**Text Generation**

**Abstract**

This project involves developing a text generation model using Long Short-Term Memory (LSTM) networks. The model is trained on a dataset of article headlines to generate coherent and contextually relevant text sequences. This report details the entire process, from data collection and preprocessing to model training and text generation.

**Table of Contents**

* Introduction
* Limitations**:** What will *not* be covered
* Setting
* Encoder-decoder models
* Decoding overview
* Literature Review
* Training overview
* Methodology
* Implementation
* Diagnostics
* Common issues
* Deployment
* Results and Discussion
* Conclusion
* References
* Appendices

**Introduction**

* **Background**

Text generation is a subset of natural language processing (NLP) that focuses on generating coherent and contextually relevant text based on a given input. Applications include chatbots, automated content creation, and translation services.

* **Objectives**

The main objective of this project is to build an LSTM-based model capable of generating meaningful text sequences when provided with a seed text.

* **Scope**

The project is limited to generating text based on article headlines. The quality and relevance of generated text will be evaluated, but the project will not delve into fine-tuning for specific use cases.

**Focus of this guide**

This guide focuses on advice for training and decoding of neural encoder-decoder models (with an attention mechanism) for text generation tasks. Roughly speaking, the source and target are

assumed to be on the order of dozens of tokens. The primary focus of the guide is on the decoding procedure.

**Limitations: What will *not* be covered**

Before continuing, we describe what this guide will *not* cover, as well as some of the current limita- tions of neural text generation models. This guide does not consider:

* Natural language understanding and semantics. While impressive work has been done in learning word embeddings [Mikolov et al., 2013, Pennington et al., 2014], the goal of learning “thought vectors” for sentences has remained more elusive [Kiros et al., 2015]. We also do not consider sequence labeling or classification tasks.
* How to capture long-term dependencies (beyond a brief discussion of attention) or maintain global coherence. This remains a challenge due to the curse of dimensionality as well as neural networks failing to learn more abstract concepts from the predominant next-step prediction training objective.
* How to interface models with a knowledge base, or other structured data that cannot be supplied in a short piece of text. Some recent work has used pointer mechanisms towards this end [Vinyals et al., 2015].
* Consequently, while we focus on natural language, to be precise, this guide does not cover *natural language generation* (NLG), which entails generating documents or longer descriptions from structured data. The primary focus is on tasks where the target is a single sentence— hence the term “text generation” as opposed to “language generation”.

Although the field is evolving quickly, there are still many tasks where older rule or template- based systems are the only reasonable option. Consider, for example, the seminal work on ELIZA [Weizen- baum, 1966]—a computer program intended to emulate a psychotherapist—that was based on pat- tern matching and rules for generating responses. In general, neural-based systems are unable

perform the dialogue state management required for such systems. Or consider the task of gener- ating a summary of a large collection of documents. With the soft attention mechanisms used in neural systems, there is currently no direct way to condition on such an amount of text.

**Setting**

We consider modeling discrete sequences of text tokens. Given a sequence *U* = (*u*1*, u*2*, . . . , uS*) over the vocabulary *V* , we seek to model

Y

*S*

*p*(*U* ) = *p*(*ut*|*u<t*) (1)

*t*=1

where *u<t* denotes *u*1*, u*2*, . . . , ut*−1, and equality follows from the chain rule of probability. Depending on how we choose to tokenize the text, the vocabulary can contain the set of characters, word- pieces/byte-pairs, words, or some other unit. For the tasks we consider in this paper, we divide the sequence *U* into an input or *source* sequence *X* (that is always provided in full) and an output or *target* sequence *Y* . For example, for machine translation tasks *X* might be a sentence in English and *Y* the translated sentence in Chinese. In this case, we model

Y

*T*

*p*(*Y* |*X*) = *p*(*yt*|*X, y<t*) (2)

*t*=1

Note that this is a generalization of (1); we consider *p*(*Y* |*X*) from here on. Besides machine translation, this also encompasses many other tasks in natural language processing—see Table 1 for more examples of sequence transduction tasks.

