Australian National University

Vanishing Gradient Problem in training Neural Networks

Muye Chen

June 2022

A thesis submitted in partial fulfilment of the requirements for the degree of Bachelor of Science with Honours in Statistics at the Australian National University.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University, and, to the best of my knowledge and belief, contains no material published or written by another person, except where due reference is made in the thesis.

Muye Chen

Acknowledgements

First and foremost, I would like to express my sincere thanks to my supervisor Professor Andrew Wood. Prior to my Honours program I have spent a semester on a smaller research thesis with Professor Wood. His knowledge, patience and advice are the keys to the completion of both theses. Professor Wood's detailed explanation of statistical and mathematical concepts helped me greatly in understanding the underlying mechanism of the models involved. Additionally I would like to express my gratitude for his advice on scientific report-writing, without which this thesis would not have been possible.

Dr Tao Zou and Dr Yanrong Yang's statistical machine learning courses have instilled my interest in deep learning research. I really appreciate RSFAS for offering these intensive but amazing courses.

Lastly I would like to thank my family and friends for their care and never-ending support over the 4 years of my Bachelor program.

Abstract

The Vanishing Gradient Problem (VGP) is a frequently encountered numerical problem in training Feedforward Neural Networks (FNN) and Recurrent Neural Networks (RNN). The gradient involved in neural network optimisation can vanish and become zero in a number of ways. In this thesis we focus on the following definition of the VGP: the tendency for network loss gradients, calculated with respect to the model weight parameters, to vanish numerically in the *back propagation* step of network training.

Due to the differences in data types on which the two types of networks are trained, the model architectures are different. Consequently the methods to alleviate the problem take different forms and focus on different model components.

This thesis attempts to introduce basic neural network model architectures to readers who are new to deep learning and using neural networks, with a particular focus on how the VGP can affect the model performance. We conduct the relevant research of RNNs in the context of a simple classification task in the field of Natural Language Processing (NLP). We have implemented and analysed the existing solutions to the VGP through mathematical details and graphical results from experiments.

The thesis is extended to two types of advanced RNN-class models which are designed to be resistant to the VGP. However our experimental results reveal that under a strict indicator, the two advanced models instead exhibit stronger tendency of the VGP, than the standard RNN model does. With regards to this finding, we introduce a different viewpoint proposed by Rehmer & Kroll (2020) regarding the VGP in RNN-class models, which support our results. We have discussed its relevance to our experiments and extended their derivations to another RNN-class model they have not covered.

Contents

Abbreviations and terminologies

VGP	Vanishing Gradient Problem
FNN	Feedforward Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
ReLU	Rectified Linear Units
ELU	Exponential Linear Units
PReLU	Parametric Rectified Linear Units
NLP	Natural Language Processing
BPTT	Back Propagation Through Time
CE	Cross-Entropy
SRNN	Standard Recurrent Neural Network. To distinguish from the general RNN-class models
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Units
BRNN, BLSTM, BGRU	Bidirectional-RNN, Bidirectional-LSTM, Bidirectional-GRU

Throughout the thesis we define dimensionality using terms in Tensorflow documentations. We use 'size' to denote the number of components in a vector. We use 'shape' to denote the dimension of a vector, matrix or 3-dimensional tensor.

1 Introduction

1.1 Motivations

A neural network consists of interconnected layers of artificial neurons joined in the form of function compositions which facilitates signal propagation. Varieties of neural networks have achieved state-of-the-art performance on applications such as natural language processing (NLP) and computer vision [Hayou et al, 2019]. Different applications prompt different model architectures. The architecture used on structured data that can be stored in data frames (i.e. rectangular data files) is known as the *feedforward neural network* (FNN). The scope of the thesis will then be extended to the *recurrent neural network* (RNN), the architecture for processing sequential data including text and time series data.

The weight parameters in neural networks are optimised by gradient-based optimisation algorithms, where the aim is to minimise the loss function. However due to causes such as inappropriate paramterisation and the *temporal-dependent* components in gradients of the loss, model training may encounter the Vanishing Gradient Problem (VGP).

As a consequence of the VGP when training FNNs, the model parameter learning process halts before the optimisation procedure reaches a sufficiently low value of the loss function.

In training RNNs, the problem manifests itself as a trade-off between efficient gradient-based learning and relaying information over long periods for predictions (Bengio et al, 1994). The optimisation of the weight parameters will not reflect the past information, and the predictive performance is undermined..

Great efforts by statistics and computer science researchers have been committed to alleviating this problem in recent decades, e.g. Graves (2013) implemented the Long Short-term Memory (LSTM) cells (Hochreiter & Schmidhuber, 1997) to modelling sequential data; Pascanu et al (2013) applied regularisation; Pascanu et al (2012) and Jozefowicz et al (2015) promoted Hessian-free Optimisation in training RNNs. It is important and worthwhile to understand and analyse how different methods are effective or ineffective in alleviating the VGP in training neural networks.

1.2 Overview of the thesis

In section 2, the FNN, standard RNN (SRNN) models are reviewed and relevant literature is discussed. We also provide the details of two existing models designed to alleviate the VGP in RNNs: the Long Short-term Memory (LSTM) cells and the Gated Recurrent Units (GRU) cells.

In section 3, the thesis will describe the FNN model structure and explain why the VGP arises in this setting. Several diagnostic checks have been devised in order to visualise the extent of vanishing gradients. We will review some widely-applied parameterisation techniques that have been designed as solutions to the VGP in training FNNs. The statistics computed from the diagnostics are presented as plots. These statistics are selected on the basis of simplicity and how effective they reflect the extent of the VGP.

We consider application to text sequence classification in section 4. Training RNNs on temporal sequential data is complex and difficult, largely due to the VGP (Sutskever et al, 2011). Visualisations of RNN training include 2 aspects. First, the performance on classification is reflected by training loss and accuracy. Second, the extent of the VGP can be visualised by an indicator first used by Noh (2021) on a different task. We have written code and provided justification for its relevance to our task.

Section 4 will then introduce how the LSTM and GRU alleviate the VGP by analysing the form

of loss gradients in these two models. The standard viewpoint is that LSTM and GRU address the VGP by preserving gradient information, therefore they have pronounced advantages for problems requiring the use of long range contextual information (Graves, 2013).

Section 5 will present a rather surprising result on the extent of the VGP in LSTM and GRU, compared to extent in the SRNN. In particular the gradient indicator we have identified has values vanished more significantly for LSTM and GRU than for SRNN. We did find similar argument in Rehmer & Kroll (2020), whose derivations and conclusions support our experimental results.

A closely related numerical instability problem is the exploding gradient problem, where the loss gradient explodes numerically in training. It is relatively easy to avoid. One can simply enforce a hard constraint over the norm of the gradient (Pascanu et al, 2012; Jozefowicz et al, 2015). The focus of this thesis will therefore be on the VGP, rather than the exploding gradient problem.

2 Literature review

2.1 Feedforward Neural Network

2.1.1 Model structure of FNN

The Feedforward Neural Network (FNN) is designed to be trained on structured data. It consists of an arbitrary number of hidden layers and one output layer on top. In practice it is possible to have any form of connectivity (fully or sparsely) between units of different layers. Here we assume the FNN has its units fully connected between consecutive layers.

Firstly we present a simple version of the FNN model with one hidden layer to clarify the terms and computations in the forward propagation and back propagation.

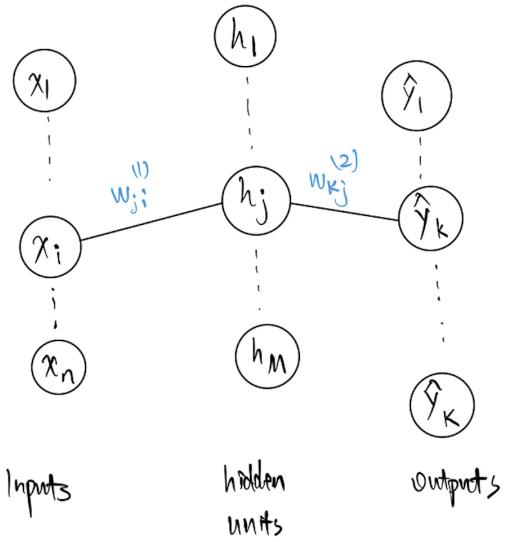

Figure 2.1: FNN with an input layer, a hidden layer and an output layer.

For clarity only the i-th input, j-th hidden unit and the k-th output are shown as connected. Now define the parameterisations. Assume the FNN is applied to a regression problem and we have square-error loss and linear output-layer activation function. The hidden layer activation function is the *hyperbolic tangent* (tanh) function defined below with its derivative:

$$f(a) = \tanh(a) = \frac{exp(a) - exp(-a)}{exp(a) + exp(-a)} \tag{2.1}$$

$$f'(a) = 1 - f(a)^2 \tag{2.2}$$

Given training data $\{(x_i, y_i)\}_{i=1}^n$, each input x_i undergoes the forward propagation, which is described by the following equations:

$$\begin{aligned} a_j^{(1)} &= \sum_{i=0}^{n} w_{ji}^{(1)} x_i \\ h_j^{(1)} &= \tanh(a_j^{(1)}) \\ a_k^{(2)} &= \sum_{j=0}^{M} w_{kj}^{(2)} h_j^{(1)} \\ \hat{y}_k &= a_k^{(2)} \\ L &= \frac{1}{2} \sum_{k=1}^{K} (\hat{y}_k - y_k)^2 \end{aligned}$$

Let $l = 1, 2$ denote the layer index. The input x_i undergoes an affine transformation with the weights on the 1st layer. The result $a_j^{(1)}$ is *activated* by applying the tanh-function to it. The activated scalars $h_j^{(1)}, j = 1, ..., M$ are known as the hidden states in the 1st hidden layer. On the output layer this activated result undergoes another affine transformation, and is further activated by the linear output-layer activation function. The final result of the propagation is the predicted output $\hat{y}_k$. Lastly the square-error loss is computed using the output and the true response y_k.

In general, common choices of the activation function on hidden layers are the sigmoid-class functions and the *rectified linear units*, ReLU-class functions. (2.3) is the standard ReLU. The

sigmoid class functions involve the hyperbolic tangent (2.1) and the logistic sigmoid (2.4).

$$\text{ReLU}(x) = \max(0, x) \tag{2.3}$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2.4}$$

The quantity $w_{j,i}^{(l)}$ is the (j, i)-th entry in the l-th layer weight matrix $W^{(l)}$. It can be visualised as the weight parameter defining the edge which connects the i-th unit in the $(l-1)$-st layer to the j-th unit in the l-th layer.

Note that the two summations for $a_j^{(1)}$ and $a_k^{(2)}$ have their sum indices starting from 0 because we have embedded the corresponding bias vector into the sums, by writing it as $w_{(\cdot,0)}^{(l)}$.

On the output layer the activation function produces outputs appropriate for the actual problem. For example, a regression problem has the linear output activation function; a multiclass classification problem has the *softmax* activation function (2.19) to produce mutually exclusive class probabilities.

2.1.2 Back propagation in FNNs

After all the inputs have been processed, the back propagation step computes the derivatives of the loss function L with respect to all the weight parameters, using the chain rule. Then the parameter values are updated with gradient-based optimisation algorithms. To express neatly the chain rule computations, Bishop (2006) introduced the *back-propagation formula*, which

can be derived as follow:

$$\begin{aligned}\delta_j^{(l)} := \frac{\partial L}{\partial a_j^{(l)}} &= \sum_k \frac{\partial L}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} \\ &= \sum_k \delta_k^{(l+1)} w_{k,j}^{(l+1)} f'(a_j^{(l)}) \\ &= f'(a_j^{(l)}) \sum_k \delta_k^{(l+1)} w_{k,j}^{(l+1)}\end{aligned} \tag{2.5}$$

With this formula, the loss gradient with respect to a weight matrix entry on the l-th layer can be expressed with terms between 2 consecutive layers $l-1$ and l:

$$\frac{\partial L}{\partial w_{j,i}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{j,i}^{(l)}} = \delta_j^{(l)} h_i^{(l-1)} \tag{2.6}$$

For the FNN in Fig 2.1 we compute the partial derivatives $\frac{\partial L}{\partial w_{j,i}^{(1)}}, \frac{\partial L}{\partial w_{k,j}^{(2)}}$. The explicit equations are:

$$\frac{\partial L}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} h_j^{(1)}$$

where,

$$\delta_k^{(2)} = \frac{\partial L}{\partial a_k^{(2)}} = \frac{\partial}{a_k^{(2)}} \frac{1}{2} \sum_{k=1}^{K} (\hat{y}_k - y_k)^2 = a_k^{(2)} - y_k$$

Then on the first layer:

$$\frac{\partial L}{\partial w_{ji}^{(1)}} = \frac{\partial L}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i$$

where,

$$\begin{aligned}
\delta_j^{(1)} &= \frac{\partial L}{\partial a_j^{(1)}} = f'(a_j^{(1)}) \sum_{k=1}^{K} w_{kj}^{(2)} \delta_k^{(2)} \\
&= (1 - (a_j^{(1)})^2) \sum_{k=1}^{K} w_{kj}^{(2)} (a_k^{(2)} - y_k) \\
w_{ji}^{(1)} &\leftarrow w_{ji}^{(1)} - \alpha \frac{\partial L}{\partial w_{ji}^{(1)}} \\
w_{kj}^{(2)} &\leftarrow w_{kj}^{(2)} - \alpha \frac{\partial L}{\partial w_{kj}^{(2)}}
\end{aligned}$$

The Stochastic Gradient Descent (SGD) is a common parameter optimisation algorithm for both FNN and RNN training. The last 2 equations are SGD updating of the weights parameters.

General forward and back propagation equations

The formulae for FNN forward propagation step on an arbitrary layer:

$$a_j^{(l)} = \sum_{i=1}^{N} w_{ji}^{(l)} h_i^{(l-1)}, \quad h_i^0 \text{ is input } x_i \tag{2.7}$$

$$h_j^{(l)} = f(a_j^{(l)}) \tag{2.8}$$

$$a_k^{(l+1)} = \sum_{j=1}^{M} w_{kj}^{(l+1)} h_j^{(l)} \tag{2.9}$$

$$\cdots$$

$$\hat{y}_n = g(a_n^{(L)}) \tag{2.10}$$

where $l = 0, ..., L$ are the indices of layers. 'Loss' is spelled out in the following equations in order to distinguish from the index of the output layer. Result (2.6) is useful for constructing an iterative algorithm. For understanding the general loss gradient in an arbitrary layer l is instead

expressed as a product of full partial derivative, starting from the output layer L to layer l:

$$\frac{\partial \text{Loss}}{\partial w_{ji}^{(l)}} = \frac{\partial a_j^{(l)}}{\partial W_{ji}^{(l)}} \frac{\partial h_j^{(l)}}{\partial a_j^{(l)}} \frac{\partial a_k^{(l+1)}}{\partial h_j^{(l)}} \cdots \frac{\partial h_\cdot^{(L-1)}}{\partial a_\cdot^{(L-1)}} \frac{\partial a_\cdot^{(L)}}{\partial h_\cdot^{(L-1)}} \frac{\partial \text{Loss}}{\partial a_\cdot^{(L)}} \tag{2.11}$$

Each of the subscript dots towards the end in (2.11) denotes an arbitrary unit in the vector. In the k-th iteration of SGD, the back propagation step updates the weight parameters component-wise, with a predetermined learning rate α:

$$w_{j,i}^{(l)} \leftarrow w_{j,i}^{(l)} - \alpha \frac{\partial \text{Loss}}{\partial w_{j,i}^{(l)}} \tag{2.12}$$

The updated weights are involved in the $(k+1)$-st iteration of forward propagation.

2.2 Recurrent Neural Network

2.2.1 RNN in general

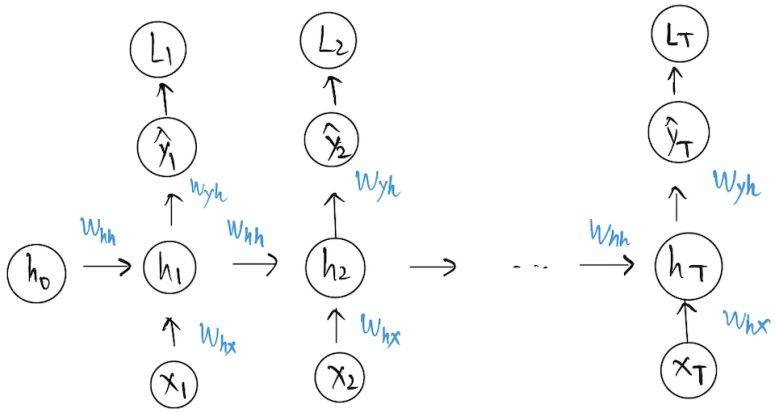

Figure 2.2: General RNN diagram

Goodfellow et al (2016) provided a basic account of the RNN model architecture in the context of a discrete multiclass response: In sequential data suppose each input sequence has timesteps ranging from $t = 1$ to τ. Forward propagation begins with a specification of the initial hidden state h_0, and computes outputs at each step:

$$
\begin{aligned}
a_t &= W_{hx}^T x_t + W_{hh}^T h_{t-1} \\
h_t &= \tanh(a_t) \\
s_t &= W_{yh}^T h_t \\
\hat{y}_t &= \text{softmax}(s_t)
\end{aligned}
$$

W_{hx} is the input-to-hidden weight, W_{hh} the hidden-to-hidden weight, W_{yh} the hidden-to-output weight. a_t, h_t are the activation and hidden state at each step. s_t is the probability scores for the next step. The output activation by the softmax function (2.19) normalises the scores into mutually exclusive class probabilities $\hat{y}_t$. At each step the multi-class *cross entropy* (CE) loss is the negative log-likelihood of the sample. The overall loss sums up the stepwise losses:

$$
\begin{aligned}
L\left(x_1, ..., x_\tau, y_1, ..., y_\tau\right) &= \sum_t L_t \\
&= -\sum_t \log p(y_t | x_1, ..., x_\tau)
\end{aligned} \tag{2.13}
$$

Besides, regression problems use linear output activation function and the squared-error loss:

$$
L\left(x_1, ..., x_\tau, y_1, ..., y_\tau\right) = \sum_t \frac{1}{2}(\hat{y}_t - y_t)^2
$$

Binary classification uses logistic sigmoid as the output activation function and the binary cross entropy loss:

$$
L\left(x_1, ..., x_\tau, y_1, ..., y_\tau\right) = -\sum_t \{y_t \log \hat{y}_t + (1 - y_t) \log\left(1 - \hat{y}_t\right)\}
$$

The back propagation in RNNs is known as *back propagation through time* (BPTT). The computation is not trivial. Details will be provided in the context of the NLP task in section 4.1.1.

2.2.2 Natural language processing

It is generally acknowledged that RNNs often outperform static networks (e.g. FNNs) in tasks which involve representing context. Learning from sequential data is one such task, whereby the model is required to process input sequences and make predictions. Typical examples of sequential data are time series data, text data, and speech data. In this thesis we will focus on the NLP task of text sequence classification where, given a sample of sentences, we are required to classify for a binary or multi-class variable associated with the text sequence. The main dataset used is the Toxic Comment Data prepared by Google Jigsaw. Each publicly available comment sequence is associated with a few binary 'toxic indices', indicating whether the sentence can be considered 'toxic', such as 'toxic', 'obscene', 'threat' and 'insult'. We will only keep 'toxic' as the classification target. Such classification task is known as *sentiment classification*.

In sequence classification we consider an input as a sequence of words. It is not possible for the neural network model to interpret and learn from text. Therefore words are first represented by numerical vectors known as *embedding* vectors, using methods such as latent semantic analysis (Dumais et al, 1988), and Word2vec (Mikolov et al, 2013).

Processing text data

Before the text sequences can be fed into the network, they will go through 2 preprocessing steps: tokenisation and embedding matching.

Tokenisation involves amassing a word dictionary containing words from the training and the test data, each word assigned with a unique index. Then a text sequence has every word converted to the corresponding index in the dictionary. Each sequence is then padded with zeros to match the length of the longest raw text sequence (say it has length τ) in the sample.

For example the first text sequence in the Toxic Comment Data contains 156 words. The longest text sequence contains $\tau = 1403$ words. In the corresponding padded converted sequence we did find 1247 zeros being padded. In addition, there are 19 words indexed 1, corresponding to the word 'the'. Therefore the first raw text sequence contains 19 'the'.

These padded converted sequences are stored in a matrix where each row is a length τ numerical vector. While these raw embedding vectors may be used directly as inputs to the RNN, practitioners will often used pre-trained embeddings to avoid learning the words from scratch. One reason for using pre-trained embeddings is that the word dictionary constructed from tokenisation assigns indices based on frequency of occurrence. In practice we also require the numerical representation of text data to map the semantic meaning into geometric space (Chollet Keras blog, 2016). For words semantically related, e.g. 'kitchen' and 'pan', reasonably their numerical representations need to be closer together. Pre-trained embeddings are trained from an extremely large vocabulary source, at the same time taking into account of the cooccurrence circumstances of different words. Therefore pre-trained embeddings have rare words better represented than raw embeddings, and have semantical relationship of words better reflected. We will use the GloVe embeddings of dimension 100 (Pennington et al, 2014), which are pre-trained from English Wikipedia vocabulary. Each word is represented with a length-100 embedding vector. These embeddings are stored row-wise as a matrix of K rows, where K refers to the number of words in the word dictionary amassed in tokenisation.

2.2.3 RNN as an NLP model

The padded, tokenised input matrix has shape (batch size (N), input length (τ)). Batch size is the number of training text sequences in the raw data. Input length corresponds the length of the padded sequences. For $i = 1, ..., N$, each input sequence $X_i = [x_1; ...; x_t; ...; x_\tau]$ is a row vector of length τ in the input matrix. At each time step $t = 1, ..., \tau$, let x_t denote a word in X_i. Each x_t is represented as a one-hot vector of length K: If x_t corresponds to the word of index

q in the vocabulary, then the q-th entry of this one-hot vector will be 1, and the rest being 0.

For text data, the model requires an *embedding layer* between the inputs and the RNN layer to transform the input text object before passing into the RNN layer. This is achieved in software by supplying the GloVe embedding matrix in the embedding layer:

```
Embedding(input_dim, output_dim, input_length= tau,
weights= [Embedding_matrix], trainable= False)
```

Input dimension is K, the length of each input one-hot indicator x_t, equivalently the size of the word dictionary. Output dimension in this case is 100, the length of each GloVe embedding vector. The RNN layer follows the embedding layer therefore output of the embedding layer is the input to the RNN layer. We use $n_i = 100$ to denote the input dimension of the RNN layer. Since they are pre-trained representations we choose not to further train the GloVe embeddings. The embedding matrix E has shape (input dim = K, output dim = 100).

Now we define the forward propagation step in a standard RNN. Note that the RNNs are introduced in their general forms. At each step a predicted output and the associated loss is produced. One can adopt a specific context of application where the models predict the next word after processing each word in sequence. Therefore each $\hat{y}_t$ is a vector of length K, where K refers to the size of the amassed word dictionary in tokenisation, i.e. number of possible classes for predicting a word.

The reason for stating the standard structure clearly is that in the experiments of sequence classification the models will adopt a slightly different structure, and will be stated clearly in

section 4.

$$e_t = E^T x_t \tag{2.14}$$

$$a_t = W_{hx}^T e_t + W_{hh}^T h_{t-1} \tag{2.15}$$

$$h_t = \text{tanh}(a_t) \tag{2.16}$$

$$s_t = W_{yh}^T h_t \tag{2.17}$$

$$\hat{y}_t = \text{softmax}(s_t) \tag{2.18}$$

At step t, one-hot vector x_t selects the corresponding embedding vector from E, and e_t is a column vector of length $n_i = 100$. Suppose the RNN layer has n_h units. It will produce an activation vector a_t involving both the information in the current input and the information (aka. memory) stored in the previous hidden state h_{t-1}. The activation is activated element-wise by the tanh-function to produce the current step hidden state h_t. Both vectors a_t, h_t have shape $(n_h, 1)$.

