6. A startup wants to train a text generation model but has access to a small dataset.

a. what is the process of text generation using RNN?

b. How does transfer learning help in text generation tasks? Why is it beneficial compared to training an RNN from scratch?

c. Build a text classifier using a fine tuned Pre-Trained model to your own dataset.

Answer

a)Recurrent Neural Networks (RNNs) are designed to handle sequential data, which makes them suitable for tasks like text generation. Here's a simplified overview of the process:

Input Encoding:

- The text is tokenized (split into words or characters) and converted into numerical representations like one-hot encoding or embeddings.

Sequential Processing:

- The RNN processes one token at a time, maintaining a hidden state that captures information about previously processed tokens.
- Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) are common variants of RNN that help manage long-term dependencies better.

Token Prediction:

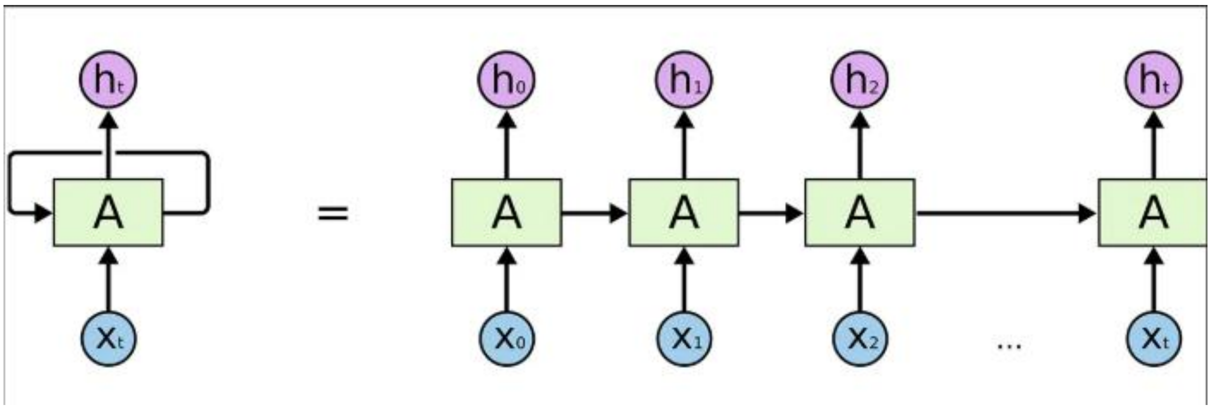
- At each time step, the model predicts the next token using the hidden state.

Training:

- The RNN is trained to minimize the difference (loss) between the predicted tokens and the actual tokens in the dataset.

Text Generation:

- During inference, a seed text (starting phrase) is provided, and the RNN generates new tokens sequentially by predicting one token at a time.



b)Transfer learning involves taking a pre-trained model (trained on a large dataset) and fine-tuning it on a smaller dataset for a specific task. For text generation:

1. **Pre-trained Language Models**:

- Models like GPT or BERT are pre-trained on extensive corpora, capturing a wide variety of language structures, grammar, and context.

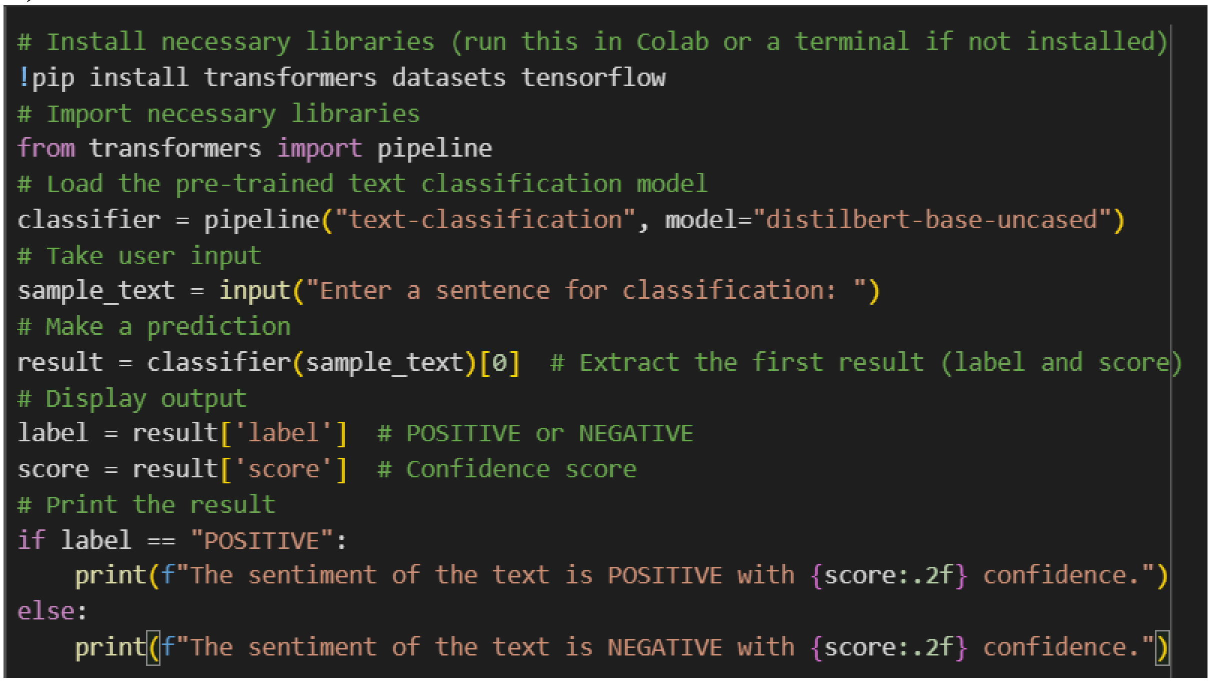
2. **Fine-tuning**:

- The startup can fine-tune the pre-trained model using its small dataset, focusing on domain-specific nuances.

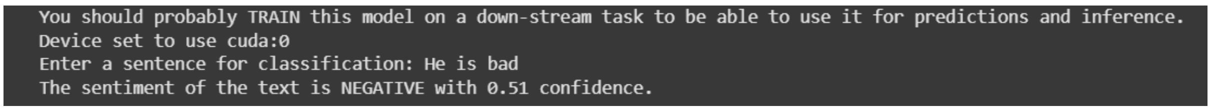
Why is Transfer Learning Beneficial Compared to Training an RNN from Scratch?

- **Data Efficiency**: Pre-trained models already understand general language patterns, so fine-tuning requires less data.
- **Faster Training**: Since the base model is pre-trained, only a small amount of training is needed to adapt it.
- **Better Performance**: Pre-trained models often achieve superior performance, especially with limited data, compared to training from scratch.

C)



