Parallelizing Legendre Memory Unit Training

by

Narsimha R. Chilkuri

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Systems Design Engineering

Waterloo, Ontario, Canada, 2021

Author's Declaration

Abstract

Recently, a new recurrent neural network (RNN) named the Legendre Memory Unit (LMU) was proposed and shown to achieve state-of-the-art performance on several benchmark datasets. Here we leverage the linear time-invariant (LTI) memory component of the LMU to construct a simplified variant that can be parallelized during training (and yet executed as an RNN during inference), resulting in up to 200 times faster training. We note that our efficient parallelizing scheme is general and is applicable to any deep network whose recurrent components are LTI systems. We demonstrate the improved accuracy and decreased parameter count of our new architecture compared to the original LMU and a variety of published LSTM and transformer networks across seven benchmarks. For instance, our LMU sets a new state-of-the-art result on psMNIST, and uses half the parameters while outperforming DistilBERT and LSTM models on IMDB sentiment analysis.

Acknowledgements

I would like to thank Chris Eliasmith for his supervision, support and encouragement over the past two years. I appreciate how he has always made time for me, despite his busy schedule.

I am thankful for my housemates and friends, for if I have retained my sanity during these uprecedented$^{\text{TM}}$ times, it is due to them.

I would also like to thank my labmates, especially Hugo and Aaron, for the many interesting and helpful conversations.

Thanks to my committee Jeff Orchard and Javad Shafiee for agreeing to read my thesis on such short notice.

Finally, I wish to thank my family – Mom, Dad, Aparanji, Vasu and Anagha – for, well, everything.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Legendre Memory Unit (LMU) [80] is an RNN cell that is constructed by coupling an LTI system to a nonlinear one. The LTI system, known as the Delay Network [81], is the core component of the LMU that projects a sliding window of length θ of the input signal onto the Legendre basis – hence the name Legendre Memory Unit. This construction, which employs the Delay Network (DN) to act as the memory and a nonlinear recurrent layer to compute arbitrary functions across time, has been shown to outperform LSTMs and other RNNs on various tasks. Of special interest to us is the ability of the LMU to handle temporal dependencies across 100,000 time-steps, which is orders of magnitude better than the LSTM.

However, the LMU, being an RNN, processes information in an inherently sequential manner. This prevents parallelization within training examples, and thus, it is slow to train. This sequential nature of RNNs is one of the critical reasons for the shift towards purely self-attention based architectures such as the transformer and its derivatives [79, 24, 14], especially in the domain of Natural Language Processing (NLP). Parallelization makes training such networks far more efficient on today's commodity GPU hardware, and thus allows them to be applied to enormous datasets – for example, the Colossal Clean Crawled Corpus (C4) which comprises 750GB of English text [67]. Hence, such models, via unsupervised pre-training, make it possible for us to exploit resources such as the internet, which produces 20TB of text data each month. A feat such as this, from the training perspective, would be unimaginable using LMU or LSTM based models.

In this thesis, we explore a simplified variant of the LMU that lends itself to parallel training. We achieve this by pruning certain connections in the original architecture such that recurrence exists only in the linear system. After showing that the training can be parallelized, we turn to the question of its effectiveness. We do so by first considering the two synthetic tasks that the original LMU was validated on: psMNIST and Mackey-Glass. We show that our variant exceeds the original LMU's performance on both of these tasks, while also establishing a new state-of-the-art result for RNNs on psMNIST. We also test our models against LSTM models of comparable sizes on several NLP tasks. We look at sentiment classification (IMDB), semantic similarity (QQP), and natural language inference (SNLI), and show that our models achieve better performance while using significantly fewer parameters (up to 650x). We then briefly investigate the transfer learning abilities of our model by training a language model on the (unlabelled) Amazon Reviews dataset and using that to improve the IMDB score. Here, we show that it outperforms DistilBert, a transfomer based model, while using 50% fewer parameters. We conclude our NLP experiments by demonstrating superior performance on language modelling (text8) and machine translation (IWSLT'15 En-Vi). Finally, we show that these architectural modifications result in training times that are up to 200 times faster, relative to the original LMU.

We note that while we experiment extensively with the Delay Network, our parallelization scheme applies generally to any deep architecture whose recurrent components are LTI systems. That is, our method is based on exploiting the general idea that linear recurrence relations can be *solved*. More specifically, a linear time-invariant (LTI) system's state update equation can be written in a non-sequential fashion [3]. This allows for the computation of the hidden state of an LTI system to be done in parallel, thus overcoming the fundamental constraint that traditional RNNs suffer from. Despite this desirable property, to our knowledge, such systems have remained underexplored. We hope that our work encourages the design of other LTI cells, with differing properties.

Before we discuss the details of the parallel LMU, we examine some of the foundational concepts in deep learning in the next chapter.

Chapter 2

Foundations of Deep Learning

Arguably, the history of deep learning is a history of priors. This is the lens through which we look at the subject in the first section of this chapter. We examine five priors (or inductive biases) that have been central to deep learning, and then present the Legendre Memory Unit as introducing another prior. Wherever pertinent, we take detours to introduce concepts that are used later on in the thesis. After examining the architectural modifications that help with generalization, in the second section, we consider the problem of how learning actually happens in the network; we introduce the currently popular first-order methods and then briefly discuss methods that make use of second-order information.

2.1 Priors

2.1.1 Why Priors?

We can borrow a simple example from [30] to illustrate the utility of priors. Imagine that we are trying to learn a function where the target (ground truth) function classifies a two-dimensional region into a checkerboard pattern; see Figure (2.1). If the only assumption we make about the function we are trying to learn is its smoothness, that is,

$$f(\boldsymbol{x} + \epsilon) \approx f(\boldsymbol{x}),$$

learning the ground truth would require at least as many labeled training examples as there are squares, which, in this simple case is 25. In the absence of training points in a particular square, there would be no way for the model, at the time of inference, to correctly classify

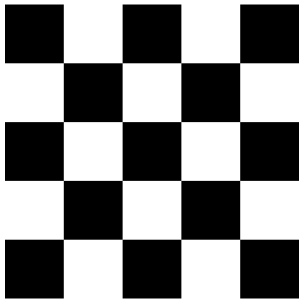

Figure 2.1: Target function that classifies a 2D region into a checkerboard pattern, where the black squares belong to a certain class and the white squares belong to another class.

previously unseen data that fall into such regions. We can reduce the reliance of the model on the availability of training data by introducing additional information, such as the fact that the target function is periodic with a fixed period. Such a prior significantly reduces the search space of functions we are trying to learn, and we can, in theory, identify the ground truth using just one labeled training example.

This approach of employing additional priors on top of the smoothness assumption is the approach that deep learning takes, except that the additional priors are general and not tailored to one specific problem, as in this example. In the following few pages, we examine five different priors that are at the core of deep learning: distributed representations, compositionality, translational equivariance, temporal equivariance and permutation invariance. We then discuss the recently introduced recurrent neural network, the Legendre Memory Unit [80], as introducing another (general) prior, which we call *delay memory*.

2.1.2 Distributed Representations

Connectionist or neural network models can be classified into two groups: those that employ *local representations* and those that employ *distributed representations*. The former class of models use a straightforward scheme whereby each neuron represents just one concept (or item; see Figure (2.2; 2.3)). Hence, each neuron comes with an identifiable interpretation. These models are thus relatively easy to design and understand. For example, consider the simple local model described in [77] that may be used to make inferences about whether a person is shy:

> You meet Alice and learn that she likes programming, so you think she might be a computer geek and therefore shy. On the other hand, you learn that she likes

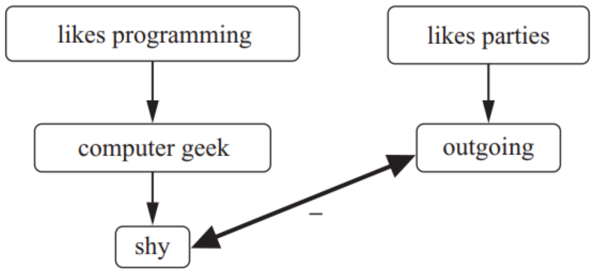

Figure 2.2: A simple local model. Boxes represent individual neural units and their degree of activation represents the applicability of a concept or truth of a preposition; thin lines represent excitatory connections; and thick lines represent inhibitory connections. Picture obtained from [77].

> parties, which suggests that she is outgoing. In forming a coherent impression of her, you have to decide whether she is actually shy or outgoing. The network in Figure (2.2) uses a unit to represent each trait and has one-way excitatory links that make activation flow from the observed behaviors to the inferred traits. It also has a symmetric inhibitory link between shy and outgoing, reflecting the fact that it is hard to be both.

In contrast, deep learning models employ *distributed representations*. Here, the models *learn* to represent concepts in more complex ways, and there is usually no one-to-one correspondence between concepts and neurons. Each neuron takes part in representing many different concepts, and many different neurons are involved in representing one concept.

Hinton et al. [39] point out that one of the important virtues of distributed representations is the ability to generalize automatically to novel situations. Consider the example of a (neural) language model which learns to assign probabilities to sentences, even new ones.

$$\text{P}(\text{``What a piece of work is a man!''}) = \,?$$

Along with learning a smooth probability distribution over the training set of sentences, we also leave it to the model to learn a distributed representation of words in the corpus. Thus, given a sentence, the model first learns to map each word to a vector – the localist approach would be to use a one-hot encoding of the words, where the i-th word is mapped to a vector whose i-th component is one and the rest are zero. For simplicity, we can assume that the model computes the sentence representation via an operation such as addition: it just sums

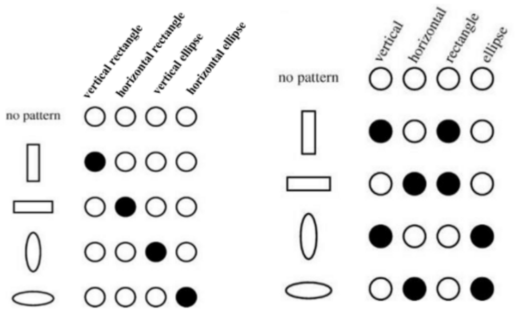

Figure 2.3: Local representation (on the left) and distributed representation (on the right) example. Image taken from [78].

all the word vectors together to come up with the sentence representation (in practice one might use a recurrent neural network). Given such a network, Bengio et al. [7] note that being exposed to a sentence such as "The cat is walking in the bedroom" in the training data, the model assigns a similar probability (and rightly so) to a new, previously unseen sentence such as "The dog is walking in the bedroom," owing to the smoothness of the function and the similarity of the representations of the words dog and cat that the model learns during training. Following the same logic, one can imagine many other sentences that the model naturally generalizes to: "A dog was running in a room," "The cat is running in a room," "A dog is walking in a bedroom," and "The dog was walking in the room."

