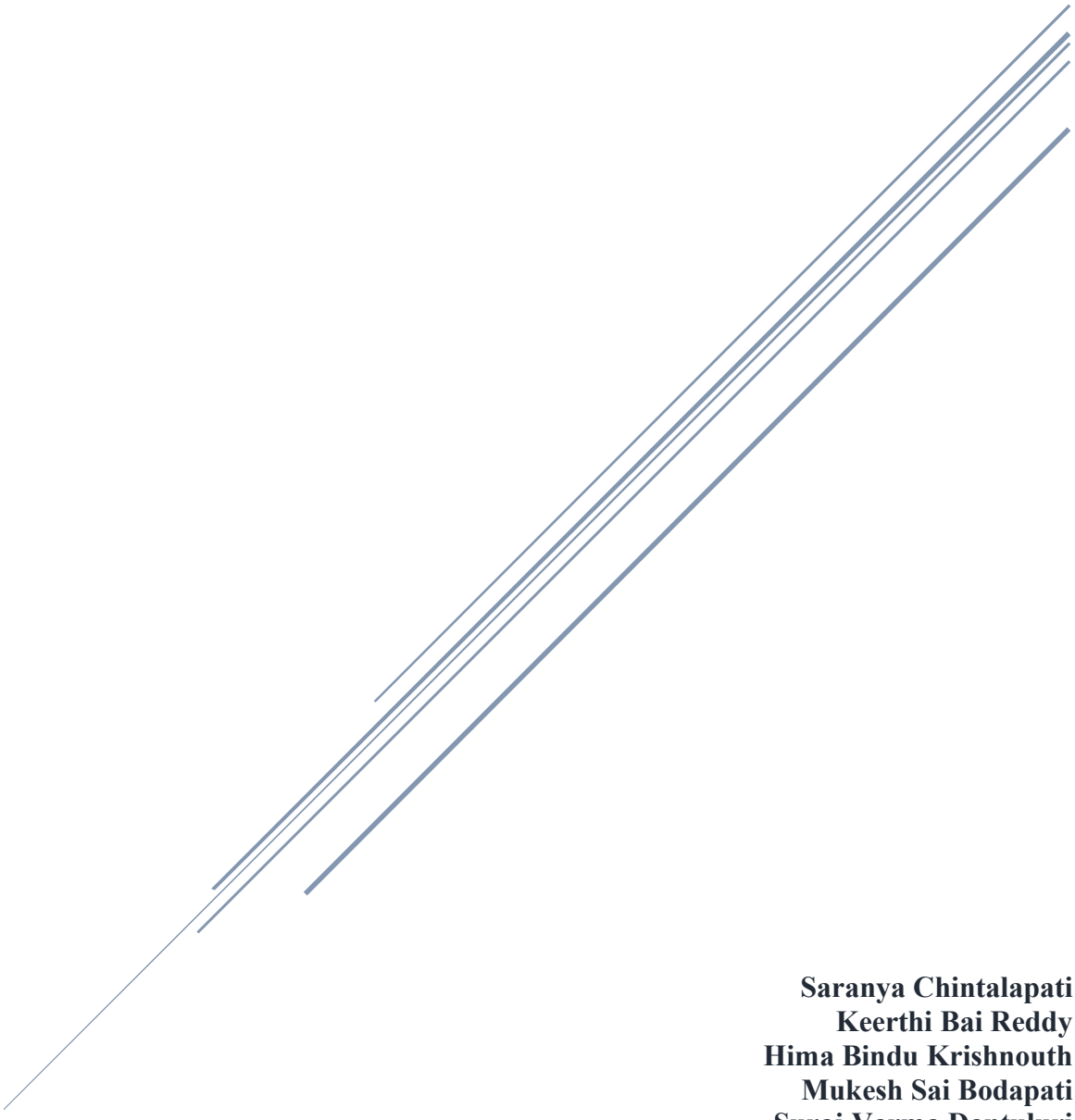
Title Generator for YouTube videos



Saranya Chintalapati
Keerthi Bai Reddy
Hima Bindu Krishnouth
Mukesh Sai Bodapati
Suraj Varma Dantuluri

Introduction:

In the digital era dominated by multimedia content, YouTube stands as a colossal platform where millions of videos are uploaded and consumed daily. The effectiveness of a video's title plays a crucial role in capturing viewers' attention and determining its discoverability. Leveraging the power of deep learning, specifically Long Short-Term Memory (LSTM) networks, this project aims to revolutionize the process of generating compelling and engaging titles for YouTube videos.