The input-to-hidden weight matrix W_{hx} has size (n_i, n_h) and the hidden-to-hidden weight matrix has size (n_h, n_h). The hidden-to-output weight matrix W_{yh} has size (n_h, K). s_t is the probability scores for the next word. The output activation by the *softmax* function normalises the scores into mutually exclusive class probabilities $\hat{y}_t$.

The softmax output activation function (abbrev. $S(\cdot)$) is defined below. We use $k = 1, ..., K$ as indices for classes, i.e., 2 arbitrary words in the vocabulary.

$$\hat{y}_t[k] = \frac{exp(s_t[k])}{\sum_{c=1}^{K} exp(s_t[c])} \tag{2.19}$$

At the end of a timestep an output $\hat{y}_t$ of size $(K, 1)$ is computed. It is the probability distribution of the next word at step $t + 1$. $\hat{y}_t[k]$ refers to the probability of the next word being the k-th word in the word dictionary.

At each timestep the cross-entropy loss L_t will be computed. Define $\hat{y}_t[k]$ as the probabil-

ity of the true word at step t being k, and the t-step loss. In the context of text sequence, each stepwise loss in the multi-class CE loss (2.13) becomes:

$$L_t = -\sum_{k=1}^{K} y_t[k] log(\hat{y}_t[k]) \tag{2.20}$$

$$L_t = -log(\hat{y}_t[c]) \tag{2.21}$$

The true response y_t is be a multivariate Bernoulli vector with 1 at the entry of the actual next word in the text sequence, and y_t is known for every t. For example it corresponds to the word indexed c. Then $y_t[c] = 1$, $y_t[k] = 0$ for all $k \neq c$. Therefore the stepwise loss can be simplified to (2.21).

In software an RNN layer consists of computations in (2.15, 2.16). For visualisation, the RNN layer can be thought of as a cell with some pre-determined number of units, stacked on top of the embedding layer. The input sequence has each step processed by the cell. This is known as *unrolling* an RNN, and is presented in Figure 2.2. RNN-unrolling exploits the dimension of timesteps, as though a copy of the RNN cell has been created for each step of the sequence.

After step τ, the overall CE loss is the sum of stepwise losses. The loss gradients with respect to the parameters are computed to update the parameters. The explicit form of these gradients will be covered in section 4.

2.2.4 Alternative RNN-architectures

While there exist other techniques to deal with the VGP such as Hessian-free optimisation and regularisation, this thesis will focus on the more advanced RNN-class models as solutions. We discuss and analyse the Long Short-Term Memory (LSTM), Gated Recurrent Units (GRU) (Cho et al, 2014) and bidirectional-structure.

(1). LSTM

Advanced RNN-class models have specially-designed cells wrapped in the standard RNN (SRNN) layer, to offer more operations and better performance, compared to the simple hidden states h_t in an RNN layer. The LSTM cell has been successful in practice and is easy to train since its model architecture is more complicated but similar to the SRNN's model architecture. In an RNN, the hidden states h_t behave like memory container, storing information of the previous steps of data. In an RNN layer characterised by LSTM cell, the hidden state at each time step is replaced with an LSTM cell illustrated below. In particular now the *cell states* c_t act as the memory holder, while the hidden states h_t are only responsible for computing outputs.

At each step t we have an LSTM cell as illustrated in Figure 2.3. Recall each input sequence x_t is a one-hot vector of shape $(K, 1)$. For consistency we still use the pretrained GloVe 100d embedding vectors. Therefore each e_t is the corresponding embedding vector of shape $(100, 1)$. Consequently x_t in the diagram is actually e_t.

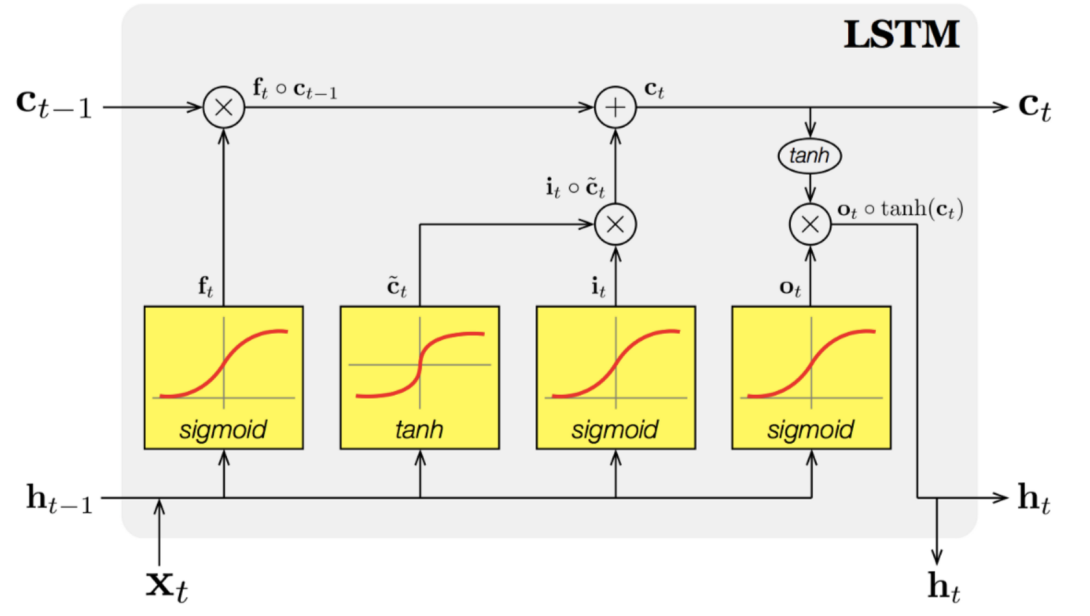

Figure 2.3: LSTM diagram. https://medium.datadriveninvestor.com/in-artificial-intelligence-new-types-of-networks-are-constantly-generated-by-scientists-a032cd7b77d9

The computations within are performed by 3 gate objects: the input gate i_t, forget gate f_t and output gate o_t. In addition at each step a candidate cell state $\tilde{c}_t$ is computed. The 4 components involve trainable recurrent weights $\{W_i, W_f, W_c, W_o\}$, and input weights $\{U_i, U_f, U_c, U_o\}$.

The LSTM forward propagation in detail:

$$i_t = \sigma(W_i^T h_{t-1} + U_i^T e_t) \tag{2.22}$$

$$f_t = \sigma(W_f^T h_{t-1} + U_f^T e_t) \tag{2.23}$$

$$\tilde{c}_t = \tanh(W_c^T h_{t-1} + U_c^T e_t) \tag{2.24}$$

$$o_t = \sigma(W_o^T h_{t-1} + U_o^T e_t) \tag{2.25}$$

The LSTM cell outputs the cell state and hidden state:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{2.26}$$

$$h_t = o_t \odot \tanh(c_t) \tag{2.27}$$

where $\odot$ is the Hadamard product of 2 vectors.

The *forget gate* f_t is a logistic sigmoid-activated vector based on the past hidden state and current input. Recall that outputs from logistic sigmoid are close to 0 and 1. Therefore the forget gate decides what information in the past cell state c_{t-1} to forget.

The *candidate cell state* $\tilde{c}_t$ is a tanh-activated vector containing the current-step information. It is exactly the hidden state in SRNN.

The *input gate* i_t is a logistic sigmoid-activated vector based on the past hidden state and current input. It selects the relevant information from the current step to include into the cell state, via the Hadamard product $i_t \odot \tilde{c}_t$. The current cell state c_t is the sum of the past memory filtered by the forget gate and the current information filtered by the input gate.

The current hidden state h_t can be thought of as a filtered scaled memory vector, as we use the *output gate* to select relevant information from the tanh-scaled current memory c_t.

Suppose at each step the LSTM cell has n_h units, then each recurrent weight W has shape (n_h, n_h), each input weight U has shape $(100, n_h)$. All 6 vectors (2.22) - (2.27) above have shape $(n_h, 1)$. The input vector has shape $(n_i, 1)$

The 3 gates have similar form, except that each is defined with weights that are independently updated, thus a different subscript. The sigmoid provides a keep-omit operation to the object that the gate is applied to. The candidate cell state is similarly defined but it is the combined and scaled information from the past hidden state and current input. As will be mentioned in section 3, the hyperbolic-tangent function is subject to VGP. One major reason that tanh-function is still commonly used in RNNs, rather than being replaced with ReLU-class functions is that the outputs of the tanh-function has mean zero, and range [-1,1]. The outputs at each step is therefore standardised to [-1,1]. Using ReLU may cause the outputs to have large fluctuations and provide difficulties in gradient-descent optimisation. The back propagation details of the LSTM will be covered in section 4.3.1.

(2). GRU

The gated recurrent unit (GRU, Cho et al 2014) is another architectural modification to SRNN as a solution to the VGP, by replacing the hidden state with a different cell parameterised by learnable gates. It is often found that GRU offers compatible predictive power as the LSTM.

A GRU cell contains 2 gates: the *update gate* (z_t) which filters both the past memory and the current-step memory; and the *reset gate* (r_t) which determines how much of the past information is to be forgotten. Again both gates are logistic sigmoid-activated vectors with outputs close to 0 and 1.

Let x_t be the $(K, 1)$ input text sequence, E, e_t be the embedding weight matrix and embedding vector associated with x_t. e_t has shape $(n_i, 1)$.

h_t is the hidden state at step t of shape $(n_h, 1)$. Let U_z, U_r be the input-to-hidden weight matrices in update and reset gates respecptively, both having shape (n_i, n_h). W_z, W_r are the hidden-to-hidden weight matrices in update and reset gates respectively, both having shape

(n_h, n_h). The forward propagation equations in a GRU cell:

$$z_t = \sigma(W_z^T h_{t-1} + U_z^T e_t) \tag{2.28}$$

$$r_t = \sigma(W_r^T h_{t-1} + U_r^T e_t) \tag{2.29}$$

$$\tilde{h}_t = \text{tanh } (W_h^T (r_t \odot h_{t-1}) + U_h^T e_t) \tag{2.30}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{2.31}$$

$$\hat{y}_t = \text{softmax}(W_{yh}^T h_t) \tag{2.32}$$

The candidate hidden state $\tilde{h}_t$ can be thought of as the current memory increment which contains information from the current input and the reset gate-filtered past information in h_{t-1}. Therefore it has the similar effect as the candidate cell state $\tilde{c}_t$ and hidden state h_t combined in LSTM. The GRU hidden state h_t is the memory holder itself as it filters both the past memory state h_{t-1} and the current-step input. It is often claimed that GRU is more efficient than LSTM. Indeed a GRU cell contains 1 fewer gate and 1 fewer state as compared to an LSTM cell, thus saving a considerable amount of computation.

2.3 Past works

Deep learning researchers have been working on efficient algorithms for neural network optimisation since the 1990s. In the past 20 years researchers have realised the impact of model parameterisation in FNN and RNN training. Since then studies on the effect of the VGP on model performance have been conducted with considerable achievements. Glorot & Bengio (2010) discussed the impacts of using saturating activation functions in FNNs and additionally introduced a carefully designed initial distribution for the weight parameter entries. He et al (2015) and Clevert et al (2016) went further on designing better-performing techniques for FNN parameterisation. Pascanu et al (2012 & 2013) have discussed the difficulties in RNN training due to the VGP. Graves (2013) proposed the LSTM cells (Hochreiter & Schmidhuber, 1997) as a solution. The GRU cells proposed by Cho et al (2014) are a similar model.

Other known solutions include the Hessian-free optimisation applied into the standard RNN (Sutskever et al, 2011); Echo State Network (Jaeger & Haas, 2004) and regularisation (Pascanu et al, 2013). Regularisation in particular is a common technique in neural network training, which aims more than reducing overfitting. For example the regularisation term proposed by Pascanu et al (2013) forces each RNN Jacobian $\frac{\partial h_t}{\partial h_{t-1}}$ to preserve norm. As will be shown in section 4, the behaviour of this Jacobian is the key aspect to the VGP in RNNs. This thesis will focus on alternative parameterisations for the VGP in FNNs and modified RNN architectures for the VGP in RNNs, but will not cover regularisation. Nonetheless in section 4 and 5 we will analyse the Jacobians in each RNN-class model. Chapter 7 of Goodfellow et al (2016) provides a holistic summary on deep learning regularisation.

The Hessian-free approach involves the 2nd-order approximation of the optimisation objective function plus an extra damping (penalty) term. Pascanu et al (2013) commented that this approach is missing detailed analysis. Due to efficiency Hessian-free optimisation is not implemented in common software libraries. For these reasons the thesis will not cover this approach.

3 The VGP in Feedforward Neural Networks

In section 3, the sources and solutions of the VGP in training FNNs are analysed. Diagnostics in software are devised to visualise the degree of the VGP.

3.1 The Problem in FNNs

A close inspection of equation (2.11) will reveal how the VGP arises in the back propagation step when training an FNN model. Note that each $\frac{\partial h}{\partial a}$ with h, a in the same layer gives the weight; each $\frac{\partial a}{\partial h}$ with a, h over 2 consecutive layers gives the derivative of the activation function.

Firstly, when the largest eigenvalue of a matrix is less than 1, raising powers of the matrix will cause the entries to vanish. Therefore more weights having such a property means a higher tendency for the entries of the product matrix to vanish to zero. Consequently the loss gradient vanishes to zero.

Secondly, the long product of the activation function derivatives in (2.11) will also shrink to zero if *saturating activation functions* are used, for example the Sigmoid-class functions including the logistic sigmoid and the tanh-functions. The Sigmoid-class functions bound the entire domain (the whole real line) into a small interval, e.g. [0,1] is the range of logistic sigmoid. Therefore they typically tend to *saturate*. An activation function saturates if the input sits in the flat regions where the output is near 0 or 1. These regions are where the derivative is close to zero. One example is the logistic sigmoid function (2.4) that has very small non-flat regions for inputs near zero.

When the loss gradients become zero, the SGD will not update the parameters when new inputs

are fed into the network. This results in information loss associated the new inputs, and thus high training loss. Once an activation function saturates, it is difficult for it to move out of saturation (Calin, 2020). In equation (2.7) we see that the activations a are affine transformation of the weights W. When the VGP occurs due to saturation, the updating of the weights will slow down significantly, or stop completely. Then with these same values of the weights, the activations in the hidden layers will get stuck in the flat regions of the activation functions. Therefore the hidden layers will remain saturated.

3.2 The Solutions in FNNs

3.2.1 ReLU-class activation functions

As mentioned earlier, the Sigmoid-class functions saturate and contribute to the VGP. One common solution for this is instead using the ReLU-class functions including the standard ReLU function (2.3) and its variants. All these functions have their range up to positive infinity, and therefore non-zero derivatives for some or all inputs.

Apart from alleviating saturation from using the sigmoid-class activation functions, the ReLU function is the maximum function, i.e. merely a comparison between the input and 0. Therefore the computation efficiency is improved. Chou et al (2019) observed that a Convolution Neural Network (CNN) parameterised with ReLU converges six times faster than one parameterised with tanh.

There are nonetheless some issues with using the standard ReLU function in the hidden layers. The first issue is known as the *Dying-ReLU problem*. Given a randomly initialised input, each ReLU unit has 50% probability of being inactive, i.e. negative input resulting in zero output. Due to the stochasticity of gradient descent, there will be occasions where the majority of the hidden layer outputs are zero, in which case the signals may die-off in the forward propagation.

The loss and the loss gradients with respect to the parameters will then be zero. Illustrations of the problem with diagnostic statistics are provided in Appendix section A.1.1.

The second issue with ReLU is that it is discontinued at zero where the gradient is undefined. The ReLU function defined in (2.3) has left limit of gradient at zero being zero, right limit being 1. It is pointed out by Goodfellow et al (2016) that this issue is not as problematic as it seems. In software computation, it is very unlikely for the algorithm to take a step at exactly zero (i.e. unlikely to evaluate $\text{ReLU}(0)$) due to machine precision. Zero machine outputs we observe are usually small numbers truncated to numerical zero. However in forward propagation the inputs to ReLU are not truncated before the outputs are produced. In the case where an input from the dataset is exactly zero, it is conventional for sotfware to set $\text{ReLU}'(0)$ as 1, i.e. the right limit.

The last issue of using the ReLU-class activation functions concerns the efficiency of gradient descent optimisation, that they do not have the *zero-centred* property. A function with its mean output being zero is a zero-centred function, and clearly ReLU is not. It is commonly acknowledged in machine learning tasks that normalisation of the input is essential for accelerating the training and improving the generalisation of neural networks (Huang et al, 2020). A zero-centred activation function provides a similar effect as normalisation for outputs from each hidden layer. When ReLU is used, the activations are always non-negative and unbounded. Thus in equation (2.12) the sign of the loss gradient with respect to the weight is always the same as that of the gradient with respect to the activation (δ in result 2.5), thereby reducing the optimisation efficiency in updating the weights.
The *batch normalisation* technique (Ioffe & Szegedy, 2015) is one effort to reduce the impact of this issue. When constructing models in practice, one would typically add a batch normalisation layer after one hidden layer parameterised by a non zero-centred activation function.

There have been variants of ReLU functions designed by researchers for better performance over time. Leaky ReLU (Maas et al, 2013) is similar to the standard ReLU but outputs ax for

negative x, where $a > 0$ is a small constant multiplier, for example 0.01. Negative inputs now have outputs of small magnitude, thus named 'leaky'.

Parametric ReLU (He et al, 2015) is similar to the leaky ReLU but the constant multiplier a is now trainable, with the form given by:

$$f(x) = \begin{cases} ax & \text{for } x \leq 0 \\ x & \text{otherwise.} \end{cases} \tag{3.1}$$

$$f'(x) = \begin{cases} a & \text{for } x \leq 0 \\ 1 & \text{otherwise.} \end{cases} \tag{3.2}$$

Exponential linear units (ELU) (Clevert et al, 2016) speeds up learning and leads to higher classification accuracies, and has the form:

$$f(x) = \begin{cases} a(\exp(x) - 1) & \text{for } x \leq 0 \\ x & \text{otherwise} \end{cases}$$

$$f'(x) = \begin{cases} f(x) + a & \text{for } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Multiple Parametric Exponential Linear Unit (MPELU) (Li et al, 2017) is a generalisation to ELU, which has the form:

$$\begin{aligned} \text{MPELU} &= \text{ELU}(\text{ PReLU}(x)) \\ &= \begin{cases} a(\exp(\beta x) - 1) & \text{for } x \leq 0 \\ x & \text{otherwise} \end{cases} \end{aligned}$$

The ELU function allows negative outputs and therefore offers more flexibility in weights-updating than ReLU. It is worth noting that ELU function saturates to $-a$ for large negative inputs in order to 'learn a more robust and stable representation'. From the shape of the ELU curve in Figure 3.1 we can deduce that the comment from Clevert et al. (2016) suggests a

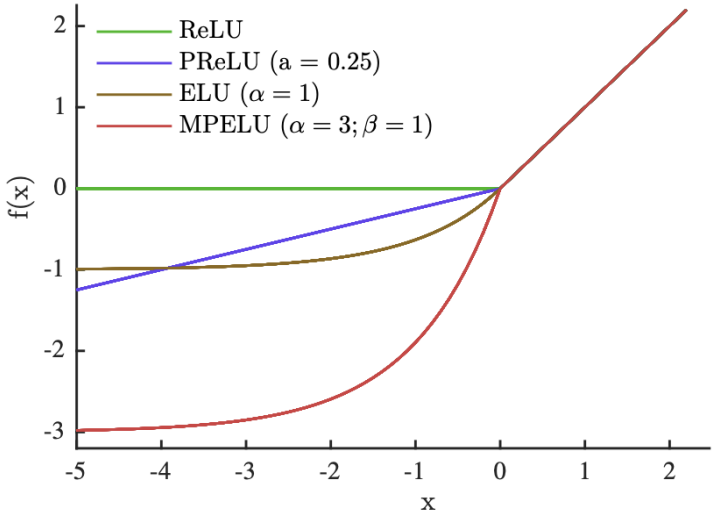

Figure 3.1: A plot from Li et al (2017) containing all the above ReLU-class variants

compromise between being bounded therefore stability in output for negative inputs, and being non-saturating for positive inputs. The parametric ReLU will be used in this thesis for the sake of demonstrating (avoiding) the VGP. However practitioners should perform a hyperparameter optimisation to find out the activation function leading to the optimal loss and accuracy on the particular data fitted on.

3.2.2 Weights initialisation

Apart from the saturation of activation functions, the other source of the VGP pointed out in section 3.1 is the multiplication of weight matrices over the layers. Typically the entries of weight matrix in each hidden layer are initialised from random uniform or normal distribution with mean zero, for example uniform $[-a, a]$ random variables. However the loss gradient (2.11) will suffer from large variance in magnitude if the largest singular value of each weight matrix is far from 1, i.e. potentially vanishing and exploding gradients.

Glorot Uniform Initialisation (Glorot & Bengio, 2010) is one of the widely-applied initialisation schemes that maintain the variance in forward and back propagation. The endpoints of

the uniform distribution are derived based on the motivation that the variances of the forward-propagated signals and the back-propagated gradients do not vanish or explode when processed across the layers. We will present the full derivation with motivation and extra details.

Motivation

As pointed out by Calin (2020), the stability of back-propagated loss gradients can be quantised by the variance of forward-propagated signals. He inspected the variance of the hidden layer outputs, at the same time taking the weights and bias to be deterministic, in order to monitor the behaviour of the variance under small and large values of weights.

Let $a^{(l)}, h^{(l)}, b^{(l)}, W^{(l)}$ be respectively the activation and hidden state vector, bias vector and weight matrix on the l-th layer. The scalar entries within each vector or matrix are assumed to be independently and identically distributed. Recall the forward propagation step:

$$h_j^{(l)} = f(W_{j,i}^{(l)} h_i^{(l-1)} + b_j^{(l)})$$

Suppose a random variable X has mean $\mathbb{E}(X)$. Approximating $f(X)$ at the mean of X by its Taylor expansion gives

$$f(X) \approx f(\mathbb{E}(X)) + f'(\mathbb{E}(X))(X - \mathbb{E}(X))$$

Assume a zero-centred activation function is used for each hidden layer, therefore $\mathbb{E}(X) = 0$. Therefore $V(f(X)) \approx f'(\mathbb{E}(X))^2 V(X)$. Now X is $W_{j,i}^{(l)} h_i^{(l-1)} + b_j^{(l)}$, $f(X)$ is $h_j^{(l)}$. Assume the entries of weights are constant:

$$V(h_j^{(l)}) \approx f'(\sum_i w_{j,i}^{(l)} \mathbb{E}(h_i^{(l-1)}) + b_j^{(l)})^2 \{\sum_i (w_{j,i}^{(l)})^2 V(h_i^{(l-1)})\}$$

The above expression differs from result (6.3.35) in Calin (2020) as we changed the rightmost $w_{j,i}^{(l)}$ into $(w_{j,i}^{(l)})^2$. Under the variance operation, we believe the constant multiplier has to be

squared. By Cauchy's inequality,

$$\sum_i (w_{j,i}^{(l)})^2 V(h_i^{(l-1)}) \leq \left(\sum_i (w_{j,i}^{(l)})^4 \right)^{1/2} \left(\sum_i V(h_i^{(l-1)})^2 \right)^{1/2}$$

Sigmoid and ReLU classes both have bounded derivatives. Therefore assume $f'(\cdot) < C$. Then express the sum of squares of variance with the above 2 results:

$$\sum_j V(h_j^{(l)})^2 < C^2 \sum_{j,i} (w_{j,i}^{(l)})^4 \sum_i V(h_i^{(l-1)})^2 \tag{3.3}$$

For small weights, the upper bound at each layer in (3.3) gets smaller, and the sum of squares of the variances decreases to zero as the signals are propagated through the layers. The hidden layer outputs stagnate. It is worth noting that this is not a result of convergence to a minimum loss in the gradient descent optimisation.

