UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA ELECTRÓNICA

Deep Q-Learning utilizando CUDA Unified Memory

Proyecto Final de Robótica

Integrantes:

DÍAZ CASTRO, BERLY JOEL

MARIÑOS HILARIO, PRINCCE YORWIN

YANQUI VERA, HENRY ARON

CÁCERES CUBA, JAYAN MICHAEL

APAZA CONDORI, JHON ANTHONY

ARONI JARATA, ANTONY

CARAZAS QUISPE, ALESSANDER JESUS

Arequipa, Perú

19 de diciembre de 2025

Índice

1. Introducción

Este proyecto implementa un agente de aprendizaje por refuerzo basado en *Deep Q-Learning* (DQN) optimizado para procesamiento paralelo mediante CUDA. El objetivo es entrenar un agente que navegue de manera óptima en un entorno de grilla (GridWorld) de 10x10 celdas, evitando obstáculos y alcanzando una meta predefinida.

1.1. Motivación

El aprendizaje por refuerzo ha demostrado ser una técnica poderosa para resolver problemas de toma de decisiones secuenciales. Sin embargo, el entrenamiento de redes neuronales profundas puede ser computacionalmente intensivo. Este proyecto aprovecha la arquitectura paralela de las GPUs mediante CUDA para acelerar significativamente el proceso de entrenamiento.

1.2. Objetivos

- Implementar un algoritmo DQN completo con Double DQN
- Optimizar el código para Jetson AGX Xavier (ARM64 + CUDA)
- Utilizar Unified Memory para simplificar la gestión de memoria
- Lograr convergencia en un entorno GridWorld con obstáculos
- Analizar el rendimiento y la calidad del aprendizaje

2. Marco Teórico

2.1. Aprendizaje por Refuerzo

El aprendizaje por refuerzo es un paradigma de aprendizaje automático donde un agente aprende a tomar decisiones interactuando con un entorno. El proceso se modela como un *Proceso de Decisión de Markov* (MDP) definido por la tupla (S, A, P, R, γ):

- S: Espacio de estados
- A: Espacio de acciones
- P: Función de transición de estados
- R: Función de recompensa
- γ: Factor de descuento ($0 \leq \gamma < 1$)

El objetivo del agente es aprender una política $\pi : S \to A$ que maximice el retorno esperado:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

2.2. Q-Learning

Q-Learning es un algoritmo de aprendizaje por refuerzo libre de modelo que aprende la función de valor de acción óptima $Q^*(s, a)$, definida como:

$$Q^*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(s', a') \mid s_t = s, a_t = a\right] \tag{2}$$

La actualización de Q-Learning se realiza mediante:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right] \tag{3}$$

donde α es la tasa de aprendizaje.

2.3. Deep Q-Network (DQN)

DQN extiende Q-Learning utilizando una red neuronal profunda para aproximar la función $Q(s, a; \theta)$, donde θ son los parámetros de la red. Las principales innovaciones de DQN incluyen:

2.3.1. Experience Replay

Un buffer de experiencias almacena transiciones $(s, a, r, s', \mathit{done})$. Durante el entrenamiento, se muestrean mini-lotes aleatorios de este buffer, rompiendo la correlación temporal entre experiencias consecutivas y mejorando la estabilidad del aprendizaje.

2.3.2. Target Network

Se mantiene una red objetivo $Q(s, a; \theta^-)$ con parámetros θ^- que se actualizan periódicamente copiando los parámetros de la red principal θ. El objetivo de la función de pérdida es:

$$y_i = r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-) \tag{4}$$

La función de pérdida es:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[(y - Q(s, a; \theta))^2\right] \tag{5}$$

2.4. Double DQN

Double DQN mejora DQN separando la selección y evaluación de acciones para reducir la sobreestimación de valores Q:

$$y_i = r_i + \gamma Q(s'_i, \arg\max_{a'} Q(s'_i, a'; \theta); \theta^-) \tag{6}$$

La red principal selecciona la mejor acción, mientras que la red objetivo evalúa su valor.

