained model, these values will most likely be way too high. Thus it would be beneficial to save the update values along with the weights for a trained model.

However, this isn't currently done in the implementation. So when resuming an experiment which uses the RPROP+ training method, -a huge spike will be observed in the beginning.

It might be that it in general is unnecessary to adjust parameters for RPROP, however, there are a few constants which are used by this training method.

In my implementation, they are configured with the following values, and they seem to work pretty well for any of the data sets I have tried out.

$$delta\_min = 10^{-6}$$

$$delta\_max = 50$$

$$neg\_eta = 0.5$$

$$pos\_eta = 1.2$$

$$zero\_tol = 10^{-12}$$

delta_min specifies the minimum value that an update value can go to, and *delta_max* specifies the maximum value that an update value can go to. The *zero_tol* is a tolerance value used to determine when a floating point value should be considered zero. At last the *neg_eta* and *pos_eta* are used for scaling the update values.

When using RPROP, we also need to keep track of two kinds of values for each weight during training.

$$previous\_weight\_change$$

$$previous\_gradient$$

The RPROP method also needs a 'sign' function to determine the sign change of a given gradient change. In the sign function, I am using a function *is_zero* which uses the *zero_tol* value to determine if the given value is zero (in the implementation, this is really just a rounding function).

Algorithm 1 $sign(grad\_change)$

```
1: if is_zero(grad_change) then
2:     return 0.0
3: else if grad_change < 0.0 then
4:     return -1.0
5: else
6:     return 1.0
7: end if
```

A function *calc_update_val* for calculating the new update value, is also defined.

Algorithm 2 $calc\_update\_val(grad\_change, old\_update\_val)$

```
1: if grad_change == 0.0 then
2:     return old_update_val
3: else if grad_change > 0.0 then
4:     return min(old_update_val * pos_eta, delta_max)
5: else
6:     return max(old_update_val * neg_eta, delta_min)
7: end if
```

It is in this function the new update value is calculated.

Furthermore, we also need a function for determining the weight change. This is essentially the function which determines if the previous weight change should be rolled back.

Algorithm 3 $calc\_weight\_change(grad\_change, gradient, update\_val, prev\_weight\_change)$

```
1: if grad_change >= 0.0 then
2:     return sign(gradient) * update_val
3: else
4:     return −prev_weight_change
5: end if
```

As mentioned earlier, the RPROP+ method is an enhancement of the original RPROP method, and what differs between these two methods is that in RPROP+ the previous weight changes are reverted if the sign changes in the current iteration.

The process of the RPROP+ method is easiest described with following pseudocode.

Algorithm 4 The RPROP+ procedure

1: $grad\_change = sign(gradient * prev\_gradient)$
2: $update\_value = calc\_update\_val(grad\_change, update\_val)$
3: $weight\_change = calc\_weight\_change(grad\_change, gradient, update\_val, update\_value, prev\_weight\_change)$
4: $prev\_weight\_change = weight\_change$
5: $weight+ = weight\_change$
6: **if** $grad\_change >= 0.0$ **then**
7: $\quad prev\_gradient = gradient$
8: **else**
9: $\quad prev\_gradient = 0.0$
10: **end if**

Where *grad_change* reflects if there has been a change in the sign of the calculated gradient. This essentially informs us if we have moved too far.

If it is the case that this value is negative, then we have moved too far, and it is time to revert the weight changes from the previous iteration.

Furthermore, when we calculate the new update value we also do it on the basis of the sign of the *grad_change*. As the *grad_change* value informs us if we are ascending or descending in the convergence, we can decrease the update value if we are descending such that we wont move beyond the minimum. And if we are ascending we increase the update value such that we will move faster in search for a minimum.

Using this method for training the ANN models, I have observed a huge speed up in the convergence.

Recurrent Neural Networks

The previously described basic ANN structure is mostly used for tasks like classification where there is no element of time in the data. However, ANN structures for handling time series data has been developed as well.

For this project, I have experimented with what is referred to as Simple Recurrent Neural Networks(SRNN) which is the first kinds of RNN developed.

The basic idea in SRNN, is to save the outputs of some of the neurons in so called context neurons. And in the next iteration we will use the weighted outputs of these context neurons, by feeding them into the model. This way we provide the model with a limited short term memory.

There are two common SRNN structures, the Elman structure and the Jordan structure.

In the Elman structure, we build a context layer based on the last hidden layer in the model, and in the Jordan structure we build a context layer based on the final output neurons in the model.

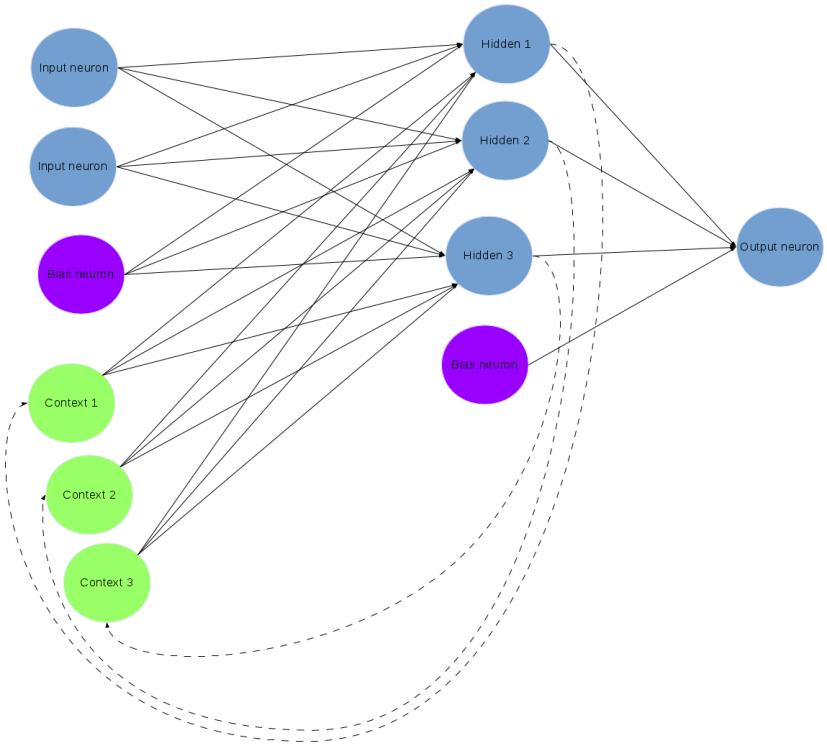

Figure 4.7: Example of Elman structured RNN, the green nodes are the context neurons and the dotted connection simply tells which neuron it is copied from

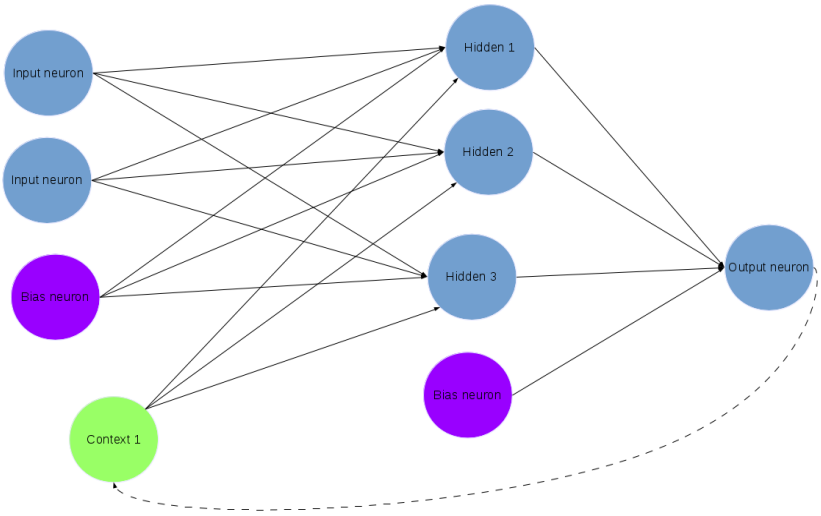

Figure 4.8: Example of Jordan structured RNN

What is common for these two structure types, is that the weighted output of the context neurons is fed into the first hidden layer.

In my implementation I have made it possible to create both structures and also to combine them.

When the RNN based ANN is initialized, the context neurons are all set to a value of zero. Furthermore, will the context layer be reset before calculating the output for a given data sequence (also during training).

Chapter 5

Data

A good predictive model needs data to be trained with, and if this data is just inputted to the model as is, it will most often lead to bad results or just very slow convergence in the training.

So when creating a predictive model it is very important to put some thought into, not only what data to serve to the model, but also if the data needs preprocessing.

