Low Resource Language Modelling

Jared Shapiro
jar.shapiro@gmail.com
Department of Computer Science
University of Cape Town
South Africa

Abstract

In the recent two decades, there has been exponential growth in the amounts of text of various types that have become available online. This significant increase in data has allowed language models, which are core components of many natural language processing applications, to greatly increase in quality. Thus, the quality of these applications has increased dramatically as well. Despite these great advances in the field of language modelling, much of the focus has been on language modelling for languages that have large amounts of training data available on them, for language models to use. This has resulted in many low-resource languages receiving far less focus. This literature review starts by explaining the core concept of language models and their background. Following this, we look at three popular language models, starting with the most widespread and simple language model, the n-gram, which is discussing in significant detail. We then pay attention to explaining the other two models, namely the recurrent neural network model, and the more recent Transformer model. Finally, we review the current literature on language modelling for low-resource languages, taking note to contrast the different methods noted in the literature. In doing this, we comment on what we think is the way forward in the field of low-resource language modelling.

CCS Concepts

• **Computing Methodologies** → **Neural Networks**; • **Information Systems** → **Language Models**.

Keywords

Language Modelling, n-gram, Neural Network, LSTM, Tranformer, Low-Resource Language

1 Introduction

Statistical language modelling can be defined as the task of fitting a probability distribution to sequences of linguistic units [22]. These models are especially prominent in natural language processing applications, such as spelling correction, machine translation, and automatic speech recognition [7].

The performance of language models is largely dependent on the size of the training data available to that model [12]. Thus, many language models perform poorly or produce undesirable results, when the size of the training data available for use, is insufficient [12]. This has largely not been an issue for many language models, as accumulating large amounts of data has recently become much easier, due to the amount of digital content available online [22]. However, many languages lack large amounts of data [22], with these languages commonly being referred to as "low-resource languages".

The performance of language models when trained on small amounts of data has not been widely studied. However, the research done relating to this issue has been mainly based on n-gram language models and recurrent neural network models. Another recent model, named the Transformer model, has shown significant improvements over other state-of-the-art language models.

This literature review will look into the background of language modelling, as well as all three of these popular language models specifically. Since n-gram language models are the most common and simple language models, they will be looked at in significantly more detail than recurrent neural networks and Transformer models. This literature review will also look at and critique the current literature on language modelling for low-resource languages, where training data is often severely limited.

2 Language Models

2.1 Background

A language model estimates the prior probability values $P(W)$ for strings of words W in a vocabulary V. The strings that W typically represent are formal sentences or other segments, including utterances relating to speech recognition [5, 21]. Language models are used in many fields, including speech recognition, machine translation, optical character recognition, handwriting recognition, and spelling correction [4, 5].

Computing the probability of a word w, given some history h, can be defined as $P(w|h)$ [12]. For example, if the history of a word is *"they were so hungry that"* and we wanted to know the probability that the next word is *"they"*, we would compute the conditional probability as:

$$P(they|they\ were\ so\ hungry\ that) \tag{1}$$

To estimate the probability of this word, we could use a significantly large corpus, determine the frequency at which we see the history of the word followed by the word itself, and divide it by the frequency at which we see the history of the word (regardless of any potential following word) [12]:

$$P(they|they\ were\ so\ hungry\ that) = \frac{C(they\ were\ so\ hungry\ that\ they)}{C(they\ were\ so\ hungry\ that)} \quad (2)$$

The general problem that the field of natural language processing faces, is that of data sparsity, which means that there is not enough training data to model a language accurately [1]. Although this simple method of estimating the probabilities of words, does work in some cases, it can often lead to undesirable results due to possible unseen sentences in the training data. This problem is often called the curse of dimensionality, which refers to how a word sequence on which the model will be tested, is often different from word sequences seen in the training data [2]. Another reason that this method is infeasible in many cases, is that calculating the joint probability of entire sequences of words that are long in nature, can be incredibly computationally expensive and space-inefficient [12]. This necessitates using other methods of estimating the probability of a word, with various techniques being used by the most popular language models.

The joint probability of all words $P(w_1, w_2, ...w_n)$ can be decomposed as follows, using the chain rule, into the probability of each word given its preceding context [12]:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)...P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned} \quad (3)$$

Equation (3) defines a statistical language model, which is represented by the conditional probability of the next word, given the word's history.

The use of Language Models has proven extremely useful in the fields of a wide array of technological fields, including speech recognition, language translation, and information retrieval [2].

