1. ¿Qué es un Transformer y qué hace este mini-GPT?

Conceptos básicos

Un **Transformer** es una arquitectura de red neuronal diseñada para procesar secuencias de texto. A diferencia de las redes recurrentes (RNN), los Transformers procesan todo el texto simultáneamente usando un mecanismo llamado **self-attention**.

¿Cómo funciona?

1. **Tokens**: El texto se divide en unidades pequeñas llamadas "tokens" (pueden ser palabras o caracteres).
2. **Embeddings** (¡Importante!): Cada token se convierte en un **vector numérico** de dimensión fija. Este proceso se llama "embedding".
    - Los embeddings son representaciones aprendidas que capturan el significado de cada token.
    - Por ejemplo, palabras similares tendrán embeddings similares.
    - El modelo aprende estos embeddings durante el entrenamiento.
3. **Self-Attention**: Este mecanismo permite que cada posición del texto "mire" a todas las otras posiciones para entender el contexto.
    - Cuando el modelo procesa la palabra "banco", puede mirar las palabras cercanas ("río" vs "dinero") para entender el contexto.
    - Cada palabra decide cuánta atención prestarle a las demás palabras.
4. **Predicción**: El modelo predice cuál es el siguiente token más probable en la secuencia.

¿Qué hace este mini-GPT específicamente?

- Es una **versión pequeña** de modelos como ChatGPT
- Aprende a predecir el **siguiente carácter** en una conversación
- Se entrena con conversaciones reales en español
- Después del entrenamiento, puede **generar texto nuevo** imitando el estilo de las conversaciones

Limitaciones

- No es ChatGPT: es mucho más simple
- Solo aprende el estilo del archivo de entrenamiento
- No tiene conocimiento general del mundo
- Genera texto basándose en patrones estadísticos, no en "comprensión" real

In [10]:
```
# Agregar el directorio gpt-trained al path de Python
import sys
sys.path.append("/kaggle/input/gpt-trained/gpt-trained")


# Importar las funciones y clases necesarias
from src.preprocess import get_infrequent_tokens, mask_tokens, drop_chars, custom_t
from src.model import GPTLanguageModel
from src.train import model_training
from src.utils import encode, decode, get_vocab

import torch
import json

print("Módulos importados exitosamente")
```

```
Módulos importados exitosamente
```

2. Preprocesamiento del texto (vocabulario y tokens)

Antes de entrenar el modelo, necesitamos convertir el texto en números que la red neuronal pueda procesar.

Conceptos clave

Token

Un **token** es la unidad básica de texto. En este modelo, cada carácter es un token (tokenización a nivel de caracteres).

Ejemplo:

- Texto: `"Hola"`
- Tokens: `['H', 'o', 'l', 'a']`

Vocabulario

El **vocabulario** es el conjunto de todos los tokens únicos que aparecen en el texto de entrenamiento.

Ejemplo:

- Si el texto es `"hola mundo"`
- Vocabulario: `['h', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd']`

Embedding

El **embedding** es la conversión de cada token en un vector numérico de tamaño fijo.

Por ejemplo:

- Token 'a' → índice 5 → embedding vector de 256 dimensiones: `[0.23, -0.45, 0.12, ..., 0.67]`
- Token 'e' → índice 10 → embedding vector de 256 dimensiones: `[0.18, -0.52, 0.08, ..., 0.71]`

¿Por qué son importantes los embeddings?

- Transforman símbolos discretos (letras) en representaciones continuas que la red puede procesar
- Los embeddings se **aprenden durante el entrenamiento**
- Capturan similitudes semánticas: letras que aparecen en contextos similares tendrán embeddings parecidos

Codificación (encoding)

La función `encode_text()` convierte cada carácter en su índice correspondiente en el vocabulario.

Ejemplo:

- Vocabulario: `['a', 'b', 'c', 'h', 'l', 'o']`
- Texto: `"hola"`
- Encoding: `[3, 5, 4, 0]` (índices en el vocabulario)

Dataset autoregresivo

El dataset se construye de manera que el modelo aprenda a predecir el siguiente carácter:

- Entrada (X): `"hol"`
- Salida esperada (Y): `"ola"`

El modelo ve `"hol"` y debe aprender a predecir `"a"`.