2.1.3 Composition of Features

The central idea that distinguishes deep learning from other machine learning approaches is *composition of features.* Simple machine learning algorithms rely exclusively on the *smoothness* prior,

$$f(\boldsymbol{x} + \epsilon) \approx f(\boldsymbol{x}),$$

in order to generalize well. Such an approach to generalization works reliably well in regions neighboring labeled training data points. Thus, as long as there is enough training data spread around regions of interest, we can expect such algorithms to work well. While this is a reasonable expectation when we are dealing with low-dimensional data, we are faced with the *curse of dimensionality* as we move to higher dimensions, which we have to do if we are to deal with central problems in AI – for example, simple image recognition models work with 256^2 dimensional inputs. How do we deal with problems that involve working with high dimensional inputs? The answer is to use additional priors.

The success of feed-forwards networks can be attributed to the additional compositional structure that we impose on the function we are trying to learn [30]. Mathematically, this can be written as:

$$f(x) = \sigma_n A_n \circ \sigma_{n-1} A_{n-1} \circ \ldots \circ \sigma_1 A_1 x,$$

where σ_i represents a non-linearity and A_i represents an affine transformation. Initially, motivation for this prior did not come from a theoretical argument but instead from the following two facts: (1) neurons in our brains also have this compositional structure; and (2) we are good at tasks such as classifying images and recognizing speech. More recent explorations into this question [50] have revealed some deep connections between this idea and physics. It is argued that the utility of this structure comes from the "ubiquity of hierarchical and compositional generative processes in physics and other machine-learning applications."

It might be worth pointing out that while feedforward networks have the universal approximation property, this is not unique to neural networks. Support Vector Machines (SVMs) have also been shown to be universal approximators [37].

2.1.4 Translational Equivariance (CNNs)

When we consider another important class of deep networks, convolutional networks, we come across other interesting priors such as *translational equivariance* and *translational invariance* that give convolutional networks an edge over traditional feedforward methods when dealing with image data. We can make the notions of equivariance and invariance a bit more precise in the following way. Given a function $g(x)$, which, say, translates the input, a function $f(x)$ is said to be equivariant to g if it satisfies

$$f(g(x)) = g(f(x)),$$

and we say that f is invariant to g if the following equality is satisfied

$$f(g(x)) = f(x).$$

The utility of these priors has to do with the translational symmetries of image data. For example, consider the images of a cat in Figure (2.4). Although the pixels that represent the cat are situated at several different places, humans can easily identify all the images to be that of a cat. The fact that the cat is translated should have no effect on the final classification. Therefore, we would like our image recognition model to behave similarly.

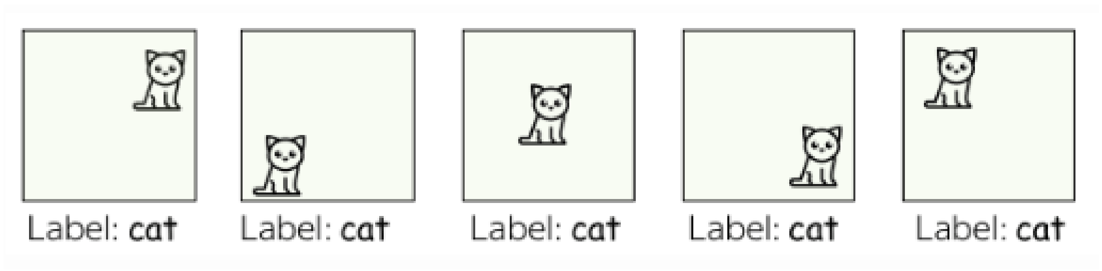

Figure 2.4: Translated images of a cat. Image taken from lena-voita's blog post.

That is, if it sees the leftmost image of the cat in the training set, we want it to generalize readily to the many different translated versions of the same image.

These two desirable properties are built into the neural network architecture by means of the convolution and pooling operations, respectively. More recent work [20] has focused on making the convolutional layers equivariant to rotations and reflections.

2.1.5 Temporal Equivariance (RNNs)

How should we deal with problems where inputs $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n\}$ are presented in a sequential manner to the system?[1] As noted in [26], the most obvious solution is to "parallelize time" by giving it a spatial representation. That is, instead of presenting the inputs sequentially, we buffer the input stream until we receive the complete sequence, and then we present it all at once to the model for processing. In such cases, the first dimension of the parallelized input vector represents the first temporal input, the second dimension the second temporal event and so on.

$$\underbrace{\{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n\}}_{\text{sequential input}} \longrightarrow \underbrace{\begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix}}_{\text{Buffered}} \longrightarrow \text{Feedforward Neural Network}$$

Elman [26] points out that such an approach of representing time, however, has a couple of issues. First, such an approach requires us to define a priori the length of the (parallelized/buffered) input vector. Having a fixed length is unnatural as it would not readily

[1]Here, we take $\boldsymbol{x}_i \in \mathbb{R}^d$ to be the input to the system at time $t = i$.

allow for the processing of something as fundamental as language, where one encounters inputs of variable lengths. Second, an approach that relies on processing the whole sequence at once cannot readily deal with patterns that are shifted temporally. This can be illustrated by considering a simple example. Suppose we train a feedforward network to recognize patterns such as "111", i.e., three ones occurring in a row. Such a network, owing to how it processes vectors, even if it sees the following two examples in the training set,

$$\{0, 1, 1, 1, 0, 0, 0, 0, 0\},$$
$$\{0, 0, 1, 1, 1, 0, 0, 0, 0\},$$

it may not generalize to recognize the same pattern in a shifted sequence such as,

$$\{0, 0, 0, 0, 1, 1, 1, 0, 0\}.$$

Natural language is full of such examples. A feedforward network trained to identify "when" something happens would have trouble with sentences such as "Yesterday, I watched a movie" and "I watched a movie yesterday."

Motivated by the above considerations, especially in the case of processing linguistic information, Elman [26] introduces a simple recurrent network architecture:

$$\boldsymbol{h}_t = \sigma_h(\boldsymbol{W}_h \boldsymbol{x}_t + \boldsymbol{U}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h), \tag{2.1}$$
$$\boldsymbol{y}_t = \sigma_y(\boldsymbol{U}_y \boldsymbol{h}_t + \boldsymbol{b}_y), \tag{2.2}$$

where $\boldsymbol{x}_t$ is the input and $\boldsymbol{y}_t$ is the output at time t; $\boldsymbol{h}_t$ is the hidden state; $\boldsymbol{W}_h \in \mathbb{R}^{h \times d}$ is the input matrix, $\boldsymbol{U} \in \mathbb{R}^{h \times h}$ are the recurrent weights, and $\boldsymbol{b} \in \mathbb{R}^h$ are the bias weights. Recurrent Neural Networks (RNN) make use of *weight sharing*, where the weights $\boldsymbol{W}, \boldsymbol{U}$ and $\boldsymbol{b}$ are shared across time-steps, and this provides RNNs with the ability to generalize to sequence lengths not seen during training, and the statistical strength across different sequence lengths and across different positions in time.

The same structure, however, introduces other significant issues such as vanishing and exploding gradients. The problem of vanishing gradients in particular hinders the ability of RNNs to learn *long-term dependencies* – according to Bengio el al. [9], a task displays long-term dependencies if computation of the desired output at time t depends on input presented at an earlier time $\tau \ll t$. Long Short-Term Memory (LSTM) networks [41] were constructed to alleviate this specific issue of capturing long-term dependencies, and since we compare our model against various LSTM models in the next chapter, we examine LSTMs in some detail next.

Long Short-Term Memory

Gated RNNs, especially Long Short-Term Memory (LSTM), have found wide adoption in practical applications. The LSTM has a long and rich history of more than two decades. The first variant of the LSTM [41] was introduced in 1995 with only two gates, the input and the output gate, and it also featured what is known as *full gate recurrence*. The forget gate was introduced later in 1999 [28], and the peephole connections were introduced subsequently [29].

The state update equations of an LSTM containing the forget gate and peephole connections can be written as:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma_g(\boldsymbol{W}_f\boldsymbol{x}_t + \boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{p}_f \circ \boldsymbol{c}_{t-1} + \boldsymbol{b}_f), && (2.3)\\
\boldsymbol{i}_t &= \sigma_g(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{p}_i \circ \boldsymbol{c}_{t-1} + \boldsymbol{b}_i), && (2.4)\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \sigma_c(\boldsymbol{W}_c\boldsymbol{x}_t + \boldsymbol{U}_c\boldsymbol{h}_{t-1} + \boldsymbol{b}_c), && (2.5)\\
\boldsymbol{o}_t &= \sigma_g(\boldsymbol{W}_o\boldsymbol{x}_t + \boldsymbol{U}_o\boldsymbol{h}_{t-1} + \boldsymbol{p}_o \circ \boldsymbol{c}_t + \boldsymbol{b}_o), && (2.6)\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \circ \sigma_h(\boldsymbol{c}_t), && (2.7)
\end{aligned}$$

where $\boldsymbol{x}_t \in \mathbb{R}^d$ is the input at time t; $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$ (both $\in \mathbb{R}^h$) are the hidden and cell states; $\boldsymbol{W}_i \in \mathbb{R}^{h\times d}$ are the input weights, $\boldsymbol{U}_i \in \mathbb{R}^{h\times h}$ are the recurrent weights, $\boldsymbol{p}_i \in \mathbb{R}^N$ are the peephole connections and $\boldsymbol{b}_i \in \mathbb{R}^h$ are the bias weights; $\boldsymbol{f}_t, \boldsymbol{i}_t, \boldsymbol{o}_t$ are the forget, input and output gates respectively.

Out of all the above mentioned architectural components, a thorough study [35] consisting of around 5400 experimental runs has identified the output activation function, σ_h, and the forget gate, $\boldsymbol{f}_t$, to be the most essential. In addition, they have found that removing the peephole connections or adding the full gate recurrence as in the first LSTM variant did not lead to significant changes. Hence, a more widely used variant of the LSTM is defined as follows:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma_g(\boldsymbol{W}_f\boldsymbol{x}_t + \boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{b}_f), && (2.8)\\
\boldsymbol{i}_t &= \sigma_g(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{b}_i), && (2.9)\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \sigma_c(\boldsymbol{W}_c\boldsymbol{x}_t + \boldsymbol{U}_c\boldsymbol{h}_{t-1} + \boldsymbol{b}_c), && (2.10)\\
\boldsymbol{o}_t &= \sigma_g(\boldsymbol{W}_o\boldsymbol{x}_t + \boldsymbol{U}_o\boldsymbol{h}_{t-1} + \boldsymbol{b}_o), && (2.11)\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \circ \sigma_h(\boldsymbol{c}_t). && (2.12)
\end{aligned}$$

Even with the above simplification, the update equations of the LSTM are still relatively cumbersome and resource intensive, and there has been a lot of work on simplifying the architecture, to make the inner-workings more comprehensible and to reduce the parameter count. Most notable of the latter attempts is the GRU [17], which uses only one state and two gates:

$$\boldsymbol{z}_t = \sigma_g(\boldsymbol{W}_z \boldsymbol{x}_t + \boldsymbol{U}_z \boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \tag{2.13}$$

$$\boldsymbol{r}_t = \sigma_g(\boldsymbol{W}_r x_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{2.14}$$

$$\hat{\boldsymbol{h}}_t = \sigma_h(\boldsymbol{W}_h \boldsymbol{x}_t + \boldsymbol{U}_h(\boldsymbol{r}_t \circ \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{2.15}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \circ \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \circ \hat{\boldsymbol{h}}_t \tag{2.16}$$

Other notable attempts include the light-GRU [68], which outperforms LSTMs on ASR tasks, and the minimal GRU [90], a single gated RNN which works as well as GRUs on a few tasks while using fewer parameters. It is also worth mentioning the work on IRNNs [48] which, with a single hidden state and a clever initialization strategy, i.e., without the use of gating mechanisms, was shown to perform comparably (although using extensive hyperparameter tuning) to LSTMs on some sequential tasks.