Traditional methods of crafting video titles often rely on intuition and creativity, leaving room for improvement in terms of maximizing click-through rates and enhancing content discoverability. By harnessing the capabilities of LSTM, a type of recurrent neural network (RNN) designed to capture and learn patterns over sequential data, we aim to develop a title generation model that understands the nuances of successful YouTube titles.

The LSTM architecture excels in processing and predicting sequences, making it an ideal candidate for capturing the underlying structures in language. Through extensive training on a diverse dataset of YouTube video titles, our model learns to generate titles that are not only contextually relevant but also aligned with the preferences of viewers.

Problem Statement:

The optimization of YouTube video titles poses a significant challenge for content creators, requiring solutions that cater to engagement, search engine visibility, and user satisfaction. This project aims to address this challenge by developing a Title Generator for YouTube Videos using deep learning models like LSTM. The research will focus on investigating the effectiveness of the title generator in generating engaging and search-optimized titles, its adaptability across diverse content types and languages, and its impact on user engagement and content recommendation systems The goal is to provide content creators with a tool that enhances the efficiency of title creation while improving overall user satisfaction and interaction with YouTube content.

Data Set:

In our project, we work with datasets derived from YouTube's trending videos across three major regions: the United States, Canada, and Great Britain. These datasets encapsulate a diverse range of content, providing our model with exposure to various topics and audience preferences. Each dataset contains essential information about video titles, categories, and associated metadata.

To enhance our contextual understanding, we complement these video datasets with JSON files. These files provide additional details, particularly category information, offering a more comprehensive view of the content landscape.

Source: YouTube trend videos data

Importing the required libraries:

Firstly, we begin by importing all the necessary packages to run the model.

Data Processing and cleaning for training the model:

Category Information Extraction:

- Extract valuable category details from JSON files associated with each dataset.
- Enhance contextual understanding of content by incorporating category information.

Creation of 'category_title' Column:

- Introduce a new dimension to the dataset by mapping category names to their corresponding IDs.

- Enrich the dataset and provide a categorical perspective for the model.

Elimination of Duplicate Videos:

- Implement a robust step to drop duplicate videos based on unique video IDs.
- Ensure dataset integrity by removing redundancies and maintaining quality.

Genre-Specific Subset Creation:

- Curate a specific subset of content, focusing on titles within the 'Entertainment' category.
- Tailor the model for a particular genre, ensuring specialization in the title generation process.

Culmination for Model Training:

- These preprocessing steps result in a refined and enriched dataset.
- Positioned for effective training of our machine learning model, enabling the generation of compelling and contextually relevant titles.

Tokenization of input data:

Tokenizer Initialization: A Tokenizer object is created. The Tokenizer class is part of the Keras library and is used for text tokenization, which involves splitting text into individual words or tokens.

Tokenizing the Corpus: The fit_on_texts method is called on the tokenizer object. This method is used to update the internal vocabulary of the tokenizer based on the text data provided in the corpus. After this step, the tokenizer has learned the vocabulary and assigns a unique integer (token) to each word in the text corpus.

Counting Total Words: The word_index attribute of the tokenizer contains a dictionary that maps words to their corresponding integer tokens. By adding 1 to the length of this dictionary, you calculate the total number of unique words (tokens) in the corpus. This is stored in the total_words variable.

Converting to Sequences of Tokens: This part of the code takes each line (sentence or text) in the corpus and processes it.For each line, it does the following:

token_list is obtained by applying the texts_to_sequences method to tokenize the line. This converts the text into a sequence of integer tokens.

It then generates n-gram sequences from the token list. An n-gram sequence is a sequence of tokens with a length ranging from 2 to the length of the token list.