In [11]:
```python
# Funciones de preprocesamiento del archivo preprocess.py

# Esta función identifica tokens que aparecen menos de min_count veces
# Sirve para eliminar caracteres raros o emojis poco frecuentes
def get_infrequent_tokens_example(tokens, min_count):
    """
    Identifica tokens que aparecen menos de un mínimo de veces.

    Ejemplo:
    tokens = "hola mundo hola"
    min_count = 2
    Resultado: {'m', 'u', 'n', 'd'} (aparecen solo 1 vez)
    """
    from collections import Counter
    counts = Counter(tokens)
```

```
    infreq_tokens = set([k for k,v in counts.items() if v < min_count])
    return infreq_tokens

# Esta función reemplaza tokens raros por un token especial <UNK> (unknown)
def mask_tokens_example(tokens, mask):
    """
    Reemplaza tokens raros por <UNK> (unknown token).

    Esto reduce el tamaño del vocabulario y ayuda a generalizar.
    """
    unknown_token = "<UNK>"
    return [t.replace(t, unknown_token) if t in mask else t for t in tokens]

# Esta función construye el vocabulario (lista de tokens únicos)
def build_vocab_example(tokens):
    """
    Construye el vocabulario: lista ordenada de tokens únicos.

    Ejemplo:
    tokens = ['h', 'o', 'l', 'a', 'h', 'o', 'l', 'a']
    Resultado: ['a', 'h', 'l', 'o']
    """
    return sorted(list(set(tokens)))

# Esta función codifica tokens a números usando el vocabulario
def encode_text_example(tokens, vocab):
    """
    Convierte cada token a su índice en el vocabulario.

    Ejemplo:
    tokens = ['h', 'o', 'l', 'a']
    vocab = ['a', 'h', 'l', 'o']
    Resultado: [1, 3, 2, 0]
    """
    token_to_idx = {token: idx for idx, token in enumerate(vocab)}
    return [token_to_idx.get(token, 0) for token in tokens]

# Ejemplo práctico
texto_ejemplo = "hola mundo"
tokens_ejemplo = list(texto_ejemplo)
vocab_ejemplo = build_vocab_example(tokens_ejemplo)
encoded_ejemplo = encode_text_example(tokens_ejemplo, vocab_ejemplo)

print("Texto original:", texto_ejemplo)
print("Tokens:", tokens_ejemplo)
print("Vocabulario:", vocab_ejemplo)
print("Codificación:", encoded_ejemplo)
```

```
Texto original: hola mundo
Tokens: ['h', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
Vocabulario: [' ', 'a', 'd', 'h', 'l', 'm', 'n', 'o', 'u']
Codificación: [3, 7, 4, 1, 0, 5, 8, 6, 2, 7]
```

3. Arquitectura del Transformer (Mini-GPT)

El modelo GPT está compuesto por varias capas que trabajan en secuencia. Aquí explicamos cada componente.

Embedding Layer (Capa de Embeddings)

¿Qué hace? Convierte cada token (número entero) en un vector denso de números reales.

¿Por qué es importante?

- Los números enteros (0, 1, 2...) no capturan relaciones semánticas
- Los embeddings permiten que tokens similares tengan representaciones similares
- Son **aprendidos** durante el entrenamiento, no predefinidos

Ejemplo:

```
# Si tenemos un vocabulario de 50 tokens y queremos embeddings de
dimensión 256:
embedding_layer = nn.Embedding(50, 256)

# Token 5 se convierte en un vector de 256 números:
token_5 = torch.tensor([5])
embedding_5 = embedding_layer(token_5)  # shape: (1, 256)
```

Positional Encoding (Codificación Posicional)

¿Qué hace? Añade información sobre la **posición** de cada token en la secuencia.

¿Por qué es necesario?

- El self-attention no tiene noción del orden de las palabras por sí solo
- "El perro muerde al hombre" ≠ "El hombre muerde al perro"
- Los positional embeddings permiten al modelo saber que el token está en la posición 0, 1, 2, etc.

Cómo funciona:

```
# Cada posición (0, 1, 2, ..., 31) tiene su propio embedding aprendido
pos_embedding = nn.Embedding(32, 256)  # 32 posiciones, 256 dimensiones

# Se suma al token embedding:
x = token_embedding + pos_embedding
```

Self-Attention (Auto-Atención)

¿Qué hace? Permite que cada token "mire" a todos los otros tokens del contexto para entender mejor su significado.

