A survey of Neural Network-Based Approaches for Language Modeling.

Kushal Arora
Supervisors: Doina Precup & Jackie Chi Kit Cheung

1 Introduction.

One of the core problems that we need to solve in the quest for Artificial General Intelligence (AGI) is language understanding. The language understanding problem can be loosely defined as endowing a machine with an ability to read some text and comprehend its meaning. This ability is essential for applications such as automated reasoning, information retrieval, machine translation, question answering, natural language inference.

At the heart of language understanding lies the problem of language modeling. In its broadest definition, language modeling can be seen as building a probability distribution over a sequence of text. This distribution will ideally assign a higher probability to a sentence that is syntactically correct and semantically agrees with the world model assumptions.

A tractable formulation of this definition is to model language as a linear chain on words and treat the language modeling problem as predicting the next word given the previous words. The traditional approaches toward building this probability distribution have been statistical in nature, relying on counts of word sequences computed from a corpus. This formulation has had empirical success in a range of language understanding tasks such as automatic speech recognition, machine translation, and information retrieval.

Neural network based language models attempt to reformulate language modeling as a continuous optimization problem, by embedding the individual words in a continuous vector space and using a parameterized function to build a probability distribution over these words. These neural models have not only proven to be more powerful than classical models in building this conditional probability, but also provided an additional advantage of being able to build a latent space language representation. These representations have been able to capture contextual semantic and syntactic regularities, and have led to a paradigm shift in approaches towards language understanding tasks such as inference, question answering and machine translation.

In this report, we will focus on various neural language models and analyze how they improve upon the classical models. We will also discuss the representational perspective of neural models in terms of word embeddings and its applications to language understanding tasks such as machine translation and language inference. We will start by discussing the classical statistical formulation of language modeling, its shortcomings and attempts to address these issues in the count-based formulation in Section 2. In Section 3, we will introduce neural language models and discuss how they address the statistical model's shortcomings. We will also analyze the computation and optimization issues with neural language models and proposed solutions to deal with these problems. In Section 4, we will look at the evaluation metrics for language models. Finally, in Section 5, we will explore the applications of language models ranging from re-ranking for automatic speech recognition and machine translation to pre-training for contextualized word embeddings.

2 Statistical N-gram Models

The language modeling problem is defined as building a probability distribution P on sequences of n words $w_1..w_n$. Usually, language is modeled as a linear chain on words. In this paradigm, language modeling can be seen as predicting next word given the history or the context, and the probability of the sequence $P(w_1..w_n)$ is factorized without loss of generality as:

$$P(w_1...w_n) = P(w_1)\prod_{i=2}^{n} P(w_i|w_1..w_{i-1})$$

For example, consider the sentence: *My dog likes eating cake.* The probability of the sentence will be modeled as

$$\begin{aligned} P(My\ dog\ like\ eating\ cake) \quad =& P(My) \times P(dog|My) \times P(likes|My\ dog) \times \\ & P(eating|My\ dog\ likes) \times P(cake|My\ dog\ like\ eating) \end{aligned}$$

In statistical models, the conditional probabilities are derived from count based statistics computed on the training corpus. Let $C(x_1..x_i)$ denotes the number of times the sequence $x_1..x_i$ occur in the training corpus. The probability $P(w_i|w_1..w_{i-1})$ can be estimated by dividing the frequency of occurrence of the sequence $w_1..w_i$ by the frequency of occurrence of the sequence $w_1..w_{i-1}$.

$$P(w_i|w_1..w_{i-1}) = \frac{P(w_1..w_i)}{\sum_{w\in V} P(w_1..w_{i-1}w)} = \frac{C(w_1..w_i)}{\sum_{w\in V} C(w_1..w_{i-1}w)} = \frac{C(w_1..w_i)}{C(w_1..w_{i-1})}$$

For example, consider the conditional probability $P(likes|My\ dog)$. If the text sequence *My dog likes* and *My dog* occur $C(My\ dog\ likes)$ and $C(My\ dog)$ times in the training corpus, the conditional probability $P(likes|My\ dog)$ is given by

$$P(likes|My\ dog) = \frac{C(My\ dog\ likes)}{C(My\ dog)}$$

One of the problems with the linear chain model is that the number of parameters of the model grows exponentially with the sequence length. Assuming the vocabulary size is $|V|$ and maximum length of the sentence in corpus is N, the space complexity for storing the distribution will be $O(|V|^N)$. This parameter space explosion problem is tackled by imposing an order-n Markovian assumption on the conditional probability distribution i.e. probability of next word will only depend upon the last $n-1$ words:

$$P(w_i|w_1..w_{i-1}) = P(w_i|w_{i-n}..w_{i-1})$$

This order-n Markov chain on words is the standard *n-gram* model [1]. This approximation reduces the space complexity of the model to $O(|V|^n)$. Most commonly used n-grams are bigrams (order 2) and trigrams (order 3) as the computational and space complexity becomes a bottleneck for order 4 or higher.

2.1 Smoothing

The standard n-gram model suffers from the *curse of dimensionality* i.e., many of the n-grams encountered at test time would not have been seen during training, and the conditional probabilities corresponding to those n-grams would be zero. This can be viewed as over-estimation of the count statistics of n-grams seen in the training corpus and under-estimation (with a value of zero) of the missing n-grams. The n-gram models deal with this sparsity problem by relying on *smoothing techniques* which redistribute the probability mass away from n-grams seen in the training data to the plausible n-grams not present in the training corpus.

An intuitive and simple smoothing technique is *additive smoothing.* In additive smoothing, the count for each n-gram seen in the training corpus is incremented by δ. If $\delta = 1$, this is called *add-one smoothing* or *Laplace smoothing.* An alternate view of additive smoothing is to view it as a prior distribution and the distribution computed from the training corpus as a posterior. In practice, additive smoothing performs rather poorly.

