FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Neuronale Netze für Smart Lighting

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Simon Strassl
Matrikelnummer 1326936

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof.Dr. Wolfgang Kastner

Wien, 6. September 2016

Simon Strassl	Wolfgang Kastner

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Artificial Neural Networks for Smart Lighting

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Simon Strassl
Registration Number 1326936

to the Faculty of Informatics
at the TU Wien
Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner

Vienna, 6th September, 2016

Simon Strassl

Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Simon Strassl
Quellenstrasse 128/34, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. September 2016

Simon Strassl

Kurzfassung

Aufgrund der steigenden Verfügbarkeit und sinkenden Kosten von Sensoren und Microcontrollern können heutzutage immer mehr Aufgaben in unseren Häusern automatisiert werden. Eine dieser Aufgaben ist das automatische Steuern der Beleuchtung, was einerseits zu einer Steigerung der Lebensqualität und des Komforts der Bewohner führt und andererseits Energieersparnisse ermöglicht. In dieser Arbeit sollen verschiedene Typen von künstlichen neuronalen Netzen bezüglich ihrer Eignung zur automatischen Steuerung von Licht evaluiert werden. Ein besonderer Fokus liegt hierbei auf rekurrenten neuronalen Netze und der Fähigkeit der neuronalen Netze Information aus den Daten von einfachen und kostengünstigen Sensoren zu gewinnen. Hierzu werden an simulierten Daten mehrere Variationen des Problems, darunter der Umgang mit fehlerhaften Sensoren und die Vorhersage zukünftiger Ereignisse untersucht. Des weiteren wird die Eignung auch anhand von real gesammelten Daten evaluiert.

Abstract

The widespread availability of cheap sensors and microcontrollers, enables automation of more and more tasks in our homes. One such problem is deciding when to turn the lights in a house on and off, for which automation promises an increase in comfort for the inhabitants and energy savings. This thesis aims to evaluate artificial networks with regard to their suitability for automatic light control, with a strong focus on recurrent neural networks and the ability of neural networks to extract information from simple and cost-efficient sensors. Several variations of the problem, including fault tolerance and prediction of future events, are explored on simulated data. Finally, the neural networks are also applied to real world data.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The idea of the smart home certainly is not a recent one, going back to the mid 1940s [Moz98], but as computing power and sensors/actuators become cheaper, more widely available and integrated into various aspects of our lives, widespread adoption of smart homes moves further into the realm of reality. Already more and more households are being outfit with at least some kind of home automation system. These systems come in various shapes, sizes and levels of functionality — from the relatively cheap and easy to install Nest thermostat [Nes], over the more extensive, but still targeted at private users, Insteon [Ins], Adhoco [Adh], which is primarily targeted at large companies with promises of saving heating costs, to open standards such as KNX [Ass] and their various implementations.

1.2 Problem statement

Despite the increasing availability of home automation solutions, the current products are still far from the ideal of the truly smart home. While most manufacturers do include some kind of learning component in their systems, they are usually severely limited in their flexibility and only able to learn/affect a small part of the entire system and thus far from the ideal smart home, which would be able to recognize the needs of its inhabitants and act according to them.

Examples of such actions are:

- Turning on the lights when people are in a room and there is not enough natural light

- Decreasing the heating when nobody is at home, but raising it back to comfortable levels *before* the inhabitants return
- Opening/closing the blinds in the morning/evening
- Starting various appliances (e.g. brewing coffee in the morning)
- Alerting the owner of intruders

Obviously, considering a wide range of different inhabitants and their requirements demands a great amount of flexibility, which is why some kind of machine learning (ML) method seems to be almost obligatory to achieve this dream, especially if the system needs to be able to adapt to changing conditions. These adjustments may become necessary because of changes in lifestyle (e.g. suddenly working late at night), different inhabitants (e.g. a child moving out) or a multitude of other reasons. Preferably, this adjustment should be as smooth for the inhabitants as possible, with them doing the minimum amount of work possible to achieve the desired results.

In particular, automatic light control can free the users from having to manually switch the lights off and on. Aside from eliminating some inconveniences from the residents life, it also reduces the amount of energy (and thus also money) spent on keeping lights running because the inhabitants can no longer forget to turn off the lights.

1.3 Aim of this work

The aim of this work is to evaluate various types of neural networks in their suitability for use in home automation, with a special focus on recurrent neural networks and automatic light control. For this, various experiments are conducted (on simulated as well as on real world data) and the performances of the different kinds of neural networks are analyzed.

1.4 Structure

Chapter 2 provides a description of the different concepts, onto which the other chapters build. It is directly followed by an overview of the state of the art of neural networks (and machine learning in general) and their use in home automation, as well as alternative approaches used in home automation in Chapter 3. In the main part in Chapter 4, the conducted experiments, their setup and their results are presented and analyzed. Finally, Chapter 5 contains a summary of the experiments' results and a reflection, as well as an outlook.

CHAPTER 2

Methodology

2.1 Artificial Neural Networks

An artificial neural network (ANN) is a type of machine learning model inspired by the structure of the human brain. At its core, it is a directed graph with weighted edges, where each node corresponds to a single neuron, which applies a function (referred to as the activation function) to the weighted sum of its inputs and thus produces the output, which again may be used as input for other neurons and so on [Hay07].

A visualization of the neuron can be seen in Figure 2.1, where $x_1 \dots x_m$ denote the input of the neuron, $w_{k1} \dots w_{km}$ the corresponding weights, b_k the bias, $f(.)$ the activation function and y_k the output of the neuron k. The bias can be thought of as a constant input (i.e. it does not depend on the real inputs) and can be modeled as an additional input $x_0 = 1$ with $w_{k0} = b_k$. Popular choices for the activation function include the hyperbolic tangent (TanH), rectified linear unit (ReLU) [NH10] and the sigmoid function, but in theory every function can be used. However, linear activation functions should almost always be avoided, since they would result in the network simply describing one larger linear function which eliminates the benefits of using a neural network. The reason for this can easily be deduced when substituting the inputs with their respective functions in the formal description of a neuron in Equation 2.1.

$$y_k = f(\sum_{i=0}^{m} w_{ki} x_i) \tag{2.1}$$

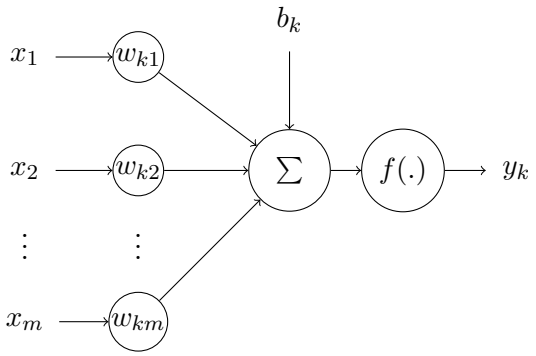


Figure 2.1: Structure of an artificial neuron

2.1.1 Training

Typically ANNs are trained by adjusting the weights of their connections via some training algorithm, with the goal of minimizing the difference of the ANN's output to the output of the real function. While various ways of training are possible, for this thesis only the simplest case is relevant: Supervised (i.e. the correct outputs are known) learning using backpropagation (BP). Backpropagation works by taking the loss (calculated by a given loss function) and propagating it backwards through the network (and adjusting the weights along the way), guided by some kind of optimizer (e.g stochastic gradient descent (SGD)). This process is repeated for every sample in the training data set. One such full pass through the training data is referred to as an *epoch*. In practice, this is done using parallel matrix calculations, usually performed on a graphics processing unit (GPU) to speed up the training process, which can often be quite slow depending on the size of the data set and the complexity of the network.

A popular, general purpose, choice for the loss function is mean square error (MSE) as defined in Equation 2.2, where act_i denotes the actual value of output i (as calculated by the ANN) and exp_i denotes the expected value of the output i.

$$Loss = \frac{1}{n}\sum_{i=0}^{n}(exp_i - act_i)^2 \tag{2.2}$$

2.1.2 Types

The various kinds of ANN architectures are numerous and diverse, with different architectures being suited for different purposes. This section aims to give a high level overview of the types of networks used in this thesis and their strengths and weaknesses.

Feed forward neural networks

One of the simplest, most common and most popular ANN architectures are feed forward neural networks (FFNNs), which simply consist of an input layer, an arbitrary amount of hidden layers between input and output and an output layer (see Figure 2.2 for an example). Each layer takes the inputs from the former layer, transforms them and passes the result to the following layer. The name stems from the fact, that information simply flows through the network in one direction (from inputs to outputs). A FFNN with at least one hidden layer is called a multilayer perceptron (MLP).

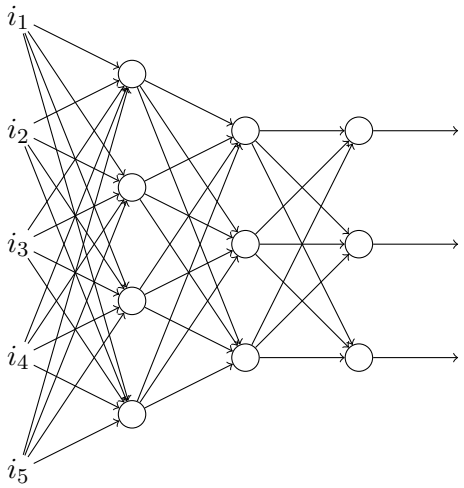


Figure 2.2: Fully connected FFNN with 2 hidden layers

Recurrent neural networks

Whereas FFNNs have a simple forward flow of information, recurrent neural networks (RNNs) contain backwards connections which create a feedback loop, allowing them to maintain an internal state of sorts [Hay07]. This memory makes them uniquely suited for tasks involving sequences because it allows them to refer to previous information at (theoretically) arbitrary points in time. In practice, RNNs are usually not created with arbitrary connections between neurons, but again organized in layers which only have recurrent connections to themselves. The result of this process is referred to as a recurrent multilayer perceptron (RMLP) as seen in Figure 2.3. Training is most commonly done using backpropagation through time (BPTT) [Wer90], where the network is unfolded through time (effectively creating something akin to a very deep FFNN) and propagating the error through the unfolded network. In practice, truncated BPTT, where the network is only unrolled a fixed number of steps, is usually used to improve performance.

From this point on, RNNs without any further enhancement (see below) will be referred to as a plain recurrent neural network (PRNN), while RNN will be used for the entire

class of recurrent ANNs.

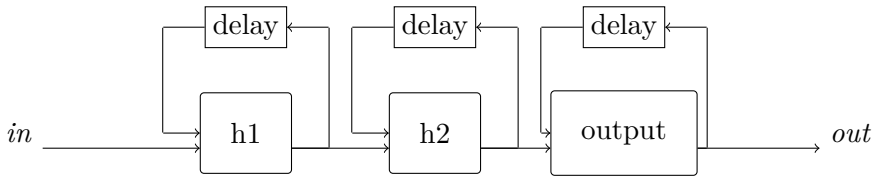


Figure 2.3: RMLP with two hidden layers

Long short-term memory neural networks

PRNNs (just as very deep FFNNs) suffer from the vanishing/exploding gradient problem, where the error either disappears or explodes when propagated too far back [GSC00]. This is especially problematic for PRNNs because the error has to be propagated back through every layer for every step in time. Consequently, this leads to PRNNs having problems with long term dependencies.

Long short-term memory neural networks (LSTMs), introduced by [HS97] prevent the vanishing gradient problem, by creating memory blocks which maintain their own internal state by means of a constant error carrousel (CEC) and control access to it via input and output gates that can be opened or closed depending on the input. Common alterations applied to the standard LSTM block are the addition of a forget gate [GSC00], which is able to clear the internal state, or peephole connections [GSS02] which allow the gates to access the internal state even when the output gate is closed.

An illustration of a LSTM block can be seen in Figure 2.4, which contains a forget gate as well as peephole connections. x_t denotes the input at time t and h_t the output at the same point in time.

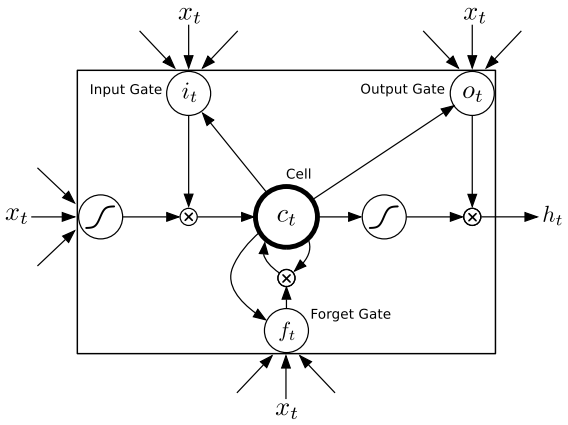


Figure 2.4: LSTM block from [GJM13]

2.2 Performance measures

There exist a multitude of ways to measure the performance of ML models, but since all problems in this work are all (multi-label) binary classification problems, only accuracy, sensitivity, specificity and the Matthews correlation coefficient (MCC) are used. These directly follow from the binary confusion matrix, where TP denotes the number of true positives, FP the number of false positives, TN the number of true negatives and FN the number of false negatives.

While the definitions given below are only applicable to single-label classification problems, they can be generalized to allow their use for multi-label classification problems [TK06], where multiple labels may be assigned to each sample. For a multi-label classification problem with n labels, one can split it into n single-label classification problems, calculating the required metric separately for each of these problems and then averaging the results.

2.2.1 Accuracy

The accuracy describes the ratio of correct predictions to total predictions, or, in other words, the probability that the model will correctly classify a given sample. The formula is given in Equation 2.3.

$$Ac = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.3}$$

2.2.2 Sensitivity

The sensitivity describes the ratio of true positives to total positives, i.e. how many (proportionally) of the positive samples were correctly classified as positive. The formula is given in Equation 2.4.

$$Se = \frac{TP}{TP + FN} \tag{2.4}$$

2.2.3 Specificity

Analogous to the sensitivity, the specificity describes the ratio of true negatives to total negatives, i.e. how many (proportionally) of the negative samples were correctly classified as negative. The formula is given in Equation 2.5.

$$Sp = \frac{TN}{TN + FP} \tag{2.5}$$

2.2.4 Matthews correlation coefficient

The MCC [Mat75; Pow11] is a measure for the correlation between predicted and actual binary classifications and as such a more balanced measure for the quality of the prediction [Bal+00]. The MCC ranges from $+1$ to -1, with $+1$ indicating a perfect correlation between actual and expected outputs (i.e all classifications were correct) and -1 indicating a perfect negative correlation (i.e no classification was correct). The definition of the MCC is given in Equation 2.6.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{2.6}$$

CHAPTER 3

State of the art

Recently ANNs have experienced a surge in popularity, mostly owed to the meteoric rise of deep learning [LBH15] in the last few years. This was made possible by the steady increase in computing power, which enabled the training of significantly larger and more complex networks on affordable hardware (GPUs are a popular choice).

Despite (or maybe because of) their simplicity, FFNNs have been successfully applied to a multitude of diverse problems and have proven to be a quite versatile tool. These problems include fraud detection [FC98; GR94], image recognition [KSH12] and many more. Unfortunately due to their rather non-sequential nature, they have problems when faced with sequential problems and require workarounds to access past information. Such workarounds include adding past data as explicit inputs to the network or pre-processing sequences down to a few key figures.

RNNs (and LSTMs in particular) on the other hand are built with sequential input in mind and have proven to be quite capable of solving various kinds of sequential problems, including (but not limited to) text generation [Gra13], speech recognition [GSS02] and even music generation [ES02].

Of course the idea of using ANNs for home automation is not a new one, having already been suggested and implemented in a real house in 1998 [Moz98], where an ANN was used to control light and heating. They have been used to classify [CDC10] and detect [RCH06; ZWB08] activities of daily living (ADLs). Rivera-Illingworth, Callaghan, and Hagras [RCH06] in particular use a RNN to capture the temporal dependencies for ADL detection and obtain very promising results on real data.