5.0.4 Preprocessing in general

When dealing with an ANN model, there is one relative simple optimization one can make before even creating and training the ANN, and that is normalization of the data.

ANNs are most often initialized with random weights which are floating point numbers in the range from -1.0 to 1.0.

Now if we imagine giving the ANN, some huge numbers as inputs, say for example values in the range 766325573883111 to 866325573883111, then it would most likely have to scale the weights a lot down.

And on the contrary, if we give the ANN some very small numbers as input, say for example 0.00000000000000006253321 to 0.00000000000000007253321, then the weights would most likely have to be scaled a lot up. And it gets even worse if we imagine a situation where we have very mixed ranges of input values.

With very mixed value ranges, the backpropagation algorithm techniques (PROP) used for training the ANN, risk of getting into situations which makes it very slow to converge. This is because the shape of the error function will end up looking like some sort of very long valley, which is very undesirable because then the backpropagation algorithm might have to travel longer to reach convergence.

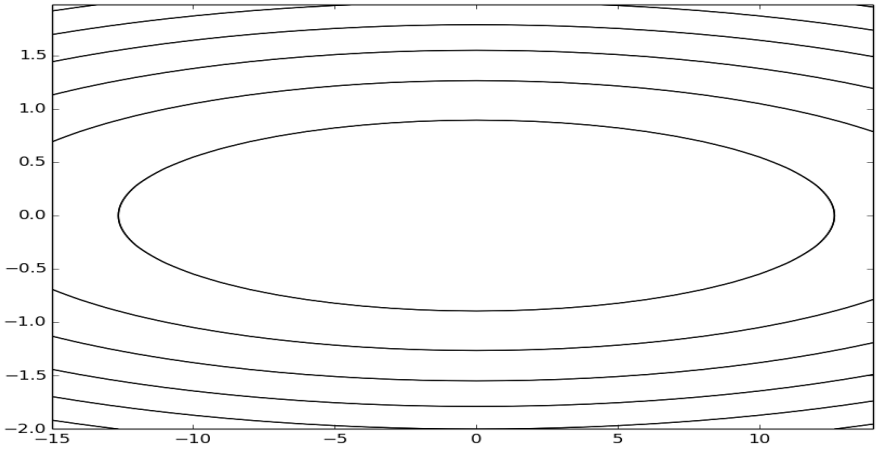

Figure 5.1: Valley shaped error function

What is more desirable, is a shape that looks more circular, this way the descend will be more uniform from all sides.

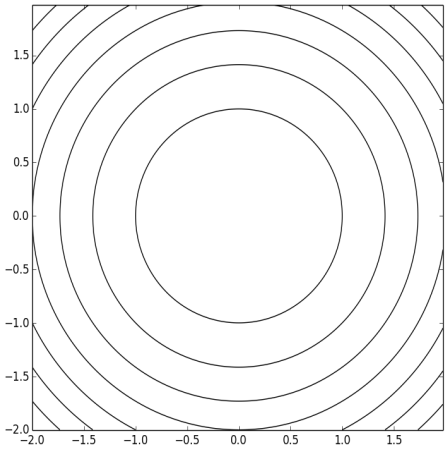

Figure 5.2: Circular shaped error function

And by normalizing the input values, we approach this circular shape.

5.0.5 Preprocessing of the data produced by the simulator

The data which the simulator generates, consists of timestamps which values are milliseconds since the Unix epoch. Already here, it is obvious that the data needs preprocessing, because of the before mentioned reasons.

The first thing I do to this data, is to find the very first time stamp among all the sensor time stamps, and then normalize all the time stamps using this value as point zero.

The next thing to do is to generate, what I refer to as data sequences, which will contain the patterns that the ANN should learn from and recognize.

Below is a diagram showing an example of time stamps for a setup with only 3 sensors, and one light.

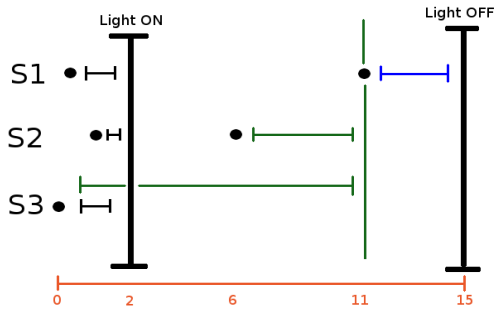

Figure 5.3: Example of registered timestamps for events from sensors S1, S2 and S3, and timestamps used for the on/off interval of a given light. Each dot represents a time stamp for the given sensor. The vertical black lines indicate the timestamp for when the given light was turned on and off. At the bottom of the diagram is the timeline, with markers for the timestamps.

Now that the timestamps are normalized with a new point zero, the next thing to do, is to compose the data sequence.

The first thing to calculate for this, is the duration from the sensor timestamps registered just before the light was turned on at time 2. These are visualized as black horizontal bars on the diagram.

The resulting values for this is example are {Sbd(1)= 2, Sbd(2)= 1, Sbd(3)= 2}, where Sbd(N) stands for Sensor-before-delta and N is the id of the sensor.

The next thing to do, is to calculate the delta duration values for the sensors between the light on and off timestamps. An example of such durations are visualized as horizontal green bars on the diagram.

It is calculated by first creating a set of sensor timestamps between the on and off events. And the resulting set from this operation, is the set of timestamps {6, 11}.

The next thing I do, is to iterate over this set, and for each value, I find biggest possible time stamp for each sensor which also fulfills the proposition of being either less than or equal to the current value from the set. And at last I subtract the current value from the set, from each of the found timestamps.

For the given example, this results in:
{Sad(1) = 6, Sad(2) = 0, Sad(3) = 6} and {Sad(1) = 0, Sad(2) = 5, Sad(3) = 11}
where Sad stands for Sensor-after-delta.

The last thing to do, is to compose the final data sequence of the parts above. However, when generating data sequences for training we also need a target value.

The target value is the value which the ANN is supposed to be able to predict given the corresponding input data. Sometimes the target value is also referred to as 'the response value' or simply 'the expected value'.

There are many ways to compose the data sequences, but I have settled on the following way to build them, based on some experimentation.

The first entry in the sequence, is the value 1.0 followed by the Sbd values followed by zeros up until the target value, and at last, the target value which is the value of the whole duration between light on and light off.

The very first value in the entries serves as an indicator that informs id the given entry is from before the light was turned on.

The next entries are created on the basis of the list of sets of Sad values, and is composed as: the value 0.0 followed by the Sbd values, followed by Sad values, and at last the target value which is the value of the duration from the zeroed time stamp in Sad, to the light off event. An example of the target value for such an entry, is visualized in the diagram as a blue horizontal bar.

The resulting data sequence for the timestamps in the example is following matrix of durations:

1	2	1	2	0	0	0	13
0	2	1	2	6	0	6	9
0	2	1	2	0	5	1	4

This is the kind of data with which the model is trained. In this case, it would use the first 7 values in an entry as inputs and the last value as target value.

Though there is on more thing I proceed to do before training a model with the data, and that is normalization of the durations themselves.

All the calculated durations up to this point are in milliseconds, and these values are undesirable because they are huge.

In my current implementation, it is also very important that the target values doesn't go above the value 1.0. This is because I am using a sigmoid activation function in the final output neuron.

So what I currently do in the implementation, is to use what I have chosen to call a "time normalization factor". This value is simply used as a denominator for dividing all the calculated durations (both input and target values, and later it is used to scale up the calculated output values of the trained ANN, to produce recommendation values.

The "time normalization factor" should be chosen, such that no target value is above 1.0. However, it is desirable to have the target values spread out in the range 0.0 to 1.0 as much as possible, -so the chosen value shouldn't be too big.

(Note that it might be possible to switch to using a linear activation function for the final output neuron in the future. And that this would eliminate the need for the target values to be below 1.0. It is, however, still desirable to keep the target values low, so the "time normalization factor" will still be needed)

It might seem obscure to repeat the delta values from "before the light on event", but the alternative would be to input these values to the same inputs as "the delta values after the turn on event". This wouldn't be very logic to do, and it only showed bad results when I did the attempt. Furthermore, I believe that the sensor registrations leading up to a light on event is too important to be ignored.

I also observed a noticeable speed up in the training process when adding an identifier value representing if the given entry is before or after the turn on event.

Providing as much information as possible to the ANN, seems to be the way to go when one wants to speed up the training.

