MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Time Series Prediction Using Neural Networks

BACHELOR THESIS

Karol Kuna

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Karol Kuna

Advisor: doc. RNDr. Tomáš Brázdil, Ph.D.

Acknowledgement

I would like to thank my advisor, Tomáš Brázdil for all the help and guidance he has provided.

Abstract

This thesis compares existing methods for predicting time series in real time using neural networks. Focus is put on recurrent neural networks (RNNs) and online learning algorithms, such as Real-Time Recurrent Learning and truncated Backpropagation Through Time. In addition to the standard Elman's RNN architecture, Clockwork-RNN is examined. Methods are compared in terms of prediction accuracy and computation time, which is critical in real-time applications. Part of the work is experimental implementation of the tested models and working applications in robotics and network traffic monitoring.

Keywords

time series, prediction, neural network, recurrent network, backpropagation, backpropagation through time, real-time recurrent learning, clockwork recurrent network

Contents

List of Tables

List of Figures

1 Introduction

Each year more and more data is collected from high velocity data streams. Machine learning algorithms are used to analyse and create models of the data. Patterns found in the data structure are then exploited to make predictions about the future which can be used to guide decision making. In this thesis, I compare various artificial neural network (ANN) architectures and learning algorithms for online prediction of time series.

One of the most popular machine learning models are ANNs which are capable of approximating unknown functions and processes and thus are good candidates for use in prediction. Inspired by biological neural networks, ANNs consist of artificial neurons wired together to form a network.

Common neural network architectures, such as time delay neural networks (TDNNs) and simple recurrent networks (SRNs) trained by Truncated Backpropagation Through Time (TBPTT) and Real-Time Recurrent Learning (RTRL), as well as latest Clockwork RNN (CW-RNN) trained by TBPTT are evaluated in this thesis. Algorithms are compared in terms of prediction accuracy as well as computation time and the trade-off between them.

Time series are sequences of data points in time, usually created by measuring output of some process in discrete time intervals. The goal of prediction is to successfully estimate output of the process in next time step or several steps. It is assumed that the process is at least partially observable and to some extent, future values can be determined by observing past values. Prediction then reduces to problem of process approximation.

Since value of stored data decreases over time, effort is put into gaining insight from data as they come. Many applications don't require storing data at all, or storing it would be impractical, therefore it is advantageous to create models and update them with every data point. Continuous online learning is well suited for such tasks and can adapt to changing environments without human intervention.

Quality and speed of prediction is tested in one synthetic test and two application scenarios. The first scenario is robotic arm simulator implemented in 3D physics library BEPUPhysics. Manipulator with three actuated joints is created, the goal is to predict future position of robot's arm. The prediction network should act as a virtual model of the body, which can be useful in control systems.

The second scenario is monitoring utilisation of computer resources such as processor, memory, disk, and network usage. Neural networks should constantly predict future volume of incoming and outgoing network traffic and detect unexpected events called anomalies. When an unexpected computer network event occurs, it needs to be logged for an administrator to examine.

2 Prediction

2.1 Time Series

Time series are sequences of data points measured over time. In this thesis, data points are real-valued vectors implemented as arrays of floating point numbers.

Data sequence is created by measuring output of a process at discrete, regular time intervals. Process may also receive input in every time step, which affects its future behaviour, or it may be purely generative receiving no input at all. For generalisation's sake, let's assume every process receives input which is a real-valued vector just like output. Generative processes then simply receive input vector of zero length. In every time step, process receives input, updates

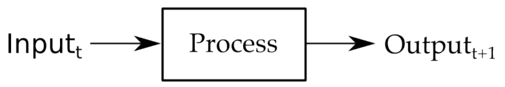


Figure 2.1: Schema of a process.

its internal state and produces output. Internal state of the process is usually hidden and can be observed only partially from output of the process.

2.2 Prediction

The goal of prediction is to approximate the process as closely as possible and hence minimise forecast error, i.e. difference between actual and forecasted value [5].

Naive approaches include methods like averaging past data points, returning previous data point or linear extrapolation. While these methods may suffice for some very simple time series, more sophisticated methods are required to cope with real-world time series. Sophisticated methods use historical data to estimate future value by finding a model of the process that empirically fits past data.

The problem of prediction can be alternatively viewed as problem of function approximation.

$$(State_{t+1}, Output_{t+1}) = Process(State_t, Input_t)$$

Process is a function from current internal state and input to next internal state and output. Unfortunately, internal state is unknown, so next output can only be estimated from history of inputs and outputs. History can be defined as an ordered set of past inputs and outputs.

$$History_t = ((Input_t, Output_t), \dots, (Input_0, Output_0))$$

Prediction of the output in next time step is then a function from current history to next output.

$$Output_{t+1} \approx Prediction_1(History_t)$$

2.3 Prediction Horizon

Some applications require estimating output of the process more than one step into the future. Prediction horizon is the number of time steps between current data point and the future predicted data point. Since future output of the process depends on inputs that will be fed in, these inputs have to be known beforehand. The modified *Prediction* function of the process output *h* steps in future is then

$$Output_{t+h} \approx Prediction_h(History_t, (Input_{t+1}, \dots, Input_{t+h}))$$

An alternative method for predicting multiple steps of the future is using one step prediction to estimate future process output, adding it to history as if it was actually measured together with the appropriate future input, and repeating this until prediction horizon is reached.

3 Artificial Neural Networks

Inspired by biological neural networks, ANNs are groups of elementary processing units called artificial neurons connected together to form a directed graph [6]. Nodes of the graph represent biological neurons and connections between them represent synapses. Unlike in biological neural networks, connections between artificial neurons aren't usually added or removed after the network was created. Instead, connections are weighted and the weights are adapted by learning algorithm.

Input signal propagates through the network in the direction of connections until it reaches output of the network. In supervised learning, learning algorithm adapts the weights in order to minimise the difference between output of the network and desired output provided by teacher.

3.1 Artificial Neuron

The complex behaviour of biological neurons was simplified to create a mathematical model of artificial neurons, also called units.

Unit receives its inputs via input connections from other units' outputs, called activations. Then it calculates a weighted sum of the inputs, called potential. Finally, unit's activation is computed from the potential and sent to other units.

Weights of connections between units are stored in a matrix w, where w_{ij} denotes weight of the connection from unit i to unit j. Every unit j has a potential p_j which is calculated as weighted sum of all of its N input units and bias.

$$p_j = \sum_{i=1}^{N+1} w_{ij} a_i$$

Bias term, also known as threshold unit, is usually represented as an extra input unit whose activation always equals one, therefore $a_{N+1} = 1$. Presence of bias term enables shifting the activation function along x-axis by changing the weight of connection from threshold unit.

Activation of the unit a_j is then computed by transforming its potential p_j by a non-linear activation function *act*.

$$a_j = act\left(p_j\right)$$

Commonly used non-linear activation function ranging from 0 to 1 is sigmoid function thanks to its easily computable derivative which is used by learning algorithms.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$
$$\frac{d\sigma(x)}{dx} = \sigma(x)\left(1-\sigma(x)\right)$$

3.2 Feedforward Neural Networks

Feedforward neural networks are a subset of ANNs whose nodes form an acyclic graph where information moves only in one direction, from input to output.

Multilayer perceptron (MLP) is a class of feedforward networks consisting of three or more layers of units. Layer is a group of units receiving connections from the same units. Units inside a layer are not connected to each other.

MLP consists of three types of layers: input layer, one or more hidden layers and output layer. Input layer is the first layer of network and it receives no connections from other units, but instead holds network's input vector as activation of its units. Input layer is fully connected to first hidden layer. Hidden layer i is then fully connected to hidden layer $i+1$. Last hidden layer is fully connected to output layer. Activation of output units is considered to be output of the network.