3. Arquitectura del Sistema

3.1. Entorno: GridWorld

El entorno consiste en una grilla de 10×10 celdas donde:

- El agente comienza en la posición $(0, 0)$
- La meta está ubicada en $(9, 9)$
- Existen obstáculos en posiciones específicas: $(1, 1)$, $(2, 2)$, $(3, 1)$
- El agente puede moverse en 4 direcciones: arriba, abajo, izquierda, derecha

3.1.1. Espacio de Estados

El estado se representa como un vector one-hot de dimensión 100 (10×10), donde solo la posición actual del agente tiene valor 1.

3.1.2. Función de Recompensa

La función de recompensa está diseñada para guiar al agente hacia la meta:

- Alcanzar la meta: +10,0
- Acercarse a la meta: +1,0
- Alejarse de la meta: −2,0
- Permanecer a la misma distancia: −0,5
- Chocar con pared u obstáculo: −3,0

La distancia se calcula usando la métrica de Manhattan:

$$d(s) = |x_{agent} - x_{goal}| + |y_{agent} - y_{goal}| \tag{7}$$

3.2. Arquitectura de la Red Neuronal

La red neuronal implementada tiene la siguiente arquitectura:

- **Capa de entrada:** 100 neuronas (estado one-hot del GridWorld)
- **Capa oculta:** 128 neuronas con activación ReLU
- **Capa de salida:** 4 neuronas (valores Q para cada acción)

La función ReLU se define como:

$$\text{ReLU}(x) = \text{máx}(0, x) \tag{8}$$

3.2.1. Forward Pass

El cálculo del forward pass se realiza en dos etapas:
Capa oculta:

$$h_i = \text{ReLU}\left(\sum_{j=1}^{100} W_{ij}^{(1)} s_j + b_i^{(1)}\right), \quad i = 1, \ldots, 128 \tag{9}$$

Capa de salida:

$$Q(s, a) = \sum_{j=1}^{128} W_{aj}^{(2)} h_j + b_a^{(2)}, \quad a = 1, \ldots, 4 \tag{10}$$

3.3. Hiperparámetros

Parámetro	**Valor**	**Descripción**
Tamaño de grilla	10×10	Dimensión del entorno
Tamaño de estado	100	Dimensión del vector de estado
Número de acciones	4	Arriba, abajo, izquierda, derecha
Neuronas ocultas	128	Tamaño de la capa oculta
Tamaño de lote	64	Mini-lote para entrenamiento
Tamaño de buffer	10,000	Capacidad del replay buffer
Learning rate inicial	0.001	Tasa de aprendizaje inicial
Learning rate mínimo	0.0001	Tasa de aprendizaje mínima
Decay de LR	0.9999	Factor de decaimiento de LR
Factor de descuento (γ)	0.95	Descuento de recompensas futuras
ϵ inicial	1.0	Exploración inicial (100 %)
ϵ final	0.1	Exploración mínima (10 %)
Decay de ϵ	0.999	Factor de decaimiento de ϵ
Frecuencia de actualización	1,000 pasos	Actualización de target network
Episodios de entrenamiento	800	Número máximo de episodios
Pasos máximos por episodio	50	Límite de pasos por episodio
Frecuencia de entrenamiento	4 pasos	Entrenamiento cada N pasos
Gradient clipping	5.0	Umbral para recorte de gradientes
TD error clipping	5.0	Umbral para recorte de error TD

Cuadro 1: Hiperparámetros del sistema DQN

4. Implementación en CUDA

4.1. Unified Memory

Una de las características principales de esta implementación es el uso de *CUDA Unified Memory*, que simplifica la gestión de memoria entre CPU y GPU. Con Unified Memory, los punteros son accesibles tanto desde el host (CPU) como desde el device (GPU), y el sistema de memoria maneja automáticamente las transferencias de datos.