These n-gram sequences are added to the input_sequences list. The result is a list of sequences, where each sequence represents a series of tokens from the original text. These sequences are used for training various natural language processing models, such as neural networks for text generation or prediction.

```
tokenizer = Tokenizer()
def get_sequence_of_tokens(corpus):
    # Get tokens
    tokenizer.fit_on_texts(corpus)
    total_words = len(tokenizer.word_index) + 1

    # Convert to sequence of tokens
    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 in Sequence Data:

Padding sequences is a crucial step in preparing text data for machine learning models, especially neural networks. In natural language processing tasks, text sequences often have varying lengths. However, for training a model, it's essential that all input sequences have the same length. Padding involves adding special tokens (usually zeros) to the beginning or end of sequences to make them uniform in length. The reason for this is that most machine learning frameworks, including neural networks, expect input data to be of consistent shape.

Finding the Maximum Sequence Length: This line calculates the maximum sequence length among all sequences in the input_sequences list. It iterates through each sequence and determines the length of the longest one Padding Sequences:

The pad_sequences function is used to ensure that all sequences in input_sequences have the same length.

Maxlen = max_sequence_len ensures that sequences are padded to the length of the longest sequence. padding='pre' indicates that the padding is added to the beginning of each sequence. Padding can be added to the beginning ('pre') or the end ('post') of the sequences.

Preparing Predictors and Labels: The sequences are split into "predictors" and "labels. "predictors contains all but the last token in each sequence, and label contains the last token in each sequence. This is a common setup for text prediction tasks, where the model is trained to predict the next word or token in a sequence given the previous words or tokens.

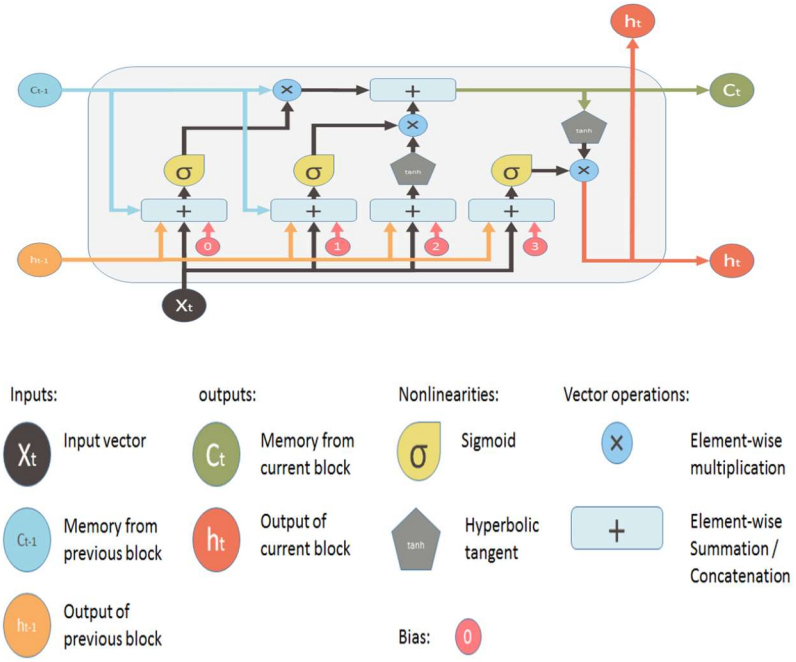
One-Hot Encoding of Labels: The labels are one-hot encoded. This means that each label (token) is converted into a binary vector where a 1 is placed at the index corresponding to the token's integer value, and all other positions contain 0s. This encoding is used for training models that predict the next token, effectively turning it into a classification problem.

```
from tensorflow.keras.utils import to_categorical

def generate_padded_sequences(input_sequences, total_words):
    max_sequence_len = max([len(x) for x in input_sequences])
    input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_len, padding='pre')
    predictors, label = input_sequences[:, :-1], input_sequences[:, -1]
    label = ku.to_categorical(label, num_classes=total_words)
    return predictors, label, max_sequence_len
```

LSTM Model:

In contrast to feedforward neural networks that propagate activation outputs in a single direction, recurrent neural networks (RNNs) operate bidirectionally, enabling information flow from input to output and vice versa. This bidirectional communication establishes loops within the neural network architecture, effectively creating a 'memory

state' for the neurons. This memory state allows RNNs to maintain and recall information across different time steps. While this memory capacity is advantageous, it can lead to issues like the vanishing gradient problem, particularly when dealing with deep networks. To address this challenge, a specialized RNN variant known as Long Short-Term Memory (LSTM) was introduced, designed to better handle, and retain information over extended periods.



Model – 1:

input_len is calculated as max_sequence_len - 1, representing the length of input sequences.

A sequential model is initialized using Keras, a popular deep learning library for Python.

The first layer added to the model is the "Input Embedding Layer," which is responsible for transforming input data into a dense vector space. It uses an embedding dimension of 10 for each word in the input sequence and has an input length of input_len.

The next layer is the "Hidden Layer 1," which is an LSTM (Long Short-Term Memory) layer with 100 units. LSTM is a type of recurrent neural network designed for sequence data. A dropout layer with a rate of 0.1 is added after the LSTM layer to prevent overfitting.

The final layer added to the model is the "Output Layer." It has several units equal to total_words, which corresponds to the total number of unique words in the vocabulary. The activation function used is softmax, which is common in text generation tasks.

The model is compiled with categorical cross-entropy as the loss function and the Adam optimizer, which is a popular optimization algorithm for training deep neural networks.

The function returns the compiled model. After creating the model, it is trained using the model.fit method. The input data (predictors) and target data (label) are used for training. It is trained for 20 epochs with a verbosity level of 5 (a high verbosity level provides fewer training details).

