

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
import random
```

```
2026-02-05 10:23:30.638649: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:467] Unable to
register cuFFT factory: Attempting to register factory for plugin
cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
E0000 00:00:1770287010.814596      55 cuda_dnn.cc:8579] Unable to
register cuDNN factory: Attempting to register factory for plugin
cuDNN when one has already been registered
E0000 00:00:1770287010.867149      55 cuda_blas.cc:1407] Unable to
register cuBLAS factory: Attempting to register factory for plugin
cuBLAS when one has already been registered
W0000 00:00:1770287011.273873      55 computation_placer.cc:177]
computation placer already registered. Please check linkage and avoid
linking the same target more than once.
W0000 00:00:1770287011.273908      55 computation_placer.cc:177]
computation placer already registered. Please check linkage and avoid
linking the same target more than once.
W0000 00:00:1770287011.273911      55 computation_placer.cc:177]
computation placer already registered. Please check linkage and avoid
linking the same target more than once.
W0000 00:00:1770287011.273913      55 computation_placer.cc:177]
computation placer already registered. Please check linkage and avoid
linking the same target more than once.
```

```
data = """
artificial intelligence is transforming modern society.
it is used in healthcare finance education and transportation.
machine learning allows systems to improve automatically with
experience.
data plays a critical role in training intelligent systems.
large datasets help models learn complex patterns.
deep learning uses multi layer neural networks.
neural networks are inspired by biological neurons.
each neuron processes input and produces an output.
training a neural network requires optimization techniques.
gradient descent minimizes the loss function.

natural language processing helps computers understand human language.
text generation is a key task in nlp.
language models predict the next word or character.
recurrent neural networks handle sequential data.
```

lstm and gru models address long term dependency problems.

transformer models changed the field of nlp.
they rely on self attention mechanisms.
attention allows the model to focus on relevant context.

education is being improved using artificial intelligence.
intelligent tutoring systems personalize learning.

ethical considerations are important in artificial intelligence.
ai systems should be designed responsibly.

text generation models can create stories poems and articles.
generated text should be meaningful and coherent.

continuous learning is essential in the field of ai.
programming skills are important for ai engineers.

"""

N-Gram Model

```
words = data.lower().replace("\n", " ").split()

ngram_model = {}

for i in range(len(words)-1):
    w1 = words[i]
    w2 = words[i+1]

    if w1 not in ngram_model:
        ngram_model[w1] = []

    ngram_model[w1].append(w2)

print("Total Keys in Ngram Model:", len(ngram_model))

Total Keys in Ngram Model: 139

def generate_ngram_text(seed, n_words=20):
    result = [seed]

    for _ in range(n_words):
        last = result[-1]
        if last in ngram_model:
            next_word = random.choice(ngram_model[last])
            result.append(next_word)
        else:
            break

    return " ".join(result)
```

```
print(generate_ngram_text("artificial", 25))
```

artificial intelligence. intelligent tutoring systems personalize learning. ethical considerations are inspired by biological neurons. each neuron processes input and coherent. continuous learning allows systems personalize learning.

LIMITATIONS OF NGRAM MODEL:

- Only looks at previous one word (short memory)
- Cannot understand long context
- Generated text may repeat or lose meaning
- No deep learning or semantic understanding

RNN Model:

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])

total_words = len(tokenizer.word_index) + 1
print("Total Vocabulary:", total_words)

Total Vocabulary: 134

input_sequences = []

for line in data.split("\n"):
    token_list = tokenizer.texts_to_sequences([line])[0]

    for i in range(1, len(token_list)):
        n_gram_seq = token_list[:i+1]
        input_sequences.append(n_gram_seq)

max_len = max(len(x) for x in input_sequences)

input_sequences = pad_sequences(input_sequences, maxlen=max_len,
padding='pre')

X = input_sequences[:, :-1]
y = input_sequences[:, -1]

y = tf.keras.utils.to_categorical(y, num_classes=total_words)

print(X.shape, y.shape)

(167, 8) (167, 134)

model = Sequential([
    Embedding(input_dim=total_words,
              output_dim=64,
```

```

        input_shape=(max_len-1,)),
        SimpleRNN(128),
        Dense(total_words, activation='softmax')
    ])

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.summary()

/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/
embedding.py:100: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(**kwargs)
I0000 00:00:1770287033.336214      55 gpu_device.cc:2019] Created
device /job:localhost/replica:0/task:0/device:GPU:0 with 13757 MB
memory: -> device: 0, name: Tesla T4, pci bus id: 0000:00:04.0,
compute capability: 7.5
I0000 00:00:1770287033.342162      55 gpu_device.cc:2019] Created
device /job:localhost/replica:0/task:0/device:GPU:1 with 13757 MB
memory: -> device: 1, name: Tesla T4, pci bus id: 0000:00:05.0,
compute capability: 7.5

Model: "sequential"