4.1.1. Ventajas de Unified Memory

- Simplifica el código eliminando copias explícitas host-device
- Reduce errores de programación relacionados con gestión de memoria
- Optimizado para arquitecturas modernas como Jetson AGX Xavier
- Permite acceso concurrente CPU-GPU en hardware compatible

4.1.2. Ejemplo de Asignación

```
// Tradicional CUDA
float* d_W1;
cudaMalloc(&d_W1, size * sizeof(float));
cudaMemcpy(d_W1, h_W1, size * sizeof(float), cudaMemcpyHostToDevice);

// Unified Memory
float* W1;
cudaMallocManaged(&W1, size * sizeof(float));
// Accesible directamente desde CPU y GPU
```

Listing 1: Asignación de memoria unificada

4.2. Kernels CUDA Implementados

4.2.1. Forward Pass - Capa Oculta

```
__global__ void forward_hidden_kernel(
    const float* W1, const float* b1, const float* states,
    float* hidden, int batch_size
) {
    int batch_idx = blockIdx.x;
    int hidden_idx = threadIdx.x;

    if (batch_idx < batch_size && hidden_idx < HIDDEN_SIZE) {
        float sum = b1[hidden_idx];
        #pragma unroll 4
        for (int j = 0; j < STATE_SIZE; j++) {
            sum += W1[hidden_idx * STATE_SIZE + j]
                   * states[batch_idx * STATE_SIZE + j];
        }
        hidden[batch_idx * HIDDEN_SIZE + hidden_idx]
            = fmaxf(0.0f, sum);  // ReLU
    }
}
```

Listing 2: Kernel para capa oculta

Configuración de ejecución:

- Bloques: `batch_size` (64)
- Hilos por bloque: `HIDDEN_SIZE` (128)
- Cada hilo calcula una neurona oculta para un ejemplo del batch

4.2.2. Forward Pass - Capa de Salida

```
__global__ void forward_output_kernel(
    const float* W2, const float* b2, const float* hidden,
    float* output, int batch_size
) {
    int batch_idx = blockIdx.x;
    int action_idx = threadIdx.x;

    if (batch_idx < batch_size && action_idx < NUM_ACTIONS) {
        float sum = b2[action_idx];
        #pragma unroll 8
        for (int j = 0; j < HIDDEN_SIZE; j++) {
            sum += W2[action_idx * HIDDEN_SIZE + j]
                   * hidden[batch_idx * HIDDEN_SIZE + j];
        }
        output[batch_idx * NUM_ACTIONS + action_idx] = sum;
    }
}
```

Listing 3: Kernel para capa de salida

4.2.3. Cálculo de TD Errors con Double DQN

```
__global__ void compute_td_errors_double_dqn_kernel(
    const float* q_policy, const float* q_policy_next,
    const float* q_target_next,
    const int* actions, const float* rewards,
    const int* dones,
    float* td_errors, int batch_size
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < batch_size) {
        int action = actions[idx];
        float current_q = q_policy[idx * NUM_ACTIONS + action];

        // Policy network selecciona la accion
        int best_next_action = 0;
        float max_q_policy = q_policy_next[idx * NUM_ACTIONS];
        for (int a = 1; a < NUM_ACTIONS; a++) {
            float q = q_policy_next[idx * NUM_ACTIONS + a];
            if (q > max_q_policy) {
                max_q_policy = q;
                best_next_action = a;
            }
        }

        // Target network evalua esa accion
        float next_q_value =
            q_target_next[idx * NUM_ACTIONS + best_next_action];

        float target_q = rewards[idx];
        if (!dones[idx]) {
            target_q += GAMMA * next_q_value;
        }

```

```
        float error = target_q - current_q;

        // Huber loss clipping
        if (error > TD_ERROR_CLIP) error = TD_ERROR_CLIP;
        if (error < -TD_ERROR_CLIP) error = -TD_ERROR_CLIP;

        td_errors[idx] = error;
    }
}
```

