Text Generation with Neural Networks

Written Preliminary Examination II Critical Review

Daphne Ippolito

ABSTRACT

The automatic generation of natural-sounding human text has been a goal of artificial intelligence since the field's inception. Up until the last decade, most approaches relied on complex multi-step algorithms, context-free-grammars, and statistical methods. With the advent of deep learning, generated text has achieved a level of fluency and coherence that would have awed early researchers in the field. In this work, I briefly survey historic methods of automatic text generation then delve into three deep learning-based approaches: recurrent neural networks, convolutional networks, and self-attention-based networks. I describe the performance trade-offs between these methods and discuss how choices pertinent to all three methods, such as vocabulary selection and decoding method, impact generated text quality.

TABLE OF CONTENTS

CHAPTER 1 : Introduction

The ability of a computer to automatically generate text that sounds like a human wrote it has been a goal of artificial intelligence since the field's inception. Early systems relied on a pipeline of algorithms. At the start, a content determination algorithm would select the pieces of information the generated text ought to convey. Intermediate steps, including discourse planning and sentence aggregation, would impose order and structure on the information, outputting a tree encoding the relationships between facts as they should appear in each generated sentence. Finally, lexicalization, referring expression generation, and linguistic realization steps would convert the tree into natural text. These algorithms would map the entities and relationships from the tree to English words, then apply the rules of grammar to connect them into syntactically and morphologically correct English (Reiter and Dale, 1997). Each of the steps in this process was a research area in its own right, and each outputted an interpretable intermediate representation.

While many of the early algorithms proposed for each of these steps were rule-based, some researchers tried to using learning-based approaches trained on real data to perform some of the tasks. For example, Janarthanam and Lemon (2010) use reinforcement learning to generate referring expressions that adapt to the user, and Kondadadi et al. (2013) train a support vector machine to choose the best sentence template given the desired sentence content. However, it wasn't until the success of deep learning systems that researchers were able to replace nearly the entire pipeline with a single learned system.

Neural net approaches replace all of the steps except content determination with a single black box. The black box is a highly non-linear function containing hundreds of thousands to millions of tuneable weight parameters. They take as input the start of a text sequence and predict a probability distribution over the next word in the sequence. A neural net that takes this form is refereed to as a neural language model. During the training process, the model is exposed to human-written text sequences. The model weights get updated using gradient descent to make the true next word as likely according to the model as possible. Neural language models are a natural progression from statistical language models, which, given a large corpus of text, compute the empirical distribution over the the next word given the previous 1-5 words. They solve one of the biggest issues with statistical language models: that word sequences never seen in the input text are assigned 0 probability.

Different inductive biases in the neural language model architecture can make training easier or harder, altering the runtime complexity, model performance, or both. In this survey, we examine three architectures that have been proposed for the task of text generation. Recurrent neural nets process the input sequence one word at a time, encoding information about the words seen so far into a single fixed-length vector (Graves, 2013). Convolutional neural networks drop the idea of recurrence and instead perform a series of 1D convolutions on the input sequence (Gehring et al., 2017). Attention-based methods find that performance is further improved by employing only feed-forward and attention mechanisms (Vaswani et al., 2017).

Each of these architectures takes two forms, an encoder-decoder form and a decoder-only form. The encoder-decoder models, also referred to as sequence-to-sequence models, condition generation on an input text sequence, often referred to as the source sequence, in order to predict a target sequence. For example, in machine translation, a French sentences is used as input in order to predict a corresponding English sentence. An encoder encodes the source text, and the decoder depends on the source's encoding in order to make its next word predictions. Decoder-only models do not have access to a source sequence; a prediction for the next word depends only on the text of the sequence so far. They are sometimes called unconditioned language models (despite actually being conditioned on the text so far).

Encoder-decoder models have been applied to machine translation (Vaswani et al., 2017; Gehring et al., 2017; Sutskever et al., 2014), abstractive text summarization (Liu et al., 2018), conditioned story generation (Fan et al., 2018), and other tasks. Decoder-only models have been applied to news article generation (Zellers et al., 2019), unconditioned story generation (Radford et al., 2019), and language modeling benchmarking tasks (Dai et al., 2019), Theoretically, the two architectures are interchangeable. An encoder-decoder architecture devolves into a decoder-only one if the source sequence is always the empty string. Conversely, a task involving paired source and target sequences can be turned into a task for a decoder-only architecture by simply prepending the source sequence to the target, although this technique does not tend to work as well in practice (Raffel et al., 2019; Bahdanau et al., 2014).

Another dichotomy that is often applied involves the nature of the generation tasks themselves. Some tasks, like machine translation or abstractive summarization, have relatively clear correct answers. A translation or summary of an input text is either correct, or it is not. These tasks have widely accepted evaluation metrics which have been shown to correlate with human judgement. In contrast, open-ended text generation tasks are ones where the breadth of possible correct generations is enormous. Chit-chat dialog and story generation fall into this category. It is often difficult to evaluate open-ended generation systems, either with human raters or with automatic metrics.

This survey starts by discussing the components that tend to be shared across language models, whether they are recurrent, convolutional, or attention-based. Then, for each of these methods, it details the encoder and encoder-decoder versions of the architecture. Experimental results are described for machine translation (encoder-decoder) and language modeling (decoder-only), both tasks with well-established evaluation metrics (Papineni et al., 2002; Jelinek et al., 1977) and standardized training datasets (Barrault et al., 2019; Chelba et al., 2013). These results are summarized in Table 2. For each architecture, interesting extensions and applications in open-ended domains are also discussed. Finally, this survey concludes with a summary of the ongoing challenges with text generation as an area of research.

CHAPTER 2 : Common Components of Neural Language Models

All neural language models have several components in common. They all begin with choosing a tokenization algorithm that takes raw text and converts it into a sequence of integers, referred to as tokens. A vocabulary, a mapping from strings of characters to their token indices, must be decided on before tokenization can be performed. Once the training set has been tokenized, language models use an embedding matrix to convert tokens indices into vectors representations of each token. Language models designed for generation nearly all produce a prediction for the embedding of the next token in the sequence given the previous tokens. They then score each token in the vocabulary by how close it is the predicted embedding. A log-likelihood loss is used to encourage the true next token's embedding to score highest. Lastly, all three of the architectures being focused on take advantage of an operation called attention to allow the sequence information seen so far to be weighed at each prediction step. This chapter goes into detail about each of these common components.

Vocab Type	Example
character-level	`['A', ' ', 'h', 'i', 'p', 'p', 'o', 'p', 'o', 't', 'a', 'm', 'u', 's',' ', 'a', 't', 'e', ' ', 'm', 'y', ' ', 'h', 'o', 'm', 'e', 'w', 'o', 'r', 'k', '.']`
subword-level	`['A', 'hip', '##pop', '##ota', '##mus', 'ate', 'my', 'homework', '.']`
word-level	`['A', 'hippopotamus', 'ate', 'my', 'homework', '.']`

Table 1: Examples of the string "A hippopotamus ate my homework." tokenized using three different vocabularies. With the subword tokenizer, the rare word "hippopotamus" gets broken up into multiple tokens.