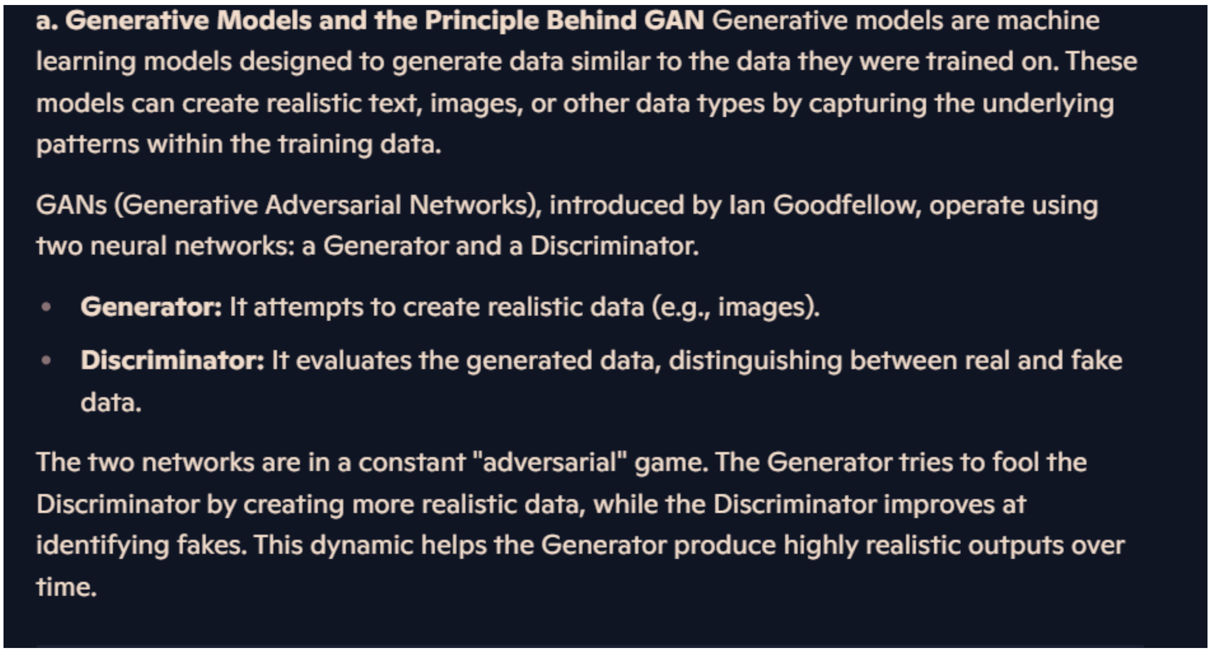
```python
# Install necessary libraries (run this in Colab or a terminal if not installed)
!pip install transformers datasets tensorflow
# Import necessary libraries
from transformers import pipeline
# Load the pre-trained text classification model
classifier = pipeline("text-classification", model="distilbert-base-uncased")
# Take user input
sample_text = input("Enter a sentence for classification: ")
# Make a prediction
result = classifier(sample_text)[0]  # Extract the first result (label and score)
# Display output
label = result['label']  # POSITIVE or NEGATIVE
score = result['score']  # Confidence score
# Print the result
if label == "POSITIVE":
    print(f"The sentiment of the text is POSITIVE with {score:.2f} confidence.")
else:
    print(f"The sentiment of the text is NEGATIVE with {score:.2f} confidence.")
```



```
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Device set to use cuda:0
Enter a sentence for classification: He is bad
The sentiment of the text is NEGATIVE with 0.51 confidence.
```

7. A company wants to generate realistic product descriptions using AI.

a. what are generative models and what is the principle behind the GAN model.

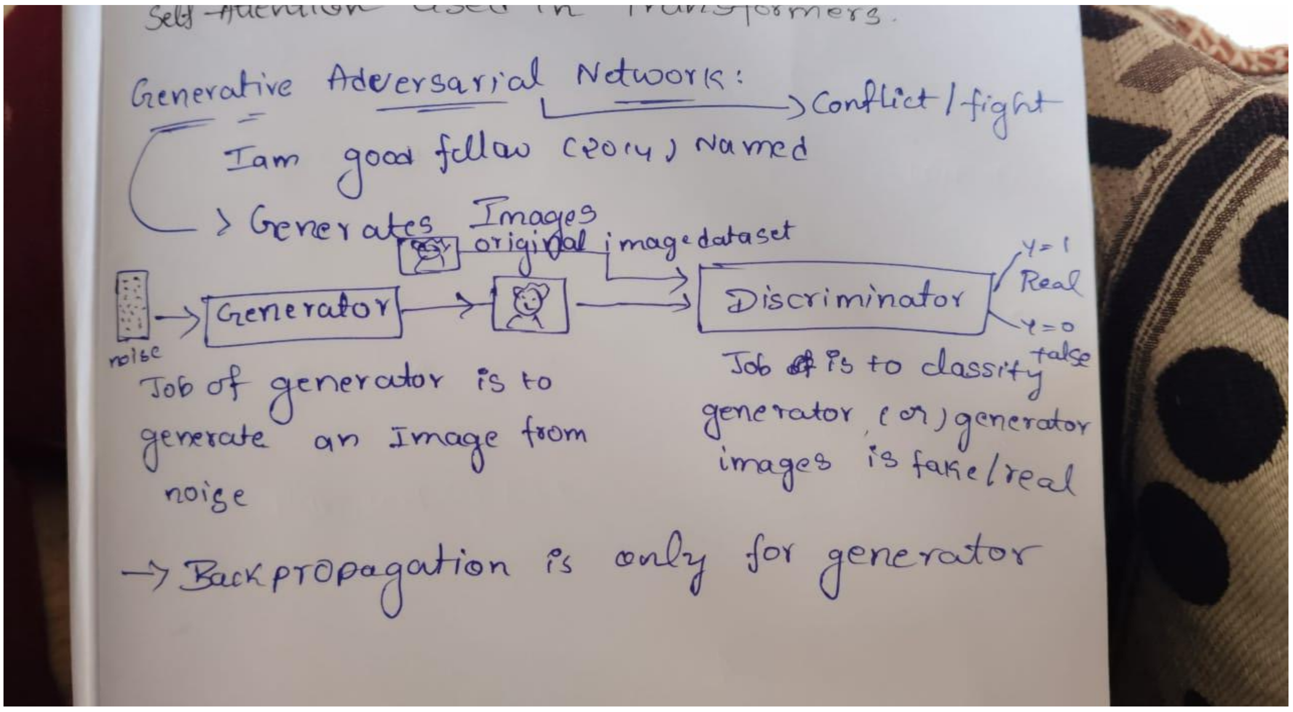


a. Generative Models and the Principle Behind GAN Generative models are machine learning models designed to generate data similar to the data they were trained on. These models can create realistic text, images, or other data types by capturing the underlying patterns within the training data.

GANs (Generative Adversarial Networks), introduced by Ian Goodfellow, operate using two neural networks: a Generator and a Discriminator.

- **Generator:** It attempts to create realistic data (e.g., images).
- **Discriminator:** It evaluates the generated data, distinguishing between real and fake data.

The two networks are in a constant "adversarial" game. The Generator tries to fool the Discriminator by creating more realistic data, while the Discriminator improves at identifying fakes. This dynamic helps the Generator produce highly realistic outputs over time.



Self-Attention used in Transformers.

