NTNU

Norwegian University of
Science and Technology

Learning event-driven time series with phased recurrent neural networks

Are Haartveit
Harald Husum

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem Description

This thesis will explore learning algorithms for time series with certain properties. The characteristics we are interested in are:

- Aperiodicity: There is no fixed, periodic sampling rate. Data points are registered when it is deemed interesting to do so.
- Asynchronicity: Data points do not reliably co-occur across channels. A reading on one channel does not mean there will be a reading on any of the other channels.
- Lengthiness: Time series that consist of many data points with long temporal relations.

These are common properties for the multi-channel event-driven signals from sensors in process industries. Finding novel machine learning techniques for data of this kind would be valuable to companies generating such data.

At the time of writing the most commonly used learning algorithms for time series (CNNs and RNNs) are not well suited for this kind of data. They expect periodic, synchronous signals as input. RNNs also have a tendency to forget previously observed information as they iterate over the time series. RNN and CNN models are still applied on data that does not fit these requirements, but usually after extensive preprocessing of raw time series to make them fit the algorithms.

The overarching goal of the thesis is to apply machine learning algorithms that have more relaxed requirements for the properties of time series, so that workarounds such as preprocessing can be avoided or be made less extensive. This approach will be contrasted with more traditional approaches in order to determine the value of the methods we apply. Performance measures like accuracy, data efficiency and time efficiency will be measured on several datasets, ranging from well-understood benchmarking problems to real sensor data from industry processes. After benchmark experiments are completed, the industry data is the main point of interest and importance, as it constitutes a point of interest for our industry partner Intelecy AS.

In order to limit the scope of the problem, we choose to experiment with one novel family of learning algorithms: Phased RNNs (Neil *et al.* [22]). They have properties that we hypothesize to be a good fit for time series of the kind we are interested in.

Abstract

We explore machine learning algorithms for time series data, particularly recurrent neural networks. For us, the most interesting methods are ones for handling long duration time series, where the sampling of data channels happen asynchronously in relation to each other, and the sampling happens with no fixed period. This kind of data frequently appears in a variety of real-world contexts. They exist in patient health records, as well as in logs from transportation safety systems, and IT intrusion detection systems. They also come in the form of event-driven sensor signals from manufacturing infrastructure, an example of which we examine in this thesis. We are motivated by the fact that improved methods for learning from this kind of data can provide value to a wide variety of industries and sectors.

We implement recurrent network models and test them on publicly available benchmark datasets, Sequential MNIST, and aperiodic sine wave classification. We also explore applications of the models on datasets consisting of sensor data from manufacturing systems in a real-world food production facility. These datasets are not publicly available.

The recurrent architectures we implement and test are the generic LSTM and GRU models, as well as a specialized model, the Phased LSTM, which is hypothesized to perform well on long, asynchronous and aperiodic signals. We also conceive, implement, and test a natural variation of the Phased LSTM, the Phased GRU, as well as code enabling the construction of multilayer phased models in TensorFlow.

On the publicly available datasets considered, we find that the concept of a time gate from the Phased LSTM shows value, and generalizes well to the GRU cell architecture. Both our Phased GRU, and the Phased LSTM it is inspired by, outperform the baseline LSTM and GRU models in terms of classification accuracy. The phased models also show accuracy performance increases when stacked in layers, as is the case for the baseline models. The Phased GRU is slightly faster, and performs with slightly better accuracy than the PLSTM on tested problems, as is the case for GRU when compared to LSTM. This indicates that performance affecting traits in the GRU and LSTM models are still relevant in the phased context. This has not been researched further in cases where LSTM might outperform GRU in accuracy. On the real, event-driven factory datasets we find some of the weaknesses of phased models, as they show clear disadvantages in accuracy compared to LSTM and GRU in short input sequence tasks. The results are more nuanced for longer input sequences and higher sampling rates, where the phased models sometimes outperform LSTM and GRU.

Sammendrag

Vi utforsker maskinlæringsalgoritmer for tidsseriedata, spesielt rekurrente nevrale nettverk. Metodene som har størst interesse for oss er de som håndterer lange tidsserier, hvor samplingen av datakanaler skjer asynkront i forhold til hverandre, og uten en fast periode. Denne typen data dukker opp i et vidt utvalg av sammenhenger i den virkelig verden. De finnes i form av pasientjournaler, så vel som logger fra sikkerhetssystemer i transportsektoren og verktøy for oppdagelse av datainnbrudd i IT-systemer. De finnes også i form av hendelsesdrevne sensorsignaler fra produksjonsprosesser, som vi utforsker nærmere i denne oppgaven. Vi lar oss motivere av verdipotensialet for mange industrier og sektorer, som ligger i forbedrede algoritmer for maskinlæring på denne type data.

Vi implementerer modeller basert på rekurrente nettverk og tester dem på åpent tilgjengelige benchmark-problemer, sekvensiell MNIST og klassifikasjon av aperiodiske sinusbølger. VI utforsker også anvendelser av modellene på datasett bestående av sensor-data fra en matvarefabrikk. Disse datasettene er ikke offentlig tilgjengelige.

Rekurrente arkitekturer vi implementerer og tester, er generiske LSTM- og GRU-modeller, så vel som en spesialisert modell, Phased LSTM, som er hypotetisert å være velegnet for lange, asynkrone og aperiodiske signaler. Vi utvikler, implementerer og tester også en naturlig variasjon av Phased LSTM, som vi kaller Phased GRU, samt verktøy for å stable rekurrente lag i "phased" modeller.

På benchmarking-datasett finner vi at konseptet om en tids-gate, hentet fra Phased LSTM generaliserer bra til GRU-cellearkitekturen. Både vår Phased GRU-modell og Phased LSTM, som den er inspirert av, presterer bedre enn ordinære GRU og LSTM modeller, målt på klassifikasjonsnøyaktighet. PLSTM og PGRU viser også økning i nøyaktighet når de stables i lag, som er tilfellet for GRU og LSTM. Phased GRU er også litt raskere og litt mer nøyaktig enn Phased LSTM på benchmark-problemene. Dette er ikke ulikt relasjonen mellom vanlige LSTM- og GRU-nettverk. Dette indikerer at egenskaper i GRU og LSTM som påvirker nøyaktighet fortsatt er relevante i "phased" modeller. Dette er ikke utforsket videre i tilfeller der det er kjent at LSTM kan være mer nøyaktig enn GRU. På de reelle, event-drevne fabrikkdatasettene finner vi noen svakheter ved "phased" modeller, da de er mindre nøyaktige enn LSTM og GRU på tester med korte sekvenser. Resultatene er mer nyanserte for lengre sekvenser og høyere sampling rater, hvor Phased LSTM og Phased GRU noen ganger gjør det bedre enn LSTM og GRU.

Preface

This text constitutes the master's thesis of the authors, written as part of a degree in computer science. It was authored through the spring of 2018 at the Department of Computer Science at the Norwegian University of Science and Technology. The thesis was supervised jointly by professor Keith Downing at NTNU, and Bertil Helseth, the CEO of Intelecy AS, a young company in the machine learning industry.

We would like to extend our gratitude to our advisers, Keith and Bertil, for many inspiring discussions, significant proofreading, interesting suggestions, and support in helping us succeed with our thesis. Likewise, we are grateful for the support, inspiration and proofreading provided by friends and family. We would also like to thank the food production company that kindly provided us real-world datasets to test our intuitions on.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter provides an introduction to the thesis. In section 1.1, we outline a justification for our interest in the subject at hand. Our preliminary goals and research questions are detailed in section 1.2. Section 1.3 lists our contributions. Finally, section 1.4 provides an overview of the chapters in the thesis.

1.1 Motivation and Background

In the manufacturing industry it is common to store extensive logs of data related to production processes, going back many years. Modern factories are brimming with sensors, tracking a variety of information. Temperatures of liquids, concentrations of gasses, speeds of rotors and other moving parts, voltages and power draws are all examples of data that can be sampled and stored.

Some of this data is used in real time for controlling and monitoring the production processes. A tank containing temperature sensitive liquid might use temperature data to control a heating element so that a stable temperature is maintained. An alarm could be triggered if too high pressure is sensed in a vessel containing toxic gas. This way of using sensor data is relatively simple in nature, and usually uses fixed thresholds or thresholds based on averages. When the data no longer details the current system state, it is stored and rarely accessed and utilized. We hypothesize that this data has a lot of unrealized value, taking into account recent advances in machine learning. Many tasks can be approached by combining historic process sensor data with real time data and the right algorithms. These include:

- *Anomaly detection* - Given historical data on how a normal, or desired production state is represented in sensor data, it should be possible to detect events - both obvious and not-so-obvious - that break with the norm, and bring attention to them. Further, one can classify these anomalies and handle them according to their class. These anomalies can be due to impending costly and dangerous failures that are desirable to detect and manage as early as possible.

- *Predictive maintenance* - In industry, maintenance is often done at predetermined periodic intervals, based on component degradation statistics. If a trend or recent event in a time series shows signs of an impending component failure, it could be used to predict such a failure, whether it happens before or after the component was scheduled to receive maintenance. Consequently, one can both extend the lifetime of components by not replacing them before it is necessary, and also avoid surprise failures of components that had a lower service lifetime than expected, avoiding the cost of unplanned maintenance.

- *Process optimization* - Utilizing large amounts of historic data for optimizing processes is challenging and often requires manual statistical review. Machine learning techniques can aid by providing improved analysis of large sets of historical data. They can be employed as recommender systems for human process operators by suggesting actions, or perform optimization directly as an agent. Although large scale process optimization driven by machine learning is a field in its infancy, promising results have been claimed. A notable example is Google DeepMind suggesting they reduced power consumption in a data center by 40% using deep learning techniques[40].

With these valuable applications in mind, optimizing machine learning algorithms for the data these processes produce is warranted. Sensor data time series from industry processes is generally of aperiodic and asynchronous nature. Fixed period sampling either fails to capture information between sample points, is very expensive to store and process if the sample frequency is high, or both. Aperiodic and asynchronous data is not handled well by state-of-the-art time series learning models, primarily neural networks, such as LSTM and GRU, at least not in its raw form. If we can develop, reapply, or improve techniques used in artificial neural networks for time series so that they function well on event-driven process sensor data, we think we can offer something of value to the machine learning field and to industries relying on sensor-monitored processes.

A recent addition to the field is the Phased LSTM cell, which is intended for long and irregular time series. We wish to expand on this work, and examine if it is a viable tool for the aforementioned factory data. Throughout the thesis we will be referencing prediction in the form of forecasting, as it is a task that is integral to or closely resembles anomaly detection, predictive maintenance and process optimization. Further, it does not require human labelling for supervised learning and gives good feedback in terms of accuracy and related metrics, so it is ideal for investigating models that will be used for the tasks mentioned above. We will include some classification tasks to show if performance differences can be rooted in the nature of forecasting tasks, such as high importance assigned to samples near the end of the input series.

1.2 Goal and Research Questions

Since we want our work to contribute to the analysis of event-driven data, we here list some overarching goals we want to work towards, and research questions we want to explore.

GOAL: To examine and expand upon recent models for long, aperiodic and asynchronous signals. To achieve higher performance than current state-of-the-art methods on these time series, particularly on real-world process industry data.

Research Questions In the process of achieving this goal, we want to explore and answer the following questions:

- **RQ1:** The Phased LSTM is a variant of the LSTM recurrent neural network model. It is designed to have improved capability to learn from long, asynchronous and aperiodic time series, when compared to the regular LSTM. In this project we propose the GRU variant Phased GRU, inspired by the architecture of the Phased LSTM. **Can the phased models PLSTM and PGRU improve upon the LSTM and GRU baseline in aperiodic, asynchronous time series tasks?**
- **RQ2:** LSTM and GRU models have been shown to perform similarly on sequences in terms of accuracy[15,18]. GRU models show lower time-complexity due to their simpler structure. Does this also hold true for their phased variants? **Do the relative properties of GRU and LSTM hold for their phased derivatives, PLSTM and PGRU?**
- **RQ3:** We have been supplied with logged sensor data from a food production facility, which includes large amounts of event-driven time series data. Tasks and points of interest have been outlined by the employees there. **Do the phased models increase performance on any of the suggested tasks on real-world factory data?**

1.3 Contributions

In this thesis we contribute the following:

- A review of a collection of recent work on problems related to learning long, asynchronous and aperiodic time series.
- A new model based on the GRU model, with the time gate from the Phased LSTM applied to the hidden state, called Phased GRU.
- An implementation of a TensorFlow wrapper class to facilitate use of PLSTM and PGRU models with multiple layers.
- A comparison of various performance measures of LSTM, GRU, PLSTM and PGRU, performed on well-understood datasets, as well as largely unexplored real-world data.

1.4 Thesis Structure

- **Chapter 1 - Introduction:** The introductory chapter provides motivation for the subjects covered in the report, a description of preliminary goals and research questions, and this outline of the report as a whole.

- **Chapter 2 - Background:** Here we discuss prerequisite theory for understanding the specific domain we focus on, further work on time series, and learning algorithms for time series data. We provide an introduction to methods and software for hyperparameter tuning, as well as some thoughts and considerations on frameworks for deep learning.
- **Chapter 3 - Related Work:** This chapter covers similar work on time series data and previous work that motivates and guides our efforts on event-driven data.
- **Chapter 4 - Methodology:** We go into detail on the technical work that was necessary to set up the experiments we wanted to perform, including model implementations and a pipeline for the factory data. We also outline our Phased GRU model.
- **Chapter 5 - Experiments and Results:** We describe experiments and their results on the MNIST dataset, a synthetic wave classification dataset and real factory data.
- **Chapter 6 - Discussion:** We discuss conclusions, insights, limitations and future work based on and motivated by previous chapters.

Chapter 2

Background

In this chapter we go through relevant background knowledge for our task. Section 2.1 details the domain where we wish to apply neural network models. Section 2.2 explains some details of time series, as they are the primary cause of difficulty for current models. Section 2.3 introduces and describes relevant neural network models for time series, both older and newer. Section 2.4 gives an overview of our hyperparameter optimization strategy. Finally, section 2.5 describes the machine learning library we work with; TensorFlow. It also outlines considerations to be made in selecting a machine learning library, and mentions some alternatives to TensorFlow.

2.1 Domain

Modern manufacturing industry operations generate large amounts of data that is recorded and stored. Process infrastructure is often equipped with a wide array of sensors monitoring different aspects of production processes. In the process and manufacturing domain, you might see sensors tracking temperature, pH, pressure, the presence of certain substances, and so on. Control systems for such equipment also generate data. This is usually information about external influences on the system, like valves opening or closing, or heating elements activating or shutting off, and what manual or automated processes affect these control systems. However, only a small portion of this data is utilized, and the data that *is* used is often used in simple ways such as hard-coded alarm thresholds.

One of the current goals of applied machine learning is to utilize data streams from these kinds of processes for a variety of tasks, such as signal classification, anomaly detection, predictive maintenance, and process optimization. Typically, there are at least two broad categories of signals recorded. The first is discrete values for the state of valves, the stage of a process and similar. The second is measurements in the system, such as temperature, volume, voltage and so on. The measurements are predominately continuous values. Both of these types of signals are typically recorded in an event-driven manner where a new value is recorded at the moment where there is a significant change in the real value being tracked. Here, what constitutes a significant change is context dependent and varies

from application to application. This means the set of signals recorded is a set of aperiodic and asynchronous time series. This also means that the signals can be represented as periodic, synchronous time series through forward filling samples, called *imputation* in the field of statistics, without loss of information. However, this comes at a significant space and time cost for computation, as well as possible loss of performance e.g. if the model's memory degrades with each state update.

2.2 Time series

2.2.1 What is a time series

In machine learning we often see that the data examples we train on exhibit some kind of internal relation between dimensions. As an example, a gray scale image is characterized by the fact that every pixel value has been acquired with the same kind of sensor, but represents a small spatial translation from its neighbour pixels. It turns out that this insight has value, and in most non-naive machine learning methods in the computer vision domain, we do not treat the image as a simple vector of values. In a *convolutional neural network* (CNN) we keep and utlize the spatial information of the image by treating it as a multidimensional array - a tensor - of shape $(h \times w \times c)$, height by width by number of channels, as illustrated in Figure 2.1. This structuring is essential in order to achieve the kind of impressive performances convolutional neural networks display on images.

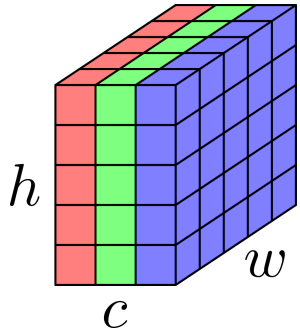


Figure 2.1: An RGB image interpreted as a tensor.

In other data than images, there might be structures that we can exploit in tailored algorithms, as seen in the case of how spatial relations are treated in CNNs. An example is the field of natural language processing. To see how language exhibits structure, just consider what happens to a sentence if you randomly rearrange the words in it. A sentence may change meaning or lose its meaning entirely if you permute its ordering. This is because a sentence is an *ordinal sequence*, a structure with a determined ordering of elements:

$$\boldsymbol{X} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_n). \tag{2.1}$$

Certain learning algorithms are well suited to handle this kind of data, among them RNNs, which we will discuss in section 2.3.

A *time series* is an extension of the concept of the ordinal sequence. Whereas an ordinal sequence can be the result of any ordering or sorting, in a time series, the ordering between elements is decided by the point in time they were sampled at. The time series differs from an ordinal sequence in that it contains this information in the form of a time stamp t_j associated with any x_j in the sequence:

$$\boldsymbol{X} = ((\boldsymbol{x}_1, t_1), ..., (\boldsymbol{x}_n, t_n)). \tag{2.2}$$

Sometimes this is made implicit by including t_j as a part of $\boldsymbol{x}_j$. Sometimes the absolute time value t_j is also replaced by time *difference* information, in the form of $\Delta t_j = t_j - t_{j-1}$.

Modern RNN architectures used today, such as Long Short-Term Memory (LSTM), discussed in 2.3.3, and the Gated Recurrent Unit (GRU), discussed in 2.3.4, are often referred to as models tailored for temporal data, but they are primarily used with constant time steps between measurements. In this case, where data arrives at a constant time interval, the data is fed to the learning algorithm without a time stamp, and is treated *no different* from an ordinal sequence. Handling the input as sequential data without time information is effective in these models, but it only solves a subset of all time series data tasks.

2.2.2 Aperiodicity

If a time series is sampled at an irregular interval, or recorded from an event-driven process running in continuous time, then we refer to it as being aperiodic. By default, regular recurrent neural network models such as LSTM have no concept of time, only of a sequence ordering, an ordinal structure. Thus they will view a large change over a large time interval the same as a large change over a very small interval, if the distribution and amount of measurements is the same. There are two commonly used methods for handling this situation in regular recurrent models. It is possible to resample the aperiodic time series to a periodic form, either by interpolation, or by imputation of time steps to a frequency high enough to regularize the series. It is also possible to feed the model time stamps as input, alongside the features at the given step. This should in theory give the model an opportunity to understand the temporal relations between samples. An example of an aperiodic time series can be seen in Figure 2.2. These methods come with their own unique deficiencies. Regularizing by interpolation would in many cases be expected to lead to loss and alteration of information in the series. It is also not clear how to properly deal with categorical features using this method. Regularizing by upsampling comes with the problem of a potentially substantial increase in the memory requirements for working with the time series. It also poses a problem for common RNN architectures, like LSTM and GRU, as they have limited ability to remember complex dependencies across longer sequences[22], and lack dedicated temporal attention mechanisms.

2.2.3 Asynchronisity

For a time series with multiple channels, that is, $dim(\boldsymbol{x}_j) > 1$, it is possible that each individual channel can be periodically sampled, but that samples do not align across channels. Say you are tracking two sensors, *a* and *b*, that each make up one channel in your

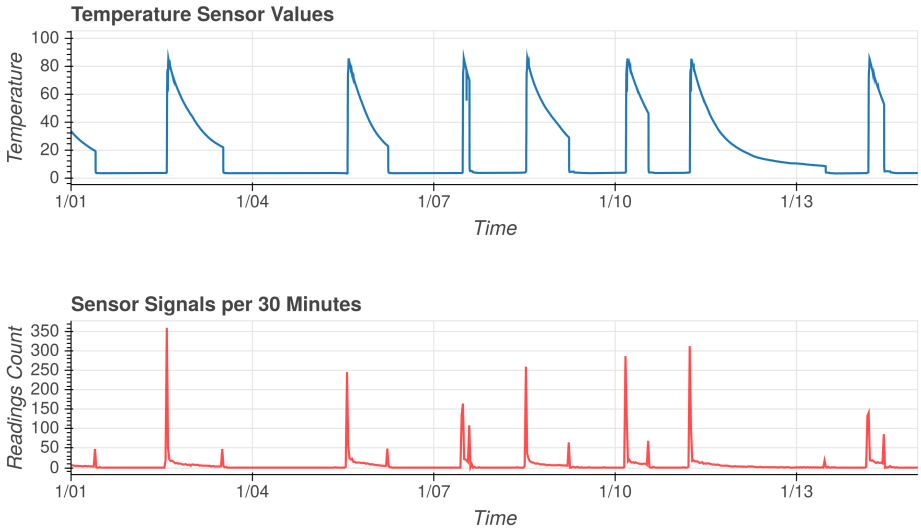


Figure 2.2: Plot of temperature and temperature sample frequency a for milk storage tank. Note the correlation between the sample frequency and the slope of the measured value.

series. If you at some time t_j have a sample from a, but not from b, we refer to the series as asynchronous. Standard implementations of recurrent models, and neural networks in general, assume and require a measurement for all features, or channels, in an input example. It thus poses the problem that we sometimes might not have this measurement. For time series, where new data is only recorded when there is a significant change in the feature being measured, it is reasonably safe to assume that the last seen value in a channel will still represent a relatively accurate depiction of the current feature state. Thus, we can forward fill this value to gain a time series with synchronous channels. This is the case for the data we consider in this thesis, but it might not always be true, and one might have to utilize more sophisticated methods of dealing with missing data for the time steps in a series. We regard tailored handling of missing data that cannot be forward filled as outside the scope of our work. However, there are other issues with asynchronisity that we deem to be of interest. The forward filling of values is associated with the same issue of increasing memory requirements as the upsampling strategy for tackling aperiodicity, as described above. Furthermore, having to consider all of the individual time steps from different channels in a series leads to more state changes in the recurrent network models applied on them, and thus worsens the problem of learning long dependencies. Finally, when there are only sparse changes in consecutive input vectors $\boldsymbol{x}_{j-1}$ and $\boldsymbol{x}_j$, it is wasteful to fully recompute operations where $\boldsymbol{x}_j$ is involved, but we have discovered no RNN implementations that realize this potential for reduction in computational cost.

2.3 Learning Algorithms for Time Series Data

In this section we cover various algorithms commonly used to learn from time series data. Inspiration for some of the explanations and figures have been drawn from Chris Olah of Google Brain's personal blog[42], though figures are (re)created by the authors.

2.3.1 Artificial Neural Networks

In the context of modern neural networks, the multilayer perceptron (MLP) can be considered the most basic starting point of other network architectures. MLP is a feedforward artificial neural network with 3 or more layers, consisting of an input layer, an output layer, and a number of hidden layers in between. The input layer is only responsible for feeding inputs to the next layer through the first set of weights, each cell there corresponding to one input channel, and thus one value for each sample in the input data. Each other layer consists of a number of nodes or cells which take as input the weighted sum of the previous layer's output and runs it through a predefined activation function, and outputs the result to the next layer. As an MLP is fully connected between layers and acyclic (or *feedforward*), input to every node is the weighted sum of all nodes in the previous layer. Optional bias nodes add a single value to each node in a layer, typically the hidden and output layers. This gives us the basic behaviour at each cell

$$output = f(\sum (weight \times input) + bias) \tag{2.3}$$

which can be written as

$$\boldsymbol{h} = f(\boldsymbol{W}_x \boldsymbol{x} + \boldsymbol{b}), \tag{2.4}$$

where $\boldsymbol{h}$ is the output of the cell, f is an activation function, $\boldsymbol{W}_x$ is the weight matrix for the matrix of input values $\boldsymbol{x}$ and $\boldsymbol{b}$ is the bias. Multilayer perceptrons are theoretically powerful models. It follows from the *universal approximation theorem*, an early formulation of which is due to Cybenko [1], that given some restraints on the activation function, f, and arbitrarily large, but finite layer sizes, a multilayer perceptron with one hidden layer can approximate any continuous function on a compact subset of $\mathbb{R}^n$. This, however, is often difficult to achieve in practice.

2.3.2 Recurrent Neural Networks

A regular feedforward network defines a parameterized mapping

$$\boldsymbol{h} = \hat{f}(\boldsymbol{x}; \boldsymbol{\theta}), \tag{2.5}$$

from a single argument $\boldsymbol{x}$ to some output $\boldsymbol{h}$. This is not practical in the case of machine learning on sequences. Like in image processing, where it turns out that it is interesting to apply a filter to the entire image, and we *share parameters* over the entire spatial extent of the image to achieve that, it turns out that parameter sharing is something to strive for in sequence learning as well.