2.2 Evaluation

2.2.1 Extrinsic

The performance of a language model can be best evaluated by embedding the model in an application, and measuring the subsequent change, or lack thereof, in performance [12]. This method of evaluation refers to extrinsic evaluation, and is the only way to know for certain if particular improvements in language model components are going to improve the application using said language model. Although this is the ideal method for evaluation, it is often very expensive, as many applications used for extrinsic evaluation are fairly elaborate and slow to compute [12].

2.2.2 Intrinsic

Quicker, cheaper and the most common methods of evaluation are intrinsic evaluation methods, which measure the quality of a language model independent of applications. These methods include cross-entropy and perplexity, which are derivatives of the probability that the model assigns to test data. In addition to the reasons listed above to use intrinsic evaluation, is that it gives us a measure of how well the model fits the data we evaluate against [12]. In practice, the use of raw probabilities as a metric of performance is not used. Instead, cross-entropy and perplexity are used.

Cross-entropy is useful if the probability distribution p that generated some data, is unknown to us. The cross-entropy $H(W)$ of a model $M = P(w_i|w_{iN+1}...w_{i1})$ on a sequence of words W is defined as [3, 12] :

$$H(W) = \frac{1}{N} \log P(w_1 w_2 ... w_N) \quad (4)$$

where N is the total number of words in the data evaluated against. This value can be interpreted as the average number of bits needed to encode each word in the test data, using the compression algorithm associated with the specific language model [3].

The perplexity $PP(W)$ of a model can be defined as the exp of the model's cross-entropy value [3]:

$$\begin{aligned} PP(W) &= 2^{H(W)} \\ &= P(w_1 w_2 ... w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} \\ &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 ... w_{i-1})}} \end{aligned} \quad (5)$$

The lower the cross-entropy and perplexity of a model is, the better the model is said to perform [3].

3 N-grams

3.1 Background

The most widely used by far and one of the simplest language models are n-gram language models [5, 21]. These models perform well across a variety of language modelling tasks, and are easily trainable, as they do not require annotated training data [6].

In an n-gram model, we make the approximation that the probability of a word depends only on the previous $(n-1)$ words. This reduces the dimensionality of the estimation problem [21] and it relates to the idea of Markov models, which assume that we can predict the probability of some future unit without looking too far into the past. In our case, this applied to future words and their history [3, 12]. This general approximation for n-grams to the conditional probability to the next word in a sequence is shown as follows:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}) \tag{6}$$

For example, trigram models, which are the most common n-gram model [5], compute the probability of a word by making use of the two words that preclude it: $P(w_n|w_{n-2}^{n-1})$. This probability can be approximated using the maximum likelihood estimate (MLE). The easiest way to calculate the MLE is to count the number of times the word sequence $(w_{n-2}w_{n-1}w_n)$ appears in some training corpus, and divide this frequency by the number of times the word sequence $(w_{n-2}w_{n-1})$ appears [3, 12]. This equation is as follows:

$$P(w_n|w_{n-2}^{n-1}) = \frac{C(w_{n-2}w_{n-1}w_n)}{C(w_{n-2}w_{n-1})} \tag{7}$$

With the general case of MLE n-gram parameter estimation as follows:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})} \tag{8}$$

Using the MLE can lead to undesirable results for many applications that use language models [3]. This is due to the potentially insufficient data and inherent data sparseness in statistical modelling [3]. This can lead to the assignment of zero probability given to sequences not seen in the training data, which would lead to inaccurate or undefined probability estimates [3]. This is because an event that has never been observed in the training data could still occur in the test data. Even in the extreme case, where all parameters of a model can be trained accurately, due to a significantly large amount of training data, data sparsity becomes an issue if one expands the model [3]. Smoothing techniques, which normally produce probabilities very close to the MLE [3], can be used to address the issue of data sparsity by modifying the MLE to produce more accurate results. The probability distributions are modified to be more uniform by decreasing significantly high probabilities and increasing significantly low probabilities, such as sequences that have an assigned probability of zero [3].

3.2 Smoothing

No matter how much data is available for use, smoothing techniques can almost always improve the performance of language models without much effort [3].

3.2.1 Additive Smoothing

One of the first and simplest smoothing techniques used in practice is additive smoothing [10, 16**?** , 17], which tackles the issue of data sparsity by pretending each n-gram occurs a certain number of times, , more than it does:

$$P_{add}(w_i|w_{i-n+1}^{i-1}) = \frac{\delta + C(w_{i-n+1}^{i})}{\delta|V| + \sum_{w_i} C(w_{i-n+1}^{i})} \tag{9}$$

Typically $0 < \delta \leq 1$ [3], however [16] and [10] advocate for $\delta = 1$.