Automatic light control has been studied by [MM08], who use a traditional FFNN combined with an algorithm that decides whether to apply the decision of the network based on the likelihood that the decision is correct, which is determined by how well the network performed at this minute of the day in the past. They obtain quite promising results, but much of their setup is designed to correct for possible errors committed by

the neural network, which is of course less than optimal (ideally the network itself would give an indicator of its confidence in a decision).

[DH14] compare ML methods including a FFNN in regards to their suitability for light control on the publicly available CASAS [Coo+13] data sets and find that the FFNN performs well, but still worse than other methods. This experiment will be further explored in Section 4.3.

CHAPTER 4

Experiments

4.1 Setup

The training was done using backpropagation/backpropagation through time, with MSE (see Equation 2.2) as the loss function and *Adam* [KB14], with the parameters suggested by the authors, as the optimizer. For all ANNs, TanH was chosen as the activation function.

Accuracy, sensitivity, specificity and MCC are calculated by first transforming each output into a binary classification by applying a simple threshold function to it. Given an output x between 0 and 1, $x \geq 0.5$ indicates the sample is in the class, $x < 0.5$ indicates it isn't. Since the problems are multi-label classification problems, the metric is then calculated using the method given at the beginning of Section 2.2.

The following models are used in the experiments (refer to Section 2.1.2 for descriptions of the different architectures):

- A baseline established by predicting constant 0. Since lights tend to be off far longer than they are on, this establishes a baseline as to how we would do if we just never turned on a light.

  Notation: *CONST*

- A small FFNN with one fully connected hidden layer (20 nodes).

  Notation: *FF-h20*

- A medium FFNN with two fully connected hidden layers (20 and 40 nodes).

  Notation: *FF-h20-h40*

- A large FFNN with two fully connected hidden layers (200 and 400 nodes).

  Notation: *FF-h200-h400*

- A PRNN with one hidden layer (20 nodes) and a sequence length of 40. The sequence length denotes how many steps in time the network is unrolled and thus limits how far back the error is propagated and how much of the past is accessible to the network when making predictions.

  Notation: *PRNN-s40-h20*

- A LSTM with one hidden layer (20 nodes) and a sequence length of 40. The LSTM includes forget gates, but no peephole connections.

  Notation: *LSTM-s40-h20*

4.1.1 Implementation

All experiments were performed using the machine learning libraries Lasagne [Die+15] (primarily) and Keras [Cho15]. Lasagne is built on the numeric computation library Theano [The16] and provides convenience functions and implementations of neural networks, while still exposing the Theano internals. Theano works on the basis of building a computation graph, which acts as an abstraction of the actual computations that can later be executed on either the CPU or GPU. In practice the GPU is preferable, because the faster floating point and matrix operations as well as the high potential for parallelism significantly decrease the time required to train the network.

Listing 1 contains the code to create a FFNN with 2 hidden layers (200 and 400 neurons) with TanH as the activation function using Lasagne. The `make_ffnn` function takes the input and output dimension as its arguments and returns an input and output layer of the required dimensions, which can later be used to train the network and make predictions. The `DenseLayers` are traditional fully connected layers with the activation function specified by the `nonlinearity` parameter. The biases are initialized to 0 by default.

Listing 2 follows the same pattern to create a LSTM with 20 hidden nodes and sequence length of 40. A PRNN can be created the same way by merely exchanging the `LSTMLayer` with a `RNNLayer`. The various `*_params` define the initialization values of the LSTM's gates and are thus not relevant when creating a PRNN. The bias of the forget gate is set to 1, as suggested in [JZS15], the other biases are initially set to 0.
Particular attention should be paid to the different input shapes: For the FFNN it is simply (`samples` × `inputs`), while for the LSTM it is (`samples` × `seq_length` × `inputs`), since a single prediction depends not only on the current inputs, but also on the history leading up to it. Thus the input data must be pre-processed by unrolling the sequences (see Figure 4.1 for an example) before it can be used to train the LSTM. The unrolling can be thought of as moving a window (the size of which is equal to the sequence length) over the data and capturing the contents of the window at each step. By default the output of the network will also be a sequence of the same length, containing the predictions for each

Listing 1 Lasagne FFNN

```
1  import lasagne as L
2  from lasagne.nonlinearities import tanh
3  from lasagne.layers import InputLayer, DenseLayer
4
5  H1 = 200
6  H2 = 400
7
8  def make_ffnn(num_inputs, num_outputs):
9      l_in = InputLayer(shape=(None, num_inputs))
10     l_h1 = DenseLayer(l_in, num_units=H1, nonlinearity=tanh)
11     l_h2 = DenseLayer(l_h1, num_units=H2, nonlinearity=tanh)
12     l_out = DenseLayer(l_h2, num_units=num_outputs,
13                        nonlinearity=tanh)
14
15     return l_in, l_out
```

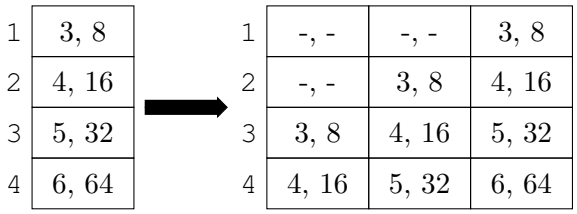


Figure 4.1: Sequence unrolling (4 samples, 2 inputs, sequence length 3)

step (i.e. the output shape is (`samples` × `seq_length` × `outputs`)). This behavior is explicitly disabled by setting `only_return_final` in line 30, which produces the same output shape as for the FFNN: (`samples` × `outputs`).

Listing 3 shows how the networks are trained. First, the final computation graph is created by defining a variable for the expected outputs (line 13), defining the loss function (line 15), defining the function that will update our networks weights (line 17) and finally compiling the computation graph into callable functions for training (line 18), calculating the loss (line 20) and predicting (line 22). At this point Theano transforms the abstract computation graph into the real code to be executed on the central processing unit (CPU) or GPU and also applies optimizations. After this, it is only a matter of calling the compiled `train` function with the training data and the expected outputs (line 25), which internally generates a prediction, calculates the loss and updates the parameters of the network. This is repeated for as many epochs as specified (40 in this case), after each of which the validation loss is calculated (line 26) to provide a measure of the training progress. The `predict` function can then be used to generate predictions on new data (even though in this example it is only applied to the validation data).

Listing 2 Lasagne LSTM

```
import lasagne as L
from lasagne.nonlinearities import tanh
from lasagne.layers.recurrent import Gate
from lasagne.layers import InputLayer, LSTMLayer, DenseLayer
from lasagne.init import Orthogonal, Constant

SEQ_LENGTH=40
HIDDEN=20

def make_lstm(num_inputs, num_outputs):
    l_in = InputLayer(shape=(None, SEQ_LENGTH, num_inputs))

    in_gate_params = Gate(W_in=Orthogonal(),
                          W_hid=Orthogonal(),
                          b=Constant(0.))
    forget_gate_params = Gate(W_in=Orthogonal(),
                              W_hid=Orthogonal(),
                              b=Constant(1.))
    out_gate_params = Gate(W_in=Orthogonal(),
                           W_hid=Orthogonal(),
                           b=Constant(0.))
    cell_params = Gate(W_in=Orthogonal(), W_hid=Orthogonal(),
                       W_cell=None, b=Constant(0.),
                       nonlinearity=tanh)
    l_lstm = LSTMLayer(l_in, num_units=HIDDEN,
                       ingate=in_gate_params,
                       forgetgate=forget_gate_params,
                       cell=cell_params,
                       outgate=out_gate_params,
                       only_return_final=True)

    l_out = DenseLayer(l_lstm, num_units=num_outputs,
                       nonlinearity=tanh)

    return l_in, l_out
```

Listing 3 Lasagne training

```
import lasagne as L
import theano
import theano.tensor as T
from lasagne.objectives import aggregate, squared_error
from lasagne.layers import get_all_params, get_output
from lasagne.updates import adam

EPOCHS = 40

def train_model(l_in, l_out,
                train_data_in, train_data_exp_out,
                val_data_in, val_data_exp_out):
    target = T.matrix()
    pred = get_output(l_out)
    loss = aggregate(squared_error(pred, target), mode="mean")
    params = get_all_params(l_out)
    updates = adam(loss, params)
    train = theano.function([l_in.input_var, target], loss,
                            updates=updates)
    compute_loss = theano.function([l_in.input_var, target_values],
                                   loss)
    predict = theano.function([l_in.input_var], pred)

    for epoch in range(0, EPOCHS):
        train(train_data_in, train_data_exp_out)
        val_loss = compute_loss(val_data_in, val_data_exp_out)
        print("({}) Val. loss: {:.4f}".format(epoch, val_loss))

    prediction = predict(val_data_in)
```

4.2 Light control in apartments (simulated)

The goal of this experiment is to compare different kinds of ANN architectures with regard to their suitability for automatic light control. The problem of automatic light control is interesting because while it is certainly one of the easier problems (compared to e.g. detecting and classifying activities of daily living), it still poses certain challenges. For one, the most decisions have to be basically instant since the light needs to turn on immediately when an inhabitant enters the room (ideally a bit earlier), while their effects can last for a long time if the inhabitant doesn't leave the room for a while.

The easiest way to implement automatic lights is to simply connect a motion sensor to the light and turn it on when the sensor detects activity. Because traditional passive infrared (PIR) sensors only detect moving people, the light is usually kept on for a constant amount of time after the sensor triggers. This works quite well for areas with lots of movement and short remain times (e.g. hallways), but turns out to be a lot less useful in areas with long remain times and little movement (e.g. living rooms) because one would have to extend the keep-alive delay significantly to prevent the light from turning off all the time.

One possible solution would be to track the number of people in a room at a given time by counting them via ultrasonic (US) or light barriers/tripwires. While this works quite well under ideal conditions, it turns out to be not quite as simple in practice because the system will most likely eventually make mistakes (e.g. when multiple people enter the room at the same time) and there needs to be a way to recover from those mistakes [HIS07]. A more robust implementation [Dan+12; Ye10] would be to use cameras in every room and video analysis to count the number of people in the room (and also possibly their activities). While this method definitely provides the most potential information, it is problematic because many inhabitants might object to having cameras record their activities in their private living spaces.

An alternative route, and the one taken in this paper, is to outfit the living space with various kinds of sensors and infer the required state of the lights from the sensor data. With sensors such as US or PIR this has the benefit of being less intrusive than cameras, while keeping costs rather low. ML techniques seem to be a promising approach here because they are more flexible than strict rule based systems.

4.2.1 Setup

Automated light control is a convenient problem insofar, that its core (lights should turn on when someone is in the room) can rather easily be simulated (compared to e.g. image recognition, where generating good training data is just as hard as the actual classification). Simulating the problem has the benefit that every parameter of the problem can be controlled, which allows us to experiment with different variants of the problem to measure their effects. Of course it also comes at the cost of not being an exact reproduction of reality, which is usually quite a bit messier than any simulation

one can imagine. In particular, the simulated inhabitants in this experiment do not show any kind of pattern in their behavior and movements, but instead merely randomly walk through the apartment, resulting in the models not being able to extract such patterns.

A real apartment was chosen as the basis for the simulation and converted into a machine-friendly form by modeling it as a graph. The floor plans of the lower and upper floor can be found in Figure A.1 and Figure A.2. The corresponding graph with the areas covered by the different sensors can be seen in Figure A.3. Note that the graph contains both floors as well as the stairs between them, but some rooms haven't been outfit with any sensors and are thus only modeled as single nodes sticking out from the graph. The upper floor is located on the right, the lower floor on the left with the stairs being split in the middle to prevent overlapping. Figure A.4 shows the same graph, but with colors for the areas of influence of the various lights. Since the stairs do not have any dedicated lights they are part of the areas of the lights at the top and at the bottom simultaneously. Additionally, every node is assigned a probability distribution which models the agent remaining on this node for more than one timestep (notable examples are the outside, the bedrooms and the dining area).

Algorithm 1 describes the process used to simulate moving agents in the apartment. The agent repeatedly chooses a target in the graph, calculates the path to the target and follows it. After each step, the agent may remain on the node for a number of ticks given by the probability distribution of the node. For each of these ticks (plus one when entering the node) a data point is generated, which describes the current state of the world. A data point consists of a tuple $(\texttt{sensor}_1, \ldots, \texttt{sensors}_m)$ of sensor readings and a tuple $(\texttt{actuator}_1, \ldots, \texttt{actuator}_n)$ of expected actuator (lights) states. Consecutive duplicates are eliminated to prevent insignificant data (e.g multiple hours of inactivity because the agent is outside the apartment) from drowning out the more important events — a transformation that can also trivially be applied in a live environment. The time sensors are ignored in this comparison, since they would always cause two data points to differ.

Samples of the generated data points can be found in Table A.1 (sensor part) and Table A.2 (actuator part).

Used sensors are:

- Time sensors (h,m,s), whose output is normalized from ($[0, 23]$, $[0, 59]$, $[0, 59]$) to $[0, 1]$ respectively.
- US sensors, whose output is normalized to $[0, 1]$, where 0 stands for nobody being in the area covered by the sensor and 1 indicates a person standing directly in front of it with the values in between corresponding to the distance of the object to the sensor. Additionally some slight noise is introduced to the sensor readings.
- PIR sensors, whose output is either 0 or 1, where 1 indicates that motion was detected. After a fixed cooldown time with no movement (again with some noise) the sensor switches back to 0.

Algorithm 1 Simulating activity in the apartment

```
1:  function Run-Simulation(world, count)
2:      result ← Empty-List
3:      path ← Empty-List
4:      time ← 0
5:      current ← Choose-Target(world)
6:      while result.size < count do
7:          if path.size = 0 then
8:              target ← Choose-Target(world)
9:              path ← Find-Path(world, current, target)
10:         current ← Poll(path)
11:         remaintime ← Sample(current.remaindist)
12:         steps ← Max(remaintime, 0) + 1
13:         for i = 0 to steps do
14:             time ← time + 1
15:             data point ← To-data point(world, time, current)
16:             Append(result, data point)
17:         Eliminate-Consecutive-Duplicates(result)
18:     return result
```

Since US sensors only trigger when someone is walking directly through their rather narrow observed area and the PIR sensors do not trigger when the person is motionless, the problem exhibits a rather sequential nature. For example, the US sensor at the bottom of the stairs triggering, then the one in the center of the stairs triggering indicates that the inhabitant is now in the upper third of the stairs, even though none of the current sensor outputs gives any indication of this. This is a rather common situation because we can rarely cover the entire living space with sensors that give us constant readings of the inhabitants position (except maybe for cameras, which have certain privacy issues). Due to this, the RNNs are expected to perform significantly better than the FFNNs, since the latter are only able to see the current sensor state. For all simulated experiments the ANNs were trained over 40 epochs on a training data set of 500 000 data points and validated on an independently generated data set of the same size.