Consider the two sentences: *"I am happy that you are here"* and *"that you are here makes me happy"*. Notice that even though the word "happy" is located in different places in the two sentences, it carries the same *meaning*. In image data, when asking the question *what do these pixels display*, the answer does not depend on *where* in the image the pixels are located. A face is a face, no matter if it is centered in the image or to the right or left. Similarly, a word often has a fixed meaning throughout a sentence, even though the context varies. Consequently, we want to apply a similar operation to all the different steps, or time steps $\boldsymbol{x}_j$ in a sequence, so we avoid having to relearn time-invariant patterns across all steps. One way this can be done is by applying a one-dimensional convolution operation

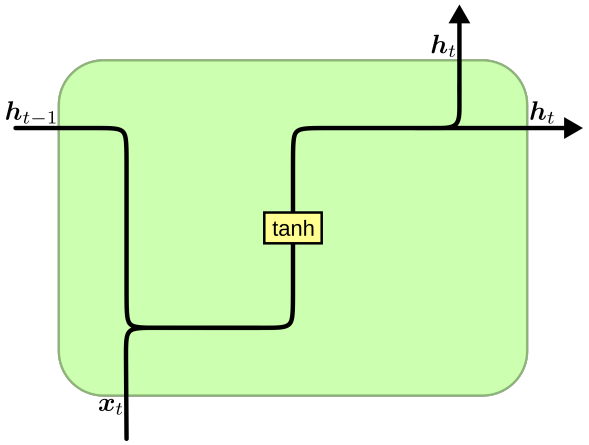


Figure 2.3: A data flow graph illustrating the structure of a layer in an Elman network. View the horizontal axis as time, and the vertical axis as the spatial location inside a larger network layer stack. An RNN layer is very similar to a standard MLP neuron, the difference being its input. We see that the cell takes $\boldsymbol{x}_t$, and $\boldsymbol{h}_{t-1}$ as input (whereas an MLP neuron only takes a vector $\boldsymbol{x}$). While $\boldsymbol{x}_t$ is either the input vector to the network or the output of an earlier layer at that particular time, $\boldsymbol{h}_{t-1}$ is the output vector from the illustrated layer, but from the preceding time step. Weighted sums of the components of both $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ are passed through an activation function, σ_h to produce $\boldsymbol{h}_t$. In this case σ_h is the hyperbolic tangent, as can be seen by the yellow square representing a simple neural network layer.

in time, as is done in a CNN. A more common method when working on sequences, however, is to utilize a recurrent mapping applied to every time step:

$$\boldsymbol{h}_t = \hat{f}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}; \boldsymbol{\theta}), \tag{2.6}$$

where the output from the map on the previous step is utilized in the calculation of the next output. Often the inferred value will then be the final output from the recurrent map, $\boldsymbol{h}_{|X|}$. A simple way of realizing this theoretical model is through a recurrent network - called an Elman network, after its inventor[2] - where each hidden layer output is defined as

$$\boldsymbol{h}_t = \sigma_h(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}). \tag{2.7}$$

In this context $\boldsymbol{x}_t$ is the time step at time t, $\boldsymbol{h}_{t-1}$ is the output from the layer at the previous time step, $\boldsymbol{W}_x$ and $\boldsymbol{W}_h$ are weight matrices, and $\boldsymbol{b}$ is a bias vector. The activation

function σ_h can be any function, but often the hyperbolic tangent is used. An illustration of such a network can be seen in Figure 2.3. Another common structure for recurrent neural networks, is the Jordan network[4]:

$$\boldsymbol{h}_t = \sigma_h(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{y}_{t-1} + \boldsymbol{b}). \tag{2.8}$$

Here $\boldsymbol{y}$ denotes the network's output. Notice the similarity with the Elman network. The only difference is that instead of taking $\boldsymbol{h}_{t-1}$ as input from the previous network state, it takes $\boldsymbol{y}_{t-1}$. These architectures are equivalent in one-layer networks, where $\boldsymbol{h} = \boldsymbol{y}$. When layers are stacked, however, every layer in the Jordan network only remember the previous *network output*, instead of their own hidden output. Since the Elman network has a more complete recurrent memory over time, it is arguably more expressive, and it is seen far more often used in practice.

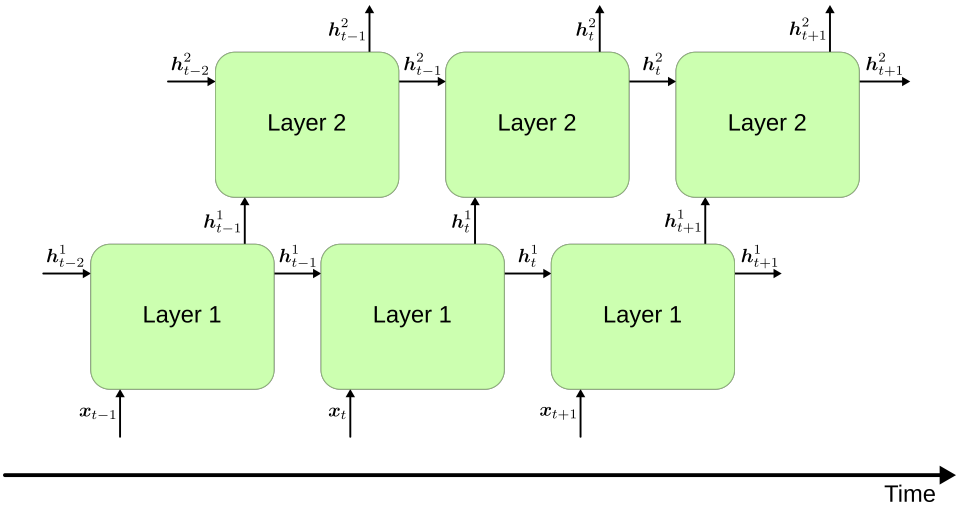


Figure 2.4: Here we see a two-layer RNN unrolled over time. Each green box corresponds to the layer structure illustrated in Figure 2.3. In each time step the layer computes its new hidden state from its previous hidden state and the current input vector. For the second layer, just like in standard feedforward networks, the input vector is the hidden state of the previous layer.

Training Recurrent Networks

For a feedforward neural network we define the gradient of the loss function $\mathcal{E}$ with respect to a parameter θ as

$$\frac{\partial \mathcal{E}}{\partial \theta} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \theta}. \tag{2.9}$$

When we fully expand this expression through use of the chain rule, and calculate the loss gradients of parameters during training, it is known as the back-propagation algorithm. This method is also applicable for the training of recurrent neural networks. If we *unroll*

the network over time, as illustrated in Figure 2.4, we see that a recurrent network can be interpreted as a acyclic, directed computational graph, just like feedforward networks are. Here, however, we have to deal with the fact that any parameter appears one time for every time step in the calculation of a network output. Also, the network produces outputs for every time step, and these outputs might all contribute to the loss of the network. The gradient of the loss $\mathcal{E}$ with regards to the parameter θ is then given as:

$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{t=1}^{T} \left(\frac{\partial \mathcal{E}}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \theta} \right), \tag{2.10}$$

Note that θ contributes to $\boldsymbol{h}_t$ both directly, at time step t, but also recursively through its effect on $\boldsymbol{h}_{t-1}$. We can express this relation as

$$\frac{\partial \boldsymbol{h}_t}{\partial \theta} = \sum_{k=1}^{t} \left(\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} \frac{\partial^+ \boldsymbol{h}_k}{\partial \theta} \right), \tag{2.11}$$

where $\frac{\partial^+ \boldsymbol{h}_k}{\partial \theta}$ signifies the *immediate* partial derivative of the state $\boldsymbol{h}_k$ with regards to θ, as defined in Pascanu *et al.* [13]. This is a partial derivative where we regard $\boldsymbol{h}_{k-1}$ as constant with respect to θ. From this we can derive a gradient for θ that covers all its influences on the loss. This method of applying backpropagation to recurrent networks is known as *backpropagation through time* (BPTT).

Vanishing and Exploding Gradients

Vanishing and exploding gradients occur in some networks when the gradient in early layers is the product of terms for all later layers. Vanishing occurs when the repeated multiplication is by terms less than 1 and exploding gradients occur when it is by terms more than 1. Exploding and vanishing gradients can occur in both deep networks and networks with long lookback through time in a sequence, as they involve gradients being the product of many terms. This can be seen as vanishing/exploding along the horizontal (time) or vertical (depth) axis in Figure 2.4. LSTM networks mitigate this issue by allowing the gates to set the multiplication of input and hidden/cell state values to 1 or near 1 for important values (forget and output gates) and updating the network state through addition instead of multiplication (after input gate). See Figure 2.7 for the layout of an LSTM cell and subsection 2.3.3 for details about LSTM.

Computational Cost of Recurrent Networks

For computational cost, at the most basic level, an RNN can be seen as a sequence of ANNs when unrolled through time, as seen along the horizontal axis in figure Figure 2.4. With this in mind, an approximation of the cost of processing a single sample with an RNN is the cost of an equivalent ANN multiplied by the length of the input sequence. In the context of modern massively parallel computing, this would not necessarily need to be problematic, if it was not for the fact that the calculations for each time step are sequentially dependent. You must calculate $\boldsymbol{h}_{t-1}$ (and $\boldsymbol{c}_{t-1}$ in the case of LSTM) to be able to calculate $\boldsymbol{h}_t$. This can to some extent be mitigated in training through increasing

batch sizes until computational resources are saturated. It does, however, mean that for longer sequences one cannot expect inference to be done in real time.

2.3.3 Long Short-Term Memory

A frequently used cell in modern deep RNNs and common baseline model for temporal data, is the LSTM cell, originally introduced in 1997 by Hochreiter & Schmidhuber [3]. A commonly-used definition of the modern LSTM follows[11]:

$$\boldsymbol{i}_t = \sigma_i(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ci}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i) \tag{2.12}$$

$$\boldsymbol{f}_t = \sigma_f(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{cf}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f) \tag{2.13}$$

$$\widetilde{\boldsymbol{c}}_t = \sigma_c(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \tag{2.14}$$

$$\boldsymbol{c}_t = \boldsymbol{i}_t \odot \widetilde{\boldsymbol{c}}_t + \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} \tag{2.15}$$

$$\boldsymbol{o}_t = \sigma_o(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{co}\boldsymbol{c}_t + \boldsymbol{b}_o) \tag{2.16}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \sigma_h(\boldsymbol{c}_t). \tag{2.17}$$

Here $\boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{o}_t$ represent the input, forget and output gates at time t respectively. $\boldsymbol{c}_t$ represents the cell state vector, $\widetilde{\boldsymbol{c}}_t$ represents a candidate state vector, $\boldsymbol{x}_t$ represents the input vector, and $\boldsymbol{h}_t$ the hidden output vector. $\odot$ denotes the entry-wise product or Hadamard product. The gates commonly use sigmoidal nonlinearities represented by σ_i, σ_f, σ_o, whereas the candidate state and hidden output often use hyperbolic tangent nonlinearities represented by σ_c and σ_h. The gates and candidate state function act like feedforward layers, but with multiple input vectors, and use weight parameters $\boldsymbol{W}_{xi}$, $\boldsymbol{W}_{xf}$, $\boldsymbol{W}_{xc}$, and $\boldsymbol{W}_{xo}$ for the input vector; $\boldsymbol{W}_{hi}$, $\boldsymbol{W}_{hf}$, $\boldsymbol{W}_{hc}$, and $\boldsymbol{W}_{ho}$ for the hidden vector from $t-1$; $\boldsymbol{W}_{ci}$, $\boldsymbol{W}_{cf}$, and $\boldsymbol{W}_{co}$ for optional peephole connections to $\boldsymbol{c}$; as well as biases $\boldsymbol{b}_i$, $\boldsymbol{b}_f$, $\boldsymbol{b}_c$, and $\boldsymbol{b}_o$. Peephole connections[5] allow the gates to look at the cell state $\boldsymbol{c}$, by giving it as input to the various gates in the cell. This might improve performance on some problems. The cell state $\boldsymbol{c}_t$ is updated to consist of an entry-wise fraction of $\boldsymbol{c}_{t-1}$ from the previous cell state, and an entry-wise fraction of a new candidate input state $\widetilde{\boldsymbol{c}}_t$. The fractions are decided by gates $\boldsymbol{f}_t$, and $\boldsymbol{i}_t$, respectively, by application of the entry-wise (or Hadamard) product $\odot$, as illustrated in Figure 2.6a. A graphical representation of the equations 2.12 to 2.17 can be seen in Figure 2.7.

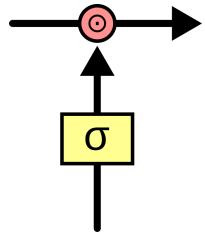
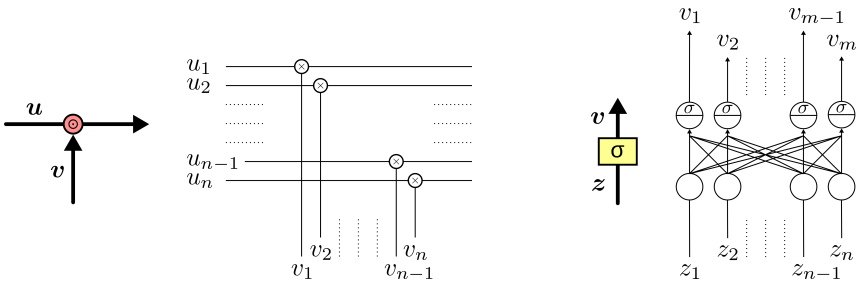


Figure 2.5: The general composition of a gate such as what is found in an LSTM cell. The horizontal line is the signal to be gated. The vertical line is the input to the sigmoidal layer and then the output, a gating mask.



(a) RNN gate utilizing the Hadamard product

(b) Sigmoidal layer of a gate

Figure 2.6: In 2.6a, we see an illustration of the Hadamard product of two vectors illustrated in two different but equivalent ways. We can interpret $\boldsymbol{v}$ as being a control signal that amplifies or attenuates the components of $\boldsymbol{u}$. If the entries in $\boldsymbol{v}$ are in the range 0 to 1, $\boldsymbol{v}$ decides how much of the information in $\boldsymbol{u}$ to let through. The horizontal lines can thus be understood as signals to be gated. In 2.6b, we see an illustration of how the gating signal $\boldsymbol{v}$ from 2.6a can be generated though a sigmoid-activated neural network layer. The output from the layer is a vector of values in the range from 0 to 1, that serve the purpose of $\boldsymbol{v}$ in 2.6a. In 2.6b, $\boldsymbol{z}$ represents an input vector for the gate.

Gating Gating warrants a short explanation. This is not intended to show the inner workings of a library such as TensorFlow (which revolves around efficient tensor operations), but to show the logic of a gate. Gates are intended to let some fraction of information through. Gates consist of a sigmoid neural net layer and an entry-wise multiplication operation, see Figure 2.5. The sigmoid layer outputs a matrix or mask of values between 0 and 1, specifying how much information should be let through. This mask is then applied to an identically-shaped signal through the entry-wise multiplication operation, so that 1 means "let everything through" and 0 means "let nothing through", see figure Figure 2.6a. The gating layer that creates the mask does so based on its own separate input, as can be seen in Figure 2.6b

2.3.4 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) introduced by Cho et al. in 2014 is an alternative to LSTM for similar sequential data. The GRU can be thought of as a simplified version of the LSTM unit. The input and forget gates found in LSTM are combined into a single "update gate" and the hidden and cell states are merged. The output gate used for the hidden state is removed due to the merge, and a reset gate is introduced to control the input to the hyperbolic tangent nonlinearity found in LSTM. The equations for the gates and states are as follows:

$$\boldsymbol{z}_t = \sigma_z(\boldsymbol{W}_{xz}\boldsymbol{x}_t + \boldsymbol{W}_{hz}\boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \tag{2.18}$$

$$\boldsymbol{r}_t = \sigma_r(\boldsymbol{W}_{xr}\boldsymbol{x}_t + \boldsymbol{W}_{hr}\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{2.19}$$

$$\widetilde{\boldsymbol{h}_t} = \sigma_h(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{2.20}$$

$$\boldsymbol{h}_t = \boldsymbol{z}_t \odot \widetilde{\boldsymbol{h}_t} + (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} \tag{2.21}$$

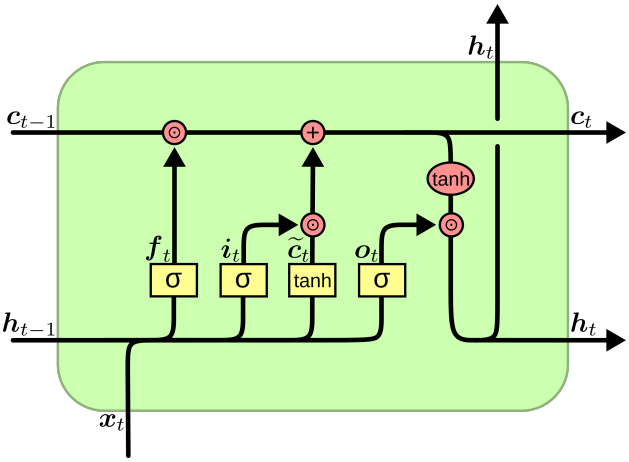


Figure 2.7: A data flow graph describing the operations of an LSTM layer. Note that peephole connections have been left out for simplicity. We see that the cell takes $\boldsymbol{x}_t$, $\boldsymbol{c}_{t-1}$ and $\boldsymbol{h}_{t-1}$ as input. It then generates $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$ through its internal dynamics, and passes them forward to itself in the future. $\boldsymbol{h}_t$ is also passed on to the next network layer as its respective $\boldsymbol{x}_t$. This description presupposes an Elman network architecture.

Where $\boldsymbol{z}_t, \boldsymbol{r}_t, \boldsymbol{h}_t, \widetilde{\boldsymbol{h}_t}$ represent the update gate, reset gate, hidden state and candidate hidden state respectively. As can be seen in the equation for the new hidden state $\boldsymbol{h}_t$, the update gate is used both to scale the candidate hidden state and the previous hidden state.

LSTM and GRU accuracy and speed on most tasks is comparable, with the GRU outperforming LSTM units somewhat in both CPU time to converge and accuracy[15], though it has been shown that adding a bias of 1 to the forget gate in LSTMs makes the difference in accuracy negligible[18].

2.4 Hyperparameter Optimization

In very early RNN models, hyperparameters could make or break model performance. Some hyperparameter choices have been made less critical due to recent work, for instance learning rates due to gradient descent optimizers such as RMSprop[9], Adagrad[8] and Adam[16]. Parameters such as cell number and layer count in neural networks can still be critical. Choosing critical parameters is largely done through empirical testing of different settings. To make this process efficient and reproducible, various algorithms have been proposed, typically search algorithms that effectively traverse the search space given a space of hyper-parameters to search through and a score for each run of a set of hyper-parameters. An overview of algorithms for hyper-parameter optimization can be found in Bergstra *et al.* [7].

As hyper-parameter optimization typically involves traversing the search space through repeatedly running the model with sets of hyper-parameters and recording the scores, it is often a very expensive process. Fortunately, in algorithms where the search is directed by the search space, what configurations are currently running, and scores from previously

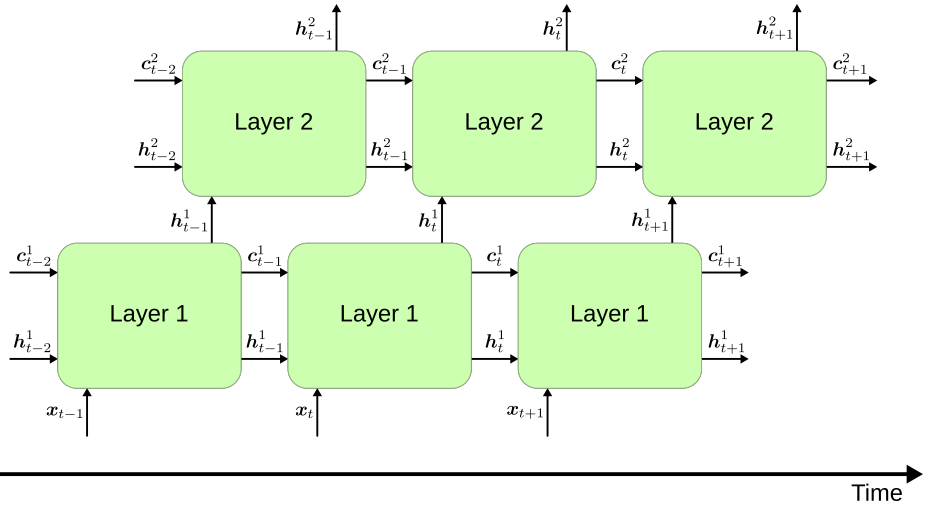


Figure 2.8: Here we see a two-layer LSTM unrolled over time. Each green box corresponds to the layer structure illustrated in Figure 2.7. Compared to the similar illustration in Figure 2.4, the important difference to note is that only part of the state of a layer is exposed to the next layer in the stack. Whereas $\boldsymbol{h}$ is transmitted both to the next layer and forwards in time as memory, $\boldsymbol{c}$ is *only* used as memory within its layer.

run configurations, distribution is trivial. In our experiments, we run Hyperopt[10] with its `DistributedTrials` class. The search space is defined through grids for cell sizes and layers, and distributions over ranges for continuous parameters such as learning rate. The search starts with an optimization instance that is given the search space and places jobs (a single run with a set of hyper-parameters) on a queue according to the Tree-structured Parzen Estimator Approach (TPE)[7] as implemented in Hyperopt. The queue and results are stored in a MongoDB instance which can be accessed by all our compute resources. Jobs are then dequeued and run in parallel to each other on the compute nodes and personal computers we have available.

2.5 Machine Learning Libraries

Training deep neural networks on large datasets is an expensive task. For instance, when running stochastic gradient descent, every time you run a mini batch through the network, you want to update every trainable parameter in the network. Every parameter update relies on gradients of the loss for every example in the mini batch. Depending on task and architecture, this process can take hours, days, or even weeks. A recent example comes from the field of reinforcement learning. When Google DeepMind created their strongest agent for playing Go, AlphaGo Zero, the best-performing network was trained on a 64 GPU and 19 CPU distributed system over 40 days[31], and was still improving at training cutoff. It follows that efficient and distributed implementations of algorithms and data structures are important when doing non-trivial work in deep learning. To perform

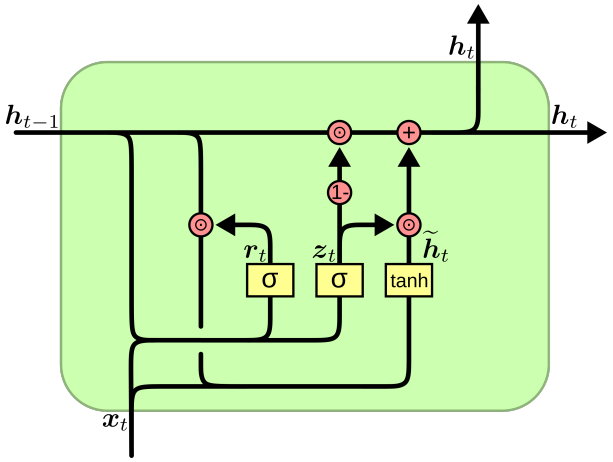


Figure 2.9: An illustration the data flow and operations inside a GRU layer. Note that as opposed to the LSTM, the GRU lacks a cell state, $\boldsymbol{c}$, only propagating the hidden state, $\boldsymbol{h}$, over time.

our own experiments, we could implement all algorithms involved from the ground up, but luckily libraries that simplify the process of developing neural network architectures exist. In this section we discuss properties of a high quality machine learning library, we describe our choice of library (TensorFlow), and we mention some alternatives that could be utilized.

2.5.1 Considerations and Quality Measures

A good deep learning library in the research setting will let you define your network through sufficiently abstract components. Most of these components should be predefined by the library, but you should have the freedom to define and redefine such components yourself. This freedom should range from low-level features like neuron connections and activation functions to higher-level structures like layers and networks. The training process should allow easily adding regularization techniques like dropout, batch normalization, and various weight initializations. Similarly, commonly used loss functions and optimization algorithms should be implemented. As with network structures, these components should also be extensible and modular. Finally we do not want to define derivatives of our trainable parameters ourselves. A good library will hide the internals of the backpropagation algorithm from you, and perform automatic differentiation while allowing modifications to the basic algorithm if necessary.

A final quality measure for a machine learning library is its ease of distribution and its scalability with computational resources. The training of neural models can happen on increasingly sophisticated hardware structures. A very simple network might only need a CPU to train in reasonable time, whereas more sophisticated networks might need to utilize a GPU. The most demanding models will be trained on distributed clusters of computers with several GPUs each, or specialized hardware optimized for neural network training, like Google's TPU. This is achievable because neural network training and execution are

inherently very parallelizable problems. But realizing this potential relies on the software implementation to facilitate it. Ease of distribution in the context of a machine learning library means low overhead for moving a model from a simple platform to a more complex one. High-level code, such as neural network graph definitions, should need little change in order to run on, and utilize the potential for parallel execution in increasingly complex hardware, both in terms of number of processing units, and types of processing units. Finally we want the performance on single processing units to be optimized. Even in the case of complex models a lot of prototyping and small scale experiments happen on simple hardware, like one CPU or a CPU and a GPU. A good library will not only be easily distributable, but also perform at a high level under resource constraints.

