Recurrent Neural Networks for Dasher

Yasen Petrov

MInf Project (Part 1)
Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

Dasher is a keyboard-free human-computer interface, which lets users enter text via gestures, possibly using just a single muscle in their body. This makes it particularly suitable for people with severe disabilities, allowing them to communicate with reasonable efficiency. It is driven by a statistical language model, which predicts the probabilities for characters the user might enter next. Under a set of assumptions, the speed with which a user can enter text with Dasher is inversely proportional to the number of bits per character down to which the model can compress the input.

In this report, I explore the potential benefits and the feasibility of substituting the current language model, Prediction by Partial Match, with a Long-Short-Term Memory Network language model, motivated by the fact that LSTM-based models achieve state-of-the-art results in character-level language modelling. I present a comparison of the language modelling performance of the two models, evaluated on the same datasets. I also analyze the amount memory required to store PPM and LSTM models with different hyperparameter settings. Finally, I compare the time taken for the two models to generate probability distributions, considering both CPU and GPU computation for LSTM models.

Acknowledgements

I would like to thank my supervisor, Dr Iain Murray for proposing the project and the patience, care and diligence he put into providing me with feedback on my work.

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 users to enter text without using a keyboard by navigating through the set of all possible strings using continuous gestures. Although entering text with Dasher is theoretically more efficient than using traditional methods like a QWERTY keyboard, the system is not used in any commercial products. A possible reason for this is that, unlike other methods, Dasher requires the user's constant visual attention.

Dasher can be used in conjunction with any device that can capture continuous 2-dimensional (or, in fact, even 1-dimensional) input, meaning that it can potentially be controlled with any single muscle in one's body. This feature makes it particularly suitable for use with eye-tracking devices, allowing severely disabled people to enter text with reasonable speed. Despite the lack of commercial success, Dasher is used by a small number of severely disabled individuals. Improving the system's performance in terms of potential writing speed can mean that such people will be able to communicate more quickly and efficiently.

There are two main features contributing to Dasher's efficiency [Ward et al., 2000]. Not coincidentally, they represent solutions to the problems that make traditional QWERTY keyboards seem inefficient. Firstly, keyboards register presses of keys, which can only be in one of two states, which means that the information rate achievable with keyboards is limited by how quickly a user can tap individual keys. Dasher, on the other hand, registers continuous gestures, which, if measured at high resolution, have the potential of conveying more information [Ward et al., 2000].

Secondly, language is predictable – if I were to type the string *LANGUA* and ask you to guess what character I would enter next, you are unlikely to guess something other than *G*. Keyboards do not make use of this fact, whereas Dasher does – after each character is entered, and internal language model outputs a probability distribution over a given alphabet, which determines the layout of the user interface.

Language models are usually compared by their ability to compress language – a popular metric is bits per character (BPC) and this is the metric that will be used for comparing language models in this report. Under a set of assumptions, discussed in

Section 2.1, the speed with which a user can enter text using Dasher is proportional to the number of bits per character down to which the language model is capable to compress the text being entered, which motivates the search for a model that compressed text better than the current one.

For its language model Dasher currently employs a variant of a compression algorithm called Prediction by Partial Match (PPM) [Cleary and Witten, 1984]. It achieves a text compression ratio of 2.2 BPC on the Canterbury corpus benchmark (`http://corpus.canterbury.ac.nz/`) and compresses the the text used for experiments in the paper introducing Dasher [Ward et al., 2000] to around 2 BPC. It predicts probabilities by keeping count of the number of occurrences of sub-strings of length no more than 5 (The algorithm is described in more detail in Section 2.3.4), making it simple and fast, which motivated its use in Dasher [Ward et al., 2000]. Another important feature of PPM is that it is easy and quick for user input to be fed back into the algorithm in real time, adapting its prediction to the user's vocabulary.

The use of neural networks has improved the state of the art in numerous fields and language modelling is no exception [Melis et al., 2017]. Recent work has led to impressive results in text prediction [Krause et al., 2016], achieving 1.19 BPC on the Raw Wikipedia dataset and 1.13 BPC on the English version of the European parliament dataset using a Long-Short Term Memory networks (LSTM), a variant of the Recurrent Neural Network (RNN) architecture.

However, reviewing literature on PPM and LSTM language models, I could not make a direct comparison of the two approaches on the same dataset. Moreover, most of figures for text compression performance were given for models that were not given any training text prior to evaluation. The results in the single paper I found that explores this possibility [Teahan and Cleary,] suggest that pre-training PPM models improves compression performance. With the results I present in Section 4.1, I aim to provide a fair comparison of LSTM and PPM models in terms of language modelling performance.

Apart from it being state-of-the-art in language modelling at the time Dasher was created [Ward et al., 2000], PPM was chosen as the language model for Dasher because of its small memory requirements, its speed and simplicity [Ward, 2001, Ward et al., 2000]. When considering whether to use another language model in the place of PPM, these factors have to be taken into account. In Sections 4.2, I aim to compare the amount of memory required to use PPM and LSTM models and whether using the latter is feasible. As Dasher needs to build a tree of related predictions, the speed at which a model can predict a distribution over the possible characters given a context is very important. In this report I aim to establish what are the requirements Dasher imposes on the prediction speed for its language model and compare how LSTM and PPM models perform with respect to those requirements. I present results on prediction speeds in Section 4.3.

As mentioned above, PPM can adapt to the user's writing style as they input text – a feature that would be desirable in any language model to be incorporated in Dasher. Research suggests recurrent networks can also update their predictions as they evaluate text and that this benefits compression performance [Krause et al., 2016]. In this report, I also aim to compare the adaptive capabilities of PPM and LSTM models. Results are presented in Section 4.1.

1.1 Main contributions

- I provide a comparison between the text compression performance of LSTM and PPM models.
- I evaluate the memory requirements for PPM and LSTM models in the context of Dasher.
- I measure and compare prediction speeds for PPM and LSTM models.
- I implement a framework for performing and monitoring sequence modelling experiments with LSTMs

Chapter 2

Background

In this section I briefly describe how the Dasher system works, I summarize material from Information Theory and Language modelling that is relevant to my project. I also give an overview of some of the most popular Recurrent Neural Network architectures. In describing Dasher, I was heavily influenced by [Ward et al., 2000] and the material on Information Theory follows [MacKay, 2005, Ward, 2001]

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 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 and an animated demonstration of entering text with Dasher can be found at `http://www.inference.org.uk/dasher/images/newdasher.gif`.

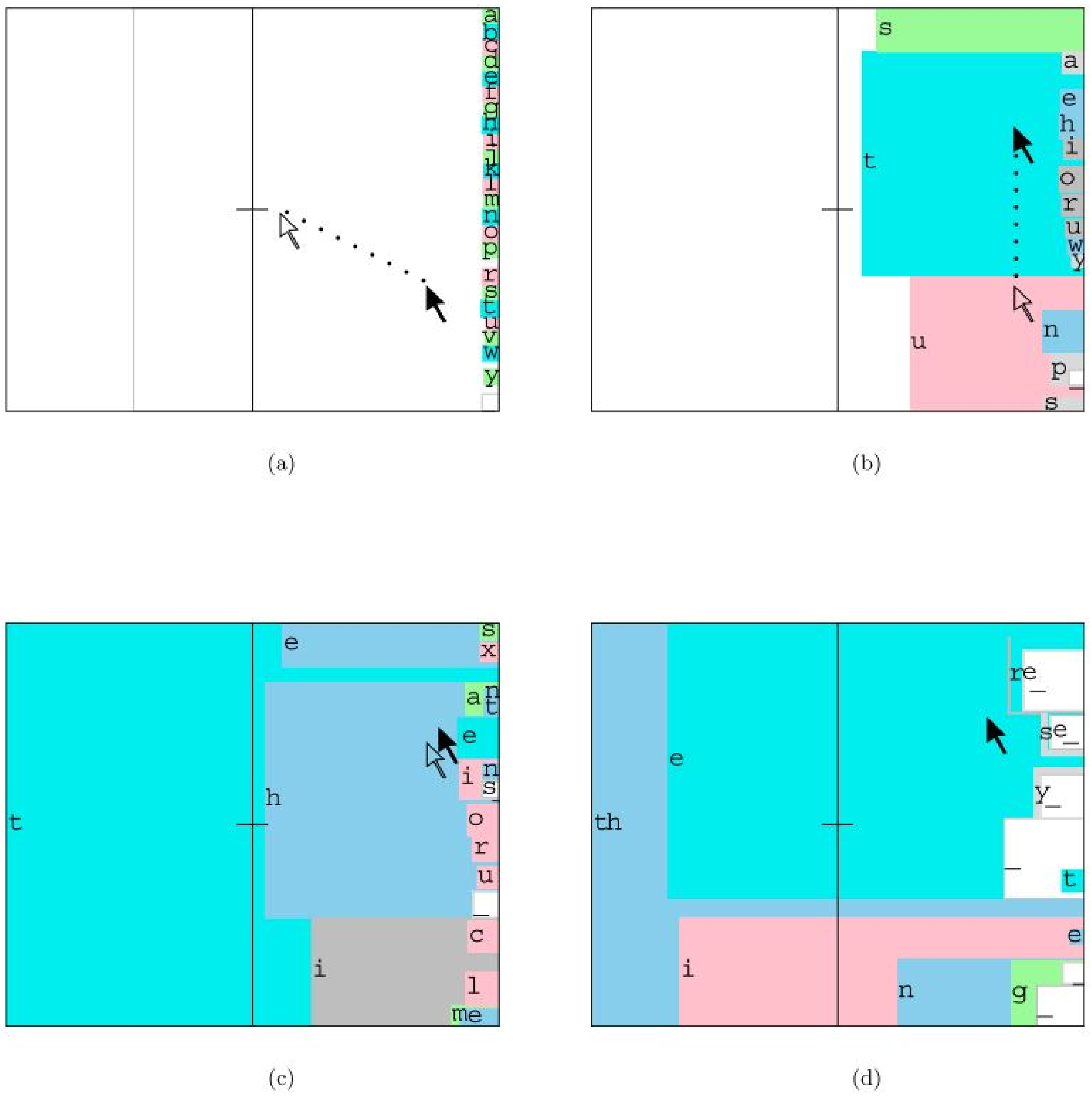


Figure 2.1: (a) Initial configuration. (b),(c),(d) Three successive screen shots from Dasher showing the string *the* being entered. The underscore symbol represents a space. The figure and caption are taken from [Ward, 2001].

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 according to the language model.

Considering 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, as in arithmetic coding, 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:

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

where each c_i is a symbol and $P_L(c_n|c_1\dots c_{n-1}$ if 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 formalize how improving predictions for the next character will improve 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 real 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 distance of the pointer from the crosshairs in the middle of the screen (The specifics of this are given in Section 4.4 of [Ward, 2001]). This dynamic entails that the number of timesteps, T_s, taken for a string s of height h to be entered is proportional to $1/\log h$. Since, as mentioned above, h is proportional to the probability assigned to this string by the language model, $P_L(s)$, we have:

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

– the time taken to input a string is proportional to the negative log probability assigned to that string by the language model.

2.1.4 Some caveats

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 rescaled 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.

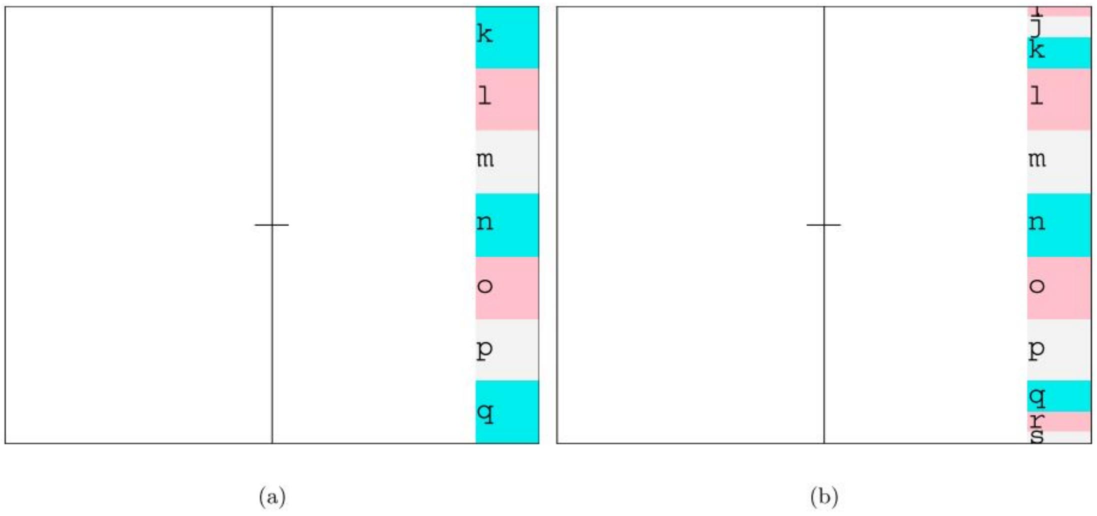


Figure 2.2: (a) shows a square aspect ratio display and (b) shows the same configuration after applying a non-linear map to the vertical coordinates of boxes. The figure is taken from [Ward, 2001].

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 ordering of the letters, 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 Information theory

As mentioned above, Dasher's efficiency is largely due to the fact that it leverages the predictability of natural language. Information theory uses the predictability of information streams to compress them. In this section, I present the formalization of the intuition that language is predictable.

2.2.1 Information content and entropy

The Shannon information content for an outcome x of a random variable is defined as

$$h(x) = -\log_2 P(x) \tag{2.3}$$

and captures the intuition that more surprising events carry more meaningful information.

The entropy of a discrete random variable X , denoted by H, is defined as the expected information content of X:

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_2 P(x_i) \tag{2.4}$$

where $x_1 \ldots x_n$ are all the possible values for X. When the base of the logarithm in the two equations above is 2, both information content and entropy are measured in bits.

Claude Shannon's source coding theorem tells us that If a statistical model assigns probability $P(s)$ to a string s, consisting on N of symbols, that string can be compressed to $-log_2 P(s)$ bits with arbitrarily small error - in this case we say that the average information content of this string is $-\frac{1}{N} log_2 P(s)$ bits per character.