Explicación simple: Imagina que estás leyendo la frase: "*El gato subió al árbol porque tenía miedo del perro*"

Cuando el modelo procesa la palabra "tenía", el self-attention le permite:

- Mirar hacia atrás y ver "gato" → entiende quién tenía miedo
- Mirar "perro" → entiende la causa del miedo

Cómo funciona técnicamente:

1. Cada token genera 3 vectores: **Query (Q)**, **Key (K)**, **Value (V)**
2. Se calcula una puntuación de atención: qué tan relevante es cada token para el token actual
3. Se usa esa puntuación para crear un promedio ponderado de los valores

Importante: En GPT usamos **causal attention** (atención causal):

- Un token solo puede mirar a tokens anteriores, no futuros
- Esto se llama "masked attention" y se logra con una máscara triangular inferior

Multi-Head Attention (Atención Multi-Cabeza)

En lugar de una sola capa de self-attention, usamos **múltiples en paralelo** (6 cabezas en este modelo).

Ventaja:

- Cada cabeza puede aprender a prestar atención a diferentes aspectos
- Una cabeza podría enfocarse en sintaxis, otra en semántica, etc.

Feed-Forward Network (Red Neuronal Feed-Forward)

Después del self-attention, cada token pasa por una pequeña red neuronal:

```
# Expande la dimensión 4x, aplica ReLU, y luego vuelve al tamaño original
Linear(256 → 1024) → ReLU → Linear(1024 → 256)
```

Esto permite que el modelo aprenda transformaciones no lineales complejas.

Bloque Transformer

Un bloque Transformer combina:

1. Layer Normalization
2. Multi-Head Self-Attention
3. Conexión residual (skip connection)
4. Layer Normalization
5. Feed-Forward Network
6. Otra conexión residual

Este modelo usa 6 bloques apilados secuencialmente.

Proyección Final (Output Layer)

La última capa convierte el embedding de 256 dimensiones en **probabilidades** sobre todo el vocabulario:

```
# Si el vocabulario tiene 80 tokens:
Linear(256 → 80) → Softmax
```

El modelo produce una distribución de probabilidad:

- Token 'a': 0.25 (25% de probabilidad)
- Token 'e': 0.15 (15% de probabilidad)
- ...

Y se elige el token con mayor probabilidad (o se muestrea según la distribución).

