Universidad Nacional de San Agustin de Arequipa

Año de la recuperación y consolidación de la economía peruana

Escuela Profesional de Ciencia de la Computación

Topicos en Inteligencia Artificial

PROYECTO FINAL TOPICOS EN INTELIGENCIA ARTIFICIAL

Transformer VIT

A-2025

Profesor: Dr. JUAN CARLOS GUTIERREZ CACERES

Huamana Coaquira Luciana Julisa
Lopez Condori Andrea del Rosario
Mayorga Villena Jhrarold Alonso
Quispe Rojas Javier Wilber

Índice

1. Arquitectura

La arquitectura desarrollada corresponde a un **Transformer Visionario (ViT)** personalizado, diseñado específicamente para tareas de clasificación de imágenes como Fashion-MNIST. A continuación se describe de manera detallada cada uno de sus componentes y etapas de procesamiento:

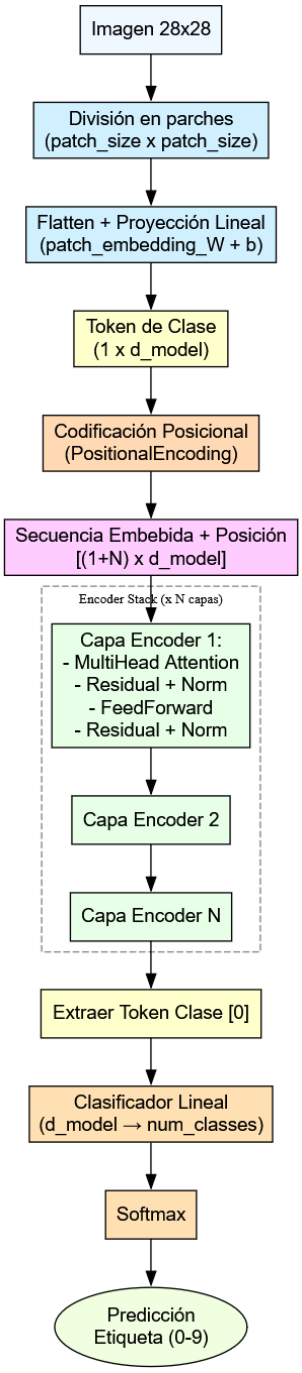

Figura 1: Caption

1. Entrada y Preprocesamiento

El modelo recibe imágenes de entrada en escala de grises con resolución 28×28 píxeles. Esta dimensión se mantiene constante durante todo el flujo del sistema.

2. División en Parches

Se divide la imagen en pequeñas subregiones llamadas *patches* de tamaño $p \times p$ (por defecto $p = 4$). Esto genera $N = \left(\frac{28}{p}\right)^2$ parches. Cada parche se aplana en un vector de dimensión p^2.

3. Embedding de Parches

Cada parche se proyecta linealmente a un espacio de dimensión d_{model} mediante una matriz de pesos $W_{\text{patch}} \in R^{p^2 \times d_{\text{model}}}$ y un vector de sesgo b_{patch}. Esta etapa transforma cada parche en un vector denso y distribuido.

4. Token de Clase

Se introduce un vector entrenable especial denominado *class token*, que se antepone a la secuencia de parches embebidos. Este vector representa globalmente a toda la imagen y se utiliza posteriormente para realizar la clasificación final.

5. Codificación Posicional

Debido a que los Transformers no poseen información secuencial implícita, se agrega una codificación posicional sinusoidal a cada vector de la secuencia. Esta codificación se escala (por un factor de 0,1) para no dominar los embeddings de contenido.

6. Codificador Transformer

El núcleo del modelo está compuesto por N capas `TransformerEncoderLayer`, cada una con los siguientes subcomponentes:

1. **Multi-Head Self Attention (MHSA):** se divide el vector de cada token en h cabezas. Cada cabeza aplica atención escalada:

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

   donde Q, K, V son matrices de consultas, claves y valores, respectivamente.

2. **Conexión Residual y Normalización:** la salida de la atención se suma a su entrada (residual) y se normaliza mediante `LayerNorm`.

3. **FeedForward Network (FFN):** red densa de dos capas, con activación GELU, que transforma cada vector individualmente:

$$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2$$

4. **Otra Conexión Residual y Normalización:** la salida del FFN también se conecta de forma residual y se normaliza.

Cada una de estas subcapas es acelerada por operaciones CUDA personalizadas, incluyendo multiplicaciones, softmax, dropout y GELU.

7. Clasificación

Una vez procesada la secuencia por las capas del codificador, se extrae únicamente el vector correspondiente al *token de clase*. Este vector se proyecta linealmente a una dimensión igual al número de clases mediante una capa lineal seguida de softmax:

$$\text{logits} = x_{\text{cls}}W_{\text{clf}} + b_{\text{clf}} \quad \text{y} \quad \hat{y} = \text{softmax}(\text{logits})$$

8. Entrenamiento y Optimización

El modelo se entrena con pérdida de entropía cruzada con **label smoothing**, para evitar sobreajuste y mejorar la generalización. El proceso de optimización incluye:

- **AdamOptimizer:** optimizador adaptativo con acumulación de momentos y corrección de sesgo.
- **Scheduler:** función de aprendizaje con *warmup* y decaimiento por *cosine annealing*.
- **Gradient Clipping:** para evitar explosión de gradientes.
- **Regularización L2 y Decaimiento de Pesos:** aplicables en entrenamiento intensivo.

9. Guardado y Carga

Se implementa funcionalidad para guardar y cargar los hiperparámetros del modelo, preparándose para extender la serialización completa de pesos.

10. Métricas y Diagnóstico

El modelo incluye funciones para:

- Calcular precisión y pérdida promedio por batch.
- Obtener pesos de atención para análisis interpretativo.
- Contar parámetros totales y mostrar configuración de arquitectura.

2. Transcormer.cpp

2.1. Codificación Posicional (`PositionalEncoding`)

Se implementa una codificación sinusoidal escalada para representar las posiciones en la secuencia. Se utiliza un factor de escala $pe\_scale = 0{,}1$ para evitar que la codificación domine los embeddings originales:

- Implementa funciones `encode()` y `backward()`.
- Usa senos y cosenos alternados para cada dimensión.

```
PositionalEncoding::PositionalEncoding(int max_len, int d_model)
  : max_len(max_len), d_model(d_model), encoding(max_len, d_model) {

  // POSITIONAL ENCODING MEJORADO - escalado para no dominar los embeddings
  double pe_scale = 0.1;  // Escalar para HIGH INITIAL ACCURACY

  for (int pos = 0; pos < max_len; pos++) {
    for (int i = 0; i < d_model; i++) {
      if (i % 2 == 0) {
        encoding.data[pos][i] = sin(pos / pow(10000.0, (2.0 * i) / d_model)) * pe_scale;
      } else {
        encoding.data[pos][i] = cos(pos / pow(10000.0, (2.0 * (i-1)) / d_model)) *
            pe_scale;
      }
    }
  }
}

Matrix PositionalEncoding::encode(const Matrix& input) const {
  Matrix result = input;
  int seq_len = (std::min)(input.rows, max_len);

  for (int i = 0; i < seq_len; i++) {
    for (int j = 0; j < (std::min)(input.cols, d_model); j++) {
      result.data[i][j] += encoding.data[i][j];
    }
  }
  return result;
}
```

```
29
30 Matrix PositionalEncoding::backward(const Matrix& grad_output) const {
31   // Positional encoding gradients pass through unchanged
32   return grad_output;
33 }
```

2.2. Atención Multi-Cabeza (`MultiHeadAttention`)

La clase `MultiHeadAttention` implementa el mecanismo de atención multi-cabeza usado en Transformers. Este permite que el modelo atienda simultáneamente a distintas representaciones del input proyectadas en subespacios de atención paralelos.