For large weights, if ReLU-class activation functions with slope parameter $a \geq 0$ are used,

$$f'(\cdot)^2 := a^2$$
$$V(h_j^{(l)}) = a^2 \sum_i (w_{j,i}^{(l)})^2 V(h_i^{(l-1)})$$

Large weights increase the signal variance and lead to instability in the outputs, thereby reducing the efficiency of optimisation.

If sigmoid-class activation functions are used, large weights lead to large values of $\sum_i w_{j,i}^{(l)} h_i^{(l-1)}$. Large inputs to sigmoid-class activation functions will make them saturate and we will have the VGP.

To point out the main idea, multiplication of weight matrices over layers in (2.11) is one way of describing the consequence of improper weights initialisation, that focuses on the actual value of the entries. Result (3.3) is another way of describing the consequence, that is focused on the

signal variance $V(h)$ over consecutive layers. With regards to the VGP, one can relate the 2 arguments by seeing that: the vanishing of the loss gradients exactly halts the weight parameter updating. Therefore the hidden layer outputs stagnate.

Glorot initialisation

Glorot & Bengio (2010) proposed to derive the initial distribution of weights under the assumption that the variances of the forward-propagated signals and back-propagated gradients are respectively invariant across the hidden layers. The assumption can be justified by the explanation regarding the upper bound of the sum of squares of the variance in (3.3).

We found some parts of section 4.2 of the paper, where the main results of the initial distribution are derived, not immediately straightforward.

1. The argument from equation (2) to equation (6) in the paper does not seem to be clear; in particular how the layer index $i + 1$ on W in equation (2) becomes i in equation (6).

2. The definition of layer size n_i is not specified precisely. Similarly the 'shared scalar variance' of all weights on layer i, $Var(W^i)$ is assumed without discussion or justification.

Given that this paper is one of the foundational analyses on the FNN model training, we have provided detailed explanation in order to assist the understanding of the authors' derivation.

The authors have defined forward propagation in a slightly non-standard manner. In addition they assumed a linear activation function f. We will temporarily adopt their notation in this section.

$$s^i = W^i z^i \tag{3.4}$$

$$z^{i+1} = f(s^i) = s^i \tag{3.5}$$

According to this definition, the internal structure of the FNN is shown in Figure 3.2 below for 2 adjacent layers i and $i + 1$.

(1) Clarification of notation.

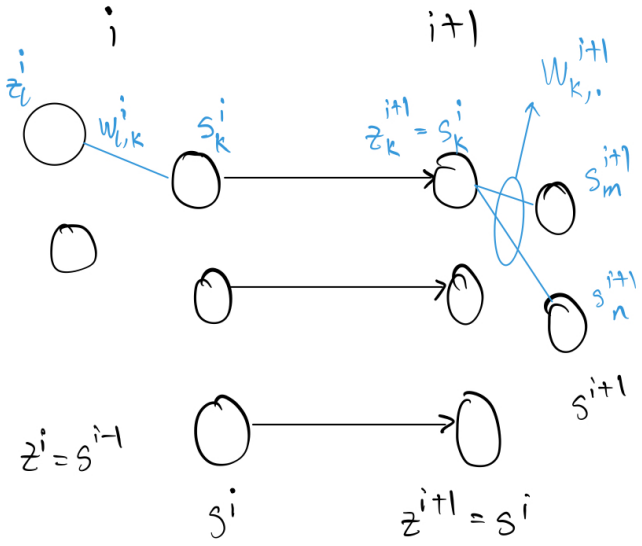

Figure 3.2: 2 arbitrary layers of FNN defined in Glorot & Bengio (2010)

Define n_i to be the length of the vector z^i. Therefore n_{i+1} is the size of s^i. This is why in Glorot & Bengio (2010), the layer size involved in forward-propagated variance is $n_{i'}$ in equation (5), and that involved in back-propagated variance is $n_{i'+1}$ in equation (6). Note that W^i therefore has shape (n_{i+1}, n_i). To avoid confusing $V(W^i)$ with the variance-covariance matrix, we define $V(w^i)$ as the shared scalar variance of all entries in the weight matrix at layer i, using the lower-case of W. $V(w^i)$ is equivalently $V(W_{k,l}^i)$ for any unit l in z^i, and any unit k in s^i.

Assumptions of distribution:

Glorot and Bengio (2010) implicitly assumed that the weights entries $w_{j,i}$ are independently distributed. The input features have the same variances. Calin (2020) went further by claim-

ing on each layer, $w_{j,i}$ are all independently and identically distributed with zero means. We also assume the entries in any per-layer input vector z^l to be independently and identically distributed.

(2) Write out the variances of forward-propagated signals z.

$$\begin{aligned} z^i &= s^{i-1} = W^{i-1} z^{i-1} = W^{i-1} W^{i-2} z^{i-2} = \cdots \\ &= W^{i-1} \cdots W^0 z^0 \\ &= \left(\prod_{i'=0}^{i-1} W^{i'} \right) x \end{aligned} \tag{3.6}$$

where signal on the zeroth (input) layer is the input x. Take variance of each independently distributed unit and we will arrive at equation (5):

$$V(z_\bullet^i) = \left(\prod_{i'=0}^{i-1} n_{i'} V(w^i) \right) V(x_\bullet) \tag{3.7}$$

Earlier in the thesis, letter i is defined as the index for a unit (a vector entry). The authors instead defined the layer index with i. Therefore to avoid confusion we use the subscript disc to denote a scalar unit in z^i above.

It is worth clarifying how the variance operation is performed with an example in the first layer. In particular how the multiplier $n_{i'}$ comes about in the product:

$$\begin{aligned} z^1 &= W^0 z^0 \\ V(z_j^1) &= \sum_i V(W_{ji}^0) V(z_i^0) = n_0 V(w^0) V(z_i^0) \end{aligned}$$

for any unit i in layer 0, and any unit j in layer 1. The multiplier n_0 refers to the summation over i. There are in total n_0 components in z^0.

As the definition of n_i is not clearly stated in Glorot & Bengio (2010), for the sake of completeness we also consider the possibility that n_i refers to the size of $s^i = z^{i+1}$. In this case n_{i-1} is the size of $s^{i-1} = z^i$. However just by looking at the first layer variance expression we have found that this version of definition is not possible, as we now see:

$$s^0 = W^0 z^0$$
$$z^1 = s^0 = W^0 z^0$$
$$V(z_j^1) = n_{0-1} V(w^0) V(z_i^0)$$

In particular, n_{-1} is undefined. Therefore we will carry on with the definition that n_i is the size of vector z^i.

(3) Write out the variances of the back-propagated gradients $\frac{\partial C}{\partial s}$.
For back propagation, it is simpler if we start from the last layer d, where we have:

$$s^d = W^d s^{d-1}$$
$$y = s^d$$

In particular s^d has shape $(n_{d+1}, 1)$, W^d has shape (n_{d+1}, n_d). Let p be an arbitrary index for vector component in the $(d-1)$-st layer, q an arbitrary index in layer d. The gradient of the cost function (eq. the loss) has the form:

$$\frac{\partial C}{\partial s_p^{d-1}} = \sum_q W_{q,p}^d \frac{\partial C}{\partial s_q^d} \tag{3.8}$$

This is equation (2) in the paper. Take variance with respect to the summation over index q, i.e. over the n_{d+1}-many units in layer d, we have multiplier n_{d+1}:

$$\begin{aligned} V(\frac{\partial C}{\partial s_p^{d-1}}) &= \sum_q V(W_{q,p}^d) V\left(\frac{\partial C}{\partial s_q^d}\right) \\ &= n_{d+1} V(w^d) V\left(\frac{\partial C}{\partial s_q^d}\right) \end{aligned} \tag{3.9}$$

On the previous layer:

$$V\left(\frac{\partial C}{\partial s_r^{d-2}}\right) = n_d V(w^{d-1}) n_{d+1} V(w^d) V\left(\frac{\partial C}{\partial s_q^d}\right)$$

Back-substitute until the i-th layer:

$$V\left(\frac{\partial C}{\partial s_k^i}\right) = \left(\prod_{i'=i+1}^{d} n_{i'+1} V(w^{i'})\right) V\left(\frac{\partial C}{\partial s_q^d}\right) \tag{3.10}$$

Result (3.10) differs from the authors' equation (6) in that the layer index in the product starts from $i+1$ instead of i. A justification for $i+1$ is that s^{i+1} is affinely-transformed from s^i with W^{i+1}: $s^{i+1} = z^{i+1} W^{i+1} = s^i W^{i+1}$.

In back propagation from layer $i+1$ to i, $s^{i+1} = z^{i+2}$ has n_{i+2} units:

$$V\left(\frac{\partial C}{\partial s^i}\right) = V\left(\frac{\partial C}{\partial s^{i+1}}\right) n_{i+2} V(w^{i+1}) \tag{3.11}$$

Therefore when back-propagating from layer d to layer i, it seems more appropriate for the product of weight variances to start at $V(w^{i+1})$, rather than at $V(w^i)$. Nonetheless it can be shown that both result (3.10) and equation (6) will lead to constraint (11) in the paper. Consequently the derivation of the Glorot Uniform Initial Distribution is unaffected.

(4) Apply the conditions on variances to produce 2 constraints:

We require the variance of the signals z to be invariant across layers in froward propagation.

Then for any 2 arbitrary layers i, i',

$$\begin{aligned} V(z_\bullet^i) &= V(z_\bullet^{i'}) \\ \left(\prod_{l=0}^{i-1} n_l V(w^l)\right) V(x_\bullet) &= \left(\prod_{l=0}^{i'-1} n_l V(w^l)\right) V(x_\bullet) \\ n_l V(w^l) &= 1 \end{aligned} \tag{3.12}$$

It is further required that the variance of the gradients $\frac{\partial C}{\partial s}$ to be invariant in back propagation:

$$\begin{aligned} V\left(\frac{\partial C}{\partial s_\bullet^i}\right) &= V\left(\frac{\partial C}{\partial s_\bullet^{i'}}\right) \\ \left(\prod_{l=i+1}^{d} n_{l+1} V(w^l)\right) V\left(\frac{\partial C}{\partial s_\bullet^d}\right) &= \left(\prod_{l=i'+1}^{d} n_{l+1} V(w^l)\right) V\left(\frac{\partial C}{\partial s_\bullet^d}\right) \\ n_{l+1} V(w^l) &= 1 \end{aligned} \tag{3.13}$$

Results (3.12) and (3.13) are constraints (10) and (11) in Glorot & Bengio (2010) on variance of weight matrix entries in an arbitrary layer. They have proposed a compromise:

$$V(w^i) = \frac{2}{n_i + n_{i+1}} \tag{3.14}$$

which is a value between $\frac{1}{n_i}$ and $\frac{1}{n_{i+1}}$. Recall that uniform distribution $\mathbb{U}_{[-a,a]}$ has variance $\frac{a^2}{3}$. Therefore the endpoint $a = \sqrt{\frac{6}{n_i+n_{i+1}}}$. This is the result in equation (16) from the paper. In practice setting Glorot Uniform Initialisation on a hidden layer will initialise all scalar entries in the weight of that layer, by sampling from this uniform distribution.

Other initialisations

Here we briefly present both some naive and more recent weights initialisations available.

1. Zero: Initialises all entries to 0.

2. Uniform $\mathbb{U}(-a, a)$ and normal $\mathbb{N}(\mu, \sigma^2)$.

3. Lecun uniform (Lecun et al, 1998)

$$W_{j,i}^{(l)} \sim \mathbb{U}\left(-\sqrt{\frac{3}{n_i}}, \sqrt{\frac{3}{n_i}}\right) \tag{3.15}$$

4. He normal and He uniform (He et al, 2015)

$$W_{j,i}^{(l)} \sim \mathbb{N}\left(0, \sqrt{\frac{2}{n_i}}\right), \tag{3.16}$$

$$W_{j,i}^{(l)} \sim \mathbb{U}\left(-\sqrt{\frac{6}{n_i}}, \sqrt{\frac{6}{n_i}}\right), \tag{3.17}$$

where i stands for the index of a component from the $(l-1)$-st layer, n_i stands for the number of hidden components on the $(l-1)$-st layer, and j stands for the index of a component from the l-th layer.

3.3 Diagnosing the Problem in FNNs

In this subsection we will demonstrate fitting FNNs consisting of 3 hidden layers, which we have found to be sufficiently deep to exhibit trends of the VGP. Both simulated concentric circle data and the Pima diabetes dataset (Smith et al, 1988) have been used to illustrate VGP in FNN and the effects of the solutions. The circle data created by `sklearn.datasets.make_circles` function contains a 2 dimensional feature X and a binary target y. The Pima dataset contains binary Outcome (y) and 8 features in X. Results of the simulated data are not much different from results of the Pima data. Therefore they are not included to avoid redundancy.
The thesis uses the `tensorflow.keras` library to construct and fit neural network models in IPython notebooks.

After fitting a model, it is crucial to check for signs of the VGP. We have written code for

3 diagnostic experiments based on the theory how the VGP arises. The diagnostics are based on extracting and visualising training statistics and internal states:

- **A. Loss/output gradients** Are the loss gradients with respect to the hidden layer weights close to zero? For the sake of convenience the model output gradient is used as a proxy.
- **B. Improvement in learning** Loss may stop decreasing after a few epochs due to the VGP. Accuracy in training may also stop improving.
- **C. Hidden layer outputs** Look for hints of saturation, whether large proportion of outputs are near 0 and 1.

A. Output gradients with respect to weight parameters on each layer:
We first create a dictionary object to store the output gradients for each epoch of training. A standard Python function is defined to compute the per-epoch gradients. The `backend` package from `keras` library offers special wrapper functions to extract intermediate results. We used `backend.gradients` function to define a wrapper function that computes the gradient of model output with respect to the weight in each layer. Then the `backend.run` function execute the wrapper function with respect to the inputs from the training set. On each epoch the dictionary stores 4 items: the output gradients with respect to weights on the 3 hidden layers plus the output layer.
A `Callback` is an object in `Tensorflow` library for computing quantities during model-fitting. Practitioners usually write custom Callbacks by defining Python functions for the computation processes in the same way above. The overall Python function is embedded in the Callback object and we have to mention this Callback in the model-fitting command, by e.g.

```
model.fit( X, y, epochs = 150, callbacks = [ grads_callback ])
```

B. The training loss and accuracy can be obtained directly from the model-fitting command.

C. Hidden layer outputs:

Similar to A, a function for computing the per-layer outputs is defined for every epoch of training. The hidden layer output-computing function is embedded in a separate Callback to be added in the model-fitting command.

With these diagnostics, we can look for signs of the VGP from fitting a model, or compare the graphical results between models of the same structure but with different hyperparameters. In the experiments, each model is fitted for 150 epochs. One epoch stands for one complete processing of the input dataset. A closely related quantity is the batch size, which refers to the number of inputs taken into the network for one fit of the model. Typically for the sake of efficiency and faster convergence (Li et al, 2014), neural networks are trained with *mini-batch* gradient descent algorithm, where at each fit only a portion of the entire sample is used. Therefore it is appropriate for visualisation to plot various quantities against the number of epochs, rather than against the number of batches. In each epoch all inputs in the entire sample will contribute to the updating of model parameters.

To demonstrate the VGP with the Pima data, we fit an FNN consisting of 3 logistic sigmoid-parameterised hidden layers, each containing 8 units. This number of hidden units arises from a suggestion in Heaton (2008). In general the number of hidden units should stay below twice the input dimension, which is 8, the number of features. The optimal number of units in each layer is actually subject to hyperparameter tuning.

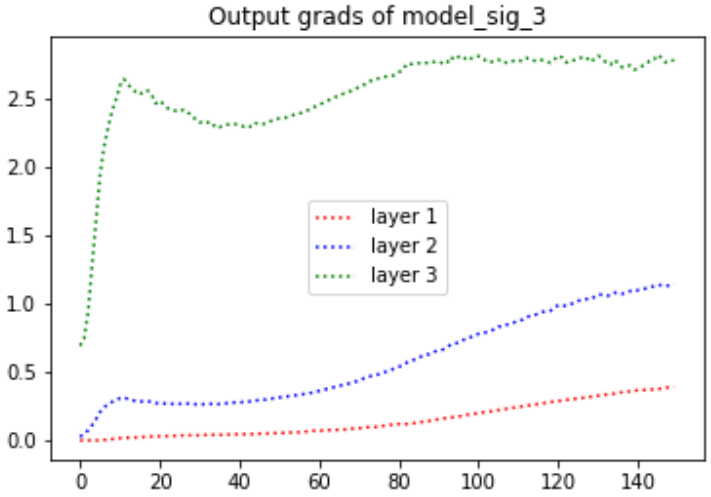

Figure 3.3: Sigmoid FNN: A. Output gradients

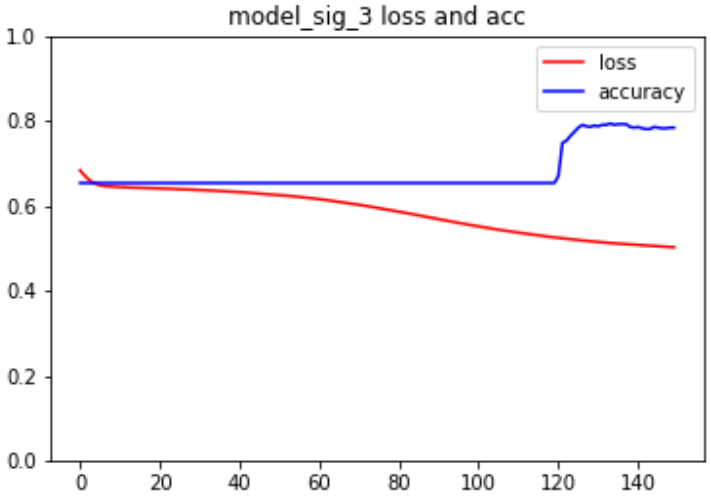

Figure 3.4: Sigmoid FNN: B. Training loss and accuracy

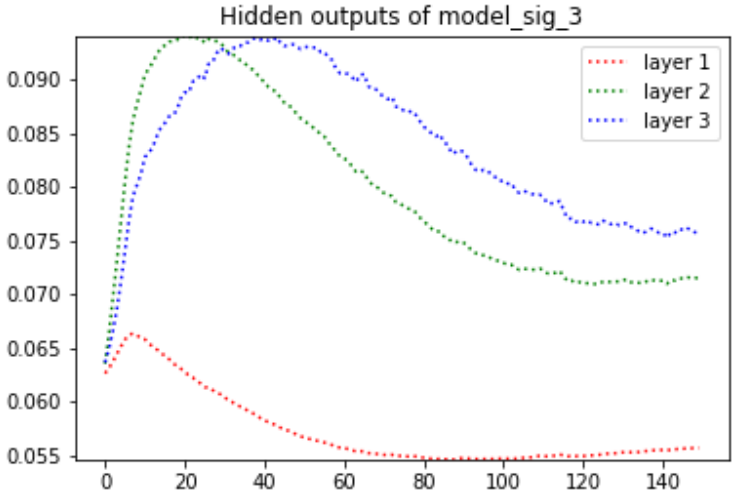

Figure 3.5: Sigmoid FNN: C. Aggregated hidden layer outputs

We can see that the output gradients are close to zero on the lower layers. This is because the gradient with respect to the weight on a lower layer will have a longer product in equation (2.11), thus stronger vanishing effect.

The training loss and accuracy have little improvement over the first 60 epochs. Models are usually trained for a much lower number of epochs due to computational expense.

The lineplot visualisation for the hidden layer outputs is used for all FNN models initially. Each output object stored is a matrix and each point in the plot is the mean absolute value of all entries in the matrix. Such an aggregation risks obscuring any sign of saturation in the hidden layers. Outputs close to 0 and 1 can not be reflected by their aggregated value. Therefore mean absolute value is not an appropriate aggregation for models parameterised with sigmoid-class activation functions. We suggest visualising the output samples from a few randomly selected epochs, via histograms as shown in Figure 3.6.

Now such a visualisation is more appropriate. We can see that on the lower layers large proportion of outputs are near 0 and 1, i.e. entered the state of saturation. On the 3rd layer, we do not have outputs saturated to near 0.

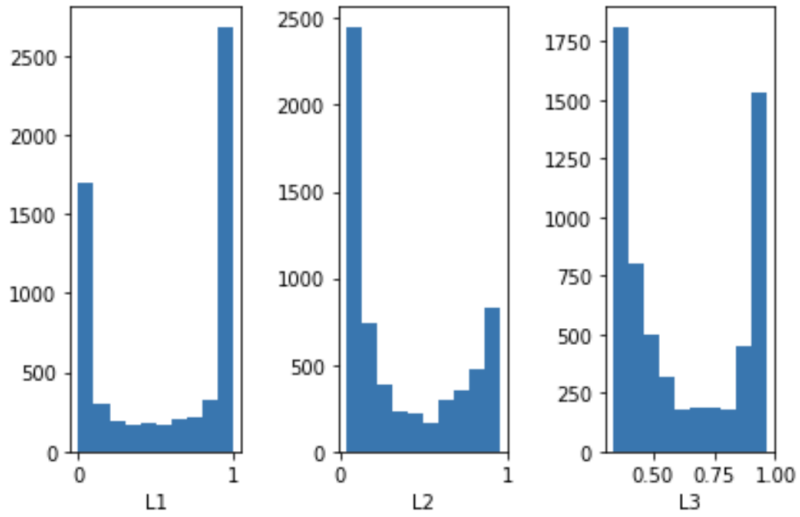

Figure 3.6: Sigmoid FNN: C. Last-epoch hidden layer outputs as histogram

3.4 Visualising the Effectiveness of Solutions

The FNN with 3 hidden layers constructed in section 3.3 is now parameterised with the PReLU activation function. The slope parameter a is set with the default value 0, but is in fact a trainable parameter which can avoid the Dying-ReLU problem mentioned in section 3.2.1. We have tuned the initial distribution of the weight in each of the 3 hidden layers, amongst a range of available choices. KerasTuner and HyperOpt are 2 commonly used software libraries for tuning neural network parameters. For this particular FNN, He-normal, random uniform, Lecun-uniform distributions are chosen based on lowest training loss. The 3 diagnostics are repeated to identify any improvement.

In Figure 3.7, the gradients have been significant enough in magnitude. There is a trend of exploding gradient on the lower layers, which could be solved by adding a Batch Normalisation layer after each hidden layer.

The training loss and accuracy in Figure 3.8 have improved significantly over the first 20

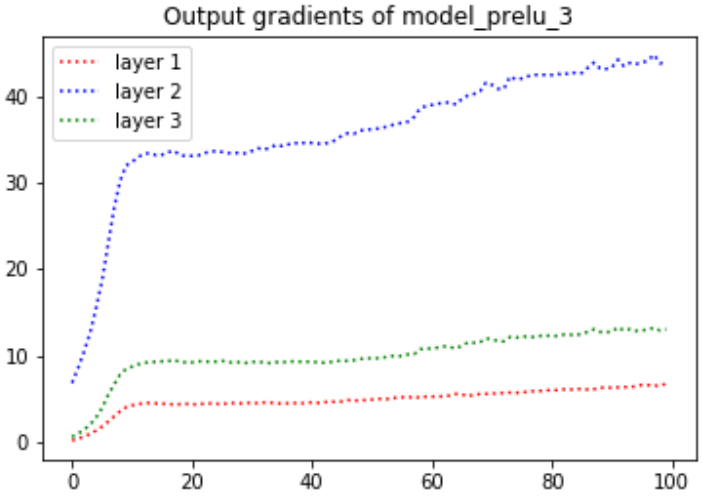

Figure 3.7: PReLU FNN with initialisation: A. output gradients

epochs, leading to a considerable improvement in optimisation efficiency.