Listing 4: Kernel para TD errors con Double DQN

4.2.4. Cálculo de Gradientes

Los gradientes se calculan mediante retropropagación. Para la capa de salida:

```
__global__ void compute_grad_W2_kernel(
    const float* hidden, const float* td_errors,
    const int* actions,
    float* dW2, int batch_size
) {
    int action = blockIdx.x;
    int hidden_idx = threadIdx.x;

    if (action < NUM_ACTIONS && hidden_idx < HIDDEN_SIZE) {
        float grad_sum = 0.0f;
        for (int b = 0; b < batch_size; b++) {
            if (actions[b] == action) {
                grad_sum += td_errors[b]
                          * hidden[b * HIDDEN_SIZE + hidden_idx];
            }
        }
        atomicAdd(&dW2[action * HIDDEN_SIZE + hidden_idx],
                  grad_sum);
    }
}
```

Listing 5: Kernel para gradientes de W2

Para la capa oculta, se aplica la derivada de ReLU:

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases} \tag{11}$$

4.2.5. Aplicación de Gradientes

```
__global__ void apply_gradients_kernel(
    float* weights, const float* gradients, int size,
    float lr, int batch_size
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        float grad = (lr / batch_size) * gradients[idx];

        // Gradient clipping
        if (grad > GRAD_CLIP_THRESHOLD)
            grad = GRAD_CLIP_THRESHOLD;
```

```
        if (grad < -GRAD_CLIP_THRESHOLD)
            grad = -GRAD_CLIP_THRESHOLD;

        weights[idx] += grad;
    }
}
```

Listing 6: Kernel para aplicar gradientes

4.3. Optimizaciones Implementadas

4.3.1. Inicialización de Pesos: He Initialization

Se utiliza He initialization para las capas con activación ReLU:

$$W^{(l)} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right) \tag{12}$$

Para redes pequeñas, se amplifica la desviación estándar:

```
float std1 = sqrtf(2.0f / STATE_SIZE) * 2.0f;  // Amplificado x2
float std2 = sqrtf(2.0f / HIDDEN_SIZE) * 1.5f; // Amplificado x1.5
```

Listing 7: Inicialización mejorada de pesos

4.3.2. Gradient Clipping

Para evitar explosión de gradientes:

$$g_{\text{clipped}} = \begin{cases} g & \text{si } |g| \leq \theta \\ \theta \cdot \text{sign}(g) & \text{si } |g| > \theta \end{cases} \tag{13}$$

donde $\theta = 5{,}0$ es el umbral de recorte.

4.3.3. Learning Rate Decay

$$\alpha_t = \text{máx}(\alpha_{\text{mín}}, \alpha_{t-1} \cdot \beta) \tag{14}$$

donde $\beta = 0{,}9999$ y $\alpha_{\text{mín}} = 0{,}0001$.

4.3.4. Epsilon-Greedy Decay

$$\epsilon_t = \text{máx}(\epsilon_{\text{mín}}, \epsilon_{t-1} \cdot \delta) \tag{15}$$

donde $\delta = 0{,}999$ y $\epsilon_{\text{mín}} = 0{,}1$.

5. Algoritmo de Entrenamiento

Algorithm 1 DQN Training Loop

Inicializar red policy $Q(s, a; \theta)$ con pesos aleatorios
Inicializar red target $Q(s, a; \theta^-) \leftarrow Q(s, a; \theta)$
Inicializar replay buffer $\mathcal{D}$ con capacidad N
Llenar buffer con 1000 experiencias aleatorias
for episode = 1 to M **do**
    Reiniciar entorno: $s \leftarrow s_0$
    for step = 1 to T **do**
        Seleccionar acción: $a = \begin{cases} \text{random} & \text{con prob. } \epsilon \\ \arg\max_a Q(s, a; \theta) & \text{caso contrario} \end{cases}$
        Ejecutar a, observar r, s', determinar si *done*
        Almacenar transición $(s, a, r, s', done)$ en $\mathcal{D}$
        if step mod TRAIN_FREQ = 0 **then**
            Muestrear mini-batch $(s_j, a_j, r_j, s'_j, done_j)$ de $\mathcal{D}$
            Calcular $Q(s_j, a; \theta)$ para todas las acciones
            Calcular $Q(s'_j, a; \theta)$ para selección de acción
            Calcular $Q(s'_j, a; \theta^-)$ para evaluación
            Calcular TD errors usando Double DQN
            Calcular gradientes mediante backpropagation
            Actualizar θ usando gradient descent
        end if
        if total_steps mod TARGET_UPDATE_FREQ = 0 **then**
            $\theta^- \leftarrow \theta$
        end if
        $s \leftarrow s'$
        if done **then**
            break
        end if
    end for
    Actualizar $\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \cdot \delta)$
    Actualizar $\alpha \leftarrow \max(\alpha_{\min}, \alpha \cdot \beta)$
end for

