Recurrent Neural Networks for Dasher

Yasen Petrov

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

In my work this year I attempt to find an approach to adapting character-level Recurrent-Neural-Network-based (RNN) language models to the writing style of a specific user that is suitable to be used in a text-entry system called Dasher. In Dasher the time to enter a string is roughly proportional to the negative log probability of that string under the language model the system uses, meaning that the higher the probability the model assigns to the string the user wants to enter, the quicker they will be able to enter that string. In the first part of this project I showed that RNN-based models achieve better language modelling performance than the finite-context model used in Dasher. However, unlike the current language model, there is no clear way of adapting the predictions of the RNN language models to the style of whoever is using the system, which is a core part of Dasher's functionality. Adapting neural networks is an active research area and there are many existing approaches, but none of them have been created with the purpose of being used in this specific system, so their suitability needs to be assessed.

I identify three existing methods for adapting RNN models – Dynamic evaluation and Sparse Dynamic evaluation, both proposed by [Krause et al., 2018] and Learning Hidden Unit Contributions (LHUC) [Swietojanski et al., 2016]. I compare how well they adapt to different types of text by looking at how they improve on a model that does not use adaptation and find that Dynamic evaluation is superior to the other two in this respect. I also perform experiments to identify which hyperparameters will be important to tune if these approaches are to be used in Dasher. In doing so, I identify a serious issue with these approaches that needs to be addressed in future work – they can unpredictably destabilize the RNN, effectively destroying its predictive capacity. I study an example of this problem, make hypotheses about its causes and propose potential solutions, which I do not evaluate due to time constraints.

Finally, I propose a simple method for improving the initial predictions of an RNN language model and show that this method significantly outperforms naive solutions.

Acknowledgements

I would like to thank Dr Iain Murray for proposing this project and for his time and dedication in providing me with useful and timely feedback on my work. I am ~~very~~ grateful to him for helping me ask the right questions and helping me realize that using the word “very” in a sentence does not necessarily make it stronger.

I also want to thank Justine, Kamen and Traiko for their support and their help in proof-reading sections of my report.

Table of Contents

Chapter 1

Introduction

Dasher is an open-source, freely distributed text-entry system invented by the late Sir David MacKay. Its interface allows the user to enter text by moving a pointer on a digital screen. One can interface with Dasher using a trackpad or an eye-tracking device, which makes the system particularly suitable for people with certain types of severe disabilities. Dasher is discussed in more detail in Section 2.1.

Crucial to Dasher's functionality is its internal character-level language model. Language modelling is discussed more formally in Section 2.2 but for the purposes of this chapter a character-level language model is something that outputs a probability distribution (over an alphabet) for the next character in a sequence given all previous characters in the sequence. As a user enters text in Dasher, the interface changes according to the predictions of the language model. A more detailed description of this mechanic can be found in Section 2.1, but at this point the reader is encouraged to view the visualization of the interface at this web page[1] or watch the beginning of this video[2] which demonstrates how Dasher is used. The keen reader can download Dasher and try to enter text using the system – the latest stable release at the time of writing can be downloaded from the Dasher project GitHub repository[3].

A key insight, which is detailed in Section 2.1, is that, under a certain set of assumptions, the better the language model is at predicting the next character (More on what "better" means and how it is quantified in Section 2.2), the faster a user will be able to enter text. Thus, improving the language model behind Dasher would allow users to communicate more efficiently.

Dasher's current language model takes into account only the last five characters the user entered when predicting the current one. With the increase in popularity of Deep Neural Networks, the state-of-the-art in language modelling has advanced well beyond what it was when Dasher was invented. Recurrent Neural Networks [Mikolov and Zweig, 2012, Merity et al., 2017, Melis et al., 2017a, Krause et al., 2018] and more recently, self-

[1]`https://web.archive.org/web/20190109082749/http://www.inference.org.uk/dasher/DasherSummary2.html`

[2]`https://youtu.be/0d6yIquOKQ0`

[3]`https://github.com/dasher-project/dasher/releases`

attention architectures [Vaswani et al., 2017, Al-Rfou et al., 2018, Khandelwal et al., 2018] have achieved results that are far superior to those of fixed-length context language models like the one used in Dasher.

It is then not unreasonable to ask if and how state-of-the-art language models can be used in Dasher. Some potential difficulties could arise from the fact that these models require a lot more data, memory and computational operations in comparison to the lightweight model currently in use in the system. My work in the first part of the project, which I describe in more detail in Section 1.1, addressed these issues.

1.1 Previous work carried out

The current language model in Dasher was originally proposed as a part of a compression algorithm called Prediction by Partial Matching (PPM) [Cleary and Witten, 1984] and I will refer to this language model as PPM. I describe finite-order language models, of which PPM is one, in more detail in Section 2.2.2. Last year I compared PPM to Recurrent Neural Network (RNN)-based language models in terms of their language modelling capabilities and how much memory and computation is required for them to be used in Dasher.

I found that RNN-based language models are better than PPM at modelling language when given a lot of training data. However, I also found that RNN-based language models could be prohibitively slow if naively implemented in Dasher but I did not investigate solutions to this problem at the time.

Another problem which I did not address last year is that of language model adaptation. The current language model has a simple mechanism through which it can quickly adapt its predictions to a specific user's use of language, which would mean that that user will be able to enter text more quickly. What is more, PPM is fast enough for us to be able to estimate its parameters on a piece of text every time Dasher is started. This means that if a user has a collection of text they have written, they can estimate the language model's parameters based on that text and get predictions that are consistent with their writing style.

In contrast, estimating the parameters of RNN-based language models requires large amounts of text and more computational resources and adapting their predictions to a specific user is less straight-forward.

These observations gave me two choices for what to work on this year. One option was to tackle the problem of how to implement RNN-based language models in Dasher by either modifying how the system uses the language model and/or finding ways to get good performance from networks with less parameters which would require less computation to output predictions.

However, if I do this I would be implementing a model that does not have the capability to adapt to a specific user, which is an important feature of Dasher (I motivate the need for adapting language models further in Section 3.1). This is why this year I focus on motivating the need for language model adaptation and assessing potential solutions to adapting RNN-based language models in the context of Dasher.

1.1.1 What I reuse from last year's work

- Except for minor corrections, Section 2.1, which describes how Dasher works is almost entirely the same as the one in my report from last year.
- The first four paragraphs of the Introduction chapter are largely the same as the ones in last year's report.
- For my experiments I reuse the experimentation framework I built last year. This year I had to implement all but the SGD rules for Dynamic evaluation (Section 4.2) and implement adaptation with LHUC (Section 4.3), which prompted some minor changes to the experimentation framework as well.
- Although I described language modelling and Recurrent Neural Networks in the background section of last year's report, I **have not reused** this material. Over the course of the last year, I have gotten a new perspective on both subjects and the relevant sections in this report, which are Section 2.2 and Section 2.3, are new work.

1.2 Adaptive language models

A few paragraphs above I mentioned that we can think of a language model as something that gives us a probability distribution for the next token in a sequence given all the tokens in that sequence so far. Using this definition, we can see how we can use a character model to assign a probability to a whole string, say *"cat"*. Using the Chain rule from probability theory, we can factor that probability as

$$P(cat) = P(c)P(a|c)P(t|ca) \tag{1.1}$$

and use the language model to get the probabilities on the RHS. As explained in Section 2.1, with a few caveats the time it takes for a user to input a string in Dasher is proportional to the negative log probability of that string under the language model, i.e. it takes less time to enter a more probable string.

The probabilities the language model outputs are usually estimated from data (usually *a lot* of it [Chelba et al., 2013]) – the model is exposed to some text and learns that some strings are more probable than others. A model trained on transcripts of conversations in English might assign a high probability to strings similar to those that are more likely to have been found in the transcripts, like *"Good morning."* and much smaller probabilities to, say, *"Morning good."*, *"Mi piace questo gelato."* or *"HghdJdPl8"*.

Keeping in mind that language models tend to assign higher probabilities to strings that are like those seen in the training data, and that in Dasher, the more probable a string is according to the language model, the quicker you can enter it, consider the following example which illustrates the problem I have focused on this year: Imagine you are in charge of implementing Dasher's language model. You go away and find a lot of data to train this model on. You look for freely distributed bunches of text and find a whole lot of books [Lahiri, 2014], news articles [Paul and Baker, 1992], or, say, European Parliament speeches [Koehn, 2005]. You train a model on those and ship it with Dasher. Then a user, call him Pete, downloads Dasher and wants to use it to chat to his friends or send tweets. Now, strings similar to *"Let's grab a pint in Teviot"* and *"lmao pluto so small it will neva b a planet"* are quite unlikely to have occurred in Wall Street Journal articles or European Parliament speeches, so these two examples will be assigned low probabilities by the model and it will take Pete ages to send his tweet. And even if you shipped Dasher with a model that was trained on records of social media interaction, Pete will tend to use some words, phrases or names more often so there will probably still be room for improvement – maybe his pet has a weird name and every time he wants to tell his friends what it has done, he will have to struggle with the Dasher interface.

A further complication is that Pete will not only use Dasher to tweet misspelled nonsense but also for writing his Master's thesis in Microbiology, having conversations with his mother, emailing his supervisor, etc. And he will probably be switching between those relatively frequently. I note here that my work this year mainly focuses on the problem of adapting a language model to the general style of a specific user's writing and provide a further discussion on the matter in the beginning of Chapter 3.

So the main challenge is to develop a language model that will adapt to the user's writing style, will give sensible predictions in various contexts and will adapt quickly.

In the current implementation of Dasher this problem is addressed in an elegant fashion. The language model used at present is quite simple – all it needs to know in order to make its prediction is how many times different contexts of length up to 5 have been seen in the training data. Because of this simplicity, it is very fast to train. Every time Dasher is launched, the model's parameters are estimated from a piece of text stored in a plain text file, which can be specified by the user. As the user types, the context counts that dictate the model's predictions are updated and the text that was just entered is appended to the training file. This means that over time the model will start assigning higher probabilities to strings the user enters more often.

In contrast to Dasher's language model, neural language models (Section 2.3) tend to be more complicated, have millions, sometimes billions [Shazeer et al., 2017] of parameters, require a lot of data and can take days to train on expensive bespoke hardware. This means that if neural language models are to be used in Dasher, taking the current approach to adaptation is simply not an option. It also means that Dasher will have to be shipped with a pre-trained language model so when creating solutions I must think about the costs of storing, distributing and maintaining these models.

Insert a couple of paragraphs about existing solutions for adapting language(and RNN in general) models

Given the problem description above, I define my goals for the project as follows:

- *Experimentally motivate the need for model adaptation* – how much is there to gain from coming up with smart methods to adapt to user text? I present the relevant experiments in Section 3.1.

- *Identify existing solutions to adapting neural models and compare those in the context of Dasher* – adapting language models to recent history is a well-researched area [Kuhn and De Mori, 1990, Kneser and Steinbiss, 1993, Mikolov and Zweig, 2012, Krause et al., 2018, Vilar, 2018, Grave et al., 2016, Chen et al., 2015, Daumé III, 2009, Reddy Gangireddy et al., 2016]. Browsing through the cornucopia of architectures and techniques and selecting the most appropriate ones given the specifics of the application is a non-trivial effort. There are techniques for adapting neural models that have been successfully applied in other domains [Finn et al., 2017] and adapting those to character-level language modelling and Dasher in particular requires making non-obvious design choices.
- *[Optional] Propose a novel architecture/technique/algorithm for adapting language models to rapid domain-shifts that improves on existing approaches or is much more suitable for Dasher.* As mentioned above, the literature on the subject is extensive so coming up with a smart approach that works better than the available solutions can be very difficult. This is why I list this as an optional goal.
- *Sketch the implementation of at least one of the identified solutions.* As the motivation behind the project is improving Dasher, it would be desirable for someone to be able to read my thesis and easily implement what I propose. If I do not identify viable solutions, I should at least describe the main obstacles to be overcome in future work.
- *[Updated] Propose a method for making good initial predictions* – Every time the user starts up Dasher or changes the context in which they are using it (say, switching between windows in the operating system), the language model needs to make predictions without having any context. The present solution to this problem depends on the properties of the current language model and is not a viable approach when using a neural language model. In Chapter 6 I provide evidence for this claim, propose a different method and show that it is an effective approach to providing good initial predictions where no context is available.

1.3 Main contributions

- Experimentally motivated the need for adapting a language model for Dasher
- Experimentally compared three methods for adapting RNN language models
- Proposed a way for providing good initial predictions with no context for neural language models in

Chapter 2

Background

2.1 Dasher

Dasher encapsulates an interesting metaphor for what writing is, namely, navigating through the library of all possible books – if I wanted to write the string *HELLO*, I would first go to the *H* section of the library. Once I am there, I would find the *E* subsection, in which I would look for the one corresponding to *L* and so on.

2.1.1 How text is entered using Dasher

Dasher's user interface (Figure 2.1) represents these sections with coloured rectangles ordered vertically on the right-hand side of the screen. Using continuous gestures, the user moves a cross-hair positioned at the center of the screen towards the rectangle containing the character they want to enter next. As they do that, the rectangles grow in size and the point of view zooms towards the direction of gesturing, giving the user the sense of navigating into the rectangles. As the user moves the cross-hair to the right, possible continuations of the current input string appear inside the rectangle the user is moving towards. Figure 2.1 shows screenshots of this process. As mentioned in the introduction, the reader is encouraged to view the visualization of the interface at this webpage[1], watch the beginning of this video[2], which demonstrates how Dasher is used, or download Dasher[3] and use it to enter a string.

2.1.2 How the probabilistic model determines the interface layout

The heights of the rectangles containing different characters of the alphabet are governed by the probabilities of those characters given the string entered so far. These probabilities

[1]https://web.archive.org/web/20190109082749/http://www.inference.org.uk/dasher/DasherSummary2.html

[2]https://youtu.be/0d6yIquOKQ0

[3]https://github.com/dasher-project/dasher/releases

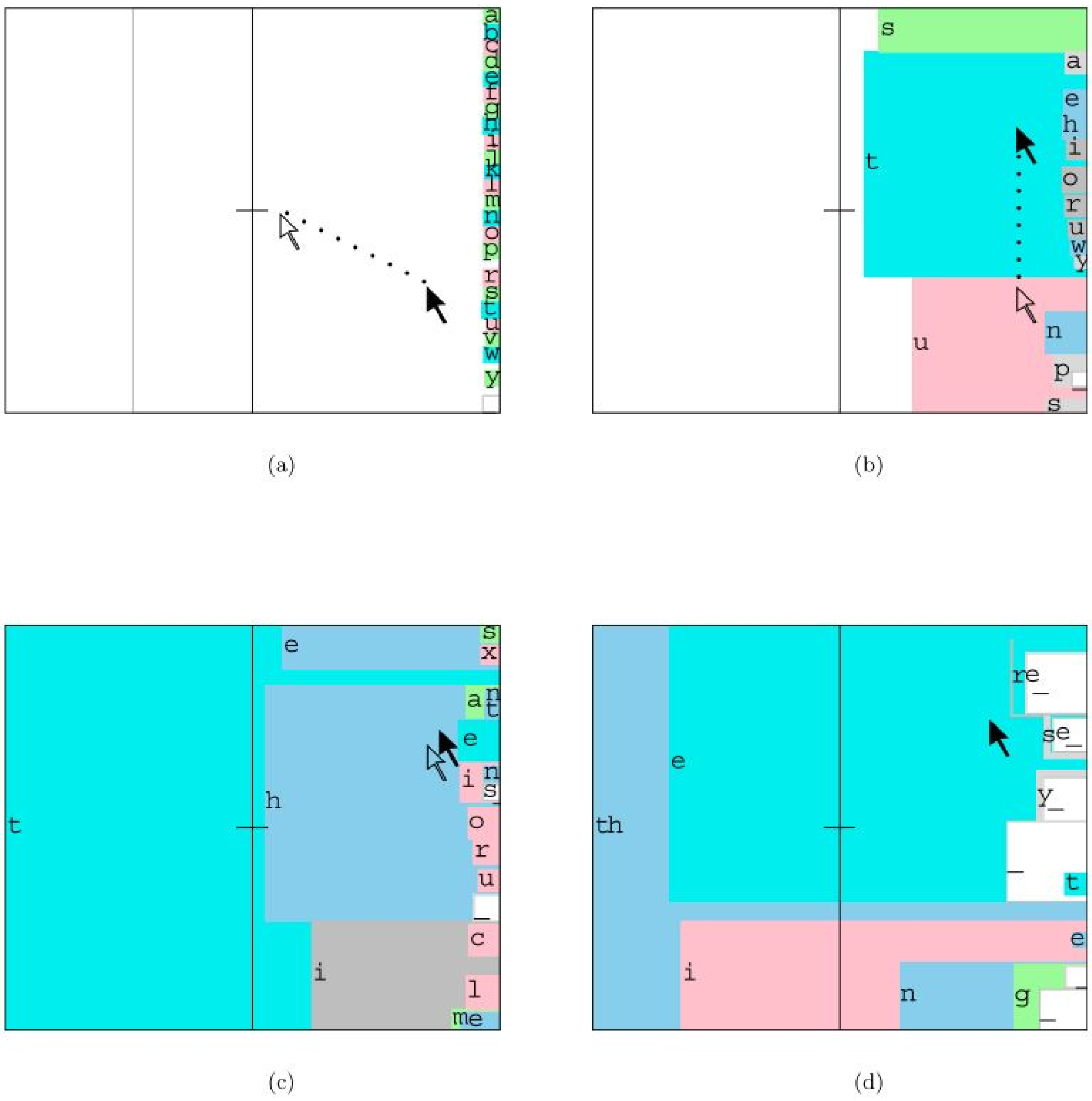


Figure 2.1: (a) Initial configuration. (b),(c),(d) Three successive screenshots from Dasher showing the string *the* being entered. The underscore symbol represents a space. The figure and caption are taken from [Ward, 2001].

are given by the language model.

In the *HELLO* example above, the size of the box containing *H* would be proportional to to the probability of *H* being the first letter of a document according to the language model. Once the user navigates into the *H* rectangle, the height of the *E* rectangle there would be much greater than that of the *Z* rectangle, for example, since strings starting with *HE* are far more common than those starting with *HZ*. This process is analogous to the division of the real line interval $[0,1)$ performed in Arithmetic coding [MacKay, 2005].

This way of dividing the space on the right-hand side of the screen means that h, the height of a box representing the string $c_1c_2\dots c_n$, will be proportional to the product of the conditional probabilities of each of the preceding letters as given by the language model and therefore to the probability of the whole string:

$$h \propto P_L(c_n|c_1 \dots c_{n-1})P_L(c_{n-1}|c_1 \dots c_{n-2}) \dots P_L(c_2|c_1)P(c_1) = P_L(c_1c_2 \dots c_n) \quad (2.1)$$

where each c_i is a symbol and $P_L(c_n|c_1 \dots c_{n-1})$ is the probability for c_n to occur after the string $c_1 \dots c_{n-1}$ as assigned by a language model L.

2.1.3 Dynamics of the Dasher interface

In order to more formally explain how improving the language model will lead to better writing speeds, I must refer to how the Dasher screen is updated, giving the sense of navigating into the boxes on the right hand of the screen as the user interacts with the system.

The part of the number line that the right-hand side of the interface corresponds to is called the *visible interval*. In the initial configuration, when no text is entered, the visible interval is [0, 1). At each timestep, the visible interval is reduced or expanded by a factor proportional to the horizontal distance between the pointer and the crosshairs in the middle of the screen (The specifics of this are given in Section 4.4 of [Ward, 2001]).

That means if we assume the user keeps the pointer at a constant horizontal distance from the middle of the screen, at each timestep the interface will zoom by a constant factor, call it z, and the time taken to zoom to a box of height h is proportional to $\frac{1}{\log h} = -\log h$

Consider T_s – the time taken to enter a string s. h_s, the height of the box corresponding to the string s will be proportional to $P(s)$ – the probability assigned to the string by the language model. So we have:

$$T_s \propto -\log h \propto -\log P(s) \propto -\log_2 P(s) \quad (2.2)$$

– the time taken to input a string is proportional to the negative log probability assigned to that string by the language model. This quantity is also proportional to the evaluation metric used to compare character-level language models. I give more details on this in Section 2.2.3.

2.1.4 Some caveats

For the sake of simplicity, I made some assumptions in the explanation above and also neglected a couple of features of the Dasher interface. In fact, the time taken to input a string is not strictly proportional to the negative log probability of that string for a few reasons, which have to be kept in mind.

Firstly, in order to enable the user to more easily enter low-probability strings, a small constant is added to the probabilities of each character, after which they are re-scaled to sum up to 1.

Secondly, the height of the box on the screen is not strictly proportional to the probability of that string – in order to "increase the maximum speed of vertical scrolling, and give users more time to react before a desired letter disappears out of view" ([Ward, 2001]), the vertical coordinates of all boxes are passed through a non-linear map before displaying the boxes on the screen, making boxes towards the edge of the screen narrower. This effect is shown in Figure 2.2.

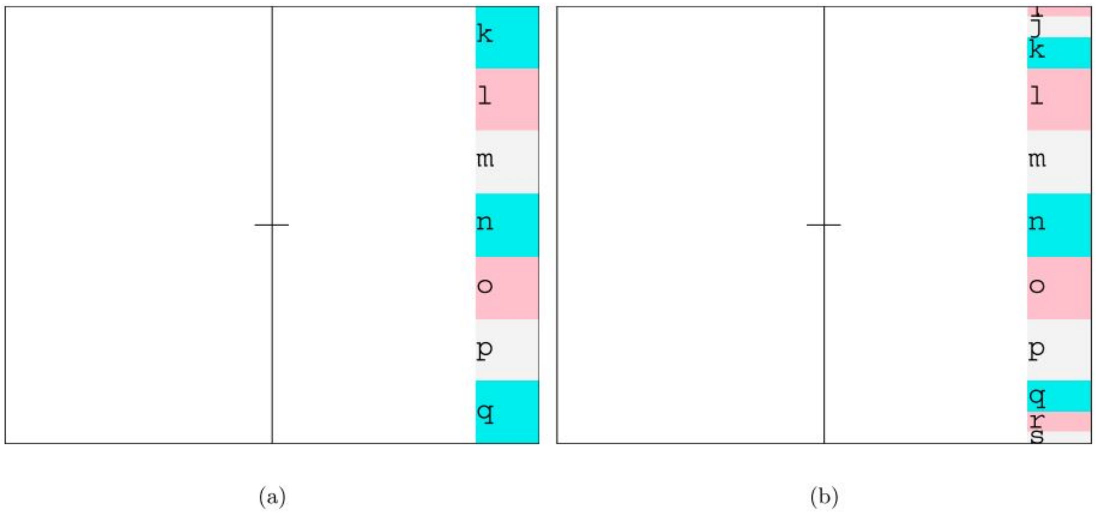


Figure 2.2: (a) shows a square aspect ratio display and (b) shows the same configuration after applying a non-linear map to the vertical coordinates of boxes. The figure is taken from [Ward, 2001].

I also assumed that the user keeps the cursor at a constant horizontal distance from the center of the screen, which will not be true. In practice, the user might have to "slow down" in order to navigate into strings with lower probability. The solution to this problem is that in the interface, the boxes with possibilities for the next symbol are in alphabetical order. Although a novice user might have to stop and think about whether *P* comes after *N*, an experienced one should be able to do this with next to no cognitive effort and will therefore be able to maintain a reasonable constant zoom rate. The last is only a hypothesis, which I have not validated experimentally. In case this is actually false, it might be the case that the smoothness of the distributions the model outputs also affects text entry speed. The effort required to organize experiments to confirm or disprove this, however, would be significant and would potentially prevent me from achieving the main project goals.

Lastly, there are other factors affecting the speed at which a string is entered, as described in [Ward et al., 2000, Ward, 2001] – the aspect ratio of the screen, the frame rate and the rate at which new letters are displayed.

The last two are of particular interest, since the speed at which a model outputs distributions imposes a tradeoff between them – if too many boxes are expanded each frame, this might take too long, reducing the frame rate, and therefore the writing speed. This motivates a thorough investigation into the speed of prediction for any model that is considered for use in Dasher.

2.2 Language modelling

As mentioned in the introduction and detailed in the previous section, Dasher's interface layout is determined by a language model and the speed with which one can enter text using Dasher is very closely related to the model's performance – having a better language model means a user will be able to communicate more efficiently. The motivation behind my efforts is that since Dasher's invention the state-of-the-art in language modelling has improved greatly and that maybe replacing the current language model with a Recurrent Neural Network (RNN)-based one will be a desirable solution – RNN-based language models have been proven to be superior to the type of model that is used in Dasher. And in the last section I explained how having a better language model – one that assigns a higher probability to the text the user enters – should lead to a user being able to enter that text more quickly.

In this section I describe language modelling more formally and explain some shortcomings of the type of language model currently used in Dasher and how models based on Recurrent Neural Networks are theoretically more powerful and achieve superior performance in practice.

2.2.1 What is a language model

A language model is a probabilistic model over sequences of tokens (usually words or characters). A character-level language model models the probability over a sequence of characters $c_1, c_2 \dots c_T$:

$$P(c_1, c_2 \dots c_T; \theta) \tag{2.3}$$

where θ are the model's parameters.

Language models work by factoring this probability using the chain rule:

$$P(c_1, c_2 \dots c_T; \theta) = \prod_{i=1}^{T} P(c_i | c_1 \dots c_{i-1}; \theta) \tag{2.4}$$

and actually modelling the conditional probabilities on the RHS of the equation above. For example, the probability of the string "DASHER" will be

$$P(DASHER) = P(D)P(A|D)P(S|DA)P(H|DAS)P(E|DASH)P(R|DASHE) \tag{2.5}$$

where the parametrization by θ is omitted for compactness.

Dasher uses these conditional probabilities in order to determine the size of the boxes in the interface and as a result of that, as explained in Section 2.1.2, the height of each box in the interface is proportional to the probability of the string it represents under the language model.

The model's parameters are usually estimated from data (a process referred to as training the language model) and so the data used for estimating them influences the predictions the probabilistic model makes. Therefore, we do not expect our model to assign high probabilities to strings generated by a different underlying distribution from the one that generated the data used to estimate the model's parameters. An extreme example would be training a language model on English text and asking it to predict the probability of a German sentence.