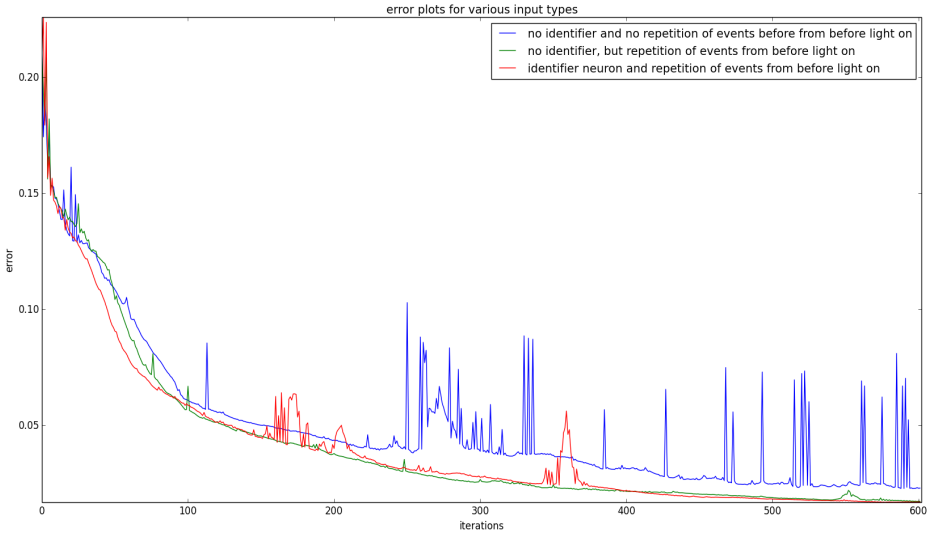

Figure 5.4: Plots of the training process for 3 different input types. It is clear to see that the input types with repetition of the events from before the light was turned on, converges significantly faster. (All the models where trained based on the same data set and with a setup with 2 layers of hidden neurons with 50 and 25 neurons and a RNN setup as a combination of Jordan and Elman. Furthermore, the networks were all initialized with Ngyuen-Widrow algorithm and used RPROP+ as training method)

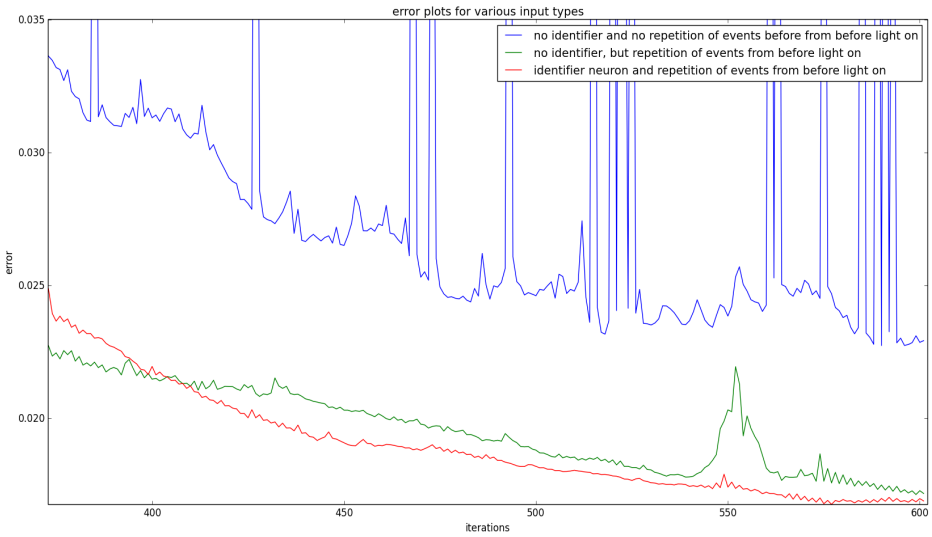

Figure 5.5: The same plot as above, but zoomed in such that it can be seen that the input type setup with identifier neuron converges slightly faster than the one without.

Chapter 6

Implementation

I chose to make the implementation of the training process in the Python programming language, using NumPy[1] for the matrix calculations and using matplotlib [2] for visualizations.

I used Python for this, because this was the most complex part. And given that I already had a lot of experience with Python it, would make the development process faster, than if I did the implementation in Scala which was completely new to me.

I only did a minimal implementation of the output (recommendation) calculation for the AI in the simulator, and chose to make a minor web service in Python with Django[3], from which the trained model could be loaded into the simulator by passing on the model information as Json.

After doing this, I realised that I could make a web interface from which the models could be trained and the results plotted.

This is much more efficient when experimenting, than adjusting the scripts and manually executing them from the console.

In the simulator I use Breeze[4] for calculating the outputs of the model. I use scalaj[5] for retrieving the model from the Django web service, and I use liftweb[6] to handle parsing of the retrieved Json in the simulator.

6.0.6 The training of the model

All the training of the ANN, is done in the class "ANN" which is located in the file ANN.py.

[1] `http://www.numpy.org/` (math module for Python)
[2] `http://matplotlib.org/` (plotting module for Python)
[3] `https://www.djangoproject.com/` MVC web framework for Python
[4] `https://github.com/scalanlp/breeze/wiki/Breeze-Linear-Algebra` (Scala math library)
[5] `https://github.com/scalaj` (a simple bare bone http client for Scala)
[6] `http://liftweb.net/` (web framework for Scala)

The basic idea with the class, is that one instantiates it with parameters describing which dataset to use, the desired structure of the neural network, and the training methods and initialization to be used. And thereafter, one initiates the actual training.

This class is the very core of the whole project itself, and many adjustments, corrections, modifications and enhancements has been made to this class while experimenting and researching.

The end result of this class, is that it makes one capable of conducting the experiments which I found to be most successful or interesting while making my experiments.

The next two sequence diagrams serves to give a quick overview of the execution flow when training an ANN via the web interface.

The flow has been split into two sequence diagrams in order for them to be readable. Furthermore, I have excluded a few method calls from the second diagram which is focused on the flow inside the ANN class.

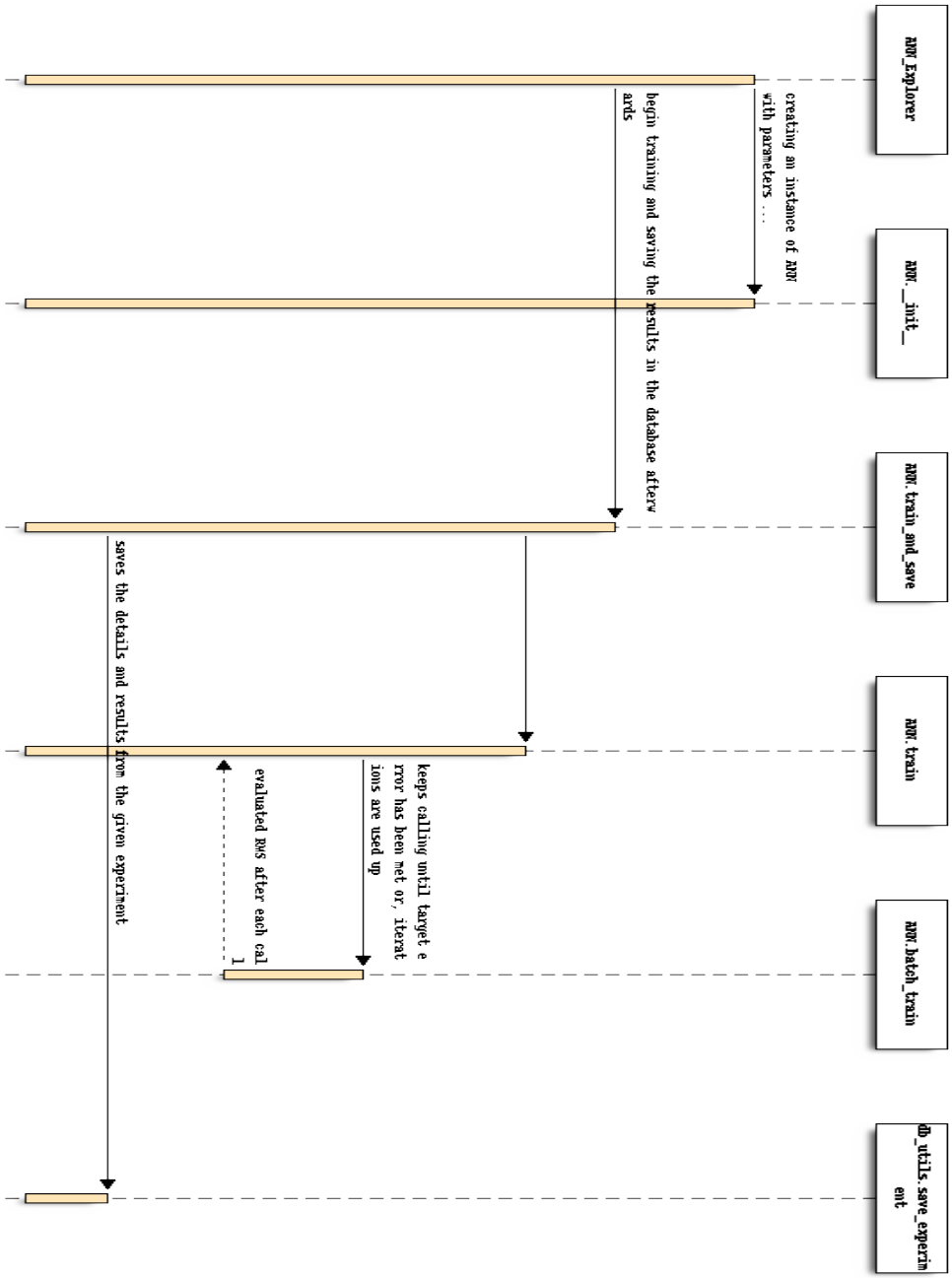

Figure 6.1: Brief overview of the execution flow of the ANN training initiated by the user via the web interface. (The diagrams are created with seqdiag `http://blockdiag.com/en/seqdiag/index.html`)