2.1. Vocabulary

Choosing a vocabulary is the critical first step toward building any text generation system. A vocabulary consists of the base units of language which can be produced at each step of generation. The simplest vocabularies are character-level–each letter of the alphabet and punctuation becomes a token. More complex vocabularies are formed by running an algorithm which joins letters together into larger units. Historically, word-level vocabularies, where each token corresponds an English word, were standard. Word-level vocabularies can be created by splitting a string at whitespace and punctuation,

More recently, subword vocabularies have been proposed. In a subword vocabulary, an English word might consist of multiple tokens. Common words typically end up as single tokens, but less common words are broken up into smaller units. Several greedy algorithms have been proposed to approximate optimally breaking up a text corpus into subwords, but byte-pair encoding (BPE) is currently the most popular (Sennrich et al., 2016). Table 1 shows the same string under a few different tokenization schemes.

Subword vocabularies were designed to be a compromise between the advantage and disadvantages of word-level and character-level vocabularies. Character-level vocabularies are usually very small, no more than a couple hundred tokens. However, the vocabulary can cover near every possible string a person could write. Word-level vocabularies cannot feasible contain the hundreds-of-thousands of words present in English text. Realistically, only the most common words are kept, and less common ones are replaced with a special `UNK` token. When text is tokenized with character-level vocabularies, the resulting sequences are very long, while word-level tokenization yields shorter sequences since there is just one token per word. Lastly, word-level representations learned by a neural net tend to more meaningful than character-level representations since a word has semantics associated with it that are common across uses. Subword vocabularies adopt the best of both worlds, using word-level tokens for common words but falling back to subword, or in the worst case, character-level, tokenization for uncommon words. This approach eliminates the need for an `UNK` token and results in tokenized sequence lengths which are somewhere between the two strategies. Nearly all current state-of-the-art text generation systems use some form of subword tokenization.

For all of the types of vocabularies discussed, a decision must be made on whether to convert strings to lowercase before vocabulary creation. Removing case allows for a more compact vocabulary, but it also removes potentially useful information about the location of proper nouns. Finally, a decision on the total vocabulary size must be made. The three papers focused on in this study make the following decisions:

- Graves (2013) run experiments with a character-level vocabulary and with a word-level vocabulary. The latter contains 10,000 words.
- Gehring et al. (2017) run experiments with both a word-level vocabulary and with BPE. For machine translate, a single joint vocabulary with 80k words is used for BPE. Separate source and target vocabularies (200k and 80k words) are used for word-level.
- Vaswani et al. (2017) only run experiments with BPE, using a vocabulary size of 37,000.

Once a vocabulary has been chosen, each vocabulary token is assigned an incrementing integer. Thus, text is reduced to a sequence of integers through the tokenization process.

2.2. Embedding Matrix

As discussed in the previous section, the input to a neural network is a sequence of integers. However, discrete spaces are difficult to operate in, so the first step of all language models is to map each token in the sequence to a vector representing the token. Working with vectors is much easier than working with discrete tokens because they facilitate statements such as

Apple is closer to pear than it is to hippopotamus because
$\|\text{embedding(apple) - embedding(pear)}\| < \|\text{embedding(apple) - embedding(hippopotamus)}\|$.

The embeddings for each token in the vocabulary are stored in an embedding matrix $\mathbf{E} \in \mathbb{R}^{\text{vocab\_size} \times \text{embedding\_size}}$. A token can be turned into an embedding by selecting the correct row from the matrix. The token that best matches an embedding predicted by a neural network can be computed by multiplying the predicted embedding with the embedding matrix. This produces a vector whose length is the vocabulary size, and where each entry can be thought of as a score for how close that token is to the prediction. Thus, the highest scoring token can be selected.

When using embedding matrices for neural language modeling, one has the choice between supplying a fixed pre-computed embedding matrix (for example, using word2vec (Mikolov et al., 2013)), or randomly initializing the embedding matrix and refining it as part of training.

2.3. Attention Mechanism

Both Gehring et al. (2017) and Vaswani et al. (2017) rely on an attention mechanisms. Attention can be thought of most intuitively in the framework of machine translation. Suppose a machine translation method is trying to translate the french sentence “Un hippopotame affamé mange.” to the English “A hungry hippo eats.” An encoder takes as input the four French words and outputs a sequence of four fixed length vectors.[1] A decoder transforms these vectors with the goal of computing a probability distribution over the first word of the English translation, following by a distribution over the second word, and so on. At the position where the decoder predicts the word ”hungry” it should naturally be paying more

[1]For the purpose of this example, we assume a simple word-level vocabulary with punctuation stripped out.

attention to the French word “affamé” than to the word “hippopotame.” An attention mechanism is a set of operations which define the input to the decoder at position i as a weighted sums of the vectors outputted by the encoder. In other words, at each decoding step, the encoder embeddings which are seen as most relevant to making a prediction at that position are given the most weight. This sort of attention is often referred to as encoder-decoder attention since it allows each input to the decoder to reference different outputs from the encoder. Thus, in the above example, for the decoder to predict “hungry” as the second output token, it can weigh the encoder's embedding in the 3rd position more heavily than the other positions.

Mathematically the attention mechanism computes a sequence of context vectors $\mathbf{c}_1, \ldots, \mathbf{c}'_T$. The context vectors depend on $\mathbf{h}^{\text{enc}}_1, \ldots, \mathbf{h}^{\text{enc}}_T$, the sequence of “hidden” vectors encoding the input sequence as well as $\mathbf{h}^{\text{dec}}_1, \ldots, \mathbf{h}^{\text{dec}}_{T'}$, the sequence of “hidden” vectors encoding the target sequence. It is worth nothing that T' does not necessarily equal T; for example, the French sentence “Je ne suis pas triste.” is one word longer than its English translation: “I am not sad.” For simplicity, the $\mathbf{h}^{\text{enc}}_i$ can be concatenated into a matrix $\mathbf{H}^{\text{enc}} \in \mathbb{R}^{d \times T}$, where d is the embedding dimension.

The context vector at position t is a weighted sum of the encoder outputs:

$$\mathbf{c}_t = \mathbf{H}^{\text{enc}} \boldsymbol{\alpha}_t \tag{2.1}$$

The vector $\boldsymbol{\alpha}_t \in \mathbb{R}^T$ is typically referred to as the attention vector. It is computed by scoring how well $\mathbf{h}^{\text{dec}}_t$ matches each of the encoder hidden states. By convention, the entries of $\boldsymbol{\alpha}_t$ are all non-negative and sum to one. A softmax is applied to ensure this property. The ith entry of $\boldsymbol{\alpha}_t$ is computed as:

$$\alpha_t[i] = \text{softmax}(\text{att\_score}(\mathbf{h}^{\text{dec}}_t, \mathbf{h}^{\text{enc}}_i) \tag{2.2}$$

Several different scoring functions have been proposed in the literature to compute the closeness of $\mathbf{h}^{\text{dec}}_t$ and $\mathbf{h}^{\text{enc}}_i$ (Neubig, 2017). These include:

$$\text{att\_score}(\mathbf{h}^{\text{dec}}_t, \mathbf{h}^{\text{enc}}_i) = \begin{cases} \mathbf{h}^{\text{dec}}_t \cdot \mathbf{h}^{\text{enc}}_i & \text{dot product} \\ \mathbf{h}^{\text{dec}}_t \mathbf{W}_a \mathbf{h}^{\text{enc}}_i & \text{bilinear function} \\ \mathbf{w}^\top_{a1} \tanh\left(\mathbf{W}_{a2}[\mathbf{h}^{\text{dec}}_t, \mathbf{h}^{\text{enc}}_i]\right) & \text{MLP} \end{cases} \tag{2.3}$$

The bilinear function and the muli-layer perceptron (MLP) contain learned weights. The dot product attention assumes the encoder and decoder hidden states are in the same space rather than learning a mapping between them. For the rest of this document, all mentions of attention refer to dot-product attention.