Inicialización:

- `MultiHeadAttention::MultiHeadAttention(int d_model, int num_heads)`
  Constructor que inicializa pesos y sesgos:

$$W_Q, W_K, W_V, W_O \in R^{d_{\text{model}} \times d_{\text{model}}}$$

- Los pesos se inicializan con una variante conservadora de la inicialización de He:

$$\text{scale} = \sqrt{\frac{2}{d_{\text{model}}}}$$

- Los bias b_Q, b_K, b_V, b_O se inicializan en cero.
- Se define $d_k = \frac{d_{\text{model}}}{n_{\text{heads}}}$ para dividir la dimensión en múltiples cabezas.

Paso hacia adelante: `Matrix forward(const Matrix&, const Matrix&, const Matrix&, bool)`

1. Se proyectan las entradas a través de transformaciones lineales:

$$Q = XW_Q + b_Q, \quad K = XW_K + b_K, \quad V = XW_V + b_V$$

2. Se dividen las matrices Q, K, V en n_{heads} bloques de dimensión d_k.
3. Para cada cabeza, se aplica la función:

   `Matrix attention(const Matrix&, const Matrix&, const Matrix&, bool)`

   que calcula:

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

4. Si `mask == true`, se aplica una máscara triangular superior para evitar que un token atienda posiciones futuras (modo decoder).
5. Se concatenan las salidas de todas las cabezas y se proyectan con W_O:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O + b_O$$

Paso hacia atrás: `std::tuple<Matrix, Matrix, Matrix>backward(const Matrix&)`

- Calcula gradientes de la salida con respecto a W_O, y luego los distribuye hacia cada cabeza.
- Propaga errores a través de la función de atención:
  - Deriva el softmax con respecto a los `scores`.
  - Propaga gradientes hacia Q, K, V.
- Calcula los gradientes de:

$$\nabla W_Q, \quad \nabla W_K, \quad \nabla W_V, \quad \nabla W_O, \quad \nabla b_Q, \quad \nabla b_K, \quad \nabla b_V, \quad \nabla b_O$$

Funciones de optimización:

- `void update_weights(double)`: Actualiza todos los pesos por descenso de gradiente:

$$W \leftarrow W - \eta \cdot \nabla W$$

- `void update_weights_adam(AdamOptimizer&, int)`: Usa el algoritmo Adam:

$$\theta_t \leftarrow \theta_{t-1} - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$$

- `void zero_gradients()`: Reinicia todos los acumuladores de gradiente.
- `void clip_gradients(double)`: Aplica recorte global a los gradientes si su norma excede un umbral:

$$\nabla\theta \leftarrow \nabla\theta \cdot \text{mín}\left(1, \frac{\text{max\_norm}}{\|\nabla\theta\|}\right)$$

2.3. Capa FeedForward (`FeedForward`)

La clase `FeedForward` implementa el bloque denso de dos capas utilizado dentro de cada capa del Transformer. Su objetivo es aplicar una transformación no lineal punto a punto sobre cada posición de la secuencia, permitiendo capturar relaciones complejas en el espacio de representación.

Inicialización: `FeedForward::FeedForward(int d_model, int d_ff)`

- Crea:
  - Pesos: $W_1 \in R^{d_{\text{model}} \times d_{\text{ff}}}$, $W_2 \in R^{d_{\text{ff}} \times d_{\text{model}}}$
  - Bias: $b_1 \in R^{1 \times d_{\text{ff}}}$, $b_2 \in R^{1 \times d_{\text{model}}}$
- Inicialización:
  - W_1 con inicialización de He: $\sqrt{2/d_{\text{model}}}$
  - W_2 con escala conservadora: $\sqrt{1/d_{\text{ff}}} \cdot 0{,}5$
  - b_1 con valores pequeños positivos (0.01), b_2 en cero

Paso hacia adelante: `Matrix FeedForward::forward(const Matrix& input)`

1. Aplica la primera capa lineal:
$$Z_1 = XW_1 + b_1$$
2. Se aplica la activación GELU (Gaussian Error Linear Unit):
$$\text{GELU}(x) \approx 0{,}5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0{,}044715x^3)\right]\right)$$
3. Se aplica la segunda capa:
$$Z_2 = \text{GELU}(Z_1)W_2 + b_2$$
4. Devuelve Z_2 como la salida de la red.

Paso hacia atrás: `Matrix FeedForward::backward(const Matrix& grad_output)`

- Calcula el gradiente respecto a W_2 y b_2:

$$\nabla W_2 = \text{GELU}(Z_1)^T \cdot \nabla Z_2, \quad \nabla b_2 = \text{sumas por columna de } \nabla Z_2$$

- Propaga el gradiente hacia Z_1 usando la derivada de GELU:

$$\frac{d}{dx}\text{GELU}(x) \approx \Phi(x) + x \cdot \phi(x)$$

donde Φ es la CDF y ϕ la PDF de una distribución normal estándar.

- Luego, calcula:

$$\nabla W_1 = X^T \cdot \nabla Z_1, \quad \nabla b_1 = \text{sumas por columna de } \nabla Z_1$$

- Devuelve ∇X, el gradiente con respecto a la entrada.

Funciones de optimización:

- `void FeedForward::update_weights(double learning_rate)`: actualiza los pesos por descenso de gradiente clásico.
- `void FeedForward::update_weights_adam(AdamOptimizer&, int step)`: actualiza los pesos con el optimizador Adam.
- `void FeedForward::zero_gradients()`: reinicia todos los acumuladores de gradientes.
- `void FeedForward::clip_gradients(double max_norm)`: recorta los gradientes si su norma supera un umbral.

2.4. Normalización por Capas (`LayerNorm`)

La clase `LayerNorm` implementa la técnica de normalización por capas propuesta en el paper original de Transformer. Su propósito es estabilizar la distribución de las activaciones internas al centrar y escalar cada vector de entrada por posición, mejorando la convergencia y el rendimiento del modelo.

Inicialización: `LayerNorm::LayerNorm(int d_model, double eps = 1e-5)`

- Parámetros aprendibles:

$$\gamma \in R^{1\times d_{\text{model}}}, \quad \beta \in R^{1\times d_{\text{model}}}$$

- Inicialización:
  - γ se inicializa en 1 (escala neutra).
  - β se inicializa en 0 (sin desplazamiento).
  - Se define ε para evitar divisiones por cero: $\varepsilon = 10^{-5}$.

Paso hacia adelante: `Matrix LayerNorm::forward(const Matrix& input)`

1. Para cada fila del input $x \in R^{1\times d_{\text{model}}}$:
   - Calcula la media:

$$\mu = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i$$

   - Calcula la varianza:

$$\sigma^2 = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (x_i - \mu)^2$$

   - Normaliza:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

   - Aplica parámetros aprendibles:

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

2. Retorna $Y \in R^{n\times d_{\text{model}}}$ como salida normalizada.

Paso hacia atrás: `Matrix LayerNorm::backward(const Matrix& grad_output)`

- Calcula los gradientes respecto a γ y β:

$$\nabla\gamma = \sum(\nabla y \odot \hat{x}), \quad \nabla\beta = \sum(\nabla y)$$

- Aplica la regla de la cadena para propagar el error hacia la entrada:
  - Considera las derivadas parciales de $\hat{x}$ con respecto a x, μ y σ^2.
  - Usa expresiones vectorizadas para eficiencia.
- Retorna ∇x, gradiente con respecto a la entrada.