```

Layer (type) Param #	Output Shape	
embedding (Embedding) 8,576	(None, 8, 64)	
simple_rnn (SimpleRNN) 24,704	(None, 128)	
dense (Dense) 17,286	(None, 134)	

Total params: 50,566 (197.52 KB)

Trainable params: 50,566 (197.52 KB)

Non-trainable params: 0 (0.00 B)

```
history = model.fit(X, y, epochs=100, verbose=1)
```

Epoch 1/100

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR

I0000 00:00:1770287039.782705 125 service.cc:152] XLA service 0x7a4680004580 initialized for platform CUDA (this does not guarantee that XLA will be used). Devices:

I0000 00:00:1770287039.782748 125 service.cc:160] StreamExecutor device (0): Tesla T4, Compute Capability 7.5

I0000 00:00:1770287039.782754 125 service.cc:160] StreamExecutor device (1): Tesla T4, Compute Capability 7.5

I0000 00:00:1770287040.098936 125 cuda_dnn.cc:529] Loaded cuDNN version 91002

1/6 ————— 14s 3s/step - accuracy: 0.0312 - loss: 4.8933

I0000 00:00:1770287041.321151 125 device_compiler.h:188] Compiled cluster using XLA! This line is logged at most once for the lifetime of the process.

6/6 ————— 4s 210ms/step - accuracy: 0.0156 - loss: 4.9022

Epoch 2/100

6/6 ————— 0s 7ms/step - accuracy: 0.0796 - loss: 4.8116

Epoch 3/100

6/6 ————— 0s 7ms/step - accuracy: 0.1443 - loss: 4.7097

Epoch 4/100

6/6 ————— 0s 7ms/step - accuracy: 0.1289 - loss: 4.5894

Epoch 5/100

6/6 ————— 0s 7ms/step - accuracy: 0.1014 - loss: 4.5135

Epoch 6/100

6/6 ————— 0s 7ms/step - accuracy: 0.1074 - loss: 4.4213

Epoch 7/100

6/6 ————— 0s 7ms/step - accuracy: 0.1563 - loss: 4.2166

Epoch 8/100

6/6 ————— 0s 7ms/step - accuracy: 0.2265 - loss: 4.1719

Epoch 9/100

6/6 ————— 0s 7ms/step - accuracy: 0.1721 - loss: 4.0914

Epoch 10/100

6/6 ————— 0s 7ms/step - accuracy: 0.1611 - loss: 4.0109

Epoch 11/100

6/6 ————— 0s 7ms/step - accuracy: 0.1530 - loss: 3.8803

Epoch 12/100

6/6 ————— 0s 7ms/step - accuracy: 0.2120 - loss: 3.8193

Epoch 13/100

6/6 ————— 0s 7ms/step - accuracy: 0.2337 - loss: 3.6625

Epoch 14/100

6/6 ————— 0s 7ms/step - accuracy: 0.2559 - loss: 3.4858

Epoch 15/100

6/6 ————— 0s 7ms/step - accuracy: 0.3259 - loss: 3.3753

Epoch 16/100

6/6 ————— 0s 7ms/step - accuracy: 0.3962 - loss: 3.2449

Epoch 17/100

6/6 ————— 0s 7ms/step - accuracy: 0.4253 - loss: 3.0899

Epoch 18/100

6/6 ————— 0s 7ms/step - accuracy: 0.4104 - loss: 3.0397

Epoch 19/100

6/6 ————— 0s 7ms/step - accuracy: 0.4872 - loss: 2.8808

Epoch 20/100

6/6 ————— 0s 7ms/step - accuracy: 0.5324 - loss: 2.6109

Epoch 21/100

6/6 ————— 0s 8ms/step - accuracy: 0.6622 - loss: 2.4946

Epoch 22/100

6/6 ————— 0s 8ms/step - accuracy: 0.6524 - loss: 2.4148

Epoch 23/100

6/6 ————— 0s 8ms/step - accuracy: 0.6601 - loss: 2.3209

Epoch 24/100

6/6 ————— 0s 8ms/step - accuracy: 0.6841 - loss: 2.1140


Epoch 25/100