While this section describes attention between two different sequences of vectors, one representing the source sentence and the other representing the target sentence, Vaswani et al. (2017) take advantage of self-attention, in which the same sequence is used for both source and target ($T = T'$, and $\forall t, \mathbf{h}^{\text{dec}}_t = \mathbf{h}^{\text{enc}}_t$). The motivation for self-attention is discussed in detail in Chapter 5.

2.4. Training Objective

Typical neural language models emit $\hat{x}_t$, an embedding for the tth position in the sequence given the previous token embeddings. This can be written as

$$\hat{\mathbf{x}}_t = f(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}) \tag{2.4}$$

where f is the neural network and $\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}$ are the embeddings of the previous tokens in the sequence. To train the weights in f, we measure how how close the emitted embedding $\hat{\mathbf{x}}_t$ is to the embedding of the true next token.

As mentioned in Section 2.2, $\mathbf{E}\hat{\mathbf{x}}_t$ computes a score for each vocabulary token. A softmax transformation is then applied to normalize this vector, forcing it to look like a probability distribution summing to 1. Let Y_t be a random variable representing the vocabulary item predicted for the tth position. We then have:

$$P(Y_t = i | \mathbf{x}_1, \ldots, \mathbf{x}_{t-1}) = \frac{\exp(\mathbf{E}\hat{\mathbf{x}}_t[i])}{\sum_j \exp(\mathbf{E}\hat{\mathbf{x}}_t[j])} \tag{2.5}$$

In Equation 2.5, i and j are indexes into the vocabulary.

Finally, a log-likelihood loss is used to encourage the model to put more probability mass on the true next token than the alternatives:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(Y_t = i^* | \mathbf{x}_1, \ldots, \mathbf{x}_{t-1}) \tag{2.6}$$

where i^* is the label of the of the true next token.

The different neural approaches described in the next chapter vary in their definition of f and in how many forward passes through the network are needed in order to compute the loss. However, they all generally follow this setup.

2.5. Evaluation

Evaluating the quality of language generation systems is difficult. Many evaluation metrics vary with vocabulary size; others only correlate will with human judgement on less open-ended tasks like machine translation. Human raters are perhaps the safest way to measure quality, but they are expensive to acquire and are not without their own biases.

One of the simplest performance measures is next-step classification error rate: given a sequence of text from the test set, what fraction of the time is the model's prediction for the next token incorrect. Problematically, the baseline error rate of random guessing grows with vocabulary size.

Perplexity of a test set according to the language model is another standard measure of model quality. Perplexity can be thought of as the likelihood of a text sequence according to a language model. Let $y_1, \ldots, y_T$ represent a token sequence from the test set. Then mathematically, the perplexity of this sequence is defined as:

$$\text{PPL}(y_1, \ldots, y_T) = \exp\big(\frac{-\sum_t \log P(Y_t = y_t)}{T}\big) \tag{2.7}$$

For any given vocabulary, lower test set perplexity is known to correlate well with generation quality. However, because perplexity is vocabulary-dependent it does not allow easy comparison between models trained with different vocabularies. For each of the neural architectures, we report their test perplexity on the Google Billion Word dataset, which has a standardized vocabulary (Chelba et al., 2013).

For machine translation, BLEU score, a word overlap-based metric not dependent on vocabulary choice, is standard. We, therefore, report BLEU score on English-to-German translation for each of the architectures (Papineni et al., 2002).

Lastly, we discuss the runtime performance of each architecture by reporting the cost of a forward pass through the network. We also report the worst-case number of times an input token at position i is operated on before a prediction is made at position $i + k$ (an operation is defined as a multiplication or convolution with a weight matrix). These numbers are indications of the best theoretical performance for each architecture.

CHAPTER 3 : Recurrent Neural Networks

Before the introduction of recurrent neural networks (RNNs), neural architectures were designed to take a fixed-size vector as input. The approach worked well for images, which can be scaled or cropped to a specified size, and for tasks with a fixed, discrete set of features, such as predicting whether a person will buy a particular hippopotamus toy given their age, sex, and wealth bracket. Text and other forms of sequential data do not fit nicely into a fixed-sized vector. Recurrent neural networks were proposed as a means of performing a flexible number of computation steps proportional to the length of the input sequence instead of a fixed number in order to make a prediction.

At their simplest, recurrent neural nets consist of a recurrent unit that take as input the embedding of a single token, uses it to update the unit's hidden state, which is a fixed-size vector that stores information about the entirety of the sequence seen so far, then predicts an embedding for the next token in the sequence by applying a transformation to the hidden state. Learned weight matrices control how the hidden state is updated and how is it transformed to produce an embedding. The "recurrent" part of recurrent neural networks comes from the fact that the recurrent unit is called over and over again, once for each token in the sequence. At every step, a new hidden state and a prediction for the next position are emitted. Hence, the f from Equation 2.4 actually entails many sequential passes through the neural network. A major challenge of designing recurrent neural networks is devising strategies to prevent the hidden state from "forgetting" far-in-the-past information as it continuously gets updated.

In Graves (2013), the recurrent neural network is implemented using a long short term memory (LSTM) unit, which is a type of recurrent unit devised to ameliorate the forgetting problem and reduce instabilities leading to vanishing or exploding gradients.

3.1. Architecture

This section first describes the mechanics behind an LSTM unit, as defined in Graves (2013). It then describes how they can be composed into both unconditioned language models and encoder-decoder models.

An LSTM is a type of recurrent unit that incorporates a gated memory mechanism into the hidden state in order to better preserve long-range dependencies (Hochreiter and Schmidhuber, 1997). While a basic RNN unit keeps track of a single hidden state vector that gets updated as each token in the input sequence is processed, LSTMs augment the hidden state with a cell state vector which also gets updated. The intention of the cell state vector is to act as a gating mechanism that controls when the hidden state gets altered.

An LSTM processes an input sequence one token at a time. Let $\mathbf{x}_t$ be the embedding for the next token to be processed, and $\mathbf{h}_{t-1}$ and $\mathbf{c}_{t-1}$ represent the current hidden state and cell state respectively. All three have the same dimension. The new hidden state $\mathbf{h}_t$ and cell state $\mathbf{c}_t$ are computed using the following operatons.

$$\begin{aligned}
\text{input gate} \quad & \mathbf{i}_t = & \sigma\left(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{ci}\mathbf{c}_{t-1} + \mathbf{b}_i\right) \quad (3.1)\\
\text{forget gate} \quad & \mathbf{f}_t = & \sigma\left(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{cf}\mathbf{c}_{t-1} + \mathbf{b}_f\right) \quad (3.2)\\
\text{cell state} \quad & \mathbf{c}_t = & \mathbf{f}_t\mathbf{c}_{t-1} + \mathbf{i}_t \tanh\left(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c\right) \quad (3.3)\\
\text{output gate} \quad & \mathbf{o}_t = & \sigma\left(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{co}\mathbf{c}_{t} + \mathbf{b}_o\right) \quad (3.4)\\
\text{hidden state} \quad & \mathbf{h}_t = & \mathbf{o}_t \tanh(\mathbf{c}_t) \quad (3.5)
\end{aligned}$$

Each of the $\mathbf{W}$ and $\mathbf{b}$ are learned weight matrices and bias vectors respectively. The terms hidden size, number of hidden units, and number of LSTM units are used interchangeably to mean the dimension of the $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{c}_t$, $\mathbf{o}_t$, and $\mathbf{h}_t$ vectors. As an additional note, the LSTM formulation in Graves (2013) is a bit non-standard. It incorporates "peepholes" which allow the three gates to look at the cell state directly (Gers and Schmidhuber, 2000). The weight matrices involved in these ($\mathbf{W}_{ci}$, $\mathbf{W}_{cf}$, and $\mathbf{W}_{co}$) are diagonal. The very first step of the LSTM is a special case–when $t = 0$ the initial hidden state is usually set to the zero vector.