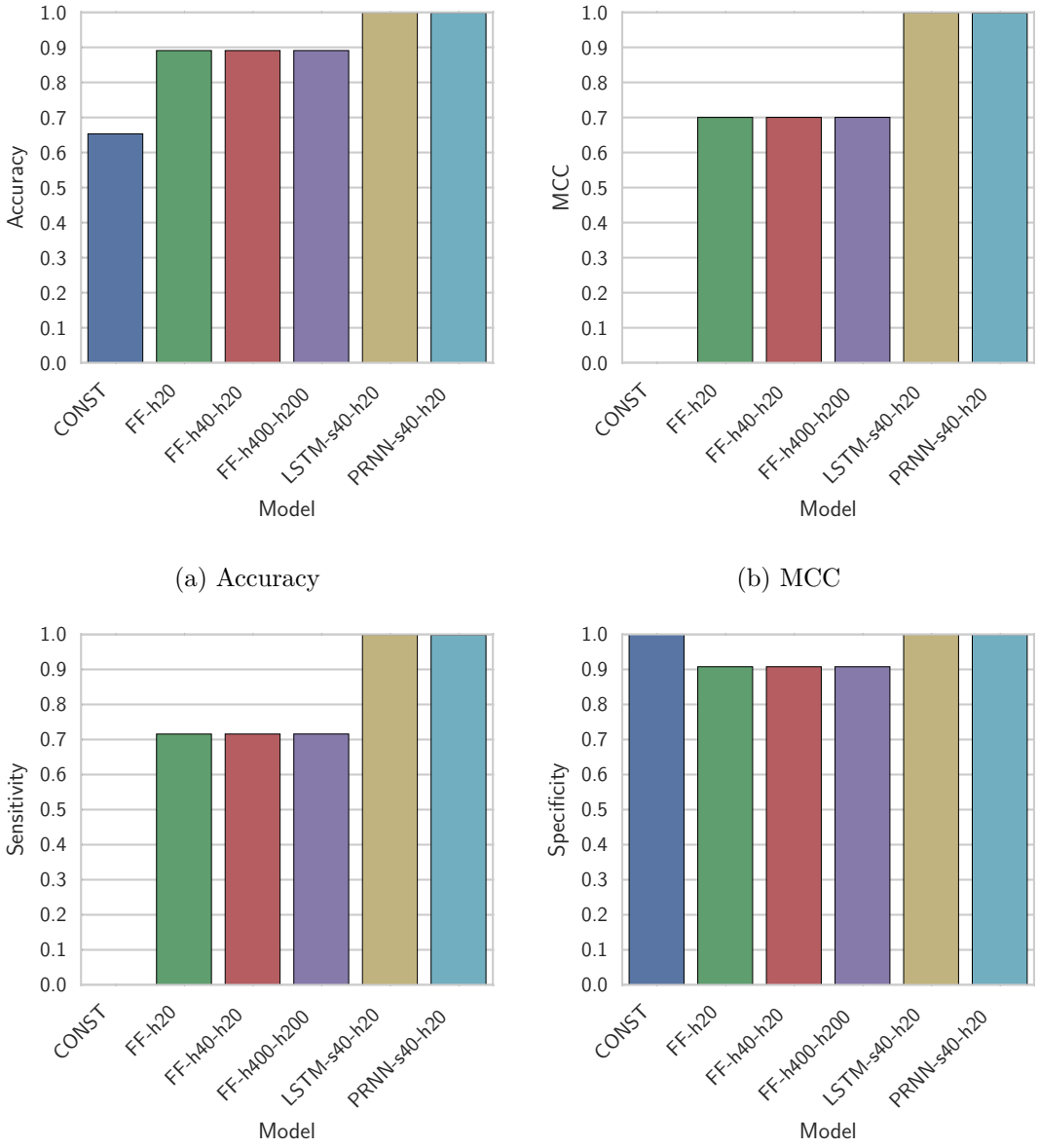
4.2.2 Single inhabitant

The results of the experiment can be seen in Figure 4.2 and Table 4.1 showing the performance on the validation set after 40 epochs of training. As expected due to the rather simple problem, even the FFNNs perform quite well, achieving an accuracy of 89%, with the size of the FFNN not affecting the performance. Nevertheless they are significantly outperformed by the RNNs, which are able to fit the function almost perfectly, with only a negligible difference between the LSTM and PRNN and display a near perfect accuracy of 99.9%. This suggests that the performance displayed by the FFNNs is close to an upper bound for predictions without any access to past information. For all ANNs, the specificity is higher than the sensitivity (although the difference is negligible for RNNs), indicating that it is easier to correctly predict the *OFF* states, than the *ON* states.

The difference between the two kinds of predictions can easily be seen in Figure 4.13 (FFNN) and Figure 4.12 (LSTM), where the FFNNs exhibit heavy spikes depending on the current sensor input, while the RNNs are able to bridge these gaps without sensor input due to their internal memory.

Model	Accuracy	Specificity	Sensitivity	MCC
RAND	0.5003	0.5003	0.5011	0.0000
CONST	0.6531	1.0000	0.0000	0.0000
FF-h20	0.8907	0.9075	0.7155	0.7000
FF-h40-h20	0.8907	0.9075	0.7156	0.7001
FF-h400-h200	0.8908	0.9075	0.7157	0.7001
PRNN-s40-h20	0.9990	0.9995	0.9982	0.9975
LSTM-s40-h20	0.9995	0.9997	0.9993	0.9990

Table 4.1: Validation performance on single inhabitant simulation



(a) Accuracy

(b) MCC

(c) Sensitivity

(d) Specificity

Figure 4.2: Validation performance on plain single inhabitant simulation

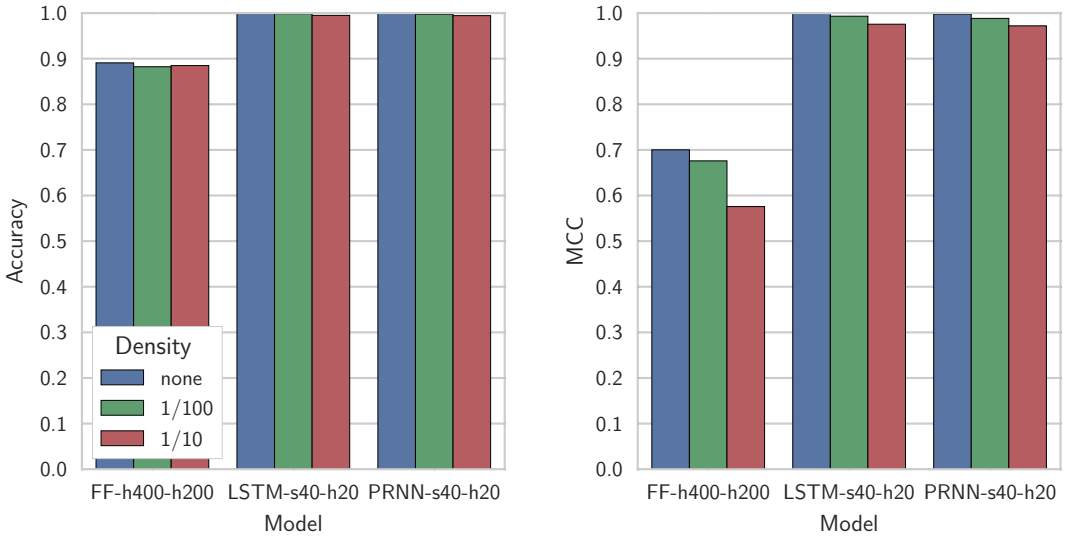
Minimum data point density

One problem with the compaction via duplicate elimination given above is that decisions made by the ANN can potentially last for either a very short (if there is a new event right afterwards) or a very long time (if nothing happens for a while). The second case

is especially problematic because it magnifies errors made at such key decision points, without giving the ANN any way to recover from the error. For example, if the ANN decides not to turn off the lights when the resident leaves the apartment the light might stay on until he returns and triggers new sensor inputs. One way to alleviate this problem is by periodically asking the network to make a new decision, even if nothing happened, effectively enforcing a minimum temporal density of events. The densities included in this comparison are one data point every 10 steps, one every 100, as well as the simulation without any minimum density.

When comparing the accuracy in Figure 4.3a, the changes in density do not seem to have a significant effect on the performance of the models, but comparing the MCC in Figure 4.3b paints a different picture. The RNNs show barely any MCC decrease, while the FFNN shows a significant drop.

The reason for this can be found in the growing amount of *OFF* events as the density increases. This is not surprising, since the increase in density causes long periods of inactivity to generate more data points, which in turn results in more *OFF* data points being generated because the lights are usually off during the times of inactivity (e.g. when nobody is in the apartment). The FFNN simply becomes very conservative in its predictions and only predicts *ON* rarely, while the RNNs are able to bridge these periods with their memory. This is an important results insofar, that it shows that RNNs are able to handle irrelevant sensor inputs polluting their history without a large decrease in performance.



(a) Accuracy

(b) MCC

Figure 4.3: Validation performance of simulations with varying minimum density

Random faults

Sensors are not perfect and sometimes they produce readings that we do not expect. PIR sensors, for example, may be triggered by pets or even just hot air. A smart home should of course be able to filter out or correct these erroneous readings to prevent selection of the wrong actions. To test the capability of the ANNs to cope with bad sensor readings, whenever the output from a sensor is requested it is randomly altered with a probability p. For US sensors this means adding or subtracting a random value from the output, for the PIR sensors it means flipping the real output. The ideal results would be the network realizing that the output is wrong and simply ignoring it. For this experiment, the values of p chosen were 0.001, 0.01 and 0.1, resulting in one of thousand, hundred or ten sensor outputs to be altered. Note that this is the probability that a single sensor output was altered, the probability that a single data point contains a fault is therefore significantly higher with 0.0168, 0.1570 and 0.8332, respectively (given by $1 - (1 - p)^{17}$). The simulation without any faults ($p = 0$) is also shown to establish a best-case scenario.

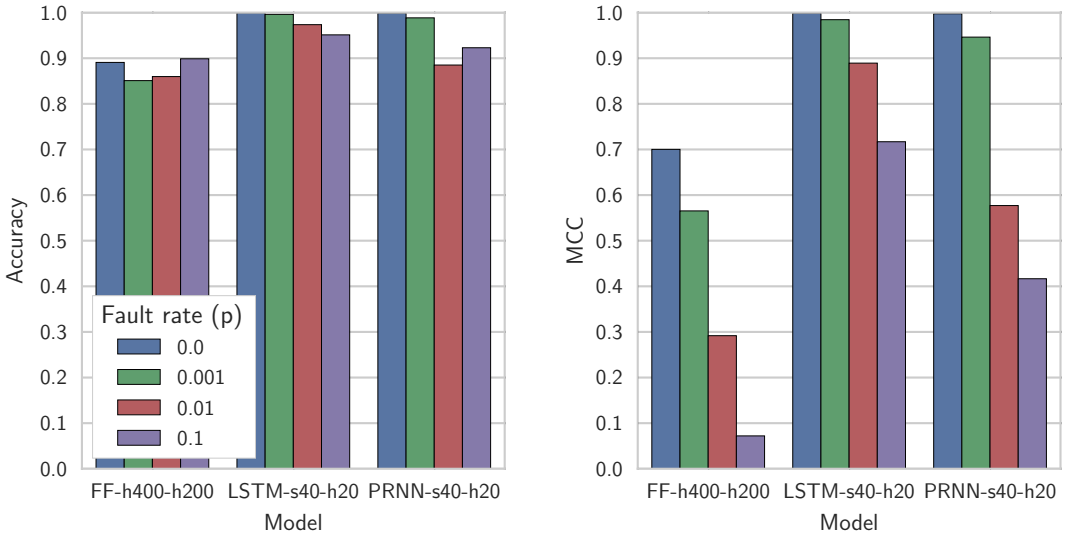
The results can be seen in Figure 4.4. Again the accuracy only decreases slightly, with the FFNN actually showing an increase in accuracy for the highest fault rate. When looking at Figure 4.4b however, all ANNs show a large drop of the MCC, which is far more pronounced in the FFNN than the RNNs. Particularly for $p = 0.1$ the MCC is less than 0.1 for the FFNN while the RNNs are able to handle the faults far better, with the LSTM performing significantly better than the PRNN on $p = 0.01$ (MCC +0.31) and $p = 0.1$ (MCC +0.30).

The reason for the change in the FFNNs behavior is similar as the one given in Section 4.2.2 — the introduction of random faults decreases the effectiveness of the duplicate elimination, again causing the FFNN to revert to more conservative predictions (i.e. more *OFF* predictions). It is likely that the RNNs are able to handle the faults better because they are aware of the context surrounding the event and can thus recognize that it does not fit with the previous events.

4.2.3 Multiple inhabitants

While the experiments in the previous sections assumed a single resident, this is rarely the case in reality. Usually multiple people inhabit the same living space, with (semi)regular visitors who further complicate the problem.

In order to simulate n residents, the process described in Algorithm 1 can be adapted by simply running n simulations in parallel, merging the data points for each tick and only applying the duplicate elimination afterwards. The merge strategy depends on the exact kinds of sensors used, but in this case simply taking the maximum value for each sensor/actuator is sufficient. While this still does not take into account more complex patterns, such as residents inhabiting separate rooms or meeting in a common area, one would expect the problem to increase in difficulty because the model has to take more variables into account. Examples of problematic situations include one person leaving the room while the other remains, people triggering sensors at the same time, etc.



(a) Accuracy

(b) MCC

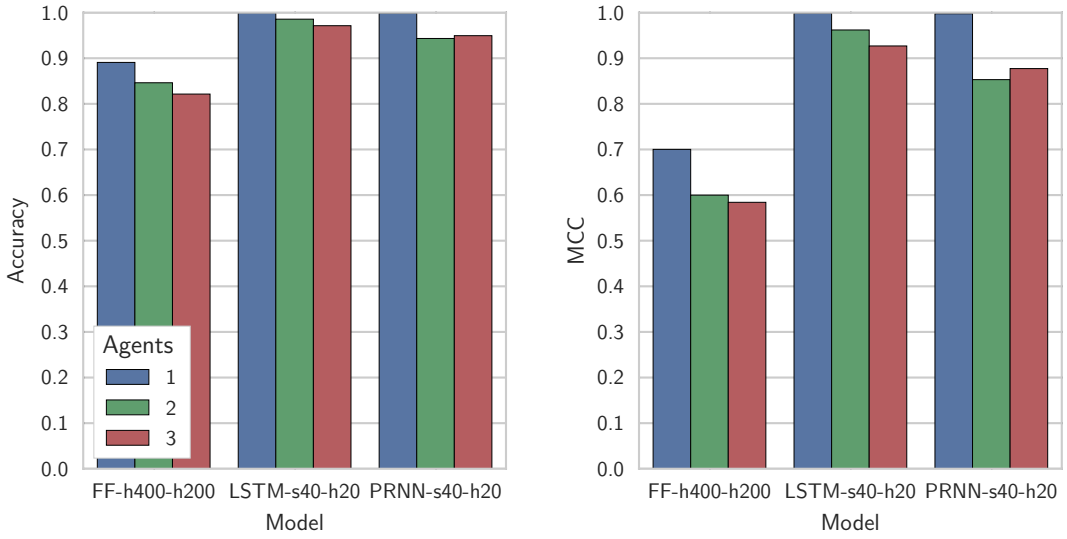
Figure 4.4: Validation performance of simulations with varying fault rate

As one can see in Figure 4.5a, the accuracy of the *CONST* prediction actually decreases in this experiment because of the increasing amount of lights being *ON* (more agents naturally cover a larger area). At the same time accuracy and MCC decrease for all ANNs (see Figure 4.5), with the effect only being slightly more pronounced for the FFNN than for the RNNs. This confirms the expectations given above, that multiple inhabitants make it more difficult for the model to correctly predict if someone is currently in a room or not.

A short look at a segment of the LSTM's prediction in Figure 4.14, compared to its prediction on the single inhabitant simulation in Figure 4.12, shows the differences: The amount of sensor events has increased, while lights stay on for longer timeframes, which does seem to give the LSTM trouble, as can seen when looking at light-1, where it sometimes tries to turn the light off again. However, it is usually able to correct wrong decisions within a relatively short timeframe, once contradicting data is available.

4.2.4 Predicting the future

Another highly interesting use case is predicting the future state of the lights instead of the current one. Assuming a perfect prediction, this would make it possible to never have the resident enter a dark room because the lights would turn on before he actually crosses the threshold. Of course in practice it is not possible to predict an arbitrary amount of time ahead, since the uncertainty increases with every passing second, but short term prediction (around 10 steps) might well be within the realm of the possible



(a) Accuracy (b) MCC

Figure 4.5: Validation performance of simulations with varying amount of agents

and might already result in a significant increase in usability.