More practical smoothing approaches that work well empirically are based on *Good-Turing* discounting. The Good-Turing discounting reallocates the probability mass of the n-grams that occur $r+1$ times in the training corpus to the n-grams that occur r times. This re-weighting scheme treats the missing n-grams as if they occurred exactly once in the corpus. Let n_{r+1} and

n_r be the number of n-grams that occur exactly $r+1$ and r time respectively. The re-weighted counts for an n-gram that occurred r times will then be

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

Now, the probability of the sequence $x_1...x_i$ that occurred r times is given by

$$p(x_1..x_i; C(x_1..x_i) = r) = \frac{r^*}{N}$$

where $N = \sum_{r=0}^{\infty} n_r r^* = \sum_{r=0}^{\infty} n_{r+1}(r+1) = \sum_{r=0}^{\infty} n_r r$.

Good-Turing discounting is not directly used as the estimates for higher r are often noisy. For example, the revised probability for a sequence $x_1..x_i$ occurring r times in the corpus might be 0 if no sequence occurs $r+1$ times i.e. $n_{r+1} = 0$.

Katz smoothing [2] extends the idea of Good-Turing discounting by using a well-behaved approximation as well as relying on the idea of backing-off to a lower-order model for unseen n-grams. The adjusted counts for a bigram Katz smoothing can be computed as:

$$C_{katz}(w_{i-1}w_i) = \begin{cases} d_r r & r > 0 \\ \alpha(w_{i-1})P(w_i) & r = 0 \end{cases}$$

where r is the number of times bigram $w_i w_{i-1}$ occurs in the corpus, d_r is an approximation of the Good-Turing discounting ratio such that, for a given hyper-parameter k, $d_r = 1$ for $r > k$ or else the Katz count estimate falls-back to the approximate Good-Turing estimate. $\alpha(w_{i-1})$ is a normalizing constant that ensures that the normalized Katz counts match the corpus count for the prefix w_i i.e. $\sum_{w_i} C_{katz}(w_{i-1}w_i) = \sum_{w_i} C(w_{i-1}w_i)$. The value of $\alpha(w_{i-1})$ and d_r for $r \leq k$ can be computed as follows:

$$d_r = \frac{\frac{r^*}{r} - \frac{(k+1)n_{k+1}}{n_1}}{1 - \frac{(k+1)n_{k+1}}{n_1}} \qquad \alpha(w_{i-1}) = \frac{1 - \sum_{w_i:c(w_{i-1}w_i)>0} P_{katz}(w_i|w_{i-1})}{1 - \sum_{w_i:c(w_{i-1}w_i)>0} P(w_i)}$$

where $P_{katz} = C_{katz}(w_{i-1}w_i) / \sum_j C_{katz}(w_{i-1}w_j)$.

2.2 Generalization and Longer Context Dependencies

Smoothed n-gram language models are widely used in applications such as speech recognition and statistical machine translation for hypothesis re-scoring. Despite their empirical success, these models fall short in capturing two major linguistic phenomena: generalization and long context dependencies.

2.2.1 Generalization and Class-Based Models

Suppose our training corpus contains two sentences: *Party will be on Tuesday* and *Party will be on Friday.* Assuming the word *Wednesday* is present in the vocabulary, the model should be able to generalize to the sentence *Party will be on Wednesday.* This expectation is based on the assumption that the model should inherently learn to "cluster" the words used in a similar context in the corpus. The n-gram based language models fail to capture this semantic or syntactic relatedness of the words because they solely rely on n-gram and lower order corpus statistics.

Class-based models like [3] address this generalization issue by mapping each word in the vocabulary to one or more predetermined classes and then building an n-gram model over the words and the classes. The classes can either be manually curated by experts or learned from the training corpus.

In their paper, Brown et al. [3] decompose the language modeling problem as predicting the class $c(w_t)$ given the class $c(w_{t-1})$ and then predicting the word w_t given the class $c(w_t)$, where c is a function that maps each word to one of the given k word classes.

$$p(w_1..w_n) = \prod_{t=1}^{n} P(w_t|c(w_t))P(c(w_t)|c(w_{t-1})) \qquad (1)$$

In the same paper, the authors also introduced a clustering algorithm that maps words to classes. This scheme is popularly known as *Brown clustering.* It relies on the bigram assumption and uses the log-likelihood of the corpus as a quality measure of the cluster. This log-likelihood is further decomposed into the entropy of the sequence and the mutual information among clusters. The clusters are selected by maximizing the mutual information measure. Let $c : V \to \{1, \dots, k\}$ be the clustering function and $Q(c)$ be the measure of quality of the cluster.

$$
\begin{aligned}
Q(c) &= \frac{1}{n}\sum_{t=1}^{n} \log\Big(P(w_t|c(w_t))P(c(w_t)|c(w_{t-1}))\Big) \\
&= \sum_{ww'} \frac{n(ww')}{n} \log\Big(P(w'|c(w'))P(c(w')|c(w))\Big) && \because \text{bigram assumption} \\
&= \sum_{ww'} \frac{n(ww')}{n} \log \frac{P(c(w), c(w'))}{P(c(w))P(c(w'))} + \sum_{ww'} \frac{n(ww')}{n} \log\Big(P(w')\Big) \\
&= \sum_{ww'} \frac{n(c(w)c(w'))}{n} \log \frac{P(c(w), c(w'))}{P(c(w))P(c(w'))} + \sum_{w'} \frac{n(w')}{n} \log\Big(P(w')\Big) && \because n(c(w), c(w')) = n(w, w') \\
&= I(c) - H(P),
\end{aligned}
$$

where $H(P)$ is the entropy of the word distribution.

As only the mutual information term $I(c)$ depends on c, the algorithm finds the clusters by maximizing the mutual information. The classes are discovered in a greedy fashion by an iterative algorithm that maximizes the mutual information at each step. This is done by considering each word as its own class and then iteratively merging the two classes that most increase the mutual information of the classes, until the desired number of classes C are left. The naive greedy approach is $O(V^5)$ but clusters can be computed in an efficient manner in $O(V^3)$.

2.2.2 Long Context Dependencies and Structured Language Models

Let's consider another example sentence: *The sky above our head is blue today.* The word *blue* in the given sentence is related to the word *sky.* But, if we use the bigram model for language modeling, the word *sky* will lie outside the Markovian boundary and hence will not influence the prediction of the word *blue.* This pattern where related words lie at a distance is common all throughout language. The n-gram based language models fail to capture such longer context dependencies.

Structured language models (SLM) [4] address this issue by augmenting the n-gram model by incorporating syntactic structure, in the form of the last p exposed headwords of the constituents present in the history[1]. These headwords can lie beyond the context boundary and can augment the information present in the local context.