The output of the network is calculated in a process called forward propagation in three steps :

1. Network's input is copied to activations of input units
2. Hidden layers compute their activations in topological order
3. Output layer computes its activation and copies it to network's output

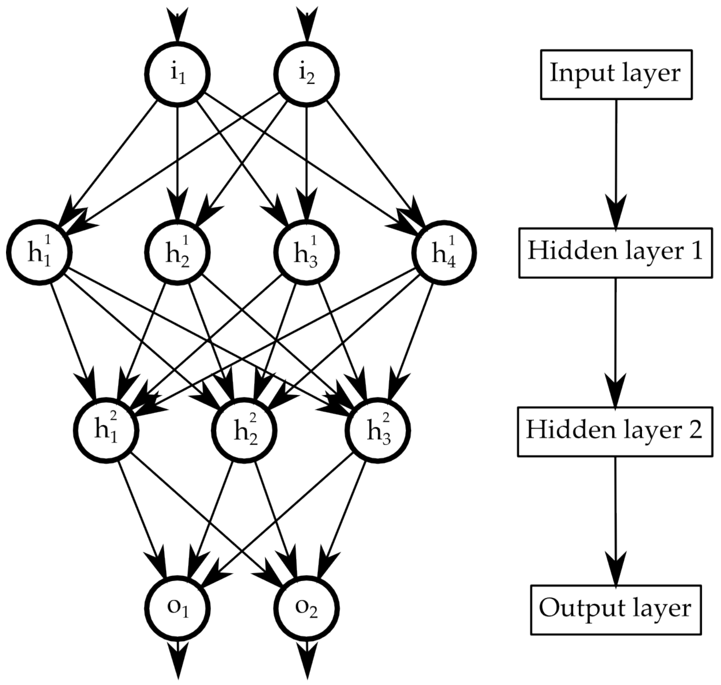


Figure 3.1: On the left, MLP consisting of input layer with two units, two hidden layers with four and three units respectively, and output layer with two units. Schematic diagram of the MLP's layers on the right.

MLPs are often used to approximate unknown functions from their inputs to outputs. MLP's capability of approximating any continuous function with support in the unit hypercube with only single hidden layer and sigmoid activation function was first proved by George Cybenko [4].

3.2.1 Backpropagation

Backpropagation, or backward propagation of errors, is the most used supervised learning algorithm for adapting connection weights of feedforward ANNs. Weights of the network are tuned so as to minimise square error

$$E = \frac{1}{2}\sum_{i=1}^{N}(target_i - output_i)^2$$

where *target* denotes desired output provided by teacher and *output* is network's prediction of the output for the corresponding input, both of size N.

Considering error E as a function of network's weights w, backpropagation can be seen as optimisation problem and standard gradient descent method can be applied. Local minimum is approached by changing weights along the direction of negative error gradient

$$-\frac{\partial E}{\partial w}$$

by weight change Δw_{ij} proportionally to α, which is constant positive value called learning rate. Fraction of previous weight change called momentum rate β can be added to the current weight change, which often speeds up learning process [12].

$$new\ \Delta w_{ij} = \beta \Delta w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}}$$

$$new\ w_{ij} = w_{ij} + \Delta w_{ij}$$

The central part of the algorithm is finding the error gradient. Let there be an MLP with L layers in topological order, first being input and last being output layer. Layer k has U_k units and holds a weight matrix w_{ij}^k representing weights of connections from unit i in layer $k-1$ to unit j in layer k. Input layer has no incoming connections. The computation can be then divided into three steps:

1. **Forward propagation**. Input vector is copied to activations a_i^1 of input layer units i. For every hidden or output layer k in topological order, compute for every unit i its potential (weighted input) p_i^k and activation a_i^k.

2. **Backward propagation**. Compute Δ_i^L i.e. the derivative of error E w.r.t. activation a_i^L of output layer unit i as

$$\Delta_i^L = \left(target_i - a_i^L\right) \frac{\partial act\left(p_i^L\right)}{\partial p_i^L}$$

For hidden layer h in reverse topological order starting from last hidden layer $h = L - 1$ down to first input layer $h = 2$ and its units i compute error term as

$$\Delta_i^h = \sum_{j=1}^{U_{h+1}} \Delta_j^{h+1} w_{ij}^{h+1} \frac{\partial act\left(p_i^h\right)}{\partial p_i^h}$$

3. **Weights update**. Change weights in layer k according to

$$new\ \Delta w_{ij}^k = \beta \Delta w_{ij}^k + \alpha \Delta_i^{k+1} a_j^k$$

$$new\ w_{ij}^k = w_{ij}^k + \Delta w_{ij}^k$$

3.2.2 Time-Delay Neural Networks

Time-Delay Neural Network (TDNN) is a modification of feedforward network designed to capture dynamics of modelled process [13]. As FFNs have no internal memory to store information about past, they are insufficient for processing temporal sequences. To overcome this, memory of past is introduced by means of extending network's input with sliding window of previous inputs, also known as tapped delay line. The information about the past is thus stored in the network input itself. Because there are no modifications to the network topology, standard backpropagation algorithm can be used.

Size of the sliding window determines how many past inputs are stored and consequently how much past data can be correlated to future output. Too small window may not capture necessary dynamics of the system, because network is blind to anything that happened before, and thus create inferior results. Whereas too large window can drastically prolong learning time.

To generalise TDNN to processes that also receive input, prediction of process output in next time step can be defined as approximation of function $Prediction_{TDNN}$ with window size w

$$Output_{t+1} \approx Prediction_{TDNN}((Input_t, Output_t), \dots, (Input_{t-w+1}, Output_{t-w+1}))$$

3.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a subclass of ANNs which allow units to form a cyclic graph [14]. This allows the network to store an internal state and consequently process sequences of inputs and thus perform temporal tasks.

Since RNNs accumulate previous activity in the activation of units that form directed cycles, there is no need to feed the network with history of previous inputs and outputs like in TDNN. Rather than approximating a function, RNNs try to model a process. Therefore, prediction of future process output can be described as

$$Output_{t+1} \approx Prediction_{RNN}\left(NetworkState, Input_t, Output_t\right)$$

where *NetworkState* is activity of units that form directed cycles.

3.3.1 Elman's Simple Recurrent Network

One of the simplest and most popular RNN architectures is Elman's simple recurrent network (SRN). SRN resembles a three-layer feed-forward network due to its structure composed of input, hidden and output layer, with addition of a context layer. Input and context layer project to hidden layer, which projects to output layer. Context layer is a copy of hidden layer's activation in previous time step and thus acts as network's memory of previous activity.

Recurrent network's output in time t is computed in four steps:

1. Input vector is copied to activation of input layer units
2. Activation of hidden layer units in time $t-1$ is copied to current activation of context layer units
3. Hidden layer units compute their activations
4. Output layer units compute their activations and copy them to network's output

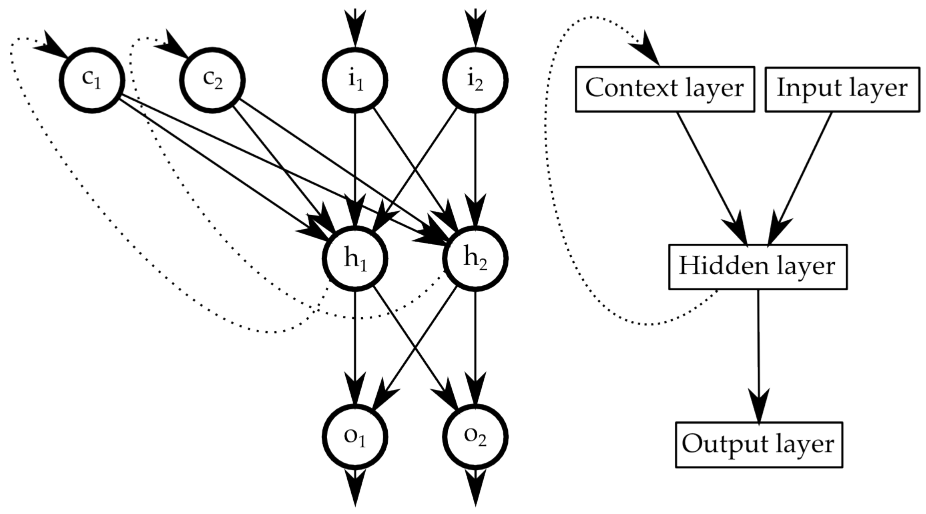


Figure 3.2: On the left, Elman's recurrent network with input, hidden, context, and output layer, each containing two units. On the right, schema of layers in Elman network. Dotted link signifies copying activity of source units to target units.