2.5.2 TensorFlow

A commonly used open source machine learning library is TensorFlow. Initially developed by Google as an internal framework for neural networks and computational graphs, it was open sourced in 2015 and developed further by a strong community in addition to Google. It provides APIs for many languages, but the Python API is the most widely used and best supported. TensorFlow lets you define numerical computations in terms of data flow graphs, where nodes are operations and edges correspond to the data communicated between them. In the case of TensorFlow, this data is represented by tensors, which are n-dimensional generalizations of vectors and matrices. When a computational graph is defined, it is compiled into a highly optimized program, which can later be executed on input data. In this sense, it is helpful to think of TensorFlow programming as a form of meta-programming. While your code is written in Python, most of the program execution happens in the compiled graph structure outside of Python, in the TensorFlow Distributed Execution Engine.

For this reason it can be hard to debug TensorFlow models with standard Python tools like the Python Debugger. TensorFlow provides utilities for debugging and examining how the program performs. For visualizing the computational graph and various statistics for training runs, you can use TensorBoard. It is a web interface that is built to display TensorFlow summary information in an accessible way. There is also the TensorFlow Debugger, which gives you access to the internals of the graph at runtime.

In TensorFlow you can define this computational graph at multiple layers of abstraction. You can define a network in terms of tensor products, sums and non-linearities, or you can use predefined 'layer' operations. In addition, TensorFlow includes an Estimator class, which lets you abstract away much of the training loop. This hierarchy is illustrated in Figure 2.10.

A final significant reason for choosing to work with TensorFlow is the significant community of developers using it. As it is, at the time of writing, the most popular deep learning library by far[37], most problems you encounter will have been discussed in public fora in some form already, and there is a plethora of open source code to gain inspiration from.

2.5.3 Other Frameworks

In addition to TensorFlow there are some other notable library options to consider. It is a complex task to compare them, and we consider it outside of the scope of this project, but

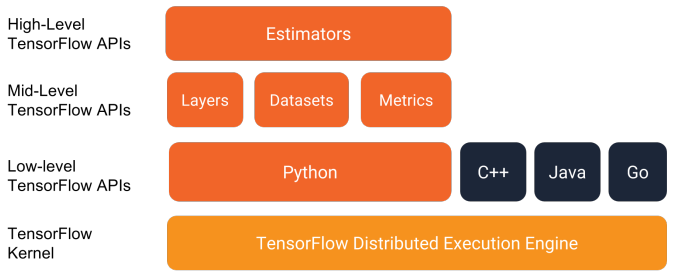


Figure 2.10: An illustration of the various levels of abstraction one can utilize when building machine learning models in TensorFlow. Lifted from the TensorFlow web page[46], under the Creative Commons Attribution 3.0 License[39].

mention them briefly here.

- *Caffe2* - Originating as a Facebook project, Caffe2 is a library intended to be suited for production applications and mobile platforms.
- *PyTorch* - Whereas TensorFlow used to only build static computational graphs outside of Python, PyTorch introduced a fundamentally different approach. The PyTorch computational graph is dynamic and amendable during runtime, its API is more closely related to traditional Python packages like NumPy of the SciPy[6] ecosystem. Representing an interesting take on neural network programming, the library is still early in development. Like Caffe2, the library originates at Facebook, but is primarily intended for research applications. The idea of a dynamic graph has since the introduction of PyTorch been incorporated into TensorFlow as an optional feature.
- *Microsoft Cognitive Toolkit* - Like TensorFlow originated from the internal codebase at Google, Microsoft Cognitive Toolkit (CNTK) was released as open source by Microsoft. It shares some aspects with TensorFlow, like the static computational graph. CNTK proposes a list of reasons why you might consider it over TensorFlow, including claimed speed advantages[41].

Chapter 3

Related Work

This chapter details related work on learning models for aperiodic, asynchronous time series or similar challenges relating to continuous time data and some of the lessons learned from these attempts. In addition, we include some works relating to general sequence modelling to justify our choice to explore LSTM-related networks.

Section 3.1 details the Time-LSTM models, three variations on LSTM specialized for time-sensitive recommender system data. Section 3.2 details Phased LSTM which adds an open/closed state to the network with trainable frequency and duration, allowing the cells in the network to operate at different time scales. We have chosen this model for further work and testing. Section 3.3 describes Clockwork RNN, an attempt at modelling long term dependencies, which involves differing frequencies of operation within a layer, similar to Phased LSTM. Section 3.4 describes Continuous-Time GRU (CT-GRU) which attempts to handle discrete events in continuous time, by using multiple hidden states with different decay rates. Section 3.5 briefly covers a recent approach to utilize LSTM or related architectures effectively in a hybrid model with One Class Support Vector Machines (OC-SVM) or Support Vector Data Description (SVDD) for anomaly detection. Section 3.6 details an attempt to generate improved recurrent architectures through genetic algorithms, which found that both LSTM and GRU are well-performing architectures even among the many tested mutations on a wide selection of problems. Section 3.7 describes how attention can be employed to assign varying degrees of importance to different steps in a sequence. This is a technique that could be utilized to increase a model's abilities to understand long-term dependencies or disregard noisy information. Section 3.8 presents the Skip RNN. This model is able to learn to judge how many time steps can be safely ignored before performing new updates of the hidden state. This reduces computational complexity, with the added benefit of improving learning of long-term dependencies. Section 3.9 introduces the Nested LSTM; a model that provides an alternative to layer stacking in RNNs by nesting recurrent cells within one another. Section 3.10 contains brief glances at other works that we deem to have some degree of relevance to our work. Section 3.11 gives a summary of the models reviewed in this chapter.

3.1 Time-LSTM

Zhu *et al.* [32] explore RNN solutions with time information designed for recommender systems. They introduce a separate time gate which receives feature input and time interval input. They propose three versions of Time-LSTM with different roles for the time gate. In their best-performing model they use coupled input and forget gates[27] with two time gates where one is designed to exploit time intervals for current recommendations (influences output) and one to store time intervals for future recommendations (influences cell state which affects later timesteps). They show improved results over LSTM with time input, and also the Phased LSTM model described in the next section. However, this model is, as stated by the authors, specifically designed to handle their datasets (user action sets for recommender systems), where data is very sparse, time intervals between recent actions are critical, and where long-term and short-term dependencies show very different trends.

3.2 Phased LSTM

The Phased LSTM model introduced by Neil *et al.* [22] is an extension of the LSTM model. It adds the concept of a trainable time gate to the LSTM cell. The time gate controls when the cell state can and cannot be updated through an oscillating function. This is hypothesized to be useful primarily for three scenarios; long time series, time series with asynchronous sampling rates for different data points, and finally event-based time series where there is no fixed sampling rate at all. The gating function is defined as follows:

$$\phi_t = \frac{(t-s) \mod \tau}{\tau}, \quad k_t = \begin{cases} \frac{2\phi_t}{r_{on}} & \text{if } \phi_t < \frac{1}{2} r_{on} \\ 2 - \frac{2\phi_t}{r_{on}} & \text{if } \frac{1}{2} r_{on} < \phi_t < r_{on} \\ \alpha\phi_t & \text{otherwise} \end{cases} \tag{3.1}$$

ϕ_t is a simple saw-tooth function, rising from 0 to 1 over τ time. It also has a phase shift of s. k_t, the gating function, peaks during an interval r_{on} of the driver function. τ, s, and r_{on} are trainable parameters for each cell. α is a leak rate that allows propagation of important gradients even when the gate is closed. It is usually set to a small positive value, like 0.05, during training, and 0 at inference. The activation behaviour of the time gate k_t can be seen illustrated in Figure 3.1.

The PLSTM cell equations are an extension of the normal LSTM equations. Since we can apply the former to asynchronous time series, we might not have fixed distances between time steps. For this reason we use the notation t_j and t_{j-1} instead of t and $t-1$, as to reference the sequence ordering, rather than a fixed step in time. In the following PLSTM definition, j is shorthand for t_j, and $j-1$ is shorthand for t_{j-1}:

$$\widehat{\boldsymbol{c}_j} = \boldsymbol{i}_j \odot \widetilde{\boldsymbol{c}_j} + \boldsymbol{f}_j \odot \boldsymbol{c}_{j-1} \tag{3.2}$$

$$\boldsymbol{c}_j = \boldsymbol{k}_j \odot \widehat{\boldsymbol{c}_j} + (1 - \boldsymbol{k}_j) \odot \boldsymbol{c}_{j-1} \tag{3.3}$$

$$\widehat{\boldsymbol{h}_j} = \boldsymbol{o}_j \odot \sigma_h(\widehat{\boldsymbol{c}_j}) \tag{3.4}$$

$$\boldsymbol{h}_j = \boldsymbol{k}_j \odot \widehat{\boldsymbol{h}_j} + (1 - \boldsymbol{k}_j) \odot \boldsymbol{h}_{j-1}. \tag{3.5}$$

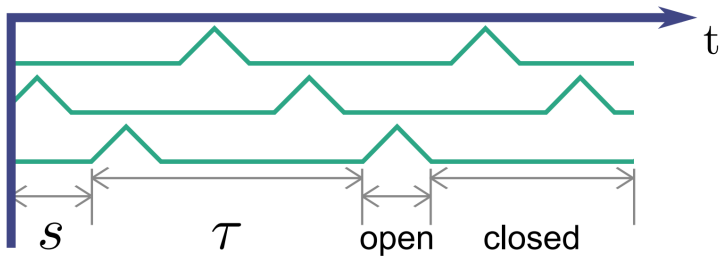


Figure 3.1: Here we see the time gate behaviour for three different neurons. The plot is retrieved from Neil *et al.* [22].

As can be seen, the extension on the LSTM equations involves re-purposing the expressions that used to define $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$ in Equation 2.15 and Equation 2.17 respectively, to define a candidate state $\widehat{\boldsymbol{c}_j}$ and candidate output $\widehat{\boldsymbol{h}_j}$. Then, the final cell state and output, $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$, are redefined as the time gate selections between candidate values and preceding values. We see that unless the gate $\boldsymbol{k}_t$ is active, the PLSTM cell maintains its state and output over time, in effect remembering past input. Whereas if the gate is open, the cell state and output is modifiable. An illustration of a PLSTM cell can be seen in Figure 3.2.

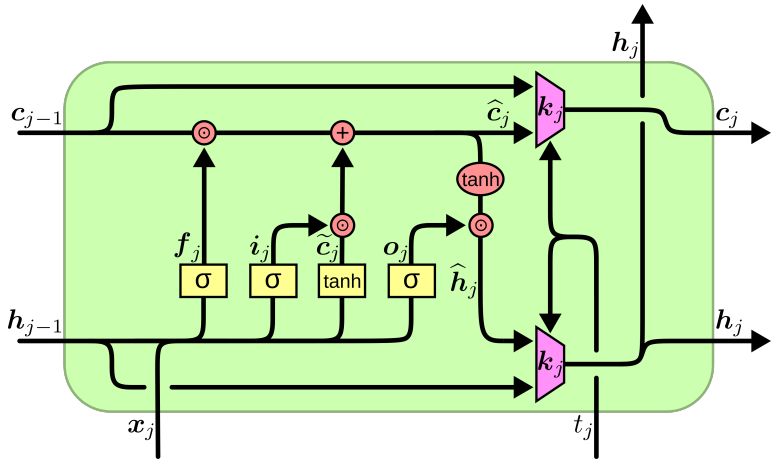


Figure 3.2: An illustration of how a Phased LSTM cell is constructed. Here we can clearly see how the phased version extends the LSTM model, with the addition of time gates that control the final cell output.

When combining many PLSTM cells in a layer, if the different cells have learned a wide selection of frequencies for openness, the network may keep some cells open for any irregular input frequency. In addition, some of them might learn to keep their state for a long time, thus remembering the past, while others can emphasize rapid state updating and thus keep track of frequently changing signals.

3.3 Clockwork RNN

As an attempted solution to the problem of modelling long-term dependencies in sequences, in 2014 Koutnik *et al.* [17] suggested a model called clockwork RNN. It differs from a regular RNN in that the hidden layer is subdivided into sets of neurons called modules that operate at different clock frequencies. Whereas some modules might recalculate output for every time step, others might freeze for longer periods. The authors use an exponential period selection scheme for the modules, where module i has period $T_i = 2^i$. This model performed better at modeling long dependencies than a plain LSTM in some experiments. The clockwork RNN model can be seen as a predecessor of the Phased LSTM model. The idea of having separate parts of a layer operate at differing frequencies is a commonality, and as future work the paper discusses letting the model learn appropriate periods on its own, as is done in Phased LSTM.

3.4 Continuous-time GRU

A recent attempt at handling discrete events in continuous time is the continuous-time GRU (CT-GRU), as proposed by Mozer *et al.* [29]. The authors extend the hidden state in traditional GRU to a set of hidden states or memory with different decay rates, to account for patterns over different time scales. The real-valued time stamp is given to the model for memory control. This scheme will result in different time horizons for different parts of the memory, similar to what is seen in a Phased LSTM model. Interestingly, the results from experiments show no significant improvement over traditional GRU with time as a simple input feature, and the authors describe it as essentially identical performance, and as a null result. They suggest GRU and CT-GRU can encode roughly the same solutions, given that CT-GRU has no more trainable parameters (in fact less). To us, this suggests GRU/LSTM can effectively handle time if given it as an input. Thus, alterations that restrict the network's trainable parameters to handle time stamp input in specific ways are less promising, and standard LSTM/GRU with time t or delta time Δt appended to x is an important baseline to beat.

Even though the results achieved in this paper are disappointing, it is worth mentioning because they show the potential difficulty in engineering solutions for handling time in specialized manners. It is also worth mentioning because, even though the method failed to show results, the motivation they provide for pursuing it is remarkably clear and well thought out. They present the inductive biases of convolutional neural networks as follows:

- *Spatial locality* - features at nearby locations in an image are more likely to have joint causes and consequences than more distant features.
- *Spatial position homogeneity* - features deemed significant in one region of an image are likely to be significant in other regions.
- *Spatial scale homogeneity* - spatial locality and position homogeneity should apply across a range of spatial scales.

They then go on to suggest desired inductive biases for models that are to be used on time series data. They do this by assuming that such models should hold inductive biases in the

time domain that are *isomorphic* to the ones possessed by CNNs in the spatial domain:

- *Temporal locality* - events closer in time are more likely to have joint causes and consequences than more distant events.
- *Temporal position homogeneity* - event patterns deemed significant at one point in time are likely to be significant at other points.
- *Temporal scale homogeneity* - temporal locality and position homogeneity should apply across a range of time scales.

They also list a non-isomorphic bias that they hypothesize to be helpful:

- *Temporal scale interactions* - sequences have different structure at different scales and these scales interact.

We feel that these inductive biases are reasonable and clear, and might provide guidance and motivation in future work on developing machine learning algorithms for the time domain.

3.5 Unsupervised and Semi-supervised Anomaly Detection with LSTM Neural Networks

In the introduction, we argue for the usefulness of RNNs in anomaly detection, predictive maintenance and process optimization. Ergen *et al.* [26] show a promising approach for utilizing RNNs along with SVDD or SVM algorithms for semi-supervised and unsupervised anomaly detection. This approach, as shown in their paper, is generic so that the RNN portion can be replaced with a better performing alternative.

Their approach is to take an arbitrary length sequence and feed it into a stacked LSTM (or similar) neural network which utilizes recurrent information, output a fixed length sequence which is then forwarded to an OC-SVM or SVDD algorithm. The final SVM or SVDD then outputs a decision identifying the sample as anomalous or normal. The LSTM architecture and SVM or SVDD parameters are jointly optimized. The authors propose a quadratic programming approach and a gradient-based approach. To apply the gradient-based method, the original OC-SVM algorithm must be smoothly approximated, and the authors show their approximation converges to the actual, original formulation. Their approach requires modification of OC-SVM and SVDD designed specifically for this task as exemplified above, but require no extensive modifications to the RNN architecture. Thus, we can utilize the improved characteristics of any approach similar to LSTM if they better suit the task, such as GRU (as shown in their paper) or the Phased LSTM architecture discussed in our thesis.

3.6 An Empirical Evaluation of Recurrent Network Architectures

A large portion of our work in this paper is on implementation and utilization of the Phased LSTM/GRU architecture. Phased LSTM is a proposed improvement on a small subset of

RNN algorithms, which is a subset of time series learning models. Is it reasonable to explore such a narrow improvement? The work in Jozefowicz *et al.* [18] motivates our decision further. The authors perform an extensive search through RNN architectures. They find that some models outperform GRU on some tasks, but no tested architecture significantly outperforms GRU on all tested tasks and their best-performing architecture resembles GRU. The search procedure follows. The search algorithm maintains a list of the 100 best architectures encountered, where performance of an architecture is the performance of its best hyperparameter setting. The top-100 list is initialized with only LSTM and GRU which have been evaluated for all considered hyperparameter settings. Then, at each step of its execution, the algorithm takes an architecure from its top-100 list and either evaluates it on new hyperparameters or creates a new architecture by mutating the chosen one. A brief overview of each option follows:

1. Select a random architecture from the top-100 list and evaluate it on randomly chosen hyperparameter settings on each of the three test tasks chosen by the authors, to update its performance. The stored performance estimate can be expressed as:

$$min_{task} = \frac{architecture's\ best\ accuracy\ on\ task}{GRU's\ best\ accuracy\ on\ task} \tag{3.6}$$

   Where best accuracy refers to the best hyperparameter setting. Taking the minimum over all tasks ensures the algorithms look for an architecture that works for every task.

2. Propose a new architecture by choosing one from the top-100 list and mutating it. First, this new architecture is tested on a simple memorization problem where a sequence of 5 symbols (out of 26, like the English alphabet) are read in sequence and reproduced in the same order. The task is easy for the LSTM, which solves it with 25000 parameters at all but the most extreme hyperparameter settings. The architecture is discarded if performance is less than 95%, but can be re-examined with new hyperparameter settings later. If the architecture passes the initial stage, it is run on the first task with 20 hyperparameter settings. If its performance exceeds 90% of the GRU's best performance, it is run with the same procedure on the second task and then the third. It can then be added to the top-100 list. If it does not reach the top 100, the hyperparameters are stored and if the same architecture is randomly chosen in the future, new hyperparameters will simply be run instead.

New architectures in the search procedure are created through random mutation. Candidate architectures are represented with a computational graph, similar to ones easily expressed in Theano or Torch (Tensorflow has a similar concept). Each layer is represented with a node, which takes M+1 nodes as inputs where M is the hidden state of the RNN at the previous timestep and the last M+1th input is the external input for the current timestep. It then outputs M (the hidden state) nodes to the next layer. The mutation is done by choosing a probability p uniformly from [0, 1] and independently applying a random chosen transformation to each node in the graph. with probability p. The transformations include:

1. If the current node is an activation function, replace it with any of sigmoid, tanh, ReLU or (scaled) linear activations.

2. If the current node is an element-wise operation, replace it with another (multiplication, addition, subtraction).

3. Insert a random activation function between the current node and one of its parents.

4. Remove the current node if it has one input and one output.

5. Replace the current node with one of its ancestors (reducing the size of the graph).

6. Replace the current node with the sum, product or difference of an ancestor of this node and the ancestor of another node.

The three tasks tested are basic arithmetic (output sum or difference of two numbers), predicting the next character in a synthetic XML dataset and language modelling on the Penn TreeBank dataset. The best models are also compared on a music dataset where notes have to be predicted. The formula for their best-performing mutated model (MUT1) is compared to the standard GRU below:

GRU:

$$\boldsymbol{z}_t = sigm(\boldsymbol{W}_{xz}\boldsymbol{x}_t + \boldsymbol{W}_{hz}\boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \tag{3.7}$$
$$\boldsymbol{r}_t = sigm(\boldsymbol{W}_{xr}\boldsymbol{x}_t + \boldsymbol{W}_{hr}\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{3.8}$$
$$\widetilde{\boldsymbol{h}_t} = tanh(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{3.9}$$
$$\boldsymbol{h}_t = \boldsymbol{z}_t \odot \widetilde{\boldsymbol{h}_t} + (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} \tag{3.10}$$

MUT1:

$$\boldsymbol{z}_t = sigm(\boldsymbol{W}_{xz}\boldsymbol{x}_t + \boldsymbol{b}_z) \tag{3.11}$$
$$\boldsymbol{r}_t = sigm(\boldsymbol{W}_{xr}\boldsymbol{x}_t + \boldsymbol{W}_{hr}\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{3.12}$$
$$\boldsymbol{h}_t = tanh(\boldsymbol{W}_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + tanh(\boldsymbol{x}_t) + \boldsymbol{b}_h) \odot \boldsymbol{z}_t + \boldsymbol{h}_{t-1} \odot (1 - \boldsymbol{z}_t) \tag{3.13}$$

The results were that GRU outperforms LSTM on all tasks except language modelling. Their best-performing model (MUT1) outpeforms GRU on all tasks except language modelling where it matched the GRU. LSTM significantly outperforms all other architectures on the language modelling task when dropout was allowed. While MUT1 was best on two of the music datasets, LSTM without input gates and LSTM without output gates was best when dropout was allowed.

We note that their best-performing model is very similar to the GRU, and that the LSTM is still superior in certain scenarios (notably with dropout for large networks). We also note that some architectures significantly outperformed GRU on certain tasks. To us this suggests GRU/LSTM are among the best-performing RNN architectures, and that improvements in performance on specific tasks can be found by specializing these architectures.

3.7 An End-to-End Spatio-Temporal Attention Model for Human Action Recognition from Skeleton Data

Typically, audio and visual data or skeleton data processed by neural networks is regularly sampled by the source sensor, but the recorded process (e.g. a human moving) occurs in continuous time. In practice, this and the phases in human movements means some video frames capture less meaningful information, or even carry misleading information, while some frames carry more discriminative information[12]. The problem of less meaningful frames also occurs in event-driven sequences in general, and especially when these are resampled to a periodic sequence, as the e.g. imputated frames in event-driven sequences are copies of earlier frames. The Phased LSTM network can be seen as implementing a type of temporal attention-based model where the attention at certain times is zero and no update of the network occurs.

The paper Song *et al.* [23] includes an example of a temporal attention-based subnetwork in their proposed network utilizing LSTM, used for human action recognition from skeleton data. The temporal attention module automatically pays different levels of attention $\boldsymbol{\beta}$ to different frames. Based on the output $\boldsymbol{z}_t$ of the main LSTM network and the temporal attention value $\boldsymbol{\beta}_t$ at each timestep $\boldsymbol{t}$, the scores for $\boldsymbol{C}$ classes are the weighted summation of the scores at all time steps

$$\boldsymbol{o} = \sum_{t=1}^{T} \beta_t \cdot \boldsymbol{z}_t \tag{3.14}$$

where $\boldsymbol{o} = (o_1, o_2, ..., o_C)^T$, T denotes the length of the sequence. Figure 3.3 illustrates how the temporal attention output β is incorporated into the main LSTM network. The predicted probability being the i^{th} class given a sequence X is

$$p(C_i|X) = \frac{e_i^o}{\sum_{j=1}^{C} e_j^o}, k = 1, ..., C. \tag{3.15}$$

As illustrated in Figure 3.4 found in the authors' paper, the attention module is composed of an LSTM layer, a fully connected layer, and a ReLU non-linear unit. It does soft frame selection. The activation of the frame-selection gate can be computed as

$$\beta_t = ReLU(\boldsymbol{w}_{x\sim}\boldsymbol{x}_t + \boldsymbol{w}_{h\sim}\boldsymbol{h}_{\tilde{t}-1} + b_\sim). \tag{3.16}$$

which depends on the current input $\boldsymbol{x}_t$, and the hidden variables $\boldsymbol{h}_{\tilde{t}-1}$ of the time step $t-1$ from an LSTM layer. The gate controls the amount of information of each frame to be used for making the final classification decision. The main LSTM network and the temporal attention subnetwork can be jointly trained to implicitly learn the temporal attention model.

The authors' approach effectively sets different importance to frames by allocating different attention weights to different frames, allowing the network to focus on frames

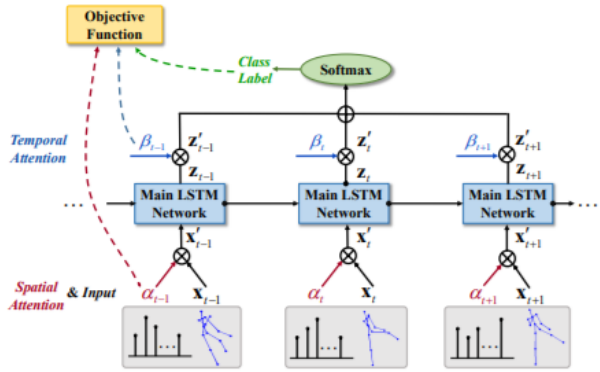


Figure 3.3: Illustration from Song *et al.* [23] showing how temporal attention is combined with the main LSTM network.