Refer to the PReLU derivatives in (3.2), PReLU activation has no risk of saturation. For the sake of demonstration we have presented in Figure 3.9 the histogram of the hidden layer outputs at the last epoch. The first layer outputs have mean -33; the second layer has mean 17; the third layer has mean -4.

In fact, the lineplots for aggregated hidden layer outputs can be used to visualise the convergence behaviour of the weight parameters, when the model is parameterised with non-saturating activation functions. This is implemented and shown in Figure 3.10. The outputs for each hidden layer have been relatively stable after 20 epochs.

3.5 Conclusions

3.5.1 Sigmoid-parameterised FNNs

Sigmoid FNNs encounter the VGP by having the loss (output) gradients with respect to the weights close to zero for long periods of training. This is particularly significant for weights on

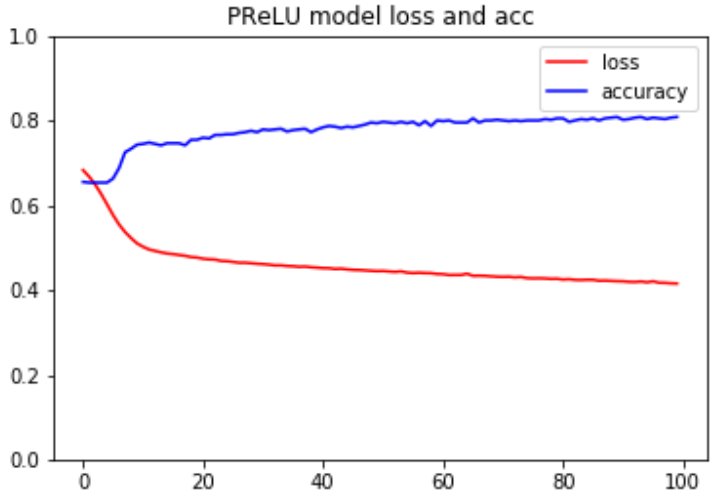

Figure 3.8: PReLU FNN with initialisation: B. Training loss and accuracy

the lower layers for which the loss gradient products involve more multiplicative components. The multiplicative components involve the sigmoid-function derivatives and the weights. The derivatives are bounded in $[0, 1]$, therefore contributing to the vanishing effect when present as a component in product (2.11).

Suboptimal initialisation of weights have detrimental effect on the variance of network signals and loss gradients. It is a cause for both the VGP and exploding gradients.

The VGP in Sigmoid FNNs reduces the model training performance. When the VGP is present, training loss and accuracy tend to remain at a poor level for long periods.

3.5.2 ReLU-parametreised FNNs

The ReLU-class activation functions avoid the saturation aspect of the VGP. The standard ReLU outputs zero for all negative inputs. This property creates sparsity in the hidden layer outputs that arguably reduces overfitting in training (Dubowski, 2020). However, zero outputs correspond to zero gradients, therefore contributing to the VGP. Appendix section A.1.1 contains plots for the 3 diagnostics of the standard ReLU model trained on Pima data. We can observe that zero-gradient problem in standard ReLU completely prevented learning of data.

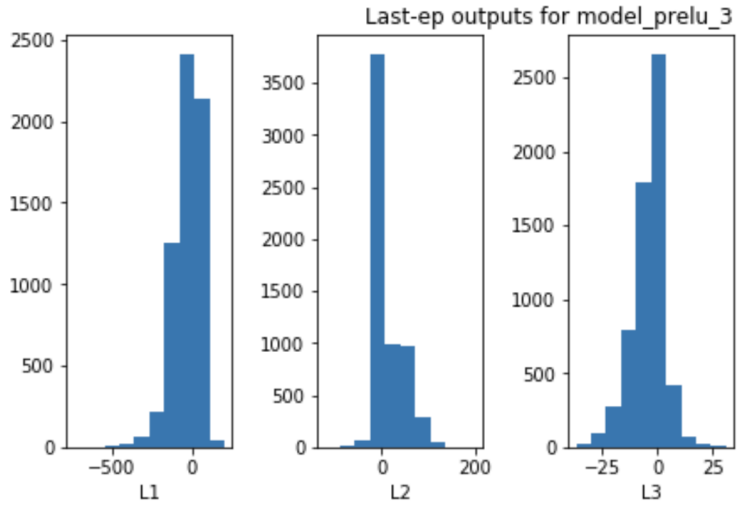

Figure 3.9: PReLU FNN with initialisation: C. last-epoch hidden layer outputs

The variant PReLU function offers significant reduction in the VGP and thus improved model training scores. It is worth noting that as more epochs are trained, the output gradients exhibit a steadily increasing trend in absolute value. As implied in the SGD algorithm (2.12), when we take too large a step in finding the loss minimum, we have to set smaller learning rate α. Therefore large magnitude of loss gradients may undermine optimisation efficiency. Techniques such as batch normalisation and gradient clipping (Pascanu et al, 2012) are common practice to alleviate this issue.

3.5.3 Diagnostics

It is expected that every deep learning practitioner will assess the model performance with the training loss and accuracy. However, the behaviour of loss gradients is only one factor affecting the performance. The output gradient plots offer straightforward visualisation of the VGP.

For models parameterised with saturating activation functions, it is more appropriate to use histograms (Figures 3.6, 3.9) to visualise the samples of hidden layer outputs to inspect the extent of saturation. Readers can typically plot the outputs at a few randomly selected epochs,

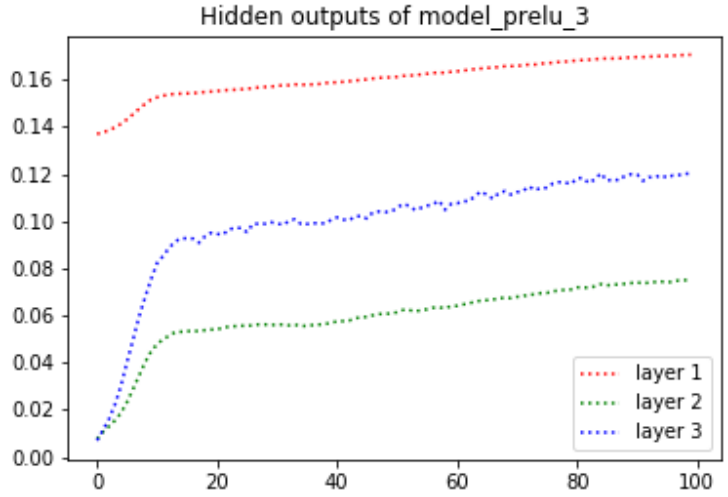

Figure 3.10: PReLU FNN with initialisation: C. aggregated hidden layer outputs

paying particular attention to the histograms corresponding to the lower layers of the model. For models parameterised with non-saturating activation functions, readers can aggregate the outputs and visualise with the lineplots, where the trend over epochs is clearly presented.

4 The VGP in Recurrent Neural Networks

In section 4 we first provide the details of a rigorous expression of the RNN loss gradients with respect to any weight parameter. Based on the form of the full loss gradient, we introduce an indicator for assessing the extent of the VGP in training RNNs.

The effectiveness of the 2 advanced RNN-class models, LSTM and GRU are analysed instead by expressing the arbitrary multiplicative component in the full loss gradients: (4.14) and (4.19) respectively. The full loss gradient expression as shown in section 4.1.1 will become very cumbersome for LSTM and GRU.

We finally compare the classification performances of the 3 models on the Toxic Comment Data in section 4.4.1.

4.1 VGP in an NLP Task

4.1.1 Computing RNN loss gradients

In RNN forward propagation equations (2.15, 2.17) one should take note that the 3 weights are involved in the computation at every timestep, also illustrated in Figure 4.1. Therefore when applying the chain rule to differentiate with respect to each of these weights, we have to consider the temporal dependency of the weight at each of the past steps.

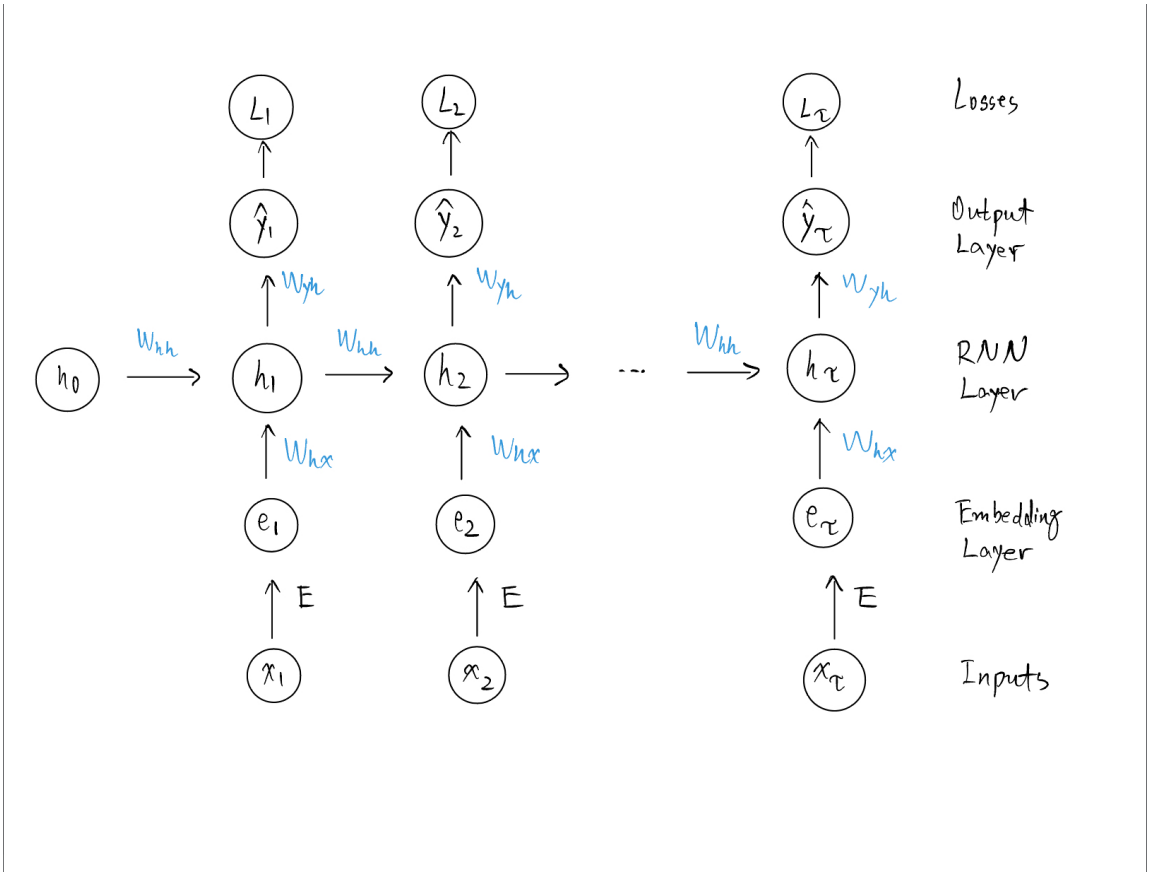

Figure 4.1: The standard RNN model applied to NLP

For the VGP in RNNs we are concerned with the loss gradients with respect to the weight matrices $\{W_{hx}, W_{hh}, W_{yh}\}$. First we will specify the derivative of softmax function (2.19) as $S'(\cdot)$, which is one element in the RNN loss gradient:

$$\begin{aligned} S'(s_t) = \frac{\partial \hat{y}_t^T}{\partial s_t} &= \left(\frac{\partial \hat{y}_t[k]}{\partial s_t[c]} \right)_{c,k=1,\dots,K} \\ &= \left((\delta_{k,c} - \hat{y}_t[c]) \hat{y}_t[k] \right)_{c,k=1,\dots,K} \\ &= \hat{Y}_t (I_K - \hat{Y}_t) \end{aligned} \tag{4.1}$$

where $\delta_{k,c} = 1$ if $k = c$; $\delta_{k,c} = 0$ otherwise. I_K is a (K, K) identity matrix. $\hat{Y}_t$ is a diagonal matrix with the diagonal being $\hat{y}_t$. Therefore $\frac{\partial \hat{y}_t^T}{\partial s_t}$ is a matrix of shape (K, K).

Differentiating the CE loss (2.20) gives the CE derivative $\frac{\partial L_t}{\partial \hat{y}_t}$:

$$\begin{aligned}\frac{\partial L_t}{\partial \hat{y}_t} &= \left(\frac{\partial L_t}{\partial \hat{y}_t[1]}, \cdots, \frac{\partial L_t}{\partial \hat{y}_t[K]}\right)^T \\ &= \left(-\frac{y_t[k]}{\hat{y}_t[k]}\right)_{k=1,\ldots,K}\end{aligned} \tag{4.2}$$

The above partial derivative is a vector of shape $(K, 1)$.

The product of softmax derivative (4.1) and the CE derivative (4.2) is therefore:

$$\begin{aligned}\frac{\partial \hat{y}_t^T}{\partial s_t}\frac{\partial L_t}{\partial \hat{y}_t} &= \hat{Y}_t(I_K - \hat{Y}_t)\left(-\frac{y_t[k]}{\hat{y}_t[k]}\right)_{k=1,\ldots,K} \\ &= I_K(\hat{Y}_t - I_K)y_t \\ &= \hat{Y}_t y_t - y_t\end{aligned}$$

A reminder of chain rule will be presented as follows. If x, y, z are all scalars with relationship defined as:

$$z = f(y)$$

$$y = g(x)$$

The partial derivative $\frac{\partial z}{\partial x}$ is straightforward a scalar. In the same setting where x, z are scalars, and $\mathbf{y}$ is now a vector of length n:

$$\frac{\partial z}{\partial x} = \frac{\partial \mathbf{y}^T}{\partial x}\frac{\partial z}{\partial \mathbf{y}}$$

where the right-hand side is multiplication between a $(1, n)$ vector and an $(n, 1)$ vector. The derivative is again a scalar.

In the case with $\mathbf{x} \in \mathbb{R}^{\mathbf{m}}, \mathbf{y} \in \mathbb{R}^{\mathbf{n}}$ being vectors of different lengths, and z still a scalar:

$$z = f(\mathbf{y})$$

$$\mathbf{y} = g(\mathbf{x})$$

The derivative from the chain rule is now:

$$\frac{\partial z}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial z}{\partial x_1} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial \mathbf{y}}{\partial x_1}\frac{\partial z}{\partial \mathbf{y}} \\ \vdots \\ \frac{\partial \mathbf{y}}{\partial x_m}\frac{\partial z}{\partial \mathbf{y}} \end{pmatrix} = \frac{\partial \mathbf{y}^T}{\partial \mathbf{x}}\frac{\partial z}{\partial \mathbf{y}}$$

where we have multiplication between a matrix of shape (m, n) and a vector of shape $(n, 1)$. The resulting derivative is of shape $(m, 1)$.

It is non-trivial to express the RNN loss gradients with the chain rule, because now we will have a matrix X of shape (m, n) in place of vector $\mathbf{x}$. The partial derivative $\frac{\partial \mathbf{y}^T}{\partial X}$ is no longer a matrix but a 3-dimensional array. Many resources fail to provide full explanation for chain rule application in this setting. For example in Pascanu et al. (2013) equation (4) involves such a partial derivative but without clarification on the dimensionality. One consequence is that such an expression cannot be written as a matrix product involving the explicit terms. Other texts e.g. Miller (Stanford CS231n) avoided this issue by suggesting to write formulae in scalar form: $\frac{\partial y_i}{\partial X_{j,k}}$.

It is in our mind to write the loss gradients in full detail, at the same time reflecting the dimensions of the quantities involves in RNN forward propagation. Therefore we present the rigorous way of writing the RNN loss gradients by vectorising the matrix on the denominator.

Magnus (1988) defined the 'only sensible definition' of a matrix derivative. Suppose a matrix function $F : S \to \mathbb{R}^{m \times p}$ is differentiable on a set $S \in \mathbb{R}^{n \times q}$. Matrix X with shape (n, q) will be mapped to matrix $F(X)$ of shape (m, p). Then the Jacobian of F at X is the following

matrix with shape (mp, nq):

$$\frac{\partial \text{vec}(F(X))}{\partial (\text{vec}(X))^T} \tag{4.3}$$

In BPTT the matrix function maps to a scalar loss in $\mathbb{R}$ instead. Therefore we only need to vectorise the weight matrices. Vectorisation of a product of more than 2 matrices will involve the Kronecker product.

(a) Kronecker product of 2 matrices:

Let A, B be matrices of shapes $(m, n), (p, q)$. The Kronecker product of A and B is $A \otimes B$, with shape (mp, nq) having the structure:

$$\begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ & \cdots & \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

where a_{ij} is the ij-th unit in matrix A.

A useful property (Magnus, 1988, result 1.15) is: $(A \otimes B)^T = A^T \otimes B^T$.

(b) Vectorisation of a matrix:

The vec$(\cdot)$ operator stacks the columns of a matrix into a long column vector. If a_j is the j-th column of the above matrix A, $j = 1 \cdots n$, then

$$\text{vec}(A) = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \in \mathbb{R}^{mn}$$

where each a_j has shape $(m, 1)$. Hence vec(A) has shape $(mn, 1)$. In BPTT we wish to express

$$\frac{\partial L_1}{\partial \text{vec}(W_{hx}^T)^T} = \frac{\partial L_1}{\partial h_1^T} \frac{\partial h_1}{\partial \text{vec}(W_{hx}^T)^T} \tag{4.4}$$

To simplify the notation, take transpose of the equation and move the transpose from the de-

nominator to the numerator:

$$\frac{\partial L_1}{\partial \text{vec}(W_{hx}^T)} = \frac{\partial h_1^T}{\partial \text{vec}(W_{hx}^T)} \frac{\partial L_1}{\partial h_1} \tag{4.5}$$

By definition, W_{hx} has shape (n_i, n_h), $\text{vec}(W_{hx}^T)$ will have shape $(n_h n_i, 1)$.

Recall $a_1 = W_{hx}^T e_1 + W_{hh}^T h_0; h_1 = \tanh(a_1)$, where tanh-function applies element-wise. Let $f = W_{hx}^T e_1 = I_{n_h} W_{hx}^T e_1$ where I_{n_h} is an identity matrix of size (n_h, n_h). f has shape $(n_h, 1)$.

In result (1.21) of Magnus (1988): $\text{vec}(ABC) = (C^T \otimes A)\text{vec}(B)$.

Applying it to $f = I_{n_h} W_{hx}^T e_1$:

$$\begin{aligned}
\text{vec}(f) &= (e_1^T \otimes I_{n_h})\text{vec}(W_{hx}^T) \\
d\text{vec}(f) &= df = (e_1^T \otimes I_{n_h})\, d\text{vec}(W_{hx}^T) \\
\frac{\partial f}{\partial \text{vec}(W_{hx}^T)^T} &= e_1^T \otimes I_{n_h} \\
&= \frac{\partial a_1}{\partial \text{vec}(W_{hx}^T)^T}
\end{aligned}$$

Then,

$$\begin{aligned}
\frac{\partial h_1}{\partial \text{vec}(W_{hx}^T)^T} &= \frac{\partial h_1}{\partial a_1^T} \frac{\partial a_1}{\partial \text{vec}(W_{hx}^T)^T} \\
&= \frac{\partial h_1}{\partial a_1^T} (e_1^T \otimes I_{n_h})
\end{aligned}$$

e_1 has shape $(n_i, 1)$, thus $e_1^T \otimes I_{n_h}$ has shape $(n_h, n_i n_h)$. $\frac{\partial h_1}{\partial \text{vec}(W_{hx}^T)^T}$ has shape $(n_h, n_i n_h)$.

In the loss gradient we need $\frac{\partial h_1^T}{\partial \text{vec}(W_{hx}^T)}$:

$$\begin{aligned}\frac{\partial h_1^T}{\partial \text{vec}(W_{hx}^T)} &= \frac{\partial a_1^T}{\partial \text{vec}(W_{hx}^T)} \frac{\partial h_1^T}{\partial a_1} \\ &= (e_1^T \otimes I_{n_h})^T \frac{\partial h_1^T}{\partial a_1} \\ &= (e_1 \otimes I_{n_h})(I_{n_h} - H_1^2)\end{aligned}$$

having shape $(n_i n_h, n_h)$. The overall first-step vectorised loss gradient:

$$\begin{aligned}\frac{\partial L_1}{\partial \text{vec}(W_{hx}^T)} &= \frac{\partial h_1^T}{\partial \text{vec}(W_{hx}^T)} \frac{\partial L_1}{\partial h_1} \\ &= (e_1 \otimes I_{n_h})(I_{n_h} - H_1^2) W_{yh} \hat{Y}_1 (\hat{Y}_1 - I_K) \left(\frac{y_1[k]}{\hat{y}_1[k]} \right)_{k=1,\ldots,K} \\ &= (e_1 \otimes I_{n_h})(I_{n_h} - H_1^2) W_{yh} (\hat{Y}_1 y_1 - y_1) \qquad (4.6)\end{aligned}$$

which has shape $(n_i n_h, 1)$.

Overall loss gradient

Using vectorisation in vector-matrix differentiation we can express the partial derivatives over time in such a manner that reflects the temporal dependency in applying the chain rule. Here we will show $\frac{\partial L}{\partial \text{vec}(W_{hh}^T)}$.
At steps 1 and 2:

$$\begin{aligned}\frac{\partial L_1}{\partial \text{vec}(W_{hh}^T)} &= (h_0 \otimes I_{n_h})(I_{n_h} - H_1^2) W_{yh} (\hat{Y}_1 y_1 - y_1) \\ \frac{\partial L_2}{\partial \text{vec}(W_{hh}^T)} &= \frac{\partial h_2^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial L_2}{\partial h_2} + \frac{\partial h_1^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial h_2^T}{\partial h_1} \frac{\partial L_2}{\partial h_2} \\ &= (h_1 \otimes I_{n_h})(I_{n_h} - H_2^2) W_{yh} (\hat{Y}_2 y_2 - y_2) \\ &\quad + (h_0 \otimes I_{n_h})(I_{n_h} - H_1^2) W_{hh} (I_{n_h} - H_2^2) W_{yh} (\hat{Y}_2 y_2 - y_2)\end{aligned}$$

The overall loss gradient has form:

$$\begin{aligned}
\frac{\partial L}{\partial \text{vec}(W_{hh}^T)} &= \sum_{t=1}^{\tau} \frac{\partial L_t}{\partial \text{vec}(W_{hh}^T)} \\
&= \frac{\partial L_1}{\partial \text{vec}(W_{hh}^T)} + \frac{\partial L_2}{\partial \text{vec}(W_{hh}^T)} + \cdots + \frac{\partial L_\tau}{\partial \text{vec}(W_{hh}^T)} \\
&= \frac{\partial h_1^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial L_1}{\partial h_1} + \left(\frac{\partial h_2^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial L_2}{\partial h_2} + \frac{\partial h_1^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial h_2^T}{\partial h_1} \frac{\partial L_2}{\partial h_2} \right) + \cdots \quad (4.7) \\
&+ \left(\frac{\partial h_\tau^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial L_\tau}{\partial h_\tau} + \frac{\partial h_{\tau-1}^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial h_\tau^T}{\partial h_{\tau-1}} \frac{\partial L_\tau}{\partial h_\tau} + \cdots \right. \\
&\left. + \frac{\partial h_1^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial h_2^T}{\partial h_1} \cdots \frac{\partial h_\tau^T}{\partial h_{\tau-1}} \frac{\partial L_\tau}{\partial h_\tau} \right) \\
&= \left[(h_0 \otimes I_{n_h})(I_{n_h} - H_1^2) W_{yh} (\hat{Y}_1 y_1 - y_1) \right] \quad (4.8) \\
&+ \left[(h_1 \otimes I_{n_h})(I_{n_h} - H_2^2) W_{yh} (\hat{Y}_2 y_2 - y_2) + \right. \\
&\left. (h_0 \otimes I_{n_h})(I_{n_h} - H_1^2) W_{hh} (I_{n_h} - H_2^2) W_{yh} (\hat{Y}_2 y_2 - y_2) \right] + \quad (4.9) \\
&\cdots + \\
&+ \left[(h_{\tau-1} \otimes I_{n_h})(I_{n_h} - H_\tau^2) W_{yh} (\hat{Y}_\tau y_\tau - y_\tau) + \right. \\
&(h_{\tau-2} \otimes I_{n_h})(I_{n_h} - H_{\tau-1}^2) W_{hh} (I_{n_h} - H_\tau^2) W_{yh} (\hat{Y}_\tau y_\tau - y_\tau) + \\
&\cdots + \\
&\left. (h_0 \otimes I_{n_h})(I_{n_h} - H_1^2) \left(\prod_{t=2}^{\tau} W_{hh} (I_{n_h} - H_t^2) \right) W_{yh} (\hat{Y}_\tau y_\tau - y_\tau) \right] \quad (4.10)
\end{aligned}$$

where τ is the final timestep. In particular, (4.8) is the loss gradient at step 1, (4.9) that at step 2, and (4.10) that at step τ.