3.3.2 Backpropagation Through Time

Standard backpropagation algorithm is not suited for networks with cycles in them. Fortunately, RNN can be modified to look like a feedforward network by unfolding the network in time as shown in figure 3.3 and then trained by Backpropagation Through Time (BPTT) algorithm first laid out by Rumelhart, Hinton and Williams [11].

The network unfolding process begins with a SRN in current time step t, denoted as SRN_t. Since context layer of a SRN is just a copy of hidden layer activation from previous step, cycles in the network can be avoided by replacing context layer with an identical copy of the SRN network from previous step, SRN_{t-1}. Hidden layer of SRN_{t-1} is then connected to hidden layer of SRN_t. This procedure is repeated until time step 0 is reached, in which case the context layer is not replaced, but rather stays set to its initial activity. The number of SRN copies represents depth of the unfolded network and each copy of the network uses the same set of connection weights.

Once the SRN has been unfolded into a feedforward network, backpropagation can be used. The algorithm again consists of 3 steps:

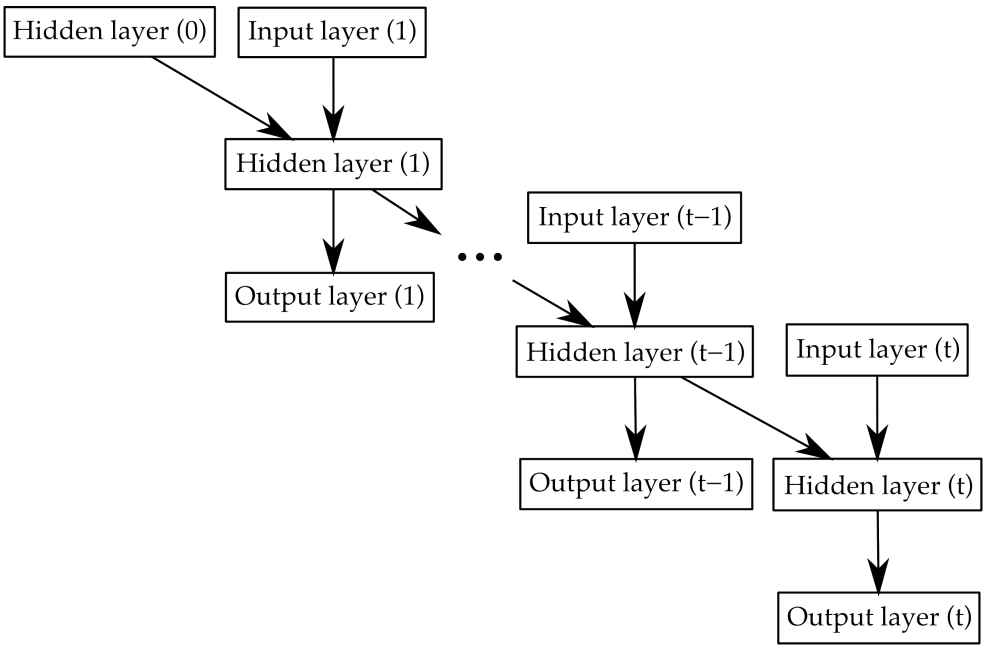


Figure 3.3: Elman's recurrent network unfolded in time.

1. **Forward propagation**. Signal is propagated through the unfolded network in the standard fashion, from top to bottom. In this case, from the SRN copy furthest in the past to the most recent copy.

2. **Backward propagation**. Error term $\Delta_i^{out}(t)$ of output layer unit i from SRN copy in time t is computed as

$$\Delta_i^{out}(t) = \left(target_i(t) - a_i^{out}(t)\right) \frac{\partial act\left(a_i^{out}(t)\right)}{\partial a_i^{out}(t)}$$

where $target_i(t)$ is desired output of output layer unit i of the SRN in time t and $a_i^{out}(t)$ is its actual activation in time t.

For every hidden layer unit i of unfolded SRN in time step t, let j iterate over all units that receive connections from unit i according to unfolded network topology and let Δ_j be their error terms. Weight matrix w_{ij}^h holds weights of connections from unit i to unit j. Error term of hidden unit i denoted as $\Delta_i^{hid}(t)$ is then computed as

$$\Delta_i^{hid}(t) = \left(\sum_j \Delta_j w_{in}^h \right) \frac{\partial act\left(a_i^{hid}(t)\right)}{\partial a_i^{hid}(t)}$$

3. **Weights update** in the original SRN. For every unit j in hidden or output layer l of the original network that receives connections from units i in layer k, update the weight according to

$$new\ w_{ij}^l = w_{ij}^l + \alpha \sum_{\tau=0}^{t} \Delta_i^k(\tau)\, a_j^l(\tau)$$

where w_{ij}^l is weight matrix of connections from unit i to unit j in layer l, $\Delta_i^k(\tau)$ is error term of unit i in layer k of unfolded network copy in time step τ, $a_j^l(\tau)$ is activation of unit j in layer l of unfolded network copy in time τ and α is learning rate. One can also add momentum rate analogously to backpropagation described in section 3.2.1.

3.3.3 Truncated Backpropagation Through Time

In BPTT, Number of SRN copies in the unfolded network is equal to current time step t. Should this algorithm be used in online manner, it would be impractical, since its memory footprint grows linearly with time. To overcome this, online version of the BPTT algorithm called Truncated Backpropagation Through Time (TBTT) can be used [14].

TBPTT works analogously to BPTT, except the maximum depth d of the unfolded network is limited. Networks are unfolded from current time step t up until $t - d + 1$ when unfolding stops. Context layer of the SRN copy in time $t - d - 1$ holds activation of network's hidden layer in time $t - d$.

TBPTT typically performs backward pass in the unfolded network either every d time steps or at each time step. The latter is examined in this thesis. Since weights of connections are changed every time step, it is appropriate to either store past weights or use current weights but perform a forward propagation pass in the unfolded network.

3.3.4 Real-Time Recurrent Learning

Real-Time Recurrent Learning (RTRL) algorithm is a gradient descent method suitable for online learning of recurrent networks [15].

Let there be a SRN whose each unit i has in time step t potential $p_i(t)$ and activation $a_i(t)$. For convenience, let U_{in}, U_{hid} and U_{out} be sets of indices of input, hidden and output units respectively. Weight of connection from unit i to unit j is denoted by w_{ij}. In time step t, let $target_i(t)$ be the teacher given desired activation of output unit i.

In time step t, instantaneous error $E(t)$ is

$$E(t) = \frac{1}{2} \sum_{i \in U_{out}} (target_i(t) - a_i(t))^2$$

and the minimised total error $E_{total}(t)$ in time t is

$$E_{total}(t) = \sum_{\tau=1}^{t} E(\tau)$$

The error is minimised by adjusting weights along the negative gradient of error $-\partial E_{total}(t)/\partial w_{ij}$. This can be computed by accumulating negative gradient of instant error in each time step

$$-\frac{\partial E(t)}{\partial w_{ij}} = \sum_{i \in U_{out}} (target_i(t) - a_i(t)) \frac{\partial a_i(t)}{\partial w_{ij}}$$

$\partial a_i(t)/\partial w_{ij}$ can be computed by differentiating the network dynamics equation, resulting in the derivative $v_{ij}^k(t)$ of hidden or output[1] unit k activation w.r.t. every weight w_{ij} in time t

$$v_{ij}^k(t+1) = \frac{\partial a_k(t+1)}{\partial w_{ij}} = act'(p_k(t)) \left[\left(\sum_{l \in U_{hid} \cup U_{out}} w_{lk} v_{ij}^l(t) \right) + \delta_{ki} a_j(t) \right]$$

where δ_{ki} is Kronecker's delta

$$\delta_{ki} = \begin{cases} 1, & \text{if } k = i, \\ 0, & \text{if } k \neq i. \end{cases}$$

1. Activations of input units do not depend on networks' weights.

This creates dynamical system with variables v^k_{ij} for all hidden and output units. Since the initial state of the network is independent from its weights, initial value can be set $v^k_{ij}(0) = 0$. Network's weights are then updated along the accumulated trajectory

$$new\ w_{ij} = w_{ij} - \alpha \sum_{k \in U_{out}} \left[(a_k(t) - target_k(t))\, v^k_{ij} \right]$$

One can also add momentum rate analogously to backpropagation described in section 3.2.1.