In [12]:
```python
# Código del modelo GPTLanguageModel (desde model.py)

import torch
import torch.nn as nn
from torch.nn import functional as F
import math

# CONFIGURACIÓN (estos valores vienen de config.py)
block_size = 32      # Tamaño máximo de contexto (32 caracteres)
embed_size = 256     # Dimensión de los embeddings
n_heads = 6          # Número de cabezas de atención
n_layer = 6          # Número de bloques Transformer
dropout = 0.2        # Tasa de dropout para regularización

# 1. HEAD: Una sola cabeza de self-attention
class Head(nn.Module):
    """Una cabeza de self-attention"""
    def __init__(self, head_size):
        super().__init__()
        # Proyecciones lineales para Query, Key, Value
        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  # Batch, Time-steps, Channels

        # Calcular Q, K, V
        k = self.key(x)    # (B, T, head_size)
        q = self.query(x)  # (B, T, head_size)

        # Calcular puntuaciones de atención: Q @ K^T / sqrt(d_k)
        wei = q @ k.transpose(-2, -1)  # (B, T, T)
        wei /= math.sqrt(k.shape[-1])  # Escalado

        # Aplicar máscara causal (no mirar al futuro)
        tril = torch.tril(torch.ones(T, T))
        wei = wei.masked_fill(tril == 0, float("-inf"))
```

```
        wei = F.softmax(wei, dim=-1)  # Normalizar a probabilidades
        wei = self.dropout(wei)

        # Aplicar atención a los valores
        v = self.value(x)  # (B, T, head_size)
        out = wei @ v      # (B, T, head_size)
        return out


# 2. MULTI-HEAD ATTENTION: Múltiples cabezas en paralelo
class MultiHeadAttention(nn.Module):
    """Múltiples cabezas de self-attention en paralelo"""
    def __init__(self):
        super().__init__()
        head_size = embed_size // n_heads
        # Crear n_heads cabezas en paralelo
        self.heads = nn.ModuleList([Head(head_size) for _ in range(n_heads)])
        # Proyección final
        self.linear = nn.Linear(n_heads * head_size, embed_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Ejecutar todas las cabezas en paralelo y concatenar
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.linear(out)
        out = self.dropout(out)
        return out


# 3. FEED-FORWARD: Red neuronal simple
class FeedFoward(nn.Module):
    """Red feed-forward con expansión 4x"""
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(embed_size, 4 * embed_size),  # Expandir
            nn.ReLU(),
            nn.Linear(4 * embed_size, embed_size),  # Contraer
            nn.Dropout(dropout),
        )

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


# 4. BLOCK: Un bloque Transformer completo
class Block(nn.Module):
    """Un bloque Transformer: self-attention + feed-forward"""
    def __init__(self):
        super().__init__()
        self.sa = MultiHeadAttention()      # Self-attention
        self.ffwd = FeedFoward()            # Feed-forward
        self.ln1 = nn.LayerNorm(embed_size) # Layer norm 1
        self.ln2 = nn.LayerNorm(embed_size) # Layer norm 2

    def forward(self, x):
```

```
        # Conexiones residuales + layer norm
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x


# 5. MODELO COMPLETO: GPTLanguageModel
class GPTLanguageModel(nn.Module):
    """Modelo GPT completo"""
    def __init__(self, vocab_size):
        super().__init__()

        # EMBEDDINGS
        self.token_embedding = nn.Embedding(vocab_size, embed_size)
        self.pos_embedding = nn.Embedding(block_size, embed_size)

        # BLOQUES TRANSFORMER (apilados secuencialmente)
        self.blocks = nn.Sequential(*[Block() for _ in range(n_layer)])

        # SALIDA
        self.ln_output = nn.LayerNorm(embed_size)
        self.linear_output = nn.Linear(embed_size, vocab_size)

        # Inicializar pesos
        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

        # EMBEDDINGS: token + posición
        tok_emb = self.token_embedding(idx)           # (B, T, embed_size)
        pos_emb = self.pos_embedding(torch.arange(T)) # (T, embed_size)
        x = tok_emb + pos_emb                         # (B, T, embed_size)

        # BLOQUES TRANSFORMER
        x = self.blocks(x)                            # (B, T, embed_size)

        # SALIDA
        x = self.ln_output(x)                         # (B, T, embed_size)
        logits = self.linear_output(x)                # (B, T, vocab_size)

        # CALCULAR LOSS (si tenemos targets)
        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits_flat = logits.view(B*T, C)
            targets_flat = targets.view(B*T)
```

```
            loss = F.cross_entropy(logits_flat, targets_flat)

        return logits, loss

print("Modelo GPT definido exitosamente")
print(f"Configuración:")
print(f"  - Tamaño de contexto: {block_size} caracteres")
print(f"  - Dimensión de embeddings: {embed_size}")
print(f"  - Número de cabezas: {n_heads}")
print(f"  - Número de bloques: {n_layer}")
```

```
Modelo GPT definido exitosamente
Configuración:
  - Tamaño de contexto: 32 caracteres
  - Dimensión de embeddings: 256
  - Número de cabezas: 6
  - Número de bloques: 6
```

4. Entrenamiento del modelo con texto en español

¿Cómo se entrena un modelo de lenguaje?

El entrenamiento de un modelo GPT sigue un esquema **autoregresivo**:

Paso 1: Cargar el texto en español

Se carga un archivo `.txt` con conversaciones reales en español (por ejemplo, `chat.txt`).

Paso 2: Crear pares de entrada-salida

Para cada secuencia de texto, creamos:

- **X (entrada)**: Una secuencia de N caracteres
- **Y (salida esperada)**: Los mismos caracteres desplazados 1 posición

Ejemplo:

```
Texto: "Hola mundo"

X: "Hola mund"  →  Y: "ola mundo"
```

El modelo ve `"Hola mund"` y debe aprender a predecir `"o"` como siguiente carácter.

Paso 3: Forward pass (predicción)

1. Se codifica X en índices del vocabulario
2. Se pasa por los embeddings
3. Se procesa con los bloques Transformer
4. Se obtienen las probabilidades de cada token

Paso 4: Calcular el error (loss)

Se compara la predicción del modelo con Y usando **Cross-Entropy Loss**:

- Si el modelo predice correctamente el siguiente carácter → loss bajo
- Si predice mal → loss alto

Paso 5: Backpropagation y actualización

1. Se calculan los gradientes (derivadas del loss respecto a los parámetros)
2. Se actualizan los pesos usando el optimizador AdamW
3. El modelo aprende a predecir mejor el siguiente carácter

Paso 6: Repetir

Se repite este proceso miles de veces (iteraciones) con diferentes fragmentos del texto.