3.2.2 Good-Turing Estimate

The Good-Turing estimate [8] forms the basis for many smoothing techniques, such as Katz smoothing [3]. This method dictates that for any n-gram that occurs r times, we should pretend that it occurs r^* times where n_r is the number of n-grams seen exactly r times in the training data [8]. This is as follows:

$$r^* = (r+1)\frac{n_r + 1}{n_r} \tag{10}$$

In order to convert this count into a probability, we normalize this value by:

$$P_{GT}(w_{i-n+1}^{i}) = \frac{r^*}{N} \tag{11}$$

The Good-Turing estimate is not used, in practice, by itself for n-gram smoothing. This is due to the estimate not including the combination of higher-order models with lower-order models, which is necessary for good performance [3].

3.2.3 Interpolated vs Backoff Smoothing Models

Interpolation techniques mix the probability estimates from all the n-gram estimators, weighing and combining trigram, bigram and unigram counts [12]. They therefore discount some probability mass from high probability seen words and reassign this mass to low probability and unseen words. For example, in simple linear interpolation, to estimate the trigram probability $P(w_n|w_{n2}w_{n1})$, we mix together the unigram, bigram and trigram probabilities according to weights λ_i [12]:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) =& \lambda_1 P(w_n|w_{n-2}w_{n-1}) \\ &+ \lambda_2 P(w_n|w_{n-1}) \\ &+ \lambda_3 P(w_n)\end{aligned} \tag{12}$$

The result is that trigrams that consist of higher probability bigrams and unigrams will be assigned higher probabilities than trigrams that consist of lower probability bigrams and unigrams.

Backoff models take a different approach. This approach is that we use higher-order n-grams if there is enough evidence for it, otherwise a lower-order n-gram is used [3]. This means that we "back off" to lower n-grams if we lack evidence for higher-order n-grams, and we continue to "back off" until we reach a specific order of n-gram history that has some counts. For backoff models to give a correct probability distribution, they must discount some probability mass from higher-order n-grams to assign to lower-order n-grams. If this is not done, and we use the undiscounted MLE probability, replacing a higher-order n-gram with a lower-order n-gram will result in the total probability of all strings in the

language model being greater than 1 [12]. Backoff with discounting is used in Katz backoff, which will be discussed later.

3.2.4 Jelinek-Mercer Smoothing

Consider the simple smoothing technique of additive smoothing, where we may pretend that each bigram occurs once more than it does. Now consider that where we train a language model where we have not seen the bigram *"fly the"* nor *"fly thou"*. If we apply additive smoothing to this model, the same probability will be assigned to both sequences of words. Intuitively this seems incorrect, as the word *"thou"* after a verb seems much less common than the word *"the"* after a verb. To address this issue, a smoothing technique named Jelinek-Mercer [11] can be used.

This interpolation technique makes use of the fact that when there is insufficient data for directly estimating an n-gram probability, useful information can be provided by the corresponding (n-1)-gram. In this way, lower-order n-grams, which have the advantage of being estimated from more data, can provide more information for the higher-order n-grams [3]. [3] denotes an elegant way of performing this interpolation:

$$\begin{aligned} P_{interp}(w_i|w_{i-n+1}^{i-1}) =& \lambda_{w_{i-n+1}^{i-1}} PML(w_i|w_{i-n+1}^{i-1}) \\ &+ (1-\lambda_{w_{i-n+1}^{i-1}}) P_{interp}(w_i|w_{i-n+2}^{i-1}) \end{aligned} \tag{13}$$

TALK ABOUT THIS

3.2.5 Katz Smoothing

Katz smoothing [14] is a non-linear method that is like Good-Turing estimates in that it computes adjusted counts, however, it extends the intuitions of the estimates by adding the combination of higher-order models with lower-order models. The probability of an n-gram with zero counts is approximated by backing off to the (n-1)-gram until an n-gram with a non-zero count history is reached. The model can be expressed as:

$$P_{Katz}(w_i|w_{i-N+1}^{i-1}) = \begin{cases} P^*(w_i|w_{i-N+1}^{i-1}) & \text{if } C(w_{i-N+1}^{i}) > 0 \\ \alpha(w_{i-N+1}^{i-1}) P_{Katz}(w_i|w_{i-N+2}^{i-1}) & \text{otherwise.} \end{cases} \tag{14}$$

Where P* is the Good-Turing discounted probability and $\alpha(w_{i-N+1}^{i-1})$ is the normalizing constant, which determines how some probability masses seen at higher-order n-grams are redistributed to unseen n-grams, according to the lower-order distribution.