Thinking in the context of Dasher, if the text the language model is trained on is very different from the text the user wants to enter, they will get poor predictions and will enter that text more slowly. The fact that the user would probably use Dasher for different purposes and that every person has a separate writing style means that there will always be a difference between the distribution used to generate the training data and that used to generate the data the user enters. This leads to the problem mentioned above – the language model will assign smaller probabilities to the strings a user enters than it should and the user will not be able to enter text as quickly as they could. A reasonable approach to solving this is to change the model's parameters based on the text the user enters (and/or on some text they have entered in the past) so that we improve the model's predictions. This process is usually referred to as adaptation. The main focus of my work this year is on how to best adapt Recurrent Neural Network language models (More on these models in Section 2.3) in the context of Dasher.

A final note is that all of the conditional probabilities in Equation 2.4 are over some alphabet and the size of that alphabet has an effect on the performance of the model. In general, the bigger the alphabet, the smaller the probability of a given string will be. Intuitively, if an alphabet A_1 is a subset of alphabet A_2, the strings of length n I can generate with A_2 will be a superset of those I can generate with A_1 and a character-level model over A_2 will have to spread probability mass over more possible sequences. In the Dasher interface, a user can select from a range of alphabets with varying sizes. One shortcoming of my work this year is that I do not explore the effects of alphabet size on adaptation performance.

2.2.2 Finite-order language models

The language model currently used in Dasher was proposed as a part of a compression algorithm called Prediction by Partial Matching (PPM) [Cleary and Witten, 1984] and I will refer to this language model as PPM. PPM is a finite-order language model, meaning that it considers only the last o characters when making a prediction for the next one, where o is called the order of the model. So the probability of a string of length T under an order-o character-level language model would be

$$P(c_1, c_2 \dots c_T; \theta) = \prod_{i=1}^{T} P(c_i | c_{i-o} \dots c_{i-1}; \theta) \tag{2.6}$$

.

For example, under a second-order language model, the probability of the string DASHER would be

$$P(DASHER) = P(D)P(A|D)P(S|DA)P(H|AS)P(E|SH)P(R|HE) \qquad (2.7)$$

One of the simplest finite-order models is an n-gram model (An n-gram is a sequence of n tokens). Its predictions for the conditional probabilities in Equation 2.6 are given by

$$P(c_i|c_{i-o}\dots c_{i-1}) = \frac{C(c_{i-o}\dots c_i)}{C(c_{i-o}\dots c_{i-1})} \qquad (2.8)$$

where $C(c_{i-o}\dots c_i)$ is the number of times the string $c_{i-o}\dots c_i$ was observed in the training data.

The assumption that the next character in a sequence only depends on a fixed number of preceding characters is clearly false. For example, the PPM version currently used in Dasher is of order 5, meaning that it will output the same distribution after seeing the two contexts below:

1. *The theory of relativity was developed by Albert Einste*

2. *Do not use water, but put milk inste*

whereas it is clear that a good model should make different predictions given these contexts.

In the example above simply increasing the order of the model by 1 would probably greatly improve the quality of predictions. But one can think of dependencies which cannot be captured by a finite-order model, no matter how big we make o – for instance, we can have arbitrarily long sentences surrounded by quotes. A model that considers only a finite history cannot learn that in all well-formed English strings all opening quotes should always be followed by closing ones at some point in the future.

At this point a reasonable question arises: despite this theoretical limitation, can we make o big enough to capture all dependencies we care about when making predictions about what character someone will enter next? Here we run into the problem of data sparsity. The parameters of this type of models are estimated using n-gram counts from the training text and the number of possible strings of length n is $|A|^n$ where $|A|$ is the size of the alphabet used. Even if our alphabet contains only lowercase Latin letters plus a space character, the number of strings of length 60 we can make is bigger than the number of atoms in the observable universe. Of course most of these will never occur in natural language so we do not care about them but the number of those that will is still big enough that we need huge amounts of text in order to estimate these probabilities reliably. Figure 2.3 demonstrates the problem of sparsity for a real dataset.

Two popular solutions to this problem, which are also used in PPM, are *interpolation* and *backoff*. Order-o models which use interpolation interpolate between the predictions

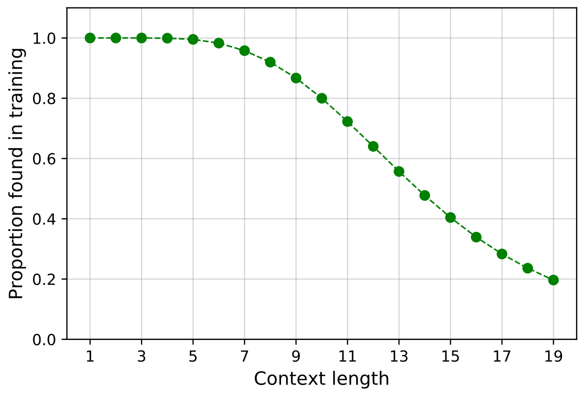


Figure 2.3: Proportion of substrings in the test data that were also found in the training data. Test and training data are the first 80 million and the following 10 million characters from the text8 dataset, which is a 2006 snapshot of Wikipedia pre-processed to contain only the 26 lowercase letters of the English alphabet plus spaces. Note that even for this restrictive alphabet, the proportion is quite low for longer substrings. This demonstrates the problem of data sparsity – if I am using an n-gram or a PPM model, after estimating a model's parameters on 80 million characters, I can only make use of dependencies that span 19 characters or longer in 20% of cases when I need to predict the next character.

of all orders up to o. Backoff is the technique of resorting to (backing off to) predictions of lower orders if higher-order ones are not available – if I am trying to predict the next character in the context of "DASHE" and I have not seen the string "DASHE" in my training text, I will try to make a prediction in the context of "ASHE" – if I have not seen that either, I will predict in the context of "SHE" and so on. The fact that PPM uses lower-order predictions when higher-order ones are not available becomes relevant in the discussion in Section 6.1, which is about how Dasher makes good initial predictions when no context is available.

There is another problem with high-order models. Apart from needing a lot of data to reliably estimate parameters, we need to store the counts for all contexts we have observed. Since PPM uses interpolation and backoff, it needs counts for all contexts of lengths *up to* the maximal order and storing these can require a lot of memory. Even though the current implementation of Dasher is optimized for low memory usage, in my work last year I found that using contexts of length more than 10 would impose memory requirements that are unreasonable for a product that is to be used on consumer-grade hardware.

2.2.3 Evaluating language models

Since the overall goal of my work is to find a better language model for Dasher, I need to have a way to compare language models. This section discusses the way language models are usually compared and why this is appropriate in the context of Dasher.

For most Natural Language Processing tasks one talks about the difference between *intrinsic evaluation* – evaluating a model's performance via some metric that is inherent to the task – and *extrinsic evaluation* – measuring how the model affects some downstream task it is used in, which is often what we ultimately care about. In an ideal situation, we would have such metrics for intrinsic evaluation that when comparing two models intrinsically, we would choose the one that will do better in practice without needing to perform extrinsic evaluation. However, devising metrics that satisfy this condition is usually challenging.

Fortunately, in the case of Dasher, there are good reasons to think that the way language models are compared intrinsically is a good proxy for the downstream system's performance.

The way language models are compared is by having them assign a probability to a given text – if model A assigns a higher probability to some text than model B, we say A is a better model for that text than B. Since computing the probability of a string involves multiplying many conditional probabilities, all smaller or equal to 1 and doing this naively can lead to numerical underflow, we usually work in the log domain. The number actually reported is the negative log probability assigned by the model, normalized by the length of the string:

$$-\frac{1}{N}\sum_{i=1}^{N}\log_2 \hat{P}_i(c_i|c_1\dots c_{i-1}) \tag{2.9}$$

where N is the length of the string and $\hat{P}_i(c_i|c_1\dots c_{i-1})$ is the probability the model assigned to the true character at position i.

The quantity in Equation 2.9 is usually referred to as Bits Per Character (BPC). The following paragraph gives a brief explanation of that name but is by no means essential for understanding the rest of my thesis.

The reason for the name Bits Per Character comes from a result in Information Theory, which tells us that if we have a stream of outcomes (characters in our case) generated from some distribution p and we use a coding scheme that is optimal for some other distribution q to encode this stream, the number of bits we would need to encode every character on average is equal to the cross entropy between p and q. The quantity in Equation 2.9 is an estimate of the cross-entropy between the distribution predicted by the model and the true distribution of characters in the text:

$$\begin{aligned} BPC(model, string) &= \frac{1}{N}\sum_{i=1}^{N} H(P_i, \hat{P}_i) = \frac{1}{N}\sum_{i=1}^{N}\sum_{c\in|A|} -P_i(c)\log_2 \hat{P}_i(c) = \\ &= -\frac{1}{N}\sum_{i=1}^{N}\log_2 \hat{P}_i(c_i) \end{aligned} \tag{2.10}$$

where $H(P,Q)$ denotes the cross-entropy between P and Q, $P_i(c)$ is the true probability of character c at position i, $\hat{P}_i(c)$ is the one estimated by our model and in the last term c_i denotes the true character at position i. The first and second equalities are the definition of average cross entropy and the third one follows from the fact that $P_i(c)$ will be 1 for c_i and 0 for all other characters. The $P_i(c)$ and $\hat{P}_i(c)$ terms are conditioned on the preceding context like in Equation 2.9 but in Equation I omit the conditioning 2.10 for brevity.

The reason why this metric is a good proxy for the performance in Dasher comes from the discussion in Section 2.1.3. There I argue that under a set of assumptions the time needed for a user to enter a given string is nearly proportional to the negative log of the probability of that string under the language model. That means that it is also proportional to the Bits Per Character of that text under the model.

In all experiments I perform I compare approaches on the basis of this metric.

2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a popular tool for modelling sequential processes and are widely used in solving a variety of tasks in multiple domains. RNNs can be used to assign a probability to a sequence of discrete inputs, which is precisely the task we care about in language modelling, as discussed in Section 2.2. In this section I describe how this is done and how RNNs differ from finite-order models.

Let us assume we have an ordered sequence of T inputs $I = \{i_1, i_2 \dots i_T\}$ – in the case of character level language modelling these would be the characters in a string we want to assign a probability to, indexed by their position in the string. An RNN will consume these characters one by one, storing an internal representation of the string it has processed. At each timestep the model will process a character, update this internal representation and use it to produce a probability distribution for the next character. I describe the process in detail below.

The first step we need to take is to embed the inputs in a high-dimensional space: When doing language modelling, we would map each character to a vector in $\{0,1\}^{|A|}$, where $|A|$ is the size of the alphabet we use (The reader can refer back to Section 2.2.2 for a discussion on the choice of alphabet). We do this mapping from character to vectors using a technique called one-hot-encoding (also known as dummy coding or 1-of-K coding). In order to create a one-hot embedding of a character into $\{0,1\}^{|A|}$, we assume an (arbitrary) ordering of the alphabet we have chosen. The one-hot encoding version of the ith character in the ordered alphabet is a vector in $\{0,1\}^{|A|}$, which has all of its elements set to zero, except for the ith one, which is set to 1. For example, if our alphabet is $A = \{a,b,c,d\}$ and we assume alphabetical ordering, the one-hot encoded version of the character c would be $[0,0,1,0]^T$.

Let $X = \{\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_T\}$ be the sequence of embeddings of the elements in the input sequence. The task is to estimate $P(\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_T)$. In an RNN model this factorizes as

$$\hat{P}(\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_T) = \prod_{t=1}^{T} \hat{P}(\mathbf{x}_t | \mathbf{x}_1 \dots \mathbf{x}_{t-1}) \tag{2.11}$$

.

The conditional probabilities above are given by

$$\hat{P}(\mathbf{x}_t|\mathbf{x}_1 \dots \mathbf{x}_{i-1}) = \sigma(\mathbf{o}_t) \tag{2.12}$$

$$\mathbf{o}_t = g(\mathbf{h}_t; \theta_o) \tag{2.13}$$

where $\sigma(\mathbf{z})$ is the softmax function given by

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.14}$$

$g(\mathbf{x}; \theta_o)$ is usually a linear map and is always a differentiable operator and $\mathbf{h}_t$ is called the network's *hidden state* at time t and can be viewed as the model's representation of the language context at that timestep.

The reason why this model is effective for modelling sequential data is the way the hidden state at every timestep is obtained. It is computed recursively (hence the name of the model) from the hidden state at the previous timestep and the current input, where f is a differentiable function:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta_h) \tag{2.15}$$

The function f can usually be broken down in two separate operations:

$$\begin{aligned} \mathbf{a}_t &= f_1(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta_h) \\ \mathbf{h}_t &= a(\mathbf{a}_t) \end{aligned} \tag{2.16}$$

where f_1 is a linear operator, a is a nonlinear function and the elements of $\mathbf{a}_t$ are called the *activations* at timestep t.

A graph of the computations in a Recurrent Network with no outputs is given in Figure 2.4 . The figure illustrates the concept of *unfolding* the computation graph for the network, which is important when it comes to training recurrent networks – we can represent the recurrent computation as a chain of applications of the recurrent function f, where its parameters are *tied* across timesteps.

2.3.1 The expressive power of RNN models

Here we can make a contrast between RNN-based and finite-order models. As discussed in Section 2.2.2, the latter are not capable of capturing dependencies across more timesteps than the model's order – the probability of an event at timestep t is independent of all events at timesteps before $t - o$, where o is the model order (See Equation 2.6). In RNN models, however, because of the recurrence in the computation of the hidden state (Equation 2.15), the output of the network at timestep t depends on all inputs up to and including timestep t. In the context of language modelling this means that, at

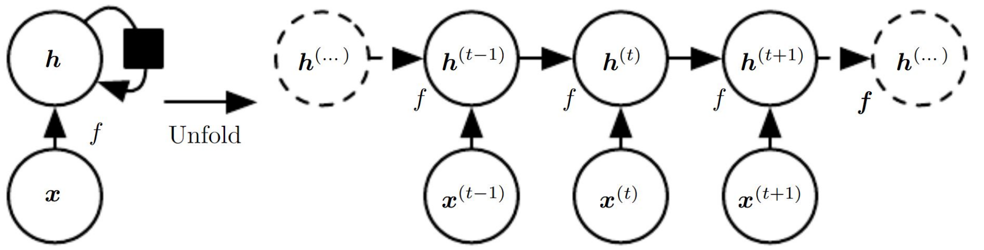


Figure 2.4: (The figure and this caption are taken directly from Chapter 10 of [Goodfellow et al., 2016]) "A recurrent network with no outputs. This recurrent network just processes information from the input $\mathbf{x}$ by incorporating it into the state $\mathbf{h}$ that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of a single timestep. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance."

least in theory, RNN-based models can capture dependencies that span an arbitrary number of timesteps. When discussing the limitation of finite-order models in Section 2.2.2, I argued that a finite-order model would not be able to learn, for example, that well-formed English sentences should always have a pair of closing quotation marks following every opening one. To contrast that with RNN models, [Graves, 2013] used an RNN language model trained on Wikipedia data to generate sequences, which contained, among other things, valid URLs and valid XML. In a popular blog post[4], Alex Karpathy used the same approach to generate Latex, which "almost compiles". [Harada et al., 2001] show that RNN models can learn simple context-free grammars with recursive rules, which is impossible for a finite-order model. These works contain evidence that RNN-based models can learn complex syntactic dependencies that span across multiple timesteps, which is clearly a desirable property for any language model.

2.3.2 Types of RNNs

The choice of the function f in Equation 2.15 is what differentiates between different "flavors" of RNNs. The simplest type of RNN is the Simple Recurrent Network (SRN) [Elman, 1990] in which the recurrent computation is given by

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; V, U, \mathbf{b}_h) = a(V\mathbf{h}_{t-1} + U\mathbf{x}_t + \mathbf{b}_h) \tag{2.17}$$

where $V \in \mathbb{R}^{H\times H}$, $U \in \mathbb{R}^{H\times I}$ and H and I are the dimensionalities of the hidden state and the input vectors, respectively. The function a is some nonlinearity, for example, the logistic sigmoid. A sketch of the dynamics of an SRN is given in Figure 2.5.

Although in theory SRNs could learn dependencies spanning arbitrarily long sequences, there is a problem with their dynamics, which prevents them from learning long-range dependencies effectively. This problem is commonly known as the *vanishing gradient*

[4]`https://web.archive.org/web/20190321235642/https://karpathy.github.io/2015/05/21/rnn-effectiveness/`

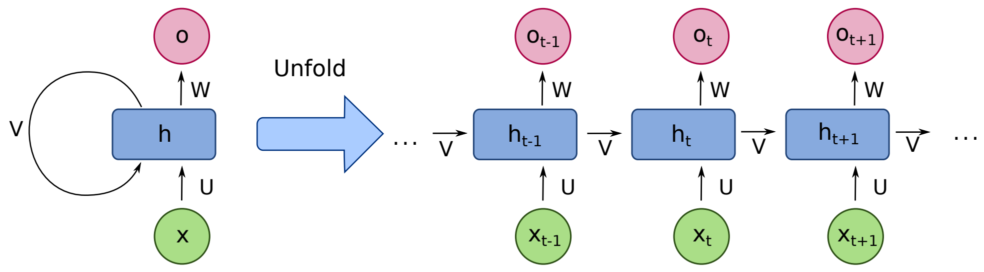


Figure 2.5: A schematic of the dynamics of a Simple Recurrent Network. The matrices that parametrize the different operations are given next to the arrows which represent these operations. (Image credit: Francois Deloche: `https://commons.wikimedia.org/wiki/User:Ixnay`)

problem [Kolen and Kremer, 2001] and even though it is not immediately relevant to the work I do this year, I provide a short discussion of it here as it motivates the existence of more complicated RNN variants, the most popular of which is the Long-Short-Term Memory (LSTM) Network, which I use in my experiments.

The intuition behind the vanishing gradient problem stems from the fact that the recurrent computation is the mechanism through which information from previous parts of the sequences is carried through to the current timestep. Let us consider how the input at timestep $t-n$ contributes to the hidden state, and therefore the output, at timestep t by expanding the term for $\mathbf{h}_{t-1}$ in Equation 2.17:

$$\begin{aligned}\mathbf{h}_t &= f(\mathbf{h}_{t-1}, \mathbf{x}_t) = f(f(\dots f(\mathbf{h}_{t-n-1}, \mathbf{x}_{t-n}), \mathbf{x}_{t-n+1}), \dots), \mathbf{x}_t) = \\ &= V^{n-1}U\mathbf{x}_{t-n} + V^{n-2}U\mathbf{x}_{t-n+1} + \dots + VU\mathbf{x}_{t-1} + U\mathbf{x}_t\end{aligned} \tag{2.18}$$

The only term in 2.18 that depends on $\mathbf{x}_{t-n}$ is $V^{n-1}U\mathbf{x}_{t-n}$. If we consider the case where the dimensionality of the hidden state is 1 and so the matrix V is actually a scalar, we can reason that since the contribution of $\mathbf{x}_{t-n}$ is exponential in V, it will either become very close to 0 in case $|V| < 1$ or become extremely large in the case when $|V| > 1$. This instability is the reason why it is difficult to estimate parameters for which a simple recurrent networks can effectively pass information over many timesteps. I elaborate on how this problem relates to estimating the parameters of an RNN in the next section.

Another problem with SRNs, the explanation for which is less rigorous, is that the hidden state, and, by extension, the weights which govern the recurrent dynamics of the network, have to serve two functions simultaneously: the hidden state has to firstly contain information which is relevant to the current prediction, and secondly, maintain and propagate information which will be useful for future predictions.

Arguably the most widely used RNN variant, the LSTM Network [Hochreiter and Urgen Schmidhuber, was proposed as a solution to these problems. The way the hidden state is computed in an LSTM means that there is a path in the computation graph that connects the input at timestep $t-n$ and the hidden state at timestep t that does not involve repeated multiplication by the same matrix, thereby going some way towards solving the vanishing gradient problem. The complete dynamics of the recurrent computation in an LSTM are more complicated and are not immediately relevant to the discussion in the rest of my thesis, so I will omit giving the full formulation here.

Another popular architecture is the Gated Recurrent Unit (GRU) network [Cho et al., 2014], which is conceptually similar to the LSTM network but has a more compact recurrent computation and has been found to achieve similar performance to that of LSTMs [Chung et al., 2014]. Although the simpler recurrent computation might potentially make GRUs more desirable as a solution for Dasher, I chose to use an LSTM language model since such models are more popular in language modelling literature and so it would be easier for me to compare my results to existing ones.

In recent years there have been many architectures proposed specifically for the task of language modelling, which have set state-of-the-art results [Krause et al., 2018, Merity et al., 2017, Chung et al., 2016, Zoph and Le, 2016]. However, in 2017 a study [Melis et al., 2017b] found that, when properly regularized, ordinary LSTM models perform at least on par with at least some of the more modern architectures.

Another note on expressiveness So far I have argued that RNN language models, unlike finite-order ones, can, in theory, learn dependencies spanning an arbitrary number of tokens in the past. I also explained why it is difficult to make Simple Recurrent Networks do that because of the multiplicative connections in their recurrent computation and that LSTM networks mitigate that issue. It would be interesting to know, then, how far in the past an LSTM language model actually looks when making predictions.

In a recent study [Khandelwal et al., 2018] asked this very question. They tried to find out how much context an LSTM language model uses through ablation studies – they perturbed parts of the text (changed tokens, shuffled them and added other types of noise) at a varying distance in the past and analyzed how that affected performance. They found that tokens further than around 200 timesteps in the past have almost no effect on prediction quality and that word order matters only for the most recent 20-50 tokens. They experimented solely with word-level language models – I think it is not clear whether the models *could not* or *did not need to* make use of longer context – it might be the case that character-level language models use longer context but I have not performed experiments to test this hypothesis.

The fact that language models only retain information from recent context will become relevant in the discussion of a method for adapting language models in Chapter 4. There I discuss LHUC [Swietojanski et al., 2016] and sparse Dynamic evaluation [Krause et al., 2018], both methods which perform adaptation by scaling the entries of the RNN's hidden state. If we want our model to adapt to recent history and we want to use its hidden state to do so, we need to keep in mind how much context is stored in that hidden state.

2.3.3 Training RNN language models

In the previous section I talk about the difficulties in estimating the parameters of RNN models. Here I briefly describe how this is done, giving more detail on the parts of the process that are particularly relevant to the work I do this year.

As mentioned previously, RNN language models model the probability of the next token given all the preceding ones at every timestep (See Equations 2.11 and 2.12). Assuming that each token is generated via a probabilistic process according to some distribution p, our goal is to find parameters that minimize the cross-entropy between p and $\hat{p}$, the estimates given by our network. By definition, this is given by

$$H(p,\hat{p}) = \mathbb{E}_p[-\log_2 \hat{p}] = -\sum_{c \in A} p(c) \log_2 \hat{p}(c) \tag{2.19}$$

where A is the support of p and $\hat{p}$, in the case of character-level models – the alphabet over which the model operates.

Since the true distribution p is not known, we treat the training text as a sample from that distribution and we minimize the Negative-Log Loss (NLL) of the model on that text:

$$NLL = -\frac{1}{T} \sum_{t=1}^{T} \log_2 \hat{p}(c_t) \tag{2.20}$$

which is a Monte-Carlo estimate of the true cross-entropy. In Section 2.2.3 I introduced this quantity as the Bits Per Character Score of the model, which is the metric I use to compare language models and, under some assumptions, is proportional to the time taken for users to enter text in Dasher when a particular language model is used.

In order to evaluate the NLL, we produce prediction targets for our network by shifting the inputs one timestep in the future, so that the target at timestep t is equal to the input at timestep $t+1$. A simple schematic of this is given in Figure 2.6.

Having a sequence of inputs $X = \{\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_T\}$, which are one-hot encodings of the characters in a string and a set of targets $Y = \{\mathbf{y}_1, \mathbf{y}_2 \dots \mathbf{y}_T\}$, which are the same encodings, shifted by one timestep, we process the sequence using Equations 2.12 to 2.15, ending up with a sequence of estimates for the conditional probabilities $\hat{Y} = \{\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2 \dots \hat{\mathbf{y}}_T\}$. We would then use the estimated probabilities and the targets to compute the NLL as

$$NLL(Y,\hat{Y}) = -\frac{1}{T} \sum_{t=1}^{T} \log_2 \hat{\mathbf{y}}_t^T \mathbf{y} = -\frac{1}{T} \sum_{t=1}^{T} \log_2 \hat{y}_{ti} \tag{2.21}$$

where $\hat{y}_{ti}$ is the element of $\hat{\mathbf{y}}_t$ at the index at which $\mathbf{y}_t$ has a value of 1 ($\mathbf{y}_t$ is a one-hot encoded vector).

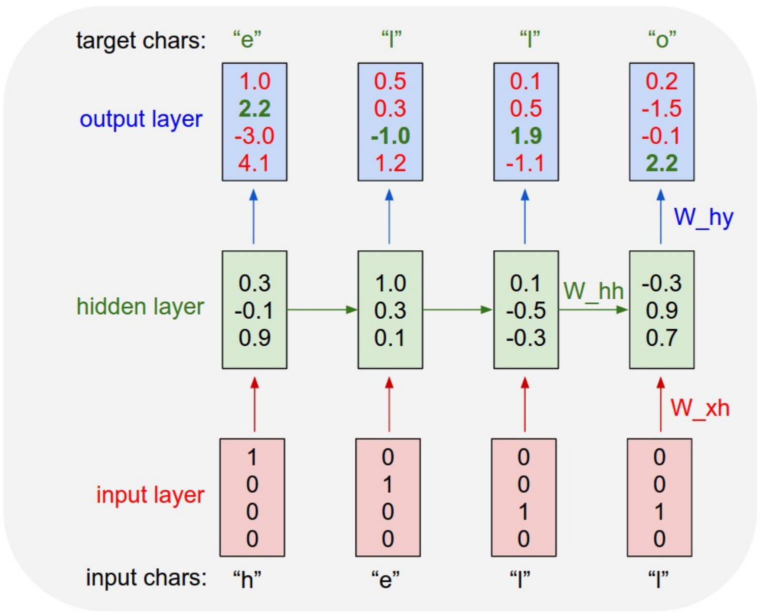


Figure 2.6: An example of a simple recurrent network unrolled through time – note that the targets are just the inputs shifted by one timestep. (Image taken from Andrey Karpathy's blog: `https://web.archive.org/web/20190321235642/https://karpathy.github.io/2015/05/21/rnn-effectiveness/`)

Back-propagation We can look at the process of computing the outputs of the network from its inputs as carrying information provided by the inputs through the computation graph of the network, which is why this process is known as *forward propagation*. Since all of the operations that are required to compute the estimates and the loss from the input vectors are differentiable, we can compute the gradient of the NLL with respect to the model's parameters $\nabla_\theta NLL$ by repeatedly applying the Chain Rule from calculus. This can be viewed as carrying information about the loss back through the computation graph and is known as *back-propagation*. The details of this process are not immediately relevant to my work this year so for them I refer the reader to Section 6.5 of [Goodfellow et al., 2016].