Beyond the tasks described in the first half of Table 1, many of the techniques described in this paper also extend to tasks at the intersection of text and other modalities. For example, in

|  |  |  |
| --- | --- | --- |
| **Task** | *X* **(example)** | *Y* **(example)** |
| language modeling | none (empty sequence) | tokens from news corpus |
| machine translation | source sequence in English | target sequence in French |
| grammar correction | noisy, ungrammatical sentence | corrected sentence |
| summarization | body of news article | headline of article |
| dialogue | conversation history | next response in turn |
| *Related tasks (may be outside scope of this guide)* | | |
| speech transcription | audio / speech features | text transcript |
| image captioning | image | caption describing image |
| question answering | supporting text + knowledge base + question | answer |

**Encoder-decoder models**

Encoder-decoder models, also referred to as sequence-to-sequence models, were developed for ma- chine translation and have rapidly exceeded the performance of prior systems depite having com- paratively simple architectures, trained end-to-end to map source directly to target.

Before neural network-based approaches, count-based methods [Chen and Goodman, 1996] and methods involving learning phrase pair probabilities were used for language modeling and translation. Prior to more recent encoder-decoder models, *feed-forward fully-connected neural networks* wereshown to work well for language modeling. Such models simply stack affine matrix transforms followed by nonlinearities to the input and each following hidden layer [Bengio et al., 2003]. However, these networks have fallen out of favor for modeling sequence data, as they require defining a fixed context length when modeling *p*(*yt*|*y<t*), do not use parameter sharing across timesteps, and have been surpassed in performance by subsequent architectures.

At the time of this writing, several different architectures have demonstrated strong results.

* *Recurrent neural networks* (RNNs) use shared parameter matrices across different time steps and combine the input at the current time step with the previous hidden state summarizing all previous time steps [Graves, 2013, Mikolov et al., 2010, Sutskever et al., 2014, Cho et al., 2014]. Many different gating mechanisms have been developed for such architectures to try and ease optimization [Hochreiter and Schmidhuber, 1997, Cho et al., 2014].
* *Convolutional neural networks* (CNNs). Convolutions with kernels reused across timesteps can also be used with masking to avoid peeking ahead at future inputs during training (see Section 2.3 for an overview of the training procedure) [Kalchbrenner et al., 2016, Gehring et al., 2017]. Using convolutions has the benefit during training of parallelizing across the time dimension instead of computing the next hidden state one step at a time.
* Both recurrent and convolutional networks for modeling sequences typically rely on a per timestep *attention* mechanism [Bahdanau et al., 2014] that acts as a shortcut connection between the target output prediction and the relevant source input hidden states. At a high- level, at decoder timestep *i*, the decoder representation *Di* is used to compute a weight *αij* for each encoder representation *Ej*. For example, this could be done by using the dot product *Di*T*Ej* as the logits before applying the softmax function. Hence

Σ

*S*

*αij* = exp(*Di*T*Ej*)*/* exp(*Di*T*Ek*)

*k*=1 *j*=1

The weighted representation Σ*S αijEj* is then fed into the decoder along with *X* and *y<i*.

More recent models which rely purely on attention mechanisms with masking have also been shown to obtain as good or better results as RNN or CNN-based models [Vaswani et al., 2017]. We describe the attention mechanism in more detail in Section 2.5.

Unless otherwise indicated, the advice in this guide is intended to be agnostic of the model archi- tecture, as long as the following two conditions hold:

* The model performs next-step prediction of the next target conditioned on the source and previous targets, i.e. it models *p*(*yt*|*X, y<t*).
* The model uses an attention mechanism (resulting in an attention matrix *A*), which eases training, is simple to implement and reasonably efficient to compute in most cases, and has become a standard component of encoder-decoder models.1

**Literature Review**

* **Historical Background**

Early text generation methods include n-gram models and Markov chains. Recent advancements leverage neural networks, particularly RNNs and LSTMs, which have shown significant improvements in generating contextually relevant text.

**Example**

N-gram models: These models predict the next word in a sequence by considering the previous 'n' words. For instance, a bigram model (n=2) would generate text by looking at one preceding word.

Markov chains: These probabilistic models generate text by considering the current state (word) and transitioning to the next state based on probability distributions derived from training data.