Generative Adeversarial Network: → Conflict / fight

Iam good fellow (2014) Named

> Generates Images

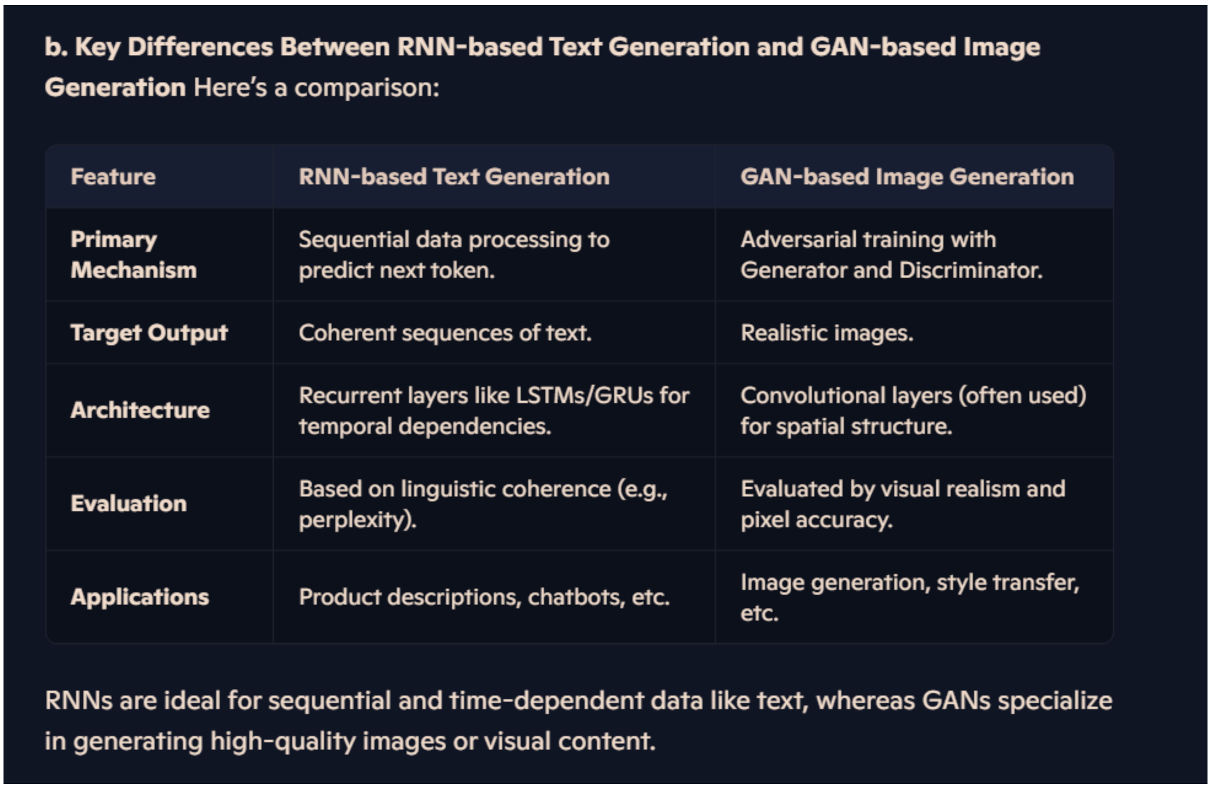
original image dataset

noise → Generator → → Discriminator → y=1 Real / y=0 false

Job of generator is to generate an Image from noise

Job ~~of~~ is to classify generator (or) generator images is fake/real

→ Backpropagation is only for generator

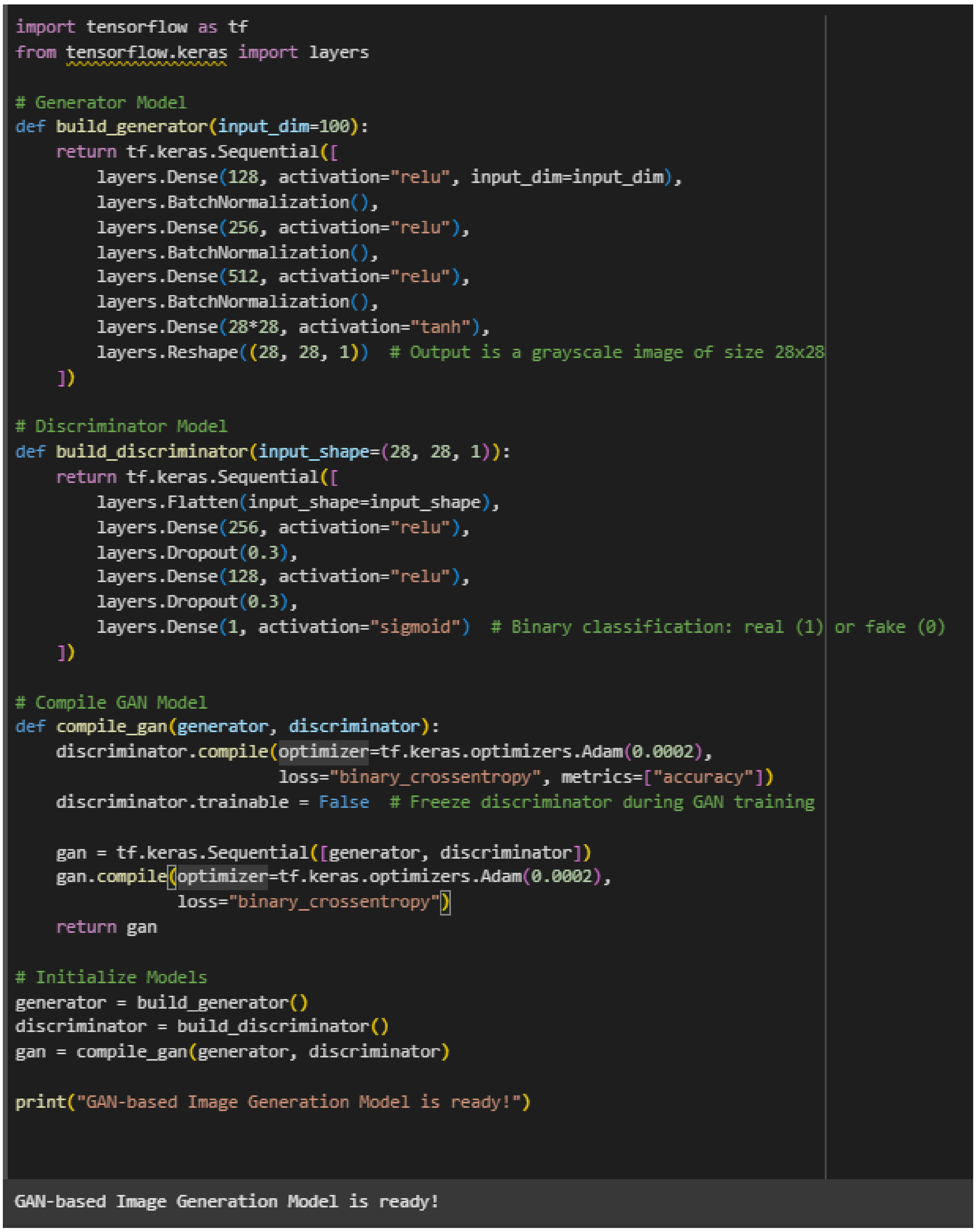


b. Key Differences Between RNN-based Text Generation and GAN-based Image Generation Here's a comparison:

Feature	RNN-based Text Generation	GAN-based Image Generation
Primary Mechanism	Sequential data processing to predict next token.	Adversarial training with Generator and Discriminator.
Target Output	Coherent sequences of text.	Realistic images.
Architecture	Recurrent layers like LSTMs/GRUs for temporal dependencies.	Convolutional layers (often used) for spatial structure.
Evaluation	Based on linguistic coherence (e.g., perplexity).	Evaluated by visual realism and pixel accuracy.
Applications	Product descriptions, chatbots, etc.	Image generation, style transfer, etc.

RNNs are ideal for sequential and time-dependent data like text, whereas GANs specialize in generating high-quality images or visual content.

c. Build a Generative Adversarial Network based Image Generation Model and discuss its architecture.a