RTRL algorithm is computationally expensive [8], because calculating $v^k_{ij}(t+1)$ requires $O(n^4)$ operations, where n is number of hidden and output units. This limits its usage to very small networks.

3.4 Clockwork Recurrent Network

SRNs have trouble capturing capturing long-term dependencies in input sequences due to vanishing gradient problem [7]. Clockwork recurrent neural network (CW-RNN) is a modification of Elman's SRN architecture designed to solve this problem by having hidden layer split into M modules running at different clocks [10]. Each module i is assigned a clock rate T_i. In time step t only modules with clock rate that satisfies $(t \ mod \ T_i) = 0$ compute their activations, other modules retain their previous activation.

Like in SRN, input layer is connected to hidden layer, context layer stores activation of hidden layer from previous time step and hidden layer is connected to output layer. The difference is that module of hidden layer with clock rate T_i connects to module in context layer with period T_j only if $T_i <= T_j$ as shown in figure 3.4. This allows slower modules to focus on long-term dependencies in the input sequence, while faster modules focus on short-term information with context provided by slower modules.

To adapt network weights, BPTT learning algorithm can be used. The algorithm works analogously with the only difference compared to SRN being that error propagates backwards only from active modules executed at time t. Error term of inactive modules is retained from previous step.

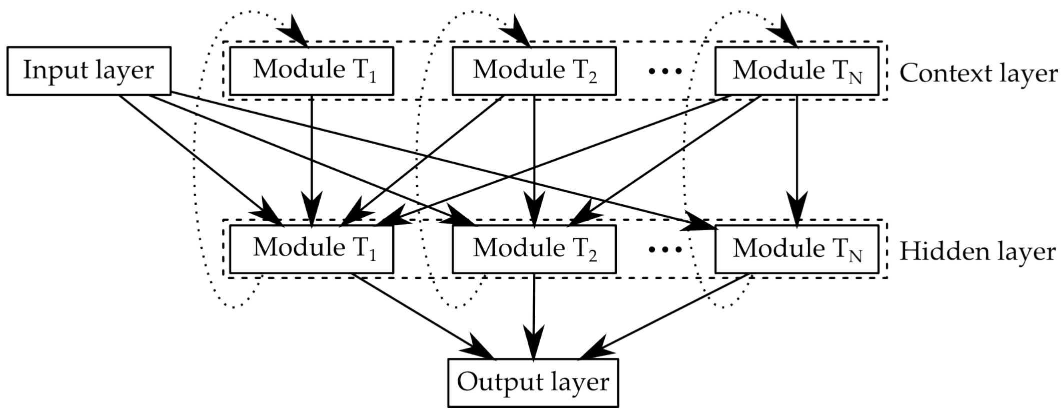


Figure 3.4: Clockwork-RNN.

Since a CW-RNN[2] with the same number of units as a SRN has fewer connections and activations of hidden units are computed less frequently, CW-RNN should run faster than SRN with the same number of either weights or units.

2. CW-RNN with at least two modules running at different clock rates.

4 Implementation

4.1 Neural Prediction Library

As a part of this thesis, Neural Prediction Library (NPL) is built for experimenting with various neural network models in online time series prediction tasks. NPL supports all tested neural network architectures as well as learning algorithms. In addition to that, the tool allows easy manipulation with data points and datasets. NPL is written in C++ programming language and is structured into classes dealing with data manipulation and neural networks.

4.1.1 Data Manipulation

Data point in NPL is represented using a **MemoryBlock** object, which encapsulates an array of floating point numbers and provides convenience methods and arithmetic vector operations.

DataSet class enables offline training of the networks by reading data points from a file, where each data point is on a separate line and its values are separated by spaces.

DstatParser class allows reading raw output produced by Dstat tool and converting it to data point format suitable for prediction.

4.1.2 Neural Networks

All implemented neural networks provide the same interface described by abstract class **NeuralNetwork** consisting of forward propagation of input and retrieval of output.

Neural networks are composed of layers implemented by **NeuralLayer** class which holds activations of its units and weights of incoming connection and on top of that supports forward propagation of input signal and backward propagation of errors using standard backpropagation algorithm.

FeedforwardNetwork implements an MLP with variable number of hidden layers and units in them. The topology of the network can be further modified beyond MLP if required.

CWRecurrentNetwork implements a CW-RNN with a variable number of hidden layer modules with specified clock rates.

SimpleRecurrentNetwork derives from CWRecurrentNetwork to implement a SRN, since Elman's SRN is just a special case of CW-RNN network with single hidden layer module with clock rate equal one.

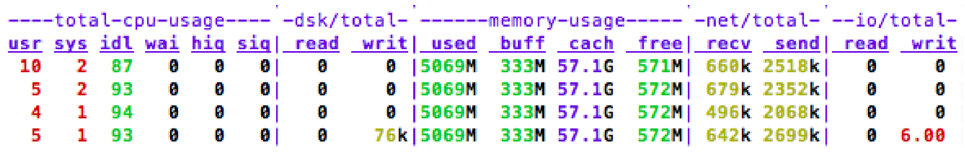
4.1.3 Learning Algorithms

LearningAlgorithm is an abstract class providing common interface for all learning algorithms prescribing a method to train the network with teacher given target and reading current square error.

Backpropagation, TBPTT, RTRL classes are descendants of LearningAlgorithm implementing respectively named algorithms. Backpropagation class is used for training a FeedforwardNetwork, RTRL for training a SimpleRecurrentNetwork. TBPTT supports training both SimpleRecurrentNetwork and CWRecurrentNetwork.

4.2 Network Monitor

Network monitor is an application built on top of NPL that works with assistance of UNIX system resources monitoring tool Dstat [3]. Dstat provides real-time measurements of system resources and displays them in a format of table that grows by one row each second. Columns of this table represent various measured variables and can be specified with command line options. This interval between measurements can be modified with command line options as well.



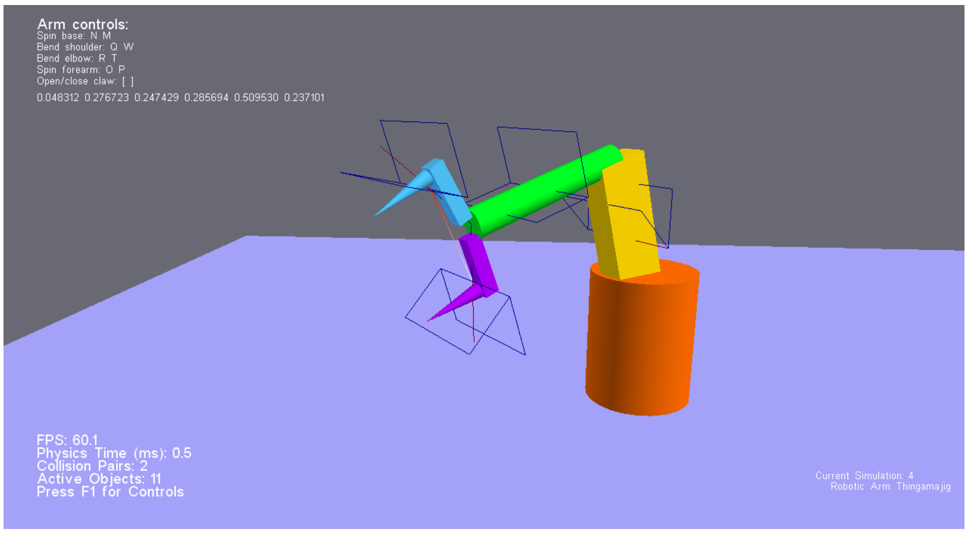
```
----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- --io/total-
usr sys idl wai hiq siq| read  writ| used  buff  cach  free| recv  send| read  writ
 10   2  87   0   0   0|   0     0 |5069M  333M 57.1G  571M| 660k 2518k|   0     0
  5   2  93   0   0   0|   0     0 |5069M  333M 57.1G  572M| 679k 2352k|   0     0
  4   1  94   0   0   0|   0     0 |5069M  333M 57.1G  572M| 496k 2068k|   0     0
  5   1  93   0   0   0|   0   76k|5069M  333M 57.1G  572M| 642k 2699k|   0  6.00
```

Figure 4.1: Sample output from Dstat.