2.2 Current State of the Art

Modern techniques such as Transformers (e.g., GPT-3) have set new benchmarks in text generation. However, LSTMs remain popular for their effectiveness and relative simplicity.

**Example**

Transformers: Models like GPT-3 use attention mechanisms to consider the importance of all previous words when generating the next word, rather than relying on sequential processing like RNNs and LSTMs.

* **Gaps in Existing Research**

Many models struggle with generating long, coherent texts. This project aims to explore the effectiveness of LSTMs in generating concise text sequences like headlines.

**Training overview**

During training, we optimize over the model parameters *θ* the sequence cross-entropy loss

Σ

*T*

*l*(*θ*) = − log *p*(*yt*|*X, y<t*; *θ*)*.* (3)

*t*=1

thus maximizing the log-likelihood of the training data. Previous ground truth inputs are given to the model when predicting the next index in the sequence, a training method sometimes referred to (unfortunately) as *teacher forcing*. Due to the inability to fit current datasets into memory as well as for faster convergence, gradient updates are computed on minibatches of training sentences. *Stochastic gradient descent* (SGD) as well as optimizers such as Adam [Kingma and Ba, 2014] have been shown to work well empirically.

Recent research has also explored other methods for training sequence models, such as by using reinforcement learning or a separate adversarial loss [Goodfellow et al., 2014, Li et al., 2016b, 2017, Bahdanau et al., 2016, Arjovsky et al., 2017]. As of this writing, however, the aforementioned training method is the primary workhorse for training such models.

**Decoding overview**

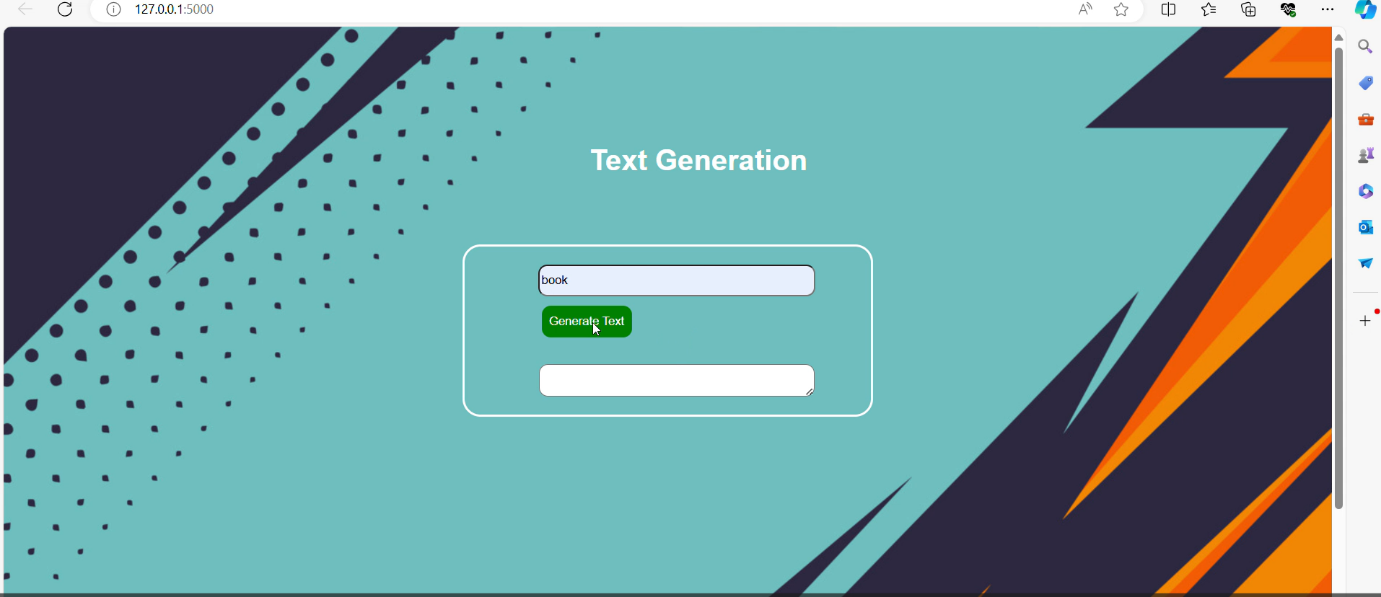
During decoding, we are given the source sequence *X* and seek to generate the target *Y*ˆthat maximizes some scoring function *s*(*Y*ˆ).2 In *greedy decoding*, we simply take the argmax over the softmax output distribution for each timestep, then feed that as the input for the next timestep. Thus at any timestep we only have a single hypothesis. Although greedy decoding can work surprisingly well, note that it often does *not* result in the most probable output hypothesis, since there may be a path that is more probable overall despite including an output which was not the argmax (this also holds true for most scoring functions we may choose).