The ANN class is instantiated from the ANN_Explorer web interface using the parameters specified in the interface. After this the function *train_and_save* is called on the class to initiate the training and save the resulting model afterwards.

The *train_and_save* function calls a *train* method. In the *train* method, the *batch_train* method is called until the specified number of max iterations is used up or the desired global error value has been reached.

In the next diagram, the execution process of the training itself is shown.

This process is very similar to the description of the RPROP+ training method found in the theory chapter.

In the *batch_train* function, we start with iterating over the data set containing the data sequences. For each of these data sequences we call the function $calc\_summed\_grads\_for\_seq$ which will calculate gradients for each of the entries in the sequence and return the sum of these gradients. Note that the gradients are contained in a list of matrices, where each matrix represents the weights between two layers.

Next, all these summed gradients for the data sequences are averaged, and at last used for the RPROP+ process described in the theory chapter.

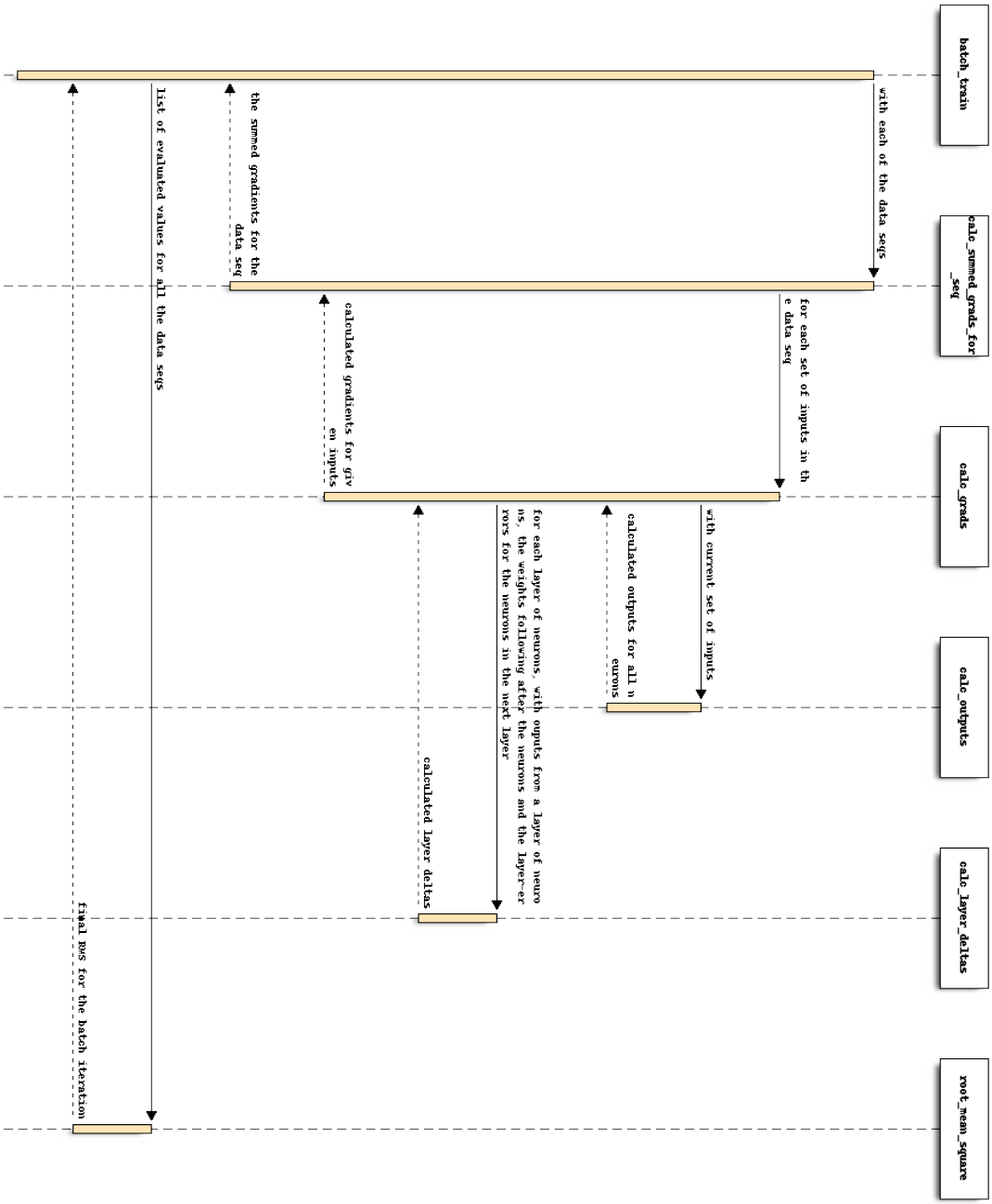

Figure 6.2: Brief overview of the execution flow of the ANN training inside the ANN class.

(Please refer to the code itself and to the Theory chapter about neural networks for details about the various calculations in the functions.)

6.0.7 The database

From the beginning, I decided to add a database to the system, because I would like to save details about the experiments and be able to reload weights and plot results.

For this project I am using an SQLite database[7]

A SQLite database simply consist of one file. This file is located in the Python project at following location: /ANN_Explorer/neural_explorer.db

Below is an overview of the tables in the database, and after this I will provide a brief description of the purpose of the various tables.

[7] http://www.sqlite.org/

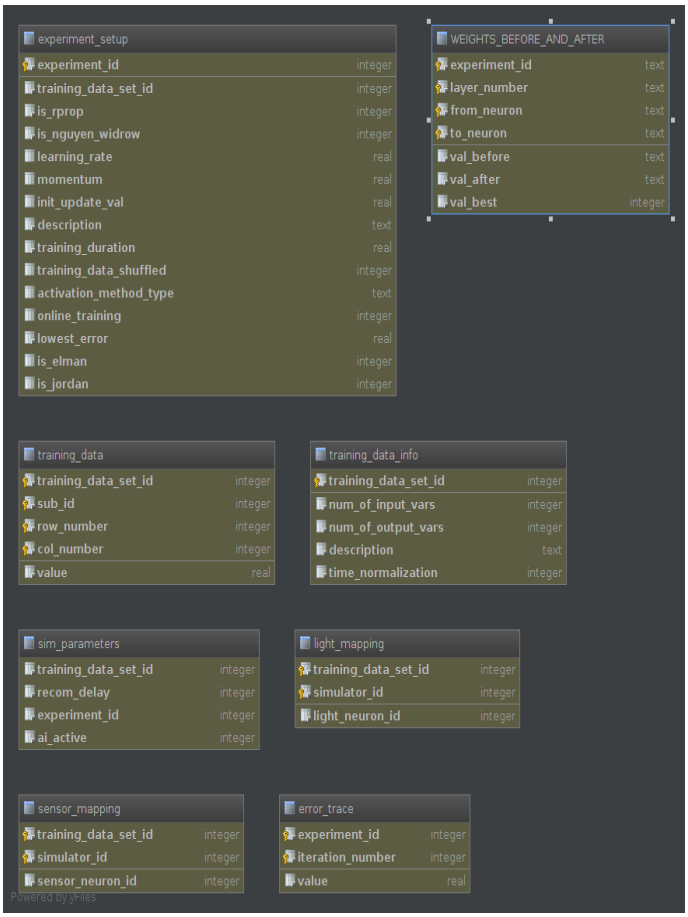

Figure 6.3: Overview of the database setup used in neural_explorer.db. The types of the fields aren't too important, as SQLite is dynamically typed `http://sqlite.org/datatype3.html`. (In this diagram it can be seen that the types in the weights_before_and_after table are very strange. This is probably due to an accident while recreating the tables.)

experiment_setup

This is one of the core tables in the database.

