Text generation is a fascinating area of natural language processing (NLP) where models learn to produce human-like text. You can achieve this using various architectures, with GPT (Generative Pre-trained Transformer) and LSTM (Long Short-Term Memory) being prominent choices.

While GPT models (especially larger ones like GPT-2, GPT-3, or GPT-4) are generally more powerful and capable of generating highly coherent and contextually relevant paragraphs due to their transformer architecture and massive pre-training, LSTMs offer a simpler and more accessible entry point for building text generation models from scratch with smaller datasets.

This deliverable will focus on demonstrating text generation using **LSTM** as it allows for a clearer understanding of the underlying principles without requiring access to large-scale pre-trained GPT models or their associated computational resources for fine-tuning on custom topics. For GPT, we'll demonstrate using a readily available pre-trained model.

**Text Generation Model using LSTM**

LSTM networks are a type of recurrent neural network (RNN) particularly well-suited for sequence prediction tasks like text generation because they can learn long-term dependencies in sequential data.

**How it Works (LSTM)**

1. **Data Preparation**:
   * **Collect Text Data**: A large corpus of text is needed. The quality and diversity of this data directly impact the generated text.
   * **Tokenization**: The text is broken down into smaller units, typically words or characters. Each unique token is assigned a numerical ID.
   * **Sequence Creation**: The data is transformed into sequences of input-output pairs. For example, if you want to predict the next word, an input sequence would be "The quick brown" and the output would be "fox".
   * **Padding**: Since sequences often have varying lengths, they are padded to a uniform length to be fed into the neural network.
2. **Model Architecture**:
   * **Embedding Layer**: Converts the numerical token IDs into dense vector representations (embeddings). This helps the model capture semantic relationships between words.
   * **LSTM Layer(s)**: The core of the model. LSTMs process the input sequences, maintaining an internal "memory" of past information to predict the next token.
   * **Dense (Output) Layer**: A fully connected layer that outputs a probability distribution over the entire vocabulary, indicating the likelihood of each token being the next in the sequence.
   * **Softmax Activation**: Normalizes the output probabilities.
3. **Training**:
   * The model is trained to minimize the difference between its predicted next token probabilities and the actual next token. This is typically done using categorical cross-entropy loss.
   * During training, the model learns the patterns, grammar, and style of the input text.
4. **Text Generation**:
   * Start with a "seed" prompt (a sequence of words).
   * The model predicts the next word based on the seed.
   * The predicted word is then appended to the seed, and the process repeats, generating new text word by word.
   * **Temperature**: A crucial parameter that controls the randomness of the generation. A lower temperature (e.g., 0.2) makes the model more deterministic and predictable, often leading to common words. A higher temperature (e.g., 1.0) increases randomness, leading to more creative but potentially less coherent output.

**Python Notebook Demonstration (LSTM)**

Let's create a simple LSTM-based text generation model using TensorFlow/Keras.

Python

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

import random

# --- 1. Data Preparation ---

# Sample text data for demonstration. In a real-world scenario, you'd use a much larger corpus.

# We'll use a short passage to keep the training time manageable for a notebook.

text\_corpus = """

The quick brown fox jumps over the lazy dog.

A journey of a thousand miles begins with a single step.

Machine learning is a fascinating field.

Artificial intelligence is transforming the world.

Natural language processing is a subfield of AI.

Deep learning uses neural networks.

"""

# Tokenize the text

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts([text\_corpus])

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

# Create input sequences (n-gram sequences)

input\_sequences = []

for line in text\_corpus.split('\n'):

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)

# Pad 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'))

# Split into input (X) and output (y)

X, y = input\_sequences[:, :-1], input\_sequences[:, -1]

y = tf.keras.utils.to\_categorical(y, num\_classes=total\_words)

print(f"Total words (vocabulary size): {total\_words}")

print(f"Max sequence length: {max\_sequence\_len}")

print(f"Number of training sequences: {len(X)}")

# --- 2. Model Architecture ---

model = Sequential()

model.add(Embedding(total\_words, 100, input\_length=max\_sequence\_len-1)) # Embedding dimension 100

model.add(LSTM(150, return\_sequences=True)) # LSTM units 150

model.add(Dropout(0.2))

model.add(LSTM(100)) # Another LSTM layer

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

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

print("\nModel Summary:")

model.summary()

# --- 3. Training ---

print("\nTraining the LSTM model (this might take a few moments)...")

history = model.fit(X, y, epochs=500, verbose=0) # Train for more epochs on small data for better results

print(f"\nTraining complete. Final accuracy: {history.history['accuracy'][-1]:.4f}")

# --- 4. Text Generation Function ---

def generate\_text(seed\_text, next\_words, model, max\_sequence\_len, tokenizer, temperature=1.0):

"""

Generates text using the trained LSTM model.