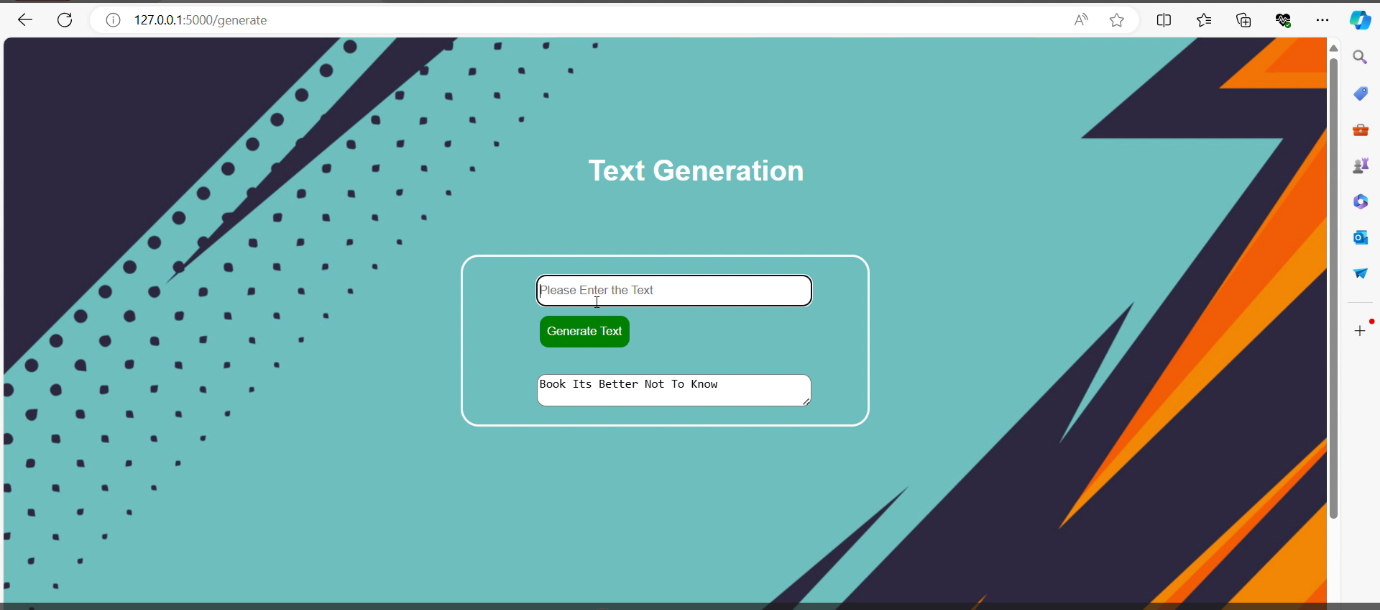
Since it’s usually intractable to consider every possible *Y*ˆ due to the branching factor and number of timesteps, we instead perform *beam search*, where we iteratively expand each hypotheses one token at a time, and at the end of every search iteration we only keep the *k* best (in terms of *s*(·)) hypotheses, where *k* is the *beam width* or *beam size*. Here’s the full beam search procedure, in more detail:

* We begin the beam procedure with the start-of-sequence token, ⟨sos⟩. Thus our set of hypoth- esis H = {[⟨sos⟩]} consisting of the single hypothesis *H* = [⟨sos⟩], a list with only the start token.
* Repeat, for *t* = 1*,* 2*, . . . , T*max:
* Repeat, for *H* ∈ H:
* Repeat, for every *u* in the vocabulary *V* with probability *put* = *p*(*u*|*X, H*):
* Add the hypothesis *H*new = [⟨sos⟩*, h*1*, . . . , ht*−1*, u*] to H.
* Compute and cache *s*(*H*new). For example, if *s*(·) simply computes the cumulative log-probability of a hypothesis, we have *s*(*H*new) = *s*(*H*) + log *put*.