Back-propagation through time The procedure for performing back-propagation in RNNs is called back-propagation through time(BPTT), but although it has a special name, it is not conceptually different from back-propagation in any other type of neural network. Working with the unfolded computation graph in Figure 2.5, we can apply the Chain Rule starting from the loss at the final timestep and compute all the required gradients. The reader can find a more detailed description of this procedure in Section 10.2.2 of [Goodfellow et al., 2016].

Vanishing gradients In Section 2.3.2 I described the *vanishing gradient* problem (also known as the *exploding gradient* problem) in the context of forward propagation by explaining how, because of the multiplicative interactions in the recurrent computation, information from the inputs in the past either shrinks to zero or grows without bound as we compute new states. Since computing the gradient of $\mathbf{h}_{t-1}$ with respect to $\mathbf{h}_t$ also involves multiplication, the same problem is present in back-propagation: the gradient of the loss at time t with respect to the hidden state at time $t-n$ also shrinks to zero (vanishes) or grows without bounds (explodes).

Truncated back-propagation through time When performing BPTT on the unrolled computation graph, we need to store all of the input vectors and hidden state activations in order to be able to propagate information about the loss at the final timestep back to the start of the network. For very long sequences this can be computationally infeasible. Also, since we cannot efficiently propagate gradient information arbitrarily far back in the past because of the vanishing gradient problem we would not benefit much from performing BPTT on the whole sequence. Instead, we divide our sequence $\{x_1, x_2, \ldots x_T\}$ into M shorter sub-sequences $s_1 = x_{1:n}, s_2 = x_{n+1:2n}, s_3 = x_{2n+1:3n}, \ldots, s_M$, each of length n. We then perform BPTT on each sequence in turn, updating our weights after processing each sequence, which means we only have to store activations and inputs for n timesteps at a time. This approach is called Truncated back-propagation through time (TBPTT). The length of the sub-sequences becomes a training hyperparameter which needs to be tuned and this comes up again when I discuss different adaptation methods in Chapter 4.

Gradient descent optimization Let us call the NLL on the ith sub-sequence $\mathcal{L}_i$. The goal of the training procedure will be to minimize the sum of the losses on all sub-sequences. Since we know how to compute the gradient of the loss with respect to the network parameters on one sequence, we can compute the average of that over all sequences:

$$\nabla_\theta \mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \mathcal{L}_i \tag{2.22}$$

After computing it, we can move our parameters in the direction of minimizing this average loss:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L} \tag{2.23}$$

where η is a hyperparameter controlling the size of this step and is called the *learning rate*. We can then repeat this process until we reach some local optimum – a setting of the hyperparameters for which the loss on our training set is minimized. This optimization technique is called gradient descent and the whole process of repeatedly processing inputs, computing losses and gradients and updating the weights in an attempt to minimize the loss is referred to as training.

Batch training Instead of computing the average loss over all sub-sequences in our dataset, we can divide the dataset into smaller subsets of size B called *batches*. If B

is large enough, computing the average loss on a single batch would give us a good estimate of $\nabla_\theta \mathcal{L}$ without having to process the whole dataset. We would process batches independently, updating the weights after processing each one, usually speeding up the training process. This way of learning the weights is called Mini-Batch Gradient Descent and when the batch size is one it is called Stochastic Gradient Descent (SGD) although the terms are often used interchangeably. There is a trade-off between using smaller batches and being able to make updates more often and using bigger batches and getting better estimates of the true average loss and so the batch size is another training hyperparameter.

Per-parameter adaptive gradients Numerous modifications to the basic SGD optimization method have been proposed in literature (See [Ruder, 2016] for a review of some of the popular ones). The specifics of these approaches are not relevant to my thesis with the exception of RMSprop [Tieleman and Hinton, 2012], which is the motivation behind one of the approaches to domain adaptation I consider (See "Dynamic evaluation with an RMSprop learning rule", Section 4.2). The core idea behind RMSprop is that different learning rates are needed for different parameters. It determines the individual learning rates by dividing the learning rate by a the square root of a running average of the squared gradient for that parameter:

$$\begin{aligned} E\left[g^2\right]_t &= 0.9E\left[g^2\right]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \varepsilon}} g_t \end{aligned} \tag{2.24}$$

where g_t is $\nabla_\theta \mathcal{L}$ after processing the tth batch and ε is a stabilization parameter.

2.3.4 RNNs in Dasher

Some of the differences between RNN and finite-order language models need to be taken into consideration when thinking about how to incorporate RNNs in Dasher. I discuss these in more detail in me report from last year. Here I provide a summary of the relevant points.

Training in Dasher State-of-the-art RNN language models have millions of parameters and sometimes take days to train on specialized hardware. Estimating the parameters of PPM is cheap enough that it can be done on a piece of text every time the system is started.

This entails some changes to how the language model for Dasher will be delivered. Currently there are no ready parameters shipped with the system – the user can choose any text they want for the parameters to be estimated on before the system is started and can even change these texts depending on what they want to use the system for at the given time.

This approach is simply not an option for an RNN-based model. If one is to be used in Dasher it would would need to be trained on a server and the weights of the model would be shipped with the software. If a user wants to have different models for different purposes, they would need to have different sets of weights. I discuss this in more detail in Section 3.2.

Prediction in Dasher In the current version of Dasher, the implementation of PPM and the implementation of the dynamics of the interface are tightly coupled. Obtaining a distribution over the next character for a box in the interface involves following pointers to parent nodes, which contain the number of times the context they represent has been seen and then doing arithmetic with those counts. Since Dasher builds a tree of predictions and so this procedure has to be done multiple times when rendering every frame, it has been optimized for minimal memory usage and also takes very few computational steps

In contrast, obtaining a single distribution over the next character using an RNN requires performing several matrix-matrix and matrix vector multiplications, which are more computationally expensive. These operations can be ran be parallelized and performed more quickly, especially if the user has a machine with a GPU but due to the sequential nature of how RNNs process information, parallelizing operations is less helpful with RNNs than with other types of neural networks. We can potentially run in parallel the prediction for a single character over many branches but we cannot batch predictions for more than one step on a single branch. Implementing RNN language models naively and not making use of any parallelism can result in the system demanding too much computational resources. As I mentioned in the introductory chapter, alongside with how to perform adaptation, this is the other big issue that needs to be tackled before RNNs can work in Dasher.

2.3.5 A note on the state-of-the-art

At the time when I began working on this project, RNN-based language models achieved state-of-the-art results in both word- and character-level language modelling [Krause et al., 2018, Józefowicz et al., 2016, Mujika et al., 2017]. Here I have to note that since then new state-of-the-art results in character-level language modelling have been achieved by using a different type of network called a Transformer [Vaswani et al., 2017, Al-Rfou et al., 2018]. Transformers have other potentially desirable properties since unlike RNN models they do not process the tokens in a sequence one at a time.

One of the approaches to adaptation, namely Dynamic evaluation [Krause et al., 2018] (Section 4.2) can be modified to be used with Transformer networks but I do not see a straightforward way to do this with the other two, which I also discuss in Chapter 4. I do not experiment with Transformers nor do I mention them in the rest of my thesis since they are outside of the defined scope of my work which is to find methods for adapting RNN models.

2.3.6 Implementation of RNNs in my experimental framework

I use the PyTorch framework [Paszke et al., 2017] for implementing neural language models. Most of the experimentation framework I use I built for the first part of the project last year. The code for it can be found on my project's GitHub repository[5].

[5] `https://github.com/YasenPetrov/char-rnn-experiments`

Chapter 3

Adaptive language models

The layout of Dasher's interface is dictated by the probabilities of the strings that could be entered (Section 2.1.2). These probabilities are given by Dasher's language model (Section 2.2). In Section 2.1.3 I explained how improving the language model's predictions can allow users to enter text more quickly. In Section 1.2 I gave an example motivating the need for any language model that is to be used in Dasher to be able to adapt its predictions to the user's writing style and the type of text they are currently entering.

It has to be noted that *style*, *genre*, *domain*, *topic*, *type* and other similar terms refer to distinct types of categories used to separate text. I should also mention that linguists are not necessarily in agreement about what all of those mean and how textual pieces should be separated according to them [Lee, 2001]. For the purposes of my thesis, I engage in a slight abuse of terminology and use these terms interchangeably. When I say that two texts have different genres, types or come from different domains I mean that I assume the two texts were generated from two different underlying distributions.

Before I continue with the rest of this section, I introduce three more terms. In domain adaptation tasks, one usually makes the distinction between in-domain (ID) data – data from the domain the model will be used in, out-of-domain (OOD) data – data that is not ID data, and general domain (GD) data – data from various domains and styles. For example, if building a translation system for technical manuals, we might treat some large parallel corpus, say Europarl [Koehn, 2005] as GD data and have some small collection of manuals translated by humans, which would be our ID data.

In the context of Dasher, we would have abundant GD data (we can train on any of the widely available huge corpora). We then have an important choice to make when it comes to what we call ID data. One approach we can take is to say that all the text a single user enters is data from a single domain – we would treat the problem as adapting to that user's writing style. An alternative view stems from the argument I gave in Section 1.2 that Dasher will presumably be used as a general-purpose text-entry system – different parts of the text the user enters will come from different domains so we will have many types of ID data, effectively treating the problem as Multi-Domain adaptation.

In this thesis I limit myself to treating the problem as adapting a language model to one domain, namely the text that would be entered by a single user. The main reason is that this is a simpler problem and a reasonable starting point – while it is true that in some (or even most) cases different parts of what a user enters will come from different distributions, treating the problem as single-domain adaptation should still bring performance improvements and is a simpler formulation of the problem that can serve as a good starting point. Moreover, there is not a mechanism in the current Dasher implementation that addresses the problem that the model might have to adapt to multiple domains. In this sense, we would not be losing anything by putting a neural language model in place of the existing one.

3.1 Do we really need adaptation?

So far I have claimed that adapting language models to novel text will improve the quality of their predictions, but I have not backed these claims experimentally.

In domain adaptation problems in general, and in the problem I am viewing in specific, it is usually the case that one has plenty of OOD data and little ID data (insufficient for building a good model). The aim is then to use the OOD data to learn some general characteristics of language and the ID data for extracting domain-specific ones. At this point, a simple question arises – how well could we do if we did in fact have abundant ID data? Obtaining experimental data for this would give us an upper bound for how much performance improvement we can gain from any adaptation method. In case the gap in performance between a model trained on a lot of OOD data and that of a model trained on a lot of ID data is insignificant, perhaps spending a lot of effort on adaptation is not well motivated.

3.1.1 What do we gain from in-domain data?

In order to answer this question, I designed several simple experiments. I trained three language models on three different corpora – the collected works of Sir Arthur Conan Doyle and those of Herbert George Wells and one on the Europarl corpus(I have described the corpora I have chosen for evaluation, how I have pre-processed them and how I have split them into training, validation and test sets in detail in Section 3.3). I then evaluated all three models on the validation sets of the A.C.Doyle and H.G.Wells corpora, aiming to find out how much better a model trained on ID data performs compared to a model trained on OOD data.

I trained two LSTM language models with hidden state sizes of 512 and 1024 (More on LSTM networks in Section 2.3) and evaluated them using both static and dynamic evaluation. When I say I evaluate some text with a model I mean I obtain the average negative log probability (Bits Per Character) of the text under the model, as I discussed in Section 2.2.3. And I use the term “static evaluation” to mean I do this evaluation without performing any adaptation, using the parameters obtained during training to obtain the predictions for each character.

Dynamic evaluation is one of the methods I consider as a solution to the problem of model adaptation and I describe it in detail in Chapter 4. In short, this means that during evaluation, the model's parameters are adjusted so that the model's predictions are closer to the distribution of the data that it is being evaluated on. More specifically, for the experiments in this section, I use dynamic evaluation with an SGD learning rule – a cross-entropy loss between the predicted and true label is calculated at every step and at every 30 steps the weights of the network are updated via a Stochastic Gradient Descent learning rule with a learning rate of 0.3.

The results of this simple experiment are presented in Table 3.1. We can see that there is a gap in the performance of a model trained on ID data and that of one trained on OOD data and that can be as large as 12% (relative performance). Considering the fact that in our case ID and OOD data are not too dissimilar – they are works of fiction from two authors, both British, both worked in the second half of the 19th century – I can say that this experiment strengthens the argument for the need of adaptive models.

3.1.2 What if we had more data?

Based on the experiments in the previous section, I argued that I can motivate the need for adaptation by the gap in performance between models trained on OOD and ID data. There is one potential caveat, though. The A.C.Doyle and H.G.Wells training sets consist of 12 million characters each, which is relatively small. Presumably, if we had more data from the authors to train these models, as we increase the amount of training data, the performance of these models would improve up to a point at which the model will not benefit from more training data (Figure 3.1).

Although it is true that with the current sizes of the training set we see the ID models perform better than the OOD ones, it might be the case that if the models were given enough data to get to the point of saturation described above, that gap in performance would disappear. As there is no more data from the authors to train these models on, I could not directly test this hypothesis. What I did instead is I experimented with reducing the amount of data I used for training. The rationale behind this was that I could see whether 12 million characters of text was enough to saturate the models. I performed the exact same experiment two more times, using 6 and 9 million characters for training. I only experimented with LSTM models with hidden dimensionality of 1024.

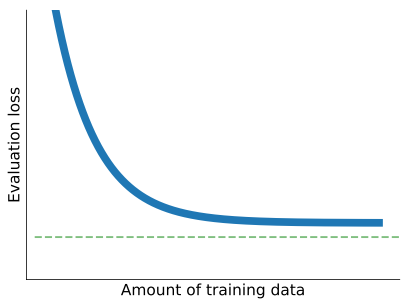


Figure 3.1: Theoretical relationship between training set size and evaluation performance

The results of these experiments are plotted in Figure 3.2. Figures 3.2a and 3.2b show the performance in bits/character (lower is better) for both models when evaluated (both statically and dynamically) on the A.C.Doyle and H.G.Wells validation sets respectively. We can notice that more training data invariably leads to a smaller loss during evaluation,

Evaluated on A.C. Doyle

Trained on	Model	Evaluation	BPC	Ratio
A.C.Doyle	LSTM 1024	Static	1.66	1.00
H.G.Wells	LSTM 1024	Static	1.85	**1.12**
Europarl EN	LSTM 1024	Static	2.68	1.62
A.C.Doyle	LSTM 1024	Dynamic	1.58	0.95
H.G.Wells	LSTM 1024	Dynamic	1.69	1.02
Europarl EN	LSTM 1024	Dynamic	1.95	1.17
A.C.Doyle	LSTM 512	Static	1.74	1.00
H.G.Wells	LSTM 512	Static	1.94	**1.11**
A.C.Doyle	LSTM 512	Dynamic	1.64	0.94
H.G.Wells	LSTM 512	Dynamic	1.76	1.01

Evaluated on H.G.Wells

Trained on	Model	Evaluation	BPC	Ratio
A.C.Doyle	LSTM 1024	Static	1.88	**1.08**
H.G.Wells	LSTM 1024	Static	1.75	1.00
Europarl EN	LSTM 1024	Static	2.67	1.53
A.C.Doyle	LSTM 1024	Dynamic	1.71	0.98
H.G.Wells	LSTM 1024	Dynamic	1.64	0.94
Europarl EN	LSTM 1024	Dynamic	1.95	1.11
A.C.Doyle	LSTM 512	Static	1.93	**1.04**
H.G.Wells	LSTM 512	Static	1.84	1.00
A.C.Doyle	LSTM 512	Dynamic	1.75	0.95
H.G.Wells	LSTM 512	Dynamic	1.73	0.94

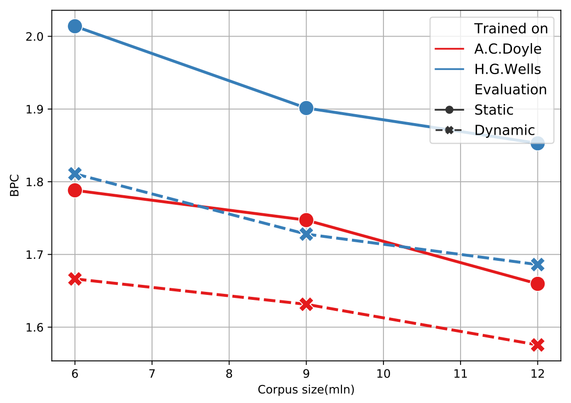
Table 3.1: Results for LSTM models with hidden state sizes 512 and 1024 trained on the works of one of A.C.Doyle and H.G.Wells and evaluated on the validation sets for both authors. The values of the Ratio column (lower is better) are the ratios of the performance of the model at that row to that of the model trained on ID data and evaluated statically. The values in bold are the ratios of the performance of a model trained on OOD data to that of a model trained on ID data (when evaluated statically). Looking at those, we see that if we had ID data to train on, we could perform to up to 12% better in these cases – this is the gap we are aiming to close with adaptation. We can also see that simple dynamic evaluation (Described in Chapter 4) is definitely helpful in adapting OOD models, in some cases even outperforming models trained on ID data and evaluated statically.

meaning that most probably 12 million characters of training text are not enough to saturate the model.

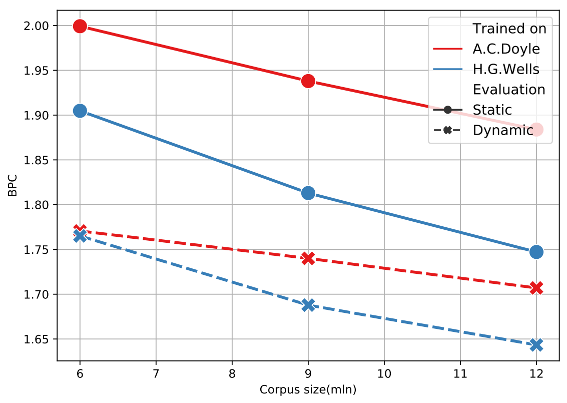
Figures 3.2c and 3.2d show the ratios of a model to that of a model of the same size trained on ID data as a function of the training set size. One thing we can notice is that these ratios do not go down as we increase the amount of data used for training. This might be interpreted as suggesting that even if we feed more training data, the relative gap in performance between models trained on ID and OOD data will not shrink.

However, I found such an extrapolation to be insufficient as an argument. This, and the fact that in some of the initial experiments the gap between the performance of ID and OOD models was as small as 4% prompted me to perform one last set of experiments to demonstrate the need for adaptation.

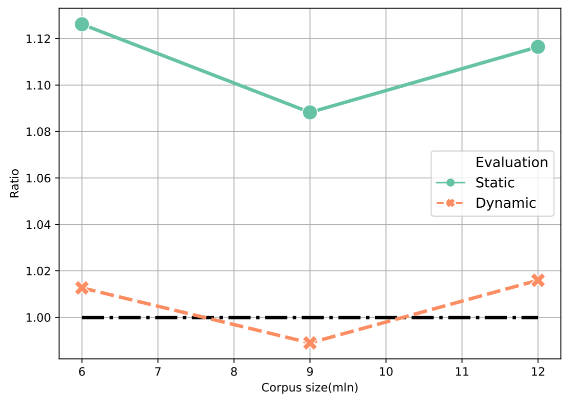
I trained an LSTM model on 60 million characters from the English part of the Europarl corpus. This is much larger than the works of A.C.Doyle and H.G.Wells and contains transcripts of speeches from the European Parliament, which differ significantly in style from 19th century British prose. I then evaluated the model on the validation sets of the corpora from the two authors using both static and dynamic evaluation. The results are also listed in Table 3.1. We can see that even though the training corpus used is larger, the performance of the model is much worse. We observe that even using dynamic evaluation does not compensate for the difference in the underlying distributions – when evaluating a text from one author, even with dynamic evaluation, the model trained on Europarl incurs a bigger loss than a statically evaluated model that was trained on text from the other author.



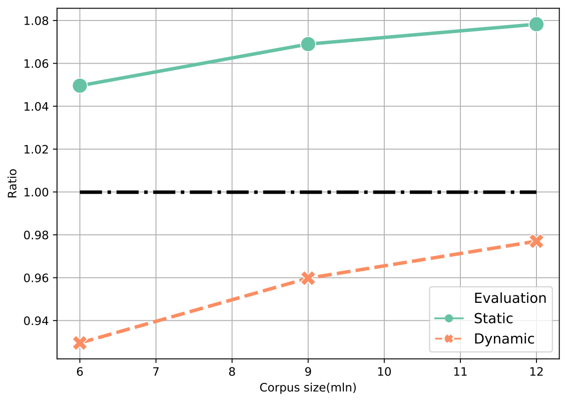
(a) Models performance. Evaluated on A.C.Doyle



(b) Models performance. Evaluated on H.G.Wells



(c) GD/ID ratio. Evaluated on A.C.Doyle



(d) GD/ID ratio. Evaluated on H.G.Wells

Figure 3.2: **(Best viewed in color)** Performance for LSTM 1024 models trained and evaluated on two authors (3.2a and 3.2b) and ratios of the performance of the OOD model to that of the ID model (3.2c and 3.2d). The top two plots show the performance in BPC when evaluated on A.C.Doyle **(Top Left)** and on H.G.Wells **(Top Right)** when different amounts of training data has been used. We see that when the models are trained on more data, we get better performance. The bottom two figures show the ratio of the performance of a model trained on one author and evaluated on the other, again using both static and dynamic evaluation, again for different amounts of training data. For example, the bottom left plot shows the performance of a model trained on H.G.Wells and evaluated on A.C.Doyle. If the lines on the bottom plots had negative slopes, this would suggest that as we train models on more data, the gap between models trained on ID and OOD data shrinks and so we could solve adaptation by just feeding the model a lot of OOD data. These experiments suggest that this is not the case.

3.2 One versus many models for Dasher

So far I have argued that if we have trained a model on a lot of OOD data it is worth putting in effort to adapt it to the specific domain it is going to be used in. Also, as I discussed in the introductory Section 1.2, Dasher's case is less trivial because the system will presumably be used for entering text from many different domains. As I noted in the beginning of the previous section, I am treating the problem as single-domain adaptation (adapting to a specific user). However, an obvious future direction would be to address the challenge of multiple domains. In this section I offer a discussion on how this should be approached. I see three high-level approaches to tackling the problem of multi-domain adaptation in Dasher.

Option One – Interpolating between multiple domain-specific models is to train multiple domain-specific models and at test time estimate the probabilities we need by interpolating between the predictions of all models.

$$P(c_i|c_1 \dots c_{i-1}) = \sum_{d \in D} \lambda_d P_d(c_i|c_1 \dots c_{i-1}) \tag{3.1}$$

subject to $\sum_{d \in D} \lambda_d = 1$ where d ranges over all domains we have trained models on, P_d is the probability assigned by the model for domain d and λ_d is the interpolation weight assigned to that model. These weights could be set heuristically or based on a sample of text provided by the user. They could also be estimated by a separate model, say a neural network (This is the idea behind *Mixture of experts* [Jacobs et al., 1991]) and perhaps dynamically adapted to recent history while the user is typing. Other ways to interpolate between the predictions can also be used.

Regardless of how we interpolate between the models and how we choose the weights, it is still the case that we would have to identify multiple domain-specific corpora and train, maintain and ship multiple models. But most importantly, the time and space complexity of the resulting model would be that of a single language model multiplied by the number of domains we identify. In the first part of my project I showed that getting a single neural model to work efficiently in Dasher would be a non-trivial task. Having many of these making predictions in parallel would potentially impose unrealistic demands on the hardware a user needs to run the system, which is clearly undesirable.

Option Two – Let the user choose a model is to train many domain specific models like in Option One but let the user choose which ones they need – say, if they know they will be using Dasher mainly for casual conversation and for writing up their work in Molecular Chemistry, they can select suitable options for language models when downloading Dasher. While using Dasher, the interface would allow them to choose what type of text they will be entering and the corresponding model will be used. Alternatively, these discrete switches could be made automatically based on recent history. In this way the need for many domain-specific models to run in parallel like in Option One is eliminated. This is similar to what the user can do in Dasher at the moment – since the current model's parameters are estimated on some piece of text every time the system is started, the user can provide any text for that to be done on.

With this option we are putting extra responsibility on the user – it is likely that Dasher's users would want to use the system in many contexts and switch between them often. Forcing them to change Dasher's mode every time they switch context would potentially make for a frustrating user experience.

However, if we assume that the user is willing to do this, we have to solve another problem – unlike the current language model, training an RNN is requires a lot of data and time and so cannot be done every time the system is started. We would either have to have many pre-trained domain-specific models and/or provide the user with the option to train a model on any corpus they choose.

A problem with having pre-trained domain-specific models is that we would again have to identify multiple domain-specific corpora and would have to make sure they cover as much of the space of possible domains as possible.

If choose to let the user train their own model, we would have less control over the training process and no control over what data they use so it will be difficult to make any guarantees on the performance of the resulting model.

Option Three – Have a single model that can adapt to multiple domains This option involves doing away with training multiple language models. Instead, we would train a single model on a lot of data from various domains. At test time, this model would be adapted to whatever text is being entered in an unsupervised fashion, meaning that it will be given no explicit information about the domain of said text. This approach has some obvious advantages to the previous two – we would not need to train, store and distribute multiple models, we would not ask the user to think what they might want to use the system for, and we would only have one model occupying computational resources on the machine Dasher is being used. I believe this is the smartest and most elegant of the approaches I discussed. Moreover, developing a general model that can adapt to multiple domains is a non-trivial task and an open research area. For these reasons, when choosing what solutions to investigate, I keep in mind that in an ideal scenario Dasher would have one model that can adapt to multiple domains.