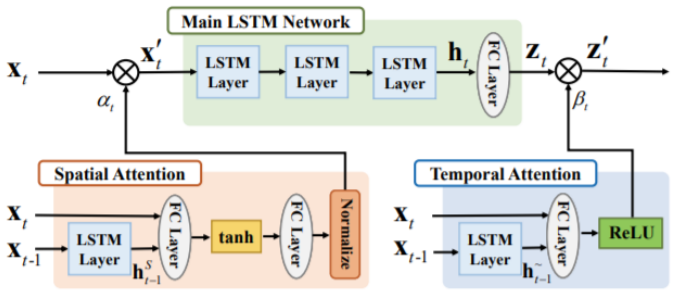


Figure 3.4: Illustration from Song *et al.* [23] showing the layout of the combined LSTM-based architecture, including how the temporal attention module is constructed.

deemed important and be less affected by less important ones, such as noisy frames, context frames as described in the paper, or possibly repeated frames stemming from resampling an event-driven time series to a regular frequency.

3.8 Skip RNN: Learning to Skip State Updates in Recurrent Neural Networks

Models like Phased LSTM and Continous time GRU try to learn time series in the general case, periodic or aperiodic. This is a promising approach, as resampling an event signal into a periodic signal in order to avoid aperiodicity comes with the side effect of increasing the length of time windows. This results in difficulties learning longer dependencies and increased computational cost. The Skip RNN model, introduced by Campos *et al.* [24], is an attempt to find solutions to these two problems. The general idea is for an RNN to learn when to update its hidden state and when not to do so. This is motivated by noting that choosing to not update the hidden state both preserves the models memory and also

avoids the need to *compute* new state updates, thus reducing the size of the computational graph.

This is done by augmenting the network with a binary state update gate $u_t \in \{0, 1\}$, that determines whether the state is updated or copied from the preceding step. At time t, a probability distribution for u_{t+1} is generated: $\tilde{u}_{t+1}$.Thereafter u_{t+1} is generated through a binarization of $\tilde{u}_{t+1}$. Whereas the paper uses the simple method $u_{t+1} = round(\tilde{u}_{t+1})$, other methods are proposed, like a stochastic sampling from a Bernoulli distribution. The updated equations for a generic RNN would look as follows:

$$u_t = f_{binarize}(\tilde{u}_t) \tag{3.17}$$

$$\boldsymbol{h}_t = u_t \widetilde{\boldsymbol{h}}_t + (1 - u_t)\boldsymbol{h}_{t-1} \tag{3.18}$$

$$\Delta\tilde{u}_t = \sigma(\boldsymbol{W}_p \boldsymbol{h}_t + \boldsymbol{b}_p) \tag{3.19}$$

$$\tilde{u}_{t+1} = u_t \Delta\tilde{u}_t + (1 - u_t)(\tilde{u}_t + min(\Delta\tilde{u}_t, 1 - \tilde{u}_t)) \tag{3.20}$$

Here $\widetilde{\boldsymbol{h}}_t$ is equivalent to the calculated $\boldsymbol{h}_t$ that would automatically be calculated in an ordinary RNN. We see that if $u_t = 0$, this value is not in need of calculation. It is also noteworthy that $\tilde{u}$ accumulates over time by the increment $\Delta\tilde{u}$, until it is reset to $\Delta\tilde{u}$ when a state update occurs. This encodes the intuition that the likelihood of updating the hidden state increases the more time passes since the last time it happened. Through this definition we can derive ahead of time when the next state update occurs:

$$N_{skip}(t) = \left\lceil \frac{0.5}{\Delta\tilde{u}} \right\rceil - 1 \tag{3.21}$$

This implies that no computing at all is needed when $u_t = 0$. The technique proposed is general enough that it can be applied to most RNN cell types, including LSTM and GRU. It also does not add complexity to the loss function, as it is fully differentiable. It is theorized that skipping steps completely is a more efficient way to achieve computational cost reduction than to reduce computation at each step. The Skip RNN achieves comparable performance to models like LSTM, on many problems, but at significantly lower computational cost.

3.9 Nested LSTMs

Although the Nested LSTM, as seen in Moniz & Krueger [34], holds common goals with the Skip RNN, the approach is different. The Nested LSTM also attempts to learn longer term dependencies than regular LSTMs, but it does so by attempting to improve the capabilities of the LSTM memory, or *cell state* $\boldsymbol{c}_t$. The expression for $\boldsymbol{c}_t$ in a standard LSTM is given in 2.15. The Nested LSTM computes this value as well, but does not use it in the role of $\boldsymbol{c}_t$ directly, but rather as input to an *inner* LSTM cell. The cell state of the outer cell, $\boldsymbol{c}_t$, is then defined as the *hidden state* of this inner cell:

$$\boldsymbol{c}_t = \widetilde{\boldsymbol{h}}_t \tag{3.22}$$

Here $\widetilde{h}_t$ is the hidden state of the inner cell. The intuition behind this structure is that the memory of the inner, nested cell might maintain information that is not currently relevant, but might become relevant at a later point in time. It is obvious that a Nested LSTM cell is more complex than a standard LSTM cell, with double the parameter count. But the model is not intended to compete with LSTMs in the same layer configuration. The nesting of LSTM cells is rather presented as an alternative to the practice of stacking LSTM layers. In this context a fair comparison would be between a one-layer Nested LSTM and a two-layer regular LSTM. This comparison is relevant because stacking more layers is one hypothesized method of gaining capacity for learning long-term correlations.

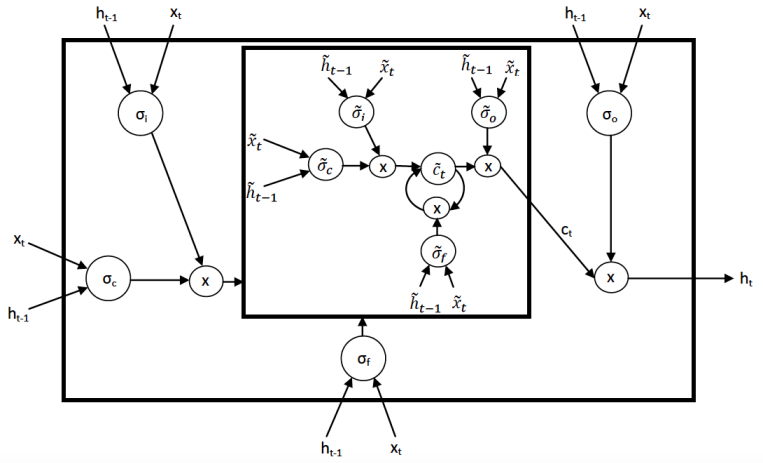


Figure 3.5: Illustration from Moniz & Krueger [34] showing the structure of an LSTM cell with one nested cell.

3.10 Other Notable Papers

3.10.1 Wider and Deeper, Cheaper and Faster: Tensorized LSTMs for Sequence Learning

He *et al.* [28] propose the Tensorized LSTM in which the hidden states are represented by tensors and updated via a cross-layer convolution. By increasing the tensor size, the network can be widened efficiently without additional parameters since the parameters are shared across different locations in the tensor. By delaying the output, the network can be deepened implicitly with little additional runtime since deep computations for each time step are merged into temporal computations of the sequence.

3.10.2 Dilated Recurrent Neural Networks

Chang *et al.* [25] proposes dilated recurrent skip connections, inspired by the dilated CNN, to tackle learning of complex dependencies in time, while avoiding vanishing and explod-

ing gradients, and improving the parallelization over regular RNNs. A *dilated recurrent skip connection* is the fundamental building block of the Dilated RNN. It is defined as:

$$\boldsymbol{h}_t = \hat{f}(\boldsymbol{x}_t, \boldsymbol{h}_{t-s}; \boldsymbol{\theta}), \tag{3.23}$$

where $s > 1$. They find that the Dilated RNN trains faster, requires less hyperparameter tuning, and needs fewer parameters to achieve state-of-the-art results.

3.10.3 Low-pass Recurrent Neural Networks – A memory architecture for longer-term correlation discovery

Stepleton *et al.* [35] propose a memory strategy that can extend the window over which backpropagation through time can learn without requiring longer traces. The architecture, called Low-pass RNN, allocate varying representational precision to events of different frequency/recency. This is accomplished through chains of low-pass filtering pools.

3.10.4 The Unreasonable Effectiveness of the Forget Gate

In van der Westhuizen & Lasenby [36] the JANET architecture is proposed. It is an LSTM variant with improved performance on long sequences. JANET is a significant simplification of the LSTM architecture, defined as follows:

$$\boldsymbol{f}_t = \sigma_f(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \tag{3.24}$$

$$\widetilde{\boldsymbol{c}_t} = \sigma_c(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \tag{3.25}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + (1 - \boldsymbol{f}_t) \odot \widetilde{\boldsymbol{c}_t} \tag{3.26}$$

$$\boldsymbol{h}_t = \boldsymbol{c}_t. \tag{3.27}$$

Here σ_f is the sigmoid function, and σ_c is the hyperbolic tangent. We see that this is an LSTM-like structure, but lacking the output gate and letting the forget gate also do the job of the input gate, not unlike in a what is done in the GRU architecture. With a tailored initialization, the JANET architecture outperforms the standard LSTM on a selection of tasks involving long sequences.

3.10.5 Learning the Joint Representation of Heterogeneous Temporal Events for Clinical Endpoint Prediction

Liu *et al.* [33] proposes Heterogeneous Event LSTM(HE-LSTM) for learning the joint representation of heterogeneous event sequences. This model is an extension of the Phased LSTM model we are considering in this thesis. According to the authors, one important application is clinical endpoint prediction, that is, predicting whether a disease, a symptom or an abnormal lab test will affect a patient in the future, from the data in the patients' history records. The salient change with regards to Phased LSTM is that the time gates are replaced by *event gates*. These are functions, not only of time, but of the event type the model is observing at a given time step. The event gate is defined as $\boldsymbol{j}_{s,t} = \boldsymbol{e}_s \odot \boldsymbol{k}_t$, where $\boldsymbol{k}_t$ is defined in 3.1, and $\boldsymbol{e}_s$ is defined as

$$\boldsymbol{e}_s = \sigma(W_{em}tanh(W_{ms}\boldsymbol{s} + \boldsymbol{b}_m) + \boldsymbol{b}_e). \tag{3.28}$$

3.11 Summary

Most of the papers in this section attempt to solve a few issues with current RNNs and related architectures. First, they are slow when compared to other state-of-the-art neural networks, largely due to deficiencies in parallelism. Second, their memory degrades over time, leading to issues with long-term dependencies in long sequences (hundreds of samples) and when important information is sparsely distributed in the input. Third, they are not designed to handle irregular time series. A common approach that applies to all of these issues is to selectively decide not to update a cell. Not updating a cell at a time step preserves its memory, reduces the amount of computation necessary, and if done carefully can handle irregular time series, for instance, by letting sets of cells operate at different regular frequencies until they collectively cover all relevant frequencies in the input data. Phased LSTM, Skip RNN and Clockwork RNN all perform some variation on this concept with different goals. Attention models that give a weight to each sample reduce the importance of an update instead of removing it completely. Continuous-time GRU and Nested LSTM are approaches that instead more directly try to enhance the memory of the network.

Chapter 4

Methodology

In this chapter we outline the technical work done in order to run all our desired experiments.

Section 4.1 explains the data pipeline, from raw data stored in factory databases, to data the model can process. Section 4.2 discusses the implementation of the model and related training script. Section 4.3 explains the layer stacking wrapper operation we added that was necessary for stacking PLSTM and PGRU layers in TensorFlow. Section 4.4 details our Phased GRU model based on GRU and PLSTM. Finally, Section 4.5 gives a summary of the chapter.

4.1 Data Pipeline

In this thesis we are handling data from the MNIST dataset, a synthetic sine wave dataset, and sensor data with associated meta data from an industrial scale dairy. The latter is not trivial as there is no readily available previous work, or solution, for this data. The data originates from relational databases using some form of SQL, organized by sensor and sensor location, with a time stamp for each sensor value. In addition, we have received related data, such as batch data which allows sensor readings to be associated with batches of products which have quality measurements such as later pH tests associated with them. To facilitate machine learning activities on this data, we implement a data processing pipeline that will prepare the data for a model to be trained. As input to the data pipeline, we have been given the result of simple queries for the interesting time frame and factory areas, supplied to the program as CSV files. In the CSV files for sensor data, readings are stored as a single value column with associated tag, timestamp and quality data per reading. We cast the input to suitable data types for preprocessing, though ultimately the TensorFlow model uses float32 for all normal inputs and float64 or float32 for time input. Faulty or missing readings are removed or replaced depending on the case. As each new reading represents the first change (above a threshold) in value for the sensor, forward filling can safely be used to upsample the signal without creating an incorrect representation, at a space and computational cost. Forward filling can also be used to replace missing or faulty values as

an estimate of the value, depending on the signal. Downsampling can be done to reduce the size of the dataset, reducing the cost of runs. Upsampling by imputation can be done to reduce time between samples. Resampling to periodic signals with the aid of interpolation is a technique that can be used to handle aperiodicity. All these sampling techniques will be done in several experiments, as described in detail in section 5.2. Values are separated to acquire one time series per sensor and divided into NumPy arrays suitable as input for the TensorFlow model. When data is loaded and prepared, we cache it on disk in a binary format for efficient load and so that expensive preprocessing steps are not repeated unnecessarily.

4.1.1 Combining Data Sources

Some data sources require combining, in the sense of associating time series readings with contextual data. For instance, we need to associate a recipe list for a product with its final quality metrics, and all related sensor data readings when the product was processed throughout the system. The dairy possesses dedicated databases to track a specific product batch through all processing steps. For any product batch that is created in the dairy, it is possible to query when it was produced, where it was produced, which ingredients were included in the production, how much of the ingredients were used, where the ingredients came from, and the respective batch ids of the ingredients. Another set of databases store lab test results associated with product batches. In the case of the curdling pH regression problem, detailed in subsection 5.2.3, we developed the necessary code to group relevant features with each cheese batch so that they can be used in the machine learning models.

4.1.2 Forecasting

In the case of the factory data, there are no labels available. To create a supervised learning problem, we can use the task of predicting future signal values based on current and past observations, without time-consuming manual labelling. The input is a sequence of values of some chosen length, ending at $\boldsymbol{x}_t$, and the target is a value $\boldsymbol{y}_{t+n}$ to be predicted. t is the final time in the input sequence and n is a parameter representing how far into the future the value we are predicting is. Note that $\boldsymbol{y}$ might take its value from the same time series as $\boldsymbol{x}$, with a slight offset, giving a simple "predict the future value of the input time series n steps ahead" problem. For this setup, we can use a sliding window method of input sequence creation. We let a sliding window representing the input sequence move over our training set with some step size to create samples of input sequences. Each target value per window is then sampled at $t + n$. The input sequences and targets can then be fed to the model according to the training strategy. The choice of prediction delay n is dependent on the use case of the prediction algorithm, and choosing larger n usually results in a more difficult problem. The aforementioned is a common setup for time series forecasting.

4.1.3 Bad Data

We have observed and confirmed a number of cases of bad data in the factory dataset. Bad data comes in the form of faulty sensor readings and, as some quality metrics are manually

recorded, mistakes in the measurements and manual input. There is also misleading data in the sense that the sensor is not measuring what is intended in all conditions. For instance, the sensor for the amount of liquid in a tank is measuring the height of the liquid, which can be impacted by a stirring implement which produces rapidly oscillating measurements for the liquid level. As another example, a pH measure reading can be intended for the liquid in the tank, but then be impacted by a cleaning process when the tank is empty. In some cases this information is useful, as machine learning algorithms can learn patterns not immediately apparent in a dataset, but it produces processes that are difficult to learn from and predict. In our dataset, depending on the task, we exclude some readings that are marked as poor (according to the Open Platform Communications Quality value), we manually exclude periods with particularly poor data quality in key measurements, and we automatically exclude some data values that are too extreme to be part of the normal state of the process. That said, we are constrained by time, resources and have limited access to expert knowledge, so the datasets still contain cases of bad data. As this is typical for real-world raw data, and an interesting use-case for neural network models that can handle noisy data, we present the results on the factory datasets with this caveat.

4.2 Training Script and Model

Primarily for forecasting tasks on factory data, we build a generic and configurable model, based on recurrent networks. We also write a training script that utilizes the input pipeline mentioned in section 4.1, feeding its output into our model for training. TensorFlow handles optimizing a graph through backpropagation automatically given the graph definition. We use Stochastic Gradient Descent (SGD) with common variants such as RMSProp[9] and Adam[16] available. We currently default to Nadam[21].

We use a standard three-way split of the total set of input samples, into a training, validation and test set. The three-way split occurs before any shuffling of the samples. This can cause problems if the nature of the data changes after some point in the input set, giving a different problem for instance for training and testing. It is possible to avoid this by creating samples with the temporal dependencies involved intact, and then shuffle these before the three-way split. However, in a real-life production scenario, the same issue will occur if a model trained on historic data encounters real-time data that is different in nature. In this case, the model must still perform predictions on the real-time data and can not shuffle it into the training set until later, so we consider this a real problem that must be handled gracefully by the network.

The evaluation depends on the network functionality. For a forecasting task, for instance similar to what is described in subsection 4.1.2, this is a form of regression problem where a mean loss function can summarize the error for all targets in the test set. The validation and evaluation after training is then as follows:

1. Run predictions from all samples in the test set.

2. Compare to target values.

3. Calculate mean loss according to for instance root mean squared error (RMSE).

In the case of classification we simply replace the final fully connected layer with a softmax layer, and use a cross-entropy loss function, while the rest of the program structure remains the same.

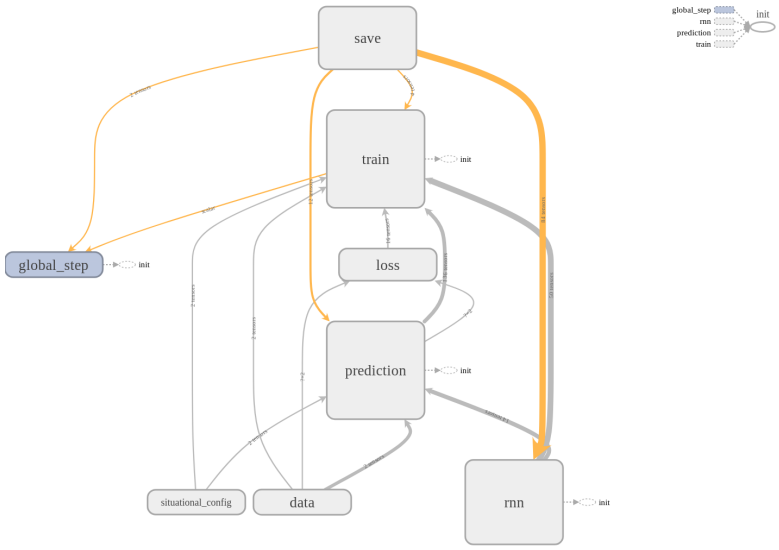


Figure 4.1: The TensorBoard graph representation of our RNN model.

Several loss functions are available, though we are defaulting to RMSE, one of the most popular metrics for regression problems.

The model construction uses a variety of typical neural network parameters as well as some specific to LSTM/GRU and PLSTM/PGRU models. The type of model can be specified, with LSTM, GRU, PLSTM, PGRU currently available. The number of layers and the width of each layer can be specified, changing the model capacity. The size of minibatches for the gradient descent can be specified. The number of epochs for training can be specified, and early stopping when validation set accuracy does not increase after a set amount of epochs has been added to prevent overfitting. For the phased network versions, the starting ratio of the on state versus the off state can be set, though this parameter is trainable at runtime. The model allows for dropout on the recurrent layers.

We use TensorFlow's implementation of PLSTM, with our own layer stacking wrapper operation as this was not available. The wrapper is described in section 4.3. We use our own implementation of PGRU, described in section 4.4. A simplified data flow graph for our TensorFlow model can be seen in 4.1.

4.3 Layer Stacking Wrapper Operation

The default recurrent network implementations in TensorFlow are provided with a wrapper class that easily enables stacking of recurrent layers. Although TensorFlow also has a Phased LSTM implementation, this implementation breaks with the regular RNN interface in that its input should be a tuple $(t, \boldsymbol{x})$ containing the signal time stamp in addition to the signal, whereas regular TensorFlow RNNs only take $\boldsymbol{x}$ as input. This deviation from the regular RNN interface breaks compatibility with the default layer's stacking wrapper. Since our PGRU implementation, detailed in section 4.4, follows the same interface as the TensorFlow Phased LSTM, we found it to be valuable to implement a layer stacking wrapper specifically for our phased architectures. The implementation itself is not very extensive. A simple sub-classing of the original layer stacking wrapper allowed us to mend the function serving input to the recurrent layers. It is worth mentioning because it adds useful functionality to TensorFlow that was lacking, and is shown to be beneficial for the phased models in subsection 5.1.3. The implementation of this wrapper can be found in Appendix B.

4.4 Phased GRU

Inspired by the the novel Phased LSTM idea, we decided to see if we could apply the same modification to a GRU. Previous work has shown that for periodic, synchronous data, GRUs usually perform on par or above LSTM in terms of accuracy, but at faster training and inference times[15,18]. If this performance relation between LSTM and GRU holds for their phased variants on relevant data, we can expect a PGRU to perform similarly to a PLSTM on most tasks, but with benefits in training and inference speed. At the time of implementation, we could not find any references to a Phased GRU in related literature, so the model proposed is our own work.

The PGRU we propose is derived from GRU in a way closely resembling how PLSTM is derived from LSTM. We utilize the time gates $\boldsymbol{k}_t$ to select between a proposed hidden state value $\widehat{\boldsymbol{h}_j}$ and $\boldsymbol{h}_{j-1}$. Since GRU does not have a cell state and only propagates $\boldsymbol{h}$ through time, the cell utilizes the time gates only in one operation, as opposed to PLSTM, where it is applied in the selection of both $\boldsymbol{c}_j$ and $\boldsymbol{h}_j$. The relevant update equations for Phased GRU are:

$$\widehat{\boldsymbol{h}_j} = \boldsymbol{z}_j \odot \widetilde{\boldsymbol{h}_j} + (1 - \boldsymbol{z}_j) \odot \boldsymbol{h}_{j-1} \tag{4.1}$$

$$\boldsymbol{h}_j = \boldsymbol{k}_j \odot \widehat{\boldsymbol{h}_j} + (1 - \boldsymbol{k}_j) \odot \boldsymbol{h}_{j-1}, \tag{4.2}$$

were $\widetilde{\boldsymbol{h}_j}$ is the aperiodic equivalent of $\widetilde{\boldsymbol{h}_t}$ defined in Equation 2.20. A data flow graph of the proposed model can be seen in Figure 4.2.

Based on the model proposed here, we implemented the Phased GRU as a TensorFlow operation, drawing inspiration from the GRU and PLSTM operations already implemented in the TensorFlow code base. Our implementation of PGRU can be found in Appendix A.

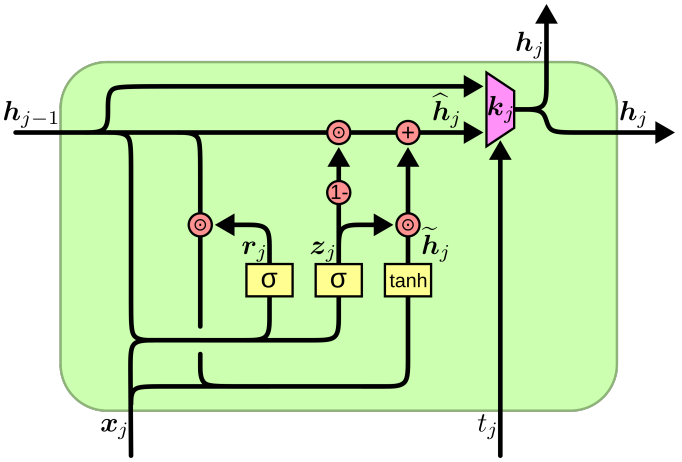


Figure 4.2: A Phased GRU cell data flow graph. We see how the model is similar to a GRU model, but extended by the addition of the time gate and its additional inputs, determining the final hidden state output.

4.5 Summary

As with any real-world machine learning problem, there is significant overhead in creating the datasets that are run through our models to present the final results. We created a data pipeline to efficiently load, cache and preprocess raw data from CSV files from the factory database. We included logic for combining data from different databases according to product batches, by utilizing the batch system the factory uses. We then create datasets primarily in the form of forecasting problems, which are supervised learning problems that avoid manual labelling, but also a dataset for predicting or classifying the quality metrics of batches for a certain product. We have observed, verified, and excluded bad data, but recognize we are unlikely to have caught all cases of it. For the tasks we test, gracefully handling bad data is crucial, so we see this as part of the learning problem. Our training script is very typical for machine learning problems. It is a graph-based approach written in TensorFlow, which takes as input the output of our data pipeline, with common optimizers for neural networks available. To test more aspects of the phased concept, we have included code for our PGRU model and for stacking layers of phased models in general in TensorFlow.