This model calculates the conditional probability of a word proportional to the MLE of the n-gram, given that specific n-gram has a frequency higher than k (Which is 0 in the above equation), which [14] suggests being equal to 5. Otherwise, the conditional probability is calculated as the back-off probability of the (n-1)-gram. Since the Katz model being defined recursively in terms of the Katz (n-1)-gram model, the end of recursion happens at the Katz unigram model, which is taken to be the maximum likelihood unigram model [3].

3.2.6 Witten-Bell Smoothing

The Witten-Bell smoothing model [26] can be considered an instance of the Jelinek-Mercer smoothing model and was initially developed for use in text compression [3]. The model is defined recursively as a linear interpolation between the nth-order maximum likelihood model and the (n-1)th-order smoothed model and can be defined as follows:

$$P_{WB}(w_i|w_{i-n+1}^{i-1}) = \frac{C(w_{i-n+1}^{i}) + N_{1+}(w_{i-n+1}^{i-1}\bullet) P_{WB}(w_i|w_{i-n+2}^{i-1})}{\sum_{w_i} C(w_{i-n+1}^{i}) + N_{1+}(w_{i-n+1}^{i-1}\bullet)} \tag{15}$$

where N_{1+} represents the number of words that have one or more counts, and the $\bullet$ represents a free variable that is summed over.

3.2.7 Absolute Discounting

Absolute discounting [19], is similar to Jelinek-Mercer smoothing, in that it involves the interpolation of higher-order models and lower-order models. However, in contrast to Jelinek-Mercer smoothing, Absolute discounting does not multiply the higher-order maximum-likelihood distribution by $\lambda_{w_{i-n+1}^{i-1}}$, rather creating the higher-order distribution by subtracting a fixed discount $D \leq 1$ from each n-gram, of some order, seen in the training data. This equation is as follows [3]:

$$P_{abs}(w_i|w_{i-n+1}^{i-1}) = \frac{max[C(w_{i-n+1}^{i}) - D, 0]}{\sum_{w_i} C(w_{i-n+1}^{i})} + (1-\lambda_{w_{i-n+1}^{i-1}}) P_{abs}(w_i|w_{i-n+2}^{i-1}) \tag{16}$$

3.2.8 Kneser-Ney

In the case of n-gram models, we have discussed how the (n-1)-gram model can be used for backing-off when there are cases of unseen events. However, [15] claims that when the normal probability distribution of lower-order models is taken for backing-off, it can lead to undesirable results. [15] illustrates this issue by pointing out that the word *"dollars"* is very unlikely to occur after a word that is not a number or the name of a country. However, if the unigram probability $P(Dollars)$ is taken for backing-off, the smoothed model will assign a much higher probability to the word than is appropriate. [15] states that this suggests backing-off distributions should be different from the normal probability distribution. The model has the following from:

$$P_{KN}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \frac{max[C(w_{i-n+1}^{i})-D,0]}{\sum_{w_i} C(w_{i-n+1}^{i})} & \text{if } C(w_{i-n+1}^{i}) > 0 \\ \gamma(w_{i-n+1}^{i-1}) P_{KN}(w_i|w_{i-n+2}^{i-1}), & \text{if } C(w_{i-n+1}^{i}) = 0 \end{cases} \tag{17}$$

where $\gamma(w_{i-n+1}^{i-1})$ is chosen to make the distribution sum to 1.

3.2.9 Modified Kneser-Ney

Modified Kneser-Ney [3] smoothed models achieve the best performance ,within the context of n-gram models, when compared to all other smoothing techniques. It also performs substantially better than regular Kneser-Ney smoothing [3].

Instead of using a single discount D for all nonzero counts as in Kneser-Ney smoothing, three or more parameters D_1, D_2, D_{3+} are applied to n-grams that have been seen once, twice, and three or more times, respectively, in the training data.

3.2.10 Stupid-Backoff

Stupid Backoff is a smoothing technique that is mainly used for significantly large datasets, and differs from the above techniques in that it does not generate normalized probabilities and does not make use of applying any discounting, rather using relative frequencies [7]:

$$S(w_i|w_{i-n+1}^{i}) = \begin{cases} \frac{C(w_{i-n+1}^{i})}{C(w_{i-n+1}^{i-1})} & \text{if } C(w_{i-n+1}^{i}) > 0 \\ \alpha S(w_i|w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases} \tag{18}$$

Where is the backoff factor and may be made to depend on n.

In distributed environments, [7] claims that Stupid Backoff is inexpensive to calculate and approaches the quality of Kneser-Ney smoothing for large amounts of data.

