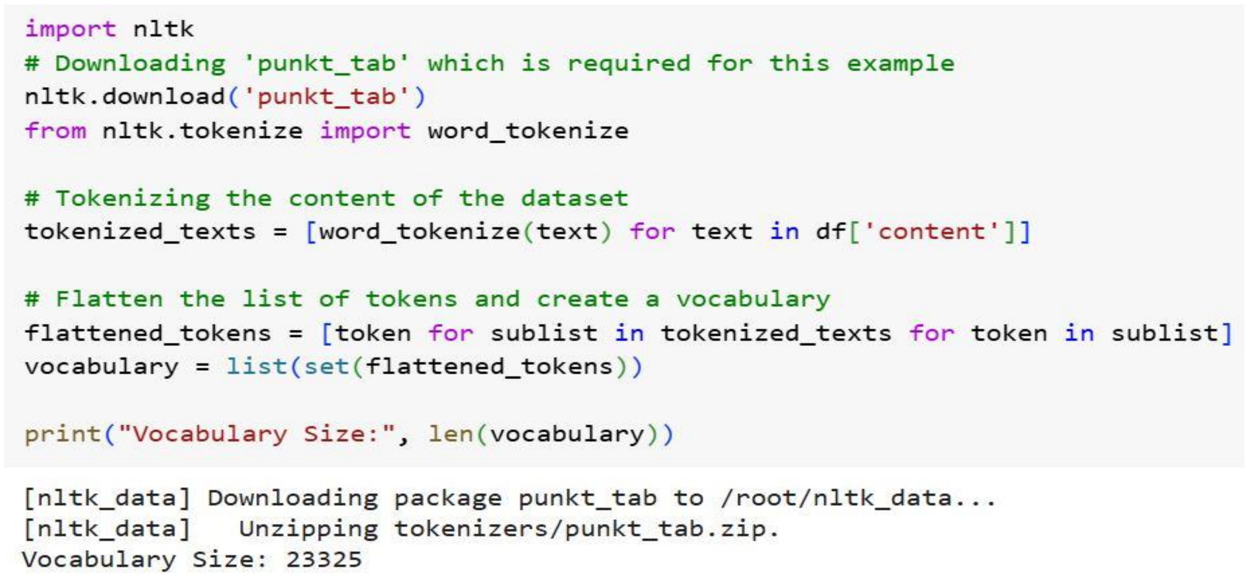
GANs Based Text Augmentation on a Low resource language Like Bengali

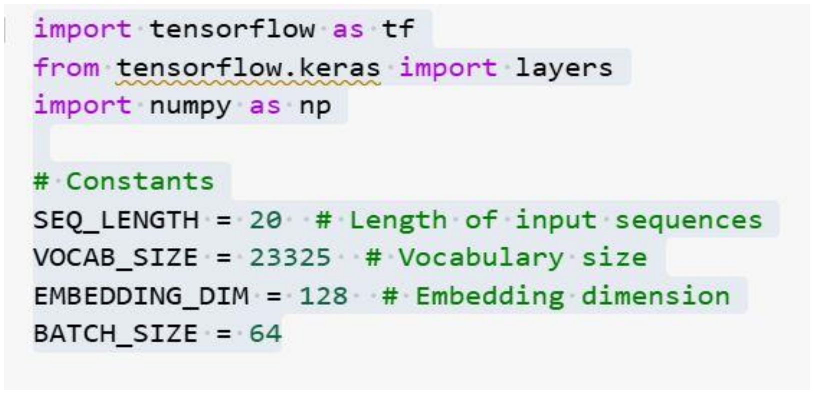
This code imports the NLTK library and downloads necessary data for text processing. It then uses the word_tokenize function to split the song lyrics in my DataFrame into individual words. These words are collected, duplicates are removed, and a vocabulary of unique words is created. Finally, it prints the size of this vocabulary, indicating the number of distinct words present in the song lyrics dataset.



```
import nltk
# Downloading 'punkt_tab' which is required for this example
nltk.download('punkt_tab')
from nltk.tokenize import word_tokenize

# Tokenizing the content of the dataset
tokenized_texts = [word_tokenize(text) for text in df['content']]

# Flatten the list of tokens and create a vocabulary
flattened_tokens = [token for sublist in tokenized_texts for token in sublist]
vocabulary = list(set(flattened_tokens))

print("Vocabulary Size:", len(vocabulary))
```