Some of the architectures used by Graves (2013) are deep; that is, they stack multiple layers of LSTMs together in order to make a final prediction. Let L represent the number of layers in the network. We can incorporate a new index i so that $\mathbf{h}_t^i$ represents the hidden state computed at sequence position t in the ith LSTM layer. For the very first layer ($i = 0$), the token embeddings $\mathbf{x}_1, \ldots, \mathbf{x}_T$ are used as input to the LSTM, exactly as written in Equation 3.1. For all subsequent layers ($i \geq 1$), the inputs to Equation 3.1 are the hidden states $\mathbf{h}_0^i, \ldots, \mathbf{h}_T^i$ emitted by the previous layer.

An unconditioned language model makes a prediction $\hat{\mathbf{x}}_t$ for the embedding of the next token in the sequence by taking a weighted sum of the hidden states from each layer:

$$\hat{\mathbf{x}}_t = f(\mathbf{x}_1, \ldots, \mathbf{x}_t) = \mathbf{b}_{\hat{x}} + \sum_{i=1}^{L} \mathbf{W}_{h^i\hat{x}}\mathbf{h}_{t-1}^i \tag{3.6}$$

where $\mathbf{b}_y$ and $\mathbf{W}_{h^n\hat{x}}$ are additional learned bias and weight matrices.

Encoder-decoder models are similar, except that they contain two stacks of LSTMs, one for the encoder and another for the decoder. In their simplest form, the input sequence is run through the encoder, then the final hidden states of the encoder at each layer are used at the initial hidden states (in lieu of initializing with the zero vector) for the decoder. The target sequence is then run through the decoder, and a prediction is made for each position, just as described in Equation 3.6 (Sutskever et al., 2014).

The performance of this basic approach is limited by the fact that no matter how long the source sequence is, only a fixed amount of information gets passed to the decoder. Bahdanau et al. (2014) and others extended the method by incorporating an attention mechanism between the source sequence embeddings and the target sequence embedding. At each step of the decoder, a context vector is computed as a weighted sum of the hidden states from each position of the encoder outputs. The decoder's prediction is conditioned on both this context vector and the hidden vector outputted by the previous step of the decoder.

Attention is advantageous because it eliminates the need for the encoder to encapsulate the entire source sentence into a single vector, and it eliminates the need for the decoder to carry over information about the source sentence across many steps of hidden state updates.

	WMT'14 EN-DE		Google Billion Word	
	BLEU	Model Size	PPL	Model Size
LSTM	20.9	32M	30.6	1.8B
CNN	25.16	215M	31.9	405M
Transformer	28.4	213M	23.5	460M

Table 2: Model performance on machine translation and language modeling. The Transformer language model scores come from Dai et al. (2019), which uses a bunch of extra tricks to optimize the transformer architecture for language modeling. The LSTM WMT score comes from Luong et al. (2015), who use a local-attention based sequence-to-sequence model. The LSTM GBW perplexity was taken from Jozefowicz et al. (2016). All other numbers come from the papers in which the architectures were introduced.

3.2. Complexity and Model Size

Let D be the maximum dimension of the hidden state, T be the maximum sequence length, and N be the number of layers. Then the computational complexity of a forward pass through the network is $O(NTD^2)$. At each layer, for each position in the sequence, a constant number of D-dimensional vectors are multiplied by $D \times D$ weight matrices. Moreover, T sequential operations are performed (the computation for position $t-1$ must be completed before the computation for position t). This leads to RNNs being somewhat slow to train. They also struggle to capture long-term dependencies since $O(k)$ sequential operations will have been performed on the value of the input sequence at position i before a prediction can be made at position $i+k$.

We can compute the number of trainable parameters in an LSTM model as follows. Each LSTM layer has an input dimension, with for the first layer is the vocabulary size, and an output dimension D_i. The input dimension for all layers $i > 1$ is the output dimension from the previous layer. Thus it is possible to compute the number of weights in one LSTM layer. For the ith layer, there are:

- 4 weight matrices of size $D_{i-1} \times D^i$.
- 4 weight matrices of size $D^i \times D^i$.
- 3 diagonal "peephole" weight matrices of size D^i.
- 4 bias vectors of dimension D^i.

This leads to $4D_{i-1}D_i + 4D_i(D_i+1)$ learned parameters per layer. In addition, the model has an embedding matrix with dimension vocab_size $\times$ D_0. For all of the neural networks described in the paper, the output dimensions of each layer are the same.

3.3. Experiments

Graves (2013) trains both word-level and character-level language models on the Penn Treebank (Marcus et al., 1993). The word-level language models have a vocabulary size of 10,000 (all less common words replaced with UNK), and the character-level language models have a vocabulary size of 49. A single-layer LSTM with a hidden size of 1000 is used as the architecture for both sets of experiments. However, since the embedding size is exactly the vocabulary size, the word-level LM has a total of 54M weights while the character-level LM has only 4.3M. Since the Penn Treebank is a relatively small dataset, both models overfit. The paper attempts to combat the overfitting by training some number of extra steps with normally distributed weight noise. Ultimately, they achieve 117 perplexity with the word-level model and 122 perplexity with the character level model. A more recent paper by Jozefowicz et al. (2016) applied the same LSTM architecture to the Google Billion Word Benchmark. With a model size of 1.8B parameters, they achieve perplexity if 30.6, which at the time far surpassed non-LSTM approaches that got test perplexity of 51.3 or more.

In terms of LSTM encoder-decoder models, Luong et al. (2015) experimented extensively with different types of attention mechanisms for machine translation. They found that dot-product attention yielded the best results, with a BLEU score of 20.9 for a model with 32M parameters.

3.4. Extensions

Since the publication of Graves (2013), LSTM-based language models have been used extensively across generative tasks. They were a key a key component to many of the first deep learning solutions that really "worked" on textual data. The sequence-to-sequence setting proved to be more popular than the decoder-only one. From machine translation, sequence-to-sequence models were rapidly applied to other more open-ended domains, including chat-chat dialog (Vinyals and Le, 2015), abstract summarization (Chopra et al., 2016), and image captioning (Vinyals et al., 2015). Along the way, substantial research efforts went into improving the performance of sequence-to-sequence LSTM models through architectural improvements. These included modifications to the way encoder-decoder attention was computed (Bahdanau et al., 2014; Yang et al., 2017), better vocabularies that avoided out-of-vocabulary tokens (Luong et al., 2014; Sennrich et al., 2015), and the addition of bidirectionality to the model by having one LSTM stack process the source sequence left-to-right and another process it right-to-left (Sundermeyer et al., 2014).