The SLMs operate left-to-right like an n-gram language model but jointly model the probability of the sequence $w_1..w_n$ as well as the binary branching parse tree T over the sequence: $P(w_1 \dots w_n, T)$. The headword annotations $h_i^{w_1 \dots w_n}$ can then be extracted from the given parse tree T using heuristics. The tree T and headword annotations in T are recursively built in a similar fashion to a left-to-right parser.

The models can be sub-divided into two modules: PARSER and PREDICTOR. The PREDICTOR predicts the next word w_t given the n-gram local context and last p exposed headwords derived from the the partial parses of $w_1 \dots w_{t-1}$. The PARSER is a standard left-to-right parser which generates the partial parse trees for the span $w_1 \dots w_t$ recursively from the word w_t and

[1]The headwords in the English language are the words that determine the nature of the phrase and capture the most important lexical information in the constituent. The exposed headword, in turn, is the topmost headword of the incomplete parses in the history.

the parses for the span $w_1 \dots w_{t-1}$. So, the probability distribution can then be factorized as:

$$P(w_1 \dots w_n, T) = \prod_{k=1}^{n} p(w_k | h_{-p+1}^{w_1 \dots w_{k-1}} \dots h_0^{w_1 \dots w_{k-1}}, w_1 \dots w_{k-1})$$

where $h_{-i}^{w_1 \dots w_{k-1}}$ is the ith rightmost exposed headword for the sequence $w_1 \dots w_{k-1}$.

3 Neural Language Models

The current state-of-the-art in language modeling are neural network based models. The main reason for their empirical success is their ability to implicitly cluster together words appearing in a similar context. These approaches work by learning a linear embedding function $C : V \to \mathbb{R}^d$ which maps words from the vocabulary in a continuous d-dimensional latent space. The embeddings of the context are then fed to a neural network g that builds a probability distribution $P(w_t | w_1 \dots w_{t-1})$. The model is optimized to learn both the parameters of the neural network g and the word embedding function C. An alternate way to look at neural network based models is as class-based models with the class random variable modeled as a continuous n-dimensional real-valued vector instead of a discrete value.Both feed-forward neural networks and recurrent neural networks can be used to build such models, as we explain in detail below.

3.1 Feed-Forward Neural Language Model

The *Feed-Forward Neural Network Language Model* (NNLM) [5] is a direct extension of n-gram models to continuous vector spaces. As in n-gram models, a fixed history of the last $n-1$ words is the input to the model. Each of the $n-1$ words is first mapped to a 1-of-V vector which is 1 at the index corresponding to word w and zero at the other $|V|-1$ positions. This one-hot encoding vector is then fed to the embedding function, leading to a fixed length $(n-1)d$-dimensional continuous vector, which we denote x:

$$x = C(w_{t-n})C(w_{t-n-1}) \dots C(w_{t-1})$$

The NNLM's architecture is two-layered, with one h-dimensional hidden linear layer followed by a hyperbolic tangent non-linearity. The output from the hidden layer is then projected to a $|V|$-dimensional output space. In parallel, the embedded input x is projected into the same output space via an optional linear layer. Let y be the output corresponding to the input x:

$$y = g(x) = b + Wx + U \tanh(b_h + Hx),$$

where b (a $|V|$-dimensional vector), W (a $|V| \times (n-1)d$-dimensional matrix), U (a $|V| \times h$-dimensional matrix), H (a $h \times (n-1)d$-dimensional matrix) , b_h (a h-dimensional vector) and embedding function C (a $d \times |V|$-dimensional matrix) are the parameters of the model.

The probability distribution over the next word w_t given the context $w_{t-n} \dots w_{t-1}$ is built by stacking a *softmax* layer on top of the output layer:

$$P(w_t | w_1 \dots w_{t-1}) = \text{softmax}(y) = \frac{e^{y_{w_t}}}{\sum_w e^{y_w}}$$

where y_{w_t} is the index corresponding to word w_t in the vocabulary.

3.2 Recurrent Neural Language Model

One of the limitations of NNLM is the reliance on a fixed-length context window, which fails to take into account longer context dependencies. The *Recurrent Neural Language Model* (RNNLM) [6] addresses this issue by modeling the neural network function g as a simple recurrent neural network (RNN).

RNNLM captures longer context dependencies by learning a more effective representation of history during training. In the recurrent model, the history is represented as a continuous state vector $s(t)$ which is updated recursively from the previous state $s(t-1)$ and the current word

$w(t)$. This effectively means that theoretically, the entire history can be used for predicting the next word.

As in the feed-forward neural model, the input word at time t is mapped to a $|V|$-dimensional 1-of-V vector $w(t)$. This one-hot encoded vector is projected into the embedding space using an embedding matrix C and is then added to the recurrent projection of the state vector $s(t-1)$ to generate the input vector $x(t)$. This input vector is then projected to a latent space using an h-dimensional hidden layer which is followed by a sigmoid non-linearity. The result is the state vector $s(t)$, which approximates the history up to time step t. The state vector $s(t)$ is then projected to the output space with a $|V|$-dimensional output layer. A softmax layer is used in similar fashion to NNLM to normalize the output into a conditional probability $P(w_{t+1}|w_1 \dots w_t)$. RNNLM can be expressed mathematically as:

$$\begin{aligned} x(t) &= Cw(t) + U_{rec}s(t-1) \\ s(t) &= \sigma(x(t)) \\ y(t) &= Ws(t) \\ P(w_{t+1}|w_1 \dots w_t) &= \text{softmax}(y(t)), \end{aligned}$$

where $\sigma(z) = 1/1 + e^{-z}$ is the sigmoid function, C (a $h \times |V|$-dimensional matrix), U_{rec} (a $h \times h$-dimensional matrix) and W (a $|V| \times h$ dimensional matrix) are the parameters of the model.