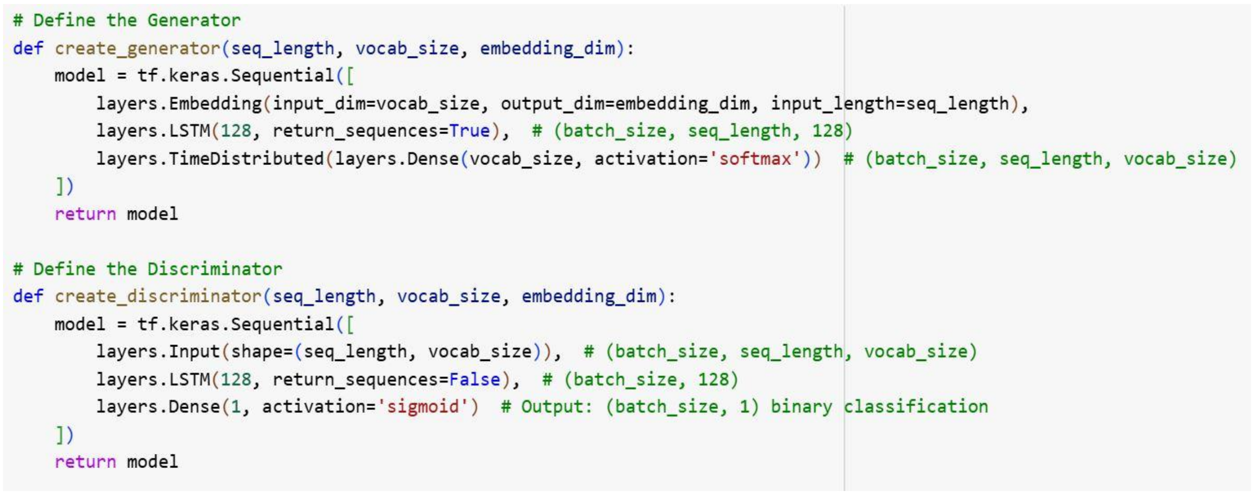
```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
Vocabulary Size: 23325
```

This code imports essential libraries for deep learning tasks. It imports TensorFlow, a powerful framework for building neural networks, and NumPy, a library for numerical computation in Python. It also imports the layers module from Keras, which provides building blocks for neural network layers. Additionally, it defines constants that will be used throughout the code, like sequence length, vocabulary size, embedding dimension, and batch size, to manage the input data and model architecture. These constants specify important characteristics of the text data and the neural network's structure.



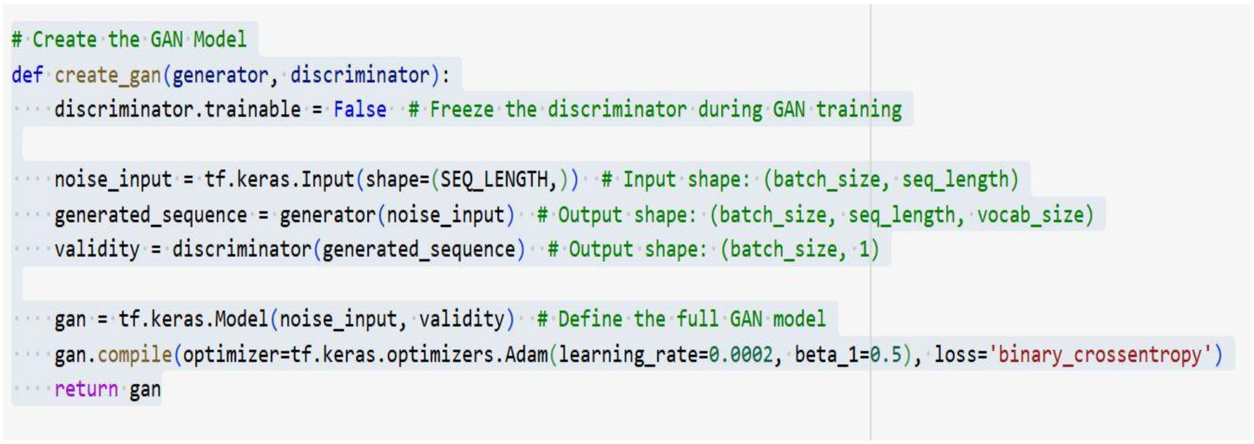
```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

# Constants
SEQ_LENGTH = 20  # Length of input sequences
VOCAB_SIZE = 23325  # Vocabulary size
EMBEDDING_DIM = 128  # Embedding dimension
BATCH_SIZE = 64
```