Output of Dstat is redirected to the network monitoring program. Raw Dstat output is automatically parsed and preprocessed to serve as input for a neural network of choice. As a proof of concept, the network is constantly predicting incoming and outgoing network traffic and training to improve its precision. Should the difference between prediction and actual value exceed predefined threshold, this incident is logged to a file as an anomaly. Please refer to the *README.txt* file for instructions on how to launch and use the network monitor.

4.3 Robotic Arm Simulator

Robotic arm simulator application is implemented in BEPUPhysics 3D physics library [1]. Manipulator with three degrees of freedom from BEPU robotic arm demo [2] is constructed in 3D world. Three controlled joints are situated in base, "shoulder" and "elbow" of the arm and rotate in one axis.


```
Arm controls:
Spin base: N M
Bend shoulder: Q W
Bend elbow: R T
Spin forearm: O P
Open/close claw: [ ]
0.048312 0.276723 0.247429 0.285694 0.509530 0.237101

FPS: 60.1
Physics Time (ms): 0.5
Collision Pairs: 2
Active Objects: 11
Press F1 for Controls

Current Simulation: 4
    Robotic Arm Thingamajig
```

Figure 4.2: Robotic arm in BEPUPhysics 3D physics simulator.

Simulator renders the scene, calculates physics and supplies information about current rotation of joints as well as position of claw to a neural network of choice, which provides prediction of claw position in next time step. The prediction network thus creates a forward model of the process, in this case a robotic arm, which can find its use in control system, such as Internal Model Control [9].

NPL is linked and integrated with the robotic arm simulator. As a proof of concept, line showing direction from current position of claw to the predicted position is drawn. For instructions on how to launch and use the robotic simulator, please refer to the *README.txt* file.

5 Experiments

In the following experiments, I attempt to find the best performing network and its hyper parameters for prediction in three different scenarios with restricted computation time.

Networks are tested on the same prepared datasets to ensure equal testing conditions. Units of all networks use logistic activation function. Weights of connections in all networks are initialised randomly with uniform distribution on interval [-0.25, 0.25]. Since the initial setting of network influences its overall performance, all experiments described below are run ten times and only the average of them is presented as the result.

Experiments measuring computation time are run on 1.7GHz Intel Core i5 2557M. While the absolute computation times should vary depending on system configuration, I expect them to retain their ratio relative to each other. Computation time is measured from the moment just before first data point is processed until the last one is processed.

As a measure of prediction accuracy, total error E_{total} collected over T time steps is used. Network that predicts entire sequence flawlessly would have zero total error.

$$E_{total} = \sum_{\tau=1}^{T} \frac{1}{2} \sum_{i=1}^{N} (Target_i - Output_i)^2$$

Networks are trained online in every time step. In order to capture both their accuracy and ability to adapt quickly, there is no traditional division into training and testing period. Instead, networks are evaluated online as they learn in all time steps.

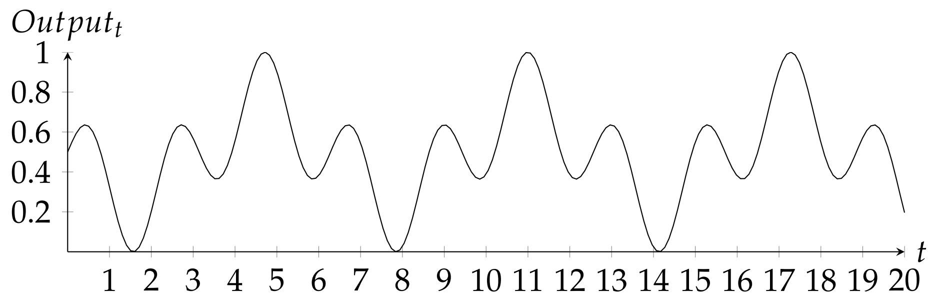
5.1 Testing Scenarios

Networks are tested on three different datasets with varying complexity collected from three scenarios. Each scenario is designed to test a specific aspect of prediction, such as modelling generative processes, noisy network traffic or substantially time constrained robotic manipulator.

5.1.1 Goniometric function

The first simple testing scenario is predicting next step of a generative process with no inputs whose output vector of length one is defined in time t as

$$Output_t = \frac{1 + sin(t)cos(2t)}{2}$$



The goniometric function's range is [0,1] which matches logistic activation function's range [0,1], therefore no preprocessing is needed. This function is periodical with period of 2π.

In time step t, a TDNN with sliding window of size w receives input consisting of $Output_\tau$ in time steps τ from $t - w + 1$ to t and is tasked to predict $Output_{t+1}$. Similarly, RNNs are tasked to predict $Output_{t+1}$, but they only receive $Output_t$ on their inputs.

5.1.2 Network Usage

Second testing scenario is predicting computer network usage measured by monitoring tool Dstat. Each second, Dstat measures usage of system resources such as CPU load, memory usage, disk throughput, number of open sockets as well as network incoming and outgoing traffic. Prediction has to be computed in less time than it takes Dstat to collect network usage, which poses a loose time constraint of computation time being less than one second. Dataset consists of 3,600 data points created by monitoring real application server for one hour with enabled cpu, disk, memory, network, I/O and socket stats command line options.

Dstat measures 26 variables of the monitored system, two of which are incoming and outgoing network traffic. Values of these two variables in next time step shall be predicted. Since the values observed by Dstat have huge dynamic range, preprocessing is necessary. Values are linearly squashed into interval [0, 1] by dividing the observed value by the expected maximum.

In time step t, a TDNN with sliding window of size w receives input consisting of the 26 measured variables in time steps τ from $t - w + 1$ to t and is tasked to predict two variables — incoming and outgoing traffic in time step$t + 1$. Similarly, RNNs are tasked to predict the same two values, but they only receive the 26 measured variables in time step t.

5.1.3 Robotic Arm

Third testing scenario is predicting position of robotic arm's claw in BEPUPhysics 3D physics library. Manipulator with three controlled joints is constructed. At each simulation step, network receives rotations of all three joints and their respective control signals. The goal is to predict next X, Y and Z position of the claw in next time step. The data points are squashed into interval [0,1] by the simulator so they don't require preprocessing.

This task is heavily time constrained. The 3D simulator is required to run at 60 frames per second for smooth control, which leaves only $\frac{1}{60}$ of a second for computing the prediction. Dataset consisting of 18,000 data points is created by controlling the robot with random commands for a period of five minutes.

In time step t, a TDNN with sliding window of size w receives input consisting of X, Y and Z position of the claw, three angles of joints and their three control signals in time steps τ from $t - w + 1$ to t and is tasked to predict claw's X, Y and Z position in time step $t + 1$. Similarly, RNNs are tasked to predict the same X, Y and Z position, but they only receive the three joint angles and their three control signals in time step t.

5.2 Tested Methods

5.2.1 Time Delay Neural Network

The role of sliding window size is examined w.r.t. prediction accuracy and computation time. TDNNs with a single hidden layer consisting of 64, 128 and 256 units are examined. Learning and momentum rates are informally chosen to the best performing value for each dataset according to the following table.

Scenario	Learning rate	Momentum rate
Goniometric function	0.05	0.9
Network usage	0.001	0.9
Robotic arm simulator	0.01	0.9

Table 5.1: Learning and momentum rates of TDNN in the experiments.

5.2.2 SRN trained by TBPTT

SRNs with 64, 128 and 256 hidden units trained using TBPTT are tested with varying unfolding depth in order to find its impact on precision and computation time.