3.3 Feed-Forward Neural Network Language Model

Modern neural networks are formed from singular compute units, which process a vector of input values and produce a single output value. Feedforward Neural Networks (FFNNs) were the first type of neural networks introduced to the field of language modelling [23], and adopt the paradigm of supervised learning, which means that a "teacher" provides output targets for each input pattern, followed by explicitly correcting the network's errors [24]. As the name of this architecture suggests, FFNNs estimate probabilities of words and word sequences via the use of a neural network, in contrast to traditional count-based n-gram models that use smoothing methods to estimate probabilities. Like count-based models, FFNNs are based on the Markov assumption [23], such that only the most recent (n-1) words of a word's history are used to calculate the probability of the current word w_i.

Perceptrons are a simple type of neural network comprised of binary threshold units that can be used to compute certain functions [24]. However, single layer perceptrons can only find a set of weights to correctly classify patterns in training sets if these patterns are linearly separable, meaning that there is no way to draw a line between two distinct classes of a pattern [24]. A specific example of this failing of single layer perceptrons is the inability to calculate the XOR function. Multi-layer perceptrons are more complex to train than single-layer perceptions, however, they can theoretically compute any function [24]. FFNNs follow similar architecture, however, differ from multi-layer perceptrons in that perceptrons are purely linear, while modern FFNNs consist of units that have non-linear activation functions [13].

The simplest type of FFNNs have three types of layers; the input layer, hidden layer, and output layer which consist of input, hidden, and output units respectively. There is always one input layer and output layer, however, there can be multiple hidden layers [13]. The input layer is often not counted when enumerating layers. Each layer is fully-connected, meaning that each unit of a layer processes as input all the separate outputs of the units in the previous layer. Similarly, a layer that is followed by another, has units that send their output to all separate units in the following layer. Therefore, there is a connection between all pairs of units between two adjacent layers [13]. Each neural unit in FFNNs takes a weighted sum of its inputs, adds a bias term, and applies a non-linear function to it, whose output is called the activation value. This can be represented using vector notation as follows[13]:

$$y = a = f(z) = f(w + b) \tag{19}$$

where w is the weight vector, x is the input vector, b is the bias scalar, a is the activation value, z is the intermediate output of the node, and y is the final output of the network, which in this case, consists of a single unit.

The three most popular non-linear functions used by FFNNs to compute activation values are the sigmoid, tahn, and rectified linear unit (ReLU) [13]. The sigmoid function can be defined as follows:

$$a = \sigma(z) = \frac{1}{1 + \exp^{-z}} \tag{20}$$

Tahn function:

$$a = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{21}$$

ReLU function:

$$a = max(x, 0) \tag{22}$$

In practice, the sigmoid function is rarely used and ReLU is the most common [13]. The benefit of using ReLU over the other two mentioned functions is that the result of the ReLU function is close to linear. Additionally, an issue with using sigmoid or tanh functions is that very high intermediate outputs lead to a final output value being extremely close to 1. [13].

A way for FFNNs to be able to perform some hidden layer computation more efficiently is to use a single matrix W for the weights of some hidden layer so that the computation can be done with simple matrix operations [13]. Since each hidden unit in some hidden layer has a weight vector w, and a bias scalar b, we can represent these parameters for some hidden layer by combined these parameters for each hidden unit i, with weight w_i and bias b_i, into a single weight matrix W and a single bias vector b. This weight matrix now represents all the parameters for some entire hidden layer, with each element W_{ij} of W representing the weight applied to the ith input unit x_i to the jth hidden unit h_j. The computation of the hidden layer can now be done in three steps: taking the input vector, x, multiplying it by the weight matrix W, adding the bias vector b, and applying the activation function to the result. The resulting value, h, can be "fed forward" to later hidden layers or the output layer as their input. The output of the hidden layer can now be defined as follows [13]:

$$h = \sigma(Wx + b) \tag{23}$$

Like the hidden layer, the output layer has a weight matrix, U, that is used in combination with its inputs (outputs from the last hidden layer) and bias scalar. While the output of hidden layers can be a real-valued number, the result of the output layer is used to make classification decisions, and so this final output must be focused on the case of classification [13]. A function that can be used as the activation function for the output layer, that converts a vector of real-valued numbers into a vector that encodes a probability distribution is the softmax function:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} 1 \leq i \leq d \tag{24}$$

where z is the vector of the intermediate output of the output layer that has dimensionality d.