Amongst the 3 loss gradients $\frac{\partial L_t}{\partial \text{vec}(W_{hx}^T)}$, $\frac{\partial L_t}{\partial \text{vec}(W_{hh}^T)}$, and $\frac{\partial L_t}{\partial \text{vec}(W_{yh}^T)}$, $\frac{\partial L_t}{\partial \text{vec}(W_{yh}^T)}$ does not suffer from the VGP. This is because each W_{yh} is not involved in the computation of $\hat{y}_t$ for any timestep via h_t. Under the chain rule, it is therefore a short product.

However for the other 2 weights W_{hx}, W_{hh}, hidden states computed using these weights are

involved in computations in the future time steps, again using the 2 same weights. As a result each stepwise loss gradient will sum over partial derivative products of different of temporal dependencies. In (4.7) the last bracket stands for $\frac{\partial L_\tau}{\partial \text{vec}(W_{hh}^T)}$, in which the first product represents the dependency over 0 step. The last product in $\frac{\partial L_\tau}{\partial \text{vec}(W_{hh}^T)}$ is dependency over $\tau - 1$ steps.

This longer partial derivative product involves the multiplications of weight matrices and the tanh-derivative matrices.

- The hidden state derivative $I_{n_h} - H_t^2$ is less than 1. In the product over time the long-term partial derivative products effectively vanish to zero.

- Pascanu et al (2013) have also shown that if the spectral radius of the weight matrix W_{hh} is less than 1, then powers of the same matrix will cause the entries to vanish.

The causes of the VGP in standard RNN have been elegantly summarised by Jozefowicz et al (2015): Loss signals exhibit exponential decay as they are back-propagated through time. The decay in BPTT leads to the long-term components in the loss gradients being effectively lost as they are overwhelmed by undecayed short-term components.

4.1.2 Consequences of the VGP in RNNs

Long-term dependency is the property that information in an early step of the sequence is carried through the timesteps for making a prediction. For example, in the sentence 'John felt sick this morning. He had his breakfast, took a bath before heading to the ___', with the last word being 'clinic'. The prediction of the last word requires information at the start of the sentence. If the model captures and relays the message 'felt sick' over the steps, then an accurate prediction can be made. In sequence classification of the comment sequences, the model may incorrectly classify a sentence as non-toxic if it cannot remember the offensive content in the earlier part of the sentence.

Referring to lines (4.7) - (4.10), the longer partial derivative products within each $\frac{\partial L_t}{\partial \text{vec}(W^T)}$ are more prone to vanishing. If for many steps the longer products vanish, the entire loss gradient will update in the same way as if the text sequence is much shorter, that back propagation through time does not reach the earlier steps. Consequently the RNN is not good at making predictions which depend on earlier information.

Therefore we have found that the term 'vanishing gradients' is not precise in the context of training RNNs. Researchers acknowledge that the overall loss gradients may not necessarily vanish entirely, whereas in FNNs we have seen them totally vanished. Instead, it is the vanishing of those longer partial derivative products that are associated with poor predictions.

4.2 Diagnosing the Problem in RNNs

4.2.1 An experiment for long-term dependency

In this subsection we use the Toxic Comment Data to demonstrate that RNN has difficulty capturing long-term dependency in making predictions. This is carried out by comparing the training loss and accuracy from model fitted on short and long sentences.

5000 comment sequences with length less than 10 words are randomly selected as 'short sentences'. 5000 long comments with more than 90 words are selected separately. The RNN model consists of a GloVe embedding layer followed by a standard RNN layer consisting of 50 units. Finally a one-unit sigmoid output layer is needed for binary classification. It is trained separately on short and long sequences for 20 epochs. The loss and accuracy are plotted:

On the last epoch, the RNN on short sequences obtained training CE loss of 0.0617, accuracy of 0.9812 $\%$. The RNN on long sequences obtained CE loss of 0.1213, and accuracy of 0.9615$\%$. Within same number of epochs the RNN model achieved better loss and accuracy on shorter senquences. However the considerable difference in sentence length means that this difference

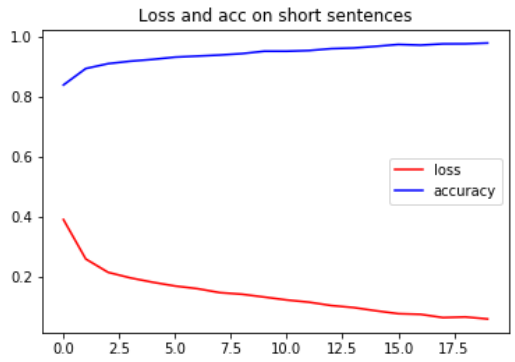

Figure 4.2: Training loss and accuracy on short sentences

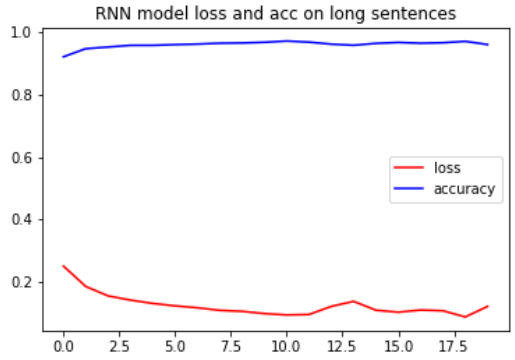

Figure 4.3: Training loss and accuracy on long sentences

in training performance is not significant.

Considering the nature of RNN that it processes an input sentence temporally, updating its internal states h_t which serve as memory, we have looked closely at comments labelled 'toxic'. We found that regardless of the length, toxic comments generally contain unfriendly and offensive words throughout. If a model is unable to capture the toxicity in the earlier steps of a sequence, it is likely that the model gets 'reminded' of the toxicity at later steps.

One possible modification is to experiment with those long sequences having the last few words replaced with some words that are neutral and irrelevant, thereby diluting the memory of the RNN at the end. This modification itself did not present a significant drop in loss and accuracy. We attribute this to the low concentration of toxic comments in the dataset. In the raw data there are 202165 sequences labelled 'non-toxic', and only 21384 labelled 'toxic'. Therefore on

the training set it has a high tendency to classify a comment as 'non-toxic'.

Therefore 2000 toxic long sequences and 2000 non-toxic long sequences are selected as a new set of training data. Now toxic comments account for half of the training data. With that we found the loss and accuracy fitted on long sequences having significantly worsened. In Figure 4.4, one can observe a slight improvement over the training process.

In addition, we replaced the last 20 words of the comments in this new training set with word

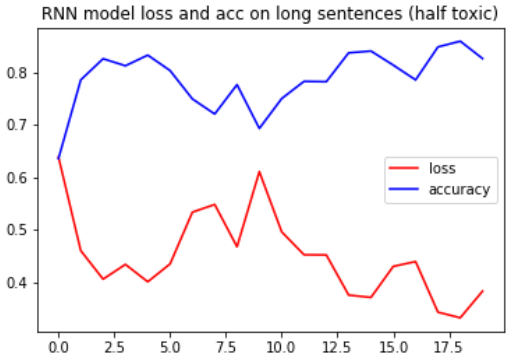

Figure 4.4: Training loss and accuracy on half toxic long sequences

'the', a non-toxic word.

In Figure 4.5, the loss and accuracy were further worsened, and there was no improvement

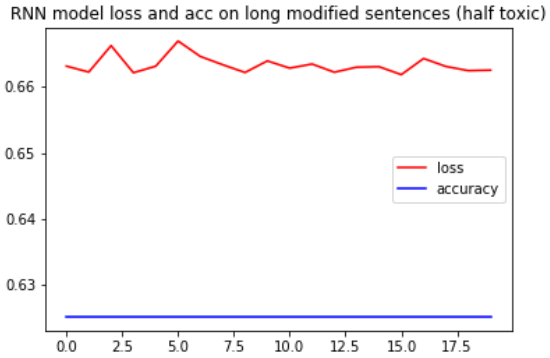

Figure 4.5: Training loss and accuracy on half toxic long modified sentences

over the training process. It is also worth noting that the loss and accuracy from fitting on modified long sequences have not improved in training, regardless of whether the concentration of

toxic comments has been adjusted.

With the results of the model performance statistics (loss and accuracy), we have experimentally shown that due to the VGP, standard RNN fails to capture long-term dependency for accurate classification. Essential information for making the overall prediction can be overwritten as forward propagation proceeds.

4.2.2 An indicator for VGP

An appropriate indicator is required to assess the extent of VGP in training RNNs, since the overall loss gradients do not necessarily vanish. Below we have used a compact expression from Pascanu et al (2013) to express each step-wise loss gradient:

$$\frac{\partial L_t}{\partial \text{vec}(W_{hh}^T)} = \sum_{k=1}^{t} \frac{\partial^+ h_k^T}{\partial \text{vec}(W_{hh}^T)} \frac{\partial h_t^T}{\partial h_k} \frac{\partial L_t}{\partial h_t} \tag{4.11}$$

$$\frac{\partial h_t^T}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i^T}{\partial h_{i-1}} \tag{4.12}$$

In particular, each product $\frac{\partial h_t^T}{\partial h_k}$ is subject to vanishing effect of different degree, depending on the value of $t - k$.

The superscript $+$ in the numerator stands for 'immediate' partial derivative. $\partial^+ h_k^T$ means that we differentiate the step-k hidden state with respect to the vectorised hidden-to-hidden weight. The earlier states $h_1, ..., h_{k-1}$ also contain the same weight parameter but we treat them as constants, in the k-th component of summation (4.11).

Noh (2021) inspected the behaviour of $\frac{\partial h_t^T}{\partial h_0}$ to assess the VGP. This statistic can be considered as a proxy for $\frac{\partial h_t^T}{\partial h_1}$ which is part of the longest partial derivative product in each $\frac{\partial L_t}{\partial \text{vec}(W_{hh}^T)}$. It is also the strictest-case indicator for the VGP because we demand information as early at

step 1 to be relayed to the end. The author applied this statistic to a time-series dataset for a regression task, but did not provide the code. We have produced the code for computing $\frac{\partial h_t^T}{\partial h_0}$ within model-fitting, and used it as an indicator for the extent of the VGP on our text dataset.

Implementing Noh's indicator

While the indicator $\frac{\partial h_t^T}{\partial h_0}$ can be expressed as a product of partial derivatives, in practice it is not necessary to compute each component and then the product. From the `Tensorflow` library, the `GradientTape` function keeps track of the initial state h_0, computes the step-t state h_t and computes the partial derivative.

It is worth noting that throughout the thesis we have defined the models under the general case, e.g. the forward propagation equations in section 2.2.3, where at each step an output and its associated loss is computed. However in sequence classification tasks there are no intermediate outputs and losses because it does not make sense to classify before the whole sequence has been processed. The overall loss gradient is in fact the gradient at the final step: $\frac{\partial L}{\partial \text{vec}(W_{hh}^T)} = \frac{\partial L_\tau}{\partial \text{vec}(W_{hh}^T)}$ where τ is the final step. Therefore we cannot compute the indicator $\frac{\partial h_t^T}{\partial h_0}$ for each step t. Instead $\frac{\partial h_\tau^T}{\partial h_0}$ is computed for 20 epochs of training. The graphical results from applying this indicator will be presented in section 5.

4.3 The Solutions in RNNs

At first glance one might be prompted to resort to using the ReLU-class activation functions in RNNs in order to alleviate the vanishing effect brought by the tanh-functions.

One justification for insisting on the tanh-function is that it has zero-centred outputs, which has the effect of maintaining the variance of signals in forward propagation. In the case of training RNNs, the forward propagation is computed for every timestep. Using ReLUs may lead to extremely high variance in outputs at later timesteps, especially when training on long sequences. The normalisation technique mentioned in section 3.2.1 can control the output variance but is

difficult to implement in practice. Because an RNN is constructed by stacking layers, while normalisation is needed for every timestep. In software we can only add a normalisation layer after the RNN layer, however in this case the normalisation applies to the wrong dimension, and will not address the high variance in stepwise outputs.

While there exist other techniques dealing with the VGP such as Hessian-free optimisation and regularisation, this thesis will focus on the more advanced RNN-class models as solutions. They are the Long Short-Term Memory (LSTM), Gated Recurrent Units (GRU) and bidirectional models. For these models the overall loss gradients are extremely long, therefore we will not present expressions similar to (4.7) - (4.10). Instead the form of the smallest multiplicative components (e.g. 4.14) will be shown.

4.3.1 Back propagation in LSTM

Again we use the chain rule to write out the stepwise loss gradient, with respect to an arbitrary gate weight matrix $W \in \{W_i, W_f, W_c, W_o, U_i, U_f, U_c, U_o\}$. Using vectorisation for matrix differentiation, and writing in Pascanu's compact expression:

$$\frac{\partial L_t}{\partial \text{vec}(W^T)} = \sum_{k=1}^{t} \frac{\partial^+ c_k^T}{\partial \text{vec}(W^T)} \frac{\partial c_t^T}{\partial c_k} \frac{\partial h_t^T}{\partial c_t} \frac{\partial L_t}{\partial h_t} \tag{4.13}$$

where $\frac{\partial c_t^T}{\partial c_k} = \prod_{k<j\leq t} \frac{\partial c_j^T}{\partial c_{j-1}}$ is a matrix of shape (n_h, n_h).

In standard RNN the hidden state partial derivatives $\frac{\partial h_j^T}{\partial h_{j-1}}$ is the product of tanh-derivative and the weight matrix. The product over timesteps of $\frac{\partial h_j^T}{\partial h_{j-1}}$ has the vanishing effect. In LSTM the cell states store the information in inputs, and the long-term dependency in sequences is carried by the cell states, thus we need to inspect $\frac{\partial c_j^T}{\partial c_{j-1}}$. Note that j, k, t are all time steps.

Take derivative of c_j^T with respect to c_{j-1} via each possible path:

$$
\begin{aligned}
c_j &= c_{j-1} \odot f_j + \tilde{c}_j \odot i_j \\
\frac{\partial c_j^T}{\partial c_{j-1}} &= \frac{\partial^+ c_j^T}{\partial c_{j-1}} + \frac{\partial h_{j-1}^T}{\partial c_{j-1}} \frac{\partial f_j^T}{\partial h_{j-1}} \frac{\partial c_j^T}{\partial f_j} + \frac{\partial h_{j-1}^T}{\partial c_{j-1}} \frac{\partial \tilde{c}_j^T}{\partial h_{j-1}} \frac{\partial c_j^T}{\partial \tilde{c}_j} + \frac{\partial h_{j-1}^T}{\partial c_{j-1}} \frac{\partial i_j^T}{\partial h_{j-1}} \frac{\partial c_j^T}{\partial i_j}
\end{aligned} \tag{4.14}
$$

A reminder on the derivative of a Hadamard product. Suppose x, y are vectors of shape $(n, 1)$, $f = x \odot y$:

$$
\frac{\partial f}{\partial x^T} = \text{diag}(y) := Y
$$

A diagonal matrix with diagonal elements being the elements in y.

A reminder on derivatives of sigmoid and hyperbolic functions. Suppose x, y, z are vectors with the same length. Sigmoid and tanh functions are applied element-wise, with $y = \sigma(x), z = \tanh(y)$:

$$
\begin{aligned}
\frac{\partial y}{\partial x^T} &= \sigma'(x) = \sigma(x)(1 - \sigma(x))^T \\
\frac{\partial y^T}{\partial x} &= \sigma'(x)^T = (1 - \sigma(x))\sigma(x)^T \\
\frac{\partial z}{\partial y^T} &= (I_n - Z^2) = \frac{\partial z^T}{\partial y}
\end{aligned}
$$

where $Z = \text{diag}(z)$. Taking differentials for terms needed in (4.14):

$$\begin{aligned}
dc_j &= F_j dc_{j-1} \\
dh_{j-1} &= (I_{n_h} - \text{diag}(\tanh(c_{j-1})^2)) O_{j-1} dc_{j-1} \\
df_j &= W_f^T f_j (1_{n_h} - f_j)^T dh_{j-1} \\
dc_j &= C_{j-1} df_j \\
d\tilde{c}_j &= W_c^T (I_{n_h} - \tilde{C}_j^2) dh_{j-1} \\
dc_j &= I_j d\tilde{c}_j \\
di_j &= W_i^T i_j (1_{n_h} - i_j)^T dh_{j-1} \\
dc_j &= \tilde{C}_j di_j
\end{aligned}$$

where each capital letter with a timestep subscript denotes the diagonal matrix with the diagonal being the vector represented by the lower-case letter. 1_{n_h} is a column vector of ones. Plugging in all the partial derivatives to (4.14):

$$\begin{aligned}
\frac{\partial c_j^T}{\partial c_{j-1}} &= F_j + O_{j-1}(I_{n_h} - \text{diag}(\tanh(c_{j-1})^2))(1_{n_h} - f_j) f_j^T W_f C_{j-1} \\
&\quad + O_{j-1}(I_{n_h} - \text{diag}(\tanh(c_{j-1})^2))(I_{n_h} - \tilde{C}_j^2) W_c I_j \\
&\quad + O_{j-1}(I_{n_h} - \text{diag}(\tanh(c_{j-1})^2))(1_{n_h} - i_j) i_j^T W_i \tilde{C}_j \\
&:= A_j + B_j + C_j + D_j
\end{aligned} \tag{4.15}$$

Therefore the step t loss gradient has the form

$$\frac{\partial L_t}{\partial \text{vec}(W^T)} = \sum_{k=1}^{t} \frac{\partial^+ c_k^T}{\partial \text{vec}(W^T)} \{ \prod_{k<j\le t} \big(A_j + B_j + C_j + D_j\big) \} \frac{\partial h_t^T}{\partial c_t} \frac{\partial L_t}{\partial h_t} \tag{4.16}$$

Properties of LSTM that alleviates VGP

Firstly the gates are constructed from trainable parameters, i.e. the gate weights $\{W, U\}$. Theoretically they will accurately learn the data and decide to forget or retain information where appropriate.

Secondly, the additive expression in the product within the cell state partial derivatives makes it more well-behaved than the partial derivative product in RNN loss gradients (4.8 - 4.10). $\{A_j + B_j + C_j + D_j\}$ altogether contain learnable gates $i_j, f_j, \tilde{c}_j, o_j$. At each step they balance the sum such that the total loss gradient over all the timesteps does not vanish.

Lastly LSTM (GRU as well) computes the memory holder in 2 steps. The candidate state is the tanh-activated quantity but the actual cell state is a linear combination involving the candidate state. Therefore the tanh-derivative is not directly involved in the product $\prod_{k<j\leq t}$ in (4.16), partly alleviating the VGP.

4.3.2 Back-propagation in GRU

For a GRU cell the vectorised loss gradient has form:

$$\frac{\partial L_t}{\partial \text{vec}(W^T)} = \sum_{k=1}^{t} \frac{\partial^+ h_k^T}{\partial \text{vec}(W^T)} \frac{\partial h_t^T}{\partial h_k} \frac{\partial L_t}{\partial h_t} \tag{4.17}$$

$$\frac{\partial h_t^T}{\partial h_k} = \prod_{k<j\leq t} \frac{\partial h_j^T}{\partial h_{j-1}} \tag{4.18}$$

where W is an arbitrary weight from $\{W_z, U_z, W_r, U_r, W_h, U_h\}$. Again we inspect the smallest multiplicative component. Take derivative of h_j^T with respect to h_{j-1} via each possible path:

$$\begin{aligned} h_j &= z_j \odot h_{j-1} + (1 - z_j) \odot \tilde{h}_j \\ \frac{\partial h_j^T}{\partial h_{j-1}} &= \frac{\partial \tilde{h}_j^T}{\partial h_{j-1}} \frac{\partial h_j^T}{\partial \tilde{h}_j} + \frac{\partial z_j^T}{\partial h_{j-1}} \frac{\partial h_j^T}{\partial z_j} + \frac{\partial^+ h_j^T}{\partial h_{j-1}} \\ &= \left(\frac{\partial r_j^T}{\partial h_{j-1}} \frac{\partial \tilde{h}_j^T}{\partial r_j} + \frac{\partial^+ \tilde{h}_j^T}{\partial h_{j-1}} \right) \frac{\partial h_j^T}{\partial \tilde{h}_j} + \frac{\partial z_j^T}{\partial h_{j-1}} \frac{\partial h_j^T}{\partial z_j} + \frac{\partial^+ h_j^T}{\partial h_{j-1}} \end{aligned} \tag{4.19}$$

Note that r_j is dependent on h_{j-1} and thus we have differentiated r_j in (4.19).

Take differentials of the GRU forward propagation equations, based on terms needed in (4.19):

$$
\begin{aligned}
dh_j &= (I_{n_h} - Z_j)d\tilde{h}_j \\
d\tilde{h}_j &= H_{j-1}W_h^T(I_{n_h} - \tilde{H}_j^2)dr_j \\
dr_j &= W_r^T r_j(1_{n_h} - r_j)^T dh_{j-1} \\
d\tilde{h}_j &= R_j W_h^T(I_{n_h} - \tilde{H}_j^2)dh_{j-1} \\
dh_j &= (H_{j-1} - \tilde{H}_j)dz_j \\
dz_j &= W_z^T z_j(1_{n_h} - z_j)^T dh_{j-1} \\
dh_j &= Z_j dh_{j-1}
\end{aligned}
$$

Therefore the overall partial derivative is:

$$
\begin{aligned}
\frac{\partial h_j^T}{\partial h_{j-1}} = &\Big((1_{n_h} - r_j)r_j^T W_r(I_{n_h} - \tilde{H}_j^2)W_h H_{j-1} + (I_{n_h} - \tilde{H}_j^2)W_h R_j\Big)(I_{n_h} - Z_j) \\
&+ (1_{n_h} - z_j)z_j^T W_z(H_{j-1} - \tilde{H}_j) + Z_j
\end{aligned} \tag{4.20}
$$

Similar to $\frac{\partial c_j^T}{\partial c_{j-1}}$ in LSTM, $\frac{\partial h_j^T}{\partial h_{j-1}}$ in GRU has the form of a summation instead of a product, as in a standard RNN cell.

4.3.3 Effect of gates

Outputs from a sigmoid function are often close to 1 and 0. A gate is thus often assumed to be either active or inactive. The effect of the gates are often misinterpreted in sources online due to incomplete application of chain rule. Lectures slides and blog articles often treat the reset gate as constant, thus not differentiating r_j with respect to h_{j-1} in computing $\frac{\partial h_j^T}{\partial h_{j-1}}$, e.g. [14] Guerzhoy CS Toronto lecture, or provide the correct expression for the partial derivatives, but missing the interpretation completely, e.g. [32] Li, BPTT tutorial. In the original paper Cho et

al (2014) have briefly described the implication of the states of the 2 GRU gates using the hidden states h_j. Here we attempt to elaborate on their description, and additionally demonstrate the effect using the hidden state partial derivatives $\frac{\partial h_j^T}{\partial h_{j-1}}$.