2The scoring function may also take *X* or other tensors as input, but for simplicity we consider just *Y*ˆ.

**How The Website Works :**

****

****

**Methodology**

* **Data Collection**

Article headlines are collected from CSV files stored in a specified directory. The headlines are then filtered to remove any entries marked as "Unknown".

**code:**

import os

import pandas as pd

curr\_dir = r"C:\Users\swetha\Downloads\ps6\text generation/"

all\_headlines = []

for filename in os.listdir(curr\_dir):

if 'Articles' in filename:

article\_df = pd.read\_csv(curr\_dir + filename)

all\_headlines.extend(list(article\_df.headline.values))

break

all\_headlines = [h for h in all\_headlines if h != "Unknown"]

* **Data Preprocessing**

The text is cleaned by removing punctuation, converting to lowercase, and ensuring it is in ASCII format.

With increasingly advanced libraries for building computation graphs and performing automatic differentiation, a more significant portion of the software development process is devoted to data preparation.4

Broadly speaking, once the raw data has been collected, there remains cleaning, tokenization, and splitting into training and test data. An important consideration during cleaning is setting the character encoding—for example ASCII or UTF-8—for which libraries such as Python’s unidecode can save a lot of time. After cleaning comes the less easily-specified tasks of splitting the text into sentences and tokenization. At present, we recommend Stanford CoreNLP5 for extensive options and better handling of sentence and word boundaries than other available libraries.

An alternative to performing tokenization (and later detokenization) is to avoid it altogether. Instead of working at the word level, we can instead operate at the character level or use intermediate subword units [Sennrich et al., 2015]. Such models result in longer sequences overall, but empirically subword models tend to provide a good trade-off between sequence length (speed) and handling of rare words [Wu et al., 2016]. Section 5.2.1 discusses the benefits of subword models in more detail. Ultimately, if using word tokens, it’s important to use a consistent tokenization scheme for all inputs to the system—this includes handling of contractions, punctuation marks such as quotes and hyphens, periods denoting abbreviations (nonbreaking prefixes) vs. sentence boundaries, character escaping, etc.

**code:**

import string

def clean\_text(txt):

txt = "".join(v for v in txt if v not in string.punctuation).lower()

txt = txt.encode("utf8").decode("ascii",'ignore')

return txt

corpus = [clean\_text(x) for x in all\_headlines]

* **Tokenization and Sequence Creation**

The cleaned text is tokenized, and sequences of tokens are created for n-gram models.

**code:**

from keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer()

def get\_sequence\_of\_tokens(corpus):

tokenizer.fit\_on\_texts(corpus)

total\_words = len(tokenizer.word\_index) + 1

input\_sequences = []

for line in corpus:

token\_list = tokenizer.texts\_to\_sequences([line])[0]

for i in range(1, len(token\_list)):

n-gram\_sequence = token\_list[:i+1]

input\_sequences.append(n-gram\_sequence)

return input\_sequences, total\_words

inp\_sequences, total\_words = get\_sequence\_of\_tokens(corpus)

* **Padding Sequences**

Sequences are padded to ensure uniform length, and predictors and labels are created for model training.

**code:**

from keras.preprocessing.sequence import pad\_sequences

from keras.utils import np\_utils

import numpy as np

def generate\_padded\_sequences(input\_sequences):

max\_sequence\_len = max([len(x) for x in input\_sequences])

input\_sequences = np.array(pad\_sequences(input\_sequences, maxlen=max\_sequence\_len, padding='pre'))

predictors, label = input\_sequences[:,:-1], input\_sequences[:,-1]

label = np\_utils.to\_categorical(label, num\_classes=total\_words)

return predictors, label, max\_sequence\_len

predictors, label, max\_sequence\_len = generate\_padded\_sequences(inp\_sequences)

* **Model Creation**

An LSTM-based model is created with an embedding layer, an LSTM layer, a dropout layer, and a dense output layer.