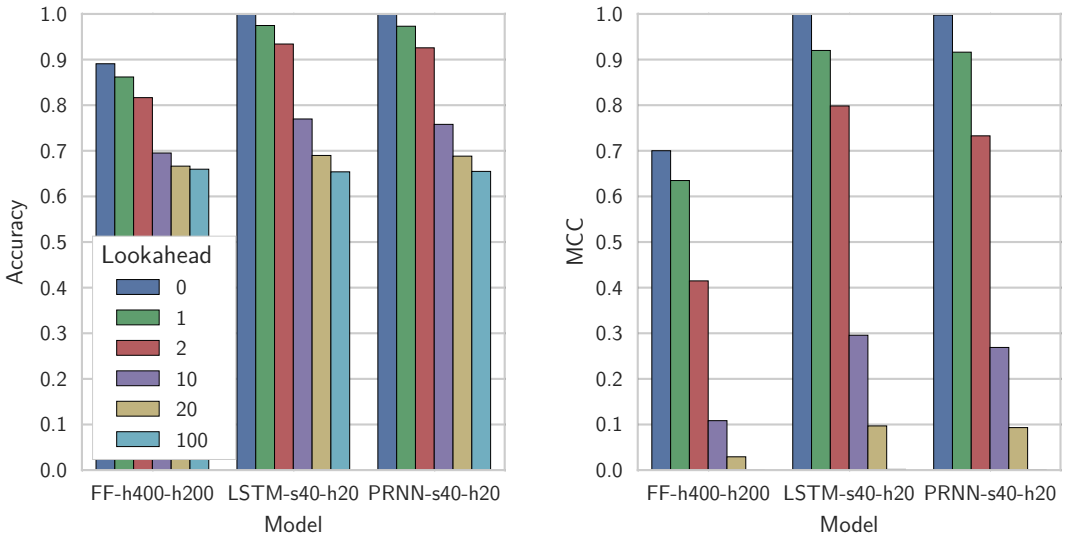
The prediction lengths chosen for this experiment were 0 (i.e. the basic single inhabitant simulation), 1, 2, 10, 20 and 100 steps. 0 and 1 are expected to be trivial, 2 rather easy, 10 and 20 already challenging and 100 close to impossible.

Static prediction

The simplest way, and the approach used in this thesis, to implement predicting n steps ahead, is by shifting the expected outputs in the training/validation data set by n, so that for each sample the expected output is the one n steps in the future. Of course this results in the model only being usable for predicting exactly n steps ahead, which is why it shall be referred to as *static prediction*. One can work around this limitation by training a separate model for each offset (e.g. for 1 steps, 2 steps, ..., n steps) and selecting the required one dynamically.

The results in Figure 4.6 confirm the expectations, with 1 lookahead prediction performing slightly worse than 0, 2 again a bit worse and 10 already only barely viable, which only gets worse for 20 steps and produces basically random results for 100 steps. The RNNs maintain their lead for all lookahead values but 100, where all ANNs perform no better than random prediction. This far in the future, the ANNs are most likely simply not able to find a reasonable prediction any more and thus revert to predicting a function close to the average.

It should be noted however, that this simulation is not a prediction-friendly problem, since it (intentionally) contains lots of movement and thus changes and the agents exhibit no real pattern in their movements. This kind of prediction might work better on real data with less random movement and more high level patterns.



(a) Accuracy

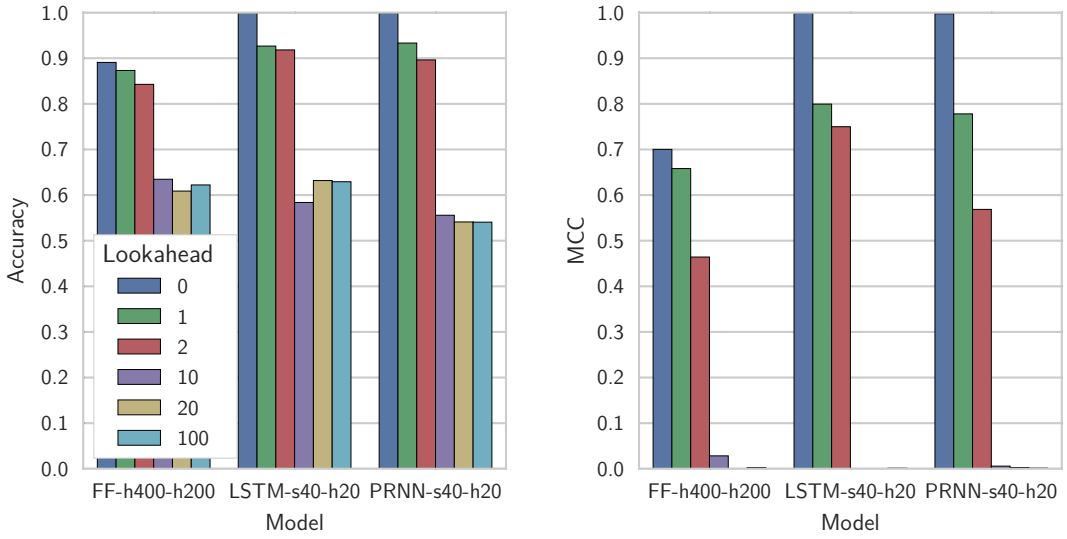
(b) MCC

Figure 4.6: Validation performance of simulations with varying prediction lookahead (static)

Dynamic prediction

Alternatively, one could use the approach used for text generation in [Gra13], where the network is trained to predict next inputs, which are then fed back into the network. By repeating this process, predictions of arbitrary lookahead length can be generated without having to train a separate model.

The results however (Figure 4.7) are significantly worse for all ANNs than the static prediction, with no prediction after (and including) 10 steps being remotely viable. As can be seen in Figure 4.17 the LSTM displays wildly chaotic behavior when predicting just 10 steps into the future. A likely reason for this is, that the output is fed *directly* back into the input, without a classification step in between, as is used for the text generation in [Gra13], which causes the unstable behavior. Unfortunately in this case, it is not as easy as text generation because some inputs are not discrete (US sensors and time in particular) and there exist a large amount of hidden dependencies among the inputs (some sensor can't trigger at the same time, for example). Another problem is of course the temporal compression, which forces the network to predict not only the content of the next events, but also when it will happen.



(a) Accuracy

(b) MCC

Figure 4.7: Validation performance of simulations with varying prediction lookahead (dynamic)

4.3 Light control in apartments (real data)

[DH14] already experimented with various ML methods for light control in apartments. Their comparison includes a MLP, a naive Bayes classifier (NB), a very fast decision tree (VFDT) and two other variations of the VFDT (C4.5 and VFDT++). In their offline experiment based on the 2011–2012 Kyoto data sets from CASAS [Coo+13] (the month from 24.05.2011 to 24.06.2011 to be exact) the VFDT (and variants) outperform the FFNN and the NB (the exact accuracies can be seen in Table 4.2)

The expected state of the lights is generated by taking the light switch events from the data set and tracking if the light is currently on or off. While this is the only way to easily extract this data from the given data set, it is slightly problematic because it depends on the inhabitants choosing the correct state. This may not appear like a problem at first glance, after all, the inhabitants should usually know the ideal configuration for themselves, but it suffers from the fact that the residents probably won't always choose the perfect state, since they may forget to turn off lights, may leave them on due to simple laziness, may choose to leave them off because the switch is in an uncomfortable position and so on. The model might learn some of these suboptimal patterns, resulting in outputs that, while closely mirroring human behavior, are less than ideal. Of course given enough training data the ANN could, theoretically, learn to ignore the noise and extract the real patterns underneath, but the problem of training bad habits into the network remains.

Method	Accuracy
NB	81.18
FFNN	98.17
VFDT	98.57
VFDT++	98.74
C4.5	99.75

Table 4.2: Results from offline experiment from [DH14]

Unfortunately the results from [DH14] for the ANN do not hold up when the network is presented with an entirely separate data set for validation instead of cross-validation. Training on the month from 24.05.2011 to 24.06.2011 with the exact same MLP configuration as in [DH14] and validating on the month from 24.07.2011 to 25.07.2011 shows a significant decrease in performance. As an example the performance difference for light number 7 can be seen in Figure 4.8, where June/June and August/August accuracy was calculated using 10-fold cross-validation.

There are several possible explanations for this. For one, there may be a significant amount of drift in the data set which causes the model which was trained in one month to perform far worse in the following month because the behavior simply is not the same any more.

Of course this drift might not be a general one, but rather an exceptional occurrence because of the selected data. This can easily be verified by choosing a different time range and running the same experiment, in this case the training data was taken from 24.07.2011 to 24.08.2011 and the validation data from 24.08.2011 to 24.09.2011 (the same configuration, but two months later). As one can see in Figure 4.8, while the accuracy is always greatest when validation was performed using cross-validation, July is apparently a rather problematic month since it shows a strong drop in accuracy for both the network trained on June (Accuracy -0.259) and the one trained on August (Accuracy -0.249).

Nevertheless the glaring loss of performance when validating on a different month can not be denied and requires consideration. It may very well be, that one month simply does not contain enough diverse data in order for the model to perform well on other timeframes. The reason for the higher cross-validation accuracy might be, that, given 30 days of data and 10-fold cross-validation, the timeframe from which the validation data is taken (3 days) is rather short and thus probably similar to the temporally close training data.

Furthermore, the sensors chosen in [DH14] include light sensors, which vastly simplify the problem in an unrealistic way since they are highly correlated with the current state of the lights. If the light is on, the light sensors will obviously show that, but this is not information the model has access to in the real world, since the lights would not be controlled by the residents any more (which obviously would defeat the purpose of the

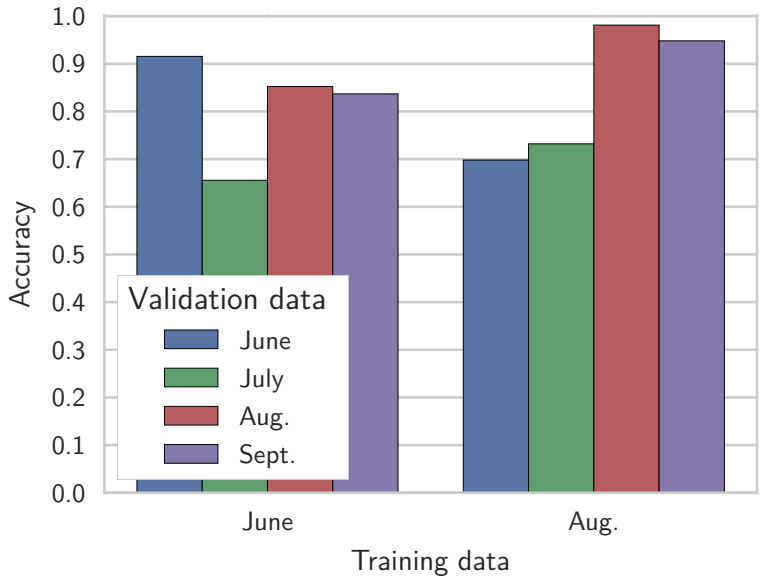


Figure 4.8: Validation accuracy of MLP on light 7 of the CASAS data set June/June and August/August were calculated using 10-fold cross-validation

entire setup).

To see the effect these sensors have on the networks' prediction, one only needs to take a look at Figure 4.18 which shows the prediction of *FF_h20*. The network apparently seems to fit L6 near perfectly and with very high confidence, but when one compares the prediction with the input of LS205 it can be seen that the light sensor goes low at exactly the same time as the light turns off (around 320). The same behavior can be found with L7 and LS41, but the network commits the error of not reacting to the first drop in brightness (around 95) which corresponds with the light turning off, but rather waits for the sensors output to go completely down (around 190) before turning the light off. Likewise the beginning of L4 correlates with a rise of LS204 (around 10) and the end with a drop of LS204 (around 395).

In order to reduce these problems, the data set used for the following experiments is larger (3 months from 24.05.2011 to 24.8.2011) and validation is performed on a data set from a different timeframe (1 month from 24.8.2011 to 24.9.2011). The the only sensors used are motion sensors $(1, 15, 16, 18, 19, 20, 23, 24, 27, 28, 29, 36, 37, 41, 42, 43)$, area motion sensors $(201 - -207)$ and normalized time sensors (hours, minutes, seconds, milliseconds). The value of a motion sensor is always kept until a new event changes it, i.e. if a motion sensor turns on at time t and off at $t + k$, its value will be on for the k steps in between.

As one would expect, the performance is significantly worse than on the simulated data,

with only the LSTM showing even remotely acceptable performance (see Figure 4.11). All ANNs fall back to conservatively predicting *OFF* far more, resulting in a high specificity (Figure 4.11d), but rarely correctly predict that a light should be *ON*, which results in a very low sensitivity (Figure 4.11c) and MCC (Figure 4.11b). Thus all ANNs seem to be quite good at predicting when a light should be off, but rather bad at predicting when it should be on, with the LSTM as the leader with a sensitivity of 0.22.

The MCC for individual lights can be seen in Figure 4.9, which is significantly worse than the performance on the data set with light sensors in Figure 4.10. Nevertheless there are interesting bits of information one can deduce from it. For one, the LSTM consistently performs better than the FFNNs, particularly on L5, L6 and L7. On L1 and L2 *FF-h20* has a MCC of 0, suggesting that it was completely unable to predict those lights, while *FF-h400-h200* is able to make useful predictions, indicating that the small FFNN is not expressive enough. L10 seems to be the easiest light to predict for all ANNs, with hardly any difference between *FF-h20* and *FF-h400-h200* and a lead of just below 0.1 for the LSTM. On L6 and L7 only the LSTM is able to perform reasonably, indicating that these lights are highly dependent on past events, while no ANN performs well on L9.

As one can see in the component plot for *LSTM-s40-h20* (Figure 4.19), the LSTM does show definite signs of fitting the given function. For instance, it (mostly) correctly predicts when L6, L7 and L10 should be *ON*/*OFF*, but mistakenly turns on L5 at the same time as L6 and L7.

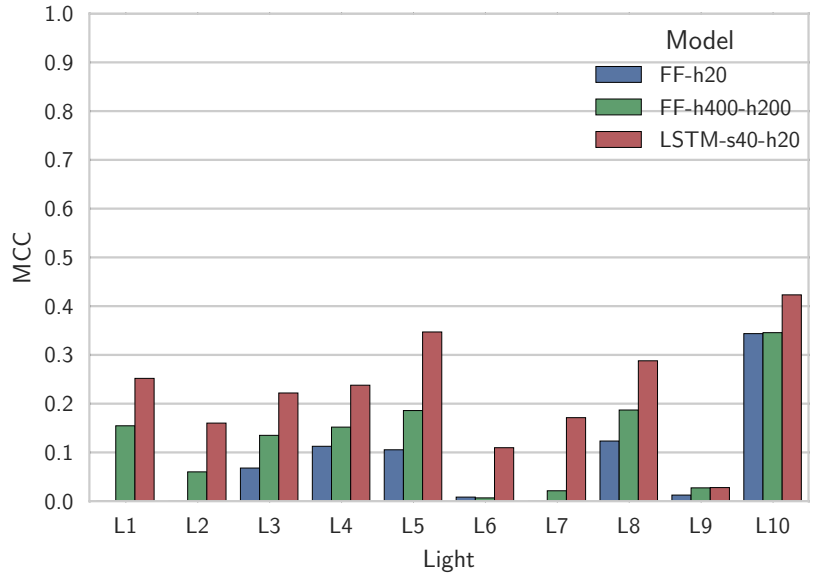


Figure 4.9: Validation MCC on different lights of the CASAS data set with motion sensors only