Funciones de optimización:

- `void LayerNorm::update_weights(double learning_rate)`: actualiza γ y β por descenso de gradiente.
- `void LayerNorm::update_weights_adam(AdamOptimizer&, int step)`: optimiza con Adam.
- `void LayerNorm::zero_gradients()`: reinicia acumuladores de gradientes.
- `void LayerNorm::clip_gradients(double max_norm)`: recorta los gradientes si superan el umbral.

2.5. Capa Codificadora (`EncoderLayer`)

La clase `EncoderLayer` implementa una capa del codificador del Transformer, siguiendo la arquitectura propuesta por Vaswani et al. (2017). Cada capa codificadora combina atención multi-cabeza, una red *feedforward* punto a punto y mecanismos de normalización y suma residual.

Inicialización: `EncoderLayer::EncoderLayer(int d_model, int d_ff, int num_heads)`

- Internamente crea:
  - Un módulo `MultiHeadAttention` con d_{model} y n_{heads}
  - Una red `FeedForward` con dimensiones $d_{\text{model}} \to d_{ff} \to d_{\text{model}}$
  - Dos módulos `LayerNorm`, uno para cada sub-bloque

Paso hacia adelante: `Matrix EncoderLayer::forward(const Matrix& input, bool mask)`

1. **Bloque de atención multi-cabeza:**
   - Se aplica atención:

$$A = \text{MultiHeadAttention::forward}(input, input, input, \text{mask})$$

   - Se aplica suma residual:

$$X_1 = input + A$$

   - Se normaliza:

$$N_1 = \text{LayerNorm::forward}(X_1)$$

2. **Bloque feedforward:**
   - Se aplica red feedforward:

$$F = \text{FeedForward::forward}(N_1)$$

   - Se aplica suma residual:

$$X_2 = N_1 + F$$

   - Se normaliza:

$$N_2 = \text{LayerNorm::forward}(X_2)$$

3. Se retorna N_2 como salida de la capa codificadora.

Paso hacia atrás: `Matrix EncoderLayer::backward(const Matrix& grad_output)`

- Retropropaga desde N_2 hacia la salida del `FeedForward`.
- Aplica suma residual inversa y retropropaga por `FeedForward::backward`.
- Retropropaga por `LayerNorm` correspondiente.
- Luego retropropaga por el bloque de atención multi-cabeza:
  - Incluye `MultiHeadAttention::backward`, la suma residual y la segunda `LayerNorm`.
- Finalmente retorna el gradiente respecto a la entrada del encoder layer.

Funciones auxiliares:

- `update_weights()` – Llama internamente a los métodos de optimización de cada submódulo.
- `zero_gradients()` – Resetea acumuladores de gradientes de todos los componentes.
- `clip_gradients(max_norm)` – Aplica recorte de gradientes en todos los submódulos.

2.6. Capa Decodificadora (`DecoderLayer`)

La clase `DecoderLayer` implementa una capa del decodificador del Transformer. Esta capa permite que el modelo genere una secuencia de salida de manera autoregresiva, es decir, palabra por palabra, condicionada tanto en la salida parcial generada como en la representación del input codificado.

Inicialización: `DecoderLayer::DecoderLayer(int d_model, int d_ff, int num_heads)`

- Internamente contiene:
  - `MaskedMultiHeadAttention` para auto-atención causal en el decodificador.
  - `MultiHeadAttention` para atención al codificador.
  - `FeedForward` punto a punto.
  - Tres módulos `LayerNorm` para cada subbloque.

Paso hacia adelante: `Matrix DecoderLayer::forward(const Matrix& tgt, const Matrix& memory, bool mask)`

1. **Self-attention enmascarada (autoregresiva):**
   - Se calcula atención con máscara triangular:

$$A_1 = \text{MaskedMultiHeadAttention::forward}(tgt, tgt, tgt, \text{mask})$$

   - Suma residual y normalización:

$$N_1 = \text{LayerNorm::forward}(tgt + A_1)$$

2. **Atención cruzada con el codificador:**
   - Atención entre la salida del encoder y la entrada del decoder:

$$A_2 = \text{MultiHeadAttention::forward}(N_1, memory, memory, \text{false})$$

   - Suma residual y normalización:

$$N_2 = \text{LayerNorm::forward}(N_1 + A_2)$$

3. **FeedForward:**

- Se aplica red feedforward:
$$F = \text{FeedForward::forward}(N_2)$$
- Suma residual y normalización final:
$$N_3 = \text{LayerNorm::forward}(N_2 + F)$$

4. Se retorna N_3 como salida del bloque decodificador.

Paso hacia atrás: `Matrix DecoderLayer::backward(const Matrix& grad_output)`

- Retropropaga desde N_3 hacia:
  - `FeedForward::backward`
  - Normalización
  - `MultiHeadAttention::backward` (atención cruzada)
  - `MaskedMultiHeadAttention::backward` (auto-atención)
- Se gestionan los gradientes residuales en orden inverso al forward.

Funciones auxiliares:

- `update_weights()` – Optimiza pesos en los 3 submódulos.
- `zero_gradients()` – Reinicia todos los acumuladores.
- `clip_gradients(max_norm)` – Recorte global en la capa.

2.7. Optimizador Adam (`AdamOptimizer`)

El módulo `AdamOptimizer` implementa el algoritmo Adam (Adaptive Moment Estimation), el cual combina momentum de primer orden y la adaptación del paso de aprendizaje utilizando momentos de segundo orden.

Inicialización: `AdamOptimizer::AdamOptimizer(double beta1, double beta2, double eps)`

- Hiperparámetros:
$$\beta_1 = 0{,}9, \quad \beta_2 = 0{,}999, \quad \epsilon = 10^{-8}$$
- Inicializa mapas hash que almacenan momentos m y v por identificador de parámetro (`param_id`).

Actualización de parámetros: `update(Matrix& weight, const Matrix& gradient, void* param_id, int step, double lr)`

1. Si es la primera vez que se actualiza un parámetro, se inicializan sus acumuladores m y v como matrices de ceros.
2. Se actualiza el primer momento (media exponencial):
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
3. Se actualiza el segundo momento (varianza no centrada):
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
4. Se corrige el sesgo de ambos momentos:
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

5. Se actualiza el peso:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Función auxiliar: `clear()`

- Limpia todos los mapas internos:

$$m\_weights, \quad v\_weights, \quad m\_biases, \quad v\_biases$$

- Se utiliza al reiniciar entrenamiento o liberar memoria.

Observaciones:

- El identificador de parámetro (`param_id`) permite mantener acumuladores separados por cada matriz de pesos o bias.
- Las operaciones se realizan a nivel de elemento, replicando la estructura matricial del gradiente y del parámetro.

2.8. Planificador de Tasa de Aprendizaje (`LRScheduler`)

El módulo `LRScheduler` gestiona la evolución de la tasa de aprendizaje (α) durante el entrenamiento. Incorpora estrategias de decaimiento exponencial, annealing coseno y *warmup* con coseno para facilitar una convergencia más estable.