6/6 ————— 0s 8ms/step - accuracy: 0.7404 - loss: 1.9837


Epoch 26/100

6/6 ————— 0s 8ms/step - accuracy: 0.6835 - loss: 1.9439


Epoch 27/100

6/6  0s 8ms/step - accuracy: 0.7345 - loss: 1.8326


Epoch 28/100

6/6  0s 8ms/step - accuracy: 0.7758 - loss: 1.6328


Epoch 29/100

6/6  0s 8ms/step - accuracy: 0.7232 - loss: 1.5600

Epoch 30/100

6/6  0s 8ms/step - accuracy: 0.7819 - loss: 1.5176


Epoch 31/100

6/6  0s 8ms/step - accuracy: 0.8080 - loss: 1.3870

Epoch 32/100

6/6  0s 8ms/step - accuracy: 0.8325 - loss: 1.3174


Epoch 33/100

6/6  0s 8ms/step - accuracy: 0.8240 - loss: 1.2960


Epoch 34/100

6/6  0s 8ms/step - accuracy: 0.8647 - loss: 1.1160

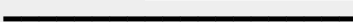
Epoch 35/100

6/6  0s 8ms/step - accuracy: 0.8520 - loss: 1.1602

Epoch 36/100

6/6  0s 8ms/step - accuracy: 0.8381 - loss: 1.1263


Epoch 37/100

6/6  0s 8ms/step - accuracy: 0.8342 - loss: 1.0644

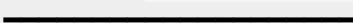
Epoch 38/100

6/6  0s 8ms/step - accuracy: 0.8181 - loss: 0.9858

Epoch 39/100

6/6  0s 9ms/step - accuracy: 0.8610 - loss: 0.9850


Epoch 40/100

6/6  0s 8ms/step - accuracy: 0.8923 - loss: 0.8999

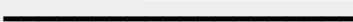
Epoch 41/100

6/6  0s 8ms/step - accuracy: 0.8997 - loss: 0.8410

Epoch 42/100

6/6  0s 8ms/step - accuracy: 0.9179 - loss: 0.8402

Epoch 43/100

6/6  0s 8ms/step - accuracy: 0.9045 - loss: 0.8198

Epoch 44/100

6/6 ————— 0s 8ms/step - accuracy: 0.9208 - loss: 0.7731

Epoch 45/100

6/6 ————— 0s 8ms/step - accuracy: 0.9393 - loss: 0.7140

Epoch 46/100

6/6 ————— 0s 8ms/step - accuracy: 0.9349 - loss: 0.6496

Epoch 47/100

6/6 ————— 0s 8ms/step - accuracy: 0.9144 - loss: 0.6779

Epoch 48/100

6/6 ————— 0s 8ms/step - accuracy: 0.9360 - loss: 0.5800

Epoch 49/100

6/6 ————— 0s 8ms/step - accuracy: 0.9364 - loss: 0.6196

Epoch 50/100

6/6 ————— 0s 8ms/step - accuracy: 0.9289 - loss: 0.5921

Epoch 51/100

6/6 ————— 0s 8ms/step - accuracy: 0.9688 - loss: 0.4813

Epoch 52/100

6/6 ————— 0s 8ms/step - accuracy: 0.9715 - loss: 0.4599

Epoch 53/100

6/6 ————— 0s 8ms/step - accuracy: 0.9680 - loss: 0.4880

Epoch 54/100

6/6 ————— 0s 8ms/step - accuracy: 0.9689 - loss: 0.4904

Epoch 55/100

6/6 ————— 0s 8ms/step - accuracy: 0.9695 - loss: 0.4127

Epoch 56/100

6/6 ————— 0s 8ms/step - accuracy: 0.9743 - loss: 0.4008

Epoch 57/100

6/6 ————— 0s 8ms/step - accuracy: 0.9797 - loss: 0.4314

Epoch 58/100

6/6 ————— 0s 8ms/step - accuracy: 0.9702 - loss: 0.3791

Epoch 59/100

6/6 ————— 0s 8ms/step - accuracy: 0.9754 - loss: 0.3400

Epoch 60/100

6/6 ————— 0s 8ms/step - accuracy: 0.9827 - loss: 0.3708

Epoch 61/100

6/6 ————— 0s 8ms/step - accuracy: 0.9769 - loss: 0.3915

Epoch 62/100

6/6 ————— 0s 10ms/step - accuracy: 0.9814 - loss: 0.3190

Epoch 63/100