In this table, the details about a given experiment is saved after it has been trained.

The purpose of saving all this information, is not only to examine the experiment details afterwards, but also to be able to resume training for a trained network later on.

error_trace

This table contains the recorded error for each iteration of the training of a given experiment.

This information is of interest, because we can plot error_trace data for various experiments together, and based on this we can evaluate which setup performs the best in the training.

weights_before_and_after

This table is actually the most important in the database, as it contains the weights for a given experiment which make up the actual ANN model.

I save both the initial weights, the weights after the training. (The table also contains a column best weights. The reason for this were that the error curves were incredible unstable at some point during development, so at that point it was beneficial to save the best weights seen thus far. However, I found the mistake that caused the unstable progress and fixed it. This means that this column actually contains the same values as the one with weights after training.)

There are several reasons to save the weights.

The most important reason is that we might want to repeat an experiment with other parameters, and since the weights are initialized randomly, we have to save them in order to do this.

Another reason is that we might wish to compare the starting weights to the trained weights.

And the last two reasons are simply that we need the weights in order to use a trained ANN model, or to resume training of a trained ANN model.

It is possible to recreate a trained ANN model, by using the information from this table together with the related information in the experiment_setup table.

training_data_info

This table contains basic information about a given training data set.

training_data

This table contains the actual values of the training data sets.

Together with the training_data_info table, a given data set can be recreated and used for training an ANN model.

sim_parameters

This table should only contain one entry.

The purpose of this table, is simply to contain the parameters for which experiment model to use in the simulator, how much "recommendation delay" to add to the recommendations in the simulator, which training set to display in the GUI and finally if the AI should be active or inactive in the simulator.

light_mapping and sensor_mapping

These two tables are used by the AI implementation in the simulator to correctly map the sensors of a given setup to the input neurons of the loaded model.

6.0.8 The user interface

The web user interface is composed of HTML5, javascript and the Django view template language[8].

The user interaction is mainly handled with javascript and Ajax[9] calls to the Django service which exposes the functions found in the views.py file.

The javascript and css based module Bootstrap UI[10] has been used for the layout, and the javascript module jQuery[11] has been used to handle the Ajax calls and the dynamic manipulation of the web interface.

The purpose of the web interface is to make it easy to load data sets into the database, set parameters for the simulator, train ANN models on the data sets and visualize the results. Furthermore, the web interface, exposes a button for activating or deactivating the AI in the simulator.

6.0.9 The AI implemented in the simulator

A minimal implementation for an AI, has been made into the simulator. It is basically just a simple control unit that creates the data sequences, and then inputs them to the trained ANN model in order to get a recommendation for when to turn off a given light source.

[8] https://docs.djangoproject.com/en/dev/topics/templates/
[9] http://en.wikipedia.org/wiki/Ajax_(programming)
[10] http://angular-ui.github.io/bootstrap/
[11] http://jquery.com/

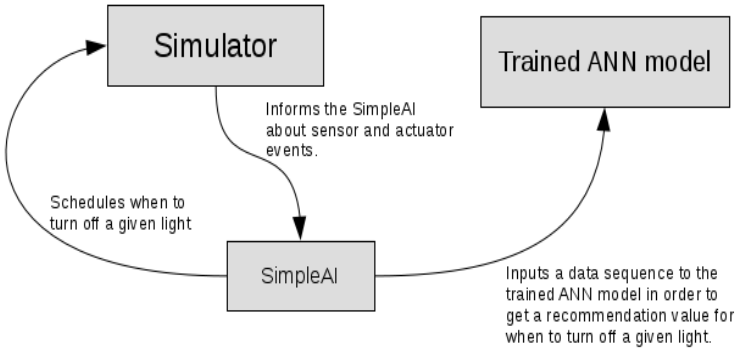

Figure 6.4: Diagram displaying the basic interaction between the SimpleAI, Simulator and trained ANN model.

The flow of the AI is controlled from the "SimpleAi" class which subscribes to sensor and light switch events in the simulator.

When the simulator is started up, it will initialize the SimpleAi class, and during this initialization, the SimpleAi class will perform a call to the Django web service to fetch the configuration of the AI and the trained model for the AI to use.

The trained ANN model is initialized into the simulator based on the fetched data, and the sensor ID mapping to the input neurons and the various other parameters are initialized.

If the AI is defined to be active via the user interface, then the AI will pay attention to the events in the simulator.

When events are registered by the AI, they are appended to a list which is referenced in a map with the given sensor or light ID.

When sensor events are registered, it is also determined whether or not to proceed with the calculation of the data sequence and the recommendation calculation given by the ANN model.

The possibility of adding a small extra delay to the recommendation has been added. This delay value is simply added to the calculated recommendation value of which the ANN model is responsible.

This delay value is necessary, because the recommendation values given by the ANN model sometimes are a little too optimistic. These optimistic recommendations will not only stress the simulator, but will also lead to flickering light which is very undesirable.

I believe that setting the delay value to approximately the same value as the cooldown value of the sensors (2000 ms in the simulator) is sufficient.

Chapter 7

Experimenting and testing

During the development and after, I have done a lot of tests, both in the simulator and when exploring the training of the ANN models.

In this chapter I will mention some of my experiences.

7.1 Experiments in the simulator

I have thought a lot about how to properly set up experiments in the simulator.

A general problem when setting up scenarios meant for training, is that the execution order of the scenarios when recording the data, is important.

My current model uses values for all the sensors each time it tries to calculate a recommendation for when to turn off a light. This means that the sensor registrations for the previous played scenario might play a role for the next scenario. So when testing I have sometimes tried to first play and record the scenarios in the order which I created them, and after that I have either played them from the last to the first again or in a random order. I have also tried to repeat the scenarios with slight variations.

It is hard to draw any conclusion on the effectiveness on this approach, but it provides more data to the data set and that with repeated pattern sequences which hopefully will help the model learn which patterns are important.

Another important thing to think about when setting up tests, is the fact that the patterns has to be more or less consistent. After all we are trying to capture human patterns which in large are based on habits.

Yet another related thing to think about, is how humans would behave if they moved around in a house where they were forced to turn off the light upon leaving a room. Most likely the human would first enter the next room, turn on the light in there, then go back and turn off the light in the previous room. I did some minor tests this way, but then decided to make it easier for myself to setup scenarios by consistently turning off the light upon leaving a room.

7.1.1 The tests

I have mainly played around with three basic setups. After starting out with some complex scenarios, I realised that I had to start with something extremely basic.

So I created a setup in the simulator with only one light in a room and 6 sensors spread out in the given room and a hallway. Furthermore the patterns for this scenario had to be simple and consistent, so I created two basic patterns.

The first pattern is an actor entering from the left in the hallway, then briefly entering the room and turning the light on and off and quickly leaving again. And the second pattern is an actor entering the hallway from the right, who enters the room, turns on the light then stays in the room for approximately 15 seconds, and then leaves turning off the light behind him.

Evaluating the model for this pattern went pretty well, as expected. The model would predict to turn off the light quickly when an actor, coming from the left would enter. And when an actor coming from the right would enter, it would correctly recommend a longer duration before turning off the light. Furthermore, these longer predictions were all in the range 12 to 18 seconds which were also the durations used during the training data generation.

The setup for these tests is found in the file *one_light.xml*.

The idea for the next setup, was two use two rooms, add lights in the hallway and having up to two actors.

The patterns for these scenarios are more less described by the following pattern descriptions.

- An actor goes through the hallway.
- An actor enters one of the rooms and stays there for for a duration then leaves the building.
- Two actors enters the two rooms, more or less simultaneously and stays there for a while before leaving the building. Sometimes the same room and sometimes a room for each.

Based the before mentioned scenarios, I made up around 30 scenarios to record and train a model with, and 4 scenarios to test the model with afterwards.

I ran the tests 20 times in random order and observed the results of the model predictions. I used a model which had received a lot of training on one with less training. The model with the lowest RMS had the fewest errors.

The results of the tests can be found in appendix B.

7.2 Experiments with training of various models

Regarding the training of the model itself, there are a lot of ways to combine training methods, initialization and structures.

A general observation, was that ANN structures with two hidden layers of for example 50 and 25 neurons would converge the fastest.

But the most interesting things I noticed while trying out different configurations, was that a basic non-RNN model actually were able to converge fairly well if it was configured with 2 hidden layers composed of for example 50 and 25 neurons. When configured this way, the non-RNN model would actually converge almost as well as the RNN configurations.