2.3 Statistical Language modelling

2.3.1 Notation

This section and a lot of the rest of this report deal with operations over characters and strings. For the sake of clarity, I summarize the notation used when presenting equations involving characters and strings here.

- Strings are denoted by uppercase Latin letters – S
- Characters are denoted by lowercase Latin letters – c
- S_i denotes the ith character of the string S – S_1 is the first character of S
- S_n^m denotes a substring – If S is the string *ASTROPHYSICS*, S_1^5 is the string *ASTRO*
- Concatenation – $c_1 c_2$ is the concatenation of the characters c_1 and c_2. Sc_3 is the concatenation of the string S and the character c_3
- $P(c|S)$ denotes the probability of the character c occurring immediately after the string S
- $\wedge$ and \$ are used as special start-of-text and end-of-text characters, respectively

2.3.2 Basics

A statistical language model is a probability distribution $P(S)$ over strings S that approximates the probability of S occurring as a sentence. For instance, a good language model for conversational English would assign a higher probability to the string

GOOD MORNING. than to the string *PERCOLATORS OSCILLATE VIGOROUSLY.* since the first phrase is far more likely to occur in everyday discourse. Statistical language models use the statistics of training corpora (volumes of text) to estimate these probabilities.

Statistical language models can be classified into character-, word- and context-level, depending on the unit of text they operate on. The mode of operation of Dasher (Refer to Section 2.1) dictates that the language model of choice for the system must be able to output a probability distribution for the next character over all characters in a given alphabet given the string entered by the user so far (the history). Given a string S of length l and a character a, we are interested in the probability that a is seen immediately after S, that is $P(a|S)$

2.3.3 N-gram language models

One way to estimate the probability $P(a|S)$ is to use relative frequency counts – given a large corpus of text, we can compute

$$P(a|S) = \frac{C(S)}{C(S)} \tag{2.5}$$

where $C(s)$ is the number of times the string S occurred in the corpus and $C(Sa)$ - the number of times the string Sa occurred. The problem with this method is that the as S gets longer, the number of possible values for S grows exponentially with the vocabulary size, which means that $C(S)$ will often be equal to 0 for a long S. N-gram language models give an approximate solution to this problem by making the assumption that the string S can be approximated by only its last $n-1$ tokens.

An n-gram is a sequence on n tokens (characters, phonemes, words) found in a given piece of text – *AB* would be an example of a character bigram ($n = 2$) and *I PLAY THE GUITAR* would be one of a word 4-gram.

N-gram models approximate the conditional probability of a character occurring immediately after a given history by approximating the history by the last $n-1$ tokens, which are usually referred to as the *context*. The length of the context is known as its *order*. For instance, under a bigram model $P(N|KITTE) = P(N|E)$ – the probability of seeing N in the context of E, which is of order 1. The order of the model is defined as the order of the context used to make predictions. In general, under an n-gram model:

$$P(a|S) = P(a|S_{l-n+1}^{l}) \tag{2.6}$$

In situations where $l > n$, the context is padded with special beginning-of-string characters to length n.

2.3.3.1 Computing probabilities

One way to compute the probability in 2.6 is to use maximum likelihood estimation. The maximum likelihood estimate for the probability of a occurring after a string S under an n-gram model is

$$P(a|S) = \frac{C(S^l_{l-n+1}a)}{C(S^l_{l-n+1})} = \frac{C(S^l_{l-n+1}a)}{\sum_c C(S^l_{l-n+1}c)} \tag{2.7}$$

Where the sum in equation 2.7 is over all characters in a given alphabet.

If we consider a small training corpus containing only the words (*SMOKER*, *COSMOS*, *SMITE*), we can compute the probability of seeing the character *I* following the string *SM* as given by a trigram model as

$$P(I|SM) = \frac{C(SMI)}{C(SM)} = \frac{1}{3} \tag{2.8}$$

2.3.3.2 Sparsity and Smoothing

Let us consider a 6-gram model operating on the alphabet of all printable ASCII characters, of which there are 95. In order to compute the conditional probability using Equation 2.7 we need the number of times we have seen a context of order 6 for the numerator. For an alphabet size of 95, there are $95^6 \approx 7 \times 10^9$ possible 6-grams, which is in the order of the number of characters in the largest freely available English corpus (https://googlebooks.byu.edu/x.asp).

All of this is to illustrate the problem of *sparsity* – the fact that because of the huge number of possibilities, we will not have seen some of these strings in our training texts. This means that our model will assign zero probabilities to some strings. For instance, a trigram model ($n = 3$) trained on the toy corpus above would tell us that it is impossible for us to see the word *COD*, since

$$\begin{aligned} P(COD) &= P(C|\wedge\wedge)P(O|\wedge C)P(D|CO) \\ &= \frac{C(\wedge\wedge C)}{C(\wedge\wedge)} \times \frac{C(\wedge CO)}{C(\wedge C)} \times \frac{C(COD)}{C(CO)} \\ &= \frac{1}{3} \times \frac{1}{1} \times \frac{0}{1} = 0 \end{aligned} \tag{2.9}$$

Moreover, if the context has not been seen in the training corpus, the denominator in Equation 2.7 will be 0 and $P(a|S)$ will be undefined.

To avoid this, probability mass is taken from more frequent events and distributed to novel ones in a process called *smoothing* or *discounting*. One of the simplest ways to perform this is called *additive smoothing*. When performing additive smoothing, a fixed quantity denoted by δ is added to the n-gram counts. To ensure probabilities sum up to 1, $\delta|A|$ is added to the denominator, where A is the alphabet the model is operating over and $|A|$ is its size. The probability of a character occurring in a context, assigned by an n-gram model with additive smoothing is given by

$$P(a|S) = \frac{\delta + C(S^{l}_{l-n+1}a)}{\delta|A| + C(S^{l}_{l-n+1})} \tag{2.10}$$

Other smoothing techniques are available, some providing better estimates of probabilities – an extensive summary of these is given in [Chen and Goodman, 1998]. In this project I use n-grams as a simple baseline for language modelling so I opted for using additive smoothing in my implementation as it is simple and does not require the use of lower-order contexts.

2.3.4 Prediction by Partial Match

Prediction by Partial Match (PPM) [Cleary and Witten, 1984] is a text compression algorithm that uses n-gram statistics in conjunction with arithmetic coding and was state-of-the-art when Dasher was developed [Ward et al., 2000, Ward, 2001]. This, and the fact that it is simple and fast, were the reasons for it to be the language model of choice for Dasher [Ward et al., 2000, Ward, 2001].

In this section I describe the PPM algorithm and the slightly modified version of it that is used in Dasher.

2.3.4.1 Context blending

As mentioned in 2.3.3, the *order* of a model is the length of the context used to generate predictions – a 4-gram model is a model of order 3. In the previous section I explained how using higher-order contexts introduces the issue of sparsity and how smoothing is used to mitigate that. Another solution to this problem is called $blending$.

Blending is the mixing of predictions of higher- and lower-order models. Let $P_o(c|S)$ be the probability of c following S assigned by a model of order o. A blended model would predict $P(c|S)$ as

$$P(c|S) = \sum_{o=-1}^{m} w_o P_o(c|S) \tag{2.11}$$

where m is the order of the highest-order model used and w_o is a weight, associated with the model or order o. Weights are normalized to sum up to 1. An order 0 model uses the relative frequencies of characters to generate predictions and $P_{-1}(c|S) = 1/|A|$ for all characters in the alphabet. The order -1 model ensures non-zero predictions.

2.3.4.2 Escape probabilities

The weights in Equation 2.11 are determined by what are known as *escape probabilities*. The escape probability for a model of order o is denoted by e_o and is defined in [Cleary and Witten, 1984] as the probability of seeing a novel character in that context. In essence, what the escape mechanism does is, for each context order, divide the probability space into two portions - one for its own predictions, which is weighed by $(1-e_o)$ and one for the predictions of lower order models, which is weighed by e_o. This is formalized in the following equation:

$$w_o = \begin{cases} (1-e_o)\prod_{i=-1}^{o} e_i & \text{for } -1 \leq o < m, \\ 1-e_m & \text{for } o = m \end{cases} \tag{2.12}$$

As discussed in [Cleary and Witten, 1984], when tackling problem of choosing values for the probability of a novel event occurring, there is no theoretical basis for choosing one solution over another. There are, however, numerous methods for computing escape probabilities for PPM, which have been compared experimentally in literature [Cleary and Witten, 1984, Moffat, 1990] including the possibility of not putting an upper bound on the order of the highest-order context used in prediction [Cleary and Teahan,] . A comparison of these is outwith the scope of this report, so I just present the method for calculating escape probabilities used in Dasher.

The literature I managed to find on Dasher [Ward et al., 2000, Ward et al., 2000] only specifies that a modified version of a variant of PPM called PPMD+ is used. As the method for calculating the escape probabilities was not specified in the single version of the paper introducing PPMD+ I could find and I found no further information on this on the official development page of the Dasher project (`http://www.inference.org.uk/Dasher/Develop.html`) either, after close examination of the Dasher codebase, I have inferred that the following formulae for calculating escape probabilities and predictions are used in Dasher:

$$e_o = \frac{\alpha + n_o\beta}{C_o + \alpha} \tag{2.13}$$

$$P_o(c|S) = \begin{cases} \frac{C_o(c)-\beta}{C_o+\alpha} & \text{if } c \text{ has been seen in this context of order } o \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

where $C_o(c) = C(S^l_{l-o}c)$ – the number of times the character c occurred in the context of order o, $C_o = C(S^l_{l-o})$ – the number of times the context of order o has been seen, n_o is the number of distinct characters that have occurred in the context of order o and α and β are tunable hyperparameters. The values for these that are used in Dasher are fixed to

$$\begin{aligned}\alpha &= 0.49\\ \beta &= 0.77\end{aligned} \tag{2.15}$$

As described in Section 4.6 of [Ward et al., 2000], since it is difficult for a user to select a character on the screen when the probability assigned to it is very small, a small, fixed value is added to the probabilities of each character, after which the probabilities are normalized. The default value for this parameter in Dasher is 0.002.

2.4 Neural networks

Neural networks are a mathematical model that computes a function of an input vector $\mathbf{x}$ by passing it through a series of linear and nonlinear transformations.

The simplest Neural network model is called a Multi-Layer Perceptron. It consists of a in sequence of vectors, called *layers*, which are computed in sequence by multiplying the previous layer by a matrix an passing the results through a differentiable non-linear function, called an *activation* function. The first layer contains the inputs and is called the *input* layer. The last layer is called the *output* layers and all other layers are referred to as *hidden layers*. The computation of a hidden layer $\mathbf{h}^{(l)}$ from the previous one, $\mathbf{h}^{(l-1)}$ is given by

$$\mathbf{h}^{(l)} = g(\mathbf{W}^{(\mathbf{l})}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \tag{2.16}$$

where $\mathbf{W}^{(l)}$ is a $K^{(l)} \times K^{(l-1)}$ matrix ($K^{(l)}$ being the size of $\mathbf{h}^{(l)}$) called a *weight matrix* and $\mathbf{b}^{(l)}$ is a vector of size $K^{(l)}$, called a *bias vector*. g is any smoothly differentiable function referred to as an *activation function* or *nonlinearity*, a popular choices is the logistic sigmoid (denoted by σ) function given by

$$\sigma(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}}} \tag{2.17}$$

Classification is the task of predicting which of N classes an object falls into. Classification can be performed by neural networks by representing the object with a vector, passing them through the layers of network that outputs a final layer, $\mathbf{y}$ of size N. In this scenario, the nonlinearity used to compute the final layer is often the $softmax$ function given by

$$softmax(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{K} e^{x_i}} \tag{2.18}$$

It has the property that the sum of the entries in the vector output sum up to 1. This vector can then be treated as a probability distribution over the K classes, with $\mathbf{y_i}$ representing the probability of class i according to the neural network model. In the context of predicting the next character in a piece of text – if we can represent a context as a vector, a neural network can output a probability distribution for the next character over a given alphabet.

The set of entries in the weight matrices and bias vectors $\mathbf{W} = \{\mathbf{W}^{(1)} \dots \mathbf{W}^{(L)}, \mathbf{b}^{(1)} \dots \mathbf{b}^{(L)}\}$ (L denotes the number of layers in the network, excluding the input layer), are called the *parameters* of the model.

Under the paradigm of *supervised learning*, models are trained to output correct predictions by updating their weights after observing a set of correctly labelled input-output pairs $\mathbf{D} = \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \dots (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$. Each $\mathbf{x}^{(i)}$ is an input vector, representing an object to be classified and each $\mathbf{y}^{(i)}$ is a K-dimensional vector of zeroes, containing a single 1 at position k_i, with k_i being the correct class for example i (This way of encoding a class into a vector is called *one-hot encoding*). After processing each example, we can compute a discrepancy between the network prediction and the correct label. This discrepancy is called the *error* on example i, denoted by $E^{(i)}$. The discrepancy is computed using a differentiable function, which is called the *loss function* and denoted by L:

$$E(\mathbf{x}^{(i)}, \mathbf{W}) = L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \tag{2.19}$$

where $\hat{\mathbf{y}}^{(i)}$ is the output of the network on the ith example.

Averaging the errors over all examples, we can obtain the average loss for the dataset, $E(\mathbf{D}, \mathbf{W})$.

Since all steps mapping an input to an error are differentiable, we can use the chain rule to compute the gradient of the loss function with respect to each w of the parameters of the neural network. We can then use a gradient descent method to optimize L w.r.t. the parameters of the network – we update each parameter by taking a small step in the direction of minimizing L:

$$w \leftarrow w - \eta \frac{\delta E}{\delta w} \tag{2.20}$$

the value η is called the *learning rate*.

By repeating this process many times for all parameters, we can get close to a value of the parameters that minimizes the error. This process is referred to as *parameter fitting* or *training*

Instead of perform the update step after computing the error on the whole dataset, we can split the dataset into subsets we call *batches* (sometimes *minibatches*), compute an estimate of $E(\mathbf{x}^{(i)}, \mathbf{W})$ and perform the update step on each of the batches. This process is called *minibatch training* and generally speeds up convergence.