Args:

seed\_text (str): The initial text to start generation from.

next\_words (int): The number of words to generate.

model (tf.keras.Model): The trained Keras model.

max\_sequence\_len (int): The maximum sequence length used for training.

tokenizer (Tokenizer): The Keras Tokenizer fitted on the training data.

temperature (float): Controls randomness. Lower values (e.g., 0.2) make it

more deterministic, higher values (e.g., 1.0) make it

more creative.

Returns:

str: The generated text.

"""

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\_probabilities = model.predict(token\_list, verbose=0)[0]

# Apply temperature to probabilities

predicted\_probabilities = np.log(predicted\_probabilities) / temperature

exp\_preds = np.exp(predicted\_probabilities)

predicted\_probabilities = exp\_preds / np.sum(exp\_preds)

predicted\_word\_index = np.argmax(np.random.multinomial(1, predicted\_probabilities, 1))

output\_word = ""

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

if index == predicted\_word\_index:

output\_word = word

break

seed\_text += " " + output\_word

return seed\_text

# --- Demonstrate Generated Text based on User Prompts ---

print("\n--- Generated Text ---")

# Prompt 1: Technology topic

prompt1 = "Artificial intelligence is"

generated\_text1 = generate\_text(prompt1, 10, model, max\_sequence\_len, tokenizer, temperature=0.7)

print(f"Prompt: \"{prompt1}\"")

print(f"Generated Text (Temp 0.7): \"{generated\_text1}\"\n")

# Prompt 2: General knowledge topic

prompt2 = "A journey of"

generated\_text2 = generate\_text(prompt2, 8, model, max\_sequence\_len, tokenizer, temperature=0.9)

print(f"Prompt: \"{prompt2}\"")

print(f"Generated Text (Temp 0.9): \"{generated\_text2}\"\n")

# Prompt 3: Another general topic

prompt3 = "Deep learning"

generated\_text3 = generate\_text(prompt3, 7, model, max\_sequence\_len, tokenizer, temperature=0.5)

print(f"Prompt: \"{prompt3}\"")

print(f"Generated Text (Temp 0.5): \"{generated\_text3}\"\n")

# Prompt 4: Experiment with higher temperature

prompt4 = "The quick brown"

generated\_text4 = generate\_text(prompt4, 6, model, max\_sequence\_len, tokenizer, temperature=1.2)

print(f"Prompt: \"{prompt4}\"")

print(f"Generated Text (Temp 1.2): \"{generated\_text4}\"\n")

# Prompt 5: Longer generation

prompt5 = "Natural language"

generated\_text5 = generate\_text(prompt5, 15, model, max\_sequence\_len, tokenizer, temperature=0.8)

print(f"Prompt: \"{prompt5}\"")

print(f"Generated Text (Temp 0.8): \"{generated\_text5}\"\n")

**Text Generation Model using GPT (Pre-trained)**

For GPT, we typically leverage **pre-trained models** from libraries like Hugging Face's transformers. Training a GPT model from scratch requires immense computational resources and an extremely large dataset, which is beyond the scope of a typical notebook demonstration. However, using a pre-trained model like **GPT-2** is straightforward and demonstrates the power of these models.

**How it Works (GPT with Hugging Face)**

1. **Load Pre-trained Model and Tokenizer**: You simply load a pre-trained GPT-2 model (e.g., gpt2, gpt2-medium, gpt2-large) and its corresponding tokenizer from the transformers library.
2. **Generate Text**: The generate() method of the model handles the entire generation process, including sampling strategies (e.g., greedy, beam search, top-k, top-p) and parameters like max\_length and temperature.

**Python Notebook Demonstration (GPT-2)**

Python

from transformers import GPT2LMHeadModel, GPT2Tokenizer

import torch

# --- 1. Load Pre-trained Model and Tokenizer ---

model\_name = "gpt2" # You can try "gpt2-medium" or "gpt2-large" for potentially better results

tokenizer = GPT2Tokenizer.from\_pretrained(model\_name)

model = GPT2LMHeadModel.from\_pretrained(model\_name)

# Set the pad\_token\_id to eos\_token\_id to avoid warnings for GPT-2

tokenizer.pad\_token = tokenizer.eos\_token

model.config.pad\_token\_id = model.config.eos\_token\_id

print(f"Loaded pre-trained GPT-2 model: {model\_name}\n")

# --- 2. Text Generation Function ---

def generate\_gpt2\_text(prompt, max\_length=100, temperature=0.7, top\_k=50, top\_p=0.95):

"""

Generates text using a pre-trained GPT-2 model.