It was not until the introduction of non-recurrent architectures, as described in the next two chapters, that LSTM-based language models were knocked off their pedestal. Even now, LSTMs continue to be reasonably popular for smaller-scale language generation tasks.

CHAPTER 4 : Convolutional Neural Networks

When convolutional neural networks' (CNNs) were first successfully applied to task of image recognition, they revolutionized computer vision. (Krizhevsky et al., 2012)'s AlexNet paper has over 55,000 citations. By taking advantage of fast matrix operations on GPUs, CNNs were able to efficiently learn rich image representations. CNNs for computer vision alternately convolve learned kernel matrices with the input and apply simple non-linear transformations.

Interest in using CNNs for text applications has existed at least since 2014, when they were proposed for sentiment analysis and other text classification problems (Dos Santos and Gatti, 2014; Lai et al., 2015). The paper being focused on in this chapter, Gehring et al. (2017), was the most successful among several methods introduced around 2016-2017 that attempted to incorporate convolutional methods into generative language models (Bradbury et al., 2016; Kalchbrenner et al., 2016). The approach was concurrently applied to unconditioned language modeling (Dauphin et al., 2017) and machine translation (Gehring et al., 2017). Later applications included story generation, task-oriented dialog, and more (Fan et al., 2018; Schuster et al., 2018).

The main advantage of CNNs for language models is the ability to parallelize computations across time. While in a traditional LSTM, the hidden state must be updated over and over again, once for each position in the sequence, in CNN-based architectures, weight vectors called kernels can be convolved with the input embedding sequence in a single operation.

4.1. Architecture

This section first explains the convolutional block used in Dauphin et al. (2017) and Gehring et al. (2017). It then describes the the decoder-only version of the architecture as introduced in Dauphin et al. (2017), and the encoder-decoder version introduced in Gehring et al. (2017).

A convolutional block takes an input sequence of token embeddings and passes it through several convolutional layers. The output is a sequence of hidden states of the same length as the input. However, the hidden vectors do not necessarily have the same dimension as the input vectors.

For simplicity we will represent the input sequence of embeddings as a matrix $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{emb}}}$ where T is the sequence length and d_{emb} is the input embedding size. The first convolutional layer takes $\mathbf{X}$ as input and outputs a hidden state $\mathbf{H}_0 \in \mathbb{R}^{T \times d_0}$. Each subsequent layer takes as input the output from the previous layer. Mathematically, the computation of the new hidden state sequence is as follows:

$$\mathbf{H}_i(\mathbf{X}) = (\mathbf{X} * \mathbf{W}_{\text{proj}} + \mathbf{b}_{\text{proj}}) \otimes \sigma(\mathbf{X} * \mathbf{W}_{\text{gate}} + \mathbf{b}_{\text{gate}}) \tag{4.1}$$

where i is the index of the current layer; $\mathbf{W}_{\text{proj}} \in \mathbb{R}^{k \times d_{i-1} \times d_i}$ and $\mathbf{W}_{\text{gate}} \in \mathbb{R}^{k \times d_{i-1} \times d_i}$ are learned weight matrices; $\mathbf{b}_{\text{proj}} \in \mathbb{R}^{d_i}$ and $\mathbf{b}_{\text{gate}} \in \mathbb{R}^{d_i}$ are learned bias terms. The weight matrices used in convolutions are sometimes referred to as the set of kernels or filters. The result of applying a single kernel to an input is referred to as a feature map. In the above dimensions, d_i is the number of output feature maps for the ith layer and k is the kernel size.

The first convolutional term in Equation 4.1 is a linear projection of the input sequence. The second convolutional term forms a gating mechanism inspired by LSTMs. The gate consists of an element-wise sigmoid, σ, which forces each of the entries in $(\mathbf{X} * \mathbf{W}_{\text{gate}} + \mathbf{b}_{\text{gate}})$ to be between 0 and 1. Element-wise multiplication (represented by $\otimes$) between the gate and the linear projection modulates which values are passed to the next layer.

There are several problems with the architecture described thus far. First, unlike the RNN which can handle variable-length sequences, the CNN expects all of the input token sequences to be the same length. To achieve this, shorter sequences are padded and longer ones truncated. Second, in the naive implementation described above, the receptive field for the final hidden state as position i could contain positions that are in the future. When a language model is used to generate text, it does not have access to positions in the future. Dauphin et al. (2017) address this by zero-padding the input sequence with k-1 elements on the left. This prevents the feature maps from accessing future content. Lastly, a convolutional layer has no conception of position within the sequence. A convolution involves sliding the kernel across the entire sequence; at each position, the kernel is multiplied with the sequence's values within the window, and the result is added to a running sum. Information about the relative positions of tokens less than the kernel width apart is maintained, but longer-term structure is absent. The problem is resolved by adding in position embeddings. In addition to learning an embedding per token in the vocabulary, the model now also learns an embedding for each position is the sequence. The input embedding sequence is computed as the sum of the token and position embeddings.

To summarize, a convolutional block operates on a fixed sequence of input embeddings and consists of several convolutional layers.

The decoder-only architecture proposed in Dauphin et al. (2017) consists of several convolutional blocks. Residual connections add the the input of a block to its output, speeding up training. The final sequence of hidden states from the final block are multiplied by the vocabulary embedding matrix and passed through a softmax function, exactly as described in Section 2.4.

The sequence-to-sequence architecture introduced in Gehring et al. (2017) is composed of two stacks of convolutional blocks, one to transform the source sequence and one to transform the target sequence. For the encoder of the source sequence, the convolution operation *is* allowed access to positions to the right of the current position in the sequence. A multi-step attention mechanism computes how much the final hidden states outputted by the encoder should contribute to each layer of the decoder. The formulation is a bit different from 2.3, both because attention is now computed at every decoder layer, and also because the embeddings of the original sequence elements are added to the hidden states before performing the attention computation. The modified attention equations are as follows.

$$\mathbf{d}_t^l = (\mathbf{W}_{dl}\mathbf{h}_t^l + \mathbf{b}_{dl}) + \mathbf{g}_t \tag{4.2}$$

$$\boldsymbol{\alpha}_t^l[i] = \text{softmax}(\text{att\_score}(\mathbf{d}_t^l, \mathbf{z}_i^L) \tag{4.3}$$

$$\mathbf{c}_t^l = \sum_{i=1} T\boldsymbol{\alpha}_t^l[i](\mathbf{z}_i^L + \mathbf{e}_i) \tag{4.4}$$

where $\mathbf{h}_t^l$ is the hidden vector at the tth position of the lth layer of the decoder; $\mathbf{z}_t^l$ is the hidden vector at the tth position of the lth layer of the encoder; $\mathbf{e}_t$ is the the embedding of the tth element in the input sequence; $\mathbf{g}_t$ is the embedding of the tth element in the target sequence; L is the number of layers; T is the input sequence length; and $\mathbf{W}_{dl}$ and $\mathbf{b}_{dl}$ are learned weight and bias.

4.2. Complexity and Model Size

Let D be the maximum dimension of any hidden states, T be the maximum sequence length, L be the number of layers, and K be the maximum kernel size. Then the computational complexity of a forward pass through the network is $O(LTKD^2)$. This comes from the fact that a convolutional layer involves moving a sliding window across the input sequence, and for each of the T positions in the sequence, K multiplications with the weight matrix are performed, costing D^2 each.