2.4.1 Overfitting, generalization and early stopping

When fitting the parameters of a network to a training set, we run the risk of adapting them too closely to the training data, which can lead to the network performing poorly on novel examples – in this case we say the network *generalizes* poorly to unseen examples. This is called *overfitting*. One way to combat overfitting is, during the training of the network, to periodically measure the error on a separate dataset, called the *validation set*, which is not used to fit the parameters. This gives us an estimate of how well our network will perform on examples it has not seen.

To avoid overfitting, we can stop training the network when the validation loss starts increasing and set of parameters that yielded the smallest validation loss. This process is called *training with early stopping*.

2.4.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) take as input sequences of vectors instead of single vectors Each input to the networks takes the form

$$\mathbf{X} = \{\mathbf{x}_1 \ldots \mathbf{x}_T\} \tag{2.21}$$

Processing the input involves computing a hidden state for each timestep using the equation

$$\mathbf{h}_t = g(W\mathbf{h}_{t-1} + U\mathbf{x}_t + \mathbf{b}_h) \tag{2.22}$$

where g is a non-linear smoothly differentiable function.

The output at each step is given by:

$$\mathbf{y}_t = (V\mathbf{h}_t + \mathbf{b}_y) \tag{2.23}$$

The trainable parameters of the network are the matrices and bias vectors in the equations above. The fact that in computing the hidden state, the hidden state from the previous timestep is used allows the network to incorporate information from previous timesteps in the prediction for the current timestep.

The size of the hidden state of a recurrent network is referred to as its *hidden dimensionality*.

The repeated multiplication by the matrix W introduces the problem of *vanishing and exploding gradients*, which prevents classical RNN models to incorporate information from many timesteps in the past in their predictions.

2.4.3 LSTMs

Long-short Term Memory Networks (LSTMs) [Hochreiter and Urgen Schmidhuber, 1997] are a variant of RNNs that solves the problem of exploding and vanishing gradients [Hochreiter and Urgen Schmidhuber, 1997], allowing for information from further back in time to be used in predicting the output at timestep t. LSTMs compute their hidden state in a more complicated way compared to classical RNNs. The equations defining LSTM networks are given below:

$$\mathbf{f}_t = \sigma(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.24}$$

$$\mathbf{i}_t = \sigma(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.25}$$

$$\mathbf{o}_t = \sigma(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.26}$$

$$\mathbf{c}_t = f_t \circ \mathbf{c}_{t-1} + i_t \circ \tanh(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2.27}$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t) \tag{2.28}$$

$$\mathbf{y}_t = V \mathbf{h}_t + \mathbf{b}_y \tag{2.29}$$

where $\circ$ denotes the element-wise product between two vectors.

The vector $\mathbf{c}_t$ is called the *cell state* of the LSTM. Intuitively, it serves the purpose of storing long-term memory. It is only used in element-wise addition and multiplication operations, which is the key to how LSTMs solve the exploding and vanishing gradients problem.

Each of the vectors $\mathbf{i}_t$, $\mathbf{o}_t$, $\mathbf{f}_t$ is called a *gate*. Intuitively, all gates serve the following purposes by taking part in the element-wise multiplications in the equations above:

- $\mathbf{i}_t$ – the *input* gate – controls what information from the input at timestep t is ”leaked” to the hidden state – ”what is relevant about the current input”
- $\mathbf{f}_t$ – the *forget* gate – controls what information from the cell state (the ”long-term memory” of the model) is ”leaked” to the current cell state – ”what about what we know is worth forgetting, given what we have seen”
- $\mathbf{o}_t$ – the *output* gate – controls what information from the cell state is ”leaked” to the current hidden state – ”what part of the long-term memory is relevant to the current prediction”

2.4.4 Character-level text modelling with recurrent networks

When predicting the next symbol of a text with a recurrent neural networks, each character of the input is one-hot encoded to a vector of size $|A|$ – the size of the alphabet and fed through the network. The of the network at time t, $\mathbf{y}_t$ is also of size $|A|$ is passed through a $softmax$ layer and the outputs are treated as a probability distribution over all characters of the alphabet.

During training batches of sequences of data are fed to the network and the weights are updated using Backpropagation through time.

When text is processed one character at a time, as it has to be in Dasher, the cell state and the hidden state have to be stored for processing the next timestep.

Chapter 3

Implementation and data

3.1 Language models implementation

In order to asses the benefits and feasibility of incorporating LSTM language models in Dasher, I compare LSTMs to the current language model in use – Prediction by Partial Match (PPM, Section 2.3.4) – on three criteria – how well can they model language, how much memory they need and how quickly can they output predictions. I also use n-gram models (Section 2.3.3) as a baseline for language modelling performance. Therefore, I needed implementation of all three models and a framework in which I can compare them. The most time-consuming part of this project in terms of system-building was implementing such a framework that is robust, logged and stored all necessary information and trained models, made it easy for me to compare results of multiple experiments and allowed me to set these experiments up quickly. I briefly describe the experimentation framework implementation in this section.

3.1.1 LSTM implementation

A major decision I had to make is choosing a machine learning framework for my LSTM implememtaion from the zoo of possibilities, most of which offer GPU training and come with pre-defined LSTM models, which are two basic requirements I had. I made the call to go with pyTorch [Paszke et al., 2017]. I chose pyTorch because I am familiar with Python, the framework's documentation is reasonably detailed and mostly, it uses dynamic graph evaluation, which makes debugging pyTorch models much easier in comparison with those implemented in a framework that uses static graph evaluation, such as TensorFlow [Abadi et al., 2015], for example.

3.1.2 N-gram implementation

I use n-gram models (Section 2.3.3) as a baseline for language modelling experiments. I use my own naive implementation of n-gram models, which uses dictionaries to store counts for the number of times each character has been seen in each context.

3.1.3 PPM implementation

Aiming at the most fair comparison with the language model implemented in Dasher, I extracted the implementation of PPM from the Dasher codebase (written in C++) and used that in my experiments.

3.1.4 Experimental framework

For LSTM and n-gram language model experiments, I set up an experimental framework that reads experiment specifications stored in `.json` files, tests models with all combinations of hyperparameters in the experiment specification, records and stores training times, best-performing models, validation and training set losses and produces plots of those errors to be used for quick visual inspection of the training procedure. For LSTM models, the framework stores snapshots of the models, which can be used for training to be resumed at a later stage. This feature was very useful when training models with many hyperparameters on big text corpora. A sample specification file is provided in Section A.1.

3.2 Datasets

In comparing the language modelling performance I chose datasets that have been used in literature for measuring LSTM text compression performance in order to be able to compare my implementation and PPM to state-of-the-art models.

I use the English and Spanish versions the Europarl [Koehn, 2005] English-Spanish parallel corpus. These consist of 100 million characters, which, as done in literature, I split into 90 million to training, 5 million for validation and 5 million for testing. The English version contains 214 unique Unicode condepoints and the Spanish one – 218.

The next dataset I use is text8, containing 100 million characters from a 2006 snapshot of Wikipedia and contains only the 26 lowercase letters of the English alphabet plus spaces. I adopt the same 90-5-5 training-validation-test split. The dataset can be found at `http://mattmahoney.net/dc/textdata`.

The last dataset I use is the text used to train English models in Dasher. It contains 318 thousand characters, defining 87 unique symbols. I use the same training-validation-test proportions when evaluation in this dataset.

Chapter 4

Evaluation

4.1 Language modelling

The first goal of my report is to compare how well LSTMs and PPM compare text. In this section I describe the experiments I performed to answer this question.

4.1.1 N-gram experiments

Results from experiments with N-gram models (Section 2.3.3) serve as a baseline for future experiments.

4.1.1.1 Comparing models of different orders

The order of the context used to make predictions is the most important hyperparameter in the n-gram model. In order to examine the effect it has on model performance, I trained n-gram models on the training set and measured BPC on the validation set using both static and dynamic evaluation for each of the corpora described in Section 3.2. The value of the smoothing parameter δ (See Equation 2.10) was set to 0.01 for all experiments.

The main purpose of these experiments was to obtain a baseline for language modelling experiments with PPM and LSTM models. Examining the results, however, can provide useful insights.

Figure 4.1 shows errors on the training and validation sets for the English training text in Dasher and the English version of the Europarl English-Spanish parallel corpus. For the details of the corpora and the training-validation splits, refer to Chapter 3. Comparing the two plots on Figure 4.1, one can notice that the optimal value for the context length (the one that minimizes the loss on the validation set) is different for the two datasets – the 4-gram model has the smallest validation loss for the Dasher text and the 6-gram model performs best for the Europarl text.

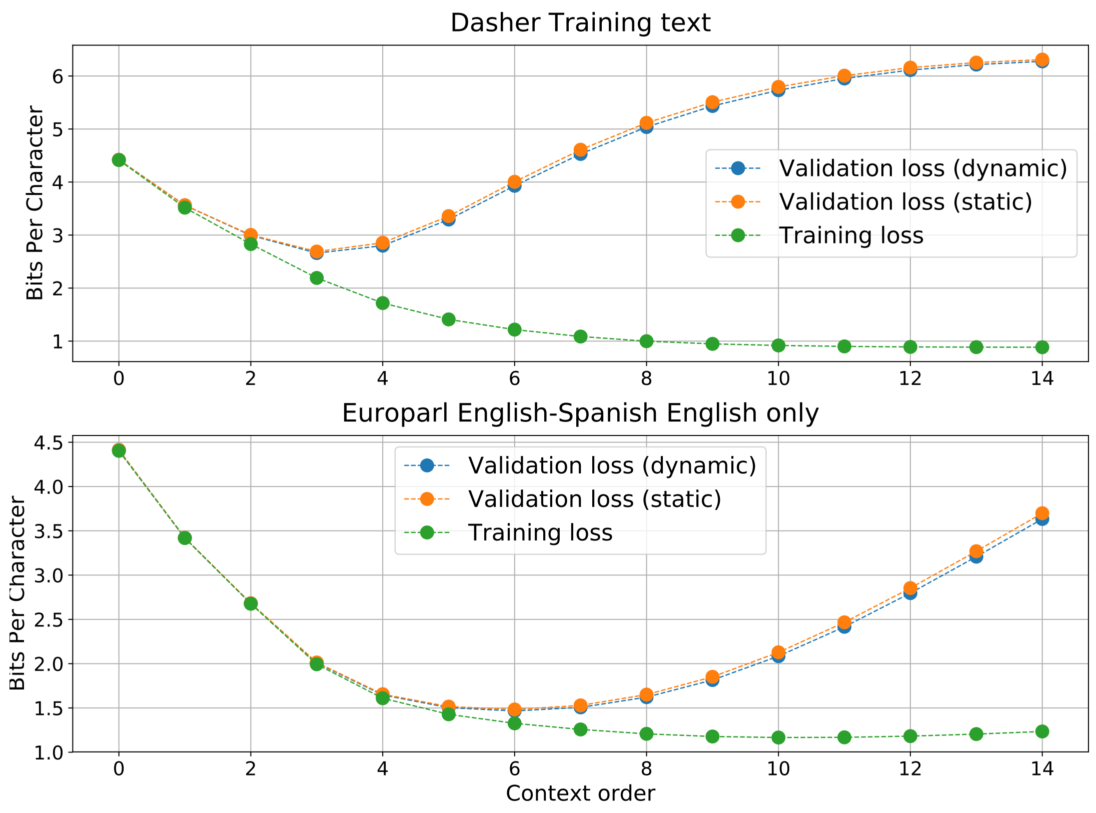


Figure 4.1: N-gram training and validation set losses in bits per character for the text8 corpus (top) and the English version of the Europarl English-Spanish parallel corpus (bottom). Validation loss is given for both static and dynamic evaluation. Note the different scale on the y-axis for the two plots. Also note that the x-axis denotes context order – a context order of 4 corresponds to a 5-gram model.

The reason for that is most probably the difference in the sizes of the two training sets – 280 thousand characters for the Dasher text and 95 million for the Europarl one. A model using a long context to generate predictions that was provided with only a small amount of training text would often have to give predictions for unseen contexts, in which case it would spread probability mass uniformly over all characters in the alphabet, giving a prediction of $1/|A|$ (Equation 2.10), where $|A|$ is the alphabet size. As the size of the training text grows, so does the number of distinct contexts the model sees, making it less likely it is for this model to fall back to a uniform prediction. This suggests that optimal value for the context length will grow with the training set size.

To test this hypothesis, I set up an experiment where n-gram models were trained on different amounts of text from the same corpus and validation loss was measured on the same piece of text for both. One model was trained on the first 95 million characters of the English-only version of the Europarl English-Spanish parallel corpus, and the other – only on the first 10 million. Both models were evaluated on the 5 million-character validation set from the same corpus. Validation losses for this experiment are shown on Figure 4.2.