Different states of GRU gates

In general, the update gate z controls the amount of memory in the previous hidden state to be propagated into the current h_j. The reset gate r drops any previous information found to be irrelevant to the future, as r_j is associated with h_{j-1} in $\tilde{h}_j$ with Hadamard product (see 2.30). When the update gate is active, $z_j = 1_{n_h}$ no matter the state of the reset gate, the memory in the previous hidden state is allowed to pass on fully. The derivative $\frac{\partial h_j^T}{\partial h_{j-1}} = Z_j = I_{n_h}$. Therefore, $h_j = h_{j-1}$. In this case GRU is capable of fully capturing the dependency between 2 steps. Update gate z has to be (relatively) active from step j to τ to capture long-term dependency if the prediction requires the memory at step j.

When the reset gate is inactive, *and* when the update gate is also inactive, the candidate hidden state ignores the previous hidden state but only picks up the current input: $\tilde{h}_j = \tanh(U_h^T e_j)$. This is also reflected in the partial derivatives. When $z_j, r_j = 0$, $\frac{\partial h_j^T}{\partial h_{j-1}} = 0$. That is, the current hidden state does not depend on the previous hidden state at all, but on the current input. There is no dependency on all the earlier information stored in h_{j-1}, hence *resetting* the network memory.

In the case where the reset gate is active and the update gate inactive, hidden state vector h_j and vector partial derivative $\frac{\partial h_j^T}{\partial h_{j-1}}$ have the forms:

$$
\begin{aligned}
h_j = \tilde{h}_j &= \tanh(W_h^T h_{j-1} + U_h^T e_j) \\
\frac{\partial h_j^T}{\partial h_{j-1}} &= (I_{n_h} - \text{diag}(\tanh(W_h^T h_{j-1} + U_h^T e_j)^2))W_h
\end{aligned}
$$

We see that both h_{j-1}, e_j are propagated to the current hidden state. Cho et al described this situation as: 'Those (GRU) units that learn to capture short-term dependencies will tend to have reset gates that are frequently active'. Indeed, new inputs included at every step will consistently dilute the memory in h_{j-1} at every further step, therefore capturing dependencies over a shorter time span.

Further implications

It is worth clarifying that capturing long- or short-term dependencies is not a trend to or away from VGP, but instead the capability of GRU in learning the temporal dependencies required. The gates are parameterised by the trainable weights. The states of the gates are therefore adjusted in the process of learning the dataset.

From above, different states of the 2 gates provide the effects of retaining long-term information, capturing short-term dependencies and resetting the memory. These features prompt the question of necessity: do we have to demand the model to capture long-term dependency for all steps and all inputs? The answer is clearly no, as both LSTM and GRU are designed to provide instant clearing of memory and short-term retaining of memory.

We describe the resetting of hidden states by GRU (eq. cell states in LSTM) as a case of zero-gradient resulted from proper learning of data and parameter optimisation, rather than a consequence of the VGP. However a point to stress is that, the VGP may take place in another way when computing the loss gradients.

For each product in the summation of the stepwise loss gradient (4.17), an arbitrary multiplicative component (4.19) has the form of a summation. However when the product (4.18) involves a large number of timesteps ($t >> k$), a product of sums (4.18) can still become a sum of long products. Therefore the vanishing effect cannot be entirely removed. The resulted loss gradients may lead to suboptimal updating of the weights and hence incorrect decisions of the gates, because the state of a gate is fully determined by the weights. This will eventually undermine the learning of data and predictions made.

Therefore in section 4.4.1 and section 5, we compare both the model training scores between the SRNN, LSTM and GRU, as well as the behaviours of Noh's indicator in the 3 models. This will hopefully draw some connections between the extent of vanishing gradients and the performance of learning the data.

4.4 Visualising the Effectiveness of Solutions

Before the experiments, we will briefly mention another existing model structure: the bidirectional RNN (Schuster & Paliwal, 1997). It offers an extra forward propagation step in the opposite direction, aggregates the hidden states of the forward and backward directions before computing the outputs. We would like to point out that for out task where the entire sequences are processed before classification, the bidirectional structure does not offer significant improvement in learning performance. The model structure and experimental results are provided in Appendix section A.2.1 for justification.

4.4.1 Comparing the models on sequence classification

To demonstrate how far LSTM and GRU improve the training performance as compared to SRNN in sequence classification, we have picked 1000 toxic and 1000 non-toxic sequences, replaced each of the last 30% of words in the sequences with the non-toxic word 'the', with the motivation stated in 4.2.1. All 3 models are fitted for 20 epochs. Both the training and validation losses and accuracies are plotted:

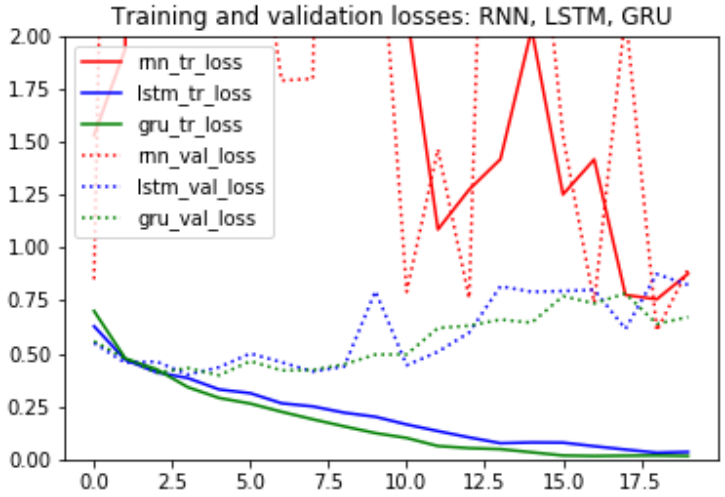

Figure 4.6.

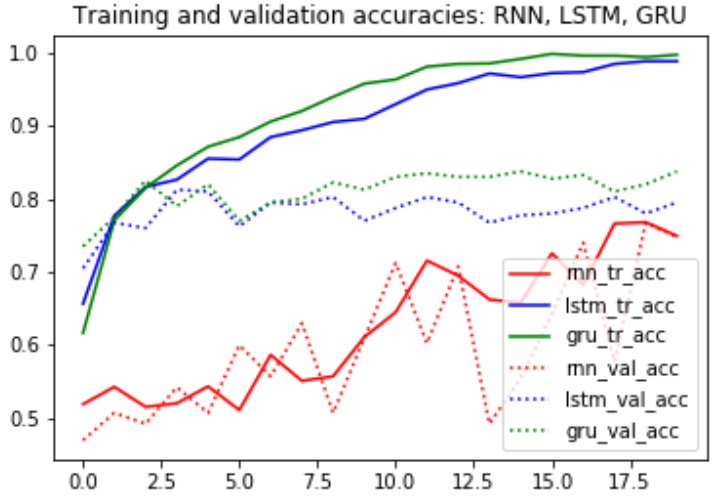

Figure 4.7.

Both LSTM and GRU offer significant improvement in training and validation performances, even without tuning the hyperparameters. The losses and accuracies of the GRU are even better. In addition GRU offers more efficient model-fitting due to the fewer model components. On average the GRU model requires 25% less time than the LSTM model to complete one epoch of training.

5 Behaviours of RNN loss gradients

We implement Noh (2021)'s indicator to the LSTM, GRU and SRNN to assess the extent of the VGP over epochs of training. Experiments in section 4.4.1 demonstrate the superior performance of the LSTM and GRU over SRNN on sequence classification. Experiments in section 5.1 aim to show whether the improved classification performance is a consequence of the reduction in the VGP.

Results in 5.1 however are opposite to our earlier expectations. We have observed instead a greater trend of the VGP in the LSTM and GRU. Rehmer & Kroll (2020) provided justification to this observation to some extent, but covered only the GRU and SRNN. For completeness, we have applied their comparison analysis to the LSTM and SRNN in section 5.2.2.

5.1 Comparing the extent of vanishing gradients

For SRNN Noh (2021) actually used $\frac{\partial s_t^T}{\partial h_0}$ as the indicator, instead of $\frac{\partial h_t^T}{\partial h_0}$. Recall that $s_t = W_{yh}^T h_t$. For GRU the author used $\frac{\partial h_t^T}{\partial h_0}$. We are not completely convinced by the appropriateness of using $\frac{\partial s_t^T}{\partial h_0}$ as the VGP indicator for SRNN in the sequence classification task, due to the following reason: Softmax of s_t gives the output $\hat{y}_t$, see (2.17) and (2.19). However in sequence classification forward propagation computes output only at the final step. It is possible to involve the output weights W_{yh} and compute s_t at earlier steps $t < \tau$, but not necessary. Therefore $\frac{\partial h_t^T}{\partial h_0}$ is a more appropriate general VGP indicator for both SRNN and GRU. In software at each step t the partial derivative is a matrix of shape $(\text{batch size}, n)$ where batch size is the size of the portion of all inputs fed into the model at one time; and n is the number of

units in a cell. Log mean absolute value is used as the aggregation for the matrix entries. For LSTM, Noh (2021) computed the quantities $\frac{\partial c_t^T}{\partial c_0}$ instead. The definition of the term $\frac{\partial c_t^T}{\partial c_k}$ in (4.13) justifies why $\frac{\partial c_t^T}{\partial c_0}$ is the general VGP indicator for LSTM.

Over epochs

As explained earlier, in sequence classification the RNN models only compute the final-step output and loss in each epoch of training. Therefore we computed $\frac{\partial h_\tau^T}{\partial h_0}$ matrix for SRNN and GRU, and $\frac{\partial c_\tau^T}{\partial c_0}$ matrix for LSTM. Each model is trained for 20 epochs and has the aggregated value of the indicator matrix recorded and plotted.

We can observed that the Noh's indicator of SRNN is in general significantly larger than LSTM and GRU. The values of GRU have typically vanished.

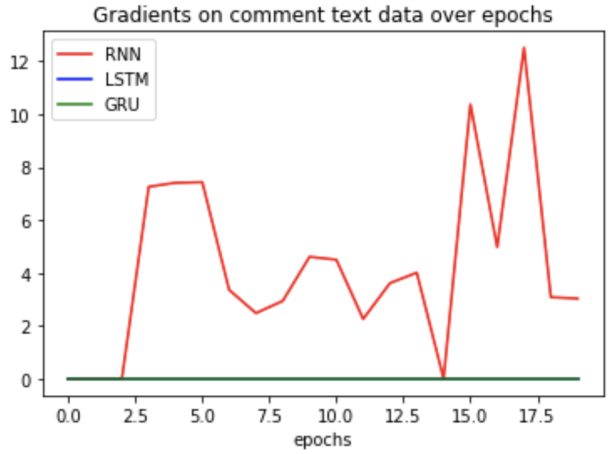

Figure 5.1: Noh's (2021) indicator over epochs

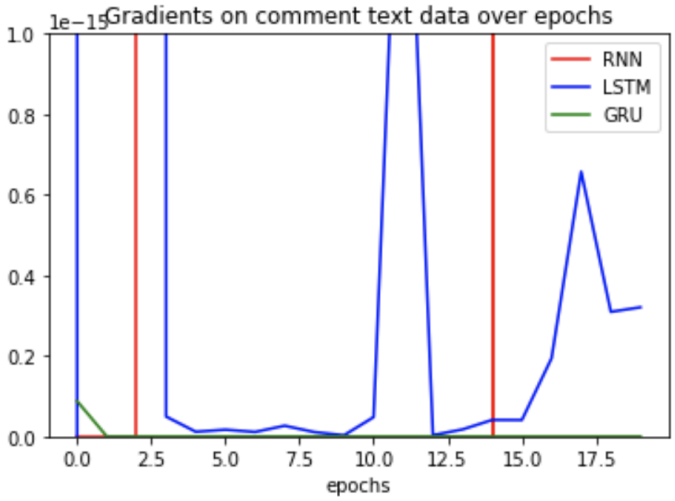

Figure 5.2: Figure 5.1 zoomed in

Over steps

In the model construction and gradient computation the true response y is not involved. Therefore it is possible to compute the indicators $\frac{\partial h_t^T}{\partial h_0}$ and $\frac{\partial c_t^T}{\partial c_0}$ for every t, as though we ignore the nature of the response y and the context of sequence classification. Here we went slightly beyond the task context in order to look at the behaviour of the indicator over the steps. Aggregated indicator values of the 3 models over 300 timesteps are plotted in Figure 5.3.

SRNN has the indicator values greater than LSTM and GRU throughout the steps, and the

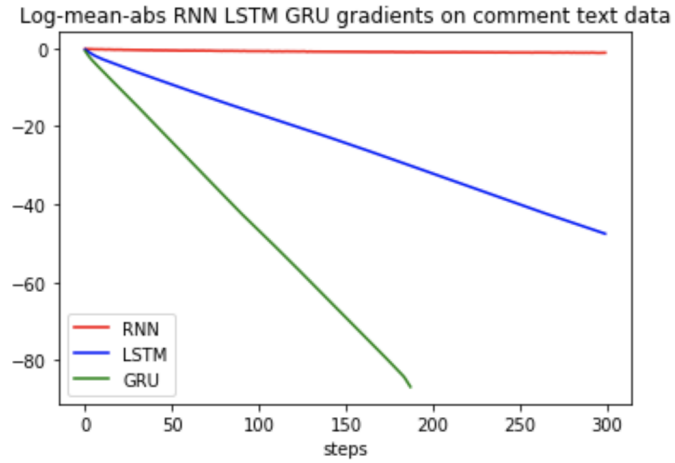

Figure 5.3: Noh's (2021) indicator over steps

GRU has the smallest values. To some degree it also acknowledges the unexpected behaviour in Figures 5.1 and 5.2.

5.2 An alternative viewpoint on VGP in RNN

5.2.1 Comparing RNN and GRU gradients

Our results suggests that under a strict indicator the SRNN is more vanishing gradient-resistant than LSTM and GRU. This is opposite to the mainstream understanding that LSTM and GRU are designed to combat the VGP. Here we would like to present an alternative viewpoint of the VGP in training RNNs due to Rehmer & Kroll (2020), whose conclusions support our findings at least to some extent.

In section 3 of Rehmer & Kroll (2020), the authors have shown that the hidden state gradients with respect to any GRU weight parameter are equal to those of the SRNN when the update gate is closed. For all other gate status the GRU gradients are smaller than the RNN gradients. Subsequently, the authors have argued that smaller gradients in GRU actually *help* gradient-based optimisation, in producing a smooth loss function without large gradients. They attribute this benefit to the superior performance of GRU over SRNN.

Rehmer & Kroll (2020) have parameterised each gate with only one weight parameter W, instead of separately input and hidden weights. In addition to simplify the comparison they have eliminated the reset gate by assuming it is always active, i.e. $r_t = 1$ for all t. We repeated the comparisons but for the standard parameterisation, and for gradients with respect to all 6 GRU weights. The same conclusion is shown.

The weights and thus the loss gradients are expressed in scalar form, in order to be consistent with the derivations in Rehmer & Kroll (2020). This is an alternative way of expressing the loss gradients, e.g. as done in Miller's notes (Stanford CS231n).

For the purpose of comparison, the authors have assumed the magnitude of the weight entries

$|w| \leq 1$ for any weights. In addition $\tanh(x) < 1$ has an upper bound.

Standard RNN hidden state gradients

Referring to the scalar form of equations (2.15) and (2.16), differentiate the SRNN hidden state with respect to the scalar weights:

$$\begin{aligned} \left|\frac{\partial h_t^T}{\partial w_{hh}}\right| &= \left|(h_{t-1} + w_{hh}\frac{\partial h_{t-1}^T}{\partial w_{hh}})\right|(1-h_t^2) \\ &\leq \left|h_{t-1} + w_{hh}\frac{\partial h_{t-1}^T}{\partial w_{hh}}\right| \end{aligned} \tag{5.1}$$

$$\begin{aligned} \left|\frac{\partial h_t^T}{\partial w_{hx}}\right| &= \left|(e_t + w_{hh}\frac{\partial h_{t-1}^T}{\partial w_{hx}})\right|(1-h_t^2) \\ &\leq \left|e_t + w_{hh}\frac{\partial h_{t-1}^T}{\partial w_{hx}}\right| \end{aligned} \tag{5.2}$$

Note that each hidden state gradient is a vector, all the comparisons in this chapter are made in an arbitrary vector norm, for instance the Euclidean norm.

GRU hidden state gradients with respect to the update gate

Differentiate the GRU hidden state h_t with respect to the hidden update weight w_z:

$$\begin{aligned} \frac{\partial h_t^T}{\partial w_z} &= \frac{\partial z_t^T}{\partial w_z}h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial w_z} + \left(-\frac{\partial z_t^T}{\partial w_z}\tilde{h}_t + (1-z_t)\frac{\partial \tilde{h}_t^T}{\partial w_z}\right) \\ &= z_t(1-z_t)\frac{\partial h_{t-1}^T}{\partial w_z}h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial w_z} - z_t(1-z_t)h_{t-1}\tilde{h}_t + (1-z_t)(1-\tilde{h}_t^2)w_h r_t\frac{\partial h_{t-1}^T}{\partial w_z} \end{aligned} \tag{5.3}$$

When $z_t = 0, r_t = 0$, $\frac{\partial h_t^T}{\partial w_z} = (1-\tilde{h}_t^2)w_h r_t\frac{\partial h_{t-1}^T}{\partial w_z} = 0$.
When $z_t = 0, r_t = 1$, $\left|\frac{\partial h_t^T}{\partial w_z}\right| \leq \left|\frac{\partial h_{t-1}^T}{\partial w_z}\right|$.
When $z_t = 1$, it can be shown that regardless of the parameter, the hidden state gradient at the current step is equal to the hidden state gradient at the previous step.

Similarly, differentiate the GRU hidden state h_t with respect to the input update weight u_z:

$$\frac{\partial h_t^T}{\partial u_z} = z_t(1-z_t)(w_z\frac{\partial h_{t-1}^T}{\partial u_z} + e_t)h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial u_z} - z_t(1-z_t)(w_z\frac{\partial h_{t-1}^T}{\partial u_z} + e_t)\tilde{h}_t$$
$$+ (1-z_t)(1-\tilde{h}_t^2)\left(w_h(r_t(1-r_t)w_r\frac{\partial h_{t-1}^T}{\partial u_z}h_{t-1} + r_t\frac{\partial h_{t-1}^T}{\partial u_z}) + e_t\right) \tag{5.4}$$

When $z_t = 0, r_t = 0$, $\left|\frac{\partial h_t^T}{\partial u_z}\right| = (1-\tilde{h}_t^2)\,|e_t| \leq |e_t|$.

When $z_t = 0, r_t = 1$, $\left|\frac{\partial h_t^T}{\partial u_z}\right| = (1-\tilde{h}_t^2)\left|w_h\frac{\partial h_{t-1}^T}{\partial u_z} + e_t\right| \leq \left|\frac{\partial h_{t-1}^T}{\partial u_z} + e_t\right|$.

Compare the gradients on hidden weight: $\frac{\partial h_t^T}{\partial w_{hh}}$ with $\frac{\partial h_t^T}{\partial w_z}$, and gradients on input weight: $\frac{\partial h_t^T}{\partial w_{hx}}$ with $\frac{\partial h_t^T}{\partial u_z}$. We see that the GRU gradients are either smaller or equal to the RNN gradients.

Gradients with respect to the remaining parameters $\frac{\partial h_t^T}{\partial w_h}, \frac{\partial h_t^T}{\partial u_h}, \frac{\partial h_t^T}{\partial w_r}, \frac{\partial h_t^T}{\partial u_r}$ can be computed in a similar manner. The detailed computations are provided in the Appendix section A.2.2. What can be shown is that, the authors' comparison conclusion still applies under the standard parameterisation and for all weights.

We will justify how results of the indicator used in the experiments are related to the arguments made in Rehmer & Kroll (2020). The stepwise hidden state gradients with respect to parameter $\frac{\partial h_t^T}{\partial \text{vec}(W^T)}$, and the stepwise indicator $\frac{\partial h_t^T}{\partial h_0}$ are closely-related quantities. Our indicator is similar to the longest product in each stepwise hidden state gradient, with the first multiplier $\frac{\partial h_1^T}{\partial \text{vec}(W^T)}$ replaced by $\frac{\partial h_1^T}{\partial h_0}$. See equation (4.7) for the explicit form. We claim that each $\frac{\partial h_t^T}{\partial h_0}$ is the part most subject to the vanishing effect in $\frac{\partial h_t^T}{\partial \text{vec}(W^T)}$, because the indicator captures the longest product over steps. Therefore the authors' conclusion do support our experimental results that GRU and LSTM VGP-indicator are smaller or equal to the value of SRNN.

5.2.2 Comparing RNN and LSTM gradients

We extend the comparison in Rehmer & Kroll (2020) to the LSTM, by applying the same comparative computations to SRNN and LSTM hidden state gradients. Since the cell states of LSTM are memory holder like the hidden states in SRNN and GRU, we compare the LSTM cell state gradients $\frac{\partial c_t^T}{\partial w}$ with the SRNN hidden state gradients $\frac{\partial h_t^T}{\partial w}$. In total we need to compute the gradients $\frac{\partial c_t^T}{\partial w_i}, \frac{\partial c_t^T}{\partial u_i}, \frac{\partial c_t^T}{\partial w_f}, \frac{\partial c_t^T}{\partial u_f}, \frac{\partial c_t^T}{\partial w_c}, \frac{\partial c_t^T}{\partial u_c}, \frac{\partial c_t^T}{\partial w_o}, \frac{\partial c_t^T}{\partial u_o}$ and compare with equation (5.1) or (5.2).

However in the particular case where all 3 gates of LSTM are active we cannot show that the LSTM gradients are smaller.

LSTM cell state gradients with respect to the input gate.

Derivative of h_{t-1} with respect to any weight w or u:

$$\frac{\partial h_{t-1}^T}{\partial w} = o_{t-1}(1 - \tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w}$$

Differentiate c_t with respect to the hidden input weight w_i.

$$\begin{aligned}
\frac{\partial c_t^T}{\partial w_i} &= \frac{\partial f_t^T}{\partial w_i} c_{t-1} + f_t \frac{\partial c_{t-1}^T}{\partial w_i} + \frac{\partial i_t^T}{\partial w_i}\tilde{c}_t + i_t \frac{\partial \tilde{c}_t^T}{\partial w_i} \\
&= f_t(1-f_t)w_f o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_i} c_{t-1} + f_t \frac{\partial c_{t-1}^T}{\partial w_i} \\
&+ i_t(1-i_t)(h_{t-1} + w_i o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_i})\tilde{c}_t \\
&+ i_t(1-\tilde{c}_t^2)w_c o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_i}) \qquad (5.5)
\end{aligned}$$

When $f_t = 0, i_t = 0$, $\frac{\partial c_t^T}{\partial w_i} = 0$.

When $f_t = 0, i_t = 0, o_{t-1} = 0$, $\frac{\partial c_t^T}{\partial w_i} = 0$.

When $f_t = 0, i_t = 0, o_{t-1} = 1$, $\left|\frac{\partial c_t^T}{\partial w_i}\right| \leq \left|w_c \frac{\partial c_{t-1}^T}{\partial w_i}\right|$.

Comparing with (5.1), under the case where the forget gate f is closed we can show that the

LSTM cell state gradients are smaller or equal to those of SRNN.

When $f_t = 1, i_t = 0$, $\frac{\partial c_t^T}{\partial w_i} = \frac{\partial c_{t-1}^T}{\partial w_i}$.