División train/validation

- **90% del texto** → entrenamiento
- **10% del texto** → validación (para monitorear el progreso)

Hiperparámetros importantes

- **Batch size**: 32 (número de secuencias procesadas simultáneamente)
- **Learning rate**: 3e-4 (qué tan grandes son los pasos de actualización)
- **Max iterations**: 5000 (número total de iteraciones de entrenamiento)
- **Eval interval**: Cada 500 iteraciones se evalúa el modelo en validación

In [13]:
```
# Código de entrenamiento (simplificado desde train.py)

import torch
from torch.utils.data import Dataset

# Configuración de entrenamiento
learn_rate = 3e-4
max_iters = 3000
eval_interval = 500
batch_size = 32
block_size = 32

def get_batch(data, batch_size, block_size):
    """Genera un batch aleatorio de datos"""
    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, data, eval_iters=200):
    """Estima el loss promedio en un conjunto de datos"""
    model.eval()
```

```
    losses = torch.zeros(eval_iters)
    for k in range(eval_iters):
        X, Y = get_batch(data, batch_size, block_size)
        logits, loss = model(X, Y)
        losses[k] = loss.item()
    model.train()
    return losses.mean()

def train_model(text_path, vocab_size):
    """Entrena el modelo GPT con texto en español"""

    # 1. Cargar y preprocesar texto
    with open(text_path, 'r', encoding='utf-8') as f:
        text = f.read()

    print(f"Longitud del texto: {len(text):,} caracteres")

    # 2. Construir vocabulario
    chars = sorted(list(set(text)))
    vocab_size = len(chars)
    print(f"Vocabulario: {vocab_size} caracteres únicos")
    print(f"Primeros 50 caracteres del vocabulario: {''.join(chars[:50])}")

    # 3. Crear mapeos char <-> int
    stoi = {ch: i for i, ch in enumerate(chars)}
    itos = {i: ch for i, ch in enumerate(chars)}
    encode = lambda s: [stoi[c] for c in s]
    decode = lambda l: ''.join([itos[i] for i in l])

    # 4. Codificar todo el texto
    data = torch.tensor(encode(text), dtype=torch.long)
    print(f"Datos codificados: {len(data):,} tokens")

    # 5. Split train/val
    n = int(0.9 * len(data))
    train_data = data[:n]
    val_data = data[n:]
    print(f"Train: {len(train_data):,} tokens")
    print(f"Validation: {len(val_data):,} tokens")

    # 6. Inicializar modelo
    model = GPTLanguageModel(vocab_size)
    n_params = sum(p.numel() for p in model.parameters())
    print(f"\nModelo inicializado con {n_params:,} parámetros")

    # 7. Configurar optimizador
    optimizer = torch.optim.AdamW(model.parameters(), lr=learn_rate)

    # 8. Loop de entrenamiento
    print(f"\nIniciando entrenamiento por {max_iters} iteraciones...")
    print("="*60)

    for iter in range(max_iters):

        # Evaluar cada eval_interval iteraciones
        if iter % eval_interval == 0 or iter == max_iters - 1:
```

```
            train_loss = estimate_loss(model, train_data)
            val_loss = estimate_loss(model, val_data)
            print(f"Iter {iter:4d} | train loss: {train_loss:.4f} | val loss: {val_

        # Obtener batch
        xb, yb = get_batch(train_data, batch_size, block_size)

        # Forward pass
        logits, loss = model(xb, yb)

        # Backward pass
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()

    print("="*60)
    print("Entrenamiento completado!")

    # 9. Guardar modelo
    torch.save({
        'model_state_dict': model.state_dict(),
        'vocab': chars,
        'stoi': stoi,
        'itos': itos
    }, 'gpt_model_espanol.pt')
    print("Modelo guardado en: gpt_model_espanol.pt")

    return model, stoi, itos

print("Función de entrenamiento lista")
```

```
Función de entrenamiento lista
```