However, I suspect that the non-RNN model will have problems with calculating correct predictions if it is faced with some very long and complex sequences.

All the patterns which I have generated in the simulator, has been fairly simple. The sequences between the light on and off events has been relative short and non-complex, in the sense that I have mostly made the actor stand still for a fixed amount of time.

Following table contains some details about some of the trained models trained with the data generated by the setup found in *labs.xml*.

Experiment ID	lowest error	hidden neurons	Elman	Jordan	Nguyen-Widrow
76	0.013352453371426324	[50, 25]	yes	no	no
75	0.014040338489100744	[50, 25]	no	yes	no
77	0.015111727441964833	[50, 25]	no	no	no
92	0.01591702220357906	[50, 25]	yes	yes	yes

(Note that a non-RNN model simply is created by deactivating both the Elman and Jordan structure)

Chapter 8

Future Development

> "It is vain to do with more what can be done with fewer"
>
> William of Ockham

When we create software we do our best to deal with the current problems at hand, and then we try to predict how it will be used.

One can spend countless hours and resources on trying to predict how software will be used, how it will perform in various scenarios etc. But if too much time is used on this, the software will never get released or will be outdated when it is released. It also often turns out in the end that a lot of these predictions and assumptions were wrong or unnecessary anyways.

Furthermore, when software is being taken into use in real life situations, it will be exposed to new unpredictable situations all the time, simply because the world constantly changes. It is also very common that bugs and errors first are discovered when the software is taken into use.

An excellent heuristic one can use in software development, when faced with a lot of choices, is to go for the simple solutions.

I think it is very important to keep these arguments in mind, both when developing and when choosing what to focus on in the future. And these thoughts are also good arguments for doing some real life testing outside the isolated sandbox environment that a simulator provides.

8.1 Improvements

I propose following improvements to the system, in order for the model to be used most effectively in a real life implementation.

With some of these enhancements, I believe that there is a good chance that the system effectively will output the best possible recommendations for turning off the light.

8.1.1 Optimization of the recommendation calculations

In the current implementation there is a lot of repetition in the calculations of recommendation values, due to the fact that each time a sensor is activated the sequence has to be calculated all over again, and then the same calculations has to be done again in the ANN model to ensure that the context neurons has the correct values.

A fairly easy, but important, optimization in the system, would be to save both the values of these sequences and the last values of the context neurons, in relation to a given light source. These values should then be declared invalid or overwritten when the given light source is turned off.

8.1.2 Evaluation Feedback

The current implementation is static, in the sense that the training of the neural network is initiated manually. In a future implementation it would be necessary to update or retrain the neural network once in a while. To do this autonomously, some feedback from to the system is needed. This feedback could for example be when a sensor is activated shortly after the related light has been turned off. When this happens, the data sequence of sensor registrations between the light on and off events has to be saved, and then the neural network has to resume its training with the new extended data set.

But when this kind of situation arises, the system has a problem with determining when to turn off the light. If the neural network is asked for recommendation, it will most likely give a wrong prediction since this is a new and unknown pattern. This problem could be countered by just defaulting to a static fall-back recommendation value, or maybe a percentage of the first given recommendation for the given pattern. If the same situation plays out at the end of this duration, it simply uses the fall-back value again for a new recommendation. The recorded data sequence for these kind of situations, should last from when the light is first turned on until the ending of the last given fall-back recommendation, ignoring any turn off events in between.

But now we face a new problem. If the system uses before mentioned fall-back mechanism, then the recommended values might only ever increase, and patterns where the recommendations should be shorter, aren't handled. This might result in a situation where the recommended durations only grows longer and longer.

This problem can maybe be countered by paying attention to situations where no sensor registrations are registered shortly before the light is turned off by the recommendation. In this kind of situation, the actor has most likely left the room much earlier than expected. The solution to this kind of situation, could be to subtract a percentage of the first given recommendation, and then use this new value in the recorded data sequence, which is submitted for retraining of the neural network.

8.1.3 Feeding more parameters into the model

An obvious improvement to the system is to use more parameters for the neural network. This could be values like time of the day and current weather.

It could also be very interesting to use timestamps from events like living-room-television on/off, oven on/off etc.
In short, all electronic house devices that humans intact with.

Furthermore, it could be interesting to count numbers of active MAC-addresses originating from smart phones on the router, and use this value as input.
This could inform the model about the number of persons in the house, as most people carry smart phones which automatically connects to the WI-FI when possible.

Using more parameters, will supply the RNN model with more information about the context of of a given pattern of data, thus improving the quality of the predictions.

8.1.4 Trashing outdated and invalid data

Another necessary enhancement of the system, is to trash outdated data sequences. This could be done by automatically performing some analysis on the recorded data sequences once in a while. The job of this analysis would be to detect if a given old pattern significantly differ from similar new recorded patterns. The analysis has to take values like time of the day, season, and weather into account.

The situations where such an analysis should be performed, could for example be when a family gets a child and therefore changes their behavioral patterns in the house significantly.

8.1.5 Optimizing the training process

Yet another thing that might be of interest for future investigation, is optimization of the training process.

The networks used for the simulator has all been fairly small and so has the data sets. In a real implementation, the networks will be larger and the data sets might be huge, both resulting in longer training durations.

Use an existing ANN framework

An easy way to gain increased performance on the training process, might be to simply use an existing neural network framework, like Encog [1], or PyBrain [2].

These frameworks are created and maintained by multiple researchers, who should have a lot more knowledge and experience with neural networks, than what a bachelor level student can gain in a couple of months, and therefore theres a good chance that their implementations will perform significantly better.

Use another training method

If it is chosen to reuse the current implementation, then it might be possible to gain better performance on those data sets, by using a better training method for the RNN.

Training methods worth investigating could be iRPROP+, which is an enhancement of RPROP+, or the LMA (Levenberg-Marquardt algorithm) training algorithm.

Some research has shown that iRPROP+ is the optimum RPROP type algorithm [3], and LMA is an even more sophisticated training method than RPROP, and is claimed to outperform RPROP based algorithms in many cases. [4]

Exploit parallel processing methods

Yet another way to gain better performance could be to exploit the massive parallel processing of GPU's, or maybe make an FPGA implementation of the training process.

Research in GPU programming promises great performance gains for systems that are performing a lot of matrix computations. And FPGA's are often used in the industry to speed up programs by reimplementing parts of the software programs, that can be parallelized, in a hardware description language and deploying it to an FPGA.

A point in the code where some relative simple parallelization could be made, is when the summed gradients are calculated for each sequence. A relative simple optimization would be to calculate the sequences in parallel as the process used for calculating the summed gradients for a sequence, can be made independent.

[1] `http://www.heatonresearch.com/wiki/Main_Page`
[2] `http://pybrain.org/`
[3] `http://www.heatonresearch.com/wiki/Resilient_Propagation`
[4] `http://www.heatonresearch.com/wiki/LMA`

Random sampling

Random sampling could be used to deal with the problem of huge data sets. We could simply shuffle the training data sequences at the beginning of each iteration (not the content of the sequences themselves!), and then use, for example, the first half of the data for training and the last half for evaluation at the end of each training iteration.

However, if the data set is too small, then this technique might just lead to longer training time, and the training process will likely be extremely fuzzy, with a lot of ups and downs in the error curve.

I would hypothesize that it might be possible to determine whether or not a data set is ready for random sampling based on this fuzziness.

8.1.6 Finding sensor-light relationship with correlations

A limitation of the current implemented system, is that it relies on user provided information about which sensor is connected to which light.

It might be possible to automatically derive this information, by simply calculating correlations between the light on/off events and the sensor registrations.

However, it might be difficult to implement this in a real life implementation, because one might have to deal with multiple sensors connected to one light, or some sensors might be connected to multiple lights.

Determining how many sensors are connected to a given light might not be so trivial.

8.2 Active control from the beginning

Another limitation for the described system, is the fact that it has to be passive for the first duration while it gathers data from the habitants about their patterns.

Furthermore, it will very likely adopt any bad habits of the habitants not turning off the light properly during this period.

8.2.1 Using dumb recommendations from the beginning

It might be a good idea to simply use static estimations from the beginning, instead of relying on the habitants to correctly turn the light on and off when they enter and leave rooms.

Then the sensors would act exactly like the plain old dumb sensors do, until enough data has been gathered.

This could effectively be combined with the methods previously explained in subsection "Evaluation Feedback".

And by doing this we make the system even more autonomous.