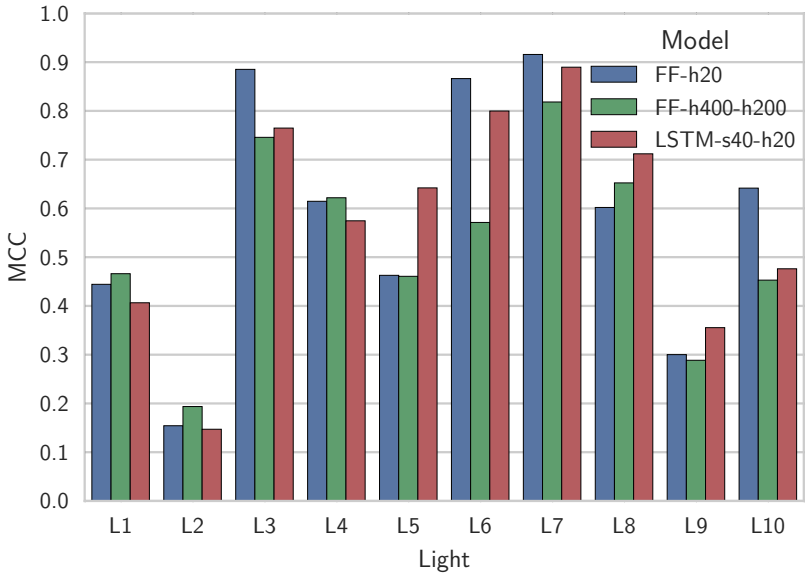
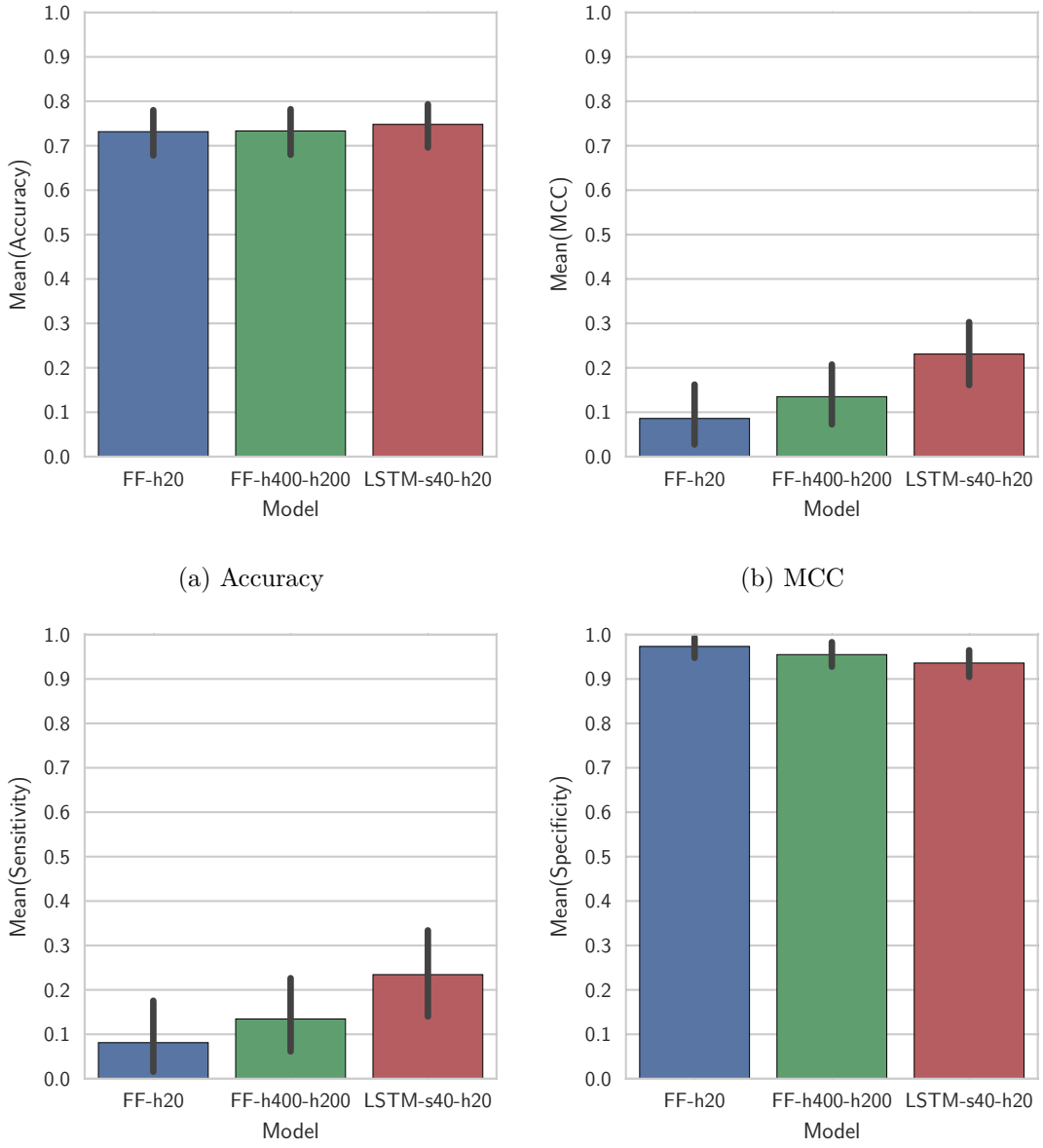


Figure 4.10: Validation MCC on different lights of the CASAS data set with motion and light sensors



(a) Accuracy

(b) MCC

(c) Sensitivity

(d) Specificity

Figure 4.11: Validation performance on CASAS data set with motion sensors only

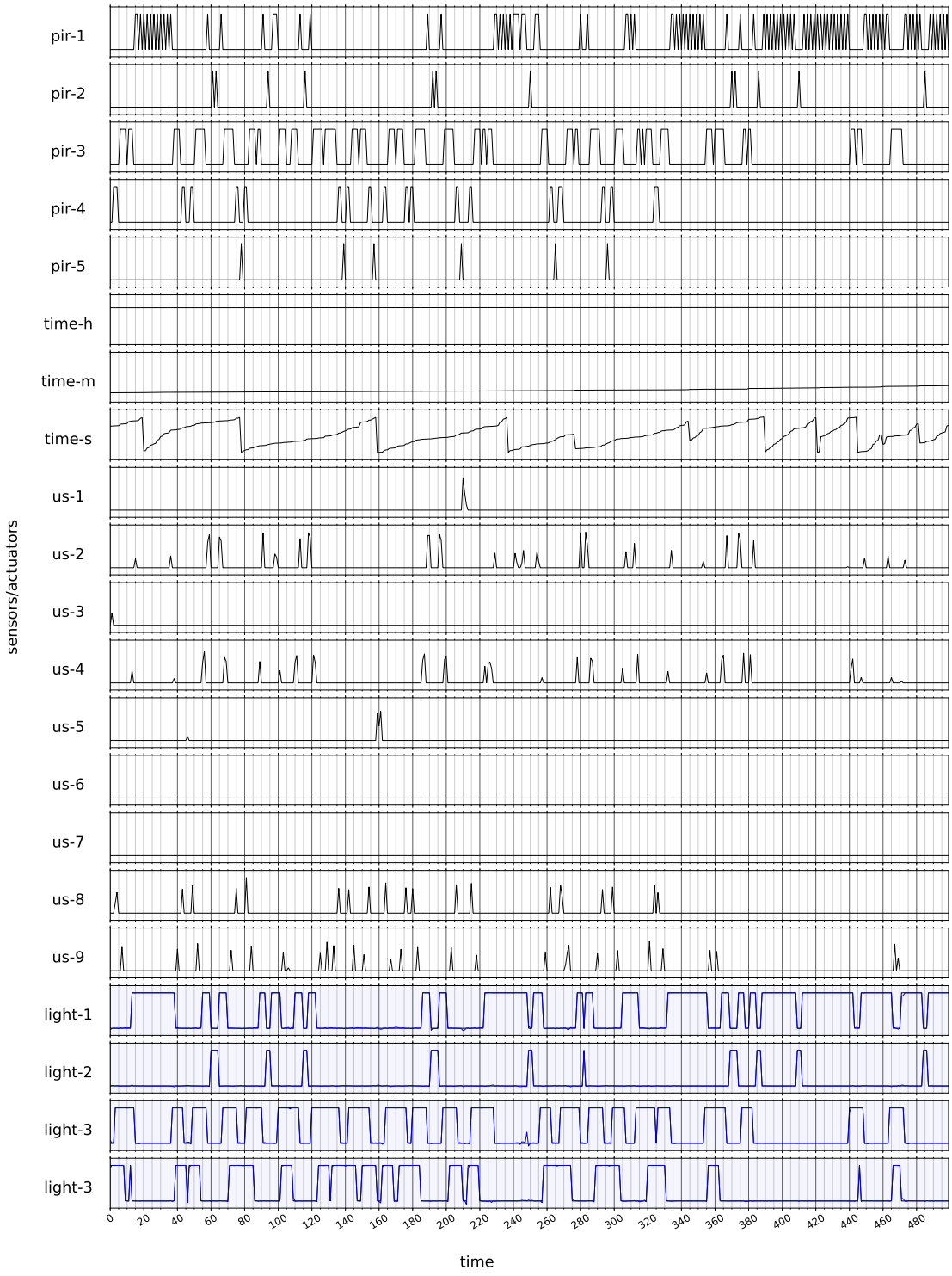


Figure 4.12: Sensor/actuator values and predictions (blue) of *LSTM-s40-h20* for single inhabitant simulation

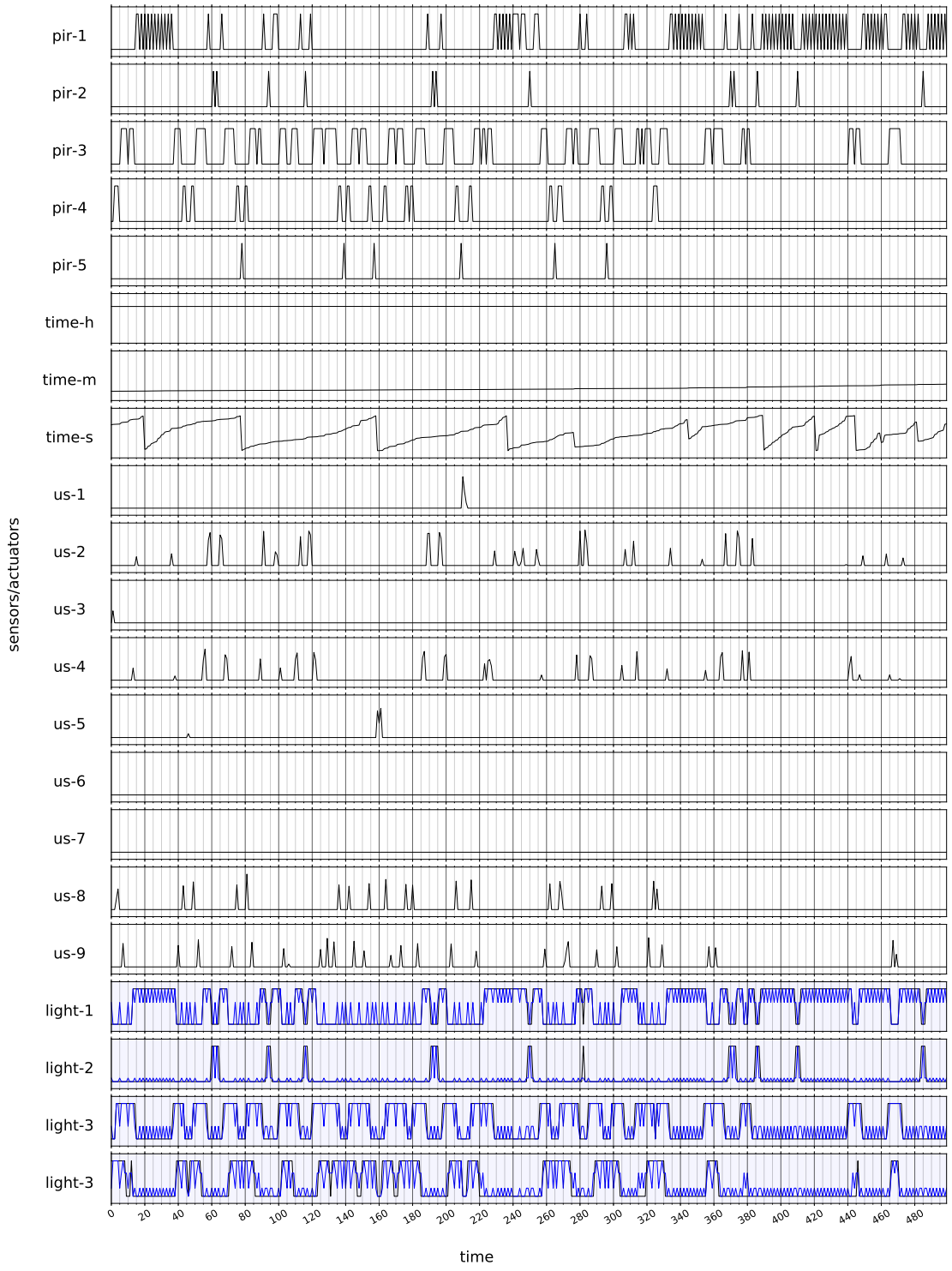


Figure 4.13: Sensor/actuator values and predictions (blue) of *FF-h400-h200* for single inhabitant simulation

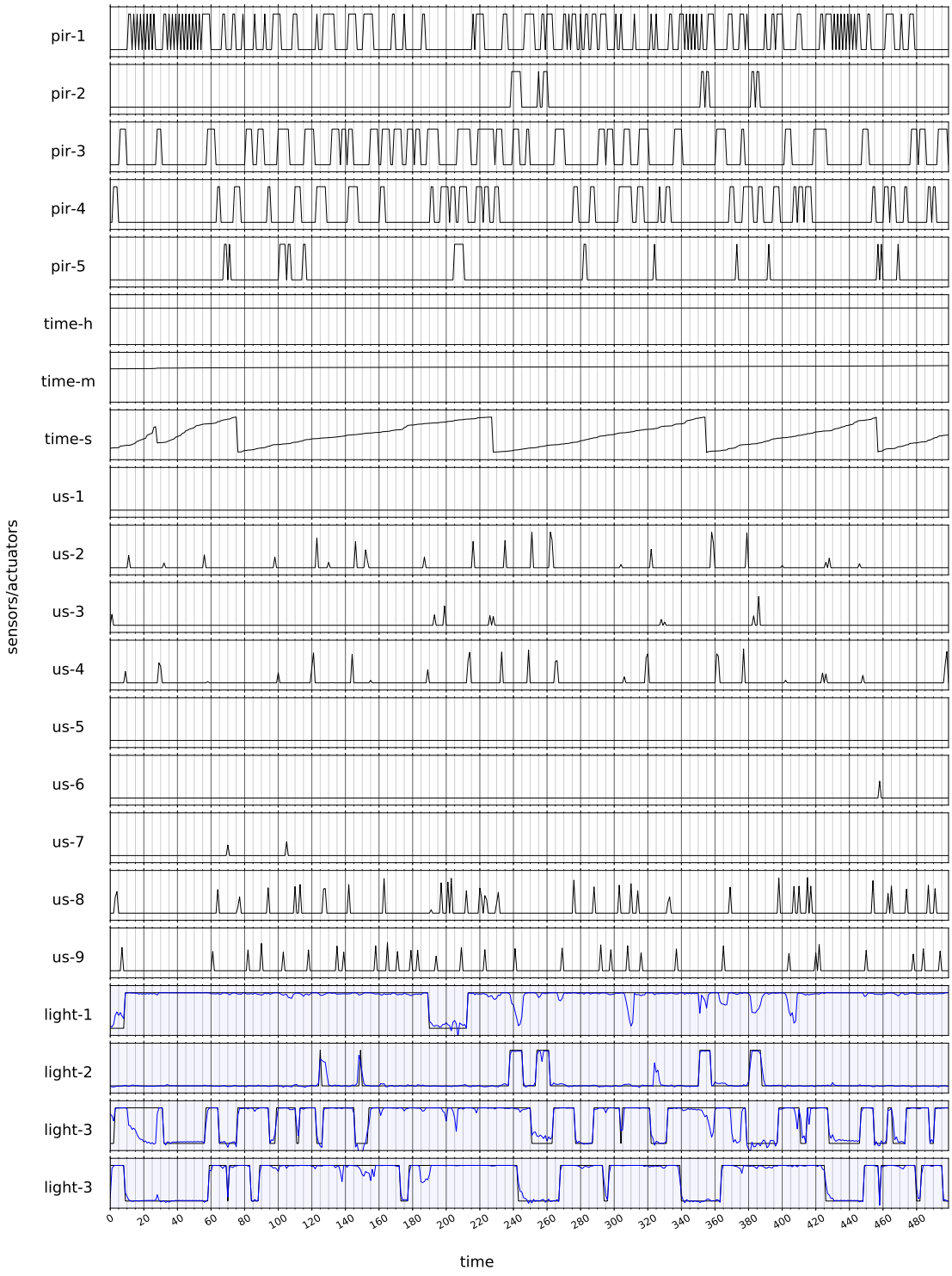


Figure 4.14: Sensor/actuator values and predictions (blue) of *LSTM-s40-h20* for simulation with 3 agents