The results presented in Figure 4.2 suggest that the hypothesis that the optimal value

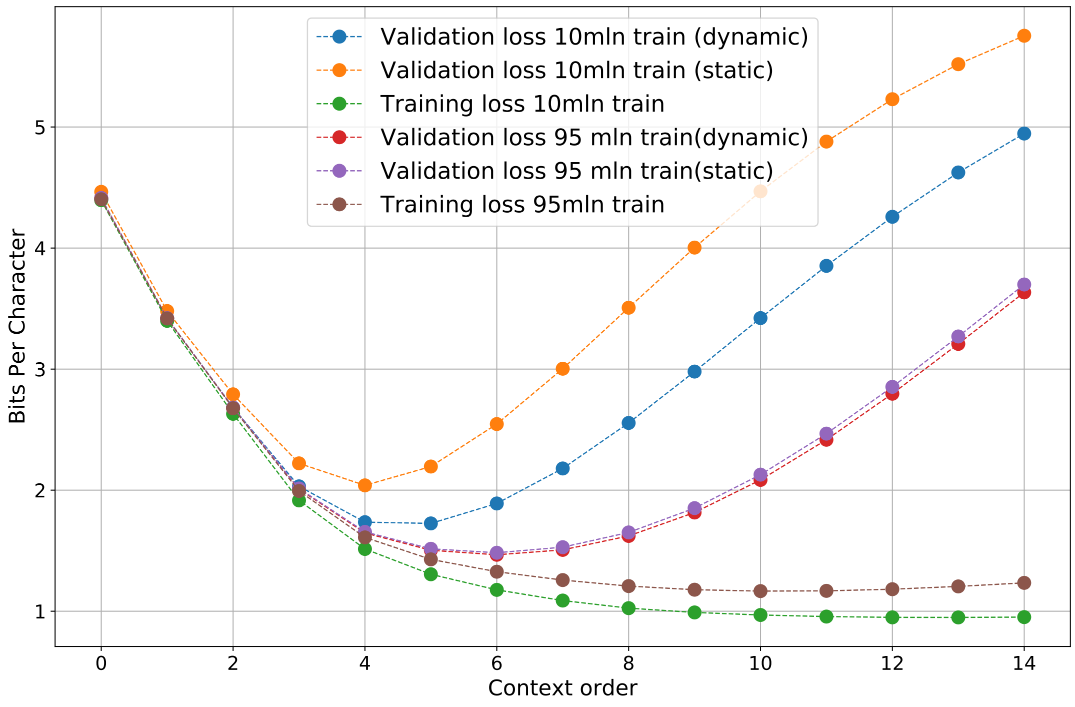


Figure 4.2: N-gram training and validation set losses in bits per character for the full English version of the Europarl English-Spanish parallel corpus and the first 10 million characters of the same corpus. Validation loss is given for both static and dynamic evaluation. Note that the x-axis denotes context order – a context order of 4 corresponds to a 5-gram model.

for the context length grows with the training set size is true – the smallest validation loss for the model trained on 10 million characters is achieved for $n = 6$ (the blue and orange curves on Figure) 4.2, and that for the model trained on 95 million characters – for $n = 7$ (the red and purple curves).

Another interesting artefact seen in Figure 4.2 is the difference in performance gain from using dynamic evaluation for the two different training set sizes. Models trained on the smaller corpus benefit significantly more from using dynamic evaluation – this is testified by the size of the gap between the orange and blue curves, compared to the that of the gap between the purple and red curves.

4.1.2 PPM experiments

The main reason for conducting language modelling experiments with PPM is to obtain results for the performance of version of the algorithm that is currently used in Dasher. However, since the choice of the maximum context order and blending parameters for the PPM is not motivated by experimental results I can find in the literature on Dasher and might have been dictated by hardware restrictions at the time when it was made, the question of whether the current implementation of the algorithm can perform better with different settings for those parameters and more training text is a valid one. The experiments in this section aim to answer this question.

4.1.2.1 Comparing models of different orders

For the initial set of experiments, I evaluated the performance of the PPM algorithm for different maximum context orders with the blending parameters set to their default values of $\alpha = 0.49$ and $\beta = 0.77$ (Refer to Equations 2.13 and 2.14 for the blending method employed by the PPM version used in Dasher).

BPC for Models that have done a single pass over the training sets was measured on the corresponding validation sets using both static and dynamic evaluation. BPC was also measured on the validation sets for models that have not seen any training text. Results are shown in Figure 4.3. Results from these experiments are given in Figure 4.3.

Examining the plots in Figure 4.3, we notice that, similarly to the experiments with N-gram models, the context order for which validation loss is minimized is bigger when training on larger texts. Optimal context orders for the four different corpora are shown in Table 4.1.

However, if we compare the plots in Figure 4.3 to those in Figure 4.1, we see that for PPM, unlike for N-gram models, the validation loss does not increase significantly as we increase the length of the context order beyond its optimal value. This shows how PPM benefits from its escape mechanism (Refer to Section 2.3.4) - when predictions from higher-order contexts are not available, the model assigns a greater portion of the probability space to predictions from lower-order contexts. Careful tuning of the blending parameters in PPM may result in the penalty incurred for using higher-order contexts to be reduced further still.

Table 4.1: Context orders that minimize validation set error for pre-trained PPM models with default smoothing parameters.

Corpus	Training text size	Optimal context order
Dasher text	2.8×10^5	5
text8	9.5×10^7	12
Europarl English-only	9.5×10^7	12
Europarl Spanish-only	9.5×10^7	13

4.1.2.2 On the use of training text

Something I notice on the plots in Figure 4.3 is that the benefit of using a training text is smaller for bigger corpora – the gap between the green line and the orange and blue lines is much greater in the topmost plot compared to the ones below.

It would not be unreasonable to ask, if we do not reap great benefits in language compression from using a training text, why go through the trouble of doing it? The main reason for this is that an untrained model would perform poorly until it sees enough

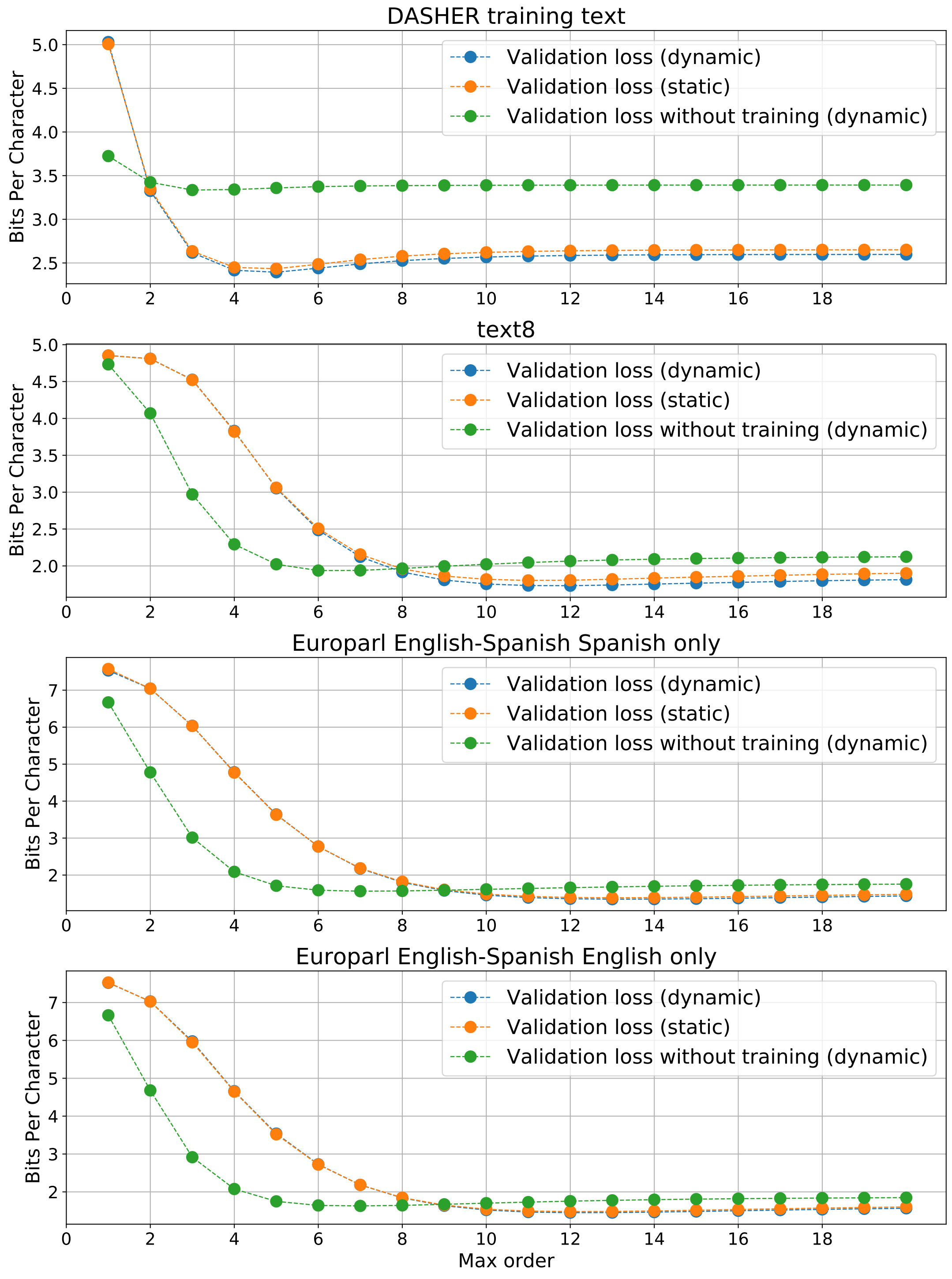


Figure 4.3: PPM validation set losses in bits per character for different corpora. Validation loss is given for both static and dynamic evaluation as well as for untrained versions of PPM. Note the different scales on the y-axes.

data in order to make accurate predictions. This is illustrated in Figure 4.4. To generate the data for the plots on the figure, I kept a running average of the validation loss for PPM while making a pass through the text8 corpus validation set, which was updated after every 10 thousand characters. The orange line shows the validation loss for a model that has made a pass through the training set for this corpus and the blue one – the loss for a model that has not seen any data prior to evaluation. It is clear from the plots that the latter performs much worse at the beginning – it would not be acceptable for a user to have to input tens of thousands of characters before the language model begins outputting reasonable predictions.

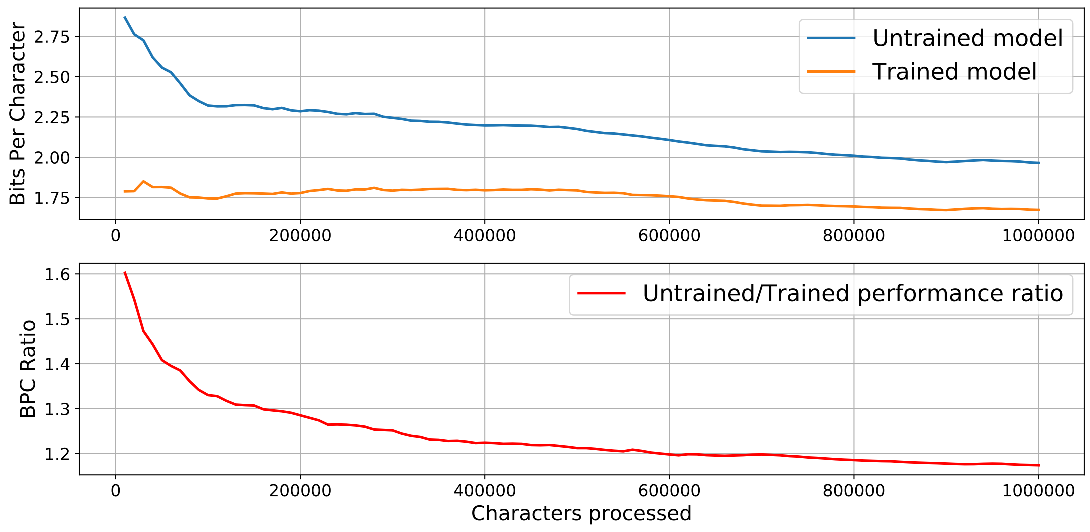


Figure 4.4: Running average of the loss over the first 1 million characters of the text8 validation set for a model that has seen the training text and a one that has not (top) and the ratio of the two averages (bottom)

4.1.2.3 Blending parameters

Reading though the literature I found on Dasher, I was not able to find any motivation behind the choice of the blending parameters α and β for the PPM model used in the system (Refer to Equations 2.13 and 2.14 for the blending method employed by the PPM version used in Dasher).

In order to check if these we reasonable settings of the parameters, I measured BPC for compressing the validation set of the Dasher text for models trained on the training set for different settings of the two parameters. Values in the range $[0.01; 0.97]$ were considered for both parameters. The maximum context order was set to 5, as this was the optimal setting from the initial experiments. The results are shown in Figure 4.5.

The BPC on the validation set for the default parameters was 2.39 and the BPC for the optimal settings of the parameters on the grid, which was $\alpha = 0.97$ and $\beta = 0.77$ was 2.38. This experiment shows that the setting of the parameters in Dasher, however it was decided on, is reasonable in this setting.

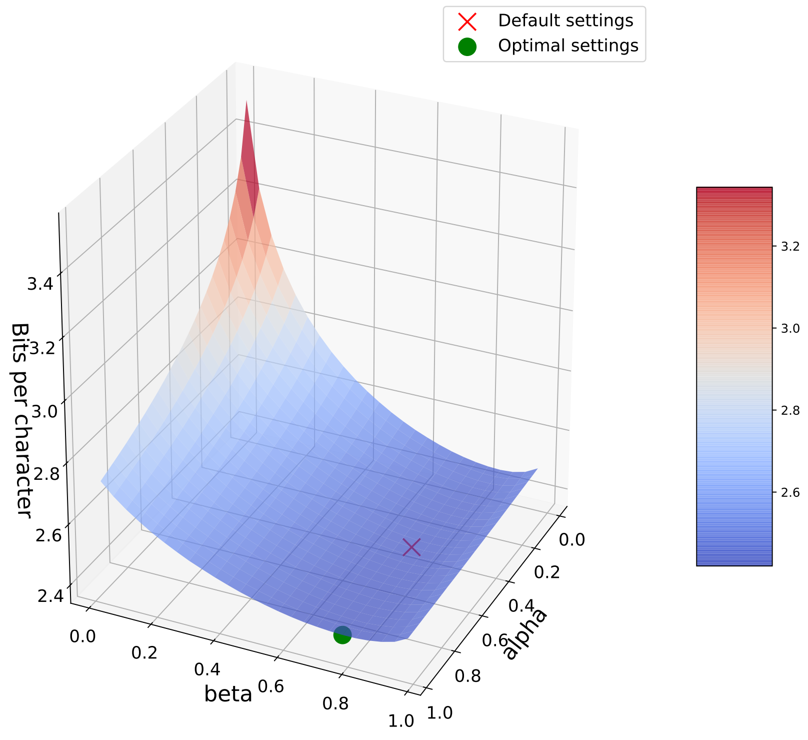


Figure 4.5: BPC on the Dasher validation set for a pre-trained PPM5 model for different settings of the α and β blending parameters

4.1.3 LSTM experiments

The main motivation behind investigating the possibility of incorporating an LSTM language model (Section 2.4.3 into Dasher is that LSTMs have achieved state-of-the-art performance in character-level language modelling tasks [Krause et al., 2016, Chung et al., 2016]. In order to perform a comparison between the language modelling capabilities of LSTM and PPM models, I trained and evaluated LSTM models on the datasets described in Section 3.2 – the same ones used when measuring the performance of PPM, aiming at a fair comparison.