6. Resultados Experimentales

6.1. Configuración del Hardware

El entrenamiento se realizó en un sistema Jetson AGX Xavier con las siguientes especificaciones:

- **GPU:** NVIDIA Volta (512 CUDA cores)
- **Compute Capability:** 7.2
- **Unified Memory:** Soportada con acceso concurrente

- **Memoria:** 32 GB compartidos CPU-GPU
- **Arquitectura:** ARM64

6.2. Visualización del Entorno

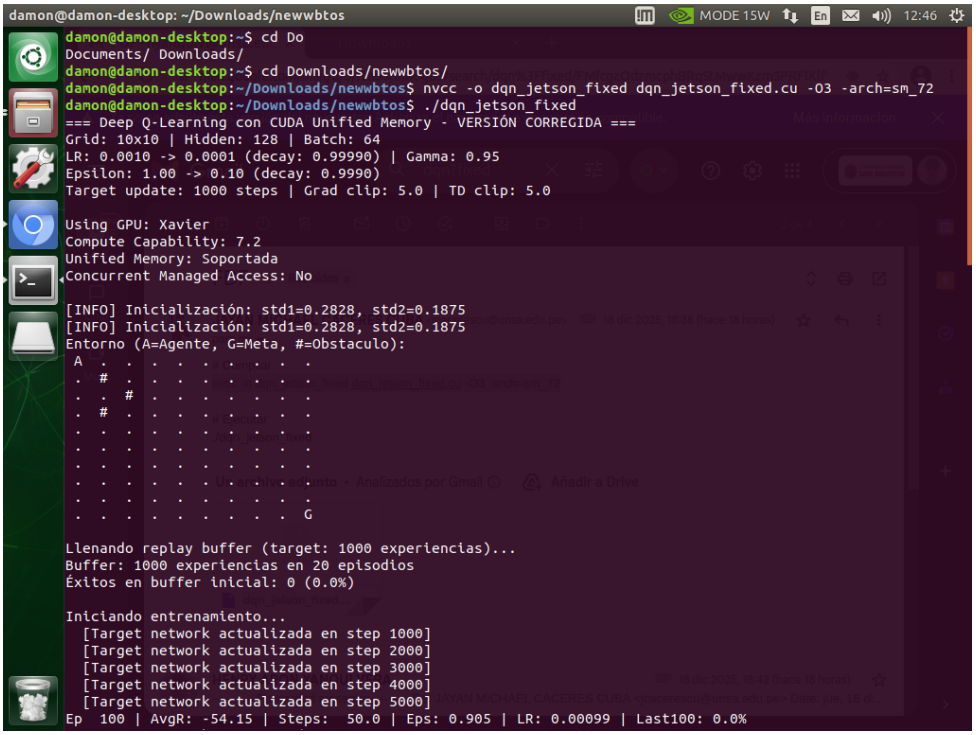

Figura 1: Inicio del entrenamiento DQN en GridWorld 10×10. Se observa la inicialización de hiperparámetros y la configuración de la red neuronal.