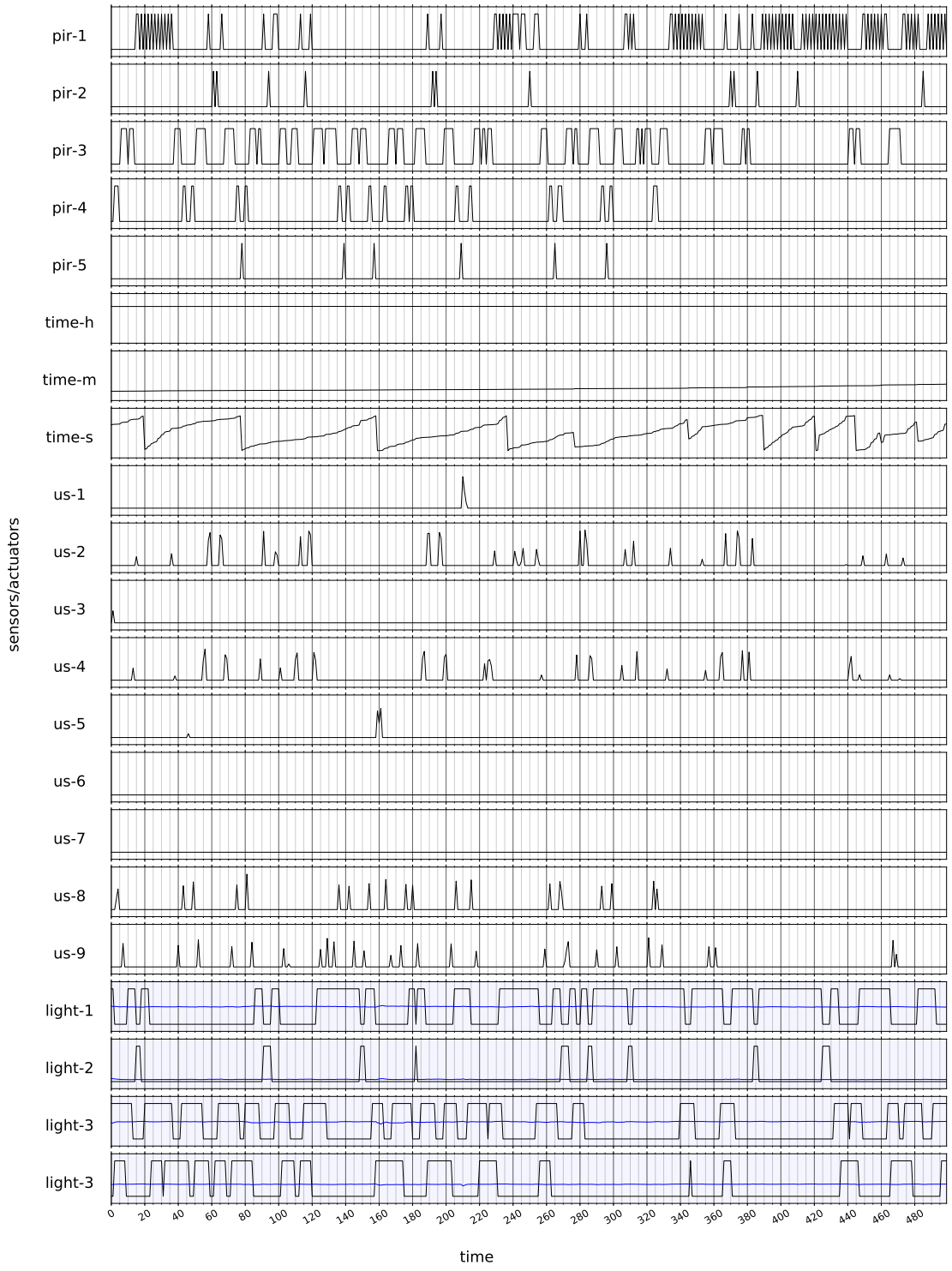


Figure 4.15: Sensor/actuator values and predictions (blue) of *LSTM-s40-h20* for prediction with 100 steps lookahead (static)

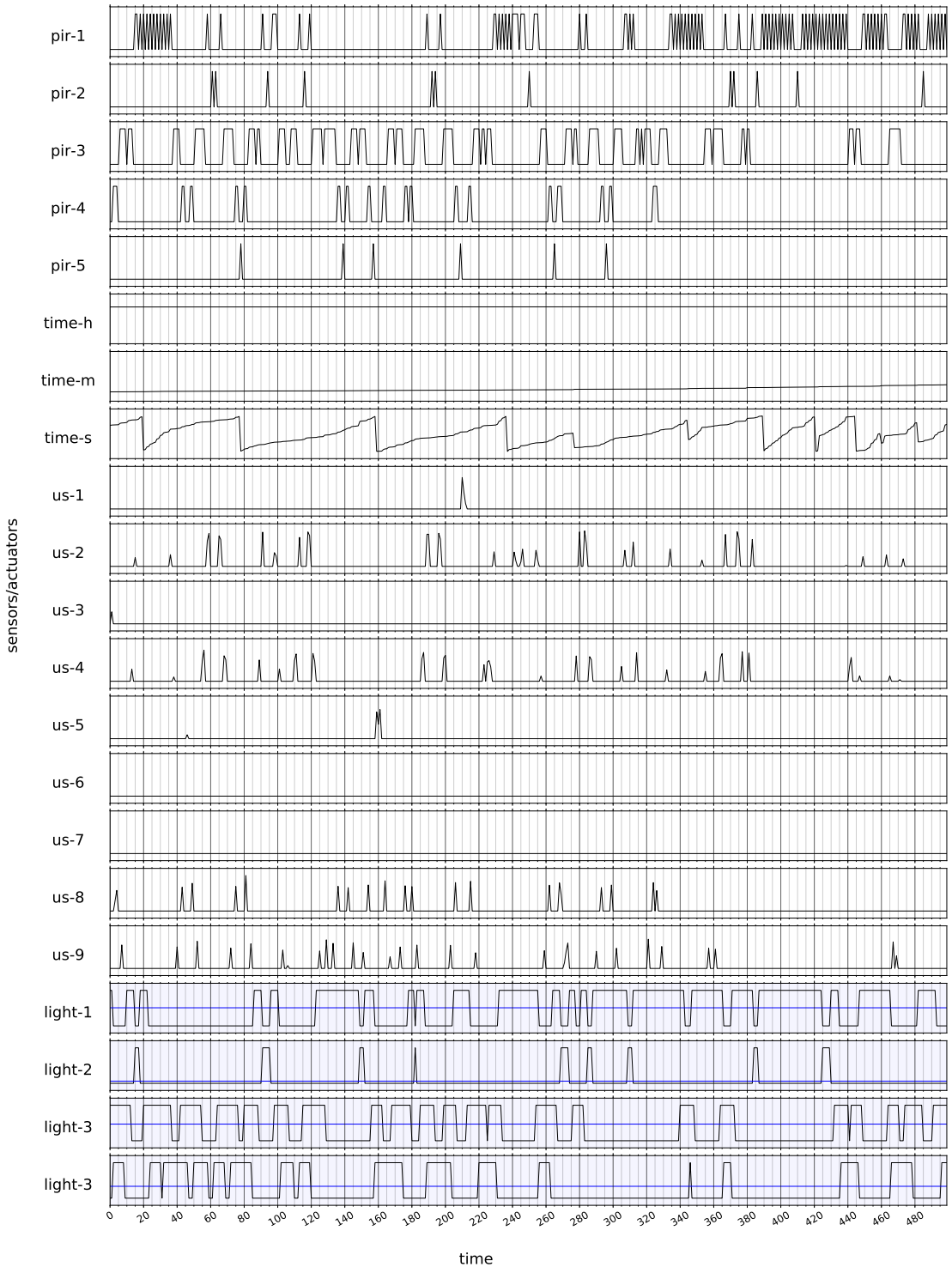


Figure 4.16: Sensor/actuator values and predictions (blue) of *FF-h400-h200* for prediction with 100 steps lookahead (static)

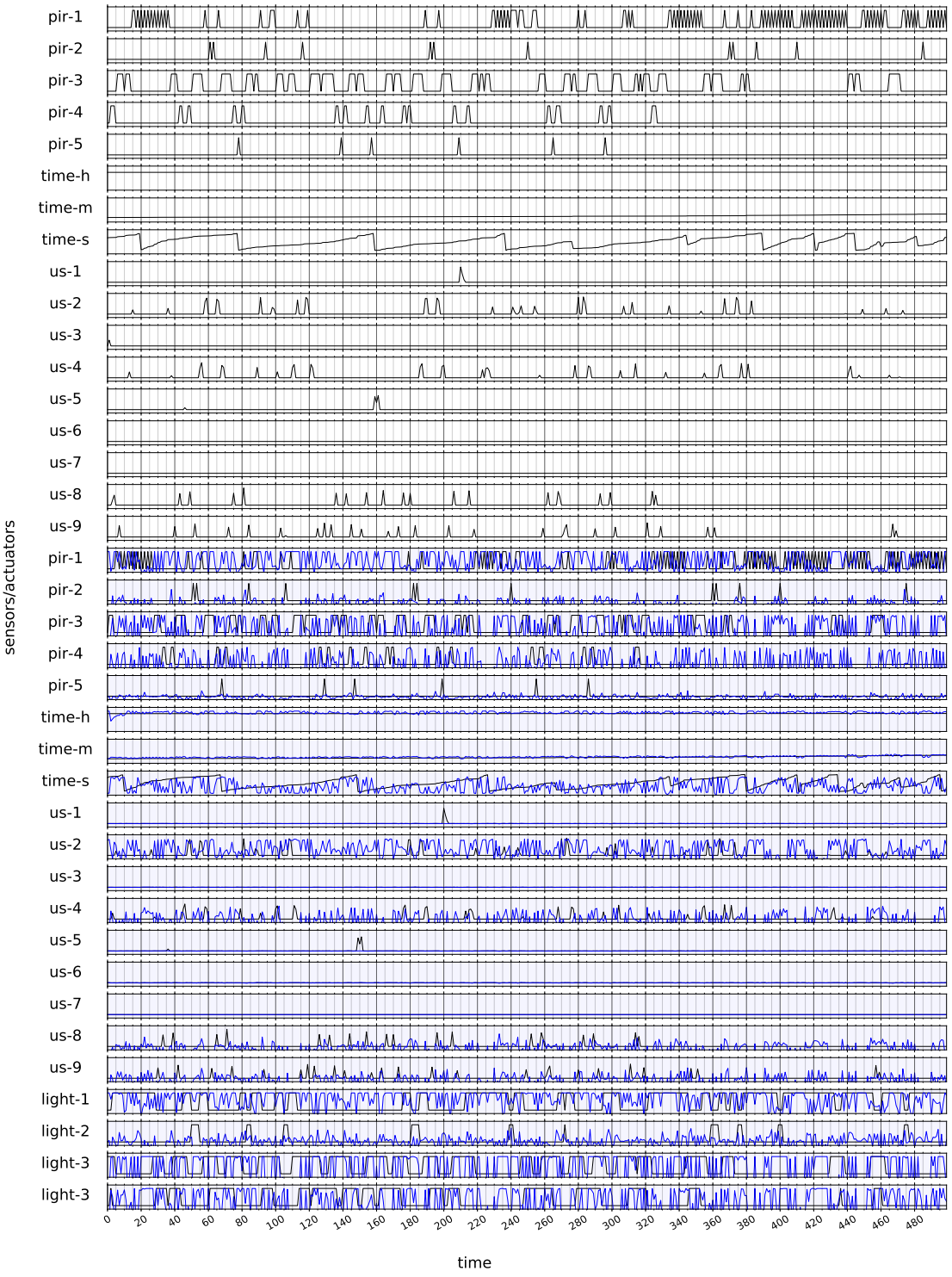


Figure 4.17: Sensor/actuator values and predictions (blue) of *LSTM-s40-h20* for prediction with 10 steps lookahead (dynamic)

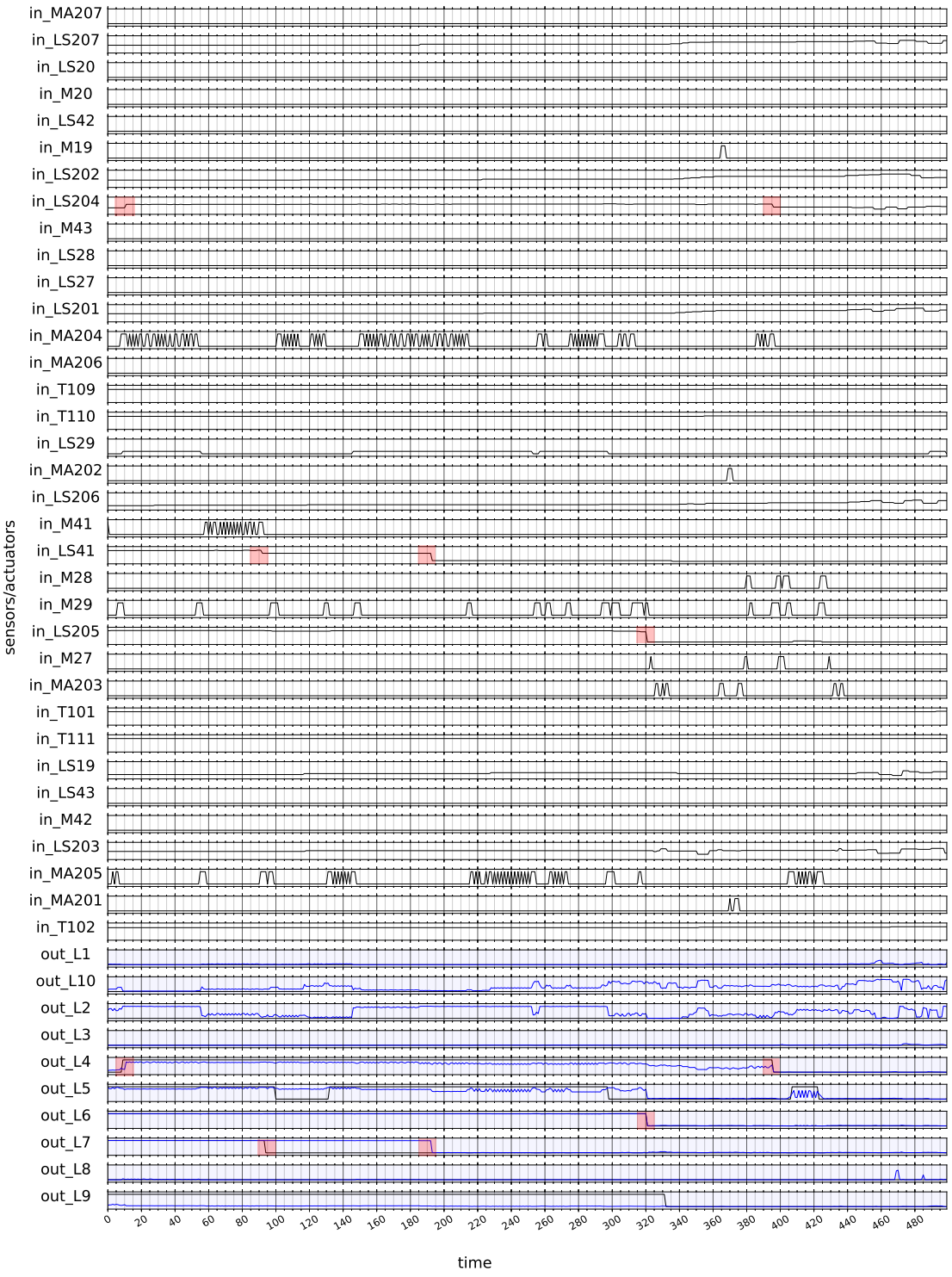


Figure 4.18: Sensor/actuator values and predictions (blue) of *FF_h20* for CASAS data set with sensors from [DH14] with relevant section marked (red)