3.3 Datasets for adaptation

3.3.1 Datasets I use for experiments

Europarl The Europarl corpus [Koehn, 2005] contains documents from the proceedings of the European parliament in eleven languages. In my experiments I use the first 100 million characters of the English version of the corpus. I split them 90-5-5 into training, validation and test data, which is standard in literature for this dataset. I chose this dataset since it is often used to benchmark language models and it is a large collection of text from a single domain.

text8 This is a dataset first proposed by [Mikolov et al., 2015] and then widely used to evaluate language models in literature, including two of the adaptation approaches I consider – Dynamic and Sparse Dynamic evaluation, which were proposed by [Krause et al., 2018]. It is a subset of a snapshot of Wikipedia, which has been stripped from metadata, converted to lowercase and pre-processed to contain only the lowercase Latin letters, dots and spaces. It contains 100 million characters and I also split it 90-5-5 into training, validation and test sets. I chose it because in Dasher the user can choose to use alphabets of different sizes, including an alphabet consisting of lowercase Latin letters and results on this dataset would give me a good proxy for how the methods I assess will handle small alphabets.

Enron emails The Enron Emails corpus [Klimt and Yang, 2004] contains internal emails between around 150 employees of Enron, which were made public during an investigation into the company after it declared bankruptcy in 2001.

The emails in the dataset contain a lot of metadata and forwarded emails and replies usually contained the original message. I had to perform some cleaning steps before I obtained a version that contained just bodies of the emails, without repetition of the original messages.

I originally chose this corpus because the motivation behind my project is adapting to different user's writing styles and in this corpus I could separate emails by authors. Even though I did not manage to perform experiments on adapting to specific users from the corpus, it still served me in my experiments as a big body of domain-specific text with a rich alphabet.

3.3.2 Choosing datasets for Dasher

Here I briefly lay out some considerations for choosing corpora for training our models.

Various types of text If we choose to have one big model that can adapt to multiple domains we would probably need to make sure we expose it to data from as many domains as possible during training.

Licensing We need to make sure that whoever prepared the datasets we use allows for their use in open-source software. If we use data that was collected from different people, for example chat room conversations, we need to make sure that when it was collected those people gave their consent to the data being used in the way we intend to.

Bias The topic of algorithmic bias is becoming more and more prevalent in the debate around systems that do any form of automated reasoning based on data. We know that these algorithms exploit statistical regularities in data and are thus likely to reproduce any human bias that is present in that data. This problem is being widely researched in the context of Natural Language Processing. We know that algorithms can reflect racial [Blodgett and O'Connor, 2017] and gender [Lu et al., 2018] biases in the data they were trained on. This is an important consideration to make when choosing data for our models to be trained on – we definitely do not want for our model to assign higher probabilities to negative words if an African-American name is present in the sentence [Caliskan et al., 2017], for example.

Chapter 4

Approaches to adaptation

There is a zoo of approaches and architectures addressing the problem of domain adaptation and I discuss some of them in detail in this section. Before I do that, I offer a brief discussion of what I think are the main challenges one needs to address while developing an adaptive model for Dasher.

4.1 Challenges and concerns

Data If a model is to be able to adapt to a broad range of topics, it would ideally have been exposed to text from those during training. This means that a lot of data has to be collected, pre-processed and potentially labeled with topics, which would amount to a serious effort. For the purposes of my project I will not concern myself with collecting enough data to cover all possible domains. However, I will need to keep this point in mind and think of how the models I described will be trained on huge amounts of data and how the approaches I devise might perform if the number of domains they have to adapt to is significantly bigger.

Supervised versus non-supervised training In an ideal scenario, we would not need to label the data we feed to the model during training with domain information. If I find that two approaches achieve similar results but one does not require us to tell it what exactly we are feeding it during training, I will probably want to choose this one.

Time and space complexity One of the motivations behind going for a single-model approach is that it will require less memory and CPU time than interpolating between multiple models, for example. When choosing a solution, I have to think of how much memory and computational operations it requires and how it can be adapted to Dasher's particular situation, namely the need for building a tree of predictions in real-time.

Instability RNN models can be seen as a dynamic systems with the network's weights governing their dynamics. As such, they are prone to becoming destabilized – [Sengupta and Friston, 2018] analyzed the spectra of the weight matrices that parametrize the hidden state transition functions in several RNN variants trained to perform different tasks and found that "a tiny perturbation of the weight matrix can completely alter the spectral signature of the underlying recurrent network, making these networks non-robust". Any method that modifies this matrix would potentially be more prone to causing instability in the network.

With these concerns in mind I chose to explore two methods of adapting language models which I describe in the next two sections – Dynamic evaluation [Mikolov et al., 2010, Krause et al., 2018] and Learning Hidden Unit Contributions (LHUC) [Swietojanski et al., 2016]. Although I apply them to LSTM language models in my experiments, both approaches can be used with any RNN and in fact with any neural network.

4.2 Dynamic evaluation

Dynamic evaluation is one idea for a solution to the problem of adapting a model to shifts in the generating distribution during inference. It was first proposed by [Mikolov et al., 2010] as a method for adapting language models based on Simple Recurrent Networks, but was not the main focus of the publication. In my thesis I mainly refer to the work of [Krause et al., 2018], which explores dynamic evaluation in depth and proposes changes which improve on the approach described by [Mikolov et al., 2010].

4.2.1 How it works

The material in this section follows the treatment of Section 3 in [Krause et al., 2018].

The idea behind Dynamic evaluation is to adapt θ_g, the parameters of the network estimated at training time, to $P_l(x)$, the generating distribution for a local sequence of the test data. The aim is to learn parameters θ_l, which reduce the cross-entropy between the predictions and $P_l(x)$.

For the purposes of Dynamic evaluation, we divide a sequence of inputs $\{x_1, x_2 \dots x_t\}$ into shorter sequences $s_1, s_2, \dots s_M$ of length n:

$$s_1 = x_{1:n},\ s_2 = x_{n+1:2n},\ s_3 = x_{2n+1:3n},\ \dots,\ s_M \tag{4.1}$$

After processing each s_i, we update the adapted parameters via a gradient descent step to obtain a new set of parameters θ_l^i, where we set the initial adapted parameters θ_l^0 to the global parameters θ_g. Updating the parameters after making a prediction on a short sequence is analogous to training an RNN with truncated back-propagation through time (TBPTT), which I describe in Section 2.3.3. Like in TBPTT, the length of the sub-sequences is a hyperparameter, which needs to be tuned.

In order to perform the update step at time i, we take the predictions the network made for s_i, which I denote by $\hat{P}(s_i;\theta_i)$. We compute $\mathcal{L}(s_i)$, the cross-entropy loss between the predictions and the targets and $\nabla\mathcal{L}(s_i)$, its gradient with respect to the model's weights and then use that to perform a gradient update step, just like in TBPTT. In Dasher, the targets would be the characters that the user entered by navigating into the corresponding boxes (See Section 2.1.1 for a reminder of how text is entered using Dasher).

4.2.2 Update rules

After we have computed the predictions and gradients for a sequence s_i we obtain the next setting for the local parameters θ_i by applying an *update rule*. The original update rule, proposed by [Mikolov et al., 2010] is a simple gradient descent step:

$$\theta_i \leftarrow \theta_{i-1} - \eta\nabla\mathcal{L}(s_i) \tag{4.2}$$

where, like in training, η is called the *learning rate* and is a hyperparameter which needs to be tuned.

The first change to the original method that [Krause et al., 2018] propose is adding what they call a global decay prior. The main motivation behind this is based on the assumption that we would like to put more importance on recent history when adapting the parameters. The update rule, which the authors call SGD (for Stochastic Gradient Descent) with a global prior achieves that by decaying the contributions of the gradients at a given timestep exponentially over time:

$$\theta_i \leftarrow \theta_{i-1} - \eta\nabla\mathcal{L}(s_i) + \lambda\left(\theta_g - \theta_l^{i-1}\right) \tag{4.3}$$

The hyperparameter λ is called the *decay rate* and it also needs to be tuned. Setting it to 0 results in the simple update rule given in Equation 4.2. One of the claims made by [Krause et al., 2018] is that setting λ to a value bigger than 0 helps with test-time performance. However, as I discuss in 5.3.2, my experimental results suggest that this is not always true and depends on the type of text being evaluated.

The next modification the authors propose is an update rule inspired by the RMSprop learning rule [Tieleman and Hinton, 2012], which I describe in Section 2.3.3. RMSprop scales the update for each weight by a moving average of the squared gradients at all previous timesteps. The authors argue that "in dynamic evaluation, near the start of a test sequence, RMSprop has had very few gradients to average, and therefore may not be able to leverage its updates as effectively". However, they provide no experimental evidence to support this claim. Because of time constraints, neither do I.

Assuming this is indeed an issue, the authors propose solving this problem by collecting mean squared gradients on the training data, rather than on the test data. After we finish training the model, we make another pass through the training data, making predictions and recording the squared gradient for each batch. The mean squared gradient estimated on the training data is then given by

$$MS_g = \frac{1}{N_b} \sum_{k=1}^{N_b} (\nabla \mathcal{L}_k)^2 \tag{4.4}$$

where $\nabla \mathcal{L}_k$ is the gradient on the kth batch and N_b is the number of batches. The number of batches depends on the batch size we choose, which can be different from the batch size used in training and is another tunable hyperparameter. The resulting update rule, which the authors call RMS with a global prior is then given by

$$\theta_l^i \leftarrow \theta_l^{i-1} - \eta \frac{\nabla \mathcal{L}(s_i)}{\sqrt{MS_g} + \varepsilon} + \lambda \left(\theta_g - \theta_l^{i-1} \right) \tag{4.5}$$

where ε is a stabilizing parameter, like in the original RMSprop learning rule I described in Section 2.3.3. After inspecting the code[1] provided by [Krause et al., 2018], I set that to the same value used by them, which is 2×10^{-5}.

As a final alteration, the authors suggest to "scale the decay rate for each parameter proportionally to $\sqrt{MS_g}$, since parameters with a high RMS gradient affect the dynamics of the network more.". The update rule they call RMS with an RMS global prior is given by

$$\theta_l^i \leftarrow \theta_l^{i-1} - \eta \frac{\nabla \mathcal{L}(s_i)}{\sqrt{MS_g} + \varepsilon} + \lambda \left(\theta_g - \theta_l^{i-1} \right) \odot RMS_{\text{norm}} \tag{4.6}$$

where

$$RMS_{\text{norm}} = \max \left(\frac{1}{\lambda}, \frac{\sqrt{MS_g}}{\text{avg}\left(\sqrt{MS_g}\right)} \right) \tag{4.7}$$

Clipping RMS_{norm} to $\frac{1}{\lambda}$ is done to ensure the decay rate does not exceed 1.

4.2.3 Sparse Dynamic evaluation

The authors of [Krause et al., 2018] propose one more solution to adaptation, which is less memory-intensive. They motivate the need for this by the necessity to store different parameter settings for each batch sequence when doing dynamic evaluation on a batch of inputs, if doing Dynamic evaluation on the full set of parameters (full Dynamic

[1] `https://github.com/benkrause/dynamic-evaluation`

evaluation). Adapting the network's parameters to a body of text provided by the user before they start using Dasher can be computationally expensive and time-consuming. Doing so in batches is one approach to speeding the process up and sparse Dynamic evaluation can reduce the memory cost associated with this. Moreover, while text is being entered, we might need to store the adaptation parameters and reuse them if the user deletes text. Sparse Dynamic evaluation has less parameters and will require less memory in this scenario, too. I discuss the implementation issues related to adapting to existing user text and performing adaptation while text is being entered in Section 4.4. Finally, this method does not modify the weights learned during training, potentially making it more stable.

They call the method, perhaps misleadingly (No operations with sparse matrices are involved), *sparse dynamic evaluation*. Instead of adapting θ_g, the parameters estimated at training time, the idea is to introduce a new adaptation matrix, $\mathcal{M}$, which is used to modify the hidden state of the network at every timestep t:

$$\mathbf{h}'_t = \mathbf{h}_t + \mathcal{M}\mathbf{h}_t \tag{4.8}$$

The modified hidden state $\mathbf{h}'_t$ is then used instead of $\mathbf{h}_t$ in both computing the output at timestep t and in computing the next hidden state. All elements of the matrix $\mathcal{M}$ are initialized to 0, which means that in the beginning of adaptation, the predictions of the network are not modified ($\mathbf{h}'_t = \mathbf{h}_t$). Like in previous approaches, after processing every short sequence of inputs s_i, we calculate the cross-entropy loss for the predictions and use its gradient with respect to $\mathcal{M}$ to update the adaptation parameters using gradient descent. However, they do not specify the update rule they use. In my experiments I first assume they use a simple SGD update rule:

$$\mathcal{M} \leftarrow \mathcal{M} - \eta\nabla\mathcal{L}(s_i) \tag{4.9}$$

After some unsuccessful initial experiments, I also add an L2 penalty to the loss :

$$\mathcal{L}(s_i) = \mathrm{NLL}(s_i, \hat{s}_i) + \lambda\|\mathcal{M}\|_F^2 \tag{4.10}$$

where $\mathrm{NLL}(s_i, \hat{s}_i)$ is the negative cross-entropy between the predictions and the targets for the ith sub-sequence and $\|\mathcal{M}\|_F^2$ is the squared Frobenius norm – the sum of the squares of all elements of $\mathcal{M}$, and λ is a hyperparameter controlling the strength of the L2 penalty.

Perhaps the reason this approach is called *sparse* is that the authors propose to only change a subset of H units of the hidden state, making $\mathcal{M}$ an $H \times H$ matrix and drastically reducing the number of parameters used for adaptation. The authors do not specify how to choose the subset of hidden units to be modified. One can argue that this does not matter since the ordering of units in the hidden state is arbitrary. While that is true, it may be the case that for a given setting of the parameters of the network it might be more beneficial to change the activations of a given set of units than changing the activation of another set, since different units in a neural network are sensitive to different patterns in the data. However, because of the lack of promising results in my initial experiments with sparse dynamic evaluation, I do not explore this line of questioning further.

4.3 Learning Hidden Unit Contributions (LHUC)

Learning Hidden Unit Contributions (LHUC) is another approach to adapting neural networks to shifts in the generating distribution between training and test time. It was originally proposed by [Swietojanski et al., 2016] as a method for adapting automatic speech recognition systems to specific speakers, but has also been shown to be effective in adapting Machine Translation systems [Vilar, 2018]. Parts of my description of LHUC in this section follow the treatment of [Vilar, 2018].

The method can be applied to any neural network and involves scaling the activations of different units in its hidden layers by different amounts, essentially amplifying or dampening their effect on the output of the network. The intuition behind the approach is based on the observation that different hidden units respond to different signals in the input. For instance, [Le et al., 2011] trained an autoencoder on a set of frames from YouTube videos and later found that one particular neuron in the activations of the network was particularly discriminative for faces – it tended to have a high activation value when a face was present in the image that was fed to the network, and a low one when one was not. In a similar, perhaps more relevant to this thesis experiment, [Radford et al., 2017] trained an RNN-based character-level language model on a set of product reviews and found that the value of one neuron in the hidden state of the network was a good predictor for the sentiment of the text that was being processed by the network. This blog post[2] by Andrej Karpathy also contains several examples of particular neurons in a character-level language model being responsive to different signals in the text.

It is, then, not unreasonable to assume that some of these hidden units will be more important than others when making predictions for a given generating distribution at test time. For example, if Dasher's RNN language model has a neuron that responds to sentiment, we would imagine that if someone uses Dasher to write a lot of product reviews, that neuron's activation should be more relevant to the output. In the next section, I describe how LHUC captures this intuition.

4.3.1 How does LHUC work?

As I mention above, the approach is general and can be applied to any neural network, but, because of the nature of my thesis, here I focus on how it works in an RNN language model.

[2]https://web.archive.org/web/20190321235642/https://karpathy.github.io/2015/05/21/rnn-effectiveness/

As a reminder, each hidden state of an RNN is computed from the hidden state at the previous timestep and the current input (More details back in Section 2.3):

$$\mathbf{h}_t = f\left(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta_h\right) \tag{4.11}$$

Much like sparse dynamic evaluation (Section 4.2.3), LHUC substitutes the hidden state with a modified version given by

$$\mathbf{h}_t^{(LHUC)} = \mathbf{h}_t \circ a(\mathbf{r}) \tag{4.12}$$

where $\circ$ denotes element-wise multiplication (Hadamart product), $\mathbf{r}$ is a vector of the same dimensionality as $\mathbf{h}_t$, the elements of which I will refer to as *LHUC scalers* and $a(x)$ is an activation function, applied element-wise, usually a sigmoid function with amplitude 2 (so values larger than 1 would increase the contribution of a neuron and values smaller than 1 would inhibit it):

$$a(x) = \frac{2}{1 + \exp(-x)} \tag{4.13}$$

The resulting modified hidden state $\mathbf{h}_t^{(LHUC)}$ is used to produce the output of the network at time t. Whether $\mathbf{h}_t^{(LHUC)}$ or $\mathbf{h}_t$ is used in computing the next hidden state is a modelling choice. Possible downsides of using the modified state is that doing so could destabilize the dynamics of the system.

Referring back to the example of the sentiment neuron above, if a Dasher user is currently writing a review of a product and the ith unit in the hidden state is responsive to sentiment, perhaps the ith unit of $a(\mathbf{r})$ will be bigger than 1, increasing the contribution of the ith neuron to the output of the network.

What I have not said so far is how to actually use this approach in order to adapt a model. The parameters which we can tune at adaptation time are the LHUC scalers, $\mathbf{r}$. All of the elements of $\mathbf{r}$ are initialized to 0, resulting in $a(\mathbf{r})$ being a vector of ones, which means that initially the predictions of the network will remain unchanged.

Exactly like in Dynamic evaluation, when performing LHUC adaptation, we can think of the input sequence being broken up into shorter sub-sequences $s_1, s_2 \ldots s_M$, the length of which is a hyperparameter for the LHUC adaptation process. After processing each sub-sequence, we compute $\mathcal{L}(s_i)$, the cross-entropy loss between the predictions and the targets. Next, since all operations in Equation 4.12 are differentiable, we can compute $\eta\nabla\mathcal{L}(s_i)$, the gradient of the loss with respect to $\mathbf{r}$. We then update $\mathbf{r}$ by taking a gradient step in the direction of minimizing the loss:

$$\mathbf{r} \leftarrow \mathbf{r} - \eta\nabla\mathcal{L}(s_i) \tag{4.14}$$

As in Dynamic evaluation, η is called the *learning rate* and is, again, a tunable hyperparameter.

4.4 How can these methods be implemented in Dasher?

Because of the differences of RNN-based language models to the current language models in Dasher, we would have to make some changes to the process of adaptation. In this section I mention some of the major concerns in this area.

4.4.1 Hyperparameter tuning

As noted previously, unlike the current language model in Dasher, the parameters of which are estimated anew every time the system is launched, an RNN language model would need to be trained beforehand and the resulting weights would have to be stored in the cloud and shipped with the software when a user downloads it. Using Dynamic evaluation or LHUC would entail some additional server-side work. Firstly, the hyperparameters for the adaptation method we choose would need to be tuned and secondly, if Dynamic evaluation with an RMS update rule is used, the estimates of the per-parameter mean squared gradients in Equation 4.4 would need to be made on the server and also shipped alongside the weights.

4.4.2 Adapting to text provided by the user

When a user opens Dasher for the first time, the predictions which dictate the layout of the screen can be given by an RNN model with the original downloaded weights. Another option would be to allow the user to provide some text they have entered before if they have any. We would then perform dynamic evaluation on that text, adapting the model's parameters to it – this should result in the model assigning higher probabilities to the strings the user enters when they start using Dasher, compared to using the original weights.

A decision has to be made on whether this evaluation should happen on the user's machine or on a server. If it is possible to do so, performing this adaptation on a user's machine would be preferable for two reasons: Firstly, there would be no need for maintaining the server infrastructure needed and paying the related costs. And secondly, adapting the model on a remote machine would mean that the user would be forced to send us potentially sensitive information. Currently, Dasher is a stand-alone open-source piece of software that does not have a team of permanent developers or even a strong developer community around it. Adding any maintenance cost to the system should ideally be avoided. An alternative would be to provide the user with the option to pay for computation time on a cloud-based service, where the model would be adapted and the final weights would be downloaded to the user's machine. Ideally, we would also avoid a situation in which the user has to spend money in order to use the full functionality of Dasher so this is not an ideal solution either.

This brings us back to the question of whether we can perform initial adaptation on a chunk of user text on their own machine. This depends on how much text the user wants to adapt the weights to, which adaptation method we use and what hardware the user has available. For instance, adapting the parameters of an LSTM model with 1024 hidden unit, using Dynamic evaluation on a 5-million-character-long piece of text, using a modern GPU, takes around 40 minutes. Doing that on a powerful CPU takes around 4 times longer. This time can be reduced by performing adaptation using mini-batches (Discussed in Section 2.3.3), especially if using a GPU. I note that performing mini-batch adaptation would presumably require further hyperparameter tuning as the size of the mini-batches will be a new hyperparameter. Five million characters is a significant amount of text – the Latex source code for this document is around 200 thousand characters long, for example – we would expect the average user to provide a smaller volume of text. Given that this adaptation would only have to happen once, it should be feasible to perform it on the user's machine.

4.4.3 Adaptation during use

Next I consider how adaptation will be performed while the user is entering a string.

How and when do we update the parameters? Currently, the parameters of the language are just the number of times different contexts have been observed (See Section 2.2.2). These are updated every time the user enters a character by moving the crosshairs into the box that corresponds to that character (See Figure 2.1).

When using any of the methods of this chapter, we would want to update the weights of the RNN after processing a certain number of characters, which is a hyperparameter for the adaptation procedure. However, this is not as simple as with the current language model. We would have to first perform back-propagation to the start of the current sub-sequence, which takes time linear in the length of the sequence. The computational cost of adapting the parameters to a single sequence is roughly double that of just evaluating the sequence – propagating the gradients back through the network involves the same number of operations as the forward pass and there is a small overhead from updating the weights. However, overall adaptation will not double the computation cost compared to static evaluation. The main computational cost in Dasher comes from the fact that a tree of possible continuations of the current input has to be built (See Section 2.1) and so we need to make many forward passes through the network in order to make predictions in all possible contexts. However, when adapting the weights, we would only need to back-propagate through the branch that corresponds to the string entered by the user.

What happens when characters are deleted? A user deletes a character they have entered by moving the crosshairs from of the box corresponding to that character to either its parent box or to one of its sibling boxes, in which case the last character is replaced by the character in that sibling box. When that happens, it is easy for the current model to update its parameters based on it – it just changes the appropriate counts for the contexts it has observed.

However, doing this with an RNN-based language model using one of the methods proposed in this chapter can be more complicated. Let us assume we update the RNN's weights after every 10 steps. After the user enters the 10th character in the string, we perform back-propagation to the beginning of the sub-sequence and update the adaptation parameters. Consider what happens if the user then deletes the last character and enters another. This would be a common situation, since when using the Dasher interface, a user trying to navigate to a given box often has to move the crosshairs through its siblings, effectively entering and then deleting the corresponding characters. If we have updated our parameters based on a string that the user did not want to enter, we should undo this operation or otherwise we would have adapted to the wrong string.

One approach to doing this would be to do the following: when updating the parameters to θ_t after processing sub-sequence s_t, we keep the old set of parameters θ_{t-1} in memory and store a reference to it. When characters are deleted, we would use the stored parameters to make predictions instead of the most recent ones. Let us say a user deletes characters up to the point where the string they have entered is n characters long. We would then use θ_k where $k = \lfloor \frac{n}{m} \rfloor$ and m is the length of the sub-sequences we use during adaptation. If the user enters a very long string, we could assume they will not delete most of it and "forget" the oldest sets of parameters, which is what Dasher does at the moment, decreasing the memory required to store the parameter back-ups.

A way in which we can decrease the amount of computation required would be to not consider a character as "truly entered" until a user enters several more characters following this one by navigating deeper into the tree. If we update the parameters only when a character is "truly entered", we would alleviate the need to constantly switch parameters when navigating over sibling nodes in the scenario described above. This would also mean that we are adding one further caveat to the ones listed in Section 2.1.4 when talking about the proportionality of the Bits Per Character the model achieves on a string and the time required to enter the string. In my experiments, after updating the adaptation parameters at timestep t, I use the new parameters to predict the character at timestep $t+1$ and in the method I propose in this paragraph this is not the case. Since we are trying to solve the problem of moving between sibling nodes, we would not need to wait too long to consider a character as "truly entered" and so the results we get if we do so should not be greatly different, I have not proven this claim experimentally.

How expensive storing and reloading the parameters will be depends on which adaptation method we use and on the length of the sub-sequences we use for adaptation. Let us say we have an LSTM language model with 1000 hidden units (In my experiments I get competitive language modelling performance using a networks with 1024 hidden units). Let us also assume we are adapting a subset of 200 units if we use sparse Dynamic evaluation ([Krause et al., 2018] report good adaptation performance when adapting less than 20% of the units). Assuming we use single-precision floating-point numbers to represent parameters, the amount of memory required to store the adaptation parameters depending on the method is as follows:

- LHUC – we only need one vector of the size of the hidden state: 4KB
- Sparse Dynamic evaluation – we store a 200×200 matrix, so that would require 160 Kilobytes

- Dynamic evaluation – Since we are adapting all of the network's parameters, we would need to back-up all of them. The total number of parameters depends on the alphabet size. Let us assume the biggest alphabet size for Dasher, which is of size 105. In this case we would need just upwards of 18 Megabytes to store the full set of parameters.

This means that especially if we are using Dynamic evaluation to adapt all of the network's parameters, we would have to use relatively long sub-sequences in order to make adaptation computationally feasible.

What happens when a user exits Dasher Currently, when the user closes the Dasher window, they can decide whether or not to append what they have just written to the file used for estimating the language model's parameters every time the system is started. Essentially they are choosing whether the model will consider what they wrote in this Dasher session when they open a new one.