6/6 ————— 0s 8ms/step - accuracy: 0.9688 - loss: 0.3395

Epoch 64/100

6/6 ————— 0s 8ms/step - accuracy: 0.9650 - loss: 0.3371

Epoch 65/100

6/6 ————— 0s 9ms/step - accuracy: 0.9717 - loss: 0.3010

Epoch 66/100

6/6 ————— 0s 8ms/step - accuracy: 0.9807 - loss: 0.3296

Epoch 67/100

6/6 ————— 0s 8ms/step - accuracy: 0.9922 - loss: 0.2604

Epoch 68/100

6/6 ————— 0s 8ms/step - accuracy: 0.9922 - loss: 0.2605

Epoch 69/100

6/6 ————— 0s 8ms/step - accuracy: 0.9801 - loss: 0.2277

Epoch 70/100

6/6 ————— 0s 8ms/step - accuracy: 0.9900 - loss: 0.2559

Epoch 71/100

6/6 ————— 0s 8ms/step - accuracy: 0.9710 - loss: 0.2247

Epoch 72/100

6/6 ————— 0s 8ms/step - accuracy: 0.9807 - loss: 0.2165

Epoch 73/100

6/6 ————— 0s 8ms/step - accuracy: 0.9881 - loss: 0.2274

Epoch 74/100

6/6 ————— 0s 8ms/step - accuracy: 0.9963 - loss: 0.1817

Epoch 75/100

6/6 ————— 0s 8ms/step - accuracy: 0.9881 - loss: 0.1913


Epoch 76/100

6/6 ————— 0s 8ms/step - accuracy: 0.9881 - loss: 0.1840


Epoch 77/100

6/6 ————— 0s 8ms/step - accuracy: 0.9974 - loss: 0.1836


Epoch 78/100

6/6  0s 8ms/step - accuracy: 0.9926 - loss: 0.1664


Epoch 79/100

6/6  0s 8ms/step - accuracy: 0.9948 - loss: 0.1709


Epoch 80/100

6/6  0s 8ms/step - accuracy: 0.9926 - loss: 0.1756


Epoch 81/100

6/6  0s 8ms/step - accuracy: 0.9926 - loss: 0.1287


Epoch 82/100

6/6  0s 8ms/step - accuracy: 0.9963 - loss: 0.1424

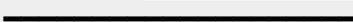
Epoch 83/100

6/6  0s 8ms/step - accuracy: 0.9963 - loss: 0.1340


Epoch 84/100

6/6  0s 8ms/step - accuracy: 0.9948 - loss: 0.1374


Epoch 85/100

6/6  0s 8ms/step - accuracy: 0.9974 - loss: 0.1348

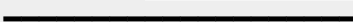
Epoch 86/100

6/6  0s 8ms/step - accuracy: 0.9974 - loss: 0.1199


Epoch 87/100

6/6  0s 8ms/step - accuracy: 0.9888 - loss: 0.1112


Epoch 88/100

6/6  0s 8ms/step - accuracy: 0.9948 - loss: 0.1286

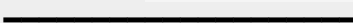
Epoch 89/100

6/6  0s 8ms/step - accuracy: 0.9948 - loss: 0.1073

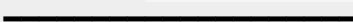
Epoch 90/100

6/6  0s 9ms/step - accuracy: 0.9881 - loss: 0.1084


Epoch 91/100

6/6  0s 8ms/step - accuracy: 0.9807 - loss: 0.1015

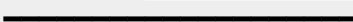
Epoch 92/100

6/6  0s 8ms/step - accuracy: 0.9829 - loss: 0.1066

Epoch 93/100

6/6  0s 8ms/step - accuracy: 0.9881 - loss: 0.1056

Epoch 94/100

6/6  0s 8ms/step - accuracy: 0.9881 - loss: 0.0964

Epoch 95/100

6/6 ————— 0s 8ms/step - accuracy: 0.9974 - loss: 0.0829

Epoch 96/100

6/6 ————— 0s 8ms/step - accuracy: 0.9926 - loss: 0.0937

Epoch 97/100

6/6 ————— 0s 8ms/step - accuracy: 0.9963 - loss: 0.0844

Epoch 98/100

6/6 ————— 0s 8ms/step - accuracy: 0.9855 - loss: 0.0884

Epoch 99/100

6/6 ————— 0s 8ms/step - accuracy: 0.9983 - loss: 0.0789

Epoch 100/100

6/6 ————— 0s 8ms/step - accuracy: 0.9926 - loss: 0.0775