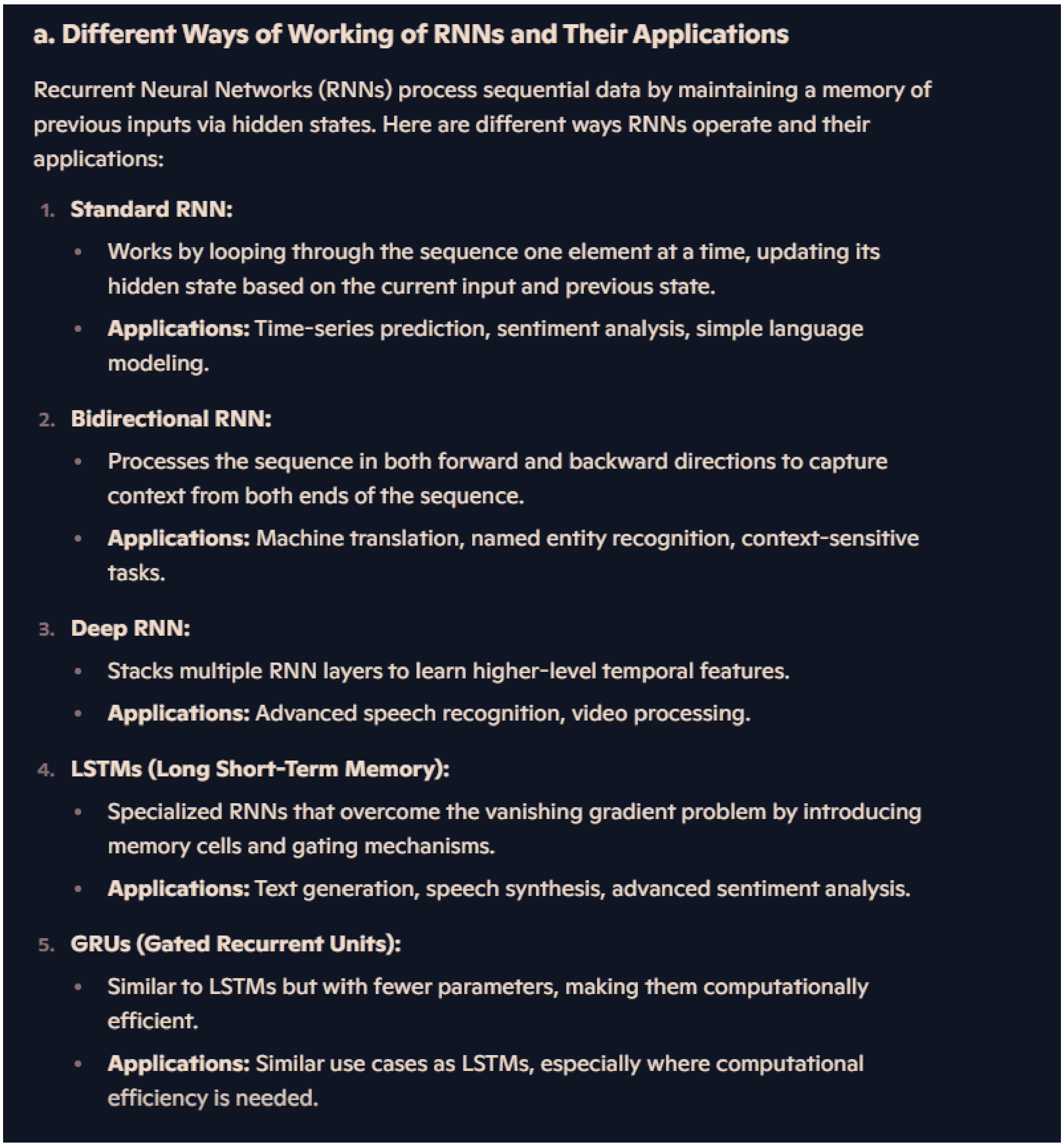


```python
import tensorflow as tf
from tensorflow.keras import layers

# Generator Model
def build_generator(input_dim=100):
    return tf.keras.Sequential([
        layers.Dense(128, activation="relu", input_dim=input_dim),
        layers.BatchNormalization(),
        layers.Dense(256, activation="relu"),
        layers.BatchNormalization(),
        layers.Dense(512, activation="relu"),
        layers.BatchNormalization(),
        layers.Dense(28*28, activation="tanh"),
        layers.Reshape((28, 28, 1))  # Output is a grayscale image of size 28x28
    ])

# Discriminator Model
def build_discriminator(input_shape=(28, 28, 1)):
    return tf.keras.Sequential([
        layers.Flatten(input_shape=input_shape),
        layers.Dense(256, activation="relu"),
        layers.Dropout(0.3),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.3),
        layers.Dense(1, activation="sigmoid")  # Binary classification: real (1) or fake (0)
    ])

# Compile GAN Model
def compile_gan(generator, discriminator):
    discriminator.compile(optimizer=tf.keras.optimizers.Adam(0.0002),
                          loss="binary_crossentropy", metrics=["accuracy"])
    discriminator.trainable = False  # Freeze discriminator during GAN training

    gan = tf.keras.Sequential([generator, discriminator])
    gan.compile(optimizer=tf.keras.optimizers.Adam(0.0002),
                loss="binary_crossentropy")
    return gan

# Initialize Models
generator = build_generator()
discriminator = build_discriminator()
gan = compile_gan(generator, discriminator)

print("GAN-based Image Generation Model is ready!")
```

```
GAN-based Image Generation Model is ready!
```

8. A team is training an RNN-based language model, but training becomes unstable as the sequence length increases.

a. what are the different ways of working of RNN and their applications.

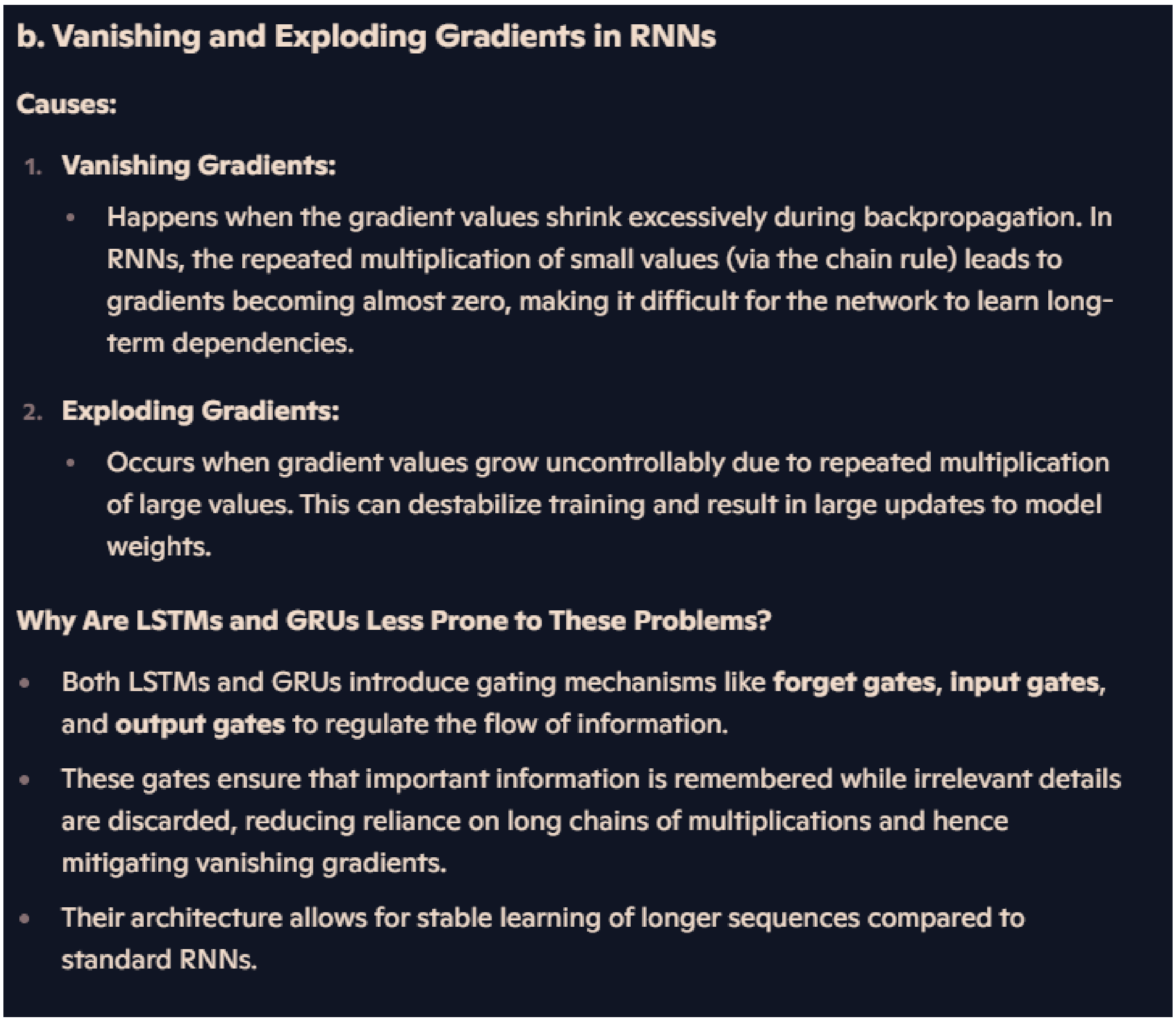


a. Different Ways of Working of RNNs and Their Applications

Recurrent Neural Networks (RNNs) process sequential data by maintaining a memory of previous inputs via hidden states. Here are different ways RNNs operate and their applications:

1. **Standard RNN:**
   - Works by looping through the sequence one element at a time, updating its hidden state based on the current input and previous state.
   - **Applications:** Time-series prediction, sentiment analysis, simple language modeling.
2. **Bidirectional RNN:**
   - Processes the sequence in both forward and backward directions to capture context from both ends of the sequence.
   - **Applications:** Machine translation, named entity recognition, context-sensitive tasks.
3. **Deep RNN:**
   - Stacks multiple RNN layers to learn higher-level temporal features.
   - **Applications:** Advanced speech recognition, video processing.
4. **LSTMs (Long Short-Term Memory):**
   - Specialized RNNs that overcome the vanishing gradient problem by introducing memory cells and gating mechanisms.
   - **Applications:** Text generation, speech synthesis, advanced sentiment analysis.
5. **GRUs (Gated Recurrent Units):**
   - Similar to LSTMs but with fewer parameters, making them computationally efficient.
   - **Applications:** Similar use cases as LSTMs, especially where computational efficiency is needed.