Convolutional neural nets are better at capturing long-term dependencies than recurrent neural networks. Although the outputs from the first layer only have access to a receptive field of size K, the size of the receptive field grows with the number of layers. The output of the Lth layer has a receptive field of $((K-1)L+1)$. This means that when making a prediction at position i, the furthest-in-the-past token embedding the model has access to is at position $i - \frac{1}{2}((K-1)L+1)$ and has been operated on L times.

We can compute the number of trainable parameters in the ith convolution layer as a function of d_{i-1}, the number of input feature maps; d_i, the number of output feature maps; and k, the kernel size. Each layer in the model has two weight matrices of size $k \times d_{i-1} \times d_i$. In addition the embedding lookup table has dimension vocab_size $\times$ d_0,.

4.3. Experiments

Dauphin et al. (2017) conduct experiments in the decoder-only language modeling setting, showing results on the Google Billion Word and Wiki-103 dataset. On the Billion Word benchmark, which contains sentences from news articles, they achieved competitive but not state-of-the-art results, with a best perplexity of 31.9. On Wiki-103, which contains much longer article-sized text sequences, they did reach state-of-the-art with a perplexity of 37.2.

The model for the Billion Word benchmark has a total of 405M parameters, coming from 14 convolutional blocks with kernel sizes ranging from 1 to 6 and feature map dimensions varying from 128 to 4096. The paper finds that using a tiny kernel size of 1 for the final layer has a bottleneck effect that improves performance. Interestingly, taking advantage of the CNNs' better capacity for capturing long-range-dependencies relative to RNNs did not actually seem to help much. The authors find that there were diminishing returns to increasing the receptive field beyond 40 tokens or so (model size being held equal). However, without model size being held constant, increasing the number of layers results in lower perplexity.

Gehring et al. (2017) show machine translation and abstractive summarization results for the sequence-to-sequence version of the architecture. For machine translation, the paper uses a model with 15 layers for the encoder and another 15 for the decoder. The feature map dimensions are between 512 and 2048 and the kernel is uniformly 3. This leads to a total model size of about 475M parameters.

At the time of publication, all machine translation results were close to (or in the case of Romanian, better) than state-of-the-art at the time. On, WMT'16 English-Romanian, the method achieved 30.02 BLEU, WMT'14 English-German 25.16 BLEU, and on WMT'14 English-French 40.51 BLEU. Surprisingly, the authors found that the position embeddings only improved translation quality a little bit, suggesting that explicit position information, one of the motivators behind recurrent architectures, is less important than thought.

4.4. Extensions

Convolutional neural networks for text as defined by Gehring et al. (2017) and Dauphin et al. (2017) have not seen a huge amount of extensions, largely because Transformers, covered in the next chapter, came out contemporaneously and ended up being more popular. However, one particularly interesting extension is in Fan et al. (2018). The authors use the encoder-decoder model designed for machine translation to instead generate stories given a writing prompt. Their method builds off the originally proposed architecture by making the attention mechanism multi-headed and multi-scale (each attention head sees a different downscale version of the input sequence). Since the story dataset is relatively small, they develop a fusion method that allows the attention mechanisms of the main encoder-decoder model to attend to the outputs from an auxiliary convolutional language model that was pre-trained on a large amount of out-of-domain text.

CHAPTER 5 : Attention-based Neural Networks

Attention-based architectures drop both the concepts of recurrence and convolutional layers. They instead contain only attention mechanisms and feed-forward layers. The development of attention models by Vaswani et al. (2017) occurred concurrently with the work on CNNs described in Chapter 4. The authors gave their attention-only architecture a fancy name:

the Transformer.

The CNN and Transformer architectures were conceived out of the same motivation: recurrence is computationally expensive and struggles to capture long-term dependencies. The encoder-decoder versions of the Transformers, CNNs, and RNNs all support an attention mechanism between the output embeddings of the source encoder and the predictions of the decoder. This allows the prediction at a given position to access all positions of the source sequeunce. Transformers go a step further by adding self-attention, a mechanism that allows the embeddings outputted by a given layer to attend to all of the embeddings outputted by the previous layer. This leads to a couple of advantages over the CNNs. First, while the receptive field for CNNs grows linearly with the number of layers, attention-based layers always have access to the entirety of the sequence. Second, for tasks like machine translation where the sequence lengths tend to be smaller than the embedding size, Transformers have better computational complexity.

5.1. Architecture

The Transformer was originally proposed as an encoder-decoder model. The encoder and decoder each consist of a sequence of attention layers. This section describes the construction of the attention layers in the encoder and in the the decoder. It then explains the attention mechanism used by the Transformer.

An attention layer takes as input a sequence of embeddings and outputs a new sequence of the same length. Those outputs are used as the input to the subsequent layer. Each operation in the attention layer revises the embeddings. While the encoder only contains self-attention mechanisms, the decoder has both encoder-decoder attention and self-attention mechanisms.

In the encoder, the first step of an attention layer is a self-attention mechanism. A residual connection then adds the input to its output, and the result is normalized. Next, a feed-forward fully-connected layer, also surrounded by a residual connection, transforms the sequence of embeddings. Let $\mathbf{H}_{i-1}^{\text{enc}} \in \mathbb{R}^{T \times d_{\text{model}}}$ be the sequence of embeddings (concatenated into a matrix) that are the input to the ith layer of the encoder. For the first layer, these are just the token embeddings of the input sequence. Mathematically, we can write the encoder layer as:

$$\mathbf{A} = \text{LayerNorm}(\text{MultiHeadAtt}(\mathbf{H}_{i-1}^{\text{enc}}, \mathbf{H}_{i-1}^{\text{enc}}, \mathbf{H}_{i-1}^{\text{enc}}) + \mathbf{H}_{i-1}^{\text{enc}}) \tag{5.1}$$

$$\mathbf{H}_{i}^{\text{enc}} = \text{FFN}(\mathbf{A}) + \mathbf{A} \tag{5.2}$$

A layer in the decoder is similar to a layer in the encoder, except that in inserts an encoder-decoder attention mechanism in between the self-attention mechanism and the feed-forward layer. Let $\mathbf{H}_{i-1}^{\text{dec}} \in \mathbb{R}^{T' \times d_{\text{model}}}$ be the sequence of decoder embeddings (concatenated into a matrix) that are the input to the ith layer of the decoder. For the first layer, these are just the token embeddings of the target sequence. Then we can write the decoder as:

$$\mathbf{A} = \text{LayerNorm}(\text{MultiHeadAtt}(\mathbf{H}_{i-1}^{\text{dec}}, \mathbf{H}_{i-1}^{\text{dec}}, \mathbf{H}_{i-1}^{\text{dec}}) + \mathbf{H}_{i-1}^{\text{enc}}) \tag{5.3}$$

$$\mathbf{A}' = \text{LayerNorm}(\text{MultiHeadAtt}(\mathbf{A}, \mathbf{H}_{i}^{\text{enc}}, \mathbf{H}_{i}^{\text{enc}}) + \mathbf{A}) \tag{5.4}$$

$$\mathbf{H}_{i}^{\text{dec}} = \text{FFN}(\mathbf{A}') + \mathbf{A}' \tag{5.5}$$

FFN is a two-layer feed-forward fully connected network with a ReLU activation between the layers. It is applied to the embedding at each position separately and independently.