However, none of the simplified variants have managed to outperform the LSTM on a wide range of tasks, and LSTMs have been the mainstay of deep learning when it comes to dealing with sequential data, until the introduction of self-attention based architectures such as the transformer [79], which we consider below.

2.1.6 Permutation Invariance (Transformers)

Self-attention based architectures such as the Transformer [79] have been extremely successful in dealing with sequential data, especially in the domain of NLP [79, 66, 24]. In addition to parallelizing training and reducing computation complexity,[2] self-attention based architectures are much better at handling long-range dependencies relative to LSTMs. The improved capacity to handle long-range dependencies could be attributed to fact that the self-attention mechanism, which is nothing but a weighted sum, is invariant to the permutation of the set over which the sum is computed. This is advantageous because the operation is blind to the order of sequence, which in turn means that, as far as the attention mechanism is concerned, the term “long-range” has no meaning – since there is no distinction between long-range and short-range.

More formally, the self-attention function computes, for every position, a weighted average of the feature representations of all other positions with a weight proportional to a

[2] In cases where the dimension of the representation is much greater than the length of the sequence.

similarity score between the representations [42]. Let $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ denote the sequence of n feature vectors of dimension d, and let $\boldsymbol{W}_Q \in \mathbb{R}^{d\times p}, \boldsymbol{W}_K \in \mathbb{R}^{d\times p}$ and $\boldsymbol{W}_V \mathbb{R}^{d\times p}$ represent the three matrices that project the input sequence. The self-attention mechanism then executes the following computation:

$$\begin{aligned}
\boldsymbol{Q} &= \boldsymbol{X}\boldsymbol{W}_Q,\\
\boldsymbol{K} &= \boldsymbol{X}\boldsymbol{W}_K,\\
\boldsymbol{V} &= \boldsymbol{X}\boldsymbol{W}_V,\\
\text{Self-attention}(\boldsymbol{X}) &= \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{p}}\right)\boldsymbol{V}.
\end{aligned}$$

The matrices resulting from projecting the input, $\boldsymbol{Q}, \boldsymbol{K}$, and $\boldsymbol{V}$ are known as the query, key and value matrices, respectively. It should be added that the softtmax operation is applied row-wise to $\boldsymbol{Q}\boldsymbol{K}^T$.

Self-attention is closely related to the attention mechanism which operates on two input sequences of potentially different lengths. If we take the example of neural machine translation [4, 51], and define the source and target sentences as $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ and $\boldsymbol{X'} \in \mathbb{R}^{n'\times d}$, then the attention (not self-attention) computes the following

$$\begin{aligned}
\boldsymbol{Q} &= \boldsymbol{X}'\boldsymbol{W}_Q,\\
\boldsymbol{K} &= \boldsymbol{X}\boldsymbol{W}_K,\\
\boldsymbol{V} &= \boldsymbol{X}\boldsymbol{W}_V,\\
\text{Attention}(\boldsymbol{X}) &= \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{p}}\right)\boldsymbol{V},
\end{aligned}$$

where the resulting sequence would be of the form $\mathbb{R}^{n'\times p}$, instead of $\mathbb{R}^{n\times p}$. In other words, the output sequence is of the same length as the target sentence. Notice that the only difference in the above computation is in how we compute the query matrix $\boldsymbol{Q}$.

It should be noted that the attention mechanism itself has been around for a while. Alex Graves [32], in 2013, used a location based attention mechanism to help with handwriting synthesis, which involves generating a sequence of pen trajectories as output given a sequence of characters as input. Bahdanu [4] first employed the attention mechanism to help with the problem of neural machine translation using an RNN based encoder-decoder architecture. The use of attention for translation applications using RNNs was further explored by Luong and Manning [51]. Vaswasni et al. [79] later proposed the transformer, which completely gets rid of recurrence.

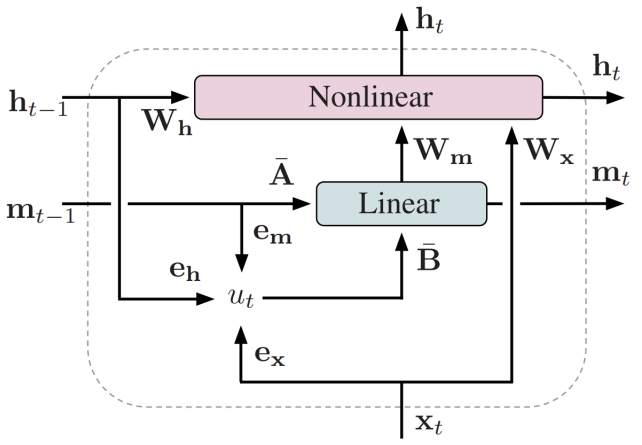

Figure 2.5: Architecture of the Legendre Memory Unit. Image taken from [80].

2.1.7 Delay Memory (LMUs)

With the above discussion of priors, one crucial thing to notice is that the priors are very general. There is not a lot of task specific or dataset specific information in them. For example, convolution networks can be applied to a variety of structured data such as images, and recurrent networks can be applied to a variety of sequential data.

In the same spirit, the newly proposed RNN named the Legendre Memory Unit (LMU) [80] is built around a linear time-invariant (LTI) system where the matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ are pre-specified, i.e., they need not be learned during training. This LTI system optimally realizes the delay operation [81], and hence, we refer to this prespecified structure as *delay memory*; see Figure (2.5). The LTI system is then coupled to a non-linear dynamical system in which the parameters are learned as usual. As we show later, the prior of delay memory is a very general assumption and is not tailored to any one specific dataset. It is this structure that gives LMUs advantage over LSTMs on various synthetic tasks; for example, on a variant of the pathological copy task – called the capacity task [80], which involves storing and reproducing up to 100,000 continuous values – the LMU exhibits far superior performance than the LSTM.

In this work, we compare a variant of the LMU to LSTMs and transformer based approaches on several standard sequential tasks.

2.2 Learning

In the previous chapter, we studied the priors that make deep learning architectures effective. However, specifying the neural network architecture – which includes the structure that comes from priors – only roughly identifies the search space of functions, the family of functions we are interested in. We are still a long way from having a functional model, i.e., a network that does something useful. For example, in the case of a simple feedforward network, specifying that we are looking for a function of the form

$$f(x) = \sigma_n A_n \circ \sigma_{n-1} A_{n-1} \circ \ldots \circ \sigma_1 A_1 x,$$

only identifies a subset of the functional space to search, those that are smooth and have the compositional structure, leaving open infinitely many possibilities. We still have to identify what the millions of weights that parameterize the network ought to be.

How do we specify the weights so that the network yields a desired behavior? We use data as well as the answers expected from data to traverse the last mile; see Figure (2.6) for an illustration. The idea is to *train* the model on many stimulus-response pairs to identify the weights, instead of manually specifying the parameters of the model. The model *learns* the weights, and there is little explicit programming. Surprisingly, it turns out that such an approach – where we first pick a neural network architecture to roughly identify the space of functions and then use data to learn the explicit final form – is very effective when it comes to solving tasks such as speech recognition, image recognition and machine translation.

In the next section, we get into the details of how the weights are learned in a network.

2.2.1 Gradient Based Optimization

Gradient based learning has been a critical component of deep learning, so much so that some[3] have proposed rebranding deep learning, which itself is a rebranding of neural networks, as differential programming. Gradient based methods rely on updating the parameters of the model iteratively using first and second-order gradient information. To do so, we start by defining an error function that quantifies how poorly the network is doing at achieving its goals, and then follow the direction defined by a matrix-vector product in order to minimize the error objective. More formally, given an error function, l, that

[3]LeCun's post on Facebook: https://www.facebook.com/yann.lecun/posts/10155003011462143, and Christopher Olah's post: http://colah.github.io/posts/2015-09-NN-Types-FP/

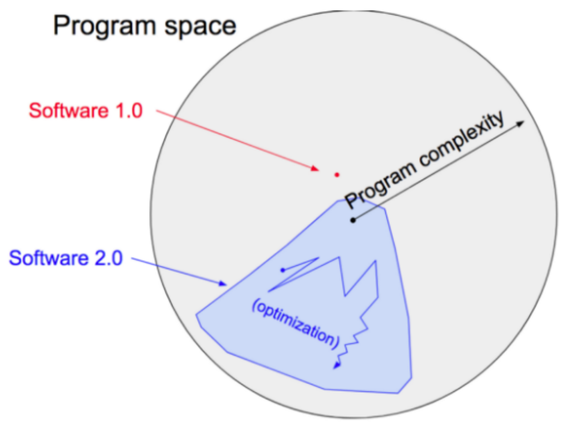

Figure 2.6: The image is taken from a blog post by Andrej Karpathy titled 'Software 2.0'. In the picture, software 1.0 refers to the classical way of programming, and software 2.0 refers to deep learning, which according to Karpathy, "represents the beginning of a fundamental shift in how we develop software." The image illustrates the idea that specifying the neural network architecture identifies the rough skeleton of the code, and gradient based learning uses data to figure out the rest of the details.

measures the discrepancy between the output of the neural network and the ground truth, the total error $\mathcal{L}$ can be written as

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} l(f_\theta(\boldsymbol{x}_i), y_i),$$

where N represents the number of training examples. Given such a function, the parameters of the neural network, θ, are usually updated iteratively as

$$\theta_{k+1} = \theta_k - \eta \boldsymbol{F}^{-1} \nabla \mathcal{L}(\theta_k), \tag{2.17}$$

where k is the iteration number, η is the step-size and $\boldsymbol{F}$ is known as the curvature matrix. In the case of first-order methods, we set the curvature matrix, $\boldsymbol{F}$, to the identity matrix, which then reduces the above equation to the familiar vanilla gradient descent:

$$\theta_{k+1} = \theta_k - \eta \nabla \mathcal{L}(\theta_k).$$

It should be noted that a more general definition of the first-order methods can be given as follows:

$$\theta_{k+1} = \theta_k - \text{Linear Combination}\{\nabla\mathcal{L}(\theta_0), \nabla\mathcal{L}(\theta_1), \ldots, \nabla\mathcal{L}(\theta_k)\},$$

which not only takes into account the vanilla gradient descent, but also covers augmented first-order methods such as momentum and conjugate gradient descent. In case of vanilla gradient descent for example, we simply use the most recent gradient scaled by a constant (step-size).