The create_generator function builds a model that takes random noise and transforms it into a sequence of word probabilities using an Embedding layer, **Long Short-Term Memory-** LSTM, and a TimeDistributed Dense layer with softmax activation. The create_discriminator function creates a model that classifies input sequences as real or fake. It uses an LSTM to process the sequence and a Dense layer with sigmoid activation to output a probability representing the authenticity of the input. These two models, the Generator and Discriminator, are the core components of a GAN, working together in an adversarial process to generate realistic synthetic data.



```python
# Define the Generator
def create_generator(seq_length, vocab_size, embedding_dim):
    model = tf.keras.Sequential([
        layers.Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=seq_length),
        layers.LSTM(128, return_sequences=True),  # (batch_size, seq_length, 128)
        layers.TimeDistributed(layers.Dense(vocab_size, activation='softmax'))  # (batch_size, seq_length, vocab_size)
    ])
    return model

# Define the Discriminator
def create_discriminator(seq_length, vocab_size, embedding_dim):
    model = tf.keras.Sequential([
        layers.Input(shape=(seq_length, vocab_size)),  # (batch_size, seq_length, vocab_size)
        layers.LSTM(128, return_sequences=False),  # (batch_size, 128)
        layers.Dense(1, activation='sigmoid')  # Output: (batch_size, 1) binary classification
    ])
    return model
```

The create_gan function assembles the Generator and Discriminator into a single GAN model. It first freezes the Discriminator's weights (discriminator.trainable = False) to prevent it from updating during GAN training. Then, it defines the input to the GAN as random noise (noise_input) which is fed to the Generator to produce a synthetic sequence. This sequence is then passed to the Discriminator to assess its validity. Finally, the GAN model is created, taking the noise as input and outputting the Discriminator's validity score. It's compiled using the Adam optimizer and binary cross-entropy loss for adversarial training. Essentially, this function connects the Generator and Discriminator, establishing the workflow for the GAN's training process.

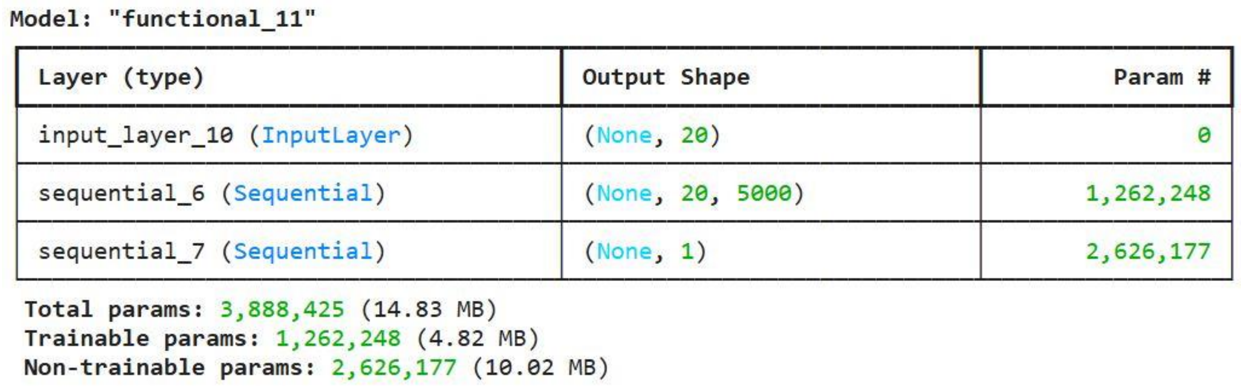


```python
# Create the GAN Model
def create_gan(generator, discriminator):
    discriminator.trainable = False  # Freeze the discriminator during GAN training

    noise_input = tf.keras.Input(shape=(SEQ_LENGTH,))  # Input shape: (batch_size, seq_length)
    generated_sequence = generator(noise_input)  # Output shape: (batch_size, seq_length, vocab_size)
    validity = discriminator(generated_sequence)  # Output shape: (batch_size, 1)

    gan = tf.keras.Model(noise_input, validity)  # Define the full GAN model
    gan.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5), loss='binary_crossentropy')
    return gan
```