3.3 Training Neural Networks based Language Models

Neural network based language models are trained by maximizing the log likelihood of the training corpus. Let T be number of words in the corpus and θ be the parameters of the model. The training objective is defined as:

$$L(\theta) = \frac{1}{T}\sum_{i=0}^{T} \log(P(w_i|w_1 \dots w_{i-1}); \theta) + R(\theta),$$

where $R(\theta)$ is a regularization term, such as weight decay. The training is done using stochastic gradient ascent and gradients are computed usually using the *Backpropagation* algorithm. The gradient computation for RNNs uses a specific formulation of backpropagation called *Backpropagation Through Time* (BPTT) where the recurrent neural network is unrolled across time and is treated as a very deep feed-forward neural network with parameters shared across the layers.

3.4 Exploding and Vanishing Gradients

RNNs in their original formulation have proven to be difficult to train due to exploding and vanishing gradients. This problem can be attributed to BPTT, as the model can theoretically have an unbounded number of layers, and propagating gradients through a large number of layers can lead the gradient to either diverge or vanish. Pascanu et al. [7] explain this problem in detail.

Let $L_t(\theta)$ be the loss at time t. The gradient of L_t w.r.t parameters θ can then be computed as:

$$\frac{\partial L_t}{\partial \theta} = \sum_{k=1}^{t} \Big(\frac{\partial L_t}{\partial s(t)} \frac{\partial s(t)}{\partial s(k)} \frac{\partial^+ s(k)}{\partial \theta} \Big)$$

where $\frac{\partial^+ s(k)}{\partial \theta}$ is the immediate partial derivative of state $s(k)$ w.r.t. θ. The $\partial s(t)/\partial s(k)$ term can further be expanded as:

$$\frac{\partial s(t)}{\partial s(k)} = \prod_{i=k}^{t} \frac{\partial s(i+1)}{\partial s(i)}$$

Pascanu et al. explain in their paper that the exploding and vanishing gradient can be traced back to the norm of the Jacobian matrix $\partial s(i)/\partial s(i-1)$. Assuming that $\forall i \ \ \|\partial s(i)/\partial s(i-1)\| <$

$\eta < 1$, the gradient contribution of the term $s(t-1)$ towards $\partial L_t/\partial\theta$ is proportional to η^{t-k}:

$$\frac{\partial L_t}{\partial\theta} = \sum_{k=1}^{t}\Big(\frac{\partial L_t}{\partial s(t)}\prod_{i=k}^{t}\frac{\partial s(i+1)}{\partial s(i)}\frac{\partial^+ s(t)}{\partial\theta}\Big) \le \sum_{k=1}^{t}\Big(\frac{\partial L_t}{\partial s(t)}\eta^{t-k}\frac{\partial^+ s(t)}{\partial\theta}\Big) \tag{2}$$

For $t \gg k$, as $\eta < 1$, the gradient contribution of the term corresponding to the input at k will go to zero exponentially fast and hence hinder learning the long-term dependencies. Similarly, if $\eta > 1$, for $t \gg k$, the contribution of the term at k will blow up, leading to the exploding gradient problem.

The exploding gradient problem is fairly easy to handle. Mikolov et al. [6] proposed a simple solution that works well empirically. They suggested clipping the gradient to keep it in the range $[-\gamma, \gamma]$:

$$\frac{\partial L}{\partial\theta} = \begin{cases} \frac{\partial L}{\partial\theta} & \text{if } \left\|\frac{\partial L}{\partial\theta}\right\| < \gamma \\ \frac{\gamma}{\left\|\frac{\partial L}{\partial\theta}\right\|} & \text{if } \left\|\frac{\partial L}{\partial\theta}\right\| \ge \gamma \end{cases}$$

A number of solutions have been proposed to handle the vanishing gradient problem. The most popular approach that addresses this problem is Long Short-Term Memory[8] networks, which we review next.

3.4.1 Long Short-Term Memory:

The *Long Short Term Memory* (LSTM) [8] network is a type of RNN with a specific architecture. A cell in the LSTM network is shown in Figure 1 reproduced from [9]. Mathematically, LSTM functionality can be expressed using the following set of equations:

$$i(t) = \sigma\Big(W_i[x(t)s(t-1)]^T\Big) \tag{3}$$
$$f(t) = \sigma\Big(W_f[x(t)s(t-1)]^T\Big) \tag{4}$$
$$o(t) = \sigma\Big(W_o[x(t)s(t-1)]^T\Big) \tag{5}$$
$$g(t) = \tanh\Big(W_g[w(t)s(t-1)]^T\Big) \tag{6}$$
$$c(t) = f(t)\circ c(t-1) + i(t)\circ g(t) \tag{7}$$
$$s(t) = o(t)\circ \tanh(c(t)) \tag{8}$$

Equations (3-5) correspond to the computation of three gates: the input gate $i(t)$, the output gate $o(t)$ and the forget gate $f(t)$. The value of these gates is between 0 and 1, with extremes corresponding to the gate being fully closed or opened respectively. $g(t)$ is the network's input at time t and $\circ$ denotes element-wise multiplication.

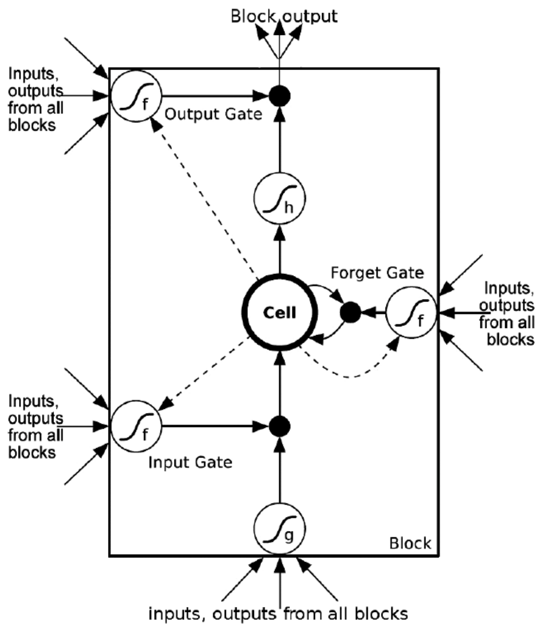
Block output
Inputs, outputs from all blocks
Output Gate
h
f
Forget Gate
Cell
Inputs, outputs from all blocks
Inputs, outputs from all blocks
Input Gate
g
Block
inputs, outputs from all blocks

Figure 1: A single LSTM cell. [9]

The central idea behind the LSTM is the use of the cell state $c(t)$ in Equation 7. The first part of the equation is a self-loop modulated by the forget gate. The function of this self-loop is to retain information over time steps. If all the other gates are closed and the forget gate is fully opened, this will lead to a constant error flow i.e. $\partial c(t)/\partial c(t-1) = 1$. This implies that in the absence of a new input (input gate closed) or an error signal (output gate closed), the gradient from the previous step doesn't change, hence avoiding the vanishing or exploding gradient problem.