**code:**

from keras.models import Sequential

from keras.layers import Embedding, LSTM, Dense, Dropout

def create\_model(max\_sequence\_len, total\_words):

input\_len = max\_sequence\_len - 1

model = Sequential()

model.add(Embedding(total\_words, 10, input\_length=input\_len))

model.add(LSTM(100))

model.add(Dropout(0.1))

model.add(Dense(total\_words, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam')

return model

model = create\_model(max\_sequence\_len, total\_words)

model.summary()

**4. Implementation**

* **Training the Model**

A few heuristics should be sufficient for handling many of the issues when training such models. Start by getting the model to overfit on a tiny subset of the data as a quick sanity check. If the loss explodes, reduce the learning rate and clip the gradient until it doesn’t. If the model overfits, apply dropout [Srivastava et al., 2014, Zaremba et al., 2014] and weight decay until it doesn’t. For SGD and its variants, periodically annealing the learning rate when the validation loss fails to decrease typically helps significantly.

A few useful heuristics that should be robust to the hyperparameter settings and optimization settings you use:

* Sort the next dozen or so batches of sentences by length so each batch has examples of roughly the same length, thus saving computation [Sutskever et al., 2014].
* If the training set is small, tuning regularization will be key to performance [Melis et al., 2017]. Noising (or “token dropout”) is also worth trying [Xie et al., 2017]. Though we only touch on this issue briefly, amount of training data will be in most cases the primary bottleneck in the performance of these models.
* Measure validation loss after each epoch and anneal the learning rate when validation loss stops decreasing. Depending on how much the validation loss fluctates (based off of validation set size and optimizer settings) you may wish to anneal with patience (wait for several epochs of non-decreasing validation loss before reducing the learning rate).

The model is trained on the predictors and labels for 50 epochs.

**code:**

model.fit(predictors, label, epochs=50, verbose=5)

* **Generating Text**

The model is used to generate text based on seed text and a specified number of next words.

**code:**

def generate\_text(seed\_text, next\_words, model, max\_sequence\_len):

for \_ in range(next\_words):

token\_list = tokenizer.texts\_to\_sequences([seed\_text])[0]

token\_list = pad\_sequences([token\_list], maxlen=max\_sequence\_len-1, padding='pre')

predict\_x=model.predict(token\_list)

predicted=np.argmax(predict\_x,axis=1)

output\_word = ""

for word, index in tokenizer.word\_index.items():

if index == predicted:

output\_word = word

break

seed\_text += " " + output\_word

return seed\_text.title()

print(generate\_text("united states", 5, model, max\_sequence\_len))

print(generate\_text("president trump", 4, model, max\_sequence\_len))

**Decoding**

Suppose you’ve trained a neural network encoder-decoder model that achieves reasonable perplexity on the validation set. You then try running decoding or generation using this model. The simplest way is to run greedy decoding, as described in Section 2.4. From there, beam search decoding should yield some additional performance improvements. However, it’s rare that things simply work. This section is intended for use as a quick reference when encountering common issues during decoding.7

7Many examples are purely illustrative excerpts from *Alice’s Adventures in Wonderland* [Carroll, 1865].

**Diagnostics**

First, besides manual inspection, it’s helpful to create some diagnostic metrics when debugging the different components of a text generation system. Despite training the encoder-decoder network to map source to target, during the decoding procedure we introduce two additional components:

* A scoring function *s*(*H*) that tells us how “good” a hypothesis *H* on the beam is (higher is better).
* Optionally, a language model trained on a large corpus which may or may not be similar to the target corpus.