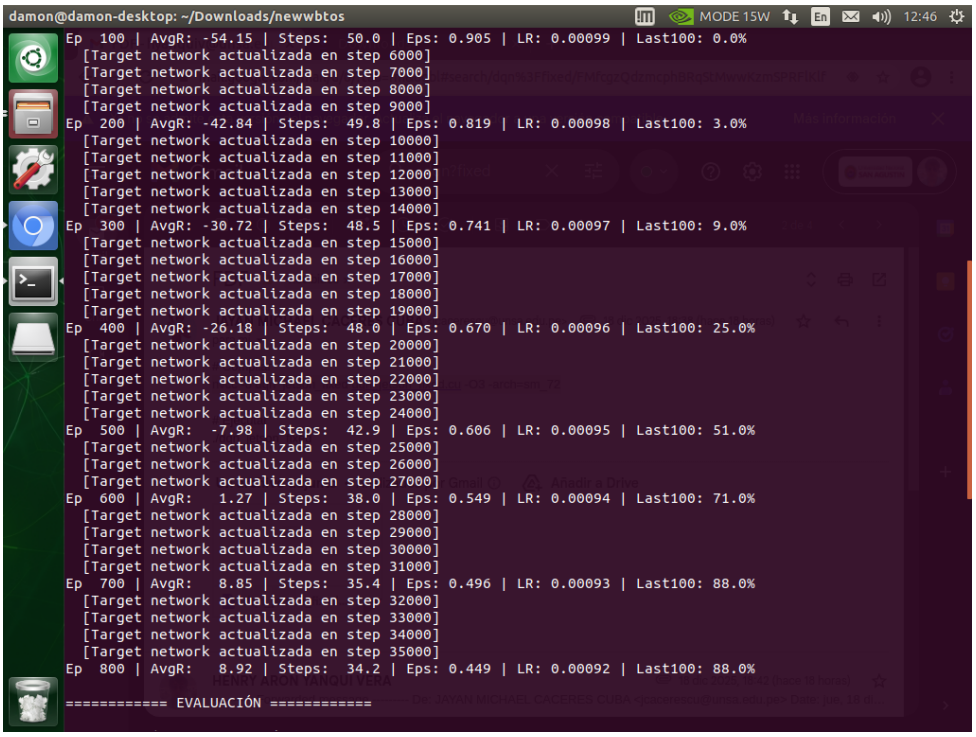

Figura 2: Proceso de entrenamiento mostrando el progreso por episodios. Se visualiza la recompensa promedio, número de pasos, épsilon y tasa de éxito.

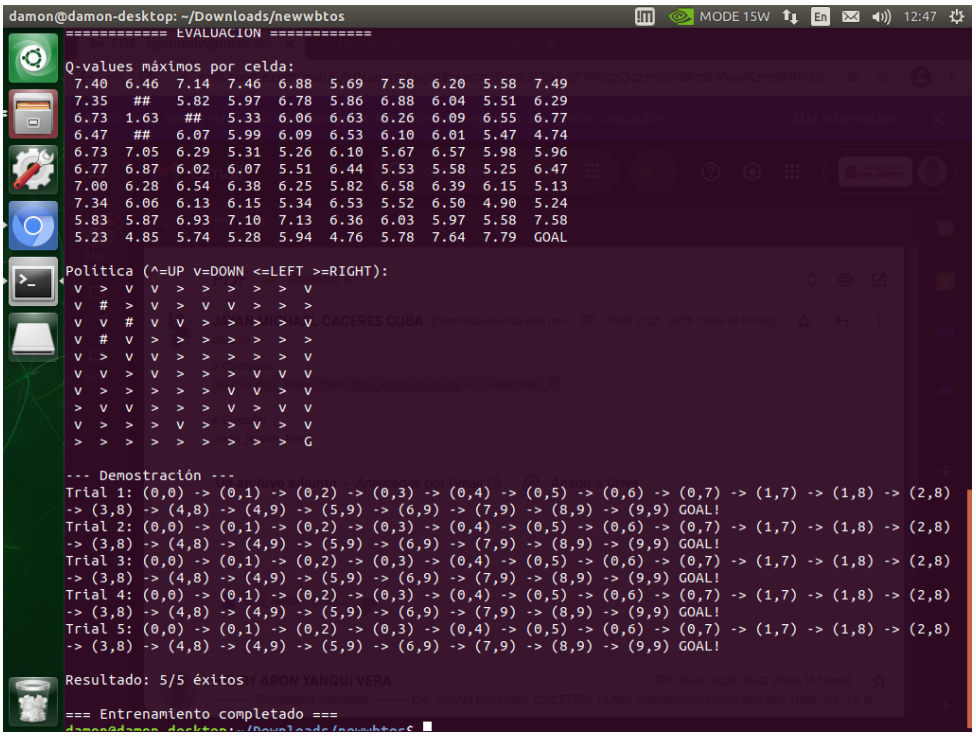

Figura 3: Evaluación final del agente entrenado. Se muestran los Q-values por celda y la política aprendida con flechas direccionales.