When using an RNN-language model we will not be training it on text every time the system is started, as discussed previously. Instead, we will load the model's weights from a file stored somewhere on the user's system. In order to provide the functionality described in the paragraph above, when a user quits Dasher, we can either keep the old parameters or overwrite them with the new ones, obtained via adaptation during the current session.

4.4.4 When adaptation goes wrong

In my experiments, I found that when adapting an RNN-language model using any of the techniques described in this chapter, what sometimes happens is that the model's state is destabilized, it starts outputting very poor predictions and the BPC loss it suffers explodes. I discuss this in Chapter 5 and study a specific example of how it happens in Section 5.2. What is relevant to the discussion here is the fact that after this happens, the newly obtained adaptation weights are unusable and that I have not found a definitive cause for this problem.

This unpredictable behaviour can cause problems both while adapting the model to a piece of text the user provided and while we are adapting the model as the user is entering text.

What if this happens while adapting to user-provided text? As I discussed above, adapting the model to some text the user provides before starting to use Dasher can be computationally expensive and could even involve the user paying for a cloud-based service. If it is possible for the loss to explode during this process, we cannot guarantee the user that they will get good adapted weights. Spending time and/or money on this and getting nothing in return would surely be a very unpleasant user experience.

What if this happens while the user is entering text? If this happens during a Dasher session, the user will get noticeably poor predictions. One could argue that this is less problematic since they can just restart the interface, choosing not to save the "broken" adapted weights. However, when they restart the interface and try to enter the same string, they would run into the same problem. Even though this seems to happen for quite specific strings, it is still quite problematic.

How could we solve this? Given that I do not know for sure why exploding losses happen, I do not have a clear answer for this question. Based on my hypothesis of what are the causes, I have ideas for potential solutions, none of which can guarantee this issue will not occur at all. I also note that I have not evaluated these ideas experimentally.

Assuming this happens for very specific, rare strings, before adapting the model to text provided by the user, we could pre-process the text so as to remove such strings. We could first evaluate the text without adapting the model's weights and then remove sub-strings which have extremely low probability under the model. We could also do this in steps – pre-process a chunk of the text, adapt the model to it, pre-process the next chunk using the new weights and so on. This should alleviate the potential problem of removing sub-strings that have low probability just because the distribution of the user text is very different from the distribution of the training text.

Splitting a provided user text into batches and performing adaptation, as suggested earlier in this section could also alleviate the problem since we would be averaging errors over many examples in a batch, which should lead to getting less extreme gradients. This has the already mentioned downside of having to tune another hyperparameter, namely the batch size.

A possible, although ad-hoc solution for losses exploding while text is being entered is to store checkpoints of the adapted weights and substitute them in when the adaptation starts going wrong. While adapting, we could store copies of previous parameters in memory. We should be able to detect when adaptation is going wrong – for instance, if the model is confidently wrong for a few timesteps in a row, that would be a good indicator. At this point, we can change the "bad" adaptation parameters for the ones we stored some timesteps ago and refresh the screen. This would probably make it obvious to the user that something had just gone wrong but should mean that the model destabilizing will not render the system unusable.

Judging from the examples of exploding losses I studied, a likely cause for the problem is the model being confidently wrong for several timesteps in a row, causing it to suffer big losses and therefore make big changes to the parameters because of the big values for the gradients. Assuming this is correct, an idea for a more principled solution is to discourage the model from making very confident predictions. To some extent, Dasher already does this. As discussed in Section 2.1.4, Dasher adds a small amount to the probability of each character and then re-normalizes the probabilities in order for the user to be able to enter characters which are assigned a very low probability by the language model. One for how to discourage an RNN-model from making confident predictions is to add an extra term to the loss function that is the entropy of the predicted distribution. Adding this term to the negative log loss, [Pereyra et al., 2017] found that it improves the performance of models in several domains.

Chapter 5

Adaptation experiments

My aim for this year's project is to compare methods to adapting RNN-based language models in the context of Dasher.

In Section 2.2, in which I presented the relevant elements of language modelling, I also discussed how language models are evaluated. I stated that the main metric used for evaluating language models is the average negative log probability assigned by the language model to a character in the text that is being evaluated. I also mentioned that this metric is called Bits Per Character (BPC) and that it is suitable for evaluating language models in the context of Dasher, since in Dasher the time needed to enter a string is nearly proportional to the BPC of that string under the language model the system uses. This is why in this section when I compare two approaches, I compare them in terms of the BPC they achieve on some piece of text.

In most of my experiments I talk about the improvement some method makes over static evaluation. Like in Chapter 3, when I say I evaluate a text under a model using static evaluation I mean I measure the BPC the model achieves on this text without performing any adaptation.

5.1 Hyperparameters for Dynamic evaluation

When presenting Dynamic evaluation in Section 4.2, I mentioned several hyperparameters for the different variants of the method. In this section I present experiments in which I tune these. The reason for doing so is three-fold. Firstly, if any of the methods is to be used in Dasher, it would be useful to know what hyperparameters it is worth spending more time optimizing. Secondly, I do this in order to make a fair comparison between the different Dynamic evaluation variants and LHUC. And lastly, in the first part of my project, I ran several experiments with Dynamic evaluation with the "SGD with a global prior" update rule (For the reader's convenience, I have copied the different update rules in Table 5.1), which showed some surprising results. Namely, I obtained better adaptation performance when I set the decay parameter λ to 0, which seemed to contradict the findings of [Krause et al., 2018] but because of time constraints I did

Rule name	Parameter update equation
SGD with a global prior	$\theta_i \leftarrow \theta_{i-1} - \eta\nabla\mathcal{L}(s_i) + \lambda\left(\theta_g - \theta_l^{i-1}\right)$
RMS with a global prior	$\theta_l^i \leftarrow \theta_l^{i-1} - \eta\frac{\nabla\mathcal{L}(s_i)}{\sqrt{MS_g}+\varepsilon} + \lambda\left(\theta_g - \theta_l^{i-1}\right)$
RMS w/ an RMS global prior	$\theta_l^i \leftarrow \theta_l^{i-1} - \eta\frac{\nabla\mathcal{L}(s_i)}{\sqrt{MS_g}+\varepsilon} + \lambda\left(\theta_g - \theta_l^{i-1}\right) \odot RMS_{\text{norm}}$

Table 5.1: Update rules for Dynamic evaluation. η is the *learning rate* and λ is the *decay rate*, controlling by how much we bias the local parameters towards the original (global) ones at each update. $\sqrt{MS_g}$ is the per-parameter mean squared gradient norm, estimated on the training set and RMS_{norm} is its normalized version (divided by the sum over all parameters). For more details, refer back to Section 4.2.

not explore this contradiction further at the time. Note that the goal of the experiments in this section **is not** to find a single best combination of hyperparameters for each learning rule and prescribe that to be used in Dasher.

The hyperparameters that need to be tuned are:

- **Sub-sequence length** – when performing Dynamic evaluation, we effectively break the test sequence into chunks and update the parameters after processing each chunk. How long these chunks are can change how well we adapt so this is a hyperparameter we need to tune – longer sequences would give us a better estimate of the local distribution but making the length too big might mean we adapt too slowly.
- **Learning rate** η – the size of the gradient step we take, controlling the speed of adaptation. If we make it too small, we would not be doing any adaptation at all. If we make it too big, we risk destabilizing the system.
- **Decay rate** λ – controls by how much we bias the update towards the original weights of the network. Contrary to [Krause et al., 2018] I find that not using a global prior (setting λ to 0) yields better results in my experiments. I discuss this further in the rest of the section.
- MS_g **estimate batch size** – The batch size used to estimate the per-parameter mean squared gradients on the training set (See Equation 4.4) According to [Krause et al., 2018] "larger mini-batches will result in smaller mean squared gradients". Looking at the update equations in Table 5.1, this means that for bigger batches we would have a bigger effective learning rate.
- **Using an RMS global prior** – whether or not to decay parameters with large mean squared gradients faster.
- **Alphabet size** – As I discussed previously, the size of the alphabet can affect the model's performance – if a bigger alphabet is used, the model has to spread probability mass over more events (characters) and so we would expect its performance to be worse. I have also noted that Dasher offers several alphabets for the user to choose from. In my experiments, I use two types of alphabets – either the full set of characters available in the training set or the most comprehensive

English alphabet available in Dasher, which contains 102 characters. During training and evaluation I substitute any character that is not in the alphabet with a special `<UNK>` token. Adding that increases the sizes of the alphabets by 1. It also means that when making predictions, the model will assign some probability to this `<UNK>` token, which will not be a valid input symbol. I see two possible solutions to this. One is to spread the probability mass assigned to that symbol evenly across all other symbols, which would mean that it will be less true that the sizes of the boxes in the interface are proportional to the probability of the strings they represent under the model (See Section 2.1.2). The other option is to remove any tokens that are not in the alphabet from the training text, which would mean that we will be presenting the model with some "false" patterns during training. I have gone for substituting all characters with `<UNK>` tokens since it is standard practice in language modelling. Because of time constraints I have not explored the effects of removing unknown tokens from the training set.

Although they do not specify the exact grid over which they searched, [Krause et al., 2018] say they found hyperparameter tuning to be important in getting good performance out of using Dynamic evaluation. The authors observed that the most important hyperparameter is by far the learning rate but also found benefits in tuning the decay rate. They do not mention tuning the number of timesteps. However, I find that to also be important, especially when larger learning rates are used for the SGD update rule.

5.1.1 Experiment setup

For my first set of experiments, I use an LSTM model with a hidden dimensionality of 1024 (Section 2.3) trained on the first 90 million characters of the English version of the Europarl corpus [Koehn, 2005]. For the experiments in this section I use the biggest alphabet available in Dasher, which consists of 104 characters, plus the `<UNK>` symbol. The Europarl corpus contains symbols which are not present in this alphabet (around 0.08% of all symbols in the dataset). I substitute these with the `UNK` symbol. This is different from what [Krause et al., 2018] do. UTF-8 encodes every symbol (Unicode code point) with one to four bytes. Instead of predicting the symbols, the authors divide their data into bytes. Their approach is clearly not suitable for Dasher – if a user wants to enter a non-ASCII symbol, they would not be doing this byte by byte.

One of the aims of these experiments is to compare different hyperparameter settings. However, using the whole validation set to do this would require too much time, as evaluating a single hyperparameter setting requires a pass through the whole set. This is why I follow the advice of [Krause et al., 2018] who found that "it is possible to use a small subset of the validation set to tune hyper-parameters and achieve a similar performance".

For my first experiment I use the first 100 thousand characters of the validation set for the English version of the Europarl corpus. Note that I use a standard 90-5-5 split between training, validation and test sets (See Section 3.3 for more details). I evaluate this piece of text with multiple combinations of hyperparameter settings – the values that I use for different hyperparameters when using the SGD and RMS update rules are given in Tables 5.2 and 5.3, respectively. Note that for each update rule, I run Dynamic evaluation for each combination in the cross product of the lists of values listed in the corresponding table. [Krause et al., 2018] do not provide a list of hyperparameters that were searched, so I try a coarse grid of values which differ by orders of magnitude. The aim here is to understand how different parameters affect the adaptation process, rather than producing the best performing model by fine-tuning.

In the experiment described above, the validation and test sets are part of the same corpus of European parliament speeches. This means that the generating distribution at training and test time will be similar. It would make sense, however, that if they are less similar, the optimal hyperparameter setting will differ – we might want to adapt faster or might not want to decay the weights towards the original ones, which have been estimated on data that came from a very different distribution. In order to test this hypothesis, I also perform hyperparameter search on the 100 thousand characters of the Enron emails corpus, which is a dump of email from the Enron corporation. I describe this dataset and how I process it in more detail in Section 3.3. The distributions that generated European parliament speeches and corporate emails will presumably differ a lot. Evidence suggests that is true – the model trained on Europarl that I use in this Section achieves a BPC of 3.61 when evaluated on 5 million characters of the Enron corpus whereas the same model, when trained on 60 million characters of Enron data and evaluated on the same 5 million (not present in the training set) characters achieves the much lower BPC of 1.84.

Hyperparameter	**Values**
Sub-sequence length	1, 5, 10, 20, 50
Learning rate	0.001, 0.01, 0.1, 0.3, 0.5, 0.7, 1
Decay rate	0, 10^{-4}, 10^{-3}

Table 5.2: Hyperparameter settings for experiments with SGD update rules. I experiment with all possible combinations of these on a small subset of the validation set.

Hyperparameter	**Values**
Sub-sequence length	5, 10, 20, 30, 50
Learning rate	0.001, 0.01, 0.1
Decay rate	0, 10^{-4}, 10^{-3}
MS_g batch size	100, 250, 500

Table 5.3: Hyperparameter settings for experiments with RMSprop update rules. I experiment with all possible combinations of these on a small subset of the validation set. Larger values for the learning rate were ruled out by initial experiments.

5.1.2 Exploding losses

A problem I encounter during adaptation is that sometimes the loss during Dynamic evaluation explodes without any apparent reason (See Figure 5.1). A hypothesis is that this is caused by numerical instability in the gradient calculations, resulting in

`NaN` values for some gradients, but I have managed to rule this out by monitoring the gradient in cases where the loss explodes. These exploding losses are problematic for two reasons – first, if this happens while the language model is in use in Dasher, it will result in the user getting extremely poor predictions, making the system unusable. I discuss this issue in Section 4.4.4. The second reason why this is problematic is that it makes my estimates of good hyperparameter configurations less reliable since for some of them the loss explodes and there is not always a discernible pattern that would explain why that happens by looking at the hyperparameter configurations. With this caveat in mind, I proceed to analyze the results of the hyperparameter search.

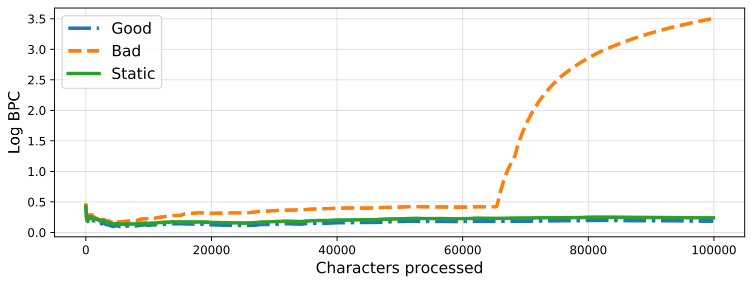


Figure 5.1: Log BPC against number of characters processed during evaluation. The bottom two lines represent static evaluation and a successful Dynamic evaluation. The third line is an instance of Dynamic evaluation going wrong – the loss suddenly goes up after processing a certain number of characters. This would be problematic if it happens in Dasher and I discuss potential solutions in Section 4.4.4.

5.1.3 Hyperparameters for the SGD update rule

The results of hyperparameter tuning on both evaluation datasets are given in the heatmaps in Figure 5.2 – the empty squares correspond to hyperparameter settings for which the loss exploded.

What is immediately obvious is that the improvements from Dynamic evaluation are greater when the discrepancy between the training and test distributions is bigger – we get more than 30% improvement over static evaluation when evaluating Enron data. This is unsurprising, since when evaluating on data from the same domain used in training, the original parameters are presumably already quite close to optimal for the test set.

The results in Figure 5.2 also support the claim by [Krause et al., 2018] that the learning rate is the most hyperparameter with the most impact on performance. It would make sense, then, that when training a language model that is to be integrated in Dasher, more effort is put into fine-tuning the learning rate.

5.1.3.1 Learning rate and number of timesteps

Another visible pattern is that in both cases when we use a bigger learning rate, we need to increase the sequence length in order to get better results. And if we combine a big learning rate with a small sub-sequence length, adaptation usually fails. This is especially true when adapting to data similar to the one used in training. Here is one possible explanation for why this happens. When we use a smaller sub-sequence length, we get a noisier estimate of the true conditional probability for the next character in the sequence and therefore of the gradients of our parameters with respect to the loss. If we have a noisy estimate of the gradient and we make a big move in that direction (using a high learning rate), we are more likely to end up with a "bad" setting of our parameters, destabilizing the network's dynamic state and/or causing the adaptation procedure to diverge. This also seems to be more of a problem when the parameters are close to some local optimum, since this is presumably true when evaluating on Europarl data and looking at the bottom-left corners of the top row of heatmaps in Figure 5.2, we see that for all settings there the loss exploded.

When a user starts using Dasher, the parameters of the model will probably be far from optimal for the text they are entering. However, with time, especially if they are using Dasher to write text from a single domain, the parameters will move close to a local optimum. These experiments suggest that when that happens, we should slow down adaptation by either reducing the learning rate or increasing the sub-sequence length. While this is not enough to give us a reliable solution, it should reduce the chance of the loss exploding. Ideally, we would have some way to speed up adaptation in case the user starts entering text from a very different domain – adaptive gradient update schemes like RMSprop (Section 2.3.3) come back to mind here. Perhaps extending the RMS Dynamic evaluation update rule by continuing to collect gradients while evaluating would be a candidate solution. However, I have not explored this or other solutions due to time constraints.

It is also possible that how close the parameters are from the optimum is not that important. In Section 5.2 I show an example of how evaluating a particular sequence destabilizes the network. It is not impossible that just by chance the validation sequence from Europarl contains more of these sequences. Due to time constraints I do not test this theory further.

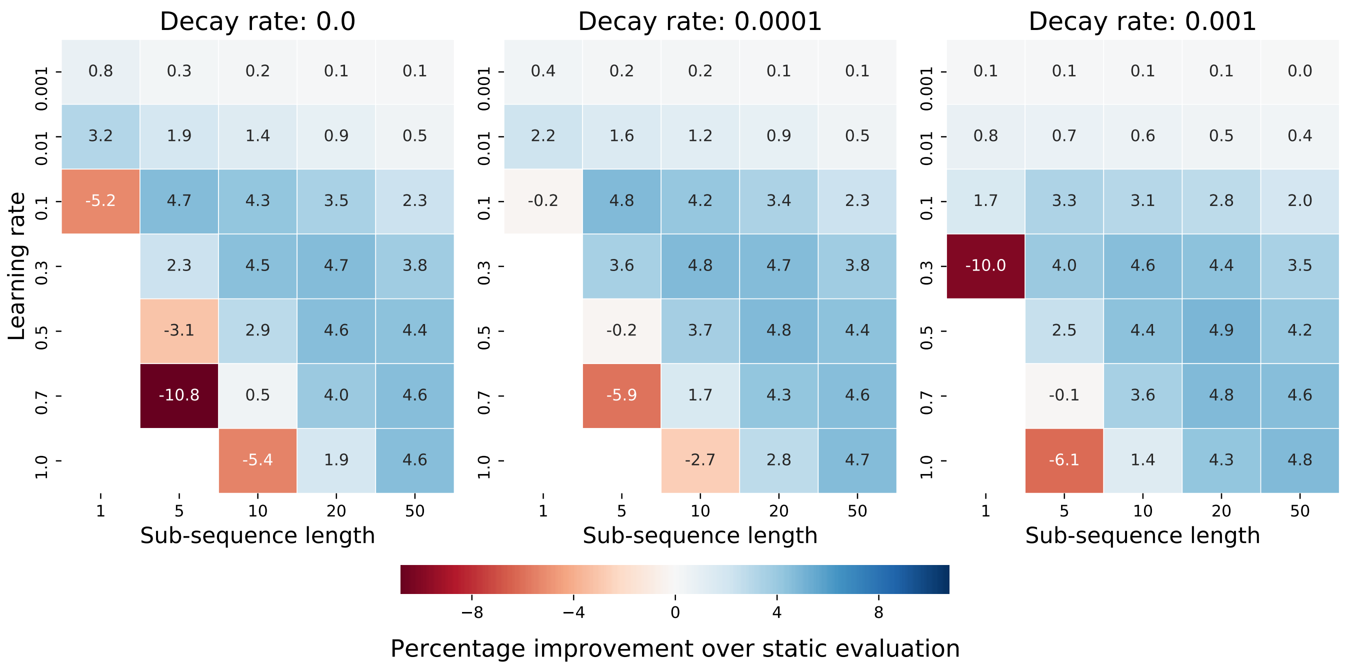
5.1.3.2 Decay rate

Overall, my results support the claim of [Krause et al., 2018] that tuning the decay rate has less of an impact than tuning other hyperparameters.

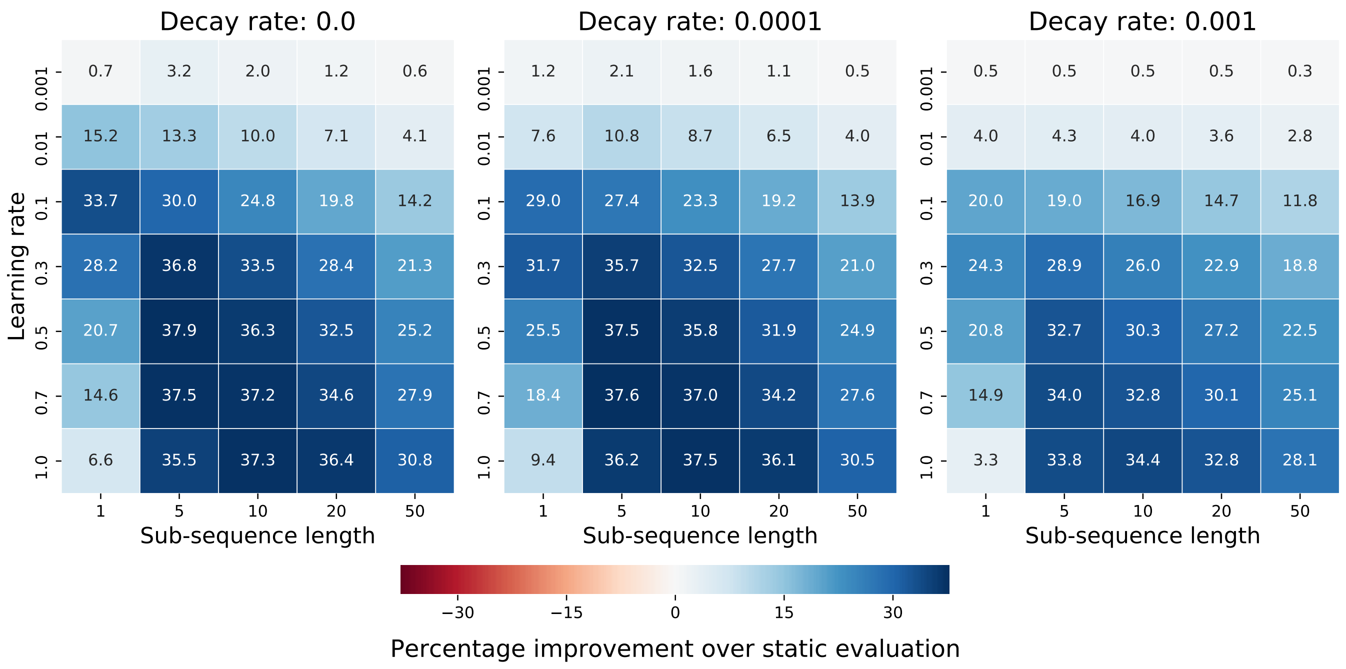
One of the questions I set out to answer in this section is whether there is any benefit from adding a decay term to the update function. Looking at the results in Figure 5.2 it seems that the answer is that it depends on the generating distribution for the test set – when adapting to Enron data, the value for the decay rate in the best hyperparameter combination was 0, unlike in the case when we adapt to more Europarl data when it was 0.001. I also performed dynamic evaluation on the full validation sets with the hyperparameter combinations that performed best on the subset I used for tuning, taking the best among combinations that used a decay rate of 0 and the best among ones that used a decay rate bigger than 0 separately. The results are given in Table 5.4.

The experiments in this section suggest that using a global prior is to be preferred when the global parameters are close to the maximal likelihood estimate for the local distribution, which makes sense – if we have good parameters to start with, we might not want to move too far away from them. In the context of Dasher this again means that we might want to vary this hyperparameter as the parameters of the network move closer to optimal ones during adaptation.

When the shift between the distributions at training and test time is bigger, these experiments indicate that we might not want to use a global prior. However it is true that there might be a better value for the decay rate between 0 and 0.0001, which is why I ran another experiment on the first 100 thousand characters of the Enron validation set, this time using smaller decay rates. I found that even decay rates as small as 10^{-7} do not provide an improvement over using no decay.



(a) Evaluated on Europarl data (same as training)



(b) Evaluated on Enron data – Dynamic evaluation leads to much greater improvements compared to when evaluating on Europarl data.

Figure 5.2: **(Best viewed in color)** BPC for different settings of the *decay rate*, *learning rate* and *sub-sequence length* parameters. The results are obtained by using Dynamic evaluation with a SGD update rule to evaluate 100 thousand characters of Europarl *(Top)* and Enron *(Bottom)* data using and LSTM model with 1024 hidden units trained on 90 million characters of Europarl data. The values in the boxes are relative improvements (in %) over the BPC for the corresponding text when not using Dynamic evaluation (1.27 for Europarl and 3.86 for Enron). The empty boxes correspond to settings for which Dynamic evaluation failed – the final loss was more than 15% worse than when not using Dynamic evaluation. This behaviour would be problematic in Dasher and I discuss it in Section 4.4.4.

5.1.4 Hyperparameters for the RMS update rule variants

Figures A.2 and 5.4 show the results from experiments with the RMS update rules. The two figures show results for evaluating a model trained on Europarl on Europarl data and on Enron data, respectively. The results are given for the RMS *without* global prior rule but this choice was arbitrary since the differences in results for experiments with and without using a global prior were minimal.

5.1.4.1 Learning rate and number of timesteps

Overall, these results tell a similar story to the experiments with the SGD learning rule. Again, it looks like the learning rate and the number of timesteps are tied together – if we use a bigger learning rate, we want to update our parameters more rarely.

Again, making big update steps when estimating the gradient over short sequences causes the loss to explode. It is evident that per-parameter learning rates have not alleviated this problem.

5.1.4.2 Batch size for mean squared gradient estimation

The proposition that bigger batch sizes result in smaller mean squared gradients and therefore bigger update steps is backed by these experimental results. When using bigger batch sizes when evaluating the Europarl validation set, for example (Bottom left of Figure A.2) we tend to see losses explode.

