Chatbot Tutor de Matemáticas y Física basado en Transformer

Informe Final — Curso de Profundización en Deep Learning

Melissa Cardona — Carrera de Física, 2026

Nota para el profesor

Este informe documenta el diseño, implementación y evaluación de un **Transformer Encoder–Decoder** entrenado desde cero para resolver problemas de matemáticas y física con soluciones paso a paso.

Para probar el chatbot interactivo, abra el notebook de demo:

```
notebooks/03_demo_profesor.ipynb
```

El modelo ya viene entrenado en `checkpoints/`. No es necesario reentrenar.

Instrucciones rápidas:

1. Clonar o descargar desde: `https://github.com/MelissaCardona2003/Chat-bot-de-matemacticas-y-f-sica-con-TensorFlow.git`
2. Crear entorno: `python -m venv .venv && source .venv/bin/activate && pip install -r requirements.txt`
3. Abrir: `jupyter notebook notebooks/03_demo_profesor.ipynb`
4. Ejecutar todas las celdas — la interfaz Gradio se abrirá automáticamente.

Funciona en **CPU**. No se requiere GPU.

Resumen

Se presenta el diseño, implementación y evaluación de un modelo **Transformer Encoder–Decoder** construido completamente desde cero en TensorFlow, sin utilizar modelos pre-entrenados ni APIs externas. El sistema recibe problemas de matemáticas (aritmética, álgebra) y física (cinemática, dinámica, termodinámica, circuitos eléctricos) y genera soluciones paso a paso.

El modelo cuenta con **7,476,615 parámetros**, utiliza tokenización a nivel de carácter (135 tokens) y fue entrenado sobre un dataset combinado de **12,568 problemas** provenientes de GSM8K, MATH (con soluciones generadas por LLM) y problemas de física generados paramétricamente.

Los resultados muestran una **token accuracy del 82.1%** en validación, lo cual indica que el modelo aprende correctamente la distribución de caracteres en soluciones matemáticas/físicas.

Sin embargo, el **exact match es del 0%**: el modelo genera texto con formato correcto (*"Step 1:... Answer: X"*) pero las respuestas numéricas son incorrectas. Se analiza en profundidad por qué ocurre esto y qué se necesitaría para superarlo.

El proyecto incluye un pipeline completo: recolección y preparación de datos, entrenamiento con técnicas modernas (label smoothing, learning rate scheduling, early stopping, decoder token masking), evaluación rigurosa, y despliegue con una interfaz interactiva Gradio.

Palabras clave: Transformer, Encoder-Decoder, Attention, Deep Learning, matemáticas, física, resolución de problemas, TensorFlow.

---

1. Introducción

1.1 Motivación

La resolución automática de problemas de matemáticas y física mediante modelos de lenguaje es una de las fronteras más activas de la inteligencia artificial. Sistemas como GPT-4, Gemini y Claude han demostrado capacidades notables en razonamiento matemático, pero operan con cientos de miles de millones de parámetros, infraestructura masiva y datasets propietarios.

Este proyecto nace de una pregunta pedagógica fundamental: **¿qué se puede lograr construyendo un Transformer desde cero, con recursos limitados, para resolver problemas de matemáticas y física?**

La motivación es doble:

1. **Pedagógica**: Implementar un Transformer completo (no solo usar `transformers.AutoModel`) fuerza a comprender cada componente: atención escalada, positional encoding, máscaras causales, residual connections, layer normalization, y el flujo de información entre encoder y decoder.

2. **Científica**: Explorar las limitaciones de un modelo compacto (7.4M parámetros, tokenización carácter a carácter) en tareas que requieren razonamiento numérico, y entender *por qué* los LLMs modernos necesitan escalas mucho mayores.

1.2 Formulación del problema

El problema se formula como una transformación secuencia-a-secuencia. Dado un problema $x = (x_1, x_2, \ldots, x_T)$ en lenguaje natural, el objetivo es generar una solución $y = (y_1, y_2, \ldots, y_{T'})$ que contenga los pasos de resolución y la respuesta final.

Formalmente, se busca maximizar la probabilidad condicional:

$$\hat{y} = \arg\max_{y} P(y \mid x) = \arg\max_{y} \prod_{t=1}^{T'} P(y_t \mid y_{<t}, x)$$

donde cada y_t es un carácter generado condicionado en el problema completo x (a través del encoder) y en los caracteres previamente generados $y_{<t}$ (a través del decoder autoregresivo).

1.3 Alcance

El sistema abarca dos dominios:

- **Matemáticas**: Problemas aritméticos y algebraicos de nivel escolar (sumas, restas, multiplicaciones, proporciones, porcentajes) con razonamiento verbal (*word problems*).
- **Física**: Problemas de cinemática, dinámica newtoniana, termodinámica y circuitos eléctricos con aplicación directa de fórmulas.