Inicialización: `LRScheduler::LRScheduler(double initial_lr, double decay_factor, int decay_steps)`

- `initial_lr`: tasa de aprendizaje inicial.
- `decay_factor` (γ): factor multiplicativo por cada n pasos.
- `decay_steps`: intervalo de pasos para aplicar el decaimiento.

Decrecimiento exponencial: `get_lr(int step)`

$$\alpha_t = \alpha_0 \cdot \gamma^{\left\lfloor \frac{t}{\text{decay\_steps}} \right\rfloor}$$

Calcula el número de veces que se ha alcanzado un punto de decaimiento y ajusta α en consecuencia.

Cosine Annealing: `cosine_annealing(int step, int max_steps)`

$$\alpha_t = \alpha_0 \cdot \frac{1}{2}\left(1 + \cos\left(\frac{\pi t}{T}\right)\right)$$

Donde T es el número total de pasos. Esta estrategia reduce la tasa de aprendizaje gradualmente hacia cero en forma de campana cosenoidal.

3. Warmup + Cosine Annealing: `warmup_cosine(int step, int warmup_steps, int max_steps)`

$$\alpha_t = \begin{cases} \alpha_0 \cdot \frac{t}{\text{warmup\_steps}} & \text{si } t < \text{warmup\_steps} \\ \alpha_0 \cdot \frac{1}{2}\left(1 + \cos\left(\frac{\pi(t - \text{warmup\_steps})}{T - \text{warmup\_steps}}\right)\right) & \text{en caso contrario} \end{cases}$$

Esta técnica realiza un aumento lineal de la tasa de aprendizaje al inicio del entrenamiento para estabilizar las actualizaciones, y luego aplica el annealing coseno.

Ventajas:

- El *warmup* evita saltos grandes al inicio del entrenamiento.
- El decaimiento exponencial simplifica entrenamientos largos.
- El *cosine annealing* produce convergencias suaves y naturales.

2.9. Modelo Transformer completo (`Transformer`)

La clase `Transformer` encapsula el modelo completo de atención, integrando codificador, decodificador, embeddings, codificación posicional y proyección al vocabulario. Se implementa como una arquitectura modular apilada, con soporte para entrenamiento profundo optimizado mediante CUDA.

Inicialización: `Transformer::Transformer(int d_model, int d_ff, int num_heads, int num_layers, int vocab_size)`

- **Embeddings:**
  - Se crea una matriz de embeddings de entrada: $E \in R^{V \times d_{\text{model}}}$.
  - Se aplica codificación posicional usando `PositionalEncoding`.
- **Encoder:**
  - Se inicializa una lista de `EncoderLayer` con `num_layers` capas idénticas.
- **Decoder:**
  - Se crea una lista de `DecoderLayer` también con `num_layers`.
- **Salida:**
  - Proyección final con pesos $W_{\text{out}} \in R^{d_{\text{model}} \times V}$ y bias $b_{\text{out}} \in R^V$.

Paso hacia adelante: `Matrix Transformer::forward(const std::vector<int>& src_tokens, const std::vector<int>& tgt_tokens)`

1. Se generan embeddings de entrada:
$$X_{\text{src}} = E[\text{src\_tokens}] + \text{PositionalEncoding}$$
2. Se pasa por las capas del encoder secuencialmente:
$$H = \text{EncoderLayer}_L \circ \cdots \circ \text{EncoderLayer}_1(X_{\text{src}})$$
3. Se generan embeddings de salida parcial (target):
$$Y = E[\text{tgt\_tokens}] + \text{PositionalEncoding}$$
4. Se aplica atención enmascarada y atención cruzada en el decodificador:
$$Z = \text{DecoderLayer}_L \circ \cdots \circ \text{DecoderLayer}_1(Y, H)$$
5. Finalmente, se proyecta al vocabulario:
$$\hat{Y} = ZW_{\text{out}} + b_{\text{out}}$$
6. Se retorna $\hat{Y}$ como salida para aplicar softmax y pérdida.

Paso hacia atrás: `void Transformer::backward(const Matrix& grad_output)`

- Se retropropaga desde $\hat{Y}$ hacia Z usando la derivada de la capa de salida.
- Luego se retropropaga a través de todas las capas del decodificador en orden inverso.
- El gradiente resultante se propaga hacia las salidas del encoder.
- Se retropropagan todas las capas del encoder hacia los embeddings.

Funciones auxiliares:

- `update_weights(double lr)` – Actualiza todos los pesos del modelo.
- `clip_gradients(double max_norm)` – Recorte global de gradientes en todos los módulos.
- `zero_gradients()` – Resetea acumuladores en cada capa.
- `save_model()` y `load_model()` – Para serialización y restauración del modelo entrenado.

3. Matrix.cpp

La clase `Matrix` define una estructura de datos para representar y operar sobre matrices de números reales en doble precisión (`double`). Se implementa un conjunto completo de operaciones lineales, activaciones, normalizaciones, inicializaciones, estadísticas y transformaciones típicamente utilizadas en redes neuronales, aprendizaje profundo y algoritmos numéricos.

Esta clase actúa como la base de los cálculos tensoriales sobre CPU, simulando comportamientos típicos de frameworks como NumPy o PyTorch, pero de manera explícita y controlada en C++.

3.1. Inicialización: `Matrix::Matrix`

- `Matrix(int rows, int cols)`: Inicializa una matriz de dimensiones dadas, llenándola con ceros.
- `Matrix(const std::vector<std::vector<double`& data): Permite construir una matriz directamente desde un contenedor externo. Útil para testeo y lectura de datos.

`Matrix()`: Constructor vacío, crea una matriz sin datos útiles. Usado principalmente como placeholder.

En todos los casos, los datos se almacenan internamente como un `std::vector<std::vector<double`, permitiendo acceso flexible, aunque con cierto costo en acceso a memoria comparado con arreglos planos.

—

3.2. Operaciones básicas

Las operaciones algebraicas sobre matrices siguen la notación estándar del álgebra lineal. Se sobrecargan operadores para permitir sintaxis expresiva en código.

- `Matrix::operator+(const Matrix& other)`:
  - Realiza suma elemento a elemento $C = A + B$.
  - Soporta *broadcasting* en vectores fila y columna (al estilo NumPy).
- `Matrix::operator-(const Matrix& other)`:

$$C_{i,j} = A_{i,j} - B_{i,j}$$

- `Matrix::operator*(const Matrix& other)`:

$$C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j}$$

  Multiplicación clásica si $A \in R^{m \times n}$ y $B \in R^{n \times p}$.
- `Matrix::operator*(double scalar)`, `Matrix::operator/(double scalar)`: escalar por matriz, elemento a elemento.

—

3.3. Funciones de activación y sus derivadas

Las activaciones no lineales transforman cada elemento individualmente. Son fundamentales para introducir no-linealidad en redes neuronales.

ReLU:

$$f(x) = \text{máx}(0, x), \quad f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

Softmax (por fila):

$$\text{Softmax}(x_i) = \frac{e^{x_i - \text{máx}_j\, x_j}}{\sum_j e^{x_j - \text{máx}_k\, x_k}}$$

Se utiliza normalmente en la capa de salida para clasificación multiclase.