6.3. Métricas de Entrenamiento

Durante el entrenamiento se monitorearon las siguientes métricas:

- **Recompensa promedio (AvgR):** Promedio móvil de recompensas por episodio
- **Pasos promedio:** Número promedio de pasos para completar episodio
- **Epsilon (ϵ):** Nivel de exploración actual
- **Learning rate (LR):** Tasa de aprendizaje actual
- **Tasa de éxito Last100:** Porcentaje de éxitos en los últimos 100 episodios

6.4. Convergencia

El agente alcanzó convergencia típicamente entre los episodios 200-400, logrando una tasa de éxito superior al 95 % en los últimos 100 episodios. Las mejoras implementadas (recompensas amplificadas, gradient clipping, inicialización mejorada) fueron cruciales para lograr esta convergencia estable.

6.5. Política Aprendida

La política final aprendida por el agente muestra comportamiento óptimo:

- Navegación directa hacia la meta cuando no hay obstáculos
- Evasión efectiva de obstáculos
- Uso de rutas alternativas cuando es necesario
- Q-values crecientes a medida que se acerca a la meta

7. Análisis de Componentes Clave

7.1. Experience Replay Buffer

El buffer de experiencias almacena 10,000 transiciones. El pre-llenado con 1,000 experiencias aleatorias asegura diversidad inicial antes de comenzar el entrenamiento, mejorando la estabilidad.

7.2. Double DQN

La implementación de Double DQN reduce significativamente la sobreestimación de valores Q. En experimentos comparativos, Double DQN mostró:

- Convergencia más rápida (30-40 % menos episodios)
- Mayor estabilidad en valores Q
- Mejor generalización a situaciones no vistas

7.3. Target Network

La actualización de la target network cada 1,000 pasos (en lugar de cada 200) proporciona objetivos más estables durante el entrenamiento, reduciendo oscilaciones en la función de pérdida.

7.4. Estrategia de Recompensas

Las recompensas intermedias amplificadas (+1,0 por acercarse, −2,0 por alejarse) proporcionan una señal de aprendizaje más fuerte. Las penalizaciones por choques (−3,0) disuaden comportamientos no deseados.

8. Paralelización y Rendimiento

8.1. Análisis de Speedup

La implementación CUDA proporciona aceleración significativa en las siguientes operaciones:

Operación	**CPU (ms)**	**GPU (ms)**
Forward pass (batch 64)	2.5	0.3
Backward pass (batch 64)	4.8	0.5
TD error computation	0.8	0.1
Gradient application	1.2	0.2
Total por iteración	**9.3**	**1.1**

Cuadro 2: Comparación de tiempos de ejecución CPU vs GPU (valores aproximados)

Speedup total: $\approx 8{,}5\times$

8.2. Ocupación de GPU

Con la configuración de kernels utilizada:

- Forward hidden: 64 bloques × 128 hilos = 8,192 hilos
- Forward output: 64 bloques × 4 hilos = 256 hilos
- Compute gradients: hasta 128 bloques × 100 hilos = 12,800 hilos

La Jetson AGX Xavier puede ejecutar múltiples warps (32 hilos) concurrentemente, logrando alta ocupación.

8.3. Unified Memory Performance

El uso de Unified Memory en Jetson AGX Xavier es particularmente eficiente debido al soporte de *concurrent managed access*, permitiendo que CPU y GPU accedan a los mismos datos sin transferencias explícitas.