8.2.2 Data gathering and analysis of general patterns

By gathering data from hundreds or thousands of houses, it might be possible to analyse this data and find some general patterns which can be used to, sort of, kick start the training of the ANN.

There is a very good chance that a lot of the gathered data patterns from all these houses will be more or less similar.

Examples of similar patterns could be, the pattern of a bedroom or the pattern of a bathroom, or the pattern of a kitchen.

Of course, it might not be, that all these houses and habitants share common patterns, so maybe the best thing to do would be to make some sort of cluster analysis on the data sets and try to settle on a few general house patterns.

8.2.3 Using another RNN model

Elman and Jordan type RNNs, are not the only types of RNN. Some of the new members to the RNN family are, for example Long Short Term Memory (LSTM)[5] [6] and Echo State Network (ESN)[7].

There seems to be particularly much fuss about and research with LSTM, and there are claims that it is possible to handle very long time patterns with this model, so it might be worth trying out.

[5] http://en.wikipedia.org/wiki/Long_short_term_memory

[6] http://www.idsia.ch/~juergen/rnn.html

[7] http://en.wikipedia.org/wiki/Echo_state_network

Chapter 9

Discussion

Some obvious criticism for the current implementation, is that it is very static, in the sense that one has to build up some training data, train the model with this data and then attempt to evaluate the model by testing it live in the simulator.

There is no feedback to the system, such that it would be possible to retrain the model automatically.

I have only implemented a bare minimum for calculating the recommendation. And there is no fall back mechanisms when the recommendations are wrong.

However, I would argue that what I have implemented, is just enough to give an idea about how plausible it is to use ANN techniques in a real life context for the problem at hand.

9.1 Bad habits

A very serious limitation for the current idea about the implementation of the system, is that it relies on the habitants generating some data for an initial period, before the system effectively can be taken into use. The weakness of this idea, is the under lying assumption that the habitants correctly will turn off the light when they leave a room.

When the ANN is trained with the data generated by the habitants, it will adopt any bad habits of not turning off the light. However, this flaw might be eliminated through time if the system, when it has been active for a while, is retrained with new data.

9.2 Limited tests

A very limited amount of tests has been performed, and only within the simulator. Furthermore, it has been troublesome to test large multi actor scenarios in the simulator, thus only small scale scenarios with one or two actors has been

conducted.

The tested data sequences has also been relative short, so it is unknown if the derived model will be able to handle predictions well when the sequences get very long.

However, I do think that the model developed so far, seems to display some potential for usage in a real life setting. But it is necessary with some reliable results from a real life implementation before the final conclusion can be drawn about using ANN techniques for managing the lighting control in a house.
Getting rid of this uncertainty is very important.

9.3 Main drawback of using ANN techniques

I think that, the main drawback of using ANN techniques is that there really exists no rules for how exactly to use them. It is for example necessary to conduct experiments in order to best determine how to configure the hidden layers.
However, some of the configuration has already been eliminated, simply by using the RPROP training method, instead of the standard backpropagation algorithm, where one has to adjust both learning rate and momentum.

Through more experiments, it will most likely be possible to settle on some scaling values for configuring the hidden layers relative to numbers of inputs.
I believe that it is necessary to conduct experiments on bigger data sets gathered in a real life setting, before one can settle on some rules of thumb not only for configuring the hidden layer, but also choice of initialization and training methods. This is also the reason why I chose to create an interface for training and exploring trained ANNs.

Chapter 10

Conclusion

I have explored various methods for designing and training neural networks, and have successfully implemented some of them. I have also managed to model the data from the problem in a way, such that an RNN can be trained with it, and perform relative well in the Intelligent House simulator.

And this serves as a proof of concept, that theres a good chance that using ANN techniques is a viable option for controlling the light in houses, based on human behavior patterns.

My efforts in this project has mainly been focused on learning just enough about neural networks to be able to make an usable application of one in the given problem context. Next on gathering data from the simulator and doing proper preprocessing on it. And next on optimizing the training method, to be viable enough to converge relatively fast, with more than just a few inputs and with a bigger data set. And at last the integration with the simulator to prove the usability of the trained neural network.

Thus, there is still much I would like to try out and explore, which is also necessary if this work is to be used in a real life setting.

However, I feel that I have laid out great foundation for continuing the work, and that, with a lot of ideas for what to explore next to take it to the next level.

On a personal level, my goal of learning about ANN techniques in a practical setting has been reached, and I am indeed very satisfied with what I have learned during this period.

Bibliography

[1] Long Chen, Hao Si, and Hongqing Fang. Recurrent neural network for human activity recognition in smart home. *Lecture Notes in Electrical Engineering*, 254:341–348, 2013.

[2] Jeff Heaton. *Introduction to the Math of Neural Networks*. Heaton Research, 2012.

[3] Jeff Heaton and multiple other contributors. Heaton research wiki, 2011.

[4] Sune Keller and Martin Skytte Kristensen. Simulation and visualization of intelligent light control system (bachelor thesis), 2010.

[5] David Emil Lemvigh and Andreas Møller. Machine learning in intelligent buildings, 2012.

Appendix A

Known bugs and problems in the simulator

The first part of the project was focused on upgrading the simulator from Scala 2.7.3 to Scala 2.9.3.

The simulator was developed in the winter of 2010, and thus it is 3 years old, which is a long time in the world of software development. If it was simply used as is, then any new libraries introduced in the system might cause problems. It would also be harder be harder to troubleshoot, as Scala is a fairly young and very fast moving programming language with a lot of development from version to version.

Yet another motivation for performing the upgrade, was that it would be beneficial, not only for myself for future development on my system, but also for other developers who wish to explore the possibilities of creating intelligent control of lighting in houses.

I successfully managed to upgrade the simulator. But when upgrading the simulator, I might have introduced some bugs. Not only due to not having any past experience with Scala, but also because the simulator is relative complex.

However, it might be that some of these bugs were inherited from the old implementation.

I have also, only used the simulator on Linux platforms, so it might be that it behaves slightly different when being used in for example a Windows environment.

A.1 Delay time definitions

When creating scenarios, it is sometimes desired to make the actor in the simulation, wait for a certain amount of time, without making any movements. This is possible in the simulator, but the time definitions currently has to be adjusted by editing the XML file containing the setup for the simulation.

A.2 Problems with multi actor scenarios

When working with multi actor scenarios, I discovered that not all sensors were activated correctly. Sometimes it would seem that they somehow were blocking out each other.

Attempts to tweak various parameters were made, but without success. So I had to live with the, sometimes, missing sensor registrations for multi actor scenarios during development and testing.

A.3 Problem with initial time

I have observed, that the simulator sometimes will use the Unix epoch [1], as starting time.

This will cause trouble, if one is trying to generate some data and then the system suddenly registers some timestamps which are very different from the first registered timestamps, because it switched from current time to Unix time epoch, or vice versa.

My implementation of the AI, might also make the simulator crash when this happens, because it relies on the timestamps being durations from a uniform point in time.

I have observed that the "Interactive simulator" pane always seems to display the correct time, and that this can be exploited to ensure correct time when generating data or testing with the "Scenario simulator". If one opens the "Interactive simulator" pane, before loading a setup, then it seems to use the correct time in the "Scenario simulator" as well.

A.4 Bugs when creating scenarios or drawing patterns

Multiple, and possible related, bugs has been observed when drawing patterns for scenarios and drawing patterns in the interactive simulator mode.

If one delete a node in a pattern it might cause weird behavior when attempting to append the pattern. And therefore, I recommend to delete the given pattern and start over instead, until this has been fixed.

Furthermore, I have observed weird "extra" trails, when creating patterns in the interactive simulator. However, this bug is just visual, and has no influence on the given simulations.

[1] `http://en.wikipedia.org/wiki/Unix_time`

A.5 Time speed up and slow downs during simulation

Sometimes the simulator seems to speed up time or slow down the visual part of the simulator.

The exact reason is unclear to me, but it seems to happen more often when the simulator has been running for a long time or when the computer has been under a lot of stress.

A simple restart of the simulator should suffice to fix this, until the bug has been properly investigated and fixed.

Appendix B

Simulator test results

Here are some registered tests performed with the simulator.

The tests has been performed by observing if a light was turned off too early and by using stop watches to check if a given light is turned off too late, where 'too late' is defined as more than 5 seconds.

If a light is turned off too early, it is registered with an x, and if one is turned off too late the time is noted down.

There can be multiple x's and registered times for one test scenario.

All the tests has been performed with a 1000ms recommendation delay.

B.1 Results from 'simple_tests.xml'

Following tests where performed on a model trained to a RMS value of 0.14796 based on the scenarios in the *simple_tests.xml* file.