Experiment setup I trained models in minibatches of size 100, using truncated back-propagation through time (2.4) on sequences of length 100. The hidden state of the network was initialized as the zero vector and reset every 10000 steps in an effort to capture longer-term dependencies as suggested in [Krause et al., 2016]. I train the network to minimize the cross-entropy between the network's predictions and the vectors, containing the one-hot encodings of the correct symbols, updating the weights using RM-SProp (`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`) with a learning rate of 0.01. I use no other form of regularization but early stopping – after processing every 500 batches, I measure the loss on the validation set and the training procedure returns the model with the lowest validation loss.

I performed experiment with single-layer networks with hidden dimensionality ranging from 32 to 1024. I also performed experiments with a larger network, with hidden dimensionality of 2048. I only experimented on the text8 dataset with this network because of time constraints. The networks trained on the Europarl Spanish and English and text8 corpora were trained for 10 epochs and those that were trained on the Dasher training text – for 50 epochs.

As the Dasher training text corpus was smaller than the datasets, I explored more of the hyperparameter space when training models on it. I experimented with different settings for the number of steps backpropagation through time is truncated, the number of steps the hidden state is reset at and the batch size.

I acknowledge the fact that I have not explored different values for the most of the hyperparameters of the LSTM model for bigger datasets. This is mainly due to time constraints, as training big models on these datasets is time-consuming. In hindsight, especially after obtaining the results in Section 4.3, which suggest that using bigger networks in Dasher might not be a viable option, due to the time taken for them to predict a character, I should have spent more time exploring how well can smaller networks perform in text compression.

Challenges during training In the experimental framework I implemented, I monitor training and validation loss, recording them every 500 batches. While training some models, both of these would suddenly increase drastically. I was able to trace the cause for this to the problem of exploding gradients, which is a common issue with recurrent networks, as described in [Pascanu et al., 2012]. I deal with this issue by scaling down all gradient values larger than a certain threshold, as proposed in the same paper. I used a value of 1 for the threshold. This change stopped the losses from skyrocketing in most training sessions.

Dynamic evaluation I selected the best-performing models for each dataset from my initial static evaluation experiments and evaluated them on the corresponding validation sets using dynamic evaluation with Stochastic Gradient Descent as proposed by [Krause et al., 2016]. I experimented with learning rates ranging from 0 to 10 and decay coefficients in the same range. Dynamic evaluation improved the validation set performance for all models. Best performance was achieved for values of the decay coefficient of 0, suggesting that decaying the weights towards the best setting found with static evaluation had no beneficial effect.

4.1.4 Results

Results on the validation sets for the best-performing models on all datasets are given in Table 4.2. The PPM models that results are reported for use the default values of the smoothing parameters α and β and n-gram models use a value of 0.01 for the δ smoothing parameter.

Both PPM and LSTM models perform better than the n-gram baseline on all datasets but text8, where a 7-gram model outperforms the best PPM model. The gap in performance between PPM and the baseline is quite small on all other datasets but the Dasher training text, where the PPM model provides a 10% improvement over the baseline. This suggests that the default settings of the smoothing parameters for PPM might be poorly tuned to deal with larger datasets.

I chose the models with the smallest validation losses and evaluated them on the test set for each dataset. These are presented in Table 4.3 alongside state-of-the-art results from literature and in Table 4.4

Model	Europarl English	Europarl Spanish	Dasher	text8
LSTM 1024 (dynamic)	**1.15**	**1.09**	2.43	**1.53**
LSTM 1024 (static)	1.21	1.12	2.46	1.55
LSTM 2048 (static)	—	—	—	1.70
LSTM 256 (dynamic)	1.31	1.19	2.43	1.61
LSTM 256 (static)	1.35	1.24	2.46	1.70
LSTM 512 (dynamic)	1.25	1.18	2.38	1.53
LSTM 512 (static)	1.27	1.23	2.40	1.62
LSTM 32 (dynamic)	2.02	1.93	2.00	2.14
LSTM 32 (static)	2.07	2.00	2.04	2.23
LSTM 64 (dynamic)	1.69	1.67	**1.98**	1.94
LSTM 64 (static)	1.73	1.70	2.00	2.00
PPM 10 (dynamic)	1.52	1.46	2.57	1.75
PPM 10 (static)	1.54	1.48	2.62	1.82
PPM 12 (dynamic)	**1.45**	**1.35**	2.59	**1.73**
PPM 12 (static)	1.48	1.39	2.64	1.81
PPM 13 (dynamic)	**1.45**	**1.35**	2.59	1.74
PPM 13 (static)	1.48	1.39	2.64	1.82
PPM 5 (dynamic)	3.54	3.64	**2.39**	3.05
PPM 5 (static)	3.52	3.63	2.43	3.06
4-gram (dynamic)	2.01	2.01	**2.65**	2.30
4-gram (static)	2.01	2.01	2.67	2.30
7-gram (dynamic)	**1.47**	**1.43**	3.93	**1.68**
7-gram (static)	1.48	1.44	4.00	1.72
8-gram (dynamic)	1.51	1.44	4.53	1.71
8-gram (static)	1.53	1.46	4.61	1.76

Table 4.2: Validation set losses in bits per character for different models on the four different corpora I experiment on. The numbers for PPM models in the leftmost column are the maximum context order for the model and those next to LSTM models – the hidden dimensionalities for those models.

Model	Europarl English	Europarl Spanish	text8
LSTM 1024 (dynamic)	**1.15**	**1.08**	**1.56**
PPM 12 (dynamic)	1.46	1.38	1.74
7-gram (dynamic)	1.48	1.43	1.74
BN-LSTM [Cooijmans et al., 2016]	–	–	1.36
HM-LSTM [Chung et al., 2016]	–	–	**1.29**
mLSTM [Krause et al., 2016]	**1.05**	**0.95**	1.40

Table 4.3: Test errors in bits per character on the test sets for three different corpora. The best results for n-grams(my baseline), PPM and LSTM are compared to state-of-the-art LSTM results. The numbers for PPM models in the leftmost column are the maximum context order for the model and those next to LSTM models – the hidden dimensionalities for those models.

Model	Test set error on the Dasher training set (bit/char)
LSTM 64 (dynamic)	**2.00**
PPM 5 (dynamic)	2.36
4-gram (dynamic)	2.65

Table 4.4: Test errors in bits per character on the test set for the Dasher training text corpus. The numbers for PPM models in the leftmost column are the maximum context order for the model and those next to LSTM models – the hidden dimensionalities for those models.

4.1.5 Comparing adaptive capabilities

Quickly adapting to qualitatively different text from the one they have been trained on is an important property for any model that is to be used in Dasher – every user will probably have their unique writing style and might use the software for different purposes – the language one would use to write their thesis is different from the one they will use when communicating with friends online.

In order to compare the capabilities of different models to adapt to qualitatively different text, I take the best models (with smallest validation loss) of each type (PPM, n-gram, LSTM) trained on the first 90 million characters of the English version of the Europarl corpus and evaluate them on the full Dasher training text. I use both static and dynamic evaluation. For LSTM, dynamic evaluation evaluation with Stochastic Gradient Descent as proposed by [Krause et al., 2016], with gradient computed after every 30 characters. I also present results for a ”cold start” – when no training text is given to the model prior to evaluating it on the Dasher training text. The Dasher default smoothing parameters we used for PPM models and n-gram models used additive smoothing (Section 2.3.3) with $\delta = 0.01$.Results are given in Table 4.5.

The first thing to notice is that adapting the weights as the text is being evaluated leads to better performance across the board compared to static evaluation – adapting the weights helps adapt to novel text, as expected.

Another interesting observation is that PPM models with short contexts that have not seen any training text perform better than longer-context models trained on a a lot of irrelevant text. One reason for this might be the poor setting of the hyperparameters for PPM. Another is that PPM stores context counts – if in its huge training text, in the context *ABC* (let us say this was a common substring), the following letter was almost always *D*, the model will be very confident in its predictions of *D* in this context. If, however, this *ABC* is a common context in the novel text but there it is very often followed by *E* instead of *D*, it would take many occurrence of the substring *ABCE* to make the model less confident in predicting *D* after seeing *ABC*; and the number of times it needs to see *ABCE* in order for this to happen grows with the size of the text it was pre-trained on. I have not explored the reasons for why more training text does not help adapting to novel text (or solutions to this problem) further because of time limitations but it is something that should be kept in mind if the the maximum-order of the context in Dasher is to be increased.

In contrast, pre-training on a large corpus yields much greater benefits when using an LSTM model – compared to PPM, LSTMs make better use of large training sets. In combination with the fact that, unlike with PPM, the memory requirements for LSTM models do not grow with the size of the training set (This is discussed in Section), this is a very good argument for using LSTM models in Dasher. This argument, of course, hinges on the assumption that with carefully choosing the smoothing parameters, PPM models cannot be tuned to perform better than LSTMs when adapting to novel text after learning from large corpora.

Model	Pre-trained on Europarl	Cold start
LSTM 1024 (dynamic)	**1.99**	2.88
LSTM 1024 (static)	2.17	–
LSTM 256 (dynamic)	2.19	2.59
LSTM 256 (static)	2.44	–
PPM 12 (dynamic)	2.67	2.68
PPM 12 (static)	2.82	–
PPM 9 (dynamic)	2.58	2.63
PPM 9 (static)	2.72	–
PPM 5 (dynamic)	3.57	**2.52**
7-gram (dynamic)	2.77	3.51
7-gram (static)	2.87	–
6-gram (dynamic)	2.55	3.12
6-gram (static)	2.64	–
4-gram (dynamic)	2.74	2.64
4-gram (static)	2.80	–

Table 4.5: Losses in bits per character on the Dasher training text corpus for models trained on the English version of the Europarl English-Spanish parallel corpus and for models that have not seen any training text (cold start). The numbers for PPM models in the leftmost column are the maximum context order for the model and those next to LSTM models – the hidden dimensionalities for those models.

4.2 Addressing memory requirements

4.2.1 PPM

The results presented in Section 4.1.2 suggest that the PPM model benefits from using a longer context and more data. These are simple changes that can be incorporated in the current system quickly and with little effort. If that is to be done, however, it has to be made sure that higher-order models trained on large corpora can fit in memory.

4.2.1.1 How PPM is implemented in Dasher

The PPM model used in Dasher is implemented using a Trie data structure. The implementation is very similar to the one proposed in [Moffat, 1990].

Each node in the trie represents a context and stores:

- An integer, representing the last symbol in the context
- An integer, storing the number of times this context has been seen
- A pointer to a node representing a context that is one character shorter and is a suffix of the current context – this is also known as the vine pointer. For example, if a node represents the context *ABCD*, its vine pointer will represent the context *BCD*
- A collection of pointers to children nodes – all contexts that are a character longer and have the current context as their prefix. When the number of child nodes is more than a quarter of the alphabet size, this collection becomes an array of the size of the alphabet.
- An integer, the value of which indicates what type of collection the children nodes are stored in.

Additionally, since the class that implements a node defines virtual methods, each node has a pointer to that class' virtual table. If we denote the number of children a node has by C, the size of a node is $28+8C$ bytes on a 64-bit machine and $20+4C$ bytes on a 32-bit one.

Three simple optimizations would be using `byte` type for the indicator the collection type and a variable of type `unsigned short` to store the symbol (It is highly unlikely that we would need an alphabet with a size of more than 2^{16}) as well as getting rid of the virtual functions. These would bring down the size of a node to $16+8C$ and $12+4C$ bytes for 64- and 32-bit machines, respectively. Since these changes are straightforward to implement, I will use latter estimates for node sizes when I calculate the memory required by the language model in Dasher.

To obtain a formula for the amount of memory the PPM language model implementation requires, let us denote the nodes in the trie for a given model with $n_{1...N}$ and let C_i

be the number of children n_i has. Then the numbers of bytes to store such on a 32-bit and on a 64-bit system, denoted by B_{32} and B_{64} respectively, are

$$\begin{aligned} B_{32} &= \sum_{i=1}^{N} 12 + 4C_i = 12N + 4\sum_{i=1}^{N} C_i \\ B_{64} &= \sum_{i=1}^{N} 16 + 8C_i = 16N + 8\sum_{i=1}^{N} C_i \end{aligned} \tag{4.1}$$

4.2.1.2 Experimental results

In order to record memory usage, I trained PPM models of different order on several corpora and periodically recorded the number of nodes in the PPM trie and the total number of pointers to children nodes. This allowed me to assess the effects of context order and corpus size on the memory required for the model.

The effect of the model's order Figure 4.6 shows the amount of memory required on a 64-bit machine to store models of different orders trained on the first 90 million characters of three different corpora and validation set losses in BPC (Refer to Section 2.2) for the same models on the corresponding validation sets (Refer to Chapter 3 for more information on the data sets used).

We can see that for the text8 corpus, we need more than 6 Gigabytes of memory to store the model of order 13, which achieves the smallest loss on the corresponding data set (For detailed text compression results refer to Section 4.1.2).

6 gigabytes is slightly below the average amount of RAM found on desktop computers today (`https://techtalk.pcpitstop.com/research-charts-memory/`) and is also below the minimum memory requirements for some modern computer games (`http://store.steampowered.com/app/377160/Fallout_4/`). It is, therefore, not an unrealistic requirement for a modern piece of software. However, imposing it would mean that users would have to invest in hardware if they want to use Dasher, especially if they want to use the system alongside other memory-intensive applications. Moreover, the C++ project in the most recent fork of the Dasher project has only a 32-bit configuration, limiting the amount of addressable memory to 4 Gigabytes. This means that if the process requires more than this, the code has to be ported to a 64-bit version. This should not be a monumental effort but is worth keeping it in mind.