El formato de salida esperado es:

```
Step 1: Identify the given values...
Step 2: Apply the relevant formula...
Step 3: Calculate...
Answer: [valor numérico]
```

---

2. Entorno y Herramientas

2.1 Framework de Deep Learning

- **TensorFlow 2.x / Keras**: Framework principal para la implementación del modelo, incluyendo todas las capas (`tf.keras.layers`), el loop de entrenamiento con `GradientTape`, y la gestión de checkpoints.

2.2 Cómputo numérico y evaluación

- **NumPy**: Operaciones vectoriales, muestreo probabilístico (top-k sampling), cálculo de métricas.
- **SymPy**: Validación simbólica de respuestas matemáticas (equivalencia algebraica).
- **Matplotlib**: Visualización de curvas de entrenamiento y distribuciones de atención.

2.3 Datos

- **HuggingFace Datasets**: Descarga de GSM8K y MATH directamente desde el hub.
- **JSON**: Formato de almacenamiento para todos los datasets procesados.
- **Funciones propias**: Generación paramétrica de problemas de física, conversión de formatos, filtrado de calidad.

2.4 Interfaz

- **Gradio 4.x**: Interfaz web interactiva para demostración del chatbot.

2.5 Hardware

- **GPU**: NVIDIA RTX 5060 (arquitectura Blackwell, 16 GB VRAM)
- **Nota**: La GPU Blackwell requirió workarounds específicos para XLA y operaciones de cast (ver Sección 5.6).
- **Inferencia**: Funcional en CPU sin modificaciones.

---

3. Marco Teórico

3.1 La arquitectura Transformer

El Transformer, introducido por Vaswani et al. (2017), reemplaza las redes recurrentes por un mecanismo de **atención puro** que permite procesar secuencias completas en paralelo. La arquitectura consta de un encoder y un decoder, ambos compuestos por capas apiladas de atención y redes feed-forward.

3.1.1 Scaled Dot-Product Attention

El mecanismo fundamental es la atención escalada por producto punto. Dadas las matrices de queries Q, keys K y values V:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

donde d_k es la dimensión de las keys. El factor $\frac{1}{\sqrt{d_k}}$ previene que los productos punto crezcan demasiado para valores grandes de d_k, lo cual empujaría al softmax hacia regiones de gradiente pequeño.

3.1.2 Multi-Head Attention

En lugar de aplicar una sola función de atención, se proyectan Q, K y V en h subespacios diferentes:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

donde cada cabeza se calcula como:

$$\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

con $W_i^Q, W_i^K \in R^{d_{\text{model}} \times d_k}$, $W_i^V \in R^{d_{\text{model}} \times d_v}$ y $W^O \in R^{hd_v \times d_{\text{model}}}$.

En nuestra implementación: $d_{\text{model}} = 256$, $h = 8$, por lo tanto $d_k = d_v = 256/8 = 32$.

3.1.3 Positional Encoding

Dado que el Transformer no posee recurrencia ni convoluciones, se inyecta información posicional mediante funciones sinusoidales:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

Estas funciones permiten al modelo inferir posiciones relativas, ya que PE_{pos+k} puede expresarse como transformación lineal de PE_{pos}.

3.1.4 Estructura del Encoder

Cada capa del encoder consta de:

1. **Multi-Head Self-Attention**: Cada posición atiende a todas las demás posiciones de la entrada.
2. **Add & Layer Normalization**: Conexión residual seguida de normalización.
3. **Feed-Forward Network (FFN)**: Dos capas densas con activación ReLU:

$$\text{FFN}(x) = max(0, xW_1 + b_1)W_2 + b_2$$

con $W_1 \in R^{d_{\text{model}} \times d_{ff}}$ y $W_2 \in R^{d_{ff} \times d_{\text{model}}}$. En nuestro caso, $d_{ff} = 1024$.

1. **Add & Layer Normalization**.

Se aplica dropout ($p = 0.2$) después de cada sub-capa.

3.1.5 Estructura del Decoder

Cada capa del decoder añade una sub-capa adicional:

1. **Masked Multi-Head Self-Attention**: Atención causal que impide que la posición t atienda a posiciones futuras $t' > t$. Se implementa con una máscara *look-ahead*:

$$M_{ij} = \begin{pmatrix} 0 & \text{si } i \geq j \\ -\infty & \text{si } i < j \end{pmatrix}$$

1. **Cross-Attention** (Encoder–Decoder): Las queries provienen del decoder, mientras que las keys y values provienen del encoder. Esto permite al decoder "leer" el problema de entrada.

2. **FFN** + Add & Norm (igual que en el encoder).

3.1.6 Capa de salida

La salida del decoder pasa por una capa lineal que produce logits sobre el vocabulario:

$$\text{logits}_t = W_{\text{out}} \cdot h_t^{(L)} + b_{\text{out}} \in R^{|V|}$$

donde $|V| = 135$ es el tamaño del vocabulario de caracteres.