The memory cell lacks the ability to reset itself or to remove irrelevant information accumulated over a long period of time. This might lead the value in each cell to grow in an unbounded manner. The forget gate $f(t)$ helps regulate the cell by acting as a contraction multiplier for the cell's value at time $t-1$ during the update step.

The input gate controls the flow of input from the other cells at the previous time step $s(t-1)$ and the input $x(t)$. It ensures that only input relevant for this cell is added to the memory cell i.e., if the input gate is fully closed, the input will be ignored. The output cell acts in a similar fashion to the input gate but it controls the gradient or error signal flow to the memory cell. It ensures that only the relevant gradient reaches the previous input states.

The LSTM architecture has proven to be highly effective in dealing with the vanishing and exploding gradient problem in recurrent neural networks and is currently the backbone of most of the recurrent neural network based architectures for language modeling and understanding tasks.

3.5 Handling Large Vocabulary

One of the bottlenecks in training and doing inference with neural language models is the cost of computation of the softmax at the output layer. The output layer is $|V| \times h$-dimensional matrix, so the cost of computing the conditional probabilities scales linearly with the vocabulary size.

Let $u(w, c; \theta)$ be the un-normalized output of the final layer of the neural network. The conditional probability is given by

$$P(w|c;\theta) = \frac{e^{u(w,c;\theta)}}{\sum_{w' \in V} e^{u(w',c;\theta)}}$$

The gradient $\partial \log(P(w|c;\theta))/\partial\theta$ is calculated in terms of $\partial u(w,c;\theta)/\partial\theta$ as

$$\frac{\partial \log(P(w|c;\theta))}{\partial\theta} = \frac{\partial u(w,c;\theta)}{\partial\theta} - \mathbb{E}_p\left[\frac{\partial u(w,c;\theta)}{\partial\theta}\right]$$

Again, computing the contribution of a single word w to the gradient is expensive as the expectation term in the gradient scales linearly with the vocabulary size.

Noise Contrastive Estimation (NCE) [10] and *Hierarchical Softmax* [11, 12] are two methods that have been proposed to reduce the computation cost of training neural language models, and which we review next.

3.5.1 Noise Contrastive Estimation

The basic idea of the NCE [10] is to reduce the problem of density estimation to that of binary classification. The classification problem is formulated as discriminating between samples from the data distribution and samples from a known noise distribution. In the language modeling use case, the data distribution is the conditional probability of generating word w_k given context c, i.e. $P_\theta(w_k|c)$. The authors propose using the uni-gram probability of the word w_k i.e. $P_u(w_k)$ as the noise distribution. Assuming the noise distribution is sampled k times more frequently than the conditional distribution, the posterior probability that the sample w came from the noise $(D = 0)$ or the data $(D = 1)$ distribution is given by:

$$P(D=1|w,c;\theta) = \frac{P_\theta(w|c)}{P_\theta(w|c) + kP_u(w)}$$

$$P(D=0|w,c;\theta) = \frac{kP_u(w)}{P_\theta(w|c) + kP_u(w)}$$

Additionally, as the normalization is still required for computing $P_\theta(w|c)$, the authors propose to model the normalization constant for the context c as a learning problem with parameter θ_c.

$$P(w|c;\theta') = u(w,c;\theta)\exp(\theta_c),$$

where $\theta' = [\theta; \theta_c]$.

Now, the language modeling problem is posed as maximizing the log-likelihood of the posterior under the mixture of noise and sample. Mathematically, this can be expressed as:

$$L_{NCE_k} = \sum_{w,c}\left(\log(P(D=1|c,w) + \sum_{i=1,\bar{w}\sim P_u}^{k} \log(P(D=0|c,\bar{w}))\right).$$

The gradient $\partial L_{NCE_k}/\partial\theta'$ then can be computed as

$$\frac{\partial L_{NCE_k}}{\partial\theta'} = \sum_{w,c}\left(P(D=0|c,w)\frac{\partial u(w|c;\theta')}{\partial\theta'} - \sum_{i=1,\bar{w}\sim P_u}^{k}\log(P(D=1|c,\bar{w}))\frac{\partial u(w'|c;\theta')}{\partial\theta'}\right)$$

With the NCE training approach, there is no need to compute the softmax while estimating the posterior or the gradient. This leads to an approximate speedup of $|V|/k$, because instead of computing the softmax, we are just sampling k negative samples at each step.

3.5.2 Hierarchical Softmax

The Hierarchical Softmax based language models [11, 12] draw inspiration from class-based models, but instead of one level of decomposition (words into classes), it decomposes the output layer hierarchically to form a balanced binary tree. Each word in this formulation is a leaf node of this tree and is represented by a binary vector $(b_1(v), \dots b_m(n))$ where the ith bit indicates the decision to go left or right in the binary tree at the ith level. Mathematically, this formulation of language modeling problem can be decomposed as

$$P(w_t|w_1\dots w_t) = \prod_{j=1}^{m} P(b_j(w_t)|b_1(w_t)\dots b_{j-1}(w_t), w_1\dots w_{t-1})$$

i.e. predicting the next bit $b_j(w_t)$ of the word representation w_t given the context $w_1\dots w_{t-1}$ and previously predicted bits $b_1(w_t)\dots b_{j-1}(w_t)$ for the word w_t.

The probability $P(b_j|b_1\dots b_{j-1}, w_t - 1\dots w_{t-n+1})$ is modeled as

$$P(b_j|b_1\dots b_{j-1}, w_t - 1\dots w_{t-n+1}) = \sigma(b + Wx + U\tanh(b_h + Hx + KN_j))$$

where b, W, x, U, b_h and H are the same parameters as NNLM and K (a $h\times m$ matrix) is a projection matrix which projects the 1-of-m vector N_j representing the position or node level of the bit to be predicted into a continuous space. Training and inference with the hierarchical formulation of the language modeling problem gives an approximate speed up of $|V|/log_2(|V|)$.

