Tarea 3 Ejercicio 3 - Introducción a las Redes Neuronales y Deep Learning

Nombre: Bruno Morici

ROL USM: 202373555-8

Curso: INF395, Introducción a las Redes Neuronales y Deep Learning

Profesor: Alejandro Veloz

Fecha: 9/11/2025

Entrenar red Transformer con texto en español

Este notebook extrae y documenta las funciones y clases del proyecto `gpt-trained/` (carpeta `src`) y añade celdas para preparar datos en español, entrenar el modelo y generar texto de ejemplo. Sigue las celdas en orden y ejecuta cada una.

Flujo general del ejercicio

1. **Configurar el entorno**: instalar dependencias y fijar rutas a `gpt-trained/`.
2. **Preparar datos en español**: limpiar un chat, tokenizarlo y guardar `train.pt`, `valid.pt`, `vocab.txt` y `contacts.txt`.
3. **Definir el modelo Transformer**: copiar las clases `Head`, `MultiHeadAttention`, `Block` y `GPTLanguageModel` del proyecto base y explicarlas.
4. **Entrenar el modelo**: reutilizar `get_batch`, `estimate_loss` y el bucle `model_training` adaptados a celdas.
5. **Generar texto**: cargar el modelo y muestrear respuestas de ejemplo sin depender de `prompt_toolkit`.

Cada sección replica el comportamiento de los scripts originales (`config.py`, `preprocess.py`, `model.py`, `train.py` y `chat.py`) para tener un flujo reproducible desde el notebook.

0. Instalación de dependencias

Instalar las dependencias declaradas en `gpt-trained/requirements.txt`.

In [1]:
```
%pip install --quiet -r "gpt-trained/requirements.txt"
```

```
Note: you may need to restart the kernel to use updated packages.
[notice] A new release of pip is available: 23.1.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
```

1. Importaciones, rutas y configuración de hiperparámetros

Copiamos los hiperparámetros de `config.py` y definimos rutas base para reutilizarlas en todas las celdas posteriores.

In [9]:
```
from pathlib import Path
import json
import math
import random
import re
import time
from collections import Counter
from datetime import datetime
from typing import List, Set, Tuple, Union

import torch
import torch.nn as nn
from torch.nn import functional as F
from nltk.tokenize import RegexpTokenizer

# Barra de progreso para el entrenamiento
from tqdm import tqdm
```

In [10]:
```
# Rutas principales
BASE_DIR = Path("gpt-trained").resolve()
ASSETS_DIR = BASE_DIR / "assets"
INPUT_CHAT_PATH = ASSETS_DIR / "input" / "chat.txt"
OUTPUT_DIR = ASSETS_DIR / "output"
MODEL_DIR = ASSETS_DIR / "models"

OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
MODEL_DIR.mkdir(parents=True, exist_ok=True)
```

In [11]:
```
# Hiperparametros del modelo
block_size = 32
embed_size = 256
dropout = 0.2
n_heads = 6
n_layer = 6
eval_iters = 50
batch_size = 32

# Hiperparametros de entrenamiento y preprocesamiento
learn_rate = 3e-4
max_iters = 2000
```

```
eval_interval = 200
min_count_chars = 1
min_count_tokens = 1
end_token = "<END>"
unknown_token = "<UNK>"
n_chats = 5
```

2. Preprocesamiento de chats en español

Replicamos `preprocess.py` y `utils.py` para limpiar conversaciones de WhatsApp en español. El flujo es:

1. Eliminar caracteres muy raros.
2. Detectar remitentes y tratarlos como tokens especiales.
3. Tokenizar el texto respetando nombres de contacto y el token `<END>`.
4. Reemplazar vocabulario infrecuente por `<UNK>`.
5. Crear tensores `train.pt` y `valid.pt`, además de `vocab.txt` y `contacts.txt`.

Declaramos las funciones necesarias extraídas del ejemplo entregado:

In [12]:
```
def get_infrequent_tokens(tokens: Union[List[str], str], min_count: int) -> List[st
    counts = Counter(tokens)
    return [k for k, v in counts.items() if v <= min_count]


def mask_tokens(tokens: List[str], mask: Set[str]) -> List[str]:
    return [unknown_token if t in mask else t for t in tokens]


def drop_chars(txt: str, drop: Set[str]) -> str:
    return txt.translate(str.maketrans("", "", "".join(drop)))


def flatten_tuple(txt: List[Tuple[str, str]]) -> str:
    return "".join([contact + ":" + msg + end_token for contact, msg in txt])


def custom_tokenizer(txt: str, spec_tokens: List[str], pattern: str = "|\\d|\\w+|[^
    pattern = "|".join(spec_tokens) + pattern
    tokenizer = RegexpTokenizer(pattern)
    return tokenizer.tokenize(txt)


def get_vocab(text: Union[List[str], str]) -> List[str]:
    return sorted(list(set(text)))


def encode(tokens: List[str], vocab: List[str]) -> torch.Tensor:
    rand_token = random.randint(0, len(vocab) - 1)
    token_to_idx = {token: idx for idx, token in enumerate(vocab)}
    enc = [token_to_idx.get(token, token_to_idx.get(unknown_token, rand_token)) for
    return torch.tensor(enc, dtype=torch.long)
```

```
def decode(tensor: torch.Tensor, vocab: List[str]) -> str:
    token_to_idx = {token: idx for idx, token in enumerate(vocab)}
    idx_to_token = {idx: token for token, idx in token_to_idx.items()}
    return " ".join(idx_to_token[i.item()] for i in tensor)


def current_time() -> str:
    return datetime.now().strftime("%H:%M:%S")


def print_delayed(s: str, delay: float = 0.05) -> None:
    for char in s:
        print(char, end="", flush=True)
        time.sleep(delay)
    print()
```

2.1 Función `make_train_test`

Leemos `chat.txt`, tokeniza y guarda los tensores necesarios para el entrenamiento.

In [13]:
```
def make_train_test(input_path: Path = INPUT_CHAT_PATH) -> None:
    text = input_path.read_text(encoding="utf-8")

    infreq_chars = get_infrequent_tokens(text, min_count=min_count_chars)
    text = drop_chars(text, set(infreq_chars))

    pattern = r"\[(.*?)\] (.*?): (.*)"
    matches = re.findall(pattern, text)
    text = [(contact, msg.lower()) for _, contact, msg in matches if not msg.starts

    contacts = list({contact + ":" for contact, _ in text})
    spec_tokens = contacts + [end_token]

    text_flat = flatten_tuple(text)
    tokens = custom_tokenizer(txt=text_flat, spec_tokens=spec_tokens)

    infreq_tokens = set(get_infrequent_tokens(tokens, min_count=min_count_tokens))
    tokens = mask_tokens(tokens, infreq_tokens)

    vocab = get_vocab(tokens)
    print(f"El corpus tiene {len(vocab)} tokens únicos.")

    data = encode(tokens, vocab)
    n = int(0.9 * len(data))
    train_data = data[:n]
    valid_data = data[n:]

    torch.save(train_data, OUTPUT_DIR / "train.pt")
    torch.save(valid_data, OUTPUT_DIR / "valid.pt")
    (OUTPUT_DIR / "vocab.txt").write_text(json.dumps(vocab), encoding="utf-8")
    (OUTPUT_DIR / "contacts.txt").write_text(json.dumps(contacts), encoding="utf-8"
```

```
    print("Datos guardados en:")
    print(f"- {OUTPUT_DIR / 'train.pt'}")
    print(f"- {OUTPUT_DIR / 'valid.pt'}")
    print(f"- {OUTPUT_DIR / 'vocab.txt'}")
    print(f"- {OUTPUT_DIR / 'contacts.txt'}")
    print("SUCCESS")
```