The two ways we can easily decrease memory consumption is by using a lower-order model or decreasing the training text size.

Looking back to Figure 4.6, we can see that the blue line on the bottom plot is relatively flat after $x = 9$ – we do not get a significant increase in text compression performance. however, looking at the top plot, we see that a model of order 9 requires less than 2 Gigabytes of RAM - 3 times less than the optimal one of order 13, which gives a text compression performance improvement of only 3 and 4 percent over the order 9 model for static and dynamic evaluation, respectively. Carefully adjusting the blending hyperparameters for the PPM algorithm (Described in Section 2.3.4) could potentially make the language performance gap even smaller.

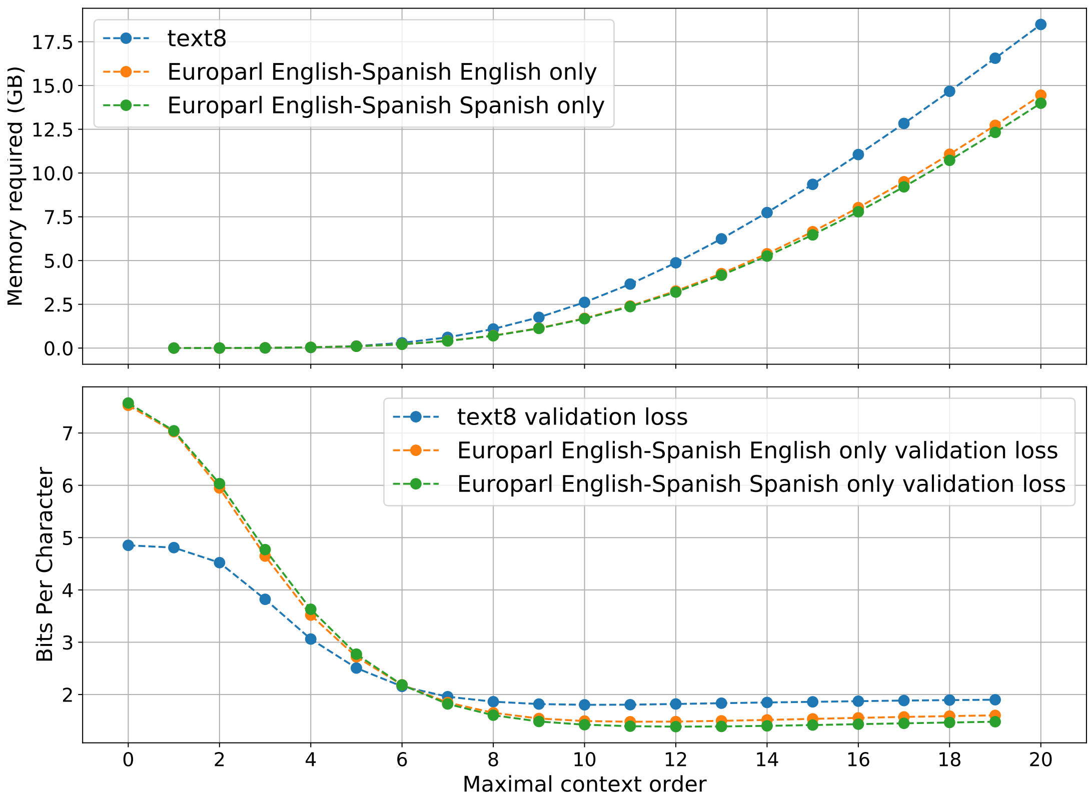


Figure 4.6: Top plot: Memory required (on a 64-bit machine) for models of different orders trained on the first 90 million characters of the text8 corpus and the two monolingual variants of the Europarl English-Spanish parallel corpus. Bottom plot: Losses in Bits Per Character (Lower is better) on the validation sets for the same models and corpora measured using static evaluation

The effect of corpus size As mentioned, the memory footprint of the PPM model in can also be reduced by simply training the model on a smaller piece of text. In order to explore this possibility, I evaluated PPM models of different orders on the text8 validation set after every 10 million characters of the training set they saw. Results are given in Figure 4.7. Noticeably, the beneficial effect of larger training sets is larger for higher-context models.

4.2.2 LSTM

Unlike for the PPM implemented in Dasher, for LSTM models (Refer to Section **??**), the amount of memory required to store a model does not depend on the size of the training text – the model is fully defined by the weight matrices of the LSTM, which do not change size during training.

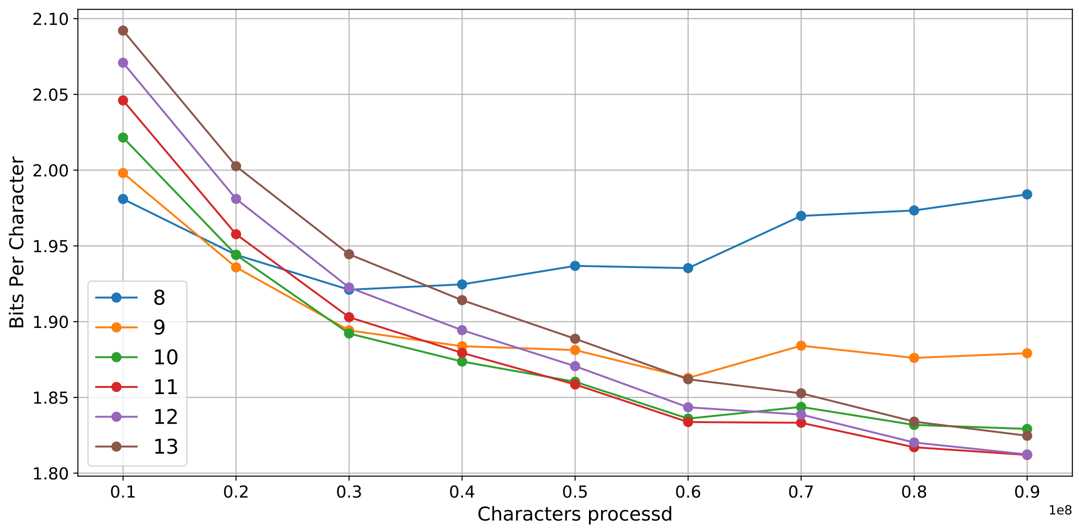


Figure 4.7: Bits per character on the text8 validation set after training on different amount of characters from the training set.

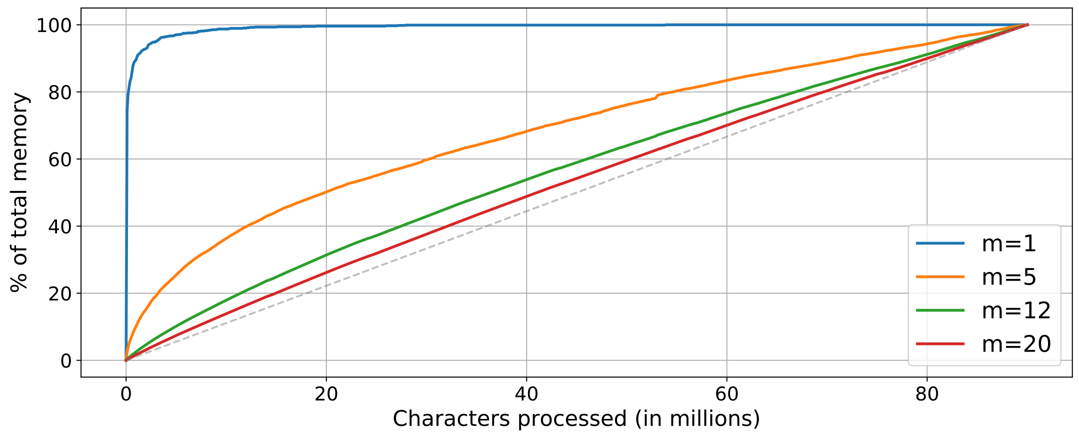


Figure 4.8: Memory consumption during training for models with different maximal orders (m). The curves show what fraction of the memory required for processing the whole text is required after training on a given number of characters.

The size of the weight matrices of an LSTM model depends on two numbers – the size of the hidden state I will denote with H and the number of symbols in the alphabet, which I denote with A. There are three weight matrices in an LSTM (Described in 2.4.3):

- W – the weights of the input connections – W is a $(4H) \times A$ matrix.
- U – the weights of the recurrent connections – U is a $(4H) \times H$ matrix.
- V – the matrix that maps the hidden state to the output – V is an $H \times A$ matrix.

For each of these multiplications matrices, there is a corresponding bias vector – two of them are of size $4H$ and the bias vector for the output layer is of size A.

This means that the total number of parameters I will denote by T, is given by:

$$\begin{aligned} T =& 4HA + 4HH + HA + 8H + A = \\ =& H(5A + 4H + 8) + A \end{aligned} \tag{4.2}$$

Notice that T is $\Theta(H^2)$ and $\Theta(A)$.

The matrix entries are represented by single- or double-precision floating-point numbers, which means that the size of a model in memory will be $4T$ or $8T$ bytes, depending on which one is chosen.

The biggest model I tested is an LSTM with a hidden size of 2048 on the Spanish version of the Europarl English-Spanish Parallel corpus (Refer to Chapter 3 for details on the data sets I used), which has 218 unique characters. The size of this model when using double-precision floating point numbers to store the parameters is just upwards of 150 Megabytes.

This seems to put LSTMs in a great advantage compared to PPM models – we need Gygabytes to store PPM models that use contexts of no more than a dozen symbols (See Section 4.2.1) whereas LSTM models that achieve near-state-of-the-art results (See Section 4.1.3) and can theoretically make use of contexts of unlimited length need several hundred Megabytes regardless of how much text they are trained on.

Parallel prediction Only using the size needed to store the trained model does not make for a fair comparison between PPM and LSTM language models in the context of Dasher. The reason is that comparing them in this manner, I do not take in account how the information about the context in which predictions are made is stored by the two models.

And this is important in Dasher since a tree of predictions needs to be built – when a user enters a string, they see not only the possibilities for the next character but also possible continuations with high probability – see Figure 4.9. Each node in the tree of predictions, displayed as a box in the Dasher interface, defines a unique context.

Building such a tree of predictions is a non-trivial task since the number of nodes in the prediction tree grows exponentially with its depth – the number of nodes in the full tree of possible predictions of length up to 6 for an alphabet of size 100 is in the order of 10^{12}. Storing only the pointers to this many objects would take Terabytes of memory – clearly, a smarter strategy for building a tree of predictions needs to be employed.

In Dasher this strategy is governed by what in the system's implementation is called an *expansion policy*. Based on the positions on the screen of all nodes in the prediction tree, such a policy dictates which nodes to be expanded and collapsed at each frame. This decision is made based on a heuristic for how beneficial it is for a node to be displayed. In the current implementation of Dasher this heuristic is the height of the node's rectangle on the screen (taller boxes are more beneficial), which is proportional to the probability of the string represented by that node (This is described in more detail

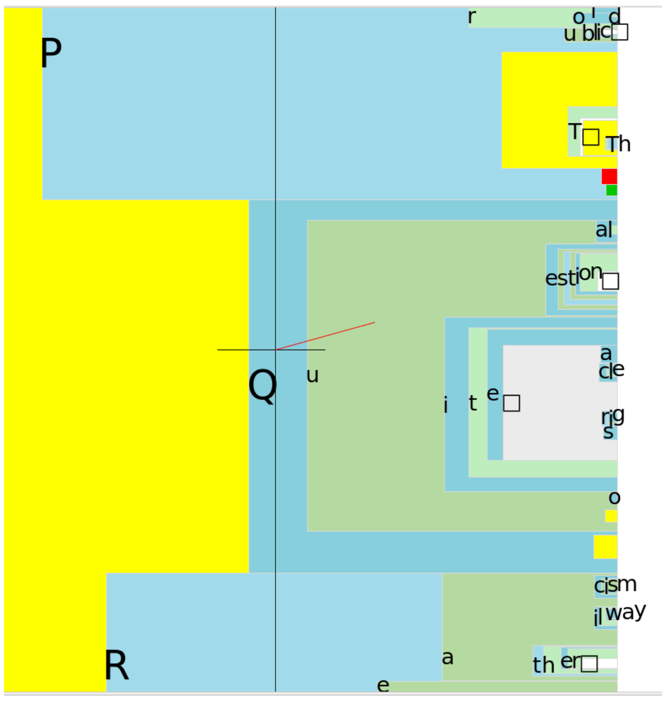


Figure 4.9: Inputting text with Dasher – a tree of related predictions is built and displayed at each frame. Notice that out of the children nodes of the *Q* node, only the *u* node is expanded. Also notice that nodes corresponding to high-probability strings are expanded as long as the probability of the string is large enough for the next node to be displayed (The nodes corresponding to the string *Question* are expanded to depth 8, for example.)

in Section 2.1). Under this policy low-probability branches in the tree of predictions are not explored, greatly reducing the number of nodes that need to be kept in memory. An example of this can be seen in Figure 4.9 – After inputting the letter Q, the only child node that is expanded in the one corresponding to u, since the language model puts almost all of the probability mass on that letter, which is to be expected, since in English, the letter Q is almost always followed by a u. The expansion policy employed in the latest distribution of Dasher also puts an upper bound on the number of nodes kept in memory at the same time – when that is exceeded, the least beneficial nodes currently in the tree of predictions are deleted. This bound is currently set at 3000 nodes.

As mentioned above, every node in the tree of predictions is associated with a context. In the current implementation, these contexts are defined by nodes in the language model tree as described in Section 4.2.1.1. This means that every node in the prediction tree only needs to store a single pointer to a context node.

In an LSTM language model, however, contexts are represented by hidden states (Refer to Section 2.4.3) – each node in the prediction tree would need to store two vectors of length H (the hidden state size, as above) – one for the hidden state and one for the cell state of the LSTM. Assuming 64-bit pointers and double-precision float numbers are used, a representation of a context for an LSTM language model uses $2H$ times more memory than that of a context for a PPM model, which is a single pointer.