In their paper, Morin et al. [11] use a handcrafted balanced binary tree derived from WordNet IS-A relationships. Mnih et al. [12] extend this model to learn these hierarchies in a bootstrap fashion. They start with a random balanced binary tree and learn a language model using hierarchical softmax. They then generate a contextual representation for each word by averaging its context embeddings[2]. Then a mixture of two Gaussians is fit to these contextual embeddings and the embeddings are partitioned into two clusters based on their responsibilities of the two mixture components. This partitioning is applied recursively to each of the two clusters until only two words are left in a cluster.

4 Language Model Evaluation

4.1 Extrinsic Evaluation

Language models are extrinsically evaluated as a part of automatic speech recognition (ASR) or machine translation (MT) systems. In these setups, the language model is used as hypothesis re-ranker. A benchmarking ASR or MT system is first used to generate an n-best list which is then re-ranked by the language model. The top re-ranked entry is then evaluated using a task specific metric such as word error rate (WER) or BLEU score. These task-specific metrics are treated as a proxy for the quality of the language model, and a relative better performance on downstream tasks indicates a superior language model.

[2]In their paper, Mnih et al. use the Log Bi-Linear variant of NNLM for which the context embeddings are d dimensional but the algorithm holds for $(n-1)\times d$ dimensional projection of context in continuous space by NNLM. We can choose n and d accordingly depending on our vocabulary size.

4.2 Perplexity

Extrinsic evaluation is often computationally expensive and needs an established ASR or MT evaluation pipeline. An alternative approach is to define an easy to compute intrinsic evaluation metric which can serve as a proxy for the quality of the model. Perplexity is the most common intrinsic evaluation metric used to measure language model performance. The perplexity (per word) of a text sequence $w_1 \dots w_n$ under model m is the inverse of the probability of the sequence under model m, averaged geometrically over the length of the sequence:

$$PPL(w_1 \dots w_n; m) = \left(\frac{1}{m(w_1 \dots w_n)} \right)^{1/n}$$

There are various ways to interpret perplexity. One of the most intuitive interpretations is as effective vocabulary size i.e., a perplexity value of x means the model m will be as confused on a test dataset, with a much larger vocabulary, as if it had to choose the next word uniformly out of only x words. Another interpretation is the ability of model m to compress or fit the data. From this perspective, perplexity can be seen as an exponentiation of the number of bits per word needed to encode the data in binary format.

An information-theoretic view of the same metric is as exponentiated cross-entropy, with cross-entropy approximated as:

$$H(p, m) = -\frac{1}{n} \log_2 m(w_1 \dots w_m)$$

This perspective helps us understand why perplexity is a good metric for language model evaluation. The cross-entropy of two probability distributions p and m measures how far the modeled probability distribution m is from the original distribution p. So, a model with lower cross-entropy (hence lower perplexity) will build a probability distribution that will be closer to the oracle distribution p.

5 Neural Language Model Applications

5.1 Sequence to Sequence models

The traditional application of language models has been as a linguistic decoder in the noisy channel model for statistical machine translation and speech recognition. An alternate perspective is to treat the language model as a hypothesis re-ranker. In this formulation, the acoustic model generates an n-best list of hypotheses and a language model is used to re-rank these hypotheses based on the possibility that they could have been generated by the language model. Neural language models such as [6] have been used for this purpose. A recent trend in machine translation and speech processing is to treat these problems as an end-to-end sequence to sequence learning (Seq2Seq) problem [13].

RNN based models can only learn to map input and output sequences of the same length. It is not straightforward to use an RNN for applications such as machine translation, where input and output sequences have different lengths. The Seq2Seq model extends the RNNLM by using two recurrent nets. The first RNN, called encoder, is used to map the input sequence to a fixed length vector. The other RNN, called decoder, is then used to map this fixed length vector to the target sequence. The fixed length vector is usually the final hidden state of the encoder.

The decoder RNN can be seen as a conditional RNN based language model, where conditioning is done on the encoder output. Let $x_1 \dots x_T$ be the input sequence and $s(T)$ be the encoder hidden state corresponding to the final input token x_T. This is the fixed length vector that will be the input to the decoder. Let $y_1 \dots y_{T'}$ be the output sequence. The problem then becomes to model:

$$P(y_1 \dots y_{T'} | x_1 \dots x_T) = \prod_{t=1}^{t=T'} P(y_t | y_1 \dots y_{t-1}, s(T)).$$

The model is trained to maximize the log-likelihood of the target sequence in a similar fashion to other neural network based language models. The major difference with the Seq2Seq models is that the model parameters will include both the encoder and the decoder RNN parameters, and the loss will be back-propagated through both RNNs.

In their paper, Sutskever et al. applied the Seq2Seq learning approach to machine translation and showed that it performed better than other contemporary approaches. They used deep LSTMs (4 layers) for both encoder and decoder. Also, they found that feeding the input sequence in the reverse order lead to better empirical performance on the BLEU metric.

5.2 Word Embeddings

The embeddings learned during neural language model training have shown the ability to cluster words contextually in latent space as well as capture the semantic and syntactic relatedness among words [14]. Additionally, these pre-trained embeddings have been used to initialize the embedding layer of neural network models for downstream tasks such as named entity recognition (NER) and chunking and have lead to considerable performance improvement on task-specific metrics. This has lead to considerable interest in learning high-quality embeddings as a standalone task.

One of the earliest attempts at learning word embeddings is the Word2Vec class of models by Mikolov et al. [15]. This paper introduces two models: the *Continuous Bag of Words* (CBOW) model and the *Skip-Gram* model. These models are based on the NNLM, with the main difference that they have no hidden layer. This optimization is motivated by the fact that in NNLM with hierarchical softmax, the dominating factor in the computational cost is the input to the hidden layer projection. Removing the hidden layer can help avoid this costly computation, making training on a large corpus feasible.

The CBOW model formulates the learning problem as predicting the word given the context which contains r words before and after the current (middle) word. Another deviation from the NNLM architecture is that the embedding layer maps all $2r$ context words to a single d-dimensional embedding by averaging the projections. This assumption is similar to the bag of words model, as the order of words in context does not influence the projection. The computational complexity of each step of this model is $2r \times d + d \times \log_2(|V|)$.