5.1.4.3 RMS global prior

In the experiments I ran, I saw no benefit of scaling the decay rate according to mean squared gradients estimated on the training set. This is not necessarily a contradiction with the claims of [Krause et al., 2018]. Although they report improvements in word-level modelling when using an RMS global prior, they do not give a comparison for character-level language models.

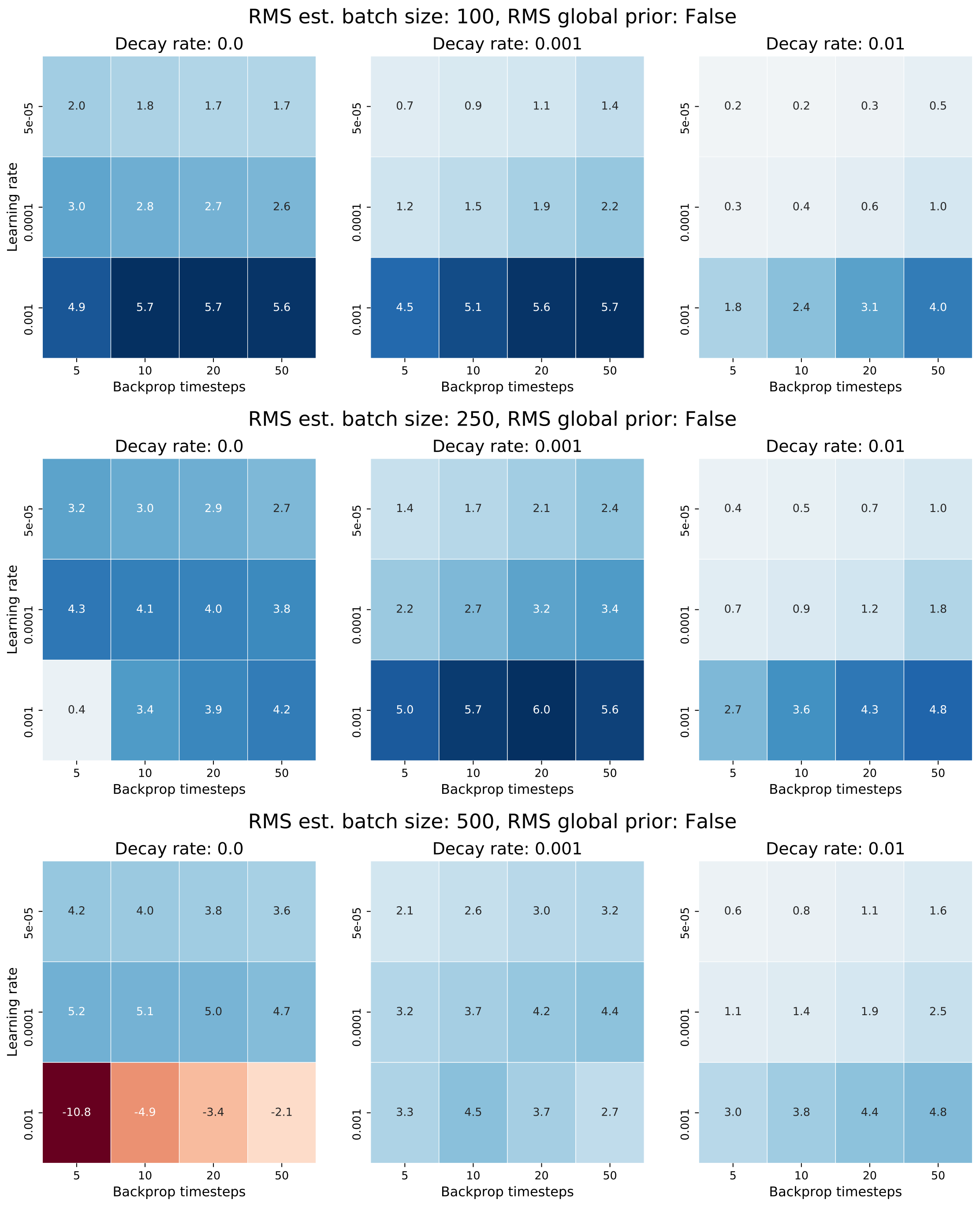


Figure 5.3: **(Best viewed in color) (RMS Dynamic evaluation of Europarl)**: BPC for different settings of the *decay rate* **(across rows)**, *RMS estimation batch size* **(across columns)**, *learning rate* and *sub-sequence length* parameters. The results are obtained by using Dynamic evaluation with a RMS update rule to evaluate 100 thousand characters of Europarl *(Top)* using and LSTM model with 1024 hidden units trained on 90 million characters of Europarl data. The values in the boxes are relative improvements (in %) over the BPC for the corresponding text when not using Dynamic evaluation. We can see that, like with the SGD update rule, the most important hyperparameters are the learning rate and the sub-sequence length and they work together to control the speed of adaptation. Exploding losses are still a problem (Bottom left). They are more common when a bigger batch size is used, suggesting that bigger batch sizes lead to bigger effective learning rates.

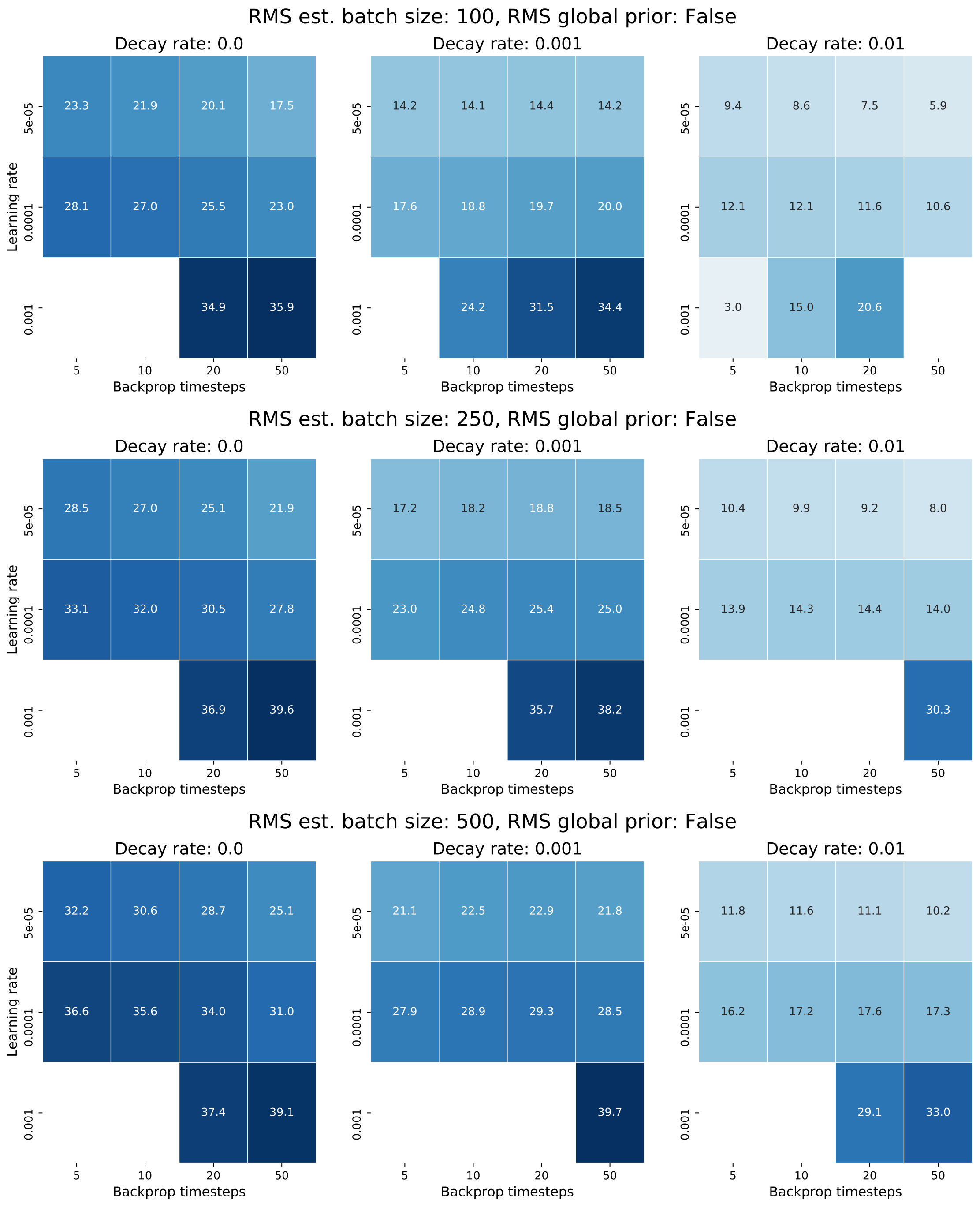


Figure 5.4: **(Best viewed in color) (RMS Dynamic evaluation of Enron)**: BPC for different settings of the *decay rate* **(across rows)**, *RMS estimation batch size* **(across columns)**, *learning rate* and *sub-sequence length* parameters. The results are obtained by using Dynamic evaluation with a RMS update rule to evaluate 100 thousand characters of Europarl *(Top)* using and LSTM model with 1024 hidden units trained on 90 million characters of Europarl data. The values in the boxes are relative improvements (in %) over the BPC for the corresponding text when not using Dynamic evaluation. The empty boxes correspond to settings for which Dynamic evaluation failed – the final loss was more than 15% worse than when not using Dynamic evaluation. The improvements over static evaluation are minimal, although the optimal parameters are near the edge of my grid, suggesting that slightly bigger learning rates and longer sub-sequence lengths can be explored.

5.1.5 Does the alphabet size matter?

The question I ask here is whether if I use a dataset with a smaller alphabet for training and evaluation, the optimal settings for the hyperparameters of the adaptation methods change. I trained a model on the training portion of the text8 dataset, which contains only lowercase Latin letters, dots and spaces (I describe it in Section 3.3). Then I evaluated the first 100 characters of the validation set using Dynamic evaluation with an SGD learning rule with the same grid of hyperparameters I used in the rest of the experiments in this section (Given in Table 5.2).

I found that the optimal parameters are in the same region as when evaluating in-domain data for a model trained on Europarl (Figure 5.2) and that the different parameters interact in a similar way. The results for the full hyperparameter grid can be found in Appendix A.1. These results, although not conclusive, suggest that the alphabet size does not affect the optimal values for the adaptation hyperparameters.

A more rigorous experiment would have involved modifying the Europarl data to lowercase and using results for training and evaluating on the modified data but I omit this due to time constraints.

5.1.6 Comparison between update rules

The results from the hyperparameter search I performed suggested that the RMS update rules leads to slightly better adaptation performance in comparison to the SGD ones. In order to confirm this I selected the best hyperparameter combinations for all variations of the two rules and evaluated the full validation sets of the Enron and Europarl datasets. The results are given in Table 5.4. They show the RMS rules are marginally better when evaluating Europarl data. Also, it seems that the benefit of using a global prior I saw during hyperparameter tuning diminishes when evaluating over the full dataset. When evaluating Enron data, however, the SGD rule performs much better and using the RMS rule caused the loss to explode for two of the selected settings. This stands to reason – the longer the sequence we are evaluating, the bigger the chance that it will contain a sequence that will destabilize the network.

Overall, my results do not suggest we should have a clear preference to one of the rules when only looking at language modelling performance. However, the RMS learning rules carry with them the extra complexity of having to estimate the mean squared errors on the training dataset and having to tune the batch size for this procedure. Considering this, I cannot say that based on my results it is well justified to use RMS learning rules for Dynamic evaluation in Dasher.

5.2 An example of how Dynamic evaluation fails

As I discussed previously, sometimes adapting a network's weights fails spectacularly, with the loss exploding. If that happens in Dasher, it can render the system useless and

Update rule	Europarl					Enron				
	BPC	η	ts	λ	bs	BPC	η	ts	λ	bs
Static evaluation	1.17	–	–	–	–	3.55	–	–	–	–
SGD	1.14	0.1	5	–	–	**1.93**	0.7	5	–	–
SGD, global prior	1.14	0.5	20	10^{-4}	–	2.36	0.7	5	0.001	–
RMS, global prior	1.14	0.001	20	0.001	250	71.37	0.001	50	0.001	250
RMS, no global prior	**1.13**	0.001	20	–	100	2.27	0.001	50	–	250
RMS, RMS global prior	1.15	0.001	50	0.001	250	55.09	0.001	50	0.001	250

Table 5.4: Best performance on the full 5-million character validation sets for Enron and Europarl. The results are obtained using the same model I have tuned hyperparameters for. I have evaluated the full validation sets using the best hyperparameter setting I found while evaluating the first 100 thousand characters of the validation sets. Next to each result I have listed the values for the learning rate η, the sub-sequence length ts, the decay rate λ and the batch size used for the RMS update rule bs. Where the hyperparameter is not applicable to the learning rule I have left a dash. The numbers in red correspond to cases in which the network was destabilized and the loss exploded.

I discuss how we could handle it in Section 4.4.4.

In this section I investigate how this happens – it is possible that this is just a bug in my code or perhaps the way I handle unknown characters in my experiments. If so, perhaps there is an easy fix for this and we do not need to worry about it in the context of Dasher.

However, I find an example of exploding losses that can happen during perfectly normal use of Dasher. This happens for certain settings of the hyperparameters when processing the following sequence from the Europarl validation set :

“Commission can accept fully, in part, or in principle Amendments 1, 2, 5, 6, 8, 9, 12, 13, 14, 15, 17, 19, 20, 22, 24, 25, 26, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 48, 55, 57, 58, 59, 60, 61, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 81, 82, 83, 84, 85, 86, 87, 89, 90, 91, 92, 97, 103, 106, 113. The Commission cannot accept Amendments 3, 4, 7, 10, 11, 16 . . . ”

The plots in Figure 5.5 track the predictions and losses of two dynamic and one static model while evaluating this sequence. The update rule for the two dynamic models is the RMS with a global prior rule. The difference between them is that the one for which the loss exploded had used a bigger batch size while estimating the mean squared gradients from the training set (See Section 4.2 for a reminder). [Krause et al., 2018] claim that larger batches result in smaller mean-squared gradients, which means that the effective step size for the update would be bigger for the model whose gradients were estimated using bigger batches. This seems like a reasonable explanation of why this model, unlike the other dynamic one did not manage to recover after processing the sequence of numbers.

This is not to say that using a smaller batch size or a smaller learning rate solves the problem. What this experiment shows is that the problem with exploding losses is due to an issue that is inherent to RNNs – their dynamic state is prone to destabilization when they are exposed to unexpected inputs and modifying their parameters amplifies this issue. What is more, there does not seem to be a principled way to detect this. This unpredictability is highly undesirable in Dasher and I discuss this problem in more detail in Section 4.4.4.

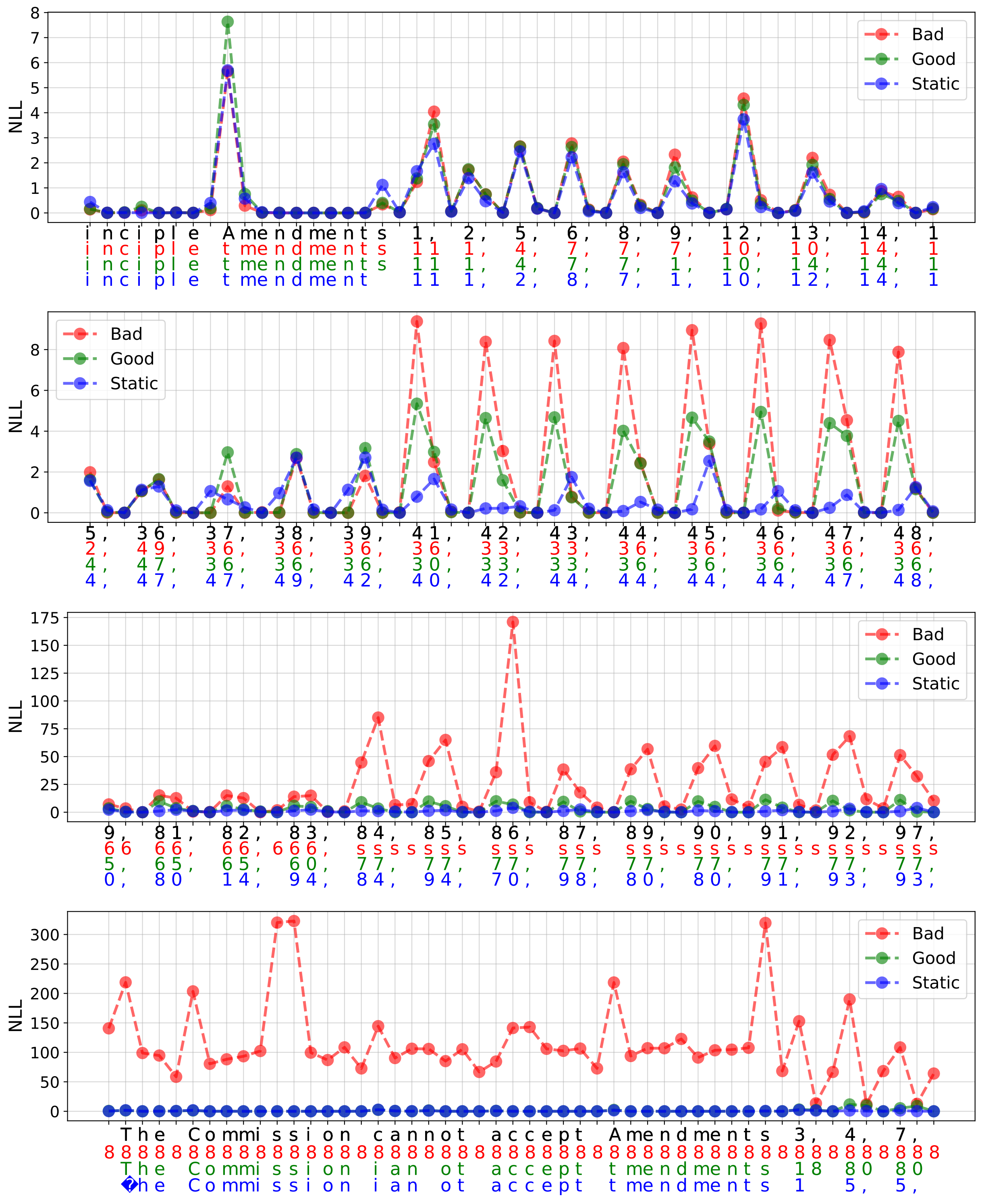


Figure 5.5: **(Best viewed in color)** Four plots tracking different stages of an example of exploding losses during dynamic evaluation of the sequence given in the beginning of the section. The lines are the losses suffered by the model at each character of the sequence for three types of evaluation – a dynamic one for which the loss explodes **(Red)**, a dynamic one for which it does not **(Green)** and static evaluation **(Blue)**. The text in black under the plots is the target sequence and the colored texts are the characters assigned the highest probability under the corresponding evaluation scheme. While evaluating sequences like $31, 32, \ldots 39$ dynamic models adapt their weights so as to predict a 3 after a comma and a space and confidently predict that when the sequence changes to $40, 41 \ldots$, suffering big losses. It seems that after that one of the models' dynamic state becomes destabilized and it begins producing nonsensical predictions, while the other two models manage to recover from this.

5.3 Sparse Dynamic evaluation

Sparse Dynamic evaluation, which I describe in Section 4.2.3 is proposed by [Krause et al., 2018] as a way to adapt an RNN with less computation and memory. As a reminder, in sparse Dynamic evaluation we substitute the hidden state of the network at each timestep with a modified version.

$$\mathbf{h}'_t = \mathbf{h}_t + \mathcal{M}\mathbf{h}_t \tag{5.1}$$

where the entries of $\mathcal{M}$ are the adaptation parameters we change at test time. The authors also propose to only change a subset of the hidden units and achieve only 5% worse BPC compared to full Dynamic evaluation by modifying less than 18% of their network's hidden units.

5.3.1 Setup

When using this approach, then, we need to decide on three hyperparameters – the learning rate for the gradient step when adapting $\mathcal{M}$, the number of hidden units to adapt and the length of the sub-sequences after which we update the weights. There is also a third decision that has to be made – whether to use the original or modified hidden state in the recurrent computation. [Krause et al., 2018] use the modified version and do not mention whether or not they experiment with using the original one. My hypothesis is that only using the modified hidden state for computing the output will mean that it is less likely to destabilize the dynamic state of the network and lead to poor predictions. If doing so leads to similar improvements in performance and my hypothesis is true, this would be a desirable modification to the algorithm, since destabilizing the network while it is in use in Dasher would be very bad news, as mentioned previously.

The questions I am asking in this section are

- How does the performance of sparse Dynamic evaluation compare to that of the other Dynamic evaluation methods?
- How is performance affected if we do not use the modified state in the recurrent computation? If there were instability issues, does this help?
- How important are the two hyperparameters? This would serve to guide fine-tuning of parameters if this approach is to be used in Dasher.

For performing hyperparameter search I use the same model (trained on Europarl) and experimental setup as described in Section 5.1.1 – I evaluate the first 100 thousand characters of the validation sets of both the Europarl and Enron corpora. The grid of hyperparameters I use is given in Table 5.5.

Hyperparameter	**Values**
Number of units to adapt	200, 500, 1024 (all)
Learning rate	0.0001, 0.0005, 0.001, 0.01
Sub-sequence length	1, 5, 10, 20, 30, 50

Table 5.5: Hyperparameters values considered for sparse Dynamic evaluation. 200 units is around 19% of all (1024) units in the hidden state and [Krause et al., 2018] report good results with adapting 18% of their units. Initial experiments ruled out using larger learning rates.

5.3.2 Results

Surprisingly, I got no improvement over static evaluation using any of the hyperparameter combinations, regardless of whether I used the modified state in the recurrent computation or not. This was true for both datasets I evaluated on. The smallest of the losses were within 1% to the performance of static evaluation and, like with full dynamic evaluation, in a lot of cases the loss exploded.

As with exploding losses during the other variants of dynamic evaluation, I tried to find `NaN` gradients by attaching debug hooks to my models but did not find such problems.

Since [Krause et al., 2018] do not specify the update rule they use when performing Sparse evaluation or any hyperparameters in my experiments I assumed the simplest possible option – SGD with no weight decay and no momentum. Since the authors report good performance using this method, they must have done something different from what I am doing here.

In an attempt to get better BPC using sparse dynamic evaluation, I tried two things – first, since in a lot of the cases the losses were exploding, I tried reducing the learning rate and also adding an L2 penalty term in an attempt to constrain the adaptation weights. And second, I tried changing the optimization method by adding momentum and trying adaptive methods. I describe my attempts below and the results I got are given in Table 5.7.

5.3.3 Regularization and smaller learning rates

I evaluated the first 100 characters of the Enron validation dataset, again, this time adding regularization and attempting smaller learning rates. I tried all combinations of the parameters in Table 5.5, plus the parameters in Table 5.6.

Hyperparameter	**Values**
L2 weight decay	100, 10, 1, 0.1, 0.01
Learning rate	10^{-7}, 10^{-6}, 10^{-5}

Table 5.6: Additional hyperparameters for sparse dynamic evaluation

Adding a weight penalty term did help – the best results I got when using weight decay with combination with the original hyperparameter grid in Table 5.5 was an improvement of 9% relative to static evaluation. Using a smaller learning rate helped increase that to 12%. Given that using when full Dynamic evaluation I got around 38% relative improvement, this is still unimpressive.

5.3.4 Using adaptive gradient methods

I also attempted updating the weights with adaptive per-parameter optimization schemes. I attempted Adagrad [Duchi et al., 2010], Adam [Kingma and Ba, 2015] and Adadelta [Zeiler, 2012] using the default hyperparameters recommended in the corresponding papers. Using Adagrad I managed to get an improvement of 16% over static evaluation.

Method	**Improvement**
Sparse DE (SGD)	-0.8%
Sparse DE (SGD + L2)	9%
Sparse DE (SGD + L2 + smaller learning rate)	12%
Sparse DE (Adagrad + L2)	**16%**
Full DE (SGD update rule)	38%

Table 5.7: Results for sparse dynamic evaluation for a model trained on Europarl and evaluated on the first 100 thousand characters of the Enron emails validation set. The column on the right is the improvement in BPC over static evaluation. Despite my efforts I could not get sparse Dynamic evaluation to perform nearly as well as full dynamic evaluation (bottom row).

5.3.5 Conclusion

I tried to answer three questions about sparse dynamic evaluation. First, I asked how it compares to full Dynamic evaluation. I tried multiple optimization techniques but even the best results for sparse dynamic evaluation were not nearly as good as the results for full Dynamic evaluation. Looking at the running averages of the BPC while evaluating the first 20 thousand characters of the Enron dataset (Figure 5.6) it seems that sparse Dynamic evaluation is much slower in adapting its predictions to a large shift in the generating distribution then full Dynamic evaluation. This is not great news for a method that is to be used in Dasher – we would not want the user to have to enter tens of thousands of characters before getting good predictions.

The second and third questions I wanted to answer were how important different hyperparameters are and whether we should use the modified hidden state in the recurrent computation. The learning rate and sub-sequence length seem to have the same effect they have in full Dynamic evaluation. I found that adding L2 regularization and using small learning rates is important in order to get good performance on the datasets I evaluated on. As to the use of the modified hidden state I found that using it in the recurrent computation usually produces better results and does not lead to the loss exploding more often. I did not investigate these further as I think sparse Dynamic

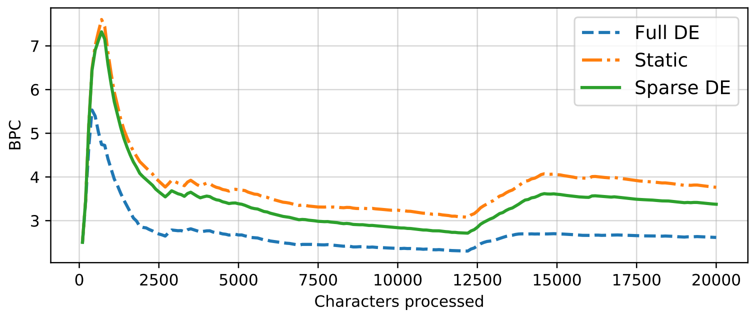


Figure 5.6: Average BPC on the first 20000 thousand characters of the Enron validation set using static evaluation, full dynamic evaluation and sparse dynamic evaluation. Full dynamic evaluation adapts more quickly and handles surprising inputs better than sparse dynamic evaluation.

evaluation is a worse fit for Dasher than full Dynamic evaluation and therefore I think I should be directing my efforts elsewhere.