b. What causes vanishing and exploding gradients in RNNs? Why are LSTMs and GRUs less prone to this problem?



b. Vanishing and Exploding Gradients in RNNs

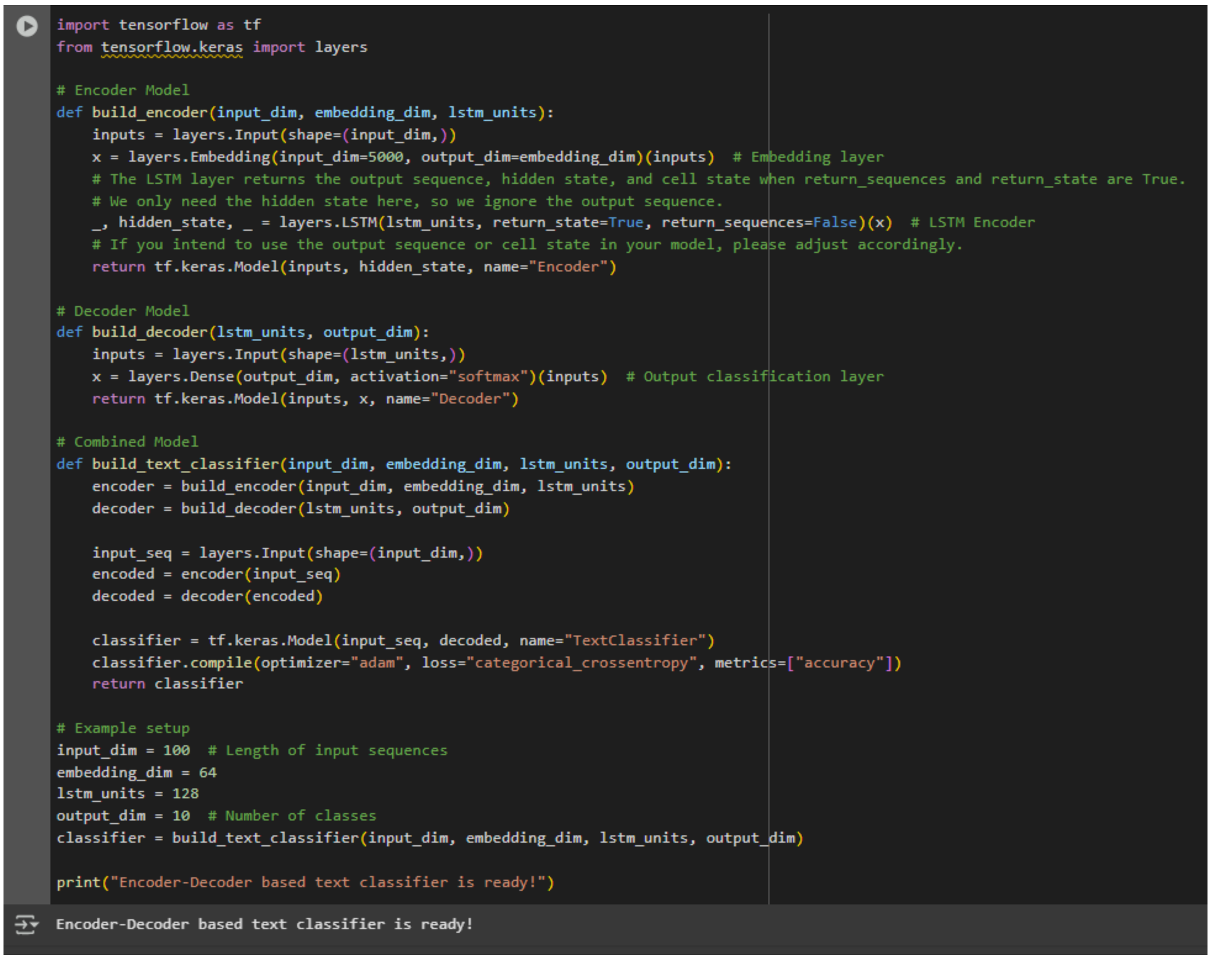
Causes:

1. **Vanishing Gradients:**
   - Happens when the gradient values shrink excessively during backpropagation. In RNNs, the repeated multiplication of small values (via the chain rule) leads to gradients becoming almost zero, making it difficult for the network to learn long-term dependencies.
2. **Exploding Gradients:**
   - Occurs when gradient values grow uncontrollably due to repeated multiplication of large values. This can destabilize training and result in large updates to model weights.

Why Are LSTMs and GRUs Less Prone to These Problems?

- Both LSTMs and GRUs introduce gating mechanisms like **forget gates, input gates**, and **output gates** to regulate the flow of information.
- These gates ensure that important information is remembered while irrelevant details are discarded, reducing reliance on long chains of multiplications and hence mitigating vanishing gradients.
- Their architecture allows for stable learning of longer sequences compared to standard RNNs.

c. Build a encoder-decoder based text classifier discuss its working procedure.

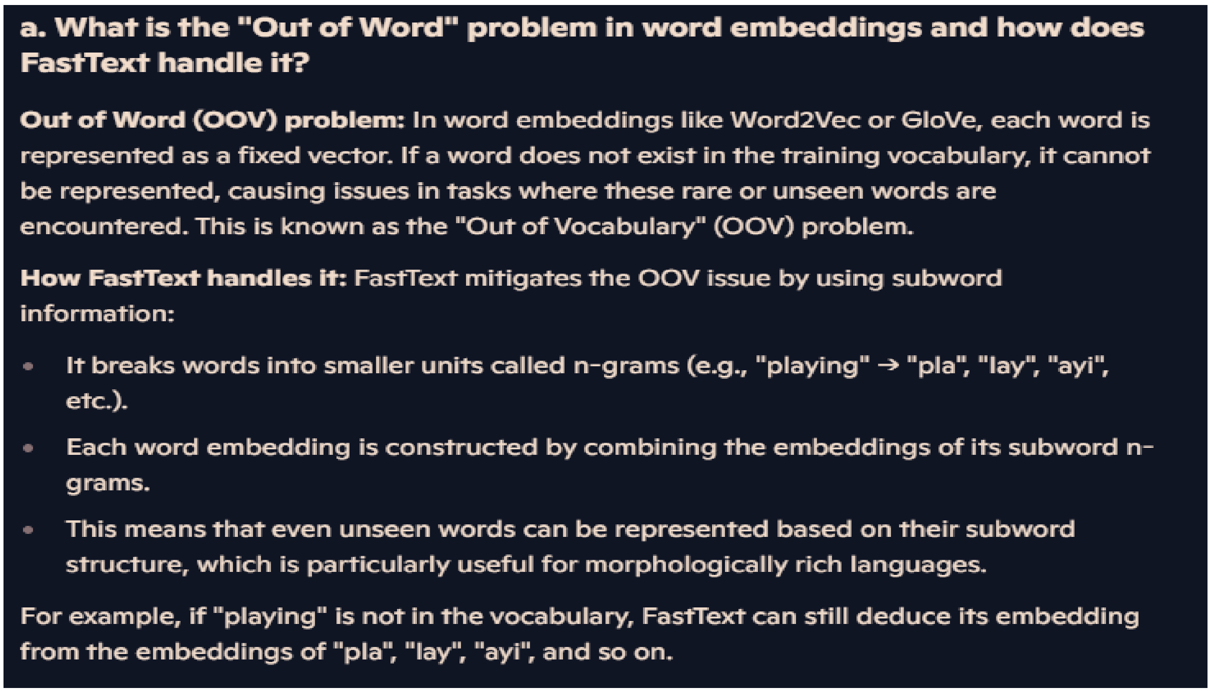


```
import tensorflow as tf
from tensorflow.keras import layers

# Encoder Model
def build_encoder(input_dim, embedding_dim, lstm_units):
    inputs = layers.Input(shape=(input_dim,))
    x = layers.Embedding(input_dim=5000, output_dim=embedding_dim)(inputs)  # Embedding layer
    # The LSTM layer returns the output sequence, hidden state, and cell state when return_sequences and return_state are True.
    # We only need the hidden state here, so we ignore the output sequence.
    _, hidden_state, _ = layers.LSTM(lstm_units, return_state=True, return_sequences=False)(x)  # LSTM Encoder
    # If you intend to use the output sequence or cell state in your model, please adjust accordingly.
    return tf.keras.Model(inputs, hidden_state, name="Encoder")

# Decoder Model
def build_decoder(lstm_units, output_dim):
    inputs = layers.Input(shape=(lstm_units,))
    x = layers.Dense(output_dim, activation="softmax")(inputs)  # Output classification layer
    return tf.keras.Model(inputs, x, name="Decoder")

# Combined Model
def build_text_classifier(input_dim, embedding_dim, lstm_units, output_dim):
    encoder = build_encoder(input_dim, embedding_dim, lstm_units)
    decoder = build_decoder(lstm_units, output_dim)

    input_seq = layers.Input(shape=(input_dim,))
    encoded = encoder(input_seq)
    decoded = decoder(encoded)

    classifier = tf.keras.Model(input_seq, decoded, name="TextClassifier")
    classifier.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])
    return classifier

# Example setup
input_dim = 100  # Length of input sequences
embedding_dim = 64
lstm_units = 128
output_dim = 10  # Number of classes
classifier = build_text_classifier(input_dim, embedding_dim, lstm_units, output_dim)

print("Encoder-Decoder based text classifier is ready!")
```

```
Encoder-Decoder based text classifier is ready!
```