Chapter 5

Experiments and Results

5.1 Benchmark Experiments

Having implemented non-trivial new TensorFlow operations in the `PhasedGRUCell` (Appendix A) and `MultiPRNNCell` (Appendix B), we deemed it important to make sure they were working as intended while testing their performance. A sensible reference point would be previous experiments done on the the performance of the Phased LSTM. The paper describing Phased LSTM details some experiments that could be used for testing. One of these, classification of aperiodically sampled sine waves, exists as an open source TensorFlow implementation[38]. We implement similar code that is tailored for our system and model interface. Another possible point of comparison is the sequential MNIST classification benchmark, one implementation of which exists open source, credit to Philippe Rémy[43]. We again implement this benchmark for our own system and model interface. These open source benchmarks are well-suited for illustrating performance differences between the models we are working on in a way that is reproducible, as well as giving a way to compare our implementation with others on the same models, making sure there are no obvious flaws in our implementation.

5.1.1 Sequential MNIST

We perform experiments on the MNIST dataset, using classification as the reference task. The MNIST dataset consists of images of handwritten digits, 28 pixels high and 28 pixels wide. Some examples images can be seen in Figure 5.1. To turn this data into a challenging classification problem for RNNs, we unfold them into one-channel sequences of length 784, such that the RNN iterates over the pixels in the image, row by row. We compare LSTM, GRU, PLSTM and PGRU, both with one and two layers, under equivalent conditions. We measure training loss, training accuracy, and training speed over 3000 batches of 256 examples. We also measure the inference speed of the final trained model over a single batch. We average performances over five runs for each experiment.

Our models consist of an Elman-connected recurrent part, feeding into a 10-unit soft-

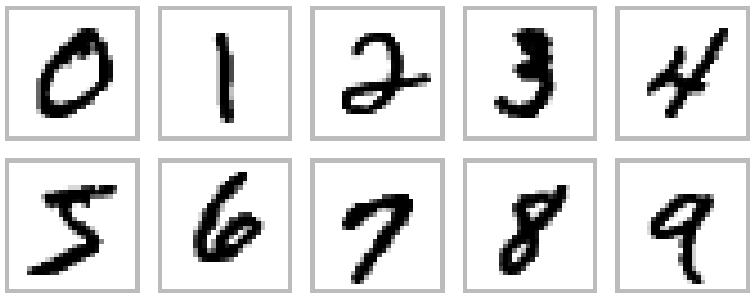


Figure 5.1: Some example images from the MNIST dataset. Image retrieved from the TensorFlow web page[44], under Creative Commons Attribution 3.0 License[39].

max layer, outputting a probability distribution for each digit class. The recurrent part of the network can consist of one layer of 32 cells, or two layers of 32 cells, each only passing state information onto itself through time. The models are trained using the Adam optimizer with initial learning rate 0.001, and cross-entropy loss. There are slight differences in the phased models and the normal recurrent models. Whereas the LSTM and GRU model get only the pixel intensity value as input for each step, the phased models also get the ordinal number of the pixel as input for their time gates. Since the distances between samples are regular, the LSTM and GRU models should not be at a disadvantage due to the lack of ordinal numbers as features. Similarly, when stacking recurrent layers, LSTM and GRU use TensorFlow's native layer stacking wrapper `tensorflow.contrib.rnn.MultiRNNCell`[49], whereas Phased LSTM and Phased GRU use our `MultiPRNNCell` wrapper.

5.1.2 Model Comparison - One Recurrent Layer

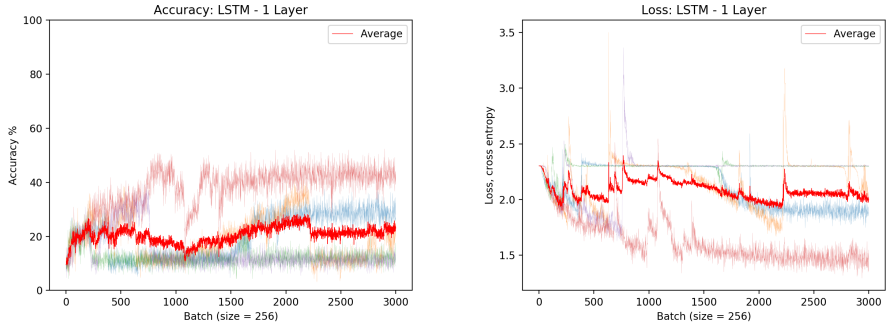
We compare our one-layer implementations of LSTM, GRU, PLSTM, and PRGU. We consider measures of efficiency both in terms of data efficiency and time complexity.

Data Efficiency

We examine the cross-entropy loss and accuracy of models as a function of observed training data.

LSTM In Figure 5.2 we see the accuracy of the one-layer LSTM over 5 runs. During one run, the accuracy stabilizes around 40%, but in general we observe that the model struggles to learn much from the examples it observes. It seems reasonable to conclude that the model might not have the capacity to properly learn the target function.

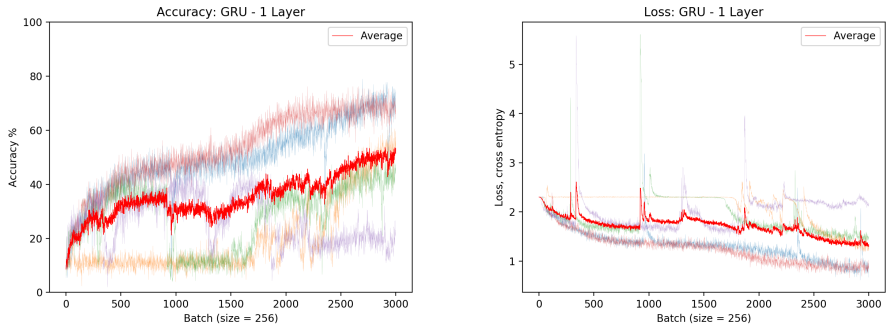
GRU Our one-layer GRU model seems to fare a little better. Observing the accuracy and loss displayed in Figure 5.3, we see that the model in some cases manages to steadily learn, still improving by the time it reaches around 70% accuracy, as the training run finishes.



(a) Accuracy **(b)** Loss

Figure 5.2: Training accuracy and loss for a one-layer LSTM over five runs.

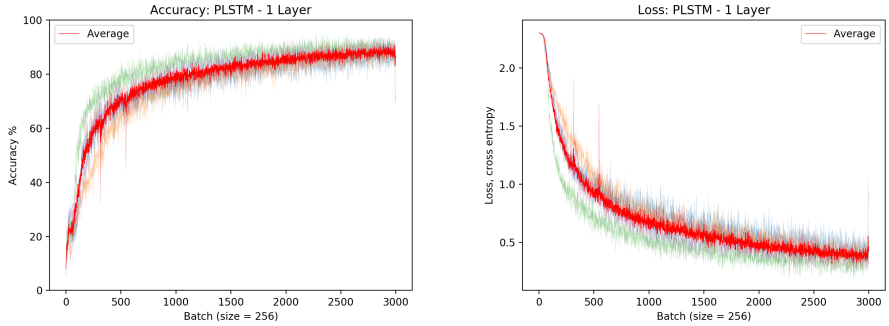
Other times the model learns little. Considering that GRUs and LSTMs are conjectured to perform similarly on most problems, it could be that both models in fact have the capacity to learn the target function, but might need considerable hyper-parameter tuning from the TensorFLow defaults, but occasionally the models perform quite differently.



(a) Accuracy **(b)** Loss

Figure 5.3: Training accuracy and loss for a one-layer GRU over five runs.

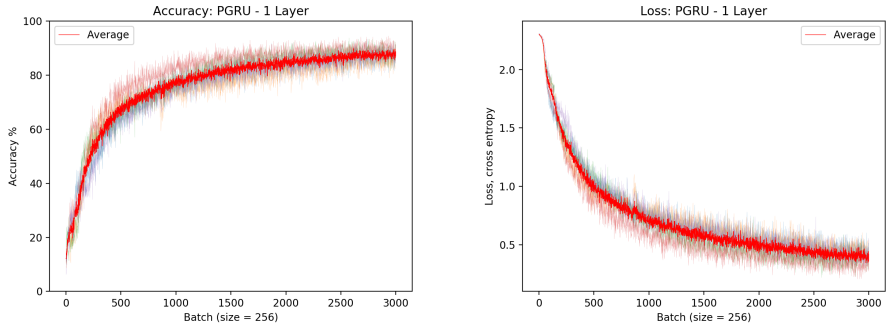
PLSTM We observe that the PLSTM model performs quite well on the task, considering it is a recurrent network trying to do image classification. As can be seen in Figure 5.4, every run is consistently able to learn from the data, and the average accuracy approaches 90% by the end of training. It is clear that the model has some advantage over both the regular LSTM and GRU. The most likely explanation is that a 784 step sequence is long enough for a regular gated cell to start struggling to maintain memory of the earliest significant information it has encountered when reaching the end of the sequence.



(a) Accuracy **(b)** Loss

Figure 5.4: Training accuracy and loss for a one-layer PLSTM over five runs.

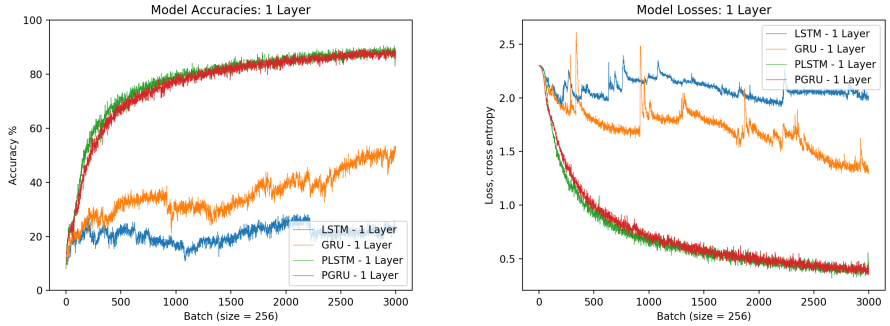
PGRU We note that our PGRU implementation performs similarly to what was the case for the PLSTM model. In Figure 5.5 we see the same pattern: all runs learn relatively quickly, reaching an average accuracy around 90%. This suggests to us that the PGRU has similar suitability as the PLSTM on this problem, and significantly better performance than the regular gated models.



(a) Accuracy **(b)** Loss

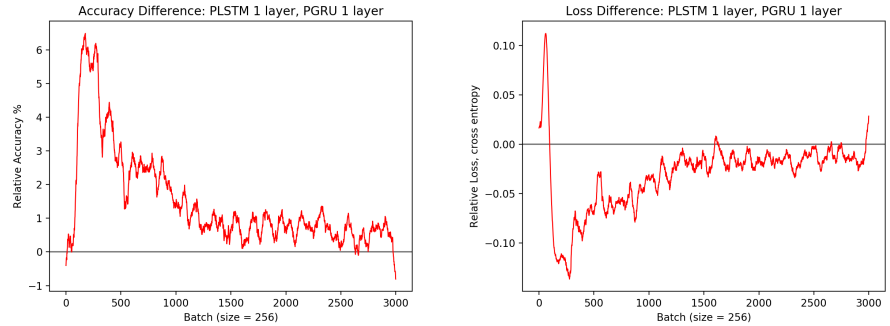
Figure 5.5: Training accuracy and loss for a one-layer PGRU over five runs.

Comparison To see the differences in performance of the models more clearly, we plot the average performances in Figure 5.6. We see the phased models dominating their non-phased equivalents, needing less training data to reach much higher performances. An interesting detail is that the PGRU on average seems to perform just slightly worse than the PLSTM. We show a smoothed plot of this difference in Figure 5.7. It is hard to say if this is just an artifact from little statistical data, something specific to this problem, or a general trend.



(a) Accuracy **(b)** Loss

Figure 5.6: Average training accuracy and loss for the one-layer models over five runs, as a function of training data.



(a) Accuracy Difference **(b)** Loss Difference

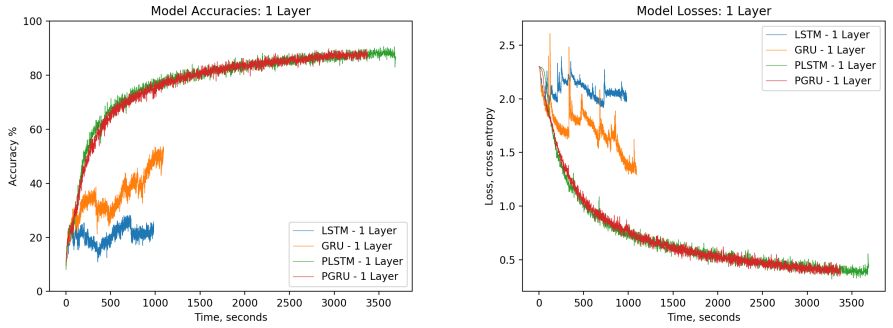
Figure 5.7: Difference in performance of one-layer PLSTM compared to a one-layer PGRU.

Time Complexity

We consider the time cost of training models, and performing inference tasks with them.

Training We want to examine not just performance as a measure of accuracy and loss in relation to the training data observed, but also in relation to the *time* needed to reach adequate performance levels. In Figure 5.8 we see the same performance graphs as earlier, but this time plotted over the time. It becomes immediately clear that the PLSTM and PGRU have a much higher time cost associated with training than LSTM or GRU. This can be the result of many factors. The phased models are more complex than LSTM or GRU, but could also be punished because their time gate uses non-standard operations that are not yet well supported by the underlying TensorFlow software stack or the underlying hardware it runs on. We still note, however, that even in the context of accuracy at a given time, the phased models consistently outperform the regular models. We also see that the

difference between the PLSTM and the PGRU is smaller in the context of time, because the PGRU trains slightly faster as a function of time.



(a) Accuracy **(b)** Loss

Figure 5.8: Average training accuracy and loss for the one-layer models over five runs, as a function of time.

Inference A consideration when implementing machine learning algorithms in production settings, is their time performance when doing inference. If the trained model is to be stored and used over time, this can be even more important than training time performance. In Table 5.1 we show the inference times for our one-layer models for one batch. We see that the GRU is clearly faster at inference than the other models. The PGRU is also a little faster than the PLSTM, but the phased models are several times slower than the standard gated units.

Table 5.1: Batch (256 examples) inference times for the one-layer models, five batch average.

	LSTM	GRU	PLSTM	PGRU
Inference Time - 1 Layer	0.1029 s	**0.0777 s**	0.5112 s	0.4545 s

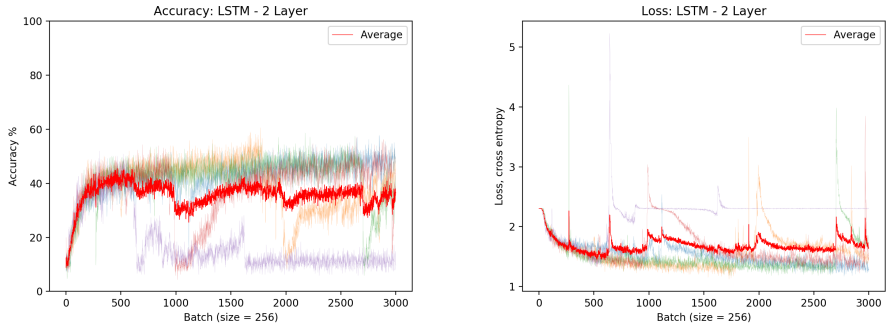
5.1.3 Model Comparison - Two Recurrent Layers

We compare our two-layer implementations of LSTM, GRU, PLSTM, and PRGU. We consider measures of efficiency both in terms of data efficiency and time complexity.

Data Efficiency

We examine the cross-entropy loss and accuracy of models as a function of observed training data.

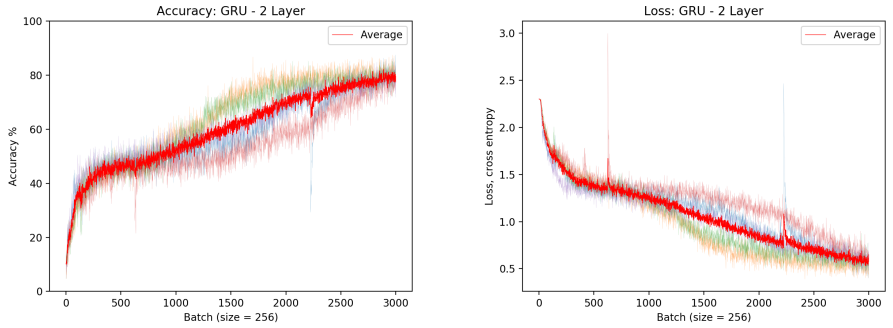
LSTM In Figure 5.9 we see the accuracy of the two-layer LSTM over 5 runs. Compared to the one-layer LSTM, the average accuracy is higher. All of the runs reach around 40% accuracy, but they repeatedly drop a lot lower for considerable amounts of time. One run is unable to start learning again after dropping to 10% accuracy - near equivalent to random guessing.



(a) Accuracy **(b)** Loss

Figure 5.9: Training accuracy and loss for a two-layer LSTM over five runs.

GRU The two-layer GRU, with accuracy and loss shown in Figure 5.10, also improves on its one-layer counterpart. Giving the GRU two layers results in all runs successfully learning. All runs reach around 80% accuracy by the final batch, and seem to possibly still be learning at that point.

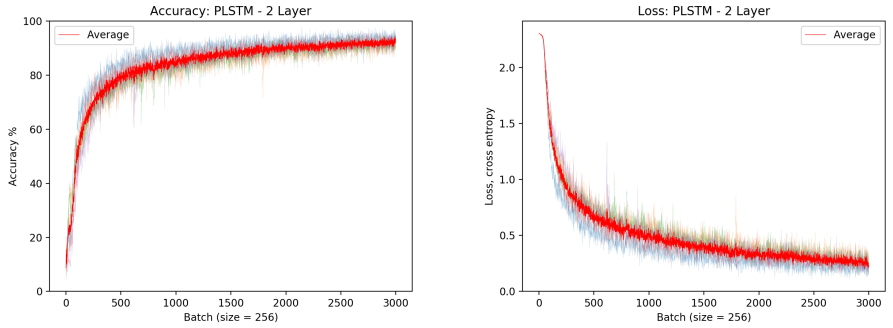


(a) Accuracy **(b)** Loss

Figure 5.10: Training accuracy and loss for a two-layer GRU over five runs.

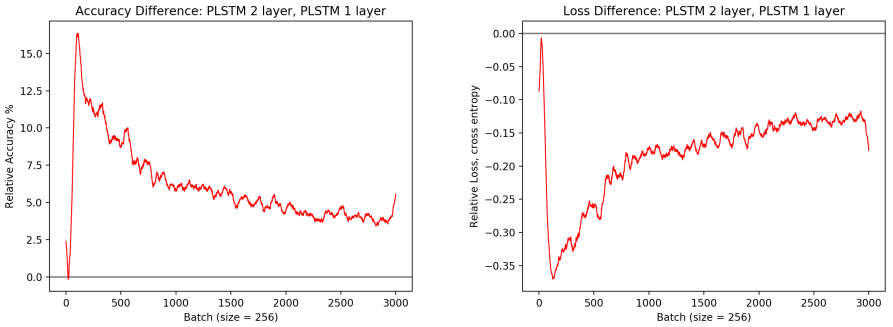
PLSTM We note that PLSTM still performs well when given two layers. It is hard to judge by the performance shown in Figure 5.11 whether we see an improvement over

the one-layer PLSTM. Accuracy in this case is also close to 90% by the end of training. We plot the smoothed relative performance of the one-layer and two-layer PLSTM in Figure 5.12, and see that there is an increase in accuracy as a result of adding one layer. This suggests our layer stacking wrapper is functioning correctly.



(a) Accuracy **(b)** Loss

Figure 5.11: Training accuracy and loss for a two-layer PLSTM over five runs.

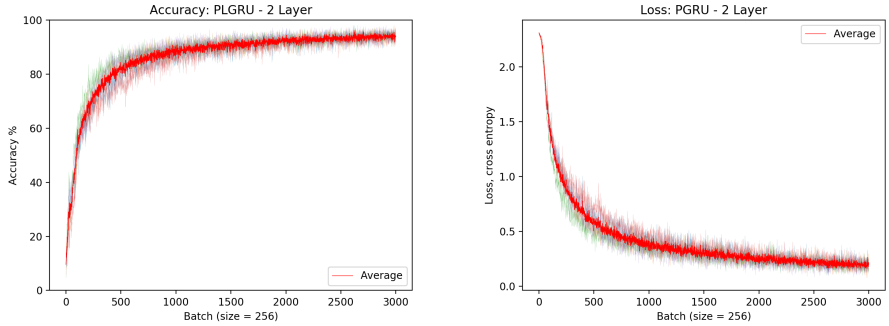


(a) Accuracy Difference **(b)** Loss Difference

Figure 5.12: Difference in performance of two-layer PLSTM compared to a one-layer PLSTM.

PGRU As was the case for the PLSTM model, we observe that the PGRU model also benefits from the addition of a layer. Performance can be seen in Figure 5.13.

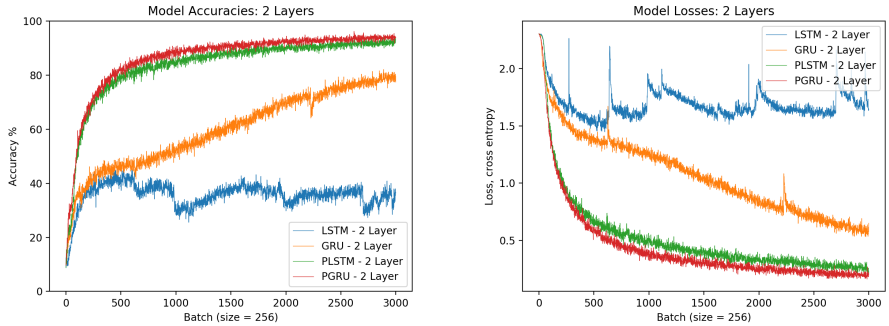
Comparison To see the differences in performance of the models more clearly, we plot the average performances in Figure 5.14. We see the phased models still dominating their non-phased equivalents, when using two recurrent layers. This time, however, we see that the GRU model is not underperforming as much as was the case with one recurrent layer. In light of that, it is interesting to note that the LSTM model is still underperforming. We



(a) Accuracy **(b)** Loss

Figure 5.13: Training accuracy and loss for a two-layer PGRU over five runs.

also see that PGRU is performing slightly better than the PLSTM, opposite of what was the case for the one-layer models. This can be seen in detail in Figure 5.15.



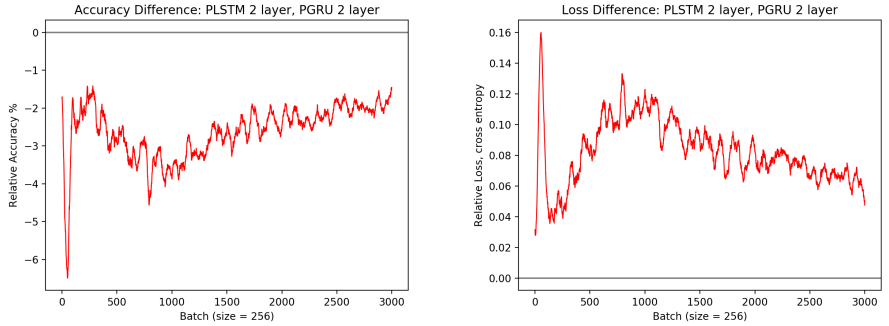
(a) Accuracy **(b)** Loss

Figure 5.14: Average training accuracy and loss for the two-layer models over five runs, as a function of training data.

Time Complexity

We consider the time cost of training models, and performing inference tasks with them.

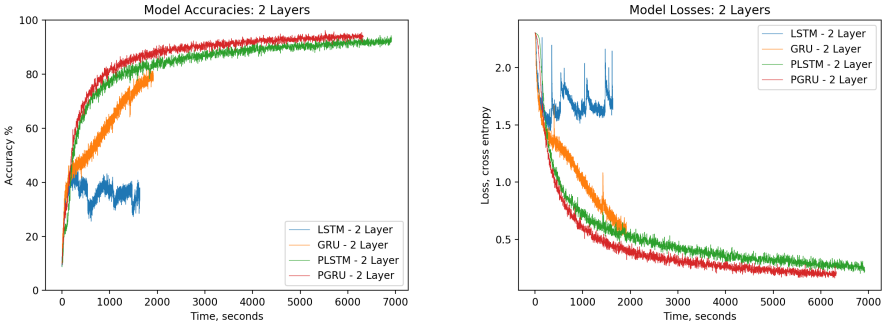
Training We are not surprised to see the time cost is near double for all models when adding a recurrent layer. This can be seen in Figure 5.16 when compared to Figure 5.8. The final softmax layer also contributes to the cost, but only executes at the final step of the sequence that is being classified, and is significantly less complex in structure than our recurrent layers. When seeing the performance as a function of time, it illustrates that the GRU is not too far behind in performance compared to the phased models. We note,



(a) Accuracy Difference **(b)** Loss Difference

Figure 5.15: Difference in performance of two-layer PLSTM compared to a two-layer PGRU.

however, that the GRU sees more training examples than the phased models during this time, and as such would be more reliant on data augmentation to avoid overfitting, and that phased models are yet to be optimized for speed.



(a) Accuracy **(b)** Loss

Figure 5.16: Average training accuracy and loss for the two-layer models over five runs, as a function of time.