```
# Generate padded sequences
predictors, label, max_sequence_len = generate_padded_sequences(inp_sequences, total_words)

def create_model(max_sequence_len, total_words):
    input_len = max_sequence_len - 1
    model = Sequential()

    # Add Input Embedding Layer
    model.add(Embedding(total_words, 10, input_length=input_len))

    # Add Hidden Layer 1 - LSTM Layer
    model.add(LSTM(100))
    model.add(Dropout(0.1))

    # Add Output Layer
    model.add(Dense(total_words, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    return model

model = create_model(max_sequence_len, total_words)
history = model.fit(predictors, label, epochs=20, verbose=1)  # Set verbose to 1 for progress updates
```

Testing the model:

The function takes four parameters:

seed_text: The initial text or phrase from which text generation begins.

next_words: The number of words to generate after the seed_text.

model: The trained neural network model used for text generation.

max_sequence_len: The maximum sequence length used for padding sequences (typically derived from training data).

A for loop is used to generate the specified number of words (next_words) based on the provided seed_text.

The seed_text is tokenized using the tokenizer (presumably the same one used for training the model) to convert it into a sequence of word indices.

The tokenized seed_text is padded to match the max_sequence_len - 1 length. Padding is applied to the beginning of the sequence using padding='pre'. This is necessary to match the input format expected by the model.

The model.predict_classes method is used to predict the next word based on the tokenized and padded seed_text. The predicted variable stores the index of the predicted word.

The code then looks up the word corresponding to the predicted index using the tokenizer.word_index dictionary. This word becomes the output_word.

The seed_text is updated by appending a space followed by the output_word. This extended seed_text is then used for the next iteration to predict the next word.

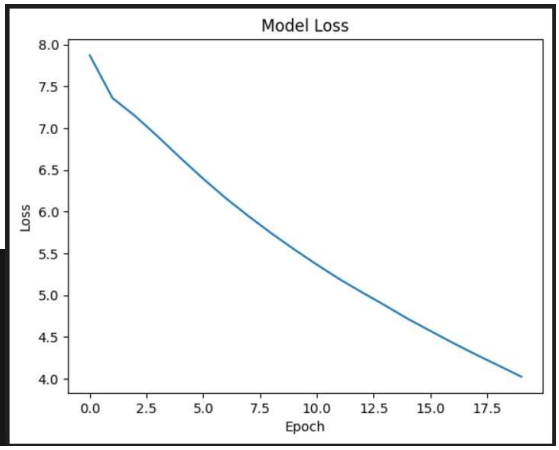
After generating the specified number of words, the function returns the generated text, which is title cased.

```python
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')
        predicted_probs = model.predict(token_list, verbose=0)

        # Find the word with the highest probability
        predicted = np.argmax(predicted_probs)

        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()
```

Model 1 Results:

We utilized Matplotlib to create a line plot of the training loss over epochs.

```python
import matplotlib.pyplot as plt

# Plot the training loss
plt.plot(history.history['loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



Printing results for Model -1:

```
print(generate_text("Stand-up", 5, model, max_sequence_len))
✓ 0.2s
Stand-Up With Laugh On The Time
```

Model 2: Added L2 regularization:

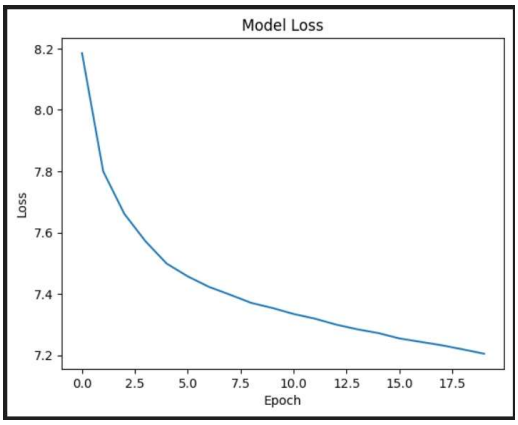
We have model few changes to see if the model can be improved or the changes that happen when there are a few hyperparameters, addition of one more LSTM layer and added a different optimizer.