A two-layer FFNN that uses ReLU as the activation function for all intermediate outputs, except the final layer intermediate output, which uses the softmax function, can be represented as follows:

$$\begin{aligned} z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= ReLU(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= softmax(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{25}$$

where the superscripts in square brackets refer to the layer number.

FFNNs can take a sliding window approach, where, at each time step t, the network can take n words (which form the window), and produce the next word in the sequence [13].

The main advantages that neural language models such as FFNN models have over traditional n-gram language models are that they can handle much larger word histories, don't need smoothing, and can generalize to unseen data better through the use of embeddings, which represent the prior context of a word [13].

Although, like traditional n-gram models, FFNNs usage can be problematic in that it limits the context from which information can be extracted [13]. In a sliding window approach of FFNNs, words outside the context window have no impact on current decisions, meaning that possible important information is lost. This is exemplified in language tasks that require information from words at that are far away, in terms of distance in number of words, from the current point in the process. Another problem is that FFNNs have difficulty learning systematic patterns in languages [13]. Recurrent Neural Networks (RNNs) solve these issues by directly dealing with the temporal aspect of languages.

4 LSTM Neural Networks

4.1 Background

Long short-term memory networks (LSTMs) are a form of recurrent neural networks (RNNs), which are neural sequence models that perform significantly well when compared to other models, and serve as the basis for tasks such as machine translation, language modeling, and question answering, among other sequence learning tasks [18, 28]

An RNN can be classified as some network that consists of units, which have values that are directly or indirectly dependent on earlier outputs as an input. Like FFNNs, an RNN works by multiplying some input vector, which represents the current input element, by a weight matrix, applying an activation function, and passing the activation value to the hidden layer. The hidden layer then calculates its output. However, in contrast to FFNNs, RNNs have a recurrent link between hidden layers, allowing the computation at the hidden layer to use the activation value of the hidden layer from the preceding point in time [9]. In this way, hidden layers from previous time steps provide context that can be used in future computations. The way an RNN makes use of past context in future computations

is by using a set of weights, U, that connect a hidden layer from a previous timestep to a current layer [9]. As discussed previously, this is how RNNs deal with the temporal nature of languages

Although the hidden layers of previous timesteps provide context for future hidden layers, it is difficult to train RNNs for such tasks that make use of history that is significantly distant from the current point of processing [13]. This is due to the information in hidden states being mainly influenced by recent hidden layers. Thus, much of the distant information is lost. These failings of RNNs to substantially retain critical information can be explained by two main reasons. The first reason is the requirement of the hidden layer weights, U, to be used to provide information about the current decision, as well as retain information required for future decisions [13]. The second reason relates to the vanishing gradients problem, meaning the gradients eventually reduce to zero, which results from the need to backpropagate the error signal back through time[13].

LSTM networks attempt to solve these problems by adding an additional context layer to the original RNN. This context layer, the "long-term" memory, consists of a vector of memory cells, which are used to store information for long periods. With this new context layer, at each time step, the LSTM can overwrite, retrieve, or retain memory cells for the next time step [28].

Formally an RNN can be described as follows: [9]

$$\begin{aligned} a^t &= b + Wh^{t-1} + Ux^t \\ h^t &= \tanh(a^t) \\ o^t &= c + Vh^t \\ \hat{y}^t &= softmax(o^t) \end{aligned} \tag{26}$$

and an LSTM defined as follows: [27]

$$LSTM : R^m \times R^n \times R^n \rightarrow R^n \times R^n$$

$$LSTM(x, c_{prev}, h_{prev} = (c, h)$$

The updated state c and the output h can then be computed as follows:

$$\begin{aligned} f &= \sigma(W^{fx}x + W^{fh}h_{prev} + b^f) \\ i &= \sigma(W^{ix}x + W^{ih}h_{prev} + b^i \\ j &= \tanh(W^{jx}x + W^{jh}h_{prev} + b^j \\ o &= \sigma(W^{ox}x + W^{oh}h_{prev} + b^o) \\ c &= f \odot c_{prev} + i \odot j \\ h &= o \odot \tanh(c) \end{aligned} \tag{27}$$

where σ is the logistic sigmoid function, $\odot$ is the elementwise product, W^* and b are weight matrices and biases

5 Transformers

5.1 Background

Despite the continued iteration and progress relating to RNNs and LSTMs, both suffer from the fundamental constraint of sequential computation. This constraint prevents these models from parallelization within training data, which leads to poor performance at significantly long sequence lengths, due to memory constraints limiting batching across data [25].

A fairly recent model architecture named the Transformer, proposed by [25], avoids the use of recurrence in favor of relying entirely on attention mechanisms to draw global dependencies between input and output, being the first transduction model to do so. This model architecture outperforms RNNs and LSTMs relating to long sequence lengths, as it allows for a significant amount of parallelization [25].