3.2 Función de pérdida

Se utiliza entropía cruzada categórica dispersa con **label smoothing** ($\epsilon = 0.1$):

$$L = -\sum_{t=1}^{T'} \left[(1-\epsilon) \log P\left(y_t^{\text{true}}\right) + \frac{\epsilon}{|V|} \sum_k \log P(k) \right) \cdot 1\left[y_t \neq \text{PAD}\right]$$

El label smoothing previene que el modelo se vuelva excesivamente confiado en sus predicciones, actuando como regularizador.

Las posiciones correspondientes al token `<PAD>` se enmascaran para no contribuir a la pérdida.

3.3 Learning Rate Schedule

Se utiliza el esquema propuesto en el paper original (Vaswani et al., 2017):

$$lr = d_{\text{model}}^{-0.5} \cdot min\left(\text{step}^{-0.5}, \text{step} \cdot \text{warmup}^{-1.5}\right)$$

con $\text{warmup\_steps} = 2000$. El learning rate crece linealmente durante los primeros 2000 pasos y luego decae proporcionalmente a $\text{step}^{-0.5}$.

---

4. Datasets y Preparación de Datos

4.1 Fuentes de datos

La construcción del dataset fue uno de los retos principales del proyecto. Se requieren pares **(problema, solución paso a paso)** donde la solución contenga razonamiento explícito y una respuesta final.

4.1.1 GSM8K (Grade School Math 8K)

Dataset de OpenAI (Cobbe et al., 2021) con 8,792 problemas aritméticos verbales de nivel escolar. Cada problema incluye una solución con razonamiento paso a paso y una respuesta final numérica.

- **Conversión**: Se descarga vía HuggingFace y se convierte al esquema unificado del proyecto con `data/convert_gsm8k.py`.
- **Limpieza**: Eliminación de anotaciones tipo `#### ANSWER`, normalización de formato, filtrado por longitud.
- **Resultado**: 8,638 problemas limpios (7,319 train + 1,319 test).

4.1.2 MATH Dataset con soluciones LLM

El dataset MATH (Hendrycks et al., 2021) contiene problemas de competencia matemática con soluciones en LaTeX. Dado que las soluciones originales usan notación LaTeX compleja incompatible con tokenización carácter a carácter, se implementó un pipeline alternativo:

1. **Descarga y filtrado** (`data/download_dataset.py`, `data/filter_dataset.py`): Se seleccionaron niveles 1-3 en álgebra, combinatoria y prealgebra.
2. **Generación de soluciones** (`data/generate_math_solutions_llm.py`): Se utilizó un LLM para generar soluciones en texto plano paso a paso, reemplazando el LaTeX por lenguaje natural.
3. **Resultado**: 1,895 problemas con soluciones legibles.

4.1.3 Problemas de Física (Generación Paramétrica)

No se encontró un dataset público de física análogo a GSM8K. Se optó por generar problemas de forma paramétrica (`data/generate_physics_templates.py`):

Se definieron templates en 5 áreas:

Área	Fórmulas	Ejemplo
Cinemática	$v = v_0 + a\,t$, $x = v_0\,t + \frac{1}{2}a\,t^2$	"A car accelerates at 3 m/s² for 5s..."
Dinámica	$F = m\,a$, $W = F\,d$	"A 10 kg box is pushed with 50 N..."
Termodinámica	$Q = m\,c\,\Delta T$	"Heat needed for 2 kg water, ΔT=30°C..."
Circuitos	$V = I\,R$, $P = I\,V$	"Circuit with 12V and 4Ω..."
Caída libre	$h = \frac{1}{2}g\,t^2$	"Object dropped from 80 m..."

Cada template genera problemas con parámetros aleatorios dentro de rangos físicamente razonables, junto con la solución correcta calculada analíticamente.

- **Resultado**: 2,035 problemas de física (2,019 paramétricos + 16 manuales).

4.2 Esquema Unificado

Todos los datos se normalizan a un esquema JSON común (`data/schema.py`):

```
{
  "problem": "A car accelerates from rest at 3 m/s² for 5 seconds...",
  "solution": "Step 1: Identify... Step 2: Apply... Answer: 15 m/s",
  "answer": "15",
  "domain": "physics",
  "source": "physics_templates",
  "topic": "kinematics",
```

```
  "split": "train"
}
```

La función de validación verifica que cada entrada tenga los campos requeridos, que `problem` y `solution` no estén vacíos, que la longitud sea razonable, y que el `domain` sea `math` o `physics`.