In my language modelling experiments, I achieve near-state-of-the-art results using models with $H = 1024$. Given that the number of nodes in memory at any given time is capped at 3000 by the expansion policy, the maximum memory overhead for using LSTM contexts in place of the current PPM contexts is that of storing 6000 double-precision floating-point number vectors of length 1024, which is just over 49 Megabytes. At present the average amount of RAM in personal computers is around 7 Gigabytes and that in portable devices is over 6 Gigabytes (`https://techtalk.pcpitstop.com/research-charts-memory/`), this is a perfectly reasonable memory requirement. Moreover, hidden states for much larger networks would fit in memory.

So far in this chapter I have shown that LSTMs outperform PPM language models in language modelling and that using an LSTM language model instead of PPM in Dasher will not increase the system's memory footprint to a level modern devices cannot handle easily.

4.3 Prediction speed

The next step in comparing the PPM model in Dasher to an LSTM language model is to compare how quickly they perform the required operations, namely training a model and outputting a probability distribution over a given alphabet given a context. The latter is arguably more important, since predictions have to be made in real-time, whereas training can happen before releasing a new version of the software, which can be shipped with the pre-trained models. However, since I consider models that perform dynamic evaluation, I must analyze the time required for a model to ”learn” a symbol in order to make a fair comparison.

In this section I present a theoretical analysis of the number of operations required for both models as well as results from experiments in which I measure the time each model takes to predict and learn text.

4.3.1 PPM time complexity

As detailed in Section 4.2.1.1, the PPM model in Dasher is implemented using a trie data structure. Each node in the trie represents a context and stores the number of times this context has been seen, a collection of pointers to children nodes and a pointer to a node, representing a context that is one symbol shorter, known as the vine pointer (the vine pointer of a node representing the context *ABCD* would point to a node representing the context *BCD*). Both generating a probability distribution and learning a symbol involve traversing a branch of the trie by following vine pointers. These two processes are detailed below.

Learning a symbol When a symbol is learned by a PPM model of maximum order m (Refer to Section 2.3.4 for details on how the PPM model works), the counts for that symbol have to be updated in the nodes corresponding to all contexts of length up to m. For example, if the symbol D is learned in the context ABC by a model with a maximum context order $m = 3$, the counter in the node, corresponding to $ABCD$ will be incremented, then the vine pointer of this node will be followed, leading to the node representing BCD. This will be repeated until the vine pointer leads to the root of the trie.

Extra complexity is added to the operation if the symbol being learned has not been seen in the context its count has to be incremented. In this case, a new node has to be created and added to the collection of children for the context node. In the worst case this collection has to be resized to all the symbols in the alphabet, which takes $O(|A|)$ operations, where $|A|$ is the size of the alphabet.

Since this has to be done for all contexts of length up to m, the worst-case complexity for learning a symbol is $O(m|A|)$.

Outputting a distribution As described in Section 2.3.4, PPM blends the predictions from contexts of order 0 up to m when outputting a probability distribution over the symbols of the alphabet given a context. In Dasher's implementation this is achieved by counting the number of times each symbol has occurred in the maximum-order context, following the vine pointer to the node that represents a context that is one character shorter and repeating this until the root of the trie is reached. At each node counts for up to $|A|$ symbols have to be obtained and $m+1$ context nodes are visited so the number of operations for obtaining a probability distribution is $O(m|A|)$.

It has to be noted that these are worst-case estimates. A more thorough amortized analysis, which I omit because of time constraints should reveal that the number of operations also depends on the size of the training text, since the number of operations performed at each node depends on the number of children that node has, which increases with the number of characters processed (See Figures 4.6 and 4.8).

4.3.2 LSTM time complexity

In contrast with PPM's trie, the structure of an LSTM language model is static – the sizes of the weight matrices and hidden vectors is fixed for a given model – making counting the number of operations required more straightforward. This also means that, like the amount of memory required to store the model, the time taken to output a distribution will not vary with the amount of training text.

Details on how the LSTM models works are given in Section 2.4.3 and the equations defining an LSTM network are copied below for the reader's convenience.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{4.3}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{4.4}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{4.5}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{4.6}$$
$$h_t = o_t \circ \tanh(c_t) \tag{4.7}$$
$$y_t = V h_t + b_y \tag{4.8}$$
$$p_t = softmax(y_t) \tag{4.9}$$

Generating a probability distribution over an alphabet takes a single forward pass, that is, computing equations 4.3 to 4.9. "Learning" a symbol involves computing the loss on the prediction, performing backpropagation and updating the parameters of network. Both the forward and the backward pass take $O(W)$ operations [Sak et al., 2014], W being the number of parameters in the network (ignoring the biases vectors), given by $W = 4H|A| + 4H^2 + H|A|$ – the number of entries in the W, U and V matrices, where H is the size of the hidden state of the network and $|A|$ is the alphabet size. Therefore, the number of operations for both a forward and a backward pass is $O(H|A| + H^2)$.

	PPM	LSTM
Outputting a distribution	$O(m\|A\|)$	$O(H\|A\| + H^2)$
Learning a symbol	$O(m\|A\|)$	$O(H\|A\| + H^2)$

Table 4.6: Number of operations required for learning a symbol and outputting distributions for the two models. The LSTM model always takes the same number of operations whereas the estimate for the PPM model is for the worst case.

The estimates for the number of operations for both models are listed in Table 4.6. The number of operations for PPM grows linearly with both the alphabet size and the maximum context length, which, as discussed in Sections 4.1.2 and 4.2.1, should not exceed 13. The number of operations required by LSTM models, however grows quadratically with the size of its hidden state, which grows into the thousands (Refer to the experiments in section 4.1.3. These facts suggest that using an LSTM language model to replace PPM in Dasher can potentially be prohibitively slow.

There are several reasons why, despite the computational complexity added by matrix operations, the time taken for LSTMs to perform these operations might not be much longer that that for PPM. Firstly, matrix operations are potentially, more cache-efficient, making better use of temporal and spacial locality when accessing memory. Secondly, there are numerous highly-optimized linear algebra libraries [Anderson et al., 1999, Blackford et al., 2002] providing all the necessary routines for performing the operations required by the LSTM model. Thirdly, inputs can be fed into the network in batches, potentially increasing the cache-efficiency of the operations.

And lastly, these operations can be performed on the system's GPU, if one is available, potentially greatly increasing the number of operations per second. Multiple machine learning libraries that make use of optimized linear algebra routines and come with

GPU support and implementations of LSTMs are available for C++ [Abadi et al., 2015, Jia et al., 2014], meaning that the engineering overhead in implementing an LSTM model able to perform operations on a GPU should not be huge.

Insisting on using a GPU, however, has a couple of downsides. Firstly, in order to reap the benefits of performing operations in parallel on the GPU, enough data has to be transferred to the unit at once in order for the benefits to offset the cost of performing the transfer (`https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html`). This suggests that the inputs to the network should be batched – perhaps, before rendering the Dasher interface, instead of asking the model for all of the required distributions one by one, the contexts in which distributions are required can be batched together and the distributions can be obtained in a single batch operation. As mentioned above, batching can potentially lead to increased performance with CPU operations as well. However, implementing it will require extra engineering effort and may present challenges I have not accounted for in this report. The second downside of insisting on GPU support is that users will be have to purchase a GPU in order to use a Dasher version employing an LSTM language model.

The discussion above does not give a concrete answer to the question of whether LSTMs can perform the required operations fast enough in order for them to be used in Dasher. In an effort to give a more precise answer to this question, I designed an ran the experiments described in the next subsection.

4.3.3 Experiments with prediction speeds

In this section I aim to answer three questions – firstly, can LSTM models output probabilities quickly enough in order to be used in Dasher, secondly, how do different factors affect the predictions speeds for PPM and LSTM and finally, how do PPM and LSTM compare in terms of prediction speed.

How quick is quick enough? In order to answer the first question, I need to establish what is the lower limit that Dasher imposes on the prediction speed of its language model. One way to do this is by estimating the number of times the model will be asked for a distribution at each frame. Knowing that in Dasher a new frame is requested 40 times per second (which I found out by inspecting the source code), I can estimate the number of times per seconds the model is asked to provide a distribution. If a model cannot output this many distributions per second, this means that using it in Dasher will lead to frames being generated more slowly, causing an unpleasant user experience and reducing writing speed, which would nullify any benefits in language modelling that using this model might bring.

To make these measurements, I modified the Dasher code so as to record the number of calls to the model's function responsible for generating a probability distribution per frame. I proceeded to input arbitrary strings as quickly as possible (dragging the interface pointer to the rightmost part of the screen) with the text entry speed set to 8.2, which is most too quick for even a very experienced user. I experimented with three of the alphabets available in Dasher - *Lowercase English*, *English with limited punctuation* and *English with numerals and lots of punctuation*, which contain 27, 59 and 103 symbols, respectively. I inputted text continuously for 15 seconds (600 frames) for each alphabet. The mean and standard deviation for each run are given in Table 4.7.

Looking at the results in Table 4.7, I notice that more distributions tend to requested per frame for smaller alphabet sizes. An explanation for this is that the vertical space on the screen is divided among less characters when using a smaller alphabet. This means that boxes will tend to be taller and since the expansion policy (Section 4.2.2) favours the expansion of taller boxes, more nodes will be expanded and more distributions will be required for the model.

More importantly, the numbers in Table 4.7 give me a lower bound on the number of distributions a language model must output per second. To check these numbers, I modified the Dasher code to suspend the thread generating the distributions for a small amount of time. Using an alphabet of 27 letters, I found no discernible difference in performance when I paused the thread for 4 milliseconds ($1/250$ of a second) but when increasing that duration to 10 milliseconds, I could notice a lag when inputting text.

Alphabet size	27	59	103
Calls per second	240.3 ± 3.1	187.4 ± 2.1	162.4 ± 2.3

Table 4.7: Estimates of the mean and standard deviation of the number of calls to the language model's function that returns a probability distribution each second, obtained by recording the number of call per frame while interacting with Dasher.

LSTM prediction speeds Having established what the target prediction speed is, the next question to ask is how quickly LSTMs can perform this operation. In order to measure this, I set up a simple experiment – for a given network size, alphabet size, batch size and sequence length, I repeatedly initialize a batch of inputs with random numbers, reset the network hidden state, make a forward pass through the network and pass the outputs through a softmax layer. I measure how long each of these trials took and report the average time over multiple trials. I acknowledge that the memory access pattern in this experiment is more regular than that might not be the case when using the model Dasher, making the experiment program more cache-efficient and the results reported for this experiment will be a slight over-estimate of the time it will take for the model to produce a distribution in a live system.

I performed experiments on a machine with an Intel Core-i7-4720HQ 2.60 GHz 4-core processor, 8 Gygabytes of DDR3 RAM and an NVIDIA GeForce GTX 950M GPU with 2 Gigabytes of dedicated memory. I made all effort to minimize the number of processes running in the background and averaged values over many runs in order to minimize the effect of the thread executing the LSTM code being interrupted.

As discussed in the previous section, in the simplest case, distributions will be requested from the LSTM language model one at a time – inputs will be fed to the network with a batch size of 1. In the first experiment I present, the performance of models in this scenario. Results for the number of distributions per second for models with hidden state sizes ranging from 64 to 2048 and four different alphabet sizes are given in Figure 4.10. The dots on the plot are the mean number of distributions per second, averaged over 100 runs and the shaded regions represent the standard deviation. I report results for both CPU and GPU computation.

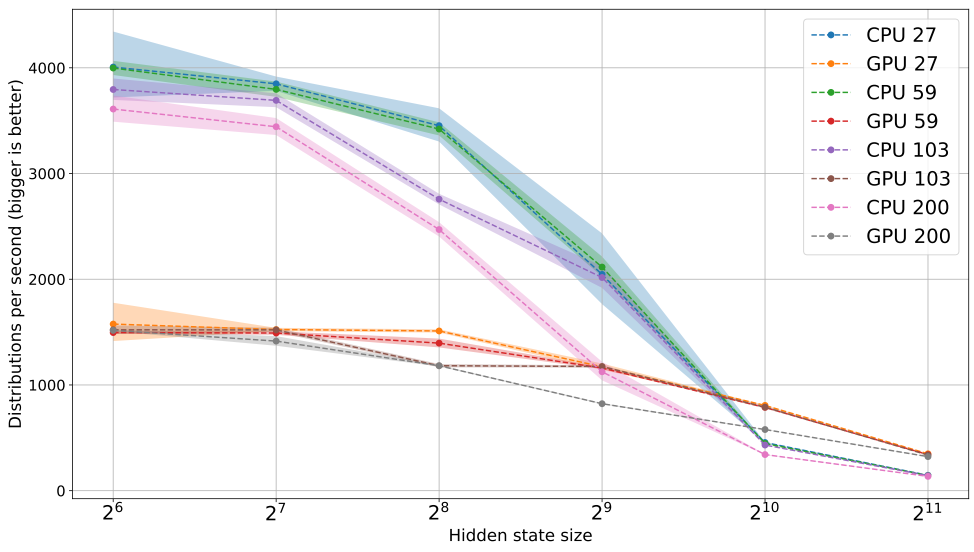


Figure 4.10: Average number of distributions produced by LSTMs of different sizes on a CPU and a GPU and standard deviations over 100 runs. Inputs were fed into the network in batches of size 1. The numbers in the legend are the sizes of the alphabet (respectively the sizes of the input and output vectors).

Examining the plots in Figure 4.10, I notice that performing the computation on the CPU is faster for smaller networks but as the size of the network grows, using the balance shifts. This is in agreement with the discussion of the benefits of GPU computation in the previous section – for larger networks, the benefit of performing many operations in parallel on the GPU outweighs the cost of transferring the data to the unit.

Another, more worrying artefact in Figure 4.10 is the dip in the performance of CPU models for larger network sizes. The exact figures for larger networks are given in Table 4.8. The numbers are pretty close to the minimum requirements for a model to be used in Dasher I established above. Considering that the results slightly overestimate LSTM performance, employing models of these sizes and predicting one character at a time might cause issues for Dasher users with less powerful hardware.