LayerNorm is function that normalizes the mean and standard deviation of the output features, and MultiHeadAtt is the attention mechanism.

The attention mechanism in a Transformer is an extension of the dot-product attention described in Section 2.3. The paper defines attention in terms of three matrices, the key $\mathbf{K} \in \mathbb{R}^{T \times d_k}$, the value $\mathbf{V} \in \mathbb{R}^{T \times d_k}$, and the query $Q \in \mathbb{R}^{T' \times d_q}$, where T is the length of the sequence being attended to, T' is the length of the target sequence, and d_k and d_q are the embedding dimensions for the keys and queries respectively. The values must have the same embedding dimension as the keys. Then the attention function is defined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^T)}{\sqrt{d_k}}\right)\mathbf{V} \tag{5.6}$$

Unlike in the definition from Section 2.3, the dot products are scaled by the dimension of the key vectors in order to combat large-magnitude dot products pushing the softmax into regions with tiny gradients.

Prior to the Transformer, most sequence-to-sequence models applied the attention function a single time, resulting in one set of weightings of the value (input) embeddings (Bahdanau et al., 2014; Gehring et al., 2017). Vaswani et al. (2017) found it beneficial to concatenate together the outputs of multiple attention functions. The paper calls this multihead attention, and it is defined as follows:

$$\text{MultiHeadAtt}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})\mathbf{W}^O \tag{5.7}$$

$$\text{where head}_\text{i} = \text{Attention}(\mathbf{QW}_i^\mathbf{Q}, \mathbf{KW}_i^\mathbf{K}, \mathbf{VW}_i^\mathbf{V}) \tag{5.8}$$

The $\mathbf{W}$ are learned weight matrices that are different for each head.

For the self-attention mechanisms, $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are all the same, since the sequence is attending to itself. For the encoder-decoder attention mechanism, the values and keys both come from the encoder. The keys come form the decoder.

Any number of attention layers can be stacked on top of each other in order to form an encoder or decoder network. Like with the CNN, position embeddings must be added to the input token embeddings as the architecture itself does not capture relative positions of tokens in any way. In addition, masking must be applied to the inputs to the decoder to prevent the model from attending to positions in the future relative to the current position. A decoder-only architecture looks identical to the encoder architecture except that it has masking of future positions.

5.2. Complexity and Model Size

Let D be the dimension of the embeddings outputted by each layer, T be the maximum sequence length, and L be total number of layers in the model. Then the computational complexity of a forward pass through the network is $O(LT^2D)$. At each layer, a constant number of attention mechanisms, each taking $O(T^2D)$, are executed.

Since each attention mechanism has access to the embeddings at all positions of the input sequence, when the model makes a prediction at position i, the sequence value at the first position of the sequence is just as accessible as the one at position $i - 1$.

5.3. Experiments

All of the experiments in the original paper are for either machine translation or constituency parsing. The paper's "big" architecture has about 213M parameters, less than the CNN described in Section 5. It contains 6 layers each in the encoder and decoder; layer output embeddings are all 1024 dimensions; and the multi-head attention mechanism has 16 heads. Despite a smaller model size than the comparable CNN, the Transformer "big" achieves a BLEU score of 28.4 on WMT'15 English-German, which at the time was state-of-the-art.

A followup paper experiments with using the Transformer model for language modeling (Dai et al., 2019). It modifies the attention mechanism to support dependencies beyond a fixed-length windows and improves upon the position embeddings. Their model with 460M parameters achieves a perplexity of 23.5 on the Google Billion Word Benchmark.

5.4. Extensions

Transformers have largely replaced LSTM-based recurrent models as the de facto architecture researchers use for language generation tasks. In nearly every task where LSTMs were previously state-of-the-art, Transformers have been able to beat them. The architecture is continually being modified and improved. For example, Shaw et al. (2018) propose a relative attention mechanism that enables training on longer sequences. (Huang et al., 2018) further modify the relative attention mechanism so that very long sequences of music can be generated efficiently. Raffel et al. (2019) introduce better architectural support for distributed training which allows for larger model sizes.

Perhaps more interesting than the architectural tweaks are the variety of applications that Transformers have been used successfully in. Open-ended text generation has long struggled with a fluency problem. Consistently generating grammatical paragraphs of text was difficult without handcrafted grammatical rules. Transformers largely solved this problem. Zellers et al. (2019) and Radford et al. (2019) have shown that Transformers can be used to generate extremely convincing-sounding text on topics ranging from new articles to Lord of the Rings fan fiction. Table 3 shows generated text from the original LSTM language modeling paper (Graves, 2013) and from a Transformer paper (Liu et al., 2018). Though both models were trained on Wikipedia, the samples from the Transformer model are vastly more fluent and coherent. However, long-term coherence and factualness remain ongoing challenging with even state-of-the-art neural language models.

CHAPTER 6 : Conclusion and Directions for Further Research

This survey considers three architectures for training generative language models. LSTMs and Transformers, especially, revolutionized research on text generation, resulting in massive performance improvements on both well-defined tasks such as machine translation and more open-ended tasks such as chit-chat dialog and story generation. Though not discussed in this survey, these approaches also yielded enormous improvements on natural language inference tasks, where a neural network needs to understand natural text in order to be able to answer conceptual questions.

Despite the tremendous success of neural language modelling in recent years, there remain several challenges to effectively employing these models for real-world applications and reproducible research. In addition, evaluation of performance on generative tasks and fair comparison between methods remain difficult. We conclude with a discussion of the ongoing

LSTM	The '''Rebellion''' (''Hyerodent'') is [[literal]], related mildly older than old half sister, the music, and morrow been much more propellent. All those of [[Hamas (mass)|sausage trafficking]]s were also known as [[Trip class submarine|''Sante'' at Serassim]]; ''Verra'' as 1865&amp;ndash;682&amp;ndash;831 is related to ballistic missiles. While she viewed it friend of Halla equatorial weapons ofTuscany, in [[France]], from vaccine homes to "individual", among [[slavery|slaves]] (such as artistual selling of factories were renamed English habit of twelve years.) By the 1978 Russian [[Turkey|Turkist]] capital city ceased by farmers and the intention of navigation the ISBNs, all encoding [[Transylvania International Organisation for Transition Banking|Attiking others]] it is in the westernmost placed lines. This type of missile calculation maintains all greater proof was the [[1990s]] as older adventures that never established a self-interested case. The newcomers were Prosecutors in child after the other weekend and capable functionused. Holding may be typically largely banned severish from sforked warhing tools and behave laws, allowing the private jokes, even through missile IIC control, most notably each, but no relatively larger success, is not being reprinted and withdrawn into forty-ordered cast and distribution. Besides these markets (notably a son of humor).
Tranformer	== Wings Over Kansas == wings over kansas is the best aviation history website i have encountered on the world wide web. it is informative, entertaining provides ever changing content, and is populated with the true voices of the mainstream aviation community . there is no better place to see where aviation has been and where it is going . centered in the midst of the greatest producer of aircraft in the world , wings over kansas reflects that aviation community to the rest of the world . == Wings Over Kansas History ==wings over kansas was established with the mission of becoming the number one online kansas aviation resource showcasing the pioneers , educators , newsmakers , manufacturers , pilots and craftsmen who have made kansas the world center for aviation production. wings over kansas was established with the mission of becoming the number one online kansas aviation resource showcasing the pioneers , educators , newsmakers , manufacturers , pilots and craftsmen ,who have made kansas the world center for aviation production . wings over kansas has been recognized by mcgraw - hill as one of the top 500 best aviation websites with visitors from over 225 countries. wings over kansas offers a unique perspective on the role of wichita and kansas in the history and development of international aviation . the featured menu offers over 1 , 500 pages on aviation news , history , education , photos , videos , careers. pioneers , quizzes and learn - to - fly categories . in addition , the special subjects section offers further aviation content pages to visit.