Overall I found that sparse Dynamic evaluation performs worse than full Dynamic evaluation over long sequences, adapts more slowly to shifts in the generating distributions yet still suffers from the problem of exploding losses, meaning that it does not give us any useful advantages over full Dynamic evaluation in the context of Dasher.

5.4 LHUC

Like sparse Dynamic evaluation, LHUC (Described in Section 4.3) also performs adaptation by modifying the hidden state at every timestep but instead of using a linear transformation it scales every element by a number in the range $[0;2]$:

$$\mathbf{h}_t^{(LHUC)} = \mathbf{h}_t \circ 2\sigma(\mathbf{r}) \tag{5.2}$$

where $\sigma(x)$ is the logistic sigmoid function.

5.4.1 Setup

The hyperparameters that need to be considered for LHUC are also the same as with sparse Dynamic evaluation, with the exception that with LHUC we adapt all units. Again, we have the choice of whether or not to use the hidden state in the recurrent computation.

As in the experiments on sparse Dynamic evaluation (Section 5.3), here I aim to find out how this method compares to the others I tried in terms of BPC, I try to find how important the two hyperparameters are and also whether or not we use the modified state in the recurrent computation makes a difference.

The model, validation datasets and experiment setup are also the same as in Section 5.3 and the grid of hyperparameters I use for my LHUC experiments is given in Table 5.8.

Hyperparameter	**Values**
Learning rate	$\mathbf{10^{-6}}$, $\mathbf{10^{-5}}$, 0.0001, 0.0005, 0.001, 0.01
Sub-sequence length	1, 5, 10, 20, 30, 50, **100**, **300**, **1000**

Table 5.8: Hyperparameters values considered for LHUC. I added the values in bold after my initial experiments indicated that I might want to explore more of the hyperparmter space.

5.4.2 Results and discussion

The results of the grid search I performed are given in Figure 5.8. On the whole LHUC performs roughly on par with static Dynamic evaluation, meaning worse than full Dynamic evaluation. When the data at test time is similar to the one at training time (the top plot of Figure 5.8), using LHUC makes little difference. I also trained a model on the text8 dataset, introduced by [Mikolov et al., 2015], which consists of 100 million characters of Wikipedia data converted to lower-case and stripped of punctuation. I tried different LHUC settings on that dataset, too, and found no improvement. The results are given in the Appendix A.2.

Adaptation speed It seems that, as in Dynamic evaluation (Figure 5.2), if we use a shorter sequence length, we need to reduce the learning rate. Again, making big steps often leads to the loss exploding (Bottom left corners of the heatmaps in Figure 5.8). One difference to Dynamic evaluation is that with LHUC we can get good performance (low BPC after processing many characters) even when updating the weights after each timestep, if our learning rate is small enough. However, this does not seem to translate to quicker adaptation in the beginning of the sequence (See Figure 5.7), which would have been good for a method which is to be used in Dasher.

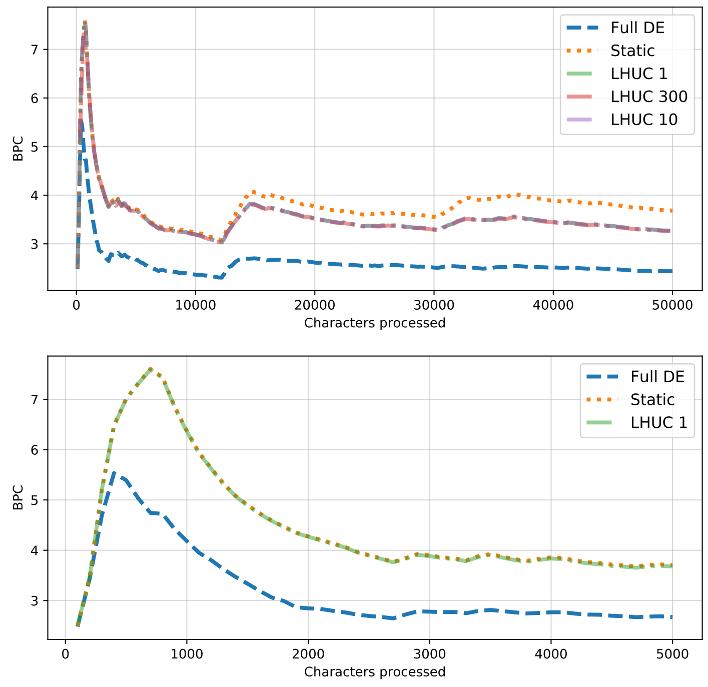
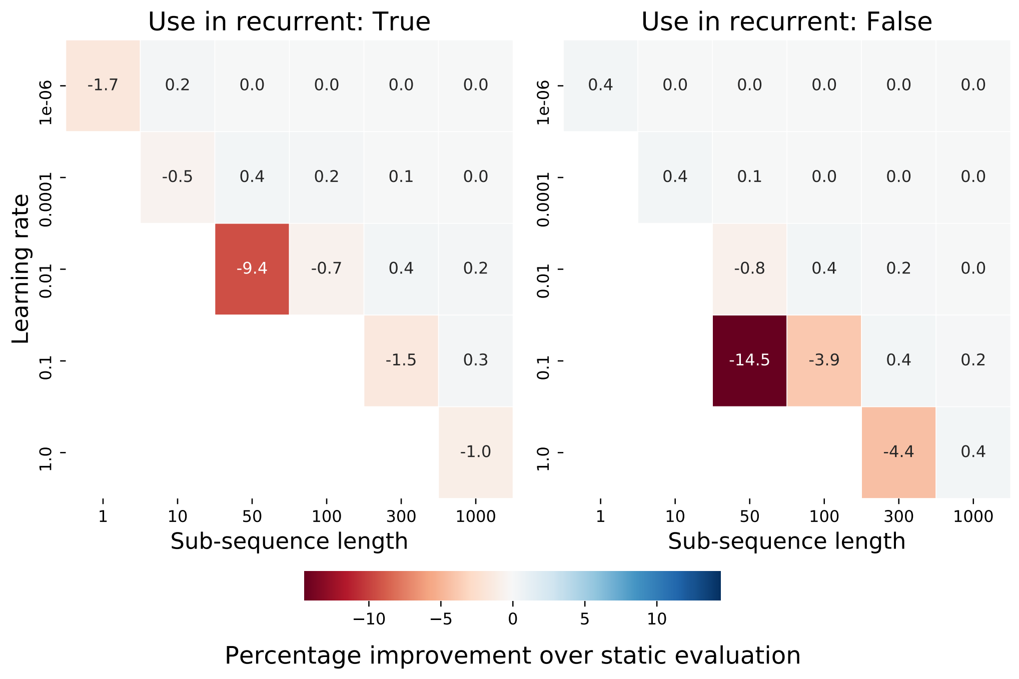
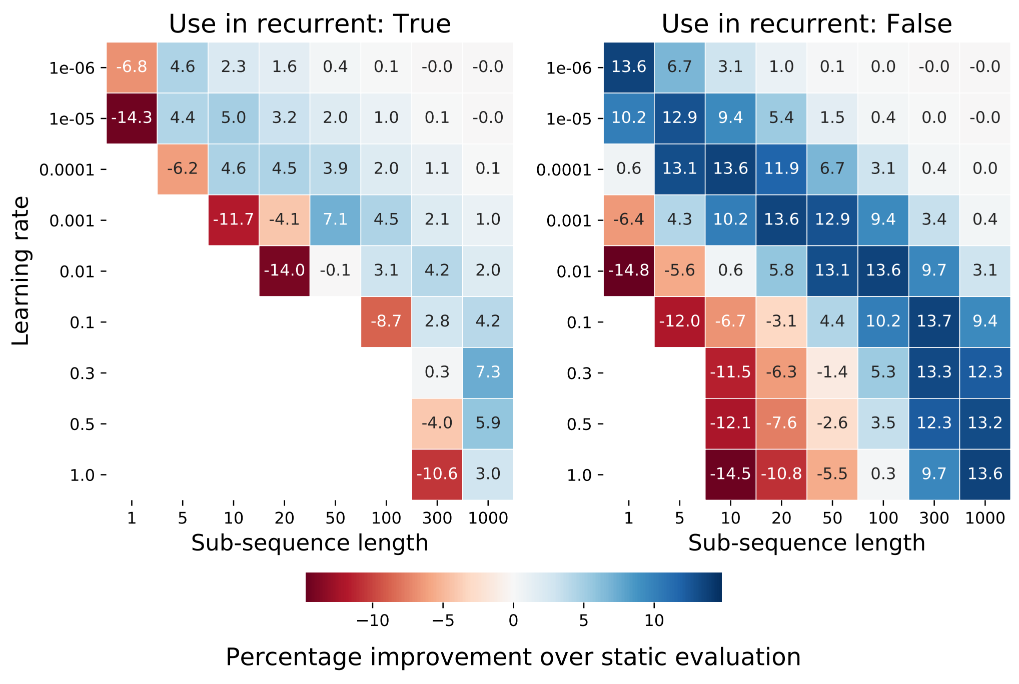


Figure 5.7: Average BPC on the first 50 thousand **(Top)** and just the first 5 thousand **(Bottom)** characters of the Enron validation set using LHUC with different number of timesteps, full dynamic evaluation and sparse dynamic evaluation. LHUC takes a long time to make a difference in the Bits per character and how quickly that happens does not seem to depend on how often the LHUC scalers are updated.



(a) Evaluated on Europarl data (same as training)



(b) Evaluated on Enron data

Figure 5.8: **(Best viewed in color)** BPC for different settings of the *decay rate*, *learning rate* and *sub-sequence length* parameters. The results are obtained by using LHUC to evaluate 100 thousand characters of Europarl *(Top)* and Enron *(Bottom)* data using and LSTM model with 1024 hidden units trained on 90 million characters of Europarl data. The values in the boxes are relative improvements (in %) over the BPC for the corresponding text when not using Dynamic evaluation. The empty boxes correspond to settings for which Dynamic evaluation failed – the final loss was more than 15% worse than when using static evaluation.

Using the modified hidden state From the results in Figure 5.8 it is clearly visible that using the modified hidden state in the recurrent computation is undesirable – it leads to poorer adaptation performance and causes the network to destabilize more often. This is in agreement with the claim that RNN models are dynamic systems which are prone to destabilizing. It seems like any changes that affect their dynamic state should be done with a lot of care.

5.4.3 Conclusion

The takeaways from the experiments with LHUC are quite similar to those from experimenting with sparse Dynamic evaluation. This method does not seem to lead to better adaptation performance over very long sequences, neither does it adapt more quickly for shorter ones. At the same time, even though we are adapting a smaller set of parameters, even when we do not use the modified hidden state in the recurrent computation, the network is often destabilized so LHUC is not superior to Dynamic evaluation in this sense, either.

5.5 Adapting a subset of the network's weights

When performing adaptation using LHUC my results suggested that using the hidden state in the recurrent computation leads to worse performance and more instability in the network's dynamic state for which I judge by the loss exploding at some point during the evaluation process. The problem of exploding losses is also present in full Dynamic evaluation (See the experimental results in Section 5.3.2), which is also the best performing method I have tried so far. Combining these two observations, I ask the question whether if we do not change the recurrent computation during dynamic evaluation, we would alleviate the problem of exploding losses. If so, will language modelling performance suffer? This is further motivated by the work of [Sengupta and Friston, 2018] I mention in the beginning of this chapter, which suggests that small perturbations in the matrix governing the hidden state transitions can easily lead to instability in the network.

5.5.1 Motivation and setup

This motivates the two simple experiments I present in this section. Instead of adapting all of the network's parameters like I have done in Section 5.3.2, I only adapt a subset of them. In one experiment I only change the weights which parametrize the function mapping from the hidden state to the output of the network (**hidden-to-out** weights), which does not affect the recurrent computation and in the other I only change the weights that govern the recurrent computation (the **hidden-to-hidden** weights).

The first question I ask is whether adapting only the hidden-to-out weights makes the network more stable and adapting only the hidden-to-hidden weights makes it more unstable. The second question is whether adapting only a subset of the weights would result in smaller improvements over static evaluation, for which we might expect the answer to be "yes" since we would have a smaller number of adaptation parameters.

In order to compare the results to full Dynamic evaluation, I use the same model described in 5.1.1 and evaluate on the first 100 thousand characters of the validation set. I performed Dynamic evaluation with an SGD with a global prior update rule with the same set of hyperparameters as listed in Tables 5.2.

5.5.2 Results and conclusion

The simple conclusion from my experiments is that they indicate that my hypothesis was false. Adapting only one subset of the weights did not seem to cause the network to destabilize more or less often. In terms of results, when adapting the full set of weights, I got an improvement of 4.8% over static evaluation for the best hyperparameter setting. And when adapting only the hidden-to-out or hidden-to-hidden parameters, I got improvements of 3.3% and 4.4% respectively. One explanation for this difference is that there are more hidden-to-hidden (1024×1024) then hidden-to-out (104×1024) parameters.

I give the full grid of results in Appendix A.3 but given the results I presented I do not pursue the line of questoning in this section further.

Chapter 6

Learning a good initial hidden state

When somebody reports a BPC (Bits Per Character) loss on some corpus, they sum up the negative log losses for all characters and divide them by the number of characters in the corpus. In the context of Dasher this means that when we say that model A has half the BPC loss of model B on some 10-million character long chunk of text, that means, roughly, that if we use model A in Dasher, we would enter those 10 million characters in half the time it would take us if we used model B (Refer back to Section 2.1.3 for a detailed explanation of why this is the case).

Just knowing that model A achieves a relatively low BPC, however, does not tell us much about its predictions on individual characters – it might very well be the case that the model was next to useless in predicting the first 10 thousand characters and was nearly perfect on the rest. Such a situation is clearly undesirable in Dasher – if the user has to enter thousands of characters before they get meaningful predictions, chances are they'll get frustrated and uninstall Dasher as quickly as they can.

So is this something I should be concerned about? Figure 6.1 suggests that the first few predictions of a generally well-performing model are quite poor but it recovers after ten or so characters. Even though that is not the thousands of characters in the example above, this situation will take place every time the user opens Dasher or perhaps even every time they switch windows – if every time they have to enter 10 characters before getting good predictions, that is still clearly undesirable. So why does this happen? The predictions of the model at every timestep depend only on the hidden state at that timestep – said hidden state should be a representation (an embedding into $\mathbb{R}^H$ where H is the hidden dimensionality, if you like) of the current context. So the initial predictions depend on what we choose the initial hidden state of the network to be. Since relatively big initial losses have negligible effect on the average loss on a big corpus, which is the number usually given in literature, people rarely care to carefully choose an initial state when evaluating models. [Krause et al., 2018] who reported then state-of-the-art results in character-level language modelling, for example, use the zero vector as an initial hidden state. This is also what I did to obtain the results in Figure 6.1.

Since this is clearly important in the context of Dasher, in this section I aim to show that it is possible to select a "good" initial hidden state so as to minimize the loss on the

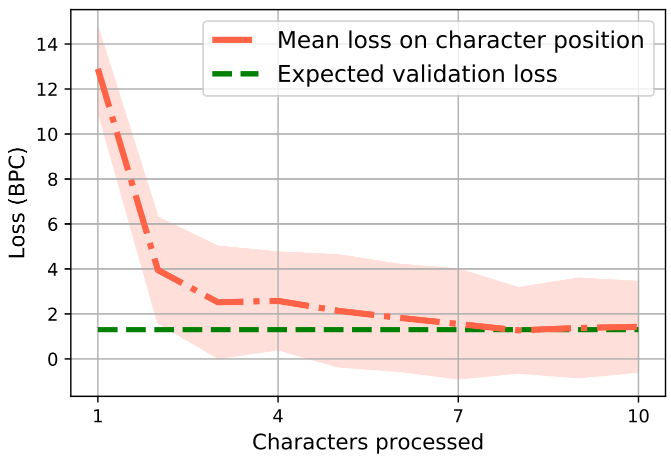


Figure 6.1: Loss on the first characters of sentences (red) suffered by an LSTM model with 1024 hidden units trained on 60 million characters from the Europarl corpus. The losses are averaged over the starts of 700 sentences from the test set. The shaded region represents one standard deviation below and above the means. The green line is the loss suffered by the model on the whole, 10 million-character-long validation set. Notice that the model suffers a very high loss on the first few characters – this means that every time a Dasher user switches context, they will be given poor predictions.

initial characters a user enters. For the benefit of this discussion it is instructive to first take a look at how Dasher solves this problem presently.

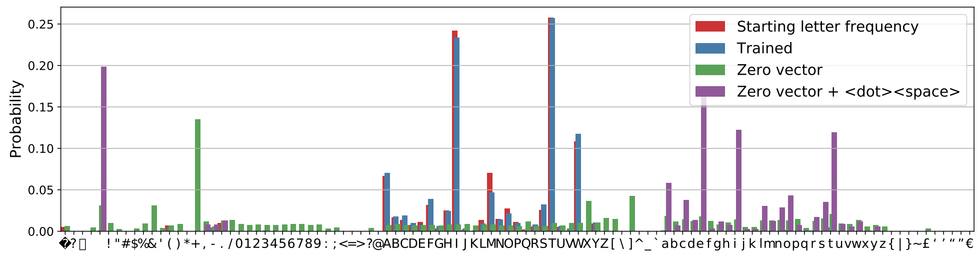
6.1 How does Dasher produce good initial predictions?

Dasher's language model resorts to predictions from lower-order contexts when higher-order ones are unavailable. In plain English this means that when it hasn't seen the particular combination of the last five letters (or such is unavailable), the model acts as if it had only seen the last four. If it has not seen those before, it looks at the last three and so on.

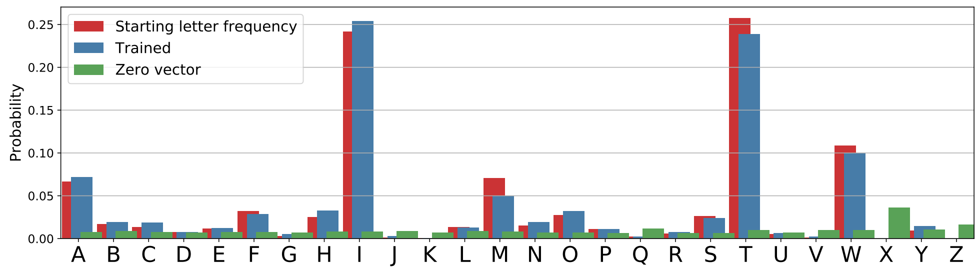
The creators of Dasher exploited this property of the language model to produce good initial predictions. Before the model is asked to produce a distribution for the first character, it is fed a dot followed by a space character. The model's predictions then depend only on the trigrams in the training data that start with a dot followed by a space. Basically, the predictions will be consistent with how sentences started in the training corpus.

6.2 How can an RNN language model produce good initial predictions?

I think the idea behind the approach taken in Dasher makes sense – it is reasonable to expect that the first character a user types would be the beginning of a sentence. Moreover, when a user switches context (changes to a different window or starts a new conversation, for example) we would probably assume that they would start a new sentence and would want to make predictions that are consistent with that assumption. So if we find a "good" initial hidden state, we could just use that to make predictions when switching between contexts.



(a) Distributions over the whole alphabet



(b) Zoomed in on capital letters only

Figure 6.2: (Best viewed in color) **Red bars**: Relative frequencies of of starting letters in the corpus used to train a good initial hidden state. The other colors represent distributions over characters obtained using three different hidden states and the same network parameters. **Blue:** An initial hidden state chosen to minimize the loss on the first 10 characters of a sentence given fixed network parameters. **Green:** A zero-vector hidden state. **Purple:** A hidden state obtained by taking a zero vector hidden state and feeding a dot and a space to the network (This is essentially the approach the Dasher system uses now). Note the absence of purple bars in the bottom figure – the probabilities assigned to capital letters were all zero in that case. We can see that the two naive approaches (in green and purple) do not give us meaningful predictions (Perhaps surprisingly, feeding a dot and a space with a zero initial hidden space seems to lead to making a prediction similar to what we would expect the distribution of unigrams to be.) We also see that the **blue** and **red** distributions are very similar – we have learned an initial hidden state from which we predict the starting letter of a sentence quite well, which is important for Dasher.

A naive approach to obtaining a good initial state for an LSTM language model would be to do what is currently done in Dasher – simply feed a dot and a space character to the model and use the resulting hidden state as an initial one. Looking at Figure 6.2 one can see that neither this, nor just using a zero vector as an initial hidden state is a good idea – even the predictions for the first character in a sequence are completely inconsistent with what we expect to see in the beginning of a sentence.

Another cheap option would be to "warm up" the network by feeding it a longer sequence of characters (We can see in Figure 6.1 that the model starts making good predictions after seeing a dozen or so characters) ending in a dot, followed by a space (This is often done when language models are used to generate text [Sutskever et al., 2011]). The problem with this approach would be exactly that the hidden state represents a summary of the text entered so far. How well the model performs in predicting the initial characters will depend on whether or not what the user enters is consistent with what text was used to warm up the model. Since we do not know in advance what the user is going to type, this will rarely be the case.

So what *would* be a good solution? A short review of relevant literature suggests that the problem of choosing a good initial state is not one that receives a lot of academic interest and is considered more of a practical issue. Some works, both published [Zimmermann et al., 2012, Mohajerin and Waslander, 2017] and unpublished[1] suggest that using a noisy initial hidden state during training or treating the hidden state as a variable to be optimized during training helps solve this issues and can even speed up convergence. Evaluating any of these approaches would require both training a model from scratch and making sure to reset the hidden state only at the starts of sentences. While both of these seem as reasonable things to do when training a production-ready model for Dasher, in the interest of time I experimented with a simpler option that proved to be effective.

Algorithm 1: Learn a good initial hidden state

```
Input: M, a pre-trained RNN language model
Input: D, a training corpus
Input: $\eta$, learning rate
Result: h, a hidden state that minimizes the loss on sentence starts
1 $\mathbf{h} \leftarrow \mathbf{0}$;
2 M.hidden ← h; // Set the initial hidden state to the zero vector
3 while not converged do
4     inputs, targets ← Sample-sentence-starts(D);
5     outputs ← M(inputs);
6     $\mathcal{L}$ ← Cross-entropy(outputs, inputs); // Compute loss
7     $\Delta\mathbf{h} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$; // Gradient w.r.t. the initial hidden state
8     $\mathbf{h} \leftarrow \mathbf{h} - \eta\Delta\mathbf{h}$; // Perform gradient descent step
9 end
```

[1] https://r2rt.com/non-zero-initial-states-for-recurrent-neural-networks.html

6.2.1 Approach

The approach I took was to take a pre-trained model and learn a vector that, when used as an initial state for this model, minimizes the expected loss suffered by the model on the first X characters of a sentence. It works by performing gradient descent on an initial hidden state while keeping the network's weights fixed and is described in Algorithm 1. A couple of details regarding the algorithm:

- In the formulation I use stochastic gradient descent. Of course, other update rules can be used.
- I have not specified the convergence criterion. In my implementation I track the loss on a set of unseen starts of sentences and stop the training procedure when that loss does not improve for four consecutive epochs.
- The training data is fed to the model in mini-batches. This means that the model actually maintains a set of hidden states that is the same size of the mini-batches. As we need to output a single hidden state, the procedure returns the average of these hidden states.

6.2.2 Results

Using the approach described above, I learned good initial hidden states by training on different sequence lengths. I started with an LSTM model with 1024 hidden units trained on the first 60 million characters of the Europarl corpus (This is the model used to produce the results in Figure 6.1). I used starts of sentences from the first 2 million characters of the training corpus to train a good initial hidden state on and sentences from the first 100 thousand characters of the validation corpus as a held-out validation set.

Figure 6.3 and Table 6.1 compare the losses incurred on the beginnings of sentences when using my proposed approach versus naive approaches. Using the learned initial hidden states drastically improved the average loss incurred on the first characters at the start of a sentence. Using numbers in Table 6.1, we can perform a simple back-of-the-envelope calculation: Currently, expert Dasher users can enter around 170 characters per minute [Ward et al., 2000]. Based on my comparison between the language model in Dasher and LSTM language models from the first part of my project, we can make the optimistic assumption that we can double that. Even if we do, on average it would take around 18 seconds for an expert user to enter 100 characters. The results in Table 6.1 suggest we can improve that by 10% for the first 100 characters – that is, our user wins back almost 2 seconds for every time the context is reset – every time they start Dasher, start a new conversation, switch between windows, etc. If they do those things, say, 60 times a day, using a pre-trained hidden state instead of a zero vector one would mean they save an hour every month.

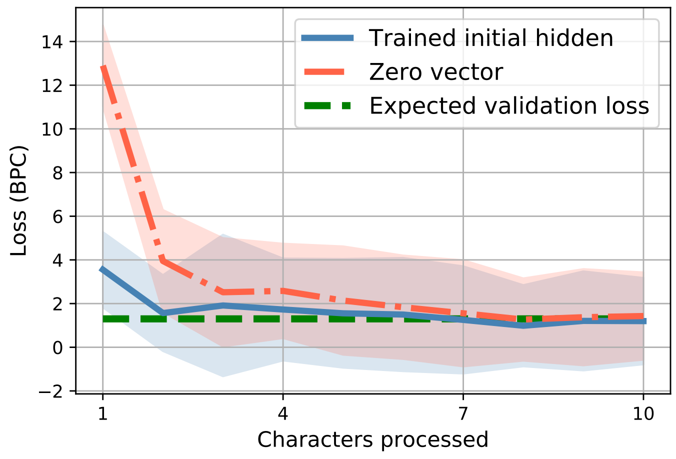


Figure 6.3: Losses suffered on the starts of sentences from the test set. The model and the data are the same ones used to produce the results in Figure 6.1. The red line, which is the same as in said figure represents the loss suffered when using a zero-vector hidden state. The blue one is for a learned hidden state, optimized to minimize the expected loss on the initial 10 characters of a sentence. The shaded regions represent one standard deviation below and above the means and the green line is the loss suffered by the model on the whole, 10 million-character-long validation set. Using a "good" initial hidden state drastically reduces the loss suffered at beginnings of sentences.