Scenario	Learning rate	Momentum rate
Goniometric function	0.01	0.99
Network usage	0.001	0.9
Robotic arm simulator	0.01	0.9

Table 5.2: Learning and momentum rates of SRN trained by BPTT in the experiments.

5.2.3 SRN trained by RTRL

Due to RTRL's severe computational complexity, SRNs with only 8 and 16 hidden units are tested for prediction accuracy and computation time. In all scenarios, learning rate is set to 0.1 and momentum to 0.9.

5.2.4 CW-RNN trained by TBPTT

Experiments examine role of unfolding depth in CW-RNN with 64, 128 and 256 hidden units on its precision and computation time. Hidden layer of the network is divided into four equally big modules with exponential clock rates: 1, 2, 4 and 8.

Scenario	**Learning rate**	**Momentum rate**
Goniometric function	0.01	0.99
Network usage	0.001	0.9
Robotic arm simulator	0.01	0.9

Table 5.3: Learning and momentum rates of CW-RNN trained by BPTT in the experiments.

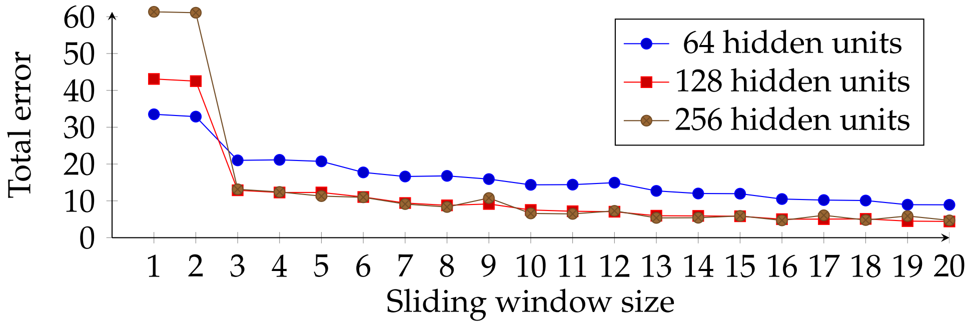
6 Results

6.1 Goniometric Function Results

6.1.1 Role of TDNN's Sliding Window Size

The results show steep decrease of error when increasing window size from two to three past time steps and mild decrease of error when extending window further. Presumably, three steps are threshold when information in the sliding window becomes sufficient to capture dynamics of the simple goniometric function. Network with only 64 hidden units performed notably worse than bigger networks. However, there is no significant gain in increasing network size beyond 128 hidden units.

Figure 6.1: Total error of predicting goniometric function with TDNN and various sliding window sizes.



6.1.2 Role of TBPTT Unfolding Depth

In both SRN and CW-RNN, increasing unfolding depth decreases total prediction error. Increasing hidden layer size helps both networks find better error minimum. In this scenario, SRN provides better results than CW-RNN with the same hidden layer size.

Figure 6.2: Total error of predicting goniometric function with SRN trained by TBPTT with various unfolding depth.

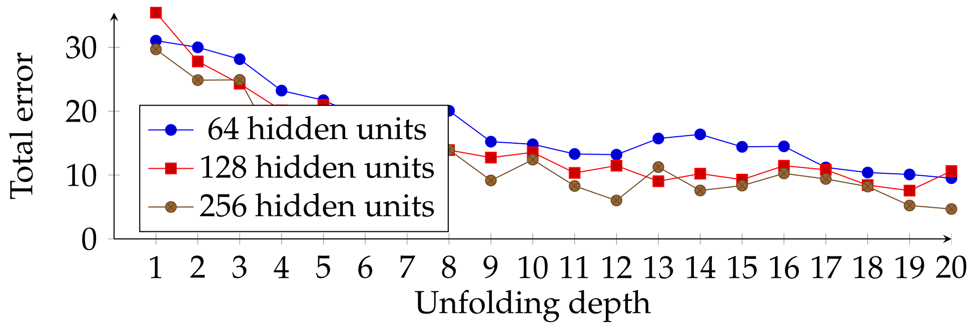
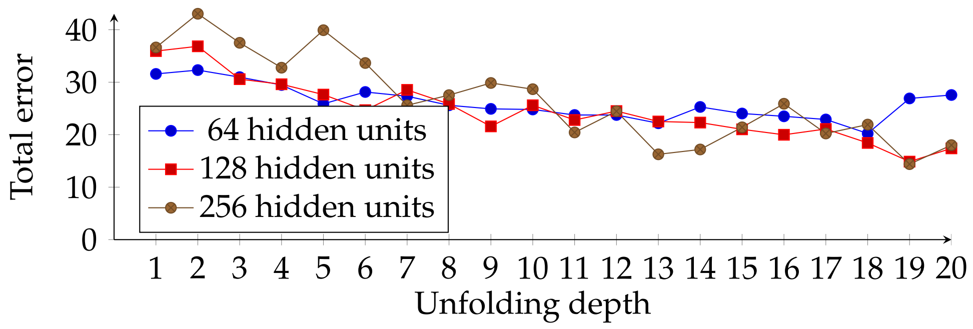


Figure 6.3: Total error of predicting goniometric function with CW-RNN trained by TBPTT with various unfolding depth.



6.1.3 Computation Time Tradeoff

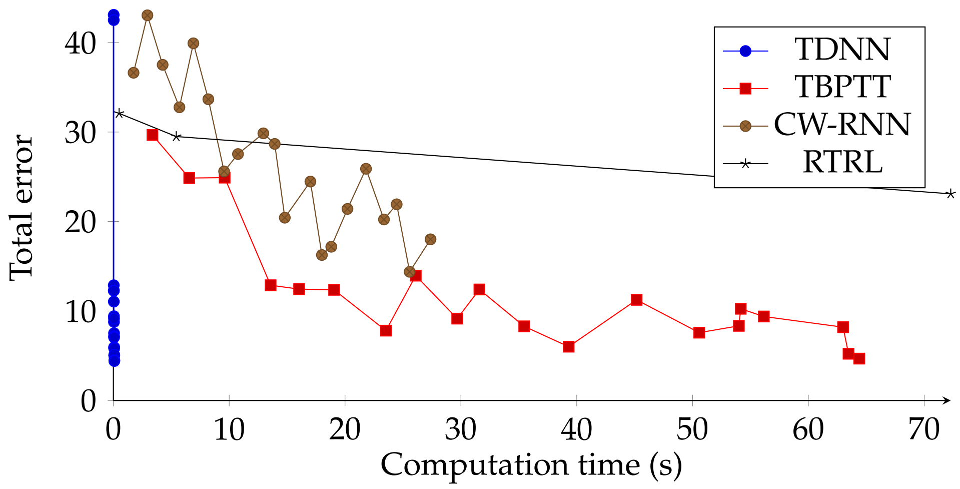
The most accurate network in predicting goniometric function was TDNN with 128 hidden units and sliding window of 20 past steps. In this simple task, TDNN shows remarkable performance while also being the least computationally extensive.

SRN trained by TBPTT achieves the second best result, though with much higher computation time. While CW-RNN requires approximately only a half of SRN's computation time with the same network size and unfolding depth, in this scenario SRN offers lower error per computation time.

SRN trained by RTRL didn't perform particularly well, primarily because of its high computation complexity which limits the maximum practical network size to 32 units.

In the following figure, all four methods are compared at how decreasing total error affects computation time. Depicted below are TDNN with 128 hidden units and various sliding window sizes, SRN and CW-RNN with 256 hidden units trained by TBPTT with various unfolding depth, and SRN trained by RTRL with 8, 16 and 32 hidden units.

Figure 6.4: Tradeoff between time and error in goniometric function scenario.



6.2 Network Usage Results

Number of hidden units, sliding window size and unfolding depth show little effect on reducing total error. This behaviour could be caused by inherent unpredictability of the network traffic or inability of the methods to find the correct model.

The best performing method was RTRL with 16 hidden units closely followed by TDNN.

Figure 6.5: Total error of predicting network traffic with TDNN and various sliding window sizes.