2.2 Ejecutar el preprocesamiento

In [14]:
```
make_train_test()

train_path = OUTPUT_DIR / "train.pt"
valid_path = OUTPUT_DIR / "valid.pt"
vocab_path = OUTPUT_DIR / "vocab.txt"
contacts_path = OUTPUT_DIR / "contacts.txt"

(train_path.exists(), valid_path.exists(), vocab_path.exists(), contacts_path.exist
```

```
El corpus tiene 142 tokens únicos.
Datos guardados en:
- C:\Users\Bruno\Desktop\INF395 - IRN\tareas\tarea_3\gpt-trained\assets\output\trai
n.pt
- C:\Users\Bruno\Desktop\INF395 - IRN\tareas\tarea_3\gpt-trained\assets\output\vali
d.pt
- C:\Users\Bruno\Desktop\INF395 - IRN\tareas\tarea_3\gpt-trained\assets\output\voca
b.txt
- C:\Users\Bruno\Desktop\INF395 - IRN\tareas\tarea_3\gpt-trained\assets\output\conta
cts.txt
SUCCESS
```

```
A module that was compiled using NumPy 1.x cannot be run in
NumPy 2.2.6 as it may crash. To support both 1.x and 2.x
versions of NumPy, modules must be compiled with NumPy 2.0.
Some module may need to rebuild instead e.g. with 'pybind11>=2.12'.

If you are a user of the module, the easiest solution will be to
downgrade to 'numpy<2' or try to upgrade the affected module.
We expect that some modules will need time to support NumPy 2.

Traceback (most recent call last):  File "<frozen runpy>", line 198, in _run_module_
as_main
  File "<frozen runpy>", line 88, in _run_code
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el_launcher.py", line 18, in <module>
    app.launch_new_instance()
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\traitle
ts\config\application.py", line 1075, in launch_instance
    app.start()
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\kernelapp.py", line 758, in start
    self.io_loop.start()
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\tornado
\platform\asyncio.py", line 211, in start
    self.asyncio_loop.run_forever()
  File "C:\Users\Bruno\AppData\Local\Programs\Python\Python311\Lib\asyncio\base_even
ts.py", line 607, in run_forever
    self._run_once()
  File "C:\Users\Bruno\AppData\Local\Programs\Python\Python311\Lib\asyncio\base_even
ts.py", line 1922, in _run_once
    handle._run()
  File "C:\Users\Bruno\AppData\Local\Programs\Python\Python311\Lib\asyncio\events.p
y", line 80, in _run
    self._context.run(self._callback, *self._args)
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\kernelbase.py", line 614, in shell_main
    await self.dispatch_shell(msg, subshell_id=subshell_id)
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\kernelbase.py", line 471, in dispatch_shell
    await result
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\ipkernel.py", line 366, in execute_request
    await super().execute_request(stream, ident, parent)
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\kernelbase.py", line 827, in execute_request
    reply_content = await reply_content
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\ipkernel.py", line 458, in do_execute
    res = shell.run_cell(
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\ipykern
el\zmqshell.py", line 663, in run_cell
    return super().run_cell(*args, **kwargs)
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\IPython
\core\interactiveshell.py", line 3116, in run_cell
    result = self._run_cell(
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\IPython
\core\interactiveshell.py", line 3171, in _run_cell
```

```
    result = runner(coro)
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\IPython
\core\async_helpers.py", line 128, in _pseudo_sync_runner
    coro.send(None)
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\IPython
\core\interactiveshell.py", line 3394, in run_cell_async
    has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\IPython
\core\interactiveshell.py", line 3639, in run_ast_nodes
    if await self.run_code(code, result, async_=asy):
  File "c:\Users\Bruno\Desktop\INF395 - IRN\tareas\tf_venv\Lib\site-packages\IPython
\core\interactiveshell.py", line 3699, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "C:\Users\Bruno\AppData\Local\Temp\ipykernel_34792\492867216.py", line 1, in
<module>
    make_train_test()
  File "C:\Users\Bruno\AppData\Local\Temp\ipykernel_34792\1821440291.py", line 23, i
n make_train_test
    data = encode(tokens, vocab)
  File "C:\Users\Bruno\AppData\Local\Temp\ipykernel_34792\714679774.py", line 32, in
encode
    return torch.tensor(enc, dtype=torch.long)
C:\Users\Bruno\AppData\Local\Temp\ipykernel_34792\714679774.py:32: UserWarning: Fail
ed to initialize NumPy: _ARRAY_API not found (Triggered internally at ..\torch\csrc
\utils\tensor_numpy.cpp:84.)
  return torch.tensor(enc, dtype=torch.long)
```

Out[14]: (True, True, True, True)

3. Definición del modelo Transformer

Pasos de las siguientes celdas:

- 1. Copiar las clases `Head` y `MultiHeadAttention`, explicando su funcionamiento.
- 2. Copiar la clase `Block`, explicando su funcionamiento.
- 3. Copiar la clase `GPTLanguageModel`, explicando su funcionamiento.