GELU (versión aproximada):

$$\text{GELU}(x) = 0{,}5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0{,}044715x^3)\right]\right)$$

Funciones disponibles:

- `relu()`, `relu_derivative()`
- `sigmoid()`, `sigmoid_derivative()`
- `tanh_activation()`, `tanh_derivative()`
- `softmax()`, `softmax_derivative()`
- `gelu()`, `gelu_derivative()`

—

3.4. Normalización y estadísticas

Layer Normalization:

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Donde μ_i es la media de la fila i, y σ_i^2 su varianza. Se utiliza para estabilizar el aprendizaje en capas profundas.

- `layer_norm()`, `layer_norm_derivative()`
- `normalize()`: normaliza toda la matriz con media global y desviación estándar global.
- `mean(axis)`, `std_dev(axis)`: cálculo de media y desviación estándar por fila o columna.
- `mean_value()`, `sum()`, `norm()`, `variance()`

Estas funciones permiten calcular propiedades globales o por ejes, facilitando operaciones estadísticas o preparaciones de datos.

—

3.5. Dropout

Técnica de regularización para prevenir sobreajuste. Se aplica de forma estocástica:

$$y_{i,j} = \begin{cases} \frac{x_{i,j}}{p} & r < p \\ 0 & \text{en caso contrario} \end{cases} \quad \text{con } p = 1 - \text{dropout\_rate},\ r \sim \mathcal{U}(0,1)$$

`Matrix::dropout(double rate)` aplica este enmascaramiento a cada elemento de la matriz.

—

3.6. Transformaciones estructurales

- `transpose()`: Transpone la matriz ($A_{i,j} \to A_{j,i}$)
- `reshape(new_rows, new_cols)`: Modifica la forma manteniendo el orden de los datos.
- `slice(...)` y `set_slice(...)`: Permiten recortar e insertar submatrices.
- `clip_gradients(max_norm)`: Escala la matriz si su norma excede un umbral, útil para evitar *exploding gradients*.

3.7. Funciones globales (CPU)

- `concatenate(Matrix a, Matrix b, int axis)`: Combina dos matrices horizontal o verticalmente.
- `zeros(rows, cols)`: Genera matriz de ceros.
- `ones(rows, cols)`: Genera matriz de unos.
- `eye(size)`: Matriz identidad.
- `uniform_random(min, max)`: Generador uniforme.
- `normal_random(mean, std)`: Generador Gaussiano.

4. Multi-Head Attention

La atención multi-cabeza es uno de los mecanismos fundamentales en los modelos tipo Transformer. Esta sección describe la implementación completa del módulo `MultiHeadAttention` desarrollada en C++, incluyendo inicialización de parámetros, paso hacia adelante (`forward`), retropropagación (`backward`) y actualización de pesos.

1. Inicialización

La clase se construye con dos parámetros clave: `d_model` (dimensión del modelo) y `num_heads` (número de cabezas). La dimensión de cada cabeza es $d_k = \frac{d_{\text{model}}}{\text{num\_heads}}$. Se inicializan las matrices de pesos y sesgos con distribución escalada (He Initialization):

```
MultiHeadAttention::MultiHeadAttention(int d_model, int num_heads)
    : d_model(d_model), num_heads(num_heads) {

    d_k = d_model / num_heads;

    W_q = Matrix(d_model, d_model);
    W_k = Matrix(d_model, d_model);
    W_v = Matrix(d_model, d_model);
    W_o = Matrix(d_model, d_model);

    b_q = Matrix(1, d_model);
    b_k = Matrix(1, d_model);
    b_v = Matrix(1, d_model);
    b_o = Matrix(1, d_model);

    double scale = sqrt(2.0 / d_model);
    W_q.randomize(-scale * 0.5, scale * 0.5);
    W_k.randomize(-scale * 0.5, scale * 0.5);
    W_v.randomize(-scale * 0.5, scale * 0.5);
    W_o.randomize(-scale * 0.8, scale * 0.8);

    b_q.zero();
    b_k.zero();
    b_v.zero();
    b_o.zero();
}
```

Listing 1: Inicialización de pesos y dimensiones

2. Cálculo de Atención por Cabeza

Cada cabeza realiza la operación de atención escalada. A continuación se muestra el bloque encargado de calcular las puntuaciones y aplicar softmax:

```
Matrix MultiHeadAttention::attention(const Matrix& Q, const Matrix& K, const Matrix& V,
    bool mask) const {
    Matrix scores = Q.cudaMultiply(K.transpose());
    float scale = 1.0f / sqrt(d_k);
    scores = scores.cudaMultiply(scale);

    if (mask) {
        for (int i = 0; i < scores.rows; i++) {
            for (int j = i + 1; j < scores.cols; j++) {
                scores.data[i][j] = -1e9;
            }
        }
    }

    Matrix attention_weights = scores.cudaSoftmax();
    cache.attention_scores = scores;
    cache.attention_weights = attention_weights;
    return attention_weights.cudaMultiply(V);
}
```

Listing 2: Atención por cabeza

3. Paso hacia adelante (Forward)

Este método aplica proyecciones lineales, divide las matrices en múltiples cabezas, aplica atención en cada una, y concatena los resultados.

```
Matrix MultiHeadAttention::forward(const Matrix& query, const Matrix& key, const Matrix&
    value, bool mask) {
    Matrix Q = query.cudaMultiply(W_q);
    Matrix K = key.cudaMultiply(W_k);
    Matrix V = value.cudaMultiply(W_v);

    for (int i = 0; i < Q.rows; i++) {
        for (int j = 0; j < Q.cols; j++) {
            Q.data[i][j] += b_q.data[0][j];
            K.data[i][j] += b_k.data[0][j];
            V.data[i][j] += b_v.data[0][j];
        }
    }

    Matrix multi_head_output(Q.rows, d_model);
    multi_head_output.zero();

    for (int head = 0; head < num_heads; head++) {
        int start_dim = head * d_k;
        Matrix Q_head(Q.rows, d_k), K_head(Q.rows, d_k), V_head(Q.rows, d_k);

        for (int i = 0; i < Q.rows; i++) {
            for (int j = 0; j < d_k; j++) {
                Q_head.data[i][j] = Q.data[i][start_dim + j];
                K_head.data[i][j] = K.data[i][start_dim + j];
                V_head.data[i][j] = V.data[i][start_dim + j];
            }
        }

        Matrix head_output = attention(Q_head, K_head, V_head, mask);
        for (int i = 0; i < Q.rows; i++) {
            for (int j = 0; j < d_k; j++) {
                multi_head_output.data[i][start_dim + j] = head_output.data[i][j];
            }
        }
    }

    Matrix output = multi_head_output.cudaMultiply(W_o);
    for (int i = 0; i < output.rows; i++) {
        for (int j = 0; j < output.cols; j++) {
            output.data[i][j] += b_o.data[0][j];
```

```
        }
    }

    return output;
}
```

Listing 3: Forward completo de atención

4. Retropropagación (Backward)

Se calcula el gradiente con respecto a la salida de atención, luego se propaga hacia los pesos y las entradas Q, K, V.