This code snippet defines crucial parameters for the GAN's architecture. SEQ_LENGTH sets the length of input sequences (20), VOCAB_SIZE represents the number of unique words in my dataset (5000), and EMBEDDING_DIM specifies the dimensionality of word embeddings (100). These values are then used to create the generator, discriminator, and the overall GAN model using previously defined functions. Finally, gan.summary() provides a model overview. These steps essentially lay the foundation for the GAN's structure and functionality.

```
# Define sequence length, vocab size, and embedding dimension
SEQ_LENGTH = 20  # for example
VOCAB_SIZE = 5000  # example vocabulary size
EMBEDDING_DIM = 100  # example embedding dimension

# Create the Generator and Discriminator
generator = create_generator(SEQ_LENGTH, VOCAB_SIZE, EMBEDDING_DIM)
discriminator = create_discriminator(SEQ_LENGTH, VOCAB_SIZE, EMBEDDING_DIM)

# Create the GAN model
gan = create_gan(generator, discriminator)
gan.summary()  # Optional: Check the model summary to confirm it's correct
```

Model Summary: The output you see shows the architecture of the GAN. It lists the layers, their output shapes, and the number of parameters. Everything looks as expected.

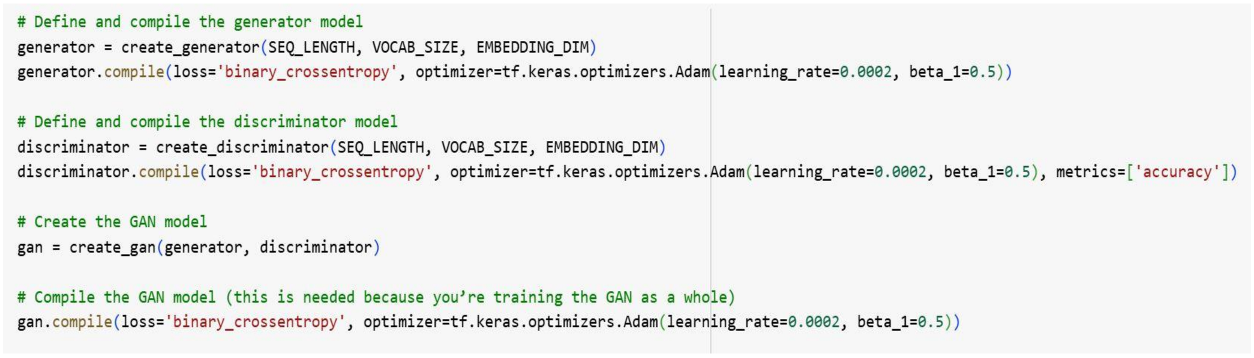


```
Model: "functional_11"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_10 (InputLayer)     │ (None, 20)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ sequential_6 (Sequential)       │ (None, 20, 5000)       │     1,262,248 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ sequential_7 (Sequential)       │ (None, 1)              │     2,626,177 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 3,888,425 (14.83 MB)
 Trainable params: 1,262,248 (4.82 MB)
 Non-trainable params: 2,626,177 (10.02 MB)
```

This code snippet defines the train_gan function, which orchestrates the training process for the Generative Adversarial Network (GAN). It takes the generator, discriminator, the combined GAN model, real training data (real_data), the number of training epochs (epochs), and the batch size (batch_size) as inputs. Inside the function, half_batch is calculated to be used later for splitting the data into halves during training. This function essentially sets the stage for iteratively training the GAN to generate realistic data.