```
def generate_rnn_text(seed_text, next_words=20):
    for _ in range(next_words):

        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_len-1,
padding='pre')

        predicted = model.predict(token_list, verbose=0)
        predicted_word = tokenizer.index_word[np.argmax(predicted)]

        seed_text += " " + predicted_word

    return seed_text
```

```
print(generate_rnn_text("artificial intelligence", 25))
```

artificial intelligence is transforming modern society and of ai
systems should be designed responsibly coherent an output term loss
problems human help experience context computers understand human

```
print("\n=====")
print(" N-GRAM vs RNN COMPARISON")
print("=====\\n")
```

```
seed = "artificial intelligence"
```

```
print("Seed Text:", seed)
print("\\n--- NGRAM OUTPUT ---")
print(generate_ngram_text("artificial", 25))
```

```
print("\\n--- RNN OUTPUT ---")
print(generate_rnn_text(seed, 25))
```

```
print("\\n--- OBSERVATION ---")
```

```

print("NGRAM:")
print("- Local word prediction only")
print("- May break sentence flow")

print("\nRNN:")
print("- Understands sequence patterns")
print("- More meaningful and coherent text")

```

```

=====
N-GRAM vs RNN COMPARISON
=====

```

Seed Text: artificial intelligence

```

--- NGRAM OUTPUT ---
artificial intelligence. ai engineers.

```

```

--- RNN OUTPUT ---
artificial intelligence is transforming modern society and of ai
systems should be designed responsibly coherent an output term loss
problems human help experience context computers understand human

```

```

--- OBSERVATION ---
NGRAM:
- Local word prediction only
- May break sentence flow

```

```

RNN:
- Understands sequence patterns
- More meaningful and coherent text

```

```

import gradio as gr

def rnn_ui(seed_text, length):
    return generate_rnn_text(seed_text, int(length))

demo = gr.Interface(
    fn=rnn_ui,
    inputs=[
        gr.Textbox(label="Enter Seed Text", value="artificial
intelligence"),
        gr.Slider(5,50,value=20,step=1,label="Generate Words")
    ],
    outputs=gr.Textbox(label="Generated Text"),
    title="RNN Text Generation UI",
    description="GenAI Lab-4 – Text Generation using RNN"
)

demo.launch(debug=True)

```

* Running on local URL: <http://127.0.0.1:7860>

It looks like you are running Gradio on a hosted Jupyter notebook, which requires `share=True`. Automatically setting `share=True` (you can turn this off by setting `share=False` in `launch()` explicitly).

* Running on public URL: <https://1f6e629edaf5aca289.gradio.live>

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Hugging Face Spaces (<https://huggingface.co/spaces>)

<IPython.core.display.HTML object>

Created dataset file at: `.gradio/flagged/dataset1.csv`