If we take $\boldsymbol{F}$ to be the Hessian, then we end up with what is known as the Newton's method, and if we set $\boldsymbol{F}$ to be the Fisher Information matrix, we obtain the method of the natural gradient [54]; in a later section, we briefly examine these two methods.

How do we go about finding derivatives of the loss function? We present four choices here, of differing strengths and weaknesses, with a focus on computing the first derivatives.

Manual Differentiation While computing derivatives by hand is exact, it is also time-consuming and error prone. Having to spend a considerable amount of time analytically solving for derivatives also disincentivises architectural modifications, which is not ideal.

Numerical Differentiation Numerical differentiation can be implemented by computing the difference of a function evaluated at two different points. For a function $f : \mathbb{R} \to \mathbb{R}$, we have

$$\frac{df}{d\theta} \approx \frac{f(\theta + h) - f(\theta)}{h}.$$

An advantage of this method is that it is simple to understand and easy to implement.

Symbolic Differentiation Symbolic differentiation[4] takes in a closed-form expression, and based on the rules of differentiation, manipulates the input to produce the output expression. For example, given the expression $f(x) = x\cos(x) + 3$, a symbolic differentiation

[4]Packages like Theano, Mathematica and Maple implement symbolic differentiation.

engine would make use of the following subset of the finite set of differentiation rules

$$\frac{dc}{dx} = 0, \ \text{ c is a constant,} \tag{2.18}$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}, \tag{2.19}$$

$$\frac{d\cos x}{dx} = -\sin x, \tag{2.20}$$

to output the derivative $\frac{df}{dx} = \cos x - x\sin x$. While numerical differentiation provides the derivative at a single point, symbolic differentiation provides a formula for every point in the domain of the function.

Automatic Differentiation Automatic differentiation (AD) relies on dividing the expression of interest into elementary operations and then using the chain rule to compute the derivatives. Automatic differentiation (AD) comes in two flavors: forward mode and reverse mode. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, forward mode is more suitable when $n \ll m$ and reverse mode is far more efficient when $n \gg m$. The computation here is exact, only constrained by machine precision.

In deep learning, we need to compute the gradient of the loss function $\mathcal{L}(\theta)$, and this function is usually of the form $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$. This puts numerical differentiation at a disadvantage because it needs to perform $O(n)$ evaluations to compute the gradient, and n often tends to be in millions – it should be noted that while this method only computes an approximation to the derivative that is dependent on both the step-size h and the round-off error, the noise resiliency of neural networks makes this a minor issue [6]. While symbolic differentiation comes with several advantages, one of the major issues with this technique however is that it results in what is known as exponential swell, where the output expressions get exponentially larger than the expression whose derivative they represent [6]. Given the form of the loss function, reverse mode AD can efficiently and exactly compute the derivatives, and this is the preferred method for computing the derivatives in deep learning. The widely used backpropagation algorithm is a specific instance of reverse mode AD.

2.2.2 Second-Order Methods

In this section we consider methods where, instead of using $\boldsymbol{F} = \boldsymbol{I}$ in equation (2.17), we set it equal to the Hessian or the Fisher Information matrix. The latter two approaches are examples of second-order methods. One way to motivate the exploration of second-order methods is by looking at the lower and upper bounds of first-order methods.

To achieve $\mathcal{L}(\theta) - \mathcal{L}(\theta^*) \leq \epsilon$, with θ^* being the optimal parameters, the number of gradient update iterations k satisfies:

1. (Worst-case) lower bound for 1st-order methods: $k \in \Omega\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$,
2. Upper bound for gradient descent: $k \in O\left(\kappa\log\frac{1}{\epsilon}\right)$,
3. Upper bound for Nesterov's momentum: $k \in O\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$,

where κ is, loosely speaking, the ratio of the maximum curvature and minimum curvature. This quantity tends to be very large for loss functions of neural networks. We see from above that Nesterov's momentum methods are, theoretically speaking, as good as it gets with first-order methods. Thus, in a sense, there cannot be any more major algorithmic improvements if we restrict ourselves to first-order methods, and this motivates the use of second-order methods.

Newton's Method

The Hessian of a function $f(\theta)$ where $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\boldsymbol{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial\theta_1^2} & \dfrac{\partial^2 f}{\partial\theta_1\,\partial\theta_2} & \cdots & \dfrac{\partial^2 f}{\partial\theta_1\,\partial\theta_n} \\ \dfrac{\partial^2 f}{\partial\theta_2\,\partial\theta_1} & \dfrac{\partial^2 f}{\partial\theta_2^2} & \cdots & \dfrac{\partial^2 f}{\partial\theta_2\,\partial\theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial\theta_n\,\partial\theta_1} & \dfrac{\partial^2 f}{\partial\theta_n\,\partial\theta_2} & \cdots & \dfrac{\partial^2 f}{\partial\theta_n^2} \end{bmatrix},$$

or equivalently, using index notation, we can write

$$[\boldsymbol{H}]_{i,j} = \frac{\partial^2 f}{\partial\theta_i\,\partial\theta_j}.$$

Setting $\boldsymbol{F} = \boldsymbol{H}$ in equation (2.17), we end up with the following:

$$\theta_{k+1} = \theta_k - \eta \boldsymbol{H}^{-1} \nabla \mathcal{L}(\theta_k). \tag{2.21}$$

This is what is know as the Newton's method. Although, in theory, it can converge faster than first-order methods, in practice, it comes with two issues, especially when it comes to optimizing neural networks. They have to do with (1) the non-convexity of the neural network loss functions; and (2) the large number of parameters that neural networks employ in order to solve problems.

First, Newton's method does not distinguish between minima, maxima and saddle points of the loss function – it just looks for stationary points. In other words, at every iteration, it first constructs a local second-order Taylor approximation of the underlying loss function, and proceeds to maximize or minimize this approximation. Thus, Newton's method can result in updates where the $\mathcal{L}(\theta_{k+1}) > \mathcal{L}(\theta_k)$. Hence, it cannot be applied directly to the problem of non-convex optimization.

Second, as the number of parameters increases, the Hessian becomes expensive to compute, difficult to store, and very expensive to invert. For example, say we have a model with a million parameters, $\boldsymbol{H}$ then contains 10^{12} entries, which, assuming 4 bytes per entry, we would need about 4TB of memory to store the matrix. Additionally, matrix inversion has a computational complexity of $O(n^3)$, which is very problematic when n is in the order of millions, which is usually the case when dealing with deep networks.

Fortunately, the first issue, Newton's method being attracted to saddle points and maxima, is relatively easy to deal with. One popular way of circumventing the issue is through second-order regularization, i.e., using $(\boldsymbol{H} + \alpha \boldsymbol{I})^{-1}$ in the update equation; this is done to make the negative eigenvalues of the Hessian positive, or at least less negative [22].

The second issue, however, is not so trivial to solve, and there are many ways of attacking the problem. One popular approach, known as the conjuagte Newton method, deals with the issue by iteratively solving for $\boldsymbol{H}^{-1}\nabla\mathcal{L}$, instead of explicitly computing the Hessian, inverting and then computing the matrix-vector product. In other words, instead of explicitly solving for the stationary point of the following second-order expansion,

$$\mathcal{L}(\theta + d) \approx \mathcal{L}(\theta) + \nabla \mathcal{L}(\theta)^T d + \frac{1}{2} d^T \boldsymbol{H}(\theta) d, \tag{2.22}$$

which, of course, is given by $d^* = -\boldsymbol{H}(\theta)^{-1}\nabla\mathcal{L}(\theta)$, we minimize the equation (2.22) iteratively using conjugate gradients.

Next, we focus on a method known as Kronecker-factored Approximate Curvature [54] that uses a block matrix approximation of the curvature matrix to alleviate the storage and computational burden.

Kronecker-factored Approximate Curvature (K-FAC)

Given a neural network that defines a conditional probability density function $f(y|x,\theta)$, the fisher information matrix $\mathcal{I}$ is defined as:

$$\begin{aligned}\left[\mathcal{I}(\theta)\right]_{i,j} &= -\operatorname{E}\left[\frac{\partial^2}{\partial\theta_i\,\partial\theta_j}\log f(y|x,\theta)\right],\\ &= \operatorname{E}\left[\left(\frac{\partial}{\partial\theta_i}\log f(y|x,\theta)\right)\left(\frac{\partial}{\partial\theta_j}\log f(y|x,\theta)\right)\right],\end{aligned}$$

where the expectation is taken with respect to the function $f(y|x,\theta)$ and the empirical training distribution. The K-FAC method [54] makes use of such a matrix in place of the Hessian to make the parameter updates.

This method hinges on three key ideas:

1. Using the Fisher matrix instead of the Hessian. The Fisher seems to provide a more global picture of the underlying function, that is, a second-order Taylor expansion that employs the Fisher gives a better approximation of the global structure, whereas the second-order expansion that uses the Hessian results in the optimal local approximation, which may not be desirable.

2. Approximating the Fisher as a block diagonal matrix, which is much easier (computationally) to invert. The inverse of a block diagonal matrix can be evaluated by inverting the blocks individually as shown below:

$$\begin{bmatrix}\mathbf{A}_1 & 0 & \cdots & 0\\ 0 & \mathbf{A}_2 & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \mathbf{A}_n\end{bmatrix}^{-1} = \begin{bmatrix}\mathbf{A}_1^{-1} & 0 & \cdots & 0\\ 0 & \mathbf{A}_2^{-1} & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \mathbf{A}_n^{-1}\end{bmatrix}.$$

3. Exploiting a property of Kronecker products to make the matrix inversion even simpler. Writing each of the block diagonal entries as a Kronecker product of two smaller matrices, the inverse of the block matrix can be computed by inverting the

(smaller) matrices involved in the product. For example,

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}.$$

In the absence of residual connections, which tend to significantly smooth the loss landscape, they show that their method leads to better generalization relative to first-order methods such as Adam and momentum-based methods.

2.3 Discussion

In this chapter, we have tried to highlight the importance of inductive biases in deep learning. In the process, we examined five priors that make core deep learning algorithms effective. Additionally, we have examined LSTMs, one of the popular gated RNNs, and transformers, models that rely completely on the attention mechanism. We also attributed LMU's effectiveness over LSTMs and other RNNs to the additional innate structure (pre-specified LTI system) that it comes with. We then considered the question of learning in deep networks, and briefly examined first-order methods, regular second-order methods and the associated challenges of computing, storing and inverting large matrices, and finally we looked at the K-FAC method, which offers a way around the aforementioned problems.

Chapter 3

Parallelizing Legendre Memory Unit

Our model is directly inspired by the success of self-attention. Self-attention based architectures have come to replace RNN based approaches for problems such as language modelling, machine translation, and a slew of other NLP tasks [66, 67]. Three properties that make self-attention desirable over RNNs are: (1) it is better at handling the challenging problem of long-range dependencies; (2) it is purely feedforward; and (3) when the sequence length is smaller than the dimension of representation, which is not uncommon in NLP applications, self-attention is computationally cheaper than an RNN.