9. A text processing system frequently encounters rare words that were not present in the training vocabulary.

a. What is an out of word problem in word embeddings how it is handled by fast text?



a. What is the "Out of Word" problem in word embeddings and how does FastText handle it?

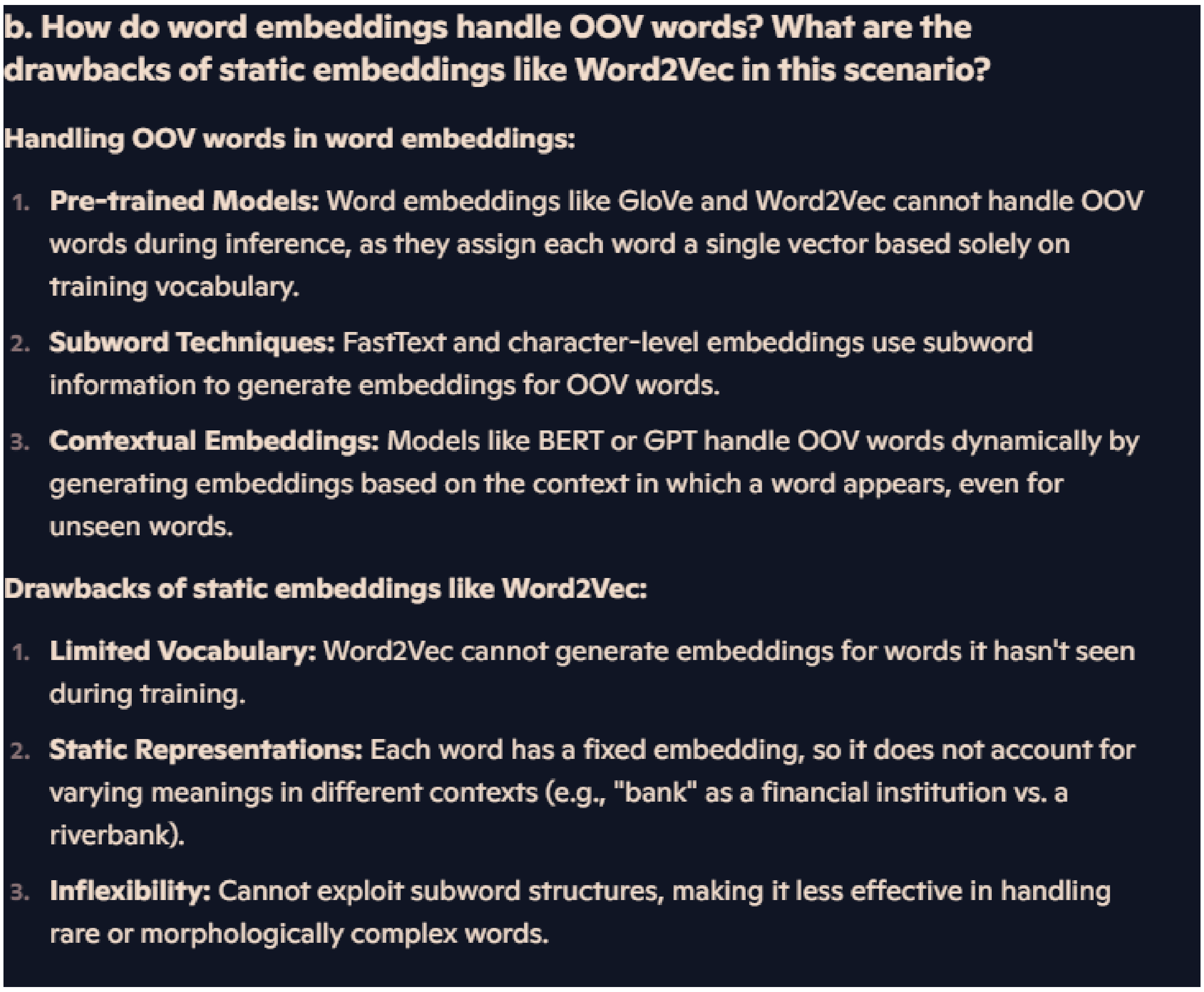
Out of Word (OOV) problem: In word embeddings like Word2Vec or GloVe, each word is represented as a fixed vector. If a word does not exist in the training vocabulary, it cannot be represented, causing issues in tasks where these rare or unseen words are encountered. This is known as the "Out of Vocabulary" (OOV) problem.

How FastText handles it: FastText mitigates the OOV issue by using subword information:

- It breaks words into smaller units called n-grams (e.g., "playing" → "pla", "lay", "ayi", etc.).
- Each word embedding is constructed by combining the embeddings of its subword n-grams.
- This means that even unseen words can be represented based on their subword structure, which is particularly useful for morphologically rich languages.

For example, if "playing" is not in the vocabulary, FastText can still deduce its embedding from the embeddings of "pla", "lay", "ayi", and so on.

b. How do word embeddings handle out-of-vocabulary (OOV) words? What are the drawbacks of static embeddings like Word2Vec in this scenario?



b. How do word embeddings handle OOV words? What are the drawbacks of static embeddings like Word2Vec in this scenario?

Handling OOV words in word embeddings:

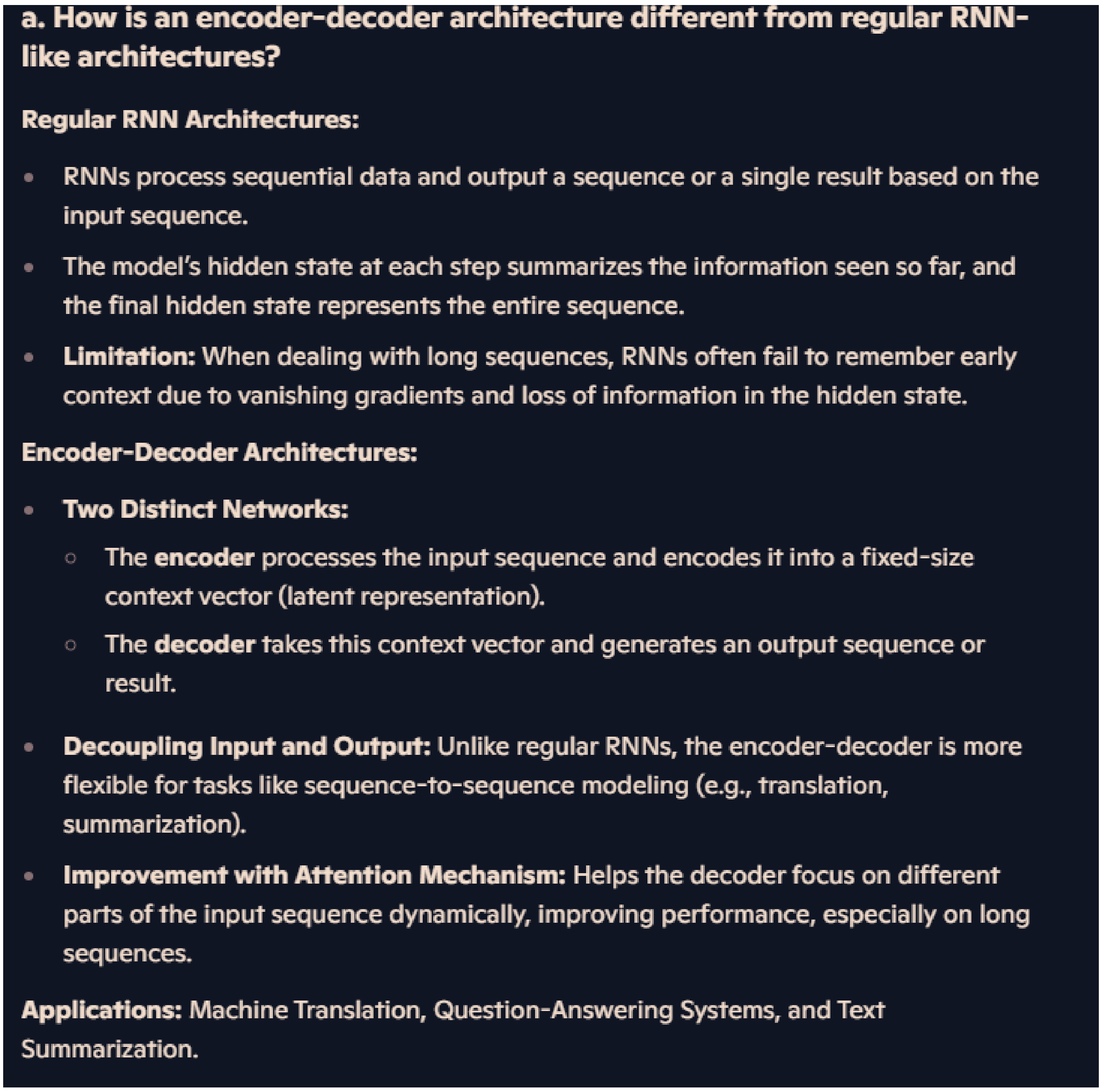
1. **Pre-trained Models:** Word embeddings like GloVe and Word2Vec cannot handle OOV words during inference, as they assign each word a single vector based solely on training vocabulary.
2. **Subword Techniques:** FastText and character-level embeddings use subword information to generate embeddings for OOV words.
3. **Contextual Embeddings:** Models like BERT or GPT handle OOV words dynamically by generating embeddings based on the context in which a word appears, even for unseen words.

Drawbacks of static embeddings like Word2Vec:

1. **Limited Vocabulary:** Word2Vec cannot generate embeddings for words it hasn't seen during training.
2. **Static Representations:** Each word has a fixed embedding, so it does not account for varying meanings in different contexts (e.g., "bank" as a financial institution vs. a riverbank).
3. **Inflexibility:** Cannot exploit subword structures, making it less effective in handling rare or morphologically complex words.

c. Build Text classifier with custom word2vec embeddings.

10. a. How an encoder-decoder architecture is different from regular RNN like architectures?



a. How is an encoder-decoder architecture different from regular RNN-like architectures?

Regular RNN Architectures:

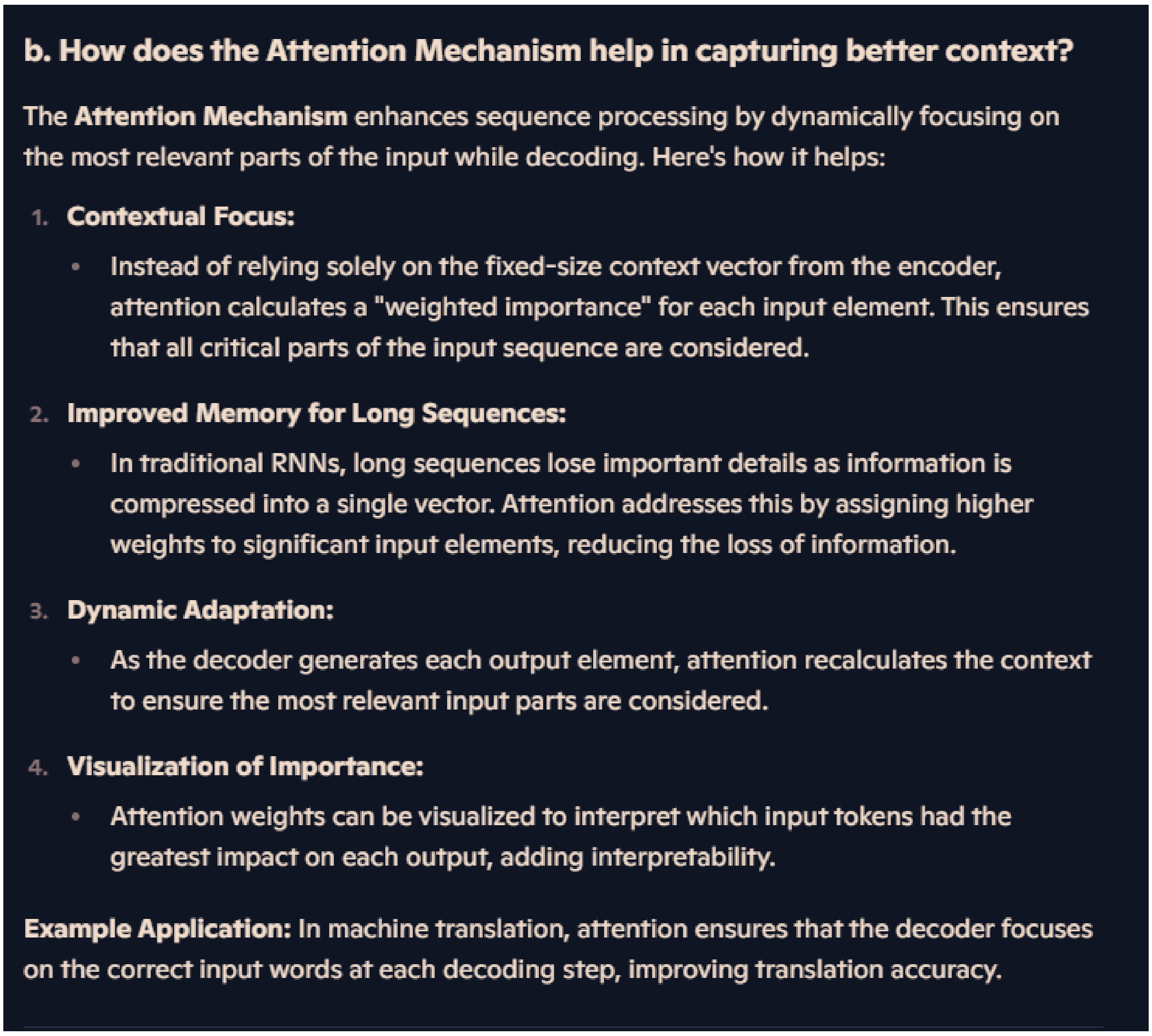
- RNNs process sequential data and output a sequence or a single result based on the input sequence.
- The model's hidden state at each step summarizes the information seen so far, and the final hidden state represents the entire sequence.
- **Limitation:** When dealing with long sequences, RNNs often fail to remember early context due to vanishing gradients and loss of information in the hidden state.

Encoder-Decoder Architectures:

- **Two Distinct Networks:**
  - The **encoder** processes the input sequence and encodes it into a fixed-size context vector (latent representation).
  - The **decoder** takes this context vector and generates an output sequence or result.
- **Decoupling Input and Output:** Unlike regular RNNs, the encoder-decoder is more flexible for tasks like sequence-to-sequence modeling (e.g., translation, summarization).
- **Improvement with Attention Mechanism:** Helps the decoder focus on different parts of the input sequence dynamically, improving performance, especially on long sequences.