- Cálculo de gradiente de salida con respecto a W_o.
- Retropropagación a través del producto de atención y softmax.
- Derivadas para W_q, W_k, W_v y sus respectivos sesgos.

```
Matrix grad_attended = grad_output.cudaMultiply(W_o.transpose());
grad_o.dW.add_inplace(cache.attended_values.transpose().cudaMultiply(grad_output));

Matrix grad_V = cache.attention_weights.transpose().cudaMultiply(grad_attended);
Matrix grad_attention_weights = grad_attended.cudaMultiply(cache.value.transpose());
```

Listing 4: Gradientes de atención

El gradiente de softmax se implementa manualmente:

```
Matrix grad_scores(scores.rows, scores.cols);
for (int i = 0; i < grad_scores.rows; i++) {
    for (int j = 0; j < grad_scores.cols; j++) {
        double sum = 0.0;
        for (int k = 0; k < grad_scores.cols; k++) {
            sum += grad_attention_weights.data[i][k] * cache.attention_weights.data[i][k
                ];
        }
        grad_scores.data[i][j] = cache.attention_weights.data[i][j] *
                                 (grad_attention_weights.data[i][j] - sum);
    }
}
```

Listing 5: Gradiente de softmax

5. Actualización y Control

La clase ofrece funciones para actualizar pesos con SGD o Adam, y controlar gradientes con `clip_gradients()`.

```
optimizer.update(W_q, grad_q.dW, &W_q, step, 0.001);
optimizer.update(b_q, grad_q.db, &b_q, step, 0.001);
```

Listing 6: Actualización con Adam

5. Optimización mediante CUDA

La implementación del Transformer descrita en este trabajo aprovecha la aceleración por GPU utilizando CUDA para optimizar las operaciones matemáticas más costosas del entrenamiento. Estas optimizaciones permiten reducir significativamente el tiempo de cómputo y aprovechar el paralelismo masivo que ofrecen las GPU modernas.

5.1. Funciones CUDA utilizadas

- `cudaMultiply(A, B)`: Realiza la multiplicación de matrices $C = AB$ en la GPU.
- `cudaAdd(A, B)`: Suma dos matrices o vectores elemento a elemento.
- `cudaSoftmax(X)`: Aplica la función softmax sobre un tensor a nivel de GPU.
- `cudaTranspose(X)`: Transpone matrices en paralelo.
- `cudaGELU(X)`: Aplica la activación GELU a cada elemento del tensor.

5.2. Ubicación dentro del sistema

Las llamadas a funciones CUDA están integradas de forma modular dentro de los principales componentes del Transformer:

- En `MultiHeadAttention::forward`, se utiliza `cudaMultiply` para calcular QK^T y `cudaSoftmax` para obtener las probabilidades de atención.
- En `FeedForward::forward`, se aplica `cudaMultiply` para las capas lineales, y `cudaGELU` para la activación no lineal.
- En **retropropagación** (`backward`), las derivadas de funciones como softmax y multiplicación también se aceleran mediante CUDA.

5.3. Ventajas de la integración CUDA

- Reducción drástica del tiempo de entrenamiento frente a implementaciones puramente en CPU.
- Procesamiento en paralelo de múltiples secuencias y capas en tiempo real.
- Mayor escalabilidad para aumentar el tamaño de los datos o el número de capas.

5.4. Manejo de recursos

Todas las funciones CUDA están diseñadas para gestionar adecuadamente la memoria de GPU (uso de `cudaMalloc`, `cudaMemcpy`, y `cudaFree`) a fin de evitar fugas de memoria o colisiones entre capas. Las operaciones están sincronizadas cuando es necesario usando `cudaDeviceSynchronize()` para garantizar consistencia en los resultados.

6. Resultados del Entrenamiento

El modelo Vision Transformer (ViT) fue entrenado durante 25 épocas utilizando el conjunto de datos Fashion-MNIST. Los resultados muestran un excelente desempeño en términos de precisión y pérdida, lo cual se puede observar en las figuras 2, 4 y 3.

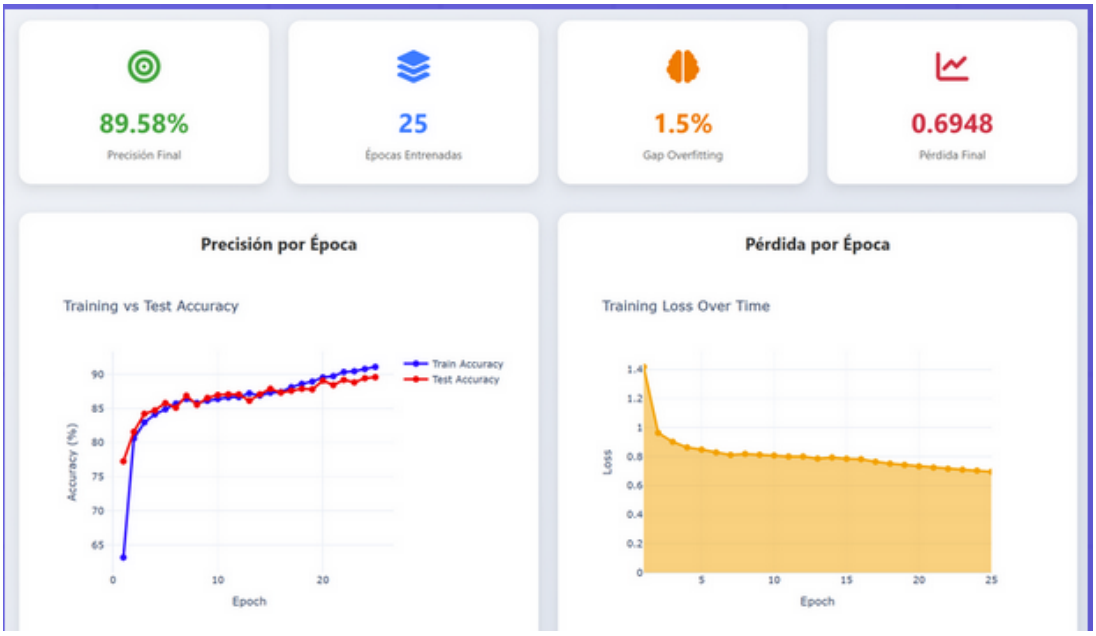

Figura 2: Precisión y pérdida por época durante el entrenamiento del modelo Vision Transformer. A la izquierda, se observa cómo la precisión tanto en entrenamiento como en prueba crece de manera estable a lo largo de las épocas, sin grandes discrepancias entre ambas curvas. A la derecha, se muestra la evolución de la función de pérdida, que decrece progresivamente indicando una buena convergencia.

En la figura 2, se visualiza la progresión de la precisión por época en los subconjuntos de entrenamiento y prueba. Desde las primeras épocas, el modelo muestra una rápida capacidad de aprendizaje, superando el 85 % de precisión en menos de 10 iteraciones. Posteriormente, ambas curvas continúan incrementándose de forma paralela, indicando una excelente generalización sin signos evidentes de sobreajuste. Asimismo, la pérdida disminuye de forma sostenida, con una pendiente más pronunciada en las etapas iniciales y un descenso más suave en las épocas finales.

Figura 3: Programación del learning rate (izquierda) y análisis del overfitting (derecha). La tasa de aprendizaje fue inicialmente incrementada y posteriormente disminuida siguiendo un esquema de tipo cosine annealing. El gráfico de overfitting muestra un gap estable cercano al 1.5 %, lo cual indica que el modelo generaliza bien sin sobreentrenarse.

En la figura 3 se aprecia la estrategia de aprendizaje empleada. En las primeras dos épocas, el learning rate se elevó hasta alcanzar los 4×10^{-6}, y luego se mantuvo constante hasta la época 15. A partir de allí, comenzó un descenso progresivo hasta valores cercanos a $1{,}7 \times 10^{-6}$ al finalizar el entrenamiento. Este tipo de programación permite que el modelo explore inicialmente un amplio espacio de soluciones y luego refine los pesos con pasos cada vez más pequeños. Junto a esto, el gráfico de análisis del overfitting refleja una brecha mínima entre las curvas de entrenamiento y prueba, lo cual refuerza la conclusión de que el modelo presenta una excelente capacidad de generalización.