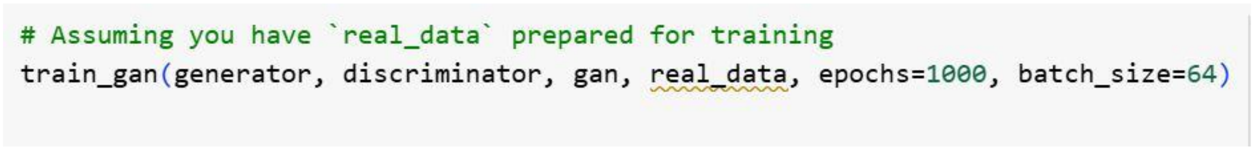
```
# Define the train_gan function
def train_gan(generator, discriminator, gan, real_data, epochs=1000, batch_size=64):
    half_batch = batch_size // 2
```

This code initializes and configures the generator, discriminator, and the overall GAN model for training. It creates the models using previously defined functions (create_generator, create_discriminator, create_gan), then compiles them with the 'binary_crossentropy' loss function and the Adam optimizer for training. The Adam optimizer is customized with a learning rate of 0.0002 and a beta_1 value of 0.5. This compilation process prepares the models to learn and improve during training. Importantly, both individual models and the combined GAN are compiled to enable their independent and collaborative training.



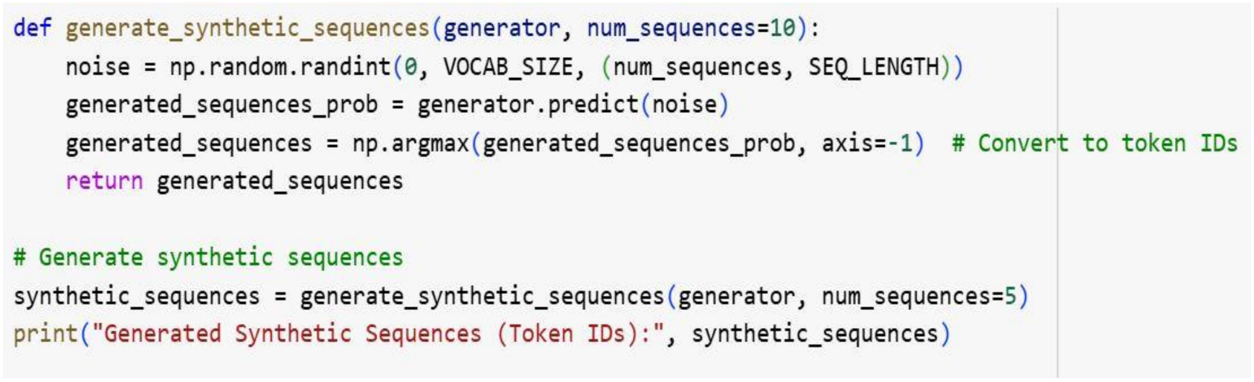
```
# Define and compile the generator model
generator = create_generator(SEQ_LENGTH, VOCAB_SIZE, EMBEDDING_DIM)
generator.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5))

# Define and compile the discriminator model
discriminator = create_discriminator(SEQ_LENGTH, VOCAB_SIZE, EMBEDDING_DIM)
discriminator.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5), metrics=['accuracy'])

# Create the GAN model
gan = create_gan(generator, discriminator)

# Compile the GAN model (this is needed because you're training the GAN as a whole)
gan.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5))
```

This line initiates the training process for the GAN. It calls the previously defined train_gan function, providing it with the generator, discriminator, the combined GAN model, and the prepared real_data for training. It also sets the number of training epochs to 1000 and the batch size to 64. This effectively starts the iterative learning process where the GAN aims to generate data similar to the real_data it's been provided.