Args:

prompt (str): The input text to start generation from.

max\_length (int): The maximum length of the generated text (including the prompt).

temperature (float): Controls the randomness of the generation (0.0 to 1.0+).

top\_k (int): The number of highest probability vocabulary tokens to consider for sampling.

top\_p (float): The cumulative probability threshold for nucleus sampling.

Returns:

str: The generated text.

"""

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

# Generate text

output\_sequences = model.generate(

input\_ids,

max\_length=max\_length,

temperature=temperature,

top\_k=top\_k,

top\_p=top\_p,

num\_return\_sequences=1,

pad\_token\_id=tokenizer.eos\_token\_id, # Ensure pad\_token\_id is set

do\_sample=True # Enable sampling for temperature, top\_k, top\_p

)

generated\_text = tokenizer.decode(output\_sequences[0], skip\_special\_tokens=True)

return generated\_text

# --- Demonstrate Generated Text based on User Prompts (GPT-2) ---

print("\n--- Generated Text (using GPT-2) ---")

# Prompt 1: Technology topic

prompt\_gpt1 = "The future of artificial intelligence will"

generated\_text\_gpt1 = generate\_gpt2\_text(prompt\_gpt1, max\_length=50, temperature=0.8)

print(f"Prompt: \"{prompt\_gpt1}\"")

print(f"Generated Text (Temp 0.8): \"{generated\_text\_gpt1}\"\n")

# Prompt 2: Creative writing

prompt\_gpt2 = "Once upon a time in a faraway land,"

generated\_text\_gpt2 = generate\_gpt2\_text(prompt\_gpt2, max\_length=70, temperature=0.9, top\_k=50)

print(f"Prompt: \"{prompt\_gpt2}\"")

print(f"Generated Text (Temp 0.9, Top-K 50): \"{generated\_text\_gpt2}\"\n")

# Prompt 3: Scientific explanation

prompt\_gpt3 = "Quantum computing is a revolutionary technology that"

generated\_text\_gpt3 = generate\_gpt2\_text(prompt\_gpt3, max\_length=60, temperature=0.7, top\_p=0.9)

print(f"Prompt: \"{prompt\_gpt3}\"")

print(f"Generated Text (Temp 0.7, Top-P 0.9): \"{generated\_text\_gpt3}\"\n")

# Prompt 4: Historical context

prompt\_gpt4 = "The Industrial Revolution brought about significant changes"

generated\_text\_gpt4 = generate\_gpt2\_text(prompt\_gpt4, max\_length=80, temperature=0.85)

print(f"Prompt: \"{prompt\_gpt4}\"")

print(f"Generated Text (Temp 0.85): \"{generated\_text\_gpt4}\"\n")

**Key Differences and Considerations**

* **LSTM**:
  + **Pros**: Easier to train from scratch on smaller, custom datasets; good for understanding foundational concepts of sequence modeling.
  + **Cons**: Limited in capturing very long-range dependencies; generated text might be less coherent or diverse compared to large pre-trained GPT models, especially with limited training data. Requires significant data and careful hyperparameter tuning for good performance.
  + **Use Cases**: Character-level text generation, simple word prediction, learning specific writing styles from moderate datasets.
* **GPT (Pre-trained)**:
  + **Pros**: Highly coherent and fluent text generation "out-of-the-box" due to vast pre-training on diverse internet text; excels at understanding context and generating relevant responses; capable of zero-shot and few-shot learning.
  + **Cons**: Training from scratch is computationally infeasible for most users; fine-tuning on custom data still requires a decent dataset and computational resources; can sometimes "hallucinate" or generate factually incorrect information.
  + **Use Cases**: Content creation (articles, stories, marketing copy), chatbots, code generation, summarization, translation, and any task requiring high-quality, human-like text.

This notebook demonstrates both approaches. For serious applications requiring high-quality, diverse, and coherent text generation, leveraging pre-trained GPT-style models is almost always the preferred and more effective strategy. LSTMs are valuable for learning fundamental concepts and for scenarios where fine-grained control over model architecture and training from scratch on very specific, smaller datasets is desired or necessary.