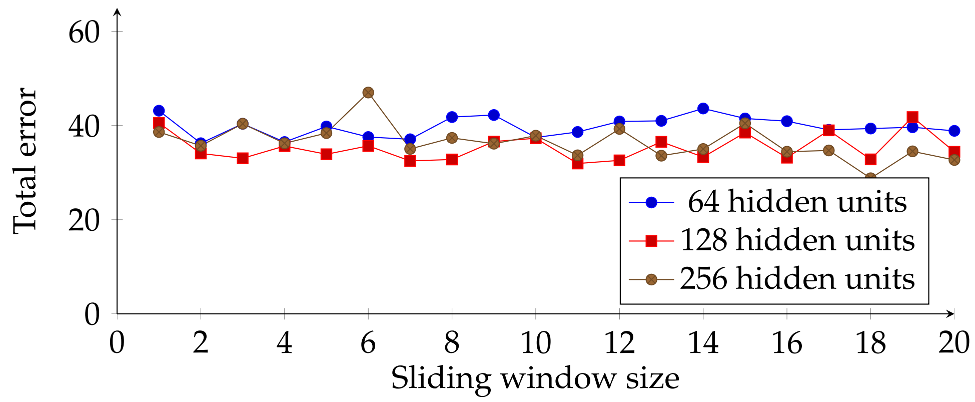


Figure 6.6: Total error of predicting network traffic with SRN trained by TBPTT with various unfolding depth.

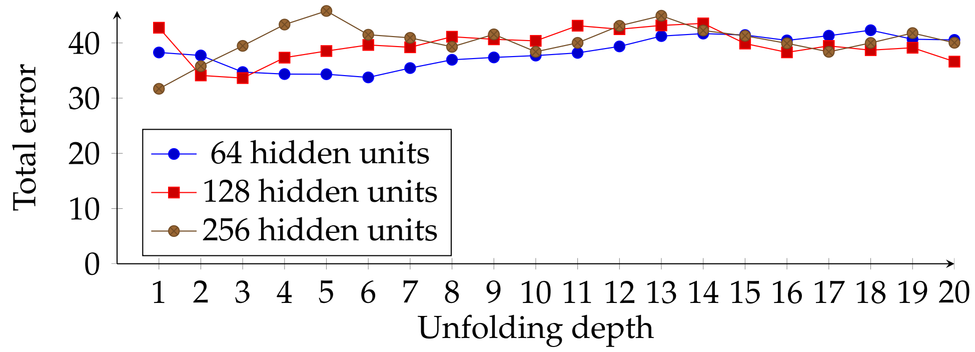
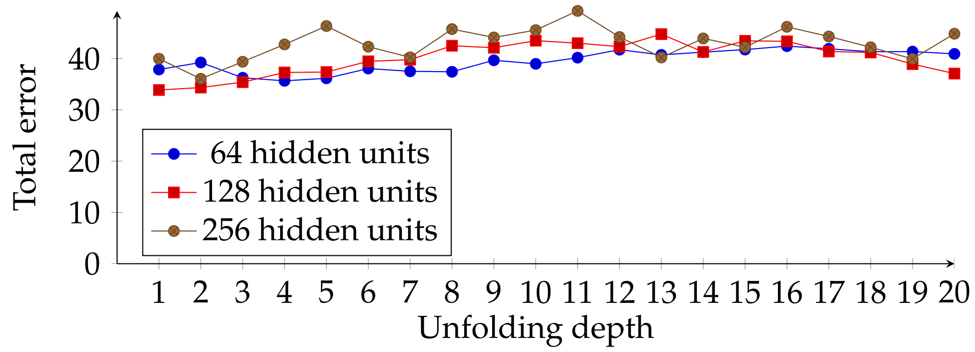


Figure 6.7: Total error of predicting network traffic with CW-RNN trained by TBPTT with various unfolding depth.

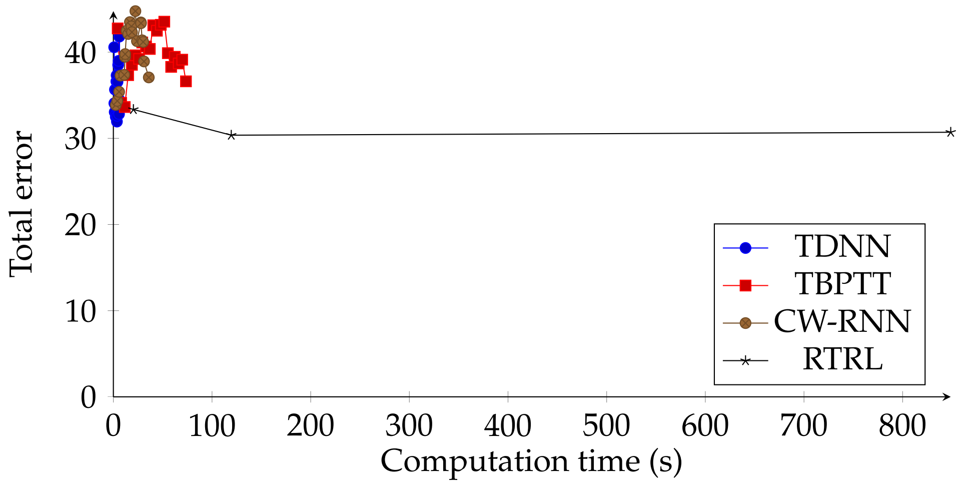


6.2.1 Computation Time Tradeoff

No tested model exceeded the maximum allowed computation time of one hour. Small networks trained by RTRL perform well and maintain viable computation time. TDNN achieved the second lowest precision while using the least computation time. Networks trained by TBPTT achieve better results with smaller unfolding depth.

In the following figure, SRN with 8, 16 and 32 hidden units trained by RTRL is compared to other models with 256 units.

Figure 6.8: Tradeoff between time and error in network usage scenario.

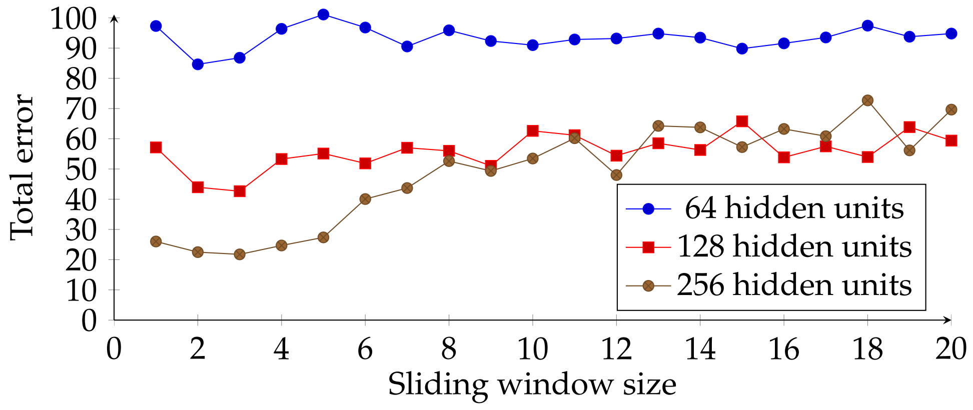


6.3 Robotic Arm Results

6.3.1 Role of TDNN's Sliding Window Size

Regardless of number of hidden units, TDNNs find their error minimum with window of size two or three. Enlarging the window further has negative impact and increases error. The larger the network, the more susceptible it is to this phenomenon. This may suggest that the state of manipulator is sufficiently captured in two or three consecutive steps. Increasing the number of hidden units is distinctly decreasing total error.

Figure 6.9: Total error of predicting manipulator's claw position with TDNN and various sliding window sizes.



6.3.2 Role of TBPTT Unfolding Depth

As unfolding depth increases, both SRN and CW-RNN show similar pattern of first sharply decreasing the total error, finding their minimum, and then error gradually rises. Networks with more hidden units perform better in this scenario.

Figure 6.10: Total error of predicting manipulator's claw position with SRN trained by TBPTT with various unfolding depth.