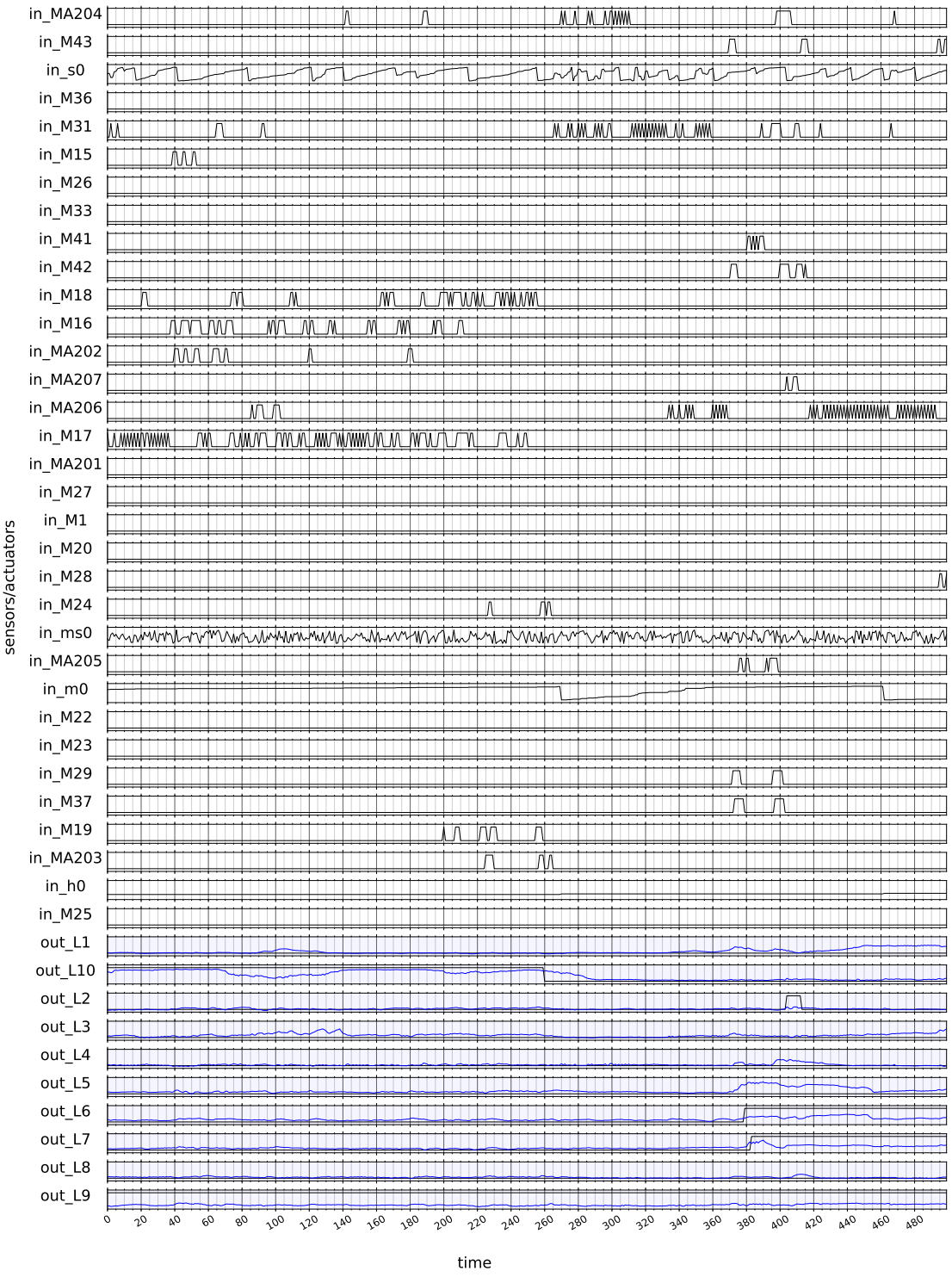


Figure 4.19: Sensor/actuator values and predictions (blue) of *LSTM-s40-h20* for CASAS data set with motion sensors only

CHAPTER 5

Conclusion

5.1 Summary

The suitability of artificial neural networks for automatic light control was evaluated by means of several simulated experiments with varying parameters and one experiment on real data from CASAS [Coo+13]. The FFNNs show acceptable performance at the simpler configurations (such as the simulation with only one resident), but are severely outperformed by the RNNs on the harder problems. Most notably, the RNNs display a rather high aptitude for recognizing and correcting faults and are able to handle multiple inhabitants far better because their internal memory allows them to use past events as the context under which they evaluate the current event. In the studied experiments, the LSTM consistently performed on par with or better than the PRNN.

While the performance on the CASAS data set is, not unexpectedly, clearly worse than on the simulated data, the LSTM still performs far better (roughly twice as good, if compared on the MCC) than the FFNN. One should keep in mind however, that no US barriers were included in this data, which are useful for detecting people entering/leaving rooms and, more importantly, that the expected output was generated from the light states selected by the inhabitants, which are subject to suboptimal decision making and unpredictable behavior. The far better results of [DH14] could unfortunately not be replicated, but it is likely due to the validation on data from completely separate months being more punishing than cross-validation and the inclusion of light sensors giving the ANN information it would not have access to in a real setting.

The largest drawback of most of the experiments presented is of course the simulated data, which creates a far easier problem than the real data, as a comparison with the experiment on real data shows. However, taking into account the drawbacks of the approach used to define the expected outputs for the real data and due to the good results on the simulated problems, ANNs, and RNNs in particular, may still perform

well in a real environment. However combining them with a higher level control systems as used in [MM08] may be necessary.

5.2 Future work

The obvious next step would be to evaluate the performance of the networks on data that was explicitly labeled with the correct light states, instead of inferring them from the inhabitants actions. After this, a real system could be created by outfitting a real home with sensors and giving the ANNs control over the lights in this home. This would allow insights into how well the model performs in a live environment, if it is able to adapt to changing conditions and if high-level patterns of the inhabitants (e.g sleeping patterns) can be recognized and learned. At this point, allowing continuous values of the lights (i.e dimming) might also be worth a consideration and allows for some interesting actions, such as only turning the light on slightly when a resident gets up at night. The approaches used in this paper may also, with small adjustments, be used to determine the rough location of the residents (e.g. on a room level), perhaps with some fault tolerance, which in turn can be used for various other home automation tasks.

Additionally, a greater variety of sensors should be incorporated as inputs into the model and their information gain evaluated, with promising leads being:

- Ambient light sensors, to determine if artificial light is necessary.
- Devices connected to the local network, to determine which (portable) devices currently reside in the home. This information is particularly useful nowadays, since almost everyone is carrying a smartphone which usually automatically connects to known networks. The signal strength could serve as an additional indicator of the persons position.

  Radio beacons [LaM+05] might also be worth considering as an interesting alternative for localization.
- Magnetic contact switches, to provide indicators of doors opening and closing. These might be especially useful to determine if someone entered or left the home.
- While cameras pose certain privacy problems, they are definitely among the sensors that provide the most information (while maintaining an affordable price tag). Combined with ANNs for automatically recognizing the contents of the video stream [Ji+13] they could provide a large amount of information for all kinds of home automation tasks, especially since surveillance cameras may already be installed in a good amount of locations.

Furthermore the suitability of RNNs for other use cases in home automation, such as home security (particularly in view of recognizing/preventing false alarms), heating and ventilation, etc., where their sequential nature may be of use.

Finally, prediction of future events should be explored in more depth, particularly the dynamic prediction, to see if the chaotic behavior of the RNNs can be reduced or eliminated.

List of Figures

Acronyms

ADL activity of daily living. 9, 16

ANN artificial neural network. 3–6, 9, 11, 16, 18–27, 29, 41, 42

BP backpropagation. 4, 11

BPTT backpropagation through time. 5, 11

CEC constant error carrousel. 6

CPU central processing unit. 12, 13

FFNN feed forward neural network. 5, 6, 9–13, 18, 19, 21–23, 26, 27, 29, 41, 45

GPU graphics processing unit. 4, 9, 12, 13

LSTM long short-term memory neural network. 6, 9, 12, 14, 19, 22, 23, 25, 29, 41, 45

MCC Matthews correlation coefficient. 7, 8, 11, 19–26, 29–31, 41, 45

ML machine learning. 2, 7, 10, 16, 26

MLP multilayer perceptron. 5, 26–28, 45

MSE mean square error. 4, 11

NB naive Bayes classifier. 26, 27

PIR passive infrared. 16–18, 22

PRNN plain recurrent neural network. 5, 6, 12, 19, 22, 41

ReLU rectified linear unit. 3

RMLP recurrent multilayer perceptron. 5, 6, 45

RNN recurrent neural network. 5, 9, 18, 19, 21–24, 41–43

SGD stochastic gradient descent. 4

TanH hyperbolic tangent. 3, 11, 12

US ultrasonic. 16–18, 22, 25, 41

VFDT very fast decision tree. 26, 27

Appendix A

pir-1	pir-2	pir-3	pir-4	pir-5	time-h	time-m	time-s	us-1	us-2	us-3	us-4	us-5	us-6	us-7	us-8	us-9
0	0	0	0	0	0.835	0.062	0.733	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0.835	0.062	0.745	0	0	0.338	0	0	0	0	0	0
0	0	0	1	0	0.835	0.062	0.747	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0.835	0.062	0.750	0	0	0	0	0	0	0	0.296	0
0	0	0	1	0	0.835	0.062	0.752	0	0	0	0	0	0	0	0.581	0
0	0	0	0	0	0.835	0.062	0.754	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0.835	0.063	0.794	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0.835	0.063	0.803	0	0	0	0	0	0	0	0	0.661
0	0	1	0	0	0.835	0.063	0.806	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0.835	0.063	0.810	0	0	0	0	0	0	0	0	0

Table A.1: Sample datapoints (sensor part)

light-1	light-2	light-3	light-4
0	0	0	1
0	0	0	1
0	0	0	1
0	0	1	1
0	0	1	1
0	0	1	1
0	0	1	1
0	0	1	1
0	0	1	1
0	0	1	0

Table A.2: Sample datapoints (actuator part)