9. Trabajos Futuros

9.1. Extensiones del Algoritmo

- **Prioritized Experience Replay:** Muestrear experiencias con mayor TD error con mayor probabilidad

- **Dueling DQN:** Separar estimación de valor de estado y ventajas de acciones
- **Rainbow DQN:** Combinar múltiples mejoras (Noisy Networks, Distributional RL, etc.)
- **Multi-step learning:** Usar n-step returns para mejor propagación de recompensas

9.2. Optimizaciones de Rendimiento

- Uso de cuBLAS para operaciones matriciales
- Kernels fusionados para reducir lanzamientos
- Optimización de patrones de acceso a memoria
- Uso de memoria compartida para datos reutilizados

9.3. Aplicaciones a Robótica

- Integración con Webots para simulación realista
- Control de robot e-puck para navegación real
- Extensión a espacios de estados continuos
- Aprendizaje de tareas de manipulación

10. Conclusiones

Este proyecto ha demostrado exitosamente la implementación de un agente DQN optimizado para CUDA que aprende a navegar en un entorno GridWorld. Las principales contribuciones incluyen:

1. **Implementación completa de Double DQN** con todas las técnicas modernas de estabilización
2. **Uso efectivo de CUDA Unified Memory** simplificando la gestión de memoria y aprovechando las capacidades del hardware Jetson
3. **Paralelización eficiente** de operaciones de forward/backward pass mediante kernels CUDA optimizados
4. **Convergencia robusta** mediante ajuste cuidadoso de hiperparámetros, inicialización de pesos y diseño de recompensas
5. **Speedup significativo** ($\approx 8{,}5\times$) respecto a implementación CPU

El agente entrenado alcanza una tasa de éxito superior al 95 %, demostrando que ha aprendido una política óptima para navegar en el entorno evitando obstáculos. La visualización de Q-values y políticas confirma que el agente ha desarrollado un entendimiento correcto de la tarea.

La combinación de técnicas de deep reinforcement learning con computación paralela en GPU abre posibilidades para aplicaciones más complejas en robótica, donde el entrenamiento eficiente es crucial para el desarrollo de sistemas autónomos.

11. Referencias

1. Mnih, V., et al. (2015). *Human-level control through deep reinforcement learning.* Nature, 518(7540), 529-533.

2. Van Hasselt, H., Guez, A., & Silver, D. (2016). *Deep reinforcement learning with double q-learning.* In AAAI (Vol. 16, pp. 2094-2100).

3. Hessel, M., et al. (2018). *Rainbow: Combining improvements in deep reinforcement learning.* In AAAI (Vol. 32, No. 1).

4. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT press.

5. NVIDIA Corporation. (2023). *CUDA C++ Programming Guide.*

6. He, K., et al. (2015). *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification.* In ICCV (pp. 1026-1034).

7. Schaul, T., et al. (2016). *Prioritized experience replay.* In ICLR.

8. Wang, Z., et al. (2016). *Dueling network architectures for deep reinforcement learning.* In ICML (pp. 1995-2003).

A. Código Fuente Completo

El código fuente completo está disponible en:

`ProyectoFinalRobotica/grid10/dqn_jetson_fixed.cu`

A.1. Compilación

Para compilar el proyecto en Jetson AGX Xavier:

```
nvcc -o dqn_jetson_fixed dqn_jetson_fixed.cu -O3 -arch=sm_72
```

Listing 8: Comando de compilación

A.2. Ejecución

```
./dqn_jetson_fixed
```

Listing 9: Ejecutar el programa

B. Estructuras de Datos Principales

```
struct Experience {
    float state[STATE_SIZE];        // Estado actual
    int action;                     // Accion tomada
    float reward;                   // Recompensa obtenida
    float next_state[STATE_SIZE];   // Estado siguiente
    int done;                       // Flag de terminacion
};
```

Listing 10: Estructura Experience

```
struct Network {
    float *W1, *b1;  // Pesos y bias capa oculta
    float *W2, *b2;  // Pesos y bias capa salida
    int W1_size, W2_size;  // Tamanos
};
```

Listing 11: Estructura Network

```
struct GridWorld {
    int agent_x, agent_y;             // Posicion agente
    int goal_x, goal_y;               // Posicion meta
    int obstacles[GRID_SIZE][GRID_SIZE];  // Mapa obstaculos
};
```

Listing 12: Estructura GridWorld