Applications: Machine Translation, Question-Answering Systems, and Text Summarization.

b. Analyze how an Attention mechanism helps in capturing better context



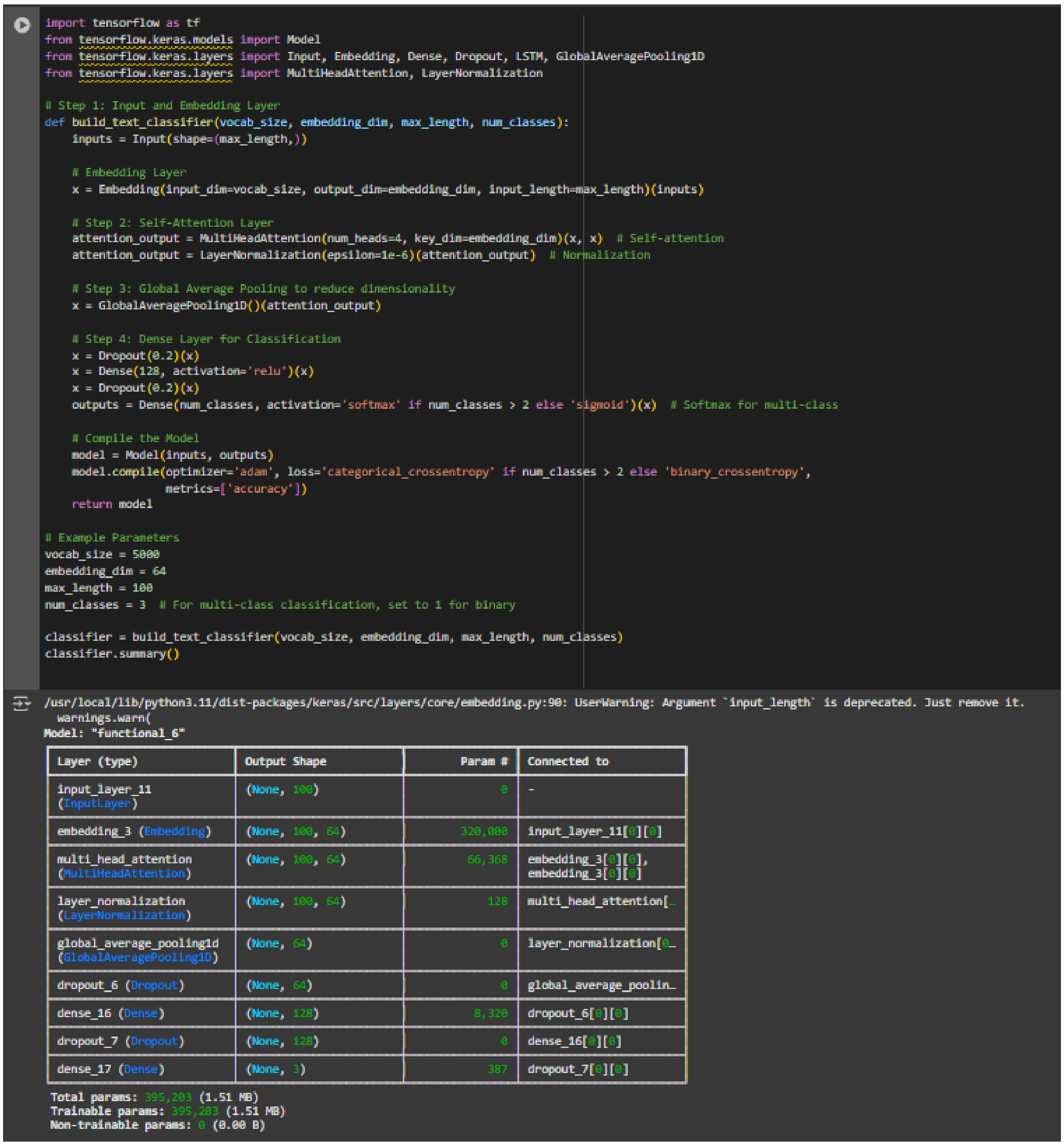
b. How does the Attention Mechanism help in capturing better context?

The **Attention Mechanism** enhances sequence processing by dynamically focusing on the most relevant parts of the input while decoding. Here's how it helps:

1. **Contextual Focus:**
   - Instead of relying solely on the fixed-size context vector from the encoder, attention calculates a "weighted importance" for each input element. This ensures that all critical parts of the input sequence are considered.
2. **Improved Memory for Long Sequences:**
   - In traditional RNNs, long sequences lose important details as information is compressed into a single vector. Attention addresses this by assigning higher weights to significant input elements, reducing the loss of information.
3. **Dynamic Adaptation:**
   - As the decoder generates each output element, attention recalculates the context to ensure the most relevant input parts are considered.
4. **Visualization of Importance:**
   - Attention weights can be visualized to interpret which input tokens had the greatest impact on each output, adding interpretability.

Example Application: In machine translation, attention ensures that the decoder focuses on the correct input words at each decoding step, improving translation accuracy.

c. Build a Text Classifier using self-attention layers and discuss the flow of the text classification.



```python
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Dense, Dropout, LSTM, GlobalAveragePooling1D
from tensorflow.keras.layers import MultiHeadAttention, LayerNormalization

# Step 1: Input and Embedding Layer
def build_text_classifier(vocab_size, embedding_dim, max_length, num_classes):
    inputs = Input(shape=(max_length,))

    # Embedding Layer
    x = Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length)(inputs)

    # Step 2: Self-Attention Layer
    attention_output = MultiHeadAttention(num_heads=4, key_dim=embedding_dim)(x, x)  # Self-attention
    attention_output = LayerNormalization(epsilon=1e-6)(attention_output)  # Normalization

    # Step 3: Global Average Pooling to reduce dimensionality
    x = GlobalAveragePooling1D()(attention_output)

    # Step 4: Dense Layer for Classification
    x = Dropout(0.2)(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.2)(x)
    outputs = Dense(num_classes, activation='softmax' if num_classes > 2 else 'sigmoid')(x)  # Softmax for multi-class

    # Compile the Model
    model = Model(inputs, outputs)
    model.compile(optimizer='adam', loss='categorical_crossentropy' if num_classes > 2 else 'binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Example Parameters
vocab_size = 5000
embedding_dim = 64
max_length = 100
num_classes = 3  # For multi-class classification, set to 1 for binary

classifier = build_text_classifier(vocab_size, embedding_dim, max_length, num_classes)
classifier.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
Model: "functional_6"
┌───────────────────────────┬────────────────────────┬────────────────┬────────────────────────┐
│ Layer (type)              │ Output Shape           │        Param # │ Connected to           │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ input_layer_11            │ (None, 100)            │              0 │ -                      │
│ (InputLayer)              │                        │                │                        │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ embedding_3 (Embedding)   │ (None, 100, 64)        │        320,000 │ input_layer_11[0][0]   │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ multi_head_attention      │ (None, 100, 64)        │         66,368 │ embedding_3[0][0],     │
│ (MultiHeadAttention)      │                        │                │ embedding_3[0][0]      │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ layer_normalization       │ (None, 100, 64)        │            128 │ multi_head_attention[… │
│ (LayerNormalization)      │                        │                │                        │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ global_average_pooling1d  │ (None, 64)             │              0 │ layer_normalization[0… │
│ (GlobalAveragePooling1D)  │                        │                │                        │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ dropout_6 (Dropout)       │ (None, 64)             │              0 │ global_average_poolin… │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ dense_16 (Dense)          │ (None, 128)            │          8,320 │ dropout_6[0][0]        │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ dropout_7 (Dropout)       │ (None, 128)            │              0 │ dense_16[0][0]         │
├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤
│ dense_17 (Dense)          │ (None, 3)              │            387 │ dropout_7[0][0]        │
└───────────────────────────┴────────────────────────┴────────────────┴────────────────────────┘
 Total params: 395,203 (1.51 MB)
 Trainable params: 395,203 (1.51 MB)
 Non-trainable params: 0 (0.00 B)
```