Of these three desirable properties, our model inherits the first one from the original LMU, and satisfies properties (2) and (3) by construction. Additionally, the capability to run our model in a recurrent manner during inference can be advantageous in situations where there are memory constraints or where low-latency is critical.

3.1 Background and LMU Variants

In this section we start by introducing the delay problem and show how a delay is optimally realized by the DN, an LTI system. We then introduce the LMU which employs a single-input DN coupled to a nonlinear dynamical system to process sequential data. Finally, we introduce our model, which is obtained by simplifying the original architecture. We show how this simplification allows for training to be parallelized (often reducing the computation complexity), while also retaining the ability to handle long range dependencies of the original formulation.

3.1.1 Delay Network

Ideal Delay

A system is said be an ideal delay system if it takes in an input, $u(t)$, and outputs a function, $y(t)$, which is the delayed version of the input. Mathematically, this can be described in the following manner:

$$y(t) = \mathcal{D}[u(t)] = \begin{cases} 0 \quad t < \theta \\ u(t-\theta) \quad t \geq \theta \end{cases}, \tag{3.1}$$

where D is the ideal delay operator and $\theta \in \mathbb{R}$ is the length of the delay. There are two things of note here: (1) the ideal delay system is linear, i.e., for any two functions, $f(t)$ and $g(t)$, and any $a, b \in \mathbb{R}$, it respects the following equation:

$$\mathcal{D}[af(t) + bg(t)] = a\mathcal{D}[f(t)] + b\mathcal{D}[g(t)]; \tag{3.2}$$

and (2) although this problem looks deceptively simple, in fact it takes a system with infinite memory to take in a continuous stream of input (with unspecified frequency content), store it for θ seconds and then reproduce the entire input without error.

These two considerations tell us that the optimal system that implements delay must be linear and that even the most optimal physical implementation can only approximately realize the delay.

Approximating Delay

If we are interested in constructing a dynamical system that implements delay, thanks to the linearity constraint, we can narrow our search space from the general system of ODEs of the form

$$\dot{\boldsymbol{m}} = \mathbf{f}(\boldsymbol{m}, u), \tag{3.3}$$

$$y = \mathbf{g}(\boldsymbol{m}, u), \tag{3.4}$$

where $\boldsymbol{m}$ and u represent the state of the system and a scalar input respectively, to just finding the four matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ that define an LTI system:

$$\dot{\boldsymbol{m}} = \boldsymbol{A}\boldsymbol{m} + \boldsymbol{B}u, \tag{3.5}$$

$$y = \boldsymbol{C}\boldsymbol{m} + \boldsymbol{D}u. \tag{3.6}$$

Following the derivation of the Delay Network in [81], we start by considering the transfer function of the delay system, which for a Single-Input-Single-Output (SISO) system is defined as:

$$G(s) = \frac{y(s)}{u(s)} = e^{-\theta s}, \tag{3.7}$$

where $y(s)$ and $u(s)$ are found by taking the Laplace transform of the input and output functions in time. As expected, this defines an infinite dimensional transfer function, capturing the intuitive difficulty of constructing a continuous delay.

The transfer function can be converted to a finite, causal state space realization *if and only if* it can be written as a proper[1] ratio of finite dimensional polynomials in s [13]. $G(s)$, however, is irrational, i.e., it cannot be written as a proper, finite dimensional ratio. Therefore, making an approximation is necessary.

We can achieve an optimal convergence rate (in the least-square error sense) for rational approximants by means of Padé approximants [61]. Choosing the order of the numerator to be one less than the order of the denominator and accounting for numerical issues in the state-space realization (see [81] for details), gives the following canonical realization:

$$A_{i,j} = \frac{(2i+1)}{\theta} \begin{cases} -1 & i < j \\ (-1)^{i-j+1} & i \geq j \end{cases}, \tag{3.8}$$

$$B_i = \frac{(2i+1)(-1)^i}{\theta}, \tag{3.9}$$

$$C_i = (-1)^i \sum_{l=0}^{i} \binom{i}{l} \binom{i+l}{j} (-1)^l, \tag{3.10}$$

$$D = 0, \quad i, j \in [0, d-1], \tag{3.11}$$

where we refer to d as the order of the system. The LTI system $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ is what is known as a Delay Network (DN). The order of the system, d, and the delay length, θ are the main hyperparameters to choose when using a DN. Higher order systems require more

[1]A ratio $\frac{a(s)}{b(s)}$ is said be proper if the order of the numerator does not exceed the order of the denominator.

resources, but provide a more accurate emulation of the ideal delay. Because we have used Padé approximants, each order is optimal for that dimension of state vector $\boldsymbol{m}$.

Legendre Polynomials

Suppose we construct a system using the $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ matrices defined above, and provide it with an input signal, $u(t)$. Given the state $\boldsymbol{m}_t$, we can use $\boldsymbol{C}$ to decode $u(t-\theta)$ to a degree of accuracy determined by the order of our system, i.e.,

$$u(t-\theta) \approx \boldsymbol{C}^T \boldsymbol{m}_t. \tag{3.12}$$

Intuitively, given $\boldsymbol{m}_t$, it seems possible to decode not only $u(t-\theta)$ but also $u(t-\theta') \ \forall \ 0 \leq \theta' \leq \theta$. This can be done using a slightly modified $\boldsymbol{C}$ for a given θ':

$$u(t-\theta') \approx \boldsymbol{C}(\theta')^T \boldsymbol{m}_t, \tag{3.13}$$

where

$$C_i(\theta') = (-1)^i \sum_{l=0}^{i} \binom{i}{l} \binom{i+l}{j} \left(-\frac{\theta'}{\theta}\right)^l, 0 \leq \theta' \leq \theta, \tag{3.14}$$

and $\boldsymbol{C}(\theta' = \theta)$ corresponds to the $\boldsymbol{C}$ defined in equation (3.10). Interestingly, the functions in (3.14) turn out to be the shifted Legendre polynomials.

3.1.2 Legendre Memory Unit

The LMU is obtained by coupling a single-input delay network to a nonlinear dynamical system. The DN orthogonalizes the input signal across a sliding window of length θ, whereas the nonlinear system relies on this memory to compute arbitrary functions across time. The state update equations that define the LMU are given below:

$$u_t = \boldsymbol{e}_x^T \boldsymbol{x}_t + \boldsymbol{e}_h^T \boldsymbol{h}_{t-1} + \boldsymbol{e}_m^T \boldsymbol{m}_{t-1}, \tag{3.15}$$
$$\boldsymbol{m}_t = \bar{\boldsymbol{A}} \boldsymbol{m}_{t-1} + \bar{\boldsymbol{B}} u_t, \tag{3.16}$$
$$\boldsymbol{h}_t = f(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_m \boldsymbol{m}_t), \tag{3.17}$$

where $\bar{\boldsymbol{A}}$ and $\bar{\boldsymbol{B}}$ are the discretized versions[2] of $\boldsymbol{A}$ and $\boldsymbol{B}$. These matrices are usually frozen during training, although they need not be. The input to the LTI system, $u(t)$, is

[2]Using zero-order hold and $dt = 1$, exact discretization gives $\bar{\boldsymbol{A}} = e^{\boldsymbol{A}}$ and $\bar{\boldsymbol{B}} = \boldsymbol{A}^{-1}(e^{\boldsymbol{A}} - \boldsymbol{I})\boldsymbol{B}$.

computed by projecting the input to the RNN cell, $\boldsymbol{x}_t \in R^{d_x}$, the hidden state, $\boldsymbol{h}_t \in \mathbb{R}^{d_h}$, and the memory state, $\boldsymbol{m}_t \in \mathbb{R}^d$, onto their respective encoding weights ($\boldsymbol{e}_x, \boldsymbol{e}_h,$ and $\boldsymbol{e}_m$). The memory state, hidden state and the input to the cell are then combined using the weight matrices ($\boldsymbol{W}_x, \boldsymbol{W}_h,$ and $\boldsymbol{W}_m$) and passed through the nonlinearity, f. The encoding vectors and the kernel matrices are learned during training.

3.1.3 Our Model

In this paper, we propose a modified LMU architecture by making two changes to the equations defined above. First, we remove certain connections in (3.15) and (3.17), namely ($\boldsymbol{e}_h^T, \boldsymbol{e}_m^T$) and ($\boldsymbol{W}_h$); this is done to aid parallelization. The following simple ablation study done on the psMNIST dataset sheds some light on the effect of pruning these connections. We tested the performance of the original LMU under five (self-explanatory) conditions: No $\boldsymbol{e}_h$; No $\boldsymbol{e}_m$; No $\boldsymbol{W}_h$; No $\boldsymbol{W}_m$; and no activation function f. We see from Table (3.1) that removing the recurrent connections ($\boldsymbol{e}_h, \boldsymbol{W}_h$) is beneficial, and although there is utility to having $\boldsymbol{e}_m$, it unfortunately hinders parallelization.

Table 3.1: Effect of removing the recurrent connections in the LMU, measured on psMNIST

Model	Accuracy
LMU (original)	97.15
LMU (no $\boldsymbol{e}_h$)	97.27
LMU (no $\boldsymbol{e}_m$)	96.82
LMU (no $\boldsymbol{W}_h$)	97.42
LMU (no $\boldsymbol{W}_m$)	20.10
LMU (no f)	89.81

Second, instead of projecting the input to the RNN cell, $\boldsymbol{x}_t$ down to a scalar as in (3.15), we implement a general affine transformation followed by an element-wise nonlinearity; the original architecture is better suited for dealing with 1D or low-dimensional inputs, and this is a straightforward generalization of the encoding equation for higher dimensional inputs. Additionally, adding a bias term to equation (3.17), we end up with the following:

$$\boldsymbol{u}_t = f_1(\boldsymbol{U}_x \boldsymbol{x}_t + \boldsymbol{b}_u), \tag{3.18}$$

$$\boldsymbol{m}_t = \bar{\boldsymbol{A}} \boldsymbol{m}_{t-1} + \bar{\boldsymbol{B}} \boldsymbol{u}_t, \tag{3.19}$$

$$\boldsymbol{o}_t = f_2(\boldsymbol{W}_m \boldsymbol{m}_t + \boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{b}_o). \tag{3.20}$$

Note that equations (3.18) and (3.20) are equivalent to having *time-distributed* dense layers before and after equation (3.19). In general, we expect our model to be modified and used in combination with other feed-forward layers, including self-attention. For example, we found a gated architecture [74] where equation (3.18) is modified to

$$\boldsymbol{u}_t = f_1(\boldsymbol{W}_u \boldsymbol{x}_t + \boldsymbol{b}_u) \cdot \boldsymbol{g} + \boldsymbol{x}_t \cdot (1 - \boldsymbol{g}),$$

to work well for the addition problem (results not shown). The gate is defined as $\boldsymbol{g} = \sigma(\boldsymbol{W}_g \boldsymbol{x}_t + \boldsymbol{b}_g)$, i.e., a sigmoid-activated affine transformation where the bias is initialized to -1.