4.3 Dataset Final Combinado

El script `data/build_combined_dataset.py` fusiona las tres fuentes y asigna splits:

	Math	Physics	Total
Train	8,616	1,621	**10,237**
Val	598	341	**939**
Test	1,319	73	**1,392**
Total	**10,533**	**2,035**	**12,568**

4.4 Tokenización

Se implementó un tokenizador a nivel de carácter (`data/tokenizer.py`) con vocabulario fijo:

$$V = \{\text{<PAD>}, \text{<START>}, \text{<END>}, \text{<UNK>}\} \cup \{c_1, \ldots, c_{131}\}$$

Los 131 caracteres incluyen letras (mayúsculas y minúsculas), dígitos, puntuación, operadores matemáticos y algunos caracteres unicode presentes en los datasets.

Justificación de tokenización carácter a carácter: Se eligió este enfoque por simplicidad de implementación y para evitar dependencias externas (SentencePiece, BPE). Como se discute en la Sección 9, esta decisión tiene implicaciones significativas en el rendimiento.

4.5 Construcción de `tf.data.Dataset`

El `DatasetBuilder` (`data/dataset_builder.py`) construye pipelines eficientes:

1. Carga el JSON del dataset combinado.
2. Tokeniza problemas y soluciones con padding a longitudes fijas (`max_encoder_len=200`, `max_decoder_len=300`).
3. Construye pares `(encoder_input, decoder_input) → decoder_target`.
4. Aplica shuffle, batching y prefetch para entrenamiento eficiente.

---

5. Implementación

Todo el código fuente se organizó como un paquete Python (`transformer_math_physics_tutor/`) con módulos separados para cada componente.

5.1 Multi-Head Attention (`models/multihead_attention.py`)

La implementación sigue fielmente el paper original. La clase `MultiHeadAttention` recibe `d_model` y `num_heads`, crea las proyecciones W^Q, W^K, W^V, W^O como capas `Dense`, y ejecuta:

1. Proyección: $Q' = QW^Q$, $K' = KW^K$, $V' = VW^V$
2. Reshape a `(batch, heads, seq_len, depth)` donde `depth = d_model / num_heads = 32`
3. Scaled dot-product attention con máscara opcional
4. Concatenación de cabezas y proyección de salida

5.2 Encoder y Decoder (`models/encoder_layer.py`, `models/decoder_layer.py`)

- **EncoderLayer**: Self-attention → Add & Norm → FFN → Add & Norm
- **DecoderLayer**: Masked self-attention → Add & Norm → Cross-attention → Add & Norm → FFN → Add & Norm

El decoder devuelve además un diccionario de pesos de atención, lo cual permite análisis post-hoc de qué tokens del encoder son más atendidos.

5.3 Transformer Completo (`models/transformer.py`)

La clase `Transformer` integra:

- Embeddings de encoder y decoder (compartiendo la misma dimensión `d_model=256`)
- Positional encoding sinusoidal
- Stack de 4 capas encoder y 4 capas decoder
- Capa de salida lineal (`Dense(vocab_size)`)
- Funciones para crear máscaras de padding y look-ahead

El forward pass: `model((encoder_input, decoder_input))` → logits `(batch, seq_len, vocab_size)`.

5.4 Loop de Entrenamiento (`training/train.py`)

Se implementó un `TransformerTrainer` con `tf.GradientTape` en lugar de `model.fit()`, lo que permite:

- **Decoder token masking**: Durante entrenamiento, se reemplazan aleatoriamente el 20% de los tokens del decoder input por un token `<UNK>`. Esto obliga al modelo a usar cross-attention para obtener información del encoder, en lugar de depender exclusivamente del contexto autoregresivo.
- **Gradient clipping** para estabilidad.
- **Checkpointing** cada 5 épocas con rotación.
- **Early stopping** con patience=10 sobre `val_loss`.

5.5 Scheduler de Learning Rate (`training/scheduler.py`)

Implementa el esquema del paper original:

$$lr(\text{step}) = d_{\text{model}}^{-0.5} \cdot min(\text{step}^{-0.5}, \text{step} \cdot \text{warmup}^{-1.5})$$

Con `warmup_steps=2000`, el LR máximo alcanza aproximadamente 3.7×10^{-4}.

5.6 Workarounds para GPU Blackwell

La GPU NVIDIA RTX 5060 (basada en la arquitectura Blackwell) presentó problemas con operaciones XLA y `tf.cast`, que fallaban con errores de segmentación. Se implementaron dos workarounds:

1. **XLA flags**: Configuración de `XLA_FLAGS` y `TF_XLA_FLAGS` para apuntar al directorio correcto de CUDA 12.8.
2. **Cast patch**: Wrapper de `tf.cast` que ejecuta conversiones de tipo en CPU para evitar el bug de Blackwell en modo eager.
3. **XLA Dropout** (`models/xla_dropout.py`): Implementación de dropout compatible con la compilación XLA, usando `tf.random.stateless_uniform` en lugar de `tf.nn.dropout`.

```
# === Configuración del entorno ===
import os, sys, json
import numpy as np

# Ruta del proyecto
project_root = os.path.abspath(os.path.join(os.getcwd(), '..'))
if project_root not in sys.path:
    sys.path.insert(0, project_root)

# Cargar configuración y training history
checkpoint_dir = os.path.join(project_root, 'checkpoints')

with open(os.path.join(checkpoint_dir, 'config.json'), 'r') as f:
    config_dict = json.load(f)

with open(os.path.join(checkpoint_dir, 'training_history.json'), 'r')
as f:
    history = json.load(f)

with open(os.path.join(checkpoint_dir, 'evaluation_report.json'), 'r')
as f:
    eval_report = json.load(f)

print("=" * 60)
print("CONFIGURACION DEL MODELO")
print("=" * 60)
for k, v in config_dict.items():
    print(f"  {k:20s}: {v}")
```

```
print(f"\n  Total parametros: {eval_report['model_params']:,}")
print(f"  Epocas entrenadas: {len(history['train_loss'])}")
```

---

6. Entrenamiento

6.1 Configuración de hiperparámetros

Hiperparámetro	Valor	Justificación
`d_model`	256	Balance entre capacidad y eficiencia
`num_heads`	8	Estándar para $d_k = 32$
`num_layers`	4	Suficiente para capturar patrones sin overfitting
`dff`	1024	Ratio 4:1 con d_{model} (como en el paper original)
`dropout_rate`	0.2	Mayor que el default (0.1) por dataset pequeño
`batch_size`	32	Limitado por GPU memory
`warmup_steps`	2000	~6 épocas de warmup
`label_smoothing`	0.1	Regularización estándar
`decoder_mask_rate`	0.20	Forzar uso de cross-attention
`max_encoder_len`	200	Cubre el 99% de los problemas
`max_decoder_len`	300	Cubre el 99% de las soluciones

6.2 Proceso de entrenamiento

El entrenamiento se ejecutó en una NVIDIA RTX 5060 durante 89 épocas (de 100 máximo), detenido por early stopping (patience=10).

Evolución del entrenamiento:

Etapa	Época	Train Loss	Train Acc	Val Loss	Val Acc
Inicio	1	3.543	24.0%	2.778	36.5%
Warmup completo	10	2.158	50.2%	1.834	62.7%
Mitad	45	1.693	69.5%	1.419	79.7%
Mejor val_loss	79	1.613	73.2%	**1.387**	81.7%
Final (early stop)	89	1.609	73.4%	1.387	81.7%

Observaciones clave:

- La val_accuracy es *mayor* que la train_accuracy. Esto es típico con dropout alto (0.2) y decoder masking (20%), que penalizan el entrenamiento pero no la validación.
- La convergencia es estable: no hay oscilaciones ni divergencia.
- El LR alcanza su máximo ($\approx 3.7 \times 10^{-4}$) alrededor de la época 6 y luego decae suavemente.

```
# === Curvas de entrenamiento ===
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

epochs = range(1, len(history['train_loss']) + 1)

# Loss
axes[0].plot(epochs, history['train_loss'], label='Train',
linewidth=2, color='#4f46e5')
axes[0].plot(epochs, history['val_loss'], label='Val', linewidth=2,
color='#f97316')
axes[0].set_title('Loss', fontsize=14, fontweight='bold')
axes[0].set_xlabel('Epoca')
axes[0].set_ylabel('Loss')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Accuracy
axes[1].plot(epochs, history['train_accuracy'], label='Train',
linewidth=2, color='#4f46e5')
axes[1].plot(epochs, history['val_accuracy'], label='Val',
linewidth=2, color='#f97316')
axes[1].set_title('Token Accuracy', fontsize=14, fontweight='bold')
axes[1].set_xlabel('Epoca')
axes[1].set_ylabel('Accuracy')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

# Learning Rate
axes[2].plot(epochs, history['learning_rates'], linewidth=2,
color='#10b981')
axes[2].set_title('Learning Rate', fontsize=14, fontweight='bold')
axes[2].set_xlabel('Epoca')
axes[2].set_ylabel('LR')
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nMejor val_loss: {min(history['val_loss']):.4f} (epoca
{history['val_loss'].index(min(history['val_loss'])) + 1})")
```

```
print(f"Mejor val_acc:  {max(history['val_accuracy']):.4f} (epoca
{history['val_accuracy'].index(max(history['val_accuracy'])) + 1})")
```

---

7. Inferencia

7.1 Generación autoregresiva

Durante entrenamiento se usa **teacher forcing**: el decoder recibe la secuencia objetivo desplazada. Durante inferencia, el decoder opera de forma autoregresiva:

1. Se codifica el problema con el encoder: $H = \text{Encoder}(x)$
2. Se inicializa el decoder con el token `<START>`
3. En cada paso t:
   - Se computan los logits: $z_t = \text{Decoder}(y_{<t}, H)$
   - Se selecciona el siguiente token $\hat{y}_t$
   - Se añade al decoder input
4. Se repite hasta generar `<END>` o alcanzar `max_length=300`

7.2 Estrategias de decodificación

Se implementaron tres estrategias en `inference/generate.py`:

Greedy decoding

$$\hat{y}_t = arg\,max_k\, z_t[k]$$

Determinístico pero propenso a generar secuencias repetitivas.