It may not be clear which of these components we should prioritize when trying to improve the performance of the combined system; hence it can be very helpful to run ablative analysis [Ng, 2007], For the language model, a few suggestions are measuring performance for *λ* = 0 and several other reasonably spaced values, then plotting the performance trend; measuring perplexity of the language model when trained on varying amounts of training data, to see if more data would be helpful or yields diminishing returns; and measuring performance when training the language model on several different domains (news data, Wikipedia, etc.) in cases where it’s difficult to obtain data

close to the target domain.

When measuring the scoring function, computing metrics and inspecting the decoded outputs vs. the gold sentences often immediately yields insights. Useful metrics include:

* Average length of decoded outputs *Y*ˆ vs. average length of reference targets *Y* .
* *s*(*Y*ˆ) vs. *s*(*Y* ), then inspecting the ratio *s*(*Y*ˆ)*/s*(*Y* ). If the average ratio is especially low, then there may be a bug in the beam search, or the beam size may need to be increased. If the average ratio is high, then the scoring function may not be appropriate.
* For some applications computing edit distance (insertions, substitutions, deletions) between *Y*ˆ and *Y* may also be useful, for example by looking at the most frequent edits or by examining cases where the length-normalized distances are highest.

**Common issues**

* **Rare and out-of-vocabulary (OOV) words**

**Decoded** And as in ⟨unk⟩ thought he stood, The ⟨unk⟩, with eyes of flame⟨eos⟩

**Expected** And as in uffish thought he stood, The Jabberwock, with eyes of flame⟨eos⟩

For languages with very large vocabularies, especially languages with rich morphologies, rare words become problematic when choosing a tokenization scheme that results in more token labels than it is feasible to model in the output softmax. One ad hoc approach that was first used to deal with this issue is simply to truncate the softmax output size (to say, 50K), then assign the remaining token labels all to the ⟨unk⟩ class [Luong et al., 2014]. The box above illustrates the resulting output (after detokenization) when rare words are replaced with ⟨unk⟩s. A more elegant approach is to use character or subword preprocessing [Sennrich et al., 2015, Wu et al., 2016] to avoid OOVs entirely, though this can slow down runtime for both training and decoding.

*Y* timestep →

*X* timestep →

0*.*0 0*.*2 0*.*4 0*.*6 0*.*8 1*.*0

⟨ ⟩

Figure 6: Example of attention matrix *A* when decoding terminates early with the eos token without having covered the input *X*.

* **Decoded output short, truncated or ignores portions of input**

**Decoded** It’s no use going back to yesterday.⟨eos⟩

**Expected** It’s no use going back to yesterday, because I was a different person then.⟨eos⟩

During the decoding search procedure, hypotheses terminate with the ⟨eos⟩ token. The decoder network should learn to place very low probability on the ⟨eos⟩ token until the target is fully gen- erated; however, sometimes ⟨eos⟩ does not have sufficiently low probability. This is because as the length of the hypothesis grows, the total log probability only decreases. Thus, if we do not normalize the log probability by the length of the hypothesis, shorter hypotheses will be favored. The box above illustrates an example where the hypothesis terminates early. This issue is exacerbated when incorporating a language model term. Two simple ways of resolving this issue are normalizing the log-probability score and adding a length bonus.

* *Length normalization*: Replace the score *s*(*Y*ˆ) with the score normalized by the hypothesis length *s*(*Y*ˆ)*/T*ˆ.
* *Length bonus*: Replace the score *s*(*Y*ˆ) with the *s*(*Y*ˆ) + *βT*ˆ, where *β >* 0 is a hyperparameter.

Note that normalizing the total log-probability by length is equivalent to maximizing the *T*ˆth root of the probability, while adding a length bonus is equivalent to multiplying the probability at every timestep by a baseline *eβ*.

Another method for avoiding this issue is with a *coverage penalty* using the attention matrix *A* [Tu et al., 2016, Wu et al., 2016]. As formulated here, the coverage penalty can only be applied once a hypothesis (with a corresponding attention matrix) has terminated; hence it can only be

* **Decoded output repeats**

**Decoded** I’m not myself, you see, you see, you see, you see, . . .

**Expected** I’m not myself, you see.⟨eos⟩

Repeating outputs are a common issue that often seem to expose neural versus template-based systems. Simple measures include adding a penalty when the model reattends to previous timesteps after the attention has shifted away. This is easily detected using the attention matrix *A* with some manually selected threshold.