For testing the 'AI test X' scenarios has been used. Of these tests number 1 and 5 are scenarios with multiple actors.

Test number	AI test	early	late	success
1	5		8	no
2	4	x		no
3	3			yes
4	2			yes
5	1	x	12	no
6	6	x		no
7	1		13	no
8	2	x		no
9	3			yes
10	4	x		no
11	5	xx		no
12	6		11	no
13	2			yes
14	4		9	no
15	3			yes
16	1	xx	6	no
17	5		6,7	no
18	6			yes

6 pure successes and 17 failures.

Following tests were performed with a model trained to a RMS value of 0.01309

Test number	AI test	early	late	success
1	4			yes
2	2		30	no
3	5			yes
4	2	x	12	no
5	3			yes
6	6			yes
7	6			yes
8	5	x		no
9	3			yes
10	1			yes
11	2			yes
12	4			yes
13	1	x	10	no
14	2			yes
15	3			yes
16	4			yes
17	5			yes
18	6			yes

14 pure successes, 6 failures.

Appendix C

Guide on how to make simulations, train and test RNN

This guide will explain how to run the project, however, you might need to install some modules first.

All development and testing has been performed on a Linux Debian platform, however there is a good chance that it will run on other platforms as well.

C.1 Installation of Python 3 and Python modules

The Python project uses Python version 3, and cannot be executed with earlier versions of Python. In case you aren't using Linux with a package manager, you can get instructions on how to install Python 3 from following link: `http://python.org/download/releases/3.0/`

Following modules has been used in python:

- numpy version 1.8
- toolz version 0.4 or higher
- Django version 1.6.1
- matplotlib version 1.3.1

For installing the necessary modules in python, I recommend to use pip.

Pip is an easy to use tool for installing and managing packages and modules for Python. Pip can either be installed with a package manager in Linux as 'python3-pip' or from `https://pypi.python.org/pypi/pip`. In windows it can be installed with binaries from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pip`.

Modules can be installed with pip by typing:

$ pip install XXXX
where XXXX is the name of a module. (The module names are case sensitive)

Furthermore, you can search for modules by typing:

$ pip search XXXX

If pip fails to install a module in Linux, the easiest to do is to attempt to install the given module with the Linux package manager instead as this will solve any missing dependencies which might be the problem. However, often pip will tell which dependencies are missing, so it is also possible to install these packages with the package manager and then try to use pip again.

If pip fails to install a module in windows you can install the given module from `http://www.lfd.uci.edu/~gohlke/pythonlibs/` instead.

C.2 Running the Python project

To run the python project, open a terminal or console and go to the root location of the python project. In this directory there will be a python source code file called 'manage.py', this is the file that starts the python project.

To start the project type:

$ python3 manage.py runserver 8000

This will start the Django service with a localhost IP address and use port 8000 for the service. Now you should be able to open a browser, go to the site `http://127.0.0.1:8000/` and see the project web GUI.

C.3 The web GUI

The web GUI, uses JavaScript and HTML5 elements, so you will have to activate JavaScript in your browser if it is disabled.

In the web GUI it is possible to control whether or not the AI should be active in the simulator. Simply click the button at the top of the site to change its status. Note that to record data from scenarios in the simulator, you need to disable the AI.

If you have recorded data from the simulator and saved it to an XML, then you can upload it to the service which will generate entries in its database based on the recorded data in the XML file. This is done in the section "Generate training data set" of the site.

The next section on the site is "Adjust simulator parameters". In this section you can choose which data set to use for training new ANNs. Choosing an entry in this list, will also update the list of the associated, already trained, ANN models. In this section you can also adjust the recommendation delay used in the simulator.

The last section on the page is the "Experiments" section. In this section, it is possible to train a new ANN model with various parameters.

Furthermore, it is in this section that the existing previous experiments are listed. In this list you can make various visualizations of the individual experiments or visualize their error plots together. Furthermore you can resume training of a given model using the 'plots' menu for a given experiment.

A last thing to mention about this list, is that this is where you choose which model to use in the simulator.

For the project hand in, I have provided a database file in which there already exists performed experiments and generated data sets.

- data set with id 13, can be used with the *one_light.xml* setup in the simulator.
- data set with id 17, can be used with the *simple_tests.xml* setup in the simulator.
- data set with id 18, can be used with the *labs.xml* setup in the simulator.

Note that if you change model or parameters related to the simulator in the web GUI, you will need to restart the simulator because the simulator only loads these values when it is started.

C.4 The simulator

Before using the simulator, I recommend that you read the appendix about "Known bugs and problems in the simulator", to be aware of the existing bugs and limitations.

Before starting the simulator, you need to have the Python project running, if it isn't running then follow the explanation in section "Running the Python project" above. It is necessary to have the python project running because the AI implementation in the simulator loads configuration and weights through the service which the Python project provides.

The simulator can be started from a terminal or console, by first moving to the root directory of the simulator project, which is called sim_fresh. After this you can run the jar file for the simulator by typing:

$ java -jar sim_fresh.jar

This will open the simulator in the "Home Builder" view where one can create a setup from scratch. However, I have provided some setups which you can load and use in the simulator.

Existing setups are contained in the following XML files:

- *one_light.xml*
- *simple_tests.xml*
- *labs.xml*

To load a setup, I recommend that you first move to the "interactive simulator" pane, to avoid a bug that causes the simulator to use the Unix Epoch as starting time. When located at this pane, you can simply load a setup from the file menu where you choose "Open setup" and locate the given setup file which you which to load.

When testing an ANN model, it is the "scenario simulator" and the "interactive simulator" panes that are of interest.

In the interactive pane, you can simply click on the floor plan to create patterns, and then you can start the simulation by clicking the play button at the right.

In the scenario simulator pane, you can choose from existing configured scenarios in the scenarios list at the bottom left. Note that when testing an ANN model, it doesn't make sense to use scenarios where there is interaction with the light switches.

My convention for scenarios has been to use "scenarios XX" for recording data, and the testing scenarios I have named "AI test X"

If you which to create your own scenario, you can do so under the "scenario editor" pane. In this pane you can add a new scenario with the add button located near the top right of the GUI, furthermore you can add paths to the given scenario, with the add button located at the bottom of the "paths" list.

When creating paths in the scenario editor, you can insert a delay by using CTRL-leftclick. Note that it is not possible to adjust the delay time itself. To

do this you have to edit the XML file in which the setup is contained. However, this is fairly straight forward, simply search for "delay" in this file and you will find the attributes to edit. When you have edited a file outside the simulator, you can reload it by reopening it from the simulator.

If you wish to create a data set from scratch to train a new model with, you can either create a new setup file from scratch, or you can empty the AI log in an existing setup file.

All the recorded data from the simulator are located in an element called "ai-log". Simply deleting everything contained in this element, will empty the recorded data, and after this the setup can be reloaded by reopening the setup file.

Appendix D

Guide to the source code

For the Python part of the project, I have utilized the Pycharm IDE[1], which I recommend viewing and browsing the source code.

For upgrading and integrating the AI into the simulator, I have used the IntelliJ IDE[2]. I recommend using this IDE for browsing and reading the source code for the simulator.

There are a lot of files in the project of which many are auto generated, or slightly irrelevant. And because of this, I will briefly sketch out the important files with their relative paths.

The important source code files in the Python project are:

- /ANN_Explorer/Intelligent_House/ANN.py (this file is where the training and generation of the ANN model is performed. It is described in more below in the subsection "The training of the model")
- /ANN_Explorer/Intelligent_House/db_utils.py (this file contains functions for saving experiments and data sets to the database, functions for retrieving data from the database and functions for fetching and visualizing data from the database)
- /ANN_Explorer/Intelligent_House/ExtractDataFromXML.py (this file contains functionality for extracting data from the XML files generated by the simulator, and performing normalization on it as described in the "Data" chapter)
- /ANN_Explorer/Intelligent_House/views.py (this file contains the functionality used in the Django web GUI and web service)

[1] http://www.jetbrains.com/pycharm/ I have used the professional edition, but there is also a free edition

[2] http://www.jetbrains.com/idea/ I have used the free community edition

- /ANN_Explorer/templates/main.html (this is the Django view template file for generating the web GUI)

The files for which I am responsible in the simulator project, are:

- /sim_fresh/src/ai/impl/SimpleAi.scala (This file was inherited from the old simulator, and originally served as an example of what it was possible to do with an AI implementation in the simulator)
- /sim_fresh/src/ai/ann/ANN.scala (Together with ANN_Functions.scala, this file contains the minimum necessities for setting up a trained ANN model and calculate outputs with it)
- /sim_fresh/src/ai/ann/ANN_Functions.scala