Top-k sampling

Se restringen los logits a los k tokens más probables, se aplica temperatura τ y se muestrea:

$$P(y_t = k) = \frac{\exp(z_t[k]/\tau)}{\sum_{j \in \text{top-}k} \exp(z_t[j]/\tau)}$$

En la demo se usa $k = 10$, $\tau = 0.3$.

Repetition penalty

Para evitar bucles, se penalizan tokens ya generados:

$$z_t[k] \leftarrow \begin{cases} z_t[k]/\alpha & \text{si } z_t[k] > 0 \text{ y } k \in \text{generados} \\ z_t[k] \cdot \alpha & \text{si } z_t[k] \leq 0 \text{ y } k \in \text{generados} \end{cases}$$

con $\alpha = 1.3$. Adicionalmente, se implementa detección de n-gram repeat ($n = 10$) para terminar anticipadamente si el modelo entra en un bucle.

7.3 Métricas en tiempo real

La demo calcula métricas basadas en los logits reales del modelo:

- **Confianza**: $\text{conf} = \frac{1}{T} \sum_{t=1}^{T} max_v P(v \mid v_{<t})$ — promedio de la probabilidad máxima por paso.
- **Perplexity**: $\text{PPL} = \exp\left(-\frac{1}{T} \sum_{t=1}^{T} \log P(y_t \mid y_{<t}) \right)$ — sobre los tokens seleccionados.

7.4 Interfaz Gradio

Se desarrolló una interfaz Gradio Blocks (`notebooks/03_demo_profesor.ipynb`) con:

- Selector de dominio (math/physics)
- Ejemplos pre-cargados por dominio
- Métricas en tiempo real (confianza, perplexity, tiempo, tokens generados)
- Sección de arquitectura del modelo
- Curvas de entrenamiento
- Resultados de evaluación
- Análisis de limitaciones

---

8. Evaluación y Resultados

8.1 Métricas utilizadas

Se evaluó el modelo con dos familias de métricas:

Token-level Accuracy

Proporción de caracteres correctamente predichos (ignorando padding).

Exact Match (Answer:)

Se extrae la línea `Answer:` de la predicción y la referencia, y se comparan. También se implementó validación simbólica con SymPy para detectar equivalencias algebraicas (e.g., `3/2` = `1.5`).

8.2 Resultados cuantitativos

```
# === Resultados de evaluacion ===
ta = eval_report['token_accuracy']
em = eval_report['exact_match']

print("=" * 60)
print("RESULTADOS DE EVALUACION")
print("=" * 60)
print(f"\n  Token Accuracy (validacion): {ta['val_acc']:.1%}")
print(f"  Token Accuracy (test):       {ta['test_acc']:.1%}")
print(f"  Val Loss:                    {ta['val_loss']:.4f}")
print(f"  Test Loss:                   {ta['test_loss']:.4f}")
print(f"\n  Exact Match: {em['correct']}/{em['total']} =
{em['pct']:.1f}%")
print(f"\n  Por dominio:")
for domain, stats in sorted(em['by_domain'].items()):
    pct = stats['correct'] / max(stats['total'], 1) * 100
    print(f"    {domain}: {stats['correct']}/{stats['total']} =
{pct:.1f}%")

print(f"\n  Ejemplos de predicciones:")
for ex in eval_report.get('examples', [])[:6]:
    icon = 'OK' if ex['correct'] else 'FAIL'
    prob = ex['problem'][:70] + '...' if len(ex['problem']) > 70 else
ex['problem']
    print(f"    [{icon}] [{ex['domain']}] \"{prob}\"")
    print(f"       Esperado: {ex['ref_answer']} | Predicho:
{ex['pred_answer'] or '(vacio)'}")
```

8.3 Análisis de los resultados

Los resultados revelan un fenómeno interesante: **alta token accuracy pero cero exact match**.

¿Cómo es posible 82% accuracy con 0% exact match?

La token accuracy mide la predicción del **siguiente carácter** dado el contexto previo. Con tokenización carácter a carácter, muchas predicciones son triviales:

- Después de `"Ste"`, el modelo predice `"p"` → correcto
- Después de `"Step "`, predice `"1"` o `"2"` → a menudo correcto
- Después de `"Answer: "`, predice un dígito → casi siempre incorrecto

El 82% refleja principalmente la capacidad del modelo para predecir **texto en inglés y formato de solución**, no razonamiento matemático.