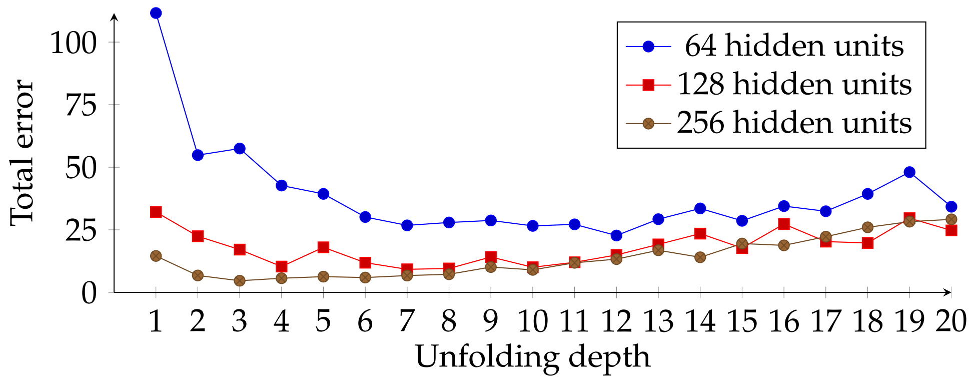
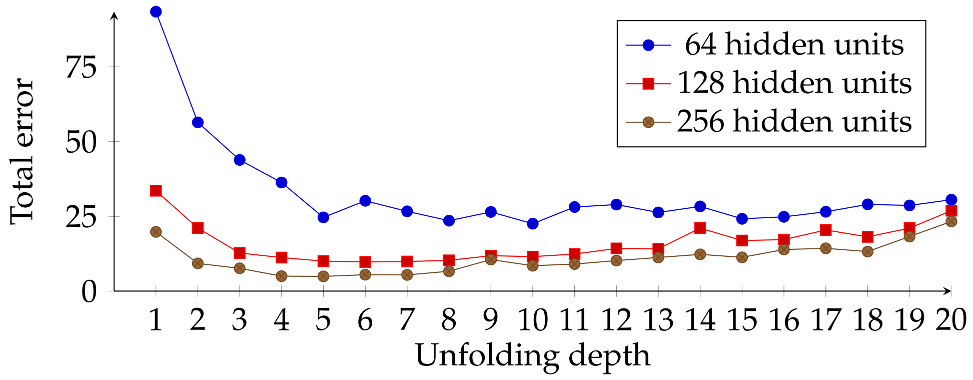


Figure 6.11: Total error of predicting manipulator's claw position with CW-RNN trained by TBPTT with various unfolding depth.



6.3.3 Computation Time Tradeoff

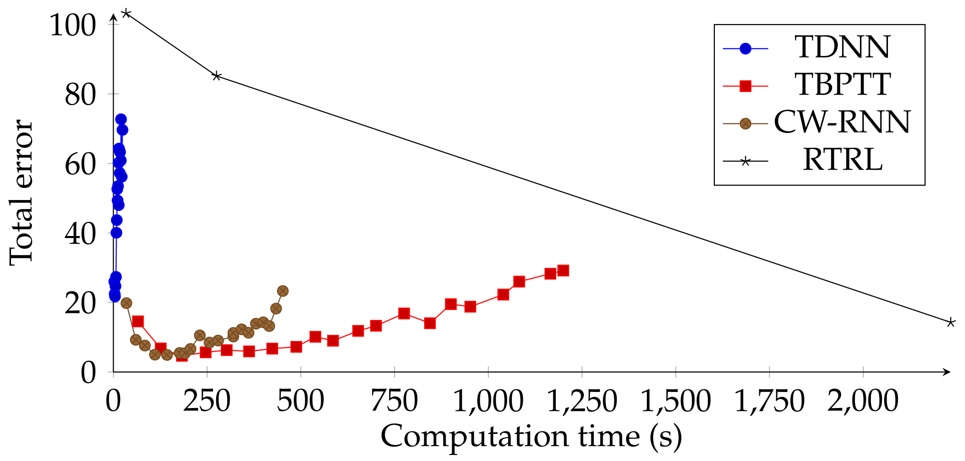
In this heavily time constrained scenario, many neural network models fail to meet the maximum computation time of 300 seconds. RTRL ends up particularly bad on this front, with its network consisting of 32 hidden units using up more than 7 times the maximum computation time. Smaller SRNs trained by RTRL that meet the requirement are not precise enough.

TDNN again shows its lightweight computation complexity and beats all networks in this regard, however it scores only third in precision.

SRN trained by TBPTT has a slight edge over CW-RNN in precision, but on the other hand, CW-RNN has a much lower computation complexity. Since both methods can fit into required maximum computation time, the final choice of the method should be based on whether one prefers increased precision or lower computation time.

In the following figure, TDNN with 256 hidden units and varying sliding window size, SRN and CW-RNN with 256 hidden units trained by TBPTT with varying unfolding depth, and SRN with 8, 16 and 32 hidden units trained by RTRL.

Figure 6.12: Tradeoff between time and error in robotic arm scenario.



7 Conclusion

This thesis compared several neural network approaches for online prediction of time series. The examined methods include TDNN, SRN trained by TBPTT and RTRL as well as CW-RNN trained by TBPTT. Role of network size and other parameters w.r.t. prediction accuracy and computation time is evaluated in one synthetic and two real-world datasets.

TDNNs have shown to be the least computationally expensive method while providing reasonably good results with appropriately sized sliding window. TBPTT shows consistently good results, albeit with higher computation time. CW-RNN slightly lags behind SRN in precision, but requires less than half of computation time with the same number of units and unfolding depth. RTRL proved to be too impractical to be included in real-world applications where smaller networks aren't sufficient due to its huge computational complexity. Methods struggled the most with predicting computer network usage, most likely due to its inherent unpredictability.

As a part of the work, a tool for experimenting with various network architectures and learning algorithms was built and incorporated into two applications. The first is a network monitoring program that detects and logs unexpected events in network traffic. The second application integrates with a robotic simulator to predict position of an arm which may later find its use in control systems.

While this thesis examines several methods of predicting time series using neural networks, there are many opportunities to extend it with comparison of other machine learning approaches in the future.

A Appendix

Source code of all tested models and experiments can be found at https://github.com/karolkuna/Time-Series-Prediction-Using-Neural-Networks

A.1 List of attachments

- NPL source code with all presented experiments
- Network monitor source code
- BEPUPhysics solution with incorporated NPL showing prediction of robot's claw
- Instructions how to launch and use the mentioned programs

Bibliography

[1] Bepuphysics. `http://www.bepuphysics.com`, May 2015.

[2] Bepuphysics demo robotic arm thingamajig. `https://bepuphysics.codeplex.com`, May 2015.

[3] Dstat: Versatile resource statistics tool. `http://dag.wiee.rs/home-made/dstat/`, May 2015.

[4] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 12 1989.

[5] R. J. Frank, N. Davey, and S. P. Hunt. Time series prediction and neural networks. *J. Intell. Robotics Syst.*, 31(1-3):91–103, May 2001.

[6] Simon S. Haykin. *Neural networks and learning machines*. Pearson, Upper Saddle River, N.J, 2009.

[7] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[8] Herbert Jaeger. A tutorial on training recurrent neural networks, covering bptt, rtrl, ekf and the "echo state network" approach. *GMD Report*, 159, 2002.

[9] Mitsuo Kawato. Internal models for motor control and trajectory planning. *Current Opinion in Neurobiology*, 9:718–727, 1999.

[10] Jan Koutník, Klaus Greff, Faustino J. Gomez, and Jürgen Schmidhuber. A clockwork RNN. *CoRR*, abs/1402.3511, 2014.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Internal Representations by Error Propagation, pages 673–695. MIT Press, Cambridge, MA, USA, 1988.

[12] Ilya Sutskever. *Training Recurrent Neural Networks*. PhD thesis, University of Toronto, 2012.

[13] Alexander Waibel, Toshiyuki Hanazawa, Geofrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Readings in speech recognition. chapter Phoneme Recognition Using Time-delay Neural Networks, pages 393–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[14] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2:490–501, 1990.

[15] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280, June 1989.