Inference We show the inference times for our two-layer models in Table 5.2. We see that the GRU is still the fastest model at inference, but by smaller margins. The PGRU still holds the edge over PLSTM, but the phased models are still an order of magnitude slower than the normal gated cells.

Conclusions

MNIST classification is a problem that we know the qualities of phased models are well-suited for, compared to normal recurrent models. Considering that our implementation

Table 5.2: Batch (256 examples) inference times for the two-layer models, five batch average

	LSTM	GRU	PLSTM	PGRU
Inference Time - 1 Layer	0.1765 s	**0.1713 s**	1.0006 s	0.9983 s

of the phased GRU performed on par with the Phased LSTM, we can conclude that our implementation is not obviously flawed. We also confirm that the *idea* of a Phased GRU is warranted, at least on some problems. We also confirm that our layer stacking wrapper operation functions, and facilitates the building of multi-layer phased models. Taking into account the performance of the two-layer GRU on this problem, we will not rule out that regular recurrent networks could also perform adequately on this problem, but would then probably require significant hyper-parameter tuning and larger networks. We have seen that the Phased GRU is slightly less costly in training and inference time, compared to the Phased LSTM. This is unfortunately dwarfed by the extreme time performance penalty of our phased models compared to regular gated recurrent networks. We note that this can be the result of lack of optimization on many layers, from implementation and down to hardware, some of which are mentioned in the original PLSTM paper[22]. We see a potential for at least partial remediation in the future, should phased models be proven to have significant value.

5.1.4 Aperiodic Frequency Classification

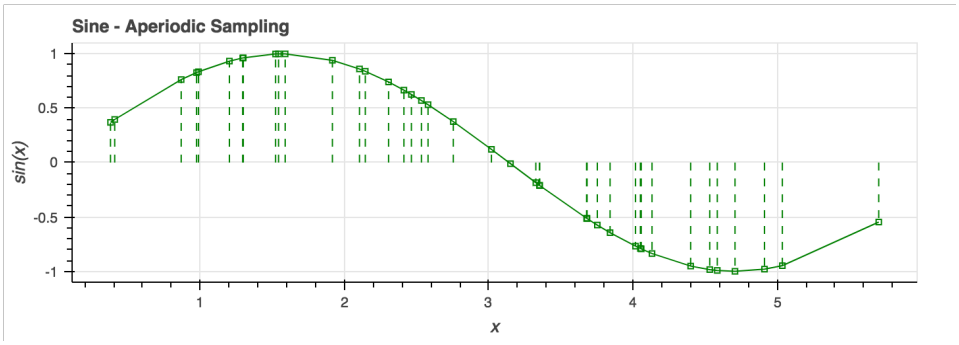


Figure 5.17: An aperiodically sampled sine function.

We have determined that the PGRU and PLSTM models are well suited for classifying long sequences. What about aperiodic time series? Inspired by an open source PLSTM benchmark script[38], we test our phased models and the non-phased models LSTM and GRU on a task that consists of classifying sinusoidal signals. We have a frequency range (5-6 Hz) that constitutes the class of signals we want to identify. All other frequencies (ranges 1-5 and 6-100 Hz) are considered negative examples. Frequencies are picked with a uniform distribution in the ranges containing positive and negative samples, such that there is an equal amount of both classes. The signals have varying lengths from

50 to 125 samples, and the samples are acquired not through a fixed, periodic sampling rate, but rather a random, uniform sampling over the length of the signal, as illustrated in Figure 5.17. The phased models get an input tuple $(\boldsymbol{t}, \boldsymbol{x})$, where $\boldsymbol{x}$ is the input vector, consisting of samples, and $\boldsymbol{t}$ are the associated time stamps. The LSTM and GRU models, on the other hand, are served a joint input matrix, where each $\boldsymbol{x}_j$ consists of an x value and the associated time stamp t. In other words, the models get the same data, but the phased models are able to use the time stamp information in their own specialized way. We train a one-layer model with 100 units for each cell type. The results of 19 training runs per model can be seen in Figure 5.18. We see that all models are able to learn to discriminate

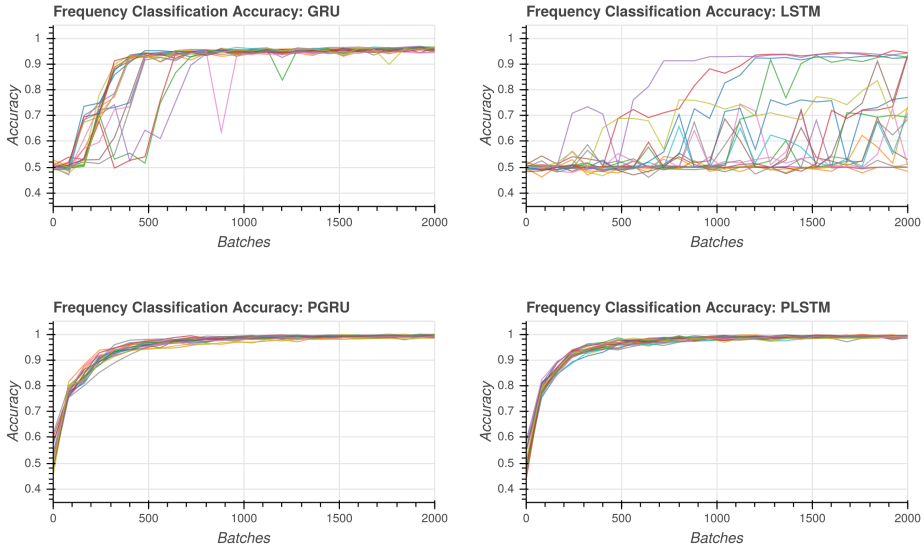


Figure 5.18: Validation accuracy for our four considered models on the frequency classification task. 19 runs per model, 32 examples per batch.

between the signals to some extent. But whereas the phased models consistently are able to learn quite fast, we see that GRU and particularly LSTM struggle with this task. It is not, however, out of the question that they could reach better performances given more time.

Table 5.3: Median test accuracy for models evaluated at the point of best validation accuracy.

	LSTM	GRU	PLSTM	PGRU
Accuracy	0.7312	0.9594	0.9922	**0.9937**

Conclusions Taking into account the results from the testing done both on the MNIST dataset and the aperiodically sampled frequency discrimination task, it seems that both

the phased LSTM and the phased GRU models are useful tools in *classification* tasks on both long and aperiodically sampled time series. If we are to solve classification tasks on process data, the PLSTM and PGRU models might be good tools to use.

5.2 Factory Data Experiments

Our goal in this section is to explore the characteristics of the baseline and phased models on real-world, event-driven data. We will run experiments on several datasets created from real-world data from sensors in a food production facility. Since the data is recorded in an event-driven manner, the resulting raw time series is aperiodic and, when multiple time series are combined, asynchronous across channels. The standard TensorFlow implementation of RNN models require input data in all channels at every step, so we forward fill data to create a synchronous, but still aperiodic time series. All models will then be trained on several versions of the data.

5.2.1 Experiment Overview

In previous sections we established that the phased models PLSTM and PGRU have a non-trivial advantage over LSTM and GRU on certain problems by using various benchmark datasets for comparisons. Next, we wish to examine the performance on real-world data believed to have similar traits. The main classes of cases we compare are: Regular frequency, created through imputation and interpolation, with Δt appended to x; irregular frequency without imputation, but potentially with downsampling and finally irregular frequency with imputation as phased models have shown promise on redundant data. *Irregular* in this section refers to irregular delta time between samples. As the datasets have been created from aperiodic, asynchronous data this irregularity is due to some combination of the two traits after preprocessing. We intended to run tests without Δt appended, but preliminary tests showed GRU and LSTM are already much faster with Δt included, so removing time information from these models seems unrealistic and the tests are omitted to focus on more interesting comparisons.

Data point imputation

As the smallest Δt found in the dataset is very small, in the order of milliseconds, a full upsampling to a periodic signal at this frequency is infeasible for practical applications. Instead, resampling, upsampling and downsampling must be done to some degree, through data point imputation where we forward fill values and one of several resampling techniques such as resampling with interpolation taking time information into account. This process costs processing time on its own, but more significantly due to additional data points having to be processed by the models, so performance relative to how costly the model is to run for different preprocessing techniques is important.

We first performed some preliminary testing on the first process dataset available to us, which consists of sensor data from a relatively simple tank for raw milk storage. Having gotten promising prediction results on the data using LSTM and various upscaling and rescaling techniques, we were hopeful that the PLSTM or PGRU might perform even

better. On the identically preprocessed data, this was *not* the case, with the PLSTM and PGRU performing with worse accuracy. It could mean that the phased models are less adept at solving prediction tasks, compared to classification tasks, or it could mean that we simply have not found the optimal working conditions for these models on prediction tasks. One of the benefits that help phased models on the classification of long sequences, is their ability to better avoid memory loss over time. When doing prediction, it can be argued that the most significant information in the time window that is used as input is the final values. Consequently, one explanation for what we are seeing could be that the LSTM and GRU are more than adept at extracting relevant information from the time window, and the phased models just represent added, unnecessary complexity, at least on data that has been preprocessed to fit the non-phased models.

Aperiodic data

As our method for creating forecasting tasks is based on the input data, different preprocessing methods give different targets for the neural network. The targets for the phased models should align closely in value and time stamp with the targets in the non-phased models' dataset for comparison. Real forecasting tasks will typically have a set time interval for forecasting. With these ideas in mind, we set the target based on time stamp instead of sample count. Due to differences in the data we still do not intend to compare models across tasks in the factory data section, we primarily compare models within each task. It will be interesting to see if the phased models can utilize the signal with less costly preprocessing to generate good predictions. In our experiments we use downsampling even for the aperiodic case, as preliminary tests on maximum resolution gave poor results for the very costly processing of microsecond resolution sensor data, when predictions are expected to be in the range of several minutes and input windows should be enlarged to match. Cycles in the dataset typically occur over hours or days, so processing these at microsecond resolution is infeasible for our computing resources. Some rapid changes occur over minutes, and so failing predictions in these cases is expected at reduced resolution.

Downsampling

Depending on the accuracy and sample resolution required for the task, downsampling is a common technique to reduce the cost of running experiments. It is especially useful when fine-grained information is not sought. For instance, millisecond frequency is unlikely to be necessary for a task forecasting hours into the future. Various methods can be used, but typically samples are grouped into bins at some regular frequency, for instance grouping all samples in each minute. Then some average can be computed, or as in our case we use the last value found in a bin. Since the bins are created at some regular frequency and the irregularity within a bin is removed, this will necessarily make the final sequence more regular. As placing bins at irregular frequencies is unusual for real tasks, we use normal regular frequency bins for downsampling but include a case with minimal downsampling. This case will be at the minimum downsampling frequency of 1 second all data is initially processed to. An illustration of downsampling making a signal more regular can be seen in Figure 5.19. Note how there is some irregularity in the downsampled signal due to the empty bin near $x = 3$, but that the signal is otherwise regular.

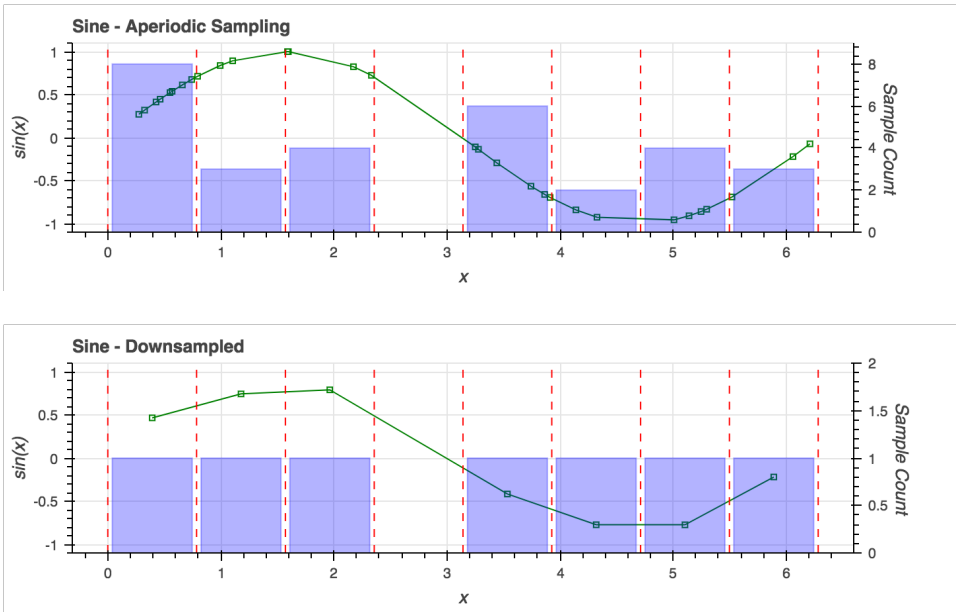


Figure 5.19: Example of traditional downsampling methods introducing regularity to the data.

Hyperopt

We will be using hyperparameter search to better represent real-world performance of the models, and to prevent poor hyperparameters from inhibiting the performance of the phased models, as good hyperparameters for the phased models are not well-documented. Additionally, this is done to explore how hyperparameters affect the phased models in general. For choosing hyperparameters we are utilizing the implementation of the Tree-of-Parzen-Estimators (TPE) algorithm[7] provided in Hyperopt[10]. Models have been run on experiments through hyperopt at least 100 times in the hyperparameter search, but most have been run more in the process of exploring which parameters and parameter ranges are relevant. Note that the upper limit on network size has been set due to runtime constraints. All runs with smaller networks than the lower limit performed worse and so are not included. The results were fairly uninteresting, as we found the largest networks to perform best, and learning rate to only marginally impact results. Learning rate at 0.002 is the suggested default for the Nadam optimizer that was used[21]. We found values slightly above the suggested default to perform best, in the range of 0.0023-0.0036 on our tasks. Unless stated otherwise, learning rate has been set to 0.0025 as it gives the most stable results in our tests, and network cell sizes set to 2 layers of 64 cells. We found marginal improvements with more cells, but have set 64 as maximum to allow a large amount of runs for an accurate representation of network performance. Note that we use early stopping, where after the patience limit of deteriorating validation performance is reached, the best epoch found so far is used, which can avoid overfitting from higher learning rates. The

best and worst setups found in an example run with PGRU can be seen in 5.4. Layer count taken from [1, 2, 3], cell count taken from [8, 16, 32, 64], learning rate in the range 0.001-0.005.

Run	lr	layers	cells	RMSE
Best	0.00233	2	64	0.03014
Worst	0.00104	1	8	0.04351

Table 5.4: An example hyperopt run with PGRU on a reduced version of the starter tank dataset detailed in section 5.2.2. 100 total evaluations.

Factory Data and Setup

We will run experiments on several datasets created from real-world data from sensors in a food production factility. Do note that this is raw data from a real factory, and so includes noise in the form of irrelevant sensor data, incorrect measurements, and processing errors. We have been given information about the intended functions of the processes, but from a machine learning and data processing perspective, they are barely explored. The tasks have been created in cooperation with the factory owners, but are ultimately decided by the authors. We have included examples we believe are interesting to attempt to solve, and illustrative in terms of what results we found. It has been shown in Neil *et al.* [22] that the PLSTM has an advantage in certain scenarios, and we have verified this advantage for both PLSTM and PGRU, but it has not been shown whether this advantage holds in real-world event-driven datasets. The data is recorded in an event-driven manner, so the resulting raw time series is aperiodic and, when multiple time series are combined, asynchronous across channels. The standard TensorFlow implementation of RNN models require input data in all channels at every step, so we forward fill data to create a synchronous, but still aperiodic time series. As the datasets are largely unknown, we will run models first with very short lookback to see if long-term information can be exploited by the model. If lookback serves no purpose, then the difference between the phased and non-phased models is expected to be negligible. Each dataset is described below in its own section. Different versions of the datasets are run to explore how the models interact with common types of changes for irregular time series to be used in RNN models. We outline the different versions of the datasets used here:

The base version is the synchronous, aperiodic time series mentioned above.

The irregular version we use is the base version with the delta time between samples in the series added as an input channel. This is a commonly used method to allow RNNs to handle time series with an irregular frequency or missing values in a time series with regular frequency.

The regular version we use is the base version resampled to a regular frequency using linear interpolation based on the time stamps of samples, with delta time added.

Note that the phased models will receive a time signal in all versions as it is used for the time gating function. Note that the irregular version will be downsampled or upsampled, as detailed in the experiments.

5.2.2 Time series Forecasting

Tuning of Hyperparameters

We have run Hyperopt on several versions of the data for both tasks, with similar results for all tasks. Networks prefer larger cell counts, with marginal improvements above 64 cells, and there is a small increase in performance from adding layers (but a very large runtime cost). We saw less increase in performance from adding layers to phased models. Learning rates perform best in the range 0.0025-0.0035, with 0.0025 producing more stable runs (0.002 is the suggested default for Nadam[21]). In accordance with the hyperopt runs, networks have been set to 64 cells in 2 layers with a learning rate of 0.0025 for most tasks. While 64 is the maximum we allow for repeated runs, cell counts up to 256 have been tested briefly with marginal improvements and some issues with convergence on most smaller tasks.

Dataset 1: Milk Storage Tank

This dataset is from a single raw milk storage tank which goes through a cycle of filling, emptying and cleaning with varying duration and frequency. The cleaning process is done with hot steam, so heats the tank rapidly, but milk will only be added when the tank has sufficiently cooled. When milk enters the tank, it is cooled rapidly further. The task is to predict the temperature in the tank 20 minutes ahead, given a window of samples from the same signal. We compared 32 sample lookback and 2 sample lookback to examine whether the task requires longer term lookback and how the models handle optimizing for very few samples. For the irregular versions A1 and A3, we downsample to 5 minute resolution. For the regular version A2, we resample to a regular frequency of 5 minutes. Run settings can be seen in Table 5.5. We tested adding signals from more sensors, but this made the results more reliant on the last one or two samples, making RNN results less interesting (1 lookback gave results close to 32 lookback in this scenario). A plot of the signals can be seen in Figure 5.20.

Dataset 2: Starter Preparation Tank

This dataset is from a tank with a cheese culture starter being prepared for addition to the cheese-making process. The task is to predict the pH value in the tank 5 minutes ahead, given a window of pH value samples. For the irregular version B1, we downsample to 1 minute resolution. For the regular versions B2 and B3, we resample to a regular frequency of 1 minute. Signals from the process can be seen in Figure 5.21. Note that the pH signal shows extreme changes when the tank is empty, while it is intended to measure the pH of the fluid inside.

Long input sequence forecasting

Phased models are intended to function better than non-phased models on long input sequences and we suggest that short-term forecasting in our tasks so far puts great emphasis on the very last samples in the lookback window and have relatively regular samples due to downsampling, which is problematic for phased model performance. In response to

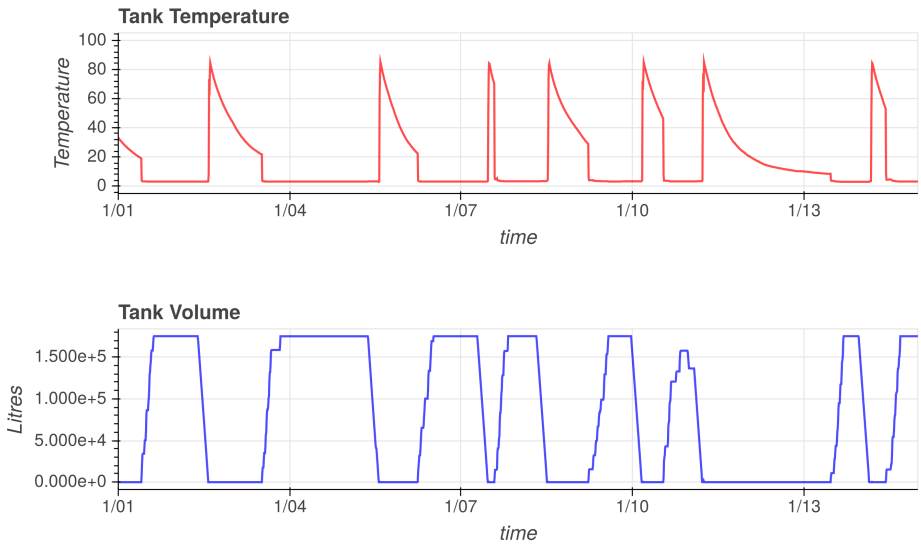


Figure 5.20: Plot of the temperature and liquid volume in a milk storage tank. Notice the tank being cleaned by hot steam in between milk storage periods.

these issues, we have included two long input forecasting tasks. The first task D1 is to predict temperature and liquid volume 240 minutes ahead in a starter tank, given a lookback window of 512 samples from the same time series as input. The dataset is processed to an irregular dataset with downsampling at 5 minute resolution. We added an irregular D2 example with downsampling at 1 second, which is computationally expensive for our tasks, but includes a higher amount of irregularity. For this task, we use 512 lookback and 10 minute prediction offset (which with minimum 1 second downsampling can mean 600 time steps ahead). Finally, we have included a comparison. One irregular task C3 with shorter lookback at 128, upsampling through imputation of values at 10 second intervals, with 10 step (as opposed to time) forecasting. For comparison, C4 shows an equivalent task without imputation. C3 and C4 get as input and predict volume, temperature and pH value in the tank.

Hypothesis or Expected Results

The advantages shown in the original paper[22] do occur when classifying sine waves sampled at an irregular interval without very long input sequences, as seen in subsection 5.1.4, so it is possible we see an improvement. However, the input windows are small and predictions might be largely determined by very recent samples, which is likely to lead to little difference between the phased and non-phased models. We can also expect downsampling that introduces a lot of regularity in the input sequence reducing the effectiveness of phased models. Some portions of the datasets involve very rapid change, so predictions are unlikely to catch these at our sampling rates, so loss is expected to be relatively high.

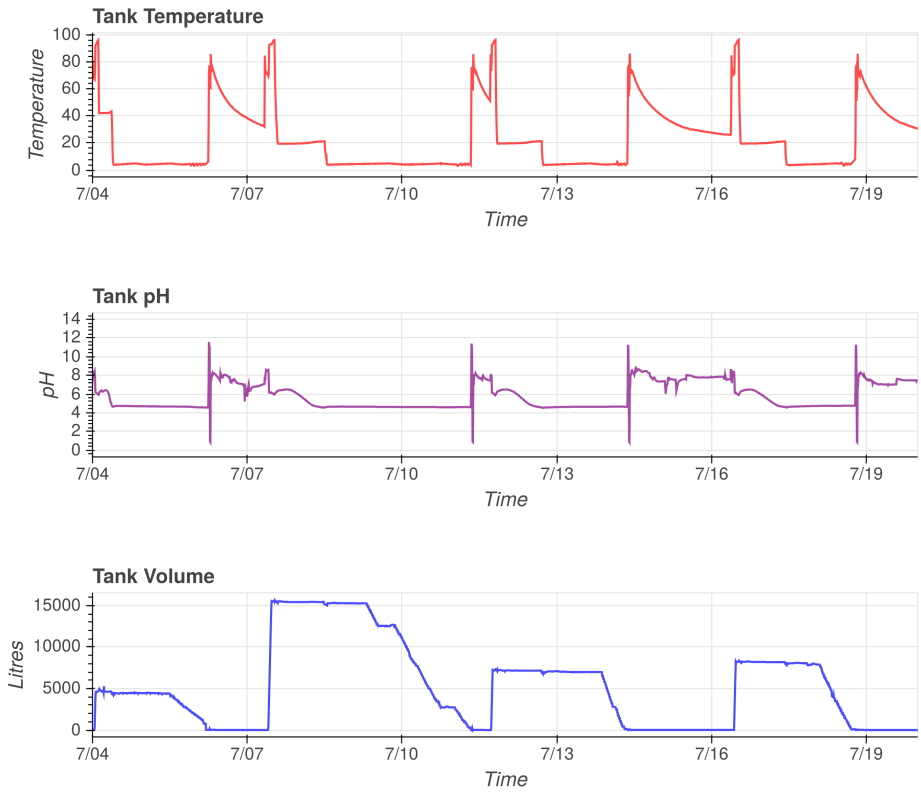


Figure 5.21: Plot of the temperature, pH and liquid volume in a starter preparation tank.

In the longer input sequence tasks we expect to see better performances for phased models overall, but it will be interesting to see if all favour phased models as they highlight different aspects of the model.

Results

Initial experiments were generic tasks given to us which only partially satisfy what we suggest are the strengths of phased models, having short input sequences, high regularity, and short-term predictions in time and number of samples. The additional long input sequence experiments are less generic tasks, but have characteristics closer to what we expect phased models are suited for. Note that results are generally not comparable across experiments, even on the same dataset, as we have used reduced versions of the dataset to achieve sufficient runs on some tasks. Within any single comparison, all models are run on the same version of the dataset, however. Note that all box plots have whiskers set to 1.5 times the Interquartile Range (IQR) maximum.