Cross-attention colapsada

El diagnóstico en el notebook de demo (`03_demo_profesor.ipynb`) revela que la **cross-attention está distribuida uniformemente** (entropía ≈ 100% del máximo). Esto significa que el

decoder no presta atención selectiva al problema de entrada, sino que genera texto "genérico" basándose solo en el contexto autoregresivo.

¿Por qué ocurre? Con tokenización carácter a carácter, el decoder puede predecir el siguiente carácter con alta accuracy usando solo el contexto previo (completar palabras, formatear pasos), sin necesidad de "leer" el problema. La cross-attention se vuelve redundante.

Decoder token masking

Se implementó **decoder token masking** (reemplazo aleatorio del 20% de tokens del decoder por `<UNK>`) para forzar al modelo a usar cross-attention. Aunque esta técnica ha demostrado efectividad en modelos más grandes, en nuestro caso la mejora fue marginal, posiblemente porque la secuencia es tan larga (~300 caracteres) que el modelo aún puede interpolar los tokens enmascarados desde el contexto local.

---

9. Limitaciones y Trabajo Futuro

9.1 Limitación principal: tokenización a nivel de carácter

La comparación con modelos que usan BPE (Byte Pair Encoding) ilustra el problema fundamental:

Aspecto	Nuestro modelo	GPT-2 (referencia)
Tokenización	Character-level (135 tokens)	BPE (~50,000 tokens)
Representación de "25"	2 tokens: `'2'`, `'5'`	1 token: `'25'`
Representación de "Step"	4 tokens: `'S'`, `'t'`, `'e'`, `'p'`	1 token: `'Step'`
Problema de 100 palabras	~500 tokens	~25 tokens
Parámetros	7.4M	117M (small)

Con tokenización BPE, el número "3200" es **un solo token**. El modelo puede aprender que cierta combinación de operaciones produce un token numérico específico. Con tokenización carácter a carácter, el modelo debe generar secuencialmente `'3'`, `'2'`, `'0'`, `'0'`, cada uno condicionado en los anteriores — una tarea combinatoriamente más difícil.

9.2 Escala del modelo

Los resultados en la literatura (Brown et al., 2020; Wei et al., 2022) muestran que el razonamiento matemático emerge como **capacidad emergente** en modelos a partir de ~10^{10} parámetros. Nuestro modelo, con 7.4×10^{6} parámetros, está tres órdenes de magnitud por debajo.

9.3 Mejoras propuestas

Para lograr respuestas numéricas correctas, se necesitaría:

1. **Tokenizer BPE/WordPiece**: Reducir las secuencias ~5x, permitiendo al modelo operar a nivel de concepto en lugar de carácter.
2. **Más parámetros**: ≥ 50M para capturar relaciones semánticas complejas.
3. **Pre-entrenamiento en corpus general**: Desarrollar comprensión lingüística previa (como BERT/GPT) antes de fine-tuning en matemáticas.
4. **Chain-of-thought fine-tuning**: Usar ejemplos de razonamiento paso a paso alineados con problemas específicos.
5. **Calculator augmentation**: Delegar cálculos aritméticos a módulos simbólicos externos.

9.4 Valor del resultado negativo

Aunque el resultado de 0% exact match pueda parecer un fracaso, es un **resultado científicamente significativo**: demuestra empíricamente que la tokenización carácter a carácter es insuficiente para razonamiento numérico en modelos de escala media, validando las observaciones teóricas de la literatura.

---

10. Aprendizajes y Contribuciones Personales

10.1 Proceso iterativo

El desarrollo del proyecto fue un proceso iterativo con múltiples versiones:

Versión 1: Solo matemáticas, dataset pequeño

- Dataset inicial: ~175 problemas generados manualmente.
- Modelo: 2 capas, d_model=128, vocab=97 (~1.5M params).
- Resultado: Overfitting extremo (96% train acc, 60% val acc). El modelo memorizó los pocos ejemplos.
- **Lección**: Se necesitan más datos.

Versión 2: GSM8K + MATH

- Se incorporó GSM8K (8,638 problemas) y MATH con soluciones LLM (1,895 problemas).
- Se escaló el modelo: 4 capas, d_model=256, 8 heads (~7.4M params).
- Resultado: Mejor generalización, pero solo en matemáticas.
- **Lección**: El modelo necesita datos de física.

Versión 3: Dataset combinado + física paramétrica

- Se generaron 2,035 problemas de física con templates paramétricos.
- Se implementó decoder token masking para mejorar cross-attention.

- Resultado: Versión final, 82% token accuracy, 0% exact match.
- **Lección**: La tokenización carácter a carácter es el cuello de botella fundamental.