The second architecture proposed in the paper inverts the language modeling problem and formulates the learning problem as predicting the context given the current (middle) word. The context, in this case, is similar to the CBOW model and given by r words on either side of the current word. The computational cost of each step of this model is $2r \times (d \times d \times \log_2(V))$.

The word embeddings learned by models such as Skip-Gram and CBOW are evaluated both intrinsically and extrinsically. The extrinsic evaluation is done using neural network based models for language understanding tasks, for example, sentiment classification, NER, and chunking. In these models, the embedding layer is initialized with the pre-trained embeddings and the performance of the model on the task is considered a proxy for the performance of the embeddings. The pre-trained embeddings are intrinsically evaluated on an array of word similarity tasks framed as analogy questions, in order to capture the semantic and syntactic relatedness among the words. For example, the analogy question *France* : *Paris* :: *Germany* :? captures the *capital-of-the-country* relation whereas *biggest* : *big* :: *small* :? is an example of analogy question designed to measure syntactic *superlative* relatedness among the embeddings.

In their previous paper, Mikolov et al. [14] showed that pre-trained embeddings are able to capture semantic and syntactic relations by simply adding the vector offset from similarly related words to the word's embeddings. For example, the embedding of the word *Berlin* can be approximated by adding the difference in embeddings for *Paris* and *France* to the word

$Germany$ i.e $x(Berlin) \approx x(Paris) - x(France) + x(Germany)$.

The authors used the same approach to evaluate CBOW and Skip-Gram embeddings and showed that Skip-Gram out-performed CBOW and all the other baselines including NNLMs and RNNLMs embeddings as well as other contemporary word embeddings.

5.3 Contextualized Embeddings:

Recently, the focus on pre-trained embeddings has shifted to learning contextualized embeddings, which try to address the criticism that word embedding fail to capture linguistic phenomena such as homonomy and polysemy. Thus, the objective of learning the contextualized embedding is to capture characteristics of the word which are dependent on its usage.

5.3.1 ELMo

The ELMo (Embeddings from Language Models) [16] model formulates the word embedding as a function of the whole sentence containing the word. Contextual word embeddings are learned by building a multi-layered bi-directional LSTM-based language model (biLM) on a sentence-segmented input corpus. The ELMo embedding of the word is then computed as a scaled weighted average of the LSTM hidden states across layers.

Bi-Directional Language Model Let the $w_1 \dots w_n$ be the input sentence. A forward (or standard) language model will build the probability distribution on a sentence by factorizing the distribution into a conditional probability of generating word w_k given the context $w_1 \dots w_{k-1}$. The backward language model is similar to the forward one except that it runs over the sequence in reverse i.e., it predicts the word w_k given $w_{k+1} \dots w_n$:

$$p(w_1 \dots w_n) = \prod_{k=1}^{n} p(w_k | w_{k+1} \dots w_n).$$

ELMo uses an L-layer deep LSTM network for modeling both the forward and the backward models.

Let x_k be the embedding for the kth word in the sentence. Let $\overrightarrow{h}_{k,j}$, $\overleftarrow{h}_{k,j}$ be the hidden state representation of the LSTM at layer j for the word at position k in the forward and the backward model respectively. The biLM representation corresponding to layer j for the kth word is defined as $h_{k,0} = x_k$ and $h_{k,j} = [\overrightarrow{h}_{k,j}, \overleftarrow{h}_{k,j}], \forall j > 1$.

The task-dependent ELMo embedding for w_k is then computed as a scaled weighted average of all the hidden layers for w_k:

$$ELMo_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{k,j}$$

where s_j^{task} are task-specific weights with a constraint $\sum_{j=0}^{L} s_j^{task} = 1$, and γ^{task} is a task-specific scaling hyper-parameter that helps the optimization during task-specific supervised training.

The ELMo model is trained in two phases. In the first, unsupervised *pre-training* phase, the biLM is trained on a large corpus by maximizing the log-likelihood of biLM given by:

$$L = \sum_{t=1}^{N} \left(log\Big(P(w_t | w_1 \dots w_{t-1}; \Theta)\Big) + \log\Big(P(w_t | w_{t+1} \dots w_n; \Theta)\Big) \right),$$

where Θ are the parameters of the L stacked LSTMs and the embedding layer X. The output of this phase are the L hidden state representations $h_{k,j}$.

The pre-training phase is followed by the supervised *task-specific training*. In this phase, the biLM model is frozen and the supervised task model and task-specific weights s_j^{task} are learned. The hyper-parameter γ_{task} is selected using the supervised task's validation set.

The ELMo model was evaluated on an array of supervised natural language understanding tasks such as natural language inference, semantic role labeling, co-reference resolution, NER, and sentiment analysis, and out-performed the contemporary state-of-the-art models on all the benchmarks. Additionally, authors demonstrated that adding ELMo embeddings to the task-specific model makes it more sample efficient both in terms of the number of updates needed to reach state-of-the-art performance as well as in training data size.

The authors also analyzed the kind of information captured by lower vs upper layers of the biLM. This was done by measuring the performance of the hidden state representations on Word Sense Disambiguation (WSD) and Part of Speech (POS) tagging tasks. The authors found that the lower level layers do better on the POS tagging task whereas the top layers perform better at the WSD task. This validated the claim that the lower layers capture the syntactic information whereas the higher layers do a better job at capturing the contextual and semantic information.

5.3.2 ULMFiT

In their paper, Howard and Ruder [17] pose learning contextual representations as an inductive transfer problem. The inspiration for their model comes from the observation that features in computer vision (CV) models transition from general in the first layer to task-specific in the final layer . This enables the CV community to train models on large object classification datasets such as ImageNet and re-use the first few layers of the model for applications like segmentation, recognition etc. These models usually are fine-tuned for the specific task and provide sample efficiency in terms of training data requirements and number of updates to reach state-of-the-art performance.

Word embeddings such as Word2Vec can be seen as an example of this inductive learning, where only the first layer is transferred and the architecture for a specific task, such as sentiment analysis, is built on top of this layer. The authors in their paper wish to invert this paradigm by pre-training a deep LSTM-based source model on a large corpus and transfer all but the top layer of the model to a task-specific model, which just adds a shallow classification network on top of the pre-trained LSTM model.