The architecture of the transformer is based on state-of-the-art sequence transduction models, which have an encoder-decoder structure. In this structure, the encoder maps an input sequence of symbol representations to a sequence of continuous representations. The decoder then processes the latter sequence and generates, one symbol at a time, an output sequence [25].

The Transformer has point-wise, fully connected layers for both encoder and decoder, with all layers of each making up a stack. Each layer in both encoder and decoder is identical, however, the layers of the encoder differ from the layers of the decoder. The encoder has six layers, which consists of two sub-layers. The first sub-layer is a multi-head self-attention mechanism, with the second sub-layer being an FFNN. Both of the sub-layers employ a residual connection around them, and the layer itself is normalized. The decoder consists of six layers that have the same two sub-layers of the encoder, however, a third sub-layer is added that performs multi-head attention over the output of the encoder stack. As with the layers of the encoder stack, residual connections are employed around each sub-layer, followed by layer normalization. The decoder stack's sub-layers are also modified to prevent positions from attending to subsequent positions, which, coupled with output embeddings being offset by one position, ensures sole dependency of the predictions for position i, on the known output positions less than i [25].

The concept of attention can be described as a function that maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are vectors. The output of an attention function is a weighted sum of values, where each value is computed by a compatibility function of the query with the corresponding key [25]. The attention function that is used by the Transformer is called Scaled Dot-Product Attention. This function's input consists of queries and keys of dimension d_k, and values of dimension d_v.

The weights on the values are calculated by computing the dot products of the query with all keys, dividing each by $\sqrt{d_k}$, and applying a softmax function to the result. The output is then computed as the weighted sum of the values [25].

The use of Multi-Head Attention in the Transformer model refers to performing the attention function in parallel on multiple linear projections of queries, keys, and values, resulting in d_v-dimensional output values. This form of attention allows the Transformer model to jointly attend to information from different representation subspaces at different positions [25].

[20] introduced a model based on the original Transformer model, commonly called GPT, that involves generative pre-training of a language model on a diverse corpus of unlabeled text, and then performs discriminative fine-tuning on each specific task. [20] found that this approach yields large performance gains over discriminatively trained models, that are designed for one specific task, in tasks such as textual entailment, question answering, semantic similarly assessment, and document classification.

6 Application for Low-Resource Languages

Recently, neural network language models have been successful across various fields of computational linguistics, however, the languages modelled on have been mainly ones that have significantly large data sets available for the training of these models. Low-resource languages have received significantly less attention, and have not currently benefited from these advances in computational linguistics [22].

[7] Compared the performance of an n-gram language model that uses modified Kneser-Ney smoothing, an FFNN, and an RNN. [7] evaluated the performance of these models across the languages of Cantonese, Pashto, Tagalog, Turkish, and Vietnamese. [7] found that both neural network models performed well in terms of perplexity and word error rate, further noting that the relative improvement against the n-gram model increased with the size of the training data used. [7] also noted that FFNNs perform better than RNNs for low-resource languages but perform worse for larger data sizes. To improve the performance of FFNNs for larger training data sets relative to RNNs, FFNNs can be interpolated with modified Kneser Ney models. In this case, the interpolated model performs better than RNNs for all training data sizes. Despite the general improvements in the performance of neural network models over state-of-the-art n-gram models, the n-gram model improved significantly better when modelled on Turkish data, which [7] notes has a rich morphology and low amounts of training data. This could suggest that n-gram models may still have the edge in these circumstances.

[22] conducted a rudimentary analysis of RNNs and compared the performance of this model to a state-of-the-art n-gram language model, which used modified Kneser-Ney smoothing. This was done within the context of low-resource South African languages, namely Xitsonga, IsiZulu ,and Afrikaans, as well as English, which is not a low-resource language. [22] found that n-grams performed better than basic RNNs, however linearly interpolating the two models outperformed both of the component models. This finding that linear interpolation of an n-gram model with a neural network model can outperform the performance of the singular models agrees with the findings of [7] discussed previously.

7 Discussion

We have presented in our review different research focusing on how the language models discussed in this review perform when being trained on low-resource languages, thus having low amounts of training data available to them. The amount of research that has been done on this matter is extremely sparse, however, there have been notable results achieved.

We have found that although there have been great strides in the field of language modelling, much of that success has not transitioned into the field of low-resource language modelling specifically.

We have seen that a state-of-the-art n-gram model using Modified Kneser-Ney [3] smoothing performs similar to or outperforms neural networks when training data is severely limited, which is often the case when modelling on low-resource languages.