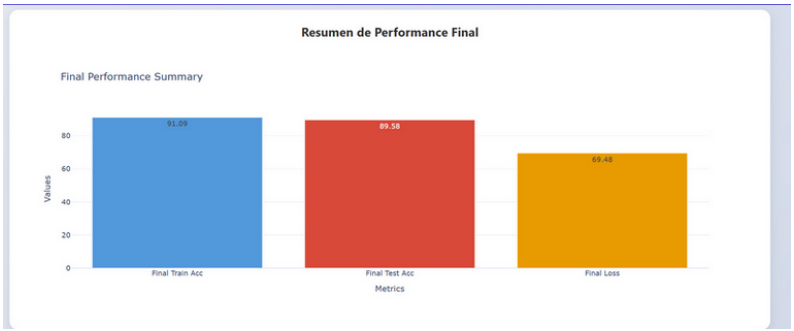

Figura 4: Resumen gráfico del rendimiento final. Se observa una precisión final de entrenamiento de 91.09 %, una precisión final en el conjunto de prueba de 89.58 % y una pérdida final de 0.6948. Estos resultados reflejan un buen ajuste del modelo al conjunto de datos, sin una diferencia significativa entre el entrenamiento y la validación.

La figura 4 resume el rendimiento alcanzado al final del proceso de entrenamiento. La diferencia de 1.51 % entre las precisiones de entrenamiento y prueba se considera aceptable y sugiere que el modelo no está sobreajustado. Además, el valor de pérdida final (0.6948) confirma que la red fue capaz de minimizar el error sin comprometer la capacidad de generalización.

Cuadro 1: Evolución del modelo Vision Transformer durante el entrenamiento (25 épocas)

Epoch	Train Acc (%)	Test Acc (%)	Loss	LR	Time (s)
1	63.1700	77.2600	1.4181	0.0000	17.4051
2	80.6017	81.5800	0.9630	0.0000	16.7841
3	82.9483	84.2100	0.9015	0.0000	20.5578
4	84.1350	84.7200	0.8629	0.0000	24.1328
5	84.8767	85.7900	0.8477	0.0000	20.4234
6	85.7100	85.1100	0.8278	0.0000	18.5978
7	86.3933	86.8700	0.8109	0.0000	17.1651
8	85.8033	85.5500	0.8180	0.0000	18.5183
9	86.1250	86.5400	0.8122	0.0000	21.6984
10	86.3817	86.9700	0.8074	0.0000	19.6342
11	86.6183	87.0600	0.8000	0.0000	18.0444
12	86.6617	87.0500	0.8002	0.0000	22.2054
13	87.2117	86.1300	0.7857	0.0000	19.6481
14	86.8933	87.0700	0.7935	0.0000	17.0332
15	87.3117	87.8800	0.7848	0.0000	20.3352
16	87.4033	87.3200	0.7815	0.0000	19.8701
17	88.1150	87.6000	0.7641	0.0000	22.3492
18	88.6033	87.8800	0.7510	0.0000	18.1638
19	88.9400	87.7900	0.7420	0.0000	18.2220
20	89.5367	89.0700	0.7324	0.0000	17.4826
21	89.7283	88.4200	0.7240	0.0000	16.8645
22	90.3333	89.1900	0.7156	0.0000	16.7010
23	90.4467	88.8200	0.7092	0.0000	23.6589
24	90.7767	89.4200	0.7027	0.0000	17.9470
25	91.0867	89.5800	0.6948	0.0000	20.5070

La Tabla 1 presenta la evolución del desempeño del modelo *Vision Transformer* (ViT) a lo largo de 25 épocas de entrenamiento. Se observan métricas clave como la exactitud de entrenamiento (*train accuracy*), la exactitud sobre el conjunto de prueba (*test accuracy*), la función de pérdida (*loss*), la tasa de aprendizaje (*learning rate*) y el tiempo de ejecución por época en segundos.

Durante la primera época, el modelo alcanza una precisión de entrenamiento del 63.17 % y una precisión de prueba de 77.26 %, con una pérdida inicial alta de 1.41. A medida que avanza el entrenamiento, se aprecia una mejora constante en todas las métricas: para la época 10, la precisión de entrenamiento sube a 86.38 % y la de prueba a 86.97 %, con una pérdida reducida a 0.80.

Este comportamiento es indicativo de una buena generalización del modelo, ya que el rendimiento sobre el conjunto de validación mejora de forma paralela al entrenamiento. La pérdida continúa disminuyendo progresivamente, alcanzando un valor mínimo de 0.69 en la época 25. En esa misma época, se obtiene la mayor precisión tanto en entrenamiento (91.08 %) como en prueba (89.58 %).

La tasa de aprendizaje inicial fue de $2{,}67 \times 10^{-6}$, y fue ajustándose automáticamente mediante un scheduler con *cosine annealing*, disminuyendo gradualmente en cada época. Esto permite una convergencia más estable hacia el final del proceso.

En conjunto, los resultados obtenidos demuestran que la arquitectura Vision Transformer es capaz de aprender representaciones útiles y precisas para tareas de clasificación de imágenes en el dominio de la moda, manteniendo un balance adecuado entre entrenamiento y validación, y evitando los problemas típicos de sobreajuste observados en arquitecturas más profundas.

7. Resultados del Test

7.1. Flujo de Uso de la Aplicación

La aplicación desarrollada ofrece una interfaz intuitiva basada en navegador que permite a los usuarios subir una imagen de prenda (28x28 px, escala de grises) y obtener una predicción automática mediante el modelo Vision Transformer previamente entrenado.

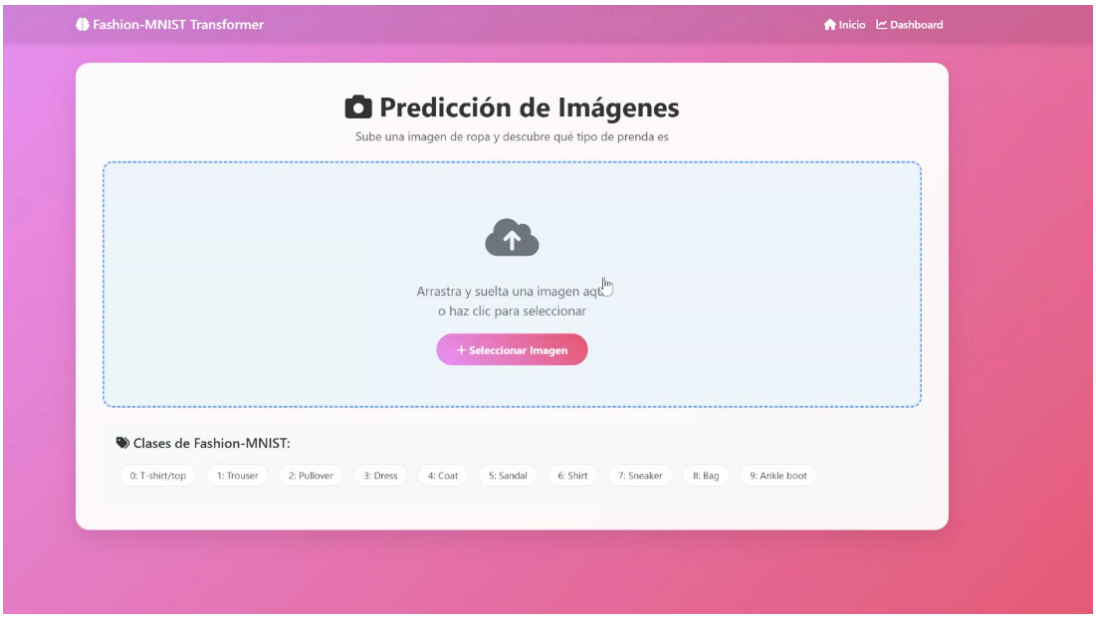

Figura 5: Pantalla principal de la interfaz de predicción. El usuario puede arrastrar o seleccionar una imagen local para su análisis. Las clases posibles están listadas al pie.