Table 3: Generations from language models trained on Wikipedia. The first comes from Graves (2013) and the second from Liu et al. (2018). The second is clearly higher quality.

problems that confront text generation and directions for future research.

6.1. Evaluation Metrics

In Section 2.5, the difficulty of coming up with reasonable automatic evaluation metrics for text generation systems was discussed. This is especially tricky for open-ended tasks where there are many reasonable choices the generative model can make. Human evaluation is often seen to be more reliable than automatic evaluation metrics, but there is little consensus in how to conduct human evaluation experiments. Ippolito et al. (2019) find that while trained human raters achieve over 70% accuracy at predicting whether a text excerpt is human-written or machine-generated, Amazon Mechanical Turk workers perform no better than random guessing. Better methods and shared tasks for evaluating open-ended text generation will be critical for the field's growth.

6.2. Difficulty of Meaningful Comparison

Even for tasks with standard datasets and evaluation metrics, such as machine translation, fair comparison of methods can be difficult. There are many small decisions which go into training and testing a neural language model, decisions I barely touch upon in this survey. These including the learning rate, optimizer, use of batch and other types of normalization, weight initialization strategy, vocabulary size, dropout rate, sampling strategy at test time, and more. Each of these can have a significant impact on performance and are rarely standardized when reporting performance numbers. Due to the cost (both computational and in human hours) of reproducing work, most papers simply paste the evaluation numbers from previous papers into their tables.

Model size is also critically important to fair comparison. For example, Table 2 shows that the Transformer achieves lower perplexity than the CNN on the Google Billion Word benchmark. However, the latter has fewer trained weights than the former. Generally larger models yield better performance than smaller ones. While some papers, such as Vaswani et al. (2017), are very good about listing their model sizes, others leave computation of model size as an exercise to the reader Dauphin et al. (2017).

Finally, it is worth noting that nearly all recent advancements in neural language modeling, including both the CNN and Transformer discussed in this survey, have come out of industry research labs (Kalchbrenner et al., 2016; Vaswani et al., 2017; Gehring et al., 2017; Dai et al., 2019). Increasingly, bigger model sizes and larger hyperparameter searches are necessary to show a method beats state-of-the-art, but this level of experimentation is beyond the reach of most academic research groups. Important research directions include the standardization of comparison procedures and methods for reducing model size without too much of a performance hit.

6.3. Datasets and Transfer Learning

Training a neural language model to generation fluent text requires a tremendous amount of training data. Most large datasets are scraped from the web; sources include news articles, Twitter and Reddit posts, copyright-free books from Project Gutenberg, and more. While these datasets contain many examples of human-written text, they are often not well-matched for the downstream tasks we are actually interested in. Often large language models are pre-trained on one of these large datasets then fine-tuned on a smaller more targeted dataset, such as text-adventure games[1] or poetry[2]. However, it is very easy to overfit when finetuning, especially when the finetuning dataset is very small.

Several exciting alternatives to finetuning have been proposed, including training an auxiliary model with its own set of weights on the smaller dataset (Fan et al., 2018), and using a trained classifier to guide the pre-trained language model's generations toward a particular class prediction (Dathathri et al., 2019). Especially with the rising computational cost of training neural language models from scratch, more research is needed in improving transfer learning procedures.

6.4. Bias and Common-Sense Reasoning

BIBLIOGRAPHY

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

L. Barrault, O. Bojar, M. R. Costa-jussa, C. Federmann, M. Fishel, Y. Graham, B. Haddow, M. Huck, P. Koehn, S. Malmasi, C. Monz, M. Muller, S. Pal, M. Post, and M. Zampieri. Findings of the 2019 conference on machine translation (wmt19). In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pages 1–61, Florence, Italy, August 2019. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W19-5301`.

[1]`https://aidungeon.io/`

[2]`https://www.gwern.net/GPT-2`

J. Bradbury, S. Merity, C. Xiong, and R. Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

S. Chopra, M. Auli, and A. M. Rush. Abstractive sentence summarization with attentive recurrent neural networks. pages 93–98, June 2016. doi: 10.18653/v1/N16-1012. URL `https://www.aclweb.org/anthology/N16-1012`.

Z. Dai, Z. Yang, Y. Yang, W. W. Cohen, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

S. Dathathri, A. Madotto, J. Lan, J. Hung, E. Frank, P. Molino, J. Yosinski, and R. Liu. Plug and play language models: a simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164*, 2019.

Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org, 2017.

C. Dos Santos and M. Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, 2014.

A. Fan, M. Lewis, and Y. Dauphin. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*, 2018.

J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org, 2017.

F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck. Music transformer. *arXiv preprint arXiv:1809.04281*, 2018.

D. Ippolito, D. Duckworth, C. Callison-Burch, and D. Eck. Human and automatic detection of generated text. *arXiv preprint arXiv:1911.00650*, 2019.

S. Janarthanam and O. Lemon. Learning to adapt to unknown users: referring expression generation in spoken dialogue systems. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 69–78. Association for Computational Linguistics, 2010.

F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1): S63–S63, 1977.

R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

N. Kalchbrenner, L. Espeholt, K. Simonyan, A. v. d. Oord, A. Graves, and K. Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

R. Kondadadi, B. Howald, and F. Schilder. A statistical nlg framework for aggregated planning and realization. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1406–1415, 2013.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

P. J. Liu, M. Saleh, E. Pot, B. Goodrich, R. Sepassi, L. Kaiser, and N. Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018.

M.-T. Luong, I. Sutskever, Q. V. Le, O. Vinyals, and W. Zaremba. Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*, 2014.

M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

M. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

G. Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*, 2017.

K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

E. Reiter and R. Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87, 1997.

S. Schuster, S. Gupta, R. Shah, and M. Lewis. Cross-lingual transfer learning for multilingual task oriented dialog. *arXiv preprint arXiv:1810.13327*, 2018.

R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, Aug. 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL `https://www.aclweb.org/anthology/P16-1162`.

P. Shaw, J. Uszkoreit, and A. Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

M. Sundermeyer, T. Alkhouli, J. Wuebker, and H. Ney. Translation modeling with bidirectional recurrent neural networks. pages 14–25, Oct. 2014. doi: 10.3115/v1/D14-1003. URL `https://www.aclweb.org/anthology/D14-1003`.

I. Sutskever, O. Vinyals, and Q. Le. Sequence to sequence learning with neural networks. *Advances in NIPS*, 2014.

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

O. Vinyals and Q. Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.

O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

Z. Yang, Z. Hu, Y. Deng, C. Dyer, and A. Smola. Neural machine translation with recurrent attention modeling. pages 383–387, Apr. 2017. URL `https://www.aclweb.org/anthology/E17-2061`.

R. Zellers, A. Holtzman, H. Rashkin, Y. Bisk, A. Farhadi, F. Roesner, and Y. Choi. Defending against neural fake news. *arXiv preprint arXiv:1905.12616*, 2019.