Name	Layers	Dataset	Sampling	Sampl. Freq.	LB	Pred.
A1	[64, 64]	1	DS	5Min	32	20Min
A2	[64, 64]	1	RS	5Min	32	20Min
A3	[64, 64]	1	DS	5Min	2	20Min
B1	[64, 64]	2	DS	1Min	32	5Min
B2	[64, 64]	2	RS	1Min	32	5Min
B3	[64, 64]	2	RS	1Min	2	5Min
C1	[64, 64]	2	DS	5Min	512	240Min
C2	[64, 64]	1	DS	1Sec	512	10Min
C3	[32, 32]	2	US	10Sec	128	10Step
C4	[32, 32]	2	DS	1Sec	128	10Step

Table 5.5: Overview of forecasting task settings. DS, US, and RS are downsampling, upsampling and resampling through interpolation respectively.

Short Input Sequence Experiments Results for the lookback comparison on dataset 1 (A1 and A3) can be seen in Table 5.6. Here, lb32 and lb2 refer to lookback and the metric shown is RMSE. The same comparison for a regular version of dataset 2 (B2 and B3) can be seen in Table 5.7. The results for the irregular version of dataset 2 B1 can be seen in Figure 5.23. The results for the regular version B2 can be seen in Figure 5.24. An unusual training strategy exemplifies that the phased models seem to converge to a performance similar or slightly better than the non-phased models. We run a model until the performance deteriorates on the validation set on B2 and store the model at the best seen epoch, and record the performance. We then restart the run from the best seen epoch, and repeat. The results can be seen in Figure 5.25.

Long Input Sequence Experiments The results for the irregular frequency task C2 with 1 second downsampling can be seen in Figure 5.22. Recall that this task has 512 lookback, which is relatively high for non-phased models. We see PGRU show an advantage over GRU on this task, which is consistent with irregular, longer input sequences being beneficial for phased models. Note that 10 minute prediction ahead with minimum 1 second frequency is 600 steps ahead maximum. The results for the long-term forecasting task C1 can be seen in Figure 5.26. Mean RMSE values of the 25 runs for each network type can be seen in Table 5.8. We see that phased models show an advantage over their non-phased counterparts on this task. This is expected as we suggested longer inputs and longer predictions would be suitable for phased models, even with regularity introduced through downsampling. Note that 240 minutes prediction ahead at minimum 5 minute frequency is 48 steps ahead maximum. The results for the task with shorter lookback at 128 with imputation (C3) can be seen in Figure 5.27b and without imputation (C4) in Figure 5.27a. Here we see imputation significantly increasing the performance of PGRU over GRU, perhaps due to reducing the irregularity to a constrained range (1-10 seconds), through greater redundancy to ease adjusting of on/off-states, or perhaps through better handling of redundant values over the GRU model.

Model	lb32	lb2
GRU	0.06904	0.09128
PGRU	0.08337	0.09800

Table 5.6: The impact of lookback on a full version of dataset 1 milk tank, irregular frequency. Settings A1 and A3. Average of 5 runs. Metric is RMSE.

Model	lb32Ver3	lb2Ver3
GRU	0.01037	0.02061
PGRU	0.01753	0.02367

Table 5.7: The impact of lookback on a full version of dataset 2, regular frequency, of dataset 2 starter tank. Settings B2 and B3. Mean of 5 runs. Metric is RMSE.

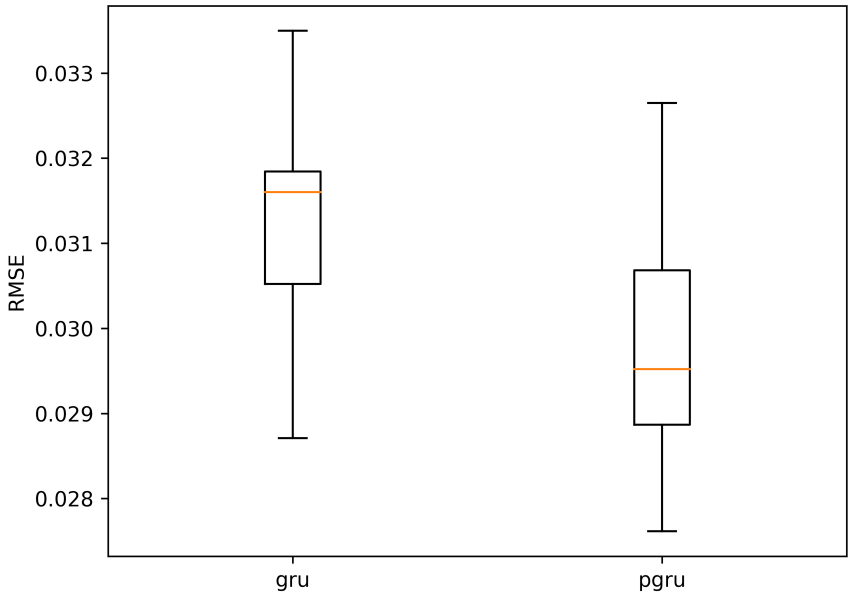


Figure 5.22: Box plot of results on a long input sequence task on dataset 1 milk tank. Settings C2. 25 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$.

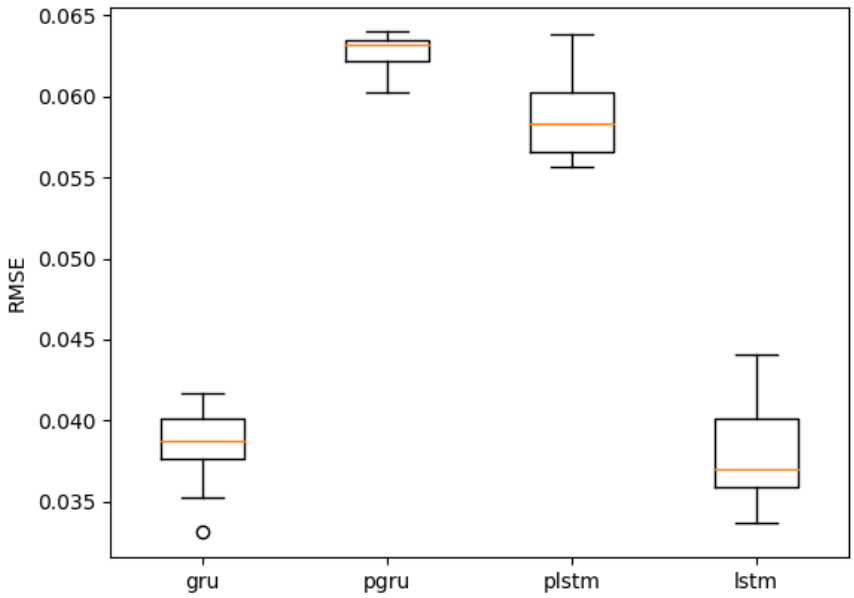


Figure 5.23: Box plot of results on irregular frequency, of dataset 2 starter preparation tank. Settings B1. 15 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$.

Model	RMSE
LSTM	0.1480
PLSTM	0.1459
GRU	0.1450
PGRU	0.1439

Table 5.8: Mean RMSE from 25 runs of all four models on the long-term prediction case on dataset 2 starter tank. Settings C1.

5.2.3 Prediction of Cheese pH 4 Hours After Curdling

Description of Dataset

This dataset is from the cheese production process in a dairy. The dataset includes the ratio of starter culture to milk used (single value per label) and time series data from the cheese curdling tanks and the starter preparation tanks. Which subset of tags we use for prediction might vary, but the tags available are the same as those listed in the previous sections. Of particular interest are the pH readings, temperature and volume readings in the starter tanks, as the dairy is interested in the effect nuances in the starter culture have on

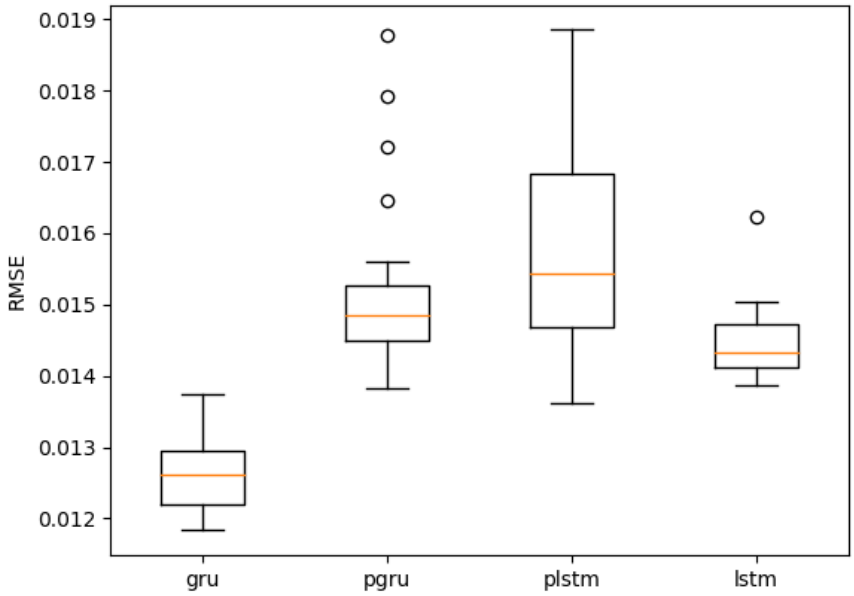


Figure 5.24: Box plot of results on regular frequency of dataset 2 starter preparation tank. Settings B2. 15 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$.

the final product quality. The task is to predict the pH value of the cheese 4 hours after the curdling process has completed, as measured by a laboratory. For cheese, this is a relevant quality metric that offers an indication of the quality and features of the final product. For each pH label, relevant time series data from the starter tanks are included, creating long series of up to 6000 time steps depending on sampling.

Hypothesis or Expected Results

Since the input includes sequences of over 1000 time steps it is possible we see some of the advantages outlined in the original PLSTM paper[22] for long time series. Given the clear advantage over standard LSTM on long sequences seen in the paper, if we do not see an improvement over non-phased models, handling long-term information in the input is likely not to be important for the final accuracy. Since a regular LSTM might only reasonably be expected to maintain memory of no more than the 100 last steps at prediction time, we hypothesize that it might miss out on important information that is needed for pH prediction.

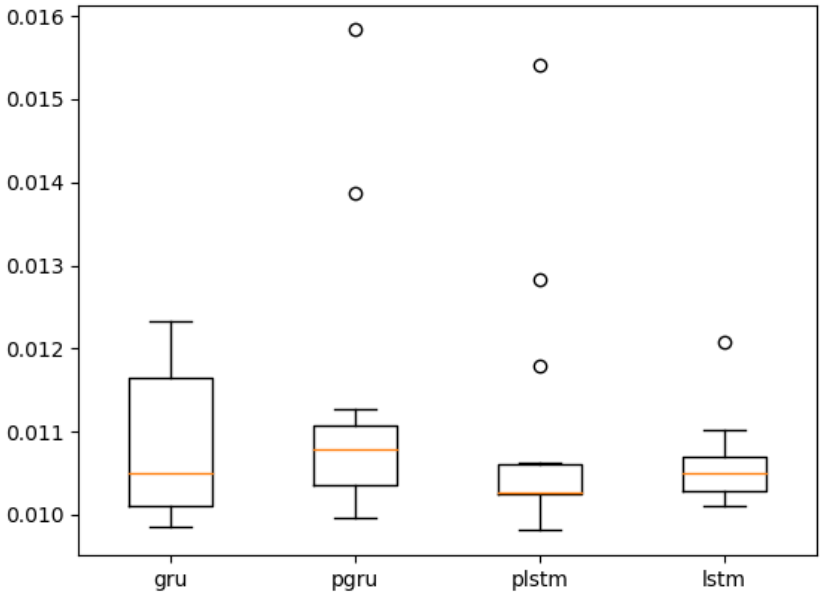


Figure 5.25: Box plot of an unusual training strategy irregular frequency of dataset 2 starter preparation tank. Settings B2. 15 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$.

Preliminary Results

Having run the various models on a wide range of features, both in terms of time series and scalar features, we get a result that does not offer grounds for proper comparison between the models. All models learn to predict approximately the mean pH value of the examples in the training set for all examples. A typical pH regression pattern from our experiments can be seen in Figure 5.29.

The problem was also formulated as a classification problem, where the classes correspond to the pH being considered too low, too high or acceptable. These thresholds were provided by the manufacturer. The classification score for the optimizer is set to compensate for the lower frequency of the low and high pH classes by increasing the penalty for incorrectly classifying those. In this context none of the models outperform the very simple strategy of classifying all examples as the most frequent class, acceptable pH, which would give a true positive rate of 44%.

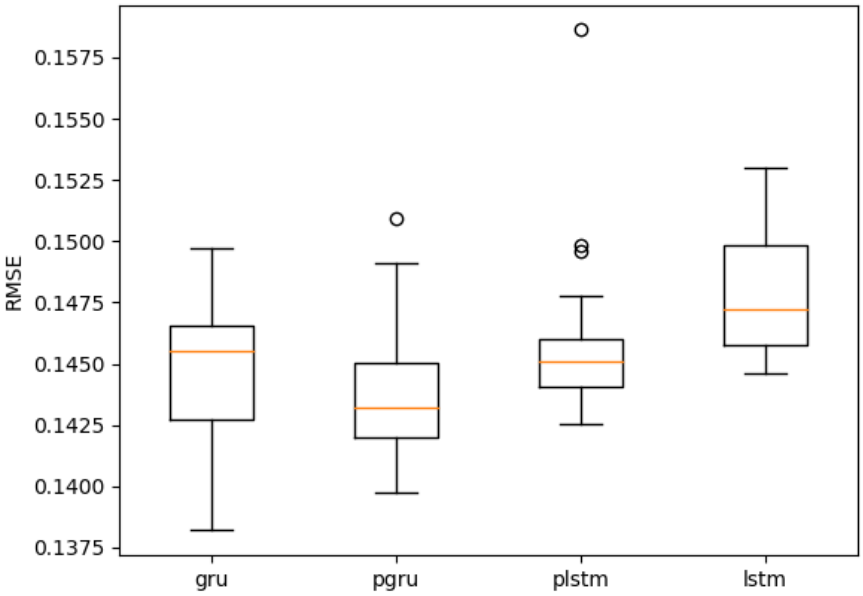
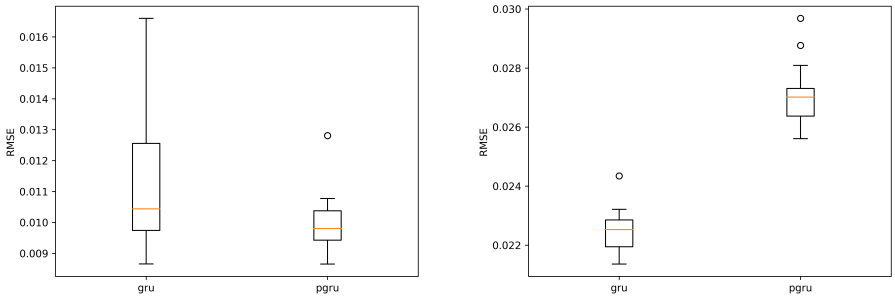


Figure 5.26: Box plot of the long-term prediction case on irregular frequency of dataset 2 starter preparation tank. Settings C1. 25 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$.

5.3 Results Discussion

5.3.1 Benchmark Experiments Discussion

To validate our implementation and compare our proposed variation on the PLSTM, PGRU and the performance of layer stacking, we performed experiments on classification of aperiodically sampled sine waves, previously explored in Neil *et al.* [22], as well as sequential MNIST classification. In these tests, all tested models behave as expected. Non-phased models struggle with aperiodic sampling and long sequences or high resolution sampling. Phased models handle the aforementioned well. PGRU outperforms PLSTM similarly to how GRU outperforms LSTM on similar tasks. A comparison of the phased models can be seen in Figure 5.7. That is, GRU shows slightly faster runtime and better performance on some tasks, MNIST being a clear example. For the MNIST task, the phased models are significantly faster to train in terms of epochs and even time for 1 layer networks. At 2 layers, the difference decreases between the phased models and GRU, but it is still significant, as seen in figures 5.14 for data and 5.16 for time. Note that we do not claim an optimal implementation of the phased models, and as noted by the PLSTM authors, the sparsity of the network updates are not properly exploited in practice[30]. Stacking phased cells appears to produce better results, similar to stacking non-phased cells, as expected. Note that



(a) C3 RMSE **(b)** C4 RMSE

Figure 5.27: In 5.27a, a box plot of C3 on dataset 2 starter preparation tank. 20 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$. In 5.27b, Box plot of C4 dataset 2 starter preparation tank. 20 runs per type. Metric RMSE. Whiskers at $1.5 \cdot IQR$.

Figure 5.28: Open top curdling vats where the milk proteins are denatured into cheese. These are not the actual curdling vats we are reviewing in this task. Retrieved from Wikipedia[52] under the Creative Commons 3 licence[39].

these experiments were classification tasks. In classification tasks, there is not necessarily a significant focus on the last part of the sequence, as opposed to in forecasting/prediction tasks.

5.3.2 Factory Data Experiments Discussion

The phased models showed poor performance on all short input sequence tasks, regular or irregularly sampled. They showed an increase in performance over non-phased on long input sequence tasks, both in the longer-term forecasting task with 5 minute downsampling

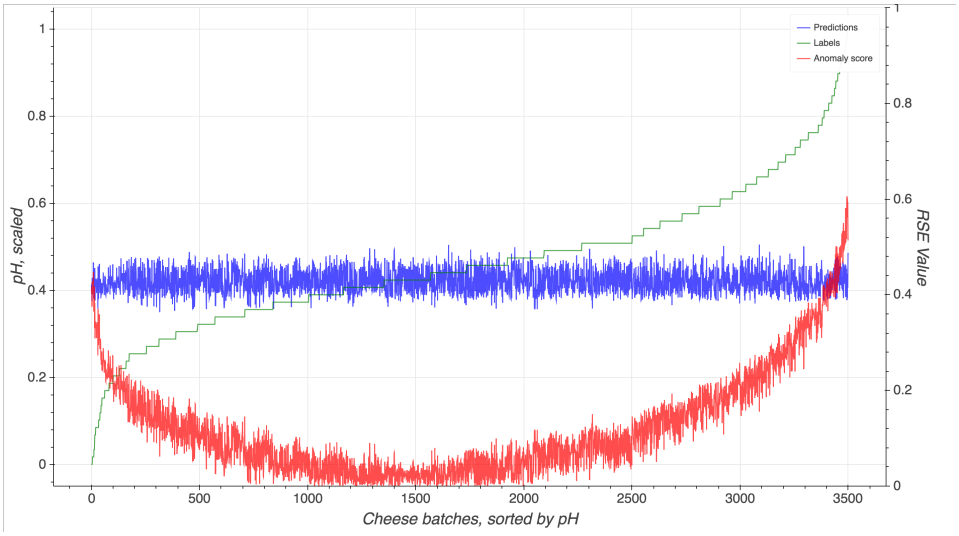


Figure 5.29: pH values after 4 hours for one kind of cheese in our dataset, as predicted by GRU. Examples are sorted for ease of plotting. Note that our models only learn to predict values close to the average pH in the dataset, and cannot identify examples with particularly high or low pH.

and the shorter-term forecasting task with 1 second downsampling (and thus a higher degree of irregularity). This is in line with our general expectations in both cases according to our own understanding and previous works. Interestingly, we also saw phased models perform much better than their non-phased counterparts on a dataset with imputation at 10 seconds than without.

Forecasting and Lookback The results of the lookback comparison show a relatively minor increase in performance from increasing lookback from one. In these situations, we believe the models find the cycles in the data largely too irregular to predict, and gain most of the observed performance from reproducing the last seen value with some adjustments from looking at very recent values. In the situation with lookback of 2, the recurrent portion of our models will only be used for 1 step, but we still see a difference between phased and non-phased models, with phased models performing noticably worse. At 32 lookback, which places a larger emphasis on recurrence, we see the difference between phased and non-phased models increase, favouring non-phased models. Differences can be seen in tables 5.6 and 5.7.

For longer lookbacks the picture is not as clear, but favours phased models. Both our 512 lookback cases show phased models produce better results, in the case of long-term prediction and short-term prediction. When using lookback of 128 and predicting 10 steps ahead, we see GRU outperforming Phased GRU on the data only dowsampled to 1 second. When upscaling this signal with values every 10 seconds, we see the Phased GRU perform marginally better than the GRU model. This can be seen in Figure 5.27a and Figure 5.27b.

pH Regression and Classification The pH regression and classification tasks were extremely difficult for our models, and with no variation on the models or task were we able to produce useful predictions from the raw time series of the starter tanks. The regression approach was only able to learn the average pH value of the seen examples. As a classification problem we could not produce results better than placing all samples in the most frequent class, even when the scores for classes were balanced in the optimizer not to favour the most frequent class. The aforementioned is a naive strategy that can be seen in some classifier models when they learn nothing of interest from the input data. Since neither phased models, GRU nor LSTM were able to beat naive strategies on this dataset, we deem the result to be inconclusive.

Phased Model Weaknesses There is not a large body of work detailing phased model behaviour, so causes we suggest are speculative. With an on ratio, r_{on}, of 1.0 the model is approaching the behaviour of an inefficient (due to time calculations) LSTM/GRU model. Advantages of the phased models are reduced memory degradation and better internal representation of irregular sequences. Neither of these advantages come into play for improving accuracy on a prediction only based on the very last samples in an input sequence. Disadvantages that can cause the loss of performance compared to non-phased models are increasing trained parameters, and reducing available cells (as some are off at any given time). Another possible angle, as noted by the original authors[22], is that PLSTM can be viewed as performing a form of dropout. Dropout generally harmed results in our tests. Jozefowicz *et al.* [18] performed tests on GRU, LSTM and mutations, and found that only large networks of LSTM saw a benefit from dropout. Note that with a large amount of cells, the phased models can have enough cells even if some are off, and that trainable r_{on} can eventually reach 100%, which in both cases can be expected to lead to near or equivalent performance to the non-phased models, though at a much higher cost. That said, larger networks are harder to train and might experience overfitting. We see something that indicates phased models can converge to non-phased models in experimental runs where models that were restarted from best epoch when their performance deteriorated would converge to performance similar to the non-phased models. Note that we have performed a large number of explorative runs to find interesting tasks, and that results from these support the idea that the phased models struggle on short input sequence forecasting tasks even when irregularity is introduced, on the factory dataset.

Chapter 6

Discussion

To give the discussion below additional context, we will briefly outline the behaviours of phased models we understand as the most significant. Each individual cell in a phased model has an *on ratio*, r_{on}, and a *cycle frequency*, τ, associated with them. This allows the models to handle different time scales in the dataset, as each cell can be susceptible to having its hidden state updated at a different moment in time. The memory of a cell can for instance not be changed for several steps in a sequence in order to retain a memory of some salient information in the past. It also allows individual cells in a model to receive data only at a regular frequency regardless of input frequency, as the cell ignores all updates in its off-state. For instance, a dataset where a signal is comprised of samples from several sampling frequencies can have one group of cells operating at each frequency to only handle changes occurring at each respective frequency. A dataset with aperiodic sampling can have a variety of cells operating at different frequencies, to ensure all data is read by some cell, while each cell only gets updates at a regular interval, giving the cell a consistent view of change over time. With this view of phased models in mind, the new behaviours primarily optimize for the relations between samples in input sequences.

6.1 Experiments

6.1.1 Benchmark Experiments

From our experiments with the MNIST dataset (subsection 5.1.1) we see that the concept of a phased model can be an effective tool for the classification of long sequences. It is likely that the ability to prevent state updates allows some cells in the network to remember important information from early on in the sequence, and that this leads to superior performance compared to regular RNN models. We also confirm that a phased RNN can receive a performance increase when stacked in layers, not unlike regular RNNs. From the frequency classification task (subsection 5.1.4) we can gather that phased models show promise in the classification of aperiodically sampled sequences. The sequences in this experiment are still short enough that architectures like LSTM or GRU should maintain

memory through most of the sequence. In this case we conjecture that the favourable performance of the phased models is due to phased models utilizing the time information about the steps in the sequence better than non-phased models. Although the regular RNN models receive time stamps as well, the lack of a bias towards a certain well-suited method for handling time information leads to the models having to learn this skill from the ground up - a seemingly challenging task. From both experiments it seems like the idea of applying the time gate to a GRU cell adds value, with performance of the Phased GRU matching or surpassing that of the Phased LSTM, both in speed of execution and accuracy.

6.1.2 Factory Data Experiments

The factory dataset shows us that phased models are not a direct upgrade to non-phased models in their current state. They are more expensive. In most of our short-term forecasting experiments, they take longer to train, give slower inference, converge slower and give less stable results. These results hold for irregular sampling, regular sampling and different sampling frequencies, but do not hold in a short-term forecasting task with long input sequences and highly irregular data (1 sec to days between samples). We also see that on one long-term forecasting task, the phased models show a slight advantage over the non-phased models. Note that phased model performance on the short-term tasks was often very poor, so the performance increase from phased models on short input sequences to long input sequences is significant in both cases. Phased models showed an advantage in a shorter input sequences at 128, when imputation was introduced. Our understanding of this is that the model can better handle redundant data than a conventional RNN, and can effectively utilize the new limited range of delta times between samples the imputation creates, but it can also be due to non-phased models struggling when imputation introduces samples between the samples in the original data, increasing memory required to handle relations. This is opposite to downsampling, where a lot of regularity is introduced, without reducing the range of possible delta values much, so the difficulties in choosing appropriate r_{on} and τ are significant, while less memory is required for relations. These experiments do not let us track the relative contribution of effective handling of aperiodicity versus effective handling of long input sequences in phased models precisely, but it does show that there is potential for phased models even on forecasting tasks on factory data. We expect to see the difference between phased and non-phased models to grow as the input sequences increase in size and forecasting offset increases, especially on irregular data.