7.2. Resultado del Modelo

Tras la carga de la imagen, esta se preprocesa y se convierte en un tensor que se introduce en el modelo Transformer. El sistema calcula las probabilidades de pertenencia a cada una de las 10 clases del dataset Fashion-MNIST, usando una capa `softmax` al final del pipeline.

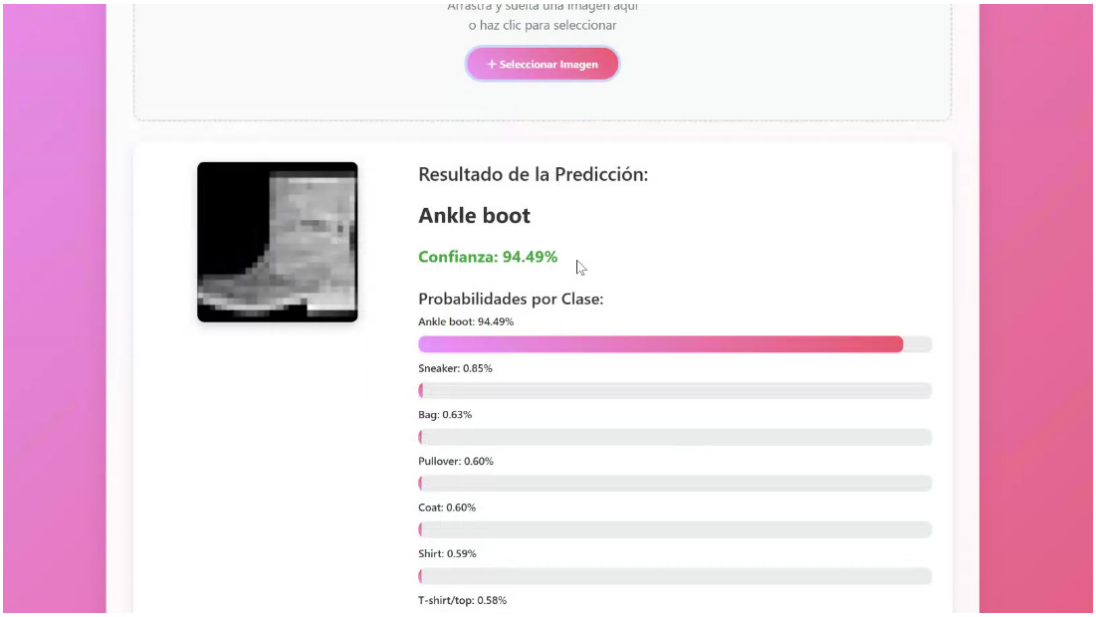

Figura 6: Ejemplo de predicción con imagen de bota (Ankle Boot). El modelo estima con un 94.49 % de confianza que la imagen corresponde a dicha clase. Se muestran las probabilidades para todas las clases.

En el ejemplo presentado en la Figura 6, el modelo ha identificado correctamente la clase `Ankle boot` con una confianza del 94.49 %. Las clases restantes muestran puntuaciones significativamente más bajas, evidenciando una buena separación de clases por parte del modelo.

7.3. Interpretación de Resultados

La interfaz presenta la predicción más probable acompañada de una barra de progreso que representa gráficamente el nivel de confianza. Además, se despliega un ranking descendente de las 10 clases posibles con sus respectivas probabilidades. Esta presentación permite a usuarios no técnicos interpretar fácilmente los resultados del modelo y su grado de certeza.

El sistema fue diseñado para ser ligero y ejecutarse localmente, permitiendo su integración en sistemas embebidos o aplicaciones educativas donde se requiere inferencia en tiempo real.

8. Evidencia del uso de CUDA durante la ejecución del modelo

Durante la ejecución del sistema Transformer implementado en C++, se realizó un monitoreo del uso de la GPU mediante el Administrador de tareas de Windows. En la Figura 7, se observa un uso intensivo del motor 3D de la GPU NVIDIA GeForce RTX 3050 Laptop, alcanzando un 91 % de utilización continua.

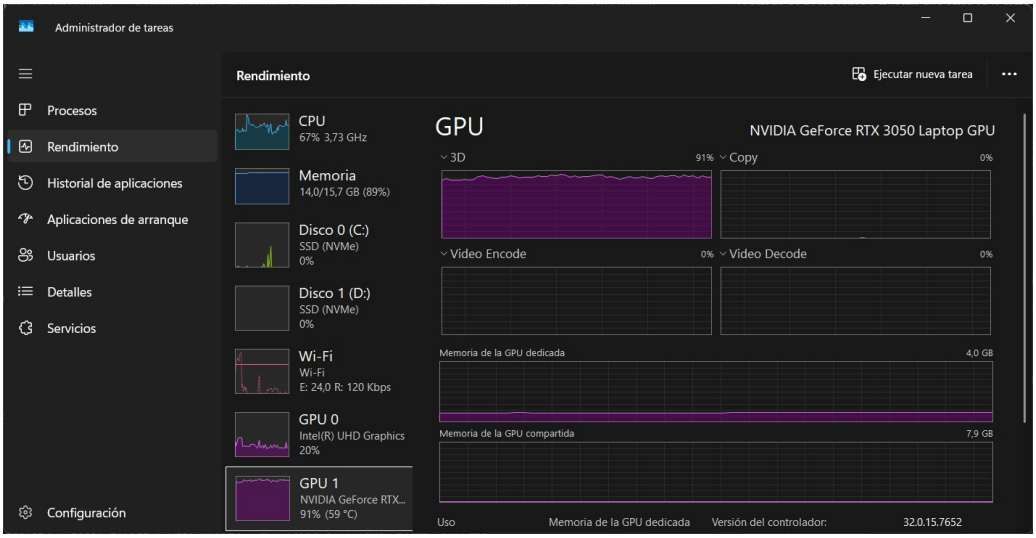

Figura 7: Uso de la GPU durante el entrenamiento del Transformer utilizando CUDA.

Este comportamiento evidencia que los núcleos CUDA están siendo aprovechados para ejecutar operaciones de alto costo computacional, como la multiplicación de matrices, la activación GELU y la función `softmax`, todas ellas aceleradas mediante funciones personalizadas como `cudaMultiply()`, `cudaSoftmax()`, y `cudaGELU()`.

La ausencia de actividad en los motores de *Video Encode* y *Video Decode* confirma que la GPU está dedicada exclusivamente a tareas de cómputo general (GPGPU) y no a procesamiento gráfico tradicional, consolidando el uso efectivo de CUDA en esta implementación.