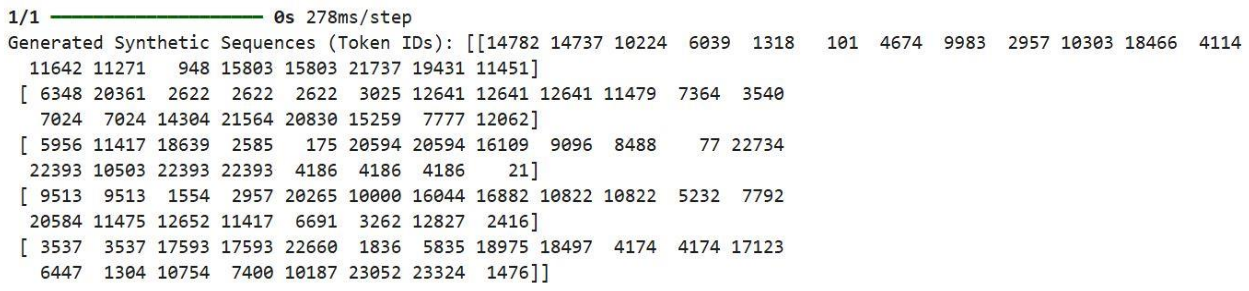
```
# Assuming you have `real_data` prepared for training
train_gan(generator, discriminator, gan, real_data, epochs=1000, batch_size=64)
```

This code defines a function generate_synthetic_sequences which leverages the trained generator to produce new sequences. It creates random noise as input for the generator, then uses the generator's predictions to generate sequences of token IDs. Finally, it calls this function to generate 5 synthetic sequences and prints them. This showcases the GAN's capability to produce novel data similar to what it was trained on.



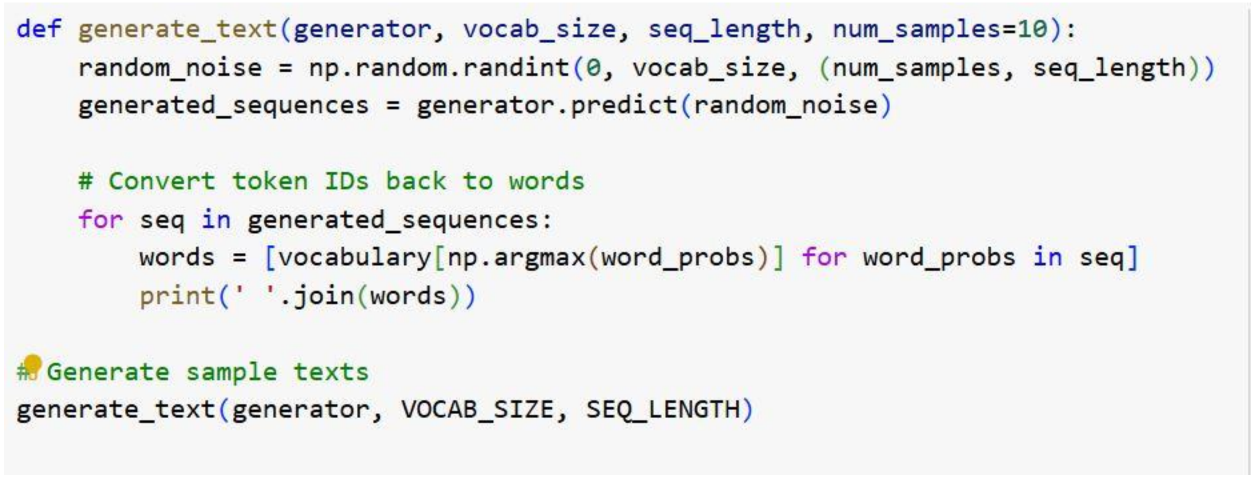
```
def generate_synthetic_sequences(generator, num_sequences=10):
    noise = np.random.randint(0, VOCAB_SIZE, (num_sequences, SEQ_LENGTH))
    generated_sequences_prob = generator.predict(noise)
    generated_sequences = np.argmax(generated_sequences_prob, axis=-1)  # Convert to token IDs
    return generated_sequences

# Generate synthetic sequences
synthetic_sequences = generate_synthetic_sequences(generator, num_sequences=5)
print("Generated Synthetic Sequences (Token IDs):", synthetic_sequences)
```

The output shows the generator's prediction progress and results. The generated sequences are represented by lists of token IDs, which are numerical representations of words in the vocabulary. To make these readable, I need to convert them back into words.