Additionally, now that the input to the LTI system in this case is a vector, $\boldsymbol{u}_t \in \mathbb{R}^{1 \times d_u}$, we have that $\boldsymbol{m}_t \in \mathbb{R}^{d \cdot d_u}$, and equation (3.16) can be thought of as implementing the following:

$$\boldsymbol{m}_t = \text{reshape}(\bar{\boldsymbol{A}} \ \text{reshape}(\boldsymbol{m}_{t-1}, (d,\ d_u)) + \bar{\boldsymbol{B}} \boldsymbol{u}_t,\ d \cdot d_u). \tag{3.21}$$

Parallel Training

One of the motivations for the above mentioned architectural changes is that the model now has only one recurrent connection: $\boldsymbol{m}_t$'s dependence on itself from the past in equation (3.19). But because this is an LTI system, standard control theory [3] gives us a non-iterative way of evaluating this equation as shown below[3]

$$\boldsymbol{m}_t = \sum_{j=1}^{t} \bar{\boldsymbol{A}}^{t-j} \bar{\boldsymbol{B}} u_j. \tag{3.22}$$

[3]Restricting ourselves to 1D input for the purposes of illustration.

Table 3.2: Complexity per layer and minimum number of sequential operations of various architectures. n is the sequence length, d_x is the input dimension, d is the order, and k is the size of the kernel. First three rows are as reported in [79].

Layer Type	Complexity	Sequential Ops
RNN	$O(n \cdot d_x^2)$	$\surd$
Convolution	$O(k \cdot n \cdot d_x^2)$	$\times$
Attention	$O(n^2 \cdot d_x)$	$\times$
DN (3.19)	$O(n \cdot d^2 \cdot d_x)$	$\surd$
DN (3.24)	$O(n^2 \cdot d \cdot d_x)$	$\times$
DN (3.25)	$O(n \cdot d \cdot d_x)$	$\times$
DN (3.26)	$O(n \cdot \log n \cdot d \cdot d_x)$	$\times$

Defining $\boldsymbol{H} = \begin{bmatrix} \bar{\boldsymbol{A}}^0 \bar{\boldsymbol{B}} & \bar{\boldsymbol{A}} \bar{\boldsymbol{B}} & \ldots \end{bmatrix} \in \mathbb{R}^{d \times n}$ and

$$\boldsymbol{U} = \begin{bmatrix} u_1 & u_2 & u_3 & \ldots & u_n \\ & u_1 & u_2 & \ldots & u_{n-1} \\ & & u_1 & \ldots & u_{n-2} \\ & & & \ddots & \vdots \\ & & & & u_1 \end{bmatrix} \in \mathbb{R}^{n \times n}, \tag{3.23}$$

the above convolution equation can alternatively be written as a matrix multiplication:

$$\boldsymbol{m}_{1:n} = \boldsymbol{H}\boldsymbol{U}, \tag{3.24}$$

where n is the sequence length. Given that $\boldsymbol{H}$ is the impulse response of the LTI system, in practice we compute $\boldsymbol{H}$ by feeding in an impulse to the RNN version of the DN (equation (3.19)). Note that in our method the $\bar{\boldsymbol{A}}$ and $\bar{\boldsymbol{B}}$ matrices are frozen during training, so the impulse response need only be computed once. In case of multi-dimensional inputs, we would repeat the above computation several times, once for each dimension of the input. It is also evident from the structure of the $\boldsymbol{U}$ matrix that although this reformulation turns the DN into a feedforward layer, it still respects causality. In other words, the state $\boldsymbol{m}_t$ depends only on the inputs seen until that point of time, i.e., $u_i : i \leq t$.

Complexity

With the new formulation, we immediately see that it is computationally (i.e., in terms of the number of operations) advantageous in situations where we only need the final state (`return_sequences=False` in Keras terminology). Instead of using (3.19) to explicitly simulate the first $n-1$ steps in-order to compute the state at the time-step n, we can instead just compute

$$\boldsymbol{m}_n = \boldsymbol{H}\boldsymbol{U}_{:n}, \tag{3.25}$$

thus reducing the complexity of computing the final state, $\boldsymbol{m}_n$, from $O(n \cdot d^2 \cdot d_x)$ to $O(n \cdot d \cdot d_x)$, where n is the sequence length, d is the order, and d_x is the dimension of the input. We show in Section (3.2.6) that using this implementation results in up to 200x speedup.

The more general computation (3.24), although parallelizable, results in a complexity of $O(n^2 \cdot d \cdot d_x)$. This can be made more efficient by employing the convolution theorem which gives us an equivalent way of evaluating the convolution in the Fourier space as[4]

$$\boldsymbol{m}_{1:n} = \mathcal{F}^{-1}\{\mathcal{F}\{\boldsymbol{H}\} \cdot \mathcal{F}\{\boldsymbol{U}_{:n}\}\}. \tag{3.26}$$

Thanks to the fast Fourier transform, the above operation has a complexity of $O(n \cdot \log_2 n \cdot d \cdot d_x)$. These, other algorithms, and their complexities are reported in Table (3.2).

It was argued in [79] that a self-attention layer is cheaper than an RNN when the representation dimension of the input, d_x, is much greater than the length of the sequence, n, which is seen in NLP applications. For example, standard word or sub-word based machine translation systems work with sequence lengths of about 100 and representation dimension ranging from 300 to 512. The same argument holds for the DN layer too, since we have found the inequality $\log_2 n \cdot d \ll d_x$ to hold in all our word-based NLP experiments – this excludes text8, which works at the level of characters [58]. More specifically, we use $d \leq 4$ for our word-based models.

Recurrent Inference

Machine learning algorithms are usually optimized for training rather than deployment [21], and because of that models need to be modified, sometimes non-trivially, to be more

[4]Assuming a padded Fourier transform across the appropriate axis and automatic broadcasting when computing element-wise multiplication. See `LMUFFT` code for details.

suitable for inference. For example, one of the metrics that is crucial for applications such as online Automatic Speech Recognition (ASR) is low latency. Transformers have shown faster training times and better performance than RNNs on ASR, but since they employ (global) self-attention, which requires the entire input to begin processing, they are not natural fits for such a task [84]. Look-ahead [88] and chunk-based [57] approaches are usually used to alter the architecture of self-attention for such purposes. While our model can be trained in parallel, because of the equivalence of equations (3.19) and (3.26), it can also be run in an iterative manner during inference, and hence can process data in an online or streaming fashion during inference.

3.2 Experiments

In the following experiments we compare our model against the LMU, LSTMs and transformers. With these experiments, we focus on benchmarking rather than establishing new state-of-the-art results. Hence, we constrain ourselves to train all the models, with the exception of text8, using the Adam optimizer [43] with all the default parameter settings, which we believe attests to the ease with which our model can be trained. For text8, we found it helpful to reduce the learning rate by a factor of 10 halfway into training. Although unusual, we use the default optimization settings even when transfer learning.

With models we compare against, we use results found in published work, even when they make use of more sophisticated architectures, learning schedules or regularization schemes.

In the original LMU paper, the authors consider the capacity task [80] (a variant of the pathological copy task) which involves storing and reproducing up to 100,000 continuous values. We need not do so here because they employ just the delay layer without nonlinear dynamics in order to deal with this problem, which makes their architecture essentially the same as ours. In other words, our model also exhibits the same performance on this task, which is orders of magnitude better than the LSTM (in terms of MSE).

3.2.1 psMNIST

The permuted sequential MNIST (psMNIST) dataset was introduced by [48] to test the ability of RNNs to model complex long term dependencies. psMNIST, as the name suggests, is constructed by permuting and then flattening the (28×28) MNIST images. The permutation is chosen randomly and is fixed for the duration of the task, and in order to

Table 3.3: psMNIST results. The first three rows are from [80], and the fourth row is from [36].

Model	Accuracy
LSTM	89.86
NRU	95.38
LMU	97.15
HiPPO-LegS	98.3
Our Model	**98.49**

stay consistent, we use the same permutation as [15] and [80]. The resulting dataset is of the form (samples, 784, 1), which is fed into a recurrent network sequentially, one pixel at a time. We use the standard 50k/10k/10k split.

Architecture As was pointed out in [80], in order to facilitate fair comparison, RNN models being tested on this task should not have access to more than $28^2 = 784$ internal variables. Keeping that in mind, and in order to make direct comparisons to the original LMU model, we consider a model with $d = 468$ dimensions for memory. We set the dimension of the output state to 346 and use $\theta = 784$. Our model uses 165k parameters, the same as all the models reported in Table (3.3), except for the original LMU model, which uses 102k parameters, and the HiPPO-LegS model, which is reported to use 512 hidden dimensions (number of parameters is unknown).

Results & Discussion Test scores of various models on this dataset are reported in Table (3.3). Our model not only surpasses the LSTM model, but also beats the current state-of-the result of 98.3% set by HiPPO-LegS [36] recently. Thus, our model sets a new state-of-the art result for RNNs of 98.49% on psMNIST. It is interesting that our model, despite being simpler than the original LMU, outperforms it on this dataset. This suggests that the main advantage of the LMU over past models is the quality of its temporal memory, implemented by the DN.

3.2.2 Mackey-Glass

Mackey-Glass equations are a set of two nonlinear differential equations that were originally developed to model the quantity of mature cells in the blood [53]. The second of these

Table 3.4: Mackey-Glass results.

Model	NRMSE
LSTM	0.059
LMU	0.049
Hybrid	0.045
Our Model	0.044

equations is interesting because it can result in chaotic attractors and is used to construct the dataset at hand. This is a time-series prediction task where we need to predict 15 time-steps into the future.

Architecture [81] compare three architectures on this dataset. The first one uses 4 layers of LSTMs ($h = 28$), the second one replaces LSTMs with LMUs ($d = 4, \theta = 4$), and the final one replaces the first and third layers in the first model with LMUs. We use a relatively simple architecture where we combine our model (single layer) with an additional dense layer. We use $d = 40$, $\theta = 50$, and 1 and 140 units in the input and output layers, with the additional dense layer containing 80 units. We did not try other variations. All the models contain about 18k parameters and are run for 500 epochs.

Results & Discussion NRMSE scores on the test set are presented in Table (3.4). We see that our model outperforms the other three models in accuracy and training time. In the original paper [81] hypothesize that the superior performance of the hybrid model is due to the presence of gates, but given that our model lacks gating mechanisms, we think that it might have to do with the LMU model being misparametrized.

3.2.3 Sentiment, Semantics and Inference

In this section, we explore the supervised and semi-supervised capabilities of our model. More specifically, we first look at the tasks of sentiment analysis (IMDB), semantic similarity (QQP) and natural language inference (SNLI), and then improve upon the IMDB score using a language model pre-trained on the (unlabelled) Amazon Reviews dataset [59].