When $f_t = 1, i_t = 1, o_{t-1} = 0$, $\frac{\partial c_t^T}{\partial w_i} = \frac{\partial c_{t-1}^T}{\partial w_i}$.

When $f_t = 1, i_t = 1, o_{t-1} = 1$, $\left|\frac{\partial c_t^T}{\partial w_i}\right| \leq \left|\frac{\partial c_{t-1}^T}{\partial w_i} + w_c \frac{\partial c_{t-1}^T}{\partial w_i}\right|$.

In the above 3 cases, the forget gate is open, c_{t-1} is allowed to be passed into the next state. In the last case where all 3 gates are open, the upper bound of $\frac{\partial c_t^T}{\partial w_i}$ cannot be shown to be smaller or equal to the upper bound of SRNN hidden state gradient (5.1).

Similarly differentiate c_t with respect to the input gate weight associated with the inputs u_i:

$$\begin{aligned}\frac{\partial c_t^T}{\partial u_i} &= f_t(1-f_t)w_f o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial u_i}c_{t-1} + f_t\frac{\partial c_{t-1}^T}{\partial u_i} \\ &+ i_t(1-i_t)(w_i o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial u_i} + e_t)\tilde{c}_t \\ &+ i_t(1-\tilde{c}_t^2)w_c o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial u_i}) \end{aligned} \tag{5.6}$$

When $f_t = 0, i_t = 0$, $\frac{\partial c_t^T}{\partial u_i} = 0$.

When $f_t = 0, i_t = 0, o_{t-1} = 0$, $\frac{\partial c_t^T}{\partial u_i} = 0$.

When $f_t = 0, i_t = 0, o_{t-1} = 1$, $\left|\frac{\partial c_t^T}{\partial u_i}\right| \leq \left|w_c \frac{\partial c_{t-1}^T}{\partial u_i}\right|$.

When $f_t = 1, i_t = 0$, $\frac{\partial c_t^T}{\partial u_i} = \frac{\partial c_{t-1}^T}{\partial u_i}$.

When $f_t = 1, i_t = 1, o_{t-1} = 0$, $\frac{\partial c_t^T}{\partial u_i} = \frac{\partial c_{t-1}^T}{\partial u_i}$.

When $f_t = 1, i_t = 1, o_{t-1} = 1$, $\left|\frac{\partial c_t^T}{\partial u_i}\right| \leq \left|\frac{\partial c_{t-1}^T}{\partial u_i} + w_c \frac{\partial c_{t-1}^T}{\partial u_i}\right|$.

Again in the last case the upper bound is not always smaller than that of SRNN.

The LSTM cell state gradients with respect to the forget gate weights have very similar forms and comparison conclusions. The details will not be included here.

LSTM cell state gradients with respect to the candidate gate

Differentiate c_t with respect to the hidden candidate weight w_c.

$$\begin{aligned}\frac{\partial c_t^T}{\partial w_c} &= f_t(1-f_t)w_f o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_c}c_{t-1} + f_t\frac{\partial c_{t-1}^T}{\partial w_c} \\ &+ i_t(1-i_t)w_i o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_c}\tilde{c}_t \\ &+ i_t(1-\tilde{c}_t^2)(h_{t-1} + w_c o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_c}) \end{aligned} \tag{5.7}$$

Apply the same assumptions and comparison, the particular case where the LSTM gradient upper bound is actually larger than that of SRNN is when all 3 gates are open.
$\left|\frac{\partial c_t^T}{\partial w_c}\right| \leq \left|\frac{\partial c_{t-1}^T}{\partial w_c} + h_{t-1} + w_c\frac{\partial c_{t-1}^T}{\partial w_c}\right|$.
Differentiate c_t with respect to the input candidate weight u_c.

$$\begin{aligned}\frac{\partial c_t^T}{\partial u_c} &= f_t(1-f_t)w_f o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial u_c}c_{t-1} + f_t\frac{\partial c_{t-1}^T}{\partial u_c} \\ &+ i_t(1-i_t)w_i o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial u_c}\tilde{c}_t \\ &+ i_t(1-\tilde{c}_t^2)(w_c o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial u_c} + e_t) \end{aligned} \tag{5.8}$$

When f_t, i_t, o_{t-1} are all open, $\left|\frac{\partial c_t^T}{\partial u_c}\right| \leq \left|\frac{\partial c_{t-1}^T}{\partial u_c} + w_c\frac{\partial c_{t-1}^T}{\partial u_c} + e_t\right|$ is the only case the upper bound is larger than that in (5.2).

LSTM cell state gradients with respect to the output gate

Note that in computing $\frac{\partial h_{t-1}^T}{\partial w_o}$ and $\frac{\partial h_{t-1}^T}{\partial u_o}$, we do not differentiate o_{t-1} with respect to w_o or u_o because gradients at t are expressed only in terms of quantities at steps $t-1, t$, and should

not involve $t-2$. Differentiate c_t with respect to the hidden output weight w_o.

$$\begin{aligned}\frac{\partial c_t^T}{\partial w_o} &= f_t(1-f_t)w_f o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_o}c_{t-1} + f_t\frac{\partial c_{t-1}^T}{\partial w_o} \\ &+ i_t(1-i_t)w_i o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_o}\tilde{c}_t \\ &+ i_t(1-\tilde{c}_t^2)w_c o_{t-1}(1-\tanh^2(c_{t-1}))\frac{\partial c_{t-1}^T}{\partial w_o} \end{aligned} \tag{5.9}$$

When all 3 gates are open, $\left|\frac{\partial c_t^T}{\partial w_o}\right| \leq \left|\frac{\partial c_{t-1}^T}{\partial w_o} + w_c\frac{\partial c_{t-1}^T}{\partial w_o}\right|$.

Similarly for the input weight in the output gate: $\left|\frac{\partial c_t^T}{\partial u_o}\right| \leq \left|\frac{\partial c_{t-1}^T}{\partial u_o} + w_c\frac{\partial c_{t-1}^T}{\partial u_o}\right|$.

We cannot claim whether these upper bounds are smaller than (5.1) and (5.2).

6 Conclusion

The main objective of the thesis is to discuss the behaviour, the causes and impacts of the VGP in training FNNs and RNNs. We have given clear definition of the VGP:

- **In FNNs:** Numerical vanishing trend of loss gradients with respect to the lower-layer weights.
- **In RNNs:** Numerical vanishing trend of those components within the loss gradients that reflect longer temporal dependencies.

There exist other miscellaneous cases of zero-gradients due to a variety of causes, for example the dying ReLU problem when training FNNs and the GRU having the reset gate being zero at some steps. The thesis is concerned only with the numerical vanishing gradient case.

6.1 Summary of contributions

For each type of the networks we have analysed the sources of the vanishing gradients and the effectiveness of existing solutions. This is achieved by understanding the back propagation computations and thus identifying the diagnostic statistics directly affected by the VGP. The diagnostic statistics can reflect the behaviour of the gradients over time for one particular combination of hyperparameters, as well as for comparing models with different parameterisations.

For FNNs we have identified 3 aspects of model fitting to watch out for: predictive performance; the magnitude of gradients; and the behaviour of the per-layer outputs (in particular, the tendency to saturate). The diagnostic statistics to compute and visualise these aspects are respectively: the training loss and accuracy, the output gradients with respect to the weights,

and the per-layer hidden state outputs. We have demonstrated the relative improvement in the VGP between an FNN parameterised with sigmoid activation and the default Glorot uniform initialisation, and one with PReLU activation, and optimised weights initialisation for each layer. Although details are not provided, we would like to mention that hyperparameter optimisation has been conducted in order to determine the best hyperparameter combination for the model fitted in section 3.4.

For RNNs, the thesis focuses on the NLP task of sequence classification. We have provided details of rigourous computation of the loss gradients. Based on the explicit form, we have identified and produced code for a strict indicator which reflects the extent of the VGP. This indicator originally used by Noh (2021) on a regression problem with time-series data. On LSTM, this indicator has been modified for the different architecture.
Experiments have been conducted to verify the mainstream viewpoint that LSTM and GRU alleviate the VGP in SRNN. In particular, we have compared their predictive performances, and the indicator behaviours. When the VGP indicator is compared we notice that it has the largest values for SRNN, rather than for LSTM and GRU. The results seemingly contradicting the mainstream viewpoint are nonetheless supported by Rehmer & Kroll (2020). We have extended their comparisons for SRNN and LSTM.

6.2 Future research directions

For the VGP in FNNs, one could extend our research by involving regularisation. Apart from introducing penalty terms there exist structural regularisation techniques such as early stopping of training (Sjöberg & Ljung, 1995), and dropout of a proportion of hidden units (Srivastava et al., 2014). In practice one needs to conduct a hyperparameter optimisation to determine the option that results in the best training or validation performance. The diagnostics in the thesis will also be helpful in visualising the effectiveness of regularisation in reducing the VGP.

For the VGP in RNNs, Rehmer & Kroll (2020) have compared RNN with the GRU only, but have implicitly suggested that the LSTM belongs to the class of gated models and will experience the VGP as well. Experimentally, we have the VGP indicator for the LSTM smaller than that of SRNN. However we cannot show in closed form that the LSTM state gradients are consistently smaller than those of the RNN, in section 5.2.2. The key reason lies in the LSTM cell state equation (2.26) and the GRU hidden state equation (2.31). In (2.31) the multipliers are $z_t, 1 - z_t$. Therefore as $z_t = 0, 1$, either one of the multipliers will be zero. The multipliers in LSTM (2.26) correspond to different gates therefore will always involve one more term in the state gradients (5.5) - (5.9).

On the other hand, we believe that the LSTM does not have to maintain a smaller state gradient for *every* combination of gates. Rather than reducing vanishing gradients in every scenario, the LSTM learns the data more appropriately and reduces the VGP where *needed*. Therefore it achieves superior classification performance in our experiments. It would be valuable if future research can draw concrete conclusions on this issue through derivations and experiments.

Lastly, it is generally acknowledged that the LSTM and GRU cannot completely avoid the VGP on extremely long sequences. The Transformer model (Vaswani et al, 2017) exploits the linguistic relevance in text for prediction. It avoids processing sequences in steps, therefore does not suffer from the VGP. The structural difference between the Transformer model and RNNs can be studied as a topic of future research.

A Appendix

A.1 FNN

A.1.1 Dying-ReLU illustration

Figures A.1 to A.3 demonstrate the Dying-ReLU problem mentioned in section 3.2.1, with diagnostics from an FNN parameterised with standard ReLU activation function (2.3). Output gradients and hidden layer outputs are close to zero. The loss and accuracy are therefore unchanged over the training.

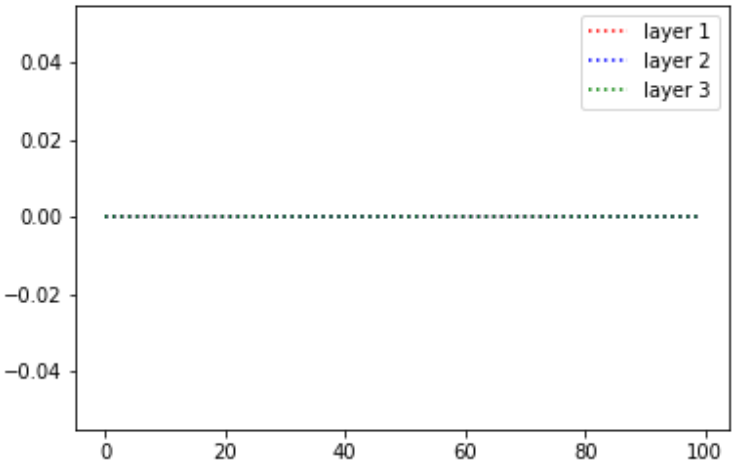

Figure A.1: Output gradients for ReLU FNN

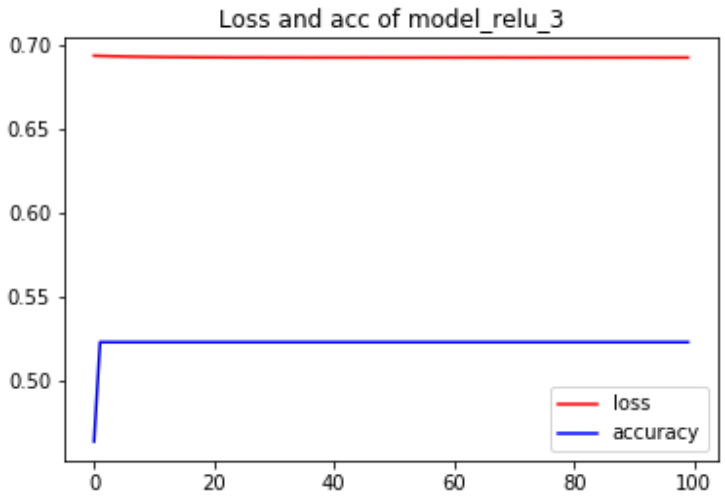

Figure A.2: Loss and accuracy for ReLU FNN

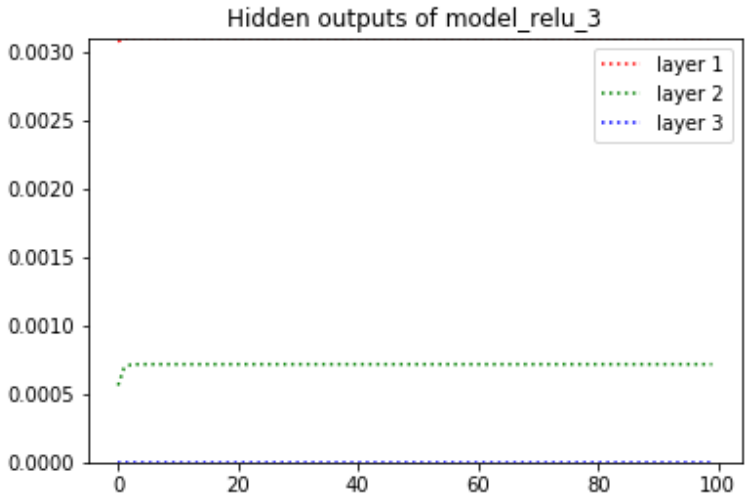

Figure A.3: Hidden layer outputs for ReLU FNN

A.2 RNN

A.2.1 Bidirectional structure

A bidirectional model stacks 2 RNN-class layers, which process the input sequences in opposite directions. In practice any of the standard RNN, LSTM, or GRU cell can be used in the bidirectional structure.

Take the standard RNN layer for example, the hidden states aggregated for both layers are combined at each step, via element-wise summation, multiplication or averaging. The Tensorflow library has a specific argument for specifying the aggregation instruction in the Bidirectional model-building functions. For consistency we will assume summation in this thesis and software experiments.

The forward propagation equations of a Bidirectional RNN (BRNN) are similar to those of the standard RNN:

$$
\begin{aligned}
h_t^f &= \text{tanh}(U_f^T e_t + W_f^T h_{t-1}^f) \\
h_t^b &= \text{tanh}(U_b^T e_t + W_b^T h_{t+1}^b) \\
h_t &= h_t^f + h_t^b \\
\hat{y}_t &= \text{softmax}(W_{yh}^T) h_t
\end{aligned}
$$

The forward and backward layers of RNN are parameterised independently with the input and hidden weights, which are distinguished with subscripts f, b. If the forward and backward layer hidden states are summed, BPTT equations for loss gradients with respect to a particular weight are the same as in the standard RNN BPTT.

Similarly a Bidirectional LSTM, BLSTM (Graves & Schmidhuber, 2005) computes cell states of both directions c_t^f, c_t^b and thus the hidden states h_t^f, h_t^b to be aggregated, by creating a backward copy of LSTM forward propagation equations.

A Bidirectional GRU (BGRU) creates a backward copy of equations, and aggregates the hidden

states of both directions.

At each step the hidden state contains information to the left (of the past) and to the right (of the future) of the current step input. Therefore in problems such as predicting a word in the middle of a sentence, the bidirectional structure offers greater predictive power.
It is less intuitive whether this characteristic of the bidirectional models also improves the performance in the task of sequence classification, whereby the entire input sequences are processed before any output or loss is computed. We shall compare on the original toxic-comment dataset the loss and accuracy of each pair of models: {RNN-BRNN, LSTM-BLSTM, GRU-BGRU}.

We have picked and shuffled a training set containing 1000 toxic and 1000 non-toxic comments. Each word in an input sequence is represented by its GloVe 6B-100d embedding vector. The bidirectional models have the same construction as the uni-directional models, except that the RNN layer is wrapped by a `keras.Bidirectional` layer. For example a BGRU layer is constructed as

```
Bidirectional(RNN(GRUCell(200)), merge_mode = 'sum' )
```

All the pairs have training and validation losses plotted together; training and validation accuracies together in a separate plot.

Both the standard RNN and BRNN models are trained for 80 epochs.

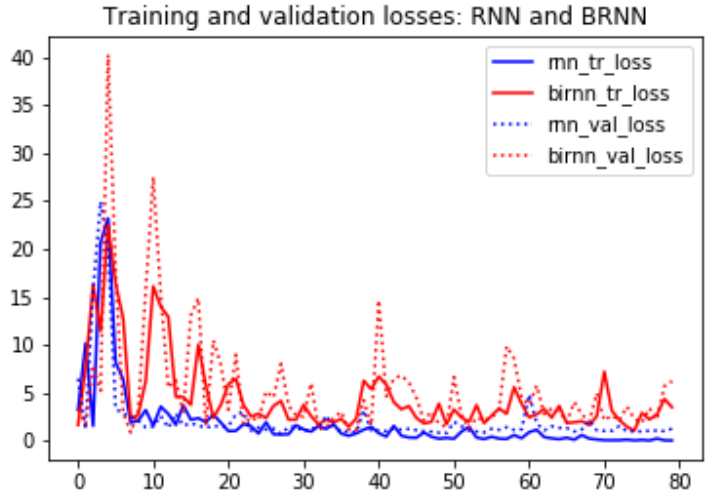

Figure A.4

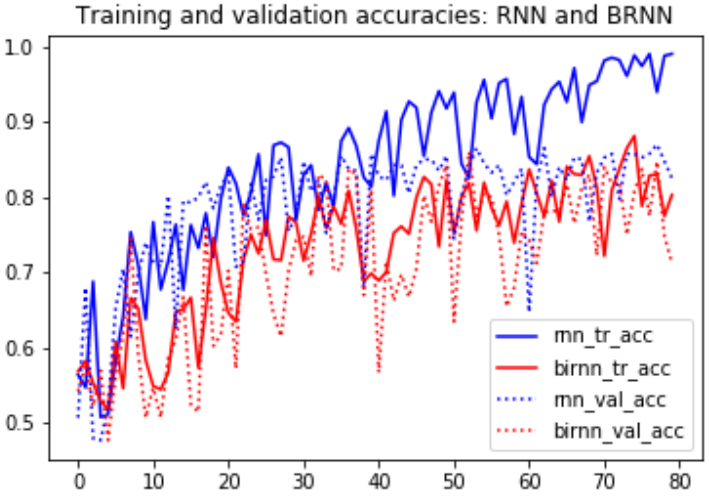

Figure A.5

In the long term BRNN is observed to generally have higher losses and lower accuracies.

Both the LSTM and BLSTM are trained for 50 epochs. Each LSTM cell computes 4 gates and 1 candidate state for all time steps, in contrast to RNN cell having only 1 computation of the hidden state. Therefore LSTM models are more expensive to fit and thus the reduction in the number of epochs.

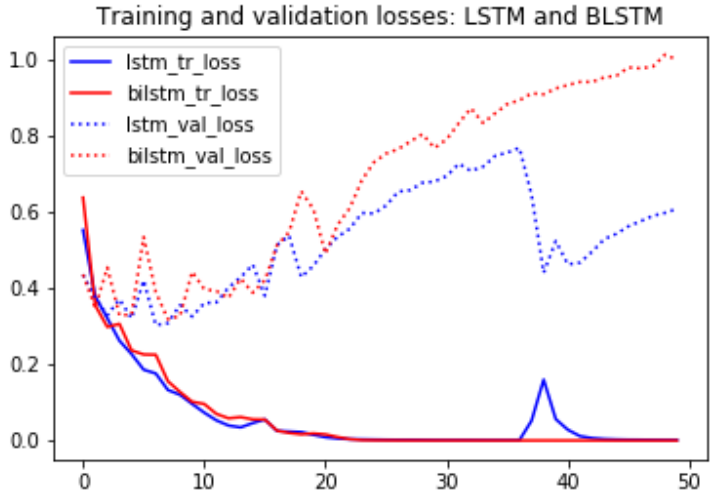

Figure A.6

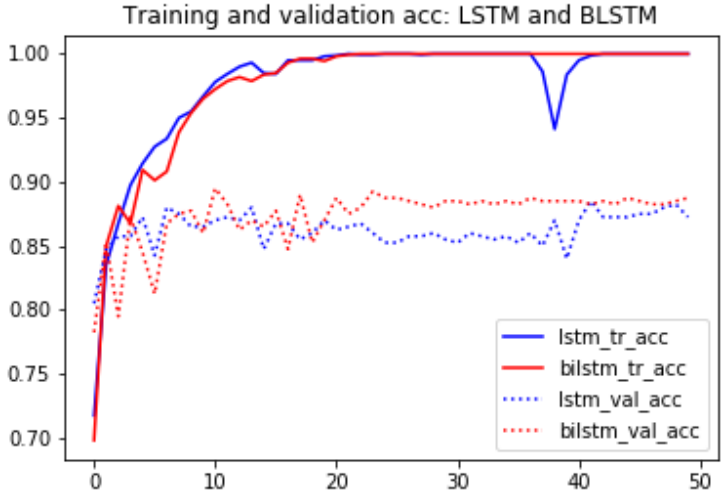

Figure A.7

BLSTM generally exhibited higher losses. BLSTM has better validation accuracy in the long term. However no decisive advantage.

GRU and BGRU are also trained for 50 epochs.

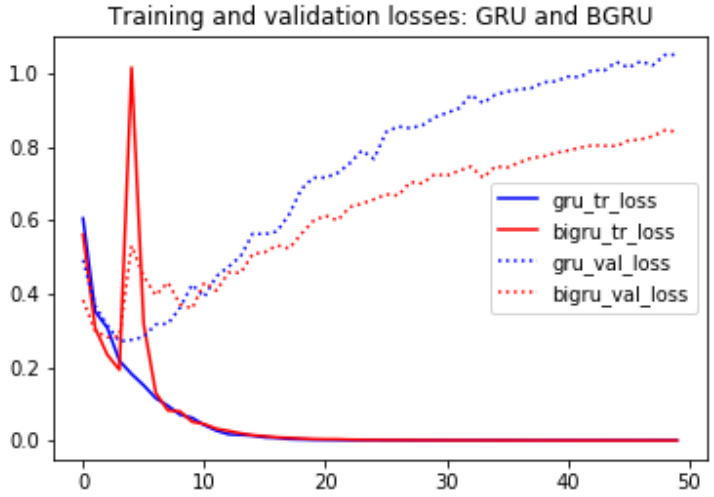

Figure A.8

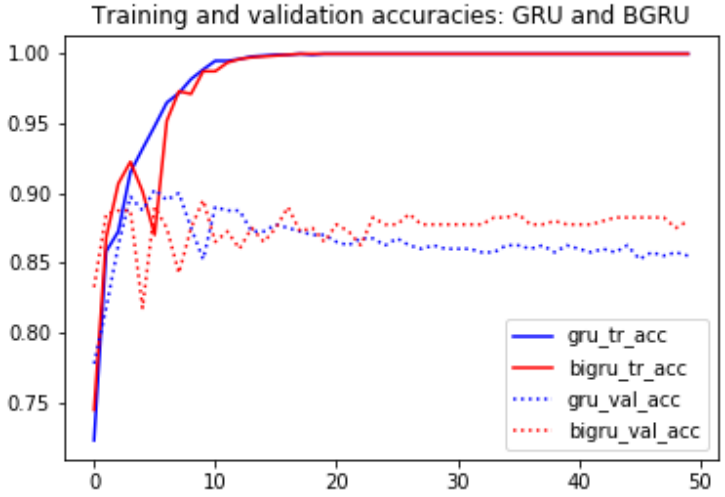

Figure A.9

In the long term BGRU had lower losses and higher accuracies. However it is outperformed within 20 epochs.