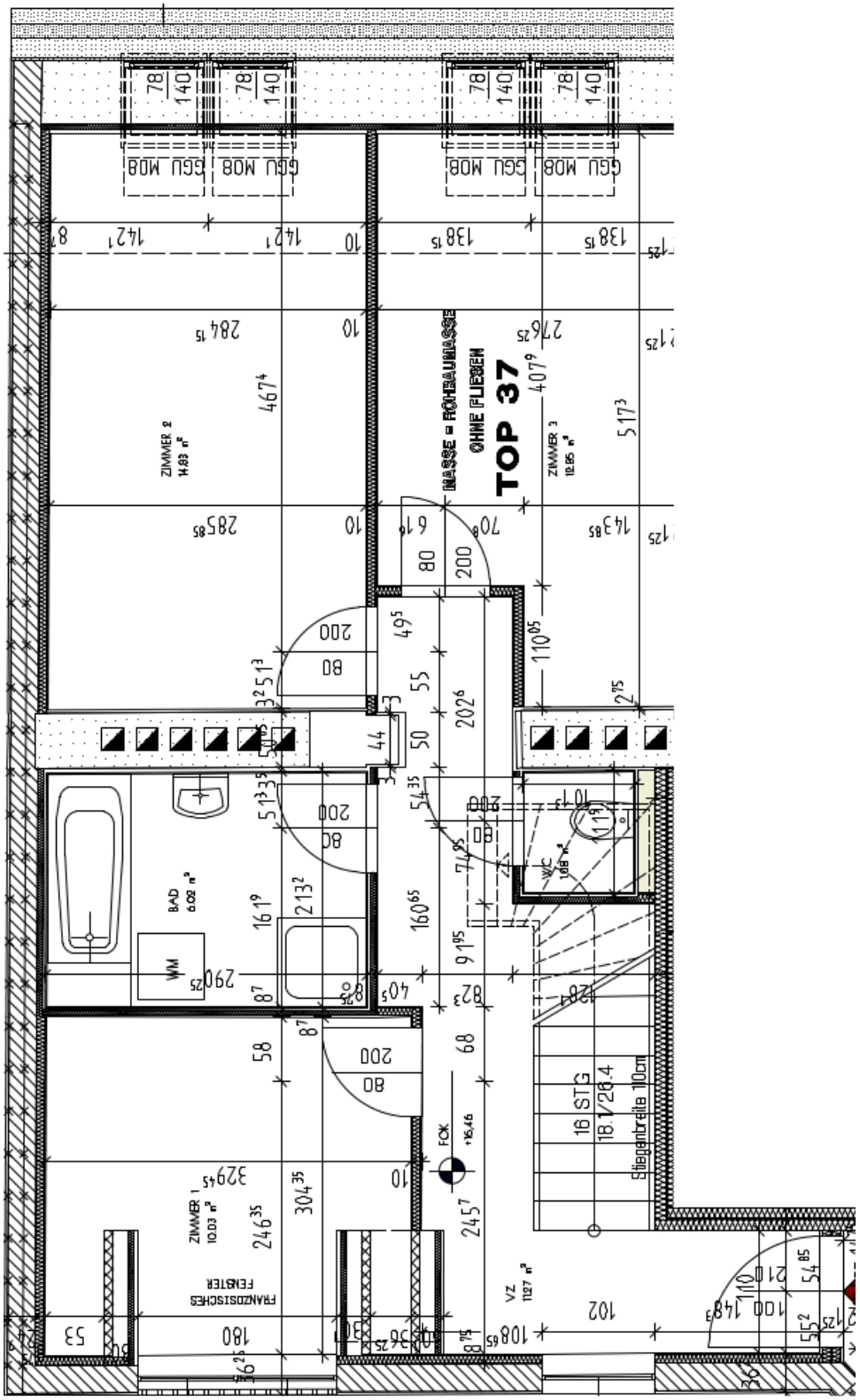


Figure A.1: Lower floor of the apartment

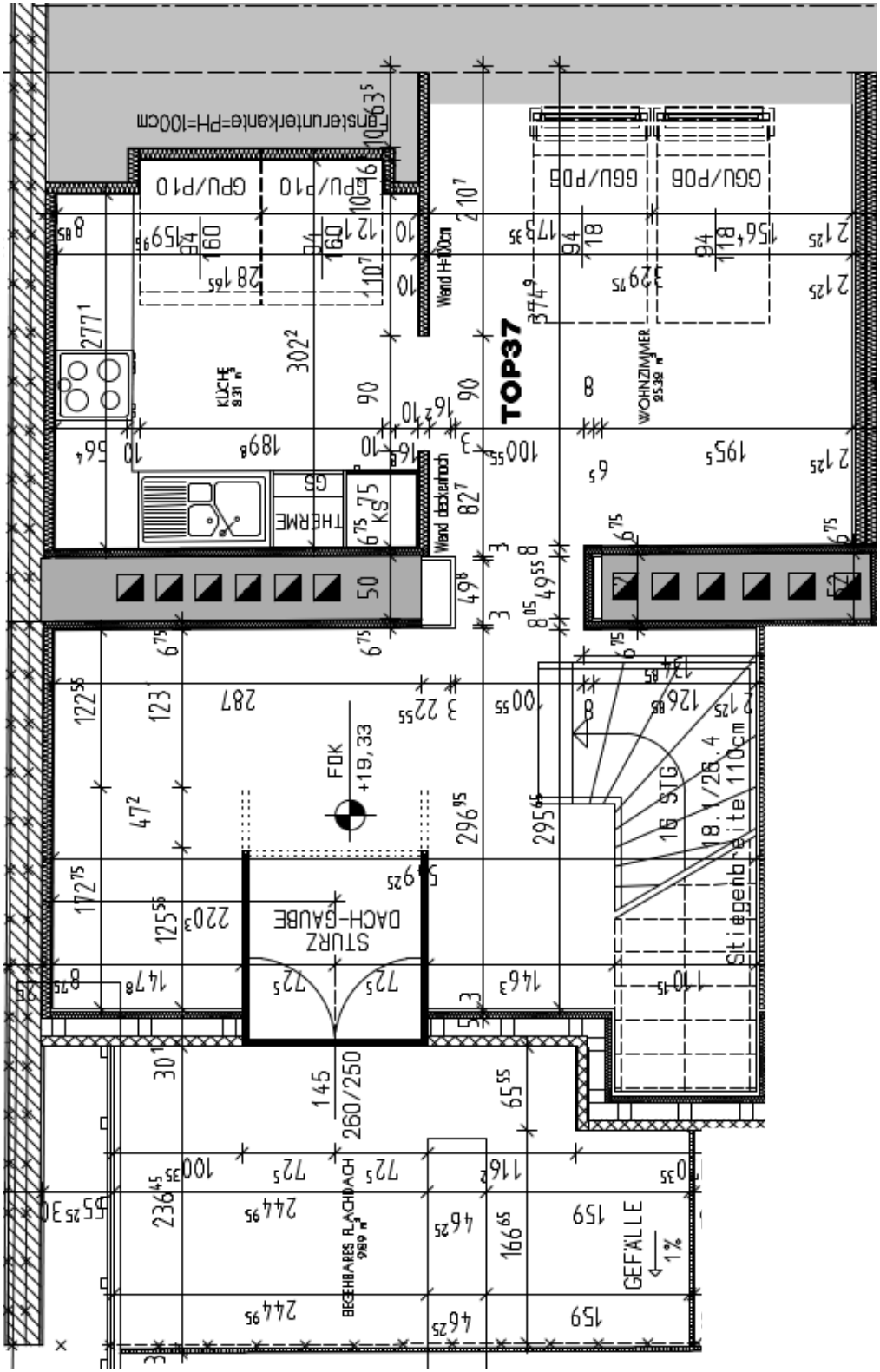


Figure A.2: Upper floor of the apartment

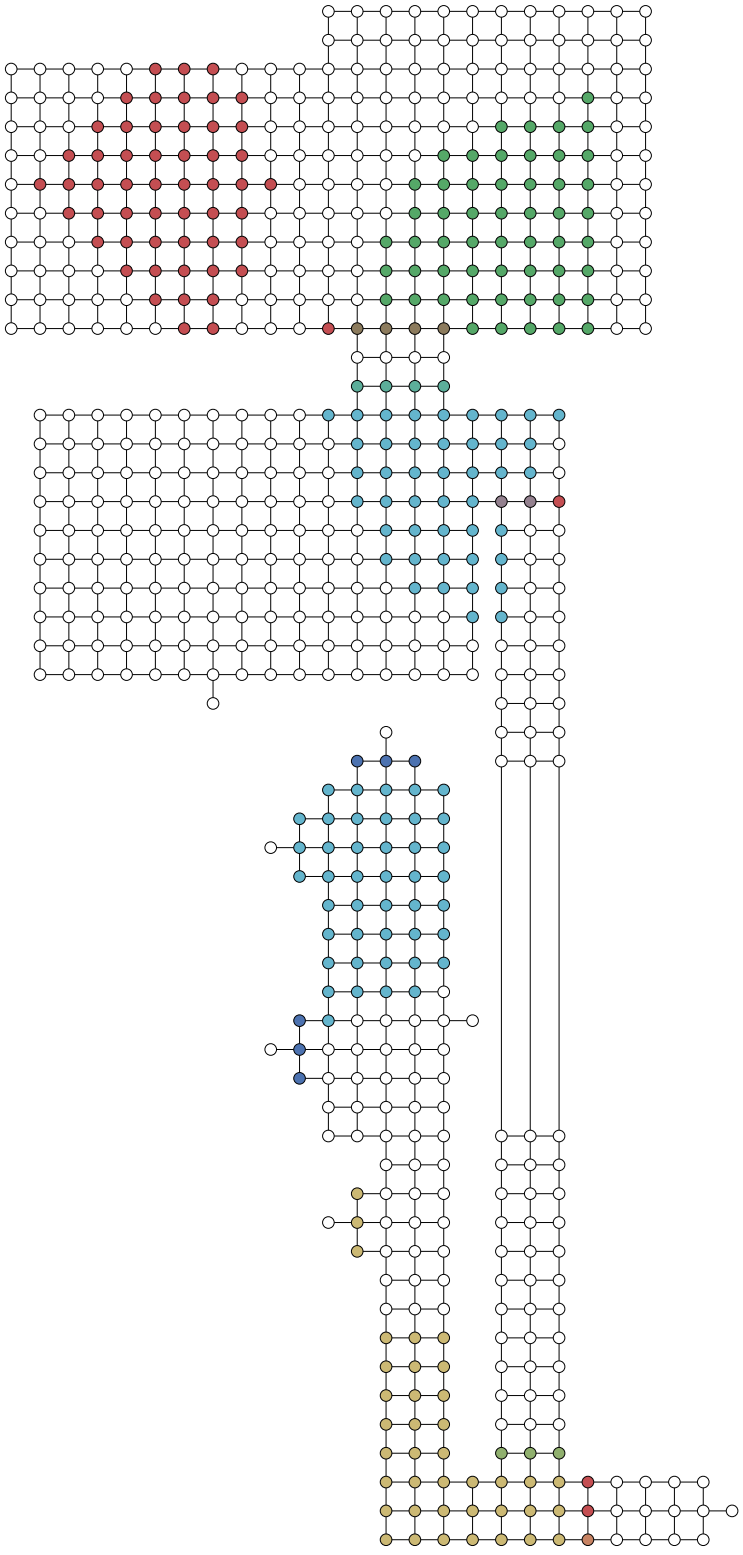


Figure A.3: Apartment graph with areas covered by sensors

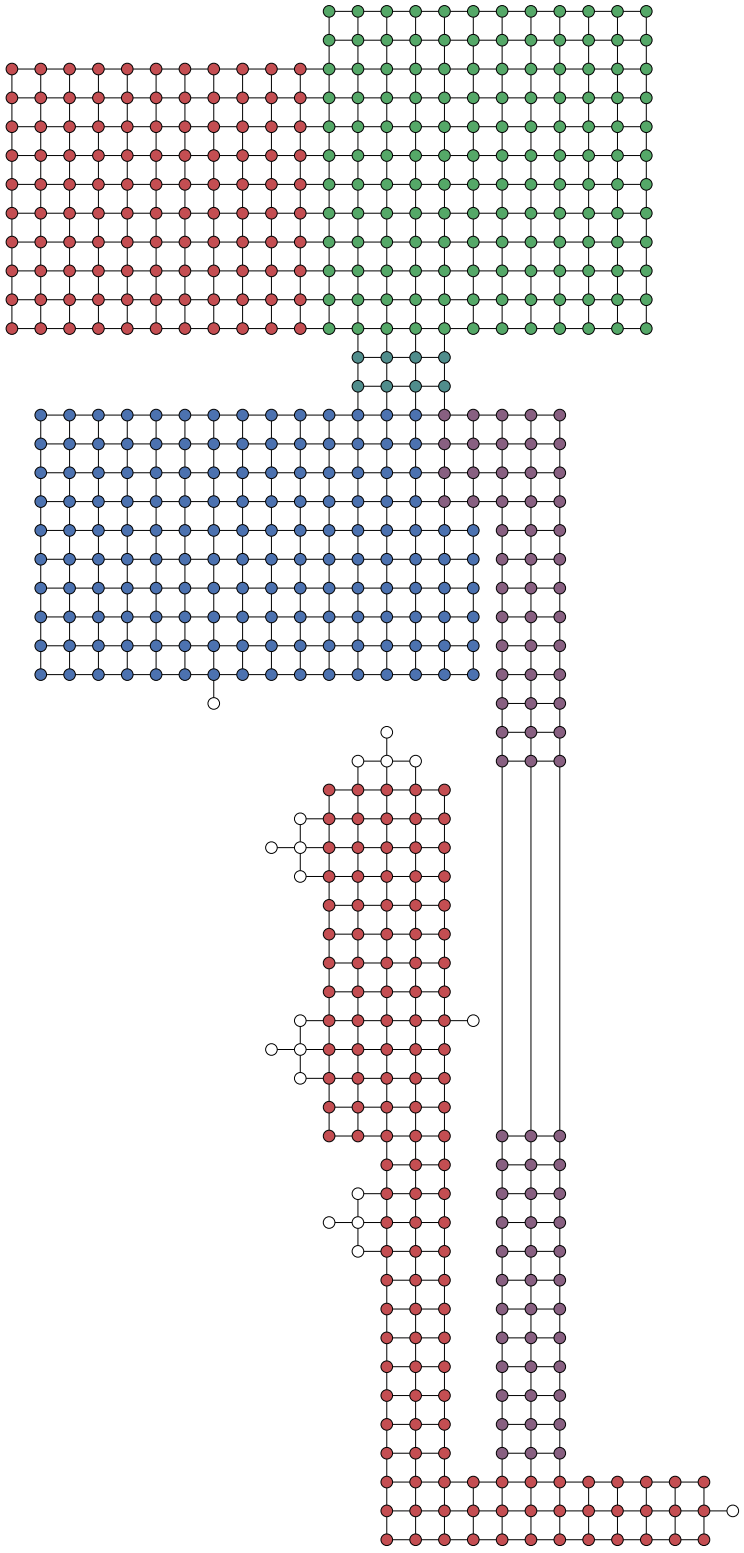


Figure A.4: Apartment graph with areas covered by actuators (lights)

Bibliography

[Adh] Adhoco. *adhoco adaptive building control*. URL: http://www.adhoco.com/ (visited on 03/11/2016).

[Ass] KNX Association. *KNX Association*. URL: https://www.knx.org (visited on 09/03/2016).

[Bal+00] Pierre Baldi et al. „Assessing the accuracy of prediction algorithms for classification: an overview". In: *Bioinformatics* 16.5 (2000), pp. 412–424.

[CDC10] C. Chen, B. Das, and D. J. Cook. „A Data Mining Framework for Activity Recognition in Smart Environments". In: *2010 Sixth International Conference on Intelligent Environments (IE)*. July 2010, pp. 80–83.

[Cho15] François Chollet. *Keras*. https://github.com/fchollet/keras. 2015.

[Coo+13] D. J. Cook et al. „CASAS: A Smart Home in a Box". In: *Computer* 46.7 (July 2013), pp. 62–69.

[Dan+12] B. K. Dan et al. „Robust people counting system based on sensor fusion". In: *IEEE Transactions on Consumer Electronics* 58.3 (Aug. 2012), pp. 1013–1021.

[DH14] I. B. P. P. Dinata and B. Hardian. „Predicting smart home lighting behavior from sensors and user input using very fast decision tree with Kernel Density Estimation and improved Laplace correction". In: *2014 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. 2014 International Conference on Advanced Computer Science and Information Systems (ICACSIS). Oct. 2014, pp. 171–175.

[Die+15] Sander Dieleman et al. *Lasagne: First release.* 2015. DOI: 10.5281/zenodo.27878. URL: http://dx.doi.org/10.5281/zenodo.27878.

[ES02] D. Eck and J. Schmidhuber. „Finding temporal structure in music: blues improvisation with LSTM recurrent networks". In: *Proceedings of the 2002 12th IEEE Workshop on Neural Networks for Signal Processing, 2002*. Proceedings of the 2002 12th IEEE Workshop on Neural Networks for Signal Processing, 2002. 2002, pp. 747–756.

[FC98] Kurt M. Fanning and Kenneth O. Cogger. „Neural network detection of management fraud using published financial data“. In: *International Journal of Intelligent Systems in Accounting, Finance & Management* 7.1 (1998), pp. 21–41.

[GJM13] A. Graves, N. Jaitly, and A. r Mohamed. „Hybrid speech recognition with Deep Bidirectional LSTM“. In: *2013 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. 2013 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU). Dec. 2013, pp. 273–278.

[GR94] Sushmito Ghosh and Douglas L. Reilly. „Credit card fraud detection with a neural-network“. In: *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*. Vol. 3. IEEE, 1994, pp. 621–630.

[Gra13] Alex Graves. „Generating Sequences With Recurrent Neural Networks“. In: *arXiv:1308.0850 [cs]* (Aug. 4, 2013). arXiv: 1308.0850.

[GSC00] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. „Learning to forget: Continual prediction with LSTM“. In: *Neural computation* 12.10 (2000), pp. 2451–2471.

[GSS02] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. *Learning Precise Timing with LSTM Recurrent Networks*. 2002.

[Hay07] Simon Haykin. *Neural Networks: A Comprehensive Foundation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007. ISBN: 9780131471399.

[HIS07] Jon Hutchins, Alexander Ihler, and Padhraic Smyth. „Modeling count data from multiple sensors: a building occupancy model“. In: *Computational Advances in Multi-Sensor Adaptive Processing, 2007. CAMPSAP 2007. 2nd IEEE International Workshop on*. IEEE, 2007, pp. 241–244.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. „Long short-term memory“. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Ins] Insteon. *Insteon*. URL: http://www.insteon.com/ (visited on 03/11/2016).

[Ji+13] Shuiwang Ji et al. „3D convolutional neural networks for human action recognition“. In: *IEEE transactions on pattern analysis and machine intelligence* 35.1 (2013), pp. 221–231.

[JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. „An empirical exploration of recurrent network architectures“. In: *Journal of Machine Learning Research* (2015).

[KB14] Diederik Kingma and Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: *arXiv:1412.6980 [cs]* (Dec. 22, 2014). arXiv: 1412.6980.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LaM+05] Anthony LaMarca et al. „Place lab: Device positioning using radio beacons in the wild“. In: *International Conference on Pervasive Computing*. Springer, 2005, pp. 116–133.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. „Deep learning“. In: *Nature* 521.7553 (May 28, 2015), pp. 436–444.

[Mat75] Brian W. Matthews. „Comparison of the predicted and observed secondary structure of T4 phage lysozyme“. In: *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405.2 (1975), pp. 442–451.

[MM08] Carlos Machado and José A. Mendes. „Automatic light control in domotics using artificial neural networks“. In: *World Academy of Science, Engineering and Technology* 44 (2008), pp. 813–818.

[Moz98] Michael C. Mozer. „The neural network house: An environment hat adapts to its inhabitants“. In: *Proc. AAAI Spring Symp. Intelligent Environments*. Vol. 58. 1998.

[Nes] Nest. *Nest*. URL: `https://nest.com/` (visited on 03/11/2016).

[NH10] Vinod Nair and Geoffrey E. Hinton. „Rectified linear units improve restricted boltzmann machines“. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814.

[Pow11] David Martin Powers. „Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation“. In: (2011).

[RCH06] F. Rivera-Illingworth, V. Callaghan, and H. Hagras. „Automated Discovery of Human Activities inside Pervasive Living Spaces“. In: *2006 First International Symposium on Pervasive Computing and Applications*. Aug. 2006, pp. 77–82.

[The16] Theano Development Team. „Theano: A Python framework for fast computation of mathematical expressions“. In: *arXiv e-prints* abs/1605.02688 (2016). URL: `http://arxiv.org/abs/1605.02688`.

[TK06] Grigorios Tsoumakas and Ioannis Katakis. „Multi-label classification: An overview“. In: *Dept. of Informatics, Aristotle University of Thessaloniki, Greece* (2006).

[Wer90] Paul J. Werbos. „Backpropagation through time: what it does and how to do it“. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[Ye10] Qing Ye. „A robust method for counting people in complex indoor spaces“. In: *2010 2nd International Conference on Education Technology and Computer*. 2010 2nd International Conference on Education Technology and Computer. Vol. 2. June 2010, pp. V2–450–V2–454.

[ZWB08] H. Zheng, H. Wang, and N. Black. „Human Activity Detection in Smart Home Environment with Self-Adaptive Neural Networks“. In: *IEEE International Conference on Networking, Sensing and Control, 2008. ICNSC 2008*. Apr. 2008, pp. 1505–1510.