Finally, a more fundamental issue to consider with repeating outputs is poor training of the model parameters. Passing in the attention vector *Ai*−1 as part of the decoder input when predicting *yi* is another training-time method

* **Lack of diversity**

In dialogue and QA, where there are often very common responses for many different conversation turns, generic responses such as “I don’t know” are a common problem. Similarly, in problems where many possible source inputs map to a much smaller set of possible target outputs, diversity of outputs can be an issue.

Increasing the temperature *τ* of the softmax exp(*zi/τ* )*/ j* exp(*zj/τ* ) is a simple method for trying to encourage more diversity in decoded outputs. In practice, however, a method penalizing low-ranked siblings during each step of the beam search decoding procedure has been shown to work

well [Li et al., 2016a]. Another more sophisticated method is to maximize the mutual information between the source and target, but is significantly more difficult to implement and requires generating *n*-best lists [Li et al., 2015].

**Deployment**

Although speed of decoding is not a huge concern when trying to achieve state-of-the-art results, it is a concern when deploying models in production, when real-time decoding is often a requirement. Beyond gains from using highly parallelized hardware such as GPUs or from using libraries with optimized matrix-vector operations, we now discuss some other techniques for improving the runtime of decoding.

Consider the factors which determine the runtime of the decoding algorithm. For the beam search algorithms we consider, the runtime should scale linearly with the beam size *k* (although in practice, batching hypotheses can lead to sublinear scaling). The runtime will often scale approximately quadratically with the hidden size of network *n*, and finally, quadratically with the number of timesteps *t* if using attention.8 Thus decoding might have a complexity of *O*(*kn*2*t*2).

Thus, (a jumbled collection of) possible methods for speeding up decoding include developing heuristics to prune the beam, finding the best trade-off between size of the vocabulary (softmax) and decoder timesteps, batching multiple examples together, caching previous computations (in the case of CNN models), and performing as much computation as possible within the compiled computation graph.

**5. Results and Discussion**

* **Model Performance**

The model’s performance is evaluated based on the coherence and relevance of the generated text. Sample outputs include:

"United States Is Expected To Be"

"President Trump Says He"

These results demonstrate the model's ability to generate contextually relevant and grammatically correct text sequences.

* **Analysis**

While the model performs well on short sequences, generating longer and more complex sentences remains a challenge. The model may occasionally produce repetitive or less meaningful sequences.

**6. Conclusion**

* **Summary of Findings**

The LSTM-based text generation model is effective in generating coherent short text sequences, such as headlines. The results demonstrate the potential of LSTM networks for text generation tasks.

* **Contributions**

This project contributes to the field by providing a practical implementation of an LSTM-based text generation model and highlighting its strengths and limitations.

* **Future Work**

Future research could explore more advanced architectures like Transformers, fine-tuning on larger and more diverse datasets, and generating longer text sequences with improved coherence.

**7. References**

Papers on text generation techniques and LSTM networks.

Keras and TensorFlow documentation.

Articles and tutorials on NLP and deep learning.

**8. Appendices**

Full code implementation.

Additional data samples.

Extended results and analyses.

Detailed Explanation with Theoretical Examples

To provide a deeper understanding, let's break down some theoretical examples used in this project.

**N-gram Models**

An n-gram model generates text by predicting the next word based on the previous 'n-1' words. For example, in a bigram model (n=2):

Given the sequence "I am", it predicts the next word.

If "I am" is frequently followed by "happy" in the training data, the model might predict "happy".

**Markov Chains**

A Markov chain is a probabilistic model that transitions from one state (word) to another based on learned probabilities. For instance:

From the state "sunny", the chain might transition to "day" with a high probability if "sunny day" is common in the training data.

**LSTM Networks**

LSTMs are a type of recurrent neural network (RNN) designed to handle long-term dependencies by using memory cells and gates (input, forget, output) to regulate information flow. Here's a simplified example:

When generating the text "The cat sat", the LSTM learns to remember the context of "The cat" to predict "sat".

The memory cells help retain information about earlier words in