10.2 Qué aprendí

Sobre Transformers

- La implementación from-scratch de Multi-Head Attention requiere manejo cuidadoso de shapes: el reshape a `(batch, heads, seq_len, depth)` y de vuelta es propenso a errores sutiles.
- Las máscaras (padding, look-ahead, combined) son el aspecto más delicado del Transformer. Un error en la máscara produce "data leakage" que no se detecta en training pero arruina la inferencia.
- El positional encoding sinusoidal funciona sin entrenamiento, pero su contribución al modelo depende de la longitud de las secuencias.

Sobre datos

- La preparación de datos consume más tiempo que la implementación del modelo (~60% del esfuerzo total).
- La calidad del dataset importa más que la cantidad: 12K problemas bien formateados superan a 100K problemas ruidosos.
- La generación paramétrica de problemas de física es una técnica efectiva cuando no hay datasets públicos disponibles.

Sobre entrenamiento

- El warmup del learning rate es esencial: sin warmup, el entrenamiento diverge en las primeras 10 épocas.
- Label smoothing (ε=0.1) y dropout (0.2) previenen overfitting efectivamente, incluso con 12K ejemplos.
- Early stopping con patience=10 es conservador pero robusto.

Sobre GPU y sistemas

- Las GPUs de nueva generación (Blackwell) pueden requerir workarounds inesperados que no están documentados.
- XLA compilation puede generar errores misteriosos que solo se manifiestan con ciertas operaciones.

---

11. Conclusiones

Se implementó exitosamente un **Transformer Encoder-Decoder desde cero** en TensorFlow para resolver problemas de matemáticas y física con soluciones paso a paso.

Logros principales:

1. **Arquitectura completa from-scratch**: Scaled dot-product attention, multi-head attention, positional encoding sinusoidal, encoder-decoder con máscaras, capa de salida — todo implementado sin usar capas Transformer pre-hechas.

2. **Pipeline de datos robusto**: Sistema modular para descargar, convertir, validar y combinar datasets de múltiples fuentes (GSM8K, MATH+LLM, física paramétrica) en un esquema unificado.

3. **Entrenamiento con técnicas modernas**: Learning rate scheduling (warmup + inverse sqrt decay), label smoothing, gradient clipping, early stopping, y decoder token masking — técnicas que demuestran comprensión de las mejores prácticas actuales.

4. **Evaluación rigurosa y honesta**: Se reportan tanto las métricas favorables (82% token accuracy) como las desfavorables (0% exact match), con un análisis detallado de por qué ocurre la discrepancia.

5. **Despliegue funcional**: Interfaz Gradio interactiva que permite probar el modelo en tiempo real con métricas de confianza.

Conclusión principal:

El proyecto demuestra que un Transformer de 7.4M parámetros con tokenización carácter a carácter puede aprender el **formato y estilo** de soluciones matemáticas y físicas (82% token accuracy), pero no el **razonamiento numérico** necesario para producir respuestas correctas (0% exact match). Esto confirma que el razonamiento matemático requiere: (a) representación a nivel de subpalabra (BPE), (b) escala significativamente mayor (>100M parámetros), y (c) idealmente pre-entrenamiento en corpus general.

El valor del proyecto reside en la **implementación completa y rigurosa de un pipeline de deep learning** desde la recolección de datos hasta el despliegue, pasando por la implementación from-scratch de una arquitectura que ha revolucionado la inteligencia artificial.

---

12. Referencias

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). **Attention Is All You Need.** *Advances in Neural Information Processing Systems, 30* (NeurIPS 2017). https://arxiv.org/abs/1706.03762

2. Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., & Schulman, J. (2021). **Training Verifiers to Solve Math Word Problems.** https://arxiv.org/abs/2110.14168

3. Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., & Steinhardt, J. (2021). **Measuring Mathematical Problem Solving With the MATH Dataset.** *Advances in Neural Information Processing Systems, 34* (NeurIPS 2021). https://arxiv.org/abs/2103.03874

4. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). **Language Models are Unsupervised Multitask Learners.** OpenAI.

5. Brown, T. B., et al. (2020). **Language Models are Few-Shot Learners.** *Advances in Neural Information Processing Systems, 33* (NeurIPS 2020). https://arxiv.org/abs/2005.14165

6. Wei, J., et al. (2022). **Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.** *Advances in Neural Information Processing Systems, 35* (NeurIPS 2022). https://arxiv.org/abs/2201.11903

7. Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). **Layer Normalization.** https://arxiv.org/abs/1607.06450

8. He, K., Zhang, X., Ren, S., & Sun, J. (2016). **Deep Residual Learning for Image Recognition.** *CVPR 2016*. https://arxiv.org/abs/1512.03385

9. Sennrich, R., Haddow, B., & Birch, A. (2016). **Neural Machine Translation of Rare Words with Subword Units.** *ACL 2016*. https://arxiv.org/abs/1508.07909

10. Kingma, D. P., & Ba, J. (2015). **Adam: A Method for Stochastic Optimization.** *ICLR 2015*. https://arxiv.org/abs/1412.6980