In [14]:
```
# ENTRENAR EL MODELO

# Ruta al archivo de texto en español
text_path = "/kaggle/input/gpt-trained/gpt-trained/assets/input/chat.txt"

# Entrenar
model, stoi, itos = train_model(text_path, vocab_size=100)  # vocab_size se calcula
```

```
Longitud del texto: 8,512 caracteres
Vocabulario: 77 caracteres únicos
Primeros 50 caracteres del vocabulario:
 !",-.0123456789:?ABCDEFGHIJLMNOPQRSTVWY[]abcdefg
Datos codificados: 8,512 tokens
Train: 7,660 tokens
Validation: 852 tokens

Modelo inicializado con 4,757,581 parámetros

Iniciando entrenamiento por 3000 iteraciones...
============================================================
Iter    0 | train loss: 4.4337 | val loss: 4.4220
Iter  500 | train loss: 0.6749 | val loss: 1.9581
Iter 1000 | train loss: 0.4055 | val loss: 2.4193
Iter 1500 | train loss: 0.3528 | val loss: 2.6068
Iter 2000 | train loss: 0.3083 | val loss: 2.8044
Iter 2500 | train loss: 0.2645 | val loss: 2.9338
Iter 2999 | train loss: 0.2209 | val loss: 3.1124
============================================================
Entrenamiento completado!
Modelo guardado en: gpt_model_espanol.pt
```

5. Generar texto en español

Una vez entrenado el modelo, podemos usarlo para **generar texto nuevo**.

¿Cómo funciona la generación?

1. **Inicio**: Damos al modelo un texto semilla (seed), por ejemplo: `"Hola"`
2. **Predicción**: El modelo predice el siguiente carácter más probable
   - `"Hola"` → modelo predice `" "` (espacio)
3. **Actualización**: Añadimos el carácter predicho al contexto
   - Nuevo contexto: `"Hola "`
4. **Repetición**: Repetimos el proceso
   - `"Hola "` → predice `"c"`
   - `"Hola c"` → predice `"ó"`
   - `"Hola có"` → predice `"m"`
   - Y así sucesivamente...
5. **Parada**: Nos detenemos cuando:
   - Se genera un token especial de fin `<END>`
   - O llegamos al límite de caracteres generados

Muestreo (Sampling)

En lugar de siempre elegir el token más probable, **muestreamos** de la distribución de probabilidad:

- Permite más creatividad y variabilidad
- Evita que el modelo repita siempre lo mismo

Ejemplo:

```
Distribución del siguiente carácter:
  'a': 40%
  'e': 30%
  'o': 20%
  'i': 10%

En lugar de siempre elegir 'a', muestreamos según estas
probabilidades.
```

In [15]:
```python
# Función para generar texto

def generate_text(model, seed_text, max_new_tokens=200, stoi=None, itos=None):
    """
    Genera texto nuevo a partir de un texto semilla.

    Args:
        model: Modelo GPT entrenado
        seed_text: Texto inicial (semilla)
        max_new_tokens: Máximo de caracteres nuevos a generar
        stoi: Diccionario char -> int
        itos: Diccionario int -> char

    Returns:
        Texto generado completo (semilla + generación)
    """
    model.eval()

    # Codificar texto semilla
    encode = lambda s: [stoi[c] for c in s if c in stoi]
    decode = lambda l: ''.join([itos[i] for i in l])

    # Convertir a tensor
    context = torch.tensor(encode(seed_text), dtype=torch.long).unsqueeze(0)  # (1,

    # Generar tokens uno por uno
    with torch.no_grad():
        for _ in range(max_new_tokens):
            # Crop context si es muy largo (máximo block_size)
            context_crop = context[:, -block_size:]

            # Obtener predicciones
            logits, _ = model(context_crop)

            # Enfocarse solo en el último paso de tiempo
            logits = logits[:, -1, :]  # (1, vocab_size)
```

```
            # Aplicar softmax para obtener probabilidades
            probs = F.softmax(logits, dim=-1)  # (1, vocab_size)

            # Muestrear de la distribución
            idx_next = torch.multinomial(probs, num_samples=1)  # (1, 1)

            # Añadir al contexto
            context = torch.cat((context, idx_next), dim=1)  # (1, T+1)

    # Decodificar y retornar
    generated = decode(context[0].tolist())
    return generated

print("Función de generación lista")
```

```
Función de generación lista
```