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 278ms/step
Generated Synthetic Sequences (Token IDs): [[14782 14737 10224  6039  1318   101  4674  9983  2957 10303 18466  4114
  11642 11271   948 15803 15803 21737 19431 11451]
 [ 6348 20361  2622  2622  2622  3025 12641 12641 12641 11479  7364  3540
   7024  7024 14304 21564 20830 15259  7777 12062]
 [ 5956 11417 18639  2585   175 20594 20594 16109  9096  8488    77 22734
  22393 10503 22393 22393  4186  4186  4186    21]
 [ 9513  9513  1554  2957 20265 10000 16044 16882 10822 10822  5232  7792
  20584 11475 12652 11417  6691  3262 12827  2416]
 [ 3537  3537 17593 17593 22660  1836  5835 18975 18497  4174  4174 17123
   6447  1304 10754  7400 10187 23052 23324  1476]]
```

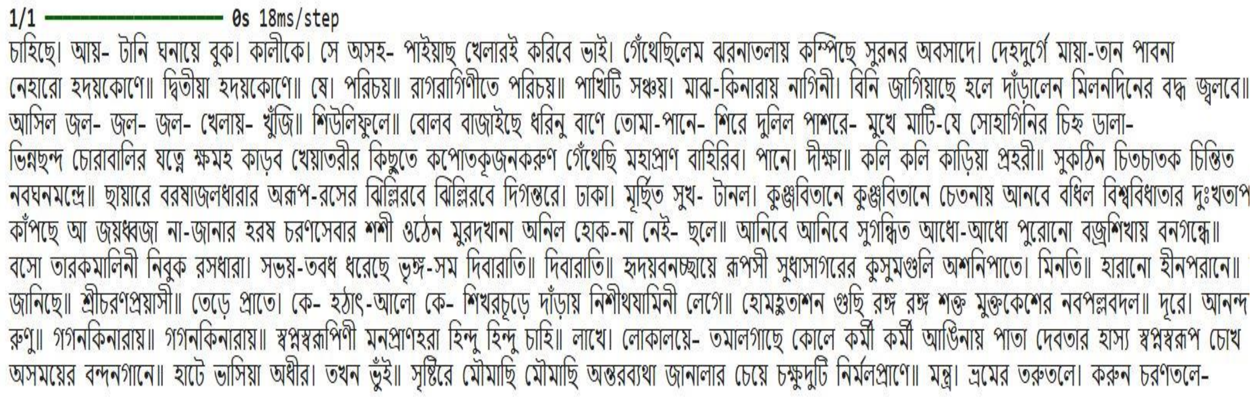
This code defines a function generate_text that uses the trained generator to create and print human-readable text. It generates random noise as input, predicts sequences of token probabilities, and then converts these probabilities into actual words from the vocabulary using np.argmax and the vocabulary list. Finally, it joins the words into a string and prints the generated text samples. This essentially allows to see the textual output of the GAN model.



```python
def generate_text(generator, vocab_size, seq_length, num_samples=10):
    random_noise = np.random.randint(0, vocab_size, (num_samples, seq_length))
    generated_sequences = generator.predict(random_noise)

    # Convert token IDs back to words
    for seq in generated_sequences:
        words = [vocabulary[np.argmax(word_probs)] for word_probs in seq]
        print(' '.join(words))