One possible way around this issue might be using bigger batch sizes. As suggested by Figure 4.11, which gives the performance for different batch sizes over an alphabet of 103 symbols, this would speed up predictions. Figure 4.11a shows that batching several dozens of characters together can mean that even models with a hidden size of 2048 can generate a few thousand distributions per second, which should be more than enough. However, since Dasher usually needs less than 10 distributions per frame (Table 4.7, Dasher operates at 40 frames per second), using bigger batches each frame

Alphabet size	27	59	103
LSTM 1024 CPU	455 ± 3	446 ± 3	431 ± 3
LSTM 2048 CPU	145 ± 5	144 ± 1	142 ± 1
LSTM 1024 GPU	806 ± 3	787 ± 4	790 ± 10
LSTM 2048 GPU	348 ± 2	341 ± 1	336 ± 1
Required	240 ± 3	187 ± 2	162 ± 2

Table 4.8: Number of distributions per second (averages and standard deviations) when predicting with a batch size of 1 and an estimate of the number of distributions required in Dasher

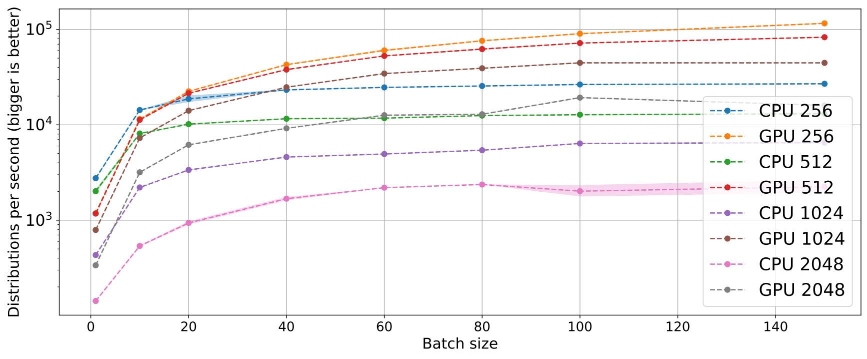
is not an option. The speedup we get for using batches of size up to 10 is considerable smaller, as shown in Figure 4.11b.

Another way to solve this issue is by simply using smaller models – models with hidden dimensionality of 256 can output upwards of 2000 distributions per second, which is on order of magnitude larger than the minimum requirement and still perform better than PPM in language modelling (Section 4.1.3). An interesting approach that could improve the performance of smaller networks is *network distillation* [Hinton et al., 2015]. This is not an approach I have explored this year because of time constraints but is a potential avenue for exploration for the second part of my project.

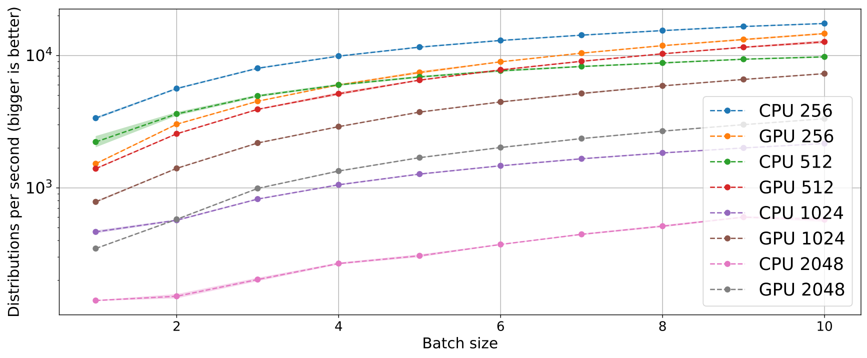
PPM prediction speeds Experiments in Section 4.1.2 show that increasing the maximal context order for PPM models leads to an increase in language modelling performance. With this in mind, it would not be a bad idea to simply increase the maximum context order for the model in the Dasher implementation (and probably tuning the smoothing parameters for PPM). In Section 4.2.1 I establish that fitting models with longer contexts in memory is feasible. One question that remains is whether these models can output probability distributions quickly enough. Noe that I have established what quickly enough means, I

In order to obtain a measure for PPM prediction speeds, I time the language modelling experiments described in Section 4.1.2. In said set of experiments, I process the validation set text character by character, polling the model for a distribution after each one. I record the time taken each chunk of 10 thousand characters for larger texts and 1 thousand characters for the Dasher training text. This gives me a point estimate of the number of distributions PPM can output per second. I report the mean and standard deviation of the estimates for all chunks in the validation set. The hardware setup for the experiments is the same as the one for the LSTM timing experiments, described a few paragraphs above. I tested models with maximal context length up to 12, as that was the highest-order model that could fit in 8 Gigabytes of memory when trained on the largest corpora I use and 12 was also the value that minimizes validation set accuracy for these corpora (Section 4.1.2).

After examining the results, I found that higher-order models operating over larger alphabets tend to be slower in comparison with smaller models but none of the exper-



(a) Large batches



(b) Smaller batches

Figure 4.11: Average number of distributions produced by LSTMs of different sizes on a CPU and a GPU and standard deviations over 100 runs for different batch sizes. The numbers in the legend are the hidden state sizes of the models. Notice the log scale on the y-axis.

iments averaged less than 11000 distributions per second (the variance in the speeds was negligible) – two orders of magnitude faster than what is required.

As a sanity check I modified the Dasher code to use an order 12 model trained on the first 90 million characters of the English version of the Europarl English-Spanish parallel corpus. There was no discernible difference in performance.

Training speeds In the current implementation of Dasher, the PPM model is trained every time the system is started. When the training set size is increased, so does the time required to train the model – training an order 12 model on the first 90 million characters of the English version of the Europarl English-Spanish parallel corpus with the hardware configuration described above takes around a minute – not a disaster in terms of start-up time for a system, but surely an inconvenience.

With LSTM models, training at start-up is out of the question – some state-of-the art models take days to train on a GPU and even training a model with the very restrictive hidden dimensionality of 256 for 10 epochs on the Europarl English corpus took more than 3 hours.

If LSTM models are incorporated in Dasher, loading pre-trained models is a must. The current Dasher implementation does not support the saving and loading of PPM models; is is advisable for this feature to be implemented if higher-order models trained on large bodies of text are to be used.

Chapter 5

Conclusions and future work

The main goal of my work was to assess the benefits and feasibility of incorporating LSTM language models into Dasher. In order to do so, I set myself three sub-goals.

The first sub-goal was providing a fair comparison, which was not present in literature, of the language modelling capabilities of LSTMs and Prediction by Partial Match (PPM) models, which are the current model choice. I compared the models on four datasets – The English- and Spanish- only versions of the Europarl parallel corpus, the text8 corpus and the training text used for the Dasher model in the latest version of the software (Details on the corpora are in Section 3.2). These experiments are described in Section 4.1

LSTM models compress the test sets of the 4 corpora to 1.15, 1.08, 1.56 and 2.00 (in the order listed above) bits per character, whereas PPM models achieved 1.46, 1.38, 1.74 and 2.36 bits per character, respectively. LSTMs compressed text to around 20% less bits per character across all datasets. Considering that the LSTM results I got on the first three corpora are arond 10% worse than state-of-the art results in literature, this supports the intial expectation that LSTM models outperform PPM.

PPM models only marginally outperformed the baseline no-gram models on all corpora but the Dasher training text. This suggests that maybe the smoothing parameters for the PPM model were overfitted to the Dasher training text. Due to time constraints, I have not attempted to tune these parameters for better performance on large datasets.

In also compared the adaptive capabilities of LSTMs and PPM by training models on the Europarl English training set and evlauating them on text generated by a different distribution – the Dasher training set. LSTM models compressed the Dasher training text to 1.99 bits per character and PPM models did not benefit from the use of training text, achieving best performance of 2.52 bits per character for compressing the Dasher training text from a cold start.

For PPM, the best results for large corpora were achieved by higher-order models (order 12), suggesting that increasing the defualt model order of 5 in Dasher and combining that with larger training text can improve performance. However, the fact that I saw no benefit in using a big training set when compressing the Dasher text suggests that this might not be the case. I have not explored the relationship between the amount of training text and adaptive performance in PPM. This is an important question and I have to address it in future. In general, the comparison of the adaptive performance of the two models I present is incomplete and needs to be cleaned up, which I leave as future work due to time constraints.

The next question I set out to answer was how much memory each model takes if incorporated into Dasher. I established that using models of order 12 and training them on large corpora will take up around 6 Gigabytes of memory if trained on the English version of the Eurpoparl corpus so simply increasing the model order and using a lot of training text is not a viable option – this can be reduced by reducing the size of the training text and the context order. The most viable option is probably tuning the smoothing parameters of the model but this is something I leave as future work. In contrast, I established that the memory footprint of LSTM models does not depend on the training set size and that the fact that the memory overhead for storing their contexts is larger, this will not impose unreasonable memory requirements on the system.

The final question I had to answer is how quickly predictions need to be made. Experimentally, I established minimal requirements for the number of distributions per second any model used in Dasher should produce, ranging from 160 to 240, depending on the number of characters in the alphabet used. I established that even higher-order PPM models easily satisfy that requirement. Large LSTM models, however, can potentially be too slow to implement in Dasher. I identify two possible solution to these problems, the exploration of which I leave as future work. the first one is using LSTM models with smaller hidden dimensionality. These still outperform PPM in language modelling but by a smaller margin than large models. The other potential solution is coming up with a smarter policy for expanding the nodes in the Dasher interface that makes use of the performance improvement gained by batching inputs for LSTMs. The two are, of course, not mutually exclusive and exploring them is a potential task for the second part of my project.

Exploring ways to obtain better performance from small LSTM models is an interesting potential avenue of research – network distillation [Hinton et al., 2015] has already been applied to various sequence modelling tasks [Kim and Rush, 2016, Prabhavalkar et al., 2016] and exploring the application to character-level language modelling is interesting.

To summarize my conclusion: LSTM models outperform PPM on language modelling tasks but I need a cleaner comparison of their capabilities to adapting to novel text. Incorporating LSTMs that compress text better than PPM in Dasher is feasible in terms of the amount of memory required but models that achieve state-of-the-art results are too slow to be naively implemented in Dasher. More work is needed in order to incorporate such models into the system.

Chapter 6

Objectives for part 2 of the project

As explained in my conclusions to this year's work, there are several branches of exploration I can take. Taking these into account, I set the following objectives for part 2 of my project:

- Improving the comparison between the language modelling capabilities of PPM and LSTM in two ways: Firstly, I intend to explore the relationship of the smoothing parameters for PPM, the amount of training text used and compression performance – this should fix any inaccuracies in the current performance that are caused by using the default smoothing parameters. Secondly, I intend to explore the adaptive capabilities of the two models in more depth by choosing more appropriate set of corpora, designing experiments to better understand the adaptive capabilities of PPM and exploring the literature for ways to improve LSTM adaptive performance.
- Exploring the implication of network distillation to LSTMs for character-level language modelling
- Exploring ways to make parallel predictions with LSTM models in the context of Dasher, perhaps designing a more appropriate data structure for using LSTM models in Dasher in order to avoid the bottleneck of prediction speed.

Bibliography

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Anderson et al., 1999] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.

[Blackford et al., 2002] Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., et al. (2002). An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151.

[Chen and Goodman, 1998] Chen, S. F. and Goodman, J. (1998). An Empirical Study of Smoothing Techniques for Language Modeling An Empirical Study of Smoothing Techniques for Language Modeling An Empirical Study of Smoothing Techniques for Language Modeling.

[Chung et al., 2016] Chung, J., Ahn, S., and Bengio, Y. (2016). Hierarchical Multiscale Recurrent Neural Networks.

[Cleary and Teahan,] Cleary, J. G. and Teahan, W. J. Unbounded Length Contexts for PPM.

[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.

[Cooijmans et al., 2016] Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., and Courville, A. (2016). Recurrent Batch Normalization.

[Hinton et al., 2015] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the Knowledge in a Neural Network.

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

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

[Kim and Rush, 2016] Kim, Y. and Rush, A. M. (2016). Sequence-level knowledge distillation. *arXiv preprint arXiv:1606.07947*.

[Koehn, 2005] Koehn, P. (2005). Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, pages 79–86.

[Krause et al., 2016] Krause, B., Lu, L., Murray, I., and Renals, S. (2016). Multiplicative LSTM for sequence modelling. pages 1–9.

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

[Melis et al., 2017] Melis, G., Dyer, C., and Blunsom, P. (2017). On the State of the Art of Evaluation in Neural Language Models.

[Moffat, 1990] Moffat, A. (1990). Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921.

[Pascanu et al., 2012] Pascanu, R., Mikolov, T., and Bengio, Y. (2012). On the difficulty of training Recurrent Neural Networks.

[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.

[Prabhavalkar et al., 2016] Prabhavalkar, R., Alsharif, O., Bruguier, A., and McGraw, L. (2016). On the compression of recurrent neural networks with an application to lvcsr acoustic modeling for embedded speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 5970–5974. IEEE.

[Sak et al., 2014] Sak, H., Senior, A., and Beaufays, F. (2014). Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*.

[Teahan and Cleary,] Teahan, W. J. and Cleary, J. G. THE ENTROPY OF ENGLISH USING PPM-BASED MODELS.

[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.

Appendix A

Language modelling framework

A.1 Example specification file

```
{
  "data_folder" : "dasher/train_GB",
  "model"       : "rnn",
  "alphabet_file": "",
  "hyperparams" : {
    "1": {
      "batches_between_stats": [640],
      "network_type"      : ["lstm"],
      "hidden_size"       : [16, 32, 64, 128, 256],
      "num_layers"        : [1],
      "batch_size"        : [10],
      "num_timesteps"     : [16],
      "reset_state_every" : [16, 128, 512, 1024],
      "optimizer"         : [
          { "name": "rmsprop", "kwargs": {} }
        ],
      "learning_rate"     : [0.1],
      "num_epochs"        : [5],
      "l2_penalty"        : [0],
      "recurrent_dropout" : [0],
      "linear_dropout"    : [0]
    }
}
```

Listing 1: Example JSON file specifying an experiment