L2 regularization is a technique used to prevent overfitting in machine learning models. It works by adding a penalty term to the loss function that encourages the model to have small weights. The penalty term is proportional to the square of the magnitude of the weights. This means that large weights are heavily penalized, while small weights are not penalized as much. By adding this penalty term, the model is encouraged to learn simpler patterns that generalize better to new data. L2 regularization is also known as weight decay or ridge regression.

```
# Add Output Layer
model.add(Dense(total_words, activation='softmax', kernel_regularizer=l2(0.01)))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Model 2 results:

This addition of regularizer has given more meaningful results like you see before:




```
print(generate_text("Stand-up", 3, model, max_sequence_len))
```
✓ 0.2s

```
Stand-Up With Daily Show
```

Model 3: Added a additional LSTM layer with unit size 50 & used optimizer RMS prop:

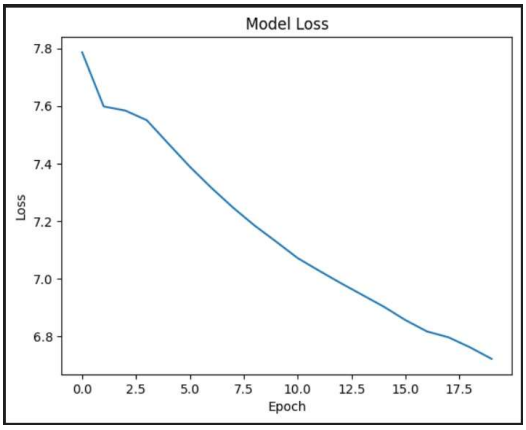
RMSprop, short for Root Mean Square Propagation, is an optimization algorithm commonly used for training artificial neural networks, particularly deep learning models. It is designed to address some of the limitations of the basic stochastic gradient descent (SGD) optimization algorithm.

```
#Increase embedding dimension
model.add(Embedding(total_words, 50, input_length=input_len))

model.add(LSTM(100, return_sequences=True))  # Add another LSTM layer
model.add(LSTM(50))  # Add another LSTM layer
```

```
# Add Output Layer
model.add(Dense(total_words, activation='softmax'))  # Try 'relu' activation

model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

Model – 3 results:

The results shown for comedy is shown below:




```
print(generate_text("Stand-up", 4, model, max_sequence_len))
✓ 0.7s
Stand-Up A The The The
```

Conclusion:

In conclusion, after evaluating the performance of the three models, it is evident that Model 2 outperforms the others. The titles generated by Model 2 exhibit greater appropriateness relative to the input words, demonstrating its superiority in capturing and generating contextually relevant content.

Future Research Questions:

Ethical Considerations:

What ethical considerations should be addressed in the development and deployment of LSTM-based title generators, particularly in the context of potential biases in generated titles?

How can the responsible use of AI be ensured to avoid unintended consequences in title generation?

Domain-Specific Title Generation:

How well do LSTM-based title generators perform in specific domains, such as scientific literature, news articles, or social media content?

Can domain-specific fine-tuning improve the relevance and coherence of generated titles?

Multimodal Title Generation:

How can LSTM models be extended to generate titles for multimodal data, such as images or audio, in addition to text?

What are the challenges and opportunities in integrating multiple modalities for title generation?

Multilingual Title Generation:

How well do LSTM-based models perform in generating titles for languages other than English?

What challenges and opportunities arise in adapting title generators for multilingual contexts?

References:

Data source: https://github.com/amankharwal/Website-data/blob/master/Data.rar

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

https://github.com/AngusTheMack/title-generator

https://thecleverprogrammer.com/2020/10/05/title-generator-with-machine-learning/

https://towardsdatascience.com/generating-scientific-papers-titles-using-machine-learning-98c8c9bc637e