Supervised

IMDB The IMDB dataset [52] is a standard sentiment classification task containing a collection of 50k highly polar reviews, with the training and testing sets containing 25k reviews each. We use the standard pre-processed dataset available from the Keras website,[5] consider a vocabulary of 20k words, and set the maximum sequence length to 500 words.

QQP For Quora Question Pairs (QQP),[6] given a pair of sentences, the task is to identify whether the two sentences are semantically similar. In this case, we experiment on two train/dev/test splits: 390k/8k/8k like in [73], and 280k/80k/40k like in [72]. We use a vocabulary of 20k words and truncate the sentences to be less than 25 words.

SNLI The Stanford Natural Language Inference[7] was released to serve as a benchmark for evaluating machine learning systems on the task of natural language inference. Given a premise, the task is to determine whether a hypothesis is true (entailment), false (contradiction), or independent (neutral). We use the standard 550k/10k/10k split, consider a vocabulary of 20k words, and set the maximum sequence length to 25 words.

Architecture In our experiments, confusingly, we found the use of the DN, without any nonlinearities, to work well. Therefore, we construct parameter efficient models that employ just the DN layer, with $d = 1$ and $\theta = \texttt{maxlen}$. We use 300D Glove embeddings (840B Common Crawl; [62]) for all our models. For the IMDB task, which is a single sentence task, we encode the sentence and pass the encoded vector to the final classification layer. For two-sentence tasks, QQP and SNLI, we encode the two sentences to produce two vectors, and then pass the vector obtained by concatenating the two vectors, their absolute difference, and their element-wise product to the final classification layer. We compare our models against the LSTM models described in [36] for IMDB, both [73] and [72] for QQP, and [11] for SNLI. They all use at least an order of magnitude more parameters than our models.

Results & Discussion We present the results from the three experiments in Table (3.5). As we can see, our simple models based on the DN alone do indeed outperform the LSTM

[5]https://keras.io/api/datasets/imdb/.
[6]https://www.kaggle.com/c/quora-question-pairs
[7]Dataset and published results are available at https://nlp.stanford.edu/projects/snli/.

Table 3.5: IMDB, QQP and SNLI results. IMDB result is from [36], QQP results are from [73] and [72] respectively, and SNLI is from [11].

	LSTM		Our Model	
	Param.	Acc.	Param.	Acc.
IMDB	50k	87.29	301	**89.10**
QQP	-/800k	82.58/81.4	1201	**86.95/85.36**
SNLI	220k	77.6	3.6k	**78.85**

models. It is also noteworthy that our models use significantly fewer parameters than the LSTM models: 160x on IMDB, 650x on QQP and 60x on SNLI.

Semi-Supervised

Amazon Reviews NLP over the past couple of years has been dominated by transfer learning approaches, where language models pre-trained on large corpora are used to initialize general purpose fine-tuning architectures to help with various downstream tasks. We therefore consider using a pre-trained language model on the Amazon Reviews dataset[8] to improve the previously achieved score on IMDB. Here, we make direct comparisons to the LSTM model described in [65]. While they use the entire dataset (82 million reviews) for pre-training and train their model for a month on 4 GPUs, due to resource constraints, we use less than 5% (3.7 million reviews) and train our model for about 12 hours on a single GPU. We use a vocabulary of 30k words.

Architecture We have found the repeating block architecture, where each block is composed of our model, a few highway layers [74], and a dense layer, when used with skip connections to work well (see Figure (3.1)). Since the inputs, d_x, are high dimensional, using a large θ, which would have to be accompanied by a large order d to maintain resolution, would result in vectors that are $d_x \cdot d$ dimensional, which is not desirable. Therefore, we instead work with smaller θ (and hence smaller d) and use several repeating blocks to take long-term dependencies into account. This is similar to how convolutional networks are used: small kernel sizes but many convolutional layers. If θ_i is the setting we use with

[8]https://jmcauley.ucsd.edu/data/amazon/

Table 3.6: IMDB results with pre-training. First row is from [65], and the second row is from [69].

Model	# parameters (Millions)	Accuracy
LSTM	75	92.88
DistilBERT	66	92.82
Our Model	34	**93.20**

the DN in the i^{th} block, then the effective delay, $\theta_e = \sum_i \theta_i$. In this specific case, we have $\theta_i = 6 \; \forall \; i$, and $\theta_e = 30$. Thus, the model has access to the past 30 tokens when making a prediction. In terms of the fine-tuning performance, like [63], we found that using deep representations, i.e., a linear combination of representations from all the blocks, to be better than using just the output of the top block. For fine-tuning, we compute a weighted sum of the outputs from the language model, and use that to classify a given review as expressing a positive or negative sentiment. Even during fine-tuning, we use the Adam optimizer with all the default settings. We did not experiment with smaller learning rates or other learning schedules.

Results & Discussion Results for this experiment are presented in Table (3.6). Despite using a much smaller pre-training dataset, training for just 12 hours, and using less than 50% of the parameters, our model outperforms the LSTM model described in [65]. We also include a self-attention based model, DistilBert [69], for comparison; it must be noted however that DistilBert was trained on a more general yet much larger dataset (concatenation of English WikiPedia and Toronto Book Corpus).

3.2.4 Language Modelling

text 8 We evaluate our model on character level language modelling using the text8 dataset[9]. It contains 100MB of clean text from Wikipedia and has an alphabet size of 27. As is standard, we use the first 90MB as the training set, the next 5MB as the validation set and the final 5MB as the test set. Following [58] and [89], we set the sequence length to 180.

[9]http://mattmahoney.net/dc/textdata.

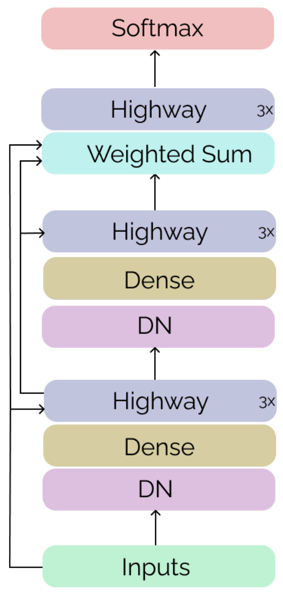

Figure 3.1: Illustration of the language model used for pre-training on the Amazon Reviews dataset. Although the actual model uses five blocks (combination of DN, Dense and Highway), we only show two blocks in the above figure.

Architecture This architecture is similar to the language model used in the above section, except for the use of deep representations; we simply work with the output from the top block.[10] For text8, minor changes were made to adapt the model to deal with longer sequences, which is a consequence of modelling at the level of characters, and to parameter match it to the LSTM model in [89], which uses around 3.2 million weights. We employ three blocks in this case and use all three DNs with the setting $\theta = 15$.

Results & Discussion We report the scores in bits per character in Table (3.7). We see that our model performs better than the LSTM model of similar size. Two observations regarding model optimization are: 1) we noticed that the training loss plateaus around the twelfth epoch, so we found it helpful it to decrease the learning rate by a factor of 10 around then (this is the only dataset for which we alter the optimization parameters); 2) despite that, the training slows down after a few more epochs, and we stop the optimization after 20 epochs; we believe that a more carefully designed learning schedule might be able

[10]We did not test the use of deep representations in this context.

to help with faster convergence. We also observed that using a self-attention layer, which takes into account the whole sequence and not just the previous 45 tokens, after the final block helps with generalization; this perhaps points to the fact that the model needs a context that is longer than 45 tokens in order to make predictions.

3.2.5 Translation

IWSLT'15 En-Vi IWSLT'15 English to Vietnamese[11] is a medium-resource language translation task containing 133k sentence pairs. Following [51], we do not perform any pre-processing other than replacing words that occur less frequently than 5 by the `<unk>` token, which leaves us with vocabulary sizes of 17k and 7.7k for English and Vietnamese, respectively. We use the TED tst2012 as the validation set and TED tst2013 as the test set. We use 300D representation for source and target embeddings.

Architecture For translation, we use a standard encoder-decoder architecture inspired by the Amazon reviews language model, and we also employ an attention layer to help with translation. Our model's about the same size as the 24 million LSTM model described in [51]. Due to time constraints, the architecture and hyperparamters for this problem were relatively underexplored.

Results & Discussion Case sensitive BLEU scores are reported in Table (3.7). Our model brings in an improvement of 2.5 BLEU over the LSTM model. When tested on lower-cased text (without altering the training procedure), we obtained a higher score of 26.2 BLEU. One major limiting factor of this analysis is that we use a small, fixed vocabulary (17k and 7.7k words), with no way of dealing with out-of-vocabulary words. For future work, we intend to experiment with an open-vocabulary encoding scheme such as byte pair encoding [71].

3.2.6 Training Time

Here we explore the effect of parallelization on training time. We refer to architectures that implement the DN using equation (3.19) as the *LTI version* and the ones that use either (3.25) or (3.26), depending on whether `return_sequences` is false or true, as the *parallel version.*

[11]https://nlp.stanford.edu/projects/nmt/

Table 3.7: Language modelling and translation results. text8 score (reported in bpc) is from [89], and IWSLT score (reported in BLEU) is from [51]. ^{a}(case sensitive), ^{b}(case insensitive).

Model	text8	IWSLT'15
LSTM	1.65	23.3
Our Model	**1.61**	$\mathbf{25.9}^{a}/\mathbf{26.7}^{b}$

Results are presented in Figure (3.2). On the left, we compare the speedup we obtain by going from the original LMU to our model, in both LTI and parallel forms, on psMNIST and Mackey-Glass. We first notice that switching from the LMU to the LTI version results in non-trivial gains. This is solely due to the reduction in the number of recurrent connections. On top of this, switching to the parallel version, owing to long sequences (784 and 5000) and, in the case of psMNIST, a reduction in the computational burden, results in substantial improvements in training times of 220x and 64x respectively.[12]

On the right of Figure (3.2), we demonstrate how varying the sequence length of the inputs affects the time it takes to complete one epoch. We see that the LTI version exhibits a linear growth whereas the parallel one is nearly constant over this scale.

3.3 Discussion

With the intention of alleviating the long standing problem of speeding up RNN training, we introduce a variant of the LMU architecture that can process data in a feedforward or sequential fashion. When implemented as a feedforward layer, it can utilize parallel processing hardware such as GPUs and thus is suitable for large scale applications, and when run as an RNN, it is useful in applications where the amount of available memory is a limitation. We also briefly consider the question of computational complexity of our model, and argue for its suitability to large scale applications in the domain of NLP, a direction we will pursue in the future.

While our model has shown competitive performance on several tasks, there is, however, room for improvement. More specifically, although in theory the depth of our model used for

[12]The parallel version of our model is also 34x faster than a parameter-matched cudaLSTM on psMNIST. Additionally, We note that for Mackey-Glass we use a (parameter matched) one layer LMU model instead of the 4 layer model used in [80]; the difference with respect to the original architecture is more drastic, with our model (parallel) being about 200x faster.