To further verify our hypothesis that bidirectional structure does not significantly help with the sequence classification tasks, we have applied the 3 pairs of models to a different dataset. The Spooky Author Identification data is taken from the Kaggle competition in 2018. Each sequence in the data is a sentence taken from text, labelled with its author index. In total it contains 3 author indices: {EAP: Edgar Allan Poe; HPL: HP Lovecraft; MWS: Mary Shelley}.

It is not within the scope of this thesis to explain precisely the linguistic differences between these 2 text datasets. However an intuitive observation is that the Spooky Author dataset is more demanding of the classifiers. The comment text sentences are classified by detecting 'toxic' words. The toxicity is mutually exclusive for each word. Sequences all adapted from horror novels on the contrary are more subtle and therefore difficult for the model to classify the corresponding authors.

A subset of the dataset of 5000 sequences is used as the training set, containing 2005 sequences labeled EAP, 1407 labeled HPL and 1588 labeled MWS. The same experiment is conducted.

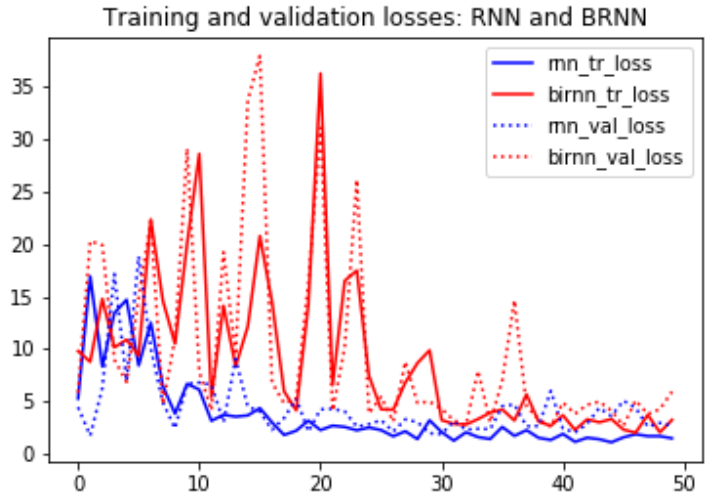

Figure A.10

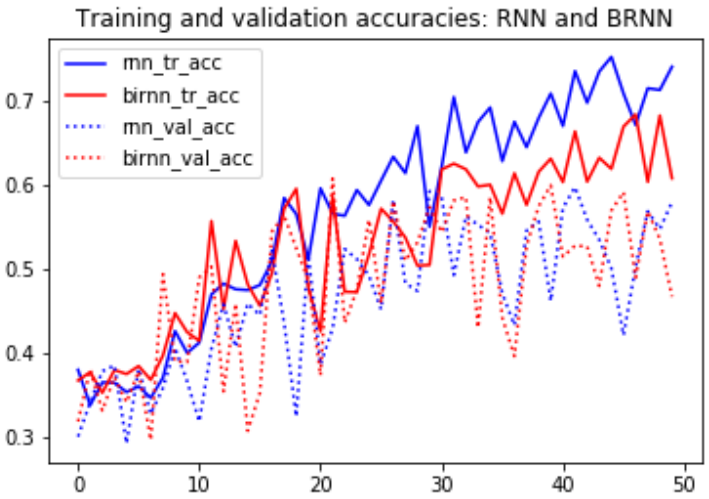

Figure A.11

In the long term RNN outperformed BRNN. However the losses and accuracies of either model are poor, compared to LSTM and GRU.

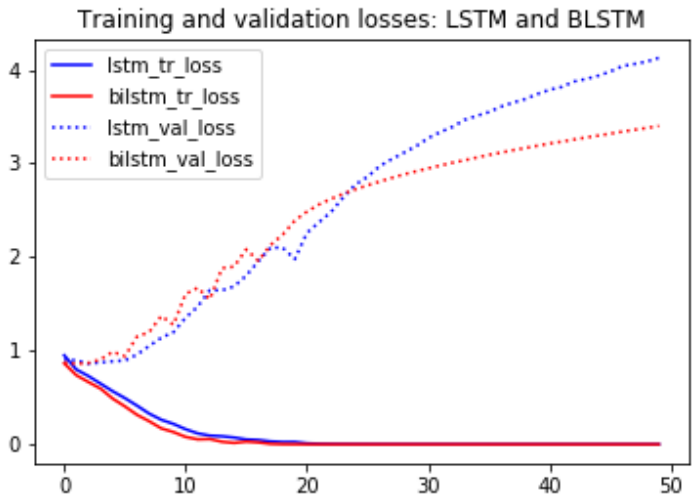

Figure A.12

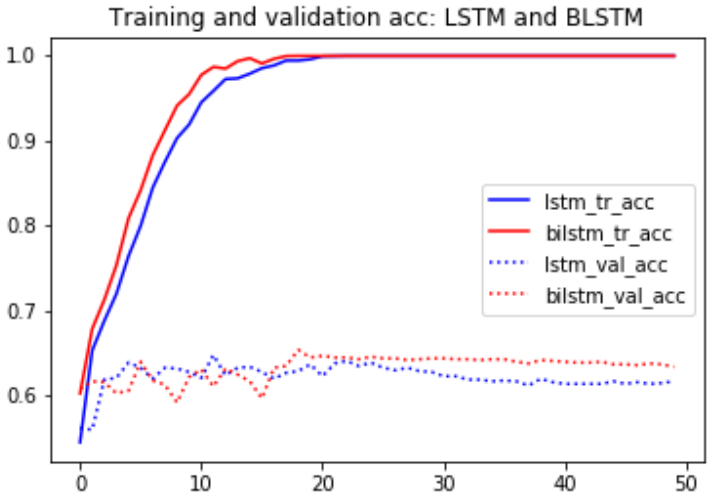

Figure A.13

LSTM and BLSTM exhibited similar losses but BLSTM outperformed in the long term. BLSTM also outperformed in accuracies in general.

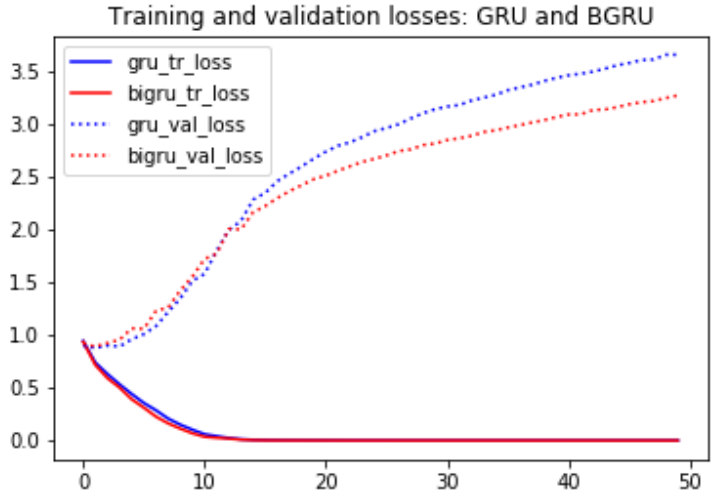

Figure A.14

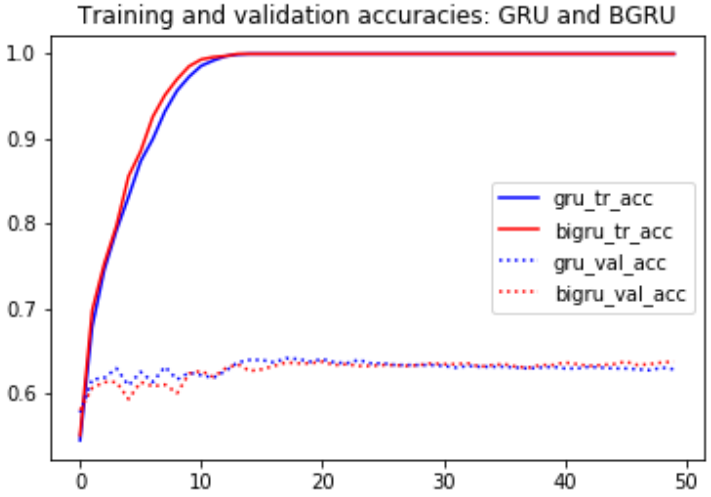

Figure A.15

BGRU was consistently better, although not significantly.

Experimentally we have shown that in the task of sequence classification, the bidirectional models do not significantly improve the predictive power of their uni-directional counterparts. Therefore results of bidirectional structure are included in the appendix.

Theoretically, the additional backward layer is parameterised independently as the forward layer. In tasks where the whole sequences are available, there will be additional loss gradients corresponding to the additional parameters. However the form of each loss gradient is not changed.

A.2.2 Hidden state gradients comparison (GRU vs SRNN)

Computations for $\frac{\partial h_t^T}{\partial w_h}, \frac{\partial h_t^T}{\partial u_h}, \frac{\partial h_t^T}{\partial w_r}, \frac{\partial h_t^T}{\partial u_r}$ are provided in this section. Differentiate the GRU hidden state h_t with respect to the input update weight U_z:

$$\begin{aligned}\frac{\partial h_t^T}{\partial u_z} &= z_t(1-z_t)(w_z\frac{\partial h_{t-1}^T}{\partial u_z}+e_t)h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial u_z} - z_t(1-z_t)(w_z\frac{\partial h_{t-1}^T}{\partial u_z}+e_t)\tilde{h}_t \\ &+ (1-z_t)(1-\tilde{h}_t^2)\left(w_h(r_t(1-r_t)w_r\frac{\partial h_{t-1}^T}{\partial u_z}h_{t-1} + r_t\frac{\partial h_{t-1}^T}{\partial u_z}) + e_t\right)\end{aligned}$$

When $z_t = 0, r_t = 0$,

$$\frac{\partial h_t^T}{\partial u_z} = (1-\tilde{h}_t^2)e_t \leq e_t$$

When $z_t = 0, r_t = 1$,

$$\begin{aligned}\frac{\partial h_t^T}{\partial u_z} &= (1-\tilde{h}_t^2)(w_h\frac{\partial h_{t-1}^T}{\partial u_z}+e_t) \\ &\leq \frac{\partial h_{t-1}^T}{\partial u_z}+e_t\end{aligned}$$

Compare the gradients on hidden weight: $\frac{\partial h_t^T}{\partial w_{hh}}$ with $\frac{\partial h_t^T}{\partial w_z}$, and gradients on input weight: $\frac{\partial h_t^T}{\partial w_{hx}}$ with $\frac{\partial h_t^T}{\partial u_z}$. We see that the GRU gradients are either smaller or equal to the RNN gradients.

GRU hidden state gradients with respect to the candidate state.

Differentiate the GRU hidden state h_t with respect to the hidden candidate weight W_h:

$$\begin{aligned}\frac{\partial h_t^T}{\partial w_h} &= z_t(1-z_t)w_z\frac{\partial h_{t-1}^T}{\partial w_h}h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial w_h} - z_t(1-z_t)w_z\frac{\partial h_{t-1}^T}{\partial w_h}\tilde{h}_t \\ &\quad + (1-z_t)(1-\tilde{h}_t^2)w_h\left(r_t(1-r_t)w_r\frac{\partial h_{t-1}^T}{\partial w_h}h_{t-1} + r_t\frac{\partial h_{t-1}^T}{\partial w_h}\right)\end{aligned}$$

When $z_t = 0, r_t = 0$,

$$\frac{\partial h_t^T}{\partial w_h} = 0$$

When $z_t = 0, r_t = 1$,

$$\begin{aligned}\frac{\partial h_t^T}{\partial w_h} &= (1-\tilde{h}_t^2)w_h\frac{\partial h_{t-1}^T}{\partial w_h} \\ &\leq \frac{\partial h_{t-1}^T}{\partial w_h}\end{aligned}$$

Differentiate the GRU hidden state h_t with respect to the input candidate weight U_h:

$$\begin{aligned}\frac{\partial h_t^T}{\partial u_h} &= z_t(1-z_t)w_z\frac{\partial h_{t-1}^T}{\partial u_h}h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial u_h} - z_t(1-z_t)w_z\frac{\partial h_{t-1}^T}{\partial u_h}\tilde{h}_t \\ &\quad + (1-z_t)(1-\tilde{h}_t^2)\left(w_h(r_t(1-r_t)w_r\frac{\partial h_{t-1}^T}{\partial u_h}h_{t-1} + r_t\frac{\partial h_{t-1}^T}{\partial u_h}) + e_t\right)\end{aligned}$$

When $z_t = 0, r_t = 0$,

$$\frac{\partial h_t^T}{\partial u_h} = (1-\tilde{h}_t^2)e_t \leq e_t$$

When $z_t = 0, r_t = 1$,

$$\begin{aligned}\frac{\partial h_t^T}{\partial u_h} &= (1 - \tilde{h}_t^2)(w_h \frac{\partial h_{t-1}^T}{\partial u_h} + e_t) \\ &\leq \frac{\partial h_{t-1}^T}{\partial u_h} + e_t\end{aligned}$$

Again the RNN gradients are greater or equal to the GRU gradients.

GRU hidden state gradients with respect to the reset gate.

Differentiate the GRU hidden state h_t with respect to the hidden reset weight W_r:

$$\begin{aligned}\frac{\partial h_t^T}{\partial w_r} &= z_t(1 - z_t)w_z \frac{\partial h_{t-1}^T}{\partial w_r} h_{t-1} + z_t \frac{\partial h_{t-1}^T}{\partial w_r} - z_t(1 - z_t)w_z \frac{\partial h_{t-1}^T}{\partial r_h} \tilde{h}_t \\ &+ (1 - z_t)(1 - \tilde{h}_t^2)w_h \left(r_t(1 - r_t)w_r \frac{\partial h_{t-1}^T}{\partial w_r} h_{t-1} + r_t \frac{\partial h_{t-1}^T}{\partial r_h} \right)\end{aligned}$$

When $z_t = 0, r_t = 0$,

$$\frac{\partial h_t^T}{\partial w_h} = 0$$

When $z_t = 0, r_t = 1$,

$$\begin{aligned}\frac{\partial h_t^T}{\partial w_h} &= (1 - \tilde{h}_t^2)w_h \frac{\partial h_{t-1}^T}{\partial w_r} \\ &\leq \frac{\partial h_{t-1}^T}{\partial w_r}\end{aligned}$$

Differentiate the GRU hidden state h_t with respect to the input reset weight U_r:

$$\begin{aligned}\frac{\partial h_t^T}{\partial u_r} &= z_t(1-z_t)w_z\frac{\partial h_{t-1}^T}{\partial u_r}h_{t-1} + z_t\frac{\partial h_{t-1}^T}{\partial u_r} - z_t(1-z_t)w_z\frac{\partial h_{t-1}^T}{\partial u_r}\tilde{h}_t \\ &\quad + (1-z_t)(1-\tilde{h}_t^2)w_h\left(r_t(1-r_t)w_r\frac{\partial h_{t-1}^T}{\partial u_r}h_{t-1} + r_t\frac{\partial h_{t-1}^T}{\partial u_r}\right)\end{aligned}$$

When $z_t = 0, r_t = 0$,

$$\frac{\partial h_t^T}{\partial u_h} = 0$$

When $z_t = 0, r_t = 1$,

$$\begin{aligned}\frac{\partial h_t^T}{\partial u_h} &= (1-\tilde{h}_t^2)w_h\frac{\partial h_{t-1}^T}{\partial u_r} \\ &\leq \frac{\partial h_{t-1}^T}{\partial u_r}\end{aligned}$$

Bibliography

[1] Bengio, Y., Simard, P., Frasconi, P., 1994. *Learning Long-Term Dependencies with Gradient Descent is Difficult*. IEEE Transactions on Neural Networks 5, 157–166. doi:10.1109/72.279181

[2] Bishop, C.M., 2014. Bishop - *Pattern Recognition And Machine Learning* - Springer 2006. Antimicrobial agents and chemotherapy 58, 7250–7.

[3] Bottou, L. *"Online Learning and Stochastic Approximations."* (1998).

[4] Calin O (2020) Activation functions. In: *Deep learning architectures*. Springer Series in the Data Sciences. Springer, Cham. https://doi.org/10.1007/978-3-030-36721-3-2

[5] Cho, K., Van Merriënboer, B., ... Bengio, Y., 2014. *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, in: EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference. Association for Computational Linguistics (ACL), pp. 1724–1734. doi:10.3115/v1/d14-1179

[6] Chou, C., Shie, C., Chang, F., Chang, J., & Chang, E. (2019). *Representation Learning on Large and Small Data*. ArXiv, abs/1707.09873.

[7] Chollet, F., 2016, *Using pre-trained word embeddings in a Keras model*, ¡https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html¿

[8] Clevert, D.A., Unterthiner, T., Hochreiter, S., 2016. *Fast and accurate deep network learning by exponential linear units (ELUs)*, in: 4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings. International Conference on Learning Representations, ICLR.

[9] Dubowski, A.P. (2020). *Activation function impact on Sparse Neural Networks*. ArXiv, abs/2010.05943.

[10] Dumais, S.T., Furnas, G.W., ... Harshman, R., 1988. *Using latent semantic analysis to improve access to textual information, in: Conference on Human Factors in Computing Systems* - Proceedings. Association for Computing Machinery, pp. 281–285. doi:10.1145/57167.57214

[11] Glorot, X., Bengio, Y., 2010. *Understanding the difficulty of training deep feedforward neural networks*, in: Journal of Machine Learning Research. pp. 249–256.

[12] Goodfellow, I.; Bengio, Y. & Courville, A. (2016), *Deep Learning* , MIT Press .

[13] Graves, A., 2013. *Supervised Sequence Labeling with Recurrent Neural Networks*, arXiv preprint arXiv:1308.0850. doi:10.1145/2661829.2661935

[14] Guerzhoy, Michael. *Learning Long-Term Dependencies with RNN* cs.toronto.edu/guerzhoy/321/lec/W09

[15] gunes (https://stats.stackexchange.com/users/204068/gunes), *Backpropagation through time: dimensions in the chain rule*, URL (version: 2022-02-26): https://stats.stackexchange.com/q/565903
Question asked by myself. Question and answer involve some degree of relevance to section 4.1.1: Gradients with respect to vectorised matrices, however limitedly helpful.

[16] Hayou, S., Doucet, A., Rousseau, J., 2019. *On the impact of the activation function on deep neural networks training*, in: 36th International Conference on Machine Learning, ICML 2019. International Machine Learning Society (IMLS), pp. 4746–4754.

[17] He, K., Zhang, X., ... Sun, J., 2015. *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in: Proceedings of the IEEE International Conference on Computer Vision. Institute of Electrical and Electronics Engineers Inc., pp. 1026–1034. doi:10.1109/ICCV.2015.123

[18] Heaton, J. 2008. *Introduction to neural networks with Java*. Heaton Research, Inc.

[19] Hochreiter, S., Schmidhuber, J.; *Long Short-Term Memory*. Neural Comput 1997; 9 (8): 1735–1780. doi: https://doi.org/10.1162/neco.1997.9.8.1735

[20] Huang, L. et al. (2020) *"Normalization Techniques in Training DNNs: Methodology, Analysis and Application."* arXiv. doi:10.48550/ARXIV.2009.12836.

[21] Ioffe, S., Szegedy, C., 2015. *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in: 32nd International Conference on Machine Learning, ICML 2015. International Machine Learning Society (IMLS), pp. 448–456.

[22] Jaeger, H., Haas, H., 2004. *Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication*. Science 304, 78–80. doi:10.1126/science.1091277

[23] Jozefowicz, R., Zaremba, W., Sutskever, I., 2015. *An empirical exploration of Recurrent Network architectures*, in: 32nd International Conference on Machine Learning, ICML 2015. International Machine Learning Society (IMLS), pp. 2332–2340.

[24] Learned-Miller, E., 2022. *Vector, Matrix, and Tensor Derivatives*. [online] Cs231n.stanford.edu. Available at: ¡http://cs231n.stanford.edu/vecDerivs.pdf¿.

[25] LeCun, Y. et al., 1998. *Efficient BackProp. Neural Networks: Tricks of the Trade*, pp.9–50. Available at: http://dx.doi.org/10.1007/3-540-49430-8-2.

[26] Li, M., 2014. *Efficient mini-batch training for stochastic optimization*, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, pp. 661–670. doi:10.1145/2623330.2623612

[27] Li, Y., Fan, C., ... Ming, Y., 2018. *Improving deep neural network with Multiple Parametric Exponential Linear Units*. Neurocomputing 301, 11–24. doi:10.1016/j.neucom.2018.01.084

[28] Maas, A.L., Hannun, A.Y., Ng, A.Y., 2013. *Rectifier nonlinearities improve neural network acoustic models*, in: In ICML Workshop on Deep Learning for Audio, Speech and Language Processing.

[29] J. R. Magnus, *Linear structures*, Oxford, Oxford University Press, 1988. ISBN 0-19-520655-X, cloth, 49.95, pp. xii + 205.

[30] Mikolov, T., Chen, K., ... Dean, J., 2013. *Efficient estimation of word representations in vector space*, in: 1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings. International Conference on Learning Representations, ICLR.

[31] Mikolov, T., Chen, K., ... Dean, J., 2006. *Distributed Representations of Words and Phrases and their Compositionality*. Neural information processing systems 1, 1–9.

[32] Li, Minchen. (2016). *A Tutorial On Backward Propagation Through Time (BPTT) In The Gated Recurrent Unit (GRU) RNN*. 10.13140/RG.2.2.32858.98247.

[33] Noh, S.H., 2021. *Analysis of gradient vanishing of RNNs and performance comparison*. Information (Switzerland) 12. doi:10.3390/info12110442

[34] Pascanu, R., Mikolov, T., Bengio, Y., 2013. *On the difficulty of training recurrent neural networks*, in: 30th International Conference on Machine Learning, ICML 2013. International Machine Learning Society (IMLS), pp. 2347–2355.

[35] Pascanu, R., Mikolov, T., Bengio, Y., 2012. *Understanding the exploding gradient problem*. Proceedings of The 30th International Conference on Machine Learning 1310–1318.

[36] Pennington, J., Socher, R., Manning, C.D., 2014. *GloVe: Global vectors for word representation*, in: EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference. Association for Computational Linguistics (ACL), pp. 1532–1543. doi:10.3115/v1/d14-1162

[37] Rehmer, A., Kroll, A., 2020. *On the vanishing and exploding gradient problem in gated recurrent units*, in: IFAC-PapersOnLine. Elsevier B.V., pp. 1243–1248. doi:10.1016/j.ifacol.2020.12.1342

[38] Schuster, M., Paliwal, K.K., 1997. *Bidirectional recurrent neural networks*. IEEE Transactions on Signal Processing 45, 2673–2681. doi:10.1109/78.650093

[39] Smith, J.W., Everhart, J.E., ... Johannes, R.S., 1988. *Using the ADAP learning algorithm to forecast the onset of diabetes mellitus*, in: Proceedings - Annual Symposium on Computer Applications in Medical Care. Publ by IEEE, pp. 261–265.

[40] Sutskever, I., Martens, J., Hinton, G., 2011. *Generating text with recurrent neural networks*, in: Proceedings of the 28th International Conference on Machine Learning, ICML 2011. pp. 1017–1024.

[41] Vaswani, A., Shazeer, N., ... Polosukhin, I., 2017. *Attention is all you need, in: Advances in Neural Information Processing Systems*. Neural information processing systems foundation, pp. 5999–6009.