Sequence length	1	5	10	20	100
Trained hidden	**3.55**	**2.06**	**1.64**	**1.50**	**1.52**
Zero vector hidden	12.90	4.82	3.15	2.31	1.70
Ratio	3.63	2.34	1.92	1.53	1.11

Table 6.1: Test set losses (in BPC, lower is better) suffered on starts of sentences. The results are averaged over 700 sequences. The model used is an LSTM model with 1024 hidden units trained on the first 60 million characters of the Europarl corpus. The losses for a given sequence length are averaged both over the length of the sequence and over all sequences of a given length. The Ratio columns gives the ratio of the two numbers above it – how many times better using a trained initial hidden state is. The results are all obtained by using the same hidden state optimized for minimizing the loss over sequences of length 10. We can see that using a trained hidden state reduces the loss suffered on initial characters and therefore reduces the time it will take to enter those in Dasher every time the interface is started or the context is switched.

Chapter 7

Conclusions and future work

The main goal in my work this year was to address some of the challenges associated with implementing RNN-based language models in Dasher. The major focus was evaluating different ways to adapt RNN-based language models in terms of their suitability to being incorporated in the system. In this chapter I present a summary of my results, a discussion of the problems I identified and potential future directions.

7.1 Summary

Below is a list of the goals I set in the introductory chapter of my thesis along with a paragraph summarizing what I have achieved in relation to each one:

- *Experimentally motivate the need for model adaptation* – in Section 3.1 I demonstrated that when evaluating data from a given domain using a model trained on data from a different domain, we get worse performance than if we had used data from the same domain to train the model. I showed that even when the domains are really similar, the gap in performance can be as big at 12%.

- *Identify existing solutions to adapting neural models and compare those in the context of Dasher* – I chose three existing methods for adapting RNN models – Dynamic evaluation [Krause et al., 2018], Sparse Dynamic evaluation [Krause et al., 2018] and LHUC [Swietojanski et al., 2016]. In Chapter 4 I described these methods and discussed the challenges associated with implementing them in Dasher.

  In Chapter 5 I compared the performance of these different methods and investigated the importance of the different hyperparameters associated with them. I found that adapting the full set of weights using Dynamic evaluation achieves the best performance out of the three methods by a large margin. In my experiments I got a relative improvement over static evaluation (not adapting the parameters) of 6% and 39.6% when evaluating in-domain and out-of-domain data, respectively. The best figures for the other two methods were 1% and 16% for sparse Dynamic evaluation and 0.4% and 13.6% for LHUC.

I found the most important hyperparameters for all methods to be the learning rate and the frequency of the updates (the sub-sequence length). I found some benefit in tuning the decay rate when performing dynamic evaluation, which is in agreement with the findings of [Krause et al., 2018]. However, in my experiments I found very little benefit in using the RMS update rules proposed by [Krause et al., 2018].

I also find that performing adaptation can destabilize the state of the RNN language model, leading to very poor predictions and causing the losses the model suffers to explode. In Section 5.2 I present an example of this happening during evaluation and argue that it is possible for this to happen during normal use of Dasher, which would be undesirable. I provide a high-level discussion of potential solutions in Section 4.4.

- *[Optional] Propose a novel architecture/technique/algorithm for adapting language models to rapid domain-shifts that improves on existing approaches or is much more suitable for Dasher* – I made almost no progress on this goal. In Section 5.5 I experiment with using Dynamic evaluation to adapt only a subset of the network's weights with the hypothesis that this can alleviate the problem of exploding losses. I find that this approach achieves similar, yet worse results compared to adapting the full set of weights. This approach was unsuccessful and had limited to no novelty.
- *Sketch the implementation of at least one of the identified solutions.* – In Section 4.4 I described the main problems associated with the implementation and presented possible solutions. The major problem with the adaptation methods I considered is the exploding loss. This is unpredictable and can render the system useless if not handled properly. I gave a high-level description of possible solutions, none of which definitively solves the problem.
- *[Updated] Propose a method for making good initial predictions* – In Chapter 6 I proposed a simple method for learning an initial state for an RNN language model which would incur small losses when used to make initial predictions in Dasher. I provided experimental evidence that this works better than naive solutions.

7.2 Discussion and further work

I motivate the need for adapting language models and provided evidence that the solutions I identified drastically improve the performance of language models compared to not using adaptation. And last year I found that RNN-based language models perform better than the current language model in Dasher. In this sense, if these methods can be made to work reliably in Dasher, this will allow users to enter text more quickly, improving the overall quality of the system.

However, I have identified serious issues with the reliability of these methods that need to be solved before any effort is made to implement RNN language models in Dasher. If the system cannot guarantee the user that it will successfully adapt to a text they provide or that it will not start outputting very bad predictions while the user is entering text, it is very difficult to make the case for incorporating RNN language models.

With this in mind, I think the most important next step is to work towards solving the issue of exploding losses and I believe the possibilities I discuss in Section 4.4 are good starting points.

Another problem with these models is the amount of computation that would be required for them to work smoothly in Dasher. This is a problem I identified last year and, as I stated in Section 1.1, I decided not to tackle this year for the benefit of investigating approaches to adaptation. Currently, the implementation of the language model and that of the interface dynamics are tightly coupled, which is why naively incorporating an RNN-based model into the system would lead to incurring high computational costs. The estimates of the cost associated with "reversing" adaptation when the user deletes text, which I discussed in 4.4 are also based on the assumption that the RNN model would be implemented naively. At this point it is clear that the issues with the speed of predictions and the details of implementing evaluation methods are connected and should be discussed together in future work.

Bibliography

[Al-Rfou et al., 2018] Al-Rfou, R., Choe, D., Constant, N., Guo, M., and Jones, L. (2018). Character-level language modeling with deeper self-attention. *CoRR*, abs/1808.04444.

[Blodgett and O'Connor, 2017] Blodgett, S. L. and O'Connor, B. (2017). Racial disparity in natural language processing: A case study of social media african-american english. *CoRR*, abs/1707.00061.

[Caliskan et al., 2017] Caliskan, A., Bryson, J., and Narayanan, A. (2017). Semantics derived automatically from language corpora contain human-like biases. *Science*, 356:183–186.

[Chelba et al., 2013] Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., and Koehn, P. (2013). One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005.

[Chen et al., 2015] Chen, X., Tan, T., Liu, X., Lanchantin, P., Wan, M., Gales, M., and Woodland, P. (2015). Recurrent neural network language model adaptation for multi-genre broadcast speech recognition.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

[Chung et al., 2016] Chung, J., Ahn, S., and Bengio, Y. (2016). Hierarchical multiscale recurrent neural networks. *CoRR*, abs/1609.01704.

[Chung et al., 2014] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

[Cleary and Witten, 1984] Cleary, J. G. and Witten, I. H. (1984). Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402.

[Daumé III, 2009] Daumé III, H. (2009). Frustratingly Easy Domain Adaptation. Technical report.

[Duchi et al., 2010] Duchi, J., Hazan, E., and Singer, Y. (2010). Adaptive subgradient methods for online learning and stochastic optimization. Technical Report UCB/EECS-2010-24, EECS Department, University of California, Berkeley.

[Elman, 1990] Elman, J. L. (1990). Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211.

[Finn et al., 2017] Finn, C., Abbeel, P., and Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Grave et al., 2016] Grave, E., Joulin, A., and Usunier, N. (2016). Improving Neural Language Models with a Continuous Cache.

[Graves, 2013] Graves, A. (2013). Generating Sequences With Recurrent Neural Networks.

[Harada et al., 2001] Harada, T., Araki, O., and Sakurai, A. (2001). Learning context-free grammars with recurrent neural networks. volume 4, pages 2602 – 2607 vol.4.

[Hochreiter and Urgen Schmidhuber, 1997] Hochreiter, S. and Urgen Schmidhuber, J. (1997). LONG SHORT-TERM MEMORY. *Neural Computation*, 9(8):1735–1780.

[Jacobs et al., 1991] Jacobs, R., Jordan, M., J. Nowlan, S., and E. Hinton, G. (1991). Adaptive mixture of local expert. *Neural Computation*, 3:78–88.

[Józefowicz et al., 2016] Józefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *CoRR*, abs/1602.02410.

[Khandelwal et al., 2018] Khandelwal, U., He, H., Qi, P., and Jurafsky, D. (2018). Sharp nearby, fuzzy far away: How neural language models use context. *CoRR*, abs/1805.04623.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[Klimt and Yang, 2004] Klimt, B. and Yang, Y. (2004). Introducing the enron corpus. In *CEAS*.

[Kneser and Steinbiss, 1993] Kneser, R. and Steinbiss, V. (1993). On the dynamic adaptation of stochastic language models. pages 586–589 vol.2. IEEE.

[Koehn, 2005] Koehn, P. (2005). Europarl: A Parallel Corpus for Statistical Machine Translation. *MT summit, volume 5*.

[Kolen and Kremer, 2001] Kolen, J. F. and Kremer, S. C. (2001). *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*. IEEE.

[Krause et al., 2018] Krause, B., Kahembwe, E., Murray, I., and Renals, S. (2018). Dynamic Evaluation of Neural Sequence Models.

[Kuhn and De Mori, 1990] Kuhn, R. and De Mori, R. (1990). A cache-based natural language model for speech recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(6):570–583.

[Lahiri, 2014] Lahiri, S. (2014). Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 96–105, Gothenburg, Sweden. Association for Computational Linguistics.

[Le et al., 2011] Le, Q. V., Monga, R., Devin, M., Corrado, G., Chen, K., Ranzato, M., Dean, J., and Ng, A. Y. (2011). Building high-level features using large scale unsupervised learning. *CoRR*, abs/1112.6209.

[Lee, 2001] Lee, D. Y. (2001). Genres, registers, text types, domains and styles: Clarifying the concepts and nevigating a path through the BNC jungle GENRES, REGISTERS, TEXT TYPES, DOMAINS, AND STYLES: CLARIFYING THE CONCEPTS AND NAVIGATING A PATH THROUGH THE BNC JUNGLE. 5:37–72.

[Lu et al., 2018] Lu, K., Mardziel, P., Wu, F., Amancharla, P., and Datta, A. (2018). Gender bias in neural natural language processing. *CoRR*, abs/1807.11714.

[MacKay, 2005] MacKay, D. J. C. (2005). *Information Theory, Inference, and Learning Algorithms David J.C. MacKay*, volume 100.

[Melis et al., 2017a] Melis, G., Dyer, C., and Blunsom, P. (2017a). On the state of the art of evaluation in neural language models. *CoRR*, abs/1707.05589.

[Melis et al., 2017b] Melis, G., Dyer, C., and Blunsom, P. (2017b). On the state of the art of evaluation in neural language models. *CoRR*, abs/1707.05589.

[Merity et al., 2017] Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and Optimizing LSTM Language Models.

[Mikolov et al., 2015] Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. (2015). Learning longer memory in recurrent neural networks. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*.

[Mikolov et al., 2010] Mikolov, T., Karafiát, M., Burget, L., and Khudanpur, S. (2010). Recurrent neural network based language model.

[Mikolov and Zweig, 2012] Mikolov, T. and Zweig, G. (2012). Context dependent recurrent neural network language model. In *2012 IEEE Spoken Language Technology Workshop (SLT)*, pages 234–239.

[Mohajerin and Waslander, 2017] Mohajerin, N. and Waslander, S. L. (2017). State initialization for recurrent neural network modeling of time-series data. pages 2330–2337.

[Mujika et al., 2017] Mujika, A., Meier, F., and Steger, A. (2017). Fast-slow recurrent neural networks. *CoRR*, abs/1705.08639.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. NIPS-W.

[Paul and Baker, 1992] Paul, D. B. and Baker, J. M. (1992). The design for the wall street journal-based csr corpus. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 357–362, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Pereyra et al., 2017] Pereyra, G., Tucker, G., Chorowski, J., Kaiser, L., and Hinton, G. E. (2017). Regularizing neural networks by penalizing confident output distributions. *CoRR*, abs/1701.06548.

[Radford et al., 2017] Radford, A., Józefowicz, R., and Sutskever, I. (2017). Learning to generate reviews and discovering sentiment. *CoRR*, abs/1704.01444.

[Reddy Gangireddy et al., 2016] Reddy Gangireddy, S., Swietojanski, P., Bell, P., and Renals, S. (2016). Unsupervised Adaptation of Recurrent Neural Network Language Models.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747.

[Sengupta and Friston, 2018] Sengupta, B. and Friston, K. J. (2018). How robust are deep neural networks? *CoRR*, abs/1804.11313.

[Shazeer et al., 2017] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.

[Sutskever et al., 2011] Sutskever, I., Martens, J., and Hinton, G. (2011). Generating Text with Recurrent Neural Networks.

[Swietojanski et al., 2016] Swietojanski, P., Li, J., and Renals, S. (2016). Learning Hidden Unit Contributions for Unsupervised Acoustic Model Adaptation.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Coursera: Neural networks for machine learning.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need.

[Vilar, 2018] Vilar, D. (2018). Learning Hidden Unit Contribution for Adapting Neural Machine Translation Models. Technical report.

[Ward, 2001] Ward, D. (2001). Adaptive computer interfaces. *Cambridge, University of Cambridge*, (November).

[Ward et al., 2000] Ward, D. J., Blackwell, A. F., and MacKay, D. J. C. (2000). Dasher—a data entry interface using continuous gestures and language models. *Proceedings of the 13th annual ACM symposium on User interface software and technology - UIST '00*, pages 129–137.

[Zeiler, 2012] Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

[Zimmermann et al., 2012] Zimmermann, H.-G., Tietz, C., and Grothmann, R. (2012). Forecasting with recurrent neural networks: 12 tricks. In *Neural Networks: Tricks of the Trade*.

[Zoph and Le, 2016] Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578.

Appendices

Appendix A

More experimental results

A.1 Dynamic evaluation on text8

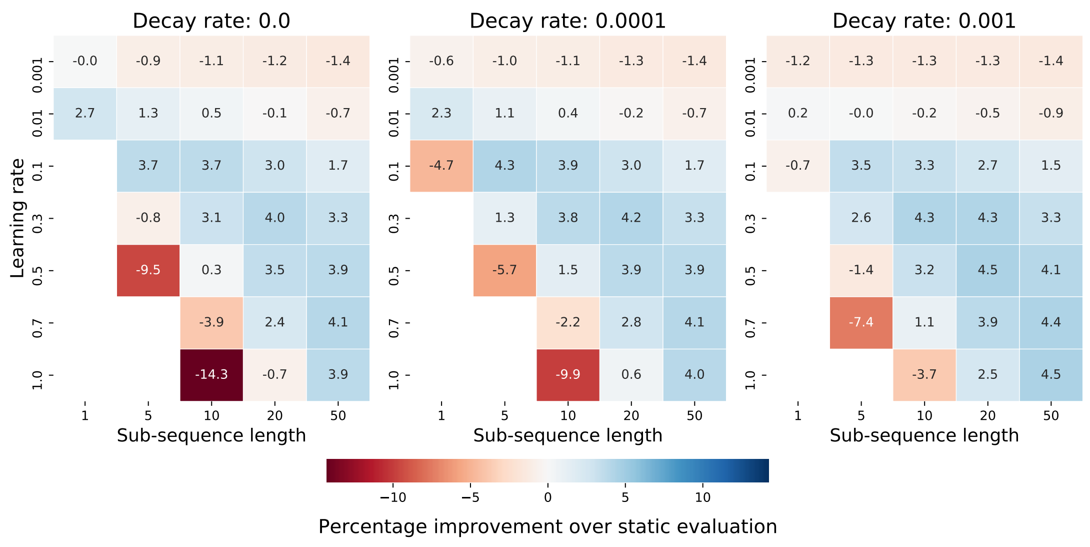


Figure A.1: **(Best viewed in color)** BPC for different settings of the *decay rate*, *learning rate* and *sub-sequence length* parameters. The results are obtained by using Dynamic evaluation with a SGD update rule to evaluate 100 thousand characters of text8 *(Top)* and Enron *(Bottom)* data using and LSTM model with 1024 hidden units trained on 90 million characters of text8 data. The area of optimal hyperparameter settings does not change in comparison to when evaluating texts with a much bigger alphabet.

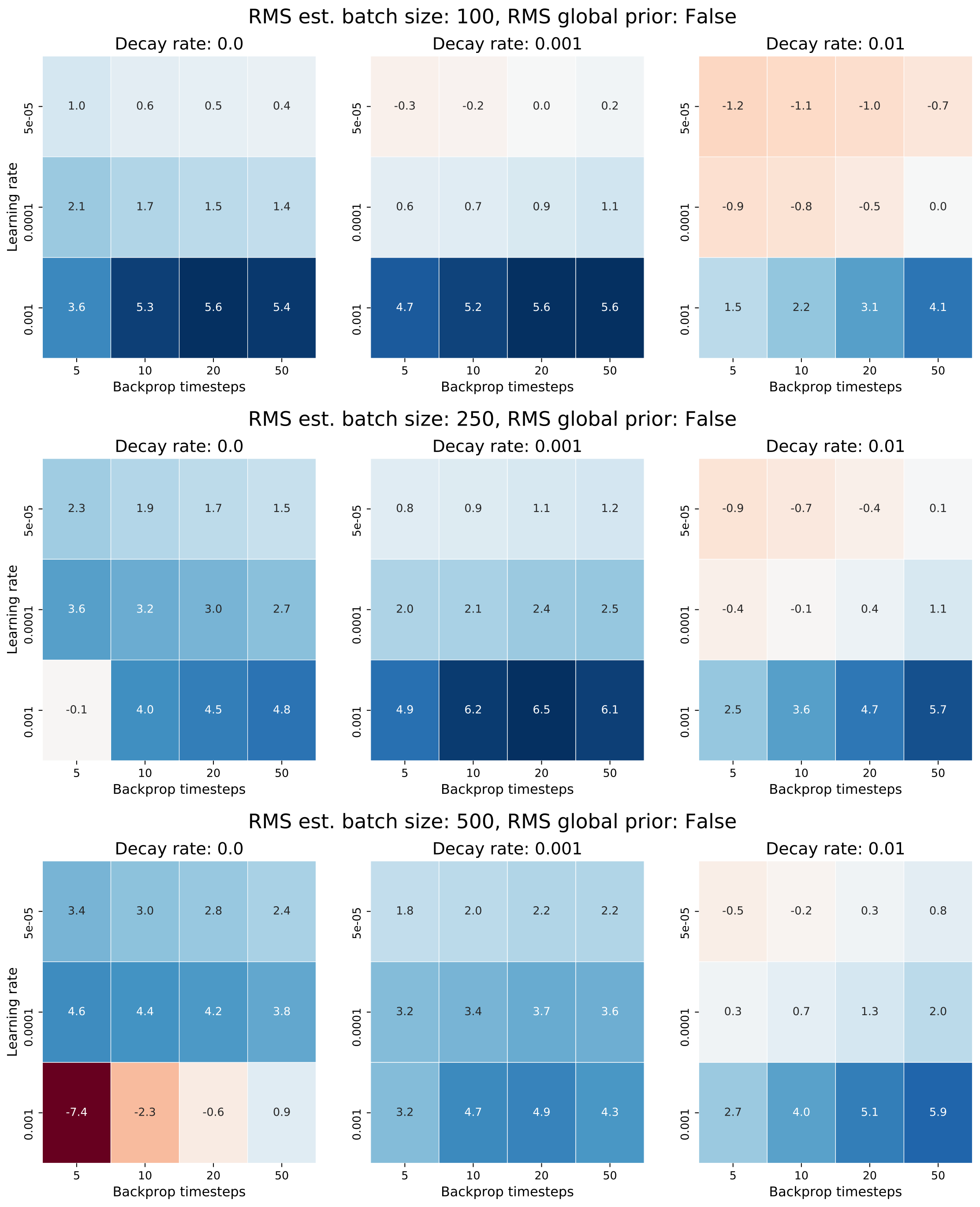


Figure A.2: **(Best viewed in color) (RMS Dynamic evaluation of Europarl)**: BPC for different settings of the *decay rate* **(across rows)**, *RMS estimation batch size* **(across columns)**, *learning rate* and *sub-sequence length* parameters. The results are obtained by using Dynamic evaluation with a RMS update rule to evaluate 100 thousand characters of text8 *(Top)* using and LSTM model with 1024 hidden units trained on 90 million characters of text8 data. The values in the boxes are relative improvements (in %) over the BPC for the corresponding text when not using Dynamic evaluation. The area of optimal hyperparameter settings does not change in comparison to when evaluating texts with a much bigger alphabet.

A.2 LHUC

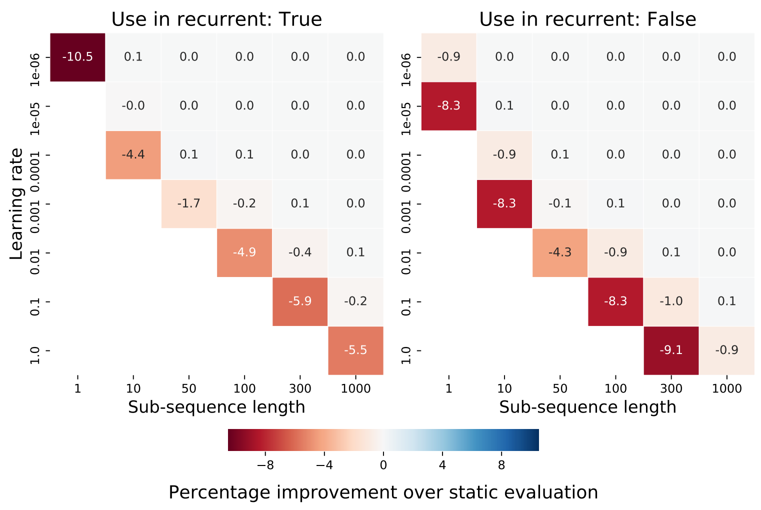
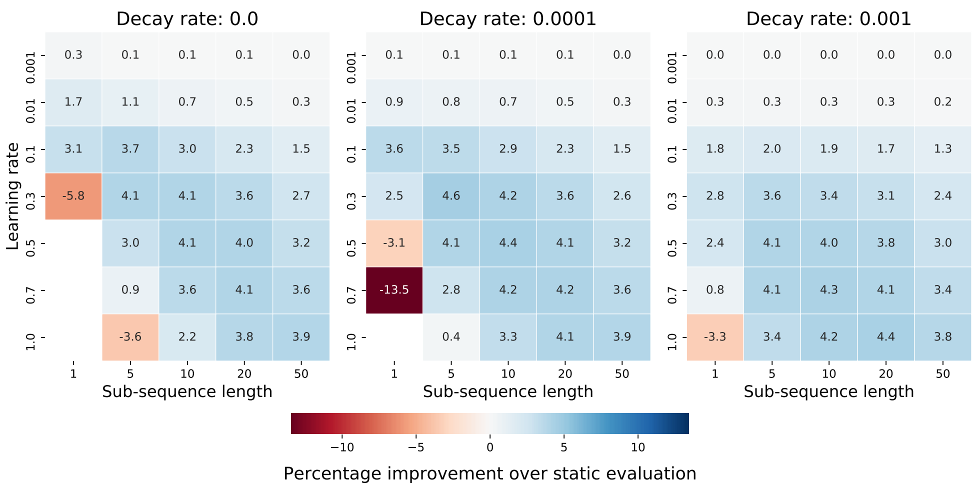
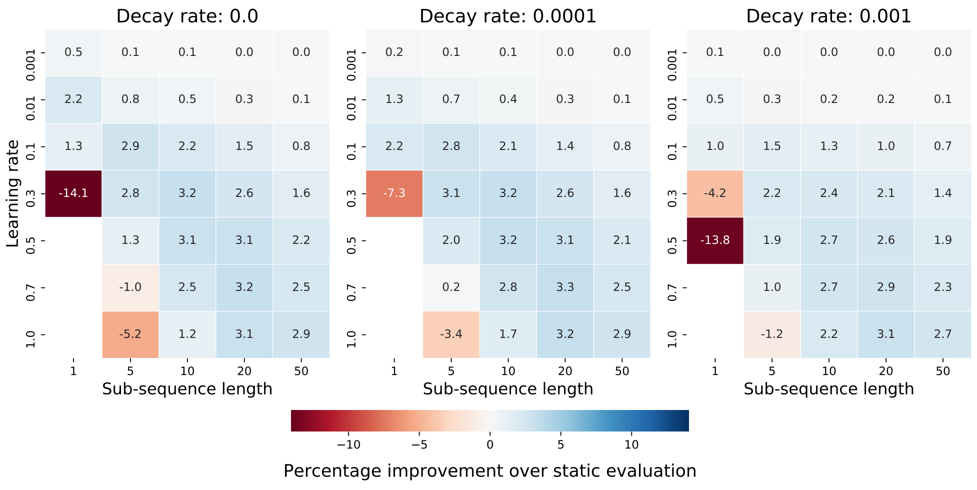


Figure A.3: **(Best viewed in color)** BPC for different settings of the *decay rate*, *learning rate* and *sub-sequence length* parameters. The results are obtained by using LHUC to evaluate 100 thousand characters of the text8 validation set using and LSTM model with 1024 hidden units trained on 90 million characters of text8 data. The values in the boxes are relative improvements (in %) over the BPC for the corresponding text when not using Dynamic evaluation. The empty boxes correspond to settings for which Dynamic evaluation failed – the final loss was more than 15% worse than when using static evaluation.

A.3 Adapting only a subset of the weights



(a) Adapting only the **hidden-to-hidden** weights



(b) Adapting only the **hidden-to-out** weights

Figure A.4: **(Best viewed in color)** BPC for different settings of the *decay rate*, *learning rate* and *sub-sequence length* parameters for dynamic evaluation when only adapting the hidden-to-hidden **(Top)** and only the hidden-to-out **(Bottom)** weights. The results are obtained for static evlaluation of the first 100 thousand characters of the Europarl dataset by a model trained on the Europarl dataset.