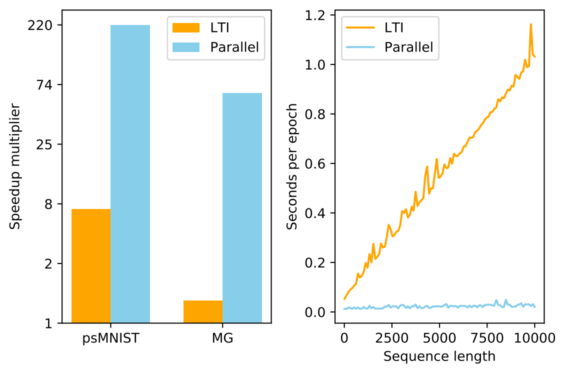

Figure 3.2: (left) The plot shows the speedup we obtain from switching from the LMU to the LTI (in orange) and parallel implementation (in blue) of our model. (right) This show the effect of increasing the sequence length on the LTI and parallel versions of our model. All results were measured using a single GTX 1080.

modelling language is decoupled from its "receptive field", that is not the case in practice. As we have discussed previously, we use our model like a standard CNN model: we use small theta but make the model very deep to increase the receptive field. With this approach, our model would have to be very deep if it is to capture long term dependencies, especially when the text sequences are much longer that what was explored in this work – in NLP, it is quite common to use text sequences longer than 1024 tokens. The major issue with such models is that training *deep* models is non-trivial and comes with a number of issues, the least of them being the model's memory footprint.

Chapter 4

Future Work

While we have used our network on various datasets, the evidence we have presented might be incomplete when it comes to large scale experiments. However, given the encouraging performance of our model on the task of semi-supervised sentiment analysis, where it outperformed LSTM and transformer based models using 50% fewer weights, it will be interesting to see if this parameter efficiency is preserved in general. More specifically, it will be interesting to pre-train the model on a large unlabelled corpus such as C4 [67], and test its performance on benchmarks such as GLUE [83] and SuperGLUE [82]. GLUE, for example, consists of a diverse set of NLP classification tasks, and the leaderboard[1] conveniently keeps track of the performance of various models on these tasks.

For all our experiments, barring text8, we restrict ourselves to use the Adam optimizer with all the default settings. While we did this to narrow the space of hyper-parameters, given the importance and sensitivity of networks to optimization parameters, it would be beneficial to experiment thoroughly with Adam and other popular optimizers. Experiments with second-order optimization methods such as K-FAC would also be very interesting. It would be valuable to compare the effectiveness of such methods on our variant of the LMU and also the original LMU. We think that the original LMU would benefit more from using second-order methods.

On a more general (and obscure) note, we believe in the utility of research focusing on discovering new priors, those that will help us tackle more complex AI tasks. In the case of language acquisition for example, even before infants learn their first word, Yang [86] points out that they are faced with a couple of non-trivial problems:

[1]https://gluebenchmark.com/leaderboard

> But before we talk of language learning we, or rather, the baby, must be clear just what counts as language. Language does not come in neat packets, ready for the child to digest and absorb. Every moment of a baby's life is a wild cocktail party, where the sound of language is embedded within the ambient acoustics that constantly bombard her eardrums. The first step in learning a language is to find it, and to find it, the baby must block out the noise.
>
> Second, unlike the words on this page, which are separable by spaces and punctuation, we—do—not—pause—between—words—in—speech. When we study the acoustic details of speech we see that most words bump into one another, without any delineable boundaries in between[...] To learn words, children have to find them in the continuous stream of speech.

These issues, of course, do not translate into real word worries for children,[2] and the answers seem to do with the linguistic assumptions that children come equipped with. Please refer to [86] and [70] for more details. Research into the addition of such priors to artificial learning systems could be fruitful.

[2]For example, children, even at the age of five days old, can not only distinguish linguistic and non-linguistic information, but also can differentiate between languages that do not fall into the same prosodic group (such as French and English).

References

[1] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[2] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pages 1120–1128, 2016.

[3] Karl Johan Åström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] David Balduzzi and Muhammad Ghifary. Strongly-typed recurrent neural networks. In *International Conference on Machine Learning*, pages 1292–1300. PMLR, 2016.

[6] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[7] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The journal of machine learning research*, 3:1137–1155, 2003.

[8] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. MIT Press, 2007.

[9] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[10] Peter Blouw, Gurshaant Malik, Benjamin Morcos, Aaron R Voelker, and Chris Eliasmith. Hardware aware training for efficient keyword spotting on general purpose and specialized hardware. *arXiv preprint arXiv:2009.04465*, 2020.

[11] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015.

[12] Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. *arXiv preprint arXiv:1603.06021*, 2016.

[13] William L Brogan. *Modern control theory*. Pearson education india, 1991.

[14] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[15] Sarath Chandar, Chinnadhurai Sankar, Eugene Vorontsov, Samira Ebrahimi Kahou, and Yoshua Bengio. Towards non-saturating recurrent units for modelling long-term dependencies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3280–3287, 2019.

[16] Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark A Hasegawa-Johnson, and Thomas S Huang. Dilated recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 77–87, 2017.

[17] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[18] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in neural information processing systems*, pages 577–585, 2015.

[19] Peter Clark, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Turney, and Daniel Khashabi. Combining retrieval, statistics, and inference to answer elementary science questions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[20] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999. PMLR, 2016.

[21] Daniel Crankshaw. *The Design and Implementation of Low-Latency Prediction Serving Systems*. PhD thesis, UC Berkeley, 2019.

[22] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*, 2014.

[23] Joost de Jong, Aaron R Voelker, Hedderik van Rijn, Terrence C Stewart, and Chris Eliasmith. Flexible timing with delay networks–the scalar property and neural scaling.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[25] William B Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[26] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[27] Mehrdad Farajtabar, Navid Azizan, Alex Mott, and Ang Li. Orthogonal gradient descent for continual learning. In *International Conference on Artificial Intelligence and Statistics*, pages 3762–3773. PMLR, 2020.

[28] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[29] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

[30] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

[31] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[32] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[33] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[34] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, 18(5-6):602–610, 2005.

[35] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[36] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Re. Hippo: Recurrent memory with optimal polynomial projections. *arXiv preprint arXiv:2008.07669*, 2020.

[37] Barbara Hammer and Kai Gersmann. A note on the universal approximation capability of support vector machines. *neural processing letters*, 17(1):43–53, 2003.

[38] Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. *arXiv preprint arXiv:1602.06662*, 2016.

[39] Geoffrey E Hinton. Distributed representations. 1984.

[40] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[41] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[42] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.

[43] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[44] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[45] Donald Knuth. *The TEXbook.* Addison-Wesley, Reading, Massachusetts, 1986.

[46] Alex M Lamb, Anirudh Goyal Alias Parth Goyal, Ying Zhang, Saizheng Zhang, Aaron C Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks. In *Advances in neural information processing systems*, pages 4601–4609, 2016.

[47] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[48] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[49] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5457–5466, 2018.

[50] Henry W Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017.

[51] Minh-Thang Luong and Christopher D Manning. Stanford neural machine translation systems for spoken language domains. In *Proceedings of the International Workshop on Spoken Language Translation*, pages 76–79, 2015.

[52] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150, 2011.

[53] Michael C Mackey and Leon Glass. Oscillation and chaos in physiological control systems. *Science*, 197(4300):287–289, 1977.

[54] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[55] Eric Martin and Chris Cundy. Parallelizing linear recurrent neural nets over sequence length. *arXiv preprint arXiv:1709.04057*, 2017.

[56] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. In *Advances in Neural Information Processing Systems*, pages 6294–6305, 2017.

[57] Haoran Miao, Gaofeng Cheng, Changfeng Gao, Pengyuan Zhang, and Yonghong Yan. Transformer-based online ctc/attention end-to-end speech recognition architecture. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6084–6088. IEEE, 2020.

[58] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8:67, 2012.

[59] Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 188–197, 2019.

[60] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[61] Jonathan R Partington. Some frequency-domain approaches to the model reduction of delay systems. *Annual Reviews in Control*, 28(1):65–73, 2004.

[62] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[63] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[64] Jonathan Pilault, Amine Elhattami, and Christopher Pal. Conditionally adaptive multi-task learning: Improving transfer learning in nlp using fewer parameters & less data. *arXiv preprint arXiv:2009.09139*, 2020.

[65] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

[66] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[67] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[68] Mirco Ravanelli, Philemon Brakel, Maurizio Omologo, and Yoshua Bengio. Light gated recurrent units for speech recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):92–102, 2018.

[69] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[70] Jan Schnupp, Israel Nelken, and Andrew King. *Auditory neuroscience: Making sense of sound*. MIT press, 2011.

[71] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[72] Lakshay Sharma, Laura Graesser, Nikita Nangia, and Utku Evci. Natural language understanding with the quora question pairs dataset. *arXiv preprint arXiv:1907.01041*, 2019.

[73] Dinghan Shen, Guoyin Wang, Wenlin Wang, Martin Renqiang Min, Qinliang Su, Yizhe Zhang, Chunyuan Li, Ricardo Henao, and Lawrence Carin. Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms. *arXiv preprint arXiv:1805.09843*, 2018.

[74] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[75] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[76] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE symposium on computational intelligence for security and defense applications*, pages 1–6. IEEE, 2009.

[77] Paul Thagard. *Mind: Introduction to cognitive science*, volume 17. MIT press Cambridge, MA, 2005.

[78] Colin R Tosh and Graeme D Ruxton. *Modelling perception with artificial neural networks*. Cambridge University Press, 2010.

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[80] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre memory units: Continuous-time representation in recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 15544–15553, 2019.

[81] Aaron R Voelker and Chris Eliasmith. Improving spiking dynamical networks: Accurate delays, higher-order synapses, and time cells. *Neural computation*, 30(3):569–609, 2018.

[82] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. *arXiv preprint arXiv:1905.00537*, 2019.

[83] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[84] Chengyi Wang, Yu Wu, Shujie Liu, Jinyu Li, Liang Lu, Guoli Ye, and Ming Zhou. Reducing the latency of end-to-end streaming speech recognition models with a scout network. *arXiv preprint arXiv:2003.10369*, 2020.

[85] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[86] Charles Yang. *The infinite gift: How children learn and unlearn the languages of the world*. Simon and Schuster, 2006.

[87] Jin Yang, Tao Li, Gang Liang, Wenbo He, and Yue Zhao. A simple recurrent unit model based intrusion detection system with dcgan. *IEEE Access*, 7:83286–83296, 2019.

[88] Qian Zhang, Han Lu, Hasim Sak, Anshuman Tripathi, Erik McDermott, Stephen Koo, and Shankar Kumar. Transformer transducer: A streamable speech recognition model with transformer encoders and rnn-t loss. In *ICASSP 2020-2020 IEEE International*

Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 7829–7833. IEEE, 2020.

[89] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Russ R Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. In *Advances in neural information processing systems*, pages 1822–1830, 2016.

[90] Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, 13(3):226–234, 2016.