In [15]:
```python
class Head(nn.Module):
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(embed_size, head_size, bias=False)
        self.query = nn.Linear(embed_size, head_size, bias=False)
        self.value = nn.Linear(embed_size, head_size, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x)
        q = self.query(x)

        wei = q @ k.transpose(-2, -1)
        wei /= math.sqrt(k.shape[-1])

        tril = torch.tril(torch.ones(T, T))
```

```
        wei = wei.masked_fill(tril == 0, float("-inf"))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)

        v = self.value(x)
        out = wei @ v
        return out


class MultiHeadAttention(nn.Module):
    def __init__(self):
        super().__init__()
        head_size = embed_size // n_heads
        self.heads = nn.ModuleList([Head(head_size) for _ in range(n_heads)])
        self.linear = nn.Linear(n_heads * head_size, embed_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        heads_list = [h(x) for h in self.heads]
        out = torch.cat(heads_list, dim=-1)
        out = self.linear(out)
        out = self.dropout(out)
        return out


class FeedFoward(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(embed_size, 4 * embed_size),
            nn.ReLU(),
            nn.Linear(4 * embed_size, embed_size),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)


class Block(nn.Module):
    def __init__(self):
        super().__init__()
        self.sa = MultiHeadAttention()
        self.ffwd = FeedFoward()
        self.ln1 = nn.LayerNorm(embed_size)
        self.ln2 = nn.LayerNorm(embed_size)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x


class GPTLanguageModel(nn.Module):
    def __init__(self, vocab_size: int):
        super().__init__()
```

```
        self.token_embedding = nn.Embedding(vocab_size, embed_size)
        self.pos_embedding = nn.Embedding(block_size, embed_size)
        block_list = [Block() for _ in range(n_layer)]
        self.blocks = nn.Sequential(*block_list)
        self.ln_output = nn.LayerNorm(embed_size)
        self.linear_output = nn.Linear(embed_size, vocab_size)
        self.apply(self.init_weights)

    def init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        tok_emb = self.token_embedding(idx)
        pos_emb = self.pos_embedding(torch.arange(T))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_output(x)
        logits = self.linear_output(x)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B * T, C)
            targets = targets.view(B * T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, vocab):
        idx_next = torch.zeros(1)
        idx_end = encode([end_token], vocab)
        idx_unk = encode([unknown_token], vocab)

        while idx_next[0] != idx_end:
            idx_cond = idx[:, -block_size:]
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
            while idx_next[0] == idx_unk:
                idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)

        return idx[0][:-1]
```

4. Entrenamiento del modelo

En esta sección:

- 1. Integramos las funciones de integramos las funciones de `utils.py` y `train.py`.
- 2. Se declaran los auxiliares `get_batch` y `estimate_loss`.
- 3. Luego el lazo `model_training` que admite un parámetro `update` para continuar entrenamientos previos.

In [16]:
```python
def get_batch(data: torch.Tensor):
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i : i + block_size] for i in ix])
    y = torch.stack([data[i + 1 : i + block_size + 1] for i in ix])
    return x, y


@torch.no_grad()
def estimate_loss(model: nn.Module, data: torch.Tensor) -> float:
    model.eval()
    loss_list = torch.zeros(eval_iters)
    for i in range(eval_iters):
        X, Y = get_batch(data)
        logits, loss = model(X, Y)
        loss_list[i] = loss.item()
    model.train()
    return loss_list.mean().item()
```

In [17]:
```python
def model_training(update: bool = False) -> GPTLanguageModel:
    train_data = torch.load(OUTPUT_DIR / "train.pt")
    valid_data = torch.load(OUTPUT_DIR / "valid.pt")
    vocab = json.loads((OUTPUT_DIR / "vocab.txt").read_text(encoding="utf-8"))

    if update:
        try:
            model = torch.load(MODEL_DIR / "model.pt")
            print("Modelo existente cargado: continúa el entrenamiento.")
        except FileNotFoundError:
            print("No se encontró un modelo previo, se inicializa uno nuevo.")
            model = GPTLanguageModel(vocab_size=len(vocab))
    else:
        print("Entrenamiento desde cero.")
        model = GPTLanguageModel(vocab_size=len(vocab))

    optimizer = torch.optim.AdamW(model.parameters(), lr=learn_rate)
    n_params = sum(p.numel() for p in model.parameters())
    print(f"Parámetros a optimizar: {n_params}")

    for i in tqdm(range(max_iters)):
        if i % eval_interval == 0 or i == max_iters - 1:
            train_loss = estimate_loss(model, train_data)
            valid_loss = estimate_loss(model, valid_data)
            print(f"{current_time()} | paso {i}: train {train_loss:.4f}, valid {val

        x_batch, y_batch = get_batch(train_data)
        logits, loss = model(x_batch, y_batch)
        optimizer.zero_grad(set_to_none=True)
```

```
        loss.backward()
        optimizer.step()

    torch.save(model, MODEL_DIR / "model.pt")
    print("Modelo guardado en", MODEL_DIR / "model.pt")
    return model, vocab
```