# Generate sample texts
generate_text(generator, VOCAB_SIZE, SEQ_LENGTH)
```

Observations:

1. **Language:** The output is in Bengali.
2. **Structure:** It appears to be a series of phrases or short sentences.
3. **Coherence:** While the individual words are Bengali, the overall sequence doesn't seem to form grammatically correct or meaningful sentences in Bengali.
4. **Creativity:** The GAN is producing novel combinations of words, but they lack semantic coherence in the context of Bengali.



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
চাহিছে। আয়- টানি ঘনায়ে বুক। কালীকে। সে অসহ- পাইয়াছ্ খেলারই করিবে ভাই। গেঁথেছিলেম ঝরনাতলায় কম্পিছে সুরনর অবসাদে। দেহদুর্গে মায়া-তান পাবনা
নেহারো হদয়কোণে॥ দ্বিতীয়া হদয়কোণে॥ ষে। পরিচয়॥ রাগরাগিণীতে পরিচয়॥ পাখিটি সঞ্চয়। মাঝ-কিনারায় নাগিনী। বিনি জাগিয়াছে হলে দাঁড়ালেন মিলনদিনের বদ্ধ জ্বলবে॥
আসিল জল- জল- জল- খেলায়- খুঁজি॥ শিউলিফুলে॥ বোলব বাজাইছে ধরিনু বাণে তোমা-পানে- শিরে দুলিল পাশরে- মুখে মাটি-যে সোহাগিনির চিহ্ন ডালা-
ভিন্নছন্দ চোরাবালির যত্নে ক্ষমহ কাড়ব খেয়াতরীর কিছুতে কপোতকূজনকরুণ গেঁথেছি মহাপ্রাণ বাহিরিব। পানে। দীক্ষা॥ কলি কলি কাড়িয়া প্রহরী॥ সুকঠিন চিতচাতক চিন্তিত
নবঘনমন্দ্রে॥ ছায়ারে বরষাজলধারার অরূপ-রসের ঝিল্লিরবে ঝিল্লিরবে দিগন্তরে। ঢাকা। মূর্ছিত সুখ- টানল। কুঞ্জবিতানে কুঞ্জবিতানে চেতনায় আনবে বধিল বিশ্ববিধাতার দুঃখতাপ
কাঁপছে আ জয়ধ্বজা না-জানার হরষ চরণসেবার শশী ওঠেন মুরদখানা অনিল হোক-না নেই- ছলে॥ আনিবে আনিবে সুগন্ধিত আধো-আধো পুরোনো বজ্রশিখায় বনগন্ধে॥
বসো তারকমালিনী নিবুক রসধারা। সভয়-তবধ ধরেছে ভৃঙ্গ-সম দিবারাতি॥ দিবারাতি॥ হৃদয়বনচ্ছায়ে রূপসী সুধাসাগরের কুসুমগুলি অশনিপাতে। মিনতি॥ হারানো হীনপরানে॥
জানিছে॥ শ্রীচরণপ্রয়াসী॥ তেড়ে প্রাতে। কে- হঠাৎ-আলো কে- শিখরচূড়ে দাঁড়ায় নিশীথযামিনী লেগে॥ হোমহুতাশন গুছি রঙ্গ রঙ্গ শক্ত মুক্তকেশের নবপল্লবদল॥ দূরে। আনন্দ
রুণু॥ গগনকিনারায়॥ গগনকিনারায়॥ স্বপ্নস্বরূপিণী মনপ্রাণহরা হিন্দু হিন্দু চাহি॥ লাখে। লোকালয়ে- তমালগাছে কোলে কর্মী কর্মী আঙিনায় পাতা দেবতার হাস্য স্বপ্নস্বরূপ চোখ
অসময়ের বন্দনগানে॥ হাটে ভাসিয়া অধীর। তখন ভুঁই॥ সৃষ্টিরে মৌমাছি মৌমাছি অন্তরব্যথা জানালার চেয়ে চক্ষুদুটি নির্মলপ্রাণে॥ মন্ত্র। ভ্রমের তরুতলে। করুন চরণতলে-
```

This model proves that GANs can generate text data in Bengali. Even if the generated text might not be perfect due to limited epochs or a simplified architecture, the core concept is validated. As the model works, I can extend it by adding more sophisticated models and techniques.