In [16]:
```
# GENERAR TEXTO NUEVO

# Texto semilla en español
seed_text = "Hola, ¿cómo estás?"

print(f"Texto semilla: '{seed_text}'")
print("\nGenerando...\n")
print("="*60)

generated = generate_text(model, seed_text, max_new_tokens=300, stoi=stoi, itos=ito

print(generated)
print("="*60)
```

```
Texto semilla: 'Hola, ¿cómo estás?'

Generando...

============================================================
Hola, ¿cómo estás?
[12.03.18, 17:13:17] Tom: ¿Qué tal juegos de mesa después de la cena?
[12.03.18, 17:11:54] Brokovski: Estoy planeando un viaje a Europa el próximo verano.
[12.03.18, 18:19:34] Alice: ¡Europa suena increíble! ¿A dónde irás?
[12.03.18, 18:21:11] Tom: Me encantaría escuchar tus planes!
[12.03.18, 19:0
============================================================
```

In [17]:
```
# Probar con diferentes semillas

seeds = [
    "Buenos días",
    "¿Qué tal?",
    "Me gusta",
    "Hoy es"
]

for seed in seeds:
    print(f"\n{'='*60}")
    print(f"Semilla: '{seed}'")
    print(f"{'='*60}")
```

```
    generated = generate_text(model, seed, max_new_tokens=150, stoi=stoi, itos=itos
    print(generated)
```

```
============================================================
Semilla: 'Buenos días'
============================================================
Buenos días.
[12.03.18, 17:21:22] Alice: También deberíamos preparar una lista de reproducción.
[12.03.18, 17:22:48] Paul: He estado pensando en unas vacaciones

============================================================
Semilla: '¿Qué tal?'
============================================================
¿Qué tal?
[12.03.18, 16:16:22] Alice: ¡Suena como una noche divertida!
[12.03.18, 16:53:53] Tom: Tendremos que encontrar una fecha que funcione para todos.
[12

============================================================
Semilla: 'Me gusta'
============================================================
Me gustarémendación de canciones?
[12.03.18, 17:27:57] Brokovski: Estoy planeando un viaje a Europa el próximo verano.
[12.03.18, 18:19:34] Alice: ¡Entonces s

============================================================
Semilla: 'Hoy es'
============================================================
Hoy estado, ¡y fue épica!
[12.03.18, 17:48:00] Alice: Viajar y explorar nuevos lugares son las mejores experie
ncias.
[12.03.18, 18:50:17] Paul: ¡Absolutamen
```

6. Conclusiones y limitaciones

Lo que aprendimos

1. **Arquitectura Transformer**:
   - Cómo funcionan los **embeddings** para convertir texto en vectores numéricos
   - El mecanismo de **self-attention** que permite que cada token mire a otros
   - La combinación de múltiples bloques para crear modelos profundos
2. **Entrenamiento**:
   - Cómo se preprocesa el texto (tokenización, vocabulario, codificación)
   - El esquema autoregresivo: predecir el siguiente carácter
   - Optimización con backpropagation y AdamW
3. **Generación**:
   - Cómo generar texto nuevo carácter por carácter
   - Muestreo de la distribución de probabilidad

Limitaciones importantes

1. **Tamaño del modelo**:
   - Este mini-GPT tiene ~1-2 millones de parámetros
   - GPT-3 tiene 175 **billones** de parámetros
   - Por eso nuestro modelo es mucho más limitado
2. **Datos de entrenamiento**:
   - Solo aprende del archivo `chat.txt`
   - No tiene conocimiento general del mundo
   - Solo puede imitar el estilo de ese archivo específico
3. **Capacidad de contexto**:
   - Solo puede "recordar" los últimos 32 caracteres
   - GPT-4 puede procesar miles de tokens
4. **Calidad del texto generado**:
   - Puede ser inconsistente o sin sentido
   - No entiende semántica profunda
   - Solo sigue patrones estadísticos

Conceptos clave para recordar

- **Embeddings**: Representaciones vectoriales aprendidas de tokens
- **Self-Attention**: Mecanismo que permite al modelo entender relaciones entre palabras
- **Autoregresivo**: Predecir el siguiente token basándose en los anteriores
- **Causal Masking**: No permitir que el modelo vea el futuro durante el entrenamiento
- **Layer Normalization**: Estabiliza el entrenamiento
- **Residual Connections**: Ayudan al flujo de gradientes en redes profundas