4.1 Ejecutar entrenamiento

In [11]:
```
model, vocab = model_training(update=False)
```

```
Entrenamiento desde cero.
Parámetros a optimizar: 4790926
  0%|          | 0/2000 [00:00<?, ?it/s]
10:15:48 | paso 0: train 5.0486, valid 5.0877
 10%|▉         | 200/2000 [02:59<15:28,  1.94it/s]
10:18:31 | paso 200: train 0.3215, valid 4.3804
 20%|██        | 400/2000 [05:09<11:36,  2.30it/s]
10:20:38 | paso 400: train 0.1653, valid 4.9521
 30%|███       | 600/2000 [06:53<10:19,  2.26it/s]
10:22:22 | paso 600: train 0.1380, valid 5.2489
 40%|████      | 800/2000 [08:30<08:20,  2.40it/s]
10:23:58 | paso 800: train 0.1319, valid 5.6288
 50%|█████     | 1000/2000 [10:06<06:50,  2.43it/s]
10:25:35 | paso 1000: train 0.1269, valid 5.7234
 60%|██████    | 1200/2000 [11:45<05:19,  2.50it/s]
10:27:13 | paso 1200: train 0.1251, valid 5.7921
 70%|███████   | 1400/2000 [13:20<04:11,  2.38it/s]
10:28:48 | paso 1400: train 0.1220, valid 5.9352
 80%|████████  | 1600/2000 [14:55<02:48,  2.38it/s]
10:30:23 | paso 1600: train 0.1157, valid 6.0960
 90%|█████████ | 1800/2000 [16:30<01:23,  2.38it/s]
10:32:00 | paso 1800: train 0.1170, valid 6.1779
100%|██████████| 1999/2000 [18:07<00:00,  2.41it/s]
10:33:35 | paso 1999: train 0.1137, valid 6.2429
100%|██████████| 2000/2000 [18:19<00:00,  1.82it/s]
Modelo guardado en C:\Users\Bruno\Desktop\INF395 - IRN\tareas\tarea_3\gpt-trained\assets\models\model.pt
```

5. Generación de texto en español

Para simplificar la interacción en notebook, cargamos el modelo guardado y generamos respuestas dadas unas pocas frases de contexto. La lógica de muestreo replica la función `conversation()` de `chat.py` sin depender de la terminal.

In [18]:
```
def load_model_and_assets():
    vocab = json.loads((OUTPUT_DIR / "vocab.txt").read_text(encoding="utf-8"))
    contacts = json.loads((OUTPUT_DIR / "contacts.txt").read_text(encoding="utf-8")
    model = torch.load(MODEL_DIR / "model.pt")
```

```
    model.eval()
    return model, vocab, contacts


def generate_response(seed_contact: str, mensaje: str, n_samples: int = 1):
    model, vocab, contacts = load_model_and_assets()
    spec_tokens = contacts + [end_token]

    prompt = f"{seed_contact}:{mensaje}{end_token}"
    tokens = custom_tokenizer(prompt, spec_tokens)
    context = encode(tokens, vocab).unsqueeze(0)
    prompt_len = context.shape[1]

    respuestas = []
    for _ in range(n_samples):
        output = model.generate(context.clone(), vocab)
        continuation = output[prompt_len:]
        texto = decode(continuation, vocab)
        texto = texto.replace(end_token, "").strip()
        respuestas.append(texto)

    return respuestas
```

5.1 Ejemplo rápido

Creamos un mensaje inicial. El modelo generará `n_samples` respuestas en español basadas en el contexto entrenado.

In [19]:
```
seed_contact = "contacto:"
mensaje = "¿Cómo va todo hoy?"
respuestas = generate_response(seed_contact, mensaje, n_samples=10)
for idx, texto in enumerate(respuestas, start=1):
    print(f"Respuesta {idx}: {texto}\n")
```

```
Respuesta 1: Brokovski: esperar para mí , pero va bien .

Respuesta 2: Brokovski: yo también estoy pensando en explorar la palomitas , ¡ no pu
edo unas vacaciones en la playa .

Respuesta 3: Brokovski: estoy planeando un viaje a europa el semana una gran una lis
ta de las películas de superhéroes .

Respuesta 4: Sandra: estoy pensando en explorar la europa , el caribe , bali y un vi
aje por carretera ¡ todos un viaje por carretera ¡ todos carretera ¡ todos todos , s
andra !

Respuesta 5: Brokovski: estoy planeando un viaje a europa el día .

Respuesta 6: Brokovski: .

Respuesta 7: Brokovski: estoy planeando un viaje a europa el es un viaje a europa el
vino !

Respuesta 8: Brokovski: estoy planeando un viaje a europa el una , ¡ europa suena in
creíble ! ¿ a

Respuesta 9: Sandra: estoy pensando en explorar la , tom . ¿ de qué a canciones ?

Respuesta 10: Brokovski: europa , el caribe , bali y un viaje por carretera ¡ todos
.
```

6. Resumen

- Encapsulamos los scripts del ejemplo entregado en clases `gpt-trained/` sin modificar la arquitectura del Transformer.
- Seccionamos las tareas en celdas para facilitar la ejecución paso a paso.
- Documentamos en proceso completo; Tokenización, preprocesamiento, definición del modelo, entrenamiento y generación de texto..