The training procedure for the ULMFiT model can be divided into three stages 1.) General-domain LM pre-training, 2) Target task LM fine-tuning, and 3) Target task classifier fine-tuning. The authors use a 3-layered deep LSTM as their base model. The first stage is similar to training an RNNLM model. The other two stages deal with the domain shift in data and task specific classifier training respectively.

Target task LM fine-tuning

This stage addresses the data distribution mismatch between the task's input data and the general-domain data used for LM pre-training. To address this issue, the model is fine-tuned with input text data for the task using the LM objective. The authors propose two novel techniques to fine-tune the LM that avoid catastrophic forgetting of the features learned during general-domain LM training.

Discriminative fine-tuning: The authors propose a different learning rate for each layer of the model, with the learning rate geometrically decreasing from the top to the bottom layer. The intuition behind this idea is that different layers of a model capture different information, ranging from least to most general from top to the bottom layer and more task-specific features at the top will need more fine-tuning.

Slanted triangular learning rates: The authors propose the use of an adaptive learning rate schedule, which they call slanted triangular learning rate. This schedule first aggressively increases the learning rate and then decays it slowly. The intuition is to make the model quickly converge to a suitable region of the parameter space at the beginning of training and then refine its parameters in this region.

Target task classifier fine-tuning

Finally, for training the classifier, the authors removed the output layer of the language model and added two linear layers on the top of the pre-trained LSTM layers. The topmost layer is followed by a softmax function to generate a probability distribution over the class labels. Each layer is also followed by a batch normalization layer and dropout is applied to each layer. The ReLU non-linearity is added as an activation between the two linear blocks. The parameters of the LSTM are only fine-tuned whereas those of the two linear layers are learned from scratch. The classifier is trained to maximize the log-likelihood of observing the label given the training example.

The authors observed that catastrophic forgetting is a major issue and proposed another novel technique called *gradual unfreezing* that is used in addition to *discriminative fine-tuning* and *slanted triangular learning rate* during this stage.

Gradual unfreezing: To avoid catastrophic forgetting, the authors proposed to start the training with all LSTM layers frozen and unfreeze them gradually, one per epoch, starting from the top layer. The intuition behind this scheme is that the top layer captures the least general features and must be fine-tuned the most to capture task-specific features. The process of unfreezing the layer is repeated until all the layers have been fine-tuned and have reached convergence.

The authors evaluated their model for transfer on six different NLP classification tasks, including sentiment analysis, question classification, topic classification etc. They showed that the ULMFiT achieved state-of-the-art performance on all the tasks. They also demonstrated that their model was able to match the previous state-of-the-art performance on these tasks with 50x to 100x fewer samples.

Both ULMFiT and ELMo built contextualized word embeddings by pre-training on the language modeling objective. Both models were able to beat previous state-of-the-art on an array of NLP benchmarks and demonstrated that language modeling is a good pre-training objective for transfer. The two models differ on how these pre-trained LSTM layers are used. The ELMo model keeps the embeddings fixed after pre-training and relies on a complex model architecture to solve the domain task. The ULMFiT model, on the other hand, doesn't need an engineered complex architecture, but only a few shallow layers on top of the pre-trained LSTM layers. This model simplicity though comes at the cost of involving complex training schemes such as *discriminative fine-tuning*, *slanted triangular learning rate* and *gradual unfreezing*.

6 Conclusion

Language modeling is one of the oldest and most widely studied problems in NLP. This report attempts to provide a concise summary of classical as well as neural network-based approaches for language modeling, highlighting the connection between various approaches as well as the language modeling perspective on NLP models like Word2Vec, Seq2Seq, ELMo and ULMFiT. The NLP research community is actively working on most of the issues discussed in this report, ranging from computationally efficient ways of solving the vanishing gradient problem, newer architectures for language and Seq2Seq models, to discovering simpler approaches for transfer from pre-trained language models to downstream tasks. We hope this exposition encourages researchers in the field to further explore these connections and leads to further improvement in both language modeling as well as its applications in downstream tasks.

References

[1] C. E. Shannon. Prediction and entropy of printed English. *The Bell System Technical Journal*, 30(1):50–64, 1 1951.

[2] Slava M. Katz. Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer, 1987.

[3] Peter F Brown, Peter V DeSouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-Based n-gram Models of Natural Language. *Association for Computational Linguistics*, 1992.

[4] Ciprian Chelba and Frederick Jelinek. Structured language modeling. *Computer Speech and Language*, 2000.

[5] Yoshua Bengio, Rjean Ducharme, Pascal Vincent, and Christian Jauvin. A Neural Probabilistic Language Model. In *Journal of Machine Learning Research*, 2003.

[6] T Mikolov, M Karafiat, L Burget, J Cernocky, and S Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010*, 2010.

[7] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks. *arXiv:1211.5063 [cs]*, 11 2012.

[8] Sepp Hochreiter and Jrgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 7 1997.

[9] Roger Grosse. Lecture 15 : Exploding and Vanishing Gradients. *cs.toronto.edu*, pages 1–11, 2017.

[10] Andriy Mnih and Yee Whye Teh. A Fast and Simple Algorithm for Training Neural Probabilistic Language Models. *arXiv:1206.6426 [cs]*, 6 2012.

[11] Frederic Morin and Yoshua Bengio. Hierarchical Probabilistic Neural Network Language Model. *In Proceedings of the international workshop on artificial intelligence and statistics*, 5:246252, 2003.

[12] Andriy Mnih and Geoffrey Hinton. A Scalable Hierarchical Distributed Language Model. *NIPS*, pages 1–8, 2009.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. pages 3104–3112, 2014.

[14] Geoffrey Zweig Tomas Mikolov, Wen-tau Yih. Linguistic Regularities in Continuous Space Word Representations. *Hlt-Naacl*, 2013.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient {Estimation} of {Word} {Representations} in {Vector} {Space}. *arXiv:1301.3781 [cs]*, 1 2013.

[16] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv:1802.05365 [cs]*, 2 2018.

[17] Jeremy Howard and Sebastian Ruder. Universal Language Model Fine-tuning for Text Classification. *arXiv:1801.06146 [cs, stat]*, 1 2018.