We have also seen that linear interpolation of a neural network and an n-gram model can significantly improve the performance of language models when they are trained on limited data, to the point at which the linearly interpolated model outperforms all individual component models.

Although Transformer models have been shown to perform better than both state-of-the-art n-gram and RNN models, there has not been any research relating to how Transformer models compare to these models when training data is severely limited.

8 Conclusions

In this literature review, we have examined what language models are and have looked at three classes of popular language models, namely n-gram models, recurrent neural networks (as well as LSTMs), and the recent Transformer model. We have also discussed how state-of-the-art n-gram models and RNNs perform in limited training data settings.

We have identified that there has been a lack of research relating to language modelling for low-resource languages, as well as how there specifically has been no research into the comparison of n-gram models, RNNs and Transformer models when all three are given a severely limited amount of training data, as is often the case when working with low-resource languages.

We feel the next step is to look at how n-gram models, LSTMs, and Transformer models perform when given limited amounts of training data, subsequently comparing all three models. We feel

as this could be done with Nguni languages, which have very low amounts of training data relating to them.

References

[1] Allison, B., Guthrie, D., and Guthrie, L. Another look at the data sparsity problem. In *International Conference on Text, Speech and Dialogue* (2006), Springer, pp. 327–334.

[2] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. A neural probabilistic language model. *Journal of machine learning research 3*, Feb (2003), 1137–1155.

[3] Chen, S. F., and Goodman, J. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language 13*, 4 (1999), 359–394.

[4] Church, K. W. A stochastic parts program and noun phrase parser for unrestricted text. In *International Conference on Acoustics, Speech, and Signal Processing,* (1989), IEEE, pp. 695–698.

[5] Clark, A., Fox, C., and Lappin, S. *The handbook of computational linguistics and natural language processing.* John Wiley & Sons, 2013.

[6] Clark, A., Giorgolo, G., and Lappin, S. Statistical representation of grammaticality judgements: the limits of n-gram models. In *Proceedings of the fourth annual workshop on cognitive modeling and computational linguistics (CMCL)* (2013), pp. 28–36.

[7] Gandhe, A., Metze, F., and Lane, I. Neural network language models for low resource languages. In *Fifteenth Annual Conference of the International Speech Communication Association* (2014).

[8] Good, I. J. The population frequencies of species and the estimation of population parameters. *Biometrika 40*, 3-4 (1953), 237–264.

[9] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[10] Jeffreys, H. *The theory of probability.* OUP Oxford, 1998.

[11] Jelinek, F. Interpolated estimation of markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice, 1980* (1980).

[12] Jurafsky, D. *Speech & language processing.* Pearson Education India, 2000.

[13] Jurafsky, D., and H. Martin, J. *Speech and Language Processing*, 2019 (accessed May 12, 2020). http://https://web.stanford.edu/~jurafsky/slp3/.

[14] Katz, S. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing 35*, 3 (1987), 400–401.

[15] Kneser, R., and Ney, H. Improved backing-off for m-gram language modeling. In *1995 International Conference on Acoustics, Speech, and Signal Processing* (1995), vol. 1, IEEE, pp. 181–184.

[16] Lindstone, G. Note on the general case of the bayes-laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries.*

[17] marquis de Laplace, P. S. *Essai philosophique sur les probabilités.* Bachelier, 1825.

[18] Merity, S., Keskar, N. S., and Socher, R. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182* (2017).

[19] Ney, H., and Essen, U. On smoothing techniques for bigram-based natural language modelling. In *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing* (1991), IEEE, pp. 825–828.

[20] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf* (2018).

[21] Rosenfeld, R. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE 88*, 8 (2000), 1270–1278.

[22] Scarcella, A. *Recurrent neural network language models in the context of under-resourced South African languages.* PhD thesis, University of Cape Town, 2018.

[23] Sundermeyer, M., Ney, H., and Schlüter, R. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing 23*, 3 (2015), 517–529.

[24] Tebelskis, J. *Speech recognition using neural networks.* PhD thesis, Carnegie Mellon University, 1995.

[25] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems* (2017), pp. 5998–6008.

[26] Witten, I. H., and Bell, T. C. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Ieee transactions on information theory 37*, 4 (1991), 1085–1094.

[27] Yogatama, D., Miao, Y., Melis, G., Ling, W., Kuncoro, A., Dyer, C., and Blunsom, P. Memory architectures in recurrent neural network language models. *International Conference on Learning Representations* (2018).

[28] Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).