6.1.3 Summary

The phased models have significant disadvantages compared to non-phased models with delta time input in some contexts, even when handling somewhat aperiodic data, but the value of the phased models outweighs these disadvantages on some tasks. We suggest this occurs when data is irregular and input sequences are long in forecasting tasks, and that these requirements lessen if the entire sequence is highly relevant, such as in our frequency classification task, and that only long input sequences are required if they are sufficiently long, such as in the MNIST task. These results do support the advantages suggested in the original Phased LSTM paper[22].

6.2 Research Questions

- **RQ1: Can PLSTM or PGRU improve upon the LSTM and GRU baseline in aperiodic, asynchronous time series tasks?** Yes, but with the current disadvantages of phased models, the advantages need to be prominent to amount to an overall increase in performance. A short, slightly irregular time series is unlikely to be suitable for phased models.
- **RQ2: Do the relative properties of GRU and LSTM hold for their phased derivatives, PLSTM and PGRU?** Our benchmark experiments show that PGRU has slightly faster runtime and better accuracy than PLSTM on tasks where a similar difference is seen in the non-phased models. Our factory data experiments support this result.
- **RQ3: Do the phased models increase performance on any of the suggested tasks on real-world factory data?** In our testing, we found significantly worse performance for phased models on most forecasting tasks with short input sequences, even with irregular input. However, we found an increase in performance with long input sequences, especially if they were highly irregular.

6.3 Phased Model Discussion

In the introduction to this chapter, we outlined some behaviours that we see as the primary cause behind phased model performance over non-phased models. The additional behaviours come at a cost, however. The added complexity of the model also reflects in its run times that are significantly slower than regular RNN models, at least in their current TensorFlow implementations. There are a multitude of reasons for this. As a result of being an experimental, cutting edge model, the TensorFlow Phased LSTM implementation (`tf.contrib.rnn.PhasedLSTMCell`)[50] is intended to be a proof of concept rather than a fully optimized model for production use. Consequently the code is based on an LSTM implementation that sacrifices performance over code readability and extensibility (`tf.contrib.rnn.BasicLSTMCell`)[45,47]. Furthermore, the time gates of the phased models introduces mathematics that are not commonly seen in machine learning models, like modulo arithmetic. In general one can assume that performance optimization work for a machine learning framework will be directed toward widely used operations like tensor products, rather than those that occur off the beaten path. Furthermore, the benefits in computational complexity that phased models promise, currently fail to materialize in practice. In theory one would assume that the ability to skip state updates for subsets of cells in a network would improve performance, but realizing this depends on strong support for sparse tensor updates. Also, due to a leak in the time gate during training, necessary to train the gate parameters, one can only utilize the benefits of sparse updates during inference. A final limiting factor on the speedups one could expect from sparse updates is the inherent temporal nature of recurrent networks. On a modern GPU a large tensor product is often not much slower to perform than a smaller one. Consequently it is a lot harder to gain time performance by reducing computation at each step, than it is by *skipping steps entirely*, as seen in Campos *et al.* [24]. One could expect this situation

to improve should phased models be widely adopted, but it is perhaps not reasonable to expect speeds on par with simpler models even in this case, at least not in the training setting.

Runtime costs aside, the model also has to optimize with its on ratios, r_{on}, and cycle frequencies, τ, in mind. These are additional parameters and thus might result in need for additional training for the model to properly converge. Imagine a hypothetical dataset, designed to be ill-suited for phased models, where the optimal way to solve a problem involves having all available cells see all input samples, as is done in a regular RNN. The phased model then has to first train itself to the point where all cells have a high r_{on} and *then* train for the actual task at hand, if it reaches that point at all before ending in a separate local minimum and/or overfitting. A traditional RNN model, on the other hand, bypasses the need for training these additional parameters. Thus, we argue that if improving the modelling of relations between samples in input sequences is not significant, then phased models are likely to perform worse than non-phased models.

6.4 Limitations

As our computational resources are limited and the phased models in their current state are quite expensive to train, we have not been able to test very large networks or very long input sequences, and appropriate tasks for this setup. The more cells are available in a phased model, the more cells can be active at any given time, so the expressive power and the coverage of phased models increases with network size, assuming the model converges properly. Because of the need for additional cells to cover the input data at any given time with sufficient cells, it is possible that phased models see a larger increase in performance when increasing the width of layers than what is seen in regular RNNs. For some problems one could then expect to see some cross-over point, such that phased models perform better than regular RNNs for wide networks, and regular RNNs perform better for smaller networks, assuming both models converge. For input with large variance in time between samples, and a wide selection of interesting frequencies that could be tracked, like our factory dataset, this might be significant for difficult tasks.

We suggest that phased models perform better when the entire input sequence is relevant, as it allows the model to utilize the different on/off frequencies it has available. The pH classification task was intended to focus more on this, but we found no good predictors for 4 hour pH in the time series dataset. We found some reasonable predictors in the available scalar values, but these are not relevant for our models. Our long-term prediction tasks are intended to alleviate this problem somewhat. As long-term information in long sequences where the entire input sequence is relevant seems to favour phased models, traditional signal classification tasks are likely to be a good fit for future work on real data.

While we see different performance exhibited by the phased models on different tasks, the cause is not always clear. One key element of this is effective handling of aperiodicity versus effective handling of long time series. While our factory data results suggest phased models primarily excel on long input sequences, the cause might still be rooted in whether the entire input sequence is highly relevant. The frequency classification task suggests this, having short inputs, using the entire sequence, and favoring phased models. The imputation comparison might suggest this, as it increases the distance between information

in the original time series, but as the input sequence in total is the same length for the input data with and without imputation, there can also be other causes. An interesting task could be a long-term forecasting task with short, irregular input that still gives long-term information. Long-term forecasting with short input sequences requires considerable downsampling in our case. With traditional methods, this would make the input data very regular so we have not been able to test long-term predictions with short, irregular input sequences.

6.5 Contributions

In this thesis we contribute a review of papers that present novel ways of overcoming the limitations of established RNN architectures. We also present our experimental findings from having applied GRU, LSTM, PGRU and PLSTM to a selection of real world data tasks. Furthermore we provide source code for two novel TensorFlow primitives that we have developed in the course of our work. The most interesting one, the `PhasedGRUCell` (Appendix A) provides a new phased architecture to the collection of publicly available TensorFlow models. It is shown to execute faster and with similar or better results than a native TensorFlow Phased LSTM implementation. It thus provides a useful alternative for others wishing to experiment with similar models. The second component we provide source code for, the `MultiPRNNCell` (Appendix B) is a wrapper class that allows stacking of multiple layers of both our own `PhasedGRUCell` and the `PhasedLSTMCell` of TensorFlow. It thus increases the flexibility and usefulness of both models.

6.6 Future Work

We suggest exploring phased models on signal classification tasks with irregular sampling or long input sequences. Long-term forecasting, perhaps with large networks, is a task that seems promising given our results. In terms of changes to the phased model, the original Phased LSTM paper[22] suggests a simpler square-wave oscillation for a more efficient model, and that libraries for fast sparse operations in neural networks can improve runtime performance significantly. We see poor runtime performance as one of the main disadvantages of current Phased LSTM, which is not a necessity in theory due to the off-state reducing state update calculations. Better support for sparse changes to tensor products in neural networks have the added benefit of enabling the realization of some performance improvement in the handling of multi-channel asynchronous time series, as well, where channels are updated one at a time as opposed to in unison.

We would like to see a combination of the most promising works in this area. Skip RNN[24], which shows faster runtime in practice, and various attention-based models for LSTM and similar architectures[14,19,20], which show stable performance. Skipping state updates as in Skip RNN can be used similarly to phased models. Skip RNN currently shows an increase in runtime performance and signs of reducing memory degradation similar to phased models. Skipping state updates can be used in attention models for when attention is very low to reduce runtime costs and reduce degradation of memory. Reducing the length of the computational graph by skipping an entire step in all cells in a network is

very significant as it reduces the issue of serial execution in RNNs.

Bibliography

1. Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* **2,** 303–314 (1989).
2. Elman, J. L. Finding Structure in Time. *Cognitive Science* **14,** 179–211. ISSN: 1551-6709 (1990).
3. Hochreiter, S. & Schmidhuber, J. Long Short-Term Memory. *Neural Computation* **9,** 1735–1780 (1997).
4. Jordan, M. I. in *Neural-Network Models of Cognition* (eds Donahoe, J. W. & Dorsel, V. P.) *Advances in Psychology* Supplement C, 471–495 (North-Holland, 1997). doi:10.1016/S0166-4115(97)80111-2. http://www.sciencedirect.com/science/article/pii/S0166411597801112.
5. Gers, F. A. & Schmidhuber, J. *Recurrent nets that time and count* in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium* **3** (IEEE, Como, Italy, 2000), 189–194 vol.3. http://dx.doi.org/10.1109/ijcnn.2000.861302.
6. Jones, E., Oliphant, T., Peterson, P., *et al. SciPy: Open source scientific tools for Python* 2001. http://www.scipy.org/.
7. Bergstra, J. S., Bardenet, R., Bengio, Y. & Kégl, B. in *Advances in Neural Information Processing Systems 24* (eds Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F. & Weinberger, K. Q.) 2546–2554 (Curran Associates, Inc., 2011). http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.
8. Duchi, J., Hazan, E. & Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* **12,** 2121–2159 (2011).
9. Tieleman, T. & Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* **4,** 26–31 (2012).

10. Bergstra, J., Yamins, D. & Cox, D. D. *Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms* in *Proceedings of the 12th Python in Science Conference* (2013), 13–20.

11. Graves, A. Generating sequences with recurrent neural networks. *ArXiv e-prints.* arXiv: 1308.0850 [cs.NE] (2013).

12. Liu, L., Shao, L. & Rockett, P. Boosted key-frame selection and correlated pyramidal motion-feature representation for human action recognition. English. *Pattern Recognition* **46.** ISSN: 0031-3203 (July 2013).

13. Pascanu, R., Mikolov, T. & Bengio, Y. *On the difficulty of training recurrent neural networks* in *Proceedings of the 30th International Conference on Machine Learning* (eds Dasgupta, S. & McAllester, D.) **28** (PMLR, Atlanta, Georgia, USA, 2013), 1310–1318. http://proceedings.mlr.press/v28/pascanu13.html.

14. Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate. *ArXiv e-prints.* arXiv: 1409.0473 [cs.CL] (2014).

15. Chung, J., Gulcehre, C., Cho, K. & Bengio, Y. in *NIPS 2014 Workshop on Deep Learning, December 2014* (2014).

16. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *ArXiv e-prints.* arXiv: 1412.6980 [cs.LG] (2014).

17. Koutnik, J., Greff, K., Gomez, F. & Schmidhuber, J. *A Clockwork RNN* in *Proceedings of the 31st International Conference on Machine Learning* (eds Xing, E. P. & Jebara, T.) **32** (PMLR, Bejing, China, 2014), 1863–1871. http://proceedings.mlr.press/v32/koutnik14.html.

18. Jozefowicz, R., Zaremba, W. & Sutskever, I. An empirical exploration of recurrent network architectures. *Journal of Machine Learning Research* (2015).

19. Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kočisky, T. & Blunsom, P. Reasoning about entailment with neural attention. *ArXiv e-prints.* arXiv: 1509.06664 [cs.CL] (2015).

20. Xu, K. *et al. Show, attend and tell: Neural image caption generation with visual attention* in *International Conference on Machine Learning* (2015), 2048–2057.

21. Dozat, T. Incorporating nesterov momentum into adam (2016).

22. Neil, D., Pfeiffer, M. & Liu, S.-C. in *Advances in Neural Information Processing Systems 29* (eds Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I. & Garnett, R.) 3882–3890 (Curran Associates, Inc., 2016). http://papers.nips.cc/paper/6310-phased-lstm-accelerating-recurrent-network-training-for-long-or-event-based-sequences.pdf.

23. Song, S., Lan, C., Xing, J., Zeng, W. & Liu, J. An End-to-End Spatio-Temporal Attention Model for Human Action Recognition from Skeleton Data (Nov. 2016).

24. Campos, V., Jou, B., Giró-i-Nieto, X., Torres, J. & Chang, S.-F. Skip RNN: Learning to Skip State Updates in Recurrent Neural Networks. *ArXiv e-prints.* arXiv: 1708.06834 [cs.AI] (2017).

25. Chang, S. *et al.* in *Advances in Neural Information Processing Systems 30* (eds Guyon, I. *et al.*) 77–87 (Curran Associates, Inc., 2017). http://papers.nips.cc/paper/6613-dilated-recurrent-neural-networks.pdf.

26. Ergen, T., Mirza, A. H. & Kozat, S. S. Unsupervised and Semi-supervised Anomaly Detection with LSTM Neural Networks (Oct. 2017).

27. Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R. & Schmidhuber, J. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems* **28,** 2222–2232. ISSN: 2162-237X (Oct. 2017).

28. He, Z. *et al.* in *Advances in Neural Information Processing Systems 30* (eds Guyon, I. *et al.*) 1–11 (Curran Associates, Inc., 2017). http://papers.nips.cc/paper/6606-wider-and-deeper-cheaper-and-faster-tensorized-lstms-for-sequence-learning.pdf.

29. Mozer, M. C., Kazakov, D. & Lindsey, R. V. Discrete-Event Continuous-Time Recurrent Nets. *ArXiv e-prints.* arXiv: 1710.04110 [cs.NE] (2017).

30. Neil, D., Pfeiffer, M. & Liu, S.-C. *Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences* Youtube. https://www.youtube.com/watch?v=ZMyVR3nwgAQ.

31. Silver, D. *et al.* Mastering the game of go without human knowledge. *Nature* **550,** 354–359 (2017).

32. Zhu, Y. *et al. What to Do Next: Modeling User Behaviors by Time-LSTM* in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17* (2017), 3602–3608. doi:10.24963/ijcai.2017/504. https://doi.org/10.24963/ijcai.2017/504.

33. Liu, L., Shen, J., Zhang, M., Wang, Z. & Tang, J. Learning the Joint Representation of Heterogeneous Temporal Events for Clinical Endpoint Prediction. *ArXiv e-prints.* arXiv: 1803.04837 [cs.AI] (2018).

34. Moniz, J. R. A. & Krueger, D. Nested LSTMs. *ArXiv e-prints.* arXiv: 1801.10308 [cs.CL] (2018).

35. Stepleton, T. *et al.* Low-pass Recurrent Neural Networks-A memory architecture for longer-term correlation discovery. *ArXiv e-prints.* arXiv: 1805.04955 [cs.LG] (2018).

36. Van der Westhuizen, J. & Lasenby, J. The unreasonable effectiveness of the forget gate. *ArXiv e-prints.* arXiv: 1804.04849 [cs.NE] (2018).

37. Allen, R. & Li, M. The Data Incubator. https://blog.thedataincubator.com/2017/10/ranking-popular-deep-learning-libraries-for-data-science/.

38. Ceolini, E. *Phased LSTM implementation in Tensorflow* https://github.com/Enny1991/PLSTM.

39. Creative Commons. *Attribution 3.0 Unported (CC BY 3.0)* https://creativecommons.org/licenses/by/3.0/.

40. Evans, R. & Gao, J. *DeepMind AI Reduces Google Data Centre Cooling Bill by 40%* https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.

41. Microsoft. *Reasons to Switch from TensorFlow to CNTK* https://docs.microsoft.com/en-us/cognitive-toolkit/reasons-to-switch-from-tensorflow-to-cntk.

42. Olah, C. *colah's blog* http://colah.github.io/ (2018).

43. Remy, P. *Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences (NIPS 2016) - Tensorflow 1.0* https://github.com/philipperemy/tensorflow-phased-lstm.

44. TensorFlow. *A Guide to TF Layers: Building a Convolutional Neural Network* https://www.tensorflow.org/tutorials/layers.

45. TensorFlow. *Performance Guide - RNN Performance* https://www.tensorflow.org/performance/performance_guide#rnn_performance.

46. TensorFlow. *Premade Estimators* https://www.tensorflow.org/get_started/premade_estimators.

47. TensorFlow. *tf.contrib.rnn.BasicLSTMCell* https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/BasicLSTMCell.

48. TensorFlow. *tf.contrib.rnn.GRUCell* https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/GRUCell.

49. TensorFlow. *tf.contrib.rnn.MultiRNNCell* https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/MultiRNNCell.

50. TensorFlow. *tf.contrib.rnn.PhasedLSTMCell* https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/PhasedLSTMCell.

51. TensorFlow. *tf.contrib.rnn.RNNCell* https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/RNNCell.

52. Unknown. *During industrial production of Emmental cheese, the as-yet-undrained curd is broken by rotating mixers.* https://en.wikipedia.org/wiki/Cheese#/media/File:Production_of_cheese_1.jpg.

Appendices

A PGRU Code

We here include our implementation of our suggested PGRU recurrent cell, as described in section 4.4. The implementation is largely based on and inspired by the implementations of PLSTM and GRU that are already found in the TensorFlow code base[47,48]. We inherit from the base `tensorflow.contrib.rnn.RNNCell`[51] class, a base class for many TensorFlow RNN implementations. The `call` method is the most interesting piece of code, as it provides the logic for how the cell behaves when being fed data. This method constitutes our implementation of the data flow graph shown in Figure 4.2. The code is tested and confirmed to work in TensorFlow 1.8.

```
from tensorflow.contrib.rnn.python.ops import rnn_cell
from tensorflow.contrib.rnn import RNNCell
from tensorflow.python.framework import dtypes
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import init_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.ops import variable_scope as vs


# NOTE: Accessing protected members of a module is usually considered
# undesirable. I am reasonably sure it should be fine in this context, though.
_Linear = rnn_cell._Linear
_random_exp_initializer = rnn_cell._random_exp_initializer


class PhasedGRUCell(RNNCell):
"""Phased GRU recurrent network cell. GRU version of the LSTM variant
published in:

https://arxiv.org/pdf/1610.09513v1.pdf
"""

```

```
  def __init__(self,
               num_units,
               activation=None,
               kernel_initializer=None,
               bias_initializer=None,
               reuse=None,
               leak=0.001,
               ratio_on=0.05,
               trainable_ratio_on=True,
               period_init_min=1.0,
               period_init_max=1000.0):
      """Initialize the Phased GRU cell.

      Args:
          num_units: int, The number of units in the Phased GRU cell.
          activation: Nonlinearity to use.  Default: `tanh`.
          kernel_initializer: (optional) The initializer to use for the
              weight and projection matrices.
          bias_initializer: (optional) The initializer to use for the bias.
          reuse: (optional) Python boolean describing whether to reuse
              variables in an existing scope.  If not `True`, and the
              existing scope already has the given variables, an error is
              raised.
          leak: float or scalar float Tensor with value in [0, 1]. Leak
              applied during training.
          ratio_on: float or scalar float Tensor with value in [0, 1]. Ratio
              of the period during which the gates are open.
          trainable_ratio_on: bool, whether ratio_on is trainable.
          period_init_min: float or scalar float Tensor. With value > 0.
              Minimum value of the initialized period. The period values are
              initialized by drawing from the distribution:
              e^U(log(period_init_min), log(period_init_max))
              Where U(.,.) is the uniform distribution.
          period_init_max: float or scalar float Tensor.
              With value > period_init_min. Maximum value of the initialized
              period.
      """
      super(PhasedGRUCell, self).__init__(_reuse=reuse)
      self._num_units = num_units
      self._activation = activation or math_ops.tanh
      self._kernel_initializer = kernel_initializer
      self._bias_initializer = bias_initializer
      self._leak = leak
      self._ratio_on = ratio_on
      self._trainable_ratio_on = trainable_ratio_on
      self._period_init_min = period_init_min
      self._period_init_max = period_init_max
      self._reuse = reuse
      self._gate_linear = None
      self._candidate_linear = None

  @property
  def state_size(self):
      return self._num_units

  @property
  def output_size(self):
      return self._num_units

  def _mod(self, x, y):
      """Modulo function that propagates x gradients."""
      return array_ops.stop_gradient(math_ops.mod(x, y) - x) + x

  def _get_cycle_ratio(self, time, phase, period):
      """Compute the cycle ratio in the dtype of the time."""
      phase_casted = math_ops.cast(phase, dtype=time.dtype)
```

```
    period_casted = math_ops.cast(period, dtype=time.dtype)
    shifted_time = time - phase_casted
    cycle_ratio = self._mod(shifted_time, period_casted) / period_casted
    return math_ops.cast(cycle_ratio, dtype=dtypes.float32)

def call(self, inputs, state):
    """Phased GRU Cell.

    Args:
        inputs: A tuple of 2 Tensors.
            The first Tensor has shape [batch, 1], and type float32 or
            float64. It stores the time.
            The second Tensor has shape [batch, features_size], and type
            float32. It stores the features.
        state: A tensor, state from previous timestep.
    Returns:
        A tuple containing:
        - A Tensor of float32, and shape [batch_size, num_units],
        representing the output of the cell.
        - A Tensor of float32, and shape [batch_size, num_units],
        representing the new state.
    """

    (time, x) = inputs

    gates_input = [x, state]

    if self._gate_linear is None:
        bias_ones = self._bias_initializer
        if self._bias_initializer is None:
            bias_ones = init_ops.constant_initializer(
                1.0,
                dtype=x.dtype)
        with vs.variable_scope("gates"):  # Reset gate and update gate.
            self._gate_linear = _Linear(
                gates_input,
                2 * self._num_units,
                True,
                bias_initializer=bias_ones,
                kernel_initializer=self._kernel_initializer)

    gate_value = math_ops.sigmoid(self._gate_linear(gates_input))
    r, u = array_ops.split(value=gate_value,
                           num_or_size_splits=2,
                           axis=1)

    r_state = r * state

    candidate_input = [x, r_state]

    if self._candidate_linear is None:
        with vs.variable_scope("candidate"):
            self._candidate_linear = _Linear(
                candidate_input,
                self._num_units,
                True,
                bias_initializer=self._bias_initializer,
                kernel_initializer=self._kernel_initializer)

    c = self._activation(self._candidate_linear(candidate_input))

    new_h = u * state + (1 - u) * c

    period = vs.get_variable(
        "period", [self._num_units],
        initializer=_random_exp_initializer(
```

```
            self._period_init_min, self._period_init_max))
    phase = vs.get_variable(
        "phase", [self._num_units],
        initializer=init_ops.random_uniform_initializer(
            0., period.initial_value))
    ratio_on = vs.get_variable(
        "ratio_on", [self._num_units],
        initializer=init_ops.constant_initializer(self._ratio_on),
        trainable=self._trainable_ratio_on)

    cycle_ratio = self._get_cycle_ratio(time, phase, period)

    k_up = 2 * cycle_ratio / ratio_on
    k_down = 2 - k_up
    k_closed = self._leak * cycle_ratio

    k = array_ops.where(cycle_ratio < ratio_on, k_down, k_closed)
    k = array_ops.where(cycle_ratio < 0.5 * ratio_on, k_up, k)

    new_h = k * new_h + (1 - k) * state

    return new_h, new_h
```

B PRNN Layer Stacking Wrapper

To enable our phased RNN cells to stack into multiple layers, we had to modify the original TensorFlow layer stacking wrapper `tensorflow.contrib.rnn.MultiRNNCell`[49]. The reason for this is that this wrapper expects RNN cells to only need input in terms of its own state from a preceding step as well as a feature vector or hidden vector. Our rewritten `call` method allows the time stamps of the steps in a sequence to be passed into all layers of the network as input, along side either an input vector or hidden state vector. This is in contrast to the `call` method of the `MultiRNNCell`, which just passes along the hidden state, and thus does not provide the information a phased layer needs to function. The code is tested and confirmed to work in TensorFlow 1.8.

```
from tensorflow.contrib.rnn import MultiRNNCell
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import variable_scope as vs
from tensorflow.python.util import nest


class MultiPRNNCell(MultiRNNCell):
    """ Phased RNN cell composed sequentially of multiple simple phased cells.

    For used with cells that take input on the form (time, x), like
    tf.contrib.rnn.PhasedLSTMCell
    """

    def call(self, inputs, state):
        """Run this multi-layer cell on inputs, starting from state."""
        cur_state_pos = 0
        (time, cur_inp) = inputs
        new_states = []
        for i, cell in enumerate(self._cells):
            with vs.variable_scope("cell_{}".format(i)):
                if self._state_is_tuple:
                    if not nest.is_sequence(state):
                        raise ValueError(
                            ("Expected state to be a tuple of length {}, "
                             "but received: {}".format(
                                len(self.state_size), state)))
                    cur_state = state[i]
                else:
                    cur_state = array_ops.slice(state, [0, cur_state_pos],
                                                [-1, cell.state_size])
                    cur_state_pos += cell.state_size
                cur_inp, new_state = cell((time, cur_inp), cur_state)
                new_states.append(new_state)

        new_states = (tuple(new_states) if self._state_is_tuple else
                      array_ops.concat(new_states, 1))

        return cur_inp, new_states
```